Principia Softwarica: The ARM Linker `51`

version 0.1

Yoann Padioleau
yoann.padioleau@gmail.com

with code from
Rob Pike

March 31, 2026

Contents

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: `5l`

I will explain in this book the code of the Plan 9 ARM linker `5l`[1], which is about 8150 lines of code (LOC). `5l` 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 `'l'` means linker.

[1] See http://plan9.bell-labs.com/magic/man2html/1/8l, which despite its name covers also `5l`.

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 `5l`, and not for instance the x86 variant `8l`, 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 `5l`.

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 `5l` 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 V2[2] and finished as a C program in UNIX V7[3]. `ld` contains 1300 LOC, which is smaller than `5l`. This is partly because `5l` 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` package[4], 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.

- LD86[6] 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 `5l`. But, to be fair, `5l` does also machine code generation, which is done instead by AS86 for LD86. `5a`+`5l` 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 `5l` 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/

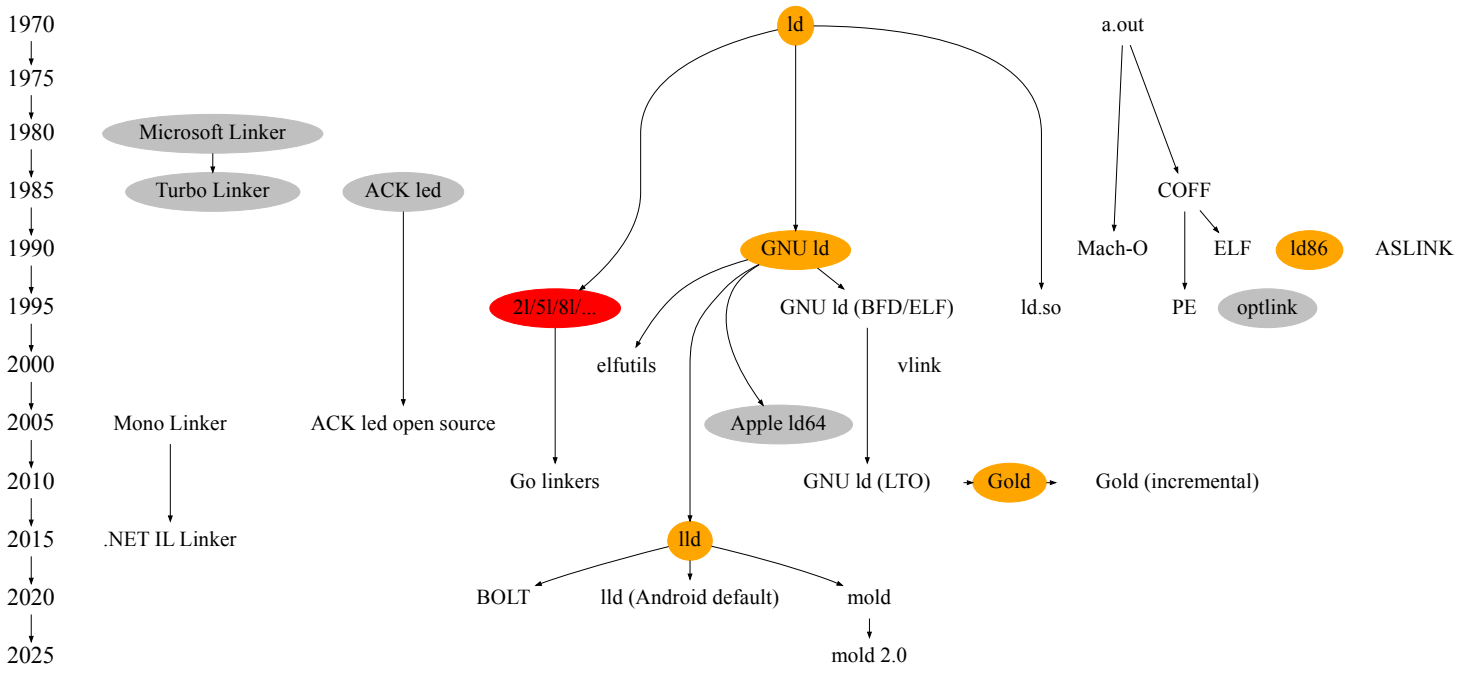


Figure 1.1: Linkers timeline

`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 language[7]. 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 Goken9cc[8], 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 Pi[9]), launchs 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 `5l`, 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 `5l`. 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.

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

1.8 Acknowledgments

I would like to acknowledge of course the author of `5l`, Rob Pike, who wrote in some sense most of this book.

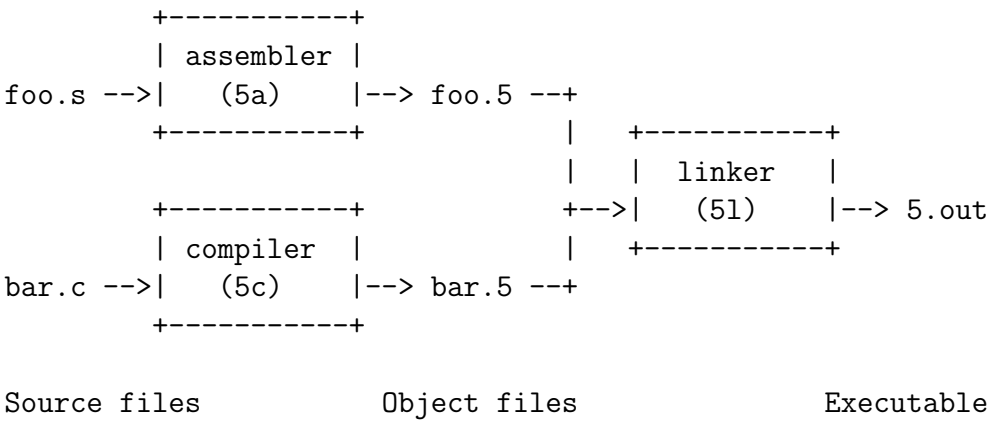


Figure 2.1: Linker inputs and output.

Chapter 2

Overview

Before showing the source code of `5l` 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 `5l`, as well as the format of the executables generated by `5l`. 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.

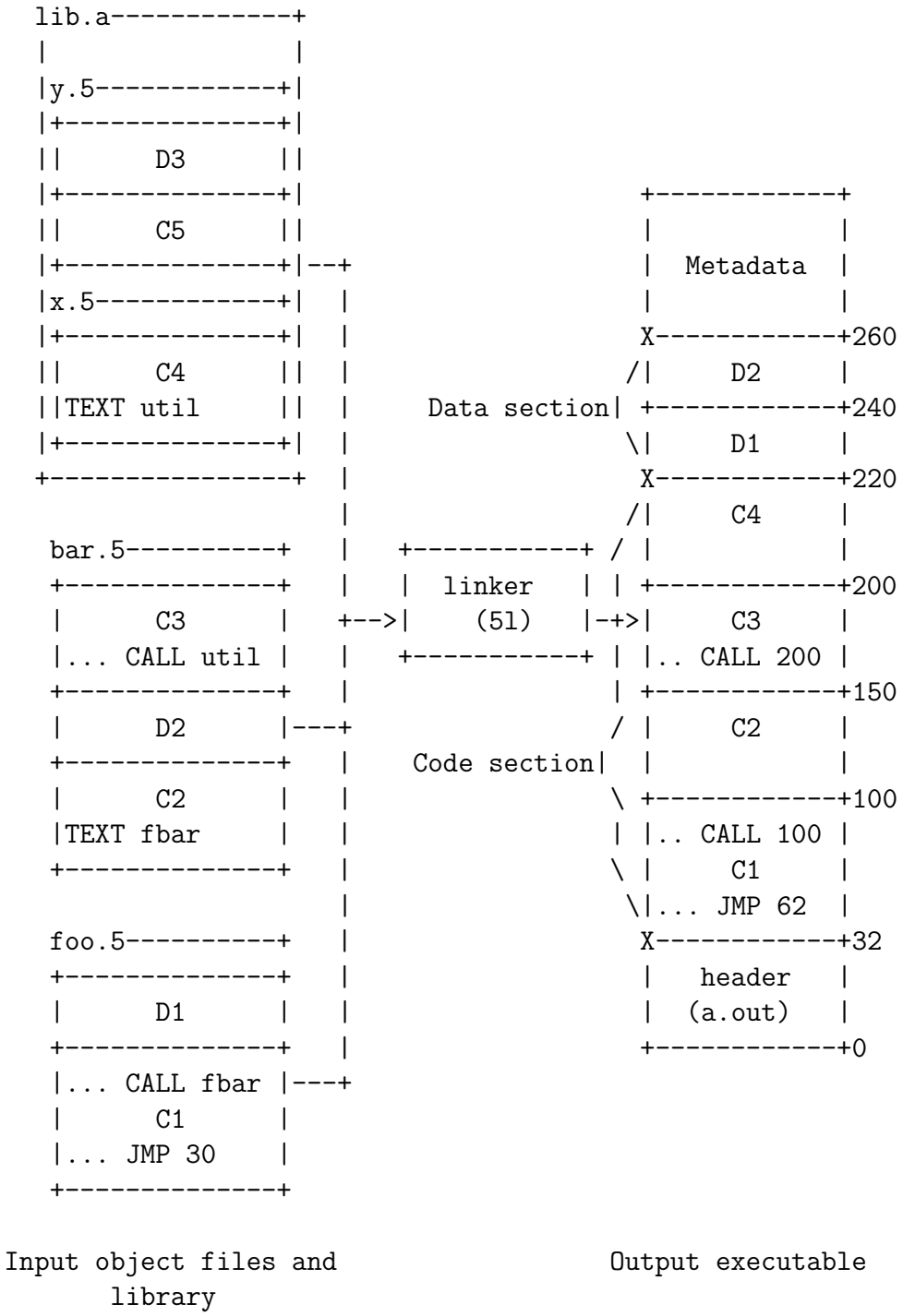


Figure 2.2: Linking process overview.

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) file[1]. 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 `5l` 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.

[1] Assuming 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.

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.

2.2 `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 `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 16a⟩≡
```
  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 16b⟩≡
```
  GLOBL   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/5l
$ 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 ,{fprint}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 {fprint}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 `fprint` 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.M ...`). 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 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 rational 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] See Section 10.3.1 for more information about line numbers.

```
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` 21⟩≡ (263i)

```
// 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 `long`s (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. 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
  ```

- `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 `5l` 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].

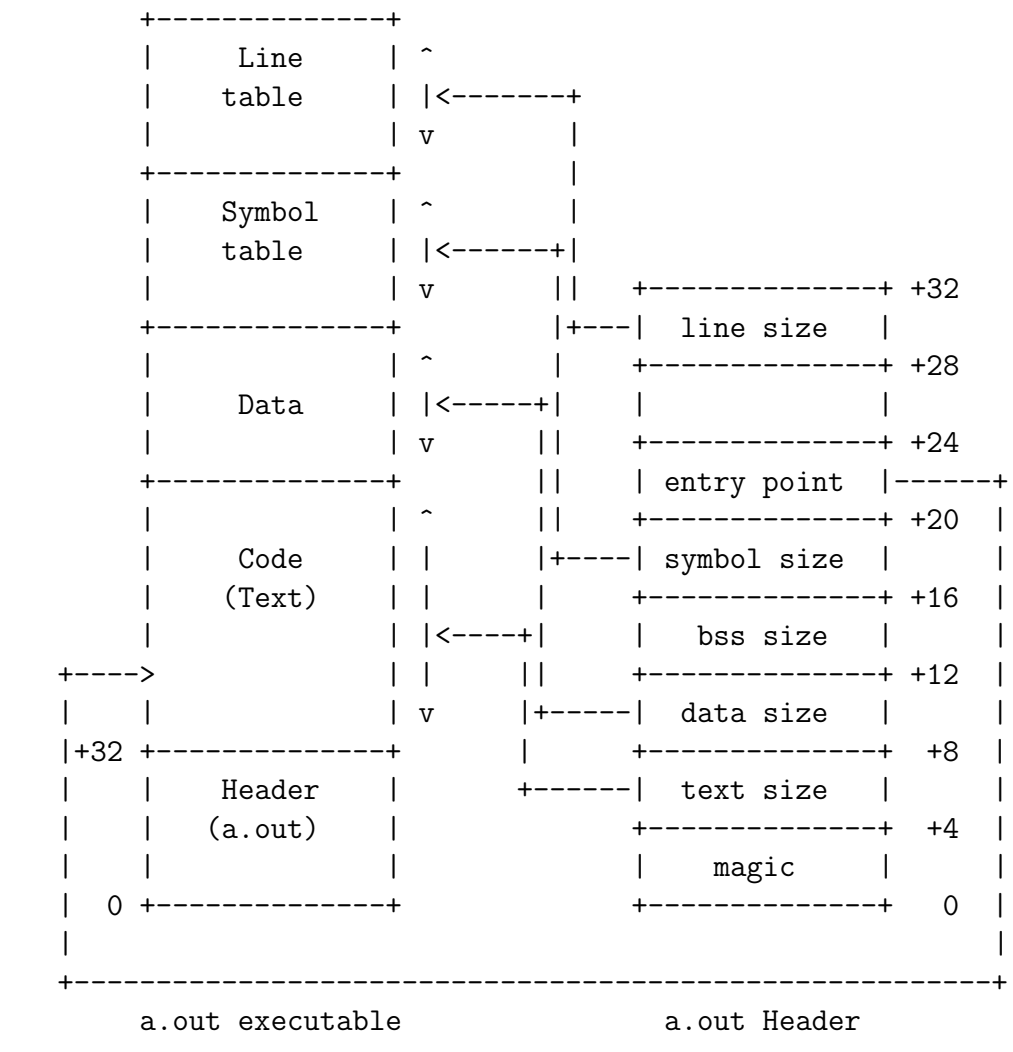


Figure 2.3: `a.out` executable format.

2.6 Code organization

Table 2.1 presents short descriptions of the source files used by `5l`, 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`[272] `Section`[29d] `Instr`[272]	471
globals	3	`globals.c`	`hash`[28a] `firstp`[32d] `datap`[33a] `pc`[49a]	168
entry point	4	`main.c`	`main()`[239j]	301
executable generation	4	`asm.c`	`asmb()`[41a]	222
data section generation	4	`datagen.c`	`datblk()`[44d]	245
loading objects	5	`obj.c`	`ldobj()`[49b]	834
loading libraries	6	`lib.c`	`loadlib()`[71e]	188
building the code instructions graph	7	`pass.c`	`patch()`[77]	372
rewriting virtual opcodes	7	`noop.c`	`noops()`[82]	415
laying out the code and data sections	7	`layout.c`	`dotext()`[89b] `dodata()`[87c]	432
ARM code-generation data structures	8	`m.h`	`Optab`[93b] `Oprange`[95a] `Operand_class`[95c]	159
ARM opcodes lookup table	8	`optab.c`	`optab`[94a]	293
ARM opcodes lookup function	8	`span.c`	`oplook()`[97a] `buildop()`[98d]	670
ARM instructions generation	9	`codegen.c`	`asmout()`[102e]	1344
debugging support	10	`debugging.c`	`asmsym()`[142] `asmlc()`[154]	244
location information	10	`hist.c`	`addhist()`[148b]	55
profiling support	11	`profile.c`	`doprof1()`[157] `doprof2()`[160a]	264
dynamic linking	12	`dynamic.c`	`asmdyn()`[171f]	412
float instructions	12	`float.c`	`ieeedtof()`[183b]	92
ELF constants	12	`elf.h`	`Ehdr32sz`	21
ELF generation	12	`elf.c`	`elf32()X`	176
dumpers	A	`fmt.c`	`prasm()`[212b] `Pconv()`[212d] `Aconv()`[213a]	355
error management	B	`error.c`	`diag()`[222d] `errorexit()`[222b]	45
input/output buffer	D	`io.c`	`buf`[227b] `wput()`[227e]	138
utilities	D	`utils.c`	`malloc()`[226a]	226
Total				8142

Table 2.1: Chapters and source files of `5l`.

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 `5l`. 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 `5l`, whereas Figure 2.5 describes the main data flow of `5l`. The main steps of the linking process of `5l` are as follows:

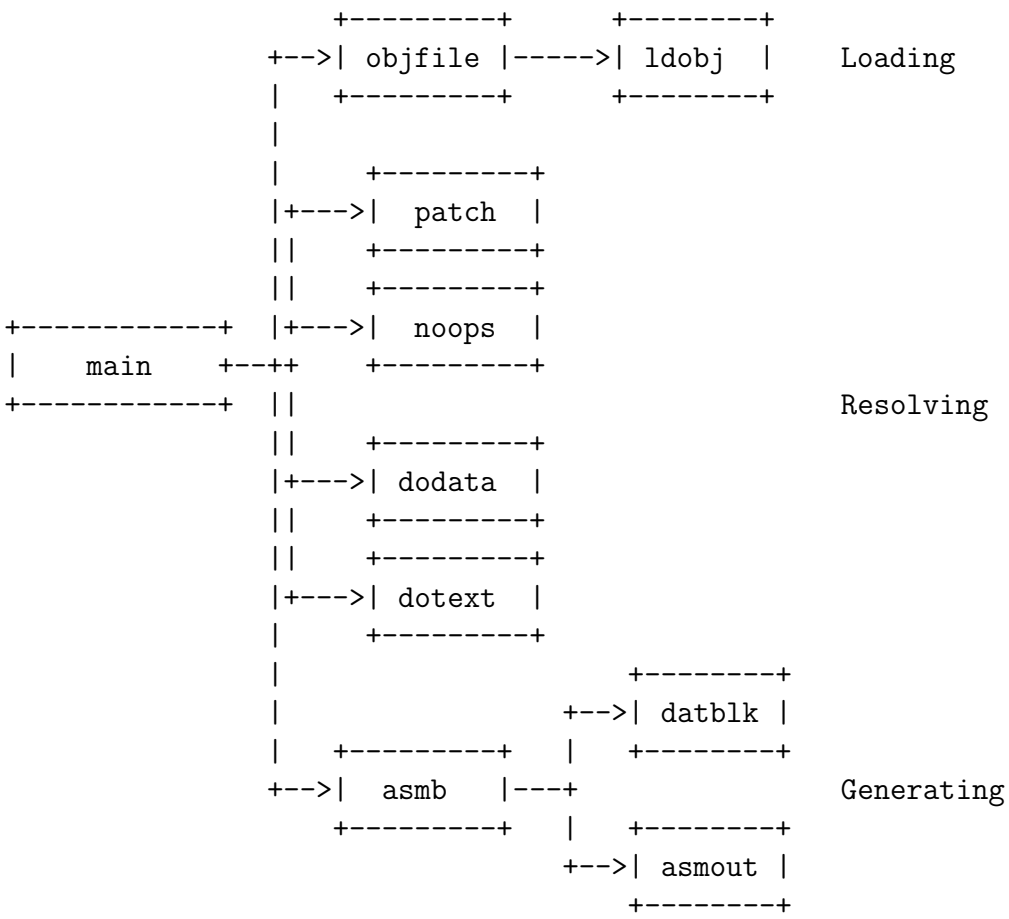


Figure 2.4: Control flow diagram of `5l`.

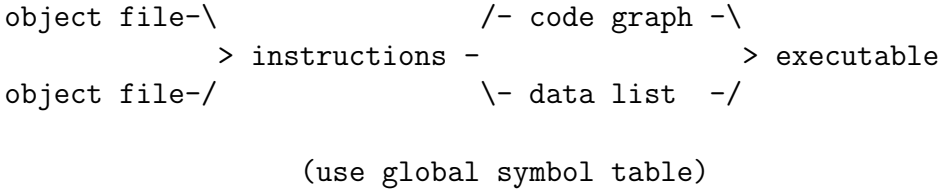


Figure 2.5: Data flow diagram of `5l`.

1. *Load* (read) the object files and libraries (via `objfile()`[40a]) and concatenate their code and data in different sections in memory (via `ldobj()`[49b])
2. *Resolve* symbol references (via `patch()`[77], ...)
3. *Generate* the executable with its header and machine code (via `asmb()`[41a])

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()`[239j] 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`[32d] and `datap`[33a] 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`[49a] 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`[28a], 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()`[82], 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 at 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 `5l` has the graph of machine code instructions and the set of globals declarations, it can start to *lay out* the code and data. `dodata()`[87c] 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()`[89b] 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`[89a] and `datsize`[87a]. `5l` can finally call `asmb()` to generate the executable and its header. This function uses two helpers functions: one to fill the data section `datblk()`[44d] and one to generate ARM instructions in the code section `asmout()`[102e].

2.8 Book structure

You now have enough background to understand the source code of `5l`. The rest of the book is organized as follows. I will start by describing the core data structures of `5l` in Chapter 3. Then, I will use a top-down approach in Chapter 4, and, starting from `main()`[239j], I will present the code, or a high-level view of the code, of some of the main functions (e.g., `asmb()`[41a]). 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 `5l`, 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 `5l`. 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 `5l` itself in Appendix A, the code to manage errors in Appendix B, and the code which profiles `5l` itself in Appendix C. Appendix D contains the code of generic utility functions used by `5l` but which are not specific to `5l`. 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` 27⟩≡ (272)

```
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 29c⟩

    ⟨Sym other fields 63c⟩
    // Extra
    ⟨Sym extra fields 28c⟩
};
```

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:

⟨*global hash linker* 28a⟩≡ (276b)
```
// hash<Sym.name * Sym.version, ref_own<Sym>> (next = Sym.link in bucket)
Sym* hash[NHASH];
```
Uses `NHASH`.

⟨*constant NHASH linker* 28b⟩≡ (271c)
```
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* 28c⟩≡ (27)
```
// list<ref<Sym>> (from = hash)
Sym*    link;
```

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

⟨*constant* `S` 28d⟩≡ (272)
```
#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`[272] in the symbol table `hash` associated with this key, or a new symbol if the key was not found:

⟨*function* `lookup` 28e⟩≡ (277b)
```
Sym*
lookup(char *symb, int v)
{
    Sym *sym;
    long h;
    int len;
    ⟨lookup() other locals 28f⟩

    ⟨lookup() compute hash value h of (symb, v) and len 29a⟩

    // 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 29b⟩
}
```
Uses `NHASH` and `S` 29d.

⟨`lookup()` *other locals* 28f⟩≡ (28e)
```
char *p;
int c;
```

⟨lookup() *compute hash value* h *of* (symb, v) *and* len 29a⟩≡ (28e)
```
// 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 &= 0xffffff;
h %= NHASH;
```

⟨lookup() *if symbol name not found* 29b⟩≡ (28e)
```
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 224c⟩
return sym;
```
Uses SNONE 29d, malloc() 226a, and nsymbol 224b.

3.2 Section

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

⟨Sym *section field* 29c⟩≡ (27)
```
//enum<Section>
short   type;
```

⟨*enum* Section*(arm)* 29d⟩≡ (272)
```
enum Section
{
    SNONE = 0,

    STEXT,
    SDATA,
    SBSS,

    SXREF,
    ⟨Section cases 150b⟩
};
```

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 5l 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()[49b], 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()[48b] 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 `5l`, 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()`[213a] for the dumper of `Opcode` (`A` because opcodes use the `A`*xxx* syntax, e.g., `ASUB`),
- see `Dconv()`[213b] for the dumper of `Operand_kind` (`D` because operand kinds use the `D`*xxx* 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 `5l`:

⟨*struct* `Adr` *(arm)* 30a⟩≡ (272)
```
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 30b⟩
};
```

`Operand`[272] 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* 30b⟩≡ (30a) 101b▷
```
// 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)* 31a⟩≡ (272)
```
struct  Prog
{
    //enum<Opcode>
    byte    as;

    // operands
    Adr from;
    Adr to;

    ⟨Prog other fields 31b⟩

    // Extra
    ⟨Prog extra fields 32a⟩
};
```

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 31b⟩≡ (31a) 31c▷
```
// 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 31c⟩+≡ (31a) ◁31b 31d▷
```
// enum<Instr_cond>
byte    scond;
```

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

⟨Prog other fields 31d⟩+≡ (31a) ◁31c 31e▷
```
long    line;
```

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

Up until now, `Instr`[272] is very similar to the type used in the Assembler book [Pad15a] to represent an instruction[1]. 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 31e⟩+≡ (31a) ◁31d 32b▷
```
// virtual program counter, and then real program counter
long    pc;
```

[1] `5a` 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`.

The virtual program counter is incremented after each code instruction is read in `ldobj()`[49b]. 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* 32a⟩≡ (31a) 32e▷
```
// 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* 32b⟩+≡ (31a) ◁31e 78d▷
```
//bitset<enum<Mark>>
short   mark;
```

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

⟨*enum* `Mark` *(arm)* 32c⟩≡ (272)
```
/* mark flags */
enum Mark {
    ⟨Mark cases 83a⟩
};
```

3.5 List of instructions

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

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` 32d⟩≡ (276b)
```
// list<ref_own<Prog>> (next = Prog.link)
Prog* firstp;
```

The instructions are linked together with the following field:

⟨`Prog` *extra fields* 32e⟩+≡ (31a) ◁32a
```
// list<ref<Prog>> (from = firstp or datap)
Prog*   link;
```

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

⟨*constant* `P` 32f⟩≡ (272)
```
#define P       ((Prog*)nil)
```

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

⟨*global* `lastp` 32g⟩≡ (276b)
```
// 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` 33a⟩≡ (276b)
```
// list<ref_own<Prog>> (next = Prog.link)
Prog* datap = P;
```
Uses `P` 32f and `datap` 33a.

`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 `firstp`[32d] 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` 33b⟩≡ (276b)
```
//option<ref<Prog>> where Prog.as == ATEXT
Prog* curtext = P;
```
Uses `P` 32f and `curtext` 33b.

⟨*global* `curp` 33c⟩≡ (276b)
```
// option<ref<Prog>>
Prog* curp;
```

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

⟨*adjust curtext when iterate over instructions p* 33d⟩≡ (178c 177d 154 89b 82 80c 77 42c)
```
if(p->as == ATEXT)
    curtext = p;
```
Uses `P` 32f and `firstp` 32d.

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)* 33e⟩≡ (272)
```
#define TNAME (curtext && curtext->from.sym ? curtext->from.sym->name : noname)
```

⟨*global noname linker* 33f⟩≡ (283a)
```
char *noname  = "<none>";
```

Chapter 4

Main Functions

I now switch to a top-down approach where I describe the main functions of `5l`, starting from `main()`[239j] down to `asmb()`[41a] which generates the executable.

4.1 `main()`

Before showing the code of `main()`[239j] 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` 34a⟩≡ (276b)
```
char thechar;
```

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

⟨*global* `thestring` 34b⟩≡ (276b)
```
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` 34c⟩≡ (276b)
```
char* outfile;
```

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

⟨*global* `cout` 34d⟩≡ (276b)
```
fdt cout = -1;
```
Uses `cout` 34d.

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)* 34e⟩≡ (285c)
```
void
main(int argc, char *argv[])
{
    ⟨main() locals(arm) 69a⟩

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

    outfile = "5.out";
```

```
    ⟨main() debug initialization(arm) 211d⟩

    ARGBEGIN {
    ⟨main() command line processing(arm) 35d⟩
    } ARGEND

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

    ⟨main() initialize globals(arm) 35c⟩

    cout = create(outfile, 1, 0775);
    ⟨main() sanity check cout 35b⟩

    // ------ main functions  ------
    ⟨main() cout is ready, LET'S GO(arm) 38d⟩

out:
    ⟨main() profile report 224a⟩
    errorexit();
}
```

⟨*function usage, linker* 35a⟩≡ (285c)

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

Uses `errorexit()` 222b.

⟨`main()` *sanity check cout* 35b⟩≡ (34e)

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

The error management functions `diag()`[222d] and `errorexit()`[222b] are described in Appendix B.

⟨`main()` *initialize globals(arm)* 35c⟩≡ (34e) 39c ▷

```
⟨main() addlibpath("/thestring/lib") or ccroot 69d⟩
⟨main() set HEADTYPE, INITTEXT, INITDAT, etc 37a⟩
⟨main() set INITENTRY 38c⟩
```

4.1.1 Arguments processing

I have mentioned before the `-o` option:

⟨`main()` *command line processing(arm)* 35d⟩≡ (34e) 36b ▷

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

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` 36a⟩≡ (276b)
```
// option<enum<Headtype>>, None = -1
short HEADTYPE = -1; /* type of header */
```
Uses `HEADTYPE` 36a.

⟨`main()` *command line processing(arm)* 36b⟩+≡ (34e) ◁35d 37c▷
```
case 'H':
    a = ARGF();
    if(a)
        HEADTYPE = atolwhex(a);
    break;
```

The `atolwhex()`[228c] 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)* 36c⟩≡ (272)
```
/*
 *  -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` 36d⟩≡ (276b)
```
long HEADR; /* length of header */
```

⟨*global* `INITTEXT` 36e⟩≡ (276b)
```
long INITTEXT = -1; /* text location */
```
Uses `INITTEXT` 36e.

⟨*global* `INITRND` 36f⟩≡ (276b)
```
long INITRND = -1; /* data round above text location */
```
Uses `INITRND` 36f.

⟨*global* `INITDAT` 36g⟩≡ (276b)
```
long INITDAT = -1; /* data location */
```
Uses `INITDAT` 36g.

⟨main() *set HEADTYPE, INITTEXT, INITDAT, etc* 37a⟩≡ (35c)
```
if(HEADTYPE == -1)
    HEADTYPE = H_PLAN9;
switch(HEADTYPE) {
⟨main() switch HEADTYPE cases(arm) 37b⟩
default:
    diag("unknown -H option");
    errorexit();
}
⟨main() sanity check INITXXX 37e⟩
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)* 37b⟩≡ (37a) 181g▷
```
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()`[89b] to contain the address of the next memory page [1] 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)* 37c⟩+≡ (34e) ◁36b 37d▷
```
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)* 37d⟩+≡ (34e) ◁37c 38b▷
```
case 'R':
    a = ARGF();
    if(a)
        INITRND = atolwhex(a);
    break;
```

⟨main() *sanity check INITXXX* 37e⟩≡ (37a)
```
if(INITDAT != 0 && INITRND != 0)
    print("warning: -D0x%lux is ignored because of -R0x%lux\n",
        INITDAT, INITRND);
```

[1] The 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` 38a⟩≡ (276b)
```
char* INITENTRY = nil;  /* entry point */
```
Uses `INITENTRY` 38a.

⟨`main()` *command line processing(arm)* 38b⟩+≡ (34e) ◁37d 68f▷
```
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* 38c⟩≡ (35c)
```
if(INITENTRY == nil) {
    INITENTRY = "_main";
    ⟨main() adjust INITENTRY if profiling 156a⟩
}
⟨main() if rare condition do not set SXREF for INITENTRY, else 70e⟩
  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 up[2]. This will create a new symbol in the symbol table `hash`[28a] 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)* 38d⟩≡ (34e)
```
// first empty instruction
firstp = prg();
lastp = firstp;

// Loading (populates firstp, datap, and hash)
while(*argv)
    objfile(*argv++);
⟨main() load implicit libraries 70d⟩

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

// Resolving
⟨main() resolving phase 74⟩
```

[2]Remember 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` 39a⟩≡ (277b)
```
Prog*
prg(void)
{
    Prog *p;

    p = malloc(sizeof(Prog));
    *p = zprg;
    return p;
}
```
Uses `malloc()` 226a and `zprg` 39b.

