

Principia Softwarica: The ARM Linker 51

version 0.1

Yoann Padioleau
`yoann.padioleau@gmail.com`

with code from
Rob Pike

April 28, 2026

The text and figures are Copyright © 2014–2026 Yoann Padioleau.
All rights reserved.

The source code is Copyright © 2021 Plan 9 Foundation.
MIT license.

Contents

1	Introduction	8
1.1	Motivations	8
1.2	The Plan 9 ARM linker: <code>5l</code>	8
1.3	Other linkers	9
1.4	Getting started	10
1.5	Requirements	11
1.6	About this document	11
1.7	Copyright	11
1.8	Acknowledgments	12
2	Overview	13
2.1	Linker principles	13
2.1.1	Separate compilation	15
2.1.2	Symbol resolution	15
2.1.3	Relocation	15
2.1.4	Disk image versus memory image	15
2.1.5	Libraries	15
2.1.6	Static and dynamic linking	16
2.1.7	Position-independent code	17
2.1.8	The linker as code generator	17
2.1.9	Linker performance	18
2.1.10	The tool nobody understands	18
2.2	<code>5l</code> command-line interface	18
2.3	Linking <code>hello.5</code> and <code>world.5</code>	19
2.3.1	The source files	19
2.3.2	The linking command	19
2.3.3	Inspecting objects with <code>nm</code>	20
2.3.4	Dumping objects with <code>5l -W</code>	20
2.3.5	Debugging machine code generation with <code>5l -a</code>	21
2.3.6	Inspecting executables with <code>nm</code>	23
2.4	Input object format	24
2.5	Output executable format: <code>a.out</code>	24
2.6	Code organization	26
2.7	Software architecture	26
2.8	Book structure	28
3	Core Data Structures	30
3.1	Symbols and <code>hash</code> table	30
3.2	<code>Section</code>	33
3.3	<code>Opcode</code> and <code>Operand</code>	33

3.4	Instruction	34
3.5	List of instructions	35
3.5.1	Code instructions: <code>firstp/lastp</code>	36
3.5.2	Data instructions: <code>datap</code>	36
3.6	Current instructions: <code>curtext/curp</code>	36
4	Main Functions	38
4.1	<code>main()</code>	38
4.1.1	Arguments processing	39
4.1.2	Executable format choice: <code>51 -H</code>	40
4.1.3	Executable entry point: <code>51 -E</code>	42
4.1.4	Main flow	42
4.2	Loading the objects and libraries: <code>objfile()</code>	44
4.3	Resolving symbols, computing addresses	44
4.4	Generating the executable: <code>asmb()</code>	45
4.4.1	Header	46
4.4.2	Text section	47
4.4.3	Data section	48
4.4.4	Symbol and line table sections	52
4.5	Checking for unresolved symbols: <code>undef()</code>	52
5	Loading Objects	53
5.1	Object file format: <code>.5</code>	53
5.2	A global and local program counter: <code>pc</code> and <code>ipc</code>	53
5.3	Object code input: <code>ldobj()</code>	53
5.3.1	Single instruction input	56
5.3.2	Operand input: <code>inopd()</code>	57
5.3.3	Buffered input: <code>buf</code>	59
5.3.4	Object file symbol table: <code>h</code> and <code>ANAME</code>	61
5.3.5	Private symbols and version	64
5.4	Opcode dispatch	64
5.4.1	<code>Axxx</code>	64
5.4.2	<code>ATEXT</code> and <code>autosize</code>	65
5.4.3	<code>AGLOBL</code>	66
5.4.4	<code>ADATA</code>	66
5.4.5	<code>AEND</code>	67
5.4.6	<code>AGOK</code>	67
5.5	Safe linking	67
5.5.1	Motivations	68
5.5.2	<code>ASIGNAME</code> and <code>5c -T</code>	68
5.5.3	Error management	69
6	Loading Libraries	70
6.1	Archive library format: <code>.a</code>	70
6.2	Loading libraries manually: <code>51 libxxx.a</code>	71
6.3	Loading libraries semi automatically: <code>51 -lxxx</code>	73
6.3.1	Library search path	73
6.3.2	<code>51 -L</code>	73
6.3.3	<code>51 -lxxx</code>	75
6.4	Loading libraries automagically: <code>#pragma lib "libxxx.a"</code>	75

7	Resolving	79
7.1	Issues in symbol resolution	79
7.1.1	Virtual program counter versus real code address	80
7.1.2	Data address and code size mutual dependency	81
7.2	Building the code instructions graph: <code>patch()</code>	82
7.2.1	Resolving branch instructions using symbols	83
7.2.2	Finding instruction at <code>pc</code>	83
7.2.3	Indexing <code>pc</code> , forward links overlay	85
7.3	Virtual opcodes rewriting: <code>noops()</code>	87
7.3.1	Leaf procedure optimisation	88
7.3.2	<code>ATEXT</code> patching	88
7.3.3	<code>ARET</code> rewriting	89
7.3.4	<code>ANOP</code> stripping	90
7.4	Laying out data: <code>dodata()</code>	91
7.5	Laying out code: <code>dotext()</code>	94
7.6	Defining special symbols: <code>etext</code> , <code>edata</code> , and <code>end</code>	95
7.7	Mutual recursion in layout and <code>SB/R12</code>	97
8	Code Generation Preparation	98
8.1	The linker as second-stage compiler	98
8.2	Additional data structures	99
8.2.1	<code>Optab</code> and <code>optab</code>	99
8.2.2	<code>Oprange</code>	100
8.2.3	<code>Operand_class</code>	101
8.3	<code>oplook()</code> and <code>buildop()</code>	102
8.3.1	<code>oplook()</code> and <code>cmp()</code>	102
8.3.2	<code>buildop()</code> and <code>ocmp()</code>	104
8.3.3	Optimizations	106
8.4	<code>asmout()</code>	108
9	ARM Machine Code Generation	111
9.1	ARM instruction format	111
9.2	Pseudo opcodes	111
9.2.1	<code>ATEXT</code>	111
9.2.2	<code>AWORD</code>	112
9.3	Operand subclasses and <code>instoffset</code>	112
9.3.1	<code>D_CONST</code> , <code>C_xCON</code> , and <code>immrot()</code>	113
9.3.2	<code>D_OREG</code> , <code>C_xOREG</code> , and <code>immaddr()</code>	114
9.3.3	<code>D_OREG</code> with globals, <code>C_xEXT</code>	115
9.3.4	<code>D_OREG</code> with automatics, <code>C_xAUTO</code>	116
9.3.5	<code>D_ADDR</code> , <code>C_xxCON</code>	116
9.4	Arithmetic and logic opcodes	118
9.4.1	Register-only operands	119
9.4.2	Small rotatable immediate constant operand	121
9.4.3	Bitshifted register	121
9.4.4	Bitshift opcodes	122
9.4.5	Byte and half word extractions	123
9.4.6	Multiplication opcodes	124
9.4.7	Large constant operand, <code>REGTMP</code> , and literal pools	125
9.5	Control flow opcodes	127

9.5.1	Direct jumps	128
9.5.2	Indirect jumps	129
9.6	Memory opcodes	130
9.6.1	Load	130
9.6.2	Store	133
9.6.3	Swaps	134
9.6.4	Symbol addresses	135
9.6.5	Half words and signed bytes	136
9.7	Software interrupt opcodes	139
9.8	Literal Pools	140
9.8.1	blitrl/elitrl	140
9.8.2	addpool()	141
9.8.3	flushpool()	143
9.8.4	checkpool()	143
9.8.5	LPOOL	144
10	Debugging Support	146
10.1	Debug tables vs DWARF	146
10.2	asmb() and the debugging tables	147
10.3	Executable symbol table	147
10.3.1	Symbol table format: putsymb()	148
10.3.2	Globals and procedures symbols: asmsym()	148
10.3.3	Stack variables symbols	150
10.3.4	Filename and line origin symbols	152
10.4	File and line information	153
10.4.1	Locations in objects: AHISTORY	153
10.4.2	Locations in 5l memory	153
10.4.3	Locations in executables	159
11	Profiling Support	162
11.1	5l -p and _mainp	162
11.2	5l -p -1 and __mcount	162
11.3	5l -p and _profin()/_profout()	165
11.4	Disabling profiling attribute: NOPROF	168
12	Advanced Topics	170
12.1	Dynamic linking	170
12.1.1	Export table: 5l -x	170
12.1.2	Dynamic loading: 5l -u	174
12.1.3	SUNDEF	175
12.1.4	XXX	176
12.1.5	Relocatable address: C_ADDR	180
12.2	Position independent code (PIC)	182
12.3	Optimizations	182
12.3.1	Opcode rewriting	182
12.3.2	Operand rewriting	182
12.3.3	Small data first	183
12.3.4	Compacting chains of AB, brloop()	183
12.3.5	Removing useless instructions, follow()	184
12.4	Overriding symbol attribute: DUPOK	187

12.5	Other executable formats	187
12.5.1	ELF (Linux)	187
12.5.2	OMach (mac OS)	189
12.5.3	PE (Windows)	189
12.6	Other instructions	189
12.6.1	Float operations	189
12.6.2	Division	201
12.6.3	Long multiplication	206
12.6.4	Multiple registers move	206
12.6.5	Status register	207
12.6.6	Half words and bytes moves since ARMv4	208
12.6.7	Shifted moves	212
12.7	Compiler-only pseudo opcodes	213
12.8	5l -E digits	213
12.9	Strings in text segment: 5l -t	214
13	Conclusion	216
13.1	Patterns and techniques	216
13.2	Connections to other books	216
13.3	Beyond the Plan 9 linker	217
A	Debugging	218
A.1	Dumpers	218
A.1.1	Instruction dumper: Pconv()	219
A.1.2	Opcode dumper: Aconv()	220
A.1.3	Operand dumper: Dconv()	220
A.1.4	Symbol kind dumper: Nconv()	222
A.1.5	Conditional execution dumper: Cconv()	223
A.1.6	String dumper: Sconv()	224
A.2	Verbose mode: 5l -v	225
A.3	Objects loading debugging: 5l -W	225
A.4	Opcode table debugging: 5l -O	225
A.5	Machine code generation debugging: 5l -a	226
A.6	Symbol table debugging: 5l -n	227
A.7	Line table debugging: 5l -V	227
B	Error Management	229
C	Profiling	231
D	Utilities	232
D.1	Memory management	232
D.2	Buffer management	233
D.2.1	Output buffer	234
D.2.2	Input buffer	235
D.3	File management	235
D.4	String processing	235
D.5	Mathematic functions	236

E	Linker-related Programs	237
E.1	include/mach.h	237
E.2	size	237
E.3	nm	238
E.4	ar	244
E.5	strip	266
F	Extra Code	270
F.1	include/	270
F.1.1	include/exec/a.out.h	270
F.1.2	include/exec/elf.h	271
F.1.3	include/obj/ar.h	273
F.2	linkers/misc/	273
F.2.1	linkers/misc/ar.c	273
F.2.2	linkers/misc/nm.c	277
F.2.3	linkers/misc/size.c	278
F.2.4	linkers/misc/strip.c	278
F.3	linkers/5l/	278
F.3.1	linkers/5l/l.h	278
F.3.2	linkers/5l/m.h	283
F.3.3	linkers/5l/globals.c	283
F.3.4	linkers/5l/optab.c	284
F.3.5	linkers/5l/utils.c	285
F.3.6	linkers/5l/error.c	285
F.3.7	linkers/5l/fmt.c	285
F.3.8	linkers/5l/layout.c	286
F.3.9	linkers/5l/pass.c	286
F.3.10	linkers/5l/datagen.c	287
F.3.11	linkers/5l/dynamic.c	287
F.3.12	linkers/5l/codegen.c	288
F.3.13	linkers/5l/io.c	289
F.3.14	linkers/5l/asm.c	289
F.3.15	linkers/5l/span.c	289
F.3.16	linkers/5l/obj.c	290
F.3.17	linkers/5l/lib.c	291
F.3.18	linkers/5l/noop.c	291
F.3.19	linkers/5l/float.c	291
F.3.20	linkers/5l/profile.c	292
F.3.21	linkers/5l/debugging.c	292
F.3.22	linkers/5l/hist.c	292
F.3.23	linkers/5l/main.c	292
	Glossary	294
	Index	295
	References	304

Chapter 1

Introduction

The goal of this book is to explain with full details the source code of a linker.

1.1 Motivations

Why a linker? Because I think you are a better programmer if you fully understand how things work under the hood, and the linker is the final piece of any development toolchain, the one generating finally the executable.

There are very few books about linkers, as opposed to a myriad of books on compilers or kernels. Linkers are usually not even studied during the computer science curriculum. This is a pity because they are central. Indeed, they make the link (no pun intended) between the compiler and the kernel: the linker generates from object files executables that the kernel will load. Without a linker there is no program to execute.

Here are a few questions I hope this book will answer:

- What is the format of object files?
- What is the format of executables? What is the format of machine instructions?
- What is the format of libraries?
- How are object files and libraries combined together and patched to form the final executable?
- What is the difference between a linker and a loader?
- What is the memory image of a program? How does it relate to the executable file? How does it relate to the original source code?
- What debugging information contains executables? How source-level debuggers can find which C source code corresponds to a specific binary instruction?

1.2 The Plan 9 ARM linker: 51

I will explain in this book the code of the Plan 9 ARM linker 51¹, which is about 8150 lines of code (LOC). 51 is written entirely in C.

Like for most books in Principia Softwarica, I chose a Plan 9 program because those programs are simple, small, elegant, open source, and they form together a coherent set. The '5' comes from the Plan 9 convention to name architectures with a single character ('0' is MIPS, '5' is ARM, '8' is x86, etc), and the '1' means linker.

¹See <http://plan9.bell-labs.com/magic/man2html/1/81>, which despite its name covers also 51.

Like for the other Principia Softwarica books covering the development toolchain, I chose the ARM architecture [Sea01] variant, in this case of the Plan 9 linker 51, and not for instance the x86 variant 81, for reasons of simplicity. Indeed RISC machines are far simpler than CISC machines. Moreover, the availability under Plan 9 of an ARM emulator, 5i, helps to understand the semantics of the machine instructions generated by 51.

The Plan 9 approach to linking is a bit different than in other operating systems. The Plan 9 linker is responsible for generating executables, but also for generating the machine code from the object files. Indeed, the object files generated by 5a and 5c are ARM-specific, but they do not contain machine code. Instead, under Plan 9 an object file is essentially the serialized form of the abstract syntax tree of an assembly source. The linker 51 generates the actual machine code. I think though that it is actually a better design because it leads to less code in total and also to simpler code. The Plan 9 linker does also some dead code elimination which reduces the size of the binaries. This is partly to compensate for the lack of dynamic shared libraries in Plan 9.

1.3 Other linkers

Here are a few linkers that I considered for this book, but which I ultimately discarded:

- The original UNIX linker, called `ld` (for link edit), was creating executables for the DEC PDP11 architecture. `ld` started as an assembly program in UNIX V2² and finished as a C program in UNIX V7³. `ld` contains 1300 LOC, which is smaller than 51. This is partly because 51 contains also the code to generate machine code (a job usually done in the assembler). However, `ld` targets an obsolete architecture (the PDP11).
- The GNU Linker (also called `ld`), part of the `binutils` package⁴, is probably the most used open source linker. It is using the Binary File Descriptor (BFD) library to read and write object files using different formats. It supports many architectures (ARM, x86, etc) as well as many executable and object file formats (ELF, `a.out`, etc). It is fairly big though: 30 000 LOC for `ld/` and 500 000 LOC for `bfd/`. Even the ARM-specific file `bfd/elf32-arm.c` has already 17 000 LOC.
- Gold [Tay08] is a faster multi-threaded ELF linker originally developed at Google. It is now part of the `binutils` package. It supports also many architectures (ARM, x86, etc). It is unfortunately also fairly big: 153 000 LOC. Again, even the ARM-specific file `gold/arm.cc` has already 13 000 LOC.
- The LLVM Linker `lld`⁵ aims to remove the current dependency to the host linker (e.g., the GNU linker under Linux) in the LLVM infrastructure. Its goal is also to be more modular and extensible than `ld`. It supports also many architectures and executable formats. It is smaller than `ld` and Gold, but still fairly large: 71 000 LOC.
- LD86⁶ is an x86 16-bit and 32-bit linker, part of Bruce Evans' C compiler (BCC). It is an historical linker used to compile old versions of Minix and Linux. It is fairly small: 6100 LOC, which is smaller than 51. But, to be fair, 51 does also machine code generation, which is done instead by AS86 for LD86. 5a+51 is made of 12 000 LOC, which is smaller than AS86+LD86 at 18 600 LOC. Moreover, the instruction format in the ARM is far simpler to understand than the one in the x86, which makes it a better candidate for Principia Softwarica.

Figure 1.1 presents a timeline of major linkers. I think 51 represents the best compromise for this book: it implements the essential features of a linker, for an architecture that is still relevant today (the ARM), while still having a small and understandable codebase (8150 LOC).

²<http://minnie.tuhs.org/cgi-bin/utree.pl?file=V2/cmd/ld1.s>

³<http://minnie.tuhs.org/cgi-bin/utree.pl?file=V7/usr/src/cmd/ld.c>

⁴<https://www.gnu.org/software/binutils/>

⁵<https://lld.llvm.org/>

⁶<http://v3.sk/~lkundrak/dev86/>

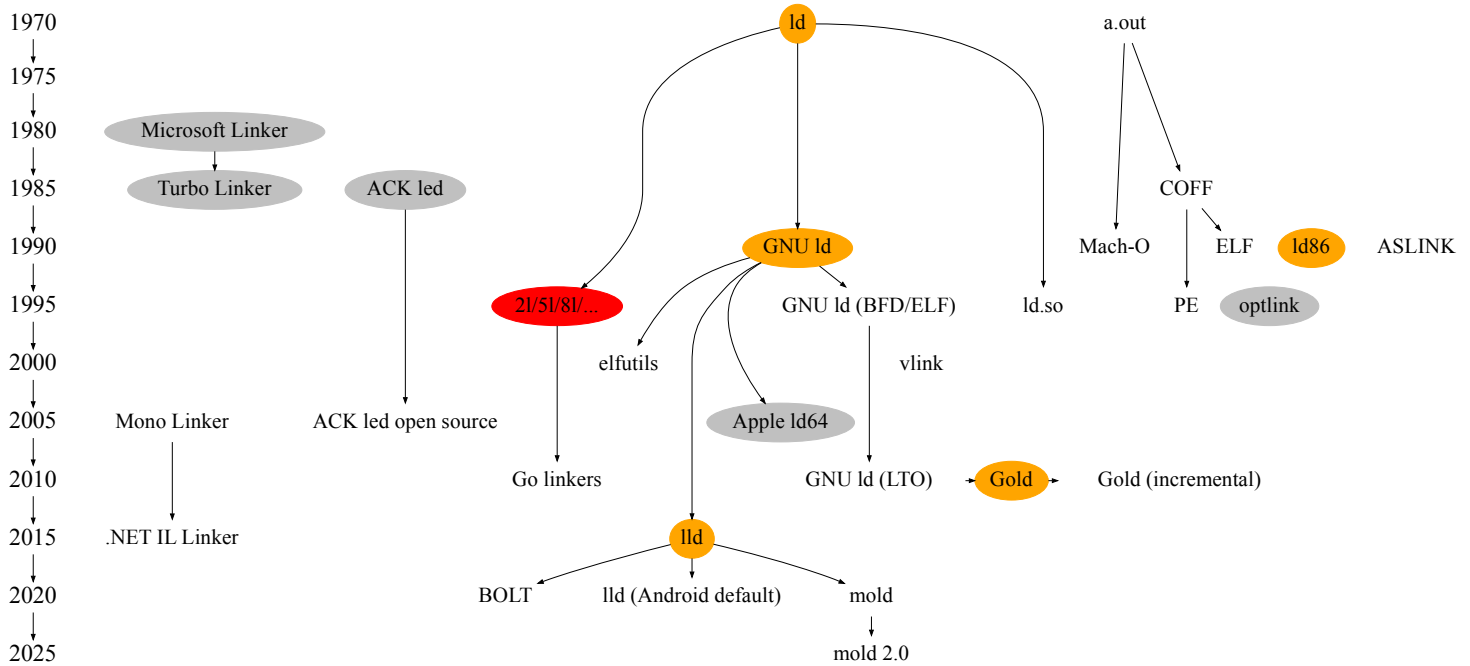


Figure 1.1: Linkers timeline

5l is obviously not as used as the GNU linker, but it is still a production-quality linker. It was used to link all Plan 9 programs at Bell Labs and it is still used in the toolchain of the Go programming language⁷. Because Go was originally conceived and used at Google, some of Google’s services are linked by a Plan 9 linker.

1.4 Getting started

To play with 5l, you will first need to install the Plan 9 fork used in Principia Softwarica (see <https://www.principia-softwarica.org/getting-started.html>). The Plan 9 toolchain (5a, 5l, 5c) can also be compiled natively on Linux, macOS, or Windows through Goken9cc⁸, so you can cross-link ARM programs from your regular machine. Once installed, you can test 5l under Plan 9 with the following commands:

```

1  $ cd /tests/5l
2  $ 5a hello.s && 5a world.s
3  $ 5l hello.5 world.5 -o hello
4  $ ./hello
5  hello world
6  $

```

The command in Line 2 assembles the simple `hello.s` and `world.s` ARM assembly programs and generates the `hello.5` and `world.5` ARM object files. Line 3 then links the object files together and generates the final ARM binary executable `hello`. Line 4, which assumes you are under an ARM machine (e.g., a Raspberry Pi⁹), launches the program.

Note that it is easy under Plan 9 to cross compile from another architecture: you can use the same commands, 5a, 5l, etc. To play with 5l under an x86 machine you just need after the linking step to use the ARM emulator 5i instead:

⁷<https://golang.org/cmd/link>

⁸<https://github.com/aryx/goken9cc>

⁹<https://www.raspberrypi.org/>

```
...
4  $ 5i hello
...
```

See the EMULATOR book [Pad15b] for more information on 5i.

1.5 Requirements

Because most of this book is made of C source code, you will need a good knowledge of the C programming language [KR88] to understand it. In fact, for Chapter 9, where you will see the code that generates ARM instructions, you will need a very good knowledge of C. Indeed, the code does lots of bit manipulations and uses many C idioms.

There are very few books explaining how a linker works. I can cite *Linkers and Loaders* [Lev99] and one chapter of *Computer Systems: A Programmer's Perspective* [BO10]. Thus, I assume most programmers do not really know how a linker works, which is why, in addition to explaining the code of 51, I will also explain most of the concepts related to linking in this book. This is a bit unusual in our Principia Softwarica series. Indeed, I usually assume the reader has read books introducing at least the concepts and theories underlying the programs I present.

Reading the ASSEMBLER book [Pad15a] is a requirement to understand this book. Indeed, many data structures introduced (and fully explained) in the ASSEMBLER book [Pad15a] are similar to data structures used by 51. This is normal because the assembler generates what the linker consumes. Those data structures will be presented only quickly in this book. The same is true for the object file format described at length in the ASSEMBLER book [Pad15a].

It is not necessary to know the ARM architecture to understand most of this book. For the machine code generation part though, in Chapter 9, I highly recommend to print the excellent colorful ARM reference card <http://re-eject.gbadev.org/files/armref.pdf>. It will help you to visually understand the binary format of the instructions. This card is very helpful to understand the code which does many bit manipulations to generate different parts of an ARM instruction.

If, while reading this book, you have specific questions on the command-line interface of 51, I suggest you to consult the manual page at `docs/man/1/81` in my Plan 9 repository (despite the name, this page documents all the Plan 9 linkers, including 51). The related pages `docs/man/1/ar`, `docs/man/1/nm`, and `docs/man/1/size` cover the auxiliary tools that manipulate the object and executable files described in this book. The “Plan 9 C Compilers” paper (at `compilers/docs/compiler.pdf` in my Plan 9 repository) also contains a short section on the linker that complements the material in this book. Finally, `docs/articles/port.ps` (“Portability and the C Compilers”) explains the portability strategy shared by the whole Plan 9 toolchain, which is useful background for Chapter 9.

1.6 About this document

This document is a *literate program* [Knu92]. It derives from a set of files processed by a tool, `syncweb` [Pad09], generating either this book or the actual source code of the program. The code and its documentation are thus strongly connected.

1.7 Copyright

Most of this document is made of source code from Plan 9, so those parts are copyright by Plan 9 Foundation. The prose is mine and is licensed under the All Rights Reserved License.

1.8 Acknowledgments

I would like to acknowledge of course the author of 51, Rob Pike, who wrote in some sense most of this book.

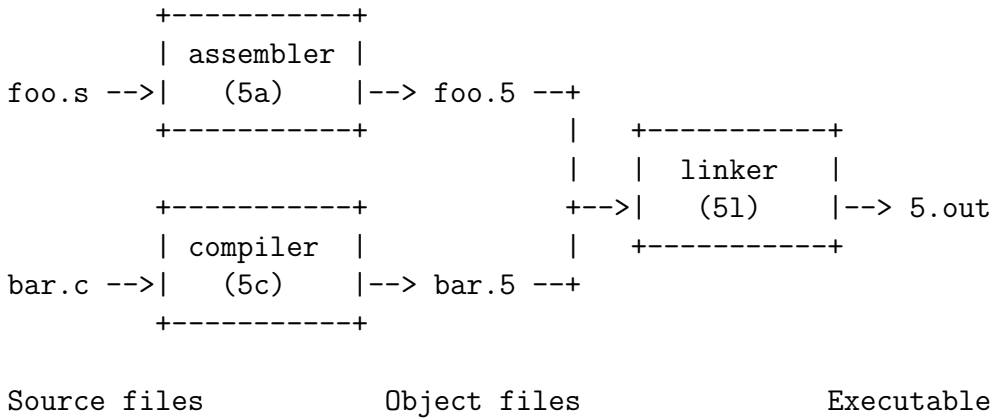


Figure 2.1: Linker inputs and output.

Chapter 2

Overview

Before showing the source code of 51 in the following chapters, I first give an overview in this chapter of the general principles of a linker. I also describe quickly the format of the object files generated by 5a and 5c and used as inputs by 51, as well as the format of the executables generated by 51. I also define terms, explain how the code is organized, and more generally give the background necessary to understand the code I will show later.

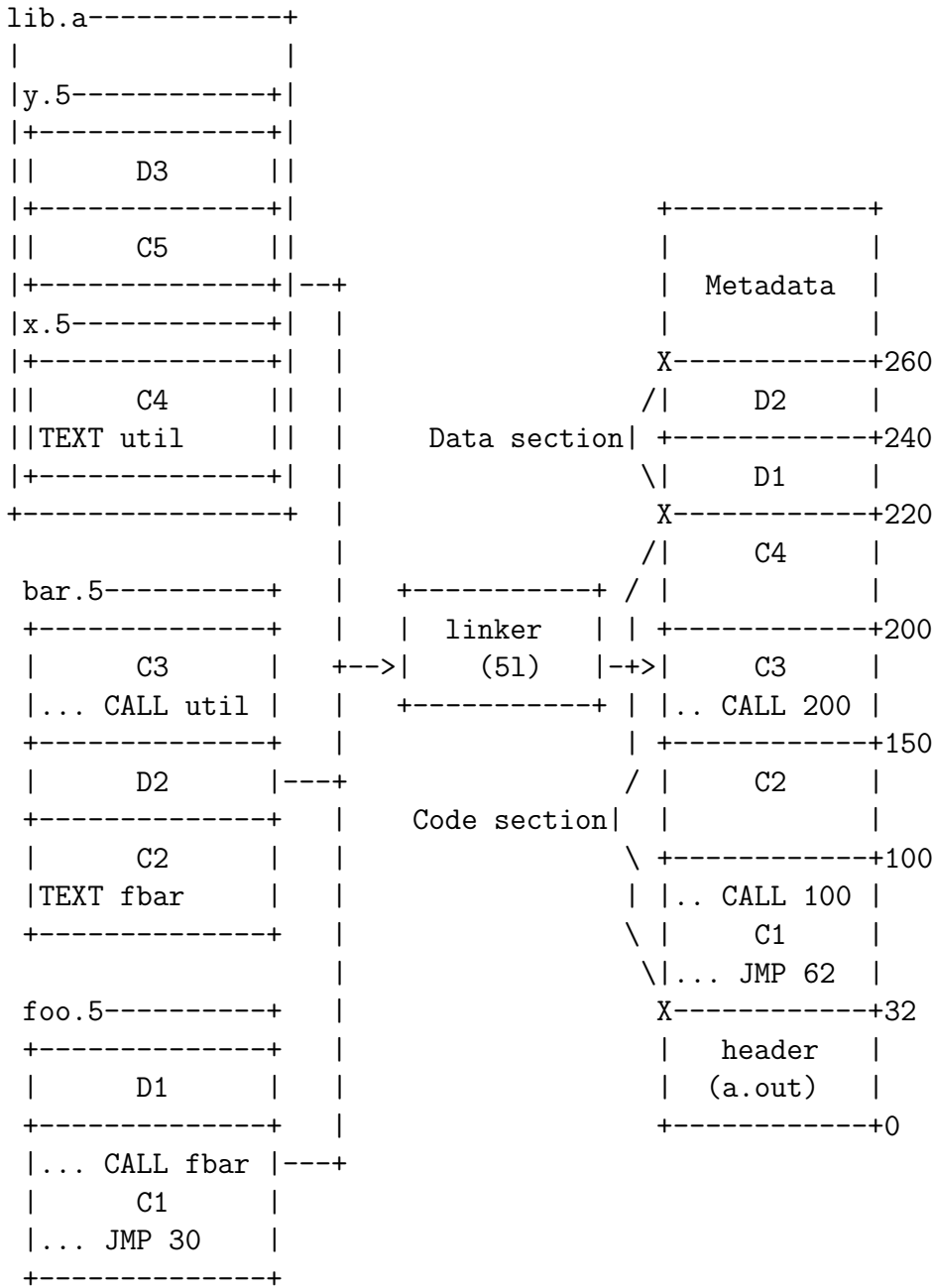
2.1 Linker principles

A *linker* is a program which takes as input multiple object files and combine them to form an executable as shown in Figure 2.1.

An *object file* is really the simplest form of a *module*. It packs code, data, and information about exported and imported entities. Object files are generated by assemblers and compilers.

An *executable* is a file containing machine instructions as well as data. The code and data are stored in different parts of the file called *sections* as shown in Figure 2.2. The size and location of those sections are stored at the beginning of the file in a part called the *header*. There are different kinds of headers corresponding to different kinds of *executable formats*. In this book I will focus on the `a.out` format which is very simple. Section 12.5 will present other formats (including the popular ELF format used in Linux).

Thanks to the header, a part of the kernel called the *loader* can know where the code and data are stored in the executable file as well as its entry point. The loader, triggered by the `exec()` system call (see the `KERNEL` book [Pad14]), can then load (copy) in memory the different sections of the executable file and start executing its code.



Input object files and library

Output executable

Figure 2.2: Linking process overview.

Figure 2.2 gives an overview of the linking process. The linker first *concatenates* the code parts of the different object files together to form the *code section* (the code parts C1 to C4), also known as the *text section*. It does the same for the data to form the *data section* (D1 and D2). Then, it links the *uses* of symbols in those code (or data) parts, e.g., the call to `fbar` in C1, to their *definitions* in possibly other code (or data) parts, here the definition of `fbar` in C2.

2.1.1 Separate compilation

One of the main operation of a linker is to concatenate parts of different files together. In fact, the program `cat` can be seen as a very primitive linker. Instead of using a linker, you could use `cat` to concatenate multiple source programs together and compile/assemble the resulting single (big) file¹. As programs grow larger, this approach becomes inconvenient though. Linkers and object files enable *separate compilation* which in the long term saves lots of compilation time.

2.1.2 Symbol resolution

Linking the uses of symbols to their definitions is also called *resolving* the references to external symbols. The linker can do so because it has access to all the code (and data). Once all the code (and data) parts are concatenated together, the final addresses of the different entities can be known. So, in Figure 2.2 the call to `fbar` in `C1` can be transformed in a machine instruction to go to the address 100 where the code of `fbar` resides (assuming `fbar` was the first procedure in `bar.5` and so `C2`).

2.1.3 Relocation

Another important operation done by the linker is called *relocation*. Remember from the ASSEMBLER book [Pad15a] that 5a can resolve the use of labels in jump instructions as both the definitions and uses of those labels must be in the same file. The instructions generated for those jumps assume though that the code of the program was loaded at the memory address 0. For instance, to jump to a label `foo` which happens to be defined just before the 30th instruction in a program `foo.s`, the instruction `JMP 30` will be generated in the object file as shown in Figure 2.2 (see `foo.5`). Once the code of object files are concatenated together and put after the header, the linker must *relocate* the jump instructions to take into account the new *memory address origin* where the object code containing the jump will be loaded. For our previous example, the linker generated the relocated instruction `JMP 62` in Figure 2.2 as `C1` is after the executable header which uses 32 bytes.

2.1.4 Disk image versus memory image

Linkers and loaders are strongly connected, just like assemblers and linkers. Indeed, one produces what the other consumes. In fact, early linkers were also called loaders as they were responsible for linking and loading in memory a program. It is important to note though, that the *disk image* of an executable generated by the linker does not necessarily match exactly the *memory image* of the program when loaded in memory by the loader. An offset in the file does not necessarily correspond to the same offset in memory, even though I assumed this was the case in Figure 2.2.

Under Plan 9, the header and code section are actually not loaded at the memory address 0 but instead after the first page at 4096 (4K). Indeed, the first page in the virtual address space of a program is reserved by the kernel and marked specially to generate faults when accessed. This helps for instance to track null pointer bugs. So, the instructions `CALL fbar` and `JMP 30` in Figure 2.2 would be actually transformed respectively in the resolved `CALL 4196` and relocated `JMP 4158` instructions with 51 under Plan 9.

2.1.5 Libraries

Object files can be concatenated together to form *libraries* using a tool called `ar` (for archive). The linker can also take as input a set of libraries as shown in Figure 2.2 with `lib.a`. Those libraries are convenient for programmers because they just have to remember the name of one file, e.g., `libc.a`, instead of a possible long list of object files.

¹Assuming the compiler/assembler could from one source file generate directly an executable.

Moreover, the linker can also take care of including only the object files in the library that matters, that is the object files containing code or data referenced by the other object files passed on the command line. For instance, in Figure 2.2 the linker decided to include in the executable the code of `x.5` (C4), which is part of `lib.a`, because it contains the definition of a procedure `util()` which is mentioned in the object file `bar.5`. It did not include the code from `y.5` though because such code would be anyway *dead code* in the executable. Such code would waste disk and memory space.

2.1.6 Static and dynamic linking

The linking I described until now is completely *static*. All the symbol references must be known at *link-time* in which case they can be fully resolved to generate an executable. This requires that all the object files (or libraries) containing those symbols get passed to the linker on the command line.

Another popular form of linking is called *dynamic linking*. Dynamic linking blurs the line between the responsibilities of the linker and loader. Indeed, with dynamic linking an executable can still contain unresolved references to external symbols; the loader at *load-time* must, before loading the program in memory, link additional objects.

Static:

```
+-----+
| a.out  |
|        |
| main:  |
| CALL sin |
|        |
| sin:   |
| (copied |
| from  |
| libm.a) |
+-----+
```

One file,
self-contained

Dynamic:

```
+-----+ +-----+
| a.out  | | libm.so |
|        | |        |
| main:  | | sin:   |
| CALL sin-->| ...   |
|        | |        |
| (no sin | +-----+
| here)   | | loaded at
+-----+ | runtime by
          | ld.so
```

Two files,
linked at load time

In practice this means the executable carries a list of shared libraries it needs (`.so` on Linux, `.dylib` on macOS, `.dll` on Windows), and a *dynamic linker* (also called a *runtime loader*) maps them into the process's address space and patches the remaining symbol references before `main` runs. The main advantage is that a shared library's code exists once on disk and once in memory, no matter how many programs use it.

Dynamic linking was introduced by Multics in the 1960s and popularized by SunOS's shared libraries in the 1980s, becoming the default on Linux, macOS, and Windows by the 1990s. The main disadvantage turned out to be complexity: the dynamic linker must find the right library version at load time (Linux's `ldconfig` and `LD_LIBRARY_PATH`, Windows' DLL search order, macOS's `install_name_tool`), version mismatches cause the infamous "DLL hell," and security-sensitive code must guard against `LD_PRELOAD`-style injection. In a reversal that vindicated Plan 9's choice, Go (2009) and Rust both default to static linking today: the simplicity of one self-contained binary outweighs the space savings on modern machines with abundant RAM.

Plan 9 opted mainly for static linking and so 5l is mainly a static linker. Static linking is far simpler than dynamic linking and is arguably also better in many respects. I will delay the discussions on dynamic linking to Section 12.1.

For more information on the principles of linkers and loaders I recommend to read *Linkers and Loaders* [Lev99] which is entirely dedicated to the topic.

2.1.7 Position-independent code

The relocation scheme from Section 2.1.4 works for static executables: the linker knows the final addresses and patches them once. But a shared library loaded by the dynamic linker cannot know its address in advance—different processes may load it at different addresses. The solution is *position-independent code* (PIC): code that works regardless of where in memory it is loaded, by accessing all global data through a level of indirection.

In a static executable, the linker can patch a call to `sin()` with its final address (say, BL 0x7200) because the executable is always loaded at the same place. A shared library does not have that luxury: process A might load `libm.so` at address 0x7f000000 while process B loads it at 0x7f400000. If the library's code contained BL 0x7f000000, it would work in A but crash in B. PIC solves this by never putting a fixed address in the code at all—every reference goes through a table that the dynamic linker fills at load time.

On most architectures the indirection uses two tables. A *Global Offset Table* (GOT) holds one pointer per global variable the library needs. PIC code reads `errno` not by its fixed address but by loading from `GOT[errno]`—a pointer that `ld.so` fills with `errno`'s actual address after loading the library. A *Procedure Linkage Table* (PLT) does the same for function calls: calling `sin()` jumps to a PLT stub which, on the first call, asks `ld.so` to resolve `sin`'s address and write it into the GOT; on subsequent calls the stub reads the cached GOT entry and jumps directly, so the overhead is paid only once:

a.out code	GOT	libm.so
+-----+	+-----+	(loaded at unknown
		address by ld.so)
load GOT[0],R0 +--->	&errno --+--->	+-----+
		errno
call PLT[sin] ++	&sin ---+---+	+-----+
		sin:
+-----+		+--> ...
	PLT stub:	+-----+
	++>+-----+	
	load &sin	1st call: ask ld.so
	from GOT	to resolve; cache
	jump to	the result in GOT.
	that addr	2nd+ call: jump
	+-----+	directly.

The cost is one extra memory load per global access and one extra jump per uncached function call—a small price for the ability to share a single copy of `libm.so` across every process on the machine. The GOT/PLT mechanism is what makes `.so` files on Linux, `.dylib` on macOS, and `.dll` on Windows work transparently—the calling code does not know or care whether `sin()` lives in the same executable or in a shared library loaded at an arbitrary address.

Plan 9's 51 does not generate PIC because Plan 9 uses static linking and each executable has a fixed load address. This is one of the simplifications that keeps 51 small—but understanding PIC and the GOT/PLT is essential for anyone who will later work with Linux or macOS shared libraries.

2.1.8 The linker as code generator

In most toolchains, the compiler generates final machine code and the linker only patches addresses. `gcc` or `clang` emit complete ARM or x86 instructions; the assembler translates mnemonics to binary; and the linker (GNU `ld`, LLVM `lld`) concatenates the binary, resolves symbols, and patches address fields in already-complete instructions. The linker never needs to know what an `ADD` instruction looks like—it only knows where the address fields sit and how to fill them.

Plan 9’s 5l splits the work differently. The compiler (5c) and assembler (5a) emit a *pseudo-code*: object files contain symbolic instructions (MOVW, ADD, BL) with symbolic operands, *not* final ARM machine code. The linker then selects the actual ARM instruction encoding, expands virtual instructions (RET becomes MOV LR, PC; a MOVW with a constant too large for an immediate field becomes a load from a *literal pool*), and allocates those literal pools. This makes 5l much more complex than a “normal” linker—it is really a linker *plus* a back-end code generator. But it has an advantage: because the linker sees *all* the code at once, it can make better decisions about instruction encoding and literal placement than a compiler or assembler that sees only one file at a time. Readers coming from the GNU or LLVM toolchains should expect this book to contain chapters that look more like compiler chapters than linker chapters—instruction selection, immediate encoding, literal pools—and this design choice is why.

2.1.9 Linker performance

On large codebases, linking is often the bottleneck: the compiler processes files in parallel, but the linker must see everything at once. GNU ld (the traditional default on Linux) is single-threaded and makes multiple passes, which can take minutes on a million-line project. Google’s gold (Ian Lance Taylor, 2008) was written specifically to be faster. LLVM’s lld improved further and became the default for Chrome and the Linux kernel (when built with clang). Most recently, mold (Rui Ueyama, 2021) achieved the fastest linking times by parallelizing aggressively across all phases—linking the Chrome executable in under two seconds where GNU ld takes over a minute. Plan 9’s 5l is fast for a different reason: it does less (no dynamic linking, no DWARF, no section-level garbage collection, no LTO) and targets a simple object format, so speed was never the bottleneck it became for Linux toolchains.

2.1.10 The tool nobody understands

Every working programmer has seen a linker error—“undefined reference to foo”, “multiple definition of bar”, “relocation truncated to fit”—and most cannot explain what the linker actually *does*. The assembler is intuitive (“translate mnemonics to binary”); the compiler is famous (“translate C to assembly”); but the linker sits between the two and the executable, doing work that is invisible when it succeeds and baffling when it fails. This invisibility is precisely why it is worth studying: the linker is where the abstractions of separate compilation (“I can compile this file without knowing what’s in that file”) meet the reality of a single address space (“every symbol must have exactly one address”). Understanding how 5l resolves symbols, assigns addresses, and patches references is what turns a linker error from an opaque incantation into a sentence you can read.

2.2 5l command-line interface

The command line interface of 5l is pretty simple:

```
$ 5l
usage: 5l [-options] objects
$ 5l foo.5 bar.5 lib.a
$ ./5.out
```

Given the set of object files foo.5 and bar.5, and a library lib.a, 5l outputs an executable file called 5.out. You can change this default behaviour by using the -o <outfile> option.

The default executable format is a.out, a classic UNIX format, but this can be changed with the -H<num> option (H for header) as explained in Section 4.1.2. You can also change the entry point of the program with -E<funcname> which by default is _main as explained in Section 4.1.3. Other options related to debugging will be described later in Appendix A.

Executables generated by C compilers in UNIX-like operating systems, e.g., `gcc` under Linux, are usually called by default `a.out`. However, under Plan 9, for the same reason ARM object files use the `.5` filename extension and not `.o`, ARM executables are called `5.out` not `a.out`. This is more convenient in an operating system supporting multiple architectures at the same time. That way, you can have in the same directory an ARM executable `5.out` and an x86 executable `8.out` without any name conflict.

Another important linker option, `-T<num>`, allows to change the memory address origin of the code section (T for text section). In Plan 9 the default value for this address is 4128. Indeed, the loader in the kernel loads executables after the first page (4096), and includes the header which is using 32 bytes, so the text section will start at the memory address 4128. The machine code generated for the jumps and calls must then assume the code will be loaded at 4128. You can also change the memory address origin of the data section as explained in Section 4.1.2. Those options are almost never used by programmers, but they are necessary for producing special executables such as kernels or boot loaders as you will see in the KERNEL book [Pad14]. Indeed, those binaries will be loaded (by the bootstrapping process or firmware) at special memory addresses (e.g., 0x80000000 for the ARM Plan 9 kernel, or 0x7c00 for an x86 bootloader).

Another set of options are used to manage libraries, which are really files encapsulating a set of object files, and will be introduced later in Chapter 6.

2.3 Linking `hello.5` and `world.5`

In this section, I adapt the `helloworld.s` example of the ASSEMBLER book [Pad15a] to illustrate the linking process with a concrete example. The goal is also to learn how to use the debugging options of 5l as well as tools such as `nm`.

2.3.1 The source files

I split the original source file `helloworld.s` in two files `hello2.s` and `world.s`. I also use the C library functions `fprint()` and `exits()` instead of using directly the `PWRITE` and `EXITS` system calls:

```
<linkers/5l/tests/hello2.s 19a>≡
TEXT main(SB), $8
    /* fprint(1,&hello) */
    MOVW $1, R0
    MOVW $hello(SB), R1
    MOVW R1, 8(R13)
    BL fprint(SB)
    /* exit(0) */
    MOVW $0, R0
    BL exits(SB)
```

```
<linkers/5l/tests/world.s 19b>≡
GLOBAL hello(SB), $12
DATA hello+0(SB)/8, $"hello wo"
DATA hello+8(SB)/4, $"rld\n"
```

If you do not understand the code, or the calling conventions, read the ASSEMBLER book [Pad15a]. Symbol definitions and uses are now spread over different files: the `hello` global is defined in `world.s` but used in `hello2.s`. Moreover, the `fprint()` and `exits()` functions are defined in source files of the C library but used in `hello2.s`.

2.3.2 The linking command

To compile the previous program do:

```

$ cd /tests/51
$ 5a hello2.s -o hello.5
$ 5a world.s
$ 5l hello.5 world.5 /arm/lib/libc.a -o hello
$ ./hello
hello world

```

The main difference with the commands shown in Section 1.4 is the use of the C library, compiled here for the ARM architecture: `/arm/lib/libc.a`. You will see in chapter 6 other ways to link libraries.

2.3.3 Inspecting objects with nm

You can use the tool `nm` to get the set of symbols defined or referenced in object files (this set is also called the name list):

```

$ nm hello.5
U exits
U fprint
U hello
T main

$ nm world.5
D hello

```

`nm` can also be used on libraries:

```

$ nm /arm/lib/libc.a | grep fprint
...
fprint.5: T fprint
...
$ nm /arm/lib/libc.a | grep exits
...
atexit.5:      T exits
...

```

U stands for undefined, T for text (a defined procedure), and D for data (a defined global). From the previous commands you can see that `hello.5` has three undefined symbol references, including `hello` which is a global data defined in `world.5`. The job of the linker is then to link those definitions to their uses.

There are other kinds of symbols and so other single letter codes used by `nm`. Those codes as well as the source of `nm` will be described fully later in Appendix E.3.

2.3.4 Dumping objects with 5l -W

Another way, more complete, to inspect the content of object files is to use the debugging option `-W` of `5l`. The effect of the option is to “dump” the instructions of the object files passed on the command line:

```

1  $ 5l -W hello.5 world.5 /arm/lib/libc.a
2  ...
3  ANAME main
4  (1) TEXT {main}00000+0(SB), $8
5  (4) MOVW $1,R0

```

```

6  ANAME hello
7  (5) MOVW ${hello}00000+0(SB),R1
8  (5) MOVW R1,8(R13)
9  ANAME fprint
10 (6) BL ,{fprint}00000+0(SB)
11 (9) MOVW $0,R0
12 ANAME exits
13 (9) BL ,{exits}00000+0(SB)
14 (10) END ,
15 ...
16 ANAME hello
17 (1) GLOBL {hello}00000+0(SB),$12
18 (2) DATA {hello}0000c+0(SB)/8,$"hell"
19 (3) DATA {hello}0000c+8(SB)/4,$"rld\n"
20 (4) END ,
21 ...
22 ANAME _main
23 (8) TEXT {_main}00000+0(SB),R1,$802
4  ANAME setR12
25 (10)MOVW ${setR12}00000+0(SB),R12
26 ANAME _tos
27 (11)MOVW R0,{_tos}00000+0(SB)
28 ...
29 (21)BL ,{main}00000+0(SB)

```

The first two fragments show the instructions of `hello.5`, from Line 3 to Line 14, and `world.5`, from Line 16 to Line 20. The output is very similar to the assembly sources of Section 2.3.1. This is normal since under Plan 9 an object file is essentially the serialized form of the abstract syntax tree of an assembly source. See the ASSEMBLER book [Pad15a] if you do not understand the opcodes or pseudo-opcodes such as `ANAME`.

Symbol references are enclosed in braces, e.g., `main` Line 4, followed by the *symbol value* in hexadecimal, e.g., `00000` Line 4. The symbol value corresponds normally to the resolved memory address of the symbol. But, almost all those values and addresses are null when using `-W` because the option dumps instructions while the object files are read into memory, at the beginning, and so when the linker has not yet resolved any of those symbols.

Once a global has been declared, e.g., `hello` Line 17, the value of its symbol is then (ab)used to store its size, here `0000c` (12) Line 18. Later you will see that the symbol value will contain the resolved address of the global.

The last fragment, starting at Line 22, shows the instructions of a procedure called `_main`. The linker `5l` is looking by default for such a procedure for the entry point of the executables it generates (unless the `-E` option is used), which is why it is included. The reason for a default called `_main`, and not `main`, even though the entry point of C programs is `main`, is because `_main()` is normally a procedure defined in the C library. This procedure, written in assembly, does some core initializations and then calls `main()`, as shown by the last dumped instruction Line 29. `5l` is usually called to link C programs, and the C compiler `5c` assumes a few core initializations has been done before the call to `main()`, hence the choice of `_main` as the default entry point.

For more information on `5l -W` see Appendix A.3.

2.3.5 Debugging machine code generation with `5l -a`

You can also display the generated machine code by using the debugging option `-a` of `5l`:

```

1  $ 5l -a hello.5 world.5 /arm/lib/libc.a
2
3 00001020:          (1) TEXT {main}01020+0(SB), $8
4 00001020: e52de00c (1) MOVW.W R14, -12(R13)
5 00001024: e3a00001 (4) MOVW $1, R0
6 00001028: e59f1884 (5) MOVW ${hello}00014+0(SB), R1
7 0000102c: e58d1008 (5) MOVW R1, 8(R13)
8 00001030: eb0000bb (6) BL , {fprintf}01324(BRANCH)
9 00001034: e3a00000 (9) MOVW $0, R0
10 00001038: eb00009a (9) BL , {exits}012a8(BRANCH)
11
12 0000103c:          (8) TEXT {_main}0103c+0(SB), R1, $80
13 0000103c: e52de054 (8) MOVW.W R14, -84(R13)
14 00001040: e59fc870 (10) MOVW ${setR12}00ffc+0(SB), R12
15 00001044: e50c0fd0 (11) MOVW R0, {_tos}0002c+0(SB)
16 ...
17 00001068: ebffffec (21) BL , {main}01020(BRANCH)
18 ...
19
20 000012a8:          (915) TEXT {exits}012a8+0(SB), R1, $16
21 000012a8: e52de014 (915) MOVW.W R14, -20(R13)
22 ...
23
24 00001324:          (874) TEXT {fprintf}01324+0(SB), R0, $20
25 00001324: e52de018 (874) MOVW.W R14, -24(R13)
26 ...
27
28 00006b50: e49df03c (890) MOVW.P 60(R13), R15

```

The first column contains the memory address in hexadecimal where the code will be loaded (e.g., 01020 Line 4). Then comes the 4 bytes in hexadecimal of the generated ARM instruction (e.g., e52de00c). Indeed ARM uses fixed-length instructions of 4 bytes. The third column contains the (global) line number in parenthesis of the instruction in the original source file (assembly or C)². Finally, the last column displays the disassembled instruction. As opposed to the previous section, the symbol values are not null anymore and contain now the resolved memory address of the symbol, e.g., 01324 Line 8 for the `fprintf` symbol. The output contains also the `TEXT` pseudo-instructions to better see the procedure boundaries, e.g., Line 3 and Line 12.

The first instruction Line 4 starts at 01020 which is equal to 4128. Indeed, as said previously in Section 2.2, the kernel loads executables after the first page (4096) and includes the `a.out` header which is using 32 bytes. Thus, the code starts at the memory address 4128.

Note that even though it looks like there are two instructions at 01020, with Line 3 and Line 4, the first instruction (`TEXT ...`) is a pseudo-instruction which got transformed by the linker in the ARM instruction Line 4 (`MOVW.W ...`). The two numbers worth chasing down are the `8` in the `TEXT` declaration and the `-12` in the generated `MOVW.W`. The source said `main` needs 8 bytes of local frame (two words). The linker added one more word (+4) to save the link register `R14`, making the total frame size 12 bytes. The single instruction `MOVW.W R14, -12(R13)` does two things at once, courtesy of the `.W` writeback flag: it stores `R14` at `R13 - 12` and then writes `R13 - 12` back into `R13`. The link register is saved and the stack pointer is bumped down in one instruction, where a less expressive ISA would need a `SUB` plus a `STR`.

Indeed, `TEXT` is an assembly directive introducing a symbol, here `main`, and symbols are resolved in concrete addresses by the linker. The generated machine code contains only concrete addresses, no symbols (except for

²See Section 10.4.1 for more information about line numbers.

debugging purposes as explained in Chapter 10). So, the reference to `main` Line 17 is resolved to the concrete address 01020. In the same way, the calls to `fprint` and `exits` Line 8 and 10 are fully resolved in respectively the memory addresses 01324 (Line 25) and 012a8 (Line 21).

The symbol value of `hello` Line 6 is 00014 which seems incorrect. Indeed, data should be stored after the code section and so the resolved address of `hello` should be beyond 06b50 (the address of the last instruction Line 28). But, the symbol value for data symbols contains the resolved address of the symbol as *an offset to the start of the data section*. The rationale for this decision will be explained later in Chapter 7. Thus, the final address of `hello` is 014 + `INITDAT` where `INITDAT` is the address where starts the data section.

For more information on `5l -a` see Appendix A.5.

2.3.6 Inspecting executables with `nm`

`nm` can also be used on executables. In that case `nm` not only displays the list of symbols, it also displays their addresses (in hexadecimal) when loaded in memory. Here it is used with the `-n` option to sort by addresses:

```
1 $ nm -n 5.out
2     1020 T main
3     103c T _main
4     ...
5     110c T _div
6     1168 T _mod
7     ...
8     12a8 T exits
9     ...
10    1324 T fprint
11    ...
12    6b58 T etext
13    7000 d onexlock
14    7000 D bdata
15    7010 D argv0
16    7014 D hello
17    7020 d _exitstr
18    ...
19    7940 D edata
20    7b44 B onex
21    7c50 B end
22    7ffc D setR12
```

You find the same resolved memory addresses we saw in the previous section with `5l -a`, e.g., 12a8 for `exits` Line 8, or 1324 for `fprint` Line 10.

You can also now see the addresses of the data symbols. `hello` Line 16 is at the address 7014. The data section starts at 7000 as indicated by the *special symbol* `bdata` (begin data) Line 14. So, `INITDAT` is 7000 which confirms that the address of `hello` is indeed 14 + `INITDAT` (7014) as mentioned in the previous section.

The symbol after `hello` is `_exitstr` Line 17 at address 7020. This confirms that `hello` is using 12 bytes ($0x7020 - 0x7014 = 0xc = 12$) as said in the source of `world.s` in Section 2.3.1.

The linker defines a set of special symbols: `etext` (end text) Line 12, `bdata` (begin data) Line 14, `edata` (end data) Line 19, and finally `end` Line 21. Those symbols are used only for their addresses and allow a form of *reflection* on the structure of the executable and its sections as explained in Section 7.6.

Another special symbol, `setR12` Line 22 plays a complex role in `5l`. This symbol is notably used by `_main` as indicated by the output of `5l -W` and `5l -a` you saw in the previous sections. Its role and its relationship with the pseudo-register `SB` will be explained later in Chapter 7.

2.4 Input object format

The object file is now the input, as opposed to the ASSEMBLER book [Pad15a] where it was the output. The format of ARM object files is actually fully described in the ASSEMBLER book [Pad15a] so I will not repeat those explanations here. Section 5.1 contains a figure summarizing this format though, which will be useful to understand the code of Chapter 5 which loads object files.

Section 2.3.4 contains the dump of a few object files and shows, as I said previously, that an object file is essentially the serialized form of the abstract syntax tree of an assembly source.

2.5 Output executable format: a.out

Plan 9 is using the very simple a.out executable format. So, 5l by default generates executables in this format. This can be changed with the -H option. Other formats will be described in Section 12.5

The file include/exec/a.out.h contains a formal description of the a.out header:

```
(struct Exec 24)≡ (270i)
// a.out header
struct Exec
{
    long magic; /* magic number */

    long text; /* size of text segment */
    long data; /* size of initialized data */
    long bss; /* size of uninitialized data */

    long syms; /* size of symbol table */

    // virtual address in [UTZERO+sizeof(Exec)..UTZERO+sizeof(Exec)+text]
    long entry; /* entry point */

    long _unused;
    // see a.out.h man page explaining how to compute the line of a PC
    long pcsz; /* size of pc/line number table */
};
```

Figure 2.3 illustrates the format of a.out executables. An a.out header is made of 8 longs (32 bytes) used as follows:

- The first four bytes in `Exec.magic` contains a *magic number* identifying the file as an a.out executable as well as an ARM executable. This magic number is recognized by tools such as `file` and by the kernel loader which will load only files having this magic number *signature*.
- Then comes the size of the text and data sections. The format of the ARM machine instructions in the text section will be described in Chapter 9.
- `Exec.bss` contains the size of the BSS section which corresponds to uninitialized data. “BSS” stands for *Block Started by Symbol*, the name of a pseudo-op in the IBM 704 assembler from the mid-1950s. The name has long outlived its meaning; today it just means “the uninitialized data section”. The actual values for those data is not defined in the executable, as opposed to the data section, but the loader must still reserve space in memory for those data. The following C code shows which section will be used depending on the declaration style of a variable:

```
int global = 1; // Data section
int another;   // BSS section
```

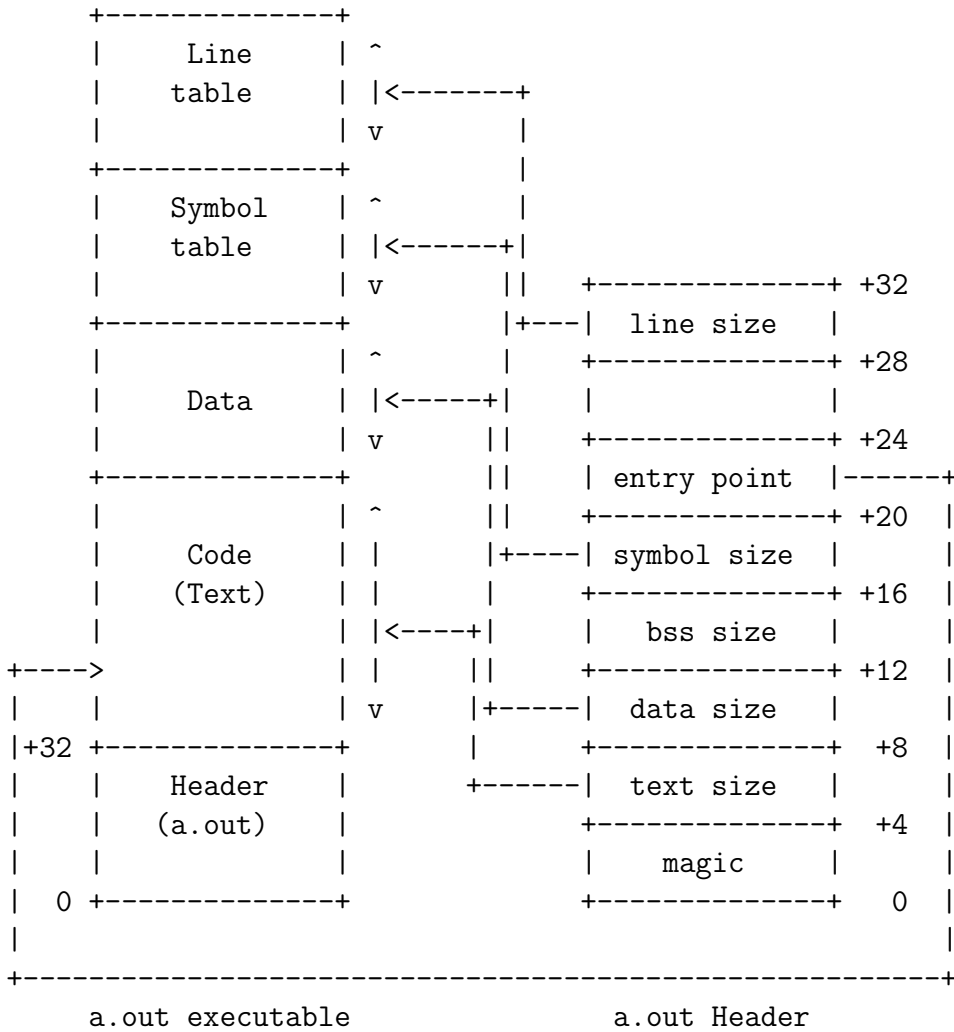


Figure 2.3: a.out executable format.

- `Exec.syms` and the executable symbol table will be discussed in Chapter 10.
- `Exec.entry` contains the memory address of the entry point of the program. This address is used by the kernel loader to start the program. It is usually the address of the `_main` procedure of the C library (unless the `-E` option is used). So, for the `hello` executable of the previous section, the value of the entry point is `0x103c`. The entry point can also be the address of the `_mainp` procedure if profiling is enabled as explained in Chapter 11.
- Finally `Exec.pcsz` and the line table will be discussed in Chapter 10.

Note that the `Exec` structure is actually not used by the code of 51 for *writing* the header. Instead, the function `asmb()`, which you will see in Section 4.4, outputs directly the bytes of the header with a series of calls to the `lput()` (for output long) utility function. The `Exec` structure is used though in programs which are *reading* the header of executables. There are many programs which need to understand the format of executables, e.g., the debugger `db`, the profiler, and small utilities like `nm`. Under Plan 9, all those programs use a common library called `libmach` which defines `Exec` as well as many other data structures and functions. Appendix E discusses the `libmach` interface and the code of utilities such as `nm`, but the code of `libmach` itself will be shown in the `DEBUGGER` book [Pad16c].

2.6 Code organization

Table 2.1 presents short descriptions of the source files used by 51, together with the main entities (e.g., structures, functions, globals) the file defines, and the corresponding chapter in this document in which the code contained in the file is primarily discussed.

Function	Ch.	File	Entities	LOC
data structures and constants	3	l.h	Sym ²⁷⁹ Section ^{33b} Instr ²⁷⁹	471
globals	3	globals.c	hash ^{31a} firstp ^{36a} datap ^{36e} pc ^{53a}	168
entry point	4	main.c	main() ^{246j}	301
executable generation	4	asm.c	asmb() ^{45a}	222
data section generation	4	datagen.c	datblk() ^{49b}	245
loading objects	5	obj.c	ldobj() ⁵⁵	834
loading libraries	6	lib.c	loadlib() ^{76e}	188
building the code instructions graph	7	pass.c	patch() ⁸²	372
rewriting virtual opcodes	7	noop.c	noops() ⁸⁷	415
laying out the code and data sections	7	layout.c	dotext() ^{94b} dodata() ^{92c}	432
ARM code-generation data structures	8	m.h	Optab ^{99a} Oprange ^{100b} Operand_class ^{101a}	159
ARM opcodes lookup table	8	optab.c	optab ^{99b}	293
ARM opcodes lookup function	8	span.c	oplook() ^{102c} buildop() ^{104c}	670
ARM instructions generation	9	codegen.c	asmout() ^{108b}	1344
debugging support	10	debugging.c	asmsym() ^{148b} asmlc() ^{160c}	244
location information	10	hist.c	addhist() ^{155a}	55
profiling support	11	profile.c	doprof1() ¹⁶³ doprof2() ^{166a}	264
dynamic linking	12	dynamic.c	asmdyn() ^{177f}	412
float instructions	12	float.c	ieeedtof() ^{189c}	92
ELF constants	12	elf.h	Ehdr32sz	21
ELF generation	12	elf.c	elf32()X	176
dumpers	A	fmt.c	prasm() ^{219b} Pconv() ^{219d} Aconv() ^{220a}	355
error management	B	error.c	diag() ^{229d} errexit() ^{229b}	45
input/output buffer	D	io.c	buf ^{234b} wput() ^{234e}	138
utilities	D	utils.c	malloc() ^{233a}	226
Total				8142

Table 2.1: Chapters and source files of 51.

An important file not included in Table 2.1 is `include/obj/5.out.h`. It contains the definition of the ARM assembly opcodes used by 51. It is not included here because its content is described instead in the ASSEMBLER book [Pad15a].

2.7 Software architecture

Figure 2.4 describes the main control flow of 51, whereas Figure 2.5 describes the main data flow of 51. The main steps of the linking process of 51 are as follows:

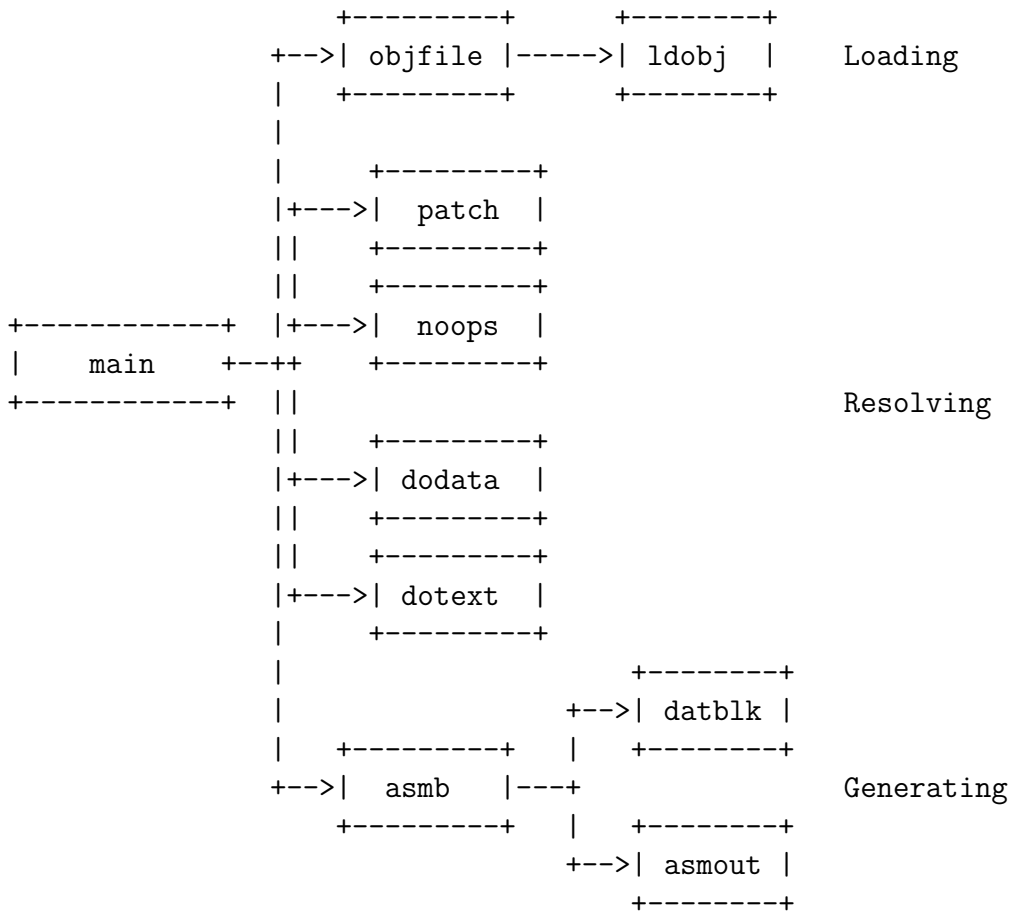


Figure 2.4: Control flow diagram of 5l.

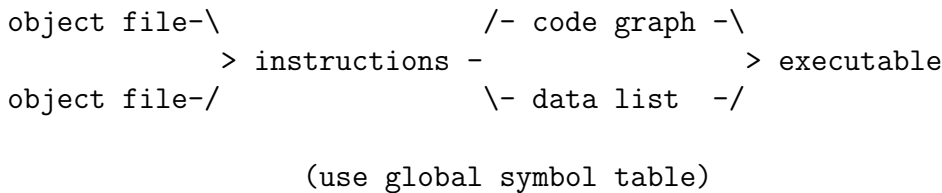


Figure 2.5: Data flow diagram of 5l.

1. *Load* (read) the object files and libraries (via `objfile()`^{44a}) and concatenate their code and data in different sections in memory (via `ldobj()`⁵⁵)
2. *Resolve* symbol references (via `patch()`⁸², ...)
3. *Generate* the executable with its header and machine code (via `asmb()`^{45a})

I will now explain briefly the control flow of 5l, starting from the left of Figure 2.4. After some basic command line processing and initializations, the function `main()`^{246j} first calls `objfile()` for each object files and libraries to link. `objfile()` opens the object file or library and calls `ldobj()` to read its object code in memory, which is mostly a *list of instructions*. Doing so, `ldobj()` modifies also the globals `firstp`^{36a} and `datap`^{36e} containing respectively the list of code instructions and data instructions. Thus, all the code and data in the object files are concatenated together in the appropriate *section*.

`ldobj()` increments also a global `pc`^{53a} after each code instruction read. `pc` represents the current value of the *virtual program counter*. Each code instruction in memory has a field `Instr.pcX` with the value of `pc` at

the moment the instruction was read. Finally, `ldobj()` populates a *symbol table* called `hash`^{31a}, which keeps track of different properties of symbols. One of this property, `Sym.valueX`, contains initially a virtual program counter for symbols corresponding to procedures, and a size for symbols corresponding to globals. This will be useful to resolve symbol references. This same field will contain at the end the resolved memory address of the procedure or global.

After loading the object files, `main()` calls a series of functions (`patch()`, `noops()`⁸⁷, etc) which use the globals `firstp`, `datap`, and `hash` to process or rewrite list of instructions. First, `patch()` transforms the list of code instructions in a *graph of instructions* where branching instructions have a field `Instr.condX` pointing to the appropriate target instruction. To build this graph `patch()` is leveraging the symbol table `hash` computed previously to find the virtual program counter corresponding to a certain procedure symbol (in its `Sym.valueX` field) and *index* all `Instr.pcX` to find the instruction pointer (`Instr*`) corresponding to a certain virtual program counter.

Once the graph has been built, there is no more need for the notion of virtual program counter. Every code references (symbols or absolute jumps) in branching instructions have been resolved. This allows in turn to transform safely some *virtual instructions* (see the ASSEMBLER book [Pad15a]) such as `RET`, `DIV`, or `NOP` in machine instructions. Indeed, `noops()` can replace a virtual instruction by multiple instructions or even delete the instruction without any consequence on the other branching instructions (as long as it maintains carefully the `Instr.condX` pointers pointing to the original instruction). Before the graph of instructions, inserting or deleting an instruction would have forced to assign a new virtual program counter to each instruction and to relocate every branching instructions.

Once 51 has the graph of machine code instructions and the set of globals declarations, it can start to *lay out* the code and data. `dodata()`^{92c} assigns a memory address in `Sym.valueX` to each globals in the symbol table as an offset to the start of the data section. Then, `dotext()`^{94b} does a similar thing for the code instructions. For those code instructions `Instr.pcX` will contain now the final code address of the instruction (and not a virtual program counter anymore).

At this point, the size of the code and data sections are known and stored respectively in the globals `textsize`^{94a} and `datsize`^{92a}. 51 can finally call `asmb()` to generate the executable and its header. This function uses two helpers functions: one to fill the data section `datblk()`^{49b} and one to generate ARM instructions in the code section `asmout()`^{108b}.

2.8 Book structure

You now have enough background to understand the source code of 51. The rest of the book is organized as follows. I will start by describing the core data structures of 51 in Chapter 3. Then, I will use a top-down approach in Chapter 4, and, starting from `main()`^{246j}, I will present the code, or a high-level view of the code, of some of the main functions (e.g., `asmb()`^{45a}). The following chapters will describe the main components of the linking pipeline: Chapter 5 will present the code to load object files, Chapter 6 the code to load libraries, Chapter 7 the code to resolve symbols, Chapter 8 the code generation preparation, and finally Chapter 9 the ARM machine code generator. In Chapter 10 I will describe the code responsible for adding debugging support in 51, which for instance adds line information in the executable. You can then know, when debugging a binary program, to which original line and which source file an instruction in the binary comes from, or what is the original name of the procedure containing this instruction. In a similar way, Chapter 11 will describe the code for adding profiling support in 51. Chapter 12 presents other features of the linker that I did not present before to simplify the explanations, for instance the support for dynamic linking or the ARM code generation of instructions involving floats. Finally, Chapter 13 concludes and gives pointers to other books in the Principia Softwarica series.

Some appendices present the code of non-functional properties: the code to help debug 51 itself in Appendix A, the code to manage errors in Appendix B, and the code which profiles 51 itself in Appendix C. Appendix D contains the code of generic utility functions used by 51 but which are not specific to 51. Ap-

pendix **E** describes the code of small programs such as `nm` and `ar` which are related to the linker.

Chapter 3

Core Data Structures

Show me your code and conceal your data structures, and I shall continue to be mystified. Show me your data structures, and I won't usually need your code; it'll be obvious.

Fred Brooks

In this chapter I will present the core data structures of 51, which are essentially: a symbol table keeping track of the memory address and section of symbols, the abstract syntax tree (AST) of the instructions contained in object files, and a set of globals pointing to lists of instructions.

3.1 Symbols and hash table

One of the main job of a linker is to resolve symbol references by linking uses of a symbol to its definition. The *symbol table*, which keeps track of those symbols, is thus a central data structure of 51. The structure below represents a symbol and its properties. It essentially associates a *key* to a *value*:

```
<struct Sym 30>≡ (279)
struct Sym
{
    // The key
    // ref_own<string>
    char    *name;
    // 0 for global symbols, object file id for private symbols
    short   version;

    // The generic value,
    // e.g., virtual pc for a TEXT procedure, size for GLOBL
    long    value;

    <Sym section field 33a>

    <Sym other fields 68c>
    // Extra
    <Sym extra fields 31c>
};
```

The key is made of a pair with the name of the symbol and a *version*. The version is used to differentiate *private symbols* (a.k.a static symbols) in different object files using the same name, e.g., `tmp<>` in `foo.5` and `tmp<>` in `bar.5`. See the ASSEMBLER book [[Pad15a](#)] for more information about private symbols. A version is a unique integer representing an object file, e.g., 1 for `foo.5` and 2 for `bar.5`, so the two previous symbols can be designated respectively by `tmp.1` and `tmp.2`. Public symbols have their version set to 0.

The meaning of `Sym.valueX` depends on the kind of the symbol. It also depends on the step in the linking pipeline. At the beginning `Sym.valueX` contains a virtual program counter for `TEXT` symbols, and a size for `GLOBL` symbols. At the end it contains a resolved memory address in the code section for procedures and a resolved offset to the start of the data section for globals.

The symbol table itself is represented by a global hash table called `hash`. It makes sense to use a global because the symbol table will be accessed by different components of the linking pipeline. One way to implement a hash table in C is to use a big array of lists, also known as an array of buckets:

Concretely, after loading two object files that both define a private `tmp` and a public `memcpy`, the table looks roughly like this (the bucket indices are hypothetical):

```

hash[NHASH]
+----+
0 |   |
+----+
|...|
+----+
42 | *---->+-----+      +-----+
+----+ | "memcpy" |      | "tmp"   |
|...|  | version=0 |---->| version=2 |----> S (nil)
+----+ | type=STEXT |      | type=STEXT |
917 | *---->+-----+      +-----+
+----+ | "tmp"     |
|...|  | version=1 |----> S (nil)
+----+ | type=STEXT |
      +-----+

```

Two observations. First, public and private symbols coexist in the same array: only the `(name, version)` pair is the key, so collisions between versions fall onto their chain-walk anyway. Second, the `link` field is embedded in `Sym`²⁷⁹ rather than stored in a separate `Bucket` struct—this intrusive-list idiom is the same trick used by 5a’s symbol table and by most Plan 9 kernel data structures, and saves one allocation per insert at the cost of making a `Sym` belong to exactly one chain.

```

⟨global hash linker 31a⟩≡ (283b)
// hash<Sym.name * Sym.version, ref_own<Sym>> (next = Sym.link in bucket)
Sym* hash[NHASH];

```

Uses `NHASH`.

```

⟨constant NHASH linker 31b⟩≡ (278c)
NHASH      = 10007,

```

One way to implement a list of something in C is to embed in this something a `link` field pointing to the next element in the list:

```

⟨Sym extra fields 31c⟩≡ (30)
// list<ref<Sym>> (from = hash)
Sym*   link;

```

The end of the list is represented by the null pointer:

```

⟨constant S 31d⟩≡ (279)
#define S      ((Sym*)nil)

```

The main interface to the symbol table is the function `lookup()` which internally uses the global `hash`. It takes a symbol name and a version, forming a full key, and returns the `Sym` in the symbol table `hash` associated with this key, or a new symbol if the key was not found:

<function lookup 32a>≡ (285a)

```
Sym*
lookup(char *symb, int v)
{
    Sym *sym;
    long h;
    int len;
    <lookup() other locals 32b>

    <lookup() compute hash value h of (symb, v) and len 32c>

    // sym = hash_lookup((symb, v), h, hash)
    for(sym = hash[h]; sym != S; sym = sym->link)
        if(sym->version == v)
            if(memcmp(sym->name, symb, len) == 0)
                return sym;

    // else
    <lookup() if symbol name not found 32d>
}
```

Uses S 33b.