⟨*global* `zprg` 39b⟩≡ (276b)
```
Prog zprg;
```

⟨`main()` *initialize globals(arm)* 39c⟩+≡ (34e) ◁35c 46e▷
⟨`main()` *set zprg(arm)* 39d⟩

⟨`main()` *set zprg(arm)* 39d⟩≡ (39c)
```
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 `5l` (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`[65a]). 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` 40a⟩≡ (283a)

```
/// main | loadlib  -> <>
void
objfile(char *file)
{
    fdt f;
    long len;
    char magbuf[SARMAG]; // magic buffer
    ⟨objfile() other locals 66c⟩

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

    ⟨objfile() adjust file if -lxxx filename 70a⟩

    f = open(file, 0);
    ⟨objfile() sanity check f 40b⟩

    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, 0L, SEEK__END); // len = filesize(f);
        seek(f, 0L, SEEK__START);

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

        close(f);
        return;
    }
    // else
    ⟨objfile() when file is a library 67⟩
}
```

Uses `errorexit()` 222b and `ldobj()` 49b.

`SARMAG`[65b] above stands for `S`ize of `AR`chive `MAG`ic string. The most important part in the code above is the call to `ldobj()`[49b] which loads in memory one (opened) object file. Chapter 5 will describe `ldobj()`.

⟨`objfile()` *sanity check f* 40b⟩≡ (40a)

```
if(f < 0) {
    diag("cannot open %s: %r", file);
    errorexit();
}
```

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, `5l` can finally generate the executable with `asmb()`:

⟨*function* `asmb` *(arm)* 41a⟩≡ (282b)
```
/// main -> <>
void
asmb(void)
{
    ⟨asmb() locals 41b⟩

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

    // Text section
    ⟨asmb() Text section 42c⟩

    // Data section
    ⟨asmb() Data section 44a⟩

    // Symbol and Line table sections
    ⟨asmb() symbol and line table sections 140⟩

    // Header
    ⟨asmb() header section 41c⟩

    cflush();
}
```
Uses DBG 272.

`asmb()`[41a] does not take any argument. It uses the global `cout`[34d], which was initialized in `main()`[239j] 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 `5l` 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* 41b⟩≡ (41a) 43a▷
```
long OFFSET;
```

4.4.1 Header

The header generation (done last) assumes the set of globals containing the size of sections (`textsize`[89a], `datsize`[87a], etc) have been computed previously:

⟨`asmb()` *header section* 41c⟩≡ (41a)
```
DBG("%5.2f header\n", cputime());

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

switch(HEADTYPE) {
⟨asmb() switch HEADTYPE (for header generation) cases(arm) 42a⟩
}
```
Uses DBG 272 and cout 34d.

⟨asmb() *switch HEADTYPE (for header generation) cases(arm)* 42a⟩≡ (41c) 182c ▷
```
// see Exec in a.out.h
case H_PLAN9:
    ⟨asmb() if dynamic module magic header adjustment(arm) 168c⟩
    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 HEADTYPE 36a, bsssize 87b, datsize 87a, entryvalue() 42b, lput() 104f, symsize 141b, and textsize 89a.

If you do not understand the order of the `lput()`[104f] 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`[38a] symbol:

⟨*function* entryvalue*(arm)* 42b⟩≡ (282b)
```
long
entryvalue(void)
{
    char *a;
    Sym *s;

    a = INITENTRY; // usually "_main"
    ⟨entryvalue() if digit INITENTRY 207d⟩

    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 171b⟩
    default:
        diag("entry not TEXT: %s", s->name);
        return 0;
    }
}
```
Uses INITDAT 36g, INITTEXT 36e, SNONE 29d, atolwhex() 228c, diag() 222d, and lookup() 28e.

4.4.2 Text section

The text section generation is of course more complicated. `asmb()`[41a] iterates over all the code instructions and calls `oplook()`[97a] to get additional information `o` about the instruction `p`. Then it calls the important function `asmout()`[102e] which generates actual ARM instructions from `p`. `asmout()` will also use the global `cout`[34d] to modify the executable file.

⟨asmb() *Text section* 42c⟩≡ (41a)
```
OFFSET = HEADR;
```

```
seek(cout, OFFSET, SEEK__START);

pc = INITTEXT;
for(p = firstp; p != P; p = p->link) {
    ⟨adjust curtext when iterate over instructions p 33d⟩
    ⟨adjust autosize when iterate over instructions p 61b⟩
    curp = p;
    ⟨asmb() in Text section generation, sanity check pc 43b⟩

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

    pc += o->size;
}
⟨asmb() before cflush, debug 219i⟩
cflush();

⟨asmb() Text section, output strings in text segment 208c⟩
}
```

Uses HEADR 36d, asmout() 102e, cout 34d, datblk() 44d, and oplook() 97a.

oplook() returns a pointer to an Optab[93b] entry I will describe later.

⟨asmb() *locals* 43a⟩+≡ (41a) ◁41b ??▷

```
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[36d] above while pc[49a] to INITTEXT[36e]. Indeed, offsets in the executable and memory addresses are different concepts, as explained in Section 2.1.4.

asmb() updates also the global pc above, but the address resolution of code instructions (in Instr.pcX) has already been done in dotext()[89b]. This is repeated here just for sanity checking:

⟨asmb() *in Text section generation, sanity check pc* 43b⟩≡ (42c)

```
if(p->pc != pc) {
    diag("phase error %lux sb %lux", p->pc, pc);
    if(!debug['a'])
        prasm(curp);
    pc = p->pc;
}
```

Uses curp 33c, debug 211a, diag() 222d, pc 49a, and prasm() 212b.

4.4.3 Data section

The data section generation assumes the layout of globals has already been done (via dodata()[87c]), just like the code section generation in the previous section assumed the layout of code has also been done (via dotext()[89b]).

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`[87a] should have been set.

⟨`asmb()` *Data section* 44a⟩≡ (41a)
```
curtext = P;
switch(HEADTYPE) {
⟨asmb() switch HEADTYPE (to position after text) cases(arm) 44b⟩
}

⟨asmb() if dynamic module, before datblk() 170f⟩

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 `datblk()` 44d and `datsize` 87a.

⟨`asmb()` *switch HEADTYPE (to position after text) cases(arm)* 44b⟩≡ (44a) 182a▷
```
case H_PLAN9:
    OFFSET = HEADR+textsize;
    seek(cout, OFFSET, SEEK__START);
    break;
```
Uses `HEADR` 36d, `HEADTYPE` 36a, `H_PLAN9` 143b, and `textsize` 89a.

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` 44c⟩≡ (280a)
```
#define Dbufslop 100
```

⟨*function* `datblk`*(arm)* 44d⟩≡ (280a)
```
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 45a⟩

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

    for(p = datap; p != P; p = p->link) {
        ⟨datblk() if sstring might continue 208d⟩
        // 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 45b⟩

        switch(p->to.type) {
        ⟨datblk() switch type of destination cases 45c⟩
        default:
            diag("unknown mode in initialization%P", p);
            break;
        }
    }
    write(cout, buf.dbuf, n);
}
```

Uses Dbufslop-7 44c, P 32f, buf 227b, cout 34d, curp 33c, datap 33a, and diag() 222d.

⟨datblk() *other locals* 45a⟩≡ (44d) 45d▷

```
long j;
```

⟨datblk() *sanity check multiple initialization* 45b⟩≡ (44d)

```
for(j=l+(c-i)-1; j>=l; j--)
    if(buf.dbuf[j]) {
        print("%P\n", p);
        diag("multiple initialization");
        break;
    }
```

Uses buf 227b and diag() 222d.

D_SCONST

⟨datblk() *switch type of destination cases* 45c⟩≡ (44d) 46a▷

```
case D_SCONST:
    for(; i<c; i++) {
        buf.dbuf[l] = p->to.sval[i];
        l++;
    }
    break;
```

Uses buf 227b.

D_CONST and endianess

⟨datblk() *other locals* 45d⟩+≡ (44d) ◁45a 47c▷

```
char *cast;
long d;
```

⟨datblk() *switch type of destination cases* 46a⟩+≡ (44d) ◁45c 187c▷
```
case D_CONST: case D_ADDR:
    d = p->to.offset;
    ⟨datblk() if D_ADDR case 47d⟩
    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 227b, curp 33c, diag() 222d, inuxi1 46b, inuxi2 46c, and inuxi4 46d.

⟨*global* inuxi1 46b⟩≡ (280a)
```
char inuxi1[1];
```

⟨*global* inuxi2 46c⟩≡ (280a)
```
char inuxi2[2];
```

⟨*global* inuxi4 46d⟩≡ (280a)
```
char inuxi4[4];
```

⟨main() *initialize globals(arm)* 46e⟩+≡ (34e) ◁39c 98c▷
```
nuxiinit(); // endianess conversion tables
```

⟨*function* nuxiinit *(arm)* 46f⟩≡ (280a)
```
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 188a⟩
```

```
    }
    ⟨nuxiinit() debug 47b⟩
    Bflush(&bso);
}
```
Uses bso 211c, find1() 47a, inuxi1 46b, inuxi2 46c, and inuxi4 46d.

⟨*function* find1 47a⟩≡ (280a)
```
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* 47b⟩≡ (46f)
```
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 211c, debug 211a, fnuxi4 187d, fnuxi8 187e, inuxi1 46b, inuxi2 46c, and inuxi4 46d.

D_ADDR

⟨datblk() *other locals* 47c⟩+≡ (44d) ◁45d 187b▷
```
Sym *v;
```

⟨datblk() *if* D_ADDR *case* 47d⟩≡ (46a)
```
v = p->to.sym;
if(v) {
    switch(v->type) {
    ⟨datblk() in D_ADDR case, switch symbol type cases 48a⟩
    }
    ⟨datblk() if dynamic module(arm) 172c⟩
}
```

⟨datblk() *in* D_ADDR *case, switch symbol type cases* 48a⟩≡ (47d) 170c ▷

```
case STEXT: case SSTRING:
    d += p->to.sym->value;
    break;
case SDATA: case SBSS:
    d += p->to.sym->value + INITDAT;
    break;
```

Uses INITDAT 36g, SBSS 164b, SDATA, SSTRING 32c, and STEXT 150b.

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 48b⟩≡ (285c)

```
/// 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 29d, SXREF, and diag() 222d.

Chapter 5

Loading Objects

Now that you have seen the code or a high-level view of the code of the main functions of `5l`, 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()`[49b].

5.1 Object file format: `.5`

Before reading the code of `ldobj()`[49b] 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()`[49b] is the *virtual program counter*:

⟨*global* `pc` 49a⟩≡ (276b)
```
long pc = 0;
```
Uses `pc` 49a.

`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)* 49b⟩≡ (51) 49c▷
```
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`[32d] or `datap`[33a] depending on the opcode.

⟨`ldobj()` *locals(arm)* 49c⟩+≡ (51) ◁49b 54a▷
```
Prog *p;
// enum<Opcode>
short o;
```

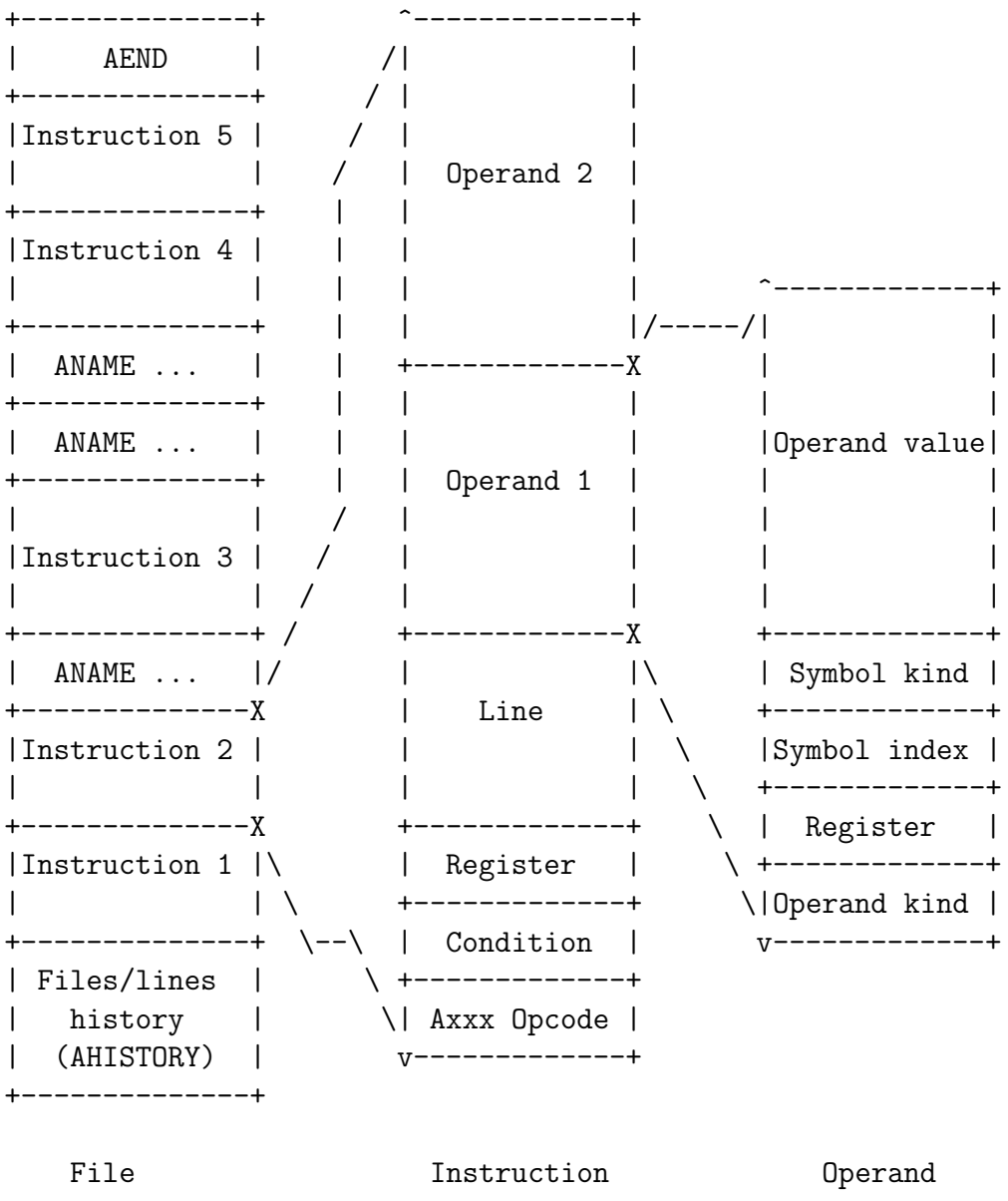


Figure 5.1: Format of a .5 object file.

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)* 51⟩≡ (283a)

```
/// main -> objfile -> <>
void
ldobj(fdt f, long c, char *pn)
{
    ⟨ldobj() locals(arm) 49b⟩

    ⟨ldobj() remember set of object filenames 64c⟩
    ⟨ldobj() bloc and bsize init 54b⟩

// can come from AEND
newloop:
    // new object file
    ipc = pc;
    ⟨ldobj() after newloop when new object file, more initializations 56b⟩

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

    ⟨ldobj() read if needed in loop:, adjust bloc and bsize 54c⟩

    o = bloc[0];  /* as */
    ⟨ldobj() sanity check opcode in range(arm) 52b⟩

    // dispatch opcode part one
    ⟨ldobj() if ANAME or ASIGNAME(arm) 56d⟩
    // else

    p = malloc(sizeof(Prog));
    p->as = o;
    ⟨ldobj() read one instruction in p 52a⟩
    p->link = P;
    p->cond = P;

    ⟨ldobj() sanity check p 52c⟩
    ⟨ldobj() debug 218c⟩

    // dispatch opcode part two
    switch(o) {
    ⟨ldobj() switch opcode cases(arm) 59d⟩
    }
    goto loop;

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

Uses `buf` 227b, `debug` 211a, and `errorexit()` 222b.

`ldobj()`[49b] 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 `5l` 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

An important chunk of `ldobj()`[49b] 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* 52a⟩≡ (51)
```
// 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()` 52d and `malloc()` 226a.

The code to read an operand uses `inopd()`[52d] 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)* 52b⟩≡ (51)
```
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");
    errorexit();
}
```
Uses `diag()` 222d and `pc` 49a.

⟨`ldobj()` *sanity check p* 52c⟩≡ (51)
```
if(p->reg > NREG)
    diag("register out of range %d", p->reg);
```
Uses `P` 32f.

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)* 52d⟩≡ (283a)
```
/// main -> objfile -> ldobj -> <>
static int
inopd(byte *p, Adr *a, Sym *h[])
{
    int size; // returned
    int symidx;
    ⟨inopd() other locals 144d⟩

    a->type = p[0];
    a->reg = p[1];
    ⟨inopd() sanity check register range 53b⟩
    symidx = p[2];
    ⟨inopd() sanity check symbol range 58a⟩
```

```
        a->sym = h[symidx];
        a->symkind = p[3];
        ⟨inopd() sanity check D_CONST ??⟩

        size = 4;

        switch(a->type) {
        ⟨inopd() cases 53a⟩
        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 144e⟩

        return size;
    }
```

Uses curauto 144c.

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**[28a].

The operands have variable size depending on the operand kind:

⟨inopd() *cases* 53a⟩≡ (52d) 182h ▷

```
    // 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:
    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() 226a.

⟨inopd() *sanity check register range* 53b⟩≡ (52d)

```
    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()`[49b] uses a global called `buf`[227b] of type `Buf`[226d] as well as many utility functions forming a sort of *input/output buffer management* library. This code is quite generic and independent of `5l` 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()`[54d]. A few locals of `ldobj()` are pointers in this array:

⟨`ldobj()` *locals(arm)* 54a⟩+≡ (51) ◁49c 56a▷
```
// 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* 54b⟩≡ (51)
```
bloc = buf.ibuf;
bsize = buf.ibuf;
```

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:

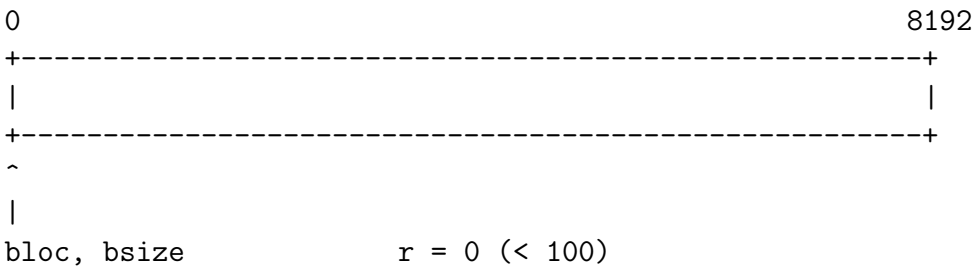
⟨`ldobj()` *read if needed in loop:, adjust bloc and bsize* 54c⟩≡ (51)
```
r = bsize - bloc;
if(r < 100 && r < c) {  /* enough for largest instruction */
    bsize = readsome(f, buf.ibuf, bloc, bsize, c);
    if(bsize == nil)
        goto eof;
    bloc = buf.ibuf; // readsome() does some memmove()
    goto loop;
}
```
Uses `buf` 227b and `readsome()` 54d.

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()`[52d] which can simply access `bloc[0]`, `bloc[1]`, etc without having to worry whether they need to call `readsome()` again.

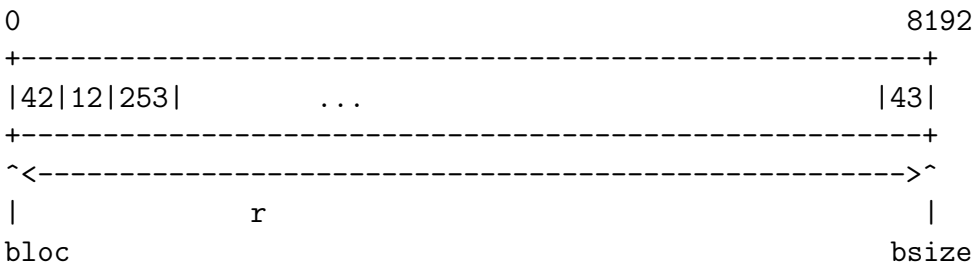
Figure 5.3 represents the evolution of the state of `buf.ibuf` while the input buffer gets filled another time by `readsome()`. The bytes at the end of the input buffer are first moved to its beginning and the rest of the buffer is filled by more data from `f`.

⟨*function* `readsome` 54d⟩≡ (282a)
```
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;
```

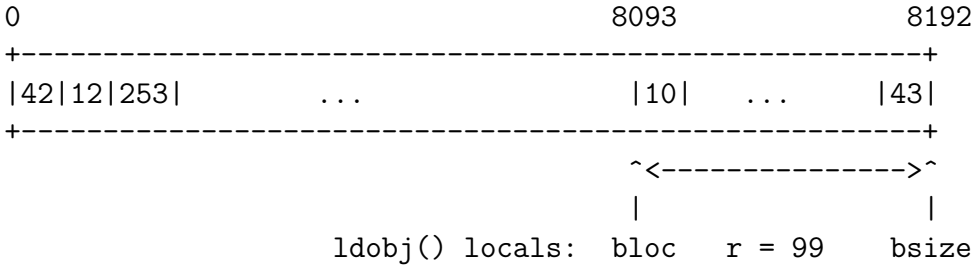


(a) Empty buffer in buf.ibuf

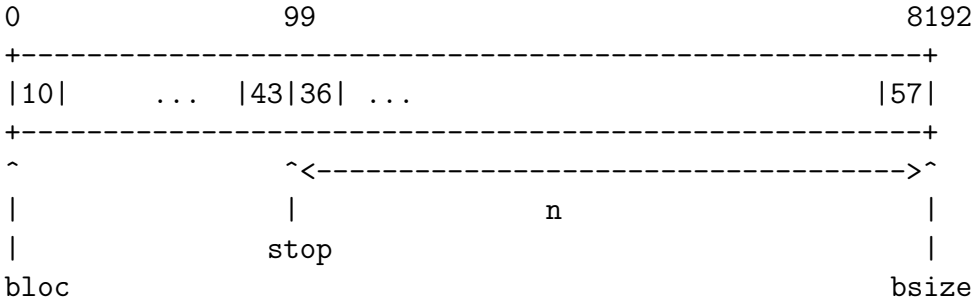


(b) After first call to readsome(): returned bsize

Figure 5.2: Evolution of input buffer and `ldobj()` pointers after `readsome()`.



(a) before second call to readsome()



(b) after second call to readsome()

Figure 5.3: Evolution of the input buffer and `readsome()` pointers.

```
    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()[49b] 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)* 56a⟩+≡ (51) ◁54a 56c▷

```
// 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 5l 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* 56b⟩≡ (51) 59b▷

```
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[28a] via lookup()[28e]. 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)* 56c⟩+≡ (51) ◁56a 60a▷

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

⟨ldobj() *if ANAME or ASIGNAME(arm)* 56d⟩≡ (51)

```
if(o == ANAME || o == ASIGNAME) {
    ⟨ldobj() if SIGNAME adjust sig 63b⟩

    stop = memchr(&bloc[3], '\0', bsize-&bloc[3]);
    ⟨ldobj() if stop is nil refill buffer and retry 58b⟩

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

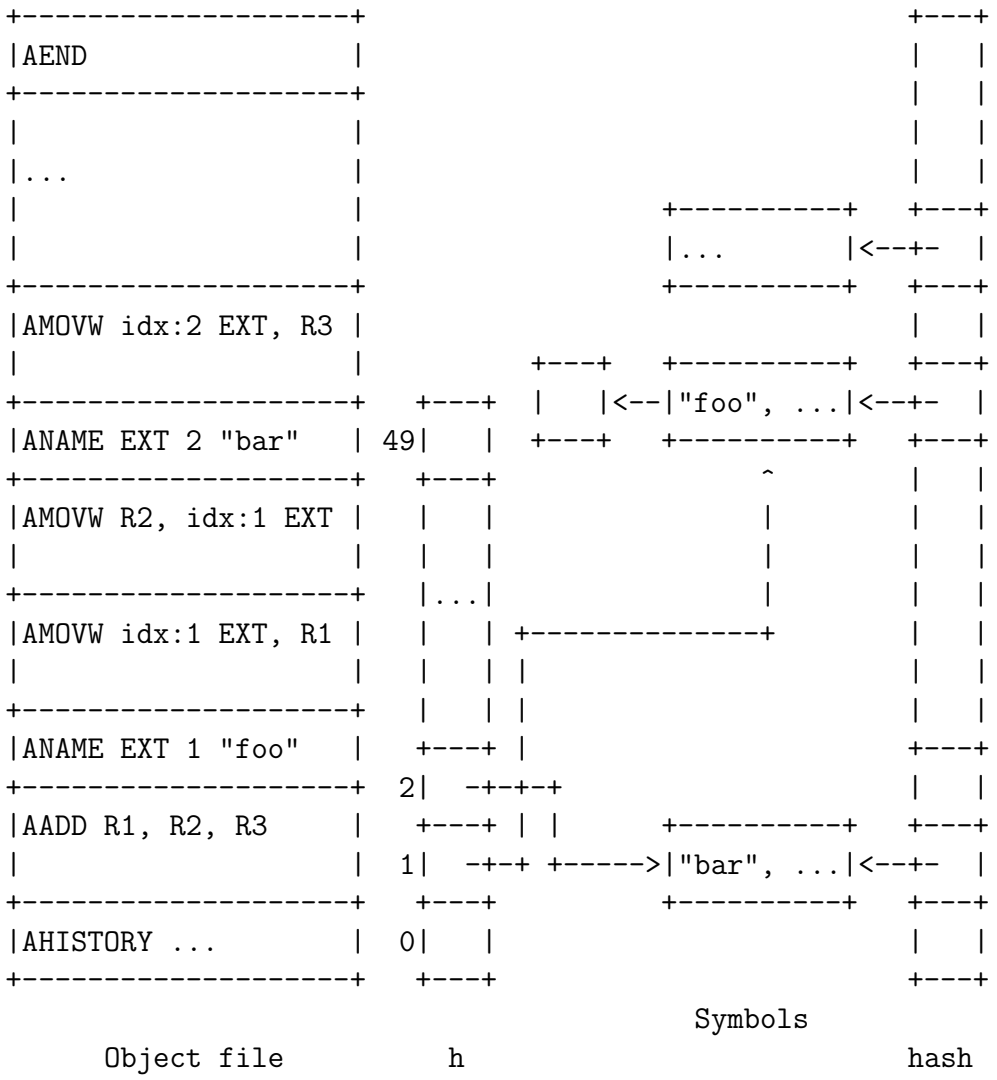


Figure 5.4: Example of object file and content of `h` and `hash`.

```
        bloc += 3;
        c -= 3;

        v = 0; // global version by default
        ⟨ldobj() when ANAME opcode, if private symbol adjust version 59c⟩

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

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

        ⟨ldobj() if sig not zero 63d⟩
        ⟨ldobj() when ANAME, debug 218d⟩

        h[symidx] = s;

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

        ⟨ldobj() when ANAME opcode, if N_FILE 150a⟩
        goto loop;
    }
```

Uses debug 211a, lookup() 28e, and version 283a.

`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()`[52d]. You can now also understand the sanity check of the symbol index in `inopd()`:

⟨inopd() *sanity check symbol range* 58a⟩≡ (52d)

```
    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 refering 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* 58b⟩≡ (56d)

```
    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){
            fprint(2, "%s: name too long\n", pn);
            errorexit();
        }
    }
```

Uses buf 227b and readsome() 54d.

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` 59a⟩≡ (283a)
```
int version = 0;
```

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

⟨`ldobj()` *after newloop when new object file, more initializations* 59b⟩+≡ (51) ◁56b 151a▷
```
version++;
```

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

⟨`ldobj()` *when ANAME opcode, if private symbol adjust version* 59c⟩≡ (56d)
```
if(k == N_INTERN)
    v = version;
```

5.4 Opcode dispatch

Once an instruction `p` with its opcode and operands have been read, `ldobj()`[49b] 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 A*xxx*

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

⟨`ldobj()` *switch opcode cases(arm)* 59d⟩≡ (51) 60b▷
```
default:
casedef:
    ⟨ldobj() in switch opcode default case, if skip 181d⟩

    // 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` 32g and `nopout()` 181f.

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`[32d] (via `lastp`[32g]) 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 `pc`[49a] 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 `hash`[28a] (see the code of `inopd()`[52d]) where `Sym.nameX` contains the procedure's name:

⟨ldobj() *locals(arm)* 60a⟩+≡ (51) ◁56c 63a▷
```
Sym *s;
```

⟨ldobj() *switch opcode cases(arm)* 60b⟩+≡ (51) ◁59d 61c▷
```
case ATEXT:
    ⟨ldobj() case ATEXT, if curtext not null adjustments for curauto 145b⟩
    curtext = p;
    ⟨ldobj() in switch opcode ATEXT case, reset skip 181e⟩
    ⟨ldobj() in switch opcode ATEXT case, set autosize 61a⟩

    s = p->from.sym;
    ⟨ldobj() sanity check for ATEXT symbol s 60c⟩
    s->type = STEXT;
    s->value = pc;

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

    p->pc = pc;
    pc++;

    ⟨ldobj() in switch opcode ATEXT case, populate textp 145f⟩
    break;
```
Uses STEXT 150b, autosize 60d, curauto 144c, diag() 222d, etextp 145e, lastp 32g, and pc 49a.

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.

⟨ldobj() *sanity check for ATEXT symbol s* 60c⟩≡ (60b)
```
if(s == S) {
    diag("TEXT must have a name\n%P", p);
    errorexit();
}
if(!(s->type == SNONE || s->type == SXREF)) {
    ⟨ldobj() case ATEXT and section not SNONE or SXREF, if DUPOK 181c⟩
    diag("redefinition: %s\n%P", s->name, p);
}
```
Uses S 29d and diag() 222d.

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

⟨*global* autosize*(arm)* 60d⟩≡ (276b)
```
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, `5l` first rounds the size at a 4 byte boundary and then increments it by 4:

⟨`ldobj()` *in switch opcode ATEXT case, set autosize* 61a⟩≡ (60b)
```
p->to.offset = rnd(p->to.offset, 4);
autosize = p->to.offset;
autosize += 4;
```

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`[33b]:

⟨*adjust autosize when iterate over instructions p* 61b⟩≡ (89b 42c)
```
if(p->as == ATEXT) {
    autosize = p->to.offset + 4;
}
```

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)* 61c⟩+≡ (51) ◁60b 61e▷
```
case AGLOBL:
    s = p->from.sym;
    ⟨ldobj() sanity check for AGLOBL symbol s 61d⟩
    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 `SNONE` 29d, `SXREF`, and `diag()` 222d.

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* 61d⟩≡ (61c)
```
if(s == S) {
    diag("GLOBL must have a name\n%P", p);
    errorexit();
}
if(!(s->type == SNONE || s->type == SXREF))
    diag("redefinition: %s\n%P", s->name, p);
```
Uses `S` 29d, `diag()` 222d, and `errorexit()` 222b.