<lookup() other locals 32b>≡ (32a)

```
char *p;
int c;
```

<lookup() compute hash value h of (symb, v) and len 32c>≡ (32a)

```
// h = hashcode(symb, v);
// len = strlen(symb);
h = v;
for(p=symb; *p; p++) {
    c = *p;
    h = h+h+h + c;
}
len = (p - symb) + 1;
h &= 0xffffffff;
h %= NHASH;
```

Uses NHASH.

<lookup() if symbol name not found 32d>≡ (32a)

```
sym = malloc(sizeof(Sym));
sym->name = malloc(len + 1); // +1 again?
memmove(sym->name, symb, len);
sym->version = v;

sym->value = 0;
sym->type = SNONE;
sym->sig = 0;

// add_hash(sym, hash)
sym->link = hash[h];
hash[h] = sym;
```

```
<lookup() profiling 231c>
return sym;
```

Uses SNONE 33b and malloc() 233a.

3.2 Section

Another important property of a symbol is in which *section* it will reside:

```
<Sym section field 33a>≡ (30)
//enum<Section>
short type;
```

```
<enum Section(arm) 33b>≡ (279)
enum Section
{
    SNONE = 0,

    STEXT,
    SDATA,
    SBSS,

    SXREF,
    <Section cases 156d>
};
```

You can see enumerations above corresponding to the usual text, data, and BSS sections of a program. Note that there is no `SSTACK` enumeration as 51 cares only about the executable file here, not the memory image of a program. The stack content and its maximum size is set initially by the kernel, not the executable.

`SXREFX` is used to represent an *unknown reference*. When object files are loaded in `ldobj()`⁵⁵, symbol references in operands are looked up in the symbol table. If the symbol has not been defined yet, a new symbol is created with its section set to `SXREFX`. Once a `TEXT` or `GLOBL` directive introduces the symbol, its section can be adjusted. At the end of the linking process there should be no more symbols with `SXREFX`. This is in fact checked by the `undef()`^{52d} function I will describe later.

3.3 Opcode and Operand

An object file contains essentially a list of instructions where each instruction is made of an *opcode* with possibly 1, 2, or 3 *operands*. The `Opcode` type, `Operand_kind` type, and a few register aliases such as `REGPC`, all used by 51, are all defined in `include/obj/5.out.h`. Those types are fully described in the `ASSEMBLER` book [Pad15a]. Dumpers for those types are described in Appendix A.1 if you need to refresh your memory:

- see `Aconv()`^{220a} for the dumper of `Opcode` (A because opcodes use the `Axxx` syntax, e.g., `ASUB`),
- see `Dconv()`^{220b} for the dumper of `Operand_kind` (D because operand kinds use the `Dxxx` syntax, e.g., `D_CONST`).

Remember that the opcodes in object files do not correspond exactly to ARM opcodes. They are mostly a superset because they also include opcodes for *pseudo-instructions* such as `ATEXT` and `AGLOBL`, as well as opcodes for *virtual instructions* such as `ARET` and `AMOVW`.

`include/obj/5.out.h` does not define an `Operand` type though as the assembler and linker have slightly different needs for this data structure. Here is the `Operand` type used by 51:

```
<struct Adr(arm) 33c>≡ (279)
struct Adr
{
    // enum<Operand_kind> (D_NONE by default)
    short type;

    // switch on Operand.type
    union {
```

```

    long    offset;
    Ieee*   ieee;
    char*   sval;
};

// option<enum<Register>> None = R_NONE
short    reg;

<Adr other fields 34a>
};

```

`Operand`²⁷⁹ is until now almost identical to the `Operand` type used by 5a and described in the ASSEMBLER book [Pad15a], except for the way float operands are represented (`Ieee*` versus a `double`). The *operand kind* is also stored in `Operand.typeX`, e.g. `D_CONST`, `D_REG`, `D_OREG`. `Operand.regX` is used again for operands involving registers. You will see later though additional fields specific to the linker.

Operands involving symbols use an `Operand.sym` field and the *symbol kind* is also stored in a `Operand.symkind` field, e.g., `N_EXTERN`, `N_PARAM`, `N_LOCAL` (which are enumerations defined also in `include/obj/5.out.h`):

```

<Adr other fields 34a>≡ (33c) 106b▷
// enum<Sym_kind>
short    symkind;
// option<ref<Sym>> (owner = hash)
Sym*     sym;

```

Symbols references in the assembly language of 5a, as well as in object files, are all represented as “offsets” to *pseudo-registers*:

- procedures and globals use the `SB` (*static base*) pseudo-register, e.g., `main(SB)`, `hello(SB)`. Their symbol kind is `N_EXTERN` (or `N_INTERN` for private symbols such as `foo<>(SB)`).
- parameters are accessed via the `FP` (*frame pointer*) pseudo-register, e.g., `p+4(FP)`. Their symbol kind is `N_PARAM`.
- locals use the `SP` (*stack pointer*) pseudo-register, e.g., `v-8(SP)`. Their symbol kind is `N_LOCAL`.

The symbol name for parameters and locals is actually optional. What matters is the additional offset number for code generation. The symbol name is mostly a comment. But, those symbols are still saved in the executable symbol table as you will see in Chapter 10, so they can be leveraged by tools such as debuggers.

3.4 Instruction

The type below connects an opcode with its operands to form a full *instruction*:

```

<struct Prog(arm) 34b>≡ (279)
struct Prog
{
    //enum<Opcode>
    byte    as;

    // operands
    Adr from;
    Adr to;

    <Prog other fields 35a>

    // Extra
    <Prog extra fields 35e>
};

```

Most opcodes use two operands, transferring or processing data *from* a source operand *to* a target operand, e.g., `MOVW R1, (R2)`. Some opcodes use three operands, e.g., `ADD R1, R2, R3`. In that case the middle operand is always a register, hence the more specialized field below representing this middle operand:

```
⟨Prog other fields 35a⟩≡ (34b) 35b▷
// option<enum<Register>>, None = R_NONE
short reg;
```

All ARM instructions have a *conditional execution* (see the ASSEMBLER book [Pad15a] or EMULATOR book [Pad15b]):

```
⟨Prog other fields 35b⟩+≡ (34b) <35a 35c>
// enum<Instr_cond>
byte scond;
```

For debugging purpose, each instruction has also a *global line number*:

```
⟨Prog other fields 35c⟩+≡ (34b) <35b 35d>
long line;
```

For more information on line numbers and debugging see Chapter 10.

Up until now, `Instr`²⁷⁹ is very similar to the type used in the ASSEMBLER book [Pad15a] to represent an instruction¹. The following field is new and specific to the linker. It initially stores the *virtual program counter* of the instruction (if the instruction is a code instruction, e.g., `ADD`):

```
⟨Prog other fields 35d⟩+≡ (34b) <35c 35f>
// virtual program counter, and then real program counter
long pc;
```

The virtual program counter is incremented after each code instruction is read in `ldobj()`⁵⁵. Later in the linking process `Instr.pcX` will be assigned a *real program counter* which will be a multiple of 4 as the ARM uses fixed-length instructions of 4 bytes.

Later in the linking process, another field, `Instr.condX`, will be used to build a *graph of instruction*. Branching instructions will have their `Instr.condX` field points to the target instruction.

```
⟨Prog extra fields 35e⟩≡ (34b) 36b▷
// option<ref<Prog>> for branch instructions
// (abused to list<ref<Prog>> (from = textp for TEXT instructions))
// (abused also for instructions using large constants)
Prog* cond;
```

Finally, the field below is used by different algorithms to temporarily *mark* instructions in the graph of instructions:

```
⟨Prog other fields 35f⟩+≡ (34b) <35d 83d>
//bitset<enum<Mark>>
short mark;
```

I will describe gradually the different kinds of marks in this document:

```
⟨enum Mark(arm) 35g⟩≡ (279)
/* mark flags */
enum Mark {
    ⟨Mark cases 88a⟩
};
```

3.5 List of instructions

In this section, you will see a set of globals keeping track of different lists of instructions.

¹5a actually uses a set of parameters to the `outcode()` function to represent an instruction. Each parameter is identical to one of the field of `Instr`.

3.5.1 Code instructions: firstp/lastp

`firstp` points to the first instruction of the list of all code instructions (everything except the `GLOBL` and `DATA` pseudo-instructions):

```
<global firstp 36a>≡ (283b)
// list<ref_own<Prog>> (next = Prog.link)
Prog* firstp;
```

The instructions are linked together with the following field:

```
<Prog extra fields 36b>+≡ (34b) <35e
// list<ref<Prog>> (from = firstp or datap)
Prog* link;
```

The end of the list is represented by the null pointer:

```
<constant P 36c>≡ (279)
#define P ((Prog*)nil)
```

5l can quickly access the end of the list by using the following global:

```
<global lastp 36d>≡ (283b)
// ref<Prog> (end from = firstp)
Prog* lastp;
```

3.5.2 Data instructions: datap

`datap` keeps track of the list of `DATA` pseudo-instructions:

```
<global datap 36e>≡ (283b)
// list<ref_own<Prog>> (next = Prog.link)
Prog* datap = P;
```

Uses `P 36c` and `datap 36e`.

`datap` points to the last data instruction. Note that `GLOBL` instructions are not part of this list. Instead, the `GLOBL` instructions, when read, modify symbols in the symbol table.

Thanks to `firstp36a` and `datap` all the code and data instructions are separated in different *sections*.

3.6 Current instructions: curtext/curp

During the linking process, 5l will iterate many times over the list of instructions. It is useful, for error messages, to store in globals in which procedure 5l currently is (`curtext`) and which instruction 5l is currently processing (`curp`):

```
<global curtext 36f>≡ (283b)
//option<ref<Prog>> where Prog.as == ATEXT
Prog* curtext = P;
```

Uses `P 36c` and `curtext 36f`.

```
<global curp 36g>≡ (283b)
// option<ref<Prog>>
Prog* curp;
```

Many algorithms will set `curtext` as follows while iterating over instructions:

```
<adjust curtext when iterate over instructions p 36h>≡ (184c 183e 160c 94b 87 85c 82 47b)
if(p->as == ATEXT)
    curtext = p;
```

Uses `curtext 36f`.

The `TNAME` macro (for Text `NAME`) is used as an argument to a few error management functions to display the name of the current procedure when there is one:

```
<constant TNAME(arm) 37a>≡ (279)  
#define TNAME (curtext && curtext->from.sym ? curtext->from.sym->name : noname)
```

```
<global noname linker 37b>≡ (290)  
char *noname = "<none>";
```

Uses `noname 37b`.

Chapter 4

Main Functions

I now switch to a top-down approach where I describe the main functions of 5l, starting from `main()`^{246j} down to `asmb()`^{45a} which generates the executable.

4.1 `main()`

Before showing the code of `main()`^{246j} I first introduce a few globals set by `main()`. A common pair of globals in Plan 9 code are `thechar` and `thestring` which both represent the current architecture. As said in the introduction, Plan 9 by convention represents architectures with a single character: '0' is MIPS, '5' is ARM, '8' is x86, etc. This character is used by 5l for the filename extension of object files (e.g., `hello.5`):

```
<global thechar 38a>≡ (283b)
char thechar;
```

`thestring` contains the longer, more readable, version of the architecture, e.g., "arm" for '5'.

```
<global thestring 38b>≡ (283b)
char* thestring;
```

This is used by 5l to find the architecture-specific library files in `/arm/lib/` as explained in Section 6.3.

Another important global is the name of the executable, which by default is `5.out` and can be modified with the `-o` option:

```
<global outfile 38c>≡ (283b)
char* outfile;
```

The file descriptor of the created executable file will be stored in the following global:

```
<global cout 38d>≡ (283b)
fdt cout = -1;
```

Uses `cout` 38d.

I can now present the code of `main()`, the entry point of 5l. The most important part is the chunk below the `-- main functions --` comment which contains the main flow of 5l. It will be described soon in Section 4.1.4:

```
<function main(arm) 38e>≡ (292d)
void
main(int argc, char *argv[])
{
    <main() locals(arm) 74a>

    thechar = '5';
    thestring = "arm";

    outfile = "5.out";
```

```

⟨main() debug initialization(arm) 218d⟩

ARGBEGIN {
⟨main() command line processing(arm) 39d⟩
} ARGEND

USED(argc);
if(*argv == nil)
    usage();

⟨main() initialize globals(arm) 39c⟩

cout = create(outfile, 1, 0775);
⟨main() sanity check cout 39b⟩

// ----- main functions -----
⟨main() cout is ready, LET'S GO(arm) 42d⟩

```

```

out:
⟨main() profile report 231a⟩
    errexit();
}

```

⟨function usage, linker 39a⟩≡

```

void
usage(void)
{
    print("usage: %s [-options] objects\n", argv0);
    errexit();
}

```

(292d)

Uses `errexit()` 229b.

⟨main() sanity check cout 39b⟩≡

```

if(cout < 0) {
    diag("cannot create %s: %r", outfile);
    errexit();
}

```

(38e)

The error management functions `diag()`^{229d} and `errexit()`^{229b} are described in Appendix B.

⟨main() initialize globals(arm) 39c⟩≡

```

⟨main() addlibpath("/thestring/lib") or ccroot 74d⟩
⟨main() set HEADTYPE, INITTEXT, INITDAT, etc 41a⟩
⟨main() set INITENTRY 42c⟩

```

(38e) 43c▷

4.1.1 Arguments processing

I have mentioned before the `-o` option:

⟨main() command line processing(arm) 39d⟩≡

```

case 'o':
    outfile = ARGF();
    break;

```

(38e) 40b▷

I will introduce the other command-line options gradually in this document. An important set of options deals with the executable format and are presented below.

4.1.2 Executable format choice: 5l -H

One of the most important option of 5l is `-H<num>` which modifies the global `HEADTYPE` recording the format of the executable:

```
<global HEADTYPE 40a>≡ (283b)
```

```
// option<enum<Headtype>>, None = -1
short HEADTYPE = -1; /* type of header */
```

Uses `HEADTYPE 40a`.

```
<main() command line processing(arm) 40b>+≡ (38e) <39d 41c>
```

```
case 'H':
    a = ARGV();
    if(a)
        HEADTYPE = atolwhex(a);
    break;
```

The `atolwhex()`^{235c} function, described in Appendix D.4, converts a string representing a number into an integer. `atolwhex()` also handles numbers written in hexadecimal, hence the name. This is not very useful for the `-H` option, but it will be useful for other options such as `-T` as you will see later.

By default 5l uses the `a.out` format which is the executable format used by Plan 9:

```
<enum headtype(arm) 40c>≡ (279)
```

```
/*
 * -H0          no header
 * -H2 -T4128 -R4096 is plan9 format
 * -H7          is elf
 */
enum Headtype {
    H_NOTHING = 0,
    H_PLAN9 = 2, // a.k.a H_AOUT
    H_ELF = 7,
};
```

The other executable formats, e.g., ELF, will be described later in Section 12.5.

The executable format dictates the size of the header, where the code section will be loaded in memory, and how the data section will follow the code section, by setting the following globals:

```
<global HEADR 40d>≡ (283b)
long HEADR; /* length of header */
```

```
<global INITTEXT 40e>≡ (283b)
long INITTEXT = -1; /* text location */
```

Uses `INITTEXT 40e`.

```
<global INITRND 40f>≡ (283b)
long INITRND = -1; /* data round above text location */
```

Uses `INITRND 40f`.

```
<global INITDAT 40g>≡ (283b)
long INITDAT = -1; /* data location */
```

Uses `INITDAT 40g`.

```

⟨main() set HEADTYPE, INITTEXT, INITDAT, etc 41a⟩≡ (39c)
    if(HEADTYPE == -1)
        HEADTYPE = H_PLAN9;
    switch(HEADTYPE) {
⟨main() switch HEADTYPE cases(arm) 41b⟩
    default:
        diag("unknown -H option");
        errexit();
    }
⟨main() sanity check INITXXX 41e⟩
    DBG("HEADER = -H%d -T0x%lux -D0x%lux -R0x%lux\n",
        HEADTYPE, INITTEXT, INITDAT, INITRND);

```

Here are the specifics for the a.out format under Plan 9:

```

⟨main() switch HEADTYPE cases(arm) 41b⟩≡ (41a) 187g▷
    case H_PLAN9:
        HEADR = 32L;
        if(INITTEXT == -1)
            INITTEXT = 4096+32; // 1 page + a.out header = 4128
        if(INITDAT == -1)
            INITDAT = 0;
        if(INITRND == -1)
            INITRND = 4096; // 1 page
        break;

```

INITDAT is initially set to 0 but it will be modified later by `dotext()`^{94b} to contain the address of the next memory page¹ after the code section. Indeed, the memory pages for the code and data section will have different properties (see the KERNEL book [Pad14]).

You can also manually set the start of the code and data sections, which is useful when producing special binaries such as kernels or boot loaders:

```

⟨main() command line processing(arm) 41c⟩+≡ (38e) <40b 41d▷
    case 'T':
        a = ARGF();
        if(a)
            INITTEXT = atolwhex(a);
        break;
    case 'D':
        a = ARGF();
        if(a)
            INITDAT = atolwhex(a);
        break;

```

```

⟨main() command line processing(arm) 41d⟩+≡ (38e) <41c 42b▷
    case 'R':
        a = ARGF();
        if(a)
            INITRND = atolwhex(a);
        break;

```

```

⟨main() sanity check INITXXX 41e⟩≡ (41a)
    if(INITDAT != 0 && INITRND != 0)
        print("warning: -D0x%lux is ignored because of -R0x%lux\n",
            INITDAT, INITRND);

```

¹The size of the page is specified by INITRND.

4.1.3 Executable entry point: 5l -E

The entry point of a program can also be modified, with the `-E` option:

```
<global INITENTRY 42a>≡ (283b)
```

```
char* INITENTRY = nil; /* entry point */
```

Uses `INITENTRY 42a`.

```
<main() command line processing(arm) 42b>+≡ (38e) <41d 73f>
```

```
case 'E':
    a = ARGF();
    if(a)
        INITENTRY = a;
    break;
```

It is set by default to `_main` under Plan 9, for reasons explained in Section 2.3.4:

```
<main() set INITENTRY 42c>≡ (39c)
```

```
if(INITENTRY == nil) {
    INITENTRY = "_main";
    <main() adjust INITENTRY if profiling 162a>
```

```
}
<main() if rare condition do not set SXREF for INITENTRY, else 75e>
lookup(INITENTRY, 0)->type = SXREF;
```

Unless some rare conditions I will explained in Section 6.4, the symbol for the entry point (usually `_main`) is looked up². This will create a new symbol in the symbol table `hash`^{31a} as the symbol table is empty at the beginning. Then, by setting its section to `SXREFX`, the symbol for the entry point is marked as a “wanted” symbol. Hopefully the object files or libraries passed on the command line to `5l` will define this symbol. Otherwise, because of `SXREFX`, an error message will be displayed at the very end such as:

```
$ 5l no_main.5
??none??: entry not text: _main
??none??: _main: not defined
```

4.1.4 Main flow

I can finally present the main control flow of `5l` with the calls to its main components. The code below follows closely the software architecture I described in Section 2.7:

```
<main() cout is ready, LET'S GO(arm) 42d>≡ (38e)
```

```
// first empty instruction
firstp = prg();
lastp = firstp;

// Loading (populates firstp, datap, and hash)
while(*argv)
    objfile(*argv++);
<main() load implicit libraries 75d>

// skip first empty instruction
firstp = firstp->link;
if(firstp == P)
    goto out;

// Resolving
<main() resolving phase 79>
```

²Remember from Section 3.1 that the second argument of `lookup()` is a version number and all global symbols use 0 for their version.

```
// Generating (writing to cout, finally)
asmb();

// Checking
undef();
```

The code steps are as follows:

1. To *load* the object files and libraries passed on the command line (as well as possibly other “implicit” libraries as explained in Section 6.4)
2. To *resolve* symbols
3. To *generate* the executable
4. To *check* finally if there are still some undefined symbols (e.g., whether `_main` has been defined)

The first line above allocates an empty instruction with `prg()`. This first instruction is used as a *sentinel* which simplifies code modifying later `firstp/lastp`. This first empty instruction is then skipped a few lines later. `prg()` uses the global `zprg` which is set to represent an empty instruction:

```
<constructor prg 43a>≡ (285a)
Prog*
prg(void)
{
    Prog *p;

    p = malloc(sizeof(Prog));
    *p = zprg;
    return p;
}
```

Uses `malloc()` 233a and `zprg` 43b.

```
<global zprg 43b>≡ (283b)
Prog zprg;
```

```
<main() initialize globals(arm) 43c>+≡ (38e) <39c 51d>
<main() set zprg(arm) 43d>
```

```
<main() set zprg(arm) 43d>≡ (43c)
zprg.as = AGOK;
zprg.scond = COND_ALWAYS;
zprg.reg = R_NONE;
zprg.from.type = D_NONE;
zprg.from.symkind = N_NONE;
zprg.from.reg = R_NONE;
zprg.to = zprg.from;
```

All of the constants above, e.g., `R_NONE`, are defined in `include/obj/5.out.h`. The `AGOK` (God Only Knows) opcode represents an *undefined instruction*. It is recognized by some error management code in 51 (see Section 5.4.6) and used as a form of defensive programming.

The following sections will each describe one of the function mentioned in the main flow above.

4.2 Loading the objects and libraries: `objfile()`

`objfile()` takes either the name of an object file (e.g., `foo.5`), or a library (e.g., `libc.a`), and reads the object code in it. First, `objfile()` opens and reads a few bytes of the file to decide whether it is an archive or an object file by checking if those bytes match an *archive magic string* (`ARMAG`^{70a}). Then it loads the file. The code below shows only the simple case where the file is an object file. The more complex case where the file is a library will be described later in Chapter 6.

```
<function objfile 44a>≡ (290)
  /// main | loadlib -> <>
  void
  objfile(char *file)
  {
    fdt f;
    long len;
    char magbuf[SARMAG]; // magic buffer
    <objfile() other locals 71c>

    DBG("%5.2f objfile: %s\n", cputime(), file);

    <objfile() adjust file if -lxxx filename 75a>

    f = open(file, 0);
    <objfile() sanity check f 44b>

    len = read(f, magbuf, SARMAG);

    // is it a regular object? (not a library)
    if(len != SARMAG || strncmp(magbuf, ARMAG, SARMAG)){
      /* load it as a regular file */
      len = seek(f, OL, SEEK__END); // len = filesize(f);
      seek(f, OL, SEEK__START);

      // the important call!
      ldobj(f, len, file);

      close(f);
      return;
    }
    // else
    <objfile() when file is a library 72>
  }
}
```

Uses `DBG` 279 and `ldobj()` 55.

`SARMAG`^{70b} above stands for Size of ARchive MAGic string. The most important part in the code above is the call to `ldobj()`⁵⁵ which loads in memory one (opened) object file. Chapter 5 will describe `ldobj()`.

```
<objfile() sanity check f 44b>≡ (44a)
  if(f < 0) {
    diag("cannot open %s: %r", file);
    errexit();
  }
}
```

Uses `diag()` 229d and `errexit()` 229b.

4.3 Resolving symbols, computing addresses

The next step, after loading the objects, is to resolve the symbol references in those objects and to compute the final addresses of those symbols. I will delay the explanations about this step to Chapter 7 where you will see

multiple functions doing multiple passes on the set of instructions and the symbol table.

4.4 Generating the executable: `asmb()`

After the symbols are resolved, and their memory addresses computed, 51 can finally generate the executable with `asmb()`:

```
<function asmb(arm) 45a>≡ (289b)  
  /// main -> <>  
  void  
  asmb(void)  
  {  
    <asmb() locals 45b>  
  
    DBG("%5.2f asm\n", cputime());  
  
    // Text section  
    <asmb() Text section 47b>  
  
    // Data section  
    <asmb() Data section 48c>  
  
    // Symbol and Line table sections  
    <asmb() symbol and line table sections 147a>  
  
    // Header  
    <asmb() header section 46a>  
  
    cflush();  
  }
```

Uses `DBG` 279 and `cflush()` 110a.

`asmb()` ^{45a} does not take any argument. It uses the global `cout` ^{38d}, which was initialized in `main()` ^{246j} with the file descriptor of the executable file, to output data in the executable.

Note that the order of operations above may seem incorrect. The code to generate the header is at the end, which seems paradoxical. But, the header specifies the size of the sections, and so 51 first needs to know those sizes before generating the header, hence the order of operations.

The code in `asmb()` uses and modifies the local variable `OFFSET` below to point to different parts of the executable file. It will be passed to the C function `seek()` to move around in the file.

```
<asmb() locals 45b>≡ (45a) 48a▷  
  long OFFSET;
```

Because the chunks above are presented in execution order (text, data, symbols, header), it is easy to lose sight of how the pieces are arranged on disk. `asmb()` writes them out of order but seeks back to each slab's offset, so the final `a.out` file looks like this:

file offset	section
0	a.out header (HEADR = 32 bytes)
HEADR	-- magic, textsize, datsize, bsssize, symsize, entry, 0, lcsiz
	TEXT section (textsize bytes)
	ARM instructions, then literal pools, then inlined strings in text segment

HEADR+textsize	DATA section (datsize bytes)
	initialised globals (no BSS: it is
	zero-filled by the kernel at exec)
+-----+	
HEADR+textsize	SYMS section (symsize bytes)
+datsize	stream of putsymb() entries
+-----+	
+ symsize	PCLN section (lcsize bytes)
	delta-encoded pc/line table
+-----+	

The header is written last because `symsize`^{147c} and `lcsize`^{160a} are only known after `asmsym()`^{148b} and `asmlc()`^{160c} run, but those sizes must be stored in the header at offset 0. `asmb()` handles this by seeking back to 0 at the very end. Note the asymmetry between text and data on one hand and BSS on the other: BSS has a size in the header but no bytes in the file, since the kernel will zero-fill those pages at `exec` time—a design shared with all `a.out/ELF` variants and a major reason executable files are much smaller than the running process image.

4.4.1 Header

The header generation (done last) assumes the set of globals containing the size of sections (`textsize`^{94a}, `datsize`^{92a}, etc) have been computed previously:

```
<asmb() header section 46a>≡ (45a)
DBG("%5.2f header\n", cputime());

OFFSET = 0;
seek(cout, OFFSET, SEEK__START);

switch(HEADTYPE) {
<asmb() switch HEADTYPE (for header generation) cases(arm) 46b>
}
```

Uses `DBG` 279, `HEADTYPE` 40a, and `cout` 38d.

```
<asmb() switch HEADTYPE (for header generation) cases(arm) 46b>≡ (46a) 188c▷
// see Exec in a.out.h
case H_PLAN9:
<asmb() if dynamic module magic header adjustment(arm) 174c>
else
    lput(0x647); /* magic */

    lput(textsize); /* sizes */
    lput(datsize);
    lput(bsssize);
    lput(symsize); /* nsyms */

    lput(entryvalue()); /* va of entry */
    lput(0L);
    lput(lcsize);
    break;
```

Uses `H_PLAN9` 150a, `bsssize` 92b, `datsize` 92a, `entryvalue()` 47a, `lcsize` 160a, `lput()` 110b, `symsize` 147c, and `textsize` 94a.

If you do not understand the order of the `lput()`^{110b} above, see Section 2.5 which describes the format of the `a.out` header used by Plan 9.

The address of the entry point is looked up in the symbol table in the `Sym.valueX` field of the `INITENTRY`^{42a} symbol:

```

<function entryvalue(arm) 47a>≡ (289b)
long
entryvalue(void)
{
    char *a;
    Sym *s;

    a = INITENTRY; // usually "_main"
    <entryvalue() if digit INITENTRY 214a>

    s = lookup(a, 0);

    switch(s->type) {
    case SNONE:
        // could warn no _main found?
        return INITTEXT; // no _main, start at beginning of binary then
    case STEXT:
        return s->value;
    <entryvalue() if dynamic module case 177b>
    default:
        diag("entry not TEXT: %s", s->name);
        return 0;
    }
}

```

Uses `INITENTRY` 42a, `INITTEXT` 40e, `SNONE` 33b, `STEXT` 156d, `diag()` 229d, and `lookup()` 32a.

4.4.2 Text section

The text section generation is of course more complicated. `asmb()`^{45a} iterates over all the code instructions and calls `oplook()`^{102c} to get additional information `o` about the instruction `p`. Then it calls the important function `asmout()`^{108b} which generates actual ARM instructions from `p`. `asmout()` will also use the global `cout`^{38d} to modify the executable file.

```

<asmb() Text section 47b>≡ (45a)
OFFSET = HEADR;
seek(cout, OFFSET, SEEK__START);

pc = INITTEXT;
for(p = firstp; p != P; p = p->link) {
    <adjust curtext when iterate over instructions p 36h>
    <adjust autosize when iterate over instructions p 66b>
    curp = p;
    <asmb() in Text section generation, sanity check pc 48b>

    o = oplook(p);
    // generate ARM instruction(s)!
    asmout(p, o);

    pc += o->size;
}
<asmb() before cflush, debug 227a>
cflush();

<asmb() Text section, output strings in text segment 214g>
}

```

Uses HEADR 40d, INITTEXT 40e, P 36c, asmout() 108b, cflush() 110a, cout 38d, curp 36g, firstp 36a, oplook() 102c, and pc 53a.

oplook() returns a pointer to an Optab^{99a} entry I will describe later.

```
<asm() locals 48a>+≡ (45a) <45b ??>
Prog *p;
Optab *o;
```

oplook() essentially looks at the opcode and operands in p and returns the kinds of actual ARM instructions that will be needed to encode the instruction p. It also returns the total size of those ARM instructions. Because the ARM uses fixed-length instructions of 4 bytes, the total size will be a multiple of 4.

Note that one instruction p can lead to the generation of multiple ARM instructions. Indeed, the instructions of the assembly language of 5a, and so the instructions in the object files, do not match exactly ARM instructions. For instance, the virtual instruction DIV has no counterpart in the ARM and is converted in a series of ARM instructions which ends with the call to the function _div() of the core C library. See Chapter 8 for more information about Optab and oplook().

Note that OFFSET is initialized to HEADR^{40d} above while pc^{53a} to INITTEXT^{40e}. Indeed, offsets in the executable and memory addresses are different concepts, as explained in Section 2.1.4.

asm() updates also the global pc above, but the address resolution of code instructions (in Instr.pcX) has already been done in dotext()^{94b}. This is repeated here just for sanity checking:

```
<asm() in Text section generation, sanity check pc 48b>≡ (47b)
if(p->pc != pc) {
    diag("phase error %lux sb %lux", p->pc, pc);
    if(!debug['a'])
        prasm(curp);
    pc = p->pc;
}
```

Uses curp 36g, debug 218a, diag() 229d, pc 53a, and prasm() 219b.

4.4.3 Data section

The data section generation assumes the layout of globals has already been done (via dodata()^{92c}), just like the code section generation in the previous section assumed the layout of code has also been done (via dotext()^{94b}). The Sym.valueX field of all data symbols should now contain the address of the global as an offset to the start of the data section, and datsize^{92a} should have been set.

```
<asm() Data section 48c>≡ (45a)
curtext = P;
switch(HEADTYPE) {
<asm() switch HEADTYPE (to position after text) cases(arm) 48d>
}
```

```
<asm() if dynamic module, before datblk() 176f>
```

```
for(t = 0; t < datsize; t += sizeof(buf)-100) {
    if(datsize-t > sizeof(buf)-100)
        datblk(t, sizeof(buf)-100, false);
    else
        datblk(t, datsize-t, false);
}
```

Uses HEADTYPE 40a, P 36c, curtext 36f, datblk() 49b, and datsize 92a.

```

⟨asmb() switch HEADTYPE (to position after text) cases(arm) 48d⟩≡ (48c) 188a▷
case H_PLAN9:
    OFFSET = HEADR+textsize;
    seek(cout, OFFSET, SEEK__START);
    break;

```

Uses HEADR 40d, H_PLAN9 150a, cout 38d, and textsize 94a.

Note again that the code seeks to HEADR+textsize, which may not be a page boundary. This is fine since offsets in the executable file are not memory addresses as explained in Section 2.1.4.

datblk()

```

⟨constant Dbufslop 49a⟩≡ (287a)
#define Dbufslop 100

```

```

⟨function datblk(arm) 49b⟩≡ (287a)

```

```

void
datblk(long s, long n, bool sstring)
{
    Prog *p;
    // absolute address of a DATA
    long a; // ulong?
    // size of a DATA
    int c;
    // index in output buffer for a DATA
    long l;
    // index in value of a DATA
    int i;
    ⟨datblk() other locals 50a⟩

    memset(buf.dbuf, 0, n+Dbufslop);

    for(p = datap; p != P; p = p->link) {
        ⟨datblk() if sstring might continue 215a⟩
        // else
        curp = p;

        a = p->from.sym->value + p->from.offset;
        l = a - s;
        c = p->reg; // size of DATA.In DATA foo+0(SB)/4, 0xa0 => 4

        i = 0;
        if(l < 0) {
            if(l+c <= 0)
                continue;
            while(l < 0) {
                l++;
                i++;
            }
        }
        if(l >= n)
            continue;
        // else

        ⟨datblk() sanity check multiple initialization 50b⟩

        switch(p->to.type) {
        ⟨datblk() switch type of destination cases 50c⟩
        default:
            diag("unknown mode in initialization%P", p);

```

```

        break;
    }
}
write(cout, buf.dbuf, n);
}

```

Uses Dbufslop-7 49a, P 36c, buf 234b, cout 38d, curp 36g, datap 36e, and diag() 229d.

<datblk() other locals 50a>≡ (49b) 50d▷
 long j;

<datblk() sanity check multiple initialization 50b>≡ (49b)
 for(j=1+(c-i)-1; j>=1; j--)
 if(buf.dbuf[j]) {
 print("%P\n", p);
 diag("multiple initialization");
 break;
 }

Uses buf 234b and diag() 229d.

D_SCONST

<datblk() switch type of destination cases 50c>≡ (49b) 50e▷
 case D_SCONST:
 for(; i<c; i++) {
 buf.dbuf[l] = p->to.sval[i];
 l++;
 }
 break;

Uses buf 234b.

D_CONST and endianness

<datblk() other locals 50d>+≡ (49b) <50a 52a▷
 char *cast;
 long d;

<datblk() switch type of destination cases 50e>+≡ (49b) <50c 193d▷
 case D_CONST: case D_ADDR:
 d = p->to.offset;
 <datblk() if D_ADDR case 52b>
 cast = (char*)&d;

 switch(c) {
 case 1:
 for(; i<c; i++) {
 buf.dbuf[l] = cast[inuxi1[i]];
 l++;
 }
 break;
 case 2:
 for(; i<c; i++) {
 buf.dbuf[l] = cast[inuxi2[i]];
 l++;
 }
 break;
 case 4:
 for(; i<c; i++) {
 buf.dbuf[l] = cast[inuxi4[i]];

```

        l++;
    }
    break;

default:
    diag("bad nuxi %d %d%P", c, i, curp);
    break;
}
break;

```

Uses buf 234b, curp 36g, diag() 229d, inuxi1 51a, inuxi2 51b, and inuxi4 51c.

```

⟨global inuxi1 51a⟩≡ (287a)
char inuxi1[1];

```

```

⟨global inuxi2 51b⟩≡ (287a)
char inuxi2[2];

```

```

⟨global inuxi4 51c⟩≡ (287a)
char inuxi4[4];

```

```

⟨main() initialize globals(arm) 51d⟩+≡ (38e) <43c 104b>
nuxiinit(); // endianness conversion tables

```

```

⟨function nuxiinit(arm) 51e⟩≡ (287a)
void
nuxiinit(void)
{

    int i, c;

    for(i=0; i<4; i++) {
        c = find1(0x04030201L, i+1);
        if(i < 2)
            inuxi2[i] = c;
        if(i < 1)
            inuxi1[i] = c;
        inuxi4[i] = c;
        ⟨nuxiinit() in loop i, fnuxi initialisation 194c⟩
    }
    ⟨nuxiinit() debug 51g⟩
    Bflush(&bso);
}

```

Uses bso 218c, find1() 51f, inuxi1 51a, inuxi2 51b, and inuxi4 51c.

```

⟨function find1 51f⟩≡ (287a)
int
find1(long l, int c)
{
    char *p;
    int i;

    p = (char*)&l;
    for(i=0; i<4; i++)
        if(*p++ == c)
            return i;
    return 0;
}

```

```

<nuxiinit() debug 51g>≡ (51e)
if(debug['v']) {
    Bprint(&bso, "inuxi = ");
    for(i=0; i<1; i++)
        Bprint(&bso, "%d", inuxi1[i]);
    Bprint(&bso, " ");
    for(i=0; i<2; i++)
        Bprint(&bso, "%d", inuxi2[i]);
    Bprint(&bso, " ");
    for(i=0; i<4; i++)
        Bprint(&bso, "%d", inuxi4[i]);
    Bprint(&bso, "\nfnuxi = ");
    for(i=0; i<4; i++)
        Bprint(&bso, "%d", fnuxi4[i]);
    Bprint(&bso, " ");
    for(i=0; i<8; i++)
        Bprint(&bso, "%d", fnuxi8[i]);
    Bprint(&bso, "\n");
}

```

Uses bso 218c, debug 218a, fnuxi4 194a, fnuxi8 194b, inuxi1 51a, inuxi2 51b, and inuxi4 51c.

```

D_ADDR
<datblk() other locals 52a>+≡ (49b) <50d 193c>
Sym *v;

```

```

<datblk() if D_ADDR case 52b>≡ (50e)
v = p->to.sym;
if(v) {
    switch(v->type) {
        <datblk() in D_ADDR case, switch symbol type cases 52c>
    }
    <datblk() if dynamic module(arm) 179a>
}

```

```

<datblk() in D_ADDR case, switch symbol type cases 52c>≡ (52b) 176c>
case STEXT: case SSTRING:
    d += p->to.sym->value;
    break;
case SDATA: case SBSS:
    d += p->to.sym->value + INITDAT;
    break;

```

Uses INITDAT 40g, SBSS 170b, SDATA, SSTRING 35g, and STEXT 156d.

4.4.4 Symbol and line table sections

The generation of the symbol and line table sections will be explained in Chapter 10

4.5 Checking for unresolved symbols: undef()

The final step is making sure there is no more undefined symbols:

```

<function undef 52d>≡ (292d)
/// main -> <>
void
undef(void)

```

```
{
    int i;
    Sym *s;

    for(i=0; i<NHASH; i++)
        for(s = hash[i]; s != S; s = s->link)
            if(s->type == SXREF)
                diag("%s: not defined", s->name);
}
```

Uses NHASH, S [33b](#), SXREF, and diag() [229d](#).

Chapter 5

Loading Objects

Now that you have seen the code or a high-level view of the code of the main functions of 51, I can start to go deeper and detail the different components of the linking pipeline. I start in this chapter with the loading of object files in memory performed mostly by `ldobj()`⁵⁵.

5.1 Object file format: .5

Before reading the code of `ldobj()`⁵⁵ it is good to have in mind the format of object files. This format is summarized in Figure 5.1. See the ASSEMBLER book [Pad15a] if you need more explanations.

5.2 A global and local program counter: `pc` and `ipc`

A very important global which will be used by `ldobj()`⁵⁵ is the *virtual program counter*:

```
<global pc 53a>≡ (283b)
long pc = 0;
```

Uses `pc` 53a.

`ldobj()` will increment `pc` after each code instruction read as you will see in Section 5.4. Note that `pc` is a *global* and so persists between different calls to `ldobj()`. Thus, all the instructions in the different object files will have a unique program counter value in their `Instr.pcX` field. `ldobj()` is also using the *local* below to store the value of the program counter at the beginning of the call:

```
<ldobj() locals(arm) 53b>≡ (55) 53c>
long ipc;
```

`ipc` will be used for relocating branching instructions as you will see in Section 5.4.1.

5.3 Object code input: `ldobj()`

I can now show the code of `ldobj()`. The function takes 3 parameters: `f` the file descriptor of the (opened) object file, `c` the size of this object file, and `pn` the name of this object file used mostly for error management. `ldobj()` essentially reads instructions from `f` in a loop where each iteration allocates a new instruction `p` with the opcode `o` and populates either `firstp`^{36a} or `datap`^{36e} depending on the opcode.

```
<ldobj() locals(arm) 53c>+≡ (55) <53b 59a>
Prog *p;
// enum<Opcode>
short o;
```

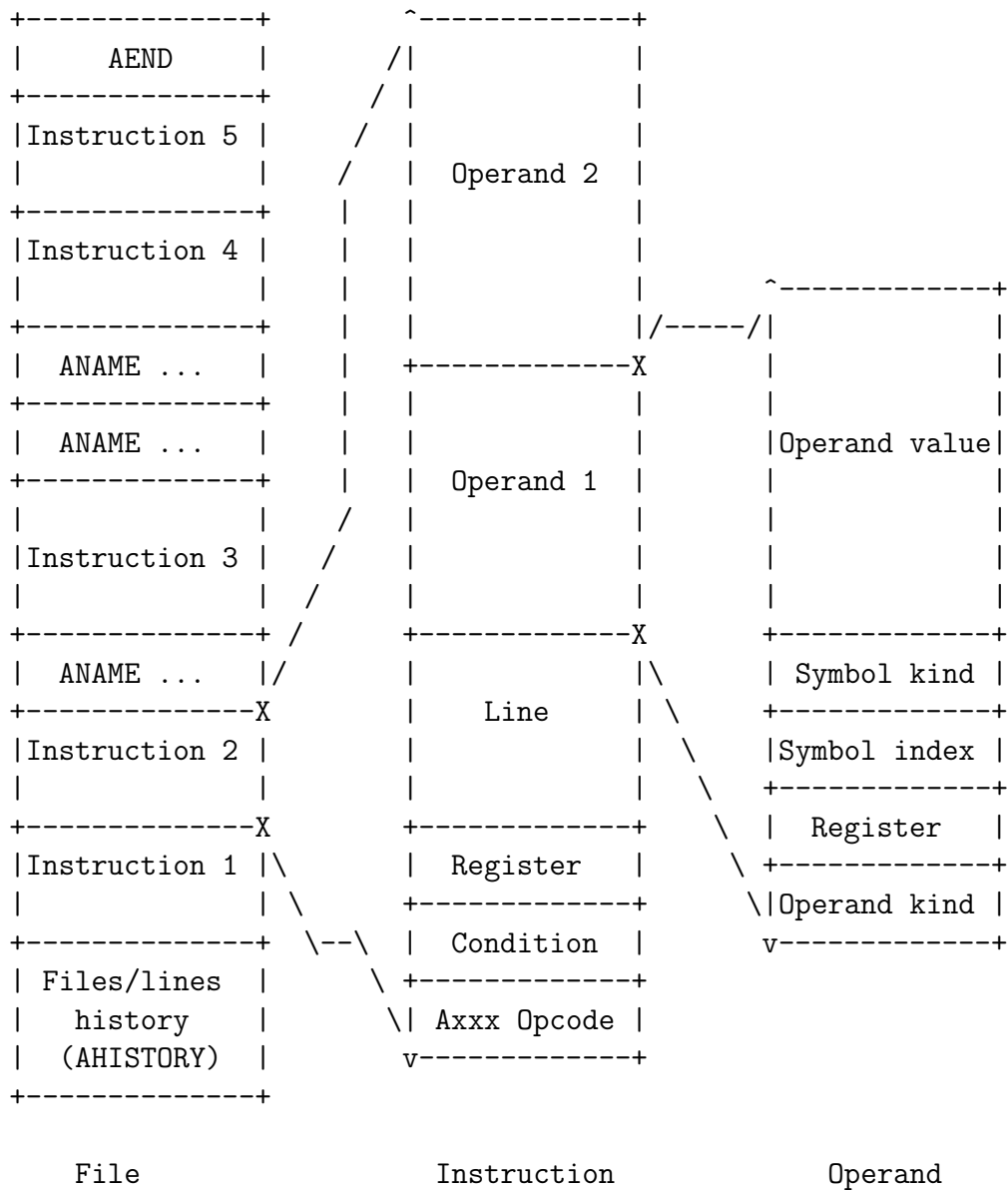


Figure 5.1: Format of a .5 object file.

An important local of `ldobj()` is `bloc` which is a *cursor* in an *input buffer* derived from `f` as explained soon in Section 5.3.3. It allows the code which parses instructions to access individual bytes of an instruction in the object file as `bloc[0]`, `bloc[1]`, etc.

```

⟨function ldobj(arm) 55⟩≡ (290)
  /// main -> objfile -> <>
  void
  ldobj(fdt f, long c, char *pn)
  {
    ⟨ldobj() locals(arm) 53b⟩

    ⟨ldobj() remember set of object filenames 69d⟩
    ⟨ldobj() bloc and bsize init 59b⟩

  // can come from AEND
  newloop:
    // new object file
    ipc = pc;
    ⟨ldobj() after newloop when new object file, more initializations 61b⟩

  loop:
    if(c <= 0)
      goto eof;

    ⟨ldobj() read if needed in loop:, adjust bloc and bsize 59c⟩

    o = bloc[0]; /* as */
    ⟨ldobj() sanity check opcode in range(arm) 56b⟩

    // dispatch opcode part one
    ⟨ldobj() if ANAME or ASIGNAME(arm) 61d⟩
    // else

    p = malloc(sizeof(Prog));
    p->as = o;
    ⟨ldobj() read one instruction in p 56a⟩
    p->link = P;
    p->cond = P;

    ⟨ldobj() sanity check p 57a⟩
    ⟨ldobj() debug 225c⟩

    // dispatch opcode part two
    switch(o) {
    ⟨ldobj() switch opcode cases(arm) 64d⟩
    }
    goto loop;

  eof:
    diag("truncated object file: %s", pn);
  }

```

Uses `P` 36c, `diag()` 229d, `malloc()` 233a, and `pc` 53a.

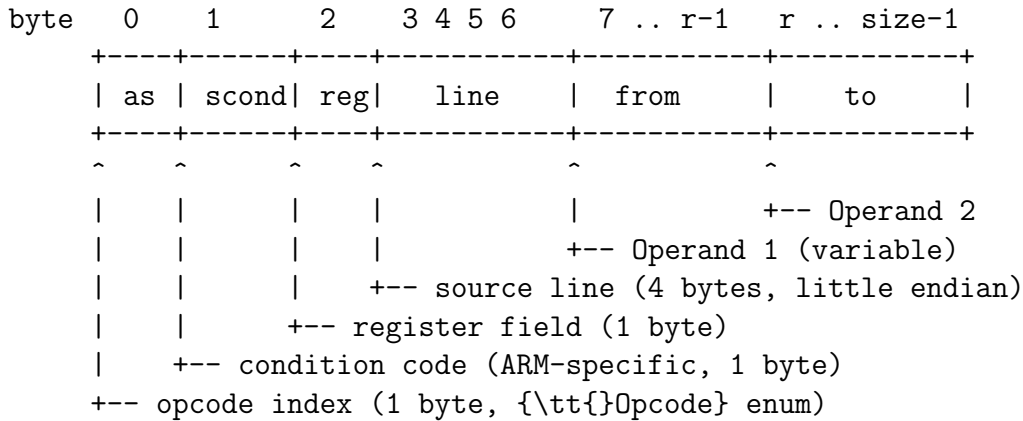
`ldobj()`⁵⁵ is a complex and very long function. I split it in many chunks to facilitate its comprehension. The rest of this chapter will detail most of those chunks.

Note that the code above looks like a loop which can finish only in an error. But, one of the chunk dealing with the AEND opcode (see Section 5.4.5), hidden in the `switch` above, contains actually a `return` which can escape the loop. In fact, AEND explains also the reason for the two loop labels `newloop` and `loop`. Indeed, `ldobj()` can also be used to read a whole library in which case 51 needs to iterate over the multiple object files

contained in one library file, which, as you will see in Chapter 6, are separated by the AEND opcode.

5.3.1 Single instruction input

Figure 5.1 shows the high-level layout of an instruction record, but it helps to see the raw bytes the reader walks through. A minimum instruction is 7 bytes (opcode header) plus at least 4 bytes per operand, so the smallest record is 15 bytes and most are 15 or 16:



Two details are worth noting. First, `line` uses a 4-byte little-endian encoding while the rest of the linker treats numbers as native-endian—this record format has to be portable between the assembler (which may cross-compile on a big-endian host) and 51, so the object file commits to little-endian on the wire. Second, the opcode is deliberately the first byte so that `ldobj()` can peek at `bloc[0]` to dispatch `ANAME`, `ASIGNAME` and `AHISTORY` entries before allocating a `Prog`²⁷⁹—those pseudo-opcodes don't represent instructions and would waste a `Prog` struct.

An important chunk of `ldobj()`⁵⁵ deals with the parsing of one instruction whose format is described in the middle of Figure 5.1. The code below does mostly the reverse operation of `outcode()` in the ASSEMBLER book [Pad15a] and reads one instruction:

```
<ldobj() read one instruction in p 56a>≡ (55)
// mostly opposite of outcode() in 5a
// p->as = bloc[0] has been done already above so continue from bloc[1]
p->scond = bloc[1];
p->reg = bloc[2];
p->line = bloc[3] | (bloc[4]<<8) | (bloc[5]<<16) | (bloc[6]<<24);
r = 7;
r += inopd(bloc+r, &p->from, h);
r += inopd(bloc+r, &p->to, h);
```

```
bloc += r;
c -= r;
```

Uses `inopd()` 57b.

The code to read an operand uses `inopd()`^{57b} which is shown in the next section. Its last argument is related to the *object file symbol table* and will be fully explained in Section 5.3.4.

```
<ldobj() sanity check opcode in range(arm) 56b>≡ (55)
if(o <= AXXX || o >= ALAST) {
    diag("%s: line %ld: opcode out of range %d", pn, pc-ipc, o);
    print(" probably not a .5 file\n");
    errexit();
}
```

Uses `diag()` 229d, `errexit()` 229b, and `pc` 53a.

```

⟨ldobj() sanity check p 57a⟩≡
    if(p->reg > NREG)
        diag("register out of range %d", p->reg);
Uses diag() 229d.

```

(55)

5.3.2 Operand input: inopd()

inopd() does mostly the reverse of outopd() in the ASSEMBLER book [Pad15a] and reads one operand from the object file. The format of an operand is described in the right of Figure 5.1. inopd() returns the number of bytes that were used to read this operand as operands can have different size:

```

⟨function inopd(arm) 57b⟩≡
    /// main -> objfile -> ldobj -> <>
    static int
    inopd(byte *p, Adr *a, Sym *h[])
    {
        int size; // returned
        int symidx;
        ⟨inopd() other locals 151a⟩

        a->type = p[0];
        a->reg = p[1];
        ⟨inopd() sanity check register range 58b⟩
        symidx = p[2];
        ⟨inopd() sanity check symbol range 63a⟩
        a->sym = h[symidx];
        a->symkind = p[3];
        ⟨inopd() sanity check D_CONST ??⟩

        size = 4;

        switch(a->type) {
        ⟨inopd() cases 58a⟩
        default:
            print("unknown type %d\n", a->type);
            p[0] = ALAST+1;
            return 0; /* force real diagnostic */
        }
        ⟨inopd() adjust curauto for N_LOCAL or N_PARAM symkind 151b⟩

        return size;
    }

```

(290)

The local `symidx` contains a *symbol index* in the object file symbol table `h` passed as a parameter. This index and the table will be explained in Section 5.3.4, but the important result is that `Operand.symX` will point to a symbol in the symbol table `hash`^{31a}.

The operands have variable size depending on the operand kind:

This is the only place in the object-file record where the encoding departs from fixed-size fields. The 4-byte opcode header (type, reg, symidx, symkind) is always present, and then a kind-dependent tail grows the record:

byte	0	1	2	3	4 .. size-1	
	+-----+-----+-----+-----+-----+-----+					
	type	reg	symidx	symkind	kind-specific	
	+-----+-----+-----+-----+-----+-----+					

kind	tail size	what it carries
------	-----------	-----------------

-----	-----	-----
D_NONE	0	(no payload)
D_REG	0	register number is in reg above
D_PSR	0	program status register reference
D_REGREG	1	second register (p[4])
D_CONST	4	signed 32-bit constant, LE
D_ADDR	4	address offset to a symbol
D_SHIFT	4	encoded shifter operand
D_OREG	4	offset for MOVW foo(Rn), ...
D_BRANCH	4	absolute virtual pc for B/BL
D_SCONST	8	inline 8-byte string constant

The trailing return value of `inopd()` tells the caller how far to advance `bloc`—there is no length field in the header, so the reader must dispatch on `type` before it can find the next operand. This is the same reason the assembler has to agree with the linker on this table: a mismatch silently corrupts the rest of the instruction stream rather than producing a parse error.

`<inopd() cases 58a>≡` (57b) 189a▷

```

// 0 byte
case D_NONE:
case D_REG:
case D_PSR:
    break;

// 1 byte
case D_REGREG:
    a->offset = p[4];
    size++;
    break;

// 4 bytes
case D_CONST:
// ??? D_ADDR is 4 bytes?? what about symbol?
case D_ADDR:
case D_SHIFT:
case D_OREG:
case D_BRANCH:
    a->offset = p[4] | (p[5]<<8) | (p[6]<<16) | (p[7]<<24);
    size += 4;
    break;

// 8 bytes (NSNAME)
case D_SCONST:
    a->sval = malloc(NSNAME);
    memmove(a->sval, p+4, NSNAME);
    size += NSNAME;
    break;

```

Uses `malloc()` 233a.

`<inopd() sanity check register range 58b>≡` (57b)

```

if(a->reg < 0 || a->reg > NREG) {
    print("register out of range %d\n", a->reg);
    p[0] = ALAST+1;
    return 0; /* force real diagnostic */
}

```

5.3.3 Buffered input: buf

I can now explain the code connecting `bloc` to the file descriptor `f`. To read object files, `ldobj()`⁵⁵ uses a global called `buf`^{234b} of type `Buf`^{233d} as well as many utility functions forming a sort of *input/output buffer management* library. This code is quite generic and independent of 51 and so most of it is described in Appendix D.2. The most important thing for this chapter is that `Buf.ibuf` contains an array of bytes, the *input buffer*, filled from time to time by the function `readsome()`^{59d}. A few locals of `ldobj()` are pointers in this array:

```
<ldobj() locals(arm) 59a>+≡ (55) <53c 61a>
// array<byte> (slice of buf.ibuf)
byte *bloc;
// ref<byte> (end pointer in buf.ibuf)
byte *bsize;
// remaining bytes, bsize - bloc
int r;
```

`bloc` is a *cursor* in the input buffer which is moved around. It allows the code which parses instructions to access individual bytes of an instruction in the object file as `bloc[0]`, `bloc[1]`, etc, as you have seen in the previous sections. It is initialized to the start of the input buffer:

```
<ldobj() bloc and bsize init 59b>≡ (55)
bloc = buf.ibuf;
bsize = buf.ibuf;
```

Uses `buf` 234b.

Figure 5.2 represents the evolution of the state of `buf.ibuf` while the input buffer gets filled by `readsome()`. The code to fill as needed the input buffer is below:

```
<ldobj() read if needed in loop:, adjust bloc and bsize 59c>≡ (55)
r = bsize - bloc;
if(r < 100 && r < c) { /* enough for largest instruction */
    bsize = readsome(f, buf.ibuf, bloc, bsize, c);
    if(bsize == nil)
        goto eof;
    bloc = buf.ibuf; // readsome() does some memmove()
    goto loop;
}
```

Uses `buf` 234b and `readsome()` 59d.

The number 100 above represents a “window” large enough to read one full instruction from the object file. At some point `bloc` will get close to `bsize` and `readsome()` will be called again, even if the input buffer contains still some unprocessed bytes. This simplifies the rest of the code in `ldobj()` and `inopd()`^{57b} which can simply access `bloc[0]`, `bloc[1]`, etc without having to worry whether they need to call `readsome()` again.

Figure 5.3 represents the evolution of the state of `buf.ibuf` while the input buffer gets filled another time by `readsome()`. The bytes at the end of the input buffer are first moved to its beginning and the rest of the buffer is filled by more data from `f`.

```
<function readsome 59d>≡ (289a)
byte*
readsome(fdt f, byte *buf, byte *good, byte *stop, int max)
{
    int n;

    n = stop - good;
    memmove(buf, good, n);
    stop = buf + n;
    n = MAXIO - n;
    if(n > max)
        n = max;
    n = read(f, stop, n);
```



```

    if(n <= 0)
        return nil;
    return stop + n;
}

```

Uses MAXIO.

5.3.4 Object file symbol table: h and ANAME

Object files contain mostly instructions. They contain also an *object file symbol table* which is *spread* in the file and where each entry starts with the ANAME opcode, as shown in the left of Figure 5.1. The object file symbol table is also a *circular* array. Its entries define symbols which are then referenced in operands of some following instructions via a *symbol index*, as shown in the right of Figure 5.1.

An important chunk of `ldobj()`⁵⁵ deals with those ANAME entries. The format of each entry starts with the pseudo-opcode ANAME followed by the kind, symbol index, and the name of the symbol as a possible long string terminated with `'\0'`. See the ASSEMBLER book [Pad15a] if you need more explanations.

The local `h` below mimics in memory this circular table:

```

<ldobj() locals(arm) 61a>+≡ (55) <59a 61c>
// array<option<ref<Sym>>>
Sym *h[NSYM];

```

NSYM is a constant defined in `include/objs/common.out.h` which is also used by 5a and so which was described already in the ASSEMBLER book [Pad15a]. `h` was a global in 5a but is a local variable in 51 as each object file contains its own symbol table. The table is reset for each new object file processed:

```

<ldobj() after newloop when new object file, more initializations 61b>≡ (55) 64b>
memset(h, 0, sizeof(h));

```

`h` is populated by `ldobj()` as the object file is read and new ANAME entries are found. `ldobj()` also populates the global symbol table `hash`^{31a} via `lookup()`^{32a}. Figure 5.4 shows the relation between those data structures. Remember that the first operand of an instruction like `MOVW foo(SB)`, `R1` is represented in the object file as 4 elements, as shown in the right of Figure 5.1, using 4 bytes: the operand kind (`D_OREG`), a possible register number (`R_NONE`), a symbol index (`1`), and a symbol kind (`N_EXTERN`). This operand is summarized as `idx:1 EXT` in Figure 5.4.

I can now show the local variables and the code of `ldobj()` dealing with ANAME (and also partially with ASIGNAME which will be explained in section 5.5):

```

<ldobj() locals(arm) 61c>+≡ (55) <61a 65a>
// enum<Sym_kind>

```

```

int k;
int symidx;
int v;
// ref<byte> (in Buf.ibuf)
byte *stop;

```

```

<ldobj() if ANAME or ASIGNAME(arm) 61d>≡ (55)

```

```

if(o == ANAME || o == ASIGNAME) {
    <ldobj() if SIGNAME adjust sig 68b>

    stop = memchr(&bloc[3], '\0', bsize-&bloc[3]);
    <ldobj() if stop is nil refill buffer and retry 63b>

    k = bloc[1]; /* type */
    symidx = bloc[2]; /* sym */

    bloc += 3;
    c -= 3;
}

```

Object file	h	Symbols	hash
AEND			
...			
		...	
AMOVW idx:2 EXT, R3			
		<-- "foo", ... <--	
ANAME EXT 2 "bar"	49	^	
AMOVW R2, idx:1 EXT			
...			
AMOVW idx:1 EXT, R1			
ANAME EXT 1 "foo"			
	2		
AADD R1, R2, R3			
	1	> "bar", ... <--	
AHISTORY ...	0		

Figure 5.4: Example of object file and content of `h` and `hash`.

```

v = 0; // global version by default
⟨ldobj() when ANAME opcode, if private symbol adjust version 64c⟩

// this will possibly create new symbols
s = lookup((char*)bloc, v);

c -= &stop[1] - bloc;
bloc = stop + 1;

⟨ldobj() if sig not zero 68d⟩
⟨ldobj() when ANAME, debug 225d⟩

h[symidx] = s;

if((k == N_EXTERN || k == N_INTERN) && s->type == SNONE)
    s->type = SXREF;

⟨ldobj() when ANAME opcode, if N_FILE 156c⟩
goto loop;
}

```

Uses SNONE 33b, SXREF, and lookup() 32a.

ldobj() first extracts the string from the ANAME entry and calls lookup() on it. This creates a new entry in the symbol table hash, unless this symbol is a global symbol (v == 0) which was defined in another object file loaded before, or if the same symbol was introduced before in the same object file due to the circular nature of the object file symbol table. ldobj() then updates h so further references of the symbol index in operands of instructions coming after will be transformed in pointers (in Operand.symX) to the right symbol entry in hash. Indeed, see the instruction a->sym = h[symidx] of inopd() ^{57b}. You can now also understand the sanity check of the symbol index in inopd():

```

⟨inopd() sanity check symbol range 63a⟩≡ (57b)
if(symidx < 0 || symidx > NSYM){
    print("sym out of range: %d\n", symidx);
    p[0] = ALAST+1;
    return 0;
}

```

Note also that ldobj() sets the symbol section to SXREFX for symbols referring to procedures or globals (e.g., foo(SB) but also tmp<>(SB)) if the symbol was just created. This symbol is now marked as “wanted”.

The reading of an ANAME entry requires to have the full string (ending with '\0') in the input buffer:

```

⟨ldobj() if stop is nil refill buffer and retry 63b⟩≡ (61d)
if(stop == nil){
    bsize = readsome(f, buf.ibuf, bloc, bsize, c);
    if(bsize == nil)
        goto eof;
    bloc = buf.ibuf;
    stop = memchr(&bloc[3], '\0', bsize-&bloc[3]);
    if(stop == nil){
        fprintf(2, "%s: name too long\n", pn);
        errexit();
    }
}
}

```

Uses buf 234b, errexit() 229b, and readsome() 59d.

5.3.5 Private symbols and version

As mentioned in Section 3.1, private symbols in different object files using the same name, e.g., `tmp<>` in `foo.5` and `tmp<>` in `bar.5`, can be differentiated thanks to a *version* number stored in a global:

```
<global version 64a>≡ (290)
    int version = 0;
```

Uses version 64a.

This version is a unique integer representing an object file by simply incrementing it each time a new object file is parsed:

```
<lobj() after newloop when new object file, more initializations 64b>+≡ (55) <61b 157f>
    version++;
```

Uses version 64a.

Every reference to a private symbol in an object file then uses the version corresponding to this object file:

```
<lobj() when ANAME opcode, if private symbol adjust version 64c>≡ (61d)
    if(k == N_INTERN)
        v = version;
```

Uses version 64a.

5.4 Opcode dispatch

Once an instruction `p` with its opcode and operands have been read, `lobj()`⁵⁵ looks at the opcode and modifies different globals in a `switch` on the opcode value. The following sections describe the different cases of this `switch` which are mostly concerned with pseudo-instructions.

5.4.1 *Axxx*

The default case corresponds to regular instructions, e.g., `AMUL`.

```
<lobj() switch opcode cases(arm) 64d>≡ (55) 65b>
    default:
    casedef:
        <lobj() in switch opcode default case, if skip 187d>

        // relocation
        if(p->to.type == D_BRANCH)
            p->to.offset += ipc;

        //add_queue(firstp, lastp, p)
        lastp->link = p;
        lastp = p;

        p->pc = pc;
        pc++;
        break;
```

Uses `lastp` 36d and `pc` 53a.

The code above modifies many of the globals I mentioned before and illustrates many of the concepts I introduced before. Here are a few notes about this code:

- Branching instructions are relocated thanks to the local program counter `ipc` which contains the new memory address origin of the object file containing the instruction
- The global `firstp`^{36a} (via `lastp`^{36d}) is updated to contain the new code instruction

- Each instruction gets its `Instr.pcX` field set to the value of the program counter at the moment the instruction is read
- The virtual program counter `pc53a` is incremented after a code instruction is read

5.4.2 ATEXT and autosize

Remember from the ASSEMBLER book [Pad15a] that pseudo-instructions use the same format than regular instructions in the object file. So, for the `TEXT` pseudo-instruction, e.g., `TEXT foo(SB), $8`, the name of the procedure is stored in the first operand (`Instr.fromX`) while the size for its local variables is stored in the second operand (`Instr.toX`). In fact, `Instr.from.sym` is a pointer to a symbol in the symbol table `hash31a` (see the code of `inopd()57b`) where `Sym.nameX` contains the procedure's name:

```
<llobj() locals(arm) 65a)+≡ (55) <61c 68a>
Sym *s;
```

```
<llobj() switch opcode cases(arm) 65b)+≡ (55) <64d 66c>
case ATEXT:
  <llobj() case ATEXT, if curtext not null adjustments for curauto 151d>
  curtext = p;
  <llobj() in switch opcode ATEXT case, reset skip 187e>
  <llobj() in switch opcode ATEXT case, set autosize 66a>

  s = p->from.sym;
  <llobj() sanity check for ATEXT symbol s 65c>
  s->type = STEXT;
  s->value = pc;

  // like in default case
  //add_queue(firstp, lastp, p)
  lastp->link = p;
  lastp = p;

  p->pc = pc;
  pc++;

  <llobj() in switch opcode ATEXT case, populate textp 152c>
  break;
```

Uses `STEXT 156d`, `curtext 36f`, `lastp 36d`, and `pc 53a`.

The code is similar to the default case you have seen in the previous section. In addition, the section and value properties of the procedure symbol are modified. In particular, `Sym.valueX` contains now the virtual program counter of the procedure, which will be useful for resolving branching instructions involving the procedure's name later in Chapter 7.

```
<llobj() sanity check for ATEXT symbol s 65c)+≡ (65b)
if(s == S) {
  diag("TEXT must have a name\n%P", p);
  errexit();
}
if(!(s->type == SNONE || s->type == SXREF)) {
  <llobj() case ATEXT and section not SNONE or SXREF, if DUPOK 187c>
  diag("redefinition: %s\n%P", s->name, p);
}
```

Uses `S 33b`, `SNONE 33b`, `SXREF`, `diag() 229d`, and `errexit() 229b`.

The size for the local variables of the current procedure is also stored in a global:

```
<global autosize(arm) 65d)+≡ (283b)
long autosize;
```

Local variables, also known as *automatic variables*, are hold in the *stack* of the process. The size for those locals is specified in the procedure declaration, e.g., 8 in `TEXT foo(sb)`, \$8, which will be enough to contain 2 locals using 4 bytes each. In fact, 51 first rounds the size at a 4 byte boundary and then increments it by 4:

```
<ldobj() in switch opcode ATEXT case, set autosize 66a>≡ (65b)
    p->to.offset = rnd(p->to.offset, 4);
    autosize = p->to.offset;
    autosize += 4;
```

Uses `autosize 65d` and `rnd() 236a`.

The reason for the 4 increment is to allocate space in the stack to save the *return address* stored in the *link register* R14 (aliased as REGLINK). See the ASSEMBLER book [Pad15a] to refresh your memory about the branch and link instruction and the Plan 9 calling conventions.

Many algorithms will update `autosize` as follows while iterating over all the instructions, just like they do with `curtext`^{36f}:

```
<adjust autosize when iterate over instructions p 66b>≡ (94b 47b)
    if(p->as == ATEXT) {
        autosize = p->to.offset + 4;
    }
```

Uses `autosize 65d`.

5.4.3 AGLOBL

The code dealing with globals is simpler than the one dealing with procedures. Remember that for the GLOBL pseudo-instruction, e.g., `GLOBL hello(SB)`, \$12, the symbol of the global is stored in the first operand (`Instr.fromX`) while its optional size is stored in the second operand (`Instr.toX`):

```
<ldobj() switch opcode cases(arm) 66c>+≡ (55) <65b 66e>
    case AGLOBL:
        s = p->from.sym;
        <ldobj() sanity check for AGLOBL symbol s 66d>
        s->type = SBSS; // for now; will be set maybe to SDATA in dodata()
        s->value = (p->to.offset > 0) ? p->to.offset : 0;
        break;
```

Uses `SBSS 170b`.

No list of instructions is modified. The only effect of a GLOBL declaration is the modification of a symbol in the symbol table. It is the DATA pseudo-instructions which populate the data section as you will see in the next section. Note that for now `Sym.valueX` contains the size of the global.

```
<ldobj() sanity check for AGLOBL symbol s 66d>≡ (66c)
    if(s == S) {
        diag("GLOBL must have a name\n%P", p);
        errexit();
    }
    if(!(s->type == SNONE || s->type == SXREF))
        diag("redefinition: %s\n%P", s->name, p);
```

Uses `S 33b`, `SNONE 33b`, `SXREF`, `diag() 229d`, and `errexit() 229b`.

5.4.4 ADATA

The code dealing with DATA pseudo-instructions is trivial. It just populates `datap`^{36e}:

```
<ldobj() switch opcode cases(arm) 66e>+≡ (55) <66c 67b>
    case ADATA:
        <ldobj() sanity check for ADATA symbol s 67a>
```

```
//add_list(datap, p)
p->link = datap;
datap = p;
```

```
break;
```

Uses `datap` 36e.

Note that 51 could set the section of the symbol to `SDATA` here with code like `s->type = SDATA;`. But this would complicate the code to detect redefinition of symbols in the previous section. Indeed, `DATA` pseudo-instructions can precede a `GLOBL` declaration. This is why the modification of the section to `SDATA` is done in `dodata()`^{92c} later instead.

```
<ldobj() sanity check for ADATA symbol s 67a>≡ (66e)
if(p->from.sym == S) {
    diag("DATA without a sym\n%P", p);
    break;
}
```

Uses `S` 33b and `diag()` 229d.

5.4.5 AEND

`AEND` is a pseudo-opcode inserted as the end of each object file by the assembler or compiler. It is a *special marker* convenient to have when dealing with libraries. Indeed, libraries are little more than object files concatenated together, as you will see in Chapter 6, and `AEND` marks represent object boundaries.

```
<ldobj() switch opcode cases(arm) 67b>+≡ (55) <66e 67c>
case AEND:
    <ldobj() case AEND, curauto adjustments with curhist 156a>
    <ldobj() case AEND, curauto adjustments 151e>
    curtext = P;
```

```
if(c)
    goto newloop;
return;
```

Uses `P` 36c and `curtext` 36f.

The important instruction above is `return` which allows to exit from `ldobj()`⁵⁵ without any error.

5.4.6 AGOK

`AGOK` (God Only Knows) is a pseudo-opcode used to initialize new instructions in 51. It is also used for the same reason in 5c. It should always be overridden at some point by a real opcode, hence the warning code below:

```
<ldobj() switch opcode cases(arm) 67c>+≡ (55) <67b 153>
case AGOK:
    diag("unknown opcode\n%P", p);
    p->pc = pc;
    pc++;
    break;
```

Uses `diag()` 229d and `pc` 53a.

5.5 Safe linking

An interesting feature of object files and 51, not present in traditional linkers, is the possibility (1) to attach *signatures* to symbols in object files, and (2) to check for *signature compatibility* when the same symbol is mentioned multiple times in different object files. This makes linking far safer.

5.5.1 Motivations

Imagine the following scenario: a file `foo.c` defines a function `foo()` taking two integer parameters and not returning anything. The function is exported in a header file `foo.h`. Another file `bar.c` is including the header and calls `foo()` with two integers. Both files are compiled resulting in the object files `foo.o` and `bar.o` which can be linked together. Later on, an additional parameter is added to `foo()` in `foo.h` and `foo.c` and `foo.c` is recompiled leading to a new `foo.o` object file. At this point, without signatures, there is nothing that prevents the new `foo.o` and the old `bar.o` to be linked together to form an executable. Traditional linkers will happily link those object files resulting in an executable which, at run-time, will not have the right behavior (`bar.o` contains a call to `foo()` with not enough parameters).

Of course, many projects use a `Makefile` where accurate file dependencies can alleviate the issue. You can specify manually for instance that `bar.o` depends also on `foo.h`. Then, any modification of `foo.h` will trigger automatically the regeneration of `bar.o`. Dependencies can also be automatically generated by tools like `gcc -MM`. Still, it is easy to make mistakes and miss some dependencies, or to forget to use shared headers, in which case there is no tool to detect the possible linking of incompatible object files (except by running the executable and get occasionally a segmentation fault).

5.5.2 ASIGNAME and `5c -T`

When two C files reference the same entity, `5l` can enforce that those two references have the same *type*. `ASIGNAME` is a pseudo-opcode generated by `5c` when using the `-T` option (T for type) which attaches a *signature* to a symbol. It is an alternate to `ANAME` with additional 4 bytes after the `ASIGNAME` opcode, and before the symbol kind, to store a signature as a `ulong`:

```
<ldobj() locals(arm) 68a>+≡ (55) <65a 69c>
    ulong sig;
```

```
<ldobj() if SIGNAME adjust sig 68b>≡ (61d)
    sig = 0;
    if(o == ASIGNAME){
        sig = bloc[1] | (bloc[2]<<8) | (bloc[3]<<16) | (bloc[4]<<24);
        bloc += 4;
        c -= 4;
    }
```

This signature is stored as another symbol property:

```
<Sym other fields 68c>≡ (30) 69b>
    // for instance last 32 bits of md5sum of the type of the symbol
    ulong sig;
```

```
<ldobj() if sig not zero 68d>≡ (61d)
    if(sig != 0){
        <ldobj() signature compatibility check 69a>
        s->sig = sig;
        <ldobj() remember file introducing the symbol 69e>
    }
```

In practice, signatures are derived from the type of an entity by the compiler. For instance, in our scenario above, the signature of `foo()` in the object file could be the result of the last 32-bits of the `md5sum` of the string representing the type, e.g., `md5sum("void(int,int)") == 0x4a2489a1`. This signature would be attached to the symbol `foo` in `foo.o` and `bar.o`. Later on, when an integer parameter is added, the signature of `foo` in

foo.5 would become `md5sum("void(int,int,int)") == 0x74100cff`. The linking of the new foo.5 and old bar.5 would then be detected and prevented by the code below:

```
<llobj() signature compatibility check 69a)&#x27E; (68d)
    if(s->sig != 0 && s->sig != sig)
        diag("incompatible type signatures %lux(%s) and %lux(%s) for %s",
            s->sig, filen[s->file],
            sig, pn,
            s->name);
```

Uses `diag()` 229d.

The actual algorithm used by 5c to compute the signature is actually not the `md5sum`. See the COMPILER book [Pad16a] to learn more about how signatures are generated.

5.5.3 Error management

To give good error messages like:

```
$ 5l foo.5 misc.5 bar.5 /arm/lib/libc.a
bar: incompatible type signatures bde91c57(foo.5) and adb3322b(bar.5)
    for foo
```

one needs to remember in the symbol which file introduced the symbol the first time. `Sym.file` below contains an *index* in the private array `filen` representing the file:

```
<Sym other fields 69b)&#x27E; (30) <68c 171h>
    // index in filen[]
    ushort file;
```

```
<llobj() locals(arm) 69c)&#x27E; (55) <68a 69f>
    // growing_array<option<string>> (grown for every 16 elements)
    static char **filen;
    // index of next free entry in filen
    static int files = 0;
```

```
<llobj() remember set of object filenames 69d)&#x27E; (55)
    <llobj() grow filen if not enough space 69g)&#x27E;
    filen[files++] = strdup(pn);
```

```
<llobj() remember file introducing the symbol 69e)&#x27E; (68d)
    s->file = files-1;
```

`filen` is actually a *growing array*:

```
<llobj() locals(arm) 69f)&#x27E; (55) <69c 187a>
    char **nfilen; // new filen
```

```
<llobj() grow filen if not enough space 69g)&#x27E; (69d)
    if((files&15) == 0){
        nfilen = malloc((files+16)*sizeof(char*));
        memmove(nfilen, filen, files*sizeof(char*));
        free(filen);
        filen = nfilen;
    }
```

Uses `free()` 233b and `malloc()` 233a.

Thanks to `Sym.fileX` and `filen`, the two conflicting files can be displayed by the (repeated) code below:

```
diag("incompatible type signatures %lux(%s) and %lux(%s) for %s",
    s->sig, filen[s->file],
    sig, pn,
    s->name);
```

Chapter 6

Loading Libraries

The next step in the linking pipeline after loading the main object files is loading libraries. A library (a `.a` archive) is a collection of object files bundled together. `5l` does not load every object in the library—it selectively loads only those that define symbols referenced but not yet resolved by the main objects. This selective loading is what keeps Plan 9 binaries small: a hello world program pulls in only the few library objects it actually needs.

6.1 Archive library format: `.a`

A `.a` archive is a flat file that concatenates multiple object files together with a header and a symbol table at the front. The format is shared across all Unix systems and dates back to the original PDP-11 tools. Each entry has an `ar_hdr` that stores metadata (name, date, size) as ASCII text—not binary—which makes archives portable and inspectable with simple tools.

```
<constant ARMAG 70a>≡ (273d)
#define ARMAG "!<arch>\n"
```

```
<constant SARMAG 70b>≡ (273d)
#define SARMAG 8
```

```
<constant ARFMAG 70c>≡ (273d)
#define ARFMAG "'\n"
```

```
<constant SARNAME 70d>≡ (273d)
#define SARNAME 16
```

```
<struct ar_hdr 70e>≡ (273d)
struct ar_hdr
{
    char name[SARNAME];

    char date[12];
    char uid[6];
    char gid[6];
    char mode[8];

    char size[10]; // use atolwhex() to get the value
    char fmag[2]; // ARFMAG
};
```

The layout of a `.a` file is:

+-----+		
ARMAG "!\<arch>\n"		8 bytes
+-----+		
ar_hdr (__.SYMDEF)		60 bytes (symbol table entry)
symbol table data		variable
+-----+		
ar_hdr (foo.5)		60 bytes
object file content		variable
+-----+		
ar_hdr (bar.5)		60 bytes
object file content		variable
+-----+		
...		
+-----+		

The first entry after the magic is always the symbol table (`__.SYMDEF`), which maps symbol names to file offsets within the archive. This allows the linker to seek directly to the object file that defines a needed symbol, without scanning every entry.

```
<constant SAR_HDR 71a>≡ (273d)
#define SAR_HDR (SARNAME+44)
```

```
<global symname linker 71b>≡ (290)
char symname[] = SYMDEF;
```

Uses `symname 71b`.