5.4.4 ADATA

The code dealing with `DATA` pseudo-instructions is trivial. It just populates `datap`[33a]:

⟨`ldobj()` *switch opcode cases(arm)* 61e⟩+≡ (51) ◁61c 62b▷
```
case ADATA:
    ⟨ldobj() sanity check for ADATA symbol s 62a⟩

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

    break;
```
Uses `datap` 33a.

Note that `5l` could set the section of the symbol to `SDATAX` 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 `SDATAX` is done in `dodata()`[87c] later instead.

⟨`ldobj()` *sanity check for ADATA symbol s* 62a⟩≡ (61e)
```
if(p->from.sym == S) {
    diag("DATA without a sym\n%P", p);
    break;
}
```
Uses `S` 29d.

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)* 62b⟩+≡ (51) ◁61e 62c▷
```
case AEND:
    ⟨ldobj() case AEND, curauto adjustments with curhist 149c⟩
    ⟨ldobj() case AEND, curauto adjustments 145c⟩
    curtext = P;

    if(c)
        goto newloop;
    return;
```
Uses `P` 32f, `curauto` 144c, and `curtext` 33b.

The important instruction above is `return` which allows to exit from `ldobj()`[49b] without any error.

5.4.6 AGOK

`AGOK` (God Only Knows) is a pseudo-opcode used to initialize new instructions in `5l`. 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)* 62c⟩+≡ (51) ◁62b 148a▷
```
case AGOK:
    diag("unknown opcode\n%P", p);
    p->pc = pc;
    pc++;
    break;
```
Uses `diag()` 222d.

5.5 Safe linking

An interesting feature of object files and `5l`, 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.5` and `bar.5` 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.5` object file. At this point, without signatures, there is nothing that prevents the new `foo.5` and the old `bar.5` 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.5` 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.5` depends also on `foo.h`. Then, any modification of `foo.h` will trigger automatically the regeneration of `bar.5`. 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)* 63a⟩+≡ (51) ◁60a 64b▷
```
ulong sig;
```

⟨`ldobj()` *if SIGNAME adjust sig* 63b⟩≡ (56d)
```
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* 63c⟩≡ (27) 64a▷
```
// for instance last 32 bits of md5sum of the type of the symbol
ulong   sig;
```

⟨`ldobj()` *if sig not zero* 63d⟩≡ (56d)
```
if(sig != 0){
    ⟨ldobj() signature compatibility check 63e⟩
    s->sig = sig;
    ⟨ldobj() remember file introducing the symbol 64d⟩
}
```

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.5` and `bar.5`. 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:

⟨`ldobj()` *signature compatibility check* 63e⟩≡ (63d)
```
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()` 222d.

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* 64a⟩+≡ (27) ◁63c 165h▷
```
// index in filen[]
ushort  file;
```

⟨`ldobj()` *locals(arm)* 64b⟩+≡ (51) ◁63a 64e▷
```
// growing_array<option<string>>  (grown for every 16 elements)
static char **filen;
// index of next free entry in filen
static int files = 0;
```

⟨`ldobj()` *remember set of object filenames* 64c⟩≡ (51)
```
⟨ldobj() grow filen if not enough space 64f⟩
filen[files++] = strdup(pn);
```

⟨`ldobj()` *remember file introducing the symbol* 64d⟩≡ (63d)
```
s->file = files-1;
```

`filen` is actually a *growing array*:

⟨`ldobj()` *locals(arm)* 64e⟩+≡ (51) ◁64b 181a▷
```
char **nfilen; // new filen
```

⟨`ldobj()` *grow filen if not enough space* 64f⟩≡ (64c)
```
if((files&15) == 0){
    nfilen = malloc((files+16)*sizeof(char*));
    memmove(nfilen, filen, files*sizeof(char*));
    free(filen);
    filen = nfilen;
}
```
Uses `malloc()` 226a.

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 65a⟩≡ (266c)
```
#define ARMAG "!<arch>\n"
```

⟨*constant* SARMAG 65b⟩≡ (266c)
```
#define SARMAG 8
```

⟨*constant* ARFMAG 65c⟩≡ (266c)
```
#define ARFMAG "`\n"
```

⟨*constant* SARNAME 65d⟩≡ (266c)
```
#define SARNAME 16
```

⟨*struct* ar_hdr 65e⟩≡ (266c)
```
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` 66a⟩≡ (266c)
```
#define SAR_HDR (SARNAME+44)
```

⟨*global symname linker* 66b⟩≡ (283a)
```
char symname[] = SYMDEF;
```

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()`[49b]. 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* 66c⟩≡ (40a)
```
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` 226d.

⟨objfile() *when file is a library* 67⟩≡ (40a)

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

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

⟨objfile() sanity check library header size and content 68a⟩

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)) {
            sprint(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 68b⟩
            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);
                errorexit();
            }
            work = true; // maybe some new SXREF has been found in ldobj()
           ⟨objfile() an SXREF was found hook 72a⟩
        }
```

```
        }
    }
    return;

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

Uses DBG 272, SNONE 29d, SXREF, atolwhex() 228c, diag() 222d, ldobj() 49b, lookup() 28e, malloc() 226a, and xrefresolv 71f.

⟨objfile() *sanity check library header size and content* 68a⟩≡ (67)

```
    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() 222d and symname 283a.

⟨objfile() *sanity check entry header* 68b⟩≡ (67)

```
    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 68c⟩≡ (283b)

```
    // growing_array<dirname>
    char** libdir;
```

⟨*global* nlibdir 68d⟩≡ (283b)

```
    // index of next free entry in libdir
    int nlibdir = 0;
```

Uses nlibdir 68d.

⟨*global* maxlibdir 68e⟩≡ (283b)

```
    // index of last free entry in libdir
    static int maxlibdir = 0;
```

Uses maxlibdir-1 68e.

6.3.2 5l -L

⟨main() *command line processing(arm)* 68f⟩+≡ (34e) ◁38b 164d▷

```
    case 'L':
        addlibpath(EARGF(usage()));
        break;
```

⟨main() *locals(arm)* 69a⟩≡ (34e) 69c▷
```
char *root;
```

⟨*constant* LIBNAMELEN 69b⟩≡ (272)
```
#define LIBNAMELEN  300
```

⟨main() *locals(arm)* 69c⟩+≡ (34e) ◁69a
```
int c;
char name[LIBNAMELEN];
char *a;
```

⟨main() *addlibpath("/thestring/lib") or ccroot* 69d⟩≡ (35c)
⟨main() *change root if ccroot* 69e⟩
```
// usually /{thestring}/lib/ as root = ""
snprint(name, sizeof(name), "%s/%s/lib", root, thestring);
addlibpath(name);
```

⟨main() *change root if ccroot* 69e⟩≡ (69d)
```
root = getenv("ccroot");

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

⟨*function* addlibpath 69f⟩≡ (283b)
```
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 69g⟩
        memmove(p, libdir, nlibdir*sizeof(*p));
        free(libdir);
        libdir = p;
    }

    libdir[nlibdir++] = strdup(arg);
}
```
Uses free() 226b, libdir 68c, malloc() 226a, maxlibdir-1 68e, and nlibdir 68d.

⟨addlibpath() *sanity check p* 69g⟩≡ (69f)
```
if(p == nil) {
    diag("out of memory");
    errorexit();
}
```
Uses diag() 222d and errorexit() 222b.

6.3.3 5l -lxxx

⟨objfile() *adjust file if -lxxx filename* 70a⟩≡ (40a)
```
if(file[0] == '-' && file[1] == 'l') {
    snprint(pname, sizeof(pname), "lib%s.a", file+2);
    e = findlib(pname);
    if(e == nil) {
        diag("cannot find library: %s", file);
        errorexit();
    }
    snprint(name, sizeof(name), "%s/%s", e, pname);
    file = name;
}
```
Uses DBG 272, diag() 222d, errorexit() 222b, and findlib() 70b.

⟨*function* findlib 70b⟩≡ (283b)
```
char*
findlib(char *file)
{
    int i;
    char name[LIBNAMELEN];

    for(i = 0; i < nlibdir; i++) {
        snprint(name, sizeof(name), "%s/%s", libdir[i], file);
        if(fileexists(name))
            return libdir[i];
    }
    return nil;
}
```
Uses LIBNAMELEN 226d, fileexists() 228b, libdir 68c, and nlibdir 68d.

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()[71e]. 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.

⟨ldobj() *in AHISTORY case, if pragma lib* 70c⟩≡ (148a)
```
if(p->to.offset == -1) {
    addlib(pn);
    histfrogp = 0;
    goto loop;
}
```
Uses addlib() 72b.

⟨main() *load implicit libraries* 70d⟩≡ (38d)
```
if(!debug['l'])
    loadlib();
```

⟨main() *if rare condition do not set SXREF for INITENTRY, else* 70e⟩≡ (38c) 207c ▷
```
if(debug['l']) {}
else
```

⟨*global* `library` 71a⟩≡ (283b)
```
// array<option<filename>>
char* library[50];
```

⟨*global* `libraryp` 71b⟩≡ (283b)
```
// index of first free entry in library array
int libraryp;
```

⟨*global* `libraryobj` 71c⟩≡ (283b)
```
char* libraryobj[50];
```

⟨*function loadlib simple version* 71d⟩≡
```
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` 71e⟩≡ (283b)
```
void
loadlib(void)
{
    int i;
    long h;
    Sym *s;
loop:
    ⟨loadlib() reset xrefresolv 71g⟩
    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 71h⟩
}
```
Uses `DBG` 272, `library` 71a, `libraryobj` 71c, `libraryp` 71b, and `objfile()` 40a.

⟨*global* `xrefresolv` 71f⟩≡ (276b)
```
bool xrefresolv;
```

⟨`loadlib()` *reset xrefresolv* 71g⟩≡ (71e)
```
xrefresolv = false;
```
Uses `xrefresolv` 71f.

⟨`loadlib()` *if xrefresolv* 71h⟩≡ (71e)
```
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` 272, `S` 29d, `SXREF`, and `xrefresolv` 71f.

⟨`objfile()` *an SXREF was found hook* 72a⟩≡ (67)
```
xrefresolv = true;
```

`addlib()`[72b] 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 `$O` with the architecture character (e.g., `'5'`) and `$M` with the architecture string (e.g., `"arm"`), then deduplicates against libraries already recorded.

⟨*function* `addlib` 72b⟩≡ (283b)
```
/// 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] == '/') {
        sprint(name, "");
        i = 1;
    } else if(histfrog[0]->name[1] == '.') {
        sprint(name, ".");
        i = 0;
    } else {
        sprint(name, "");
        i = 0;
        search = true;
    }

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

        // s/$O/<thechar>/
        for(;;) {
            p = strstr(comp, "$O");
            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){
            snprint(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 226d, diag() 222d, findlib() 70b, histfrog 150d, histfrogp 150g, library 71a, libraryobj 71c, libraryp 71b, malloc() 226a, thechar 34a, and thestring 34b.

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* 74⟩≡ (38d)

```
⟨main() if export table or dynamic module(arm) 165b⟩

patch();
⟨main() call doprofxxx() if profiling 156b⟩
noops();

dodata();
dotext();
```

`patch()`[77] builds a graph of code instructions, `noops()`[82] rewrites virtual instructions in machine instructions, `dodata()`[87c] layout data by assigning a memory address to each globals, and finally `dotext()`[89b] 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 rational 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.

[1] Remember 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()`[49b] after each code instruction read. Real code addresses on the other hand start at `INITTEXT`[36e] (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 `5l` 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 `5l` operates in three steps:

1. The function `patch()`[77] 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()`[89b] sets `pc`[49a] 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()`[97a]), 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()`[102e] when involved with a branching instruction `p` can find the target code address by simply looking at `p->cond->pc`.

[2]Note 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()`[82] 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 at 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, `5l` 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. `5l` would get a size for the code section which it could use as the starting point of the data section, to layout data. Finally, `5l` 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 `5l` to solve the mutual dependency problem operates in four steps:

1. The layout of data is done first, via `dodata()`[87c], by iterating over all the globals. But, each globals 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()`[89b] 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`[36g], 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()`[102e]) 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()`[77].

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)* 77⟩≡ (279b)

```
/// main -> <>
void
patch(void)
{
    Prog *p;
    Prog *q;
    long c; // not ulong?
    ⟨patch() other locals 78a⟩

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

    ⟨patch() initialisations 80a⟩

    // pass 1
    for(p = firstp; p != P; p = p->link) {
        ⟨adjust curtext when iterate over instructions p 33d⟩

        ⟨patch() resolve branch instructions using symbols 78b⟩

        if(p->to.type == D_BRANCH && p->cond != UP) {
            c = p->to.offset; // target pc
            ⟨patch() find Prog reference q with q->pc == c 79⟩
            p->cond = q;
        }
    }
    // pass 2
    ⟨patch() optimisation pass 177d⟩
}
```

Uses `curtext` 33b and `mkfwd()` 80c.

The extra condition `p->cond != UP` above is not very important and can be ignored for now. It will be explained later (see `UP`[272]). Normally, before the call to `patch()`[77], `Instr.condX` should be a null pointer[3] 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* 78a⟩≡ (77)
```
Sym *s;
// enum<Opcode>
int a;
```

⟨`patch()` *resolve branch instructions using symbols* 78b⟩≡ (77)
```
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 78c⟩
    }
}
```
Uses `S` 29d and `STEXT` 150b.

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

⟨`patch()` *switch section type for branch instruction, cases* 78c⟩≡ (78b) 169g▷
```
// SNONE, SXREF, etc
default:
    diag("undefined: %s\n%P", s->name, p);
    s->type = STEXT;
    s->value = 0;
    break;
```
Uses `STEXT` 150b and `diag()` 222d.

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, `5l` will not report an error. Note that there is no risk of generating a wrong executable though as the call to `diag()`[222d] 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()`[77] 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* 78d⟩+≡ (31a) ◁32b 101a▷
```
Prog*   forwd;
```

[3] See the instruction `p->cond = P;` in `ldobj()`[49b].

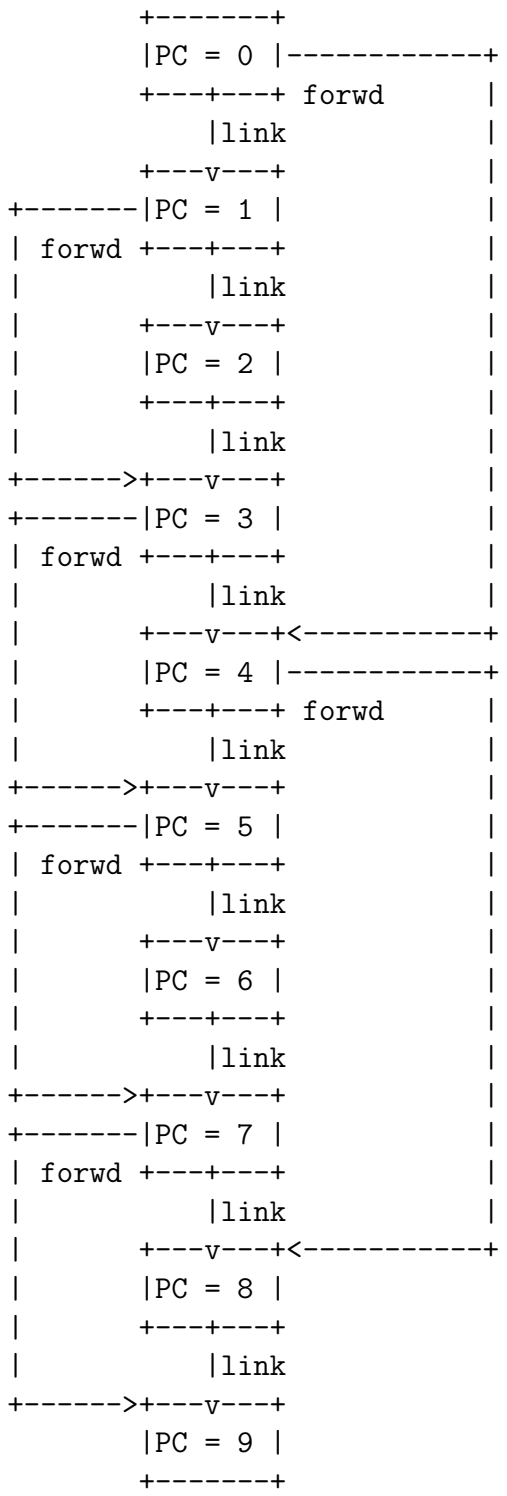


Figure 7.1: Forward pointers.

That way you can make bigger jumps sometimes:

⟨patch() *find Prog reference* q *with* q->pc == c 79⟩≡ (77)

```
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!
```

```
            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 32f, diag() 222d, and firstp 32d.

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 5l 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* 80a⟩≡ (77)
```
mkfwd();
```

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 80b⟩≡ (279b)
```
#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 80c⟩≡ (279b)
```
/// 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 81a⟩

    i = 0;
    for(p = firstp; p != P; p = p->link) {
        ⟨adjust curtext when iterate over instructions p 33d⟩
        p->forwd = P;

        ⟨mkfwd() in for loop, add forward links from past p in lst to p 81b⟩
    }
}
```

Uses LOG-15 and curtext 33b.

The code of mkfwd()[80c] 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[80b]) for a certain level i is

stored in `cnt[i]` initialized by the following code:

⟨`mkfwd()` *initializes cnt, dwn, lst* 81a⟩≡ (80c)
```
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`.

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* 81b⟩≡ (80c)
```
// 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` and `P` 32f.

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)* 82⟩≡ (284a)

```
/// 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 33d⟩

        switch(p->as) {
        ⟨noops() first pass switch opcode cases 83b⟩
        }
        q = p;
    }

    // pass 2, transform
    curtext = P;
    for(p = firstp; p != P; p = p->link) {
        ⟨adjust curtext when iterate over instructions p 33d⟩
```

```
        o = p->as;
        switch(o) {
        ⟨noops() second pass switch opcode cases 83c⟩
        }
    }
}
```

Uses DBG 272, P 32f, and curtext 33b.

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()[82]:

⟨Mark cases 83a⟩≡ (32c) 178b▷

```
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 83b⟩≡ (82) 85▷

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

case ABL:
    if(curtext != P)
        curtext->mark &= ~LEAF;
    // fallthrough
```

⟨noops() first pass switch opcode ABL fallthrough 86⟩

Uses LEAF 272, P 32f, and curtext 33b.

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. 5l 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 83c⟩≡ (82) 84b▷

```
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 84a⟩

    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 272, autosize 60d, curtext 33b, and prg() 39a.

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[33b].

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* 84a⟩≡ (83c)

```
if((autosize == 0) && !(curtext->mark & LEAF)) {
    DBG("save suppressed in: %s\n", curtext->from.sym->name);
    curtext->mark |= LEAF;
}
```

Uses DBG 272, LEAF 272, autosize 60d, and curtext 33b.

7.3.3 ARET rewriting

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

⟨noops() *second pass switch opcode cases* 84b⟩+≡ (82) ◁83c 187a▷

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

```
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.

```
    break;
```
Uses `LEAF` 272, `autosize` 60d, `curtext` 33b, and `zprg` 39b.

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()`[82] 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* 85⟩+≡ (82) ◁83b 199a▷
```
  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
```

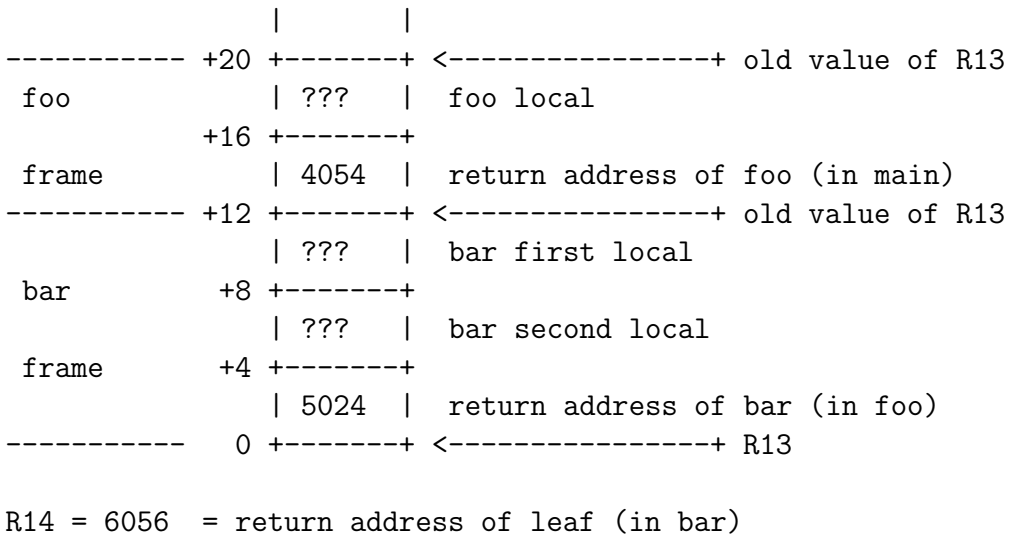


Figure 7.3: Stack before `B (R14)` in `leaf()`.

The compiler `5c` occasionally generates code using `ANOP` (see the COMPILER book [Pad16a]). `5l` 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* 86⟩≡ (83b)

```
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` 32f.

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()`[87c] 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` 87a⟩≡ (276b)
```
long datsize;
```

⟨*global* `bsssize` 87b⟩≡ (276b)
```
long bsssize;
```

Those globals are then used to generate the `a.out` header in `asmout()`[102e], 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 (`SDATAX`) instead of the BSS section (`SBSS`[164b]). 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)* 87c⟩≡ (279a)
```
/// 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 88b⟩
    }

    ⟨dodata() if string in text segment 207f⟩

    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 88a⟩
            v = rnd(v, 4); // align
            s->value = v;   // adjust
```

```
            ⟨dodata() in pass 1, if small data size, adjust orig 177b⟩
        }
    }

    /*
     * 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 177c⟩
        }
    }
    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 90d⟩
}
```

Uses BIG, DBG 272, NHASH, P 32f, S 29d, SBSS 164b, SDATA, datap 33a, datsize 87a, rnd() 229a, and xdefine() 92a.

Note that 5l is just laying out globals here. 5l uses mostly the GLOBL declarations which are stored in the symbol table hash[28a]. The use of the DATA instructions and datap[33a] to generate the data section in the executable is done in datblk()[44d], not in dodata().

⟨dodata() *sanity check GLOBL instructions, size of data v* 88a⟩≡ (87c)

```
if(v == 0) { // check
    diag("%s: no size", s->name);
    v = 1;
}
```

Uses diag() 222d.

⟨dodata() *sanity check DATA instructions* 88b⟩≡ (87c)

```
if(s->type != SDATA)
    diag("initialize non-data (%d): %s\n%P",
        s->type, s->name, p);
```

```
v = p->from.offset + p->reg;
if(v > s->value)
    diag("initialize bounds (%ld): %s\n%P",
        s->value, s->name, p);
```

Uses SDATA and diag() 222d.

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()[89b]. 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 89a⟩≡ (276b)
```
long textsize;
```

This global is also used to generate the a.out header in asmout()[102e], as shown in Section 4.4.1. dotext() also computes INITDAT[36g] 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)* 89b⟩≡ (279a)
```
/// main -> <>
void
dotext(void)
{
    Prog *p;
    Optab *o;
    // real code address
    long c; // ulong?
    // size of instruction
    int m;
    ⟨dotext() other locals 90a⟩

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

    c = INITTEXT;
    ⟨dotext() initialisation 90b⟩

    for(p = firstp; p != P; p = p->link) {
        ⟨adjust curtext when iterate over instructions p 33d⟩
        ⟨adjust autosize when iterate over instructions p 61b⟩

        // real program counter transition
        p->pc = c;

        o = oplook(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 90c⟩
            } else {
                diag("zero-width instruction\n%P", p);
            }
        } else {
            ⟨dotext() pool handling for optab o 135e⟩
        }
    }

    ⟨dotext() if string in text segment 208a⟩

    c = rnd(c, 8);

    textsize = c - INITTEXT;
    ⟨dotext() refine special symbols 92c⟩
    if(INITRND)
        INITDAT = rnd(c, INITRND);
    DBG("tsize = %lux\n", textsize);
}
```

Uses DBG 272, INITDAT 36g, INITRND 36f, INITTEXT 36e, diag() 222d, lookup() 28e, oplook() 97a, rnd() 229a, and textsize 89a.

`dotext()` relies on modifications done by the previous functions of the resolving phase:

- `noops()`[82], so `oplook()`[97a] does not have to deal with virtual instructions (except `MOVW`)
- `patch()`[77], which is needed for `noops()` to work
- `dodata()`[87c], 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()`[41a] and `asmout()`, not in `dotext()`.

⟨dotext() *other locals* 90a⟩≡ (89b) 207g▷

```
long    otxt;
```

⟨dotext() *initialisation* 90b⟩≡ (89b)

```
otxt = c;
```

Uses INITTEXT 36e.

⟨dotext() *detect if large procedure* 90c⟩≡ (89b)

```
if(c - otxt >= 1L<<17) {
    diag("Procedure %s too large\n", TNAME);
    errorexit();
}
otxt = c;
```

Uses TNAME 272, diag() 222d, and errorexit() 222b.

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* 90d⟩≡ (87c) 92b▷

```
xdefine("bdata", SDATA, 0L);
xdefine("edata", SDATA, datsize);
xdefine("end",   SBSS, datsize+bsssize);
```

Uses SDATA and xdefine() 92a.

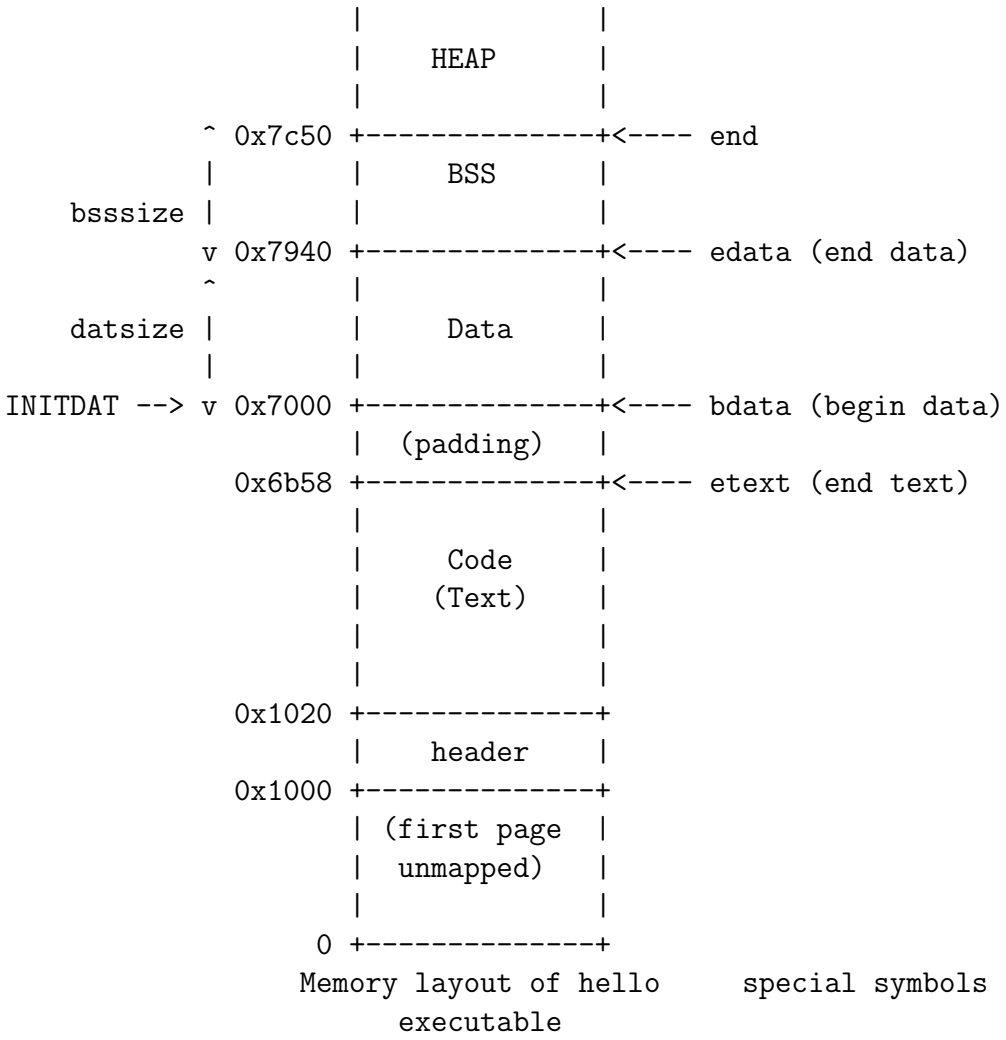


Figure 7.4: Memory layout of `hello` and special symbols.

⟨*function* `xdefine` *(arm)* 92a⟩≡ (279a)
```
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` 29d, `SXREF`, and `lookup()` 28e.

Note that the `Sym.valueX` for the `bdata` symbol is `0`, not `INITDAT`[36g]. 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()`[87c]. But, it is important to layout those special symbols in `dodata()`, before `dotext()`[89b], 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* 92b⟩+≡ (87c) ◁90d 176c▷
```
xdefine("etext", STEXT, 0L);
```
Uses `SBSS` 164b, `bsssize` 87b, `datsize` 87a, and `xdefine()` 92a.

But its final value can be set only in `dotext()`:

⟨`dotext()` *refine special symbols* 92c⟩≡ (89b)
```
lookup("etext", 0)->value = INITTEXT+textsize;
```
Uses `INITTEXT` 36e and `textsize` 89a.

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 `5l` defines is `setR12` which I mentioned before:

⟨`dodata()` *define special symbols setR12 IF unoptimized* 92d⟩≡
```
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`[36g]. 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()`[41a] 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()`[97a] and `asmout()`[102e]:

⟨`asmb()` *Text section, in iteration over instruction* `p` *(repeated code)* 93a⟩≡
```
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`[93b].

8.1 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()`[102e] 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.

8.1.1 `Optab` and `optab`

The structure below represents a rule:

⟨*struct* `Optab` *(arm)* 93b⟩≡ (276a)
```
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 125d⟩
    ⟨Optab flag field 121b⟩
};
```

A very important global is `optab` which contains all the rules of the ARM code generator:

⟨global optab (linkers/5l/optab.c)(arm) 94a⟩≡ (277a)

```
Optab optab[] =
{
    ⟨optab entries 106b⟩
    { AXXX,  C_NONE, C_NONE, C_NONE,   0, 4 },
};
```

Uses C_BRANCH 95c and C_NONE 95c.

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* marker[1]. It is recognized by `buildop()`[98d] which I will cover later.