6.2 Loading libraries manually: `5l libxxx.a`

When `5l` encounters a library on the command line, it performs selective loading: it reads the archive's symbol table into memory, then iterates over it looking for symbols that match unresolved references (`SXREF`). For each match, it seeks to the corresponding object file within the archive and loads it with `ldobj()`⁵⁵. The key subtlety is the outer `while(work)` loop: loading one object file may introduce new undefined references (the loaded code calls other functions), so the linker must make multiple passes over the symbol table until a fixpoint is reached—a pass where no new symbols are pulled in. This is the same transitive-closure algorithm that all Unix linkers use, and it is why library order on the command line matters: a library is scanned only when the linker reaches it in the argument list, so dependencies must appear after the objects that reference them.

```
<objfile() other locals 71c>≡ (44a)
struct ar_hdr arhdr;
long off, esym, cnt;
Sym *s;
char pname[LIBNAMELEN];
char name[LIBNAMELEN];
char *e, *start, *stop;
bool work;
int pass = 1;
```

Uses `LIBNAMELEN 233d`.

```

<objfile() when file is a library 72>≡
DBG("%5.2f ldlib: %s\n", cputime(), file);

len = read(f, &arhdr, SAR_HDR);

<objfile() sanity check library header size and content 73a>

esym = SARMAG + SAR_HDR + atolwhex(arhdr.size);
off = SARMAG + SAR_HDR;

/*
 * just bang the whole symbol file into memory
 */
seek(f, off, 0);
cnt = esym - off;
start = malloc(cnt + 10);
cnt = read(f, start, cnt);
if(cnt <= 0){
    close(f);
    return;
}
stop = &start[cnt];
memset(stop, '\0', 10);

work = true;
while(work) {

    DBG("%5.2f library pass%d: %s\n", cputime(), pass, file);
    pass++;
    work = false;
    for(e = start; e < stop; e = strchr(e+5, 0) + 1) {

        s = lookup(e+5, 0);
        // loading only the object files containing symbols we are looking for
        if(s->type == SXREF ||
            (s->type == SNONE && strcmp(s->name, "main") == 0)) {
            sprintf(pname, "%s(%s)", file, s->name);
            DBG("%5.2f library: %s\n", cputime(), pname);

            len = e[1] & 0xff;
            len |= (e[2] & 0xff) << 8;
            len |= (e[3] & 0xff) << 16;
            len |= (e[4] & 0xff) << 24;
            // >> >> >> >>

            seek(f, len, SEEK__START);
            len = read(f, &arhdr, SAR_HDR);
            <objfile() sanity check entry header 73b>
            len = atolwhex(arhdr.size);

            // loading the object file containing the symbol
            ldobj(f, len, pname);

            if(s->type == SXREF) {
                diag("%s: failed to load: %s", file, s->name);
                errexit();
            }
            work = true; // maybe some new SXREF has been found in ldobj()
            <objfile() an SXREF was found hook 77a>
        }
    }
}

```

```

    }
}
return;

bad:
    diag("%s: bad or out of date archive", file);
out:
    close(f);

```

Uses [DBG 279](#), [SNONE 33b](#), [SXREF](#), [atolwhex\(\) 235c](#), [diag\(\) 229d](#), [errorexit\(\) 229b](#), [ldobj\(\) 55](#), [lookup\(\) 32a](#), and [malloc\(\) 233a](#).

```

⟨objfile() sanity check library header size and content 73a⟩≡ (72)
    if(len != SAR_HDR) {
        diag("%s: short read on archive file symbol header", file);
        goto out;
    }
    if(strncmp(arhdr.name, symname, strlen(symname))) {
        diag("%s: first entry not symbol header", file);
        goto out;
    }

```

Uses [diag\(\) 229d](#) and [symname 71b](#).

```

⟨objfile() sanity check entry header 73b⟩≡ (72)
    if(len != SAR_HDR)
        goto bad;
    if(strncmp(arhdr.fmag, ARFMAG, sizeof(arhdr.fmag)))
        goto bad;

```

6.3 Loading libraries semi automatically: 5l -lxxx

Like Unix linkers, 5l supports a -l shorthand: 5l -lfoo is equivalent to 5l libfoo.a, with the linker searching a list of directories for the library. The search path defaults to /arm/lib/ and can be extended with -L.

6.3.1 Library search path

```

⟨global libdir 73c⟩≡ (291a)
    // growing_array<dirname>
    char** libdir;

```

```

⟨global nlibdir 73d⟩≡ (291a)
    // index of next free entry in libdir
    int nlibdir = 0;

```

Uses [nlibdir 73d](#).

```

⟨global maxlibdir 73e⟩≡ (291a)
    // index of last free entry in libdir
    static int maxlibdir = 0;

```

Uses [maxlibdir-1 73e](#).

6.3.2 5l -L

```

⟨main() command line processing(arm) 73f⟩+≡ (38e) <42b 170d>
    case 'L':
        addlibpath(EARGF(usage()));
        break;

```

`<main() locals(arm) 74a>≡` (38e) 74c▷
char *root;

`<constant LIBNAMELEN 74b>≡` (279)
#define LIBNAMELEN 300

`<main() locals(arm) 74c>+≡` (38e) ◁74a
int c;
char name[LIBNAMELEN];
char *a;

`<main() addlibpath("/thestring/lib") or ccroot 74d>≡` (39c)
`<main() change root if ccroot 74e>`

```
// usually /{thestring}/lib/ as root = ""
snprint(name, sizeof(name), "%s/%s/lib", root, thestring);
addlibpath(name);
```

`<main() change root if ccroot 74e>≡` (74d)
root = getenv("ccroot");

```
if(root != nil && *root != '\0') {
    if(!fileexists(root)) {
        diag("nonexistent $ccroot: %s", root);
        errexit();
    }
}
else
    root = "";
```

`<function addlibpath 74f>≡` (291a)

```
void
addlibpath(char *arg)
{
    char **p;

    // growing array libdir
    if(nlibdir >= maxlibdir) {
        if(maxlibdir == 0)
            maxlibdir = 8;
        else
            maxlibdir *= 2;
        p = malloc(maxlibdir*sizeof(*p));
        <addlibpath() sanity check p 74g>
        memmove(p, libdir, nlibdir*sizeof(*p));
        free(libdir);
        libdir = p;
    }
}
```

```
libdir[nlibdir++] = strdup(arg);
```

Uses `free()` 233b, `libdir` 73c, `malloc()` 233a, `maxlibdir-1` 73e, and `nlibdir` 73d.

`<addlibpath() sanity check p 74g>≡` (74f)

```
if(p == nil) {
    diag("out of memory");
    errexit();
}
```

Uses `diag()` 229d and `errexit()` 229b.

6.3.3 5l -lxxx

```
<objfile() adjust file if -lxxx filename 75a>≡ (44a)
if(file[0] == '-' && file[1] == 'l') {
    snprintf(pname, sizeof(pname), "lib%s.a", file+2);
    e = findlib(pname);
    if(e == nil) {
        diag("cannot find library: %s", file);
        errexit();
    }
    snprintf(name, sizeof(name), "%s/%s", e, pname);
    file = name;
}
```

Uses `diag()` 229d, `errexit()` 229b, and `findlib()` 75b.

```
<function findlib 75b>≡ (291a)
char*
findlib(char *file)
{
    int i;
    char name[LIBNAMELEN];

    for(i = 0; i < nlibdir; i++) {
        snprintf(name, sizeof(name), "%s/%s", libdir[i], file);
        if(fileexists(name))
            return libdir[i];
    }
    return nil;
}
```

Uses `LIBNAMELEN` 233d, `fileexists()` 235b, `libdir` 73c, and `nlibdir` 73d.

6.4 Loading libraries automagically: #pragma lib "libxxx.a"

This is one of the most original features of the Plan 9 toolchain, alongside safe linking (Section ??). In Plan 9, header files contain a `#pragma lib "libxxx.a"` directive. When `5c` compiles a source file that includes such a header, it embeds an `AHISTORY` instruction with a special marker (`offset == -1`) in the object file. When `5l` loads that object file, it records the library name. After all command-line objects are loaded, `5l` automatically loads those recorded libraries via `loadlib()` 76e. The result is that you never need to specify libraries on the link command line—just include the right header and the linker does the rest. This is the Plan 9 equivalent of what `pkg-config --libs` and build systems like `CMake` do on Unix, but built directly into the compiler and linker.

```
<llobj() in AHISTORY case, if pragma lib 75c>≡ (153)
if(p->to.offset == -1) {
    addlib(pn);
    histfrogp = 0;
    goto loop;
}
```

Uses `addlib()` 77b and `histfrogp` 157d.

```
<main() load implicit libraries 75d>≡ (42d)
if(!debug['l'])
    loadlib();
```

```
<main() if rare condition do not set SXREF for INITENTRY, else 75e>≡ (42c) 213g>
if(debug['l']) {}
else
```

```
<global library 76a>≡ (291a)
```

```
// array<option<filename>>
char* library[50];
```

```
<global libraryp 76b>≡ (291a)
```

```
// index of first free entry in library array
int libraryp;
```

```
<global libraryobj 76c>≡ (291a)
```

```
char* libraryobj[50];
```

```
<function loadlib simple version 76d>≡
```

```
void
loadlib(void)
{
    int i;

    for(i=0; i<libraryp; i++) {
        DBG("%5.2f autolib: %s (from %s)\n", cputime(), library[i], libraryobj[i]);
        objfile(library[i]);
    }
}
```

The full version of `loadlib()` handles mutually dependent libraries. After loading all recorded libraries, it checks whether any `SXREF` symbols remain in the hash table. If so, it loops and loads the libraries again—each pass may resolve symbols that allow the next pass to pull in more objects. This is a fixpoint computation: the loop terminates when a full pass adds nothing new.

```
<function loadlib 76e>≡ (291a)
```

```
void
loadlib(void)
{
    int i;
    long h;
    Sym *s;
loop:
    <loadlib() reset xrefresolv 76g>
    for(i=0; i<libraryp; i++) {
        DBG("%5.2f autolib: %s (from %s)\n", cputime(), library[i], libraryobj[i]);
        objfile(library[i]);
    }
    <loadlib() if xrefresolv 76h>
}
```

Uses `DBG 279`, `library 76a`, `libraryobj 76c`, `libraryp 76b`, and `objfile() 44a`.

```
<global xrefresolv 76f>≡ (283b)
```

```
bool xrefresolv;
```

```
<loadlib() reset xrefresolv 76g>≡ (76e)
```

```
xrefresolv = false;
```

Uses `xrefresolv 76f`.

```
<loadlib() if xrefresolv 76h>≡ (76e)
```

```
if(xrefresolv)
    for(h=0; h<nelem(hash); h++)
        for(s = hash[h]; s != S; s = s->link)
            if(s->type == SXREF) {
                DBG("symbol %s still not resolved, looping\n", s->name); //pad
                goto loop;
            }
}
```

Uses `DBG 279`, `S 33b`, `SXREF`, and `xrefresolv 76f`.

`<objfile() an SXREF was found hook 77a>`≡

(72)

```
xrefresolv = true;
```

Uses `xrefresolv 76f`.

`addlib()`^{77b} reconstructs a library path from the AHISTORY data that 5c embedded in the object file. The path components are stored in `histfrog[]`, a global array that `ldobj()` populates as it processes AHISTORY instructions. `addlib()` concatenates them, substituting `$0` with the architecture character (e.g., '5') and `$M` with the architecture string (e.g., "arm"), then deduplicates against libraries already recorded.

`<function addlib 77b>`≡

(291a)

```
/// ldobj(case AHISTORY and local_line == -1 special mark) -> <>
void
addlib(char *obj)
{
    char fn1[LIBNAMELEN], fn2[LIBNAMELEN], comp[LIBNAMELEN];
    char *p, *name;
    int i;
    bool search;

    if(histfrogp <= 0)
        return;

    name = fn1;
    search = false;
    if(histfrog[0]->name[1] == '/') {
        sprintf(name, "");
        i = 1;
    } else if(histfrog[0]->name[1] == '.') {
        sprintf(name, ".");
        i = 0;
    } else {
        sprintf(name, "");
        i = 0;
        search = true;
    }

    for(; i<histfrogp; i++) {
        snprintf(comp, sizeof comp, histfrog[i]->name+1);

        // s/$0/<thechar>/
        for(;;) {
            p = strstr(comp, "$0");
            if(p == nil)
                break;
            memmove(p+1, p+2, strlen(p+2)+1);
            p[0] = thechar;
        }
        // s/$M/<thestring>/
        for(;;) {
            p = strstr(comp, "$M");
            if(p == nil)
                break;
            if(strlen(comp)+strlen(thestring)-2+1 >= sizeof comp) {
                diag("library component too long");
                return;
            }
            memmove(p+strlen(thestring), p+2, strlen(p+2)+1);
            memmove(p, thestring, strlen(thestring));
        }
    }
}
```

```

    if(strlen(fn1) + strlen(comp) + 3 >= sizeof(fn1)) {
        diag("library component too long");
        return;
    }
    if(i > 0 || !search)
        strcat(fn1, "/");
    strcat(fn1, comp);
}

cleanname(name);

if(search){
    p = findlib(name);
    if(p != nil){
        snprintf(fn2, sizeof(fn2), "%s/%s", p, name);
        name = fn2;
    }
}

for(i=0; i<libraryp; i++)
    if(strcmp(name, library[i]) == 0)
        return;
if(libraryp == nelem(library)){
    diag("too many autolibs; skipping %s", name);
    return;
}

p = malloc(strlen(name) + 1);
strcpy(p, name);
library[libraryp] = p;
p = malloc(strlen(obj) + 1);
strcpy(p, obj);
libraryobj[libraryp] = p;
libraryp++;
}

```

Uses LIBNAMELEN 233d, diag() 229d, findlib() 75b, histfrog 157a, histfrogp 157d, library 76a, libraryobj 76c, libraryp 76b, malloc() 233a, thechar 38a, and thestring 38b.

Chapter 7

Resolving

Once the object files and libraries have been loaded, the next step in the linking pipeline is to resolve the symbol references in the instructions of those object files. Here is the responsible code:

```
<main() resolving phase 79>≡ (42d)
<main() if export table or dynamic module(arm) 171b>

patch();
<main() call doprofixx() if profiling 162b>
noops();

dodata();
dotext();
```

`patch()`⁸² builds a graph of code instructions, `noops()`⁸⁷ rewrites virtual instructions in machine instructions, `dodata()`^{92c} layout data by assigning a memory address to each global, and finally `dotext()`^{94b} does a similar thing for the code, assigning a real code address to each instruction.

In the first section of this chapter, I will discuss the different issues in symbol resolution. I will also explain the need for the 4 functions above and the rationale for the order of their calls. Then, the following sections will each explain one of the 4 functions above.

7.1 Issues in symbol resolution

The linker has mainly two kinds of references to resolve:

- *Code references* in branching instructions, e.g., `BL foo(SB)`, which ultimately should be transformed in an ARM instruction with, as an operand, a concrete address in the code section, e.g., `BL 0x1020`
- *Data references* in memory instructions, e.g., `MOVW hello(SB), R1`, which ultimately should be transformed in an ARM instruction with, as an operand, a concrete address in the data section, e.g., `LDR 0x7014(R0), R1`.¹

Note that some branching instructions have been partially resolved at this point. Indeed, branching to a label (e.g., `B loop`), or relative jumps using the pseudo-register PC (e.g., `B 2(PC)`), have already been resolved and converted by the assembler 5a in *absolute jumps* (e.g., B 13). See the ASSEMBLER book [Pad15a]. Those absolute jumps have even been relocated by the linker to a new memory address origin in Section 5.4.1 (e.g., B 113).

Resolving and converting the remaining branching instructions (e.g., `BL foo(SB)`) to absolute jumps is very easy thanks to the symbol table and the `Sym.valueX` field of procedure symbols. There are still some remaining issues though that makes the resolving step non-trivial as you will see in the following sections.

¹Remember from the ASSEMBLER book [Pad15a] that `MOVW` is actually a virtual instruction which unifies the two separate ARM instructions `LDR` (load register) and `STR` (store register).

7.1.1 Virtual program counter versus real code address

The first issue is that the value in the absolute jump instructions I mentioned before refers to a *virtual program counter* which is not a *real code address*².

Indeed, the virtual program counter starts at 0 and is incremented by 1 in `ldobj()`⁵⁵ after each code instruction read. Real code addresses on the other hand start at `INITTEXT`^{40e} (4128 under Plan 9), and because the ARM uses fixed-length instructions of 4 bytes, are always a multiple of 4.

Unfortunately, going from a virtual program counter to a real code address can not be done simply by multiplying by 4 and adding `INITTEXT`. Indeed, as said in Section 4.4.2, the instructions in object files do not match exactly ARM instructions. This means that one instruction in the object file may lead to the generation of multiple ARM instructions in the executable, or no instruction at all sometimes. Even though the assembly language of 5a tries to mimic closely the ARM instruction set, there is still a mismatch for a few reasons explained below:

- *Virtual instructions* such as `DIV` have no counterpart in the ARM. `DIV` and `MOD` are converted by 51 in a series of ARM instructions which ends with the call to respectively `_div()` and `_mod()` of the core C library (see Section 12.6.2) which implement in software the division and modulo algorithms (using `ADD`, `MUL`, `SUB`, etc).
- The *pseudo-instruction* `TEXT` leads to the generation of 0, 1, or many ARM instructions depending on the situation. Indeed, if the procedure use some locals, it will need an extra ARM instruction to decrement the *stack pointer* (see Section 7.3.2). Moreover, if profiling is enabled, a few more ARM instructions will be added for the instrumentation of the procedure to gather statistics (see Chapter 11). The same is true for `RET`.
- *Constraints* on immediate constants and offsets imposed by the ARM are relaxed by 5a and the object file. As said before, the ARM uses fixed-length instructions of 4 bytes, so for instructions involving immediate constants (e.g., `ADD $10256, R1, R2`) or offsets (e.g., `MOVW 54000(R0), R2`), the range of those numbers is quite limited. For arithmetic instructions, only 12 bits of the ARM instruction are available to encode the constant. This is why even a regular instruction such as `ADD` may lead to the generation of multiple ARM instructions. Indeed, if the instruction uses a big number, this number must first be loaded in a temporary register with an extra instruction (see Section 9.8).

Because of all of this, it is not trivial to convert a virtual program counter reference in a branching instruction to a real code address. The solution adopted by 51 operates in three steps:

1. The function `patch()`⁸² builds a *graph of code instructions* by transforming the use of a virtual program counter reference to a real pointer (`Instr*`). This pointer will be stored in the `Instr.condX` field of the branching instruction and will point to the target instruction having the specified virtual program counter in his `Instr.pcX`.
2. The function `dotext()`^{94b} sets `pc`^{53a} to `INITTEXT`] and iterates over all the code instructions. For each instruction, `dotext()` computes the actual number of ARM instructions which are needed for this instruction (via `oplook()`^{102c}), sets `Instr.pcX` to `pc` which both represent now *real program counters*, and increment `pc` accordingly (e.g., by 4 if only one ARM instruction was necessary).
3. The code generator `asmout()`^{108b} when involved with a branching instruction `p` can find the target code address by simply looking at `p->cond->pc`.

²Note that a *real* code address under Plan 9 is actually a *virtual* memory address. But, the use of virtual in this context would be confusing, which is why I will keep to use the term “real code address”.

Thanks to the graph of code instructions, it is also easy after `patch()` to call `noops()`⁸⁷ to rewrite virtual instructions in machine instructions. Indeed, `noops()` can replace a virtual instruction by multiple machine instructions chained together or even delete the instruction without any consequence on the other branching instructions (as long as it maintains carefully the `Instr.condX` pointers in instructions pointing to the original instruction). Before the graph of instructions, inserting or deleting an instruction would have forced to assign a new virtual program counter to all the following instructions and to relocate every branching instructions.

Thanks to the graph, it is also possible to perform some optimizations before code generation, such as dead code elimination, as you will see in Section 12.3.

7.1.2 Data address and code size mutual dependency

The second issue which makes the resolving step non-trivial is the mutual dependency between the layout of code and data. Indeed, to generate code one needs to resolve memory instructions involving globals which means one must know the address of those globals. This suggests to layout first data and then the code. But, because those globals are in the data section, which is after the code section, one needs first to know at least the size of the code section to start to layout data. This suggests to layout first the code and then the data, hence the mutual dependency.

Of course, 51 just needs the size of the code section to layout data, so it could first layout the code using fake addresses for the globals referenced in memory instructions. 51 would get a size for the code section which it could use as the starting point of the data section, to layout data. Finally, 51 would layout the code a second time using real addresses for the globals this time, to generate the real code.

One difficulty is that the size of the code section computed during the first layout would be only an estimation. Indeed, some instructions involving globals may have different size depending on the address of the global. Offsets in ARM instructions have a limited range, as mentioned in the previous section. This means some memory instructions may require the use of an extra instruction if the address of the global happens to be large. Using fake addresses during the first layout would be imprecise.

The solution adopted by 51 to solve the mutual dependency problem operates in four steps:

1. The layout of data is done first, via `dodata()`^{92c}, by iterating over all the globals. But, each global is assigned a memory address (in `Sym.valueX`) as *an offset to the start of the data section*, not an absolute address. Note that the old value in `Sym.valueX` contained the size of the global which is used to layout globals one after the other.
2. `dotext()`^{94b} then layouts code by iterating over all the instructions. But, it assumes the *reserved register* R12 will, at *run-time*, when the executable is executed, contain `INITDAT`^{40g}, the address of the start of the data section. It also assumes that operands involving globals such as `MOVW hello(SB), R1` can be transformed later (in `asmout()`^{108b}) as an *indirect with offset* operand (see the ASSEMBLER book [Pad15a]). This operand can then use R12 as the *base* and the `Sym.valueX` of the global (computed previously) as the *offset*, e.g., `LDR 14(R12), R1`. In the same way, instructions involving global addresses such as `MOVW $hello(SB), R1` can be transformed as `ADD $14, R12, R1`. That way, `dotext()` can compute accurately the size of the instruction.
3. One of the first instruction of the program must initialize R12 with the value of `INITDAT`. This value is recorded in the executable as a special symbol, `setR12`, using a form of reflection explained in Section 7.7. In practice, the function `_main()` from the core C library is the first function executed by the program (see Section 4.1.3). This is why one of its first instruction is: `MOVW $setR12(SB), R12`.
4. The instruction `MOVW $setR12(SB), R12` itself uses the address of a global but it must not be transformed as `ADD 0, R12, R12` like other similar instructions. Indeed, the whole point of the instruction is to initialize R12, and so it can not rely on R12. To *bootstrap*, this address is recognized in a special way in the code

generator in `asmout()` and leads to the generation of an instruction which uses the absolute address of `setR12`, e.g., `MOV $0x7000, R12`.

All of this explains the order of the calls you saw at the beginning of this chapter:

```
patch();
noops();

dodata();
dotext();
```

Now that you have a better picture of how things work together to resolve symbols, I can go through the code of each of the functions above, starting with `patch()`⁸².

7.2 Building the code instructions graph: `patch()`

As explained before, the goal of `patch()` is to set the `Instr.condX` field of branching instructions to point to the right target instruction. Essentially `patch()` will iterate over all the code instructions `p` and for branch instructions it will look for the target instruction `q` with the virtual program counter `q->pc` equal to the absolute jump value `p->to.offset`:

```
<function patch(arm) 82>≡ (286b)
  /// main -> <>
  void
  patch(void)
  {
    Prog *p;
    Prog *q;
    long c; // not ulong?
    <patch() other locals 83a>

    DBG("%5.2f patch\n", cputime());

    <patch() initialisations 85a>

    // pass 1
    for(p = firstp; p != P; p = p->link) {
      <adjust curtext when iterate over instructions p 36h>

      <patch() resolve branch instructions using symbols 83b>

      if(p->to.type == D_BRANCH && p->cond != UP) {
        c = p->to.offset; // target pc
        <patch() find Prog reference q with q->pc == c 84>
        p->cond = q;
      }
    }
    // pass 2
    <patch() optimisation pass 183e>
  }
}
```

Uses `DBG 279`, `P 36c`, `UP 279`, and `firstp 36a`.

The extra condition `p->cond != UP` above is not very important and can be ignored for now. It will be explained later (see [UP²⁷⁹](#)). Normally, before the call to `patch()`⁸², `Instr.condX` should be a null pointer³ which is different than `UP`.

`patch()` is a complex function. I split it in a few chunks to facilitate its comprehension. To find `q`, `patch()` needs to first initialize some data structures that makes the search more efficient. It also needs to (partially) resolve branching instructions using symbols and convert them to absolute jumps. That way all branching instructions become uniforms. The following sections will explain those different steps.

7.2.1 Resolving branch instructions using symbols

Converting the use of symbols in branching instruction (e.g., `B foo(SB)`) to absolute jumps (e.g., `B 113`) is trivial thanks to the `Sym.valueX` of procedure symbols which contains the virtual program counter of the procedure:

```
⟨patch() other locals 83a⟩≡ (82)
```

```
Sym *s;
// enum<Opcode>
int a;
```

```
⟨patch() resolve branch instructions using symbols 83b⟩≡ (82)
```

```
a = p->as;
if((a == ABL || a == AB) &&
    p->to.type != D_BRANCH && // must be D_OREG then
    p->to.sym != S) {
    s = p->to.sym;
    switch(s->type) {
    case STEXT:
        p->to.offset = s->value;
        p->to.type = D_BRANCH;
        break;
    ⟨patch() switch section type for branch instruction, cases 83c⟩
    }
}
```

Uses [S 33b](#) and [STEXT 156d](#).

If the procedure can not be found, an error is reported:

```
⟨patch() switch section type for branch instruction, cases 83c⟩≡ (83b) 175g▷
```

```
// SNONE, SXREF, etc
default:
    diag("undefined: %s\n%P", s->name, p);
    s->type = STEXT;
    s->value = 0;
    break;
```

Uses [STEXT 156d](#) and [diag\(\) 229d](#).

To avoid reporting multiple times the same error, the symbol of the undefined procedure is modified to become a (fake) defined procedure. That way, if another instruction later calls the same procedure, 51 will not report an error. Note that there is no risk of generating a wrong executable though as the call to `diag()`^{229d} will prevent such a thing (see [Appendix B](#)).

7.2.2 Finding instruction at pc

The naive way to find the instruction `q` with a certain `pc` is a linear search over all the instructions, by following `Instr.linkX`. To optimize the search, `patch()`⁸² relies on another pointer, `Instr.forwd` which will point not to the next instruction, but to a few more instructions forward (hence the name), as shown in [Figure 7.1](#).

```
⟨Prog other fields 83d⟩+≡ (34b) <35f 106a▷
```

```
Prog* forwd;
```

³See the instruction `p->cond = P`; in [ldobj\(\)](#)⁵⁵.

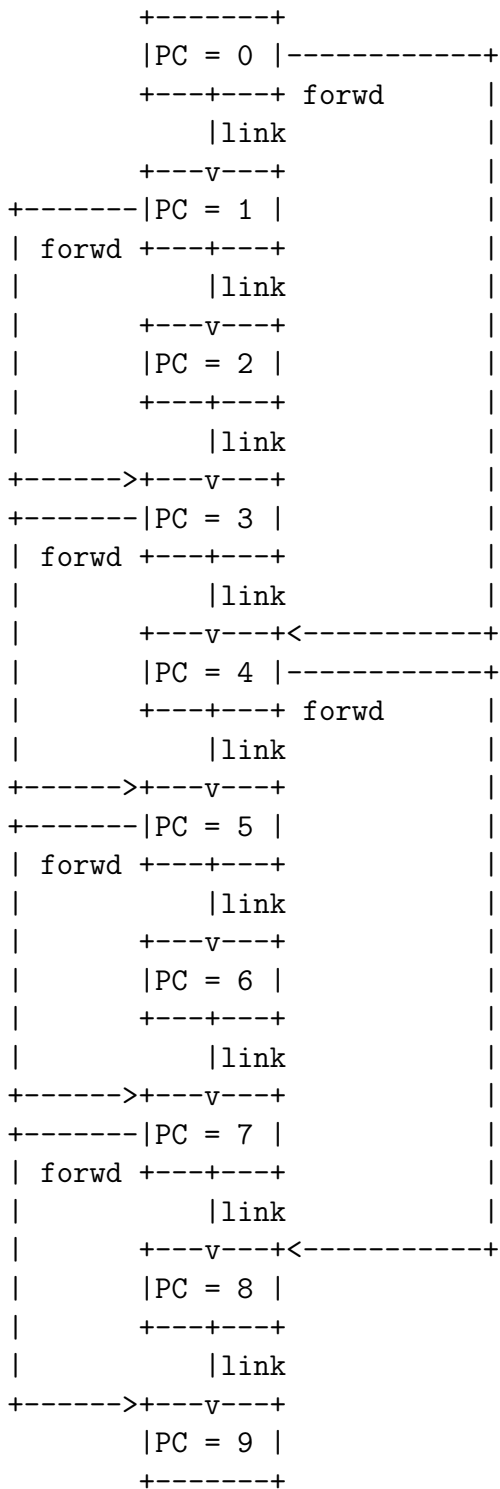


Figure 7.1: Forward pointers.

That way you can make bigger jumps sometimes:

```

⟨patch() find Prog reference q with q->pc == c 84⟩≡
for(q = firstp; q != P;) {
  if((q->forwd != P) && (c >= q->forwd->pc)) {
    q = q->forwd; // big jump
  } else {
    if(c == q->pc)
      break; // found it!

```

(82)

```

    q = q->link; // small jump
}
}
if(q == P) {
    diag("branch out of range %ld\n%P", c, p);
    p->to.type = D_NONE;
}

```

Uses P 36c, diag() 229d, and firstp 36a.

By carefully setting Instr.forwdX, the search can become a *binary search*. Indeed, to find for instance the instruction with PC==7 in Figure 7.1, shortened to p_7 , one needs only 3 steps: starting from p_0 51 can go directly to p_4 with forwd, then p_5 with link and finally p_7 with another forwd. A linear search with only link would require 7 steps.

7.2.3 Indexing pc, forward links overlay

The Instr.forwdX fields are set by mkfwd():

```

<patch() initialisations 85a>≡ (82)
    mkfwd();

```

Uses mkfwd() 85c.

mkfwd() essentially builds a *binary search tree* on top of a linked list. It relies on the constant LOG which controls at the same time the lengths of the forward arcs and the depth of the search tree:

```

<constant LOG 85b>≡ (286b)
    #define LOG 5

```

To illustrate mkfwd() though, I will use LOG=2, which leads to Figure 7.1 when mkfwd() is applied to a list of 10 instructions.

```

<function mkfwd 85c>≡ (286b)
    /// main -> patch -> <>
    void
    mkfwd(void)
    {
        long cnt[LOG]; // (length of arc)/LOG at a certain level (constant)
        long dwn[LOG]; // remaining elements to skip at a level (goes down)
        Prog *lst[LOG]; // past instruction saved at a level
        Prog *p;
        int i; // level

        <mkfwd() initializes cnt, dwn, lst 86a>

        i = 0;
        for(p = firstp; p != P; p = p->link) {
            <adjust curtext when iterate over instructions p 36h>
            p->forwd = P;

            <mkfwd() in for loop, add forward links from past p in lst to p 86b>
        }
    }

```

Uses LOG-15 85b, P 36c, and firstp 36a.

The code of mkfwd()^{85c} is very subtle which is why I split it in multiple chunks. It iterates over all the instructions p and at certain *frequencies* adds a forward link from a past instruction saved in lst[i] to the

current instruction *p*. It does so for different *levels* *i*. The frequency (divided by LOG^{85b}) for a certain level *i* is stored in `cnt[i]` initialized by the following code:

```
<mkfwd() initializes cnt, dwn, lst 86a>≡ (85c)
for(i=0; i<LOG; i++) {
    if(i == 0)
        cnt[i] = 1;
    else
        cnt[i] = LOG * cnt[i-1];
    dwn[i] = 1;
    lst[i] = P;
}
```

Uses LOG-15 85b and P 36c.

Here are the initial values of `cnt` for different LOG:

```
// LOG = 2
cnt[] = {1, 2};
// LOG = 3
cnt[] = {1, 3, 9};
// LOG = 4
cnt[] = {1, 4, 16, 64};
// LOG = 5
cnt[] = {1, 5, 25, 125, 625};
```

I can finally show the body of the loop of `mkfwd()`:

```
<mkfwd() in for loop, add forward links from past p in lst to p 86b>≡ (85c)
// first loop, the levels
i--;
if(i < 0)
    i = LOG-1;

// second loop, the frequency at a certain level
dwn[i]--;
if(dwn[i] <= 0) {
    dwn[i] = cnt[i];

    if(lst[i] != P)
        lst[i]->forwd = p; // link from past p to p
    lst[i] = p;
}
```

Uses LOG-15 85b and P 36c.

The idea is to first loop over the different levels *i*, and then loop over the frequency at a certain level *i*. So, for LOG=2, `mkfwd()` will add forward links of length 4 (every LOG * 2 instructions), and of length 2 (every LOG * 1 instructions), as illustrated in Figure 7.1.

Here is a simple trace of `mkfwd()` for LOG=2 which could help understand the algorithm. It shows the value of the different variables at the *end* of each iteration of the `for` loop:

```
// does not change
LOG = 2
cnt[] = { 1, 2 };

// before for loop
i = 0  dwn[] = { 1, 1 } lst[] = {P, P};
```

```

// end of each iteration
p=p0 i=1 dwn[]={1,2} lst[]={P, p0} // save p0
p=p1 i=0 dwn[]={1,2} lst[]={p1,p0} // save p1
p=p2 i=1 dwn[]={1,1} lst[]={p1,p0} // decrement dwn[1]
p=p3 i=0 dwn[]={1,1} lst[]={p3,p0} // + p1 -> p3 forwd link
p=p4 i=1 dwn[]={1,2} lst[]={p3,p4} // + p0 -> p4 forwd link
p=p5 i=0 dwn[]={1,2} lst[]={p5,p4} // + p3 -> p5 forwd link
p=p6 i=1 dwn[]={1,1} lst[]={p5,p4}
p=p7 i=0 dwn[]={1,1} lst[]={p7,p4} // + p5 -> p7 forwd link
p=p8 i=1 dwn[]={1,2} lst[]={p7,p8} // + p4 -> p8 forwd link
p=p9 i=0 dwn[]={1,1} lst[]={p9,p8} // + p7 -> p9 forwd link

```

For LOG=5, mkfwd() will add forward links of lengths 3125 ($5 * 625 = 5^5$), 625 (5^4), 125 (5^3), 25 (5^2), and 5 (5^1).

7.3 Virtual opcodes rewriting: noops()

As explained before, noops() rewrites pseudo and virtual instructions which do not have a corresponding ARM opcode (no op), e.g., ARET, to machine instructions which have one. It works in two passes over the list of code instructions. The first pass mostly *marks* certain instructions in Instr.markX. This pass can also rely on q which stores the previous instruction to p. The second pass possibly uses the mark to transform virtual instructions:

```

⟨function noops(arm) 87⟩≡ (291b)
  /// main -> <>
  void
  noops(void)
  {
    Prog *p, *q, *q1;
    // enum<Opcode>
    int o;

    /*
     * find leaf subroutines
     * strip NOPs
     * expand RET
     */

    DBG("%5.2f noops\n", cputime());

    // pass 1, mark or delete
    curtext = P;
    q = P;
    for(p = firstp; p != P; p = p->link) {
      ⟨adjust curtext when iterate over instructions p 36h⟩

      switch(p->as) {
        ⟨noops() first pass switch opcode cases 88b⟩
      }
      q = p;
    }

    // pass 2, transform
    curtext = P;

```

```

for(p = firstp; p != P; p = p->link) {
    <adjust curtext when iterate over instructions p 36h>

    o = p->as;
    switch(o) {
        <noops() second pass switch opcode cases 88c>
    }
}
}

```

Uses DBG 279, P 36c, curtext 36f, and firstp 36a.

The following sections will detail those passes.

7.3.1 Leaf procedure optimisation

An important concept related to the ARM architecture is whether a procedure is a leaf (see the ASSEMBLER book [Pad15a]). Such procedures will be marked in the first pass of `noops()`⁸⁷:

```

<Mark cases 88a>≡ (35g) 184b▷
LEAF = 1<<2,

```

A *leaf* is a procedure which does not call other procedures. Such a procedure is thus a leaf in the *call tree*:

```

<noops() first pass switch opcode cases 88b>≡ (87) 90▷

```

```

case ATEXT:
    p->mark |= LEAF;
    break;

case ABL:
    if(curtext != P)
        curtext->mark &= ~LEAF;
    // fallthrough
<noops() first pass switch opcode ABL fallthrough 91>

```

Uses LEAF 279, P 36c, and curtext 36f.

Depending on whether a procedure is a leaf, the code for ATEXT and ARET can be optimized to not use memory at all and just use the *link register* R14, as you will see in the following sections.

7.3.2 ATEXT patching

If a procedure is a leaf, there is no need to save R14 in the stack. Indeed, such a procedure will not contain any BL instruction, and so will not overwrite R14. 51 does this optimization though only if the procedure does not declare also any locals (e.g., ATEXT `foo(SB), $0`).

Otherwise, if the procedure is not a leaf, or use some locals, then an extra word (+4 bytes) is reserved in the stack to save the return address to the caller stored in R14.

```

<noops() second pass switch opcode cases 88c>≡ (87) 89b▷
case ATEXT:

```

```

    if(p->to.offset <= 0) {
        if(curtext->mark & LEAF) {
            // to compensate further + 4 to get autosize == 0.
            p->to.offset = -4;
        }
    }
    autosize = p->to.offset + 4;

```

```

<noops() in second pass, if size local was -4 and not a leaf 89a>

```

```

if((curtext->mark & LEAF) && (autosize == 0))

```

```

    break;
// else

// MOVW.W R14, -autosize(R13)
q1 = prg();
q1->as = AMOVW;
q1->scond |= C_WBIT;
q1->line = p->line; // origin tracking
q1->from.type = D_REG;
q1->from.reg = REGLINK;
q1->to.type = D_OREG;
q1->to.reg = REGSP;
q1->to.offset = -autosize;

// insert_after(p, q1)
q1->link = p->link;
p->link = q1;
break;

```

Uses LEAF 279, autosize 65d, curtext 36f, and prg() 43a.

The single instruction `MOVW.W R14, -autosize(R13)`, which uses the *special bit* `.W` (see the ASSEMBLER book [Pad15a]), is equivalent to the sequence:

```

MOVW R14, -autosize(R13)
SUB autosize, R13, R13

```

Note that the ATEXT instruction in `p` is kept. It is not replaced by a new instruction `q1` but instead `q1` is added after `p`. That way, further iterations over the list of instructions can still rely on `curtext`^{36f}.

In some situations, even if the procedure is not a leaf, one would like to not save R14 in the stack. In that case, the programmer needs to declare his procedure with `-4` for the size of its locals (e.g., `TEXT foo(SB), $-4`).

```

⟨noops() in second pass, if size local was -4 and not a leaf 89a)≡ (88c)
    if((autosize == 0) && !(curtext->mark & LEAF)) {
        DBG("save suppressed in: %s\n", curtext->from.sym->name);
        curtext->mark |= LEAF;
    }

```

Uses DBG 279, LEAF 279, autosize 65d, and curtext 36f.

7.3.3 ARET rewriting

ARET is complementary to ATEXT. Its transformation is pretty straightforward:

```

⟨noops() second pass switch opcode cases 89b)+≡ (87) <88c 193b>
    case ARET:
        if((curtext->mark & LEAF) && (autosize == 0)) {
            // B 0(R14)
            p->as = AB;
            p->from = zprg.from;
            p->to.type = D_OREG;
            p->to.offset = 0;
            p->to.reg = REGLINK;
        } else {
            // MOVW.P autosize(R13), R15
            p->as = AMOVW;
            p->scond |= C_PBIT;
            p->from.type = D_OREG;
            p->from.offset = autosize;
            p->from.reg = REGSP;
            p->to.type = D_REG;
        }

```

```

TEXT foo(SB), $4      5000: MOVW R14, $-8(R13)
...                  5004: SUB $8, R13, R13
                    ...
BL bar(SB)           ==> 5020: BL 6000
...                  ...
RET                  5040: ADD $8, R13, R13
                    5044: MOVW $-8(R13), R15

TEXT bar(SB), $8      6000: MOVW R14, $-12(R13)
...                  6004: SUB $12, R13, R13
                    ...
BL leaf(SB)          ==> 6052: BL 7000
...                  ...
RET                  6080: ADD $12, R13, R13
                    6084: MOVW $-12(R13), R15

TEXT leaf(SB), $0     7000: /* nothing */
...                  ==>    ...
RET                  7028: B (R14)

```

Figure 7.2: TEXT and RET translations.

```

    p->to.reg = REGPC;
}
break;

```

Uses LEAF 279, autosize 65d, curtext 36f, and zprg 43b.

The single instruction `MOVW.P autosize(R13), R15`, which uses the special bit `.P` (see the `ASSEMBLER` book [Pad15a]), is equivalent to the sequence:

```

ADD autosize, R13, R13
MOVW -autosize(R13), R15

```

Note that this time the instruction `p` is overwritten; `RET` really disappears.

Figure 7.2 summarizes the possible translations for `ATEXT` and `ARET` on a simple program with 3 procedures `foo`, `bar`, and `leaf` where `leaf` is the only leaf procedure. The numbers in the middle column, e.g., 5000, represent the code addresses of the generated code in the executable. Figure 7.3 shows the stack after `foo` has called `bar` which has called `leaf`.

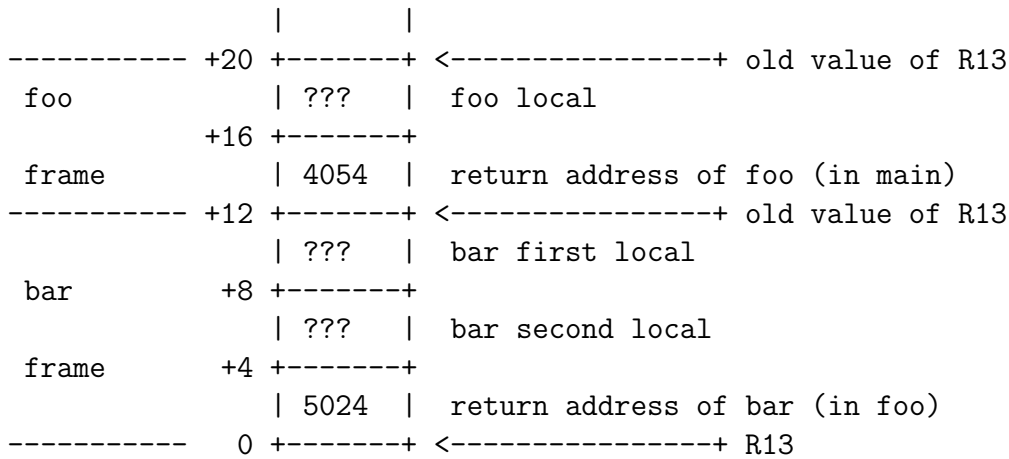
7.3.4 ANOP stripping

Another virtual instruction rewritten by `noops()`⁸⁷ is `ANOP`. The semantic of `ANOP` is to do nothing (nope), which is why it can be removed from the graph of instructions:

```

<noops() first pass switch opcode cases 90>+≡ (87) <88b 205a>
case ANOP:
    q1 = p->link;
    q->link = q1; // q is a non-ANOP before p
    q1->mark |= p->mark;
    continue; // no q = p; so q remains a non-ANOP

```



R14 = 6056 = return address of leaf (in bar)

Figure 7.3: Stack before B (R14) in leaf().

The compiler 5c occasionally generates code using ANOP (see the COMPILER book [Pad16a]). 51 does too as you will see in Section 12.4.

Note that some instructions could have this deleted ANOP instruction p as their branching target in Instr.condX. In that case their Instr.condX field must be updated:

```

<noops() first pass switch opcode ABL fallthrough 91>≡ (88b)
case AB:
case ABEQ: case ABNE:
case ABHS: case ABLO:
case ABMI: case ABPL:
case ABVS: case ABVC:
case ABHI: case ABLS:
case ABGE: case ABLT:
case ABGT: case ABLE:
case ABCASE:
    q1 = p->cond;
    if(q1 != P) {
        while(q1->as == ANOP) {
            q1 = q1->link;
        }
        p->cond = q1;
    }
    break;

```

Uses P 36c.

There are other virtual instructions rewritten by noops() beside TEXT, RET, and NOP. Those are the most important one though. Chapter 12 will present the transformation of the remaining virtual instructions: DIV, MOD, and MOVWD.

7.4 Laying out data: dodata()

As explained before, the goal of dodata()^{92c} is to layout globals by essentially modifying the Sym.valueX field of those globals. This field was originally storing the size of the global and will contain, after dodata(), an offset to the start of the data section. dodata() also computes two important globals storing the size of the data and

BSS sections:

```
<global datsize 92a>≡ (283b)
long datsize;
```

```
<global bsssize 92b>≡ (283b)
long bsssize;
```

Those globals are then used to generate the `a.out` header in `asmout()`^{108b}, as shown in Section 4.4.1.

The programmer declares globals and specifies their *size* with `GLOBL`, e.g., `GLOBL foo(SB), $4`. He can also specify their *content* with `DATA`, e.g., `DATA foo(SB)/4, "hey\n"`, in which case the global will be part of the data section (`SDATA`) instead of the BSS section (`SBSS`^{170b}). But, he does not care about the precise layout of those globals in their respective sections. `dodata()` gathers all the global declarations, spread in different object files, and places them next to each other in memory. It also *aligns* those globals at word boundaries to satisfy architecture constraints on memory addressing.

The code is pretty straightforward once all the motivations and the big picture have been presented (see Section 7.1.2):

```
<function dodata(arm) 92c>≡ (286a)
// main -> <>
void
dodata(void)
{
    Prog *p;
    Sym *s;
    // offset to start of data section
    long orig;
    // size of data
    long v;
    //enum<Section>
    int t;
    int i;

    DBG("%5.2f dodata\n", cputime());

    // DATA instructions loop
    for(p = datap; p != P; p = p->link) {
        s = p->from.sym;
        if(s->type == SBSS)
            s->type = SDATA;
        <dodata() sanity check DATA instructions 93b>
    }

    <dodata() if string in text segment 214c>

    orig = 0;

    /*
     * pass 1
     * sanity check data values, and align.
     */
    // symbol table loop
    for(i=0; i<NHASH; i++)
        for(s = hash[i]; s != S; s = s->link) {
            t = s->type;
            if(t == SDATA || t == SBSS) {
                v = s->value; // size of global for now
                <dodata() sanity check GLOBL instructions, size of data v 93a>
                v = rnd(v, 4); // align
                s->value = v; // adjust
            }
        }
}
```

```

        <dodata() in pass 1, if small data size, adjust orig 183c>
    }
}

/*
 * pass 2
 * assign (large) 'data' variables to data segment
 */
for(i=0; i<NHASH; i++)
for(s = hash[i]; s != S; s = s->link) {
    t = s->type;
    if(t == SDATA) {
        v = s->value;
        // s->value used to contain the size of the GLOBL.
        // Now it contains its location (as an offset to INITDAT)
        s->value = orig;
        orig += v;
    } else {
        <dodata() in pass 2, retag small data 183d>
    }
}
orig = rnd(orig, 8);

datsize = orig;

/*
 * pass 3
 * everything else to bss segment
 */
for(i=0; i<NHASH; i++)
for(s = hash[i]; s != S; s = s->link) {
    if(s->type == SBSS) {
        v = s->value;
        s->value = orig;
        orig += v;
    }
}
orig = rnd(orig, 8);

bsssize = orig-datsize;

<dodata() define special symbols 95d>
}

```

Uses DBG 279, NHASH, P 36c, S 33b, SBSS 170b, SDATA, bsssize 92b, datap 36e, datsize 92a, and rnd() 236a.

Note that 51 is just laying out globals here. 51 uses mostly the GLOBL declarations which are stored in the symbol table hash^{31a}. The use of the DATA instructions and datap^{36e} to generate the data section in the executable is done in datblk()^{49b}, not in dodata().

```

<dodata() sanity check GLOBL instructions, size of data v 93a>≡ (92c)
    if(v == 0) { // check
        diag("%s: no size", s->name);
        v = 1;
    }
}

```

Uses diag() 229d.

```

<dodata() sanity check DATA instructions 93b>≡ (92c)
    if(s->type != SDATA)
        diag("initialize non-data (%d): %s\nnP",
            s->type, s->name, p);
}

```

```

v = p->from.offset + p->reg;
if(v > s->value)
    diag("initialize bounds (%ld): %s\nnP",
        s->value, s->name, p);

```

Uses SDATA and `diag()` [229d](#).

Remember that `Instr.regX` is (ab)used to store the size of the DATA slice, e.g., 4 in `DATA foo+8(SB)/4`, "hey!".

7.5 Laying out code: `dotext()`

You are now ready to see the last function of the resolving phase: `dotext()` [94b](#). As explained before, the goal of `dotext()` is to layout code by essentially modifying the `Instr.pcX` of all code instructions. This field was originally storing a virtual program counter and will contain, after `dotext()`, a real program counter. `dotext()` also computes the important global below which stores the size of the text section:

```

<global textsize 94a>≡ (283b)
long textsize;

```

This global is also used to generate the `a.out` header in `asmout()` [108b](#), as shown in Section [4.4.1](#). `dotext()` also computes `INITDAT` [40g](#) which will be used to initialize R12 at run-time.

The code of `dotext()` is pretty straightforward once all the motivations and the big picture have been presented (see Section [7.1.1](#)):

```

<function span(arm) 94b>≡ (286a)
// main -> <>
void
dotext(void)
{
    Prog *p;
    Optab *o;
    // real code address
    long c; // ulong?
    // size of instruction
    int m;
    <dotext() other locals 95a>

    DBG("%5.2f span\n", cputime());

    c = INITTEXT;
    <dotext() initialisation 95b>

    for(p = firstp; p != P; p = p->link) {
        <adjust curtext when iterate over instructions p 36h>
        <adjust autosize when iterate over instructions p 66b>

        // real program counter transition
        p->pc = c;

        o = olookup(p);
        m = o->size;
        c += m;

        if(m == 0) {
            if(p->as == ATEXT) {
                if(p->from.sym != S)
                    p->from.sym->value = c;
            }
        }
    }
}

```

```

        <dotext() detect if large procedure 95c>
    } else {
        diag("zero-width instruction\n%P", p);
    }
} else {
    <dotext() pool handling for optab o 141d>
}
}

```

<dotext() if string in text segment 214e>

```

c = rnd(c, 8);

textsize = c - INITTEXT;
<dotext() refine special symbols 97c>
if(INITRND)
    INITDAT = rnd(c, INITRND);
DBG("tsize = %lux\n", textsize);
}

```

Uses DBG 279, INITDAT 40g, INITRND 40f, INITTEXT 40e, P 36c, S 33b, diag() 229d, firstp 36a, oplook() 102c, rnd() 236a, and textsize 94a.

dotext() relies on modifications done by the previous functions of the resolving phase:

- noops() ⁸⁷, so oplook() ^{102c} does not have to deal with virtual instructions (except MOVW)
- patch() ⁸², which is needed for noops() to work
- dodata() ^{92c}, so all globals have an assigned address (as an offset to R12) and so oplook() can predict the size of instructions involving globals

Note again that 5l is just laying out code here. dotext() does not generate code. The generation of the code section of the executable is done in asmb() ^{45a} and asmout(), not in dotext().

```

<dotext() other locals 95a>≡ (94b) 214d▷
long    otxt;

```

```

<dotext() initialisation 95b>≡ (94b)
otxt = c;

```

```

<dotext() detect if large procedure 95c>≡ (94b)
if(c - otxt >= 1L<<17) {
    diag("Procedure %s too large\n", TNAME);
    errexit();
}
otxt = c;

```

Uses TNAME 279, diag() 229d, and errexit() 229b.

7.6 Defining special symbols: etext, edata, and end

The linker defines a few *special symbols* which allow a form of *reflection* on the executable structure and its sections. Those symbols and their meanings are listed in Figure 7.4 which describes the memory layout of the hello executable you saw in Section 2.3.

Those symbols are introduced by 5l with xdefine():

```

<dodata() define special symbols 95d>≡ (92c) 97b▷
xdefine("bdata", SDATA, 0L);
xdefine("edata", SDATA, datsize);
xdefine("end", SBSS, datsize+bsssize);

```

Uses SBSS 170b, SDATA, bsssize 92b, datsize 92a, and xdefine() 97a.

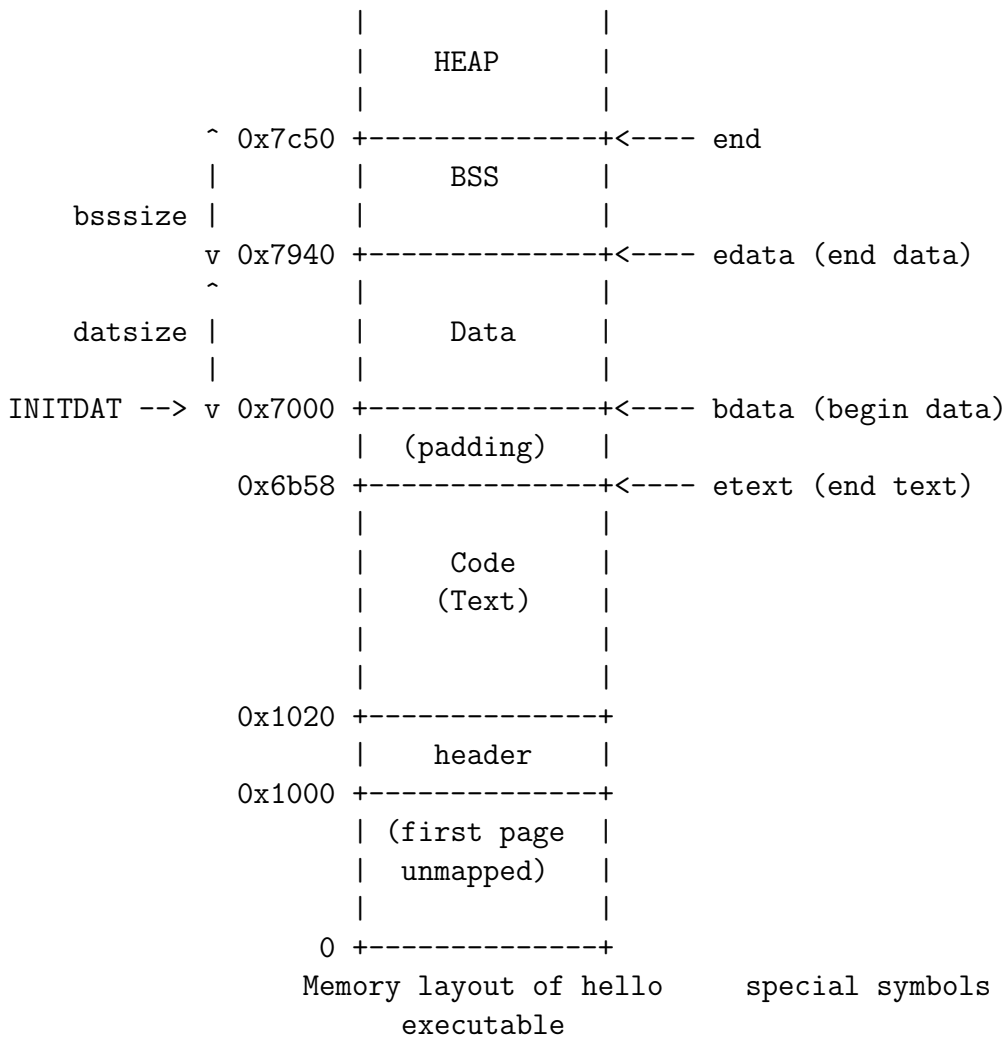


Figure 7.4: Memory layout of `hello` and special symbols.

```

⟨function xdefine(arm) 97a)≡ (286a)
void
xdefine(char *p, int t, long v)
{
    Sym *s;

    s = lookup(p, 0);
    if(s->type == SNONE || s->type == SXREF) {
        s->type = t;
        s->value = v;
    }
}

```

Uses SNONE 33b, SXREF, and lookup() 32a.

Note that the `Sym.valueX` for the `bdata` symbol is 0, not `INITDAT`^{40g}. Indeed, just like for globals, special data symbols have their addresses represented as an offset to the start of the data section. Moreover, `INITDAT` has not yet been computed in `dodata()`^{92c}. But, it is important to layout those special symbols in `dodata()`, before `dotext()`^{94b}, just like globals, as they may also be referenced in instructions.

Special symbols are similar to globals but they do not take any space in the data or BSS section. In fact, the first global of the data section will have the same address than `bdata`. They are used only for their addresses in instructions such as `MOVW $etext(SB), R1`. The Plan 9 kernel uses the addresses of those symbols to move itself in memory during the boot procedure (see the `KERNEL` book [Pad14]). The `sbrk()` system call, which is used by `malloc()`, uses `end` to know where the heap starts (see the `LIBCORE` book [Pad16b]).

`etext` is also initialized in `dodata()` so it can be referenced in instructions:

```

⟨dodata() define special symbols 97b) +≡ (92c) <95d 182d>
    xdefine("etext", STEXT, 0L);

```

Uses STEXT 156d and xdefine() 97a.

But its final value can be set only in `dotext()`:

```

⟨dotext() refine special symbols 97c)≡ (94b)
    lookup("etext", 0)->value = INITTEXT+textsize;

```

Uses INITTEXT 40e, lookup() 32a, and textsize 94a.

Note that for code symbols, `Sym.valueX` contains a real code address, not an offset.

7.7 Mutual recursion in layout and SB/R12

The last special symbol 51 defines is `setR12` which I mentioned before:

```

⟨dodata() define special symbols setR12 IF unoptimized 97d)≡
    xdefine("setR12", SDATA, 0L);

```

It is the last piece of the process I outlined in Section 7.1.2 to solve the mutual dependency issue between the layout of code and data. Note that the `Sym.valueX` of `setR12`, just like for `bdata`, is 0, not `INITDAT`^{40g}. But, The address of `setR12` when used in an instruction, e.g., `MOVW $setR12(SB), R12`, is recognized in a special way by the code generator and will be converted in an instruction which loads its *absolute address*, which will be `INITDAT`. Just like for `etext` and `edata`, the use of the special symbol `setR12` allows a form of reflection which is needed to communicate the value of `INITDAT` to the program.

Section 12.3.2 will present an optimisation around `setR12` which will actually change its value.

Chapter 8

Code Generation Preparation

The final step in the linking pipeline is the machine code generation. You saw already `asmb()`^{45a} in Section 4.4, which generates the `a.out` header and contains some of the logic to generate the code and data sections. The only missing pieces were the functions `oplook()`^{102c} and `asmout()`^{108b}:

```
<asmb() Text section, in iteration over instruction p(repeated code) 98>≡
    Optab* o = oplook(p);
    asmout(p, o);
```

Those two functions work together to generate the ARM instructions and will be covered in this chapter. But before, I will present additional data structures used by `oplook()` and `asmout()`, especially `Optab`^{99a}.

8.1 The linker as second-stage compiler

The paragraph above calls out something unusual: the final step of Plan 9’s linking pipeline is machine code generation. Most linkers (GNU `ld`, LLVM `lld`, `gold`, `mold`, and every mainstream UNIX linker) do not generate any machine code at all. They read object files whose instructions are already fully assembled by the compiler or assembler, and their job is a layout engine: concatenate sections, assign final addresses, rewrite relocation records, write the executable header, and emit the result. The instructions themselves pass through unchanged—the linker never peeks inside one, never rewrites one, never needs to know what an `ADD` looks like on the target ISA.

`5l` is different because `5a` and `5c` deliberately hand it unfinished work. The object files contain in-memory `Prog` instruction records with symbolic operands (Chapter 5), and `5l` picks actual ARM machine instructions from a rule table (`oplook/asmout`), handles ARM’s constrained rotated-immediate encoding via the literal pool and the `setR12` base-register trick, applies leaf-function optimization once it knows all call sites, inserts stack-frame code at function entries, and runs a second pass of peephole optimization after symbol resolution. `5a`’s output is closer to a “linearized AST of assembly instructions” than to finished machine code; `5l` completes what `5a` started, and the whole compiler/assembler/linker pipeline is effectively one compiler split across three processes.

The modern UNIX world has caught up with this idea under a different name: link-time optimization (LTO). With `gcc -flto` or `clang -flto`, the compiler emits GIMPLE or LLVM bitcode IR into the object file instead of finished machine code, and the linker runs the back end across all translation units, which enables inlining, devirtualization, dead-code elimination, and register allocation that see the whole program. The spirit is identical to what Plan 9 has been doing since the first version—defer code generation to the linker so the linker has more information—but LTO is layered on top of section-oriented ELF and Mach-O object files and separate front- and back-ends, so it needs its own IR format and its own plumbing. Plan 9 just designed the object format around the idea from the start.

8.2 Additional data structures

Most of the data structures presented in this section work together to provide a concise way to express the *rules* of the ARM code generator. A rule is made of a *pattern* specifying the shape of an instruction, and an *action* to be taken if the pattern *matches* an instruction. A single pattern can match a *set* of opcodes and certain *classes* of operands. An action in a rule consists mostly of an *action identifier* which is then used in `asmout()`^{108b} to choose the appropriate instruction generator.

The following sections introduce the data structures and a few concrete examples for the rules, patterns, and operand classes I just mentioned.

8.2.1 Optab and optab

The structure below represents a rule:

```
<struct Optab(arm) 99a>≡ (283a)
struct Optab
{
    // -----
    // The pattern (if)
    // -----

    // enum<Opcode> (the opcode is the representant of a set of opcodes)
    byte    as;

    // enum<Operand_class>, possible operand class for first operand (from)
    short   a1;
    // enum<Operand_class>, possible operand class for middle operand
    short   a2;
    // enum<Operand_class>, possible operand class for third operand (to)
    short   a3;

    // -----
    // The action (then)
    // -----

    // action id for the code generator (see the giant switch in asmout())
    short   type;
    // size of the corresponding machine code (should be a multiple of 4)
    short   size;

    <Optab param field 131b>
    <Optab flag field 126d>
};
```

A very important global is `optab` which contains all the rules of the ARM code generator:

```
<global optab (linkers/5l/optab.c)(arm) 99b>≡ (284)
Optab optab[] =
{
    /* struct Optab:
       OPCODE, from, prog->reg, to,   type,size,param,flag */
    <optab entries 111b>
    { AXXX, C_NONE, C_NONE, C_NONE, 0, 4 },
};
```

Uses `C_NONE` 101a.

I will present elements of this array gradually in this chapter. The last entry uses the `AXXX` opcode which is used as an *end-of-array* marker¹. It is recognized by `buildop()`^{104c} which I will cover later.

¹NULL is often used for this purpose.

Here are examples of entries in `optab`, that is rules:

```
<optab example of entries (repeated) 100a>≡
//      Pattern          Action
{ AADD,  C_REG,  C_REG, C_REG,    1, 4 },
{ AADD,  C_RCON, C_REG, C_REG,    2, 4 },
{ AADD,  C_LCON, C_REG, C_REG,   13, 8 },
```

The first entry says that *if* an instruction has for opcode `AADD` and has 3 registers (`C_REG`^{101a}) for operands, *then* the first entry (the action id 1) in the dispatch code of `asmout()`^{108b} should be used to generate the code, and 4 bytes should be necessary for this code (one ARM instruction). In fact, this first rule will also match instructions using `ASUB`, `AAND`, `AORR`, etc. Indeed, as you will see in Section 8.2.2, `Optab.as` contains the *representant* (here `AADD`) of a set of opcodes.

Up until now, the *class* of an operand (e.g., `C_REG` above), which will be described fully in Section 8.2.3, looks similar to the *kind* of an operand, which I presented quickly in Section 3.3 and fully in the `ASSEMBLER` book [Pad15a]. Indeed, `C_REG` is very similar to the operand kind for registers `D_REG`. For constants though you will start to see a deviation. Indeed, the operand kind `D_CONST` is actually divided in multiple operand classes: `C_RCON`^{113a}, `C_LCON`^{113a}, and a few more I will describe in Section 9.3.1. The reason for the division is that depending on the value of the constant, one might need to use different kinds of instructions. For instance, the third entry of `optab` above says that if the first operand is a large constant (`C_LCON`), then 2 ARM instructions will be necessary (8 bytes). If the number is small or can be rotated in a certain way (`C_RCON`), then 1 ARM instruction will be enough, as indicated by the second entry above. Indeed, this small or rotatable number can be encoded directly in the instruction.

The `optab` global will be eventually sorted in `buildop()` with the ordering function `ocmp()`^{105b} which I will describe later. All the rules about `AADD` will then be put next to each other in the array (if they were originally spread). In fact, `ocmp()` orders first by opcodes, but also then by operand classes. The order for those classes is actually important as you will see later.

`Optab` has two optional fields, `Optab.param`^{131b} and `Optab.flag`^{126d} which will be presented later in this chapter.

8.2.2 Oprange

The structure below represents a *range* of entries in the sorted global `optab`^{99b}:

```
<struct Oprange(arm) 100b>≡ (283a)
struct Oprange
{
    //starting index in (sorted) optab
    Optab* start;
    //ending index in (sorted) optab
    Optab* stop;
};
```

The global below associates a range to every opcode:

```
<global oprange(arm) 100c>≡ (289c)
// map<enum<Opcode>, Oprange>
Oprange oprange[ALAST];
```

`oprangle` is also set by `buildop()`^{104c} and contains originally the ranges for the opcodes mentioned in the `optab` array. For instance, the range for `AADD`, given the `optab` entries of the previous section, could be:

```
// range for AADD, optab[0..3[ <=> optab[0..2]
{ .start = &optab[0];
  .stop  = &optab[3];
}
```

This range will be the starting point for the algorithm in `oplook()`^{102c} to find the rule matching an instruction (see the code of `oplook`^{102c}). Indeed, given an instruction `p`, `oplook()` will look first in `oprange[p->as]` to get a range of entries. `oplook()` can then iterate over all those entries, which are rules, and check if the classes of the operands `p->from`, `p->reg`, and `p->to` match the one specified in the rule. The operand matching function is actually not just equality but a subtle function, `cmp()`^{103c}, which I will present later. The first rule matching the instruction will be returned by `oplook()`.

In addition to its role as an *indexing structure* for `oplook()`, `oprange` can also be used to *factorize* rules. Indeed, by doing for instance `oprange[ASUB] = oprange[AADD]`; at the end of `buildop()`, all the rules with `AADD` in `optab` will be tried for instructions with the opcode `ASUB`. So, in effect `AADD` becomes the *representant* of the set of opcodes `AADD`, `ASUB`. Then, one just needs to write rules with `AADD` instead of having to repeat them also for `ASUB`. I will gradually see in this chapter the different opcodes sets and their representants.

8.2.3 Operand_class

You saw already a few elements of the `Operand_class` type: `C_REG`, `C_RCON`, and `C_LCON`. As I mentioned before, the operand classes are similar to the operand kinds. In fact, many are almost identical, e.g., `C_REG` versus `D_REG`:

```
<enum Operand_class(arm) 101a>≡ (283a)
// order of entries for one kind matters! coupling with cmp() and ocmp()
enum Operand_class {
    C_NONE      = 0,

    C_REG,      // D_REG
    C_BRANCH,  // D_BRANCH

    // D_CONST
    <Operand_class C_xCON cases 113a>

    // D_OREG
    <Operand_class C_xOREG cases 114a>
    // D_OREG with symbol N_EXTERN (or N_INTERN)
    <Operand_class C_xEXT cases 115b>
    // D_OREG with symbol N_PARAM or N_LOCAL
    <Operand_class C_xAUTO cases 116c>

    // D_ADDR
    <Operand_class C_xxCON cases 116g>

    <Operand_class cases 121c>
    C_GOK, // must be at the end e.g., for xcmp[] decl, or buildop loops
};
```

Some kinds are also divided in *subclasses*, e.g. `D_CONST` in `C_RCON`^{113a}, `C_LCON`^{113a}, and a few more. I will present those subclasses gradually in this chapter.

The function below converts an operand kind to an operand class. It is used notably by `oplook()`^{102c}:

```
<function aclass(arm) 101b>≡ (289c)
/// oplook | ... -> <>
int
aclass(Adr *a)
{
    <aclass() locals 102b>

    switch(a->type) {
    <aclass() switch type cases 102a>
    }
    return C_GOK;
}
```

Uses C_GOK 101a.

```
<aclass() switch type cases 102a>≡ (101b) 113b▷  
case D_NONE:  
    return C_NONE;  
case D_REG:  
    return C_REG;  
case D_BRANCH:  
    return C_BRANCH;
```

Uses C_BRANCH 101a, C_NONE 101a, and C_REG 101a.

Up until now the code of `aclass()`^{101b} is pretty simple. Later, for operands involving constants (`D_CONST`), offsets (`D_OREG`), or addresses (`D_ADDR`), the code will be more complicated as one operand kind may lead to different subclasses. In fact, `aclass()` assumes the layout of globals has already been done (via `dodata()`^{92c}). That way `aclass()` can know for instance the section and concrete offset of a global. This is necessary to know the subclass of this operand, to know for instance whether the offset is small enough to be encoded directly in the instruction.

```
<aclass() locals 102b>≡ (101b)  
Sym *s;  
// enum<Section>  
int t;
```

8.3 `oplook()` and `buildop()`

You are now ready to understand the code of `oplook()`. `oplook()` assumes the global `optab`^{99b} has been sorted and `orange`^{100c} has been set. Those two operations are performed by `buildop()`^{104c} which I will present after `oplook()`.

8.3.1 `oplook()` and `cmp()`

`oplook` takes an instruction as a parameter and returns a rule (`Optab*`) matching the opcode and operands of the instruction. Its code is pretty simple once the different data structures it is using have been explained and its optimization hidden (for now):

```
<function oplook(arm) 102c>≡ (289c)  
/// main -> (dotext | asmb) -> <>  
Optab*  
oplook(Prog *p)  
{  
    // enum<Opcode> (to index orange[])  
    int r;  
    // enum<Operand_class>  
    int a1, a2, a3;  
    // ref<Optab> (from = optab)  
    Optab *o, *e;  
    <oplook() other locals 106c>  
  
    <oplook() if use opti, part1 107a>  
    else {  
        a1 = aclass(&p->from);  
        a3 = aclass(&p->to);  
    }  
    a2 = (p->reg != R_NONE)? C_REG : C_NONE;  
    r = p->as;
```

```

// find the range of relevant optab entries
o = oprange[r].start;
e = oprange[r].stop;
⟨oplook() sanity check o 103a⟩

⟨oplook() debug 225e⟩

⟨oplook() if use opti, part2 108a⟩
else {
    // iterate over the range
    for(; o<e; o++)
        // compare the operands
        if(o->a2 == a2)
            if(cmp(o->a1, a1))
                if(cmp(o->a3, a3)) {
                    // found a matching rule!
                    return o;
                }
}
⟨oplook() illegal combination error 103b⟩
}

```

Uses C_NONE 101a, C_REG 101a, aclass() 101b, cmp() 103c, and oprange 100c.

If no rule is found a linking error is reported:

```

⟨oplook() sanity check o 103a⟩≡ (102c)
if(o == nil) {
    o = oprange[r].stop; /* just generate an error */
}

```

Uses oprange 100c.

```

⟨oplook() illegal combination error 103b⟩≡ (102c)
diag("illegal combination %A %d %d %d", p->as, a1, a2, a3);
prasm(p);
if(o == nil)
    o = optab;
return o;

```

Uses diag() 229d, optab 99b, and prasm() 219b.

The matching of operand classes in oplook() ^{102c} is using simply == for the middle operand (a2). Indeed, this operand class is always either a register (C_REG^{101a}) or nothing (C_NONE^{101a}). But, for the other operands the function cmp() (for compatible) below is used:

```

⟨function cmp(arm) 103c⟩≡ (289c)
bool
cmp(int a, int b)
{
    if(a == b)
        return true;

    switch(a) {
        ⟨cmp() switch on a, the operand class in optab rule, cases 104a⟩
    }
    return false;
}

```

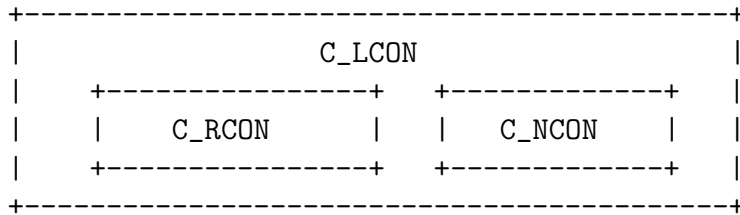
cmp() ^{103c} is used as cmp(o->a1, a1) in oplook(), so the parameter a corresponds to the class of the operand in one of the rule of optab^{99b} (o->a1), and the parameter b the actual class of the operand of the instruction (a1). Up until now cmp() is similar to ==. But, for certain classes, cmp() can also express inclusion between

classes and a notion of subclasses. For instance, if the operand class of a rule is `C_LCON`^{113a}, then it will also match the actual operand classes `C_RCON`^{113a} and `C_NCON`^{113a} of the instruction thanks to this code:

```
<cmp() switch on a, the operand class in optab rule, cases 104a>≡ (103c) 114d▷
case C_LCON:
    if(b == C_RCON || b == C_NCON)
        return true;
    break;
```

Uses `C_LCON` 113a, `C_NCON` 113a, and `C_RCON` 113a.

Indeed, a large constant (`C_LCON`) is more general than a rotatable constant (`C_RCON`) or a negative rotatable constant (`C_NCON`). Visually, the relationship is a nested set:



Thanks to this encoding of inclusion via cases of `cmp()`, a single rule can match many different kinds of instructions. Just like `orange`^{100c} you saw before allows with one opcode to actually match a set of opcodes, `cmp()` allows with one *general* operand class (e.g., `C_LCON`) to actually match also more *specialized* operand classes (e.g., `C_RCON` and `C_NCON`). I will present those general and specialized classes, and their encoding in `cmp()`, gradually in this chapter.

8.3.2 buildop() and ocmp()

`buildop()` is pretty simple. It sorts `optab`^{99b} and then sets `orange`^{100c} which is used by `oplook()`^{102c}:

```
<main() initialize globals(arm) 104b>+≡ (38e) <51d 109c▷
buildop();
```

```
<function buildop(arm) 104c>≡ (289c)
// main -> <>
void
buildop(void)
{
    int i, n;
    // enum<Opcode> representing a range
    int r;

    <buildop() initialize xcmp cache 107e>
    <buildop() initialize flags 191g>

    for(n=0; optab[n].as != AXXX; n++) {
        <buildop() adjust optab if flags, remove certain rules 191h>
    }
    // n contains now the size of optab

    qsort(optab, n, sizeof(optab[0]), ocmp);

    for(i=0; i<n; i++) {
        r = optab[i].as;

        // initialize orange for the representants
        orange[r].start = optab+i;
        while(optab[i].as == r)
```

```

        i++;
oprangle[r].stop = optab+i;
i--; // compensate the i++ of the for

// setup opcode sets of representants
switch(r)
{
<buildop() switch opcode r for ranges cases 105a>
default:
    diag("unknown op in build: %A", r);
    errexit();
}
}
}

```

Uses `diag()` 229d, `errexit()` 229b, `ocmp()` 105b, `oprangle` 100c, and `optab` 99b.

The cases of the `switch` above, which I will present gradually in this chapter, specify the opcodes sets of the representants.

```

<buildop() switch opcode r for ranges cases 105a>≡ (104c) 112a▷
case AXXX:
    break;

```

The ordering of rules in the sorted `optab` is governed by the function below:

```

<function ocmp(arm) 105b>≡ (289c)
/// main -> buildop -> qsort(..., <>)
int
ocmp(const void *a, const void *b)
{
    Optab *p1, *p2;
    int n;

    p1 = (Optab*)a;
    p2 = (Optab*)b;

    n = p1->as - p2->as;
    if(n)
        return n;
    <ocmp() if v4 flag on p1 or p2 208g>
    <ocmp() if floating point flag on p1 or p2 191e>
    n = p1->a1 - p2->a1;
    if(n)
        return n;
    n = p1->a2 - p2->a2;
    if(n)
        return n;
    n = p1->a3 - p2->a3;

    if(n)
        return n;
    return 0;
}

```

`ocmp()`^{105b} orders first by opcode (opcodes with lower enumeration values in `Opcode` are first), then by the class of the first operand, then second, and finally third operand. Note that the order of the enumerations cases in `Operand_class`^{101a} matters. Indeed, if `C_RCON`^{113a} was after `C_LCON`^{113a}, as in:

```

enum Operand_class {
    ...
    C_LCON,

```

```

...
C_RCON
...
};

```

the enumeration value of C_RCON would be greater than C_LCON. Then entries in optab will be sorted as follows:

```

// sorted optab
optab[] = {
...
{ AADD, C_LCON, C_REG, C_REG,      13, 8 },
{ AADD, C_RCON, C_REG, C_REG,      2, 4 },
...
};

```

But, C_LCON is considered more general than C_RCON according to `cmp()`^{103c}. That means that an arithmetic instruction with a small or rotatable constant as a first operand will already match the rule with C_LCON in `oplook()` which is unfortunate. Indeed, I would rather have it match the rule with C_RCON which is more specialized. This is why the order of enumerations in `Operand_class` matters. C_RCON must be before C_LCON as in:

```

enum Operand_class {
...
C_RCON,
C_LCON,
...
};

```

The enumerations of general operand classes must be after the enumerations of their subclasses.

8.3.3 Optimizations

There are a few optimizations which improve `oplook()`^{102c}, `aclass()`^{101b}, and `cmp()`^{103c}, which I will cover now.

Caching

The first set of optimizations involves *caching* (a.k.a *memoizing*). `oplook()`^{102c} is called once from `dotext()`^{94b} and another time from `asmb()`^{45a} so it makes sense to cache the result of the first call to `oplook()`, that is the rule matching an instruction, in the instruction itself:

```

<Prog other fields 106a>+≡ (34b) <83d
// option<index in optab[] (1-based)>, 0 means None, i means optab[i-1]
byte    optab;

```

In the same way, `aclass()`^{101b} is called from `oplook()` but also from `asmout()`^{108b}, so 5l can also cache the result of `aclass()` in the operand itself:

```

<Adr other fields 106b>+≡ (33c) <34a 150c>
// option<enum<Operand_class>>, 0 means None, i means i-1 is the class you want
short   class;

```

Those caching fields are used and set in `oplook()`:

```

<oplook() other locals 106c>≡ (102c) 107f>
bool useopti = true;
int i;

```

```

⟨oplook() if use opti, part1 107a)≡ (102c)
    if(useopti) {
        i = p->optab;
        if(i)
            return optab+(i-1);

        a1 = p->from.class;
        if(a1 == 0) {
            a1 = aclass(&p->from) + 1;
            p->from.class = a1;
        }
        a1--;

        a3 = p->to.class;
        if(a3 == 0) {
            a3 = aclass(&p->to) + 1;
            p->to.class = a3;
        }
        a3--;
    }

```

Uses `aclass()` 101b and `optab` 99b.

```

⟨oplook() if use opti, in part2, memoize matching rule o 107b)≡ (108a)
    p->optab = (o-optab)+1;

```

Uses `optab` 99b.

Note that if one transforms an instruction `p` after the result of `oplook()` or `aclass()` has been cached in `p` and its operands, one should reset this cache with `nocache()` below. That way it will force the next call to `oplook()` and `aclass()` to recompute things:

```

⟨function nocache(arm) 107c)≡ (291b)
    void
    nocache(Prog *p)
    {
        p->optab = 0;
        p->from.class = 0;
        p->to.class = 0;
    }

```

Precomputing

Another optimization *precomputes* the result of `cmp()`^{103c} on any pair of operand classes:

```

⟨global xcmp(arm) 107d)≡ (289c)
    bool xcmp[C_GOK+1][C_GOK+1];

```

Uses `C_GOK` 101a.

```

⟨buildop() initialize xcmp cache 107e)≡ (104c)
    for(i=0; i<C_GOK; i++)
        for(n=0; n<C_GOK; n++)
            xcmp[i][n] = cmp(n, i);

```

Uses `C_GOK` 101a, `cmp()` 103c, and `xcmp` 107d.

This in turns will improve `oplook()`^{102c}.

```

⟨oplook() other locals 107f)+≡ (102c) <106c
    bool *c1, *c3;

```

```

⟨oplook() if use opti, part2 108a⟩≡ (102c)
    if(useopti) {
        c1 = xcmp[a1];
        c3 = xcmp[a3];
        for(; o<e; o++)
            if(o->a2 == a2)
                if(c1[o->a1])
                    if(c3[o->a3]) {
                        ⟨oplook() if use opti, in part2, memoize matching rule o 107b⟩
                        return o;
                    }
            }
    }

```

Uses xcmp 107d.

8.4 asmout()

asmout() is the heart of the ARM code generator. Given an instruction *p* and a rule *o* found via oplook() ^{102c}, asmout() will dispatch the action identifier in *o->type* in a big switch which generates the ARM code:

```

⟨function asmout(arm) 108b⟩≡ (288)
    /// main -> asmb -> for p { <> }
    void
    asmout(Prog *p, Optab *o)
    {
        // ARM 32 bits instructions, set in the switch
        long o1, o2, o3, o4, o5, o6;
        ⟨asmout() other locals 111a⟩

        o1 = o2 = o3 = o4 = o5 = o6 = 0;

        // first switch, action id dispatch, set o1, o2, ...
        switch(o->type) {
            ⟨asmout() switch on type cases 112b⟩
            default:
                diag("unknown asm %d", o->type);
                prasm(p);
                break;
        }

        ⟨asmout() debug 226b⟩

        // second switch, output o1, o2, ...
        switch(o->size) {
            ⟨asmout() switch on size cases 108c⟩
        }
    }

```

Uses diag() 229d and prasm() 219b.

Note that one instruction *p* in the object file may lead to the generation of up to 6 ARM instructions (24 bytes) in the executable, hence the use of local variables up to *o6* above. Those local variables are set in the cases of the first switch (cases you will see gradually in this chapter) and passed later to lput1() ^{109d} in cases of the second switch:

```

⟨asmout() switch on size cases 108c⟩≡ (108b)
    case 4:
        ⟨asmout() when 1 generated instruction, debug 226c⟩
        lput1(o1);
        break;

```

```

case 8:
    ⟨asmout() when 2 generated instructions, debug 226d⟩
    lputl(o1);
    lputl(o2);
    break;
case 12:
    ⟨asmout() when 3 generated instructions, debug 226e⟩
    lputl(o1);
    lputl(o2);
    lputl(o3);
    break;
case 16:
    ⟨asmout() when 4 generated instructions, debug 226f⟩
    lputl(o1);
    lputl(o2);
    lputl(o3);
    lputl(o4);
    break;
case 20:
    ⟨asmout() when 5 generated instructions, debug 226g⟩
    lputl(o1);
    lputl(o2);
    lputl(o3);
    lputl(o4);
    lputl(o5);
    break;
case 24:
    ⟨asmout() when 6 generated instructions, debug 226h⟩
    lputl(o1);
    lputl(o2);
    lputl(o3);
    lputl(o4);
    lputl(o5);
    lputl(o6);
    break;
default:
    ⟨asmout() when other size, debug 226i⟩
    break;

```

Uses `lputl()` 109d.

`lputl()` is one of the function from the *input/output buffer management* library described in Appendix D.2. The most important thing for this chapter is that `lputl()` fills an array of bytes in `Buf.obuf`, the *output buffer*, through a global `cbp` setup in `main()`^{246j}:

```

⟨global cbp 109a⟩≡ (289a)
// array<byte> (slice of buf.obuf)
char* cbp;

```

```

⟨global cbc 109b⟩≡ (289a)
// remaining bytes in buf.obuf
int cbc;

```

```

⟨main() initialize globals(arm) 109c⟩+≡ (38e) <104b
cbp = buf.obuf;
cbc = sizeof(buf.obuf);

```

Then, this buffer gets *flushed* once in a while in `cflush()`^{110a}, which finally outputs the data in the executable via the global `cout`^{38d}:

```

⟨function lputl(arm) 109d⟩≡ (289a)
void

```

```

lputl(long l)
{
    cbp[3] = 1>>24;
    cbp[2] = 1>>16;
    cbp[1] = 1>>8;
    cbp[0] = 1;
    cbp += 4;
    cbc -= 4;
    if(cbc <= 0)
        cflush();
}

```

Uses cbc 109b, cbp 109a, and cflush() 110a.

```

⟨function cflush 110a⟩≡ (289a)
void
cflush(void)
{
    int n;

    n = sizeof(buf.obuf) - cbc;
    if(n)
        write(cout, buf.obuf, n);

    cbp = buf.obuf;
    cbc = sizeof(buf.obuf);
}

```

Uses buf 234b, cbc 109b, cbp 109a, and cout 38d.

Note that `lputl()` means “long put little-endian”. The lowest byte of a number have the lowest memory address. Indeed the ARM uses a little-endian architecture. There is another function `lput()`^{110b}, which does a similar thing but for big-endian architecture, where the lower bits of a number have the higher address (as in the x86):

```

⟨function lput(arm) 110b⟩≡ (289a)
void
lput(long l)
{
    cbp[0] = 1>>24;
    cbp[1] = 1>>16;
    cbp[2] = 1>>8;
    cbp[3] = 1;
    cbp += 4;
    cbc -= 4;
    if(cbc <= 0)
        cflush();
}

```

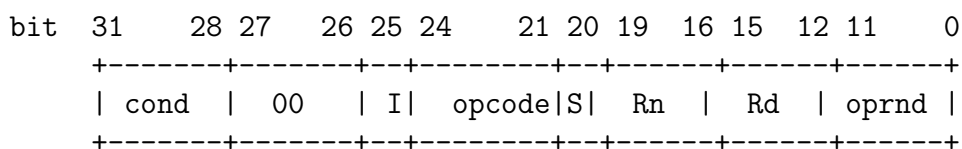
Uses cbc 109b, cbp 109a, and cflush() 110a.

In fact, this function is actually used for the `a.out` generation in Section 4.4.1. Indeed the `a.out` format imposes a big-endian order for the numbers in the header.

Chapter 9

ARM Machine Code Generation

The general shape of an ARM data-processing instruction, which most of the cases in `asmout()`^{108b} are filling in, is a single 32-bit word with the following layout:



`cond` is one of 16 condition codes (always 1110 = AL for unconditional), `I` selects between register and rotated-immediate operand, `opcode` picks ADD/SUB/AND/EOR/etc, `S` enables condition-flag updates, and `Rn/Rd` are the 4-bit register fields. Almost every line in the upcoming `asmout()` cases is ORing one of these subfields into `o1`.

The cases of the first `switch` in `asmout()` will rely on a set of temporary variables to store registers and symbols:

```
<asmout() other locals 111a>≡ (108b) ??▷
int rf; // register ‘from’
int rt; // register ‘to’
int r; // middle register
Sym *s;
```

The rest of this chapter will present gradually the different cases of the first `switch` of `asmout()`. In fact, each of those cases is associated with a set of entries in `optab`^{99b} which mention the action identifier representing the case. This is why I will present also gradually those entries. In the same way the set of opcodes in `oprange`^{100c} will be gradually explained in the following sections next to the entries of `optab` and the cases of `asmout()`. All those elements work together to express the ARM code generation rules of 51.

9.1 ARM instruction format

9.2 Pseudo opcodes

The first code generation rules you will see concern pseudo-opcodes.

9.2.1 ATEXT

ATEXT is kept in the list of code instructions only for `curtext`^{36f}, so that error messages can mention in which procedure the problematic instruction is. No code is generated for ATEXT:

```
<optab entries 111b>≡ (99b) 112c▷
{ ATEXT, C_LEXT, C_NONE, C_LCON, 0, 0 },
```

```
{ ATEXT, C_LTEXT, C_REG, C_LCON, 0, 0 },
```

Uses C_LCON 113a, C_LTEXT 115b, C_NONE 101a, and C_REG 101a.

```
<buildop() switch opcode r for ranges cases 112a>+≡ (104c) <105a 112d>
  case ATEXT:
    break;
```

```
<asmout() switch on type cases 112b>≡ (108b) 112e>
  case 0: /* pseudo ops */
    break;
```

9.2.2 AWORD

WORD is a directive which provides an adhoc way to write a word (hence the name) in the code section. It is often used to overcome some limitations of 5a which does not handle every possible opcodes, e.g. opcodes to call a coprocessor (see the ASSEMBLER book [Pad15a]). It is also used to represent large literal constants in the code section, which is very useful to support the encoding of certain instructions as you will see in Section 9.8.

```
<optab entries 112c>+≡ (99b) <111b 119a>
  { AWORD, C_NONE, C_NONE, C_LCON, 11, 4 },
Uses C_LCON 113a and C_NONE 101a.
```

Remember that C_LCON^{113a} is a general operand class and so the single rule above will also match instructions with operands having (sub)classes such as C_RCON^{113a} or C_NCON^{113a}.

```
<buildop() switch opcode r for ranges cases 112d>+≡ (104c) <112a 118c>
  case AWORD:
    break;
```

```
<asmout() switch on type cases 112e>+≡ (108b) <112b 119b>
  case 11: /* word */
    c = aclass(&p->to);
    <asmout() in AWORD case, when dlm 180e>
    o1 = instoffset;
    break;
```

Uses aclass() 101b and instoffset 112f.

In addition to returning the class of an operand, aclass()^{101b} has also for side effect to modify the global instoffset^{112f}, as you will see in the next section. This global contains the immediate constant or offset in the operand. So, o1 above will contain p->to.offset. Remember that the second switch of asmout()^{108b} will eventually output the content of the local variable o1 in the executable.

9.3 Operand subclasses and instoffset

Before seeing the ARM code generation rules for arithmetic instructions, memory instructions, and more, I will present in this section the remaining operand classes and subclasses used in 51. Those classes are referenced in many of the code generation rules. I will also finish to present the code of aclass()^{101b}. As said before, aclass() also modifies the following global, which is then used extensively in asmout()^{108b}:

```
<global instoffset(arm) 112f>≡ (283b)
  long instoffset;
```

9.3.1 D_CONST, C_xCON, and immrot()

I mentioned before the general class C_LCON (large constant) and its subclasses C_RCON (rotatable small constant) and C_NCON (rotatable small negative constant):

```
<Operand_class C_xCON cases 113a>≡ (101a)
C_RCON,    /* 0xff rotated */ // [0..0xff] range, possibly rotated
C_NCON,    /* ~RCON */
C_LCON,
```

Remember from Section 8.3.2 that the order of those enumeration matters, and that the general class C_LCON must be after its subclasses C_RCON and C_NCON.

All those classes derive from the operand kind D_CONST:

```
<aclass() switch type cases 113b>+≡ (101b) <102a 114b>
case D_CONST:
    instoffset = a->offset;
    if(a->reg != R_NONE) // when??????
        goto aconsize;

    if(immrot(instoffset))
        return C_RCON;
    if(immrot(~instoffset))
        return C_NCON;
    return C_LCON;
```

Uses C_LCON 113a, C_NCON 113a, C_RCON 113a, immrot() 113c, and instoffset 112f.

I can now define what is a rotatable small number and see the code of immrot() ^{113c}. The ARM use 12 bits to encode immediate constants in arithmetic instructions but it is using a clever trick to be able to represent numbers larger than 4096. Indeed, the 12 bits are divided in two parts: 8 bits to represent a number from 0 to 0xff, and 4 bits to represent a rotation of this number. The 16 possible rotations are not simply a bitshift to the left from 0 to 15. Indeed, this would not be enough to represent large numbers such as 2^{31} . Instead, each rotation is a double bit rotation to the right. Here are a few examples of numbers and their ARM encoding:

```
0    => rotation = 0b0000; number = 0b000000000
255 => rotation = 0b0000; number = 0b111111111
256 => rotation = 0b1100; number = 0b000000001
512 => rotation = 0b1100; number = 0b000000010
2^31 => rotation = 0b0001; number = 0b000000010
```

Not all 32 bits integers can be represented using that scheme but many important numbers like all the powers of 2 between 0 and 31 can be expressed which is very useful in operations involving bitsets. See <http://alisdair.mcdiarmid.org/arm-immediate-value-encoding/> for an excellent tutorial on the topic. This page contains also an interactive application where you can enter any number and get its encoding (when the number is rotatable).

I can now show the code of immrot():

```
<function immrot(arm) 113c>≡ (289c)
// ulong -> option<long> (None = 0)
long
immrot(ulong v)
{
    // the rotation
    int i;

    for(i=0; i<16; i++) {
        if((v & ~0xff) == 0)
            //      i,r,r opcodes      rotation      number
```

```

        return (1<<25) | (i<<8) | v ;
// inverse of 2 bits rotation to the right
v = (v<<2) |
    (v>>30);
}
return 0;
}

```

`immrot(255)` will return when `i == 0` since `255 & 255 == 0`. For 2^31 , the first iteration will not return as this number is not between 0 and `0xff`. So, `v` will be rotated to the left 2 times, and the 2 upper bits will be put back to the beginning by bitshifting the number to the right by 30. Now `i==1` and `v==2` and `immrot()` will return since `v` is now between 0 and `0xff`.

In addition to returning the ARM encoding of the rotatable constant, `immrot()` also sets the special bit 25 in the return value. This bit is set for all instructions involving an immediate constant, as shown in the ARM reference card. This will allow later code such as:

```
o1 |= immrot(instoffset);
```

9.3.2 D_OREG, C_xOREG, and immaddr()

```

<Operand_class C_xOREG cases 114a>≡ (101a)
<Operand_class cases, in C_xOREG, half case 209e>
<Operand_class cases, in C_xOREG, float cases 191b>
C_SOREG,
C_LOREG,

C_ROREG,
C_SROREG, /* both S and R */

```

```

<aclass() switch type cases 114b>+≡ (101b) <113b 117a>
case D_OREG:
    switch(a->symkind) {
    <aclass() D_OREG case, switch symkind cases 114c>
    }
    return C_GOK;

```

Uses `C_GOK` 101a.

```

<aclass() D_OREG case, switch symkind cases 114c>≡ (114b) 115c>
case N_NONE:
    instoffset = a->offset;
    t = immaddr(instoffset);
    if(t) {
    <aclass() if immfloat for N_NONE symbol 191c>
    <aclass() if immhalf for N_NONE symbol 209f>
    if(immrot(instoffset))
        return C_SROREG;
    return C_SOREG;
    }
    if(immrot(instoffset))
        return C_ROREG;
    return C_LOREG;

```

Uses `C_LOREG` 114a, `C_ROREG` 114a, `C_SOREG` 114a, `C_SROREG` 114a, `immaddr()` 115a, `immrot()` 113c, and `instoffset` 112f.

```

<cmp() switch on a, the operand class in optab rule, cases 114d>+≡ (103c) <104a 115d>
case C_SROREG:
    return cmp(C_SOREG, b) || cmp(C_ROREG, b);
case C_SOREG:
case C_ROREG:

```

```

    return b == C_SROREG || cmp(C_HFOREG, b);
case C_LOREG:
    return cmp(C_SROREG, b);

```

Uses C_HFOREG 191b, C_LOREG 114a, C_ROREG 114a, C_SOREG 114a, C_SROREG 114a, and cmp() 103c.

```

⟨function immaddr(arm) 115a⟩≡ (289c)
// ulong -> option<long> (None = 0)
long
immaddr(long v)
{
    if(v >= 0 && v <= 0xffff)
        return (v & 0xffff) |
            (1<<24) | /* pre indexing */
            (1<<23); /* up */

    if(v < 0 && v >= -0xffff)
        return (-v & 0xffff) |
            (1<<24); /* pre indexing */
    return 0;
}

```

9.3.3 D_OREG with globals, C_xEXT

```

⟨Operand_class C_xEXT cases 115b⟩≡ (101a)
⟨Operand_class cases, in C_xEXT, half case 208h⟩
⟨Operand_class cases, in C_xEXT, float cases 190d⟩
C_SEXT,
C_LEXT,

```

```

⟨aclass() D_OREG case, switch symkind cases 115c⟩+≡ (114b) <114c 116e>
case N_EXTERN:
case N_INTERN:
    ⟨aclass() D_OREG case, N_EXTERN case, sanity check a 116a⟩
    s = a->sym;
    t = s->type;
    ⟨aclass() D_OREG case, N_EXTERN case, sanity check t 116b⟩
    ⟨aclass() when D_OREG and external symbol and dlm 180b⟩
    instoffset = s->value + a->offset - BIG;
    t = immaddr(instoffset);
    if(t) {
        ⟨aclass() if immfloat for N_EXTERN symbol 190e⟩
        ⟨aclass() if immhalf for N_EXTERN symbol 208i⟩
        return C_SEXT;
    }
    return C_LEXT;

```

Uses BIG, C_LEXT 115b, C_SEXT 115b, immaddr() 115a, and instoffset 112f.

```

⟨cmp() switch on a, the operand class in optab rule, cases 115d⟩+≡ (103c) <114d 116d>
case C_SEXT:
    return cmp(C_HFEXT, b);
case C_LEXT:
    return cmp(C_SEXT, b);

```

Uses C_HFEXT 190d, C_LEXT 115b, C_SEXT 115b, and cmp() 103c.

```

<aclass() D_OREG case, N_EXTERN case, sanity check a 116a)≡ (115c)
  if(a->sym == nil || a->sym->name == nil) {
    print("null sym external\n");
    print("%D\n", a);
    return C_GOK;
  }

```

Uses C_GOK 101a.

```

<aclass() D_OREG case, N_EXTERN case, sanity check t 116b)≡ (115c)
  if(t == SNONE || t == SXREF) {
    diag("undefined external: %s in %s", s->name, TNAME);
    s->type = SDATA;
  }

```

Uses SDATA, SNONE 33b, SXREF, TNAME 279, and diag() 229d.

9.3.4 D_OREG with automatics, C_xAUTO

```

<Operand_class C_xAUTO cases 116c)≡ (101a)
  <Operand_class cases, in C_xAUTO, half case 209b>
  <Operand_class cases, in C_xAUTO, float cases 190f>
  C_SAUTO, /* -0xfff to 0xfff */
  C_LAUTO,

```

```

<cmp() switch on a, the operand class in optab rule, cases 116d)+≡ (103c) <115d 117e>
  case C_SAUTO:
    return cmp(C_HFAUTO, b);
  case C_LAUTO:
    return cmp(C_SAUTO, b);

```

Uses C_HFAUTO 190f, C_LAUTO 116c, C_SAUTO 116c, and cmp() 103c.

```

<aclass() D_OREG case, switch symkind cases 116e)+≡ (114b) <115c 116f>
  case N_LOCAL:
    instoffset = autosize + a->offset;
    t = immaddr(instoffset);
    if(t){
      <aclass() if immfloat for N_LOCAL or N_PARAM symbol 191a>
      <aclass() if immhalf for N_LOCAL or N_PARAM symbol 209d>
      return C_SAUTO;
    }
    return C_LAUTO;

```

Uses C_LAUTO 116c, C_SAUTO 116c, autosize 65d, immaddr() 115a, and instoffset 112f.

```

<aclass() D_OREG case, switch symkind cases 116f)+≡ (114b) <116e>
  case N_PARAM:
    instoffset = autosize + 4L + a->offset;
    t = immaddr(instoffset);
    if(t){
      <aclass() if immfloat for N_LOCAL or N_PARAM symbol 191a>
      <aclass() if immhalf for N_LOCAL or N_PARAM symbol 209d>
      return C_SAUTO;
    }
    return C_LAUTO;

```

Uses C_LAUTO 116c, C_SAUTO 116c, autosize 65d, immaddr() 115a, and instoffset 112f.

9.3.5 D_ADDR, C_xxCON

```

<Operand_class C_xxCON cases 116g)≡ (101a) 117d>
  C_RECON,

```

```

<aclass() switch type cases 117a)+≡ (101b) <114b 121d>
case D_ADDR:
    switch(a->symkind) {
        <aclass() D_ADDR case, switch symkind cases 117b>
    }
    return C_GOK;

```

Uses C_GOK 101a.

```

<aclass() D_ADDR case, switch symkind cases 117b)≡ (117a) 117f>
case N_EXTERN:
case N_INTERN:
    s = a->sym;
    <aclass() D_ADDR case, N_EXTERN case, sanity check s 117c>
    switch(s->type) {
        case STEXT: case SSTRING: case SUNDEF:
            instoffset = s->value + a->offset;
            return C_LCON; // etext is stable
        case SNONE: case SXREF:
            diag("undefined external: %s in %s", s->name, TNAME);
            s->type = SDATA;
            // Fall through
        case SDATA: case SBSS: case SDATA1:
            <aclass() in D_ADDR case, SDATA case, if dlm 176e>
            instoffset = s->value + a->offset - BIG;
            if(immrot(instoffset) && instoffset != 0) {// VERY IMPORTANT != 0 for setR12 bootstrapping
                return C_RECON;
            } else {
                instoffset = s->value + a->offset + INITDAT;
                return C_LCON;
            }
    }
    diag("unknown section for %s", s->name);
    break;

```

Uses BIG, C_LCON 113a, C_RECON 116g, INITDAT 40g, SBSS 170b, SDATA, SDATA1 279, SNONE 33b, SSTRING 35g, STEXT 156d, SUNDEF 33b, SXREF, TNAME 279, diag() 229d, immrot() 113c, and instoffset 112f.

```

<aclass() D_ADDR case, N_EXTERN case, sanity check s 117c)≡ (117b)
if(s == S) // no warning?
    break;

```

Uses S 33b.

```

<Operand_class C_xxCON cases 117d)+≡ (101a) <116g
C_RACON,
C_LACON,

```

```

<cmp() switch on a, the operand class in optab rule, cases 117e)+≡ (103c) <116d 209a>
case C_LACON:
    if(b == C_RACON)
        return true;
    break;

```

Uses C_LACON 117d and C_RACON 117d.

```

<aclass() D_ADDR case, switch symkind cases 117f)+≡ (117a) <117b 118a>
case N_LOCAL:
    instoffset = autosize + a->offset;
    goto aconsize;

```

Uses autosize 65d and instoffset 112f.

```

<aclass() D_ADDR case, switch symkind cases 118a)+≡ (117a) <117f 118b>
    case N_PARAM:
        instoffset = autosize + a->offset + 4L;
        goto aconsize;

```

Uses autosize 65d and instoffset 112f.

```

<aclass() D_ADDR case, switch symkind cases 118b)+≡ (117a) <118a
    aconsize:
        return immrot(instoffset)? C_RACON : C_LACON;

```

Uses C_LACON 117d, C_RACON 117d, immrot() 113c, and instoffset 112f.

9.4 Arithmetic and logic opcodes

Now that you have seen all the operand classes, I can focus on the opcodes, starting in this section with the code generation rules for the arithmetic and logic opcodes, e.g., AADD, ASUB, ACMP, etc. As explained in Section 8.2.2, AADD is made the representant of many other opcodes in the `optab`^{99b} rules thanks to the following code:

```

<buildop() switch opcode r for ranges cases 118c)+≡ (104c) <112d 118d>
    case AADD:
        oprange[ASUB] = oprange[r];

        oprange[AAND] = oprange[r];
        oprange[AEOR] = oprange[r];
        oprange[AORR] = oprange[r];

        oprange[AADC] = oprange[r];
        oprange[ASBC] = oprange[r];
        oprange[ARSC] = oprange[r];
        oprange[ARSB] = oprange[r];
        oprange[ABIC] = oprange[r];
        break;

```

Uses oprange 100c.

The same is true for ACMP which represents also ATST, ATEQ, and ACMN:

```

<buildop() switch opcode r for ranges cases 118d)+≡ (104c) <118c 118e>
    case ACMP:
        oprange[ATST] = oprange[r];
        oprange[ATEQ] = oprange[r];
        oprange[ACMN] = oprange[r];
        break;

```

Uses oprange 100c.

Even if this section is about arithmetic and logic opcodes, you will also see rules about opcodes associated more traditionally with memory instructions, e.g., AMOVW. The reason is that AMOVW is a very general instruction which can also be used to set registers as in MOVW \$1, R1. This instruction does not involve any memory and is actually converted in an ARM instruction using an arithmetic opcode as you will see soon.

```

<buildop() switch opcode r for ranges cases 118e)+≡ (104c) <118d 122b>
    case AMVN:
        break;

```

The following subsections are organized according to the shape of the operands (the operand classes), more than by the kind of the operation. This is a different organization that the one I chose in the ASSEMBLER book [Pad15a], but it better matches how instructions are organized in the ARM.

9.4.1 Register-only operands

The simplest instructions are probably the one involving only registers, e.g., `ADD R1, R2, R3`:

```
<optab entries 119a>+≡ (99b) <112c 121a>
//      From   Middle To
{ AADD,  C_REG, C_REG, C_REG,    1, 4 },
{ AADD,  C_REG, C_NONE, C_REG,    1, 4 },

{ AMOVW, C_REG, C_NONE, C_REG,    1, 4 },
{ AMVN,  C_REG, C_NONE, C_REG,    1, 4 },
{ ACMP,  C_REG, C_REG,  C_NONE,   1, 4 },
```

Uses `C_NONE` 101a and `C_REG` 101a.

They can be encoded in a single ARM instruction, hence the rule size 4 in the rules above, and the use of only `o1` in the code below:

```
<asmout() switch on type cases 119b>+≡ (108b) <112e 121b>
case 1: /* op R,[R],R */
    o1 = oprrr(p->as, p->scond);
    rf = p->from.reg;
    rt = p->to.reg;
    r = p->reg;
    <asmout() adjust r and rt 119d>
    o1 |= (r<<16) | (rt<<12) | rf;
    break;
```

Uses `oprerr()` 120a.

You will see `oprerr()`^{120a} below, but it essentially sets the *conditional execution*¹ bits (bits 28 to 32) and the bits for the arithmetic opcode (bits 21 to 24) of a partial ARM instruction returned and stored in the local variable `o1` above. Then, the code above extracts from the original instruction `p` the “from” register `rf`, which will end up as bits 0 to 3 in `o1`², the “to” register `rt`, which will end up as bits 12 to 15 in `o1` hence the `rt<<12` above, and the middle register `r`, which will end up as bits 16 to 19 in `o1` hence the `r<<16` above.

Again, I highly recommend to print the excellent colorful ARM reference card <http://re-eject.gbadev.org/files/armref.pdf> to better understand the bit manipulations in this chapter. This card represents visually in a very compact way the ARM instruction format. The only differences with the conventions used in this chapter is that `rf` is called `Rm` in the card, `rt` is called `Rd` (for destination), and `r` is called `Rn`. Moreover the instructions in the card use the Intel syntax with a *right-to-left* assignment syntax. This book, as well as 5a, use the AT&T syntax with a *left-to-right* assignment syntax. So `ADD Rd, Rn, Rm` in the card corresponds to `ADD rf, r, rt` using the conventions of this chapter.

Note that the assembler allows to write `ADD R1, R2` which is transformed by the linker in `ADD R1, R2, R2`, hence this piece of code below which will be repeated in many places in `asmout()`^{108b}:

```
<asmout() adjust r 119c>≡ (199c 197c 125b 124 123c 122d 119d)
if(r == R_NONE) // ADD FROM, TO ==> ADD FROM, TO, TO
    r = rt;

<asmout() adjust r and rt 119d>≡ (122a 121b 119b)
if(p->to.type == D_NONE)
    rt = 0;
if(p->as == AMOVW || p->as == AMVN)
    r = 0;
else
    <asmout() adjust r 119c>
```

¹See the ASSEMBLER book [Pad15a] or EMULATOR book [Pad15b].

²Remember from the ASSEMBLER book [Pad15a] that registers go from `R0` to `R15` so 4 bits are enough for their encoding in an instruction.

As mentioned above, `oprerr()` returns a partial instruction, to be completed later, with the conditional execution bits and opcode bits set according to its two parameters:

```

<function oprerr(arm) 120a>≡ (288)
long
oprerr(int a, int sc)
{
    long o;

    // bits 28 to 32
    o = (sc&C_SCOND) << 28;
    <oprerr() sign bit handling 120b>
    <oprerr() sanity check sc 120c>

    switch(a) {

        // bits 21 to 24 (and sometimes bit 20)
        case AAND: return o | (0x0<<21);
        case AEOR: return o | (0x1<<21);
        case ASUB: return o | (0x2<<21);
        case ARSB: return o | (0x3<<21);
        case AADD: return o | (0x4<<21);
        case AADC: return o | (0x5<<21);
        case ASBC: return o | (0x6<<21);
        case ARSC: return o | (0x7<<21);

        case ATST: return o | (0x8<<21) | (1<<20);
        case ATEQ: return o | (0x9<<21) | (1<<20);
        case ACMP: return o | (0xa<<21) | (1<<20);
        case ACMN: return o | (0xb<<21) | (1<<20);

        case AORR: return o | (0xc<<21);
        case AMOVW: return o | (0xd<<21); // MOV
        case ABIC: return o | (0xe<<21);
        case AMVN: return o | (0xf<<21); // MVN

        // bits 20 to 27
        <oprerr() switch cases 123a>
    }
    diag("bad rrr %d", a);
    prasm(curp);
    return 0;
}

```

Uses `curp` 36g, `diag()` 229d, and `prasm()` 219b.

`oprerr()` occasionally sets bit 20, the *S bit*, if the programmer requested it by using the *special bit S* as in `ADD.S R1, R2, R3`. See the ASSEMBLER book [Pad15a] and EMULATOR book [Pad15b] for more information on special bits and the semantic of the *S bit*.

```

<oprerr() sign bit handling 120b>≡ (120a)
    if(sc & C_SBIT)
        o |= 1 << 20;

```

This bit is always set for the comparison opcodes, as shown in the case for `ATST` and so on in the `switch` above.

```

<oprerr() sanity check sc 120c>≡ (120a)
    if(sc & (C_PBIT|C_WBIT))
        diag(".P/.W on dp instruction");

```

Uses `diag()` 229d.

9.4.2 Small rotatable immediate constant operand

You have seen in Section 9.3.1 the different subclasses of constants, especially `C_RCON`^{113a} for rotatable constants. Those constants can be encoded directly in the ARM instruction, hence the rule size 4 in the rules below:

```
<optab entries 121a>+≡ (99b) <119a 121e>
{ AADD, C_RCON, C_REG, C_REG, 2, 4 },
{ AADD, C_RCON, C_NONE, C_REG, 2, 4 },

{ AMOVW, C_RCON, C_NONE, C_REG, 2, 4 },
{ AMVN, C_RCON, C_NONE, C_REG, 2, 4 },
{ ACMP, C_RCON, C_REG, C_NONE, 2, 4 },
```

Uses `C_NONE` 101a, `C_RCON` 113a, and `C_REG` 101a.

```
<asmout() switch on type cases 121b>+≡ (108b) <119b 122a>
case 2: /* op $I, [R], R */
    aclass(&p->from);
    o1 = oprrr(p->as, p->scond);
    o1 |= immrot(instoffset); // set also bit 25 for Immediate
    rt = p->to.reg;
    r = p->reg;
    <asmout() adjust r and rt 119d>
    o1 |= (r<<16) | (rt<<12);
    break;
```

Uses `aclass()` 101b, `immrot()` 113c, `instoffset` 112f, and `oprrr()` 120a.

Remember from Section 9.3 that the call to `aclass()`^{101b} above has for side effect to set the global `instoffset`^{112f} which can then be passed to `immrot()`^{113c}.

9.4.3 Bitshifted register

In the ASSEMBLER book [Pad15a], the grammar rule for the first operand of arithmetic instructions is called `imsr` which stands for: immediate (constant) or shifted-register or register. You have seen the ARM code generation rules for the register (`C_REG`^{101a}) and immediate constant (`C_RCON`^{113a}) cases in the previous sections. I can now show the rules for the bitshifted register (`C_SHIFT`):

```
<Operand_class cases 121c>≡ (101a) 180a▷
C_SHIFT, // D_SHIFT
```

```
<aclass() switch type cases 121d>+≡ (101b) <117a 190b>
case D_SHIFT:
    return C_SHIFT;
```

Uses `C_SHIFT` 121c.

```
<optab entries 121e>+≡ (99b) <121a 122c>
{ AADD, C_SHIFT, C_REG, C_REG, 3, 4 },
{ AADD, C_SHIFT, C_NONE, C_REG, 3, 4 },

{ AMVN, C_SHIFT, C_NONE, C_REG, 3, 4 },
{ ACMP, C_SHIFT, C_REG, C_NONE, 3, 4 },
```

Uses `C_NONE` 101a, `C_REG` 101a, and `C_SHIFT` 121c.

```

<asmout() switch on type cases 122a)+≡ (108b) <121b 122d>
case 3: /* op R<<[IR],[R],R */
mov:
    aclass(&p->from);
    o1 = oprrr(p->as, p->scond);
    o1 |= p->from.offset; // set in 5a, complex bit layout
    rt = p->to.reg;
    r = p->reg;
    <asmout() adjust r and rt 119d>
    o1 |= (r<<16) | (rt<<12);
    break;

```

Uses `aclass()` 101b and `opr_rrr()` 120a.

A bitshifted register combines a register (a `D_REG`) and an immediate constant (a `D_CONST`) in a single operand. As explained in the ASSEMBLER book [Pad15a], the `Operand.offset` field for bitshifted registers directly encodes the operand in the ARM instruction format: the first 4 bits (bits 0 to 3) are used for the register number (0 to 15), bits 5 and 6 are used to encode the kind of shift, e.g., `0 << 5` for a left shift, and bits 7 to 11 to encode either a small immediate constant or another register. This is why the code above just does `o1 |= p->from.offset` as part of the ARM instruction encoding has already been done by 5a.

9.4.4 Bitshift opcodes

In addition to the use of bitshifted registers, 5a supports the more traditional bitshift instructions `SLL`, `SRL`, and `SRA`:

```

<buildop() switch opcode r for ranges cases 122b)+≡ (104c) <118e 124b>
case ASLL:
    oprange[ASRL] = oprange[r];
    oprange[ASRA] = oprange[r];
    break;

```

Uses `oprange` 100c.

```

<optab entries 122c)+≡ (99b) <121e 123b>
{ ASLL, C_RCON, C_REG, C_REG, 8, 4 },
{ ASLL, C_RCON, C_NONE, C_REG, 8, 4 },

```

Uses `C_NONE` 101a, `C_RCON` 113a, and `C_REG` 101a.

Those instructions are virtual instructions transformed by the linker in an ARM `MOV` with a bitshifted register as shown below:

```

<asmout() switch on type cases 122d)+≡ (108b) <122a 123c>
case 8: /* sll $c,[R],R -> mov (R<<$c),R */
    aclass(&p->from);
    o1 = oprrr(p->as, p->scond);
    rt = p->to.reg;
    r = p->reg;
    <asmout() adjust r 119c>
    o1 |= (rt<<12) | ((instoffset&31) << 7) | r;
    break;

```

Uses `aclass()` 101b, `instoffset` 112f, and `opr_rrr()` 120a.

Note that `MOV` is an ARM opcode with two operands. It is used to set the content of a register. It should not be confused with the virtual instruction `MOVW` of 5a, which is more general, and which can be transformed depending on its operands as the ARM instructions `LDR`, `STR`, or `MOV` when the operands are a constant and a register.

The bits encoding the shift operation are set via `opr_rrr()` ^{120a}:

```
<opr_rrr() switch cases 123a>≡ (120a) 124e>
//          MOV          shift op
case ASLL: return o | (0xd<<21) | (0<<5);
case ASRL: return o | (0xd<<21) | (1<<5);
case ASRA: return o | (0xd<<21) | (2<<5);
```

```
<optab entries 123b>+≡ (99b) <122c 123d>
{ ASLL, C_REG, C_NONE, C_REG, 9, 4 },
{ ASLL, C_REG, C_REG, C_REG, 9, 4 },
```

Uses `C_NONE` 101a and `C_REG` 101a.

```
<asmout() switch on type cases 123c>+≡ (108b) <122d 123e>
case 9: /* sll R,[R],R -> mov (R<<R),R */
    o1 = opr_rrr(p->as, p->scond);
    rf = p->from.reg;
    rt = p->to.reg;
    r = p->reg;
    <asmout() adjust r 119c>
    o1 |= (rt<<12) | (rf<<8) | (1<<4) | r ;
    break;
```

Uses `opr_rrr()` 120a.

9.4.5 Byte and half word extractions

The following instructions allow to extract parts of a register and store the result in another register:

```
<optab entries 123d>+≡ (99b) <123b 124c>
{ AMOVb, C_REG, C_NONE, C_REG, 14, 8 },
{ AMOVbU, C_REG, C_NONE, C_REG, 58, 4 },
{ AMOVh, C_REG, C_NONE, C_REG, 14, 8 },
{ AMOVhU, C_REG, C_NONE, C_REG, 14, 8 },
```

Uses `C_NONE` 101a and `C_REG` 101a.

`MOVb R1, R2` is a virtual instruction transformed in two instructions:

```
SLL $24, R1, R2 // o1
SRA $24, R2, R2 // o2
```

This is why the rule size in the rules above is 8 and the code below uses the variables `o1` and `o2` for the first time:

```
<asmout() switch on type cases 123e>+≡ (108b) <123c 124a>
case 14: /* movb/movbu/movh/movhu R,R */
    o1 = opr_rrr(ASLL, p->scond);

    if(p->as == AMOVbU || p->as == AMOVhU)
        o2 = opr_rrr(ASRL, p->scond);
    else
        o2 = opr_rrr(ASRA, p->scond);

    rf = p->from.reg;
    rt = p->to.reg;
    if(p->as == AMOVb || p->as == AMOVbU) {
        o1 |= (rt<<12) | (24<<7) | rf;
        o2 |= (rt<<12) | (24<<7) | rt;
    } else {
        o1 |= (rt<<12) | (16<<7) | rf;
        o2 |= (rt<<12) | (16<<7) | rt;
```

```

}
break;

```

Uses `oprerr()` 120a.

`MOVBU R1, R2` can be optimized and encoded using only 1 ARM instruction as `0xff` is a rotatable constant. Note that `0xffff` is not a rotatable constant hence the need for two instructions above for `MOVHU`.

```

<asmout() switch on type cases 124a>+≡ (108b) <123e 124d>
case 58: /* movbu R,R -> AND $0xff, R, R */
    o1 = oprerr(AAND, p->scond);
    o1 |= immrot(0xff);
    rt = p->to.reg;
    r = p->from.reg;
    if(p->to.type == D_NONE)
        rt = 0;
    <asmout() adjust r 119c>
    o1 |= (r<<16) | (rt<<12);
    break;

```

Uses `immrot()` 113c and `oprerr()` 120a.

9.4.6 Multiplication opcodes

Multiplication in the ARM uses a different format than the other arithmetic instructions. It supports only registers as operands:

```

<buildop() switch opcode r for ranges cases 124b>+≡ (104c) <122b 127d>
case AMUL:
    oprange[AMULU] = oprange[r];
    break;

```

Uses `oprange` 100c.

```

<optab entries 124c>+≡ (99b) <123d 125a>
{ AMUL, C_REG, C_REG, C_REG, 15, 4 },
{ AMUL, C_REG, C_NONE, C_REG, 15, 4 },

```

Uses `C_NONE` 101a and `C_REG` 101a.

```

<asmout() switch on type cases 124d>+≡ (108b) <124a 125b>
case 15: /* mul r,[r]r */
    o1 = oprerr(p->as, p->scond);
    rf = p->from.reg;
    rt = p->to.reg;
    r = p->reg;
    <asmout() adjust r 119c>
    <asmout() adjust registers when mul 124f>
    o1 |= (rt<<16) | (rf<<8) | r;
    break;

```

Uses `oprerr()` 120a.

```

<oprerr() switch cases 124e>+≡ (120a) <123a 139d>
case AMULU:
case AMUL: return o | (0x0<<21) | (0x9<<4);

```

```

<asmout() adjust registers when mul 124f>≡ (124d)
if(rt == r) {
    r = rf;
    rf = rt;
}

```

Note that the operands of `AMUL`, and in particular its result, are stored in 32 bits registers. Because the multiplication of two 32 bits numbers can easily overflow, Section 12.6.3 presents other ARM multiplication opcodes where the result is contained in 2 registers, thus supporting a 64 bits target.

Note also that there is no `DIV` opcode in the ARM so `ADIV` is a virtual instruction transformed in a series of instructions implementing the division in software, as explained in Section 12.6.2.

9.4.7 Large constant operand, `REGTMP`, and literal pools

I can finally show the translation of arithmetic instructions using a large constant (`C_LCON`^{113a}) or a negative rotatable constant (`C_NCON`^{113a}) as a first operand. Those instructions are more complex to handle than the instructions with a rotatable constant (`C_RCON`^{113a}) you saw before. Indeed, large constants can not be encoded directly in the instruction. This is why the rules in this section have a rule size of 8. I will first cover the rules and the code to deal with `C_NCON` which has a lot in common with `C_LCON` but is simpler to see first.

`C_NCON` represents a negative rotatable constant meaning a constant not directly rotatable, but with a negation that is rotatable, e.g., `-0xff` (since `0xff` is rotatable).

```
<optab entries 125a>+≡ (99b) <124c 126c>
{ AADD, C_NCON, C_REG, C_REG, 13, 8 },
{ AADD, C_NCON, C_NONE, C_REG, 13, 8 },
{ AMVN, C_NCON, C_NONE, C_REG, 13, 8 },
{ ACMP, C_NCON, C_REG, C_NONE, 13, 8 },
```

Uses `C_NCON` 113a, `C_NONE` 101a, and `C_REG` 101a.

The translation of instructions matching the rules above involves a temporary register reserved by the linker: `R11`. This register is aliased as `REGTMP` in the `ASSEMBLER` book [Pad15a] and in `include/obj/5.out.h`. So, `ADD $-0xff, R2, R3` is translated by 51 in the two following instructions:

```
MVN $0xff, R11 // o1
ADD R11, R2, R3 // o2
```

```
<asmout() switch on type cases 125b>+≡ (108b) <124d 127c>
case 13: /* op $lcon, [R], R */
    o1 = omvl(p, &p->from, REGTMP);
    <asmout() sanity check o1 126b>

    o2 = oprrr(p->as, p->scond);
    rf = REGTMP;
    rt = p->to.reg;
    r = p->reg;
    if(p->as == AMOVW || p->as == AMVN) // can be AMOVW??
        r = 0;
    else
        <asmout() adjust r 119c>
        o2 |= (r<<16) | rf;
        if(p->to.type != D_NONE)
            o2 |= rt << 12;
        break;
```

Uses `omvl()` 125c and `oprrr()` 120a.

The code above relies on `omvl()` (for move large) which returns an instruction which loads an operand `a` in a register `dr` (usually `REGTMP`):

```
<function omvl(arm) 125c>≡ (288)
long
omvl(Prog *p, Adr *a, int dr)
{
    long v, o1;
```

```

⟨omv1() when C_LCON case 127a⟩
else {
    // C_NCON case
    aclass(a);
    v = immrot(~instoffset);
    ⟨omv1() sanity check v 126a⟩
    o1 = oprrr(AMVN, p->scond&C_SCOND);
    o1 |= (dr<<12) | v;
}
return o1;
}

```

Uses aclass() 101b, immrot() 113c, instoffset 112f, and oprrr() 120a.

```

⟨omv1() sanity check v 126a⟩≡ (125c)
if(v == 0) {
    diag("missing literal");
    prasm(p);
    return 0;
}

```

Uses diag() 229d and prasm() 219b.

```

⟨asmout() sanity check o1 126b⟩≡ (138c 137b 135f 134a 132f 125b)
if(!o1)
    break;

```

I can now show the arithmetic rules using C_LCON and their translations. Those rules use the same action identifier than before, 13, as they share lots of the code to deal with C_NCON, but they have the special *rule flag* LFROM^{141c}.

```

⟨optab entries 126c⟩+≡ (99b) <125a 127b>
{ AADD, C_LCON, C_REG, C_REG, 13, 8, 0, LFROM },
{ AADD, C_LCON, C_NONE, C_REG, 13, 8, 0, LFROM },
{ AMVN, C_LCON, C_NONE, C_REG, 13, 8, 0, LFROM },
{ ACMP, C_LCON, C_REG, C_NONE, 13, 8, 0, LFROM },

```

Uses C_LCON 113a, C_NONE 101a, C_REG 101a, and LFROM 141c.

This flag is stored in an extra field of Optab^{99a}:

```

⟨Optab flag field 126d⟩≡ (99a)
// bitset<enum<Optab_flag>>
short flag;

```

As it will be explained fully later in Section 9.8, large constants in operands are transformed as data in the code section using the WORD pseudo-opcode. The instruction ADD \$0xffff, R2, R3 is thus transformed in the following instructions:

```

2000: LDR 1000(R15), R11
2004: ADD R11, R2, R3
...
...
3000: WORD $0xffff

```

When a rule with the LFROM rule flag matches an instruction p, dotext()^{94b} calls addpool()^{141f} with the from operand (p->from), as you will see in Section 9.8. addpool() extracts the constant from the operand and creates a new WORD instruction at the end of the program with this constant as an operand. This new instruction is part of what is called a *literal pool*, meaning a set of pseudo-instructions containing literals. addpool() also

modifies the passed instruction `p` and sets its `Instr.condX` field to point to this newly added instruction. Finally, `omvl` looks whether this field has been set, in which case it knows it is handling the `C_LCON` case:

```
<omvl() when C_LCON case 127a>≡ (125c)
if(p->cond) {
    // C_LCON case with Pool
    v = p->cond->pc - p->pc - 8;
    // =~ LDR v(R15), R11
    o1 = olr(AMOVW, p->scond&C_SCOND, v, REGPC, dr);
}
```

Uses `olr()` 131d.

You will see `olr()` ^{131d} later in Section 9.6.1 when I cover the memory instructions which loads data from memory in a register. The reason for the `-8` above will be explained when I cover the branching instructions in Section 9.5.

```
<optab entries 127b>+≡ (99b) <126c 128a>
{ AMOVW, C_NCON, C_NONE, C_REG, 12, 4 },
{ AMOVW, C_LCON, C_NONE, C_REG, 12, 4, 0, LFROM },
```

Uses `C_LCON` 113a, `C_NCON` 113a, `C_NONE` 101a, `C_REG` 101a, and `LFROM` 141c.

```
<asmout() switch on type cases 127c>+≡ (108b) <125b 128b>
case 12: /* movw $lcon, reg */
    o1 = omvl(p, &p->from, p->to.reg);
    break;
```

Uses `omvl()` 125c.

9.5 Control flow opcodes

The next big category of ARM instructions are branching instructions. They alter the control flow of the program.

```
<buildop() switch opcode r for ranges cases 127d>+≡ (104c) <124b 127e>
case AB:
case ABL:
    break;
```

`ABEQ` is the representant of a set of virtual opcodes to express *conditional jumps*. Those virtual instructions are converted in *unconditional jumps* (`AB`) but using the general mechanism of *conditional execution* provided by the ARM, as you will see soon.

```
<buildop() switch opcode r for ranges cases 127e>+≡ (104c) <127d 130b>
case ABEQ:
    oprange[ABNE] = oprange[r];
    oprange[ABHS] = oprange[r];
    oprange[ABLO] = oprange[r];
    oprange[ABMI] = oprange[r];
    oprange[ABPL] = oprange[r];
    oprange[ABVS] = oprange[r];
    oprange[ABVC] = oprange[r];
    oprange[ABHI] = oprange[r];
    oprange[ABLS] = oprange[r];
    oprange[ABGE] = oprange[r];
    oprange[ABLT] = oprange[r];
    oprange[ABGT] = oprange[r];
    oprange[ABLE] = oprange[r];
    break;
```

Uses `oprange` 100c.

9.5.1 Direct jumps

```
<optab entries 128a>+≡ (99b) <127b 129c>
{ AB, C_NONE, C_NONE, C_BRANCH, 5, 4, 0, LPOOL },
{ ABL, C_NONE, C_NONE, C_BRANCH, 5, 4 },
{ ABEQ, C_NONE, C_NONE, C_BRANCH, 5, 4 },
```

Uses C_BRANCH 101a, C_NONE 101a, and LPOOL 141c.

The rule flag LPOOL^{141c} will be explained later. You can see the important use of Instr.condX set by patch()⁸² in the code below to find the code address of the target instruction:

```
<asmout() switch on type cases 128b>+≡ (108b) <127c 129d>
case 5: /* bra s */
  <asmout() BRA case, if undefined target 176a>
  else
    if(p->cond != P)
      v = (p->cond->pc - p->pc) - 8;
    else
      v = -8; // warning?
  o1 = opbra(p->as, p->scond);
  o1 |= (v >> 2) & 0xffffffff;
  break;
```

Uses P 36c and opbra() 128c.

There are many interesting elements in the code above which require a small comment:

- p->cond->pc - p->pc: Jumps are *relative* in the ARM, but Instr.pcX contains an *absolute* code address, hence the subtraction.
- 0xffffffff: Jump offsets are encoded in 24 bits in the ARM.
- v >> 2: Jumps are in terms of instructions in the ARM, not bytes, hence the division by 4 since the ARM uses fixed-length instruction of 4 bytes.
- -8: The ARM implicitly does a +8 (see the EMULATOR book [Pad15b]) on the jump offset (after it multiplied it by 4). So, a jump offset of 0 in an ARM jump instruction actually jumps 2 instructions forward. The reason is probably that it seems incorrect to jump on the same instruction (hence an implicit +4), and it seems useless to jump on the next instruction (hence another implicit +4) as one could do the same by not using any jump instruction.

```
<function opbra(arm) 128c>≡ (288)
long
opbra(int a, int sc)
{

  <opbra() sanity check sc 129a>
  sc &= C_SCOND;
  if(a == ABL)
    return (sc<<28)|(0x5<<25)|(0x1<<24);
  <opbra() sanity check sc if not ABL 129b>

  switch(a) {
  // manual setting of the conditional execution
  // bits 28 to 32
  case ABEQ: return (0x0<<28)|(0x5<<25);
  case ABNE: return (0x1<<28)|(0x5<<25);
  case ABHS: return (0x2<<28)|(0x5<<25);
  case ABLO: return (0x3<<28)|(0x5<<25);
  case ABMI: return (0x4<<28)|(0x5<<25);
  case ABPL: return (0x5<<28)|(0x5<<25);
```

```

    case ABVS: return (0x6<<28)|(0x5<<25);
    case ABVC: return (0x7<<28)|(0x5<<25);
    case ABHI: return (0x8<<28)|(0x5<<25);
    case ABLs: return (0x9<<28)|(0x5<<25);
    case ABGE: return (0xa<<28)|(0x5<<25);
    case ABLT: return (0xb<<28)|(0x5<<25);
    case ABGT: return (0xc<<28)|(0x5<<25);
    case ABLE: return (0xd<<28)|(0x5<<25);
    case AB: return (0xe<<28)|(0x5<<25);
}
diag("bad bra %A", a);
prasm(curp);
return 0;
}

```

Uses curp 36g, diag() 229d, and prasm() 219b.

```

⟨opbra() sanity check sc 129a⟩≡ (128c)
    if(sc & (C_SBIT|C_PBIT|C_WBIT))
        diag(".S/.P/.W on bra instruction");

```

Uses diag() 229d.

```

⟨opbra() sanity check sc if not ABL 129b⟩≡ (128c)
    if(sc != 0xe)
        diag("redundant .EQ/.NE/... on B/BEQ/BNE/... instruction");

```

Uses diag() 229d.

9.5.2 Indirect jumps

Most jumps in a program are *static*, e.g., jumping to a procedure or a label. Even the assembly instruction B 2(PC), which uses the pseudo-register PC (to perform a relative jump), is actually converted in a static jump. Indeed, the instruction is (1) converted by the assembler in an absolute jump, (2) relocated in `ldobj()`⁵⁵ in Section ??, (3) patched to find the actual target instruction in Section 7.2, and finally (4) converted back to a static relative jump ARM instruction in the previous section.

The assembly language of 5a supports also *dynamic* jumps where the content of a regular register and an offset can specify an absolute code address to jump to, e.g., B 10(R4). Such instructions are useful to encode for instance in C calls through a function pointer as in `(*f)(1,2)`; which are dynamic. The ARM branching instructions support only 24 bits static offsets though, but you can manipulate directly the program counter register (R15) and link register (R14) to encode dynamic jumps using simple arithmetic instructions:

```

⟨optab entries 129c⟩+≡ (99b) <128a 130c>
    { AB, C_NONE, C_NONE, C_ROREG, 6, 4, 0, LPOOL },
    { ABL, C_NONE, C_NONE, C_ROREG, 7, 8 },

```

Uses C_NONE 101a, C_ROREG 114a, and LPOOL 141c.

```

⟨asmout() switch on type cases 129d⟩+≡ (108b) <128b 130a>
    case 6: /* b 0(R) -> add $0,R,PC */
        aclass(&p->to);
        o1 = oprrr(AADD, p->scond);
        o1 |= immrot(instoffset);
        r = p->to.reg;
        rt = REGPC;
        o1 |= (r<<16) | (rt<<12);
        break;

```

Uses aclass() 101b, immrot() 113c, instoffset 112f, and oprrr() 120a.

The translation of indirect calls (BL) requires two ARM instructions:

```

<asmout() switch on type cases 130a)>+≡ (108b) <129d 131a>
case 7: /* bl Offset(R) -> ADD $0,PC,LINK; add $Offset,R,PC */
    aclass(&p->to);
    o1 = oprrr(AADD, p->scond);
    rt = REGLINK;
    r = REGPC;
    o1 |= (r<<16) | (rt<<12) | immrot(0);

    o2 = oprrr(AADD, p->scond);
    r = p->to.reg;
    rt = REGPC;
    o2 |= (r<<16) | (rt<<12) | immrot(instoffset);
    break;

```

Uses `aclass()` 101b, `immrot()` 113c, `instoffset` 112f, and `opr_rr()` 120a.

Note that the first generated instruction above which adds zero to R15 and saves the result in R14 seems incorrect. Indeed, the return address saved in R14 should not be the current instruction but the next instruction, which in our case should be at R15+8 since the linker generated two ARM instructions for BL. But, again the ARM does implicitly a +8 when one of the *source* operand of an arithmetic instruction is R15 (see the EMULATOR book [Pad15b]). Note that there is no implicit +8 though when R15 is the *destination* operand as in the second generated ARM instruction above.

9.6 Memory opcodes

I will show in this section the many ARM code generation rules for AMOVW and its variants when one of its operands references a memory area.

```

<buildop() switch opcode r for ranges cases 130b)>+≡ (104c) <127e 134e>
case AMOVW:
case AMOVB:
case AMOVBU:
case AMOVH:
case AMOVHU:
    break;

```

You have seen already a few rules concerning AMOVW in Section 9.4 when the operands of AMOVW were simple registers and constants. Indeed, AMOVW is a very general virtual instruction which unifies with one virtual opcode many ARM opcodes: MOV, MVN, but more importantly for this section LDR, STR, and even more as some AMOVW can also be translated in [ADD]]. In fact, one move instruction can lead to the generation of up to 6 ARM instructions (24 bytes) as you will see soon.

9.6.1 Load

An important use of MOVW, beyond the setting of a register from another register or a constant (via the ARM instructions MOV and MVN), is to *load* data from memory in a register, e.g., MOVW 10(R1), R2. This will result in the use of the ARM instruction LDR. I will first show the rules when the offset is small, that is when the operand class has the form C_Sxxx, e.g., C_SEXT^{115b} (see the code of `immaddr()`^{115a}). In those cases the offset can be encoded directly in the ARM instruction, hence the rule size 4 below:

```

<optab entries 130c)>+≡ (99b) <129c 132e>
{ AMOVW, C_SEXT, C_NONE, C_REG, 21, 4, REGSB },
{ AMOVW, C_SAUTO,C_NONE, C_REG, 21, 4, REGSP },
{ AMOVW, C_SOREG,C_NONE, C_REG, 21, 4 },

{ AMOVBU, C_SEXT, C_NONE, C_REG, 21, 4, REGSB },

```

```
{ AMOVBU, C_SAUTO,C_NONE, C_REG, 21, 4, REGSP },
{ AMOVBU, C_SOREG,C_NONE, C_REG, 21, 4 },
```

Uses C_NONE 101a, C_REG 101a, C_SAUTO 116c, C_SEXT 115b, and C_SOREG 114a.

```
<asmout() switch on type cases 131a>+≡ (108b) <130a 132f>
case 21: /* mov/movbu 0(R),R */
    aclass(&p->from);
    rt = p->to.reg;
    r = p->from.reg;
    <asmout() adjust maybe r to rule param 131c>
    o1 = olr(p->as, p->scnd, instoffset, r, rt);
    break;
```

Uses aclass() 101b, instoffset 112f, and olr() 131d.

Remember that aclass() ^{101b} above modifies the global instoffset ^{112f} whose value depends on the operand class parameter. instoffset can contain a computed offset to R12 for external symbols (EXT, see Section 9.3.3), an adjusted offset to R13 for stack access (AUTO, see Section 9.3.4), or the given offset for the other cases (OREG, see Section 9.3.2). Each kind of offset corresponds to a different *base register*, which can actually be specified in the rule itself. Indeed, an extra field of Optab ^{99a} can be used to contain extra parameters:

```
<Optab param field 131b>≡ (99a)
// 0 | REGSB | REGSP
short param;
```

For instance REGSB is used in when the rule concerns an external symbol (C_SEXT). This extra parameter can then be accessed from asmout() ^{108b} to adjust the base register:

```
<asmout() adjust maybe r to rule param 131c>≡ (211 210 197a 196 195c 138 137b 136b 135 134a 133c 132f 131a)
if(r == R_NONE)
    r = o->param;
```

The code for case 21: above relies on olr() which factorizes most of the code related to the instruction format of memory instructions. It is actually also used for the generation of STR instructions as you will see soon.

```
<function olr(arm) 131d>≡ (288)
long
olr(int a, int sc, long v, int b, int rt)
{
    long o;

    o = (sc&C_SCOND) << 28;
    <olr() sanity check sc 132b>

    <olr() set special bits 132a>
    o |= (0x1<<26) | // memory instructions class
        (1<<20); // LDR
    if(v >= 0) {
        o |= 1 << 23; // Up bit, positive offset
    } else {
        // bit 23 unset, Down, negative offset
        v = -v;
    }
    <olr() sanity check offset v 132c>
    switch(a) {
    case AMOVB:
    case AMOVBU:
        o |= 1<<22;
        break;
    case AMOVW:
        break;
```

```

default:
    <olr() sanity check a 132d>
}
o |= (b<<16) | (rt<<12) | v;
return o;
}

```

Remember from the ASSEMBLER book [Pad15a] that memory instructions can use the .P or .W suffixes to offer additional addressing modes. Those suffixes are converted by 5a in special bits in `Instr.condX` which then leads to the setting of special bits in the ARM instruction itself:

```

<olr() set special bits 132a>≡ (131d)
if(!(sc & C_PBIT))
    o |= 1 << 24; // pre (not post)
if(sc & C_WBIT)
    o |= 1 << 21; // write back

```

```

<olr() sanity check sc 132b>≡ (131d)
if(sc & C_SBIT)
    diag(".S on LDR/STR instruction");
if(sc & C_UBIT)
    diag(".U on LDR/STR instruction");

```

Uses `diag()` 229d.

```

<olr() sanity check offset v 132c>≡ (131d)
if(v >= (1<<12))
    diag("literal span too large: %ld (R%d)\nnP", v, b, curp);

```

Uses `curp` 36g and `diag()` 229d.

```

<olr() sanity check a 132d>≡ (131d)
diag("expect move operation, not: %P", curp);

```

Uses `curp` 36g and `diag()` 229d.

The use of large offsets (`C_Lxxx`) requires an additional instruction to first load the offset in `REGTMP`:

```

<optab entries 132e>+≡ (99b) <130c 133b>
{ AMOVW, C_LEXT, C_NONE, C_REG, 31, 8, REGSB, LFROM },
{ AMOVW, C_LAUTO,C_NONE, C_REG, 31, 8, REGSP, LFROM },
{ AMOVW, C_LOREG,C_NONE, C_REG, 31, 8, 0, LFROM },

{ AMOVBU, C_LEXT, C_NONE, C_REG, 31, 8, REGSB, LFROM },
{ AMOVBU, C_LAUTO,C_NONE, C_REG, 31, 8, REGSP, LFROM },
{ AMOVBU, C_LOREG,C_NONE, C_REG, 31, 8, 0, LFROM },

```

Uses `C_LAUTO` 116c, `C_LEXT` 115b, `C_LOREG` 114a, `C_NONE` 101a, `C_REG` 101a, and `LFROM` 141c.

The code below uses also `omvl()`^{125c} which you saw in Section 9.8 for arithmetic instructions. Remember than `omvl()` internally calls `olr()`^{131d}. Indeed, it also needs here to load from a literal pool a large number, here an offset. So loading certain data from memory requires actually two LDR instructions.

```

<asmout() switch on type cases 132f>+≡ (108b) <131a 133c>
case 31: /* mov/movbu L(R),R */
    o1 = omvl(p, &p->from, REGTMP);
    <asmout() sanity check o1 126b>

```

```

    rt = p->to.reg;
    r = p->from.reg;
    <asmout() adjust maybe r to rule param 131c>
    o2 = olrr(p->as, p->scond, REGTMP, r, rt);
    break;

```

Uses `olrr()` 133a and `omvl()` 125c.

Note that *i* below represents a register number (REGTMP) which is passed to `olr()` instead of `instoffset` in previous calls. Indeed, the immediate offset or the register containing the offset are both encoded at the beginning of the ARM instruction (the low bits).

```
<function olrr(arm) 133a>≡ (288)
long
olrr(int a, int sc, int i, int b, int r)
{
```

```
    return olr(a, sc, i, b, r) | (1<<25); // Rm not immediate offset
}
```

Uses `olr()` [131d](#).

9.6.2 Store

MOVW can be used to *store* data from a register in memory, e.g., `MOVW R2, 10(R1)`. This will result in the use of the ARM instruction STR.

```
<optab entries 133b>+≡ (99b) <132e 133e>
{ AMOVW, C_REG, C_NONE, C_SEXT, 20, 4, REGSB },
{ AMOVW, C_REG, C_NONE, C_SAUTO, 20, 4, REGSP },
{ AMOVW, C_REG, C_NONE, C_SOREG, 20, 4, 0 },
```

```
{ AMOVBU, C_REG, C_NONE, C_SEXT, 20, 4, REGSB },
{ AMOVBU, C_REG, C_NONE, C_SAUTO, 20, 4, REGSP },
{ AMOVBU, C_REG, C_NONE, C_SOREG, 20, 4, 0 },
```

Uses `C_NONE` [101a](#), `C_REG` [101a](#), `C_SAUTO` [116c](#), `C_SEXT` [115b](#), and `C_SOREG` [114a](#).

```
<asmout() switch on type cases 133c>+≡ (108b) <132f 134a>
case 20: /* mov/movb/movbu R,0(R) */
    aclass(&p->to);
    rf = p->from.reg;
    r = p->to.reg;
    <asmout() adjust maybe r to rule param 131c>
    o1 = osr(p->as, p->scond, rf, instoffset, r);
    break;
```

Uses `aclass()` [101b](#), `instoffset` [112f](#), and `osr()` [133d](#).

```
<function osr(arm) 133d>≡ (288)
long
osr(int a, int sc, int r, long v, int b)
{
```

```
    return olr(a, sc, v, b, r) ^ (1<<20); // STR, unset (via xor) bit 20
}
```

Uses `olr()` [131d](#).

Note that `osr()` [133d](#) is a small wrapper around `olr()` [131d](#) as the instruction format for LDR and STR are very similar except for the bit 20.

```
<optab entries 133e>+≡ (99b) <133b 134c>
{ AMOVW, C_REG, C_NONE, C_LEXT, 30, 8, REGSB, LTO },
{ AMOVW, C_REG, C_NONE, C_LAUTO, 30, 8, REGSP, LTO },
{ AMOVW, C_REG, C_NONE, C_LOREG, 30, 8, 0, LTO },
```

```
{ AMOVBU, C_REG, C_NONE, C_LEXT, 30, 8, REGSB, LTO },
{ AMOVBU, C_REG, C_NONE, C_LAUTO, 30, 8, REGSP, LTO },
{ AMOVBU, C_REG, C_NONE, C_LOREG, 30, 8, 0, LTO },
```

Uses `C_LAUTO` [116c](#), `C_LEXT` [115b](#), `C_LOREG` [114a](#), `C_NONE` [101a](#), `C_REG` [101a](#), and `LTO` [141c](#).

```

<asmout() switch on type cases 134a>+≡ (108b) <133c 135a>
case 30: /* mov/movb/movbu R,L(R) */
    o1 = omvl(p, &p->to, REGTMP);
    <asmout() sanity check o1 126b>

```

```

    rf = p->from.reg;
    r = p->to.reg;
    <asmout() adjust maybe r to rule param 131c>
    o2 = osrr(p->as, p->scond, rf, REGTMP, r);
    break;

```

Uses omvl() 125c and osrr() 134b.

```

<function osrr(arm) 134b>≡ (288)
long
osrr(int a, int sc, int r, int i, int b)
{
    return olrr(a, sc, i, b, r) ^ (1<<20); // STR
}

```

Uses olrr() 133a.

The previous cases work also for AMOVB:

```

<optab entries 134c>+≡ (99b) <133e 134d>
{ AMOVB, C_REG, C_NONE, C_SEXT, 20, 4, REGSB },
{ AMOVB, C_REG, C_NONE, C_SAUTO, 20, 4, REGSP },
{ AMOVB, C_REG, C_NONE, C_SOREG, 20, 4, 0 },

```

Uses C_NONE 101a, C_REG 101a, C_SAUTO 116c, C_SEXT 115b, and C_SOREG 114a.

```

<optab entries 134d>+≡ (99b) <134c 134f>
{ AMOVB, C_REG, C_NONE, C_LEXT, 30, 8, REGSB, LTO },
{ AMOVB, C_REG, C_NONE, C_LAUTO, 30, 8, REGSP, LTO },
{ AMOVB, C_REG, C_NONE, C_LOREG, 30, 8, 0, LTO },

```

Uses C_LAUTO 116c, C_LEXT 115b, C_LOREG 114a, C_NONE 101a, C_REG 101a, and LTO 141c.

Indeed, storing a signed byte or an unsigned byte from a register involves the same ARM instruction. This is not the case though for loading. Indeed, loading a signed byte from memory in a register may require to fill with 1s the 24 upper bits of the register if the number in memory was negative. You will see the rules for loading and AMOVB later in Section 9.6.5.

9.6.3 Swaps

SWP and SWPBU have a format more strict than other memory operations.

```

<buildop() switch opcode r for ranges cases 134e>+≡ (104c) <130b 139a>
case ASWPW:
    oprange[ASWPBU] = oprange[r];
    break;

```

Uses oprange 100c.

```

<optab entries 134f>+≡ (99b) <134d 135c>
{ ASWPW, C_SOREG,C_REG, C_REG, 40, 4 },

```

Uses C_REG 101a and C_SOREG 114a.

```

<asmout() switch on type cases 135a>+≡ (108b) <134a 135d>
case 40: /* swp oreg,reg,reg */
    aclass(&p->from);
    <asmout() sanity check instoffset for SWP 135b>
    o1 = (0x2<<23) | (0x9<<4); // SWP
    if(p->as == ASWPBU)
        o1 |= 1 << 22;
    rf = p->from.reg;
    rt = p->to.reg;
    r = p->reg;
    o1 |= ((p->scond & C_SCOND) << 28) | (rf<<16) | (rt<<12) | r;
    break;

```

Uses `aclass()` 101b.

```

<asmout() sanity check instoffset for SWP 135b>≡ (135a)
if(instoffset != 0)
    diag("offset must be zero in SWP");

```

Uses `diag()` 229d and `instoffset` 112f.

9.6.4 Symbol addresses

The loading of an address (e.g., `MOVW $hello(SB), R1`) which happens after resolution to be a small rotatable offset to a base register (e.g., `SB`) can be encoded efficiently using simply `ADD`:

```

<optab entries 135c>+≡ (99b) <134f 135e>
{ AMOVW, C_RECON,C_NONE, C_REG, 4, 4, REGSB },
{ AMOVW, C_RACON,C_NONE, C_REG, 4, 4, REGSP },

```

Uses `C_NONE` 101a, `C_RACON` 117d, `C_RECON` 116g, and `C_REG` 101a.

```

<asmout() switch on type cases 135d>+≡ (108b) <135a 135f>
case 4: /* add $I, [R],R */
    aclass(&p->from);
    o1 = oprrr(AADD, p->scond);
    rt = p->to.reg;
    r = p->from.reg;
    <asmout() adjust maybe r to rule param 131c>
    o1 |= (r<<16) | (rt<<12) | immrot(instoffset);
    break;

```

Uses `aclass()` 101b, `immrot()` 113c, `instoffset` 112f, and `oprrr()` 120a.

The address of an entity which is not a rotatable offset must be loaded from a literal pool with `omvl()`^{125c}:

```

<optab entries 135e>+≡ (99b) <135c 136a>
{ AMOVW, C_LACON,C_NONE, C_REG, 34, 8, REGSP, LFROM },

```

Uses `C_LACON` 117d, `C_NONE` 101a, `C_REG` 101a, and `LFROM` 141c.

```

<asmout() switch on type cases 135f>+≡ (108b) <135d 136b>
case 34: /* mov $lacon,R -> LDR x(R15), R11; ADD R11, R13, R */
    o1 = omvl(p, &p->from, REGTMP);
    <asmout() sanity check o1 126b>

    o2 = oprrr(AADD, p->scond);
    rf = REGTMP;
    rt = p->to.reg;
    r = p->from.reg;
    <asmout() adjust maybe r to rule param 131c>
    o2 |= (r<<16) | (rt<<12) | rf;
    break;

```

Uses `omvl()` 125c and `oprrr()` 120a.

9.6.5 Half words and signed bytes

The last rules for memory instructions I will show in this chapter concern the loading and storing of signed and unsigned half words (2 bytes) as well as signed bytes (1 byte). Many ARM processors have special support for those instructions. In this section though I will show the simpler ARM code generation rules for old ARM processors. In those cases, MOVH, MOVHU and MOVB are virtual instructions transformed by 51 in multiple ARM instructions using LDR/STR and bitshift instructions. The rules for more recent ARM processors are described in Section 12.6.6.

Load

```
<optab entries 136a>+≡ (99b) <135e 137a>
{ AMOVH, C_SEXT, C_NONE, C_REG, 22, 12, REGSB },
{ AMOVH, C_SAUTO, C_NONE, C_REG, 22, 12, REGSP },
{ AMOVH, C_SOREG, C_NONE, C_REG, 22, 12, 0 },

{ AMOVHU, C_SEXT, C_NONE, C_REG, 22, 12, REGSB },
{ AMOVHU, C_SAUTO, C_NONE, C_REG, 22, 12, REGSP },
{ AMOVHU, C_SOREG, C_NONE, C_REG, 22, 12, 0 },

{ AMOVB, C_SEXT, C_NONE, C_REG, 22, 12, REGSB },
{ AMOVB, C_SAUTO, C_NONE, C_REG, 22, 12, REGSP },
{ AMOVB, C_SOREG, C_NONE, C_REG, 22, 12, 0 },
```

Uses C_NONE 101a, C_REG 101a, C_SAUTO 116c, C_SEXT 115b, and C_SOREG 114a.

MOVHU 10(R1), R2 is transformed in 3 instructions:

```
LDR 10(R1), R2 // load more than needed
SLL $16, R2 // reset the 16
SRL $16, R2 // upper bits
```

```
<asmout() switch on type cases 136b>+≡ (108b) <135f 137b>
case 22: /* movb/movh/movhu 0(R),R -> lr,shl,shr */
    aclass(&p->from);
    rt = p->to.reg;
    r = p->from.reg;
    <asmout() adjust maybe r to rule param 131c>
    o1 = olr(AMOVW, p->scond, instoffset, r, rt);

    o2 = oprrr(ASLL, p->scond);
    if(p->as == AMOVHU)
        o3 = oprrr(ASRL, p->scond);
    else
        o3 = oprrr(ASRA, p->scond);

    if(p->as == AMOVB) {
        o2 |= (rt<<12) | (24<<7) | rt;
        o3 |= (rt<<12) | (24<<7) | rt;
    } else {
        o2 |= (rt<<12) | (16<<7) | rt;
        o3 |= (rt<<12) | (16<<7) | rt;
    }
    break;
```

Uses aclass() 101b, instoffset 112f, olr() 131d, and oprrr() 120a.

Note that the rules for AMOVH and loading are in this section while the rules for AMOVBU and storing were presented before and equivalent to the rules for AMOVBU. This asymmetry is due to how signed integers are represented in the machine: in *two's complement* form. Loading a signed byte from memory in a 32 bits register

may require to fill with 1s the 24 upper bits of the register if the number in memory was negative. For instance the signed byte -128 is represented in binary as 0b1000_0000. The signed word -128 is represented in binary as 0b11...1000_0000 though. So one needs first to logical shift to the left (SLL) 0b1000_0000 24 times, and then do an *arithmetic* shift to the right (SRA) 24 times back so the possible presence of a 1 in bit 8 will be repeated in all the upper bits.

```
<optab entries 137a>+≡ (99b) <136a 137c>
{ AMOVH, C_LEXT, C_NONE, C_REG, 32, 16, REGSB, LFROM },
{ AMOVH, C_LAUTO,C_NONE, C_REG, 32, 16, REGSP, LFROM },
{ AMOVH, C_LOREG,C_NONE, C_REG, 32, 16, 0, LFROM },

{ AMOVHU, C_LEXT, C_NONE, C_REG, 32, 16, REGSB, LFROM },
{ AMOVHU, C_LAUTO,C_NONE, C_REG, 32, 16, REGSP, LFROM },
{ AMOVHU, C_LOREG,C_NONE, C_REG, 32, 16, 0, LFROM },

{ AMOVb, C_LEXT, C_NONE, C_REG, 32, 16, REGSB, LFROM },
{ AMOVb, C_LAUTO,C_NONE, C_REG, 32, 16, REGSP, LFROM },
{ AMOVb, C_LOREG,C_NONE, C_REG, 32, 16, 0, LFROM },
```

Uses C_LAUTO 116c, C_LEXT 115b, C_LOREG 114a, C_NONE 101a, C_REG 101a, and LFROM 141c.

```
<asmout() switch on type cases 137b>+≡ (108b) <136b 138a>
case 32: /* movh/movb L(R),R */
    o1 = omvl(p, &p->from, REGTMP);
    <asmout() sanity check o1 126b>

    rt = p->to.reg;
    r = p->from.reg;
    <asmout() adjust maybe r to rule param 131c>
    o2 = olrr(p->as, p->scond, REGTMP,r, rt);

    o3 = oprrr(ASLL, p->scond);
    if(p->as == AMOVHU)
        o4 = oprrr(ASRL, p->scond);
    else
        o4 = oprrr(ASRA, p->scond);

    if(p->as == AMOVb) {
        o3 |= (rt<<12) | (24<<7) | rt;
        o4 |= (rt<<12) | (24<<7) | rt;
    } else {
        o3 |= (rt<<12) | (16<<7) | rt;
        o4 |= (rt<<12) | (16<<7) | rt;
    }
    break;
```

Uses olrr() 133a, omvl() 125c, and oprrr() 120a.