Here are examples of entries in `optab`, that is rules:

⟨optab example of entries (repeated) 94b⟩≡

```
//           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`[95c]) for operands, *then* the first entry (the action id `1`) in the dispatch code of `asmout()`[102e] 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 8.1.2, `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 8.1.3, 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`[107e], `C_LCON`[107e], 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()`[99b] 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`[125d] and `Optab.flag`[121b] which will be presented later in this chapter.

[1] `NULL` is often used for this purpose.

8.1.2 Oprange

The structure below represents a *range* of entries in the sorted global `optab`[94a]:

⟨*struct* `Oprange` *(arm)* 95a⟩≡ (276a)
```
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)* 95b⟩≡ (282c)
```
// map<enum<Opcode>, Oprange>
Oprange oprange[ALAST];
```

`oprange`[95b] is also set by `buildop()`[98d] 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()`[97a] to find the rule matching an instruction (see the code of `oplook`[97a]). 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()`[98a], 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.

8.1.3 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)* 95c⟩≡ (276a)
```
// 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 107e⟩

    // D_OREG
    ⟨Operand_class C_xOREG cases 109a⟩
    // D_OREG with symbol N_EXTERN (or N_INTERN)
```

```
    ⟨Operand_class C_xEXT cases 110b⟩
    // D_OREG with symbol N_PARAM or N_LOCAL
    ⟨Operand_class C_xAUTO cases 111b⟩

    // D_ADDR
    ⟨Operand_class C_xxCON cases 111f⟩

    ⟨Operand_class cases 116b⟩
    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`[107e], `C_LCON`[107e], 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()`[97a]:

⟨*function* `aclass`*(arm)* 96a⟩≡ (282c)
```
/// oplook | ... -> <>
int
aclass(Adr *a)
{
    ⟨aclass() locals 96c⟩

    switch(a->type) {
    ⟨aclass() switch type cases 96b⟩
    }
    return C_GOK;
}
```
Uses C_PSR 201b.

⟨`aclass()` *switch type cases* 96b⟩≡ (96a) 108a▷
```
case D_NONE:
    return C_NONE;
case D_REG:
    return C_REG;
case D_BRANCH:
    return C_BRANCH;
```
Uses C_NONE 95c and C_REG 95c.

Up until now the code of `aclass()`[96a] 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()`[87c]). 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* 96c⟩≡ (96a)
```
Sym *s;
// enum<Section>
int t;
```

8.2 `oplook()` and `buildop()`

You are now ready to understand the code of `oplook()`. `oplook()` assumes the global `optab`[94a] has been sorted and `oprange`[95b] has been set. Those two operations are performed by `buildop()`[98d] which I will present after `oplook()`.

8.2.1 `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)* 97a⟩≡ (282c)
```
/// main -> (dotext | asmb) -> <>
Optab*
oplook(Prog *p)
{
    // enum<Opcode> (to index oprange[])
    int r;
    // enum<Operand_class>
    int a1, a2, a3;
    // ref<Optab> (from = optab)
    Optab *o, *e;
    ⟨oplook() other locals 101c⟩

    ⟨oplook() if use opti, part1 101d⟩
    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 97b⟩

    ⟨oplook() debug 218e⟩

    ⟨oplook() if use opti, part2 102d⟩
    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 97c⟩
}
```
Uses `C_NONE` 95c, `C_REG` 95c, `aclass()`, and `cmp()` 98a.

If no rule is found a linking error is reported:

⟨`oplook()` *sanity check o* 97b⟩≡ (97a)
```
if(o == nil) {
    o = oprange[r].stop; /* just generate an error */
}
```
Uses `oprange`.

⟨`oplook()` *illegal combination error* 97c⟩≡ (97a)
```
diag("illegal combination %A %d %d %d",  p->as, a1, a2, a3);
prasm(p);
```

```
  if(o == nil)
      o = optab;
  return o;
```

Uses diag() 222d and prasm() 212b.

The matching of operand classes in oplook()[97a] is using simply == for the middle operand (a2). Indeed, this operand class is always either a register (C_REG[95c]) or nothing (C_NONE[95c]). But, for the other operands the function cmp() (for compatible) below is used:

⟨*function* cmp*(arm)* 98a⟩≡ (282c)

```
  bool
  cmp(int a, int b)
  {

      if(a == b)
          return true;

      switch(a) {
      ⟨cmp() switch on a, the operand class in optab rule, cases 98b⟩
      }
      return false;
  }
```

Uses C_HFOREG 184h.

cmp()[98a] 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[94a] (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[107e], then it will also match the actual operand classes C_RCON[107e] and C_NCON[107e] of the instruction thanks to this code:

⟨cmp() *switch on a, the operand class in optab rule, cases* 98b⟩≡ (98a) 109d▷

```
  case C_LCON:
      if(b == C_RCON || b == C_NCON)
          return true;
      break;
```

Uses C_LCON 107e, C_NCON 107e, and C_RCON 107e.

Indeed, a large constant (C_LCON) is more general than a rotatable constant (C_RCON) or a negative rotatable constant (C_NCON). Thanks to this encoding of inclusion via cases of cmp(), a single rule can match many different kinds of instructions. Just like oprange[95b] 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.

8.2.2 buildop() and ocmp()

buildop() is pretty simple. It sorts optab[94a] and then sets oprange[95b] which is used by oplook()[97a]:

⟨main() *initialize globals(arm)* 98c⟩+≡ (34e) ◁46e 104c▷

```
  buildop();
```

⟨*function* buildop*(arm)* 98d⟩≡ (282c)

```
  /// main -> <>
  void
  buildop(void)
  {
      int i, n;
      // enum<Opcode> representing a range
      int r;
```

```
    ⟨buildop() initialize xcmp cache 102b⟩
    ⟨buildop() initialize flags 185c⟩

    for(n=0; optab[n].as != AXXX; n++) {
        ⟨buildop() adjust optab if flags, remove certain rules 185d⟩
    }
    // 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 oprange for the representants
        oprange[r].start = optab+i;
        while(optab[i].as == r)
            i++;
        oprange[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 99a⟩
        default:
            diag("unknown op in build: %A", r);
            errorexit();
        }
    }
}
```

Uses armv4 202a, debug 211a, diag() 222d, errorexit() 222b, ocmp() 99b, oprange, and optab 94a.

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* 99a⟩≡ (98d) 106c ▷

```
case AXXX:
  break;
```

The ordering of rules in the sorted `optab` is governed by the function below:

⟨*function* ocmp*(arm)* 99b⟩≡ (282c)

```
/// 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 202e⟩
    ⟨ocmp() if floating point flag on p1 or p2 185a⟩
    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()`[99b] 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`[95c] matters. Indeed, if `C_RCON`[107e] was after `C_LCON`[107e], 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()`[98a]. 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.

8.2.3 Optimizations

There are a few optimizations which improve `oplook()`[97a], `aclass()`[96a], and `cmp()`[98a], which I will cover now.

Caching

The first set of optimizations involves *caching* (a.k.a *memoizing*). `oplook()`[97a] is called once from `dotext()`[89b] and another time from `asmb()`[41a] 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* 101a⟩+≡ (31a) ◁78d
```
// option<index in optab[] (1-based)>, 0 means None, i means optab[i-1]
byte    optab;
```

In the same way, `aclass()`[96a] is called from `oplook()` but also from `asmout()`[102e], so `5l` can also cache the result of `aclass()` in the operand itself:

⟨`Adr` *other fields* 101b⟩+≡ (30a) ◁30b 144b▷
```
// 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* 101c⟩≡ (97a) 102c▷
```
bool useopti = true;
int i;
```

⟨`oplook()` *if use opti, part1* 101d⟩≡ (97a)
```
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()` and `optab` 94a.

⟨`oplook()` *if use opti, in part2, memoize matching rule* `o` 101e⟩≡ (102d)
```
p->optab = (o-optab)+1;
```

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)* 101f⟩≡ (284a)
```
void
nocache(Prog *p)
{
    p->optab = 0;
    p->from.class = 0;
    p->to.class = 0;
}
```

Precomputing

Another optimization *precomputes* the result of `cmp()`[98a] on any pair of operand classes:

⟨*global* `xcmp` *(arm)* 102a⟩≡ (282c)
```
bool xcmp[C_GOK+1][C_GOK+1];
```

⟨`buildop()` *initialize xcmp cache* 102b⟩≡ (98d)
```
for(i=0; i<C_GOK; i++)
    for(n=0; n<C_GOK; n++)
        xcmp[i][n] = cmp(n, i);
```
Uses `C_GOK` 95c.

This in turns will improve `oplook()`[97a].

⟨`oplook()` *other locals* 102c⟩+≡ (97a) ◁101c
```
bool *c1, *c3;
```

⟨`oplook()` *if use opti, part2* 102d⟩≡ (97a)
```
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 101e⟩
            return o;
        }
}
```
Uses `optab` 94a and `xcmp` 282c.

8.3 `asmout()`

`asmout()` is the heart of the ARM code generator. Given an instruction `p` and a rule `o` found via `oplook()`[97a], `asmout()` will dispatch the action identifier in `o->type` in a big `switch` which generates the ARM code:

⟨*function* `asmout` *(arm)* 102e⟩≡ (281)
```
/// 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 106a⟩

    o1 = o2 = o3 = o4 = o5 = o6 = 0;

    // first switch, action id dispatch, set o1, o2, ...
    switch(o->type) {
    ⟨asmout() switch on type cases 106d⟩
    default:
        diag("unknown asm %d", o->type);
        prasm(p);
        break;
    }

    ⟨asmout() debug 219a⟩
```

```
    // second switch, output o1, o2, ...
    switch(o->size) {
    ⟨asmout() switch on size cases 103⟩
    }
}
```

Uses diag() 222d and prasm() 212b.

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 lputl()[104d] in cases of the second switch:

⟨asmout() *switch on size cases* 103⟩≡ (102e)

```
case 4:
    ⟨asmout() when 1 generated instruction, debug 219b⟩
    lputl(o1);
    break;
case 8:
    ⟨asmout() when 2 generated instructions, debug 219c⟩
    lputl(o1);
    lputl(o2);
    break;
case 12:
    ⟨asmout() when 3 generated instructions, debug 219d⟩
    lputl(o1);
    lputl(o2);
    lputl(o3);
    break;
case 16:
    ⟨asmout() when 4 generated instructions, debug 219e⟩
    lputl(o1);
    lputl(o2);
    lputl(o3);
    lputl(o4);
    break;
case 20:
    ⟨asmout() when 5 generated instructions, debug 219f⟩
    lputl(o1);
    lputl(o2);
    lputl(o3);
    lputl(o4);
    lputl(o5);
    break;
case 24:
    ⟨asmout() when 6 generated instructions, debug 219g⟩
    lputl(o1);
    lputl(o2);
    lputl(o3);
    lputl(o4);
    lputl(o5);
    lputl(o6);
    break;
default:
    ⟨asmout() when other size, debug 219h⟩
    break;
```

Uses lputl() 104d.

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()`[239j]:

⟨*global* `cbp` 104a⟩≡ (282a)
```
// array<byte> (slice of buf.obuf)
char* cbp;
```

⟨*global* `cbc` 104b⟩≡ (282a)
```
// remaining bytes in buf.obuf
int cbc;
```

⟨`main()` *initialize globals(arm)* 104c⟩+≡ (34e) ◁98c
```
cbp = buf.obuf;
cbc = sizeof(buf.obuf);
```

Then, this buffer gets *flushed* once in a while in `cflush()`[104e], which finally outputs the data in the executable via the global `cout`[34d]:

⟨*function* `lputl` *(arm)* 104d⟩≡ (282a)
```
void
lputl(long l)
{

    cbp[3] = l>>24;
    cbp[2] = l>>16;
    cbp[1] = l>>8;
    cbp[0] = l;
    cbp += 4;
    cbc -= 4;
    if(cbc <= 0)
        cflush();
}
```
Uses `cbc` 104b, `cbp` 104a, and `cflush()` 104e.

⟨*function* `cflush` 104e⟩≡ (282a)
```
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` 227b, `cbc` 104b, `cbp` 104a, and `cout` 34d.

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()`[104f], 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)* 104f⟩≡ (282a)
```
void
lput(long l)
{

    cbp[0] = l>>24;
    cbp[1] = l>>16;
    cbp[2] = l>>8;
```

```
    cbp[3] = l;
    cbp += 4;
    cbc -= 4;
    if(cbc <= 0)
        cflush();
}
```

Uses `cbc` 104b, `cbp` 104a, and `cflush()` 104e.

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 cases of the first `switch` in `asmout()`[102e] will rely on a set of temporary variables to store registers and symbols:

⟨`asmout()` *other locals* 106a⟩≡ (102e) ?? ▷
```
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`[94a] 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`[95b] 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`[33b], so that error messages can mention in which procedure the problematic instruction is. No code is generated for `ATEXT`:

⟨`optab` *entries* 106b⟩≡ (94a) 107a ▷
```
{ ATEXT, C_LEXT, C_NONE, C_LCON,   0, 0 },
{ ATEXT, C_LEXT, C_REG, C_LCON,   0, 0 },
```

⟨`buildop()` *switch opcode r for ranges cases* 106c⟩+≡ (98d) ◁99a 107b ▷
```
case ATEXT:
    break;
```

⟨`asmout()` *switch on type cases* 106d⟩≡ (102e) 107c ▷
```
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* 107a⟩+≡ (94a) ◁106b 113d▷
```
{ AWORD, C_NONE, C_NONE, C_LCON,  11, 4 },
```
Uses C_LCON 107e, C_LEXT 110b, and C_REG 95c.

Remember than C_LCON[107e] is a general operand class and so the single rule above will also match instructions with operands having (sub)classes such as C_RCON[107e] or C_NCON[107e].

⟨buildop() *switch opcode r for ranges cases* 107b⟩+≡ (98d) ◁106c 113a▷
```
case AWORD:
    break;
```

⟨asmout() *switch on type cases* 107c⟩+≡ (102e) ◁106d 114a▷
```
case 11: /* word */
    c = aclass(&p->to);
    ⟨asmout() in AWORD case, when dlm 174e⟩
    o1 = instoffset;
    break;
```
Uses aclass() and instoffset 107d.

In addition to returning the class of an operand, aclass()[96a] has also for side effect to modify the global instoffset[107d], 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()[102e] 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 5l. Those classes are referenced in many of the code generation rules. I will also finish to present the code of aclass()[96a]. As said before, aclass() also modifies the following global, which is then used extensively in asmout()[102e]:

⟨*global* instoffset*(arm)* 107d⟩≡ (276b)
```
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* 107e⟩≡ (95c)
```
C_RCON,     /* 0xff rotated */ // [0..0xff] range, possibly rotated
C_NCON,     /* ~RCON */
C_LCON,
```

Remember from Section 8.2.2 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* 108a⟩+≡ (96a) ◁96b 109b▷

```
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_BRANCH` 95c, `C_RCON` 107e, `immrot()` 108b, and `instoffset` 107d.

I can now define what is a rotatable small number and see the code of `immrot()`[108b]. 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 = 0b00000000
255   => rotation = 0b0000; number = 0b11111111
256   => rotation = 0b1100; number = 0b00000001
512   => rotation = 0b1100; number = 0b00000010
2^31 => rotation = 0b0001; number = 0b00000010
```

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)* 108b⟩≡ (282c)

```
// 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 bitshiting 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* 109a⟩≡ (95c)
```
⟨Operand_class cases, in C_xOREG, half case 202l⟩
⟨Operand_class cases, in C_xOREG, float cases 184h⟩
C_SOREG,
C_LOREG,

C_ROREG,
C_SROREG,   /* both S and R */
```

⟨aclass() *switch type cases* 109b⟩+≡ (96a) ◁108a 111g▷
```
case D_OREG:
    switch(a->symkind) {
    ⟨aclass() D_OREG case, switch symkind cases 109c⟩
    }
    return C_GOK;
```
Uses C_LAUTO 111b and C_LCON 107e.

⟨aclass() D_OREG *case, switch symkind cases* 109c⟩≡ (109b) 110c▷
```
case N_NONE:
    instoffset = a->offset;
    t = immaddr(instoffset);
    if(t) {
        ⟨aclass() if immfloat for N_NONE symbol 184i⟩
        ⟨aclass() if immhalf for N_NONE symbol 203a⟩
        if(immrot(instoffset))
            return C_SROREG;
        return C_SOREG;
    }
    if(immrot(instoffset))
        return C_ROREG;
    return C_LOREG;
```
Uses C_HOREG 202l, C_SOREG 109a, C_SROREG 109a, immrot() 108b, and instoffset 107d.

⟨cmp() *switch on a, the operand class in optab rule, cases* 109d⟩+≡ (98a) ◁98b 110d▷
```
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 184h, C_ROREG 109a, C_SOREG 109a, C_SROREG 109a, and cmp() 98a.

⟨*function* `immaddr` *(arm)* 110a⟩≡ (282c)
```
// ulong -> option<long> (None = 0)
long
immaddr(long v)
{
    if(v >= 0 && v <= 0xfff)
        return (v & 0xfff) |
            (1<<24) | /* pre indexing */
            (1<<23); /* up */

    if(v < 0 && v >= -0xfff)
        return (-v & 0xfff) |
            (1<<24); /* pre indexing */
    return 0;
}
```

9.3.3 D_OREG with globals, C_xEXT

⟨`Operand_class C_xEXT` *cases* 110b⟩≡ (95c)
```
⟨Operand_class cases, in C_xEXT, half case 202f⟩
⟨Operand_class cases, in C_xEXT, float cases 184d⟩
C_SEXT,
C_LEXT,
```

⟨`aclass() D_OREG` *case, switch symkind cases* 110c⟩+≡ (109b) ◁109c 111d▷
```
case N_EXTERN:
case N_INTERN:
    ⟨aclass() D_OREG case, N_EXTERN case, sanity check a 110e⟩
    s = a->sym;
    t = s->type;
    ⟨aclass() D_OREG case, N_EXTERN case, sanity check t 111a⟩
    ⟨aclass() when D_OREG and external symbol and dlm 174b⟩
    instoffset = s->value + a->offset - BIG;
    t = immaddr(instoffset);
    if(t) {
        ⟨aclass() if immfloat for N_EXTERN symbol 184e⟩
        ⟨aclass() if immhalf for N_EXTERN symbol 202g⟩
        return C_SEXT;
    }
    return C_LEXT;
```
Uses `BIG`, `C_HEXT` 202f, `C_LOREG` 109a, `C_SEXT` 110b, and `instoffset` 107d.

⟨`cmp()` *switch on a, the operand class in optab rule, cases* 110d⟩+≡ (98a) ◁109d 111c▷
```
case C_SEXT:
    return cmp(C_HFEXT, b);
case C_LEXT:
    return cmp(C_SEXT, b);
```
Uses `C_HFEXT` 184d, `C_SEXT` 110b, `C_SROREG` 109a, and `cmp()` 98a.

⟨`aclass() D_OREG` *case,* `N_EXTERN` *case, sanity check a* 110e⟩≡ (110c)
```
if(a->sym == nil || a->sym->name == nil) {
    print("null sym external\n");
    print("%D\n", a);
    return C_GOK;
}
```

⟨aclass() D_OREG *case,* N_EXTERN *case, sanity check t* 111a⟩≡ (110c)
```
if(t == SNONE || t == SXREF) {
    diag("undefined external: %s in %s", s->name, TNAME);
    s->type = SDATA;
}
```
Uses SNONE 29d, SXREF, TNAME 272, and diag() 222d.

9.3.4 D_OREG with automatics, C_xAUTO

⟨Operand_class C_xAUTO *cases* 111b⟩≡ (95c)
```
⟨Operand_class cases, in C_xAUTO, half case 202i⟩
⟨Operand_class cases, in C_xAUTO, float cases 184f⟩
C_SAUTO,    /* -0xfff to 0xfff */
C_LAUTO,
```

⟨cmp() *switch on a, the operand class in optab rule, cases* 111c⟩+≡ (98a) ◁110d 112d▷
```
case C_SAUTO:
    return cmp(C_HFAUTO, b);
case C_LAUTO:
    return cmp(C_SAUTO, b);
```
Uses C_HFAUTO 184f, C_SAUTO 111b, C_SEXT 110b, and cmp() 98a.

⟨aclass() D_OREG *case, switch symkind cases* 111d⟩+≡ (109b) ◁110c 111e▷
```
case N_LOCAL:
    instoffset = autosize + a->offset;
    t = immaddr(instoffset);
    if(t){
        ⟨aclass() if immfloat for N_LOCAL or N_PARAM symbol 184g⟩
        ⟨aclass() if immhalf for N_LOCAL or N_PARAM symbol 202k⟩
        return C_SAUTO;
    }
    return C_LAUTO;
```
Uses C_HAUTO 202i, C_LEXT 110b, C_SAUTO 111b, autosize 60d, and instoffset 107d.

⟨aclass() D_OREG *case, switch symkind cases* 111e⟩+≡ (109b) ◁111d
```
case N_PARAM:
    instoffset = autosize + 4L + a->offset;
    t = immaddr(instoffset);
    if(t){
        ⟨aclass() if immfloat for N_LOCAL or N_PARAM symbol 184g⟩
        ⟨aclass() if immhalf for N_LOCAL or N_PARAM symbol 202k⟩
        return C_SAUTO;
    }
    return C_LAUTO;
```
Uses C_HAUTO 202i, C_LAUTO 111b, C_SAUTO 111b, autosize 60d, and instoffset 107d.

9.3.5 D_ADDR, C_xxCON

⟨Operand_class C_xxCON *cases* 111f⟩≡ (95c) 112c▷
```
C_RECON,
```

⟨aclass() *switch type cases* 111g⟩+≡ (96a) ◁109b 116c▷
```
case D_ADDR:
    switch(a->symkind) {
    ⟨aclass() D_ADDR case, switch symkind cases 112a⟩
    }
    return C_GOK;
```
Uses C_GOK 95c, C_LACON 112c, C_RACON 112c, immrot() 108b, and instoffset 107d.

⟨aclass() D_ADDR *case, switch symkind cases* 112a⟩≡ (111g) 112e▷
```
case N_EXTERN:
case N_INTERN:
    s = a->sym;
    ⟨aclass() D_ADDR case, N_EXTERN case, sanity check s 112b⟩
    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 170e⟩
        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 107e, C_RECON 111f, INITDAT 36g, SDATA, SNONE 29d, SSTRING 32c, STEXT 150b, SUNDEF 29d, SXREF, TNAME 272, diag() 222d, immrot() 108b, and instoffset 107d.

⟨aclass() D_ADDR *case,* N_EXTERN *case, sanity check s* 112b⟩≡ (112a)
```
if(s == S) // no warning?
    break;
```

⟨Operand_class C_xxCON *cases* 112c⟩+≡ (95c) ◁111f
```
C_RACON,
C_LACON,
```

⟨cmp() *switch on a, the operand class in optab rule, cases* 112d⟩+≡ (98a) ◁111c 202h▷
```
case C_LACON:
    if(b == C_RACON)
        return true;
    break;
```
Uses C_LACON 112c, C_RACON 112c, C_SAUTO 111b, and cmp() 98a.

⟨aclass() D_ADDR *case, switch symkind cases* 112e⟩+≡ (111g) ◁112a 112f▷
```
case N_LOCAL:
    instoffset = autosize + a->offset;
    goto aconsize;
```

⟨aclass() D_ADDR *case, switch symkind cases* 112f⟩+≡ (111g) ◁112e 112g▷
```
case N_PARAM:
    instoffset = autosize + a->offset + 4L;
    goto aconsize;
```

⟨aclass() D_ADDR *case, switch symkind cases* 112g⟩+≡ (111g) ◁112f
```
aconsize:
    return immrot(instoffset)? C_RACON : C_LACON;
```

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 8.1.2, `AADD` is made the representant of many other opcodes in the `optab`[94a] rules thanks to the following code:

⟨`buildop()` *switch opcode r for ranges cases* 113a⟩+≡ (98d) ◁107b 113b▷

```
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`.

The same is true for `ACMP` which represents also `ATST`, `ATEQ`, and `ACMN`:

⟨`buildop()` *switch opcode r for ranges cases* 113b⟩+≡ (98d) ◁113a 113c▷

```
case ACMP:
    oprange[ATST] = oprange[r];
    oprange[ATEQ] = oprange[r];
    oprange[ACMN] = oprange[r];
    break;
```

Uses `oprange`.

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* 113c⟩+≡ (98d) ◁113b 117a▷

```
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* 113d⟩+≡ (94a) ◁107a 115c▷

```
//          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_LCON` 107e, `C_NONE` 95c, and `C_REG` 95c.

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* 114a⟩+≡ (102e) ◁107c 116a▷
```
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 114c⟩
    o1 |= (r<<16) | (rt<<12) | rf;
    break;
```
Uses `oprrr()` 114d.

You will see `oprrr()`[114d] 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 endup as bits 0 to 3 in `o1`[2], the "to" register `rt`, which will endup as bits 12 to 15 in `o1` hence the `rt<<12` above, and the middle register `r`, which will endup 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()`[102e]:

⟨`asmout()` *adjust* `r` 114b⟩≡ (193b 191a 120a 119c 118c 117 114c)
```
if(r == R_NONE) // ADD FROM, TO ==> ADD FROM, TO, TO
    r = rt;
```

⟨`asmout()` *adjust* `r` *and* `rt` 114c⟩≡ (116 114a)
```
if(p->to.type == D_NONE)
    rt = 0;
if(p->as == AMOVW || p->as == AMVN)
    r = 0;
else
  ⟨asmout() adjust r 114b⟩
```

As mentioned above, `oprrr()` returns a partial instruction, to be completed later, with the conditional execution bits and opcode bits set according to its two parameters:

⟨*function* `oprrr` *(arm)* 114d⟩≡ (281)
```
long
oprrr(int a, int sc)
{
    long o;

    // bits 28 to 32
    o = (sc&C_SCOND) << 28;
    ⟨oprrr() sign bit handling 115a⟩
    ⟨oprrr() sanity check sc 115b⟩
```

[1] See the ASSEMBLER book [Pad15a] or EMULATOR book [Pad15b].

[2] Remember from the ASSEMBLER book [Pad15a] that registers go from `R0` to `R15` so 4 bits are enough for their encoding in an instruction.

```
    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
    ⟨oprrr() switch cases 117d⟩
    }
    diag("bad rrr %d", a);
    prasm(curp);
    return 0;
}
```

Uses curp 33c, diag() 222d, and prasm() 212b.

oprrr() 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.

⟨oprrr() *sign bit handling* 115a⟩≡ (114d)
```
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.

⟨oprrr() *sanity check sc* 115b⟩≡ (114d)
```
if(sc & (C_PBIT|C_WBIT))
    diag(".P/.W on dp instruction");
```

Uses diag() 222d.

9.4.2 Small rotatable immediate constant operand

You have seen in Section 9.3.1 the different subclasses of constants, especially C_RCON[107e] for rotatable constants. Those constants can be encoded directly in the ARM instruction, hence the rule size 4 in the rules below:

⟨optab *entries* 115c⟩+≡ (94a) ◁113d 116d▷
```
{ 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 95c, C_RCON 107e, and C_REG 95c.

⟨asmout() *switch on type cases* 116a⟩+≡ (102e) ◁114a 116e▷
```
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 114c⟩
    o1 |= (r<<16) | (rt<<12);
    break;
```
Uses aclass(), immrot() 108b, instoffset 107d, and oprrr() 114d.

Remember from Section 9.3 that the call to `aclass()`[96a] above has for side effect to set the global `instoffset`[107d] which can then be passed to `immrot()`[108b].

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: `i`mmediate (constant) or `s`hifted-register or `r`egister. You have seen the ARM code generation rules for the register (`C_REG`[95c]) and immediate constant (`C_RCON`[107e]) cases in the previous sections. I can now show the rules for the bitshifted register (`C_SHIFT`):

⟨Operand_class *cases* 116b⟩≡ (95c) 174a▷
```
C_SHIFT,   // D_SHIFT
```

⟨aclass() *switch type cases* 116c⟩+≡ (96a) ◁111g 184b▷
```
case D_SHIFT:
    return C_SHIFT;
```
Uses C_GOK 95c.

⟨optab *entries* 116d⟩+≡ (94a) ◁115c 117b▷
```
{ 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 95c, C_RCON 107e, C_REG 95c, and C_SHIFT 116b.

⟨asmout() *switch on type cases* 116e⟩+≡ (102e) ◁116a 117c▷
```
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 114c⟩
    o1 |= (r<<16) | (rt<<12);
    break;
```
Uses aclass() and oprrr() 114d.

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 bitshited registers, `5a` supports the more traditional bitshift instructions `SLL`, `SRL`, and `SRA`:

⟨`buildop()` *switch opcode r for ranges cases* 117a⟩+≡ (98d) ◁113c 119a▷
```
case ASLL:
    oprange[ASRL] = oprange[r];
    oprange[ASRA] = oprange[r];
    break;
```
Uses `oprange`.

⟨`optab` *entries* 117b⟩+≡ (94a) ◁116d 117e▷
```
{ ASLL,  C_RCON, C_REG, C_REG,   8, 4 },
{ ASLL,  C_RCON, C_NONE, C_REG,   8, 4 },
```
Uses `C_NONE` 95c, `C_REG` 95c, and `C_SHIFT` 116b.

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* 117c⟩+≡ (102e) ◁116e 117f▷
```
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 114b⟩
    o1 |= (rt<<12) | ((instoffset&31) << 7) | r;
    break;
```
Uses `aclass()`, `instoffset` 107d, and `oprrr()` 114d.

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 `oprrr()`[114d]:

⟨`oprrr()` *switch cases* 117d⟩≡ (114d) 119d▷
```
//                      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* 117e⟩+≡ (94a) ◁117b 118a▷
```
{ ASLL,  C_REG, C_NONE, C_REG,   9, 4 },
{ ASLL,  C_REG, C_REG, C_REG,   9, 4 },
```
Uses `C_NONE` 95c, `C_RCON` 107e, and `C_REG` 95c.

⟨`asmout()` *switch on type cases* 117f⟩+≡ (102e) ◁117c 118b▷
```
case 9:  /* sll R,[R],R -> mov (R<<R),R */
    o1 = oprrr(p->as, p->scond);
    rf = p->from.reg;
    rt = p->to.reg;
    r = p->reg;
    ⟨asmout() adjust r 114b⟩
    o1 |= (rt<<12) | (rf<<8) | (1<<4) | r ;
    break;
```
Uses `oprrr()` 114d.

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* 118a⟩+≡ (94a) ◁117e 119b▷
```
{ 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 95c and C_REG 95c.

`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* 118b⟩+≡ (102e) ◁117f 118c▷
```
case 14: /* movb/movbu/movh/movhu R,R */
    o1 = oprrr(ASLL, p->scond);

    if(p->as == AMOVBU || p->as == AMOVHU)
        o2 = oprrr(ASRL, p->scond);
    else
        o2 = oprrr(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 oprrr() 114d.

`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* 118c⟩+≡ (102e) ◁118b 119c▷
```
case 58: /* movbu R,R -> AND $0xff, R, R */
    o1 = oprrr(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 114b⟩
    o1 |= (r<<16) | (rt<<12);
    break;
```
Uses immrot() 108b and oprrr() 114d.