Store

```
<optab entries 137c>+≡ (99b) <137a 138b>
{ AMOVH, C_REG, C_NONE, C_SEXT, 23, 12, REGSB },
{ AMOVH, C_REG, C_NONE, C_SAUTO, 23, 12, REGSP },
{ AMOVH, C_REG, C_NONE, C_SOREG, 23, 12, 0 },

{ AMOVHU, C_REG, C_NONE, C_SEXT, 23, 12, REGSB },
{ AMOVHU, C_REG, C_NONE, C_SAUTO, 23, 12, REGSP },
{ AMOVHU, C_REG, C_NONE, C_SOREG, 23, 12, 0 },
```

Uses C_NONE 101a, C_REG 101a, C_SAUTO 116c, C_SEXT 115b, and C_SOREG 114a.

MOVHU R2, 10(R1) is transformed in 3 instructions:

```
LDRB R2, 10(R1) // load first byte
SRL $8, R2, R11 // extract second byte
LDRB R11, 11(R1) // load second byte
```

```
<asmout() switch on type cases 138a>+≡ (108b) <137b 138c>
case 23: /* movh/movhu R,0(R) -> sb,sb */
    aclass(&p->to);
    rf = p->from.reg;
    r = p->to.reg;
    <asmout() adjust maybe r to rule param 131c>
    o1 = osr(AMOVBU, p->scond, rf, instoffset, r);

    o2 = oprrr(ASRL, p->scond);
    rt = REGTMP;
    o2 |= (rt<<12) | (8<<7) | rf;

    o3 = osr(AMOVBU, p->scond, REGTMP, instoffset+1, r);
    break;
```

Uses aclass() 101b, instoffset 112f, oprrr() 120a, and osr() 133d.

Note the rule size of 24 below for the translation of AMOVH using a large offset:

```
<optab entries 138b>+≡ (99b) <137c 139b>
{ AMOVH, C_REG, C_NONE, C_LEXT, 33, 24, REGSB, LTO },
{ AMOVH, C_REG, C_NONE, C_LAUTO, 33, 24, REGSP, LTO },
{ AMOVH, C_REG, C_NONE, C_LOREG, 33, 24, 0, LTO },

{ AMOVHU, C_REG, C_NONE, C_LEXT, 33, 24, REGSB, LTO },
{ AMOVHU, C_REG, C_NONE, C_LAUTO, 33, 24, REGSP, LTO },
{ AMOVHU, C_REG, C_NONE, C_LOREG, 33, 24, 0, LTO },
```

Uses C_LAUTO 116c, C_LEXT 115b, C_LOREG 114a, C_NONE 101a, C_REG 101a, and LTO 141c.

MOVHU R2, 0xffff(R1) is transformed in 6 ARM instructions:

```
LDR xxx(R15), R11 // load offset
STR.B R2, R11(R1) // store first byte
ROR $8, R2, R2 // MOV R2@>8, R2
ADD $1, R11, R11
STR.B R2, R11(R1) // store second byte
ROR $24, R2, R2 // MOV R2@>24, R2
```

```
<asmout() switch on type cases 138c>+≡ (108b) <138a 139c>
case 33: /* movh/movhu R,L(R) -> sb, sb */
    o1 = omvl(p, &p->to, REGTMP);
    <asmout() sanity check o1 126b>

    rf = p->from.reg;
    r = p->to.reg;
    <asmout() adjust maybe r to rule param 131c>
    o2 = osrr(AMOVBU, p->scond, rf, REGTMP, r);

    o3 = oprrr(ASRL, p->scond);
    o3 |= (rf<<12) | (8<<7) | rf;
    o3 |= (1<<6); /* ROR 8 */

    o4 = oprrr(AADD, p->scond);
    o4 |= (REGTMP<<16) | (REGTMP<<12);
    o4 |= immrot(1);
```

```

o5 = osrr(AMOVBU, p->scond, rf, REGTMP,r);

// restore rf
o6 = oprrr(ASRL, p->scond);
o6 |= (rf<<12) | (24<<7) | rf;
o6 |= (1<<6); /* ROL 8 */

break;

```

Uses `immrot()` 113c, `omvl()` 125c, `opr`rr() 120a, and `osrr()` 134b.

9.7 Software interrupt opcodes

The final ARM code generation rules concern software interrupts. SWI creates a software interrupt and performs a *system call* to a kernel function. RFE returns from an *interrupt handler* in the kernel. RFE stands for “Return From Exception” as certain interrupts are due to hardware exceptions (e.g. division by zero).

```

<buildop() switch opcode r for ranges cases 139a>+≡ (104c) <134e 194d>
case ASWI:
case ARFE:
break;

```

Normally the argument to SWI in the ARM specifies an entry in an interrupt table but under Plan 9 this argument is actually not used. Instead, R0 is used to store the system call code.

```

<optab entries 139b>+≡ (99b) <138b 139e>
{ ASWI, C_NONE, C_NONE, C_NONE, 10, 4 },
{ ASWI, C_NONE, C_NONE, C_LCON, 10, 4 },
{ ASWI, C_NONE, C_NONE, C_LOREG, 10, 4 },

```

Uses `C_LCON` 113a, `C_LOREG` 114a, and `C_NONE` 101a.

```

<asmout() switch on type cases 139c>+≡ (108b) <138c 139f>
case 10: /* swi [$con] */
o1 = oprrr(p->as, p->scond);
if(p->to.type != D_NONE) {
aclass(&p->to);
o1 |= instoffset & 0xfffff;
}
break;

```

Uses `aclass()` 101b, `instoffset` 112f, and `opr`rr() 120a.

```

<oprrr() switch cases 139d>+≡ (120a) <124e 197b>
case ASWI: return o | (0xf<<24);

```

RFE is actually a virtual instruction which is transformed by 51 in an instruction using `MOVM`, the multiple register move opcode (see Section 12.6.4):

```

<optab entries 139e>+≡ (99b) <139b 180c>
{ ARFE, C_NONE, C_NONE, C_NONE, 41, 4 },

```

Uses `C_NONE` 101a.

```

<asmout() switch on type cases 139f>+≡ (108b) <139c 181b>
case 41: /* rfe -> movm.s.w.u 0(r13), [r15] */
o1 = 0xe8fd8000;
break;

```

9.8 Literal Pools

I will now fully explain the final piece of the ARM code generation: the management of *literal pools*. Those pools are used for the translation of instructions using large constants, e.g., `ADD $0xffff, R2, R3`, or large offsets, e.g., `MOVW 0xffce(R1), R2`). As explained in Section , large constants in operands are transformed as data in the code section by using the `WORD` pseudo-opcode. The two previous instructions are thus transformed, thanks to the help of data structures and functions presented in this section, in the following ARM instructions:

```
// ADD $0xffff, R2, R3
2000: LDR 1000(R15), R11
2004: ADD R11, R2, R3
// MOVW 0xffce(R1), R2
2008: LDR 996(R15), R11
2012: LDR R11(R1), R2
...
// literal pool at the end of the code section
3000: WORD $0xffff
3004: WORD $0xffce
```

9.8.1 blitrl/elitrl

Before diving into the code, it helps to visualise where pools sit in the final text layout. Unlike x86, where a 32-bit immediate can be embedded right after the opcode, ARM loads a large constant with `LDR Roff(R15), Rdst` where the 12-bit offset must reach a `WORD` datum. The linker therefore scatters small pools inside the text section, typically right after an unconditional branch so execution skips over them:

```
text addr
...
1f00: LDR 1008(R15), R11 ; from ADD $0xffff...
1f04: ADD R11, R2, R3
1f08: LDR 1004(R15), R11 ; from MOVW 0xffce(R1), R2
1f0c: LDR R11(R1), R2
1f10: B 1f20 ; branch over pool (inserted)
1f14: WORD $0xffff |
1f18: WORD $0xffce | <- literal pool
1f1c: WORD $0xdead | (blitrl..elitrl)
1f20: MOVW ... ; next real instruction
```

Two properties fall out of this layout. First, each pool has to stay within $\pm 4\text{KB}$ (a 12-bit offset) of every instruction that references it; `checkpool()`^{143d} flushes the pool early when that window threatens to close, which is why pools can appear mid-function rather than only at its end. Second, the pool itself lives in the text section, not in data, because only text has a stable PC-relative relationship with the loads: putting pools in `.data` would work on ELF but would need real relocations and a larger addressing window. Java bytecode and Swift take the opposite end of this design space, with a dedicated constant pool per class or module; ARM's limited immediates push 51 closer to a per-function micro-pool.

The literal pool, which is a list of `WORD` instructions, is represented by the globals `blitrl` and `elitrl` which respectively stores the beginning and end of this list:

```
<global blitrl(arm) 140>≡ (286a)
// list<ref_own<prog>> (next = Prog.link)
Prog* blitrl;
```

```

⟨global elitrl(arm) 141a⟩≡
// ref<Prog> (end from = blitrl)
Prog* elitrl;

```

(286a)

9.8.2 addpool()

As mentioned in Section 9.8, the addition of large constants in the literal pool is done through the function `addpool()`^{141f}. This function is called from `dotext()`^{94b}, as you will see soon, thanks to a *rule flag* indicating which operand in the instruction contains the large constant. Here are past examples of rules with a rule flag:

```

⟨optab example of entries with rule flags (repeated) 141b⟩≡
{ AADD, C_LCON, C_REG, C_REG, 13, 8, 0, LFROM },
{ AMOVW, C_REG, C_NONE, C_LOREG, 30, 8, 0, LTO },

```

The set of rule flags is defined by the type below:

```

⟨enum Optab_flag(arm) 141c⟩≡
enum Optab_flag {
// Flags related to literal pools
LFROM = 1<<0,
LTO = 1<<1,

LPOOL = 1<<2,
// Flags related architecture restrictions
⟨Optab_flag cases 191d⟩
};

```

(283a)

I will describe LPOOL later. The remaining flags are described in Chapter 12.

Remember from Section 7.1.1 that `dotext()` layouts code and calls `oplook()`^{102c} to find which rule `o` matches an instruction `p`. The presence of a rule flag in the rule `o` determines the possible call to `addpool()`:

```

⟨dotext() pool handling for optab o 141d⟩≡
switch(o->flag & (LFROM|LTO|LPOOL)) {
⟨dotext() pool handling, switch flag cases 141e⟩
}
⟨dotext() pool handling, flush if MOVW REGPC 145a⟩
⟨dotext() pool handling, checkpool 143c⟩

```

(94b)

Uses LFROM 141c, LPOOL 141c, and LTO 141c.

```

⟨dotext() pool handling, switch flag cases 141e⟩≡
case LFROM:
addpool(p, &p->from);
break;
case LTO:
addpool(p, &p->to);
break;

```

(141d) 144e>

Uses LFROM 141c, LTO 141c, and `addpool()` 141f.

`addpool()` essentially adds a new WORD instruction in the list `blitrl`¹⁴⁰ with `a` as an operand (in `Instr.toX`), the large constant or large offset in the operand parameter `a`. It also modifies the parameter `p` so that its `Instr.condX` field links to the newly added WORD instruction:

```

⟨function addpool(arm) 141f⟩≡
void
addpool(Prog *p, Adr *a)
{
Prog *q;
Prog t;
//enum<Operand_class>
int c;

```

(286a)

```

c = aclass(a);

t = zprg;
t.as = AWORD;
⟨addpool() set t.to using a 142a⟩

⟨addpool() if literal already present in pool 143a⟩
// else

q = prg();
*q = t;

⟨addpool() set pool.start and pool.size 144b⟩

// add_queue(q, blitrl, elitrl)
if(blitrl == P) {
    blitrl = q;
} else
    elitrl->link = q;
elitrl = q;

// for omvl()!
p->cond = q;
}

```

Uses P 36c, aclass() 101b, blitrl 140, elitrl 141a, prg() 43a, and zprg 43b.

Thanks to the instruction `p->cond = q` above, `omvl()`^{125c} in Section 9.8, will be able to know the offset to R15 it needs to generate, e.g., `LDR 1000(R15), ...`, to load this new literal. Note that once the literal pool will be added at the end of the list of code instructions (via `flushpool()`^{143b} which you will see in the next section), the WORD instructions in the pool will be also processed by `dotext()` to get their `Instr.pcX` field set.

`addpool()` is used with operands representing large constants (`C_LCON`^{113a}), but also large offsets (`C_LOREG`^{114a}, `C_LAUTO`^{116c}), and also with symbols which when resolved are large offsets to R12 (`C_LEXT`^{115b}, `C_LACON`^{117d}).

```

⟨addpool() set t.to using a 142a⟩≡ (141f)
switch(c) {
    ⟨addpool() switch operand class c cases 142b⟩
}

```

```

⟨addpool() switch operand class c cases 142b⟩≡ (142a) 142c▷
// C_LCON|C_xCON, C_LEXT|C_xEXT? TODO warning if other case?
default:
    t.to = *a;
    break;

```

```

⟨addpool() switch operand class c cases 142c⟩+≡ (142a) ◁142b
case C_SROREG:
case C_LOREG:
case C_ROREG:
case C_FOREG:
case C_SOREG:
case C_FAUTO:
case C_SAUTO:
case C_LAUTO:
case C_LACON:
    t.to.type = D_CONST;
    t.to.offset = instoffset;
    break;

```

Uses C_FAUTO 190f, C_FOREG 191b, C_LACON 117d, C_LAUTO 116c, C_LOREG 114a, C_ROREG 114a, C_SAUTO 116c, C_SOREG 114a, C_SROREG 114a, and instoffset 112f.

There is no need to generate a new WORD instruction if the pool contains already the literal 51 needs to add:

```
<addpool() if literal already present in pool 143a>≡ (141f)
// find_list(t.to, blitrl)
for(q = blitrl; q != P; q = q->link)
    if(memcmp(&q->to, &t.to, sizeof(Adr)) == 0) {
        // for omvl()
        p->cond = q;
        return;
    }
```

Uses P 36c and blitrl 140.

9.8.3 flushpool()

```
<function flushpool(arm) 143b>≡ (286a)
/// (dotext -> checkpool) | dotext -> <>
void
flushpool(Prog *p, bool skip)
{
    Prog *q;

    if(blitrl) {
        <flushpool() if skip or corner case 144d>

        //insert_list_after_elt(blitrl, elitrl, p)
        elitrl->link = p->link;
        p->link = blitrl;

        blitrl = nil; /* BUG: should refer back to values until out-of-range */
        elitrl = nil;
        pool.size = 0;
        pool.start = 0;
    }
}
```

Uses blitrl 140, elitrl 141a, and pool-13 144a.

9.8.4 checkpool()

```
<dotext() pool handling, checkpool 143c>≡ (141d)
if(blitrl)
    checkpool(p);
```

Uses blitrl 140 and checkpool() 143d.

```
<function checkpool(arm) 143d>≡ (286a)
void
checkpool(Prog *p)
{
    if(p->link == P)
        flushpool(p, true);
    else
        <checkpool() if special condition 144c>
}
```

Uses P 36c and flushpool() 143b.

Another global, `pool`, is used to store additional information about the pool:

```
<global pool(arm) 144a>≡ (286a)
static struct {
    // PC of first instruction referencing the pool
    ulong start;
    // a multiple of 4
    ulong size;
} pool;
```

Uses `__anon_struct_2 144a`.

```
<addpool() set pool.start and pool.size 144b>≡ (141f)
// will be overwritten when dotext() layout the pool later
q->pc = pool.size;

if(blitrl == P) {
    pool.start = p->pc;
}
pool.size += 4;
```

Uses `P 36c`, `blitrl 140`, and `pool-13 144a`.

```
<checkpool() if special condition 144c>≡ (143d)
/*
 * When the first reference to the literal pool threatens
 * to go out of range of a 12-bit PC-relative offset,
 * drop the pool now, and branch round it.
 * This happens only in extended basic blocks that exceed 4k.
 */
if(pool.size >= 0xffc ||
    immaddr((p->pc+4) + 4 + pool.size - pool.start + 8) == 0)
    flushpool(p, true);
```

Uses `flushpool() 143b`, `immaddr() 115a`, and `pool-13 144a`.

```
<flushpool() if skip or corner case 144d>≡ (143b)
if(skip){
    DBG("note: flush literal pool at %lux: len=%lud ref=%lux\n",
        p->pc+4, pool.size, pool.start);
    q = prg();
    q->as = AB;
    q->to.type = D_BRANCH;
    q->cond = p->link;

    //insert_list(q, blitrl)
    q->link = blitrl;
    blitrl = q;
}
```

<flushpool() else if not skip and corner case 145b>

Uses `DBG 279`, `blitrl 140`, `pool-13 144a`, and `prg() 43a`.

9.8.5 LPOOL

```
<dotext() pool handling, switch flag cases 144e>+≡ (141d) <141e>
case LPOOL:
    if ((p->scond&C_SCOND) == COND_ALWAYS)
        flushpool(p, false);
    break;
```

Uses `LPOOL 141c` and `flushpool() 143b`.

```
<dotext() pool handling, flush if MOVW REGPC 145a>≡ (141d)
// MOVW ..., R15 => flush
if(p->as==AMOVW && p->to.type==D_REG && p->to.reg==REGPC &&
    (p->scond&C_SCOND) == COND_ALWAYS)
    flushpool(p, false);
```

Uses flushpool() 143b.

```
<flushpool() else if not skip and corner case 145b>≡ (144d)
else if((p->pc + pool.size - pool.start) < 2048)
    return;
```

Uses pool-13 144a.

Chapter 10

Debugging Support

The Plan 9 debugger `db`, which I will describe in the `DEBUGGER` book [Pad16c], has access to lots of *metadata* in the executable to help debug programs. For instance, here is a simplified output of `db` when debugging a C program:

```
$ db hello
...
main(argv=...) /usr/.../main.c
    called from _main+26 (/sys/.../main9.s:12)
...
```

`db` knows from which *file* and which *line* a piece of code comes from. It also knows the name of the *function* containing this piece of code, the *parameters* of this function (names and values), as well as the name of the *caller* to this function. Finally it knows also the file and line of this caller function.

The metadata in the executable comes from corresponding metadata in the object files generated by 5a and 5c. Indeed, the names of the parameters, locals, and entities are kept in the *object file symbol table* which you have seen in Section 5.3.4 (and in the `ASSEMBLER` book [Pad15a]). Each entry of this table uses the special opcode `ANAME`. In the same way, the object file contains also a *file/line table* which uses the special opcode `AHISTORY` (see the `ASSEMBLER` book [Pad15a]). In this chapter, I will show how two corresponding tables are also stored in the executable: the *executable symbol table* and the *program counter and line table*.

10.1 Debug tables vs DWARF

The modern mainstream standard for debug information on UNIX systems is DWARF. DWARF was introduced in 1988 for UNIX System V Release 4; it went through several revisions (DWARF 2 in 1993, DWARF 5 in 2017) and is now roughly a thousand pages of specification. It encodes, for each compiled program, the full tree of types, scopes, variables, functions, inlined call sites, macros, source-file and line-number tables, call-frame unwind information, and much else. `gdb`, `lldb`, and every other modern UNIX debugger depend on DWARF for essentially all of their functionality, and the debug information for a typical C++ program compiled with `-g` is often larger than the program itself.

Plan 9 takes the opposite approach. The debug information 51 emits into the executable is just two small tables: a symbol table with one entry per function, global, or static (name, value, kind), and a compact PC-to-line table encoded as delta bytes on top of a base line number. There is no type information, no variable-location expressions, no call-frame unwind tables, no inlined-frame records. The symbol table alone is enough for `db` and `acid` to report which function the PC is in, what file and line it came from, and—with the help of the standard frame layout the compiler and assembler both agree on—to walk back up the call chain. The entire metadata infrastructure across the compiler, assembler, and linker is well under a thousand lines.

The trade-off is real. `acid` cannot display the type and value of a local variable the way `gdb` can, because the type is not in the executable; `5c` records just enough in the object file for `acid`'s built-in `complex` declarations to work, but nowhere near what DWARF's `DW_AT_location` expressions encode for optimized code. On the other hand, the entire debugging workflow survives compiler changes without needing new debugger support, and the resulting binaries are much smaller—a kilobyte-scale symbol table instead of a section that can easily dwarf the code. Plan 9 pays in power for what it saves in complexity, and the two small tables stay tractable enough to be described and generated in a single chapter.

10.2 `asmb()` and the debugging tables

As explained in Section 4.4, the generation of the executable is done by `asmb()`^{45a}. Here is the part of `asmb()` which generates the debugging sections:

```

<asmb() symbol and line table sections 147a>≡ (45a)
// modified by asmsym()
symsize = 0;
// modified by asmlc()
lcsize = 0;

if(!debug['s']) {
    switch(HEADTYPE) {
        <asmb() switch HEADTYPE (for symbol table generation) cases(arm) 147b>
    }
    DBG("%5.2f sym\n", cputime());
    asmsym();
    DBG("%5.2f pc\n", cputime());
    asmlc();

    <asmb() if dynamic module, call asmdyn() 177e>
    cflush();
}
else {
    <asmb() if dynamic module and no symbol table generation 177a>
}

```

Uses `DBG` 279, `HEADTYPE` 40a, `asmlc()` 160c, `asmsym()` 148b, `cflush()` 110a, `debug` 218a, `lcsize` 160a, and `symsize` 147c.

Note that you can *strip* the executable of debugging information by using the `5l -s` option. You can also use instead later the `strip` command (described in Appendix E.5) on the executable.

You saw in Section 2.5 that the debugging tables are stored after the data section in an `a.out` executable, hence the `seek()` below:

```

<asmb() switch HEADTYPE (for symbol table generation) cases(arm) 147b>≡ (147a) 188b▷
case H_PLAN9:
    OFFSET = HEADR+textsize+datsize;
    seek(cout, OFFSET, SEEK__START);
    break;

```

Uses `HEADR` 40d, `H_PLAN9` 150a, `cout` 38d, `datsize` 92a, and `textsize` 94a.

The next two sections will describe the generation of the executable symbol table, via `asmsym()`^{148b}, and the program counter and line table, via `asmlc()`^{160c}.

10.3 Executable symbol table

One side effect of `asmsym()`^{148b} is to modify the global `symsize` below, which contains the size of the symbol table. The value in this global will then be stored in the `a.out` header (see Section 4.4.1).

0	4	5	10	11
+-----+				
1020	'T'	"_main"	'\0'	
+-----+				
value	type	name	end	
		(variable size)	marker	

Figure 10.1: Executable symbol table entry format.

<global symsize 147c>≡ (283b)
long symsize;

10.3.1 Symbol table format: putsymb()

In Section 5.3.4, you saw the rather subtle format of the object file symbol table: a spread and circular array on disk. You saw also before the symbol table `hash31a`, which resides in memory, and which uses a hash table data structure. Both tables were used for resolving symbols. The format of the executable symbol table is very simple instead. It is just a list of *symbol descriptions* separated by `'\0'`. The format of a symbol description as well as an example is shown in Figure 10.1. The purpose of the executable symbol table is mostly to help debuggers to display names to the programmer instead of memory addresses.

`putsymb()` generates a new symbol description in the executable, e.g., with `putsymb("_main", 'T', 0x1020, 0)`.

<function putsymb 148a>≡ (292b)
void
putsymb(char *s, int t, long v, int ver)
{
int i, f;

<putsymb() adjust string s if file symbol 159b>

// value
lput(v);
// type
if(ver)
t += 'a' - 'A'; // lowercase(t)
cput(t+0x80); /* 0x80 is variable length */

<putsymb() if z or Z 159d>
else {
// name
for(i=0; s[i]; i++)
cput(s[i]);
// end marker
cput('\0');
}
symsize += 4 + 1 + i + 1;

<putsymb() debug 227b>
}

Uses `cput()` 234c, `lput()` 110b, and `symsize` 147c.

10.3.2 Globals and procedures symbols: `asmsym()`

I can now show the code of `asmsym()` which first calls `putsymb()`^{148a} for all the data symbols in `hash`^{31a}:

```
<function asmsym(arm) 148b>≡ (292b)
  /// main -> asmb -> <>
  void
  asmsym(void)
  {
    Sym *s;
    int h;
    Prog *p;
    <asmsym() other locals 152d>

    <asmsym() generate symbol for etext 149>

    // data symbols
    for(h=0; h<NHASH; h++)
      for(s=hash[h]; s!=S; s=s->link)
        switch(s->type) {
          case SDATA:
            putsymb(s->name, 'D', s->value+INITDAT, s->version);
            continue;
          case SBSS:
            putsymb(s->name, 'B', s->value+INITDAT, s->version);
            continue;
          <asmsym() in symbol table iteration, switch section cases 159a>
        }

    // procedure symbols
    for(p=textp; p!=P; p=p->cond) {
      s = p->from.sym;
      if(s->type == STEXT) {
        /* filenames first */
        <asmsym() call putsymb for filenames 159c>

        if(p->mark & LEAF)
          putsymb(s->name, 'L', s->value, s->version);
        else
          putsymb(s->name, 'T', s->value, s->version);

        // local symbols
        <asmsym() frame symbols 152e>
      }
    }
    if(debug['v'] || debug['n']) {
      Bprint(&bso, "symsize = %lud\n", symsize);
      Bflush(&bso);
    }
  }
}
```

Uses `INITDAT` 40g, `LEAF` 279, `NHASH`, `P` 36c, `S` 33b, `SBSS` 170b, `SDATA`, `STEXT` 156d, `bso` 218c, `debug` 218a, `putsymb()` 148a, `symsize` 147c, and `textp` 152a.

`asmsym()`^{148b} could generate the entries for the procedures by also using `hash`. Instead, it relies on `textp`^{152a} (which I will describe soon) which contains the list of all procedures. This list is a subset of the list of instructions in `firstp`^{36a}. The reason for iterating over instructions instead of the symbol table for procedure symbols is because the leaf information `p->mark` (see Section 7.3.1) is in the `TEXT` instruction of the procedure, not its symbol. Moreover, information about the parameters and local variables of a procedure are also stored in the `TEXT` instruction as you will see in the next section.

`etext` is defined via `xdefine()`^{97a} in Section 7.6 and is part of the text section (`STEXT`^{156d}). It is not a real procedure though so it would not be found in `textp`, hence the special code below:

```
<asmsym() generate symbol for etext 149>≡ (148b)
```

```
s = lookup("etext", 0);
if(s->type == STEXT)
    putsymb(s->name, 'T', s->value, s->version);
```

Uses `STEXT` 156d, `lookup()` 32a, and `putsymb()` 148a.

10.3.3 Stack variables symbols

In addition to global symbols, the executable symbol table contains also information about the local variables of a procedure and its parameters, that is stack variables. This is really useful for displaying *stack traces* in a debugger. This is also why assembly programmers write code like `MOVW count+4(FP), R1` (or why 5c generates such assembly code) even though `count` is not used to generate any machine code; `count` will be present in the executable symbol table and used by the debugger to name stack variables.

Before showing the code of `asmsym()`^{148b} which generates entries for those stack variables, I need to explain first the code which keeps track of those variables when loading the object files in `ldobj()`⁵⁵.

Auto

The `Auto` structure below is used to keep track of all the parameters and locals accessed by a procedure:

```
<struct Auto(arm) 150a>≡ (279)
```

```
struct Auto
{
    // enum<Sym_kind> (N_LOCAL, N_PARAM, or N_FILE/N_LINE)
    short   type;

    // <ref<Sym>>
    Sym*    asym;
    long    aoffset;

    // Extra
    <Auto extra fields 150b>
};
```

For instance, `p+4(FP)` will be represented by:

```
// p+4(FP)
{ .type = N_PARAM; // (FP)
  .asym = &<p Sym>; // p
  .aoffset = 4;    // +4
}
```

Those stack variables are chained together:

```
<Auto extra fields 150b>≡ (150a)
```

```
// list<ref<Auto> (head = curauto or Instr.to.autom of TEXT instruction)
Auto*  link;
```

The head of the list is stored in one of the field of one of the operand of the `TEXT` instruction:

```
<Adr other fields 150c>+≡ (33c) <106b
```

```
// list<ref_own<Auto> (next = Auto.link), only used by TEXT instruction
Auto*  autom;
```

curauto

The set of stack variables for the current procedure (`curtext36f`) is stored in `curauto` and updated at loading time by `inopd()57b` (which is called by `ldobj()55`):

```
<global curauto 150d>≡ (283b)
// list<ref<Auto>> (next = Auto.link)
Auto* curauto;
```

```
<inopd() other locals 151a>≡ (57b)
Sym *s;
// <enum<Sym_kind>>
int t;
int l;
Auto *u;
```

```
<inopd() adjust curauto for N_LOCAL or N_PARAM symkind 151b>≡ (57b)
s = a->sym;
t = a->symkind;
l = a->offset;
```

```
// a parameter or local with a symbol, e.g., p+4(FP)
if(s != S && (t == N_LOCAL || t == N_PARAM)) {
```

```
    <inopd() return if stack variable already present in curauto 151c>
    // else
```

```
    u = malloc(sizeof(Auto));
    u->asym = s;
    u->type = t;
    u->aoffset = l;
```

```
    //add_list(u, curauto)
    u->link = curauto;
    curauto = u;
```

```
}
```

Uses S 33b, `curauto 150d`, and `malloc() 233a`.

```
<inopd() return if stack variable already present in curauto 151c>≡ (151b)
for(u=curauto; u; u=u->link)
    if(u->asym == s)
        if(u->type == t) {
            if(u->aoffset > l)
                u->aoffset = l; // diag()? inconsistent offset?
            return size;
        }
```

Uses `curauto 150d`.

Once the code of a procedure has been fully read, `curauto` can be transferred to the `Instr.to.autom` field of the `TEXT` instruction of the current procedure:

```
<ldobj() case ATEXT, if curtext not null adjustments for curauto 151d>≡ (65b)
if(curtext != P) {
    <ldobj() case ATEXT, curauto adjustments with curhist 155c>
    curtext->to.autom = curauto;
    curauto = nil;
}
```

Uses P 36c, `curauto 150d`, and `curtext 36f`.

`<ldebug() case AEND, curauto adjustments 151e>≡` (67b)

```
if(curtext != P)
    curtext->to.autom = curauto;
curauto = nil;
```

Uses P 36c, curauto 150d, and curtext 36f.

Procedure declarations, textp/etextp

I can now show the pair of globals that keeps track of the list of procedures (the TEXT pseudo-instructions):

`<global textp 152a>≡` (283b)

```
// list<ref<Prog>> (next = Prog.cond)
Prog* textp = P;
```

Uses P 36c and textp 152a.

`<global etextp 152b>≡` (283b)

```
// ref<Prog> (end from = textp)
Prog* etextp = P;
```

Uses P 36c and etextp 152b.

The procedure instructions are chained together by (ab)using `Instr.condX`:

`<ldebug() in switch opcode ATEXT case, populate textp 152c>≡` (65b)

```
//add_queue(textp, etextp, p)
if(textp == P) {
    textp = p;
    etextp = p;
} else {
    etextp->cond = p;
    etextp = p;
}
```

Uses P 36c, etextp 152b, and textp 152a.

Note that the instructions in the list `textp/etextp` are a subset of the list of instructions in `firstp/lastp`.

Frame symbols

I can now show the code that leverages `Auto`²⁷⁹ to generate the symbols for the stack variables of a procedure:

`<asmsym() other locals 152d>≡` (148b)

```
Auto *a;
```

`<asmsym() frame symbols 152e>≡` (148b)

```
/* frame, auto and param after */
putsymb(".frame", 'm', p->to.offset+4, 0);
for(a=p->to.autom; a; a=a->link)
    if(a->type == N_LOCAL)
        putsymb(a->asym->name, 'a', -a->aoffset, 0);
    else
        if(a->type == N_PARAM)
            putsymb(a->asym->name, 'p', a->aoffset, 0);
```

Uses `putsymb()` 148a.

I refer you to the `DEBUGGER` book [Pad16c] to see how those symbols are used to improve the debugging experience. Note that the `.frame` symbol allows to know the size in the stack necessary to hold the local variables used by the procedure as well as the return address to the caller (the +4 above). This part of the stack is also called the *frame* of a procedure. Thanks to `.frame`, the debugger can go up the stack and identify the different frames of the different procedures in the call stack.

10.3.4 Filename and line origin symbols

The executable symbol table contains also information about the filenames of the assembly or C sources where the object files come from. Those filenames are encoded using a compact scheme I will describe in Section 10.4.3. The symbol table also stores information about the `#include` and `#line` directives used in those source files and which are kept in the object files. Those directives help to keep track of the file and line origin of any piece of machine code.

10.4 File and line information

Before seeing the code of `asm1c()`^{160c}, I need first to recall how file and line information are stored in object files, and to explain the code of `ldobj()`⁵⁵ which loads this information in memory.

10.4.1 Locations in objects: AHISTORY

In the ASSEMBLER book [Pad15a], I show that some file and line information are stored in the memory of 5a, in a list of `Hist` structures. Each element of this list represented any one of the original source filename, a `#include`, or a `#line` directive, and contained the following three elements:

- A filename (or nil to indicate the end of a `#include`)
- A *global line number* representing a line number after preprocessing
- A *local line number*, which was either 0 for the original source and `#include` directives, or a positive integer for `#line` directives (or -1 for `#pragma lib` directives, see Section 6.4)

Thanks to this list, 5l can then easily convert the global line number assigned to each instruction in the object file in `Instr.lineX` to a pair of (source) filename and (local) line information. Figure 10.2 illustrates the evolution of the global line number on the content of `/tests/cpp/foo.s` (which includes other files) as well as the list of `Hists` after having pre-processed the entire file.

I also show in the ASSEMBLER book [Pad15a] how the list of `Hist` structures is stored at the beginning of the object file in instructions using the special opcode `AHISTORY`. In fact, each `AHISTORY` instruction records just the global line number (in `Instr.lineX`) and local line number (in `Instr.to.offset`) of an `Hist`. The filename of the `Hist` is encoded in a series of `ANAME` preceding the `AHISTORY` instruction. Each `ANAME` encodes a *path element* of the filename. For instance, here is the series of instructions in the object file which encode the directive `#line 10 "/usr/foobar.c"` of Figure 10.2:

```
ANAME <usr
ANAME <foobar.c
AHISTORY 17 10
```

You will see later why each path element is prefixed by a `<` and why a filename is split in multiple path elements.

10.4.2 Locations in 5l memory

5a uses the function `outhist()` to write the `Hists` in the object file by using `AHISTORY` and `ANAME` instructions. I will now show the code of 5l that reads those `AHISTORY` and `ANAME` instructions in the different object files via `ldobj()`.

global line number	foo.s and included files	list of Hist structures
	foo.s-----+	+-----+ "foo.s", G1, L0
1	L1	+-----+
2	L2	
3	L3 #include "foo.h"	+-----+
	foo.h-----+	"foo.h", G4, L0
4	L1 #include "foo1.h"	+-----+
	foo1.h-----+	"foo1.h", G5, L0
5	L1	+-----+
	+-----+	nil, G6, L0
6	L2	+-----+
7	L3 #include "foo2.h"	+-----+
	foo2.h-----+	"foo2.h", G8, L0
8	L1	+-----+
	+-----+	nil, G9, L0
9	L4	+-----+
10	L5	+-----+
	+-----+	nil, G11, L0
11	L4	+-----+
12	L5 #include "bar.s"	+-----+
	bar.s-----+	"bar.s", G13, L0
13	L1	+-----+
14	L2	+-----+
	+-----+	nil, G15, L0
15	L6	+-----+
16	L7	+-----+
17	L8#line 10 "foobar.c"	"foobar.c", G18, L10
18	L9	+-----+
19	L10	+-----+
	+-----+	nil, G20, L0
		+-----+

Figure 10.2: File and line information for /tests/cpp/foo.s.

AHISTORY and addhist

Reading an AHISTORY in `ldobj()`⁵⁵ is pretty simple because it is using the same format than regular instructions. We can just write a new case in the opcode switch of `ldobj()` and access information from the read instruction `p`:

```
<ldobj() switch opcode cases(arm) 153>+≡ (55) <67c 182b>
case AHISTORY:
  <ldobj() in AHISTORY case, if pragma lib 75c>
  // else

  // the global line
  addhist(p->line, N_FILE); /* 'z' */
  // the local line (if needed for #line)
  if(p->to.offset)
    addhist(p->to.offset, N_LINE); /* 'Z' */
  <ldobj() in AHISTORY case, end of case, reset histfrogp 157e>
  goto loop;
```

Uses `addhist()` 155a.

As you will see in the rest of this chapter, one AHISTORY instruction in the object file will become eventually two symbols in the executable symbol table (if the local line number is non zero), hence the two calls to `addhist()`^{155a} above (and the 'z' and 'Z' comments). Indeed, `addhist()` below creates a new symbol, which I call an *Hist symbol* from now on, and a new `Auto`²⁷⁹ which is then stored in the global `curhist`^{155b}. As you will see soon, `curhist` is then copied in `curauto`^{150d} which is then assigned to one of the field of the first TEXT instruction of an object file. Ultimately, the *Hist* symbols created by `addhist()` will be stored in the executable symbol table.

```
<function addhist 155a>≡ (292c)
void
addhist(long line, int type)
{
  Auto *u;
  Sym *s;
  <addhist() other locals 157g>

  s = malloc(sizeof(Sym));

  u = malloc(sizeof(Auto));
  u->asym = s;
  u->type = type;
  u->aoffset = line;

  //add_list(u, curhist)
  u->link = curhist;
  curhist = u;

  <addhist() set symbol name to filename using compact encoding 157h>
}
```

Uses `curhist` 155b and `malloc()` 233a.

The second parameter of `addhist()` above can be either `N_FILE` or `N_LINE`, which are two new symbol kinds (see Section 3.3) used here respectively to represent a global line number and a local line number (if there is one). The reason for using two *Hist* symbols for one AHISTORY is because a symbol description in the executable symbol table can contain only one integer value (see Section 10.3.1), but an AHISTORY carry two line numbers.

The code which sets the name of an *Hist* symbol will be described soon; it is using a subtle compact encoding. The next section will describe `curhist`.

curhist and curauto

`curhist` below is used to accumulate the list of `Auto`²⁷⁹ created by `addhist()`^{155a}. This list derives from the `AHISTORY` instructions. `curhist` is similar to `curauto`^{150d} which you saw before.

```
<global curhist 155b>≡ (283b)
Auto* curhist;
```

In fact, the elements in `curhist` are gradually transferred to `curauto`:

```
<ldobj() case ATEXT, curauto adjustments with curhist 155c>≡ (151d)
histtoauto();
```

Uses `histtoauto()` 156b.

```
<ldobj() case AEND, curauto adjustments with curhist 156a>≡ (67b)
histtoauto();
```

Uses `histtoauto()` 156b.

```
<function histtoauto 156b>≡ (292c)
```

```
/// ldobj (case AEND | ATEXT) -> <>
void
histtoauto(void)
{
    Auto *l;

    // append_list(curhist, curauto); curhist = nil;
    while(l = curhist) {
        curhist = l->link;

        l->link = curauto;
        curauto = l;
    }
}
```

Uses `curauto` 150d and `curhist` 155b.

In 5a, `Hists` are stored in memory in a single global (`hist`) because 5a deals with only one assembly file at a time. Global line numbers are unique there. The `Hists` are then stored at the beginning of the generated object file. Because 5l deals with many object files, global line numbers are not unique anymore. So, the `Hists`, which became `Autos`, are spread in the first `TEXT` instructions of every input object files. Eventually, as you will see in Section 10.4.3, the `Hist` symbols will be also spread in the executable symbol table next to the symbols of the first procedures of every input object files.

ANAME and path elements

As said earlier, each `AHISTORY` instruction in the object file is preceded by a series of `ANAMES` which each encodes a path element of the `Hist`'s filename. The code in `ldobj()`⁵⁵ dealing with `ANAMES` first lookups in `hash`^{31a} the symbol `s` introduced by this `ANAME` (see Section 5.3.4) and then executes the following code:

```
<ldobj() when ANAME opcode, if N_FILE 156c>≡ (61d)
if(k == N_FILE) {
    if(s->type != SFILE) {
        s->type = SFILE;
        histgen++;
        s->value = histgen;
    }
    <ldobj() when ANAME opcode, if N_FILE, update histfrogp 157c>
    <ldobj() when ANAME opcode, if N_FILE, if no more space in histfrog 158a>
}
```

Uses `SFILE` and `histgen` 156e.

Note that because path elements are prefixed with a `<`, e.g., `<usr`, their symbols can not conflict with the symbols used for globals or procedures, e.g., `foo`. A new section is used for those new kind of symbols:

```
<Section cases 156d>≡ (33b) 170b▷
SFILE,
```

Note also that if another `ANAME` instruction contains the same path element, they will share the same symbol. That way, if filenames of different `Hists` have common roots, they will share their common parts. In fact, as shown by the code above, each path element symbol is assigned a unique integer, `histgen`, stored in `Sym.valueX`, which will act as you will see soon as an *index*.

```
<global histgen 156e>≡ (283b)
int histgen = 0;
Uses histgen 156e.
```

Full filenames and histfrog

The series of path element symbols corresponding to the series of `ANAME`s preceding an `AHISTORY` are then accumulated in the following global array:

```
<global histfrog 157a>≡ (283b)
Sym* histfrog[MAXHIST];
Uses MAXHIST 203i.
```

```
<constant MAXHIST 157b>≡ (278c)
MAXHIST = 20, /* limit of path elements for history symbols */
```

```
<lobj() when ANAME opcode, if N_FILE, update histfrogp 157c>≡ (156c)
if(histfrogp < MAXHIST) {
    histfrog[histfrogp] = s;
    histfrogp++;
}
```

Uses `MAXHIST 203i`, `histfrog 157a`, and `histfrogp 157d`.

```
<global histfrogp 157d>≡ (283b)
int histfrogp;
```

The array is reseted after each processed `AHISTORY`:

```
<lobj() in AHISTORY case, end of case, reset histfrogp 157e>≡ (153)
histfrogp = 0;
```

Uses `histfrogp 157d`.

```
<lobj() after newloop when new object file, more initializations 157f>+≡ (55) <64b 187b▷
histfrogp = 0;
```

Uses `histfrogp 157d`.

I can now finally show how are encoded the names of `Hist` symbols. `51` is using a compact encoding. Instead, of storing the full string of the full filename, the symbol name is made of a series of 16 bits integers representing each a path element. Each integer corresponds to the index `histgen`^{156e} stored in `Sym.valueX` of the corresponding path element symbol:

```
<addhist() other locals 157g>≡ (155a)
int i, j, k;
```

```

<addhist() set symbol name to filename using compact encoding 157h)≡ (155a)
    s->name = malloc(2*(histfrogp+1) + 1);
    j = 1;
    for(i=0; i<histfrogp; i++) {
        k = histfrog[i]->value;
        s->name[j+0] = k>>8;
        s->name[j+1] = k;
        j += 2;
    }

```

Uses `histfrog` 157a, `histfrog` 157d, and `malloc()` 233a.

Symbol descriptions are usually separated by a `'\0'` (see Section 10.3.1), but the 16 bits corresponding to an `histgen` can contain a null character. Indeed, an `histgen` can be less than 256 or a multiple of 256. To avoid ambiguities, the first byte of an `Hist` symbol is the null character and the name ends with a double null character, hence `malloc(2*(histfrogp+1) + 1)` in the code above. Note also that `histgen` starts at 1, which avoids any ambiguity with the double ending null characters.

For instance, the start of the object file `foo.5` resulting from the assembling of `/usr/foo.s` in Figure 10.2 could be:

```

ANAME <usr
ANAME <foo.s
AHISTORY 1 0
ANAME <usr
ANAME <foo.h
AHISTORY 4 0

```

Given this object file, here is the list of corresponding symbols created by 51 and their values:

```

{ .name = "<usr", .value = 1; } // ANAME <usr
{ .name = "<foo.s", .value = 2; } // ANAME <foo.s
{ .name = "\0\0\1\0\2\0\0", .value = 1; } // AHISTORY 1 0, /usr/foo.s
{ .name = "<foo.h", .value = 3; } // ANAME foo.h
{ .name = "\0\0\1\0\3\0\0", .value = 4; } // AHISTORY 4 0, /usr/foo.h

```

```
collapsefrog()
```

51 limits filenames to 20 path elements (see `MAXHIST`²⁰³ⁱ). A longer filename will be truncated. Nevertheless, some path elements such as `..` or `.` can be resolved, which allows to extend partially the limit thanks to the code below:

```

<lobj() when ANAME opcode, if N_FILE, if no more space in histfrog 158a)≡ (156c)
    else
        collapsefrog(s);

```

Uses `collapsefrog()` 158b.

Remember that `<` is the first character in all path elements, hence the many `+1` in the code below:

```

<function collapsefrog 158b)≡ (290)
    static void
    collapsefrog(Sym *s)
    {
        int i;

        /*
         * bad encoding of path components only allows
         * MAXHIST components. if there is an overflow,

```

```

    * first try to collapse xxx/..
    */
for(i=1; i<histfrog; i++)
    if(strcmp(histfrog[i]->name+1, "..") == 0) {
        memmove(histfrog+i-1, histfrog+i+1,
            (histfrog-i-1)*sizeof(histfrog[0]));
        histfrogp--;
        goto out;
    }

/*
 * next try to collapse .
 */
for(i=0; i<histfrog; i++)
    if(strcmp(histfrog[i]->name+1, ".") == 0) {
        memmove(histfrog+i, histfrog+i+1,
            (histfrog-i-1)*sizeof(histfrog[0]));
        goto out;
    }

/*
 * last chance, just truncate from front
 */
memmove(histfrog+0, histfrog+1,
    (histfrogp-1)*sizeof(histfrog[0]));

out:
    histfrog[histfrogp-1] = s;
}

```

Uses `histfrog` 157a and `histfrogp` 157d.

10.4.3 Locations in executables

Now that the file and line information has been loaded in memory (in the symbol table, in some `Auto`²⁷⁹s of some procedures, and in the `Instr.lineX` of every instructions) I can show how this information is stored in the executable. Just like for object files, location information is split in two parts: one part corresponding to the `Hists` is stored in the executable symbol table, while the other part corresponding to the global line numbers of every instructions is stored in a new *program counter and line table*.

Path element and Hist symbols

Because path element symbols are shared by all object files, they can be stored at the beginning of the executable symbol table, with the symbols for globals:

```

⟨asmsym() in symbol table iteration, switch section cases 159a⟩≡ (148b) 215b▷
    case SFILE:
        putsymb(s->name, 'f', s->value, s->version);
        continue;

```

Uses `SFILE` and `putsymb()` 148a.

Remember that the value of path elements symbols is an `histgen`^{156e}, an identifier referenced in the name of `Hist` symbols. The `<` prefix in path elements can be skipped:

```

⟨putsymb() adjust string s if file symbol 159b⟩≡ (148a)
    if(t == 'f')
        s++;

```

Hist symbols are stored in the `Instr.to.autom` field of the first procedure of every object files. They are written just before the symbol of the procedure (see Section 10.3.2) in the executable symbol table:

```
<asmsym() call putsymb for filenames 159c>≡ (148b)
for(a=p->to.autom; a; a=a->link)
    if(a->type == N_FILE)
        putsymb(a->asym->name, 'z', a->aoffset, 0);
    else
        if(a->type == N_LINE)
            putsymb(a->asym->name, 'Z', a->aoffset, 0);
```

Uses `putsymb()` 148a.

The character type of an Hist symbol is a 'z' to remind that the name is using a compression scheme.

```
<putsymb() if z or Z 159d>≡ (148a)
if(t == 'z' || t == 'Z') {
    cput(s[0]);
    for(i=1; s[i] != '\0' || s[i+1] != '\0'; i += 2) {
        cput(s[i]);
        cput(s[i+1]);
    }
    cput('\0');
    cput('\0');
    i++;
}
```

Uses `cput()` 234c.

Note that there is only one `i++` above. The code of `putsymb()`^{148a} (see Section 10.3.1) uses `i` to adjust `symsize`^{147c} but it does add 1 to `i` to account for the very last null character.

Program counter line table format

The naive encoding of “which source line did each instruction come from” would store one 32-bit line number per 4-byte instruction, doubling the size of the executable. `asmlc` reduces this to roughly one byte per instruction by storing *deltas* since the previous entry, exploiting the fact that adjacent instructions usually advance both PC and line by small amounts.

The encoding partitions the byte 0–255 into ranges:

```
0           : escape, next 4 bytes are a signed
              line-number delta (for big jumps and
              cross-file transitions)
1..64       : line += this value, pc += MINLC (4)
65..128     : line -= (this - 64), pc += MINLC
129..255    : pc += (this - 128) * MINLC, no line change
```

A line of source that compiled to two ARM instructions takes two bytes: one in the 129..255 range (advance pc by 4 with no line change) and one in the 1..64 range (advance both). A jump backwards by 14 lines becomes 0x4F (65 + 14, the “minus 14” bucket). The escape 0x00 followed by 4 bytes handles cross-file transitions where the line number can swing arbitrarily.

One side effect of `asmlc()`^{160c} is to modify the global `lcsz` below, which contains the size of the line table. The value in this global will then be stored in the `a.out` header (see Section 4.4.1).

```
<global lcsz 160a>≡ (283b)
long lcsz;
```

`asmlc()`

```
<constant MINLC(arm) 160b>≡ (292b)
#define MINLC 4
```

<function asmlc 160c>≡

(292b)

```
void
asmlc(void)
{
    long oldpc, oldlc;
    Prog *p;
    long v;
    long s;

    oldpc = INITTEXT;
    oldlc = 0;
    for(p = firstp; p != P; p = p->link) {
        if(p->line == oldlc || p->as == ATEXT || p->as == ANOP) {
            <adjust curtext when iterate over instructions p 36h>
            <asmlc() dump instruction p, debug 227d>
            continue;
        }
        // else
        <asmlc() dump lcsiz, debug 227e>
        v = (p->pc - oldpc) / MINLC;
        while(v) {
            s = (v < 127)? v : 127; // min(), but impossible more than 6 (o6)
            cput(s+128); /* 129-255 +pc */
            <asmlc() dump s, debug 227f>
            v -= s;
            lcsiz++;
        }
        s = p->line - oldlc;
        oldlc = p->line;
        oldpc = p->pc + MINLC;

        if(s > 64 || s < -64) {
            cput(0); /* 0 vv +lc */
            cput(s>>24);
            cput(s>>16);
            cput(s>>8);
            cput(s);
            <asmlc() dump big line change, debug 228a>
            lcsiz += 5;
        } else {
            if(s > 0) {
                cput(0+s); /* 1-64 +lc */
                <asmlc() dump small line increment, debug 228b>
            } else {
                cput(64-s); /* 65-128 -lc */
                <asmlc() dump negative line increment, debug 228c>
            }
            lcsiz++;
        }
    }
    // padding
    while(lcsiz & 1) {
        s = 129;
        cput(s);
        lcsiz++;
    }
    if(debug['v'] || debug['V'])
        Bprint(&bso, "lcsiz = %ld\n", lcsiz);
    Bflush(&bso);
}
```

Uses INITTEXT 40e, MINLC-12 160b, P 36c, bso 218c, cput() 234c, debug 218a, firstp 36a, and lcsiz 160a.

Chapter 11

Profiling Support

In many operating systems, profiling is enabled by a flag of the compiler, e.g., `gcc -p`. Surprisingly, in Plan 9 profiling support is enabled instead by a flag of the linker: `5l -p`. By *instrumenting* the instructions `TEXT` and `RET` in different ways, `5l` can provide easily different sorts of profiling which you will see in this chapter.

11.1 `5l -p` and `_mainp`

The effect of the use of the `-p` flag is twofold. First, it changes the entry point of the program from `_main` to `_mainp`:

```
<main() adjust INITENTRY if profiling 162a>≡ (42c)
    if(debug['p'])
        INITENTRY = "_mainp";
```

`_mainp` is a procedure written in assembly in `lib_core/libc/arm/main9p.s`. The main difference with `_main` is the call to `_profmain()` defined in `lib_core/libc/port/profile.c` which initializes profiling data. See the `PROFILER` book [Pad26] for more information.

The second effect of `-p` is the execution of `doprof1()`¹⁶³ or `doprof2()`^{166a} which perform different kinds of profiling:

```
<main() call doprofxxx() if profiling 162b>≡ (79)
    if(debug['p'])
        if(debug['1'])
            doprof1();
        else
            doprof2();
```

Those functions are called after `patch()`⁸² and before `noops()`⁸⁷ (see Chapter 7). Thanks to `patch()` and the graph of code instructions (see Section 7.2), you can easily instrument a program in a similar way to `noops()`. The idea is to insert new profiling instructions *after* the pseudo instruction `TEXT`, and new profiling instructions *before* the virtual instruction `RET`.

11.2 `5l -p -1` and `__mcount`

The first profiling strategy I will show counts *the number of times a function is called*. It is enabled by `5l -p -1`. Note that it does not count the *time spent* in those functions, which is another strategy you will see in the next section. Nevertheless, even if the function calls count is a rudimentary information, it can be already useful for improving the performance of a program. In fact, it can be also very useful to find bugs in a program. Indeed, unexpected statistics can be good leads for fixing code.

The main idea of `doprof1()`¹⁶³ below is to use a global array `__mcount` in which each entry corresponds to a function. Each entry uses 8 bytes: the first 4 bytes contain the address of the function, which is a simple way

to identify a function, and the last 4 bytes the function calls count. Each call to a function will then increment an element in `__mcount` at runtime.

Here is an example of how the code is instrumented by `5l -p -1` for a program with 2 procedures `foo` and `bar`:

```
TEXT foo(SB), $0 -> TEXT foo(SB), $0
                    MOVW __mcount+8(SB), R11
...                ADD $1, R11
                    MOVW R11, __mcount+8(SB)
                    ...

RET                -> RET

TEXT bar(SB), $0 -> TEXT bar(SB), $0
                    MOVW __mcount+16(SB), R11
...                ADD $1, R11
                    MOVW R11, __mcount+16(SB)
                    ...

RET                -> RET
```

```
// first entry contain #words (32 bits) used by mcount: 4 + 1
DATA __mcount+0(SB)/4, $5
// start of info about foo
DATA __mcount+4(SB)/4, $foo(SB)
// start of info about bar
DATA __mcount+12(SB)/4, $bar(SB)
```

The code of `doprof1()` is pretty straightforward:

```
<function doprof1(arm) 163>≡ (292a)
void
doprof1(void)
{
    Sym *s;
    Prog *p, *q;
    long n;

    DBG("%5.2f profile 1\n", cputime());

    s = lookup("__mcount", 0);
    n = 1;
    // start from firstp->link, not firstp so skip first instruction/procedure
    // (usually _main?) as SB might not have been set yet
    for(p = firstp->link; p != P; p = p->link) {
        if(p->as == ATEXT) {

            // DATA __mcount+n*4(SB)/4, $foo(SB) //$
            q = prg();
            q->line = p->line;
            q->as = ADATA;
            q->from.type = D_OREG;
            q->from.symkind = N_EXTERN;
            q->from.offset = n*4;
            q->from.sym = s;
```

```

q->reg = 4; // size of this DATA slice
q->to = p->from;
q->to.type = D_ADDR;

// add_list(q, datap)
q->link = datap;
datap = q;

// MOVW __mcount+ n*4+4(SB), R11
q = prg();
q->line = p->line;
q->pc = p->pc;
q->as = AMOVW;
q->from.type = D_OREG;
q->from.symkind = N_EXTERN;
q->from.sym = s;
q->from.offset = n*4 + 4;
q->to.type = D_REG;
q->to.reg = REGTMP;

// add_after(q, p)
q->link = p->link;
p->link = q;

p = q;

// ADD, $1, R11 //$
q = prg();
q->line = p->line;
q->pc = p->pc;
q->as = AADD;
q->from.type = D_CONST;
q->from.offset = 1;
q->to.type = D_REG;
q->to.reg = REGTMP;

// add_after(q, p)
q->link = p->link;
p->link = q;

p = q;

// MOVW R11, __mcount+ n*4+4(SB)
q = prg();
q->line = p->line;
q->pc = p->pc;
q->as = AMOVW;
q->from.type = D_REG;
q->from.reg = REGTMP;
q->to.type = D_OREG;
q->to.symkind = N_EXTERN;
q->to.sym = s;
q->to.offset = n*4 + 4;

// add_after(q, p)
q->link = p->link;
p->link = q;

p = q;

```

```

        n += 2;
        continue;
    }
}

// DATA __mcount+0(SB)/4, $n
q = prg();
q->line = 0;
q->as = ADATA;
q->from.type = D_OREG;
q->from.symkind = N_EXTERN;
q->from.sym = s;
q->reg = 4;
q->to.type = D_CONST;
q->to.offset = n;

// add_list(q, datap)
q->link = datap;
datap = q;

s->type = SBSS;
s->value = n*4;
}

```

Uses DBG 279, P 36c, SBSS 170b, datap 36e, firstp 36a, lookup() 32a, and prg() 43a.

Note the use of SBSS^{170b} instead of SDATAX above since the profiling instrumentation is done before `dodata()`^{92c}. The layout of data has not been done yet.

Given the instrumentation done by `doprof1()`, we can then easily modify the `main()`^{246j} of the profiled program to print data from `__mcount`. We can then display all the function calls counts or a subset of those counts.

11.3 5l -p and _profin()/_profout()

The second and default profiling strategy, implemented by `doprof2()`^{166a}, is to count *the time spent in each function*. Most of the logic for this strategy is actually implemented in `lib_core/libc/port/profile.c` in two functions: `_profin()` and `_profout()`. I refer you to the PROFILER book [Pad26] for more information on those functions. In this section, I will just explain how 5l instruments the code to call those functions.

Here is an example of how the code is instrumented by 5l -p for a program with 2 procedures `foo` and `bar`:

```

TEXT foo(SB), $0 -> TEXT foo(SB), $0
                   BL _profin(SB)
...
RET                -> BL _profout(SB)
                   RET

TEXT bar(SB), $0 -> TEXT bar(SB), $0
                   BL _profin(SB)
...
RET                -> BL _profout(SB)
                   RET

```

Again, the code of `doprof2()` is pretty straightforward:

```

<function doprof2(arm) 166a>≡ (292a)
void
doprof2(void)
{
    Sym *s2, *s4;
    Prog *p, *q;
    <doprof2() other locals 166b>

    DBG("%5.2f profile 2\n", cputime());

    // in lib_core/libc/port/profile.c
    s2 = lookup("_profin", 0);
    s4 = lookup("_profout", 0);
    <doprof2() sanity check s2 and s4 168c>

    <doprof2() find ps2, ps4, the Instr of s2 and s4 166c>

    for(p = firstp; p != P; p = p->link) {
        if(p->as == ATEXT) {
            <doprof2() if NOPROF p(arm) 169a>
            <doprof2() ATEXT instrumentation 166d>
            continue;
        }
        if(p->as == ARET) {
            <doprof2() ARET instrumentation 167>
            continue;
        }
    }
}

```

Uses P 36c, STTEXT 156d, firstp 36a, and lookup() 32a.

Before instrumenting ATEXT and ARET, 51 needs first to find the instructions containing the ATEXT of `_profin()` and `_profout()`. Indeed, later it will generate instructions that call those functions, and so 51 will need to set their `Instr.condX` fields to point to the right instruction. 51 needs to maintain what `patch()`⁸² did for the other branching instructions.

```

<doprof2() other locals 166b>≡ (166a)
    Prog *ps2 = P;
    Prog *ps4 = P;

```

```

<doprof2() find ps2, ps4, the Instr of s2 and s4 166c>≡ (166a)
    for(p = firstp; p != P; p = p->link) {
        if(p->as == ATEXT) {
            if(p->from.sym == s2) {
                ps2 = p;
                <doprof2() set TEXT attribute of _profin or _profout 169b>
            }
            if(p->from.sym == s4) {
                ps4 = p;
                <doprof2() set TEXT attribute of _profin or _profout 169b>
            }
        }
    }
}

```

The ATEXT instrumentation is trivial:

```

<doprof2() ATEXT instrumentation 166d>≡ (166a)
/*
 * BL profin

```

```

*/
q = prg();
q->line = p->line;
q->pc = p->pc;
q->as = ABL;
q->to.type = D_BRANCH;
q->cond = ps2; // _profin
q->to.sym = s2;

//insert_after(q, p)
q->link = p->link;
p->link = q;

```

```
p = q;
```

Uses `prg()` 43a.

The `ARET` instrumentation is more subtle. First, 51 needs to take care of the possible branching instructions which were originally branching to the `RET` instruction. Those instructions must now branch to the instrumented `BL _profout`. Indeed, we don't want to return before calling `_profout()`. This is why in the code below, `p`, to which branching instructions may point to via their `Instr.condX` fields, is first copied in `q` and then overwritten:

<doprof2() ARET instrumentation 167> ≡ (166a)

```

/*
 * RET
 */
q = prg();
// *q = *p;
q->as = ARET;
q->from = p->from;
q->to = p->to;
q->cond = p->cond;
q->link = p->link;
q->reg = p->reg;

// insert_after(q, p)
p->link = q;

```

<doprof2() in ARET case, if conditinal execution 168a>

```

else {
/*
 * BL profout
 */
// overwrite original RET instruction
p->as = ABL;
p->from = zprg.from;
p->to = zprg.to;
p->to.type = D_BRANCH;
p->cond = ps4; // _profout
p->to.sym = s4;
p->scond = COND_ALWAYS;

p = q;
}

```

Uses `prg()` 43a and `zprg` 43b.

The second subtlety is due to the possible setting of a conditional execution on `RET`, e.g., `RET.EQ`. In that case, we want to also conditionally execute `_profout()` hence the code below which may jump over the call to `_profout()`. Here is an example of instrumentation:

...

```
RET.EQ      -> 1000: B.NE 1012
            1004: BL _profout
            1008: RET
ADD R1, R2  -> 1012: ADD R1, R2
```

```
<doprof2() in ARET case, if conditinal execution 168a>≡ (167)
if(p->scond != COND_ALWAYS) {
    // BL _profout
    q = prg();
    q->as = ABL;
    q->from = zprg.from;
    q->to = zprg.to;
    q->to.type = D_BRANCH;
    q->cond = ps4; // _profout
    q->to.sym = s4;

    // insert_after(q, p)
    q->link = p->link;
    p->link = q;

    // overwrite original RET instruction with B.XXX
    p->as = brcond[p->scond^1]; /* complement */
    p->scond = COND_ALWAYS;
    p->from = zprg.from;
    p->to = zprg.to;
    p->to.type = D_BRANCH;
    p->cond = q->link->link; /* successor of RET */
    p->to.offset = q->link->link->pc; // useful??

    p = q->link->link;
}
```

Uses `brcond-2` 292a, `prg()` 43a, and `zprg` 43b.

The expression `p->scond^1` will reverse the last bit of the conditional execution which when used to index `brcond` below will return the complement condition:

```
<global brcond(arm) 168b>≡ (292a)
static int brcond[] =
{ABEQ, ABNE,
 ABHS, ABLO,
 ABMI, ABPL,
 ABVS, ABVC,
 ABHI, ABLS,
 ABGE, ABLT,
 ABGT, ABLE};
```

```
<doprof2() sanity check s2 and s4 168c>≡ (166a)
if(s2->type != STEXT || s4->type != STEXT) {
    diag("_profin/_profout not defined");
    return;
}
```

11.4 Disabling profiling attribute: NOPROF

You can disable profiling for certain functions by setting the `NOPROF` attribute (1<<0), e.g., with `TEXT foo(SB), 1, $0`. This attribute is declared in `5.out.h`. See the ASSEMBLER book [Pad15a] for more information on text

attributes.

```
<doprof2() if NOPROF p(arm) 169a>≡ (166a)
if(p->reg & NOPROF) {
    for(;;) {
        q = p->link;
        if(q == P || q->as == ATEXT)
            break;
        p = q;
    }
    continue;
}
```

Uses P 36c.

The code above will skip the instrumentation of the current procedure. It will actually skip all the instructions of the procedure, including its RET, until the next procedure.

Of course we do not want to instrument the `_profin()` and `_profout()` functions, otherwise 51 would generate infinite loops, hence the defensive code below:

```
<doprof2() set TEXT attribute of _profin or _profout 169b>≡ (166)
p->reg = NOPROF;
```

Chapter 12

Advanced Topics

The earlier chapters covered 5l's core pipeline: loading objects, resolving symbols, and generating machine code. This chapter covers features that extend or complicate that pipeline: dynamic linking (loading shared libraries at runtime), position-independent code, the ELF executable format, and the linker's role in supporting specific ARM features like trampolines for long branches.

12.1 Dynamic linking

```
<global dlm 170a>≡ (283b)
bool dlm;
```

```
<Section cases 170b>+≡ (33b) <156d 175d>
SIMPORT,
SEXPORt,
```

12.1.1 Export table: 5l -x

```
<global doexp 170c>≡ (283b)
// do export table, -x
bool doexp;
```

```
<main() command line processing(arm) 170d>+≡ (38e) <73f 174b>
case 'x': /* produce export table */
doexp = true;
if(argv[1] != nil && argv[1][0] != '-' && !isobjfile(argv[1]))
readundefs(ARGF(), SEXPORt);
break;
```

```
<function isobjfile 170e>≡ (290)
int
isobjfile(char *f)
{
int n, v;
Biobuf *b;
char buf1[5], buf2[SARMAG];

b = Bopen(f, OREAD);
if(b == nil)
return 0;
n = Bread(b, buf1, 5);
if(n == 5 && (buf1[2] == 1 && buf1[3] == '<' || buf1[3] == 1 && buf1[4] == '<'))
v = 1; /* good enough for our purposes */
else{
```

```

    Bseek(b, 0, 0);
    n = Bread(b, buf2, SARMAG);
    v = n == SARMAG && strcmp(buf2, ARMAG, SARMAG) == 0;
}
Bterm(b);
return v;
}

```

<global EXPTAB 171a>≡ (283b)
char* EXPTAB;

<main() if export table or dynamic module(arm) 171b>≡ (79)
if(doexp || dlm){
EXPTAB = "_exporttab";
zerosig(EXPTAB);
zerosig("etext");
zerosig("edata");
zerosig("end");

<main() if dynamic module(arm) 175a>
else
divsig();

export();
}

<function zerosig 171c>≡ (287b)
void
zerosig(char *sp)
{
Sym *s;

s = lookup(sp, 0);
s->sig = 0;
}

Uses lookup() 32a.

<global nimports 171d>≡ (287b)
int nimports;

<global nexports 171e>≡ (287b)
int nexports;

<global imports 171f>≡ (287b)
int imports;

<global exports 171g>≡ (287b)
int exports;

<Sym other fields 171h>+≡ (30) <69b
// enum<Section> too?
short subtype;

<function readundefs 172a>≡

(287b)

```
void
readundefs(char *f, int t)
{
    int i, n;
    Sym *s;
    Biobuf *b;
    char *l, buf[256], *fields[64];

    if(f == nil)
        return;
    b = Bopen(f, OREAD);
    if(b == nil){
        diag("could not open %s: %r", f);
        errexit();
    }
    while((l = Brdline(b, '\n')) != nil){
        n = Blinelen(b);
        if(n >= sizeof(buf)){
            diag("%s: line too long", f);
            errexit();
        }
        memmove(buf, l, n);
        buf[n-1] = '\0';
        n = getfields(buf, fields, nelem(fields), 1, " \t\r\n");
        if(n == nelem(fields)){
            diag("%s: bad format", f);
            errexit();
        }
        for(i = 0; i < n; i++){
            s = lookup(fields[i], 0);
            s->type = SXREF;
            s->subtype = t;
            if(t == SIMPORT)
                nimports++;
            else
                nexports++;
        }
    }
    Bterm(b);
}
```

Uses SIMPORT, SXREF, diag() 229d, errexit() 229b, lookup() 32a, nexports 171e, and nimports 171d.

<function export(arm) 172b>≡

(287b)

```
void
export(void)
{
    int i, j, n, off, nb, sv, ne;
    Sym *s, *et, *str, **esyms;
    Prog *p;
    char buf[NSNAME], *t;

    n = 0;
    for(i = 0; i < NHASH; i++)
        for(s = hash[i]; s != S; s = s->link)
            if(s->sig != 0 && s->type != SXREF && s->type != SUNDEF && (nexports == 0 || s->subtype == SEXPORT)
                n++;
    esyms = malloc(n*sizeof(Sym*));
    ne = n;
    n = 0;
}
```

```

for(i = 0; i < NHASH; i++)
    for(s = hash[i]; s != S; s = s->link)
        if(s->sig != 0 && s->type != SXREF && s->type != SUNDEF && (nexports == 0 || s->subtype == SEXPORT)
            esyms[n++] = s;
for(i = 0; i < ne-1; i++)
    for(j = i+1; j < ne; j++)
        if(strcmp(esyms[i]->name, esyms[j]->name) > 0){
            s = esyms[i];
            esyms[i] = esyms[j];
            esyms[j] = s;
        }

nb = 0;
off = 0;
et = lookup(EXPTAB, 0);
if(et->type != 0 && et->type != SXREF)
    diag("%s already defined", EXPTAB);
et->type = SDATA;
str = lookup(".string", 0);
if(str->type == 0)
    str->type = SDATA;
sv = str->value;
for(i = 0; i < ne; i++){
    s = esyms[i];
    Bprint(&bso, "EXPORT: %s sig=%lux t=%d\n", s->name, s->sig, s->type);

    /* signature */
    p = newdata(et, off, sizeof(long), N_EXTERN);
    off += sizeof(long);
    p->to.offset = s->sig;

    /* address */
    p = newdata(et, off, sizeof(long), N_EXTERN);
    off += sizeof(long);
    p->to.symkind = N_EXTERN;
    p->to.sym = s;

    /* string */
    t = s->name;
    n = strlen(t)+1;
    for(;;){
        buf[nb++] = *t;
        sv++;
        if(nb >= NSNAME){
            p = newdata(str, sv-NSNAME, NSNAME, N_INTERN);
            p->to.type = D_SCONST;
            p->to.sval = malloc(NSNAME);
            memmove(p->to.sval, buf, NSNAME);
            nb = 0;
        }
        if(*t++ == 0)
            break;
    }

    /* name */
    p = newdata(et, off, sizeof(long), N_EXTERN);
    off += sizeof(long);
    p->to.symkind = N_INTERN;
    p->to.sym = str;
    p->to.offset = sv-n;

```

```

}

if(nb > 0){
    p = newdata(str, sv-nb, nb, N_INTERN);
    p->to.type = D_SCONST;
    p->to.sval = malloc(NSNAME);
    memmove(p->to.sval, buf, nb);
}

for(i = 0; i < 3; i++){
    newdata(et, off, sizeof(long), N_EXTERN);
    off += sizeof(long);
}
et->value = off;
if(sv == 0)
    sv = 1;
str->value = sv;
exports = ne;
free(esyms);
}

```

Uses EXPTAB 171a, NHASH, S 33b, SDATA, SEXPORT 214b, SUNDEF 33b, SXREF, bso 218c, diag() 229d, exports 171g, free() 233b, lookup() 32a, malloc() 233a, newdata() 174a, and nexports 171e.

<function newdata(arm) 174a> ≡ (287b)

```

static Prog*
newdata(Sym *s, int o, int w, int t)
{
    Prog *p;

    p = prg();
    p->link = datap;
    datap = p;

    p->as = ADATA;
    p->reg = w;
    p->from.type = D_OREG;
    p->from.symkind = t;
    p->from.sym = s;
    p->from.offset = o;
    p->to.type = D_CONST;
    p->to.symkind = N_NONE;

    return p;
}

```

Uses datap 36e and prg() 43a.

12.1.2 Dynamic loading: 5l -u

<main() command line processing(arm) 174b>+≡ (38e) <170d 188f>

```

case 'u': /* produce dynamically loadable module */
    dlm = true;
    if(argv[1] != nil && argv[1][0] != '-' && !isobjfile(argv[1]))
        readundefs(ARGF(), SIMPORT);
    break;

```

<asmb() if dynamic module magic header adjustment(arm) 174c>≡ (46b)

```

if(dlm)
    lput(0x80000000|0x647); /* magic */

```

Uses dlm 170a and lput() 110b.

```

⟨main() if dynamic module(arm) 175a⟩≡ (171b)
if(dlm){
    initdiv();
    import();
    HEADTYPE = H_PLAN9;
    INITTEXT = INITDAT = 0;
    INITRND = 8;
    INITENTRY = EXPTAB;
}

```

```

⟨function import(arm) 175b⟩≡ (287b)
void
import(void)
{
    int i;
    Sym *s;

    for(i = 0; i < NHASH; i++)
        for(s = hash[i]; s != S; s = s->link)
            if(s->sig != 0 && s->type == SXREF && (nimports == 0 || s->subtype == SIMPORT)){
                undefsym(s);
                Bprint(&bso, "IMPORT: %s sig=%lux v=%ld\n", s->name, s->sig, s->value);
            }
}

```

Uses NHASH, S 33b, SIMPORT, SXREF, bso 218c, nimports 171d, and undefsym() 176b.

```

⟨enum rxxx 175c⟩≡ (279)
enum rxxx {
    Roffset = 22, /* no. bits for offset in relocation address */
    Rindex = 10, /* no. bits for index in relocation address */
};

```

12.1.3 SUNDEF

```

⟨Section cases 175d⟩+≡ (33b) <170b 183a>
SUNDEF,

```

```

⟨global undefp 175e⟩≡ (283b)
/*@Scheck: not dead, used by UP
Prog undefp;

```

```

⟨constant UP 175f⟩≡ (279)
#define UP (&undefp)

```

```

⟨patch() switch section type for branch instruction, cases 175g⟩+≡ (83b) <83c
case SUNDEF:
    if(p->as != ABL)
        diag("help: SUNDEF in AB || ARET");
    p->to.offset = 0;
    p->to.type = D_BRANCH;
    p->cond = UP;
    break;

```

Uses SUNDEF 33b, UP 279, and diag() 229d.

```

⟨asmout() BRA case, if undefined target 176a⟩≡ (128b)
    if(p->cond == UP) {
        s = p->to.sym;
        if(s->type != SUNDEF)
            diag("bad branch sym type");
        v = (ulong)s->value >> (Roffset-2);
        dynreloc(s, p->pc, 0);
    }

```

Uses Roffset, SUNDEF 33b, UP 279, diag() 229d, and dynreloc() 179c.

```

⟨function undefsym 176b⟩≡ (287b)
    void
    undefsym(Sym *s)
    {
        int n;

        n = imports;
        if(s->value != 0)
            diag("value != 0 on SXREF");
        if(n >= 1<<Rindex)
            diag("import index %d out of range", n);
        s->value = n<<Roffset;
        s->type = SUNDEF;
        imports++;
    }

```

Uses Rindex 182e, Roffset, SUNDEF 33b, diag() 229d, and imports 171f.

```

⟨datblk() in D_ADDR case, switch symbol type cases 176c⟩+≡ (52b) <52c
    case SUNDEF:
        ckoff(v, d);
        d += p->to.sym->value;
        break;

```

Uses SUNDEF 33b and ckoff() 176d.

```

⟨function ckoff 176d⟩≡ (287b)
    void
    ckoff(Sym *s, long v)
    {
        if(v < 0 || v >= 1<<Roffset)
            diag("relocation offset %ld for %s out of range", v, s->name);
    }

```

Uses Roffset and diag() 229d.

12.1.4 XXX

```

⟨aclass() in D_ADDR case, SDATA case, if dlm 176e⟩≡ (117b)
    if(dlm) {
        instoffset = s->value + a->offset + INITDAT;
        return C_LCON;
    }

```

Uses C_LCON 113a, INITDAT 40g, dlm 170a, and instoffset 112f.

```

⟨asmb() if dynamic module, before datblk() 176f⟩≡ (48c)
    if(dlm){
        char buf[8];

        write(cout, buf, INITDAT-textsize);
        textsize = INITDAT;
    }

```

Uses INITDAT 40g, cout 38d, dlm 170a, and textsize 94a.

```

<asmb() if dynamic module and no symbol table generation 177a>≡ (147a)
    if(dlm){
        seek(cout, HEADR+textsize+datsize, 0);
        asmdyn();
        cflush();
    }

```

Uses HEADR 40d, asmdyn() 177f, cflush() 110a, cout 38d, datsize 92a, dlm 170a, and textsize 94a.

```

<entryvalue() if dynamic module case 177b>≡ (47a)
    case SDATA:
        if(dlm)
            return s->value+INITDAT;

```

Uses INITDAT 40g, SDATA, and dlm 170a.

```

<struct Reloc 177c>≡ (287b)
    struct Reloc
    {
        int n;
        int t;
        byte *m;
        ulong *a;
    };

```

```

<global rels 177d>≡ (287b)
    Reloc rels;

```

```

<asmb() if dynamic module, call asmdyn() 177e>≡ (147a)
    if(dlm)
        asmdyn();

```

Uses asmdyn() 177f and dlm 170a.

```

<function asmdyn 177f>≡ (287b)
    void
    asmdyn()
    {
        int i, n, t, c;
        Sym *s;
        ulong la, ra, *a;
        vlong off;
        byte *m;
        Reloc *r;

        cflush();
        off = seek(cout, 0, 1);
        lput(0);
        t = 0;
        lput(imports);
        t += 4;
        for(i = 0; i < NHASH; i++)
            for(s = hash[i]; s != S; s = s->link)
                if(s->type == SUNDEF){
                    lput(s->sig);
                    t += 4;
                    t += sput(s->name);
                }

        la = 0;
        r = &rels;
        n = r->n;

```

```

m = r->m;
a = r->a;
lput(n);
t += 4;
for(i = 0; i < n; i++){
    ra = *a-la;
    if(*a < la)
        diag("bad relocation order");
    if(ra < 256)
        c = 0;
    else if(ra < 65536)
        c = 1;
    else
        c = 2;
    cput((c<<6)|*m++);
    t++;
    if(c == 0){
        cput(ra);
        t++;
    }
    else if(c == 1){
        wput(ra);
        t += 2;
    }
    else{
        lput(ra);
        t += 4;
    }
    la = *a++;
}

cflush();
seek(cout, off, 0);
lput(t);

DBG("import table entries = %d\n", imports);
DBG("export table entries = %d\n", exports);
}

```

Uses [DBG 279](#), [NHASH, S 33b](#), [SUNDEF 33b](#), [cflush\(\) 110a](#), [cout 38d](#), [cput\(\) 234c](#), [diag\(\) 229d](#), [exports 171g](#), [imports 171f](#), [lput\(\) 110b](#), [rels 177d](#), [sput\(\) 178a](#), and [wput\(\) 234e](#).

```

⟨function sput 178a⟩≡ (287b)
static int
sput(char *s)
{
    char *p;

    p = s;
    while(*s)
        cput(*s++);
    cput(0);
    return s-p+1;
}

```

Uses [cput\(\) 234c](#).

```

⟨global modemap 178b⟩≡ (287b)
int modemap[4] = { 0, 1, -1, 2, };

```

```
<datblk() if dynamic module(arm) 179a>≡ (52b)
if(dlm)
```

```
    dynreloc(v, a+INITDAT, 1);
```

Uses INITDAT 40g, dlm 170a, and dynreloc() 179c.

```
<enum SpanConstants(arm) 179b>≡ (287b)
```

```
enum{
    ABSD = 0,
    ABSU = 1,
    RELD = 2,
    RELU = 3,
};
```

```
<function dynreloc(arm) 179c>≡ (287b)
```

```
void
dynreloc(Sym *s, long v, int abs)
{
    int i, k, n;
    byte *m;
    ulong *a;
    Reloc *r;

    if(v&3)
        diag("bad relocation address");
    v >>= 2;

    if(s != S && s->type == SUNDEF)
        k = abs ? ABSU : RELU;
    else
        k = abs ? ABSD : RELD;
    /* Bprint(&bso, "R %s a=%ld(%lx) %d\n", s->name, a, a, k); */
    k = modemap[k];
    r = &rels;
    n = r->n;
    if(n >= r->t)
        grow(r);
    m = r->m;
    a = r->a;
    for(i = n; i > 0; i--){
        if(v < a[i-1]){ /* happens occasionally for data */
            m[i] = m[i-1];
            a[i] = a[i-1];
        }
        else
            break;
    }
    m[i] = k;
    a[i] = v;
    r->n++;
}
```

Uses ABSD-3 179b, ABSU-4 179b, RELD-5 179b, RELU-6 179b, S 33b, SUNDEF 33b, diag() 229d, grow() 179d, modemap 178b, and rels 177d.

```
<function grow 179d>≡ (287b)
```

```
static void
grow(Reloc *r)
{
    int t;
    byte *m, *nm;
    ulong *a, *na;
```

```

t = r->t;
r->t += 64;
m = r->m;
a = r->a;
r->m = nm = malloc(r->t * sizeof(byte));
r->a = na = malloc(r->t * sizeof(ulong));
memmove(nm, m, t*sizeof(byte));
memmove(na, a, t*sizeof(ulong));
free(m);
free(a);
}

```

Uses `free()` 233b and `malloc()` 233a.

12.1.5 Relocatable address: C_ADDR

```

⟨Operand_class cases 180a⟩+≡ (101a) <121c 190a>
C_ADDR, /* relocatable address */

```

```

⟨aclass() when D_OREG and external symbol and dlm 180b⟩≡ (115c)
if(dlm) {
    switch(t) {
        case STEXT: case SSTRING:
        case SUNDEF:
            instoffset = s->value + a->offset;
            break;
        case SDATA: case SBSS:
        default:
            instoffset = s->value + a->offset + INITDAT;
            break;
    }
    return C_ADDR;
}

```

Uses `C_ADDR` 180a, `INITDAT` 40g, `SBSS` 170b, `SDATA`, `SSTRING` 35g, `STEXT` 156d, `SUNDEF` 33b, `dlm` 170a, and `instoffset` 112f.

```

⟨optab entries 180c⟩+≡ (99b) <139e 180d>
{ ATEXT, C_ADDR, C_NONE, C_LCON, 0, 0 },
{ ATEXT, C_ADDR, C_REG, C_LCON, 0, 0 },

```

Uses `C_ADDR` 180a, `C_LCON` 113a, `C_NONE` 101a, and `C_REG` 101a.

```

⟨optab entries 180d⟩+≡ (99b) <180c 181a>
{ AWORD, C_NONE, C_NONE, C_ADDR, 11, 4 },

```

Uses `C_ADDR` 180a and `C_NONE` 101a.

```

⟨asmout() in AWORD case, when dlm 180e⟩≡ (112e)
switch(c) {
    case C_LCON:
        if(!dlm)
            break;
        if(p->to.symkind != N_EXTERN && p->to.symkind != N_INTERN)
            break;
        // Fallthrough
    case C_ADDR:
        if(p->to.sym->type == SUNDEF)
            ckoff(p->to.sym, p->to.offset);
        dynreloc(p->to.sym, p->pc, 1);
}

```

Uses `C_ADDR` 180a, `C_LCON` 113a, `SUNDEF` 33b, `ckoff()` 176d, `dlm` 170a, and `dynreloc()` 179c.

`<optab entries 181a>+≡` (99b) <180d 181c>

```
{ AMOVW, C_REG, C_NONE, C_ADDR, 64, 8, 0, LTO },
{ AMOVBU, C_REG, C_NONE, C_ADDR, 64, 8, 0, LTO },
{ AMOVBU, C_REG, C_NONE, C_ADDR, 64, 8, 0, LTO },
```

Uses C_ADDR 180a, C_NONE 101a, C_REG 101a, and LTO 141c.

`<asmout() switch on type cases 181b>+≡` (108b) <139f 181d>

```
/* reloc ops */
case 64: /* mov/movb/movbu R,addr */
    o1 = omvl(p, &p->to, REGTMP);
    if(!o1)
        break;
    o2 = osr(p->as, p->scond, p->from.reg, 0, REGTMP);
    break;
```

Uses omvl() 125c and osr() 133d.

`<optab entries 181c>+≡` (99b) <181a 181e>

```
{ AMOVW, C_ADDR, C_NONE, C_REG, 65, 8, 0, LFROM },
{ AMOVBU, C_ADDR, C_NONE, C_REG, 65, 8, 0, LFROM },
{ AMOVBU, C_ADDR, C_NONE, C_REG, 65, 8, 0, LFROM },
{ AMOVBU, C_ADDR, C_NONE, C_REG, 65, 8, 0, LFROM },
{ AMOVBU, C_ADDR, C_NONE, C_REG, 65, 8, 0, LFROM },
{ AMOVBU, C_ADDR, C_NONE, C_REG, 65, 8, 0, LFROM },
```

Uses C_ADDR 180a, C_NONE 101a, C_REG 101a, and LFROM 141c.

`<asmout() switch on type cases 181d>+≡` (108b) <181b 182a>

```
case 65: /* mov/movbu addr,R */
case 66: /* movh/movhu/movb addr,R */
    o1 = omvl(p, &p->from, REGTMP);
    if(!o1)
        break;
    o2 = olr(p->as, p->scond, 0, REGTMP, p->to.reg);
    if(o->type == 65)
        break;

    o3 = oprrr(ASLL, p->scond);

    if(p->as == AMOVBU || p->as == AMOVHU)
        o4 = oprrr(ASRL, p->scond);
    else
        o4 = oprrr(ASRA, p->scond);

    r = p->to.reg;
    o3 |= (r)|(r<<12);
    o4 |= (r)|(r<<12);
    if(p->as == AMOVBU || p->as == AMOVBU) {
        o3 |= (24<<7);
        o4 |= (24<<7);
    } else {
        o3 |= (16<<7);
        o4 |= (16<<7);
    }
    break;
```

Uses olr() 131d, omvl() 125c, and oprrr() 120a.

`<optab entries 181e>+≡` (99b) <181c 195a>

```
{ AMOVH, C_REG, C_NONE, C_ADDR, 67, 24, 0, LTO },
{ AMOVHU, C_REG, C_NONE, C_ADDR, 67, 24, 0, LTO },
```

Uses C_ADDR 180a, C_NONE 101a, C_REG 101a, and LTO 141c.

`<asmout() switch on type cases 182a>+≡ (108b) <181d 195c>`

```
case 67: /* movh/movhu R,addr -> sb, sb */
    o1 = omvl(p, &p->to, REGTMP);
    if(!o1)
        break;
    o2 = osr(p->as, p->scond, p->from.reg, 0, REGTMP);

    o3 = oprrr(ASRL, p->scond);
    o3 |= (8<<7)|(p->from.reg)|(p->from.reg<<12);
    o3 |= (1<<6); /* ROR 8 */

    o4 = oprrr(AADD, p->scond);
    o4 |= (REGTMP << 12) | (REGTMP << 16);
    o4 |= immrot(1);

    o5 = osr(p->as, p->scond, p->from.reg, 0, REGTMP);

    o6 = oprrr(ASRL, p->scond);
    o6 |= (24<<7)|(p->from.reg)|(p->from.reg<<12);
    o6 |= (1<<6); /* ROL 8 */
    break;
```

Uses `immrot()` 113c, `omvl()` 125c, `oprrr()` 120a, and `osr()` 133d.

12.2 Position independent code (PIC)

12.3 Optimizations

12.3.1 Opcode rewriting

`<ldobj() switch opcode cases(arm) 182b>+≡ (55) <153 182c>`

```
case ASUB:
    if(p->from.type == D_CONST)
        if(p->from.offset < 0) {
            p->from.offset = -p->from.offset;
            p->as = AADD;
        }
    goto casedef;
```

`<ldobj() switch opcode cases(arm) 182c>+≡ (55) <182b 191k>`

```
case AADD:
    if(p->from.type == D_CONST)
        if(p->from.offset < 0) {
            p->from.offset = -p->from.offset;
            p->as = ASUB;
        }
    goto casedef;
```

12.3.2 Operand rewriting

`<dodata() define special symbols 182d>+≡ (92c) <97b>`

```
xdefine("setR12", SDATA, 0L+BIG);
```

Uses `BIG`, `SDATA`, and `xdefine()` 97a.

`<constant BIG 182e>≡ (278c)`

```
//BIG = (1<<12)-4,
BIG = 0,
```

12.3.3 Small data first

`<Section cases 183a>+≡ (33b) <175d 214b>`
SDATA1,

`<constant MINSIZ 183b>≡ (278c)`
MINSIZ = 64,

`<dodata() in pass 1, if small data size, adjust orig 183c>≡ (92c)`
/*
* assign 'small' variables to data segment
* (rational is that data segment is more easily
* addressed through offset on R12)
*/
if(v <= MINSIZ) {
 s->value = orig;
 orig += v;
 s->type = SDATA1;
}

Uses MINSIZ 157b and SDATA1 279.

`<dodata() in pass 2, retag small data 183d>≡ (92c)`
if(t == SDATA1)
 s->type = SDATA;

Uses SDATA and SDATA1 279.

12.3.4 Compacting chains of AB, brloop()

`<patch() optimisation pass 183e>≡ (82)`
for(p = firstp; p != P; p = p->link) {
 <adjust curtext when iterate over instructions p 36h>

 if(p->cond != P && p->cond != UP) {
 p->cond = brloop(p->cond);
 if(p->cond != P)
 if(p->to.type == D_BRANCH)
 p->to.offset = p->cond->pc;
 }
}

Uses P 36c, UP 279, brloop() 183f, and firstp 36a.

`<function brloop(arm) 183f>≡ (286b)`
/// main -> patch -> <>
Prog*
brloop(Prog *p)
{
 Prog *q;
 int c = 0;

 for(; p!=P;) {
 if(p->as != AB)
 return p;
 q = p->cond;
 if(q <= p) {
 c++;
 if(q == p || c > 5000)
 break;
 }
 p = q;
 }

```

    }
    return P;
}

```

Uses P 36c.

12.3.5 Removing useless instructions, follow()

<function follow 184a>≡ (286b)

```

void
follow(void)
{
    DBG("%5.2f follow\n", cputime());

    firstp = prg();
    lastp = firstp;

    xfol(textp);

    lastp->link = P;
    firstp = firstp->link;
}

```

Uses DBG 279, P 36c, firstp 36a, lastp 36d, prg() 43a, textp 152a, and xfol() 184c.

<Mark cases 184b>+≡ (35g) <88a

```

FOLL      = 1<<0,

```

<function xfol(arm) 184c>≡ (286b)

```

void
xfol(Prog *p)
{
    Prog *q, *r;
    int a, i;

loop:
    if(p == P)
        return;
    <adjust curtext when iterate over instructions p 36h>
    a = p->as;

    if(a == AB) {
        q = p->cond;
        if(q != P) {
            p->mark |= FOLL;
            p = q;
            if(!(p->mark & FOLL))
                goto loop;
        }
    }

    if(p->mark & FOLL) {
        <xfol() when p is marked, for loop to copy instructions 186b>
        a = AB;
        q = prg();
        q->as = a;
        q->line = p->line;
        q->to.type = D_BRANCH;
        q->to.offset = p->pc;
    }
}

```

```

    q->cond = p;
    p = q;
}

p->mark |= FOLL;
lastp->link = p;
lastp = p;

if(a == AB || (a == ARET && p->scond == COND_ALWAYS) || a == ARFE){
    return;
}

if(p->cond != P)
    ⟨xfol() if a is not ABL and p has a link 185a⟩
p = p->link;
goto loop;
}

```

Uses FOLL 40c, P 36c, lastp 36d, and prg() 43a.

⟨xfol() if a is not ABL and p has a link 185a⟩≡

(184c)

```

if(a != ABL && p->link != P) {
    q = brchain(p->link);

    if(a != ATEXT && a != ABCASE)
        if(q != P && (q->mark & FOLL)) {
            p->as = relinv(a);
            p->link = p->cond;
            p->cond = q;
        }

    // recursive call
    xfol(p->link);

    q = brchain(p->cond);
    if(q == P)
        q = p->cond;
    if(q->mark&FOLL) {
        p->cond = q;
        return;
    }
    p = q;
    goto loop;
}

```

Uses FOLL 40c, P 36c, brchain() 185b, relinv() 186a, and xfol() 184c.

⟨function brchain(arm) 185b⟩≡

(286b)

```

static Prog*
brchain(Prog *p)
{
    int i;

    for(i=0; i<20; i++) {
        if(p == P || p->as != AB)
            return p;
        p = p->cond;
    }
    return P;
}

```

Uses P 36c.

<function relinv(arm) 186a>≡

(286b)

```
int
relinv(int a)
{
    switch(a) {
        case ABEQ: return ABNE;
        case ABNE: return ABEQ;
        case ABHS: return ABLO;
        case ABLO: return ABHS;
        case ABMI: return ABPL;
        case ABPL: return ABMI;
        case ABVS: return ABVC;
        case ABVC: return ABVS;
        case ABHI: return ABLS;
        case ABLS: return ABHI;
        case ABGE: return ABLT;
        case ABLT: return ABGE;
        case ABGT: return ABLE;
        case ABLE: return ABGT;
    }
    diag("unknown relation: %s", anames[a]);
    return a;
}
```

Uses diag() 229d.

<xfol() when p is marked, for loop to copy instructions 186b>≡

(184c)

```
for(i=0, q=p; i<4 && q != lastp; i++, q=q->link) {
    a = q->as;
    if(a == ANOP) {
        i--;
        continue;
    }
    if(a == AB || (a == ARET && q->scond == COND_ALWAYS) || a == ARFE)
        goto copy;
    if(!q->cond || (q->cond->mark & FOLL))
        continue;
    if(a != ABEQ && a != ABNE)
        continue;

// here when a is one of AB, ARET, ARFE, ABEQ, ABNE
copy:
    for(;;) {
        r = prg();
        *r = *p;

        <xfol() sanity check one, r should be marked ??>

        if(p != q) {
            p = p->link;
            lastp->link = r;
            lastp = r;
            continue;
        }
        lastp->link = r;
        lastp = r;

        if(a == AB || (a == ARET && q->scond == COND_ALWAYS) || a == ARFE)
            return;

        // r->as = relinv(a)
    }
}
```

```

    r->as = ABNE;
    if(a == ABNE)
        r->as = ABEQ;

    r->cond = p->link;
    r->link = p->cond;

    if(!(r->link->mark & FOLL))
        // recursive call
        xfol(r->link);

    <xfol() sanity check two, r->cond should be marked ??>

    return;
}
}

```

Uses FOLL 40c, lastp 36d, prg() 43a, and xfol() 184c.

12.4 Overriding symbol attribute: DUPOK

```

<ldobj() locals(arm) 187a>+≡ (55) <69f 191i>
    bool skip;

```

```

<ldobj() after newloop when new object file, more initializations 187b>+≡ (55) <157f
    skip = false;

```

```

<ldobj() case ATEXT and section not SNONE or SXREF, if DUPOK 187c>≡ (65c)
    if(p->reg & DUPOK) {
        skip = true;
        goto casedef;
    }

```

```

<ldobj() in switch opcode default case, if skip 187d>≡ (64d)
    if(skip)
        nopout(p);

```

Uses nopout() 187f.

```

<ldobj() in switch opcode ATEXT case, reset skip 187e>≡ (65b)
    skip = false; // needed?

```

```

<function nopout 187f>≡ (290)
    static void
    nopout(Prog *p)
    {
        p->as = ANOP;
        p->from.type = D_NONE;
        p->to.type = D_NONE;
    }

```

12.5 Other executable formats

12.5.1 ELF (Linux)

```

<main() switch HEADTYPE cases(arm) 187g>+≡ (41a) <41b
    case H_ELF: /* elf executable */
        HEADR = rnd(Ehdr32sz+3*Phdr32sz, 16);

```

```

if(INITTEXT == -1)
    INITTEXT = 4096+HEADR;
if(INITDAT == -1)
    INITDAT = 0;
if(INITRND == -1)
    INITRND = 4;
break;

```

`<asmb() switch HEADTYPE (to position after text) cases(arm) 188a>+≡ (48c) <48d`

```

case H_ELF:
    OFFSET = HEADR+textsize;
    seek(cout, OFFSET, 0);
    break;

```

Uses HEADR 40d, H_ELF 150a, cout 38d, and textsize 94a.

`<asmb() switch HEADTYPE (for symbol table generation) cases(arm) 188b>+≡ (147a) <147b`

```

case H_ELF:
    break;

```

Uses H_ELF 150a.

`<asmb() switch HEADTYPE (for header generation) cases(arm) 188c>+≡ (46a) <46b`

```

case H_ELF:
    debug['S'] = 1; /* symbol table */
    elf32(ARM, ELFDATA2LSB, 0, nil);
    break;

```

Uses H_ELF 150a and debug 218a.

`<global INITTEXTP 188d>≡ (283b)`

```

long INITTEXTP = -1; /* text location (physical) */

```

Uses INITTEXTP 188d.

`<main() last INITXXX adjustments 188e>≡`

```

if (INITTEXTP == -1)
    INITTEXTP = INITTEXT;

```

`<main() command line processing(arm) 188f>+≡ (38e) <174b 218b>`

```

case 'P':
    a = ARGF();
    if(a)
        INITTEXTP = atolwhex(a);
    break;

```

`<enum ElfHeaderSizes 188g>≡`

```

enum {
    Ehdr32sz = 52,
    Phdr32sz = 32,
    Shdr32sz = 40,

    Ehdr64sz = 64,
    Phdr64sz = 56,
    Shdr64sz = 64,
};

```

12.5.2 OMach (mac OS)

12.5.3 PE (Windows)

12.6 Other instructions

12.6.1 Float operations

Operand kind

```
<inopd() cases 189a>+≡ (57b) <58a 189b>  
    case D_FREG:  
    case D_FPCR:  
        break;
```

```
<inopd() cases 189b>+≡ (57b) <189a  
    case D_FCONST:  
        a->ieee = malloc(sizeof(Ieee));  
  
        a->ieee->l = p[4] | (p[5]<<8) | (p[6]<<16) | (p[7]<<24);  
        a->ieee->h = p[8] | (p[9]<<8) | (p[10]<<16) | (p[11]<<24);  
        size += 8;  
        break;
```

Uses malloc() 233a.

```
<function ieeedtof 189c>≡ (291c)  
    /// main -> objfile -> ldoobj -> <>  
    long  
    ieeedtof(Ieee *e)  
    {  
        int exp;  
        long v;  
  
        if(e->h == 0)  
            return 0;  
        exp = (e->h>>20) & ((1L<<11)-1L);  
        exp -= (1L<<10) - 2L;  
        v = (e->h & 0xffffL) << 3;  
        v |= (e->l >> 29) & 0x7L;  
        if((e->l >> 28) & 1) {  
            v++;  
            if(v & 0x800000L) {  
                v = (v & 0x7ffffL) >> 1;  
                exp++;  
            }  
        }  
        if(exp <= -126 || exp >= 130)  
            diag("double fp to single fp overflow");  
        v |= ((exp + 126) & 0xffL) << 23;  
        v |= e->h & 0x80000000L;  
        return v;  
    }
```

Uses diag() 229d.

```
<function ieeedtod 189d>≡ (291c)  
    /// Dconv -> <>  
    double  
    ieeedtod(Ieee *ieeep)  
    {
```

```

IEEE e;
double fr;
int exp;

if(ieee->h & (1L<<31)) {
    e.h = ieee->h & ~(1L<<31);
    e.l = ieee->l;
    return -ieeedtod(&e);
}
if(ieee->l == 0 && ieee->h == 0)
    return 0;
fr = ieee->l & ((1L<<16)-1L);
fr /= 1L<<16;
fr += (ieee->l>>16) & ((1L<<16)-1L);
fr /= 1L<<16;
fr += (ieee->h & (1L<<20)-1L) | (1L<<20);
fr /= 1L<<21;
exp = (ieee->h>>20) & ((1L<<11)-1L);
exp -= (1L<<10) - 2L;
return ldexp(fr, exp);
}

```

Uses `ieeedtod()` 189d.

Operand class

`<Operand_class cases 190a>+≡` (101a) <180a 206c>

```

C_FREG,
C_FCON,
C_FCR,

```

`<aclass() switch type cases 190b>+≡` (101b) <121d 206d>

```

case D_FREG:
    return C_FREG;
case D_FCONST:
    return C_FCON;
case D_FPCR:
    return C_FCR;

```

Uses `C_FCON` 190a, `C_FCR` 190a, and `C_FREG` 190a.

`<function immfloat(arm) 190c>≡` (289c)

```

static int
immfloat(long v)
{
    return (v & 0xC03) == 0; /* offset will fit in floating-point load/store */
}

```

`<Operand_class cases, in C_xEXT, float cases 190d>≡` (115b)

```

C_FEXT,
C_HFEXT,

```

`<aclass() if immfloat for N_EXTERN symbol 190e>≡` (115c)

```

if(immfloat(t))
    return imhalf(instoffset)? C_HFEXT : C_FEXT;

```

Uses `C_FEXT` 190d, `C_HFEXT` 190d, `immfloat()` 190c, `imhalf()` 209h, and `instoffset` 112f.

`<Operand_class cases, in C_xAUTO, float cases 190f>≡` (116c)

```

C_FAUTO, /* float insn offset (0 to 0x3fc, word aligned) */
C_HFAUTO, /* both H and F */

```

`<aclass() if immfloat for N_LOCAL or N_PARAM symbol 191a>≡ (116)`

```
if(immfloat(t))
    return immhalf(instoffset)? C_HFAUTO : C_FAUTO;
```

Uses C_FAUTO 190f, C_HFAUTO 190f, immfloat() 190c, immhalf() 209h, and instoffset 112f.

`<Operand_class cases, in C_xOREG, float cases 191b>≡ (114a)`

```
C_FOREG,
C_HFOREG,
```

`<aclass() if immfloat for N_NONE symbol 191c>≡ (114c)`

```
if(immfloat(t))
    return immhalf(instoffset)? C_HFOREG : C_FOREG;
/* n.b. that [C_FOREG] will also satisfy immrot */
```

Uses C_FOREG 191b, C_HFOREG 191b, immfloat() 190c, immhalf() 209h, and instoffset 112f.

VFP

`<Optab_flag cases 191d>≡ (141c) 208e▷`

```
VFP = 1<<4, /* arm vfpv3 floating point */
```

`<ocmp() if floating point flag on p1 or p2 191e>≡ (105b)`

```
n = (p2->flag&VFP) - (p1->flag&VFP); /* floating point arch */
if(n)
    return n;
```

Uses VFP 191d.

`<global vfp(arm) 191f>≡ (283b)`

```
bool vfp;
```

`<buildop() initialize flags 191g>≡ (104c) 208d▷`

```
vfp = debug['f'];
```

Uses debug 218a and vfp 191f.

`<buildop() adjust optab if flags, remove certain rules 191h>≡ (104c) 208f▷`

```
if((optab[n].flag & VFP) && !vfp)
    optab[n].as = AXXX;
```

Uses VFP 191d, optab 99b, and vfp 191f.

`<ldebug() locals(arm) 191i>+≡ (55) <187a`

```
Prog *t;
```

`<global literal(arm) 191j>≡ (290)`

```
char literal[32];
```

`<ldebug() switch opcode cases(arm) 191k>+≡ (55) <182c`

```
case AMOVDF:
    if(!vfp || p->from.type != D_FCONST)
        goto casedef;
    p->as = AMOVF;
    /* fall through */
case AMOVF:
    if(skip)
        goto casedef;

    if(p->from.type == D_FCONST && chipfloat(p->from.ieee) < 0) {
        /* size sb 9 max */
        sprintf(literal, "%lux", ieeeedtof(p->from.ieee));
        s = lookup(literal, 0);
        if(s->type == 0) {
```

```

        s->type = SBSS;
        s->value = 4;
        t = prg();
        t->as = ADATA;
        t->line = p->line;
        t->from.type = D_OREG;
        t->from.sym = s;
        t->from.symkind = N_EXTERN;
        t->reg = 4;
        t->to = p->from;
        t->link = datap;
        datap = t;
    }
    p->from.type = D_OREG;
    p->from.sym = s;
    p->from.symkind = N_EXTERN;
    p->from.offset = 0;
}
goto casedef;

case AMOVD:
    if(skip)
        goto casedef;

    if(p->from.type == D_FCONST && chipfloat(p->from.ieee) < 0) {
        /* size sb 18 max */
        sprintf(literal, "$%lux.%lux",
            p->from.ieee->l, p->from.ieee->h);
        s = lookup(literal, 0);
        if(s->type == 0) {
            s->type = SBSS;
            s->value = 8;
            t = prg();
            t->as = ADATA;
            t->line = p->line;
            t->from.type = D_OREG;
            t->from.sym = s;
            t->from.symkind = N_EXTERN;
            t->reg = 8;
            t->to = p->from;
            t->link = datap;
            datap = t;
        }
        p->from.type = D_OREG;
        p->from.sym = s;
        p->from.symkind = N_EXTERN;
        p->from.offset = 0;
    }
    goto casedef;

```

Uses SBSS 170b, chipfloat() 193a, datap 36e, ieeeedtof() 189c, literal 191j, lookup() 32a, prg() 43a, and vfp 191f.

<global chipfloats(*arm*) 192>≡

```

static Ieee chipfloats[] = {
    {0x00000000, 0x00000000}, /* 0 */
    {0x00000000, 0x3ff00000}, /* 1 */
    {0x00000000, 0x40000000}, /* 2 */
    {0x00000000, 0x40080000}, /* 3 */
    {0x00000000, 0x40100000}, /* 4 */
    {0x00000000, 0x40140000}, /* 5 */
    {0x00000000, 0x3fe00000}, /* .5 */

```

(291c)

```

    {0x00000000, 0x40240000}, /* 10 */
};

```

```

⟨function chipfloat(arm) 193a⟩≡ (291c)
int
chipfloat(Ieee *e)
{
    Ieee *p;
    int n;

    if(vfp)
        return -1;
    for(n = sizeof(chipfloats)/sizeof(chipfloats[0]); --n >= 0;){
        p = &chipfloats[n];
        if(p->l == e->l && p->h == e->h)
            return n;
    }
    return -1;
}

```

Uses chipfloats-14 192 and vfp 191f.

```

⟨noops() second pass switch opcode cases 193b⟩+≡ (87) <89b 201>
/*
 * 5c code generation for unsigned -> double made the
 * unfortunate assumption that single and double floating
 * point registers are aliased - true for emulated 7500
 * but not for vfp. Now corrected, but this test is
 * insurance against old 5c compiled code in libraries.
 */
case AMOVWD:
    if((q = p->link) != P && q->as == ACMP)
    if((q = q->link) != P && q->as == AMOVF)
    if((q1 = q->link) != P && q1->as == AADDF)
    if(q1->to.type == D_FREG && q1->to.reg == p->to.reg) {
        q1->as = AADDD;
        q1 = prg();
        q1->scond = q->scond;
        q1->line = q->line;
        q1->as = AMOVFD;
        q1->from = q->to;
        q1->to = q1->from;
        q1->link = q->link;
        q->link = q1;
    }
    break;

```

Uses P 36c and prg() 43a.

```

⟨datblk() other locals 193c⟩+≡ (49b) <52a>
long fl;

```

```

⟨datblk() switch type of destination cases 193d⟩+≡ (49b) <50e>
case D_FCONST:
    switch(c) {
    default:
    case 4:
        fl = ieeeedtof(p->to.ieee);
        cast = (char*)&fl;
        for(; i<c; i++) {
            buf.dbuf[l] = cast[fnuxi4[i]];
            l++;
        }
    }

```

```

    }
    break;
case 8:
    cast = (char*)p->to.ieee;
    for(; i<c; i++) {
        buf.dbuf[l] = cast[fnuxi8[i]];
        l++;
    }
    break;
}
break;

```

Uses buf 234b, fnuxi4 194a, fnuxi8 194b, and ieedtof() 189c.

<global fnuxi4 194a>≡ (287a)
char fnuxi4[4];

<global fnuxi8 194b>≡ (287a)
char fnuxi8[8];

<nuxiinit() in loop i, fnuxi initialisation 194c>≡ (51e)
fnuxi4[i] = c;
if(debug['d'] == 0){
 fnuxi8[i] = c;
 fnuxi8[i+4] = c+4;
}
else{
 fnuxi8[i] = c+4; /* ms word first, then ls, even in little endian mode */
 fnuxi8[i+4] = c;
}

Uses debug 218a, fnuxi4 194a, and fnuxi8 194b.

Common float instructions

<buildop() switch opcode r for ranges cases 194d>+≡ (104c) <139a 205c>

```

case AADDF:
    oprange[AADDD] = oprange[r];
    oprange[ASUBF] = oprange[r];
    oprange[ASUBD] = oprange[r];
    oprange[AMULF] = oprange[r];
    oprange[AMULD] = oprange[r];
    oprange[ADIVF] = oprange[r];
    oprange[ADIVD] = oprange[r];
    oprange[AMOVFD] = oprange[r];
    oprange[AMOVDF] = oprange[r];
    break;

case ACMPF:
    oprange[ACMPD] = oprange[r];
    break;

case AMOVF:
    oprange[AMOVD] = oprange[r];
    break;

case AMOVFW:
    oprange[AMOVWF] = oprange[r];
    oprange[AMOVWD] = oprange[r];
    oprange[AMOVDW] = oprange[r];
    break;

```

Uses oprange 100c.

<optab entries 195a>+≡

(99b) <181e 198d>

```
{ AMOVF, C_FREG, C_NONE, C_FEXT, 50, 4, REGSB },
{ AMOVF, C_FREG, C_NONE, C_FAUTO, 50, 4, REGSP },
{ AMOVF, C_FREG, C_NONE, C_FOREG, 50, 4, 0 },

{ AMOVF, C_FEXT, C_NONE, C_FREG, 51, 4, REGSB },
{ AMOVF, C_FAUTO,C_NONE, C_FREG, 51, 4, REGSP },
{ AMOVF, C_FOREG,C_NONE, C_FREG, 51, 4, 0 },

{ AMOVF, C_FREG, C_NONE, C_LEXT, 52, 12, REGSB, LTO },
{ AMOVF, C_FREG, C_NONE, C_LAUTO, 52, 12, REGSP, LTO },
{ AMOVF, C_FREG, C_NONE, C_LOREG, 52, 12, 0, LTO },

{ AMOVF, C_LEXT, C_NONE, C_FREG, 53, 12, REGSB, LFROM },
{ AMOVF, C_LAUTO,C_NONE, C_FREG, 53, 12, REGSP, LFROM },
{ AMOVF, C_LOREG,C_NONE, C_FREG, 53, 12, 0, LFROM },

{ AMOVF, C_FREG, C_NONE, C_ADDR, 68, 8, 0, LTO },
{ AMOVF, C_ADDR, C_NONE, C_FREG, 69, 8, 0, LFROM },

{ AADDF, C_FREG, C_NONE, C_FREG, 54, 4, 0 },
{ AADDF, C_FREG, C_REG, C_FREG, 54, 4, 0 },
{ AADDF, C_FCON, C_NONE, C_FREG, 54, 4, 0 },
{ AADDF, C_FCON, C_REG, C_FREG, 54, 4, 0 },
{ AMOVF, C_FCON, C_NONE, C_FREG, 54, 4, 0 },
{ AMOVF, C_FREG, C_NONE, C_FREG, 54, 4, 0 },

{ ACMPF, C_FREG, C_REG, C_NONE, 54, 4, 0 },
{ ACMPF, C_FCON, C_REG, C_NONE, 54, 4, 0 },

{ AMOVFW, C_FREG, C_NONE, C_REG, 55, 4, 0 },
{ AMOVFW, C_REG, C_NONE, C_FREG, 55, 4, 0 },
```

Uses C_ADDR 180a, C_FAUTO 190f, C_FCON 190a, C_FEXT 190d, C_FOREG 191b, C_FREG 190a, C_LAUTO 116c, C_LEXT 115b, C_LOREG 114a, C_NONE 101a, C_REG 101a, LFROM 141c, and LTO 141c.

<function regoff(arm) 195b>≡

(289c)

```
long
regoff(Adr *a)
{
    instoffset = 0;
    aclass(a);
    return instoffset;
}
```

Uses aclass() 101b and instoffset 112f.

<asmout() switch on type cases 195c>+≡

(108b) <182a 196a>

```
case 50: /* floating point store */
    v = regoff(&p->to);
    r = p->to.reg;
    <asmout() adjust maybe r to rule param 131c>
    o1 = ofsr(p->as, p->from.reg, v, r, p->scond, p);
    break;
```

Uses ofsr() 195d and regoff() 195b.

<function ofsr(arm) 195d>≡

(288)

```
long
ofsr(int a, int r, long v, int b, int sc, Prog *p)
{
```

```

long o;

if(vfp)
    return ovfpmem(a, r, v, b, sc, p);

o = (sc & C_SCOND) << 28;
if(sc & C_SBIT)
    diag(".S on FLDR/FSTR instruction");
if(!(sc & C_PBIT))
    o |= 1 << 24;
if(sc & C_WBIT)
    o |= 1 << 21;
o |= (6<<25) | (1<<24) | (1<<23);
if(v < 0) {
    v = -v;
    o ^= 1 << 23;
}
if(v & 3)
    diag("odd offset for floating point op: %ld\nnP", v, p);
else if(v >= (1<<10))
    diag("literal span too large: %ld\nnP", v, p);
o |= (v>>2) & 0xFF;
o |= b << 16;
o |= r << 12;
o |= 1 << 8;

switch(a) {
default:
    diag("bad fst %A", a);
case AMOVD:
    o |= 1<<15;
case AMOVF:
    break;
}
return o;
}

```

Uses `diag()` 229d, `ovfpmem()` 199a, and `vfp` 191f.

```

<asmout() switch on type cases 196a>+≡ (108b) <195c 196b>
case 51: /* floating point load */
    v = regoff(&p->from);
    r = p->from.reg;
    <asmout() adjust maybe r to rule param 131c>
    o1 = ofsr(p->as, p->to.reg, v, r, p->scond, p) | (1<<20);
    break;

```

Uses `ofsr()` 195d and `regoff()` 195b.

```

<asmout() switch on type cases 196b>+≡ (108b) <196a 197a>
case 52: /* floating point store, long offset UGLY */
    o1 = omvl(p, &p->to, REGTMP);
    if(!o1)
        break;
    r = p->to.reg;
    <asmout() adjust maybe r to rule param 131c>
    o2 = oprrr(AADD, p->scond) | (REGTMP << 12) | (REGTMP << 16) | r;
    o3 = ofsr(p->as, p->from.reg, 0, REGTMP, p->scond, p);
    break;

```

Uses `ofsr()` 195d, `omvl()` 125c, and `opr` 120a.

```

<asmout() switch on type cases 197a>+≡ (108b) <196b 197c>
case 53: /* floating point load, long offset UGLY */
    o1 = omvl(p, &p->from, REGTMP);
    if(!o1)
        break;
    r = p->from.reg;
    <asmout() adjust maybe r to rule param 131c>
    o2 = oprrr(AADD, p->scond) | (REGTMP << 12) | (REGTMP << 16) | r;
    o3 = ofsr(p->as, p->to.reg, 0, REGTMP, p->scond, p) | (1<<20);
    break;

```

Uses ofsr() 195d, omvl() 125c, and oprrr() 120a.

Old ARM 7500 floating point

```

<oprrr() switch cases 197b>+≡ (120a) <139d 206g>
/* old arm 7500 fp using coprocessor 1 (1<<8) */
case AADD: return o | (0xe<<24) | (0x0<<20) | (1<<8) | (1<<7);
case AADD: return o | (0xe<<24) | (0x0<<20) | (1<<8);
case AMULD: return o | (0xe<<24) | (0x1<<20) | (1<<8) | (1<<7);
case AMULF: return o | (0xe<<24) | (0x1<<20) | (1<<8);
case ASUBD: return o | (0xe<<24) | (0x2<<20) | (1<<8) | (1<<7);
case ASUBF: return o | (0xe<<24) | (0x2<<20) | (1<<8);
case ADIVD: return o | (0xe<<24) | (0x4<<20) | (1<<8) | (1<<7);
case ADIVF: return o | (0xe<<24) | (0x4<<20) | (1<<8);
/* arguably, ACPMF should expand to RNDF, CMPD */
case ACPMD:
case ACPMF: return o | (0xe<<24) | (0x9<<20) | (0xF<<12) | (1<<8) | (1<<4);

case AMOVF:
case AMOVDF: return o | (0xe<<24) | (0x0<<20) | (1<<15) | (1<<8);
case AMOVD:
case AMOVFD: return o | (0xe<<24) | (0x0<<20) | (1<<15) | (1<<8) | (1<<7);

case AMOVWF: return o | (0xe<<24) | (0<<20) | (1<<8) | (1<<4);
case AMOVWD: return o | (0xe<<24) | (0<<20) | (1<<8) | (1<<4) | (1<<7);
case AMOVFW: return o | (0xe<<24) | (1<<20) | (1<<8) | (1<<4);
case AMOVFDW: return o | (0xe<<24) | (1<<20) | (1<<8) | (1<<4) | (1<<7);

```

```

<asmout() switch on type cases 197c>+≡ (108b) <197a 198a>
case 54: /* floating point arith */
    o1 = oprrr(p->as, p->scond);
    if(p->from.type == D_FCONST) {
        rf = chipfloat(p->from.ieee);
        if(rf < 0){
            diag("invalid floating-point immediate\n%P", p);
            rf = 0;
        }
        rf |= (1<<3);
    } else
        rf = p->from.reg;
    rt = p->to.reg;
    r = p->reg;
    if(p->to.type == D_NONE)
        rt = 0; /* CMP[FD] */
    else if(o1 & (1<<15))
        r = 0; /* monadic */
    else
        <asmout() adjust r 119c>
    o1 |= rf | (r<<16) | (rt<<12);

```

```
break;
```

Uses `chipfloat()` 193a, `diag()` 229d, and `oprerr()` 120a.

```
<asmout() switch on type cases 198a>+≡ (108b) <197c 198b>
case 55: /* floating point fix and float */
    o1 = oprerr(p->as, p->scond);
    rf = p->from.reg;
    rt = p->to.reg;
    if(p->to.type == D_NONE){
        rt = 0;
        diag("to.type==D_NONE (asm/fp)");
    }
    if(p->from.type == D_REG)
        o1 |= (rf<<12) | (rt<<16);
    else
        o1 |= rf | (rt<<12);
    break;
```

Uses `diag()` 229d and `oprerr()` 120a.

```
<asmout() switch on type cases 198b>+≡ (108b) <198a 198c>
case 68: /* floating point store -> ADDR */
    o1 = omvl(p, &p->to, REGTMP);
    if(!o1)
        break;
    o2 = ofsr(p->as, p->from.reg, 0, REGTMP, p->scond, p);
    break;
```

Uses `ofsr()` 195d and `omvl()` 125c.

```
<asmout() switch on type cases 198c>+≡ (108b) <198b 198e>
case 69: /* floating point load <- ADDR */
    o1 = omvl(p, &p->from, REGTMP);
    if(!o1)
        break;
    o2 = ofsr(p->as, p->to.reg, 0, REGTMP, p->scond, p) | (1<<20);
    break;
```

Uses `ofsr()` 195d and `omvl()` 125c.

```
<optab entries 198d>+≡ (99b) <195a 199b>
{ AMOVW, C_REG, C_NONE, C_FCR, 56, 4 },
{ AMOVW, C_FCR, C_NONE, C_REG, 57, 4 },
```

Uses `C_FCR` 190a, `C_NONE` 101a, and `C_REG` 101a.

```
<asmout() switch on type cases 198e>+≡ (108b) <198c 198f>
/* old arm 7500 fp using coprocessor 1 (1<<8) */
case 56: /* move to FP[CS]R */
    o1 = ((p->scond & C_SCOND) << 28) | (0xe << 24) | (1<<8) | (1<<4);
    o1 |= ((p->to.reg+1)<<21) | (p->from.reg << 12);
    break;
```

```
<asmout() switch on type cases 198f>+≡ (108b) <198e 199c>
case 57: /* move from FP[CS]R */
    o1 = ((p->scond & C_SCOND) << 28) | (0xe << 24) | (1<<8) | (1<<4);
    o1 |= ((p->from.reg+1)<<21) | (p->to.reg<<12) | (1<<20);
    break;
```

VFP hardware instructions, 51 -f

<function ovfpmem(arm) 199a>≡

(288)

```

long
ovfpmem(int a, int r, long v, int b, int sc, Prog *p)
{
    long o;

    o = (sc & C_SCOND) << 28;
    if(sc & (C_SBIT|C_PBIT|C_WBIT))
        diag(".S/.P/.W on VLDR/VSTR instruction");
    o |= 0xd<<24 | (1<<23);
    if(v < 0) {
        v = -v;
        o ^= 1 << 23;
    }
    if(v & 3)
        diag("odd offset for floating point op: %ld\n%P", v, p);
    else if(v >= (1<<10))
        diag("literal span too large: %ld\n%P", v, p);
    o |= (v>>2) & 0xFF;
    o |= b << 16;
    o |= r << 12;
    switch(a) {
    default:
        diag("bad fst %A", a);
    case AMOVD:
        o |= 0xb<<8;
        break;
    case AMOVF:
        o |= 0xa<<8;
        break;
    }
    return o;
}

```

Uses *diag()* 229d.

<optab entries 199b>+≡

(99b) <198d 205d>

```

{ AADD, C_FREG, C_NONE, C_FREG, 74, 4, 0, VFP },
{ AADD, C_FREG, C_REG, C_FREG, 74, 4, 0, VFP },
{ AMOVF, C_FREG, C_NONE, C_FREG, 74, 4, 0, VFP },
{ ACMPF, C_FREG, C_REG, C_NONE, 75, 8, 0, VFP },
{ ACMPF, C_FCON, C_REG, C_NONE, 75, 8, 0, VFP },
{ AMOVFW, C_FREG, C_NONE, C_REG, 76, 8, 0, VFP },
{ AMOVFW, C_REG, C_NONE, C_FREG, 76, 8, 0, VFP },

```

Uses *C_FCON* 190a, *C_FREG* 190a, *C_NONE* 101a, *C_REG* 101a, and *VFP* 191d.

<asmout() switch on type cases 199c>+≡

(108b) <198f 200a>

```

/* VFP ops: */
case 74: /* vfp floating point arith */
    o1 = opvfprrr(p->as, p->scond);
    rf = p->from.reg;
    if(p->from.type == D_FCONST) {
        diag("invalid floating-point immediate\n%P", p);
        rf = 0;
    }
    rt = p->to.reg;
    r = p->reg;
    <asmout() adjust r 119c>
    o1 |= rt<<12;

```

```

if(((o1>>20)&0xf) == 0xb)
    o1 |= rf<<0;
else
    o1 |= r<<16 | rf<<0;
break;

```

Uses `diag()` 229d and `opvfprrr()` 200c.

`<asmout() switch on type cases 200a>+≡ (108b) <199c 200b>`

```

case 75: /* vfp floating point compare */
o1 = opvfprrr(p->as, p->scond);
rf = p->from.reg;
if(p->from.type == D_FCONST) {
    if(p->from.ieee->h != 0 || p->from.ieee->l != 0)
        diag("invalid floating-point immediate\n%P", p);
    o1 |= 1<<16;
    rf = 0;
}
rt = p->reg;
o1 |= rt<<12 | rf<<0;
o2 = 0x0ef1fa10; /* MRS APSR_nzcv, FPSCR */
o2 |= (p->scond & C_SCOND) << 28;
break;

```

Uses `diag()` 229d and `opvfprrr()` 200c.

`<asmout() switch on type cases 200b>+≡ (108b) <200a 205e>`

```

case 76: /* vfp floating point fix and float */
o1 = opvfprrr(p->as, p->scond);
rf = p->from.reg;
rt = p->to.reg;
if(p->from.type == D_REG) {
    o2 = o1 | rt<<12 | rf<<0;
    o1 = 0x0e000a10; /* VMOV F,R */
    o1 |= (p->scond & C_SCOND) << 28 | rt<<16 | rf<<12;
} else {
    o1 |= FREGTMP<<12 | rf<<0;
    o2 = 0x0e100a10; /* VMOV R,F */
    o2 |= (p->scond & C_SCOND) << 28 | FREGTMP<<16 | rt<<12;
}
break;

```

Uses `opvfprrr()` 200c.

`<function opvfprrr(arm) 200c>≡ (288)`

```

long
opvfprrr(int a, int sc)
{
    long o;

    o = (sc & C_SCOND) << 28;
    if(sc & (C_SBIT|C_PBIT|C_WBIT))
        diag(".S/.P/.W on vfp instruction");
    o |= 0xe<<24;
    switch(a) {
case AMOVWD: return o | 0xb<<8 | 0xb<<20 | 1<<6 | 0x8<<16 | 1<<7;
case AMOVWF: return o | 0xa<<8 | 0xb<<20 | 1<<6 | 0x8<<16 | 1<<7;
case AMOVDW: return o | 0xb<<8 | 0xb<<20 | 1<<6 | 0xd<<16 | 1<<7;
case AMOVFW: return o | 0xa<<8 | 0xb<<20 | 1<<6 | 0xd<<16 | 1<<7;
case AMOVFD: return o | 0xa<<8 | 0xb<<20 | 1<<6 | 0x7<<16 | 1<<7;
case AMOVDF: return o | 0xb<<8 | 0xb<<20 | 1<<6 | 0x7<<16 | 1<<7;
case AMOVF: return o | 0xa<<8 | 0xb<<20 | 1<<6 | 0x0<<16 | 0<<7;

```

```

case AMOVD: return o | 0xb<<8 | 0xb<<20 | 1<<6 | 0x0<<16 | 0<<7;
case ACMPF: return o | 0xa<<8 | 0xb<<20 | 1<<6 | 0x4<<16 | 0<<7;
case ACMPD: return o | 0xb<<8 | 0xb<<20 | 1<<6 | 0x4<<16 | 0<<7;
case AADDF: return o | 0xa<<8 | 0x3<<20;
case AADD: return o | 0xb<<8 | 0x3<<20;
case ASUBF: return o | 0xa<<8 | 0x3<<20 | 1<<6;
case ASUBD: return o | 0xb<<8 | 0x3<<20 | 1<<6;
case AMULF: return o | 0xa<<8 | 0x2<<20;
case AMULD: return o | 0xb<<8 | 0x2<<20;
case ADIVF: return o | 0xa<<8 | 0x8<<20;
case ADIVD: return o | 0xb<<8 | 0x8<<20;
}
diag("bad vfp rrr %d", a);
prasm(curp);
return 0;
}

```

Uses curp [36g](#), diag() [229d](#), and prasm() [219b](#).

12.6.2 Division

ADIV rewriting

<noops() second pass switch opcode cases 201>+≡

(87) <193b

```

case ADIV:
case ADIVU:
case AMOD:
case AMODU:
    <noops() second pass, ADIV rewrite, case ADIV and so on, if -M 205b>
    if(p->from.type != D_REG)
        break;
    if(p->to.type != D_REG)
        break;
    // else
    q1 = p;

    q = prg();
    q->link = p->link;
    p->link = q;
    p = q;

    /* MOV a,4(SP) */
    p->as = AMOVW;
    p->line = q1->line;
    p->from.type = D_REG;
    p->from.reg = q1->from.reg;
    p->to.type = D_OREG;
    p->to.reg = REGSP;
    p->to.offset = 4;

    q = prg();
    q->link = p->link;
    p->link = q;
    p = q;

    /* MOV b,REGTMP */
    p->as = AMOVW;
    p->line = q1->line;
    p->from.type = D_REG;
    p->from.reg = q1->reg;

```

```

if(q1->reg == R_NONE)
    p->from.reg = q1->to.reg;
p->to.type = D_REG;
p->to.reg = REGTMP;
p->to.offset = 0;

q = prg();
q->link = p->link;
p->link = q;
p = q;

/* CALL appropriate */
p->as = ABL;
p->line = q1->line;
p->to.type = D_BRANCH;
p->cond = p;
switch(o) {
case ADIV:
    p->cond = prog_div;
    p->to.sym = sym_div;
    break;
case ADIVU:
    p->cond = prog_divu;
    p->to.sym = sym_divu;
    break;
case AMOD:
    p->cond = prog_mod;
    p->to.sym = sym_mod;
    break;
case AMODU:
    p->cond = prog_modu;
    p->to.sym = sym_modu;
    break;
}

q = prg();
q->link = p->link;
p->link = q;
p = q;

/* MOV REGTMP, b */
p->as = AMOVW;
p->line = q1->line;
p->from.type = D_REG;
p->from.reg = REGTMP;
p->from.offset = 0;
p->to.type = D_REG;
p->to.reg = q1->to.reg;

q = prg();
q->link = p->link;
p->link = q;
p = q;

/* ADD $8,SP */
p->as = AADD;
p->from.type = D_CONST;
p->from.reg = R_NONE;
p->from.offset = 8;
p->reg = R_NONE;

```

```

p->to.type = D_REG;
p->to.reg = REGSP;

/* SUB $8,SP */
q1->as = ASUB;
q1->from.type = D_CONST;
q1->from.offset = 8;
q1->from.reg = R_NONE;
q1->reg = R_NONE;
q1->to.type = D_REG;
q1->to.reg = REGSP;
break;

```

Uses `prg()` 43a, `prog_div` 203a, `prog_divu` 203b, `prog_mod` 203c, `prog_modu` 203d, `sym_div-8` 203e, `sym_divu-9` 203f, `sym_mod-10` 203g, and `sym_modu-11` 203h.

<global prog_div(arm) 203a>≡ (291b)
Prog* prog_div;

<global prog_divu(arm) 203b>≡ (291b)
Prog* prog_divu;

<global prog_mod(arm) 203c>≡ (291b)
Prog* prog_mod;

<global prog_modu(arm) 203d>≡ (291b)
Prog* prog_modu;

`initdiv()`

<global sym_div(arm) 203e>≡ (291b)
static Sym* sym_div;

<global sym_divu(arm) 203f>≡ (291b)
static Sym* sym_divu;

<global sym_mod(arm) 203g>≡ (291b)
static Sym* sym_mod;

<global sym_modu(arm) 203h>≡ (291b)
static Sym* sym_modu;

<constant SIGNINTERN(arm) 203i>≡ (279)
#define SIGNINTERN (1729*325*1729)

<function sigdiv(arm) 203j>≡ (291b)
static void
sigdiv(char *n)
{
Sym *s;

s = lookup(n, 0);
if(s->type == STEXT){
if(s->sig == 0)
s->sig = SIGNINTERN;
}
else if(s->type == SNONE || s->type == SXREF)
s->type = SUNDEF;
}

Uses `SIGNINTERN` 279, `SNONE` 33b, `STEXT` 156d, `SUNDEF` 33b, `SXREF`, and `lookup()` 32a.

```

<function divsig(arm) 204a>≡ (291b)
void
divsig(void)
{
    sigdiv("_div");
    sigdiv("_divu");
    sigdiv("_mod");
    sigdiv("_modu");
}

```

Uses `sigdiv()` 203j.

```

<function sdiv(arm) 204b>≡ (291b)
static void
sdiv(Sym *s)
{
    if(s->type == SNONE || s->type == SXREF){
        /* undefsym(s); */
        s->type = SXREF;
        if(s->sig == 0)
            s->sig = SIGNINTERN;
        s->subtype = SIMPORT;
    }
    else if(s->type != STEXT)
        diag("undefined: %s", s->name);
}

```

Uses `SIGNINTERN` 279, `SIMPORT`, `SNONE` 33b, `STEXT` 156d, `SXREF`, and `diag()` 229d.

```

<function initdiv(arm) 204c>≡ (291b)
void
initdiv(void)
{
    Sym *s2, *s3, *s4, *s5;
    Prog *p;

    if(prog_div != P)
        return;
    sym_div = s2 = lookup("_div", 0);
    sym_divu = s3 = lookup("_divu", 0);
    sym_mod = s4 = lookup("_mod", 0);
    sym_modu = s5 = lookup("_modu", 0);
    if(dlm) {
        sdiv(s2); if(s2->type == SXREF) prog_div = UP;
        sdiv(s3); if(s3->type == SXREF) prog_divu = UP;
        sdiv(s4); if(s4->type == SXREF) prog_mod = UP;
        sdiv(s5); if(s5->type == SXREF) prog_modu = UP;
    }
    for(p = firstp; p != P; p = p->link)
        if(p->as == ATEXT) {
            if(p->from.sym == s2)
                prog_div = p;
            if(p->from.sym == s3)
                prog_divu = p;
            if(p->from.sym == s4)
                prog_mod = p;
            if(p->from.sym == s5)
                prog_modu = p;
        }
    if(prog_div == P) {
        diag("undefined: %s", s2->name);
        prog_div = curtext;
    }
}

```

```

}
if(prog_divu == P) {
    diag("undefined: %s", s3->name);
    prog_divu = curtext;
}
if(prog_mod == P) {
    diag("undefined: %s", s4->name);
    prog_mod = curtext;
}
if(prog_modu == P) {
    diag("undefined: %s", s5->name);
    prog_modu = curtext;
}
}

```

Uses P 36c, SXREF, UP 279, curtext 36f, diag() 229d, dlm 170a, firstp 36a, lookup() 32a, prog_div 203a, prog_divu 203b, prog_mod 203c, prog_modu 203d, sdiv() 204b, sym_div-8 203e, sym_divu-9 203f, sym_mod-10 203g, and sym_modu-11 203h.

`<noops() first pass switch opcode cases 205a>+≡ (87) <90`

```

case ADIV:
case ADIVU:
case AMOD:
case AMODU:
    if(prog_div == P)
        initdiv();
    if(curtext != P)
        curtext->mark &= ~LEAF;
    continue; // no q = p;

```

Uses LEAF 279, P 36c, curtext 36f, initdiv() 204c, and prog_div 203a.

Kernel emulation, 51 -M

`<noops() second pass, ADIV rewrite, case ADIV and so on, if -M 205b>≡ (201)`

```

if(debug['M'])
    break;

```

Uses debug 218a.

`<buildop() switch opcode r for ranges cases 205c>+≡ (104c) <194d 206a>`

```

case ADIV:
    oprange[AMOD] = oprange[r];
    oprange[AMODU] = oprange[r];
    oprange[ADIVU] = oprange[r];
    break;

```

Uses oprange 100c.

`<optab entries 205d>+≡ (99b) <199b 206b>`

```

{ ADIV, C_REG, C_REG, C_REG, 16, 4 },
{ ADIV, C_REG, C_NONE, C_REG, 16, 4 },

```

Uses C_NONE 101a and C_REG 101a.

`<asmout() switch on type cases 205e>+≡ (108b) <200b 206f>`

```

case 16: /* div r, [r,]r */
    o1 = 0xf << 28;
    o2 = 0;
    break;

```

12.6.3 Long multiplication

```
<buildop() switch opcode r for ranges cases 206a>+≡ (104c) <205c 206h>
    case AMULL:
        oprange[AMULA] = oprange[r];
        oprange[AMULAL] = oprange[r];
        oprange[AMULLU] = oprange[r];
        oprange[AMULALU] = oprange[r];
        break;
```

Uses oprange 100c.

```
<optab entries 206b>+≡ (99b) <205d 206i>
    { AMULL, C_REG, C_REG, C_REGREG, 17, 4 },
```

Uses C_REG 101a and C_REGREG 206c.

```
<Operand_class cases 206c>+≡ (101a) <190a 207d>
    C_REGREG, // D_REGREG
```

```
<aclass() switch type cases 206d>+≡ (101b) <190b 207e>
    case D_REGREG:
        return C_REGREG;
```

Uses C_REGREG 206c.

```
<asmout() other locals 206e>+≡ (108b) <?? 226a>
    int rt2;
```

```
<asmout() switch on type cases 206f>+≡ (108b) <205e 207a>
    case 17:
        o1 = oprrr(p->as, p->scond);
        rf = p->from.reg;
        rt = p->to.reg;
        rt2 = p->to.offset;
        r = p->reg;
        o1 |= (rf<<8) | r | (rt<<16) | (rt2<<12);
        break;
```

Uses oprrr() 120a.

```
<opr() switch cases 206g>+≡ (120a) <197b>
    case AMULA: return o | (0x1<<21) | (0x9<<4);
    case AMULLU: return o | (0x4<<21) | (0x9<<4);
    case AMULL: return o | (0x6<<21) | (0x9<<4);
    case AMULALU: return o | (0x5<<21) | (0x9<<4);
    case AMULAL: return o | (0x7<<21) | (0x9<<4);
```

12.6.4 Multiple registers move

```
<buildop() switch opcode r for ranges cases 206h>+≡ (104c) <206a 213c>
    case AMOVM:
        break;
```

```
<optab entries 206i>+≡ (99b) <206b 207c>
    { AMOVM, C_LCON, C_NONE, C_SOREG, 38, 4 },
    { AMOVM, C_SOREG, C_NONE, C_LCON, 39, 4 },
```

Uses C_LCON 113a, C_NONE 101a, and C_SOREG 114a.

```

<asmout() switch on type cases 207a)+≡ (108b) <206f 207b>
case 38: /* movm $con,oreg -> stm */
    o1 = (0x4 << 25);
    o1 |= p->from.offset & 0xffff;
    o1 |= p->to.reg << 16;
    aclass(&p->to);
    goto movm;

```

Uses aclass() 101b.

```

<asmout() switch on type cases 207b)+≡ (108b) <207a 207f>
case 39: /* movm oreg,$con -> ldm */
    o1 = (0x4 << 25) | (1 << 20);
    o1 |= p->to.offset & 0xffff;
    o1 |= p->from.reg << 16;
    aclass(&p->from);

```

```

movm:
    if(instoffset != 0)
        diag("offset must be zero in MOVm");
    o1 |= (p->scond & C_SCOND) << 28;
    if(p->scond & C_PBIT)
        o1 |= 1 << 24;
    if(p->scond & C_UBIT)
        o1 |= 1 << 23;
    if(p->scond & C_SBIT)
        o1 |= 1 << 22;
    if(p->scond & C_WBIT)
        o1 |= 1 << 21;
    break;

```

Uses aclass() 101b, diag() 229d, and instoffset 112f.

12.6.5 Status register

```

<optab entries 207c)+≡ (99b) <206i 210a>
{ AMOVW, C_PSR, C_NONE, C_REG, 35, 4 },
{ AMOVW, C_REG, C_NONE, C_PSR, 36, 4 },
{ AMOVW, C_RCON, C_NONE, C_PSR, 37, 4 },

```

Uses C_NONE 101a, C_PSR 207d, C_RCON 113a, and C_REG 101a.

```

<Operand_class cases 207d)+≡ (101a) <206c
C_PSR, // D_PSR

```

```

<aclass() switch type cases 207e)+≡ (101b) <206d
case D_PSR:
    return C_PSR;

```

Uses C_PSR 207d.

```

<asmout() switch on type cases 207f)+≡ (108b) <207b 208a>
case 35: /* mov PSR,R */
    o1 = (2<<23) | (0xf<<16) | (0<<0);
    o1 |= (p->scond & C_SCOND) << 28;
    o1 |= (p->from.reg & 1) << 22;
    o1 |= p->to.reg << 12;
    break;

```

```

⟨asmout() switch on type cases 208a⟩+≡ (108b) <207f 208b>
case 36: /* mov R,PSR */
    o1 = (2<<23) | (0x29f<<12) | (0<<4);
    if(p->scond & C_FBIT)
        o1 ^= 0x010 << 12;
    o1 |= (p->scond & C_SCOND) << 28;
    o1 |= (p->to.reg & 1) << 22;
    o1 |= p->from.reg << 0;
    break;

```

```

⟨asmout() switch on type cases 208b⟩+≡ (108b) <208a 210b>
case 37: /* mov $con,PSR */
    aclass(&p->from);
    o1 = (2<<23) | (0x29f<<12) | (0<<4);
    if(p->scond & C_FBIT)
        o1 ^= 0x010 << 12;
    o1 |= (p->scond & C_SCOND) << 28;
    o1 |= immrot(instoffset);
    o1 |= (p->to.reg & 1) << 22;
    o1 |= p->from.reg << 0;
    break;

```

Uses aclass() 101b, immrot() 113c, and instoffset 112f.

12.6.6 Half words and bytes moves since ARMv4

```

⟨global armv4(arm) 208c⟩≡ (283b)
bool armv4;

```

```

⟨buildop() initialize flags 208d⟩+≡ (104c) <191g
armv4 = !debug['h'];

```

Uses armv4 208c and debug 218a.

```

⟨Optab_flag cases 208e⟩+≡ (141c) <191d
V4      = 1<<3, /* arm v4 arch */

```

```

⟨buildop() adjust optab if flags, remove certain rules 208f⟩+≡ (104c) <191h
if((optab[n].flag & V4) && !armv4) {
    optab[n].as = AXXX;
    break;
}

```

Uses V4 208e, armv4 208c, and optab 99b.

```

⟨ocmp() if v4 flag on p1 or p2 208g⟩≡ (105b)
n = (p2->flag&V4) - (p1->flag&V4); /* architecture version */
if(n)
    return n;

```

Uses V4 208e.

```

⟨Operand_class cases, in C_xEXT, half case 208h⟩≡ (115b)
C_HEXTEXT,

```

```

⟨aclass() if immhalf for N_EXTERN symbol 208i⟩≡ (115c)
if(immhalf(instoffset))
    return C_HEXTEXT;

```

Uses C_HEXTEXT 208h, immhalf() 209h, and instoffset 112f.

```

⟨cmp() switch on a, the operand class in optab rule, cases 209a) +≡ (103c) <117e 209c>
    case C_HFEXT:
        return b == C_HEXT || b == C_FEXT;
    case C_FEXT:
    case C_HEXT:
        return b == C_HFEXT;

```

Uses C_FEXT 190d, C_HEXT 208h, and C_HFEXT 190d.

```

⟨Operand_class cases, in C_xAUTO, half case 209b) ≡ (116c)
    C_HAUTO, /* halfword insn offset (-0xff to 0xff) */

```

```

⟨cmp() switch on a, the operand class in optab rule, cases 209c) +≡ (103c) <209a 209g>
    case C_HFAUTO:
        return b == C_HAUTO || b == C_FAUTO;
    case C_FAUTO:
    case C_HAUTO:
        return b == C_HFAUTO;

```

Uses C_FAUTO 190f, C_HAUTO 209b, and C_HFAUTO 190f.

```

⟨aclass() if immhalf for N_LOCAL or N_PARAM symbol 209d) ≡ (116)
    if(immhalf(instoffset))
        return C_HAUTO;

```

Uses C_HAUTO 209b, immhalf() 209h, and instoffset 112f.

```

⟨Operand_class cases, in C_xOREG, half case 209e) ≡ (114a)
    C_HOREG,

```

```

⟨aclass() if immhalf for N_NONE symbol 209f) ≡ (114c)
    /* n.b. that immhalf() will also satisfy immrot */
    if(immhalf(instoffset))
        return C_HOREG;

```

Uses C_HOREG 209e, immhalf() 209h, and instoffset 112f.

```

⟨cmp() switch on a, the operand class in optab rule, cases 209g) +≡ (103c) <209c>
    case C_HFOREG:
        return b == C_HOREG || b == C_FOREG;
    case C_FOREG:
    case C_HOREG:
        return b == C_HFOREG;

```

Uses C_FOREG 191b, C_HFOREG 191b, and C_HOREG 209e.

```

⟨function immhalf(arm) 209h) ≡ (289c)
    static int
    immhalf(long v)
    {
        if(v >= 0 && v <= 0xff)
            return v |
                (1<<24) | /* pre indexing */
                (1<<23); /* pre indexing, up */
        if(v >= -0xff && v < 0)
            return (-v & 0xff) |
                (1<<24); /* pre indexing */
        return 0;
    }

```

<optab entries 210a>+≡

(99b) <207c 212d>

```
{ AMOVH, C_REG, C_NONE, C_HEXT, 70, 4, REGSB, V4 },
{ AMOVH, C_REG, C_NONE, C_HAUTO, 70, 4, REGSP, V4 },
{ AMOVH, C_REG, C_NONE, C_HOREG, 70, 4, 0, V4 },

{ AMOVHU, C_REG, C_NONE, C_HEXT, 70, 4, REGSB, V4 },
{ AMOVHU, C_REG, C_NONE, C_HAUTO, 70, 4, REGSP, V4 },
{ AMOVHU, C_REG, C_NONE, C_HOREG, 70, 4, 0, V4 },

{ AMOVVB, C_HEXT, C_NONE, C_REG, 71, 4, REGSB, V4 },
{ AMOVVB, C_HAUTO,C_NONE, C_REG, 71, 4, REGSP, V4 },
{ AMOVVB, C_HOREG,C_NONE, C_REG, 71, 4, 0, V4 },

{ AMOVH, C_HEXT, C_NONE, C_REG, 71, 4, REGSB, V4 },
{ AMOVH, C_HAUTO,C_NONE, C_REG, 71, 4, REGSP, V4 },
{ AMOVH, C_HOREG,C_NONE, C_REG, 71, 4, 0, V4 },

{ AMOVHU, C_HEXT, C_NONE, C_REG, 71, 4, REGSB, V4 },
{ AMOVHU, C_HAUTO,C_NONE, C_REG, 71, 4, REGSP, V4 },
{ AMOVHU, C_HOREG,C_NONE, C_REG, 71, 4, 0, V4 },
```

```
{ AMOVH, C_REG, C_NONE, C_LEXT, 72, 8, REGSB, LTO|V4 },
{ AMOVH, C_REG, C_NONE, C_LAUTO, 72, 8, REGSP, LTO|V4 },
{ AMOVH, C_REG, C_NONE, C_LOREG, 72, 8, 0, LTO|V4 },
```

```
{ AMOVHU, C_REG, C_NONE, C_LEXT, 72, 8, REGSB, LTO|V4 },
{ AMOVHU, C_REG, C_NONE, C_LAUTO, 72, 8, REGSP, LTO|V4 },
{ AMOVHU, C_REG, C_NONE, C_LOREG, 72, 8, 0, LTO|V4 },
```

```
{ AMOVVB, C_LEXT, C_NONE, C_REG, 73, 8, REGSB, LFROM|V4 },
{ AMOVVB, C_LAUTO,C_NONE, C_REG, 73, 8, REGSP, LFROM|V4 },
{ AMOVVB, C_LOREG,C_NONE, C_REG, 73, 8, 0, LFROM|V4 },
```

```
{ AMOVH, C_LEXT, C_NONE, C_REG, 73, 8, REGSB, LFROM|V4 },
{ AMOVH, C_LAUTO,C_NONE, C_REG, 73, 8, REGSP, LFROM|V4 },
{ AMOVH, C_LOREG,C_NONE, C_REG, 73, 8, 0, LFROM|V4 },
```

```
{ AMOVHU, C_LEXT, C_NONE, C_REG, 73, 8, REGSB, LFROM|V4 },
{ AMOVHU, C_LAUTO,C_NONE, C_REG, 73, 8, REGSP, LFROM|V4 },
{ AMOVHU, C_LOREG,C_NONE, C_REG, 73, 8, 0, LFROM|V4 },
```

Uses C_HAUTO 209b, C_HEXT 208h, C_HOREG 209e, C_LAUTO 116c, C_LEXT 115b, C_LOREG 114a, C_NONE 101a, C_REG 101a, LFROM 141c, LTO 141c, and V4 208e.

<asmout() switch on type cases 210b>+≡

(108b) <208b 210c>

```
/* ArmV4 ops: */
case 70: /* movh/movhu R,0(R) -> strh */
    aclass(&p->to);
    r = p->to.reg;
    <asmout() adjust maybe r to rule param 131c>
    o1 = oshr(p->from.reg, instoffset, r, p->scond);
    break;
```

Uses aclass() 101b, instoffset 112f, and oshr() 212a.

<asmout() switch on type cases 210c>+≡

(108b) <210b 211a>

```
case 71: /* movb/movh/movhu 0(R),R -> ldrsb/ldrsh/ldrh */
    aclass(&p->from);
    r = p->from.reg;
```

```

⟨asmout() adjust maybe r to rule param 131c⟩
o1 = olhr(instoffset, r, p->to.reg, p->scnd);
if(p->as == AMOVB)
    o1 ^= (1<<5)|(1<<6);
else if(p->as == AMOVH)
    o1 ^= (1<<6);
break;

```

Uses aclass() 101b, instoffset 112f, and olhr() 211c.

```

⟨asmout() switch on type cases 211a⟩+≡ (108b) <210c 211b>
case 72: /* movh/movhu R,L(R) -> strh */
    o1 = omvl(p, &p->to, REGTMP);
    if(!o1)
        break;
    r = p->to.reg;
    ⟨asmout() adjust maybe r to rule param 131c⟩
    o2 = oshrr(p->from.reg, REGTMP,r, p->scnd);
    break;

```

Uses omvl() 125c and oshrr() 212b.

```

⟨asmout() switch on type cases 211b⟩+≡ (108b) <211a 212e>
case 73: /* movb/movh/movhu L(R),R -> ldrsb/ldrsh/ldrh */
    o1 = omvl(p, &p->from, REGTMP);
    if(!o1)
        break;
    r = p->from.reg;
    ⟨asmout() adjust maybe r to rule param 131c⟩
    o2 = olhrr(REGTMP, r, p->to.reg, p->scnd);
    if(p->as == AMOVB)
        o2 ^= (1<<5)|(1<<6);
    else if(p->as == AMOVH)
        o2 ^= (1<<6);
    break;

```

Uses olhrr() 212c and omvl() 125c.

```

⟨function olhr(arm) 211c⟩≡ (288)
long
olhr(long v, int b, int r, int sc)
{
    long o;

    o = (sc & C_SCOND) << 28;
    if(sc & C_SBIT)
        diag(".S on LDRH/STRH instruction");

    if(!(sc & C_PBIT))
        o |= 1 << 24;
    if(sc & C_WBIT)
        o |= 1 << 21;
    o |= (1<<23) | (1<<20) | (0xb<<4);
    if(v < 0) {
        v = -v;
        o ^= 1 << 23;
    }
    if(v >= (1<<8))
        diag("literal span too large: %ld (R%d)\n%P", v, b, curp);
    o |= (v&0xf)|((v>>4)<<8)|(1<<22);
    o |= b << 16;
    o |= r << 12;
}

```

```

    return o;
}

```

Uses `curp` 36g and `diag()` 229d.

`<function oshr(arm) 212a>`≡ (288)

```

long
oshr(int r, long v, int b, int sc)
{
    long o;

    o = olhr(v, b, r, sc) ^ (1<<20);
    return o;
}

```

Uses `olhr()` 211c.

`<function oshrr(arm) 212b>`≡ (288)

```

long
oshrr(int r, int i, int b, int sc)
{
    return olhr(i, b, r, sc) ^ ((1<<22) | (1<<20));
}

```

Uses `olhr()` 211c.

`<function olhrr(arm) 212c>`≡ (288)

```

long
olhrr(int i, int b, int r, int sc)
{
    return olhr(i, b, r, sc) ^ (1<<22);
}

```

Uses `olhr()` 211c.

12.6.7 Shifted moves

`<optab entries 212d>`+≡ (99b) <210a 213d>

```

{ AMOVW, C_SHIFT,C_NONE, C_REG,  59, 4 },
{ AMOVBU, C_SHIFT,C_NONE, C_REG,  59, 4 },

{ AMOVB, C_SHIFT,C_NONE, C_REG,  60, 4 },

{ AMOVW, C_REG, C_NONE, C_SHIFT, 61, 4 },
{ AMOVB, C_REG, C_NONE, C_SHIFT, 61, 4 },
{ AMOVBU, C_REG, C_NONE, C_SHIFT, 61, 4 },

```

Uses `C_NONE` 101a, `C_REG` 101a, and `C_SHIFT` 121c.

`<asmout() switch on type cases 212e>`+≡ (108b) <211b 213a>

```

case 59: /* movw/bu R<<I(R),R -> ldr indexed */
    if(p->from.reg == R_NONE) {
        if(p->as != AMOVW)
            diag("byte MOV from shifter operand");
        goto mov;
    }
    if(p->from.offset&(1<<4))
        diag("bad shift in LDR");
    o1 = olrr(p->as, p->scond, p->from.offset, p->from.reg, p->to.reg);
    break;

```

Uses `diag()` 229d and `olrr()` 133a.

```

<asmout() switch on type cases 213a>+≡ (108b) <212e 213b>
case 60: /* movb R(R),R -> ldrsb indexed */
    if(p->from.reg == R_NONE) {
        diag("byte MOV from shifter operand");
        goto mov;
    }
    if(p->from.offset & (~0xf))
        diag("bad shift in LDRSB");
    o1 = olhrr(p->from.offset, p->from.reg, p->to.reg, p->scond);
    o1 ^= (1<<5)|(1<<6);
    break;

```

Uses `diag()` 229d and `olhrr()` 212c.

```

<asmout() switch on type cases 213b>+≡ (108b) <213a 213e>
case 61: /* movw/b/bu R,R<<[IR](R) -> str indexed */
    if(p->to.reg == R_NONE)
        diag("MOV to shifter operand");
    o1 = osrr(p->as, p->scond, p->from.reg, p->to.offset, p->to.reg);
    break;

```

Uses `diag()` 229d and `osrr()` 134b.

12.7 Compiler-only pseudo opcodes

```

<buildop() switch opcode r for ranges cases 213c>+≡ (104c) <206h
case ACASE:
case ABCASE:
    break;

```

```

<optab entries 213d>+≡ (99b) <212d
{ ACASE, C_REG, C_NONE, C_NONE, 62, 4 },
{ ABCASE, C_NONE, C_NONE, C_BRANCH, 63, 4 },

```

Uses `C_BRANCH` 101a, `C_NONE` 101a, and `C_REG` 101a.

```

<asmout() switch on type cases 213e>+≡ (108b) <213b 213f>
case 62: /* case R -> movw R<<2(PC),PC */
    o1 = olrr(AMOVW, p->scond, p->from.reg, REGPC, REGPC);
    o1 |= 2<<7;
    break;

```

Uses `olrr()` 133a.

```

<asmout() switch on type cases 213f>+≡ (108b) <213e
case 63: /* bcase */
    if(p->cond != P) {
        o1 = p->cond->pc;
        if(dlm)
            dynreloc(S, p->pc, 1);
    }
    break;

```

Uses `P` 36c, `S` 33b, `dlm` 170a, and `dynreloc()` 179c.