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* 119a⟩+≡ (98d) ◁117a 122a▷
```
case AMUL:
    oprange[AMULU] = oprange[r];
    break;
```

⟨`optab` *entries* 119b⟩+≡ (94a) ◁118a 119f▷
```
{ AMUL,  C_REG, C_REG, C_REG,  15, 4 },
{ AMUL,  C_REG, C_NONE, C_REG,  15, 4 },
```
Uses `C_NONE` 95c and `C_REG` 95c.

⟨`asmout()` *switch on type cases* 119c⟩+≡ (102e) ◁118c 120a▷
```
case 15: /* mul r,[r,]r */
    o1 = oprrr(p->as, p->scond);
    rf = p->from.reg;
    rt = p->to.reg;
    r = p->reg;
    ⟨asmout() adjust r 114b⟩
    ⟨asmout() adjust registers when mul 119e⟩
    o1 |= (rt<<16) | (rf<<8) | r;
    break;
```
Uses `oprrr()` 114d.

⟨`oprrr()` *switch cases* 119d⟩+≡ (114d) ◁117d 134c▷
```
case AMULU:
case AMUL: return o | (0x0<<21) | (0x9<<4);
```

⟨`asmout()` *adjust registers when mul* 119e⟩≡ (119c)
```
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`[107e]) or a negative rotatable constant (`C_NCON`[107e]) as a first operand. Those instructions are more complex to handle than the instructions with a rotatable constant (`C_RCON`[107e]) 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* 119f⟩+≡ (94a) ◁119b 121a▷
```
{ 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` 107e, `C_NONE` 95c, and `C_REG` 95c.

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 `5l` in the two following instructions:

```
MVN $0xff, R11  // o1
ADD R11, R2, R3 // o2
```

⟨asmout() *switch on type cases* 120a⟩+≡ (102e) ◁119c 121e▷

```
case 13: /* op $lcon, [R], R */
    o1 = omvl(p, &p->from, REGTMP);
    ⟨asmout() sanity check o1 120d⟩

    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 114b⟩
    o2 |= (r<<16) | rf;
    if(p->to.type != D_NONE)
        o2 |= rt << 12;
    break;
```

Uses omvl() 120b and oprrr() 114d.

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)* 120b⟩≡ (281)

```
long
omvl(Prog *p, Adr *a, int dr)
{
    long v, o1;

    ⟨omvl() when C_LCON case 121c⟩
    else {
        // C_NCON case
        aclass(a);
        v = immrot(~instoffset);
        ⟨omvl() sanity check v 120c⟩
        o1 = oprrr(AMVN, p->scond&C_SCOND);
        o1 |= (dr<<12) | v;
    }
    return o1;
}
```

Uses aclass(), immrot() 108b, instoffset 107d, and oprrr() 114d.

⟨omvl() *sanity check v* 120c⟩≡ (120b)

```
if(v == 0) {
    diag("missing literal");
    prasm(p);
    return 0;
}
```

Uses diag() 222d and prasm() 212b.

⟨asmout() *sanity check o1* 120d⟩≡ (133a 131c 130c 128d 127d 120a)

```
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`[135d].

⟨`optab` *entries* 121a⟩+≡ (94a) ◁119f 121d▷
```
{ 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 107e, C_NCON 107e, C_NONE 95c, C_REG 95c, and LFROM 135d.

This flag is stored in an extra field of `Optab`[93b]:

⟨`Optab` *flag field* 121b⟩≡ (93b)
```
// 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()`[89b] calls `addpool()`[136a] 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* 121c⟩≡ (120b)
```
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()` 126b.

You will see `olr()`[126b] 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* 121d⟩+≡ (94a) ◁121a 122c▷
```
{ AMOVW, C_NCON, C_NONE, C_REG,  12, 4 },
{ AMOVW, C_LCON, C_NONE, C_REG,  12, 4, 0, LFROM },
```
Uses C_LCON 107e, C_NONE 95c, C_REG 95c, and LFROM 135d.

⟨`asmout()` *switch on type cases* 121e⟩+≡ (102e) ◁120a 122d▷
```
case 12: /* movw $lcon, reg */
    o1 = omvl(p, &p->from, p->to.reg);
    break;
```
Uses `omvl()` 120b.

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* 122a⟩+≡ (98d) ◁119a 122b▷
```
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* 122b⟩+≡ (98d) ◁122a 125a▷
```
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.

9.5.1 Direct jumps

⟨optab *entries* 122c⟩+≡ (94a) ◁121d 124b▷
```
{ 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 95c, C_LCON 107e, C_NONE 95c, C_REG 95c, LFROM 135d, and LPOOL 135d.

The rule flag LPOOL[135d] will be explained later. You can see the important use of Instr.condX set by patch()[77] in the code below to find the code address of the target instruction:

⟨asmout() *switch on type cases* 122d⟩+≡ (102e) ◁121e 124c▷
```
case 5:  /* bra s */
    ⟨asmout() BRA case, if undefined target 170a⟩
    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) & 0xffffff;
    break;
```
Uses P 32f and opbra() 123a.

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.
- `0xffffff`: 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)* 123a⟩≡ (281)
```
long
opbra(int a, int sc)
{

    ⟨opbra() sanity check sc 123b⟩
    sc &= C_SCOND;
    if(a == ABL)
        return (sc<<28)|(0x5<<25)|(0x1<<24);
    ⟨opbra() sanity check sc if not ABL 124a⟩

    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` 33c, `diag()` 222d, and `prasm()` 212b.

⟨`opbra()` *sanity check sc* 123b⟩≡ (123a)
```
if(sc & (C_SBIT|C_PBIT|C_WBIT))
    diag(".S/.P/.W on bra instruction");
```
Uses `diag()` 222d.

⟨opbra() *sanity check sc if not ABL* 124a⟩≡ (123a)
```
if(sc != 0xe)
    diag("redundant .EQ/.NE/... on B/BEQ/BNE/...  instruction");
```
Uses diag() 222d.

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()`[49b] 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* 124b⟩+≡ (94a) ◁122c 125b▷
```
{ AB,  C_NONE, C_NONE, C_ROREG,  6, 4, 0, LPOOL },
{ ABL,  C_NONE, C_NONE, C_ROREG,  7, 8 },
```
Uses C_BRANCH 95c and C_NONE 95c.

⟨asmout() *switch on type cases* 124c⟩+≡ (102e) ◁122d 124d▷
```
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(), immrot() 108b, instoffset 107d, and oprrr() 114d.

The translation of indirect calls (`BL`) requires two ARM instructions:

⟨asmout() *switch on type cases* 124d⟩+≡ (102e) ◁124c 125c▷
```
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(), immrot() 108b, instoffset 107d, and oprrr() 114d.

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 implicitely 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* 125a⟩+≡ (98d) ◁122b 129c▷
```
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_S*xxx*, e.g., C_SEXT[110b] (see the code of immaddr()[110a]). In those cases the offset can be encoded directly in the ARM instruction, hence the rule size 4 below:

⟨optab *entries* 125b⟩+≡ (94a) ◁124b 127c▷
```
{ 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 95c, C_REG 95c, C_ROREG 109a, C_SAUTO 111b, C_SEXT 110b, and C_SOREG 109a.

⟨asmout() *switch on type cases* 125c⟩+≡ (102e) ◁124d 127d▷
```
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 126a⟩
    o1 = olr(p->as, p->scond, instoffset, r, rt);
    break;
```
Uses aclass(), instoffset 107d, and olr() 126b.

Remember that aclass()[96a] above modifies the global instoffset[107d] 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[93b] can be used to contain extra parameters:

⟨Optab *param field* 125d⟩≡ (93b)
```
// 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()`[102e] to adjust the base register:

⟨`asmout()` *adjust maybe r to rule param* 126a⟩≡ (204 190 189b 133a 132b 131 130 128 127d 125c)
```
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)* 126b⟩≡ (281)
```
long
olr(int a, int sc, long v, int b, int rt)
{
    long o;

    o = (sc&C_SCOND) << 28;
    ⟨olr() sanity check sc 126d⟩

    ⟨olr() set special bits 126c⟩
    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 127a⟩
    switch(a) {
    case AMOVB:
    case AMOVBU:
        o |= 1<<22;
        break;
    case AMOVW:
        break;
    default:
        ⟨olr() sanity check a 127b⟩
    }
    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* 126c⟩≡ (126b)
```
if(!(sc & C_PBIT))
    o |= 1 << 24; // pre (not post)
if(sc & C_WBIT)
    o |= 1 << 21; // write back
```

⟨`olr()` *sanity check sc* 126d⟩≡ (126b)
```
if(sc & C_SBIT)
    diag(".S on LDR/STR instruction");
if(sc & C_UBIT)
    diag(".U on LDR/STR instruction");
```
Uses `diag()` 222d.

⟨olr() *sanity check offset* v 127a⟩≡ (126b)
```
if(v >= (1<<12))
    diag("literal span too large: %ld (R%d)\n%P", v, b, curp);
```
Uses curp 33c and diag() 222d.

⟨olr() *sanity check* a 127b⟩≡ (126b)
```
diag("expect move operation, not: %P", curp);
```
Uses curp 33c and diag() 222d.

The use of large offsets (C_L*xxx*) requires an additional instruction to first load the offset in REGTMP:

⟨optab *entries* 127c⟩+≡ (94a) ◁125b 127f▷
```
{ 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 111b, C_LEXT 110b, C_LOREG 109a, C_NONE 95c, C_REG 95c, C_SOREG 109a, and LFROM 135d.

The code below uses also omvl()[120b] which you saw in Section 9.8 for arithmetic instructions. Remember than omvl() internally calls olr()[126b]. 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* 127d⟩+≡ (102e) ◁125c 128a▷
```
case 31: /* mov/movbu L(R),R */
    o1 = omvl(p, &p->from, REGTMP);
    ⟨asmout() sanity check o1 120d⟩

    rt = p->to.reg;
    r = p->from.reg;
    ⟨asmout() adjust maybe r to rule param 126a⟩
    o2 = olrr(p->as, p->scond, REGTMP, r, rt);
    break;
```
Uses olrr() 127e and omvl() 120b.

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)* 127e⟩≡ (281)
```
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() 126b.

9.6.2 Store

MOVW can also 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* 127f⟩+≡ (94a) ◁127c 128c▷
```
{ 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_LOREG 109a, C_NONE 95c, C_REG 95c, C_SAUTO 111b, C_SEXT 110b, C_SOREG 109a, and LFROM 135d.

⟨asmout() *switch on type cases* 128a⟩+≡ (102e) ◁127d 128d▷

```
  case 20: /* mov/movb/movbu R,O(R) */
      aclass(&p->to);
      rf = p->from.reg;
      r = p->to.reg;
      ⟨asmout() adjust maybe r to rule param 126a⟩
      o1 = osr(p->as, p->scond, rf, instoffset, r);
      break;
```

Uses aclass(), instoffset 107d, and osr() 128b.

⟨*function* osr*(arm)* 128b⟩≡ (281)

```
  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() 126b.

Note that osr()[128b] is a small wrapper around olr()[126b] as the instruction format for LDR and STR are very similar except for the bit 20.

⟨optab *entries* 128c⟩+≡ (94a) ◁127f 129a▷

```
  { 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 111b, C_LEXT 110b, C_LOREG 109a, C_NONE 95c, C_REG 95c, C_SOREG 109a, and LTO 135d.

⟨asmout() *switch on type cases* 128d⟩+≡ (102e) ◁128a 129e▷

```
  case 30: /* mov/movb/movbu R,L(R) */
      o1 = omvl(p, &p->to, REGTMP);
      ⟨asmout() sanity check o1 120d⟩

      rf = p->from.reg;
      r = p->to.reg;
      ⟨asmout() adjust maybe r to rule param 126a⟩
      o2 = osrr(p->as, p->scond, rf, REGTMP, r);
      break;
```

Uses omvl() 120b and osrr() 128e.

⟨*function* osrr*(arm)* 128e⟩≡ (281)

```
  long
  osrr(int a, int sc, int r, int i, int b)
  {

      return olrr(a, sc, i, b, r) ^ (1<<20); // STR
  }
```

Uses olrr() 127e.

The previous cases work also for AMOVB:

⟨optab *entries* 129a⟩+≡ (94a) ◁128c 129b▷
```
{ 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_LOREG 109a, C_NONE 95c, C_REG 95c, C_SEXT 110b, and LTO 135d.

⟨optab *entries* 129b⟩+≡ (94a) ◁129a 129d▷
```
{ 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_LEXT 110b, C_NONE 95c, C_REG 95c, C_SOREG 109a, and LTO 135d.

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* 129c⟩+≡ (98d) ◁125a 133b▷
```
case ASWPW:
    oprange[ASWPBU] = oprange[r];
    break;
```

⟨optab *entries* 129d⟩+≡ (94a) ◁129b 129g▷
```
{ ASWPW, C_SOREG,C_REG, C_REG,  40, 4 },
```
Uses C_LOREG 109a, C_NONE 95c, C_REG 95c, and LTO 135d.

⟨asmout() *switch on type cases* 129e⟩+≡ (102e) ◁128d 130a▷
```
case 40: /* swp oreg,reg,reg */
    aclass(&p->from);
    ⟨asmout() sanity check instoffset for SWP 129f⟩
    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().

⟨asmout() *sanity check instoffset for SWP* 129f⟩≡ (129e)
```
if(instoffset != 0)
    diag("offset must be zero in SWP");
```
Uses diag() 222d and instoffset 107d.

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* 129g⟩+≡ (94a) ◁129d 130b▷
```
{ AMOVW, C_RECON,C_NONE, C_REG,   4, 4, REGSB },
{ AMOVW, C_RACON,C_NONE, C_REG,   4, 4, REGSP },
```
Uses C_REG 95c and C_SOREG 109a.

⟨asmout() *switch on type cases* 130a⟩+≡ (102e) ◁129e 130c▷
```
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 126a⟩
    o1 |= (r<<16) | (rt<<12) | immrot(instoffset);
    break;
```
Uses aclass(), immrot() 108b, instoffset 107d, and oprrr() 114d.

The address of an entity which is not a rotatable offset must be loaded from a literal pool with omvl()[120b]:

⟨optab *entries* 130b⟩+≡ (94a) ◁129g 130d▷
```
{ AMOVW, C_LACON,C_NONE, C_REG,  34, 8, REGSP, LFROM },
```
Uses C_NONE 95c, C_RACON 112c, and C_REG 95c.

⟨asmout() *switch on type cases* 130c⟩+≡ (102e) ◁130a 131a▷
```
case 34: /* mov $lacon,R -> LDR x(R15), R11; ADD R11, R13, R */
    o1 = omvl(p, &p->from, REGTMP);
    ⟨asmout() sanity check o1 120d⟩

    o2 = oprrr(AADD, p->scond);
    rf = REGTMP;
    rt = p->to.reg;
    r = p->from.reg;
    ⟨asmout() adjust maybe r to rule param 126a⟩
    o2 |= (r<<16) | (rt<<12) | rf;
    break;
```
Uses omvl() 120b and oprrr() 114d.

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 5l 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* 130d⟩+≡ (94a) ◁130b 131b▷
```
{ 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_LACON 112c, C_NONE 95c, C_REG 95c, C_SAUTO 111b, C_SEXT 110b, C_SOREG 109a, and LFROM 135d.

`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* 131a⟩+≡ (102e) ◁130c 131c▷

```
case 22: /* movb/movh/movhu O(R),R -> lr,shl,shr */
    aclass(&p->from);
    rt = p->to.reg;
    r = p->from.reg;
    ⟨asmout() adjust maybe r to rule param 126a⟩
    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()`, `instoffset` 107d, `olr()` 126b, and `oprrr()` 114d.

Note that the rules for `AMOVB` and loading are in this section while the rules for `AMOVB` 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* 131b⟩+≡ (94a) ◁130d 132a▷

```
{ 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` 111b, `C_LEXT` 110b, `C_LOREG` 109a, `C_NONE` 95c, `C_REG` 95c, `C_SOREG` 109a, and `LFROM` 135d.

⟨asmout() *switch on type cases* 131c⟩+≡ (102e) ◁131a 132b▷

```
case 32: /* movh/movb L(R),R */
    o1 = omvl(p, &p->from, REGTMP);
    ⟨asmout() sanity check o1 120d⟩

    rt = p->to.reg;
```

```
r = p->from.reg;
⟨asmout() adjust maybe r to rule param 126a⟩
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() 127e, omvl() 120b, and oprrr() 114d.

Store

⟨optab *entries* 132a⟩+≡ (94a) ◁131b 132c▷

```
{ 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_LOREG 109a, C_NONE 95c, C_REG 95c, C_SAUTO 111b, C_SEXT 110b, C_SOREG 109a, and LFROM 135d.

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* 132b⟩+≡ (102e) ◁131c 133a▷

```
case 23: /* movh/movhu R,O(R) -> sb,sb */
    aclass(&p->to);
    rf = p->from.reg;
    r = p->to.reg;
    ⟨asmout() adjust maybe r to rule param 126a⟩
    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(), instoffset 107d, oprrr() 114d, and osr() 128b.

Note the rule size of 24 below for the translation of AMOVH using a large offset:

⟨optab *entries* 132c⟩+≡ (94a) ◁132a 134a▷

```
{ 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 111b, C_LEXT 110b, C_LOREG 109a, C_NONE 95c, C_REG 95c, C_SOREG 109a, and LTO 135d.

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* 133a⟩+≡ (102e) ◁132b 134b▷

```
  case 33: /* movh/movhu R,L(R) -> sb, sb */
      o1 = omvl(p, &p->to, REGTMP);
      ⟨asmout() sanity check o1 120d⟩

      rf = p->from.reg;
      r = p->to.reg;
      ⟨asmout() adjust maybe r to rule param 126a⟩
      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() 108b, omvl() 120b, oprrr() 114d, and osrr() 128e.

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* 133b⟩+≡ (98d) ◁129c 188b▷

```
  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* 134a⟩+≡ (94a) ◁132c 134d▷
```
{ 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_LOREG 109a, C_NONE 95c, C_REG 95c, and LTO 135d.

⟨asmout() *switch on type cases* 134b⟩+≡ (102e) ◁133a 134e▷
```
case 10: /* swi [$con] */
    o1 = oprrr(p->as, p->scond);
    if(p->to.type != D_NONE) {
        aclass(&p->to);
        o1 |= instoffset & 0xffffff;
    }
    break;
```
Uses aclass(), instoffset 107d, and oprrr() 114d.

⟨oprrr() *switch cases* 134c⟩+≡ (114d) ◁119d 190d▷
```
case ASWI: return o | (0xf<<24);
```

RFE is actually a virtual instruction which is transformed by 5l in an instruction using MOVM, the multiple register move opcode (see Section 12.6.4):

⟨optab *entries* 134d⟩+≡ (94a) ◁134a 174c▷
```
{ ARFE,  C_NONE, C_NONE, C_NONE,  41, 4 },
```
Uses C_LOREG 109a and C_NONE 95c.

⟨asmout() *switch on type cases* 134e⟩+≡ (102e) ◁134b 175a▷
```
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

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)* 135a⟩≡ (279a)
```
// list<ref_own<prog>> (next = Prog.link)
Prog* blitrl;
```

⟨*global* elitrl*(arm)* 135b⟩≡ (279a)
```
// ref<Prog> (end from = blitrl)
Prog* elitrl;
```

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()[136a]. This function is called from dotext()[89b], 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)* 135c⟩≡
```
{ 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)* 135d⟩≡ (276a)
```
enum Optab_flag {
    // Flags related to literal pools
    LFROM   = 1<<0,
    LTO     = 1<<1,

    LPOOL   = 1<<2,
    // Flags related architecture restrictions
    ⟨Optab_flag cases 184j⟩
};
```

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()[97a] 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* 135e⟩≡ (89b)
```
switch(o->flag & (LFROM|LTO|LPOOL)) {
⟨dotext() pool handling, switch flag cases 135f⟩
}
⟨dotext() pool handling, flush if MOVW REGPC 139b⟩
⟨dotext() pool handling, checkpool 137c⟩
```

⟨dotext() *pool handling, switch flag cases* 135f⟩≡ (135e) 139a▷
```
case LFROM:
    addpool(p, &p->from);
    break;
case LTO:
    addpool(p, &p->to);
    break;
```

Uses LFROM 135d, LPOOL 135d, LTO 135d, and addpool() 136a.

`addpool()` essentially adds a new `WORD` instruction in the list `blitrl`[135a] with 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)* 136a⟩≡ (279a)
```
void
addpool(Prog *p, Adr *a)
{
    Prog *q;
    Prog t;
    //enum<Operand_class>
    int c;

    c = aclass(a);

    t = zprg;
    t.as = AWORD;
    ⟨addpool() set t.to using a 136b⟩

    ⟨addpool() if literal already present in pool 137a⟩
    // else

    q = prg();
    *q = t;

    ⟨addpool() set pool.start and pool.size 138c⟩

    // add_queue(q, blitrl, elitrl)
    if(blitrl == P) {
        blitrl = q;
    } else
        elitrl->link = q;
    elitrl = q;

    // for omvl()!
    p->cond = q;
}
```
Uses `P` 32f, `aclass()`, `blitrl`, `elitrl` 279a, `pool-13` 279a, and `prg()` 39a.

Thanks to the instruction `p->cond = q` above, `omvl()`[120b] 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()`[137b] 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`[107e]), but also large offsets (`C_LOREG`[109a], `C_LAUTO`[111b]), and also with symbols which when resolved are large offsets to `R12` (`C_LEXT`[110b], `C_LACON`[112c]).

⟨`addpool()` *set t.to using a* 136b⟩≡ (136a)
```
switch(c) {
⟨addpool() switch operand class c cases 136c⟩
}
```

⟨`addpool()` *switch operand class* `c` *cases* 136c⟩≡ (136b) 136d▷
```
// C_LCON|C_xCON, C_LEXT|C_xEXT? TODO warning if other case?
default:
    t.to = *a;
    break;
```

⟨`addpool()` *switch operand class* `c` *cases* 136d⟩+≡ (136b) ◁136c
```
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 184f, C_FOREG 184h, C_LACON 112c, C_LAUTO 111b, C_LOREG 109a, C_ROREG 109a, C_SAUTO 111b, C_SOREG 109a, and C_SROREG 109a.

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* 137a⟩≡ (136a)

```
// 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 32f and blitrl.

9.8.3 flushpool()

⟨*function* flushpool*(arm)* 137b⟩≡ (279a)

```
/// (dotext -> checkpool) | dotext -> <>
void
flushpool(Prog *p, bool skip)
{
    Prog *q;

    if(blitrl) {
        ⟨flushpool() if skip or corner case 138e⟩

        //insert_list_after_elt(blitlr, 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, elitrl 279a, and pool-13 279a.

9.8.4 checkpool()

⟨dotext() *pool handling, checkpool* 137c⟩≡ (135e)

```
if(blitrl)
    checkpool(p);
```

⟨*function* `checkpool` *(arm)* 138a⟩≡ (279a)
```
void
checkpool(Prog *p)
{
    if(p->link == P)
        flushpool(p, true);
    else
    ⟨checkpool() if special condition 138d⟩
}
```
Uses `P` 32f and `flushpool()` 137b.

Another global, `pool`, is used to store additional information about the pool:

⟨*global* `pool` *(arm)* 138b⟩≡ (279a)
```
static struct {
    // PC of first instruction referencing the pool
    ulong start;
    // a multiple of 4
    ulong size;
} pool;
```
Uses `__anon_struct_2` 138b.

⟨`addpool()` *set pool.start and pool.size* 138c⟩≡ (136a)
```
// 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` 32f, `blitrl`, and `pool-13` 279a.

⟨`checkpool()` *if special condition* 138d⟩≡ (138a)
```
/*
 * 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 `pool-13` 279a.

⟨`flushpool()` *if skip or corner case* 138e⟩≡ (137b)
```
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* 139c⟩
Uses `DBG` 272, `blitrl`, `pool-13` 279a, and `prg()` 39a.

9.8.5 LPOOL

⟨dotext() *pool handling, switch flag cases* 139a⟩+≡ (135e) ◁135f
```
  case LPOOL:
      if ((p->scond&C_SCOND) == COND_ALWAYS)
          flushpool(p, false);
      break;
```
Uses LPOOL 135d.

⟨dotext() *pool handling, flush if MOVW REGPC* 139b⟩≡ (135e)
```
  // 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);
```

⟨flushpool() *else if not skip and corner case* 139c⟩≡ (138e)
```
  else if((p->pc + pool.size - pool.start) < 2048)
      return;
```

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 `asmb()` and the debugging tables

As explained in Section 4.4, the generation of the executable is done by `asmb()`[41a]. Here is the part of `asmb()` which generates the debugging sections:

⟨`asmb()` *symbol and line table sections* 140⟩≡ (41a)

```
// modified by asmsym()
symsize = 0;
// modified by asmlc()
lcsize = 0;

if(!debug['s']) {
    switch(HEADTYPE) {
    ⟨asmb() switch HEADTYPE (for symbol table generation) cases(arm) 141a⟩
    }
    DBG("%5.2f sym\n", cputime());
    asmsym();
    DBG("%5.2f pc\n", cputime());
    asmlc();
```

```
0          4     5              10   11
+-----------------------------------+
|  1020    | 'T' |    "_main"     |'\0'|
+-----------------------------------+
   value     type      name         end
                  (variable size) marker
```

Figure 10.1: Executable symbol table entry format.

```
    ⟨asmb() if dynamic module, call asmdyn() 171e⟩
    cflush();
}
else {
    ⟨asmb() if dynamic module and no symbol table generation 171a⟩
}
```

Uses DBG 272, asmdyn() 171f, asmsym() 142, cflush() 104e, lcsize 153d, and symsize 141b.

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)* 141a⟩≡ (140) 182b ▷

```
case H_PLAN9:
    OFFSET = HEADR+textsize+datsize;
    seek(cout, OFFSET, SEEK__START);
    break;
```

Uses HEADR 36d, HEADTYPE 36a, H_PLAN9 143b, datsize 87a, and textsize 89a.

The next two sections will describe the generation of the executable symbol table, via `asmsym()`[142], and the program counter and line table, via `asmlc()`[154].

10.2 Executable symbol table

One side effect of `asmsym()`[142] 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).

⟨*global* symsize 141b⟩≡ (276b)

```
long symsize;
```

10.2.1 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 `hash`[28a], 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 141c⟩≡ (285a)

```
void
putsymb(char *s, int t, long v, int ver)
```

```
{
    int i, f;

    ⟨putsymb() adjust string s if file symbol 153a⟩

    // value
    lput(v);
    // type
    if(ver)
        t += 'a' - 'A'; // lowercase(t)
    cput(t+0x80);   /* 0x80 is variable length */

    ⟨putsymb() if z or Z 153c⟩
    else {
        // name
        for(i=0; s[i]; i++)
            cput(s[i]);
        // end marker
        cput('\0');
    }
    symsize += 4 + 1 + i + 1;

    ⟨putsymb() debug 220a⟩
}
```

Uses cput() 227c and lput() 104f.

10.2.2 Globals and procedures symbols: asmsym()

I can now show the code of asmsym() which first calls putsymb()[141c] for all the data symbols in hash[28a]:

⟨function asmsym(arm) 142⟩≡ (285a)

```
/// main -> asmb -> <>
void
asmsym(void)
{
    Sym *s;
    int h;
    Prog *p;
    ⟨asmsym() other locals 146a⟩

    ⟨asmsym() generate symbol for etext 143a⟩

    // 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 152b⟩
            }

    // 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 153b⟩

            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 146b⟩
        }
    }
    if(debug['v'] || debug['n']) {
        Bprint(&bso, "symsize = %lud\n", symsize);
        Bflush(&bso);
    }
}
```

Uses INITDAT 36g, LEAF 272, NHASH, P 32f, S 29d, SBSS 164b, SDATA, bso 211c, debug 211a, putsymb() 141c, symsize 141b, and textp 145d.

`asmsym()`[142] could generate the entries for the procedures by also using `hash`. Instead, it relies on `textp`[145d] (which I will describe soon) which contains the list of all procedures. This list is a subset of the list of instructions in `firstp`[32d]. 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()`[92a] in Section 7.6 and is part of the text section (`STEXT`[150b]). 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* 143a⟩≡ (142)

```
s = lookup("etext", 0);
if(s->type == STEXT)
    putsymb(s->name, 'T', s->value, s->version);
```

Uses lookup() 28e.

10.2.3 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()`[142] 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()`[49b].

Auto

The `Auto` structure below is used to keep track of all the parameters and locals accessed by a procedure:

⟨*struct* Auto*(arm)* 143b⟩≡ (272)

```
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 144a⟩
};
```

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* 144a⟩≡ (143b)
```
// 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* 144b⟩+≡ (30a) ◁101b
```
// list<ref_own<Auto> (next = Auto.link), only used by TEXT instruction
Auto*   autom;
```

`curauto`

The set of stack variables for the current procedure (`curtext`[33b]) is stored in `curauto` and updated at loading time by `inopd()`[52d] (which is called by `ldobj()`[49b]):

⟨*global* curauto 144c⟩≡ (276b)
```
// list<ref<Auto>> (next = Auto.link)
Auto* curauto;
```

⟨inopd() *other locals* 144d⟩≡ (52d)
```
Sym *s;
// <enum<Sym_kind>>
int t;
int l;
Auto *u;
```

⟨inopd() *adjust curauto for* N_LOCAL *or* N_PARAM *symkind* 144e⟩≡ (52d)
```
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 145a⟩
    // else

    u = malloc(sizeof(Auto));
    u->asym = s;
    u->type = t;
    u->aoffset = l;

    //add_list(u, curauto)
```

```
    u->link = curauto;
    curauto = u;
}
```

Uses malloc() 226a.

⟨inopd() *return if stack variable already present in curauto* 145a⟩≡ (144e)

```
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 S 29d and curauto 144c.

Once the code of a procedure has been fully read, curauto can be transfered to the Instr.to.autom field of the TEXT instruction of the current procedure:

⟨ldobj() *case ATEXT, if curtext not null adjustments for curauto* 145b⟩≡ (60b)

```
if(curtext != P) {
    ⟨ldobj() case ATEXT, curauto adjustments with curhist 149b⟩
    curtext->to.autom = curauto;
    curauto = nil;
}
```

Uses histtoauto() 149d.

⟨ldobj() *case AEND, curauto adjustments* 145c⟩≡ (62b)

```
if(curtext != P)
    curtext->to.autom = curauto;
curauto = nil;
```

Uses histtoauto() 149d.

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 145d⟩≡ (276b)

```
// list<ref<Prog>> (next = Prog.cond)
Prog* textp = P;
```

Uses P 32f and textp 145d.

⟨*global* etextp 145e⟩≡ (276b)

```
// ref<Prog> (end from = textp)
Prog* etextp = P;
```

Uses P 32f and etextp 145e.

The procedure instructions are chained together by (ab)using Instr.condX:

⟨ldobj() *in switch opcode ATEXT case, populate textp* 145f⟩≡ (60b)

```
//add_queue(textp, etextp, p)
if(textp == P) {
    textp = p;
    etextp = p;
} else {
    etextp->cond = p;
    etextp = p;
}
```

Uses P 32f, etextp 145e, pc 49a, and textp 145d.

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`[272] to generate the symbols for the stack variables of a procedure:

⟨`asmsym()` *other locals* 146a⟩≡ (142)
```
Auto *a;
```

⟨`asmsym()` *frame symbols* 146b⟩≡ (142)
```
/* 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()` 141c.

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.

10.2.4 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 10.3.3. 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.

10.3 File and line information

Before seeing the code of `asmlc()`[154], I need first to recall how file and line information are stored in object files, and to explain the code of `ldobj()`[49b] which loads this information in memory.

10.3.1 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.

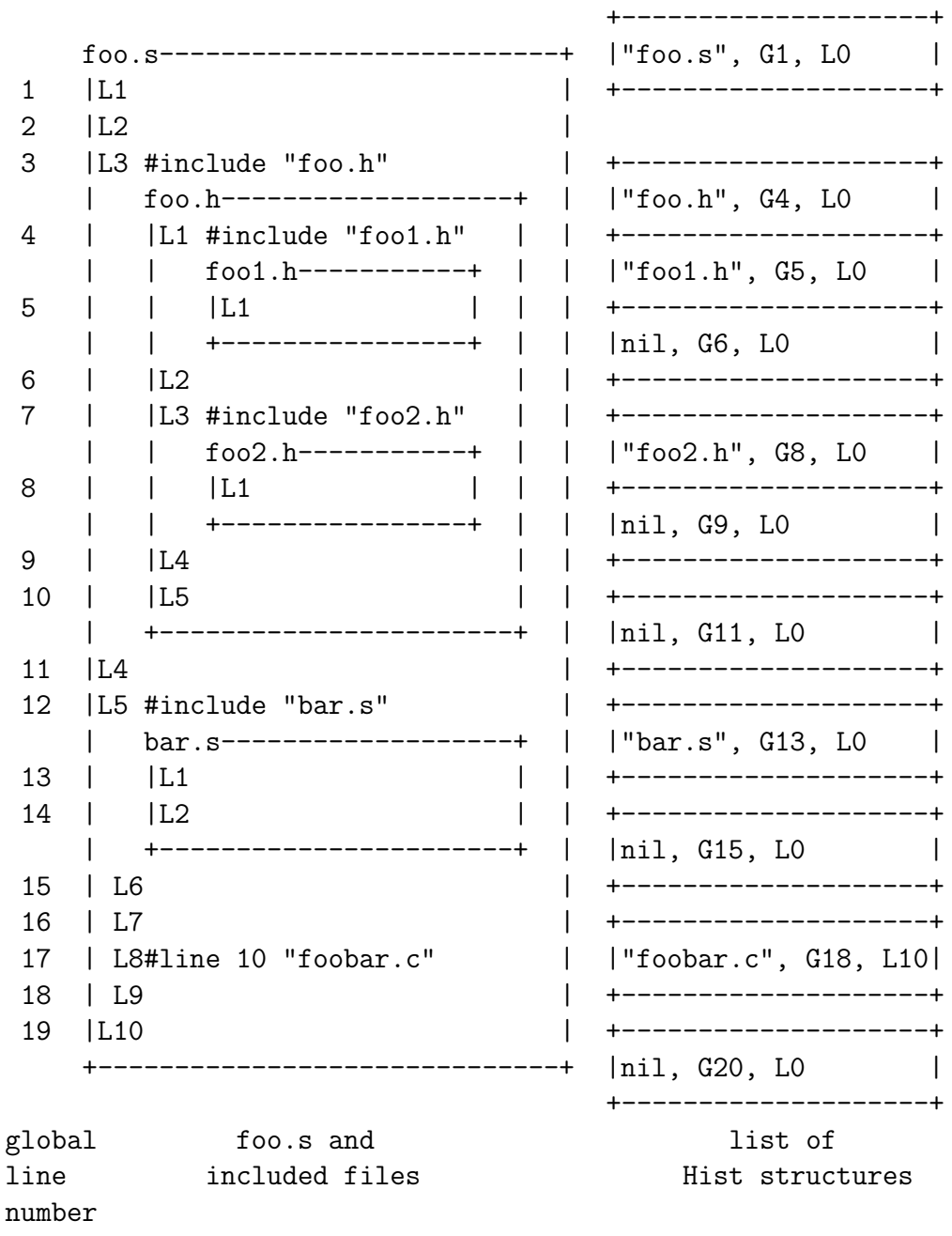


Figure 10.2: File and line information for `/tests/cpp/foo.s`.

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.

10.3.2 Locations in `5l` memory

`5a` uses the function `outhist()` to write the `Hist`s 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()`.

`AHISTORY` and `addhist`

Reading an `AHISTORY` in `ldobj()`[49b] 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)* 148a⟩+≡ (51) ◁62c 176a▷

```
case AHISTORY:
    ⟨ldobj() in AHISTORY case, if pragma lib 70c⟩
    // 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 150h⟩
    goto loop;
```

Uses `addhist()` 148b and `pc` 49a.

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()`[148b] 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`[272] which is then stored in the global `curhist`[149a]. As you will see soon, `curhist` is then copied in `curauto`[144c] 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` 148b⟩≡ (285b)

```
void
addhist(long line, int type)
{
    Auto *u;
    Sym *s;
    ⟨addhist() other locals 151b⟩
```

```
    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 151c⟩
}
```

Uses curhist 149a and malloc() 226a.

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 10.2.1), 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[272] created by addhist()[148b]. This list derives from the AHISTORY instructions. curhist is similar to curauto[144c] which you saw before.

⟨*global* curhist 149a⟩≡ (276b)

```
Auto* curhist;
```

In fact, the elements in curhist are gradually transfered to curauto:

⟨ldobj() *case ATEXT, curauto adjustments with curhist* 149b⟩≡ (145b)

```
histtoauto();
```

⟨ldobj() *case AEND, curauto adjustments with curhist* 149c⟩≡ (62b)

```
histtoauto();
```

⟨*function* histtoauto 149d⟩≡ (285b)

```
/// 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 144c and curhist 149a.

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 10.3.3, 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()[49b] dealing with ANAMEs first lookups in hash[28a] 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 150a⟩≡ (56d)
```
if(k == N_FILE) {
    if(s->type != SFILE) {
        s->type = SFILE;
        histgen++;
        s->value = histgen;
    }
    ⟨ldobj() when ANAME opcode, if N_FILE, update histfrogp 150f⟩
    ⟨ldobj() when ANAME opcode, if N_FILE, if no more space in histfrog 151d⟩
}
```
Uses SFILE and SXREF.

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* 150b⟩≡ (29d) 164b ▷
```
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 150c⟩≡ (276b)
```
int histgen = 0;
```
Uses histgen 150c.

Full filenames and histfrog

The series of path element symbols corresponding to the series of ANAMEs preceding an AHISTORY are then accumulated in the following global array:

⟨*global* histfrog 150d⟩≡ (276b)
```
Sym* histfrog[MAXHIST];
```
Uses MAXHIST 197g.

⟨*constant* MAXHIST 150e⟩≡ (271c)
```
MAXHIST     = 20,   /* limit of path elements for history symbols */
```

⟨ldobj() *when ANAME opcode, if* N_FILE, *update histfrogp* 150f⟩≡ (150a)
```
if(histfrogp < MAXHIST) {
    histfrog[histfrogp] = s;
    histfrogp++;
}
```
Uses MAXHIST 197g, histfrogp 150g, and histgen 150c.

⟨*global* histfrogp 150g⟩≡ (276b)
```
int histfrogp;
```

The array is reseted after each processed AHISTORY:

⟨ldobj() *in AHISTORY case, end of case, reset histfrogp* 150h⟩≡ (148a)
```
histfrogp = 0;
```

⟨ldobj() *after newloop when new object file, more initializations* 151a⟩+≡ (51) ◁59b 181b▷
```
histfrogp = 0;
```

I can now finally show how are encoded the names of `Hist` symbols. `5l` 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`[150c] stored in `Sym.valueX` of the corresponding path element symbol:

⟨addhist() *other locals* 151b⟩≡ (148b)
```
int i, j, k;
```

⟨addhist() *set symbol name to filename using compact encoding* 151c⟩≡ (148b)
```
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` 150d, `histfrogp` 150g, and `malloc()` 226a.

Symbol descriptions are usually separated by a `'\0'` (see Section 10.2.1), 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 `5l` 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()`

`5l` limits filenames to 20 path elements (see `MAXHIST`[197g]). 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:

⟨ldobj() *when ANAME opcode, if* `N_FILE`*, if no more space in histfrog* 151d⟩≡ (150a)
```
else
        collapsefrog(s);
```

Remember that `<` is the first character in all path elements, hence the many `+1` in the code below:

⟨*function* `collapsefrog` 152a⟩≡ (283a)
```
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<histfrogp; i++)
        if(strcmp(histfrog[i]->name+1, "..") == 0) {
            memmove(histfrog+i-1, histfrog+i+1,
                (histfrogp-i-1)*sizeof(histfrog[0]));
            histfrogp--;
            goto out;
        }

    /*
     * next try to collapse .
     */
    for(i=0; i<histfrogp; i++)
        if(strcmp(histfrog[i]->name+1, ".") == 0) {
            memmove(histfrog+i, histfrog+i+1,
                (histfrogp-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` 150d and `histfrogp` 150g.

10.3.3 Locations in executables

Now that the file and line information has been loaded in memory (in the symbol table, in some `Auto`[272]s 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 `Hist`s 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* 152b⟩≡ (142) 208e▷
```
case SFILE:
    putsymb(s->name, 'f', s->value, s->version);
```

```
    continue;
```
Uses SFILE.

Remember that the value of path elements symbols is an `histgen`[150c], an identifier referenced in the name of `Hist` symbols. The `<` prefix in path elements can be skipped:

⟨`putsymb()` *adjust string* `s` *if file symbol* 153a⟩≡ (141c)
```
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 10.2.2) in the executable symbol table:

⟨`asmsym()` *call putsymb for filenames* 153b⟩≡ (142)
```
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()` 141c.

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* 153c⟩≡ (141c)
```
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()` 227c.

Note that there is only one `i++` above. The code of `putsymb()`[141c] (see Section 10.2.1) uses `i` to adjust `symsize`[141b] but it does add 1 to `i` to account for the very last null character.

Program counter line table format

One side effect of `asmlc()`[154] is to modify the global `lcsize` 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* `lcsize` 153d⟩≡ (276b)
```
long lcsize;
```

`asmlc()`

⟨*constant* `MINLC` *(arm)* 153e⟩≡ (285a)
```
#define MINLC 4
```

⟨*function* asmlc 154⟩≡ (285a)

```
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 33d⟩
            ⟨asmlc() dump instruction p, debug 220c⟩
            continue;
        }
        // else
        ⟨asmlc() dump lcsize, debug 220d⟩
        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 220e⟩
            v -= s;
            lcsize++;
        }
        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 220f⟩
            lcsize += 5;
        } else {
            if(s > 0) {
                cput(0+s); /* 1-64 +lc */
                ⟨asmlc() dump small line increment, debug 220g⟩
            } else {
                cput(64-s); /* 65-128 -lc */
                ⟨asmlc() dump negative line increment, debug 221⟩
            }
            lcsize++;
        }
    }
    // padding
    while(lcsize & 1) {
        s = 129;
        cput(s);
        lcsize++;
    }
    if(debug['v'] || debug['V'])
        Bprint(&bso, "lcsize = %ld\n", lcsize);
    Bflush(&bso);
}
```

Uses INITTEXT 36e, MINLC-12, bso 211c, cput() 227c, debug 211a, and lcsize 153d.

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* 156a⟩≡ (38c)
```
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()`[157] or `doprof2()`[160a] which perform different kinds of profiling:

⟨`main()` *call doprofxxx() if profiling* 156b⟩≡ (74)
```
if(debug['p'])
    if(debug['1'])
        doprof1();
    else
        doprof2();
```

Those functions are called after `patch()`[77] and before `noops()`[82] (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()`[157] 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)* 157⟩≡ (284c)

```
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 272, P 32f, SBSS 164b, datap 33a, firstp 32d, lookup() 28e, and prg() 39a.

Note the use of SBSS[164b] instead of SDATAX above since the profiling instrumentation is done before dodata()[87c]. The layout of data has not been done yet.

Given the instrumentation done by doprof1(), we can then easily modify the main()[239j] 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()[160a], 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)* 160a⟩≡ (284c)
```
void
doprof2(void)
{
    Sym *s2, *s4;
    Prog *p, *q;
    ⟨doprof2() other locals 160b⟩

    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 162c⟩

    ⟨doprof2() find ps2, ps4, the Instr of s2 and s4 160c⟩

    for(p = firstp; p != P; p = p->link) {
        if(p->as == ATEXT) {
            ⟨doprof2() if NOPROF p(arm) 163a⟩
            ⟨doprof2() ATEXT instrumentation 161a⟩
            continue;
        }
        if(p->as == ARET) {
            ⟨doprof2() ARET instrumentation 161b⟩
            continue;
        }
    }
}
```
Uses `DBG` 272, `P` 32f, `firstp` 32d, and `lookup()` 28e.

Before instrumenting `ATEXT` and `ARET`, `5l` 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 `5l` will need to set their `Instr.condX` fields to point to the right instruction. `5l` needs to maintain what `patch()`[77] did for the other branching instructions.

⟨`doprof2()` *other locals* 160b⟩≡ (160a)
```
Prog *ps2 = P;
Prog *ps4 = P;
```
Uses `P` 32f.

⟨`doprof2()` *find ps2, ps4, the Instr of s2 and s4* 160c⟩≡ (160a)
```
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 163b⟩
        }
        if(p->from.sym == s4) {
            ps4 = p;
            ⟨doprof2() set TEXT attribute of _profin or _profout 163b⟩
        }
    }
}
```
Uses `P` 32f and `firstp` 32d.

The ATEXT instrumentation is trivial:

⟨doprof2() *ATEXT instrumentation* 161a⟩≡ (160a)

```
/*
 * 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() 39a.

The ARET instrumentation is more subtle. First, 5l 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* 161b⟩≡ (160a)

```
/*
 * 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 162a⟩
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() 39a and zprg 39b.

The second subtelity 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* 162a⟩≡ (161b)

```
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` 162b, `prg()` 39a, and `zprg` 39b.

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)* 162b⟩≡ (284c)

```
static int brcond[] =
 {ABEQ, ABNE,
  ABHS, ABLO,
  ABMI, ABPL,
  ABVS, ABVC,
  ABHI, ABLS,
  ABGE, ABLT,
  ABGT, ABLE};
```

⟨`doprof2()` *sanity check s2 and s4* 162c⟩≡ (160a)

```
if(s2->type != STEXT || s4->type != STEXT) {
    diag("_profin/_profout not defined");
    return;
}
```

Uses `STEXT` 150b and `diag()` 222d.

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)* 163a⟩≡ (160a)
```
if(p->reg & NOPROF) {
    for(;;) {
        q = p->link;
        if(q == P || q->as == ATEXT)
            break;
        p = q;
    }
    continue;
}
```
Uses `P` 32f.

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 `5l` would generate infinite loops, hence the defensive code below:

⟨`doprof2()` *set TEXT attribute of* `_profin` *or* `_profout` 163b⟩≡ (160)
```
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` 164a⟩≡ (276b)
```
bool dlm;
```

⟨`Section` *cases* 164b⟩+≡ (29d) ◁150b 169d▷
```
SIMPORT,
SEXPORT,
```

12.1.1 Export table: `5l -x`

⟨*global* `doexp` 164c⟩≡ (276b)
```
// do export table, -x
bool doexp;
```

⟨`main()` *command line processing(arm)* 164d⟩+≡ (34e) ◁68f 168b▷
```
case 'x': /* produce export table */
    doexp = true;
    if(argv[1] != nil && argv[1][0] != '-' && !isobjfile(argv[1]))
        readundefs(ARGF(), SEXPORT);
    break;
```

⟨*function* `isobjfile` 164e⟩≡ (283a)
```
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 && strncmp(buf2, ARMAG, SARMAG) == 0;
    }
    Bterm(b);
    return v;
}
```

⟨*global* `EXPTAB` 165a⟩≡ (276b)
```
char* EXPTAB;
```

⟨`main()` *if export table or dynamic module(arm)* 165b⟩≡ (74)
```
if(doexp || dlm){
    EXPTAB = "_exporttab";
    zerosig(EXPTAB);
    zerosig("etext");
    zerosig("edata");
    zerosig("end");

    ⟨main() if dynamic module(arm) 169a⟩
    else
        divsig();

    export();
}
```

⟨*function* `zerosig` 165c⟩≡ (280b)
```
void
zerosig(char *sp)
{
    Sym *s;

    s = lookup(sp, 0);
    s->sig = 0;
}
```
Uses `lookup()` 28e.

⟨*global* `nimports` 165d⟩≡ (280b)
```
int nimports;
```

⟨*global* `nexports` 165e⟩≡ (280b)
```
int nexports;
```

⟨*global* `imports` 165f⟩≡ (280b)
```
int imports;
```

⟨*global* `exports` 165g⟩≡ (280b)
```
int exports;
```

⟨`Sym` *other fields* 165h⟩+≡ (27) ◁64a
```
// enum<Section> too?
short   subtype;
```

⟨*function* `readundefs` 166a⟩≡ (280b)

```
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);
        errorexit();
    }
    while((l = Brdline(b, '\n')) != nil){
        n = Blinelen(b);
        if(n >= sizeof(buf)){
            diag("%s: line too long", f);
            errorexit();
        }
        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);
            errorexit();
        }
        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()` 222d, `errorexit()` 222b, `lookup()` 28e, `nexports` 165e, and `nimports` 165d.

⟨*function* `export` *(arm)* 166b⟩≡ (280b)

```
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 165a, NHASH, S 29d, SDATA, SEXPORT 207e, SUNDEF 29d, SXREF, bso 211c, diag() 222d, exports 165g, free() 226b, lookup() 28e, malloc() 226a, newdata() 168a, and nexports 165e.

⟨*function* newdata*(arm)* 168a⟩≡ (280b)

```
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 33a and prg() 39a.

12.1.2 Dynamic loading: 5l -u

⟨main() *command line processing(arm)* 168b⟩+≡ (34e) ◁164d 182f▷

```
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)* 168c⟩≡ (42a)

```
if(dlm)
    lput(0x80000000|0x647); /* magic */
```

Uses H_PLAN9 143b.

⟨main() *if dynamic module(arm)* 169a⟩≡ (165b)
```
if(dlm){
    initdiv();
    import();
    HEADTYPE = H_PLAN9;
    INITTEXT = INITDAT = 0;
    INITRND = 8;
    INITENTRY = EXPTAB;
}
```

⟨*function* import*(arm)* 169b⟩≡ (280b)
```
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 29d, SIMPORT, SXREF, bso 211c, nimports 165d, and undefsym() 170b.

⟨*enum* rxxx 169c⟩≡ (272)
```
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* 169d⟩+≡ (29d) ◁164b 176e▷
```
SUNDEF,
```

⟨*global* undefp 169e⟩≡ (276b)
```
//@Scheck: not dead, used by UP
Prog undefp;
```

⟨*constant* UP 169f⟩≡ (272)
```
#define UP   (&undefp)
```

⟨patch() *switch section type for branch instruction, cases* 169g⟩+≡ (78b) ◁78c
```
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 29d and diag() 222d.

⟨asmout() *BRA case, if undefined target* 170a⟩≡ (122d)
```
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 29d, UP 272, diag() 222d, and dynreloc() 173b.

⟨*function* undefsym 170b⟩≡ (280b)
```
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 176d, Roffset, SUNDEF 29d, diag() 222d, and imports 165f.

⟨datblk() *in* D_ADDR *case, switch symbol type cases* 170c⟩+≡ (47d) ◁48a
```
case SUNDEF:
    ckoff(v, d);
    d += p->to.sym->value;
    break;
```
Uses SUNDEF 29d and ckoff() 170d.

⟨*function* ckoff 170d⟩≡ (280b)
```
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() 222d.

12.1.4 XXX

⟨aclass() *in* D_ADDR *case,* SDATA *case, if dlm* 170e⟩≡ (112a)
```
if(dlm) {
    instoffset = s->value + a->offset + INITDAT;
    return C_LCON;
}
```
Uses INITDAT 36g, SBSS 164b, SDATA, SDATA1 272, dlm 164a, and instoffset 107d.

⟨asmb() *if dynamic module, before datblk()* 170f⟩≡ (44a)
```
if(dlm){
    char buf[8];

    write(cout, buf, INITDAT-textsize);
    textsize = INITDAT;
}
```
Uses INITDAT 36g, cout 34d, dlm 164a, and textsize 89a.

⟨asmb() *if dynamic module and no symbol table generation* 171a⟩≡ (140)

```
if(dlm){
    seek(cout, HEADR+textsize+datsize, 0);
    asmdyn();
    cflush();
}
```

Uses HEADR 36d, asmdyn() 171f, cout 34d, datsize 87a, dlm 164a, and textsize 89a.

⟨entryvalue() *if dynamic module case* 171b⟩≡ (42b)

```
case SDATA:
    if(dlm)
        return s->value+INITDAT;
```

Uses SDATA.

⟨*struct* Reloc 171c⟩≡ (280b)

```
struct Reloc
{
    int n;
    int t;
    byte *m;
    ulong *a;
};
```

⟨*global* rels 171d⟩≡ (280b)

```
Reloc rels;
```

⟨asmb() *if dynamic module, call asmdyn()* 171e⟩≡ (140)

```
if(dlm)
    asmdyn();
```

⟨*function* asmdyn 171f⟩≡ (280b)

```
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 272, NHASH, S 29d, SUNDEF 29d, cflush() 104e, cout 34d, cput() 227c, diag() 222d, exports 165g, imports 165f, lput() 104f, rels 171d, sput() 172a, and wput() 227e.

⟨*function* sput 172a⟩≡ (280b)

```
    static int
    sput(char *s)
    {
        char *p;

        p = s;
        while(*s)
            cput(*s++);
        cput(0);
        return  s-p+1;
    }
```

Uses cput() 227c.

⟨*global* modemap 172b⟩≡ (280b)

```
    int modemap[4] = { 0, 1, -1, 2, };
```

⟨datblk() *if dynamic module(arm)* 172c⟩≡ (47d)

```
    if(dlm)
        dynreloc(v, a+INITDAT, 1);
```

Uses INITDAT 36g, dlm 164a, and dynreloc() 173b.

⟨*enum SpanConstants(arm)* 173a⟩≡ (280b)

```
enum{
    ABSD = 0,
    ABSU = 1,
    RELD = 2,
    RELU = 3,
};
```

⟨*function* dynreloc*(arm)* 173b⟩≡ (280b)

```
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 173a, ABSU-4 173a, RELD-5 173a, RELU-6 173a, S 29d, SUNDEF 29d, diag() 222d, grow() 173c, modemap 172b, and rels 171d.

⟨*function* grow 173c⟩≡ (280b)

```
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() 226b and malloc() 226a.

12.1.5 Relocatable address: C_ADDR

⟨Operand_class *cases* 174a⟩+≡ (95c) ◁116b 184a▷

```
    C_ADDR,     /* relocatable address */
```

⟨aclass() *when* D_OREG *and external symbol and dlm* 174b⟩≡ (110c)

```
    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 INITDAT 36g, SBSS 164b, SDATA, SSTRING 32c, STEXT 150b, SUNDEF 29d, dlm 164a, and instoffset 107d.

⟨optab *entries* 174c⟩+≡ (94a) ◁134d 174d▷

```
    { ATEXT, C_ADDR, C_NONE, C_LCON,   0, 0 },
    { ATEXT, C_ADDR, C_REG, C_LCON,   0, 0 },
```

Uses C_NONE 95c.

⟨optab *entries* 174d⟩+≡ (94a) ◁174c 174f▷

```
    { AWORD, C_NONE, C_NONE, C_ADDR,  11, 4 },
```

Uses C_ADDR 174a, C_LCON 107e, and C_REG 95c.

⟨asmout() *in AWORD case, when dlm* 174e⟩≡ (107c)

```
    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 174a, C_LCON 107e, SUNDEF 29d, ckoff() 170d, dlm 164a, and dynreloc() 173b.

⟨optab *entries* 174f⟩+≡ (94a) ◁174d 175b▷

```
    { AMOVW, C_REG, C_NONE, C_ADDR,  64, 8, 0, LTO },
    { AMOVB, C_REG, C_NONE, C_ADDR,  64, 8, 0, LTO },
    { AMOVBU, C_REG, C_NONE, C_ADDR,  64, 8, 0, LTO },
```

Uses C_ADDR 174a, C_NONE 95c, C_REG 95c, and LTO 135d.

⟨asmout() *switch on type cases* 175a⟩+≡ (102e) ◁134e 175c▷
```
/* 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() 120b and osr() 128b.

⟨optab *entries* 175b⟩+≡ (94a) ◁174f 175d▷
```
{ AMOVW, C_ADDR, C_NONE, C_REG,  65, 8, 0, LFROM },
{ AMOVBU, C_ADDR, C_NONE, C_REG,  65, 8, 0, LFROM },
{ AMOVB, C_ADDR, C_NONE, C_REG,  66, 16, 0, LFROM },
{ AMOVH, C_ADDR, C_NONE, C_REG,  66, 16, 0, LFROM },
{ AMOVHU, C_ADDR, C_NONE, C_REG,  66, 16, 0, LFROM },
```
Uses C_ADDR 174a, C_NONE 95c, C_REG 95c, LFROM 135d, and LTO 135d.

⟨asmout() *switch on type cases* 175c⟩+≡ (102e) ◁175a 175e▷
```
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 == AMOVB || p->as == AMOVBU) {
        o3 |= (24<<7);
        o4 |= (24<<7);
    } else {
        o3 |= (16<<7);
        o4 |= (16<<7);
    }
    break;
```
Uses olr() 126b, omvl() 120b, and oprrr() 114d.

⟨optab *entries* 175d⟩+≡ (94a) ◁175b 188c▷
```
{ 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 174a, C_NONE 95c, C_REG 95c, and LFROM 135d.

⟨asmout() *switch on type cases* 175e⟩+≡ (102e) ◁175c 189b▷
```
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() 108b, omvl() 120b, oprrr() 114d, and osr() 128b.

12.2 Position independent code (PIC)

12.3 Optimizations

12.3.1 Opcode rewriting

⟨ldobj() *switch opcode cases(arm)* 176a⟩+≡ (51) ◁148a 176b▷

```
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)* 176b⟩+≡ (51) ◁176a 185g▷

```
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* 176c⟩+≡ (87c) ◁92b

```
xdefine("setR12", SDATA, 0L+BIG);
```

Uses STEXT 150b and xdefine() 92a.

⟨*constant* BIG 176d⟩≡ (271c)

```
//BIG       = (1<<12)-4,
BIG     = 0,
```

12.3.3 Small data first

⟨Section *cases* 176e⟩+≡ (29d) ◁169d 207e▷

```
SDATA1,
```

⟨*constant* MINSIZ 177a⟩≡ (271c)
```
MINSIZ      = 64,
```

⟨dodata() *in pass 1, if small data size, adjust orig* 177b⟩≡ (87c)
```
/*
 * 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 150e.

⟨dodata() *in pass 2, retag small data* 177c⟩≡ (87c)
```
if(t == SDATA1)
    s->type = SDATA;
```

12.3.4 Compacting chains of AB, brloop()

⟨patch() *optimisation pass* 177d⟩≡ (77)
```
for(p = firstp; p != P; p = p->link) {
    ⟨adjust curtext when iterate over instructions p 33d⟩

    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 32f, UP 272, brloop() 177e, and curtext 33b.

⟨*function* brloop*(arm)* 177e⟩≡ (279b)
```
/// 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 32f.

12.3.5 Removing useless instructions, `follow()`

⟨*function* `follow` 178a⟩≡ (279b)
```
void
follow(void)
{

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

    firstp = prg();
    lastp = firstp;

    xfol(textp);

    lastp->link = P;
    firstp = firstp->link;
}
```
Uses `DBG` 272, `P` 32f, `firstp` 32d, `lastp` 32g, `prg()` 39a, `textp` 145d, and `xfol()` 178c.

⟨`Mark` *cases* 178b⟩+≡ (32c) ◁83a
```
FOLL        = 1<<0,
```

⟨*function* `xfol`*(arm)* 178c⟩≡ (279b)
```
void
xfol(Prog *p)
{
    Prog *q, *r;
    int a, i;

loop:
    if(p == P)
        return;
    ⟨adjust curtext when iterate over instructions p 33d⟩
    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 180⟩
        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 179a⟩
    p = p->link;
    goto loop;
}
```

Uses FOLL 36c, P 32f, curtext 33b, lastp 32g, and prg() 39a.

⟨xfol() *if a is not ABL and p has a link* 179a⟩≡ (178c)

```
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 36c, P 32f, brchain() 179b, relinv() 179c, and xfol() 178c.

⟨*function* brchain*(arm)* 179b⟩≡ (279b)

```
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 32f.

⟨*function* relinv*(arm)* 179c⟩≡ (279b)

```
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() 222d.

⟨xfol() *when p is marked, for loop to copy instructions* 180⟩≡ (178c)

```
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 36c, lastp 32g, and prg() 39a.

12.4 Overriding symbol attribute: DUPOK

⟨ldobj() *locals(arm)* 181a⟩+≡ (51) ◁64e 185e▷
```
  bool skip;
```

⟨ldobj() *after newloop when new object file, more initializations* 181b⟩+≡ (51) ◁151a
```
  skip = false;
```

⟨ldobj() *case ATEXT and section not SNONE or SXREF, if DUPOK* 181c⟩≡ (60c)
```
  if(p->reg & DUPOK) {
      skip = true;
      goto casedef;
  }
```
Uses SNONE 29d and SXREF.

⟨ldobj() *in switch opcode default case, if skip* 181d⟩≡ (59d)
```
  if(skip)
      nopout(p);
```

⟨ldobj() *in switch opcode ATEXT case, reset skip* 181e⟩≡ (60b)
```
  skip = false; // needed?
```

⟨*function* nopout 181f⟩≡ (283a)
```
  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)* 181g⟩+≡ (37a) ◁37b
```
  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)* 182a⟩+≡ (44a) ◁44b
```
case H_ELF:
    OFFSET = HEADR+textsize;
    seek(cout, OFFSET, 0);
    break;
```
Uses HEADR 36d, H_ELF 143b, and textsize 89a.

⟨asmb() *switch HEADTYPE (for symbol table generation) cases(arm)* 182b⟩+≡ (140) ◁141a
```
case H_ELF:
    break;
```

⟨asmb() *switch HEADTYPE (for header generation) cases(arm)* 182c⟩+≡ (41c) ◁42a
```
case H_ELF:
    debug['S'] = 1;    /* symbol table */
    elf32(ARM, ELFDATA2LSB, 0, nil);
    break;
```
Uses H_ELF 143b and debug 211a.

⟨*global* INITTEXTP 182d⟩≡ (276b)
```
long INITTEXTP = -1; /* text location (physical) */
```
Uses INITTEXTP 182d.