12.8 5l -E digits

```

<main() if rare condition do not set SXREF for INITENTRY, else 213g>+≡ (42c) <75e
if(*INITENTRY >= '0' && *INITENTRY <= '9') {}
else

```

`<entryvalue() if digit INITENTRY 214a>≡` (47a)

```
if(*a >= '0' && *a <= '9')
    return atolwhex(a);
```

Uses `atolwhex()` 235c.

12.9 Strings in text segment: 51 -t

`<Section cases 214b>+≡` (33b) <183a

```
SSTRING, // arm
```

`<dodata() if string in text segment 214c>≡` (92c)

```
if(debug['t']) {
    /*
     * pull out string constants
     */
    for(p = datap; p != P; p = p->link) {
        s = p->from.sym;
        if(p->to.type == D_SCONST)
            s->type = SSTRING;
    }
}
```

Uses `P` 36c, `SSTRING` 35g, `datap` 36e, and `debug` 218a.

`<dotext() other locals 214d>+≡` (94b) <95a

```
Sym *s;
long v;
int i;
```

`<dotext() if string in text segment 214e>≡` (94b)

```
if(debug['t']) {
    /*
     * add strings to text segment
     */
    c = rnd(c, 8);
    for(i=0; i<NHASH; i++)
        for(s = hash[i]; s != S; s = s->link)
            if(s->type == SSTRING) {
                v = s->value;
                v = rnd(v, 4);
                s->value = c;
                c += v;
            }
}
```

Uses `NHASH`, `S` 33b, `SSTRING` 35g, `debug` 218a, and `rnd()` 236a.

`<asmb() locals 214f>+≡` (45a) <??

```
long etext;
```

`<asmb() Text section, output strings in text segment 214g>≡` (47b)

```
/* output strings in text segment */
etext = INITTEXT + textsize;
for(t = pc; t < etext; t += sizeof(buf)-100) {
    if(etext-t > sizeof(buf)-100)
        datblk(t, sizeof(buf)-100, true);
    else
        datblk(t, etext-t, true);
}
```

Uses `INITTEXT` 40e, `buf` 234b, `datblk()` 49b, `pc` 53a, and `textsize` 94a.

`<datblk() if sstring might continue 215a>≡ (49b)`
 `if(sstring != (p->from.sym->type == SSTRING))`
 `continue;`

Uses SSTRING 35g.

`<asmsym() in symbol table iteration, switch section cases 215b>+≡ (148b) <159a`
 `case SSTRING:`
 `putsymb(s->name, 'T', s->value, s->version);`
 `continue;`

Uses SSTRING 35g and putsymb() 148a.

Chapter 13

Conclusion

You now know how the Plan 9 ARM linker `51` works—from loading and deserializing the object files produced by `5a`, through resolving symbolic references across compilation units, to the final emission of ARM machine code into an executable binary—and more generally how many linkers work.

The Plan 9 linker is unusual in that it does far more than traditional linkers: it loads AST-based object files from the assembler, performs symbol resolution, generates actual ARM machine code, adds debugging support (line numbers, symbol tables), instruments functions for profiling, and finally writes a self-contained `a.out` executable. In most toolchains, machine code generation belongs to the assembler—here it belongs to the linker, which makes `51` both a linker and a code generator. Along the way, you have seen recurring patterns: the `Sym` hash table reused from `5a`, the `Prog` linked list as the central intermediate representation, multi-pass processing to resolve addresses and patch branches, and the `autosize()` mechanism that adjusts stack frames after the code is fully known.

13.1 Patterns and techniques

These techniques apply far beyond toolchains:

- *Symbol resolution*: the linker reconciles names defined across separate object files into one global namespace. This is the linking problem in its purest form, but the same challenge—merging independently defined names—appears in dynamic library loading (`dlopen`), module systems in Python and JavaScript, microservice discovery, and even merging Git branches (reconciling changes from different authors).
- *Retroactive fixup*: `autosize()` patches each function’s prologue after the full body is known. This “measure then fix up” pattern appears in network protocols (fill in the length field after serializing the body) and PDF generation (write the cross-reference offset at the end).
- *Table-driven encoding*: machine code generation is driven by the `orange` table and bitfield packing rather than hand-coded byte sequences. Replacing code with tables makes the encoder regular, auditable, and easy to extend—the same approach used in Huffman codebooks and Unicode case-conversion tables.

13.2 Connections to other books

- ASSEMBLER book [Pad15a]: `5a` produces the AST-based object files that `51` reads. The two tools share data structures (`Prog`, `Adr`, `Sym`) and form two halves of the same pipeline.
- COMPILER book [Pad16a]: `5c` also produces object files in the same format. The linker does not distinguish between code written in C and code written in assembly—both arrive as serialized `Prog` lists.

- **KERNEL** book [Pad14]: the kernel binary is the most important output of 51. The kernel’s boot sequence relies on the memory layout that 51 produces: text, data, and BSS segments at specific addresses.
- **DEBUGGER** book [Pad16c]: the debugger uses `libmach` to read the symbol table and line number information that 51 embeds in the executable, enabling source-level debugging.
- **PROFILER** book [Pad26]: the `-p` flag instruments every function with `_profin/_profout` calls for `prof`, and the `-p -1` flag inserts `__mcount` calls for call counting.

13.3 Beyond the Plan 9 linker

Modern linkers operate in a very different context from 51. Here are some of the features they provide:

- *Relocatable object files*: most linkers (GNU `ld`, LLVM `lld`, the macOS linker) work with relocatable object files (ELF, Mach-O, COFF) that already contain machine code with relocation entries. The linker patches addresses but does not generate machine code. 51’s approach of generating machine code from an AST is unique and keeps the assembler simpler, at the cost of a more complex linker.
- *Dynamic linking*: 51 produces static executables only. Modern systems rely heavily on shared libraries (`.so`, `.dylib`, `.dll`) loaded at runtime by a dynamic linker (`ld.so`). This requires position-independent code (PIC), Global Offset Tables (GOT), Procedure Linkage Tables (PLT), and a runtime symbol resolver—substantial machinery that Plan 9 avoids entirely.
- *Link-time optimization (LTO)*: LLVM and GCC can pass intermediate representation (LLVM IR or GIMPLE) through the linker, enabling whole-program optimizations across compilation units: inlining, dead code elimination, and interprocedural analysis. This is one of the most impactful compiler optimizations in modern toolchains.
- *Linker scripts*: GNU `ld` supports a scripting language for controlling memory layout, section placement, and symbol assignments. This is essential for embedded systems and kernel development where precise control over the memory map is needed. 51 handles layout through command-line flags (`-T`, `-R`) and conventions instead.
- *Speed*: linking large C++ projects can take tens of seconds with GNU `ld`. Google’s `gold` and LLVM’s `lld` were written specifically for speed, using parallel section merging and memory-mapped I/O. `gold`, a newer linker, pushes this further with aggressive parallelism, linking large binaries in under a second.
- *Debug information*: 51 emits a simple symbol table and line number table in Plan 9’s own format. Modern linkers process DWARF debug information—a complex standard supporting variables, types, inlined functions, and optimized-out values—and can merge gigabytes of debug data from large builds.

The core job of a linker—resolving symbols and laying out sections into an executable—has not changed since the earliest systems. 51 does this in about 10 000 lines of C while also serving as the machine code generator. Modern linkers are larger because they handle dynamic linking, complex executable formats, and whole-program optimization, but the fundamental algorithms (symbol lookup, address patching, section layout) are the same ones you have studied in this book.

Appendix A

Debugging

Like 5a, 5l uses a character-indexed `debug` array. The most useful flag is 5l `-v`, which prints timing and symbol statistics. Other debug characters are repurposed for non-debug options (e.g., `-l` for library paths, `-p` for profiling instrumentation).

```
<global debug 218a>≡ (283b)
    bool debug[128];
```

```
<main() command line processing(arm) 218b>+≡ (38e) <188f
    default:
        c = ARGV();
        if(c >= 0 && c < sizeof(debug))
            debug[c]++;
        break;
```

```
<global bso 218c>≡ (283b)
    Biobuf bso;
```

```
<main() debug initialization(arm) 218d>≡ (38e)
    Binit(&bso, STDOUT, OWRITE);
    listinit(); // fmtinstall()
```

A.1 Dumpers

The linker uses Plan 9's `fmtinstall()` mechanism (see the LIBCORE book [Pad16b]) to register custom format verbs for `print()`. Each verb converts one of the linker's data structures to a string: `\%P` prints an instruction (Prog), `\%A` prints an opcode, `\%D` prints an operand (Adr), `\%N` prints a symbol kind, `\%C` a condition code, and `\%S` a string with escapes. This convention is shared with 5a and 5c, so debug output looks consistent across the entire toolchain.

```
<function listinit(arm) 218e>≡ (285c)
    void
    listinit(void)
    {
        fmtinstall('A', Aconv);
        fmtinstall('C', Cconv);
        fmtinstall('D', Dconv);
        fmtinstall('N', Nconv);
        fmtinstall('P', Pconv);
        fmtinstall('S', Sconv);
    }
```

Uses `Aconv()` 220a, `Cconv()` 223b, `Dconv()` 220b, `Nconv()` 222, `Pconv()` 219d, and `Sconv()` 224.

```

⟨pragmas varargck type 219a⟩≡ (279)
#pragma varargck    type    "A" int
#pragma varargck    type    "A" uint
#pragma varargck    type    "C" int
#pragma varargck    type    "D" Adr*
#pragma varargck    type    "N" Adr*
#pragma varargck    type    "P" Prog*
#pragma varargck    type    "S" char*

```

A.1.1 Instruction dumper: Pconv()

```

⟨function prasm(arm) 219b⟩≡ (285c)
void
prasm(Prog *p)
{
    print("%P\n", p);
}

```

```

⟨constant STRINGSZ 219c⟩≡ (278c)
STRINGSZ    = 200,

```

```

⟨function Pconv(arm) 219d⟩≡ (285c)
// Prog -> string
int
Pconv(Fmt *fp)
{
    char str[STRINGSZ], *s;
    Prog *p;
    int a;

    p = va_arg(fp->args, Prog*);
    curp = p;
    a = p->as;

    switch(a) {
    case ASWPW:
    case ASWPBU:
        sprintf(str, "(%ld) %A%C R%d,%D,%D",
            p->line, a, p->scond, p->reg, &p->from, &p->to);
        break;

    case ADATA:

    default:
        s = str;
        s += sprintf(s, "(%ld)", p->line);
        if(p->reg == R_NONE)
            sprintf(s, " %A%C %D,%D",
                a, p->scond, &p->from, &p->to);
        else
            if(p->from.type != D_FREG)
                sprintf(s, " %A%C %D,R%d,%D",
                    a, p->scond, &p->from, p->reg, &p->to);
            else
                sprintf(s, " %A%C %D,F%d,%D",
                    a, p->scond, &p->from, p->reg, &p->to);
        break;
    }
}

```

```

    return fmtstrcpy(fp, str);
}

```

Uses STRINGSZ 232d and curp 36g.

A.1.2 Opcode dumper: Aconv()

```

⟨function Aconv(arm) 220a⟩≡ (285c)
// enum<Opcode> -> string
int
Aconv(Fmt *fp)
{
    char *s;
    int a;

    a = va_arg(fp->args, int);
    s = "???";
    if(a >= AXXX && a < ALAST)
        s = anames[a];
    return fmtstrcpy(fp, s);
}

```

A.1.3 Operand dumper: Dconv()

```

⟨function Dconv(arm) 220b⟩≡ (285c)
// Adr -> string
int
Dconv(Fmt *fp)
{
    char str[STRINGSZ];
    char *op;
    Adr *a;
    long v;

    a = va_arg(fp->args, Adr*);
    switch(a->type) {

    case D_NONE:
        str[0] = '\0';
        if(a->symkind != N_NONE || a->reg != R_NONE || a->sym != S)
            sprintf(str, "%N(R%d)(NONE)", a, a->reg);
        break;

    case D_CONST:
        sprintf(str, "$%N", a);
        break;

    // ??? this is wrong with my D_ADDR; should look at a->sym
    case D_ADDR:
        if(a->reg == R_NONE)
            sprintf(str, "$%N", a);
        else
            sprintf(str, "%N(R%d)", a, a->reg);
        break;

    case D_SHIFT:
        v = a->offset;
        op = "<<>>->@" + (((v>>5) & 3) << 1);
        if(v & (1<<4))

```

```

        sprintf(str, "R%ld%c%cR%ld", v&15, op[0], op[1], (v>>8)&15);
    else
        sprintf(str, "R%ld%c%c%ld", v&15, op[0], op[1], (v>>7)&31);
    if(a->reg != R_NONE)
        sprintf(str+strlen(str), "(R%d)", a->reg);
    break;

case D_OREG:
    if(a->reg != R_NONE)
        sprintf(str, "%N(R%d)", a, a->reg);
    else
        sprintf(str, "%N", a);
    break;

case D_REG:
    sprintf(str, "R%d", a->reg);
    if(a->symkind != N_NONE || a->sym != S)
        sprintf(str, "%N(R%d)(REG)", a, a->reg);
    break;

case D_REGREG:
    sprintf(str, "(R%d,R%d)", a->reg, (int)a->offset);
    if(a->symkind != N_NONE || a->sym != S)
        sprintf(str, "%N(R%d)(REG)", a, a->reg);
    break;

case D_FREG:
    sprintf(str, "F%d", a->reg);
    if(a->symkind != N_NONE || a->sym != S)
        sprintf(str, "%N(R%d)(REG)", a, a->reg);
    break;

case D_PSR:
    switch(a->reg) {
    case 0:
        sprintf(str, "CPSR");
        break;
    case 1:
        sprintf(str, "SPSR");
        break;
    default:
        sprintf(str, "PSR%d", a->reg);
        break;
    }
    if(a->symkind != N_NONE || a->sym != S)
        sprintf(str, "%N(PSR%d)(REG)", a, a->reg);
    break;

case D_FPCR:
    switch(a->reg){
    case 0:
        sprintf(str, "FPSR");
        break;
    case 1:
        sprintf(str, "FPCR");
        break;
    default:
        sprintf(str, "FCR%d", a->reg);
        break;
    }
}

```

```

    if(a->symkind != N_NONE || a->sym != S)
        sprintf(str, "%N(FCR%d)(REG)", a, a->reg);

    break;

case D_BRANCH: /* botch */
    if(curp->cond != P) {
        v = curp->cond->pc;
        if(a->sym != S)
            sprintf(str, "{%s}%.5lux(BRANCH)", a->sym->name, v);
        else
            sprintf(str, "%.5lux(BRANCH)", v);
    } else
        if(a->sym != S)
            sprintf(str, "{%s}%ld(APC)", a->sym->name, a->offset);
        else
            sprintf(str, "%ld(APC)", a->offset);
    break;

case D_FCONST:
    sprintf(str, "$%e", ieeedtod(a->ieeee));
    break;

case D_SCONST:
    sprintf(str, "$\"%S\"", a->sval);
    break;

default:
    sprintf(str, "GOK-type(%d)", a->type);
    break;

}
return fmtstrcpy(fp, str);
}

```

Uses P 36c, S 33b, STRINGSZ 232d, curp 36g, and ieeedtod() 189d.

A.1.4 Symbol kind dumper: Nconv()

<function Nconv(arm) 222>≡

(285c)

```

int
Nconv(Fmt *fp)
{
    char str[STRINGSZ];
    Adr *a;
    Sym *s;

    a = va_arg(fp->args, Adr*);
    s = a->sym;

    switch(a->symkind) {
case N_NONE:
    sprintf(str, "%ld", a->offset);
    break;

case N_EXTERN:
    if(s == S)
        sprintf(str, "%ld(SB)", a->offset);
    else
        sprintf(str, "{%s}%.5lux+%ld(SB)", s->name, s->value, a->offset);
    }
}

```

```

        break;

case N_INTERN:
    if(s == S)
        sprintf(str, "<>+%ld(SB)", a->offset);
    else
        sprintf(str, "{%s<>}.5lux+%ld(SB)", s->name, s->value, a->offset);
    break;

case N_LOCAL:
    if(s == S)
        sprintf(str, "%ld(SP)", a->offset);
    else
        sprintf(str, "{%s}-%ld(SP)", s->name, -a->offset);
    break;

case N_PARAM:
    if(s == S)
        sprintf(str, "%ld(FP)", a->offset);
    else
        sprintf(str, "{%s}%ld(FP)", s->name, a->offset);
    break;
default:
    sprintf(str, "GOK-name(%d)", a->symkind);
    break;

}
return fmtstrcpy(fp, str);
}

```

Uses S 33b and STRINGSZ 232d.

A.1.5 Conditional execution dumper: Cconv()

```

⟨global strcond(arm) 223a⟩≡ (285c)
char* strcond[16] =
{
    ".EQ",
    ".NE",
    ".HS",
    ".LO",
    ".MI",
    ".PL",
    ".VS",
    ".VC",
    ".HI",
    ".LS",
    ".GE",
    ".LT",
    ".GT",
    ".LE",
    "",
    ".NV"
};

```

```

⟨function Cconv(arm) 223b⟩≡ (285c)
int
Cconv(Fmt *fp)
{
    char s[20];

```

```

int c;

c = va_arg(fp->args, int);
strcpy(s, strcond[c & C_SCOND]);
if(c & C_SBIT)
    strcat(s, ".S");
if(c & C_PBIT)
    strcat(s, ".P");
if(c & C_WBIT)
    strcat(s, ".W");
if(c & C_UBIT) /* ambiguous with FBIT */
    strcat(s, ".U");
return fmtstrcpy(fp, s);
}

```

Uses `strcond` [223a](#).

A.1.6 String dumper: `Sconv()`

(function `Sconv(arm)` [224](#)) ≡

(285c)

```

// string -> string
int
Sconv(Fmt *fp)
{
    int i, c;
    char str[STRINGSZ], *p, *a;

    a = va_arg(fp->args, char*);
    p = str;
    for(i=0; i<sizeof(long); i++) {
        c = a[i] & 0xff;
        if(c >= 'a' && c <= 'z' ||
           c >= 'A' && c <= 'Z' ||
           c >= '0' && c <= '9' ||
           c == ' ' || c == '%') {
            *p++ = c;
            continue;
        }
        *p++ = '\\';
        switch(c) {
            case '\\0':
                *p++ = 'z';
                continue;
            case '\\':
            case '"':
                *p++ = c;
                continue;
            case '\\n':
                *p++ = 'n';
                continue;
            case '\\t':
                *p++ = 't';
                continue;
        }
        // else
        *p++ = (c>>6) + '0';
        *p++ = ((c>>3) & 7) + '0';
        *p++ = (c & 7) + '0';
    }
    *p = '\\0';
}

```

```

    return fmtstrcpy(fp, str);
}

```

Uses `STRINGSZ` 232d.

A.2 Verbose mode: 51 -v

The `-v` flag is the most useful debug option: it prints timing information at each phase of linking (loading, resolving, code generation), making it easy to identify bottlenecks. Rather than scattering `if(debug['v'])` throughout the code, the author introduced a `DBG()` macro that expands to a conditional `mylog()` call.

```

<macro DBG 225a>≡ (279)
#define DBG if(debug['v']) mylog

```

```

<function log 225b>≡ (285a)
void mylog(char *fmt, ...) {

    va_list arg;

    va_start(arg, fmt);
    Bvprint(&bso, fmt, arg);
    va_end(arg);
    Bflush(&bso);
}

```

Uses `bso` 218c.

A.3 Objects loading debugging: 51 -W

```

<ldobj() debug 225c>≡ (55)
if(debug['W'])
    print("%P\n", p);

```

Uses `debug` 218a.

```

<ldobj() when ANAME, debug 225d>≡ (61d)
if(debug['W'])
    print(" ANAME %s\n", s->name);

```

Uses `debug` 218a.

A.4 Opcode table debugging: 51 -0

```

<oplook() debug 225e>≡ (102c)
if(debug['0']) {
    print("oplook %P %A %d %d %d\n",
        p, (int)p->as, a1, a2, a3);
}

```

Uses `debug` 218a.

A.5 Machine code generation debugging: 5l -a

The `-a` flag produces a disassembly-like listing of the generated executable: each line shows the address, the raw machine word(s) in hex, and the instruction mnemonic. Using `-a -a` (twice) additionally prints the opcode table entry type (`o->type`) that was used to encode each instruction—invaluable when debugging why an instruction was encoded incorrectly.

```
<asmout() other locals 226a>+≡ (108b) <206e
// pc (real)
long v;
```

```
<asmout() debug 226b>≡ (108b)
if(debug['a'] > 1)
    Bprint(&bso, "%2d ", o->type);
v = p->pc; // for debugging later
```

Uses `bso 218c` and `debug 218a`.

```
<asmout() when 1 generated instruction, debug 226c>≡ (108c)
if(debug['a'])
    Bprint(&bso, " %.8lux: %.8lux\t%P\n", v, o1, p);
```

Uses `bso 218c` and `debug 218a`.

```
<asmout() when 2 generated instructions, debug 226d>≡ (108c)
if(debug['a'])
    Bprint(&bso, " %.8lux: %.8lux %.8lux%P\n", v, o1, o2, p);
```

Uses `bso 218c` and `debug 218a`.

```
<asmout() when 3 generated instructions, debug 226e>≡ (108c)
if(debug['a'])
    Bprint(&bso, " %.8lux: %.8lux %.8lux %.8lux%P\n", v, o1, o2, o3, p);
```

Uses `bso 218c` and `debug 218a`.

```
<asmout() when 4 generated instructions, debug 226f>≡ (108c)
if(debug['a'])
    Bprint(&bso, " %.8lux: %.8lux %.8lux %.8lux %.8lux%P\n",
        v, o1, o2, o3, o4, p);
```

Uses `bso 218c` and `debug 218a`.

```
<asmout() when 5 generated instructions, debug 226g>≡ (108c)
if(debug['a'])
    Bprint(&bso, " %.8lux: %.8lux %.8lux %.8lux %.8lux %.8lux%P\n",
        v, o1, o2, o3, o4, o5, p);
```

Uses `bso 218c` and `debug 218a`.

```
<asmout() when 6 generated instructions, debug 226h>≡ (108c)
if(debug['a'])
    Bprint(&bso, " %.8lux: %.8lux %.8lux %.8lux %.8lux %.8lux %.8lux%P\n",
        v, o1, o2, o3, o4, o5, o6, p);
```

Uses `bso 218c` and `debug 218a`.

```
<asmout() when other size, debug 226i>≡ (108c)
if(debug['a'])
    Bprint(&bso, " %.8lux:\t\t%P\n", v, p);
```

Uses `bso 218c` and `debug 218a`.

```

<asmb() before cflush, debug 227a>≡ (47b)
    if(debug['a']) {
        Bprint(&bso, "\n");
        Bflush(&bso);
    }

```

Uses bso 218c and debug 218a.

A.6 Symbol table debugging: 5l -n

```

<putsymb() debug 227b>≡ (148a)
    if(debug['n']) {
        <putsymb() if z or Z in debug output 227c>
        if(ver)
            Bprint(&bso, "%c %.8lux %s<%d>\n", t, v, s, ver);
        else
            Bprint(&bso, "%c %.8lux %s\n", t, v, s);
    }

```

Uses bso 218c and debug 218a.

```

<putsymb() if z or Z in debug output 227c>≡ (227b)
    if(t == 'z' || t == 'Z') {
        Bprint(&bso, "%c %.8lux ", t, v);
        for(i=1; s[i] != 0 || s[i+1] != 0; i+=2) {
            f = ((s[i]&0xff) << 8) | (s[i+1]&0xff);
            Bprint(&bso, "%x", f);
        }
        Bprint(&bso, "\n");
        return;
    }

```

Uses bso 218c.

A.7 Line table debugging: 5l -V

```

<asmlc() dump instruction p, debug 227d>≡ (160c)
    if(debug['V'])
        Bprint(&bso, "%6lux %P\n", p->pc, p);

```

Uses bso 218c and debug 218a.

```

<asmlc() dump lcsiz, debug 227e>≡ (160c)
    if(debug['V'])
        Bprint(&bso, "\t\t%6ld", lcsiz);

```

Uses bso 218c, debug 218a, and lcsiz 160a.

```

<asmlc() dump s, debug 227f>≡ (160c)
    if(debug['V'])
        Bprint(&bso, " pc+%ld*%d(%ld)", s, MINLC, s+128);

```

Uses MINLC-12 160b, bso 218c, and debug 218a.

```

⟨asmlc() dump big line change, debug 228a⟩≡ (160c)
  if(debug['V']) {
    if(s > 0)
      Bprint(&bso, " lc+%ld(%d,%ld)\n",
             s, 0, s);
    else
      Bprint(&bso, " lc+%ld(%d,%ld)\n",
             s, 0, s);
    Bprint(&bso, "%6lux %P\n",
           p->pc, p);
  }

```

Uses bso 218c and debug 218a.

```

⟨asmlc() dump small line increment, debug 228b⟩≡ (160c)
  if(debug['V']) {
    Bprint(&bso, " lc+%ld(%ld)\n", s, 0+s);
    Bprint(&bso, "%6lux %P\n",
           p->pc, p);
  }

```

Uses bso 218c and debug 218a.

```

⟨asmlc() dump negative line increment, debug 228c⟩≡ (160c)
  if(debug['V']) {
    Bprint(&bso, " lc+%ld(%ld)\n", s, 64-s);
    Bprint(&bso, "%6lux %P\n",
           p->pc, p);
  }

```

Uses bso 218c and debug 218a.

Appendix B

Error Management

51 accumulates errors (incrementing `nerrors`) and continues processing to report as many problems as possible in a single run. On exit, `errorexit()` removes the partial output file, just like 5a does.

<global nerrors 229a>≡ (285b)
`int nerrors = 0;`

Uses `nerrors 229a`.

<function errexit 229b>≡ (285b)
`void
errexit(void)
{`

```
    if(nerrors) {  
        if(cout >= 0)  
            remove(outfile);  
        exits("error");  
    }  
    exits(0);  
}
```

Uses `cout 38d`, `nerrors 229a`, and `outfile 38c`.

<pragmas varargck argpos 229c>≡ (279)
`#pragma varargck argpos diag 1`

<function diag 229d>≡ (285b)

```
void  
diag(char *fmt, ...)  
{  
    char buf[STRINGSZ];  
    char *tn;  
    va_list arg;  
  
    tn = "??none??";  
    if(curtext != P && curtext->from.sym != S)  
        tn = curtext->from.sym->name;  
    va_start(arg, fmt);  
    vseprint(buf, buf+sizeof(buf), fmt, arg);  
    va_end(arg);  
    print("%s: %s\n", tn, buf);  
  
    nerrors++;  
    if(nerrors > 20 && !debug['A']) {  
        print("too many errors\n");  
        errexit();  
    }  
}
```

}

Uses P 36c, S 33b, STRINGSZ 232d, curtext 36f, debug 218a, errexit() 229b, and nerrors 229a.

Appendix C

Profiling

The `5l -v` flag prints a summary of CPU time, symbol count, and memory usage at the end of linking. This is useful for profiling the linker itself when linking large programs.

```
<main() profile report 231a>≡ (38e)
    if(debug['v']) {
        Bprint(&bso, "%5.2f cpu time\n", cputime());
        Bprint(&bso, "%ld symbols\n", nsymbol);
        Bprint(&bso, "%ld memory used\n", thunk);

        Bprint(&bso, "%d sizeof adr\n", sizeof(Adr));
        Bprint(&bso, "%d sizeof prog\n", sizeof(Prog));
        Bflush(&bso);
    }
```

```
<global nsymbol linker 231b>≡ (283b)
    long nsymbol;
```

```
<lookup() profiling 231c>≡ (32d)
    nsymbol++;
```

Uses `nsymbol 231b`.

Appendix D

Utilities

This appendix collects the utility code shared with 5a: the same arena-based memory allocator and buffered I/O wrappers. The linker also adds string-related helpers used during symbol name manipulation.

D.1 Memory management

The linker replaces the standard `malloc()` and `free()` with a simple arena allocator that never frees memory. This makes sense because a linker allocates many small objects (instructions, symbols, operands) that all live until the program exits—there is no point in tracking individual lifetimes. The allocator requests large chunks from `sbrk()` and hands out pieces sequentially. An important side effect is that `sbrk()` returns zero-filled memory (the kernel zeroes pages before mapping them), so newly allocated structures start with all fields set to zero. Some code in 51 relies on this property.

```
<global hunk 232a>≡ (285a)
char* hunk;
```

```
<global nhunk 232b>≡ (285a)
long nhunk;
```

```
<global thunk 232c>≡ (283b)
long thunk;
```

```
<constant NHUNK linker 232d>≡ (278c)
NHUNK = 100000,
```

```
<function gethunk 232e>≡ (285a)
void
gethunk(void)
{
    char *h;
    long nh;

    nh = NHUNK;
    if(thunk >= 5L*NHUNK) {
        nh = 5L*NHUNK;
        if(thunk >= 25L*NHUNK)
            nh = 25L*NHUNK;
    }
    h = sbrk(nh);
    if(h == (char*)-1) {
        diag("out of memory");
        errexit();
    }
}
```

```

    hunk = h;
    nhunk = nh;
    thunk += nh;
}

```

Uses NHUNK, diag() 229d, errexit() 229b, hunk 232a, nhunk 232b, and thunk 232c.

```

⟨function malloc 233a⟩≡ (285a)
/*
 * fake malloc
 */
void*
malloc(ulong n)
{
    void *p;

    // upper_round(n, 8)
    while(n & 7)
        n++;

    while(nhunk < n)
        gethunk();
    p = hunk;
    nhunk -= n;
    hunk += n;
    return p;
}

```

Uses gethunk() 232e, hunk 232a, and nhunk 232b.

```

⟨function free 233b⟩≡ (285a)
void
free(void *p)
{
    USED(p);
}

```

```

⟨function setmalloctag 233c⟩≡ (285a)
/*@Scheck: looks dead, but because we redefine malloc/free we must also redefine that
void setmalloctag(void *v, ulong pc)
{
    USED(v, pc);
}

```

D.2 Buffer management

The linker uses its own I/O buffers rather than `libbio` for the performance-critical paths: writing the executable and reading object files. The output buffer (`obuf`) accumulates machine code and data before flushing to the executable file; the input buffer (`ibuf`) holds the current chunk of an object file being deserialized. The `dbuf` alias allows treating the union as a variable-size buffer.

```

⟨struct Buf 233d⟩≡ (279)
union Buf
{
    struct
    {
        char    obuf[MAXIO];          /* output buffer */
        byte    ibuf[MAXIO];         /* input buffer */
    };
}

```

```

char    dbuf[1]; // variable size
//XxX: this cause bugs in kencc under Linux
};

```

<constant MAXIO 234a>≡ (278c)
MAXIO = 8192,

<global buf 234b>≡ (289a)
union Buf buf;

Uses Buf 233d.

D.2.1 Output buffer

<function cput(arm) 234c>≡ (289a)
void
cput(int c)
{
 cbp[0] = c;
 cbp++;
 cbc--;
 if(cbc <= 0)
 cflush();
}

Uses cbc 109b, cbp 109a, and cflush() 110a.

<function wputl(arm) 234d>≡ (289a)
void
wputl(long l)
{

 cbp[0] = l;
 cbp[1] = l>>8;
 cbp += 2;
 cbc -= 2;
 if(cbc <= 0)
 cflush();
}

Uses cbc 109b, cbp 109a, and cflush() 110a.

<function wput(arm) 234e>≡ (289a)
void
wput(long l)
{

 cbp[0] = l>>8;
 cbp[1] = l;
 cbp += 2;
 cbc -= 2;
 if(cbc <= 0)
 cflush();
}

Uses cbc 109b, cbp 109a, and cflush() 110a.

```

<function strnput(arm) 235a>≡ (289a)
void
strnput(char *s, int n)
{
    for(; *s; s++){
        cput(*s);
        n--;
    }
    for(; n > 0; n--)
        cput(0);
}
Uses cput() 234c.

```

D.2.2 Input buffer

D.3 File management

```

<function fileexists 235b>≡ (285a)
int
fileexists(char *s)
{
    byte dirbuf[400];

    /* it's fine if stat result doesn't fit in dirbuf, since even then the file exists */
    return stat(s, dirbuf, sizeof(dirbuf)) >= 0;
}

```

D.4 String processing

`atolwhex()` ^{235c} (“ASCII to long, with hex”) is a number parser that handles decimal, octal (leading 0), and hexadecimal (leading 0x) formats. It is used throughout 51 to parse command-line arguments like `-T0x10000` (text segment start address) and to decode the ASCII size fields in archive headers.

```

<function atolwhex 235c>≡ (285a)
long
atolwhex(char *s)
{
    long n;
    int f;

    n = 0;
    f = 0;
    while(*s == ' ' || *s == '\t')
        s++;
    if(*s == '-' || *s == '+') {
        if(*s++ == '-')
            f = 1;
        while(*s == ' ' || *s == '\t')
            s++;
    }
    if(s[0]=='0' && s[1]){
        if(s[1]=='x' || s[1]=='X'){
            s += 2;
            for(;;){
                if(*s >= '0' && *s <= '9')
                    n = n*16 + *s++ - '0';
            }
        }
    }
}

```

```

        else if(*s >= 'a' && *s <= 'f')
            n = n*16 + *s++ - 'a' + 10;
        else if(*s >= 'A' && *s <= 'F')
            n = n*16 + *s++ - 'A' + 10;
        else
            break;
    }
} else
    while(*s >= '0' && *s <= '7')
        n = n*8 + *s++ - '0';
} else
    while(*s >= '0' && *s <= '9')
        n = n*10 + *s++ - '0';
if(f)
    n = -n;
return n;
}

```

D.5 Mathematic functions

`rnd()`^{236a} rounds a value v up to the next multiple of r . For example, `rnd(16, 8) == 16` and `rnd(17, 8) == 24`. The linker uses this constantly to align sections and data to the boundaries required by the executable format and the hardware (e.g., 4-byte alignment for ARM instructions, page alignment for text and data segments).

<function rnd 236a>≡ (285a)

```

long
rnd(long v, long r)
{
    long c;

    <rnd() if r is null or negative 236b>
    v += r - 1;
    c = v % r;
    <rnd() if v was negative 236c>
    v -= c;
    return v;
}

```

<rnd() if r is null or negative 236b>≡ (236a)

```

if(r <= 0)
    return v;

```

<rnd() if v was negative 236c>≡ (236a)

```

if(c < 0)
    c += r;

```

Appendix E

Linker-related Programs

This appendix covers programs that work with the linker's output: `libmach` (the library for reading executables and symbol tables, used by the debugger and profiler), `nm` (list symbols), `size` (report segment sizes), and `strip` (remove symbol tables). These are the Plan 9 equivalents of the `binutils` tools on Linux.

E.1 `include/mach.h`

E.2 `size`

```
<function size 237a>≡ (278a)
int
size(char *file)
{
    fdt fd;
    Fhdr f;

    if((fd = open(file, OREAD)) < 0){
        fprintf(2, "size: ");
        perror(file);
        return 1;
    }
    if(crackhdr(fd, &f)) {
        print("%ldt + %ldd + %ldb = %ld\t%s\n", f.txtsz, f.datsz,
            f.bsssz, f.txtsz+f.datsz+f.bsssz, file);
        close(fd);
        return 0;
    }
    fprintf(2, "size: %s not an a.out\n", file);
    close(fd);
    return 1;
}

<function main (linkers/misc/size.c) 237b>≡ (278a)
void
main(int argc, char *argv[])
{
    char *err;
    int i;

    ARGBEGIN {
    default:
        fprintf(2, "usage: size [a.out ...]\n");
        exits("usage");
    }
}
```

```

} ARGEND;

err = nil;
if(argc == 0)
    if(size("8.out"))
        err = "error";
for(i=0; i<argc; i++)
    if(size(argv[i]))
        err = "error";
exits(err);
}

```

E.3 nm

<enum NmConstants 238a>≡ (277)

```

enum{
    CHUNK = 256 /* must be power of 2 */
};

```

<global errs 238b>≡ (277)

```

static char *errs; /* exit status */

```

<global filename 238c>≡ (277)

```

static char *filename; /* current file */

```

<global symname 238d>≡ (277)

```

static char symname[]="__.SYMDEF"; /* table of contents file name */

```

Uses symname-36 238d.

<global multifile 238e>≡ (277)

```

static bool multifile; /* processing multiple files */

```

<global aflag (linkers/misc/nm.c) 238f>≡ (277)

```

static int aflag;

```

<global gflag 238g>≡ (277)

```

static int gflag;

```

<global hflag 238h>≡ (277)

```

static int hflag;

```

<global nflag 238i>≡ (277)

```

static int nflag;

```

<global sflag 238j>≡ (277)

```

static int sflag;

```

<global uflag (linkers/misc/nm.c) 238k>≡ (277)

```

static int uflag;

```

<global Tflag 238l>≡ (277)

```

static int Tflag;

```

<global fnames 238m>≡ (277)

```

static Sym **fnames; /* file path translation table */

```

<global symptr 238n>≡ (277)

```

static Sym **symptr;

```

<global nsym 239a>≡ (277)
static int nsym;

<global bout (linkers/misc/nm.c) 239b>≡ (277)
static Biobuf bout;

<function usage (linkers/misc/nm.c) 239c>≡ (277)
static void
usage(void)
{
fprint(STDERR, "usage: nm [-aghnsTu] file ...\\n");
exits("usage");
}

<function main (linkers/misc/nm.c) 239d>≡ (277)
void
main(int argc, char *argv[])
{
int i;
Biobuf *bin;

Binit(&bout, STDOUT, OWRITE);
argv0 = argv[0];
ARGBEGIN {
default: usage();
case 'a': aflag = 1; break;
case 'g': gflag = 1; break;
case 'h': hflag = 1; break;
case 'n': nflag = 1; break;
case 's': sflag = 1; break;
case 'u': uflag = 1; break;
case 'T': Tflag = 1; break;
} ARGEND
if (argc == 0)
usage();
if (argc > 1)
multifile = true;
for(i=0; i<argc; i++){
filename = argv[i];
bin = Bopen(filename, OREAD);
if(bin == nil){
error("cannot open %s", filename);
continue;
}
if (isarc(bin))
doarc(bin);
else{
Bseek(bin, 0, SEEK__START);
dofile(bin);
}
Bterm(bin);
}
exits(errs);
}

<function doarc 239e>≡ (277)
/*
* read an archive file,
* processing the symbols for each intermediate file in it.
*/

```

void
doar(Biobuf *bp)
{
    int offset, size, obj;
    char membername[SARNAME];

    multifile = true;
    for (offset = Boffset(bp);;offset += size) {
        size = nextar(bp, offset, membername);
        if (size < 0) {
            error("phase error on ar header %ld", offset);
            return;
        }
        if (size == 0)
            return;
        if (strcmp(membername, symname) == 0)
            continue;
        obj = objtype(bp, 0);
        if (obj < 0) {
            error("inconsistent file %s in %s",
                membername, filename);
            return;
        }
        if (!readar(bp, obj, offset+size, 1)) {
            error("invalid symbol reference in file %s",
                membername);
            return;
        }
        filename = membername;
        nsym=0;
        objtraverse(psym, 0);
        printsyms(symptr, nsym);
    }
}

```

Uses filename-35 238c, multifile-37 238e, nsym-47 239a, printsyms() 243a, psym() 242, symname-36 238d, and symptr-46 238n.

<function dofile 240a>≡

(277)

```

/*
 * process symbols in a file
 */
void
dofile(Biobuf *bp)
{
    int obj;

    obj = objtype(bp, 0);
    if (obj < 0)
        execsyms(Bfildes(bp));
    else
        if (readobj(bp, obj)) {
            nsym = 0;
            objtraverse(psym, 0);
            printsyms(symptr, nsym);
        }
}

```

Uses execsyms() 241b, nsym-47 239a, printsyms() 243a, psym() 242, and symptr-46 238n.

<function cmp_symbol 240b>≡

(277)

```

/*

```

```

* comparison routine for sorting the symbol table
* this screws up on 'z' records when aflag == 1
*/
int
cmp_symbol(void *vs, void *vt)
{
    Sym **s, **t;

    s = vs;
    t = vt;
    if(nflag)
        if((*s)->value < (*t)->value)
            return -1;
        else
            return (*s)->value > (*t)->value;
    return strcmp((*s)->name, (*t)->name);
}

```

Uses nflag-41 238i.

<function zenter 241a>≡

(277)

```

/*
* enter a symbol in the table of filename elements
*/
void
zenter(Sym *s)
{
    static int maxf = 0;

    if (s->value > maxf) {
        maxf = (s->value+CHUNK-1) &~ (CHUNK-1);
        fnames = realloc(fnames, (maxf+1)*sizeof(*fnames));
        if(fnames == 0) {
            error("out of memory", argv0);
            exits("memory");
        }
    }
    fnames[s->value] = s;
}

```

Uses CHUNK-33 238a and fnames-45 238m.

<function execsyms 241b>≡

(277)

```

/*
* get the symbol table from an executable file, if it has one
*/
void
execsyms(int fd)
{
    Fhdr f;
    Sym *s;
    long n;

    seek(fd, 0, 0);
    if (crackhdr(fd, &f) == 0) {
        error("Can't read header for %s", filename);
        return;
    }
    if (syminit(fd, &f) < 0)
        return;
    s = symbase(&n);
    nsym = 0;
}

```

```

while(n--)
    psym(s++, 0);

    printsyms(symptr, nsym);
}

```

Uses filename-35 238c, nsym-47 239a, printsyms() 243a, psym() 242, and symptr-46 238n.

<function psym 242>≡

(277)

```

void
psym(Sym *s, void* p)
{
    USED(p);
    switch(s->type) {
    case 'T':
    case 'L':
    case 'D':
    case 'B':
        if (uflag)
            return;
        if (!aflag && ((s->name[0] == '.' || s->name[0] == '$'))
            return;
        break;
    case 'b':
    case 'd':
    case 'l':
    case 't':
        if (uflag || gflag)
            return;
        if (!aflag && ((s->name[0] == '.' || s->name[0] == '$'))
            return;
        break;
    case 'U':
        if (gflag)
            return;
        break;
    case 'Z':
        if (!aflag)
            return;
        break;
    case 'm':
    case 'f': /* we only see a 'z' when the following is true*/
        if(!aflag || uflag || gflag)
            return;
        if (strcmp(s->name, ".frame"))
            zenter(s);
        break;
    case 'a':
    case 'p':
    case 'z':
    default:
        if(!aflag || uflag || gflag)
            return;
        break;
    }
    symptr = realloc(symptr, (nsym+1)*sizeof(Sym*));
    if (symptr == 0) {
        error("out of memory");
        exits("memory");
    }
    symptr[nsym++] = s;
}

```

```
}
```

Uses `aflag-38 238f`, `gflag-39 238g`, `nsym-47 239a`, `symptr-46 238n`, `uflag-43 238k`, and `zenter() 241a`.

<function printsyms 243a>≡ (277)

```
void
printsyms(Sym **symptr, long nsym)
{
    int i, wid;
    Sym *s;
    char *cp;
    char path[512];

    if(!sflag)
        qsort(symptr, nsym, sizeof(*symptr), cmp_symbol);

    wid = 0;
    for (i=0; i<nsym; i++) {
        s = symptr[i];
        if (s->value && wid == 0)
            wid = 8;
        else if (s->value >= 0x100000000LL && wid == 8)
            wid = 16;
    }
    for (i=0; i<nsym; i++) {
        s = symptr[i];
        if (multifile && !hflag)
            Bprint(&bout, "%s:", filename);
        if (s->type == 'z') {
            fileelem(fnames, (uchar *) s->name, path, 512);
            cp = path;
        } else
            cp = s->name;
        if (Tflag)
            Bprint(&bout, "%8ux ", s->sig);
        if (s->value || s->type == 'a' || s->type == 'p')
            Bprint(&bout, "%*llux ", wid, s->value);
        else
            Bprint(&bout, "%*s ", wid, "");

        Bprint(&bout, "%c %s\n", s->type, cp);
    }
}
```

Uses `Tflag-44 238l`, `bout-48 239b`, `cmp_symbol() 240b`, `filename-35 238c`, `fnames-45 238m`, `hflag-40 238h`, `multifile-37 238e`, and `sflag-42 238j`.

<function error 243b>≡ (277)

```
static void
error(char *fmt, ...)
{
    Fmt f;
    char buf[128];
    va_list arg;

    fmtfdinit(&f, 2, buf, sizeof buf);
    fprintf(&f, "%s: ", argv0);
    va_start(arg, fmt);
    fmtvprint(&f, fmt, arg);
    va_end(arg);
    fprintf(&f, "\n");
    fmtfdflush(&f);
}
```

```
    errs = "errors";
}
```

Uses errs-34 238b.

E.4 ar

```
<struct Arsymref 244a>≡ (273e)
typedef struct Arsymref
{
    char *name;
    int type;
    int len;
    vlong offset;
    struct Arsymref *next;
} Arsymref;
```

Uses Arsymref 244a.

```
<struct Armember 244b>≡ (273e)
typedef struct Armember /* Temp file entry - one per archive member */
{
    struct Armember *next;
    struct ar_hdr hdr;
    long size;
    long date;
    void *member;
} Armember;
```

Uses Armember 244b.

```
<struct Arfile 244c>≡ (273e)
typedef struct Arfile /* Temp file control block - one per tempfile */
{
    int paged; /* set when some data paged to disk */
    char *fname; /* paging file name */
    int fd; /* paging file descriptor */
    vlong size;
    Armember *head; /* head of member chain */
    Armember *tail; /* tail of member chain */
    Arsymref *sym; /* head of defined symbol chain */
} Arfile;
```

Uses Arfile 244c.

```
<struct Hashchain 244d>≡ (273e)
typedef struct Hashchain
{
    char *name;
    struct Hashchain *next;
} Hashchain;
```

Uses Hashchain 244d.

```
<constant NHASH 244e>≡ (273e)
#define NHASH 1024
```

<function HEADER_IO 245a>≡ (273e)
 /*
 * macro to portably read/write archive header.
 * 'cmd' is read/write/Bread/Bwrite, etc.
 */
 #define HEADER_IO(cmd, f, h) cmd(f, h.name, sizeof(h.name)) != sizeof(h.name)\
 || cmd(f, h.date, sizeof(h.date)) != sizeof(h.date)\
 || cmd(f, h.uid, sizeof(h.uid)) != sizeof(h.uid)\
 || cmd(f, h.gid, sizeof(h.gid)) != sizeof(h.gid)\
 || cmd(f, h.mode, sizeof(h.mode)) != sizeof(h.mode)\
 || cmd(f, h.size, sizeof(h.size)) != sizeof(h.size)\
 || cmd(f, h.fmag, sizeof(h.fmag)) != sizeof(h.fmag)

<global man 245b>≡ (273e)
 char *man = "mrxt dpq";
 Uses man 245b.

<global opt 245c>≡ (273e)
 char *opt = "uvnbailo";
 Uses opt 245c.

<global artemp 245d>≡ (273e)
 char artemp[] = "/tmp/vXXXXX";
 Uses artemp 245d.

<global movtemp 245e>≡ (273e)
 char movtemp[] = "/tmp/v1XXXXX";
 Uses movtemp 245e.

<global tailtemp 245f>≡ (273e)
 char tailtemp[] = "/tmp/v2XXXXX";
 Uses tailtemp 245f.

<global symdef 245g>≡ (273e)
 char symdef[] = "___SYMDEF";
 Uses symdef 245g.

<global aflag 245h>≡ (273e)
 int aflag; /* command line flags */

<global bflag 245i>≡ (273e)
 static int bflag;

<global cflag 245j>≡ (273e)
 int cflag;

<global oflag 245k>≡ (273e)
 int oflag;

<global uflag 245l>≡ (273e)
 int uflag;

<global vflag 245m>≡ (273e)
 int vflag;

<global allobj 245n>≡ (273e)
 int allobj = 1; /* set when all members are object files of the same type */
 Uses allobj 245n.

<global symdefsize 246a>≡ (273e)
int symdefsize; /* size of symdef file */

<global dupfound 246b>≡ (273e)
int dupfound; /* flag for duplicate symbol */

<global hash 246c>≡ (273e)
Hashchain *hash[NHASH]; /* hash table of text symbols */
Uses NHASH-16 244e.

<constant ARNAME_SIZE 246d>≡ (273e)
#define ARNAME_SIZE sizeof(astart->tail->hdr.name)

<global poname 246e>≡ (273e)
char poname[ARNAME_SIZE+1]; /* name of pivot member */
Uses ARNAME_SIZE-19 246d.

<global file 246f>≡ (273e)
char *file; /* current file or member being worked on */

<global bout 246g>≡ (273e)
Biobuf bout;

<global bar 246h>≡ (273e)
Biobuf bar;

<global comfun 246i>≡ (273e)
void (*comfun)(char*, int, char**);

<function main 246j>≡ (273e)
void
main(int argc, char *argv[])
{
char *cp;

Binit(&bout, 1, OWRITE);
if(argc < 3)
usage();
for (cp = argv[1]; *cp; cp++) {
switch(*cp) {
case 'a': aflag = 1; break;
case 'b': bflag = 1; break;
case 'c': cflag = 1; break;
case 'd': setcom(dcmd); break;
case 'i': bflag = 1; break;
case 'l':
strcpy(artemp, "vXXXXX");
strcpy(movtemp, "v1XXXXX");
strcpy(tailtemp, "v2XXXXX");
break;
case 'm': setcom(mcmd); break;
case 'o': oflag = 1; break;
case 'p': setcom(pcmd); break;
case 'q': setcom(qcmd); break;
case 'r': setcom(rcmd); break;
case 't': setcom(tcmt); break;
case 'u': uflag = 1; break;
case 'v': vflag = 1; break;
case 'x': setcom(xcmd); break;
default:

```

        fprintf(2, "ar: bad option '%c'\n", *cp);
        exits("error");
    }
}
if (aflag && bflag) {
    fprintf(2, "ar: only one of 'a' and 'b' can be specified\n");
    usage();
}
if(aflag || bflag) {
    trim(argv[2], poname, sizeof(poname));
    argv++;
    argc--;
    if(argc < 3)
        usage();
}
if(comfun == 0) {
    if(uflag == 0) {
        fprintf(2, "ar: one of [%s] must be specified\n", man);
        usage();
    }
    setcom(rcmd);
}
cp = argv[2];
argc -= 3;
argv += 3;
(*comfun)(cp, argc, argv); /* do the command */
cp = 0;
while (argc--) {
    if (*argv) {
        fprintf(2, "ar: %s not found\n", *argv);
        cp = "error";
    }
    argv++;
}
exits(cp);
}

```

Uses aflag 245h, artemp 245d, bflag-18 245i, bout 246g, cflag 245j, comfun 246i, dcmd() 249, man 245b, mcmd() 251, movtemp 245e, oflag 245k, pcmd() 250b, poname 246e, qcmd() 252b, rcmd() 247b, setcom() 247a, tailtemp 245f, tcmd() 252a, trim() 261a, uflag 245l, vflag 245m, and xcmd() 250a.

<function setcom 247a>≡ (273e)

```

/*
 * select a command
 */
void
setcom(void (*fun)(char *, int, char**))
{
    if(comfun != 0) {
        fprintf(2, "ar: only one of [%s] allowed\n", man);
        usage();
    }
    comfun = fun;
}

```

Uses comfun 246i and man 245b.

<function rcmd 247b>≡ (273e)

```

/*
 * perform the 'r' and 'u' commands
 */

```

```

void
rcmd(char *arname, int count, char **files)
{
    int fd;
    int i;
    Arfile *ap;
    Armember *bp;
    Dir *d;
    Biobuf *bfile;

    fd = openar(arname, ORDWR, 1);
    if (fd >= 0) {
        Binit(&bar, fd, OREAD);
        Bseek(&bar, seek(fd, 0, 1), 1);
    }
    astart = newtempfile(artemp);
    ap = astart;
    aend = 0;
    for(i = 0; fd >= 0; i++) {
        bp = getdir(&bar);
        if (!bp)
            break;
        if (bamatch(file, poname)) { /* check for pivot */
            aend = newtempfile(tailtemp);
            ap = aend;
        }
        /* pitch symdef file */
        if (i == 0 && strcmp(file, symdef) == 0) {
            skip(&bar, bp->size);
            continue;
        }
        if (count && !match(count, files)) {
            scanobj(&bar, ap, bp->size);
            arcopy(&bar, ap, bp);
            continue;
        }
        bfile = Bopen(file, OREAD);
        if (!bfile) {
            if (count != 0)
                fprintf(2, "ar: cannot open %s\n", file);
            scanobj(&bar, ap, bp->size);
            arcopy(&bar, ap, bp);
            continue;
        }
        d = dirfstat(Bfildes(bfile));
        if(d == nil)
            fprintf(2, "ar: cannot stat %s: %r\n", file);
        if (uflag && (d==nil || d->mtime <= bp->date)) {
            scanobj(&bar, ap, bp->size);
            arcopy(&bar, ap, bp);
            Bterm(bfile);
            free(d);
            continue;
        }
        mesg('r', file);
        skip(&bar, bp->size);
        scanobj(bfile, ap, d->length);
        free(d);
        armove(bfile, ap, bp);
        Bterm(bfile);
    }
}

```

```

}
if(fd >= 0)
    close(fd);
    /* copy in remaining files named on command line */
for (i = 0; i < count; i++) {
    file = files[i];
    if(file == 0)
        continue;
    files[i] = 0;
    bfile = Bopen(file, OREAD);
    if (!bfile)
        fprintf(2, "ar: %s cannot open\n", file);
    else {
        mesg('a', file);
        d = dirfstat(Bfildes(bfile));
        if (d == nil)
            fprintf(2, "can't stat %s\n", file);
        else {
            scanobj(bfile, astart, d->length);
            armove(bfile, astart, newmember());
            free(d);
        }
        Bterm(bfile);
    }
}
}
if(fd < 0 && !cflag)
    install(arname, astart, 0, aend, 1); /* issue 'creating' msg */
else
    install(arname, astart, 0, aend, 0);
}

```

Uses aend 273e, arcopy() 257b, armove() 257a, artemp 245d, astart 273e, bamatch() 260a, bar 246h, cflag 245j, file 246f, free() 233b, getdir() 256d, install() 258a, match() 259b, mesg() 260b, newmember() 263e, newtempfile() 263d, openar() 255a, poname 246e, scanobj() 253, skip() 257c, symdef 245g, tailtemp 245f, and uflag 245l.

<function dcmd 249>≡

(273e)

```

void
dcmd(char *arname, int count, char **files)
{
    Armember *bp;
    int fd, i;

    if (!count)
        return;
    fd = openar(arname, ORDWR, 0);
    Binit(&bar, fd, OREAD);
    Bseek(&bar, seek(fd,0,1), 1);
    astart = newtempfile(artemp);
    for (i = 0; bp = getdir(&bar); i++) {
        if(match(count, files)) {
            mesg('d', file);
            skip(&bar, bp->size);
            if (strcmp(file, symdef) == 0)
                allobj = 0;
        } else if (i == 0 && strcmp(file, symdef) == 0)
            skip(&bar, bp->size);
        else {
            scanobj(&bar, astart, bp->size);
            arcopy(&bar, astart, bp);
        }
    }
}

```

```

    close(fd);
    install(arname, astart, 0, 0, 0);
}

```

Uses `allobj` 245n, `arcopy()` 257b, `artemp` 245d, `astart` 273e, `bar` 246h, `file` 246f, `getdir()` 256d, `install()` 258a, `match()` 259b, `mesg()` 260b, `newtempfile()` 263d, `openar()` 255a, `scanobj()` 253, `skip()` 257c, and `symdef` 245g.

<function xcmd 250a>≡ (273e)

```

void
xcmd(char *arname, int count, char **files)
{
    int fd, f, mode, i;
    Armember *bp;
    Dir dx;

    fd = openar(arname, OREAD, 0);
    Binit(&bar, fd, OREAD);
    Bseek(&bar, seek(fd, 0, 1), 1);
    i = 0;
    while (bp = getdir(&bar)) {
        if(count == 0 || match(count, files)) {
            mode = strtoul(bp->hdr.mode, 0, 8) & 0777;
            f = create(file, OWRITE, mode);
            if(f < 0) {
                fprintf(2, "ar: %s cannot create\n", file);
                skip(&bar, bp->size);
            } else {
                mesg('x', file);
                arcopy(&bar, 0, bp);
                if (write(f, bp->member, bp->size) < 0)
                    wrerr();
                if(oflag) {
                    nulldir(&dx);
                    dx.atime = bp->date;
                    dx.mtime = bp->date;
                    if(dirwstat(file, &dx) < 0)
                        perror(file);
                }
                free(bp->member);
                close(f);
            }
            free(bp);
            if (count && ++i >= count)
                break;
        } else {
            skip(&bar, bp->size);
            free(bp);
        }
    }
    close(fd);
}

```

Uses `arcopy()` 257b, `bar` 246h, `file` 246f, `free()` 233b, `getdir()` 256d, `match()` 259b, `mesg()` 260b, `oflag` 245k, `openar()` 255a, `skip()` 257c, and `wrerr()` 255c.

<function pcmd 250b>≡ (273e)

```

void
pcmd(char *arname, int count, char **files)
{
    int fd;
    Armember *bp;

```

```

fd = openar(aname, OREAD, 0);
Binit(&bar, fd, OREAD);
Bseek(&bar, seek(fd,0,1), 1);
while(bp = getdir(&bar)) {
    if(count == 0 || match(count, files)) {
        if(vflag)
            print("\n<%s>\n\n", file);
        arcopy(&bar, 0, bp);
        if (write(1, bp->member, bp->size) < 0)
            wrerr();
    } else
        skip(&bar, bp->size);
    free(bp);
}
close(fd);
}

```

Uses arcopy() 257b, bar 246h, file 246f, free() 233b, getdir() 256d, match() 259b, openar() 255a, skip() 257c, vflag 245m, and wrerr() 255c.

<function mcmd 251>≡ (273e)

```

void
mcmd(char *aname, int count, char **files)
{
    int fd, i;
    Arfile *ap;
    Armember *bp;

    if (count == 0)
        return;
    fd = openar(aname, ORDWR, 0);
    Binit(&bar, fd, OREAD);
    Bseek(&bar, seek(fd,0,1), 1);
    astart = newtempfile(artemp);
    amiddle = newtempfile(movtemp);
    aend = 0;
    ap = astart;
    for (i = 0; bp = getdir(&bar); i++) {
        if (bamatch(file, poname)) {
            aend = newtempfile(tailtemp);
            ap = aend;
        }
        if(match(count, files)) {
            mesg('m', file);
            scanobj(&bar, amiddle, bp->size);
            arcopy(&bar, amiddle, bp);
        } else
            /*
             * pitch the symdef file if it is at the beginning
             * of the archive and we aren't inserting in front
             * of it (ap == astart).
             */
            if (ap == astart && i == 0 && strcmp(file, symdef) == 0)
                skip(&bar, bp->size);
            else {
                scanobj(&bar, ap, bp->size);
                arcopy(&bar, ap, bp);
            }
    }
    close(fd);
    if (poname[0] && aend == 0)

```

```

    fprintf(2, "ar: %s not found - files moved to end.\n", poname);
    install(arname, astart, amiddle, aend, 0);
}

```

Uses `aend` 273e, `amiddle` 273e, `arcopy()` 257b, `artemp` 245d, `astart` 273e, `bamatch()` 260a, `bar` 246h, `file` 246f, `getdir()` 256d, `install()` 258a, `match()` 259b, `mesg()` 260b, `movtemp` 245e, `newtempfile()` 263d, `openar()` 255a, `poname` 246e, `scanobj()` 253, `skip()` 257c, `symdef` 245g, and `tailtemp` 245f.

`<function tcmd 252a>`≡ (273e)

```

void
tcmd(char *arname, int count, char **files)
{
    int fd;
    Armember *bp;
    char name[ARNAMESIZE+1];

    fd = openar(arname, OREAD, 0);
    Binit(&bar, fd, OREAD);
    Bseek(&bar, seek(fd,0,1), 1);
    while(bp = getdir(&bar)) {
        if(count == 0 || match(count, files)) {
            if(vflag)
                longt(bp);
            trim(file, name, ARNAMESIZE);
            Bprint(&bout, "%s\n", name);
        }
        skip(&bar, bp->size);
        free(bp);
    }
    close(fd);
}

```

Uses `ARNAMESIZE-19` 246d, `bar` 246h, `bout` 246g, `file` 246f, `free()` 233b, `getdir()` 256d, `longt()` 262d, `match()` 259b, `openar()` 255a, `skip()` 257c, `trim()` 261a, and `vflag` 245m.

`<function qcmd 252b>`≡ (273e)

```

void
qcmd(char *arname, int count, char **files)
{
    int fd, i;
    Armember *bp;
    Biobuf *bfile;

    if(aflag || bflag) {
        fprintf(2, "ar: abi not allowed with q\n");
        exits("error");
    }
    fd = openar(arname, ORDWR, 1);
    if (fd < 0) {
        if(!cflag)
            fprintf(2, "ar: creating %s\n", arname);
        fd = arcreate(arname);
    }
    Binit(&bar, fd, OREAD);
    Bseek(&bar, seek(fd,0,1), 1);
    /* leave note group behind when writing archive; i.e. sidestep interrupts */
    rfork(RFNOTEGB);
    Bseek(&bar, 0, 2);
    bp = newmember();
    for(i=0; i<count && files[i]; i++) {
        file = files[i];
        files[i] = 0;
    }
}

```

```

bfile = Bopen(file, OREAD);
if(!bfile)
    fprintf(2, "ar: %s cannot open\n", file);
else {
    mesg('q', file);
    armove(bfile, 0, bp);
    if (!arwrite(fd, bp))
        wrerr();
    free(bp->member);
    bp->member = 0;
    Bterm(bfile);
}
}
free(bp);
close(fd);
}

```

Uses aflag 245h, arcreate() 255b, armove() 257a, arwrite() 265a, bar 246h, bflag-18 245i, cflag 245j, file 246f, free() 233b, mesg() 260b, newmember() 263e, openar() 255a, and wrerr() 255c.

<function scanobj 253>≡

(273e)

```

/*
 * extract the symbol references from an object file
 */
void
scanobj(Biobuf *b, Arfile *ap, long size)
{
    int obj;
    vlong offset;
    Dir *d;
    static int lastobj = -1;

    if (!allobj) /* non-object file encountered */
        return;
    offset = Boffset(b);
    obj = objtype(b, 0);
    if (obj < 0) { /* not an object file */
        allobj = 0;
        d = dirfstat(Bfildes(b));
        if (d != nil && d->length == 0)
            fprintf(2, "ar: zero length file %s\n", file);
        free(d);
        Bseek(b, offset, 0);
        return;
    }
    if (lastobj >= 0 && obj != lastobj) {
        fprintf(2, "ar: inconsistent object file %s\n", file);
        allobj = 0;
        Bseek(b, offset, 0);
        return;
    }
    lastobj = obj;
    if (!readar(b, obj, offset+size, 0)) {
        fprintf(2, "ar: invalid symbol reference in file %s\n", file);
        allobj = 0;
        Bseek(b, offset, 0);
        return;
    }
    Bseek(b, offset, 0);
    objtraverse(objsym, ap);
}

```

Uses `allobj` 245n, `file` 246f, `free()` 233b, and `objsym()` 254a.

```
<function objsym 254a>≡ (273e)
/*
 * add text and data symbols to the symbol list
 */
void
objsym(Sym *s, void *p)
{
    int n;
    Arsymref *as;
    Arfile *ap;

    if (s->type != 'T' && s->type != 'D')
        return;
    ap = (Arfile*)p;
    as = (Arsymref*)armalloc(sizeof(Arsymref));
    as->offset = ap->size;
    n = strlen(s->name);
    as->name = armalloc(n+1);
    strcpy(as->name, s->name);
    if(s->type == 'T' && duplicate(as->name)) {
        dupfound = 1;
        fprintf(2, "duplicate text symbol: %s\n", as->name);
        free(as->name);
        free(as);
        return;
    }
    as->type = s->type;
    symdefsize += 4+(n+1)+1;
    as->len = n;
    as->next = ap->sym;
    ap->sym = as;
}
```

Uses `armalloc()` 266b, `dupfound` 246b, `duplicate()` 254b, `free()` 233b, and `symdefsize` 246a.

```
<function duplicate 254b>≡ (273e)
/*
 * Check the symbol table for duplicate text symbols
 */
int
duplicate(char *name)
{
    Hashchain *p;
    char *cp;
    int h;

    h = 0;
    for(cp = name; *cp; h += *cp++)
        h *= 1119;
    if(h < 0)
        h = ~h;
    h %= NHASH;

    for(p = hash[h]; p; p = p->next)
        if(strcmp(p->name, name) == 0)
            return 1;
    p = (Hashchain*) armalloc(sizeof(Hashchain));
    p->next = hash[h];
    p->name = name;
}
```

```

    hash[h] = p;
    return 0;
}

```

Uses NHASH-16 244e and armalloc() 266b.

<function openar 255a>≡ (273e)

```

/*
 * open an archive and validate its header
 */
int
openar(char *arname, int mode, int errok)
{
    int fd;
    char mbuf[SARMAG];

    fd = open(arname, mode);
    if(fd >= 0){
        if(read(fd, mbuf, SARMAG) != SARMAG || strcmp(mbuf, ARMAG, SARMAG)) {
            fprintf(2, "ar: %s not in archive format\n", arname);
            exits("error");
        }
    }else if(!errok){
        fprintf(2, "ar: cannot open %s: %r\n", arname);
        exits("error");
    }
    return fd;
}

```

<function arcreate 255b>≡ (273e)

```

/*
 * create an archive and set its header
 */
int
arcreate(char *arname)
{
    int fd;

    fd = create(arname, OWRITE, 0664);
    if(fd < 0){
        fprintf(2, "ar: cannot create %s: %r\n", arname);
        exits("error");
    }
    if(write(fd, ARMAG, SARMAG) != SARMAG)
        wrerr();
    return fd;
}

```

Uses wrerr() 255c.

<function wrerr 255c>≡ (273e)

```

/*
 * error handling
 */
void
wrerr(void)
{
    perror("ar: write error");
    exits("error");
}

```

```

⟨function rderr 256a⟩≡ (273e)
void
rderr(void)
{
    perror("ar: read error");
    exits("error");
}

```

```

⟨function phaseerr 256b⟩≡ (273e)
void
phaseerr(int offset)
{
    fprintf(2, "ar: phase error at offset %d\n", offset);
    exits("error");
}

```

```

⟨function usage 256c⟩≡ (273e)
static void
usage(void)
{
    fprintf(2, "usage: ar [%s] [%s] archive files ... \n", opt, man);
    exits("error");
}

```

Uses man 245b and opt 245c.

```

⟨function getdir 256d⟩≡ (273e)
/*
 * read the header for the next archive member
 */
Armember *
getdir(Biobuf *b)
{
    Armember *bp;
    char *cp;
    static char name[ARNAMESIZE+1];

    bp = newmember();
    if(HEADER_IO(Bread, b, bp->hdr)) {
        free(bp);
        return 0;
    }
    if(strncmp(bp->hdr.fmag, ARFMAG, sizeof(bp->hdr.fmag)) != 0)
        phaseerr(Boffset(b));
    strncpy(name, bp->hdr.name, sizeof(bp->hdr.name));
    cp = name+sizeof(name)-1;
    *cp = '\0';
    /* skip trailing spaces and (gnu-produced) slashes */
    while(*--cp == ' ' || *cp == '/')
        ;
    cp[1] = '\0';
    file = name;
    bp->date = strtol(bp->hdr.date, 0, 0);
    bp->size = strtol(bp->hdr.size, 0, 0);
    return bp;
}

```

Uses ARNAMESIZE-19 246d, HEADER_IO-17 245a, file 246f, free() 233b, newmember() 263e, and phaseerr() 256b.

<function armove 257a>≡

(273e)

```
/*
 * Copy the file referenced by fd to the temp file
 */
void
armove(Biobuf *b, Arfile *ap, Armember *bp)
{
    char *cp;
    Dir *d;

    d = dirfstat(Bfiles(b));
    if (d == nil) {
        fprintf(2, "ar: cannot stat %s\n", file);
        return;
    }
    trim(file, bp->hdr.name, sizeof(bp->hdr.name));
    for (cp = strchr(bp->hdr.name, 0); /* blank pad on right */
         cp < bp->hdr.name+sizeof(bp->hdr.name); cp++)
        *cp = ' ';
    sprintf(bp->hdr.date, "%-12ld", d->mtime);
    sprintf(bp->hdr.uid, "%-6d", 0);
    sprintf(bp->hdr.gid, "%-6d", 0);
    sprintf(bp->hdr.mode, "%-8lo", d->mode);
    sprintf(bp->hdr.size, "%-10lld", d->length);
    strncpy(bp->hdr.fmag, ARFMAG, 2);
    bp->size = d->length;
    arread(b, bp, bp->size);
    if (d->length&0x01)
        d->length++;
    if (ap) {
        arinsert(ap, bp);
        ap->size += d->length+SAR_HDR;
    }
    free(d);
}
```

Uses arinsert() 264b, arread() 264a, file 246f, free() 233b, and trim() 261a.

<function arcopy 257b>≡

(273e)

```
/*
 * Copy the archive member at the current offset into the temp file.
 */
void
arcopy(Biobuf *b, Arfile *ap, Armember *bp)
{
    long n;

    n = bp->size;
    if (n & 01)
        n++;
    arread(b, bp, n);
    if (ap) {
        arinsert(ap, bp);
        ap->size += n+SAR_HDR;
    }
}
```

Uses arinsert() 264b and arread() 264a.

<function skip 257c>≡

(273e)

```
/*
 * Skip an archive member
```

```

*/
void
skip(Biobuf *bp, vlong len)
{
    if (len & 01)
        len++;
    Bseek(bp, len, 1);
}

```

<function install 258a>≡

(273e)

```

/*
 * Stream the three temp files to an archive
 */
void
install(char *arname, Arfile *astart, Arfile *amiddle, Arfile *aend, int createflag)
{
    int fd;

    if(allobj && dupfound) {
        fprintf(2, "%s not changed\n", arname);
        return;
    }
    /* leave note group behind when copying back; i.e. sidestep interrupts */
    rfork(RFNOTEG);

    if(createflag)
        fprintf(2, "ar: creating %s\n", arname);
    fd = arcreate(arname);

    if(allobj)
        rl(fd);

    if (astart) {
        arstream(fd, astart);
        arfree(astart);
    }
    if (amiddle) {
        arstream(fd, amiddle);
        arfree(amiddle);
    }
    if (aend) {
        arstream(fd, aend);
        arfree(aend);
    }
    close(fd);
}

```

Uses allobj 245n, arcreate() 255b, arfree() 266a, arstream() 264c, dupfound 246b, and rl() 258b.

<function rl 258b>≡

(273e)

```

void
rl(int fd)
{
    Biobuf b;
    char *cp;
    struct ar_hdr a;
    long len;

    Binit(&b, fd, OWRITE);
    Bseek(&b, seek(fd,0,1), 0);
}

```

```

len = symdefsize;
if(len&01)
    len++;
sprintf(a.date, "%-12ld", time(0));
sprintf(a.uid, "%-6d", 0);
sprintf(a.gid, "%-6d", 0);
sprintf(a.mode, "%-8lo", 0644L);
sprintf(a.size, "%-10ld", len);
strncpy(a.fmag, ARFMAG, 2);
strcpy(a.name, symdef);
for (cp = strchr(a.name, 0); /* blank pad on right */
     cp < a.name+sizeof(a.name); cp++)
    *cp = ' ';
if(HEADER_IO(Bwrite, &b, a))
    wrerr();

len += Boffset(&b);
if (astart) {
    wrsym(&b, len, astart->sym);
    len += astart->size;
}
if(amiddle) {
    wrsym(&b, len, amiddle->sym);
    len += amiddle->size;
}
if(aend)
    wrsym(&b, len, aend->sym);

if(symdefsize&0x01)
    Bputc(&b, 0);
Bterm(&b);
}

```

Uses HEADER_IO-17 245a, aend 273e, amiddle 273e, astart 273e, symdef 245g, symdefsize 246a, wrerr() 255c, and wrsym() 259a.

<function wrsym 259a>≡ (273e)

```

/*
 * Write the defined symbols to the symdef file
 */
void
wrsym(Biobuf *bp, long offset, Arsymref *as)
{
    int off;

    while(as) {
        Bputc(bp, as->type);
        off = as->offset+offset;
        Bputc(bp, off);
        Bputc(bp, off>>8);
        Bputc(bp, off>>16);
        Bputc(bp, off>>24);
        if (Bwrite(bp, as->name, as->len+1) != as->len+1)
            wrerr();
        as = as->next;
    }
}

```

Uses wrerr() 255c.

<function match 259b>≡ (273e)

```

/*

```

```

* Check if the archive member matches an entry on the command line.
*/
int
match(int count, char **files)
{
    int i;
    char name[ARNAME_SIZE+1];

    for(i=0; i<count; i++) {
        if(files[i] == 0)
            continue;
        trim(files[i], name, ARNAME_SIZE);
        if(strncmp(name, file, ARNAME_SIZE) == 0) {
            file = files[i];
            files[i] = 0;
            return 1;
        }
    }
    return 0;
}

```

Uses ARNAME_SIZE-19 246d, file 246f, and trim() 261a.

<function bamatch 260a>≡

(273e)

```

/*
* compare the current member to the name of the pivot member
*/
int
bamatch(char *file, char *pivot)
{
    static int state = 0;

    switch(state)
    {
    case 0: /* looking for position file */
        if (aflag) {
            if (strncmp(file, pivot, ARNAME_SIZE) == 0)
                state = 1;
        } else if (bflag) {
            if (strncmp(file, pivot, ARNAME_SIZE) == 0) {
                state = 2; /* found */
                return 1;
            }
        }
        break;
    case 1: /* found - after previous file */
        state = 2;
        return 1;
    case 2: /* already found position file */
        break;
    }
    return 0;
}

```

Uses ARNAME_SIZE-19 246d, aflag 245h, and bflag-18 245i.

<function mesg 260b>≡

(273e)

```

/*
* output a message, if 'v' option was specified
*/
void
mesg(int c, char *file)

```

```

{
    if(vflag)
        Bprint(&bout, "%c - %s\n", c, file);
}

```

Uses `bout` 246g and `vflag` 245m.

`<function trim 261a>`≡ (273e)

```

/*
 * isolate file name by stripping leading directories and trailing slashes
 */
void
trim(char *s, char *buf, int n)
{
    char *p;

    for(;;) {
        p = strrchr(s, '/');
        if (!p) { /* no slash in name */
            strncpy(buf, s, n);
            return;
        }
        if (p[1] != 0) { /* p+1 is first char of file name */
            strncpy(buf, p+1, n);
            return;
        }
        *p = 0; /* strip trailing slash */
    }
}

```

`<constant SUID 261b>`≡ (273e)

```

/*
 * utilities for printing long form of 't' command
 */
#define SUID 04000

```

`<constant SGID 261c>`≡ (273e)

```

#define SGID 02000

```

`<constant ROWN 261d>`≡ (273e)

```

#define ROWN 0400

```

`<constant WOWN 261e>`≡ (273e)

```

#define WOWN 0200

```

`<constant XOWN 261f>`≡ (273e)

```

#define XOWN 0100

```

`<constant RGRP 261g>`≡ (273e)

```

#define RGRP 040

```

`<constant WGRP 261h>`≡ (273e)

```

#define WGRP 020

```

`<constant XGRP 261i>`≡ (273e)

```

#define XGRP 010

```

`<constant ROTH 261j>`≡ (273e)

```

#define ROTH 04

```

<constant WOTH 262a>≡ (273e)

```
#define WOTH 02
```

<constant XOTH 262b>≡ (273e)

```
#define XOTH 01
```

<constant STXT 262c>≡ (273e)

```
#define STXT 01000
```

<function longt 262d>≡ (273e)

```
void
longt(Armember *bp)
{
    char *cp;

    pmode(strtoul(bp->hdr.mode, 0, 8));
    Bprint(&bout, "%3ld/%1ld", strtoul(bp->hdr.uid, 0, 0), strtoul(bp->hdr.gid, 0, 0));
    Bprint(&bout, "%7ld", bp->size);
    cp = ctime(bp->date);
    Bprint(&bout, " %-12.12s %-4.4s ", cp+4, cp+24);
}
```

Uses *bout 246g* and *pmode() 263b*.

<global m1 262e>≡ (273e)

```
int m1[] = { 1, ROWN, 'r', '-' };
```

Uses *ROWN-23 261d*.

<global m2 262f>≡ (273e)

```
int m2[] = { 1, WOWN, 'w', '-' };
```

Uses *WOWN-24 261e*.

<global m3 262g>≡ (273e)

```
int m3[] = { 2, SUID, 's', XOWN, 'x', '-' };
```

Uses *SUID-21 261b* and *XOWN-25 261f*.

<global m4 262h>≡ (273e)

```
int m4[] = { 1, RGRP, 'r', '-' };
```

Uses *RGRP-26 261g*.