⟨main() *last INITXXX adjustments* 182e⟩≡
```
if (INITTEXTP == -1)
    INITTEXTP = INITTEXT;
```

⟨main() *command line processing(arm)* 182f⟩+≡ (34e) ◁168b 211b▷
```
case 'P':
    a = ARGF();
    if(a)
        INITTEXTP = atolwhex(a);
    break;
```

⟨*enum ElfHeaderSizes* 182g⟩≡
```
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* 182h⟩+≡ (52d) ◁53a 183a▷
```
case D_FREG:
case D_FPCR:
    break;
```

⟨inopd() *cases* 183a⟩+≡ (52d) ◁182h
```
  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() 226a.

⟨*function* ieeedtof 183b⟩≡ (284b)
```
  /// main -> objfile -> ldobj -> <>
  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 & 0xfffffL) << 3;
      v |= (e->l >> 29) & 0x7L;
      if((e->l >> 28) & 1) {
          v++;
          if(v & 0x800000L) {
              v = (v & 0x7fffffL) >> 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() 222d.

⟨*function* ieeedtod 183c⟩≡ (284b)
```
  /// Dconv -> <>
  double
  ieeedtod(Ieee *ieeep)
  {
      Ieee e;
      double fr;
      int exp;

      if(ieeep->h & (1L<<31)) {
          e.h = ieeep->h & ~(1L<<31);
          e.l = ieeep->l;
          return -ieeedtod(&e);
      }
      if(ieeep->l == 0 && ieeep->h == 0)
          return 0;
      fr = ieeep->l & ((1L<<16)-1L);
      fr /= 1L<<16;
      fr += (ieeep->l>>16) & ((1L<<16)-1L);
      fr /= 1L<<16;
      fr += (ieeep->h & (1L<<20)-1L) | (1L<<20);
```

```
    fr /= 1L<<21;
    exp = (ieeep->h>>20) & ((1L<<11)-1L);
    exp -= (1L<<10) - 2L;
    return ldexp(fr, exp);
}
```

Uses ieeedtod() 183c.

Operand class

⟨Operand_class *cases* 184a⟩+≡ (95c) ◁174a 200a▷

```
C_FREG,
C_FCON,
C_FCR,
```

⟨aclass() *switch type cases* 184b⟩+≡ (96a) ◁116c 200b▷

```
case D_FREG:
    return C_FREG;
case D_FCONST:
    return C_FCON;
case D_FPCR:
    return C_FCR;
```

Uses C_FCON 184a, C_FREG 184a, and C_SHIFT 116b.

⟨*function* immfloat*(arm)* 184c⟩≡ (282c)

```
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* 184d⟩≡ (110b)

```
C_FEXT,
C_HFEXT,
```

⟨aclass() *if immfloat for* N_EXTERN *symbol* 184e⟩≡ (110c)

```
if(immfloat(t))
    return immhalf(instoffset)? C_HFEXT : C_FEXT;
```

⟨Operand_class *cases, in* C_xAUTO*, float cases* 184f⟩≡ (111b)

```
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* 184g⟩≡ (111)

```
if(immfloat(t))
    return immhalf(instoffset)? C_HFAUTO : C_FAUTO;
```

⟨Operand_class *cases, in* C_xOREG*, float cases* 184h⟩≡ (109a)

```
C_FOREG,
C_HFOREG,
```

⟨aclass() *if immfloat for* N_NONE *symbol* 184i⟩≡ (109c)

```
if(immfloat(t))
    return immhalf(instoffset)? C_HFOREG : C_FOREG;
    /* n.b. that [C_FOREG] will also satisfy immrot */
```

Uses immfloat() 184c.

VFP

⟨Optab_flag *cases* 184j⟩≡ (135d) 202c▷

```
VFP     = 1<<4, /* arm vfpv3 floating point */
```

⟨ocmp() *if floating point flag on p1 or p2* 185a⟩≡ (99b)
```
n = (p2->flag&VFP) - (p1->flag&VFP); /* floating point arch */
if(n)
    return n;
```
Uses VFP 184j.

⟨*global* vfp*(arm)* 185b⟩≡ (276b)
```
bool vfp;
```

⟨buildop() *initialize flags* 185c⟩≡ (98d) 202b▷
```
vfp = debug['f'];
```

⟨buildop() *adjust optab if flags, remove certain rules* 185d⟩≡ (98d) 202d▷
```
if((optab[n].flag & VFP) && !vfp)
    optab[n].as = AXXX;
```
Uses optab 94a.

⟨ldobj() *locals(arm)* 185e⟩+≡ (51) ◁181a
```
Prog *t;
```

⟨*global* literal*(arm)* 185f⟩≡ (283a)
```
char literal[32];
```

⟨ldobj() *switch opcode cases(arm)* 185g⟩+≡ (51) ◁176b
```
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 */
        sprint(literal, "$%lux", ieeedtof(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 */
            sprint(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 164b, chipfloat() 186b, datap 33a, ieeedtof() 183b, literal 283a, lookup() 28e, prg() 39a, and vfp 185b.

⟨*global* chipfloats*(arm)* 186a⟩≡ (284b)

```
  static Ieee chipfloats[] = {
        {0x00000000, 0x00000000}, /* 0 */
        {0x00000000, 0x3ff00000}, /* 1 */
        {0x00000000, 0x40000000}, /* 2 */
        {0x00000000, 0x40080000}, /* 3 */
        {0x00000000, 0x40100000}, /* 4 */
        {0x00000000, 0x40140000}, /* 5 */
        {0x00000000, 0x3fe00000}, /* .5 */
        {0x00000000, 0x40240000}, /* 10 */
  };
```

⟨*function* chipfloat*(arm)* 186b⟩≡ (284b)

```
  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 186a and vfp 185b.

⟨noops() *second pass switch opcode cases* 187a⟩+≡ (82) ◁84b 195▷

```
/*
 * 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 32f and prg() 39a.

⟨datblk() *other locals* 187b⟩+≡ (44d) ◁47c

```
long fl;
```

⟨datblk() *switch type of destination cases* 187c⟩+≡ (44d) ◁46a

```
case D_FCONST:
    switch(c) {
    default:
    case 4:
        fl = ieeedtof(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 227b, fnuxi4 187d, fnuxi8 187e, and ieeedtof() 183b.

⟨*global* fnuxi4 187d⟩≡ (280a)

```
char fnuxi4[4];
```

⟨*global* fnuxi8 187e⟩≡ (280a)

```
char fnuxi8[8];
```

⟨nuxiinit() *in loop i, fnuxi initialisation* 188a⟩≡ (46f)
```
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 211a, fnuxi4 187d, and fnuxi8 187e.

Common float instructions

⟨buildop() *switch opcode r for ranges cases* 188b⟩+≡ (98d) ◁133b 199c▷
```
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.

⟨optab *entries* 188c⟩+≡ (94a) ◁175d 192c▷
```
{ 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 174a, C_FAUTO 184f, C_FCON 184a, C_FEXT 184d, C_FOREG 184h, C_FREG 184a, C_LAUTO 111b, C_LEXT 110b, C_LOREG 109a, C_NONE 95c, C_REG 95c, LFROM 135d, and LTO 135d.

⟨*function* regoff *(arm)* 189a⟩≡ (282c)

```
long
regoff(Adr *a)
{

    instoffset = 0;
    aclass(a);
    return instoffset;
}
```

Uses aclass() and instoffset 107d.

⟨asmout() *switch on type cases* 189b⟩+≡ (102e) ◁175e 190a▷

```
case 50: /* floating point store */
    v = regoff(&p->to);
    r = p->to.reg;
    ⟨asmout() adjust maybe r to rule param 126a⟩
    o1 = ofsr(p->as, p->from.reg, v, r, p->scond, p);
    break;
```

Uses ofsr() 189c and regoff() 189a.

⟨*function* ofsr *(arm)* 189c⟩≡ (281)

```
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\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;
        o |= 1 << 8;

        switch(a) {
        default:
            diag("bad fst %A", a);
        case AMOVD:
            o |= 1<<15;
        case AMOVF:
            break;
        }
        return o;
    }
```

Uses diag() 222d, ovfpmem() 192f, and vfp 185b.

⟨asmout() *switch on type cases* 190a⟩+≡ (102e) ◁189b 190b▷

```
    case 51: /* floating point load */
        v = regoff(&p->from);
        r = p->from.reg;
        ⟨asmout() adjust maybe r to rule param 126a⟩
        o1 = ofsr(p->as, p->to.reg, v, r, p->scond, p) | (1<<20);
        break;
```

Uses ofsr() 189c and regoff() 189a.

⟨asmout() *switch on type cases* 190b⟩+≡ (102e) ◁190a 190c▷

```
    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 126a⟩
        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() 189c, omvl() 120b, and oprrr() 114d.

⟨asmout() *switch on type cases* 190c⟩+≡ (102e) ◁190b 191a▷

```
    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 126a⟩
        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() 189c, omvl() 120b, and oprrr() 114d.

Old ARM 7500 floating point

⟨oprrr() *switch cases* 190d⟩+≡ (114d) ◁134c 200e▷

```
    /* old arm 7500 fp using coprocessor 1 (1<<8) */
```

```
  case AADDD: return o | (0xe<<24) | (0x0<<20) | (1<<8) | (1<<7);
  case AADDF: 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, ACMPF should expand to RNDF, CMPD */
  case ACMPD:
  case ACMPF: 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 AMOVDW: return o | (0xe<<24) | (1<<20) | (1<<8) | (1<<4) | (1<<7);
```

⟨asmout() *switch on type cases* 191a⟩+≡ (102e) ◁190c 191b▷

```
  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 114b⟩
      o1 |= rf | (r<<16) | (rt<<12);
      break;
```

Uses chipfloat() 186b, diag() 222d, and oprrr() 114d.

⟨asmout() *switch on type cases* 191b⟩+≡ (102e) ◁191a 192a▷

```
  case 55: /* floating point fix and float */
      o1 = oprrr(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() 222d and oprrr() 114d.

⟨asmout() *switch on type cases* 192a⟩+≡ (102e) ◁191b 192b▷
```
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() 189c and omvl() 120b.

⟨asmout() *switch on type cases* 192b⟩+≡ (102e) ◁192a 192d▷
```
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() 189c and omvl() 120b.

⟨optab *entries* 192c⟩+≡ (94a) ◁188c 193a▷
```
{ AMOVW, C_REG, C_NONE, C_FCR,  56, 4 },
{ AMOVW, C_FCR, C_NONE, C_REG,  57, 4 },
```
Uses C_FREG 184a, C_NONE 95c, and C_REG 95c.

⟨asmout() *switch on type cases* 192d⟩+≡ (102e) ◁192b 192e▷
```
/* 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* 192e⟩+≡ (102e) ◁192d 193b▷
```
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, 5l -f

⟨*function* ovfpmem*(arm)* 192f⟩≡ (281)
```
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() 222d.

⟨optab *entries* 193a⟩+≡ (94a) ◁192c 199d▷

```
    { AADDF, C_FREG, C_NONE, C_FREG,  74, 4, 0, VFP },
    { AADDF, 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 184a, C_FCR 184a, C_FREG 184a, C_NONE 95c, C_REG 95c, and VFP 184j.

⟨asmout() *switch on type cases* 193b⟩+≡ (102e) ◁192e 193c▷

```
    /* 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 114b⟩
        o1 |= rt<<12;
        if(((o1>>20)&0xf) == 0xb)
            o1 |= rf<<0;
        else
            o1 |= r<<16 | rf<<0;
        break;
```

Uses diag() 222d and opvfprrr() 194b.

⟨asmout() *switch on type cases* 193c⟩+≡ (102e) ◁193b 194a▷

```
    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() 222d and opvfprrr() 194b.

⟨asmout() *switch on type cases* 194a⟩+≡ (102e) ◁193c 199e▷

```
    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 | rt<<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() 194b.

⟨*function* opvfprrr*(arm)* 194b⟩≡ (281)

```
    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 AADDD: 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 33c, diag() 222d, and prasm() 212b.

12.6.2 Division

ADIV rewriting

⟨noops() *second pass switch opcode cases* 195⟩+≡ (82) ◁187a

```
case ADIV:
case ADIVU:
case AMOD:
case AMODU:
    ⟨noops() second pass, ADIV rewrite, case ADIV and so on, if -M 199b⟩
    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()` 39a, `prog_div`, `prog_divu`, `prog_mod`, `prog_modu` 284a, `sym_div-8`, `sym_divu-9`, `sym_mod-10`, and `sym_modu-11` 284a.

⟨*global* `prog_div` *(arm)* 196a⟩≡ (284a)

```
Prog* prog_div;
```

⟨*global* `prog_divu` *(arm)* 196b⟩≡ (284a)

```
Prog* prog_divu;
```

⟨*global* `prog_mod`*(arm)* 197a⟩≡ (284a)
```
Prog* prog_mod;
```

⟨*global* `prog_modu`*(arm)* 197b⟩≡ (284a)
```
Prog* prog_modu;
```

`initdiv()`

⟨*global* `sym_div`*(arm)* 197c⟩≡ (284a)
```
static Sym* sym_div;
```

⟨*global* `sym_divu`*(arm)* 197d⟩≡ (284a)
```
static Sym* sym_divu;
```

⟨*global* `sym_mod`*(arm)* 197e⟩≡ (284a)
```
static Sym* sym_mod;
```

⟨*global* `sym_modu`*(arm)* 197f⟩≡ (284a)
```
static Sym* sym_modu;
```

⟨*constant* `SIGNINTERN`*(arm)* 197g⟩≡ (272)
```
#define SIGNINTERN  (1729*325*1729)
```

⟨*function* `sigdiv`*(arm)* 197h⟩≡ (284a)
```
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` 272, `SNONE` 29d, `STEXT` 150b, `SXREF`, and `lookup()` 28e.

⟨*function* `divsig`*(arm)* 197i⟩≡ (284a)
```
void
divsig(void)
{
    sigdiv("_div");
    sigdiv("_divu");
    sigdiv("_mod");
    sigdiv("_modu");
}
```
Uses `sigdiv()` 197h.

⟨*function* `sdiv`*(arm)* 198a⟩≡ (284a)

```
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` 272, `SIMPORT`, `SNONE` 29d, `STEXT` 150b, and `SXREF`.

⟨*function* `initdiv`*(arm)* 198b⟩≡ (284a)

```
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 32f, SXREF, UP 272, curtext 33b, diag() 222d, dlm 164a, firstp 32d, lookup() 28e, prog_div, prog_divu, prog_mod, prog_modu 284a, sdiv() 198a, sym_div-8, sym_divu-9, sym_mod-10, and sym_modu-11 284a.

⟨noops() *first pass switch opcode cases* 199a⟩+≡ (82) ◁85

```
case ADIV:
case ADIVU:
case AMOD:
case AMODU:
    if(prog_div == P)
        initdiv();
    if(curtext != P)
        curtext->mark &= ~LEAF;
    continue; // no q = p;
```

Uses P 32f, curtext 33b, initdiv() 198b, and prog_div.

Kernel emulation, 5l -M

⟨noops() *second pass, ADIV rewrite, case ADIV and so on, if -M* 199b⟩≡ (195)

```
if(debug['M'])
    break;
```

⟨buildop() *switch opcode r for ranges cases* 199c⟩+≡ (98d) ◁188b 199f▷

```
case ADIV:
    oprange[AMOD] = oprange[r];
    oprange[AMODU] = oprange[r];
    oprange[ADIVU] = oprange[r];
    break;
```

Uses oprange.

⟨optab *entries* 199d⟩+≡ (94a) ◁193a 199g▷

```
{ ADIV,  C_REG, C_REG, C_REG,  16, 4 },
{ ADIV,  C_REG, C_NONE, C_REG,  16, 4 },
```

Uses C_FREG 184a, C_NONE 95c, C_REG 95c, and VFP 184j.

⟨asmout() *switch on type cases* 199e⟩+≡ (102e) ◁194a 200d▷

```
case 16: /* div r,[r,]r */
    o1 = 0xf << 28;
    o2 = 0;
    break;
```

12.6.3 Long multiplication

⟨buildop() *switch opcode r for ranges cases* 199f⟩+≡ (98d) ◁199c 200f▷

```
case AMULL:
    oprange[AMULA] = oprange[r];
    oprange[AMULAL] = oprange[r];
    oprange[AMULLU] = oprange[r];
    oprange[AMULALU] = oprange[r];
    break;
```

Uses oprange.

⟨optab *entries* 199g⟩+≡ (94a) ◁199d 200g▷

```
{ AMULL, C_REG, C_REG, C_REGREG, 17, 4 },
```

Uses C_NONE 95c and C_REG 95c.

⟨Operand_class *cases* 200a⟩+≡ (95c) ◁184a 201b▷
```
C_REGREG,  // D_REGREG
```

⟨aclass() *switch type cases* 200b⟩+≡ (96a) ◁184b 201c▷
```
case D_REGREG:
    return C_REGREG;
```
Uses C_FCR 184a.

⟨asmout() *other locals* 200c⟩+≡ (102e) ◁?? 218f▷
```
int rt2;
```

⟨asmout() *switch on type cases* 200d⟩+≡ (102e) ◁199e 200h▷
```
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() 114d.

⟨oprrr() *switch cases* 200e⟩+≡ (114d) ◁190d
```
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* 200f⟩+≡ (98d) ◁199f 206e▷
```
case AMOVM:
    break;
```

⟨optab *entries* 200g⟩+≡ (94a) ◁199g 201a▷
```
{ AMOVM, C_LCON, C_NONE, C_SOREG, 38, 4 },
{ AMOVM, C_SOREG,C_NONE, C_LCON,  39, 4 },
```
Uses C_REG 95c and C_REGREG 200a.

⟨asmout() *switch on type cases* 200h⟩+≡ (102e) ◁200d 200i▷
```
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().

⟨asmout() *switch on type cases* 200i⟩+≡ (102e) ◁200h 201d▷
```
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(), diag() 222d, and instoffset 107d.

12.6.5 Status register

⟨optab *entries* 201a⟩+≡ (94a) ◁200g 203d▷

```
  { 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_LCON 107e, C_NONE 95c, C_PSR 201b, C_REG 95c, and C_SOREG 109a.

⟨Operand_class *cases* 201b⟩+≡ (95c) ◁200a

```
  C_PSR,      // D_PSR
```

⟨aclass() *switch type cases* 201c⟩+≡ (96a) ◁200b

```
  case D_PSR:
        return C_PSR;
```

Uses C_REGREG 200a.

⟨asmout() *switch on type cases* 201d⟩+≡ (102e) ◁200i 201e▷

```
  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* 201e⟩+≡ (102e) ◁201d 201f▷

```
  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* 201f⟩+≡ (102e) ◁201e 204a▷

```
  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(), immrot() 108b, and instoffset 107d.

12.6.6 Half words and bytes moves since ARMv4

⟨*global* `armv4`*(arm)* 202a⟩≡ (276b)
```
bool armv4;
```

⟨`buildop()` *initialize flags* 202b⟩+≡ (98d) ◁185c
```
armv4 = !debug['h'];
```
Uses `debug` 211a and `vfp` 185b.

⟨`Optab_flag` *cases* 202c⟩+≡ (135d) ◁184j
```
V4      = 1<<3, /* arm v4 arch */
```

⟨`buildop()` *adjust optab if flags, remove certain rules* 202d⟩+≡ (98d) ◁185d
```
if((optab[n].flag & V4) && !armv4) {
    optab[n].as = AXXX;
    break;
}
```
Uses `V4` 202c, `armv4` 202a, and `optab` 94a.

⟨`ocmp()` *if v4 flag on p1 or p2* 202e⟩≡ (99b)
```
n = (p2->flag&V4) - (p1->flag&V4); /* architecture version */
if(n)
    return n;
```
Uses `V4` 202c.

⟨`Operand_class` *cases, in* `C_xEXT`*, half case* 202f⟩≡ (110b)
```
C_HEXT,
```

⟨`aclass()` *if immhalf for* `N_EXTERN` *symbol* 202g⟩≡ (110c)
```
if(immhalf(instoffset))
    return C_HEXT;
```

⟨`cmp()` *switch on a, the operand class in optab rule, cases* 202h⟩+≡ (98a) ◁112d 202j▷
```
case C_HFEXT:
    return b == C_HEXT || b == C_FEXT;
case C_FEXT:
case C_HEXT:
    return b == C_HFEXT;
```
Uses `C_FEXT` 184d, `C_HEXT` 202f, and `C_HFEXT` 184d.

⟨`Operand_class` *cases, in* `C_xAUTO`*, half case* 202i⟩≡ (111b)
```
C_HAUTO,    /* halfword insn offset (-0xff to 0xff) */
```

⟨`cmp()` *switch on a, the operand class in optab rule, cases* 202j⟩+≡ (98a) ◁202h 203b▷
```
case C_HFAUTO:
    return b == C_HAUTO || b == C_FAUTO;
case C_FAUTO:
case C_HAUTO:
    return b == C_HFAUTO;
```
Uses `C_FAUTO` 184f, `C_HAUTO` 202i, `C_HFAUTO` 184f, and `C_HFEXT` 184d.

⟨`aclass()` *if immhalf for* `N_LOCAL` *or* `N_PARAM` *symbol* 202k⟩≡ (111)
```
if(immhalf(instoffset))
    return C_HAUTO;
```

⟨`Operand_class` *cases, in* `C_xOREG`*, half case* 202l⟩≡ (109a)
```
C_HOREG,
```

⟨aclass() *if immhalf for* N_NONE *symbol* 203a⟩≡ (109c)
```
   /* n.b. that immhalf() will also satisfy immrot */
  if(immhalf(instoffset))
      return C_HOREG;
```

⟨cmp() *switch on a, the operand class in optab rule, cases* 203b⟩+≡ (98a) ◁202j
```
  case C_HFOREG:
      return b == C_HOREG || b == C_FOREG;
  case C_FOREG:
  case C_HOREG:
      return b == C_HFOREG;
```
Uses C_FOREG 184h, C_HFAUTO 184f, C_HFOREG 184h, and C_HOREG 202l.

⟨*function* immhalf *(arm)* 203c⟩≡ (282c)
```
  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* 203d⟩+≡ (94a) ◁201a 206a▷
```
  { 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 },


  { AMOVB, C_HEXT, C_NONE, C_REG,  71, 4, REGSB, V4 },
  { AMOVB, C_HAUTO,C_NONE, C_REG,  71, 4, REGSP, V4 },
  { AMOVB, 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 },


  { AMOVB, C_LEXT, C_NONE, C_REG,  73, 8, REGSB, LFROM|V4 },
```

```
  { AMOVB, C_LAUTO,C_NONE, C_REG,  73, 8, REGSP, LFROM|V4 },
  { AMOVB, 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 202i, C_HEXT 202f, C_HOREG 202l, C_LAUTO 111b, C_LEXT 110b, C_LOREG 109a, C_NONE 95c, C_PSR 201b, C_RCON 107e, C_REG 95c, LFROM 135d, LTO 135d, and V4 202c.

⟨asmout() *switch on type cases* 204a⟩+≡ (102e) ◁201f 204b▷

```
  /* ArmV4 ops: */
  case 70: /* movh/movhu R,O(R) -> strh */
      aclass(&p->to);
      r = p->to.reg;
      ⟨asmout() adjust maybe r to rule param 126a⟩
      o1 = oshr(p->from.reg, instoffset, r, p->scond);
      break;
```

Uses aclass(), instoffset 107d, and oshr() 205b.

⟨asmout() *switch on type cases* 204b⟩+≡ (102e) ◁204a 204c▷

```
  case 71: /* movb/movh/movhu O(R),R -> ldrsb/ldrsh/ldrh */
      aclass(&p->from);
      r = p->from.reg;
      ⟨asmout() adjust maybe r to rule param 126a⟩
      o1 = olhr(instoffset, r, p->to.reg, p->scond);
      if(p->as == AMOVB)
          o1 ^= (1<<5)|(1<<6);
      else if(p->as == AMOVH)
          o1 ^= (1<<6);
      break;
```

Uses aclass(), instoffset 107d, and olhr() 205a.

⟨asmout() *switch on type cases* 204c⟩+≡ (102e) ◁204b 204d▷

```
  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 126a⟩
      o2 = oshrr(p->from.reg, REGTMP,r, p->scond);
      break;
```

Uses omvl() 120b and oshrr() 205c.

⟨asmout() *switch on type cases* 204d⟩+≡ (102e) ◁204c 206b▷

```
  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 126a⟩
      o2 = olhrr(REGTMP, r, p->to.reg, p->scond);
      if(p->as == AMOVB)
          o2 ^= (1<<5)|(1<<6);
      else if(p->as == AMOVH)
          o2 ^= (1<<6);
      break;
```

Uses olhrr() 205d and omvl() 120b.

⟨*function* `olhr` *(arm)* 205a⟩≡ (281)

```
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` 33c and `diag()` 222d.

⟨*function* `oshr` *(arm)* 205b⟩≡ (281)

```
long
oshr(int r, long v, int b, int sc)
{
    long o;

    o = olhr(v, b, r, sc) ^ (1<<20);
    return o;
}
```

Uses `olhr()` 205a.

⟨*function* `oshrr` *(arm)* 205c⟩≡ (281)

```
long
oshrr(int r, int i, int b, int sc)
{
    return olhr(i, b, r, sc) ^ ((1<<22) | (1<<20));
}
```

Uses `olhr()` 205a.

⟨*function* `olhrr` *(arm)* 205d⟩≡ (281)

```
long
olhrr(int i, int b, int r, int sc)
{
    return olhr(i, b, r, sc) ^ (1<<22);
}
```

Uses `olhr()` 205a.

12.6.7 Shifted moves

⟨optab *entries* 206a⟩+≡ (94a) ◁203d 206f▷
```
  { 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_LOREG 109a, C_NONE 95c, C_REG 95c, C_SHIFT 116b, LFROM 135d, and V4 202c.

⟨asmout() *switch on type cases* 206b⟩+≡ (102e) ◁204d 206c▷
```
  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() 222d and olrr() 127e.

⟨asmout() *switch on type cases* 206c⟩+≡ (102e) ◁206b 206d▷
```
  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() 222d and olhrr() 205d.

⟨asmout() *switch on type cases* 206d⟩+≡ (102e) ◁206c 207a▷
```
  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() 222d and osrr() 128e.

12.7 Compiler-only pseudo opcodes

⟨buildop() *switch opcode r for ranges cases* 206e⟩+≡ (98d) ◁200f
```
  case ACASE:
  case ABCASE:
      break;
```

⟨optab *entries* 206f⟩+≡ (94a) ◁206a
```
  { ACASE, C_REG, C_NONE, C_NONE,  62, 4 },
  { ABCASE, C_NONE, C_NONE, C_BRANCH,  63, 4 },
```
Uses C_NONE 95c, C_REG 95c, and C_SHIFT 116b.

⟨asmout() *switch on type cases* 207a⟩+≡ (102e) ◁206d 207b▷
```
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() 127e.

⟨asmout() *switch on type cases* 207b⟩+≡ (102e) ◁207a
```
case 63: /* bcase */
    if(p->cond != P) {
        o1 = p->cond->pc;
        if(dlm)
            dynreloc(S, p->pc, 1);
    }
    break;
```
Uses P 32f, S 29d, dlm 164a, and dynreloc() 173b.

12.8 5l -E digits

⟨main() *if rare condition do not set SXREF for INITENTRY, else* 207c⟩+≡ (38c) ◁70e
```
if(*INITENTRY >= '0' && *INITENTRY <= '9') {}
else
```

⟨entryvalue() *if digit INITENTRY* 207d⟩≡ (42b)
```
if(*a >= '0' && *a <= '9')
    return atolwhex(a);
```
Uses INITENTRY 38a.

12.9 Strings in text segment: 5l -t

⟨Section *cases* 207e⟩+≡ (29d) ◁176e
```
SSTRING, // arm
```

⟨dodata() *if string in text segment* 207f⟩≡ (87c)
```
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 32f, SSTRING 32c, datap 33a, and debug 211a.

⟨dotext() *other locals* 207g⟩+≡ (89b) ◁90a
```
Sym *s;
long v;
int i;
```

⟨dotext() *if string in text segment* 208a⟩≡ (89b)

```
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 29d, SSTRING 32c, debug 211a, and rnd() 229a.

⟨asmb() *locals* 208b⟩+≡ (41a) ◁??

```
long etext;
```

⟨asmb() *Text section, output strings in text segment* 208c⟩≡ (42c)

```
/* 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 36e, buf 227b, datblk() 44d, pc 49a, and textsize 89a.

⟨datblk() *if sstring might continue* 208d⟩≡ (44d)

```
if(sstring != (p->from.sym->type == SSTRING))
    continue;
```

Uses SSTRING 32c.

⟨asmsym() *in symbol table iteration, switch section cases* 208e⟩+≡ (142) ◁152b

```
case SSTRING:
    putsymb(s->name, 'T', s->value, s->version);
    continue;
```

Uses SSTRING 32c.

Chapter 13

Conclusion

You now know how the Plan 9 ARM linker `5l` 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 `5l` 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 `oprange` 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 `5l` 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 `5l`. The kernel's boot sequence relies on the memory layout that `5l` 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 `5l` 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 `5l`. 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. `5l`'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*: `5l` 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. `5l` 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. `mold`, a newer linker, pushes this further with aggressive parallelism, linking large binaries in under a second.
- *Debug information*: `5l` 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. `5l` 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` 211a⟩≡ (276b)
```
bool debug[128];
```

⟨`main()` *command line processing(arm)* 211b⟩+≡ (34e) ◁182f
```
default:
    c = ARGC();
    if(c >= 0 && c < sizeof(debug))
        debug[c]++;
    break;
```

⟨*global* `bso` 211c⟩≡ (276b)
```
Biobuf bso;
```

⟨`main()` *debug initialization(arm)* 211d⟩≡ (34e)
```
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)* 211e⟩≡ (278b)
```
void
listinit(void)
{

    fmtinstall('A', Aconv);
    fmtinstall('C', Cconv);
    fmtinstall('D', Dconv);
    fmtinstall('N', Nconv);
    fmtinstall('P', Pconv);
    fmtinstall('S', Sconv);
}
```
Uses `Aconv()` 213a, `Cconv()` 216b, `Dconv()` 213b, and `Nconv()` 215.