<global m5 262i>≡ (273e)

```
int m5[] = { 1, WGRP, 'w', '-' };
```

Uses *WGRP-27 261h*.

<global m6 262j>≡ (273e)

```
int m6[] = { 2, SGID, 's', XGRP, 'x', '-' };
```

Uses *SGID-22 261c* and *XGRP-28 261i*.

<global m7 262k>≡ (273e)

```
int m7[] = { 1, ROTH, 'r', '-' };
```

Uses *ROTH-29 261j*.

<global m8 262l>≡ (273e)

```
int m8[] = { 1, WOTH, 'w', '-' };
```

Uses *WOTH-30 262a*.

<global m9 262m>≡ (273e)

```
int m9[] = { 2, STXT, 't', XOTH, 'x', '-' };
```

Uses *STXT-32 262c* and *XOTH-31 262b*.

<global m 263a>≡ (273e)

```
int *m[] = { m1, m2, m3, m4, m5, m6, m7, m8, m9};
```

Uses m1 262e, m2 262f, m3 262g, m4 262h, m5 262i, m6 262j, m7 262k, m8 262l, and m9 262m.

<function pmode 263b>≡ (273e)

```
void
pmode(long mode)
{
    int **mp;

    for(mp = &m[0]; mp < &m[9];)
        select(*mp++, mode);
}
```

Uses m 263a and select() 263c.

<function select 263c>≡ (273e)

```
void
select(int *ap, long mode)
{
    int n;

    n = *ap++;
    while(--n>=0 && (mode & (*ap++))==0)
        ap++;
    Bputc(&bout, *ap);
}
```

Uses bout 246g.

<function newtempfile 263d>≡ (273e)

```
/*
 * Temp file I/O subsystem. We attempt to cache all three temp files in
 * core. When we run out of memory we spill to disk.
 * The I/O model assumes that temp files:
 * 1) are only written on the end
 * 2) are only read from the beginning
 * 3) are only read after all writing is complete.
 * The architecture uses one control block per temp file. Each control
 * block anchors a chain of buffers, each containing an archive member.
 */
Arfile *
newtempfile(char *name) /* allocate a file control block */
{
    Arfile *ap;

    ap = (Arfile *) armalloc(sizeof(Arfile));
    ap->fname = name;
    return ap;
}
```

Uses armalloc() 266b.

<function newmember 263e>≡ (273e)

```
Armember *
newmember(void) /* allocate a member buffer */
{
    return (Armember *)armalloc(sizeof(Armember));
}
```

Uses armalloc() 266b.

```

⟨function arread 264a⟩≡ (273e)
void
arread(Biobuf *b, Armember *bp, int n) /* read an image into a member buffer */
{
    int i;

    bp->member = armalloc(n);
    i = Bread(b, bp->member, n);
    if (i < 0) {
        free(bp->member);
        bp->member = 0;
        rderr();
    }
}

```

Uses armalloc() 266b, free() 233b, and rderr() 256a.

```

⟨function arinsert 264b⟩≡ (273e)
/*
 * insert a member buffer into the member chain
 */
void
arinsert(Arfile *ap, Armember *bp)
{
    bp->next = 0;
    if (!ap->tail)
        ap->head = bp;
    else
        ap->tail->next = bp;
    ap->tail = bp;
}

```

```

⟨function arstream 264c⟩≡ (273e)
/*
 * stream the members in a temp file to the file referenced by 'fd'.
 */
void
arstream(int fd, Arfile *ap)
{
    Armember *bp;
    int i;
    char buf[8192];

    if (ap->paged) { /* copy from disk */
        seek(ap->fd, 0, 0);
        for (;;) {
            i = read(ap->fd, buf, sizeof(buf));
            if (i < 0)
                rderr();
            if (i == 0)
                break;
            if (write(fd, buf, i) != i)
                wrerr();
        }
        close(ap->fd);
        ap->paged = 0;
    }

    /* dump the in-core buffers */
    for (bp = ap->head; bp; bp = bp->next) {
        if (!arwrite(fd, bp))
            wrerr();
    }
}

```

```
    }
```

```
}
```

Uses `arwrite()` 265a, `rderr()` 256a, and `wrerr()` 255c.

<function arwrite 265a>≡ (273e)

```
/*
 * write a member to 'fd'.
 */
int
arwrite(int fd, Armember *bp)
{
    int len;

    if(HEADER_IO(write, fd, bp->hdr))
        return 0;
    len = bp->size;
    if (len & 01)
        len++;
    if (write(fd, bp->member, len) != len)
        return 0;
    return 1;
}
```

Uses `HEADER_IO-17` 245a.

<function page 265b>≡ (273e)

```
/*
 * Spill a member to a disk copy of a temp file
 */
int
page(Arfile *ap)
{
    Armember *bp;

    bp = ap->head;
    if (!ap->paged) { /* not yet paged - create file */
        ap->fname = mktemp(ap->fname);
        ap->fd = create(ap->fname, ORDWR|ORCLOSE, 0600);
        if (ap->fd < 0) {
            fprintf(2,"ar: can't create temp file\n");
            return 0;
        }
        ap->paged = 1;
    }
    if (!arwrite(ap->fd, bp)) /* write member and free buffer block */
        return 0;
    ap->head = bp->next;
    if (ap->tail == bp)
        ap->tail = bp->next;
    free(bp->member);
    free(bp);
    return 1;
}
```

Uses `arwrite()` 265a and `free()` 233b.

<function getspace 265c>≡ (273e)

```
/*
 * try to reclaim space by paging. we try to spill the start, middle,
 * and end files, in that order. there is no particular reason for the
 * ordering.
```

```

*/
int
getspace(void)
{
    if (astart && astart->head && page(astart))
        return 1;
    if (amiddle && amiddle->head && page(amide))
        return 1;
    if (aend && aend->head && page(aend))
        return 1;
    return 0;
}

```

Uses aend 273e, amiddle 273e, astart 273e, and page() 265b.

```

⟨function arfree 266a⟩≡ (273e)
void
arfree(Arfile *ap) /* free a member buffer */
{
    Armember *bp, *next;

    for (bp = ap->head; bp; bp = next) {
        next = bp->next;
        if (bp->member)
            free(bp->member);
        free(bp);
    }
    free(ap);
}

```

Uses free() 233b.

```

⟨function armalloc 266b⟩≡ (273e)
/*
 * allocate space for a control block or member buffer.  if the malloc
 * fails we try to reclaim space by spilling previously allocated
 * member buffers.
 */
char *
armalloc(int n)
{
    char *cp;

    do {
        cp = malloc(n);
        if (cp) {
            memset(cp, 0, n);
            return cp;
        }
    } while (getspace());
    fprintf(2, "ar: out of memory\n");
    exits("malloc");
    return 0;
}

```

Uses getspace() 265c and malloc() 233a.

E.5 strip

```

⟨function error (linkers/misc/strip.c) 266c⟩≡ (278b)

```

```

void
error(char* fmt, ...)
{
    va_list arg;
    char *e, s[256];

    va_start(arg, fmt);
    e = sprintf(s, s+sizeof(s), "%s: ", argv0);
    e = vsprintf(e, s+sizeof(s), fmt, arg);
    e = sprintf(e, s+sizeof(s), "\n");
    va_end(arg);

    write(2, s, e-s);
}

```

<function usage (linkers/misc/strip.c) 267a>≡

(278b)

```

static void
usage(void)
{
    error("usage: %s -o ofile file\n\t%s file ...\n", argv0, argv0);
    exits("usage");
}

```

<function strip 267b>≡

(278b)

```

static int
strip(char* file, char* out)
{
    Dir *dir;
    int fd, i;
    Fhdr fhdr;
    Exec *exec;
    ulong mode;
    void *data;
    vlong length;

    if((fd = open(file, OREAD)) < 0){
        error("%s: open: %r", file);
        return 1;
    }

    if(!crackhdr(fd, &fhdr)){
        error("%s: %r", file);
        close(fd);
        return 1;
    }

    for(i = MIN_MAGIC; i <= MAX_MAGIC; i++){
        if(fhdr.magic == _MAGIC(0, i) || fhdr.magic == _MAGIC(HDR_MAGIC, i))
            break;
    }

    if(i > MAX_MAGIC){
        error("%s: not a recognizeable binary", file);
        close(fd);
        return 1;
    }

    if((dir = dirfstat(fd)) == nil){
        error("%s: stat: %r", file);
        close(fd);
        return 1;
    }
}

```

```

length = fhdr.datoff+fhdr.datsz;
if(length == dir->length){
    if(out == nil){ /* nothing to do */
        error("%s: already stripped", file);
        free(dir);
        close(fd);
        return 0;
    }
}
if(length > dir->length){
    error("%s: strange length", file);
    close(fd);
    free(dir);
    return 1;
}

mode = dir->mode;
free(dir);

if((data = malloc(length)) == nil){
    error("%s: malloc failure", file);
    close(fd);
    return 1;
}
seek(fd, OLL, 0);
if(read(fd, data, length) != length){
    error("%s: read: %r", file);
    close(fd);
    free(data);
    return 1;
}
close(fd);

exec = data;
exec->syms = 0;
exec->_unused = 0;
exec->pcsz = 0;

if(out == nil){
    if(remove(file) < 0) {
        error("%s: remove: %r", file);
        free(data);
        return 1;
    }
    out = file;
}
if((fd = create(out, OWRITE, mode)) < 0){
    error("%s: create: %r", out);
    free(data);
    return 1;
}
if(write(fd, data, length) != length){
    error("%s: write: %r", out);
    close(fd);
    free(data);
    return 1;
}
close(fd);
free(data);

```

```
    return 0;
}
```

Uses `free()` 233b and `malloc()` 233a.

<function main (linkers/misc/strip.c) 269>≡

(278b)

```
void
main(int argc, char* argv[])
{
    int r;
    char *p;

    p = nil;

    ARGBEGIN{
    default:
        usage();
        break;
    case 'o':
        p = ARGF();
        if(p == nil)
            usage();
        break;
    }ARGEND;

    switch(argc){
    case 0:
        usage();
        return;
    case 1:
        if(p != nil){
            r = strip(*argv, p);
            break;
        }
        /*FALLTHROUGH*/
    default:
        r = 0;
        while(argc > 0){
            r |= strip(*argv, nil);
            argc--;
            argv++;
        }
        break;
    }

    if(r)
        exits("error");
    exits(0);
}
```

Appendix F

Extra Code

F.1 include/

F.1.1 include/exec/a.out.h

```
<function _MAGIC 270a>≡ (270i)
#define _MAGIC(f, b) (((f)|((((4*(b))+0)*(b))+7))

<constant I_MAGIC 270b>≡ (270i)
#define I_MAGIC _MAGIC(0, 11) /* intel 386 */

<constant E_MAGIC 270c>≡ (270i)
#define E_MAGIC _MAGIC(0, 20) /* arm */

<constant HDR_MAGIC 270d>≡ (270i)
#define HDR_MAGIC 0x00008000 /* header expansion */

<constant MIN_MAGIC 270e>≡ (270i)
#define MIN_MAGIC 11

<constant MAX_MAGIC 270f>≡ (270i)
#define MAX_MAGIC 20 /* <= 90 */

<constant DYN_MAGIC 270g>≡ (270i)
#define DYN_MAGIC 0x80000000 /* dlm */

<struct Sym a.out.h 270h>≡ (270i)
struct Sym
{
    vlong value;
    uint sig;
    char type;
    char *name;
};

<include/a.out.h 270i>≡
typedef struct Exec Exec;
typedef struct Sym Sym;

<struct Exec 24>

<constant HDR_MAGIC 270d>

<function _MAGIC 270a>
```

```

<constant I_MAGIC 270b>
<constant E_MAGIC 270c>

#define V_MAGIC _MAGIC(0, 16) /* mips 3000 BE */

<constant MIN_MAGIC 270e>
<constant MAX_MAGIC 270f>

<constant DYN_MAGIC 270g>

<struct Sym a.out.h 270h>

```

F.1.2 include/exec/elf.h

```

<enum ElfConstants 271>≡
/* was in /sys/src/libmach/elf.h */
enum {
    /* Ehdr codes */
    MAG0 = 0, /* ident[] indexes */
    MAG1 = 1,
    MAG2 = 2,
    MAG3 = 3,
    CLASS = 4,
    DATA = 5,
    VERSION = 6,

    ELFCLASSNONE = 0, /* ident[CLASS] */
    ELFCLASS32 = 1,
    ELFCLASS64 = 2,
    ELFCLASSNUM = 3,

    ELFDATANONE = 0, /* ident[DATA] */
    ELFDATA2LSB = 1,
    ELFDATA2MSB = 2,
    ELFDATANUM = 3,

    NOETYPE = 0, /* type */
    REL = 1,
    EXEC = 2,
    DYN = 3,
    CORE = 4,

    NONE = 0, /* machine */
    M32 = 1, /* AT&T WE 32100 */
    SPARC = 2, /* Sun SPARC */
    I386 = 3, /* Intel 80386 */
    M68K = 4, /* Motorola 68000 */
    M88K = 5, /* Motorola 88000 */
    I486 = 6, /* Intel 80486 */
    I860 = 7, /* Intel i860 */
    MIPS = 8, /* Mips R2000 */
    S370 = 9, /* Amdhal */
    MIPSR4K = 10, /* Mips R4000 */
    SPARC64 = 18, /* Sun SPARC v9 */
    POWER = 20, /* PowerPC */
    POWER64 = 21, /* PowerPC64 */
    ARM = 40, /* ARM */
    AMD64 = 62, /* Amd64 */
    ARM64 = 183, /* ARM64 */

```

(273c)

```

NO_VERSION = 0, /* version, ident[VERSION] */
CURRENT = 1,

/* Phdr Codes */
NOPTYPE = 0, /* type */
PT_LOAD = 1, /* also LOAD */
DYNAMIC = 2,
INTERP = 3,
NOTE = 4,
SHLIB = 5,
PHDR = 6,

R = 0x4, /* flags */
W = 0x2,
X = 0x1,

/* Shdr Codes */
Progbits = 1, /* section types */
Strtab = 3,
Nobits = 8,

SwriteElf = 1, /* section attributes (flags) */
Salloc = 2,
Sexec = 4,
};

```

<struct Ehdr 272a>≡

(273c)

```

/*
 * Definitions needed for accessing ELF headers
 */
struct Ehdr {
    uchar ident[16]; /* ident bytes */
    ushort type; /* file type */
    ushort machine; /* target machine */
    int version; /* file version */
    ulong elfentry; /* start address */
    ulong phoff; /* phdr file offset */
    ulong shoff; /* shdr file offset */
    int flags; /* file flags */
    ushort ehsize; /* sizeof ehdr */
    ushort phentsize; /* sizeof phdr */
    ushort phnum; /* number phdrs */
    ushort shentsize; /* sizeof shdr */
    ushort shnum; /* number shdrs */
    ushort shstrndx; /* shdr string index */
};

```

<struct Phdr 272b>≡

(273c)

```

struct Phdr {
    int type; /* entry type */
    ulong offset; /* file offset */
    ulong vaddr; /* virtual address */
    ulong paddr; /* physical address */
    int filesz; /* file size */
    ulong memsz; /* memory size */
    int flags; /* entry flags */
    int align; /* memory/file alignment */
};

```

```

<struct Shdr 273a>≡ (273c)
struct Shdr {
    ulong name; /* section name */
    ulong type; /* SHT_... */
    ulong flags; /* SHF_... */
    ulong addr; /* virtual address */
    ulong offset; /* file offset */
    ulong size; /* section size */
    ulong link; /* misc info */
    ulong info; /* misc info */
    ulong addralign; /* memory alignment */
    ulong entsize; /* entry size if table */
};

```

```

<constant ELF_MAG 273b>≡ (273c)
#define ELF_MAG ((0x7f<<24) | ('E'<<16) | ('L'<<8) | 'F')

```

```

<include/exec/elf.h 273c>≡

```

```

<enum ElfConstants 271>

```

```

typedef struct Ehdr Ehdr;
typedef struct Phdr Phdr;
typedef struct Shdr Shdr;

```

```

<struct Ehdr 272a>

```

```

<struct Phdr 272b>

```

```

<struct Shdr 273a>

```

```

<constant ELF_MAG 273b>

```

F.1.3 include/obj/ar.h

```

<include/ar.h 273d>≡

```

```

<constant ARMAG 70a>

```

```

<constant SARMAG 70b>

```

```

<constant ARFMAG 70c>

```

```

<constant SARNAME 70d>

```

```

<struct ar_hdr 70e>

```

```

<constant SAR_HDR 71a>

```

F.2 linkers/misc/

F.2.1 linkers/misc/ar.c

```

<linkers/misc/ar.c 273e>≡

```

```

/*
 * ar - portable (ascii) format version
 */
#include <u.h>
#include <libc.h>
#include <bio.h>

```

```

#include <mach.h>
#include <ar.h>

/*
 * The algorithm uses up to 3 temp files.  The "pivot member" is the
 * archive member specified by and a, b, or i option.  The temp files are
 * astart - contains existing members up to and including the pivot member.
 * amiddle - contains new files moved or inserted behind the pivot.
 * aend - contains the existing members that follow the pivot member.
 * When all members have been processed, function 'install' streams the
 * temp files, in order, back into the archive.
 */

<struct Arsymref 244a>

<struct Armember 244b>

<struct Arfile 244c>

<struct Hashchain 244d>

<constant NHASH 244e>

<function HEADER_IO 245a>

    /* constants and flags */
<global man 245b>
<global opt 245c>
<global artemp 245d>
<global movtemp 245e>
<global tailtemp 245f>
<global symdef 245g>

<global aflag 245h>
<global bflag 245i>
<global cflag 245j>
<global oflag 245k>
<global uflag 245l>
<global vflag 245m>

Arfile *astart, *amiddle, *aend; /* Temp file control block pointers */

<global allobj 245n>
<global symdefsize 246a>
<global dupfound 246b>
<global hash 246c>

<constant ARNAME_SIZE 246d>

<global poname 246e>
<global file 246f>
<global bout 246g>
<global bar 246h>

void arcopy(Biobuf*, Arfile*, Armember*);
int arcreate(char*);
void arfree(Arfile*);
void arinsert(Arfile*, Armember*);
char *armalloc(int);
void armove(Biobuf*, Arfile*, Armember*);

```

```

void arread(Biobuf*, Armember*, int);
void arstream(int, Arfile*);
int arwrite(int, Armember*);
int bamatch(char*, char*);
int duplicate(char*);
Armember *getdir(Biobuf*);
int getspace(void);
void install(char*, Arfile*, Arfile*, Arfile*, int);
void longt(Armember*);
int match(int, char**);
void mesg(int, char*);
Arfile *newtempfile(char*);
Armember *newmember(void);
void objsym(Sym*, void*);
int openar(char*, int, int);
int page(Arfile*);
void pmode(long);
void rl(int);
void scanobj(Biobuf*, Arfile*, long);
void select(int*, long);
void setcom(void(*)(char*, int, char**));
void skip(Biobuf*, vlong);
int symcomp(void*, void*);
void trim(char*, char*, int);
static void usage(void);
void wrerr(void);
void wrsym(Biobuf*, long, Arsymref*);

void rcmd(char*, int, char**); /* command processing */
void dcmd(char*, int, char**);
void xcmd(char*, int, char**);
void tcmd(char*, int, char**);
void pcmd(char*, int, char**);
void mcmd(char*, int, char**);
void qcmd(char*, int, char**);

```

<global comfun 246i>

<function main 246j>

<function setcom 247a>

<function rcmd 247b>

<function dcmd 249>

<function xcmd 250a>

<function pcmd 250b>

<function mcmd 251>

<function tcmd 252a>

<function qcmd 252b>

<function scanobj 253>

<function objsym 254a>

<function duplicate 254b>

<function openar 255a>

<function arcreate 255b>

<function wrerr 255c>
<function rderr 256a>
<function phaseerr 256b>
<function usage 256c>
<function getdir 256d>
<function armove 257a>
<function arcopy 257b>
<function skip 257c>
<function install 258a>
<function rl 258b>
<function wrsym 259a>
<function match 259b>
<function bamatch 260a>
<function mesg 260b>
<function trim 261a>
<constant SUID 261b>
<constant SGID 261c>
<constant ROWN 261d>
<constant WOWN 261e>
<constant XOWN 261f>
<constant RGRP 261g>
<constant WGRP 261h>
<constant XGRP 261i>
<constant ROTH 261j>
<constant WOTH 262a>
<constant XOTH 262b>
<constant STXT 262c>
<function longt 262d>
<global m1 262e>
<global m2 262f>
<global m3 262g>
<global m4 262h>
<global m5 262i>
<global m6 262j>
<global m7 262k>
<global m8 262l>
<global m9 262m>
<global m 263a>
<function pmode 263b>
<function select 263c>

<function newtempfile 263d>
 <function newmember 263e>
 <function arread 264a>
 <function arinsert 264b>
 <function arstream 264c>
 <function arwrite 265a>
 <function page 265b>
 <function getspace 265c>
 <function arfree 266a>
 <function armalloc 266b>

F.2.2 linkers/misc/nm.c

```

<linkers/misc/nm.c 277>≡
/*
 * nm.c -- drive nm
 */
#include <u.h>
#include <libc.h>
#include <bio.h>

#include <mach.h>
#include <ar.h>

<enum NmConstants 238a>

<global errs 238b>
<global filename 238c>
<global symname 238d>
<global multifile 238e>
<global aflag (linkers/misc/nm.c) 238f>
<global gflag 238g>
<global hflag 238h>
<global nflag 238i>
<global sflag 238j>
<global uflag (linkers/misc/nm.c) 238k>
<global Tflag 238l>

<global fnames 238m>
<global symptr 238n>
<global nsym 239a>
<global bout (linkers/misc/nm.c) 239b>

int cmp(void*, void*);
static void error(char*, ...);
void execsyms(int);
void psym(Sym*, void*);
void printsyms(Sym**, long);
void doar(Biobuf*);
  
```

```

void dofile(Biobuf*);
void zenter(Sym*);

⟨function usage (linkers/misc/nm.c) 239c⟩
⟨function main (linkers/misc/nm.c) 239d⟩
⟨function doar 239e⟩
⟨function dofile 240a⟩
⟨function cmp_symbol 240b⟩
⟨function zenter 241a⟩
⟨function execsyms 241b⟩
⟨function psym 242⟩
⟨function printsyms 243a⟩
⟨function error 243b⟩

```

F.2.3 linkers/misc/size.c

```

⟨linkers/misc/size.c 278a⟩≡
#include <u.h>
#include <libc.h>
#include <bio.h>

#include <mach.h>

⟨function size 237a⟩

⟨function main (linkers/misc/size.c) 237b⟩

```

F.2.4 linkers/misc/strip.c

```

⟨linkers/misc/strip.c 278b⟩≡
#include <u.h>
#include <libc.h>
#include <bio.h>

#include <mach.h>

⟨function error (linkers/misc/strip.c) 266c⟩
⟨function usage (linkers/misc/strip.c) 267a⟩
⟨function strip 267b⟩
⟨function main (linkers/misc/strip.c) 269⟩

```

F.3 linkers/5l/

F.3.1 linkers/5l/l.h

```

⟨enum misc_constant (arm) 278c⟩≡

```

(279)

```

enum misc_constants {
    <constant BIG 182e>

    <constant STRINGSZ 219c>
    <constant NHASH linker 31b>
    <constant NHUNK linker 232d>

    <constant MINSIZ 183b>
    <constant MAXIO 234a>
    <constant MAXHIST 157b>
};

<linkers/5l/l.h 279>≡
#include <u.h>
#include <libc.h>
#include <bio.h>

#include <common.out.h>
#include <5.out.h>

#include "../8l/elf.h"

//-----
// Data structures and constants
//-----

// forward decls
typedef struct Adr Adr;
typedef struct Auto Auto;
typedef struct Prog Prog;
typedef struct Sym Sym;

<struct Adr(arm) 33c>

<struct Prog(arm) 34b>
<constant P 36c>

<struct Sym 30>
<constant S 31d>

<enum Section(arm) 33b>

<enum Mark(arm) 35g>

<enum headtype(arm) 40c>

<struct Auto(arm) 150a>

<enum rxxx 175c>
<enum misc_constant(arm) 278c>

<constant SIGNINTERN(arm) 203i>

<constant LIBNAMELEN 74b>

<struct Buf 233d>

//-----
// Globals
//-----

```

```

// io.c
extern union Buf buf;
extern int    cbc;
extern char*  cbp;

// globals.c

extern char  thechar;
extern char* thestring;

// configuration
extern short  HEADTYPE;      /* type of header */
extern long   HEADR;        /* length of header */
extern long   INITTEXT;     /* text location */
extern long   INITRND;      /* data round above text location */
extern long   INITDAT;      /* data location */
extern char*  INITENTRY;    /* entry point */
extern long   INITTEXTP;    /* text location (physical) */ // ELF

// output
extern char*  outfile;
extern fdt    cout;
extern Biobuf bso;

// core algorithm
extern Sym*   hash[NHASH];
extern long   pc;
extern Prog   zprg;

extern Prog*  firstp;
extern Prog*  lastp;
extern Prog*  datap;
extern Prog*  textp;
extern Prog*  etextp;

extern Prog*  curtext;
extern Auto*  curauto;
extern Auto*  curhist;
extern Prog*  curp;
extern long   autosize;

// sections size
extern long   textsize;
extern long   datsize;
extern long   bsssize;
extern long   symsize;
extern long   lcsiz;

extern char*  noname;
<constant TNAME(arm) 37a>

// debugging support
extern Sym*   histfrog[MAXHIST];
extern int    histfrogp;
extern int    histgen;

// library
extern int    xrefresolv;

```

```

// advanced topics
extern int armv4;
extern int vfp;
extern bool doexp;
extern bool dlm;
extern char*  EXPTAB;

extern Prog   undefp;
<constant UP 175f>

// debugging
extern bool   debug[128];
extern char*  anames[];

// utils (for statistics)
extern long   thunk;
extern long   nsymbol;

<pragmas varargck type 219a>
<pragmas varargck argpos 229c>

//-----
// Functions
//-----

// obj.c
int   isobjfile(char *f);
void  objfile(char*);

// lib.c
void  loadlib(void);
void  addlibpath(char*);
char* findlib(char *file);
void  addlib(char *obj);

// pass.c
void  patch(void);
void  follow(void);

// noops.c
void  noops(void);
void  divsig(void);
void  initdiv(void);
void  nocache(Prog*);

// layout.c
void  dodata(void);
void  dotext(void);

// span.c
void  buildop(void);
int   aclass(Adr*);
long  immrot(ulong);
long  immaddr(long);
// oplook() in m.h

long  regoff(Adr*); // for float

```

```

// datagen.c
void  nuxiinit(void);
void  datblk(long s, long n, bool sstring);

// codegen.c
// asmout() in m.h

// asm.c
void  asmb(void);

// hist.c
void  addhist(long, int);
void  histtoauto(void);

// debugging.c
void  asmsym(void);
void  asmlc(void);

// profile.c
void  doprof1(void);
void  doprof2(void);

// float.c
double  ieeeedtod(Ieee*);
long    ieeeedtof(Ieee*);
int     chipfloat(Ieee*);

// dynamic.c
void    zerosig(char*);
void    readundefs(char*, int);
void    dynreloc(Sym*, long, int);
void    asmdyn(void);
void    import(void);
void    export(void);
void    ckoff(Sym*, long);

// io.c
void    cput(int);
void    lput(long);
void    lputl(long l);
void    wput(long);
void    wputl(long);
void    cflush(void);
byte*  readsome(fdt f, byte *buf, byte *good, byte *stop, int max);

// error.c
void    diag(char*, ...);
void    errexit(void);

// utils.c
Sym*   lookup(char*, int);
Prog*  prg(void);
// and malloc/free/setmalloctag overwrite
long   atolwhex(char*);
long   rnd(long, long);
int    fileexists(char*);
void   mylog(char*, ...);

```

```

<macro DBG 225a>

// fmt.c (dumpers)
void listinit(void);
void prasm(Prog*);

```

Uses Adr 279, Auto 279, Prog 279, and Sym 279.

F.3.2 linkers/5l/m.h

```

<linkers/5l/m.h 283a>≡

typedef struct Optab Optab;
typedef struct Oprange Oprange;

<enum Operand_class(arm) 101a>

<struct Optab(arm) 99a>

<struct Oprange(arm) 100b>

<enum Optab_flag(arm) 141c>

// globals
extern Optab optab[];
extern long instoffset;

// span.c
Optab* oplook(Prog*);

// codegen.c
void asmout(Prog*, Optab*);

```

Uses Oprange 100b and Optab 99a.

F.3.3 linkers/5l/globals.c

```

<linkers/5l/globals.c 283b>≡
#include "l.h"
#include "m.h"

<global thechar 38a>
<global thestring 38b>

<global HEADR 40d>
<global HEADTYPE 40a>
<global INITDAT 40g>
<global INITRND 40f>
<global INITTEXT 40e>
<global INITTEXTP 188d>
<global INITENTRY 42a>

<global outfile 38c>
<global cout 38d>
<global bso 218c>

<global curauto 150d>

```

<global curhist 155b>
<global curp 36g>
<global curtext 36f>

*<global autosize(*arm*) 65d>*
*<global instoffset(*arm*) 112f>*

<global datap 36e>
<global etextp 152b>
<global firstp 36a>
<global lastp 36d>
<global textp 152a>

<global debug 218a>

<global textsize 94a>
<global datsize 92a>
<global bsssize 92b>
<global symsize 147c>
<global lcsiz 160a>

<global hash linker 31a>
<global pc 53a>
<global zprg 43b>

<global histfrog 157a>
<global histfrogp 157d>
<global histgen 156e>

<global xrefresolv 76f>

<global thunk 232c>
<global nsymbol linker 231b>

*<global armv4(*arm*) 208c>*
*<global vfp(*arm*) 191f>*

<global doexp 170c>
<global dlm 170a>

<global EXPTAB 171a>
<global undefp 175e>

F.3.4 linkers/5l/optab.c

```
<linkers/5l/optab.c 284>≡  
#include "l.h"  
#include "m.h"
```

*<global optab (linkers/5l/optab.c)(*arm*) 99b>*

F.3.5 linkers/5l/utils.c

```
<linkers/5l/utils.c 285a>≡
#include "l.h"

// Utilities.

<function log 225b>

<constructor prg 43a>

<function lookup 32a>

<function atolwhex 235c>

<function rnd 236a>

<function fileexists 235b>

<global hunk 232a>
<global nhunk 232b>
// thunk defined in globals.c because also used by main.c for profiling report

<function gethunk 232e>

<function malloc 233a>

<function free 233b>

<function setmalloctag 233c>
```

F.3.6 linkers/5l/error.c

```
<linkers/5l/error.c 285b>≡
#include "l.h"

<global nerrors 229a>

<function errexit 229b>

<function diag 229d>
```

F.3.7 linkers/5l/fmt.c

```
<linkers/5l/fmt.c 285c>≡
#include "l.h"

// Printing.

<function Pconv(arm) 219d>

<function Aconv(arm) 220a>

<global strcond(arm) 223a>

<function Cconv(arm) 223b>
```

<function Dconv(arm) 220b>

<function Nconv(arm) 222>

<function Sconv(arm) 224>

<function listinit(arm) 218e>

<function prasm(arm) 219b>

F.3.8 linkers/5l/layout.c

<linkers/5l/layout.c 286a>≡

```
#include "l.h"
```

```
#include "m.h"
```

```
// Data layout and relocation.
```

<global pool(arm) 144a>

<global blitrl(arm) 140>

<global elitrl(arm) 141a>

```
void checkpool(Prog*);
```

```
void flushpool(Prog*, int);
```

```
void addpool(Prog*, Adr*);
```

<function xdefine(arm) 97a>

<function dodata(arm) 92c>

<function span(arm) 94b>

<function checkpool(arm) 143d>

<function flushpool(arm) 143b>

<function addpool(arm) 141f>

F.3.9 linkers/5l/pass.c

<linkers/5l/pass.c 286b>≡

```
#include "l.h"
```

```
// Code and data passes.
```

```
// forward decls
```

```
void xfol(Prog*);
```

<function brchain(arm) 185b>

<function relinv(arm) 186a>

<function follow 184a>

<function xfol(arm) 184c>

<constant LOG 85b>

<function mkfwd 85c>

<function brloop(arm) 183f>

<function patch(arm) 82>

F.3.10 linkers/5l/datagen.c

<linkers/5l/datagen.c 287a>≡

```
#include "l.h"
```

<constant Dbufslop 49a>

<global inuxi1 51a>

<global inuxi2 51b>

<global inuxi4 51c>

<global fnuxi4 194a>

<global fnuxi8 194b>

<function find1 51f>

<function nuxiinit(arm) 51e>

<function datblk(arm) 49b>

F.3.11 linkers/5l/dynamic.c

<linkers/5l/dynamic.c 287b>≡

```
#include "l.h"
```

```
// forward decls
```

```
typedef struct Reloc Reloc;
```

<enum SpanConstants(arm) 179b>

<global modemap 178b>

<struct Reloc 177c>

<global rels 177d>

<global imports 171f>

<global nimports 171d>

<global exports 171g>
<global nexports 171e>

<function zerosig 171c>

<function readundefs 172a>

<function undefsym 176b>

*<function import(*arm*) 175b>*

<function ckoff 176d>

*<function newdata(*arm*) 174a>*

*<function export(*arm*) 172b>*

<function grow 179d>

*<function dynreloc(*arm*) 179c>*

<function sput 178a>

<function asmdyn 177f>

Uses Reloc 177c.

F.3.12 linkers/51/codegen.c

<linkers/51/codegen.c 288>≡

#include "l.h"

#include "m.h"

*<function oprrr(*arm*) 120a>*

*<function opvfprrr(*arm*) 200c>*

*<function opbra(*arm*) 128c>*

*<function olr(*arm*) 131d>*

*<function olhr(*arm*) 211c>*

*<function osr(*arm*) 133d>*

*<function oshr(*arm*) 212a>*

*<function olrr(*arm*) 133a>*

*<function olhrr(*arm*) 212c>*

*<function osrr(*arm*) 134b>*

*<function oshrr(*arm*) 212b>*

<function ovfpmem(arm) 199a>

<function ofs(arm) 195d>

<function omvl(arm) 125c>

<function asmout(arm) 108b>

F.3.13 linkers/51/io.c

<linkers/51/io.c 289a>≡

```
#include "l.h"
```

<global buf 234b>

<global cbc 109b>

<global cbp 109a>

<function readsome 59d>

<function strnput(arm) 235a>

<function cput(arm) 234c>

<function wput(arm) 234e>

<function wputl(arm) 234d>

<function lput(arm) 110b>

<function lputl(arm) 109d>

<function cflush 110a>

F.3.14 linkers/51/asm.c

<linkers/51/asm.c 289b>≡

```
#include "l.h"
```

```
#include "m.h"
```

```
// Writing object files.
```

<function entryvalue(arm) 47a>

<function asmb(arm) 45a>

F.3.15 linkers/51/span.c

<linkers/51/span.c 289c>≡

```

#include "l.h"
#include "m.h"

// Instruction layout.

<global oprange(arm) 100c>
<global xcmp(arm) 107d>

<function cmp(arm) 103c>

<function ocmp(arm) 105b>

<function buildop(arm) 104c>

<function regoff(arm) 195b>

<function immrot(arm) 113c>

<function immaddr(arm) 115a>

<function immfloat(arm) 190c>

<function immhalf(arm) 209h>

<function aclass(arm) 101b>

<function oplook(arm) 102c>

```

F.3.16 linkers/5l/obj.c

```

<linkers/5l/obj.c 290>≡
#include "l.h"
#include <ar.h>

// Reading object files.

<global noname linker 37b>
<global symname linker 71b>

<global version 64a>

<global literal(arm) 191j>

<function isobjfile 170e>

<function inopd(arm) 57b>

<function collapsefrog 158b>

<function nopout 187f>

<function ldoobj(arm) 55>

```

<function objfile 44a>

F.3.17 linkers/5l/lib.c

```
<linkers/5l/lib.c 291a>≡  
#include "l.h"
```

<global library 76a>
<global libraryobj 76c>
<global libraryp 76b>

<global libdir 73c>
<global nlibdir 73d>
<global maxlibdir 73e>

<function addlibpath 74f>

<function findlib 75b>

<function loadlib 76e>

<function addlib 77b>

F.3.18 linkers/5l/noop.c

```
<linkers/5l/noop.c 291b>≡  
#include "l.h"
```

```
// Code transformations.
```

<global sym_div(arm) 203e>
<global sym_divu(arm) 203f>
<global sym_mod(arm) 203g>
<global sym_modu(arm) 203h>

<global prog_div(arm) 203a>
<global prog_divu(arm) 203b>
<global prog_mod(arm) 203c>
<global prog_modu(arm) 203d>

<function noops(arm) 87>

<function sigdiv(arm) 203j>

<function divsig(arm) 204a>

<function sdiv(arm) 204b>

<function initdiv(arm) 204c>

<function nocache(arm) 107c>

F.3.19 linkers/5l/float.c

```
<linkers/5l/float.c 291c>≡  
#include "l.h"
```

<function ieedtoof 189c>
<function ieedtod 189d>
<global chipfloats(arm) 192>
<function chipfloat(arm) 193a>

F.3.20 linkers/5l/profile.c

```
<linkers/5l/profile.c 292a>≡  
#include "l.h"  
  
// Profiling.  
  
<function doprof1(arm) 163>  
  
<global brcond(arm) 168b>  
  
<function doprof2(arm) 166a>
```

F.3.21 linkers/5l/debugging.c

```
<linkers/5l/debugging.c 292b>≡  
#include "l.h"  
  
// Symbol table.  
  
<function putsymb 148a>  
  
<function asmsym(arm) 148b>  
  
  
<constant MINLC(arm) 160b>  
<function asmlc 160c>
```

F.3.22 linkers/5l/hist.c

```
<linkers/5l/hist.c 292c>≡  
#include "l.h"  
  
<function addhist 155a>  
  
<function histtoauto 156b>
```

F.3.23 linkers/5l/main.c

```
<linkers/5l/main.c 292d>≡  
#include "l.h"  
  
<function usage, linker 39a>
```

<function undef 52d>

<function main(arm) 38e>

Glossary

LDR = Load Register

STR = Store Register

ARM = Acorn RISC Machine

RISC = Reduced Instruction Set Computer

CISC = Complex Instruction Set Computer

PC = Program Counter

SB = Static Base register

SP = Stack Pointer

FP = Frame Pointer

Indexes

Here is a list of the identifiers used, and where they appear. Underlined entries indicate the place of definition. This index is generated automatically.

ABSD-3: [179b](#), [179c](#)
ABSU-4: [179b](#), [179c](#)
aclass(): [101b](#), [102c](#), [107a](#), [112e](#), [121b](#), [122a](#), [122d](#), [125c](#), [129d](#), [130a](#), [131a](#), [133c](#), [135a](#), [135d](#), [136b](#), [138a](#), [139c](#), [141f](#), [195b](#), [207a](#), [207b](#), [208b](#), [210b](#), [210c](#)
Aconv(): [218e](#), [220a](#)
addhist(): [153](#), [155a](#)
addlib(): [75c](#), [77b](#)
addlibpath(): [74f](#)
addpool(): [141e](#), [141f](#)
Adr: [279](#), [279](#)
Adr (typedef): [279](#)
aend: [247b](#), [251](#), [258b](#), [265c](#), [273e](#)
aflag: [245h](#), [246j](#), [252b](#), [260a](#)
aflag-38: [238f](#), [242](#)
allobj: [245n](#), [245n](#), [249](#), [253](#), [258a](#)
amiddle: [251](#), [258b](#), [265c](#), [273e](#)
arcopy(): [247b](#), [249](#), [250a](#), [250b](#), [251](#), [257b](#)
arcreate(): [252b](#), [255b](#), [258a](#)
Arfile: [244c](#), [244c](#)
Arfile.fd: [244c](#)
Arfile.fname: [244c](#)
Arfile.head: [244c](#)
Arfile.paged: [244c](#)
Arfile.size: [244c](#)
Arfile.sym: [244c](#)
Arfile.tail: [244c](#)
Arfile (typedef): [244c](#)
arfree(): [258a](#), [266a](#)
arinsert(): [257a](#), [257b](#), [264b](#)
armalloc(): [254a](#), [254b](#), [263d](#), [263e](#), [264a](#), [266b](#)
Armember: [244b](#), [244b](#)
Armember.date: [244b](#)
Armember.hdr: [244b](#)
Armember.member: [244b](#)
Armember.next: [244b](#)
Armember.size: [244b](#)
Armember (typedef): [244b](#)
armove(): [247b](#), [252b](#), [257a](#)

armv4: [208c](#), [208d](#), [208f](#)
ARNAME_SIZE-19: [246d](#), [246e](#), [252a](#), [256d](#), [259b](#), [260a](#)
arread(): [257a](#), [257b](#), [264a](#)
arstream(): [258a](#), [264c](#)
Arsymref: [244a](#), [244a](#)
Arsymref.len: [244a](#)
Arsymref.name: [244a](#)
Arsymref.next: [244a](#)
Arsymref.offset: [244a](#)
Arsymref.type: [244a](#)
Arsymref (typedef): [244a](#)
artemp: [245d](#), [245d](#), [246j](#), [247b](#), [249](#), [251](#)
arwrite(): [252b](#), [264c](#), [265a](#), [265b](#)
asmb(): [45a](#)
asmdyn(): [177a](#), [177e](#), [177f](#)
asmlc(): [147a](#), [160c](#)
asmout(): [47b](#), [108b](#)
asmsym(): [147a](#), [148b](#)
astart: [247b](#), [249](#), [251](#), [258b](#), [265c](#), [273e](#)
atolwhex(): [72](#), [214a](#), [235c](#)
Auto: [279](#), [279](#)
Auto (typedef): [279](#)
autosize: [65d](#), [66a](#), [66b](#), [88c](#), [89a](#), [89b](#), [116e](#), [116f](#), [117f](#), [118a](#)
bamatch(): [247b](#), [251](#), [260a](#)
bar: [246h](#), [247b](#), [249](#), [250a](#), [250b](#), [251](#), [252a](#), [252b](#)
bflag-18: [245i](#), [246j](#), [252b](#), [260a](#)
BIG: [115c](#), [117b](#), [182d](#)
blitrl: [140](#), [141f](#), [143a](#), [143b](#), [143c](#), [144b](#), [144d](#)
bout: [246g](#), [246j](#), [252a](#), [260b](#), [262d](#), [263c](#)
bout-48: [239b](#), [243a](#)
brchain(): [185a](#), [185b](#)
brcond-2: [168a](#), [292a](#)
brloop(): [183e](#), [183f](#)
bso: [51e](#), [51g](#), [148b](#), [160c](#), [172b](#), [175b](#), [218c](#), [225b](#), [226b](#), [226c](#), [226d](#), [226e](#), [226f](#), [226g](#), [226h](#), [226i](#), [227a](#), [227b](#),
[227c](#), [227d](#), [227e](#), [227f](#), [228a](#), [228b](#), [228c](#)
bssize: [46b](#), [92b](#), [92c](#), [95d](#)
Buf: [233d](#), [234b](#)
buf: [49b](#), [50b](#), [50c](#), [50e](#), [59b](#), [59c](#), [63b](#), [110a](#), [193d](#), [214g](#), [234b](#)
Buf.dbuf: [279](#)
buildop(): [104c](#)
cbc: [109b](#), [109d](#), [110a](#), [110b](#), [234c](#), [234d](#), [234e](#)
cbp: [109a](#), [109d](#), [110a](#), [110b](#), [234c](#), [234d](#), [234e](#)
Cconv(): [218e](#), [223b](#)
cflag: [245j](#), [246j](#), [247b](#), [252b](#)
cflush(): [45a](#), [47b](#), [109d](#), [110a](#), [110b](#), [147a](#), [177a](#), [177f](#), [234c](#), [234d](#), [234e](#)
checkpool(): [143c](#), [143d](#)
chipfloat(): [191k](#), [193a](#), [197c](#)
chipfloats-14: [192](#), [193a](#)
CHUNK-33: [238a](#), [241a](#)

ckoff(): [176c](#), [176d](#), [180e](#)
 cmp(): [102c](#), [103c](#), [107e](#), [114d](#), [115d](#), [116d](#)
 cmp_symbol(): [240b](#), [243a](#)
 collapsefrog(): [158a](#), [158b](#)
 comfun: [246i](#), [246j](#), [247a](#)
 cout: [38d](#), [38d](#), [46a](#), [47b](#), [48d](#), [49b](#), [110a](#), [147b](#), [176f](#), [177a](#), [177f](#), [188a](#), [229b](#)
 cput(): [148a](#), [159d](#), [160c](#), [177f](#), [178a](#), [234c](#), [235a](#)
 curauto: [150d](#), [151b](#), [151c](#), [151d](#), [151e](#), [156b](#)
 curhist: [155a](#), [155b](#), [156b](#)
 curp: [36g](#), [47b](#), [48b](#), [49b](#), [50e](#), [120a](#), [128c](#), [132c](#), [132d](#), [200c](#), [211c](#), [219d](#), [220b](#)
 curtext: [36f](#), [36f](#), [36h](#), [48c](#), [65b](#), [67b](#), [87](#), [88b](#), [88c](#), [89a](#), [89b](#), [151d](#), [151e](#), [204c](#), [205a](#), [229d](#)
 C.ADDR: [180a](#), [180b](#), [180c](#), [180d](#), [180e](#), [181a](#), [181c](#), [181e](#), [195a](#)
 C.BRANCH: [101a](#), [102a](#), [128a](#), [213d](#)
 C.FAUTO: [142c](#), [190f](#), [191a](#), [195a](#), [209c](#)
 C.FCON: [190a](#), [190b](#), [195a](#), [199b](#)
 C.FCR: [190a](#), [190b](#), [198d](#)
 C.FEXT: [190d](#), [190e](#), [195a](#), [209a](#)
 C.FOREG: [142c](#), [191b](#), [191c](#), [195a](#), [209g](#)
 C.FREG: [190a](#), [190b](#), [195a](#), [199b](#)
 C.GOK: [101a](#), [101b](#), [107d](#), [107e](#), [114b](#), [116a](#), [117a](#)
 C.HAUTO: [209b](#), [209c](#), [209d](#), [210a](#)
 C.HEXT: [208h](#), [208i](#), [209a](#), [210a](#)
 C.HFAUTO: [116d](#), [190f](#), [191a](#), [209c](#)
 C.HFEXT: [115d](#), [190d](#), [190e](#), [209a](#)
 C.HFOREG: [114d](#), [191b](#), [191c](#), [209g](#)
 C.HOREG: [209e](#), [209f](#), [209g](#), [210a](#)
 C.LACON: [117d](#), [117e](#), [118b](#), [135e](#), [142c](#)
 C.LAUTO: [116c](#), [116d](#), [116e](#), [116f](#), [132e](#), [133e](#), [134d](#), [137a](#), [138b](#), [142c](#), [195a](#), [210a](#)
 C.LCON: [104a](#), [111b](#), [112c](#), [113a](#), [113b](#), [117b](#), [126c](#), [127b](#), [139b](#), [176e](#), [180c](#), [180e](#), [206i](#)
 C.LEXT: [111b](#), [115b](#), [115c](#), [115d](#), [132e](#), [133e](#), [134d](#), [137a](#), [138b](#), [195a](#), [210a](#)
 C.LOREG: [114a](#), [114c](#), [114d](#), [132e](#), [133e](#), [134d](#), [137a](#), [138b](#), [139b](#), [142c](#), [195a](#), [210a](#)
 C.NCON: [104a](#), [113a](#), [113b](#), [125a](#), [127b](#)
 C.NONE: [99b](#), [101a](#), [102a](#), [102c](#), [111b](#), [112c](#), [119a](#), [121a](#), [121e](#), [122c](#), [123b](#), [123d](#), [124c](#), [125a](#), [126c](#), [127b](#), [128a](#),
[129c](#), [130c](#), [132e](#), [133b](#), [133e](#), [134c](#), [134d](#), [135c](#), [135e](#), [136a](#), [137a](#), [137c](#), [138b](#), [139b](#), [139e](#), [180c](#), [180d](#), [181a](#),
[181c](#), [181e](#), [195a](#), [198d](#), [199b](#), [205d](#), [206i](#), [207c](#), [210a](#), [212d](#), [213d](#)
 C.PSR: [207c](#), [207d](#), [207e](#)
 C.RACON: [117d](#), [117e](#), [118b](#), [135c](#)
 C.RCON: [104a](#), [113a](#), [113b](#), [121a](#), [122c](#), [207c](#)
 C.RECON: [116g](#), [117b](#), [135c](#)
 C.REGREG: [206b](#), [206c](#), [206d](#)
 C.REG: [101a](#), [102a](#), [102c](#), [111b](#), [119a](#), [121a](#), [121e](#), [122c](#), [123b](#), [123d](#), [124c](#), [125a](#), [126c](#), [127b](#), [130c](#), [132e](#), [133b](#),
[133e](#), [134c](#), [134d](#), [134f](#), [135c](#), [135e](#), [136a](#), [137a](#), [137c](#), [138b](#), [180c](#), [181a](#), [181c](#), [181e](#), [195a](#), [198d](#), [199b](#), [205d](#),
[206b](#), [207c](#), [210a](#), [212d](#), [213d](#)
 C.ROREG: [114a](#), [114c](#), [114d](#), [129c](#), [142c](#)
 C.SAUTO: [116c](#), [116d](#), [116e](#), [116f](#), [130c](#), [133b](#), [134c](#), [136a](#), [137c](#), [142c](#)
 C.SEXT: [115b](#), [115c](#), [115d](#), [130c](#), [133b](#), [134c](#), [136a](#), [137c](#)
 C.SHIFT: [121c](#), [121d](#), [121e](#), [212d](#)
 C.SOREG: [114a](#), [114c](#), [114d](#), [130c](#), [133b](#), [134c](#), [134f](#), [136a](#), [137c](#), [142c](#), [206i](#)
 C.SROREG: [114a](#), [114c](#), [114d](#), [142c](#)

datap: [36e](#), [36e](#), [49b](#), [66e](#), [92c](#), [163](#), [174a](#), [191k](#), [214c](#)
datblk(): [48c](#), [49b](#), [214g](#)
datsize: [46b](#), [48c](#), [92a](#), [92c](#), [95d](#), [147b](#), [177a](#)
DBG: [44a](#), [45a](#), [46a](#), [72](#), [76e](#), [76h](#), [82](#), [87](#), [89a](#), [92c](#), [94b](#), [144d](#), [147a](#), [163](#), [177f](#), [184a](#), [279](#)
Dbufslop-7: [49a](#), [49b](#)
dcmd(): [246j](#), [249](#)
Dconv(): [218e](#), [220b](#)
debug: [48b](#), [51g](#), [147a](#), [148b](#), [160c](#), [188c](#), [191g](#), [194c](#), [205b](#), [208d](#), [214c](#), [214e](#), [218a](#), [225c](#), [225d](#), [225e](#), [226b](#),
[226c](#), [226d](#), [226e](#), [226f](#), [226g](#), [226h](#), [226i](#), [227a](#), [227b](#), [227d](#), [227e](#), [227f](#), [228a](#), [228b](#), [228c](#), [229d](#)
diag(): [44b](#), [47a](#), [48b](#), [49b](#), [50b](#), [50e](#), [52d](#), [55](#), [56b](#), [57a](#), [65c](#), [66d](#), [67a](#), [67c](#), [69a](#), [72](#), [73a](#), [74g](#), [75a](#), [77b](#), [83c](#),
[84](#), [93a](#), [93b](#), [94b](#), [95c](#), [103b](#), [104c](#), [108b](#), [116b](#), [117b](#), [120a](#), [120c](#), [126a](#), [128c](#), [129a](#), [129b](#), [132b](#), [132c](#), [132d](#),
[135b](#), [172a](#), [172b](#), [175g](#), [176a](#), [176b](#), [176d](#), [177f](#), [179c](#), [186a](#), [189c](#), [195d](#), [197c](#), [198a](#), [199a](#), [199c](#), [200a](#), [200c](#),
[204b](#), [204c](#), [207b](#), [211c](#), [212e](#), [213a](#), [213b](#), [229d](#), [232e](#)
divsig(): [204a](#)
dlm: [170a](#), [174c](#), [176e](#), [176f](#), [177a](#), [177b](#), [177e](#), [179a](#), [180b](#), [180e](#), [204c](#), [213f](#)
doar(): [239e](#)
dodata(): [92c](#)
doexp: [170c](#)
dofile(): [240a](#)
doprof1(): [163](#)
dotext(): [94b](#)
dupfound: [246b](#), [254a](#), [258a](#)
duplicate(): [254a](#), [254b](#)
dynreloc(): [176a](#), [179a](#), [179c](#), [180e](#), [213f](#)
elitrl: [141a](#), [141f](#), [143b](#)
entryvalue(): [46b](#), [47a](#)
error(): [266c](#)
errorexit(): [39a](#), [44b](#), [56b](#), [63b](#), [65c](#), [66d](#), [72](#), [74g](#), [75a](#), [95c](#), [104c](#), [172a](#), [229b](#), [229d](#), [232e](#)
errs-34: [238b](#), [243b](#)
etextp: [152b](#), [152b](#), [152c](#)
execsyms(): [240a](#), [241b](#)
export(): [172b](#)
exports: [171g](#), [172b](#), [177f](#)
EXPTAB: [171a](#), [172b](#)
file: [246f](#), [247b](#), [249](#), [250a](#), [250b](#), [251](#), [252a](#), [252b](#), [253](#), [256d](#), [257a](#), [259b](#)
fileexists(): [75b](#), [235b](#)
filename-35: [238c](#), [239e](#), [241b](#), [243a](#)
find1(): [51e](#), [51f](#)
findlib(): [75a](#), [75b](#), [77b](#)
firstp: [36a](#), [47b](#), [82](#), [84](#), [85c](#), [87](#), [94b](#), [160c](#), [163](#), [166a](#), [183e](#), [184a](#), [204c](#)
flushpool(): [143b](#), [143d](#), [144c](#), [144e](#), [145a](#)
fnames-45: [238m](#), [241a](#), [243a](#)
fnuxi4: [51g](#), [193d](#), [194a](#), [194c](#)
fnuxi8: [51g](#), [193d](#), [194b](#), [194c](#)
FOLL: [40c](#), [184c](#), [185a](#), [186b](#)
follow(): [184a](#)
free(): [69g](#), [74f](#), [172b](#), [179d](#), [233b](#), [247b](#), [250a](#), [250b](#), [252a](#), [252b](#), [253](#), [254a](#), [256d](#), [257a](#), [264a](#), [265b](#), [266a](#),
[267b](#)
getdir(): [247b](#), [249](#), [250a](#), [250b](#), [251](#), [252a](#), [256d](#)

gethunk(): [232e](#), [233a](#)
getspace(): [265c](#), [266b](#)
gflag-39: [238g](#), [242](#)
grow(): [179c](#), [179d](#)
hash: [31a](#)
Hashchain: [244d](#), [244d](#)
Hashchain.name: [244d](#)
Hashchain.next: [244d](#)
Hashchain (typedef): [244d](#)
HEADER_IO-17: [245a](#), [256d](#), [258b](#), [265a](#)
HEADR: [40d](#), [47b](#), [48d](#), [147b](#), [177a](#), [188a](#)
HEADTYPE: [40a](#), [40a](#), [46a](#), [48c](#), [147a](#)
Headtype: [279](#)
hflag-40: [238h](#), [243a](#)
histfrog: [77b](#), [157a](#), [157c](#), [157h](#), [158b](#)
histfrogp: [75c](#), [77b](#), [157c](#), [157d](#), [157e](#), [157f](#), [157h](#), [158b](#)
histgen: [156c](#), [156e](#), [156e](#)
histtoauto(): [155c](#), [156a](#), [156b](#)
hunk: [232a](#), [232e](#), [233a](#)
H_ELF: [150a](#), [188a](#), [188b](#), [188c](#)
H_PLAN9: [46b](#), [48d](#), [147b](#), [150a](#)
ieeedtod(): [189d](#), [189d](#), [220b](#)
ieeedtof(): [189c](#), [191k](#), [193d](#)
immaddr(): [114c](#), [115a](#), [115c](#), [116e](#), [116f](#), [144c](#)
immfloat(): [190c](#), [190e](#), [191a](#), [191c](#)
immhalf(): [190e](#), [191a](#), [191c](#), [208i](#), [209d](#), [209f](#), [209h](#)
immrot(): [113b](#), [113c](#), [114c](#), [117b](#), [118b](#), [121b](#), [124a](#), [125c](#), [129d](#), [130a](#), [135d](#), [138c](#), [182a](#), [208b](#)
import(): [175b](#)
imports: [171f](#), [176b](#), [177f](#)
INITDAT: [40g](#), [40g](#), [52c](#), [94b](#), [117b](#), [148b](#), [176e](#), [176f](#), [177b](#), [179a](#), [180b](#)
initdiv(): [204c](#), [205a](#)
INITENTRY: [42a](#), [42a](#), [47a](#)
INITRND: [40f](#), [40f](#), [94b](#)
INITTEXT: [40e](#), [40e](#), [47a](#), [47b](#), [94b](#), [97c](#), [160c](#), [214g](#)
INITTEXTP: [188d](#), [188d](#)
inopd(): [56a](#), [57b](#)
install(): [247b](#), [249](#), [251](#), [258a](#)
instoffset: [112e](#), [112f](#), [113b](#), [114c](#), [115c](#), [116e](#), [116f](#), [117b](#), [117f](#), [118a](#), [118b](#), [121b](#), [122d](#), [125c](#), [129d](#), [130a](#),
[131a](#), [133c](#), [135b](#), [135d](#), [136b](#), [138a](#), [139c](#), [142c](#), [176e](#), [180b](#), [190e](#), [191a](#), [191c](#), [195b](#), [207b](#), [208b](#), [208i](#), [209d](#),
[209f](#), [210b](#), [210c](#)
inuxi1: [50e](#), [51a](#), [51e](#), [51g](#)
inuxi2: [50e](#), [51b](#), [51e](#), [51g](#)
inuxi4: [50e](#), [51c](#), [51e](#), [51g](#)
isobjfile(): [170e](#)
lastp: [36d](#), [64d](#), [65b](#), [184a](#), [184c](#), [186b](#)
lcsiz: [46b](#), [147a](#), [160a](#), [160c](#), [227e](#)
ldobj(): [44a](#), [55](#), [72](#)
LEAF: [88b](#), [88c](#), [89a](#), [89b](#), [148b](#), [205a](#), [279](#)
LFROM: [126c](#), [127b](#), [132e](#), [135e](#), [137a](#), [141c](#), [141d](#), [141e](#), [181c](#), [195a](#), [210a](#)

libdir: [73c](#), [74f](#), [75b](#)
LIBNAMELEN: [71c](#), [75b](#), [77b](#), [233d](#)
library: [76a](#), [76e](#), [77b](#)
libraryobj: [76c](#), [76e](#), [77b](#)
libraryp: [76b](#), [76e](#), [77b](#)
listinit(): [218e](#)
literal: [191j](#), [191k](#)
loadlib(): [76e](#)
LOG-15: [85b](#), [85c](#), [86a](#), [86b](#)
longt(): [252a](#), [262d](#)
lookup(): [32a](#), [47a](#), [61d](#), [72](#), [97a](#), [97c](#), [149](#), [163](#), [166a](#), [171c](#), [172a](#), [172b](#), [191k](#), [203j](#), [204c](#)
LPOOL: [128a](#), [129c](#), [141c](#), [141d](#), [144e](#)
lput(): [46b](#), [110b](#), [148a](#), [174c](#), [177f](#)
lputl(): [108c](#), [109d](#)
LTO: [133e](#), [134d](#), [138b](#), [141c](#), [141d](#), [141e](#), [181a](#), [181e](#), [195a](#), [210a](#)
m: [263a](#), [263b](#)
m1: [262e](#), [263a](#)
m2: [262f](#), [263a](#)
m3: [262g](#), [263a](#)
m4: [262h](#), [263a](#)
m5: [262i](#), [263a](#)
m6: [262j](#), [263a](#)
m7: [262k](#), [263a](#)
m8: [262l](#), [263a](#)
m9: [262m](#), [263a](#)
main-20(): [246j](#)
malloc(): [32d](#), [43a](#), [55](#), [58a](#), [69g](#), [72](#), [74f](#), [77b](#), [151b](#), [155a](#), [157h](#), [172b](#), [179d](#), [189b](#), [233a](#), [266b](#), [267b](#)
man: [245b](#), [245b](#), [246j](#), [247a](#), [256c](#)
Mark: [35g](#)
match(): [247b](#), [249](#), [250a](#), [250b](#), [251](#), [252a](#), [259b](#)
MAXHIST: [157a](#), [157c](#), [203i](#)
MAXIO: [59d](#)
maxlibdir-1: [73e](#), [73e](#), [74f](#)
mcmd(): [246j](#), [251](#)
mesg(): [247b](#), [249](#), [250a](#), [251](#), [252b](#), [260b](#)
MINLC-12: [160b](#), [160c](#), [227f](#)
MINSIZ: [157b](#), [183c](#)
mkfwd(): [85a](#), [85c](#)
modemap: [178b](#), [179c](#)
movtemp: [245e](#), [245e](#), [246j](#), [251](#)
multifile-37: [238e](#), [239e](#), [243a](#)
mylog(): [225b](#)
Nconv(): [218e](#), [222](#)
nerrors: [229a](#), [229a](#), [229b](#), [229d](#)
newdata(): [172b](#), [174a](#)
newmember(): [247b](#), [252b](#), [256d](#), [263e](#)
newtempfile(): [247b](#), [249](#), [251](#), [263d](#)
nexports: [171e](#), [172a](#), [172b](#)
nflag-41: [238i](#), [240b](#)

NHASH: [31a](#), [32c](#), [52d](#), [92c](#), [148b](#), [172b](#), [175b](#), [177f](#), [214e](#)
 NHASH-16: [244e](#), [246c](#), [254b](#)
 NHUNK: [232e](#)
 nhunk: [232b](#), [232e](#), [233a](#)
 nimports: [171d](#), [172a](#), [175b](#)
 nlibdir: [73d](#), [73d](#), [74f](#), [75b](#)
 nocache(): [107c](#)
 noname: [37b](#), [37b](#)
 noops(): [87](#)
 nopout(): [187d](#), [187f](#)
 nsym-47: [239a](#), [239e](#), [240a](#), [241b](#), [242](#)
 nsymbol: [231b](#), [231c](#)
 nuxiinit(): [51e](#)
 objfile(): [44a](#), [76e](#)
 objsym(): [253](#), [254a](#)
 ocmp(): [104c](#), [105b](#)
 oflag: [245k](#), [246j](#), [250a](#)
 ofsr(): [195c](#), [195d](#), [196a](#), [196b](#), [197a](#), [198b](#), [198c](#)
 olhr(): [210c](#), [211c](#), [212a](#), [212b](#), [212c](#)
 olhrr(): [211b](#), [212c](#), [213a](#)
 olr(): [127a](#), [131a](#), [131d](#), [133a](#), [133d](#), [136b](#), [181d](#)
 olrr(): [132f](#), [133a](#), [134b](#), [137b](#), [212e](#), [213e](#)
 omvl(): [125b](#), [125c](#), [127c](#), [132f](#), [134a](#), [135f](#), [137b](#), [138c](#), [181b](#), [181d](#), [182a](#), [196b](#), [197a](#), [198b](#), [198c](#), [211a](#), [211b](#)
 opbra(): [128b](#), [128c](#)
 openar(): [247b](#), [249](#), [250a](#), [250b](#), [251](#), [252a](#), [252b](#), [255a](#)
 Operand.class: [101a](#)
 oplook(): [47b](#), [94b](#), [102c](#)
 Oprange: [100b](#), [283a](#)
 oprange: [100c](#), [102c](#), [103a](#), [104c](#), [118c](#), [118d](#), [122b](#), [124b](#), [127e](#), [134e](#), [194d](#), [205c](#), [206a](#)
 Oprange.start: [100b](#)
 Oprange.stop: [100b](#)
 Oprange (typedef): [283a](#)
 oprrr(): [119b](#), [120a](#), [121b](#), [122a](#), [122d](#), [123c](#), [123e](#), [124a](#), [124d](#), [125b](#), [125c](#), [129d](#), [130a](#), [135d](#), [135f](#), [136b](#),
[137b](#), [138a](#), [138c](#), [139c](#), [181d](#), [182a](#), [196b](#), [197a](#), [197c](#), [198a](#), [206f](#)
 opt: [245c](#), [245c](#), [256c](#)
 Optab: [99a](#), [283a](#)
 optab: [99b](#), [103b](#), [104c](#), [107a](#), [107b](#), [191h](#), [208f](#)
 Optab.a1: [99a](#)
 Optab.a2: [99a](#)
 Optab.a3: [99a](#)
 Optab.as: [99a](#)
 Optab.flag: [126d](#)
 Optab.param: [131b](#)
 Optab.size: [99a](#)
 Optab.type: [99a](#)
 Optab (typedef): [283a](#)
 Optab.flag: [141c](#)
 opvfprrr(): [199c](#), [200a](#), [200b](#), [200c](#)
 oshr(): [210b](#), [212a](#)

oshrr(): [211a](#), [212b](#)
osr(): [133c](#), [133d](#), [138a](#), [181b](#), [182a](#)
osrr(): [134a](#), [134b](#), [138c](#), [213b](#)
outfile: [38c](#), [229b](#)
ovfpmem(): [195d](#), [199a](#)
P: [36c](#), [36e](#), [36f](#), [47b](#), [48c](#), [49b](#), [55](#), [67b](#), [82](#), [84](#), [85c](#), [86a](#), [86b](#), [87](#), [88b](#), [91](#), [92c](#), [94b](#), [128b](#), [141f](#), [143a](#), [143d](#),
[144b](#), [148b](#), [151d](#), [151e](#), [152a](#), [152b](#), [152c](#), [160c](#), [163](#), [166a](#), [169a](#), [183e](#), [183f](#), [184a](#), [184c](#), [185a](#), [185b](#), [193b](#),
[204c](#), [205a](#), [213f](#), [214c](#), [220b](#), [229d](#)
page(): [265b](#), [265c](#)
patch(): [82](#)
pc: [47b](#), [48b](#), [53a](#), [53a](#), [55](#), [56b](#), [64d](#), [65b](#), [67c](#), [214g](#)
pcmd(): [246j](#), [250b](#)
Pconv(): [218e](#), [219d](#)
phaseerr(): [256b](#), [256d](#)
pmode(): [262d](#), [263b](#)
poname: [246e](#), [246j](#), [247b](#), [251](#)
pool-13: [143b](#), [144a](#), [144b](#), [144c](#), [144d](#), [145b](#)
prasm(): [48b](#), [103b](#), [108b](#), [120a](#), [126a](#), [128c](#), [200c](#), [219b](#)
prg(): [43a](#), [88c](#), [141f](#), [144d](#), [163](#), [166d](#), [167](#), [168a](#), [174a](#), [184a](#), [184c](#), [186b](#), [191k](#), [193b](#), [201](#)
printsyms(): [239e](#), [240a](#), [241b](#), [243a](#)
Prog: [279](#), [279](#)
Prog (typedef): [279](#)
prog_divu: [201](#), [203b](#), [204c](#)
prog_div: [201](#), [203a](#), [204c](#), [205a](#)
prog_modu: [201](#), [203d](#), [204c](#)
prog_mod: [201](#), [203c](#), [204c](#)
psym(): [239e](#), [240a](#), [241b](#), [242](#)
putsymb(): [148a](#), [148b](#), [149](#), [152e](#), [159a](#), [159c](#), [215b](#)
qcmd(): [246j](#), [252b](#)
rcmd(): [246j](#), [247b](#)
rderr(): [256a](#), [264a](#), [264c](#)
readsome(): [59c](#), [59d](#), [63b](#)
readundefs(): [172a](#)
regoff(): [195b](#), [195c](#), [196a](#)
RELD-5: [179b](#), [179c](#)
relinv(): [185a](#), [186a](#)
Reloc: [177c](#), [287b](#)
Reloc.a: [177c](#)
Reloc.m: [177c](#)
Reloc.n: [177c](#)
Reloc.t: [177c](#)
Reloc (typedef): [287b](#)
rels: [177d](#), [177f](#), [179c](#)
RELU-6: [179b](#), [179c](#)
RGRP-26: [261g](#), [262h](#)
Rindex: [176b](#), [182e](#)
rl(): [258a](#), [258b](#)
rnd(): [66a](#), [92c](#), [94b](#), [214e](#), [236a](#)
Roffset: [176a](#), [176b](#), [176d](#)

ROTH-29: [261j](#), [262k](#)
ROWN-23: [261d](#), [262e](#)
rxxx: [278c](#)
S: [32a](#), [33b](#), [52d](#), [65c](#), [66d](#), [67a](#), [76h](#), [83b](#), [92c](#), [94b](#), [117c](#), [148b](#), [151b](#), [172b](#), [175b](#), [177f](#), [179c](#), [213f](#), [214e](#), [220b](#),
[222](#), [229d](#)
SBSS: [52c](#), [66c](#), [92c](#), [95d](#), [117b](#), [148b](#), [163](#), [170b](#), [180b](#), [191k](#)
scanobj(): [247b](#), [249](#), [251](#), [253](#)
Sconv(): [218e](#), [224](#)
SDATA: [52c](#), [92c](#), [93b](#), [95d](#), [116b](#), [117b](#), [148b](#), [172b](#), [177b](#), [180b](#), [182d](#), [183d](#)
SDATA1: [117b](#), [183c](#), [183d](#), [279](#)
sdiv(): [204b](#), [204c](#)
Section: [33b](#)
select(): [263b](#), [263c](#)
setcom(): [246j](#), [247a](#)
setmalloctag(): [233c](#)
SEXPORT: [172b](#), [214b](#)
SFILE: [156c](#), [159a](#)
sflag-42: [238j](#), [243a](#)
SGID-22: [261c](#), [262j](#)
sigdiv(): [203j](#), [204a](#)
SIGNINTERN: [203j](#), [204b](#), [279](#)
SIMPORT: [172a](#), [175b](#), [204b](#)
size(): [237a](#)
skip(): [247b](#), [249](#), [250a](#), [250b](#), [251](#), [252a](#), [257c](#)
SNONE: [32d](#), [33b](#), [47a](#), [61d](#), [65c](#), [66d](#), [72](#), [97a](#), [116b](#), [117b](#), [203j](#), [204b](#)
sput(): [177f](#), [178a](#)
SSTRING: [35g](#), [52c](#), [117b](#), [180b](#), [214c](#), [214e](#), [215a](#), [215b](#)
STEXT: [47a](#), [52c](#), [65b](#), [83b](#), [83c](#), [97b](#), [117b](#), [148b](#), [149](#), [156d](#), [166a](#), [180b](#), [203j](#), [204b](#)
strcond: [223a](#), [223b](#)
STRINGSZ: [219d](#), [220b](#), [222](#), [224](#), [229d](#), [232d](#)
strip(): [267b](#)
strnput(): [235a](#)
STXT-32: [262c](#), [262m](#)
SUID-21: [261b](#), [262g](#)
SUNDEF: [33b](#), [117b](#), [172b](#), [175g](#), [176a](#), [176b](#), [176c](#), [177f](#), [179c](#), [180b](#), [180e](#), [203j](#)
SXREF: [52d](#), [61d](#), [65c](#), [66d](#), [72](#), [76h](#), [97a](#), [116b](#), [117b](#), [172a](#), [172b](#), [175b](#), [203j](#), [204b](#), [204c](#)
Sym: [279](#), [279](#)
Sym (typedef): [279](#)
symdef: [245g](#), [245g](#), [247b](#), [249](#), [251](#), [258b](#)
symdefsize: [246a](#), [254a](#), [258b](#)
symname: [71b](#), [71b](#), [73a](#)
symname-36: [238d](#), [238d](#), [239e](#)
symptr-46: [238n](#), [239e](#), [240a](#), [241b](#), [242](#)
symsize: [46b](#), [147a](#), [147c](#), [148a](#), [148b](#)
sym_div-8: [201](#), [203e](#), [204c](#)
sym_divu-9: [201](#), [203f](#), [204c](#)
sym_mod-10: [201](#), [203g](#), [204c](#)
sym_modu-11: [201](#), [203h](#), [204c](#)
tailtemp: [245f](#), [245f](#), [246j](#), [247b](#), [251](#)

tcmd(): [246j](#), [252a](#)
textp: [148b](#), [152a](#), [152a](#), [152c](#), [184a](#)
textsize: [46b](#), [48d](#), [94a](#), [94b](#), [97c](#), [147b](#), [176f](#), [177a](#), [188a](#), [214g](#)
Tflag-44: [238l](#), [243a](#)
thechar: [38a](#), [77b](#)
thestring: [38b](#), [77b](#)
thunk: [232c](#), [232e](#)
TNAME: [95c](#), [116b](#), [117b](#), [279](#)
trim(): [246j](#), [252a](#), [257a](#), [259b](#), [261a](#)
uflag: [245l](#), [246j](#), [247b](#)
uflag-43: [238k](#), [242](#)
undef(): [52d](#)
undefp: [175e](#)
undefsym(): [175b](#), [176b](#)
UP: [82](#), [175g](#), [176a](#), [183e](#), [204c](#), [279](#)
usage(): [39a](#)
V4: [208e](#), [208f](#), [208g](#), [210a](#)
version: [64a](#), [64a](#), [64b](#), [64c](#)
vflag: [245m](#), [246j](#), [250b](#), [252a](#), [260b](#)
VFP: [191d](#), [191e](#), [191h](#), [199b](#)
vfp: [191f](#), [191g](#), [191h](#), [191k](#), [193a](#), [195d](#)
WGRP-27: [261h](#), [262i](#)
WOTH-30: [262a](#), [262l](#)
WOWN-24: [261e](#), [262f](#)
wput(): [177f](#), [234e](#)
wputl(): [234d](#)
wrerr(): [250a](#), [250b](#), [252b](#), [255b](#), [255c](#), [258b](#), [259a](#), [264c](#)
wrsym(): [258b](#), [259a](#)
xcmd(): [246j](#), [250a](#)
xcmp: [107d](#), [107e](#), [108a](#)
xdefine(): [95d](#), [97a](#), [97b](#), [182d](#)
xfol(): [184a](#), [184c](#), [185a](#), [186b](#)
XGRP-28: [261i](#), [262j](#)
XOTH-31: [262b](#), [262m](#)
XOWN-25: [261f](#), [262g](#)
xrefresolv: [76f](#), [76g](#), [76h](#), [77a](#)
zenter(): [241a](#), [242](#)
zerosig(): [171c](#)
zprg: [43a](#), [43b](#), [89b](#), [141f](#), [167](#), [168a](#)
__anon_enum_1: [179b](#)
__anon_enum_5: [238a](#)
__anon_struct_1.ieee: [33c](#)
__anon_struct_1.offset: [33c](#)
__anon_struct_1.sval: [33c](#)
__anon_struct_1: [33c](#)
__anon_struct_2.size: [144a](#)
__anon_struct_2.start: [144a](#)
__anon_struct_2: [144a](#), [144a](#)

Bibliography

- [BO10] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Prentice Hall, 2010. cited page(s) 11
- [Knu92] Donald E. Knuth. *Literate Programming*. Center for the Study of Language and Information, 1992. cited page(s) 11
- [KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988. cited page(s) 11
- [Lev99] John R. Levine. *Linkers and Loaders*. Morgan-Kaufman, 1999. Available at <http://www.iecc.com/linker/>. cited page(s) 11, 16
- [Pad09] Yoann Padioleau. Syncweb, literate programming meets unison, 2009. <https://github.com/aryx/syncweb>. cited page(s) 11
- [Pad14] Yoann Padioleau. *Principia Softwarica: The Plan 9 Kernel 9*. 2014. cited page(s) 13, 19, 41, 97, 217
- [Pad15a] Yoann Padioleau. *Principia Softwarica: The ARM Assembler 5a*. 2015. cited page(s) 11, 15, 19, 21, 24, 26, 28, 30, 33, 34, 35, 53, 56, 57, 61, 65, 66, 79, 81, 88, 89, 90, 100, 112, 118, 119, 120, 121, 122, 125, 132, 146, 153, 168, 216
- [Pad15b] Yoann Padioleau. *Principia Softwarica: The ARM Emulator 5i*. 2015. cited page(s) 11, 35, 119, 120, 128, 130
- [Pad16a] Yoann Padioleau. *Principia Softwarica: The Plan 9 C Compiler 5c*. 2016. cited page(s) 69, 91, 216
- [Pad16b] Yoann Padioleau. *Principia Softwarica: The Plan 9 Core Libraries*. 2016. cited page(s) 97, 218
- [Pad16c] Yoann Padioleau. *Principia Softwarica: The Plan 9 Debuggers and Tracers*. 2016. cited page(s) 25, 146, 152, 217
- [Pad26] Yoann Padioleau. *Principia Softwarica: The The Plan 9 Profilers*. 2026. cited page(s) 162, 165, 217
- [Sea01] David Seal. *ARM, Architecture Reference Manual*. Addison-Wesley, 2001. cited page(s) 9
- [Tay08] Ian Lance Taylor. A new elf linker. In *Proceedings of the GCC Developers' Summit*, 2008. Available at <http://ols.fedoraproject.org/GCC/Reprints-2008/taylor-reprint.pdf>. cited page(s) 9