⟨*pragmas varargck type* 212a⟩≡ (272)

```
#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)* 212b⟩≡ (278b)

```
void
prasm(Prog *p)
{
    print("%P\n", p);
}
```

⟨*constant* `STRINGSZ` 212c⟩≡ (271c)

```
STRINGSZ    = 200,
```

⟨*function* `Pconv`*(arm)* 212d⟩≡ (278b)

```
// 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:
        sprint(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 += sprint(s, "(%ld)", p->line);
        if(p->reg == R_NONE)
            sprint(s, " %A%C %D,%D",
                a, p->scond, &p->from, &p->to);
        else
        if(p->from.type != D_FREG)
            sprint(s, " %A%C %D,R%d,%D",
                a, p->scond, &p->from, p->reg, &p->to);
        else
            sprint(s, " %A%C %D,F%d,%D",
                a, p->scond, &p->from, p->reg, &p->to);
        break;

    }
```

```
    return fmtstrcpy(fp, str);
}
```

Uses STRINGSZ 225d and curp 33c.

A.1.2 Opcode dumper: Aconv()

⟨*function* Aconv*(arm)* 213a⟩≡ (278b)

```
// 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)* 213b⟩≡ (278b)

```
// 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)
            sprint(str, "%N(R%d)(NONE)", a, a->reg);
        break;

    case D_CONST:
        sprint(str, "$%N", a);
        break;

    case D_ADDR:
        if(a->reg == R_NONE)
            sprint(str, "$%N", a);
        else
            sprint(str, "$%N(R%d)", a, a->reg);
        break;

    case D_SHIFT:
        v = a->offset;
        op = "<<>>->@>" + (((v>>5) & 3) << 1);
        if(v & (1<<4))
            sprint(str, "R%ld%c%cR%ld", v&15, op[0], op[1], (v>>8)&15);
```

```
    else
        sprint(str, "R%ld%c%c%ld", v&15, op[0], op[1], (v>>7)&31);
    if(a->reg != R_NONE)
        sprint(str+strlen(str), "(R%d)", a->reg);
    break;

case D_OREG:
    if(a->reg != R_NONE)
        sprint(str, "%N(R%d)", a, a->reg);
    else
        sprint(str, "%N", a);
    break;

case D_REG:
    sprint(str, "R%d", a->reg);
    if(a->symkind != N_NONE || a->sym != S)
        sprint(str, "%N(R%d)(REG)", a, a->reg);
    break;

case D_REGREG:
    sprint(str, "(R%d,R%d)", a->reg, (int)a->offset);
    if(a->symkind != N_NONE || a->sym != S)
        sprint(str, "%N(R%d)(REG)", a, a->reg);
    break;

case D_FREG:
    sprint(str, "F%d", a->reg);
    if(a->symkind != N_NONE || a->sym != S)
        sprint(str, "%N(R%d)(REG)", a, a->reg);
    break;

case D_PSR:
    switch(a->reg) {
    case 0:
        sprint(str, "CPSR");
        break;
    case 1:
        sprint(str, "SPSR");
        break;
    default:
        sprint(str, "PSR%d", a->reg);
        break;
    }
    if(a->symkind != N_NONE || a->sym != S)
        sprint(str, "%N(PSR%d)(REG)", a, a->reg);
    break;

case D_FPCR:
    switch(a->reg){
    case 0:
        sprint(str, "FPSR");
        break;
    case 1:
        sprint(str, "FPCR");
        break;
    default:
        sprint(str, "FCR%d", a->reg);
        break;
    }
    if(a->symkind != N_NONE || a->sym != S)
```

```
            sprint(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)
                sprint(str, "{%s}%.5lux(BRANCH)", a->sym->name, v);
            else
                sprint(str, "%.5lux(BRANCH)", v);
        } else
            if(a->sym != S)
                sprint(str, "{%s}%ld(APC)", a->sym->name, a->offset);
            else
                sprint(str, "%ld(APC)", a->offset);
        break;

    case D_FCONST:
        sprint(str, "$%e", ieeedtod(a->ieee));
        break;

    case D_SCONST:
        sprint(str, "$\"%S\"", a->sval);
        break;

    default:
        sprint(str, "GOK-type(%d)", a->type);
        break;

    }
    return fmtstrcpy(fp, str);
}
```

Uses P 32f, S 29d, STRINGSZ 225d, curp 33c, and ieeedtod() 183c.

A.1.4 Symbol kind dumper: Nconv()

⟨*function* Nconv*(arm)* 215⟩≡ (278b)

```
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:
        sprint(str, "%ld", a->offset);
        break;

    case N_EXTERN:
        if(s == S)
            sprint(str, "%ld(SB)", a->offset);
        else
            sprint(str, "{%s}%.5lux+%ld(SB)", s->name, s->value, a->offset);
        break;
```

```
        case N_INTERN:
            if(s == S)
                sprint(str, "<>+%ld(SB)", a->offset);
            else
                sprint(str, "{%s<>}%.5lux+%ld(SB)", s->name, s->value, a->offset);
            break;

        case N_LOCAL:
            if(s == S)
                sprint(str, "%ld(SP)", a->offset);
            else
                sprint(str, "{%s}-%ld(SP)", s->name, -a->offset);
            break;

        case N_PARAM:
            if(s == S)
                sprint(str, "%ld(FP)", a->offset);
            else
                sprint(str, "{%s}%ld(FP)", s->name, a->offset);
            break;
        default:
            sprint(str, "GOK-name(%d)", a->symkind);
            break;

        }
        return fmtstrcpy(fp, str);
    }
```

Uses S 29d and STRINGSZ 225d.

A.1.5 Conditional execution dumper: Cconv()

⟨*global* strcond*(arm)* 216a⟩≡ (278b)

```
    char* strcond[16] =
    {
        ".EQ",
        ".NE",
        ".HS",
        ".LO",
        ".MI",
        ".PL",
        ".VS",
        ".VC",
        ".HI",
        ".LS",
        ".GE",
        ".LT",
        ".GT",
        ".LE",
        "",
        ".NV"
    };
```

⟨*function* Cconv*(arm)* 216b⟩≡ (278b)

```
    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 216a.

A.1.6 String dumper: Sconv()

⟨*function* Sconv*(arm)* 217⟩≡ (278b)

```
// 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 225d.

A.2 Verbose mode: 5l -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 218a⟩≡ (272)
```
#define DBG if(debug['v']) mylog
```

⟨*function* log 218b⟩≡ (277b)
```
void mylog(char *fmt, ...) {

    va_list arg;

    va_start(arg, fmt);
    Bvprint(&bso, fmt, arg);
    va_end(arg);
    Bflush(&bso);
}
```
Uses bso 211c.

A.3 Objects loading debugging: 5l -W

⟨ldobj() *debug* 218c⟩≡ (51)
```
if(debug['W'])
    print("%P\n", p);
```
Uses diag() 222d.

⟨ldobj() *when ANAME, debug* 218d⟩≡ (56d)
```
if(debug['W'])
    print(" ANAME %s\n", s->name);
```

A.4 Opcode table debugging: 5l -O

⟨oplook() *debug* 218e⟩≡ (97a)
```
if(debug['O']) {
    print("oplook %P %A %d %d %d\n",
        p, (int)p->as, a1, a2, a3);
}
```
Uses debug 211a.

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* 218f⟩+≡ (102e) ◁200c
```
// pc (real)
long v;
```

⟨asmout() *debug* 219a⟩≡ (102e)
```
if(debug['a'] > 1)
    Bprint(&bso, "%2d ", o->type);
v = p->pc; // for debugging later
```
Uses bso 211c and debug 211a.

⟨asmout() *when 1 generated instruction, debug* 219b⟩≡ (103)
```
if(debug['a'])
    Bprint(&bso, " %.8lux: %.8lux\t%P\n", v, o1, p);
```
Uses bso 211c and debug 211a.

⟨asmout() *when 2 generated instructions, debug* 219c⟩≡ (103)
```
if(debug['a'])
    Bprint(&bso, " %.8lux: %.8lux %.8lux%P\n", v, o1, o2, p);
```
Uses bso 211c and debug 211a.

⟨asmout() *when 3 generated instructions, debug* 219d⟩≡ (103)
```
if(debug['a'])
    Bprint(&bso, " %.8lux: %.8lux %.8lux %.8lux%P\n", v, o1, o2, o3, p);
```
Uses bso 211c and debug 211a.

⟨asmout() *when 4 generated instructions, debug* 219e⟩≡ (103)
```
if(debug['a'])
    Bprint(&bso, " %.8lux: %.8lux %.8lux %.8lux %.8lux%P\n",
        v, o1, o2, o3, o4, p);
```
Uses bso 211c and debug 211a.

⟨asmout() *when 5 generated instructions, debug* 219f⟩≡ (103)
```
if(debug['a'])
    Bprint(&bso, " %.8lux: %.8lux %.8lux %.8lux %.8lux %.8lux%P\n",
        v, o1, o2, o3, o4, o5, p);
```
Uses bso 211c and debug 211a.

⟨asmout() *when 6 generated instructions, debug* 219g⟩≡ (103)
```
if(debug['a'])
    Bprint(&bso, " %.8lux: %.8lux %.8lux %.8lux %.8lux %.8lux %.8lux%P\n",
        v, o1, o2, o3, o4, o5, o6, p);
```
Uses bso 211c and debug 211a.

⟨asmout() *when other size, debug* 219h⟩≡ (103)
```
if(debug['a'])
    Bprint(&bso, " %.8lux:\t\t%P\n", v, p);
```
Uses bso 211c and debug 211a.

⟨asmb() *before cflush, debug* 219i⟩≡ (42c)
```
if(debug['a']) {
    Bprint(&bso, "\n");
    Bflush(&bso);
}
```
Uses bso 211c and debug 211a.

A.6 Symbol table debugging: `5l -n`

⟨`putsymb()` *debug* 220a⟩≡ (141c)
```
if(debug['n']) {
    ⟨putsymb() if z or Z in debug output 220b⟩
    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` 211c.

⟨`putsymb()` *if z or Z in debug output* 220b⟩≡ (220a)
```
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` 211c and `debug` 211a.

A.7 Line table debugging: `5l -V`

⟨`asmlc()` *dump instruction p, debug* 220c⟩≡ (154)
```
if(debug['V'])
    Bprint(&bso, "%6lux %P\n", p->pc, p);
```

⟨`asmlc()` *dump lcsize, debug* 220d⟩≡ (154)
```
if(debug['V'])
    Bprint(&bso, "\t\t%6ld", lcsize);
```

⟨`asmlc()` *dump s, debug* 220e⟩≡ (154)
```
if(debug['V'])
    Bprint(&bso, " pc+%ld*%d(%ld)", s, MINLC, s+128);
```
Uses `cput()` 227c.

⟨`asmlc()` *dump big line change, debug* 220f⟩≡ (154)
```
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` 211c, `cput()` 227c, and `debug` 211a.

⟨`asmlc()` *dump small line increment, debug* 220g⟩≡ (154)
```
if(debug['V']) {
    Bprint(&bso, " lc+%ld(%ld)\n", s, 0+s);
    Bprint(&bso, "%6lux %P\n",
        p->pc, p);
}
```
Uses `bso` 211c, `cput()` 227c, and `debug` 211a.

⟨`asmlc()` *dump negative line increment, debug* 221⟩≡ (154)

```
if(debug['V']) {
    Bprint(&bso, " lc%ld(%ld)\n", s, 64-s);
    Bprint(&bso, "%6lux %P\n",
        p->pc, p);
}
```

Uses `bso` 211c, `cput()` 227c, and `debug` 211a.

Appendix B

Error Management

`5l` 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` 222a⟩≡ (278a)
```
int nerrors = 0;
```
Uses `nerrors` 222a.

⟨*function* `errorexit` 222b⟩≡ (278a)
```
void
errorexit(void)
{

    if(nerrors) {
        if(cout >= 0)
            remove(outfile);
        exits("error");
    }
    exits(0);
}
```
Uses `cout` 34d, `nerrors` 222a, and `outfile` 34c.

⟨*pragmas varargck argpos* 222c⟩≡ (272)
```
#pragma varargck    argpos  diag 1
```

⟨*function* `diag` 222d⟩≡ (278a)
```
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");
        errorexit();
    }
```

```
}
```

Uses P 32f, S 29d, STRINGSZ 225d, curtext 33b, debug 211a, errorexit() 222b, and nerrors 222a.

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* 224a⟩≡ (34e)
```
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* 224b⟩≡ (276b)
```
long nsymbol;
```

⟨`lookup()` *profiling* 224c⟩≡ (29b)
```
nsymbol++;
```

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 `5l` relies on this property.

⟨*global* `hunk` 225a⟩≡ (277b)
```
char* hunk;
```

⟨*global* `nhunk` 225b⟩≡ (277b)
```
long nhunk;
```

⟨*global* `thunk` 225c⟩≡ (276b)
```
long thunk;
```

⟨*constant NHUNK linker* 225d⟩≡ (271c)
```
NHUNK       = 100000,
```

⟨*function* `gethunk` 225e⟩≡ (277b)
```
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");
        errorexit();
    }
```

```
    hunk = h;
    nhunk = nh;
    thunk += nh;
}
```

Uses NHUNK, diag() 222d, errorexit() 222b, hunk, nhunk 277b, and thunk 225c.

⟨*function* malloc 226a⟩≡ (277b)

```
/*
 * 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() 225e, hunk, and nhunk 277b.

⟨*function* free 226b⟩≡ (277b)

```
void
free(void *p)
{
    USED(p);
}
```

⟨*function* setmalloctag 226c⟩≡ (277b)

```
//@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 226d⟩≡ (272)

```
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` 227a⟩≡ (271c)
```
MAXIO       = 8192,
```

⟨*global* `buf` 227b⟩≡ (282a)
```
union Buf buf;
```
Uses `Buf` 226d.

D.2.1 Output buffer

⟨*function* `cput`*(arm)* 227c⟩≡ (282a)
```
void
cput(int c)
{
    cbp[0] = c;
    cbp++;
    cbc--;
    if(cbc <= 0)
        cflush();
}
```
Uses `cbc` 104b, `cbp` 104a, and `cflush()` 104e.

⟨*function* `wputl`*(arm)* 227d⟩≡ (282a)
```
void
wputl(long l)
{

    cbp[0] = l;
    cbp[1] = l>>8;
    cbp += 2;
    cbc -= 2;
    if(cbc <= 0)
        cflush();
}
```
Uses `cbc` 104b, `cbp` 104a, and `cflush()` 104e.

⟨*function* `wput`*(arm)* 227e⟩≡ (282a)
```
void
wput(long l)
{

    cbp[0] = l>>8;
    cbp[1] = l;
    cbp += 2;
    cbc -= 2;
    if(cbc <= 0)
        cflush();
}
```
Uses `cbc` 104b, `cbp` 104a, and `cflush()` 104e.

⟨*function* `strnput` *(arm)* 228a⟩≡ (282a)

```
void
strnput(char *s, int n)
{
    for(; *s; s++){
        cput(*s);
        n--;
    }
    for(; n > 0; n--)
        cput(0);
}
```

Uses `cput()` 227c.

D.2.2 Input buffer

D.3 File management

⟨*function* `fileexists` 228b⟩≡ (277b)

```
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()`[228c] ("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` 228c⟩≡ (277b)

```
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()`[229a] 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` 229a⟩≡ (277b)

```
long
rnd(long v, long r)
{
    long c;

    ⟨rnd() if r is null or negative 229b⟩
    v += r - 1;
    c = v % r;
    ⟨rnd() if v was negative 229c⟩
    v -= c;
    return v;
}
```

⟨`rnd()` *if r is null or negative* 229b⟩≡ (229a)

```
if(r <= 0)
    return v;
```

⟨`rnd()` *if v was negative* 229c⟩≡ (229a)

```
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` 230a⟩≡ (271a)

```
int
size(char *file)
{
    fdt fd;
    Fhdr f;

    if((fd = open(file, OREAD)) < 0){
        fprint(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;
    }
    fprint(2, "size: %s not an a.out\n", file);
    close(fd);
    return 1;
}
```

⟨*function main (linkers/misc/size.c)* 230b⟩≡ (271a)

```
void
main(int argc, char *argv[])
{
    char *err;
    int i;

    ARGBEGIN {
    default:
        fprint(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* 231a⟩≡ (270)
```
enum{
    CHUNK = 256 /* must be power of 2 */
};
```

⟨*global* errs 231b⟩≡ (270)
```
static char *errs;   /* exit status */
```

⟨*global* filename 231c⟩≡ (270)
```
static char *filename;  /* current file */
```

⟨*global* symname 231d⟩≡ (270)
```
static char symname[]="__.SYMDEF"; /* table of contents file name */
```
Uses symname-36 231d.

⟨*global* multifile 231e⟩≡ (270)
```
static bool multifile;  /* processing multiple files */
```

⟨*global aflag (linkers/misc/nm.c)* 231f⟩≡ (270)
```
static int aflag;
```

⟨*global* gflag 231g⟩≡ (270)
```
static int gflag;
```

⟨*global* hflag 231h⟩≡ (270)
```
static int hflag;
```

⟨*global* nflag 231i⟩≡ (270)
```
static int nflag;
```

⟨*global* sflag 231j⟩≡ (270)
```
static int sflag;
```

⟨*global uflag (linkers/misc/nm.c)* 231k⟩≡ (270)
```
static int uflag;
```

⟨*global* Tflag 231l⟩≡ (270)
```
static int Tflag;
```

⟨*global* fnames 231m⟩≡ (270)
```
static Sym **fnames;  /* file path translation table */
```

⟨*global* symptr 231n⟩≡ (270)
```
static Sym **symptr;
```

⟨*global* nsym 232a⟩≡ (270)
```
static int nsym;
```

⟨*global bout (linkers/misc/nm.c)* 232b⟩≡ (270)
```
static Biobuf bout;
```

⟨*function usage (linkers/misc/nm.c)* 232c⟩≡ (270)
```
static void
usage(void)
{
    fprint(STDERR, "usage: nm [-aghnsTu] file ...\n");
    exits("usage");
}
```

⟨*function main (linkers/misc/nm.c)* 232d⟩≡ (270)
```
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 (isar(bin))
            doar(bin);
        else{
            Bseek(bin, 0, SEEK__START);
            dofile(bin);
        }
        Bterm(bin);
    }
    exits(errs);
}
```

⟨*function* doar 232e⟩≡ (270)
```
/*
 * 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 231c, multifile-37 231e, nsym-47 232a, printsyms() 236a, psym() 235, symname-36 231d, and symptr-46 231n.

⟨*function* dofile 233a⟩≡ (270)

```
/*
 * 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() 234b, nsym-47 232a, printsyms() 236a, psym() 235, and symptr-46 231n.

⟨*function* cmp_symbol 233b⟩≡ (270)

```
/*
```

```
 * 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 231i.

⟨*function* zenter 234a⟩≡ (270)

```
/*
 * 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 231a and fnames-45 231m.

⟨*function* execsyms 234b⟩≡ (270)

```
/*
 * 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 231c, nsym-47 232a, printsyms() 236a, psym() 235, and symptr-46 231n.

⟨*function* psym 235⟩≡ (270)

```
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 231f, gflag-39 231g, nsym-47 232a, symptr-46 231n, uflag-43 231k, and zenter() 234a.

⟨*function* printsyms 236a⟩≡ (270)

```
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 231l, bout-48 232b, cmp_symbol() 233b, filename-35 231c, fnames-45 231m, hflag-40 231h, multifile-37 231e, and sflag-42 231j.

⟨*function* error 236b⟩≡ (270)

```
static void
error(char *fmt, ...)
{
    Fmt f;
    char buf[128];
    va_list arg;

    fmtfdinit(&f, 2, buf, sizeof buf);
    fmtprint(&f, "%s: ", argv0);
    va_start(arg, fmt);
    fmtvprint(&f, fmt, arg);
    va_end(arg);
    fmtprint(&f, "\n");
    fmtfdflush(&f);
```

```
    errs = "errors";
}
```

Uses errs-34 231b.

E.4 ar

⟨*struct* Arsymref 237a⟩≡ (266d)

```
typedef struct Arsymref
{
    char *name;
    int type;
    int len;
    vlong offset;
    struct Arsymref *next;
} Arsymref;
```

Uses Arsymref 237a.

⟨*struct* Armember 237b⟩≡ (266d)

```
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 237b.

⟨*struct* Arfile 237c⟩≡ (266d)

```
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 237c.

⟨*struct* Hashchain 237d⟩≡ (266d)

```
typedef struct Hashchain
{
    char *name;
    struct Hashchain *next;
} Hashchain;
```

Uses Hashchain 237d.

⟨*constant* NHASH 237e⟩≡ (266d)

```
#define NHASH 1024
```

⟨*function* HEADER_IO 238a⟩≡ (266d)
```
/*
 * 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 238b⟩≡ (266d)
```
char *man =  "mrxtdpq";
```
Uses man 238b.

⟨*global* opt 238c⟩≡ (266d)
```
char *opt =  "uvnbailo";
```
Uses opt 238c.

⟨*global* artemp 238d⟩≡ (266d)
```
char artemp[] = "/tmp/vXXXXX";
```
Uses artemp 238d.

⟨*global* movtemp 238e⟩≡ (266d)
```
char movtemp[] = "/tmp/v1XXXXX";
```
Uses movtemp 238e.

⟨*global* tailtemp 238f⟩≡ (266d)
```
char tailtemp[] = "/tmp/v2XXXXX";
```
Uses tailtemp 238f.

⟨*global* symdef 238g⟩≡ (266d)
```
char symdef[] = "__.SYMDEF";
```
Uses symdef 238g.

⟨*global* aflag 238h⟩≡ (266d)
```
int aflag;    /* command line flags */
```

⟨*global* bflag 238i⟩≡ (266d)
```
static int bflag;
```

⟨*global* cflag 238j⟩≡ (266d)
```
int cflag;
```

⟨*global* oflag 238k⟩≡ (266d)
```
int oflag;
```

⟨*global* uflag 238l⟩≡ (266d)
```
int uflag;
```

⟨*global* vflag 238m⟩≡ (266d)
```
int vflag;
```

⟨*global* allobj 238n⟩≡ (266d)
```
int allobj = 1;   /* set when all members are object files of the same type */
```
Uses allobj 238n.

⟨*global* `symdefsize` 239a⟩≡ (266d)
```
int symdefsize;   /* size of symdef file */
```

⟨*global* `dupfound` 239b⟩≡ (266d)
```
int dupfound;   /* flag for duplicate symbol */
```

⟨*global* `hash` 239c⟩≡ (266d)
```
Hashchain *hash[NHASH];  /* hash table of text symbols */
```
Uses `NHASH-16` 237e.

⟨*constant* `ARNAMESIZE` 239d⟩≡ (266d)
```
#define ARNAMESIZE sizeof(astart->tail->hdr.name)
```

⟨*global* `poname` 239e⟩≡ (266d)
```
char poname[ARNAMESIZE+1];  /* name of pivot member */
```
Uses `ARNAMESIZE-19` 239d.

⟨*global* `file` 239f⟩≡ (266d)
```
char *file;    /* current file or member being worked on */
```

⟨*global* `bout` 239g⟩≡ (266d)
```
Biobuf bout;
```

⟨*global* `bar` 239h⟩≡ (266d)
```
Biobuf bar;
```

⟨*global* `comfun` 239i⟩≡ (266d)
```
void (*comfun)(char*, int, char**);
```

⟨*function* `main` 239j⟩≡ (266d)
```
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(tcmd); break;
        case 'u': uflag = 1; break;
        case 'v': vflag = 1; break;
        case 'x': setcom(xcmd); break;
        default:
```

```
            fprint(2, "ar: bad option `%c'\n", *cp);
            exits("error");
        }
    }
    if (aflag && bflag) {
        fprint(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) {
            fprint(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) {
            fprint(2, "ar: %s not found\n", *argv);
            cp = "error";
        }
        argv++;
    }
    exits(cp);
}
```

Uses aflag 238h, artemp 238d, bflag-18 238i, bout 239g, cflag 238j, comfun 239i, dcmd() 242, man 238b, mcmd() 244, movtemp 238e, oflag 238k, pcmd() 243b, poname 239e, qcmd() 245b, rcmd() 240b, setcom() 240a, tailtemp 238f, tcmd() 245a, trim() 254a, uflag 238l, vflag 238m, and xcmd() 243a.

⟨*function* setcom 240a⟩≡ (266d)

```
/*
 * select a command
 */
void
setcom(void (*fun)(char *, int, char**))
{

    if(comfun != 0) {
        fprint(2, "ar: only one of [%s] allowed\n", man);
        usage();
    }
    comfun = fun;
}
```

Uses comfun 239i and man 238b.

⟨*function* rcmd 240b⟩≡ (266d)

```
/*
 * 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)
                fprint(2, "ar: cannot open %s\n", file);
            scanobj(&bar, ap, bp->size);
            arcopy(&bar, ap, bp);
            continue;
        }
        d = dirfstat(Bfildes(bfile));
        if(d == nil)
            fprint(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)
            fprint(2, "ar: %s cannot open\n", file);
        else {
            mesg('a', file);
            d = dirfstat(Bfildes(bfile));
            if (d == nil)
                fprint(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 266d, arcopy() 250b, armove() 250a, artemp 238d, astart 266d, bamatch() 253a, bar 239h, cflag 238j, file 239f, free() 226b, getdir() 249d, install() 251a, match() 252b, mesg() 253b, newmember() 256e, newtempfile() 256d, openar() 248a, poname 239e, scanobj() 246, skip() 250c, symdef 238g, tailtemp 238f, and uflag 238l.

⟨*function* dcmd 242⟩≡ (266d)

```
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 238n, arcopy() 250b, artemp 238d, astart 266d, bar 239h, file 239f, getdir() 249d, install() 251a, match() 252b, mesg() 253b, newtempfile() 256d, openar() 248a, scanobj() 246, skip() 250c, and symdef 238g.

⟨*function* xcmd 243a⟩≡ (266d)

```
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) {
                fprint(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() 250b, bar 239h, file 239f, free() 226b, getdir() 249d, match() 252b, mesg() 253b, oflag 238k, openar() 248a, skip() 250c, and wrerr() 248c.

⟨*function* pcmd 243b⟩≡ (266d)

```
void
pcmd(char *arname, int count, char **files)
{
    int fd;
    Armember *bp;
```

```
    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)
                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()` 250b, `bar` 239h, `file` 239f, `free()` 226b, `getdir()` 249d, `match()` 252b, `openar()` 248a, `skip()` 250c, `vflag` 238m, and `wrerr()` 248c.

⟨*function* `mcmd` 244⟩≡ (266d)

```
void
mcmd(char *arname, int count, char **files)
{
    int fd, i;
    Arfile *ap;
    Armember *bp;

    if (count == 0)
        return;
    fd = openar(arname, 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)
```

```
        fprint(2, "ar: %s not found - files moved to end.\n", poname);
    install(arname, astart, amiddle, aend, 0);
}
```

Uses aend 266d, amiddle 266d, arcopy() 250b, artemp 238d, astart 266d, bamatch() 253a, bar 239h, file 239f, getdir() 249d, install() 251a, match() 252b, mesg() 253b, movtemp 238e, newtempfile() 256d, openar() 248a, poname 239e, scanobj() 246, skip() 250c, symdef 238g, and tailtemp 238f.

⟨*function* tcmd 245a⟩≡ (266d)

```
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 239d, bar 239h, bout 239g, file 239f, free() 226b, getdir() 249d, longt() 255d, match() 252b, openar() 248a, skip() 250c, trim() 254a, and vflag 238m.

⟨*function* qcmd 245b⟩≡ (266d)

```
void
qcmd(char *arname, int count, char **files)
{
    int fd, i;
    Armember *bp;
    Biobuf *bfile;

    if(aflag || bflag) {
        fprint(2, "ar: abi not allowed with q\n");
        exits("error");
    }
    fd = openar(arname, ORDWR, 1);
    if (fd < 0) {
        if(!cflag)
            fprint(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(RFNOTEG);
    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)
                fprint(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 238h, arcreate() 248b, armove() 250a, arwrite() 258a, bar 239h, bflag-18 238i, cflag 238j, file 239f, free() 226b, mesg() 253b, newmember() 256e, openar() 248a, and wrerr() 248c.

⟨*function* scanobj 246⟩≡ (266d)

```
    /*
     * 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)
                fprint(2, "ar: zero length file %s\n", file);
            free(d);
            Bseek(b, offset, 0);
            return;
        }
        if (lastobj >= 0 && obj != lastobj) {
            fprint(2, "ar: inconsistent object file %s\n", file);
            allobj = 0;
            Bseek(b, offset, 0);
            return;
        }
        lastobj = obj;
        if (!readar(b, obj, offset+size, 0)) {
            fprint(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 238n, file 239f, free() 226b, and objsym() 247a.

⟨*function* objsym 247a⟩≡ (266d)

```
/*
 * 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;
        fprint(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() 259b, dupfound 239b, duplicate() 247b, free() 226b, and symdefsize 239a.

⟨*function* duplicate 247b⟩≡ (266d)

```
/*
 * 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 237e and armalloc() 259b.

⟨*function* openar 248a⟩≡ (266d)

```
/*
 * 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 || strncmp(mbuf, ARMAG, SARMAG)) {
            fprint(2, "ar: %s not in archive format\n", arname);
            exits("error");
        }
    }else if(!errok){
        fprint(2, "ar: cannot open %s: %r\n", arname);
        exits("error");
    }
    return fd;
}
```

⟨*function* arcreate 248b⟩≡ (266d)

```
/*
 * create an archive and set its header
 */
int
arcreate(char *arname)
{
    int fd;

    fd = create(arname, OWRITE, 0664);
    if(fd < 0){
        fprint(2, "ar: cannot create %s: %r\n", arname);
        exits("error");
    }
    if(write(fd, ARMAG, SARMAG) != SARMAG)
        wrerr();
    return fd;
}
```

Uses wrerr() 248c.

⟨*function* wrerr 248c⟩≡ (266d)

```
/*
 *  error handling
 */
void
wrerr(void)
{
    perror("ar: write error");
    exits("error");
}
```

⟨*function* rderr 249a⟩≡ (266d)
```
void
rderr(void)
{
    perror("ar: read error");
    exits("error");
}
```

⟨*function* phaseerr 249b⟩≡ (266d)
```
void
phaseerr(int offset)
{
    fprint(2, "ar: phase error at offset %d\n", offset);
    exits("error");
}
```

⟨*function* usage 249c⟩≡ (266d)
```
static void
usage(void)
{
    fprint(2, "usage: ar [%s][%s] archive files ...\n", opt, man);
    exits("error");
}
```
Uses man 238b and opt 238c.

⟨*function* getdir 249d⟩≡ (266d)
```
/*
 * 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 239d, HEADER_IO-17 238a, file 239f, free() 226b, newmember() 256e, and phaseerr() 249b.

⟨*function* armove 250a⟩≡ (266d)

```
/*
 * Copy the file referenced by fd to the temp file
 */
void
armove(Biobuf *b, Arfile *ap, Armember *bp)
{
    char *cp;
    Dir *d;

    d = dirfstat(Bfildes(b));
    if (d == nil) {
        fprint(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 = ' ';
    sprint(bp->hdr.date, "%-12ld", d->mtime);
    sprint(bp->hdr.uid, "%-6d", 0);
    sprint(bp->hdr.gid, "%-6d", 0);
    sprint(bp->hdr.mode, "%-8lo", d->mode);
    sprint(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()` 257b, `arread()` 257a, `file` 239f, `free()` 226b, and `trim()` 254a.

⟨*function* arcopy 250b⟩≡ (266d)

```
/*
 * 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()` 257b and `arread()` 257a.

⟨*function* skip 250c⟩≡ (266d)

```
/*
 * Skip an archive member
```

```
 */
void
skip(Biobuf *bp, vlong len)
{
    if (len & 01)
        len++;
    Bseek(bp, len, 1);
}
```

⟨*function* `install` 251a⟩≡ (266d)

```
/*
 * 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) {
        fprint(2, "%s not changed\n", arname);
        return;
    }
    /* leave note group behind when copying back; i.e. sidestep interrupts */
    rfork(RFNOTEG);

    if(createflag)
        fprint(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` 238n, `arcreate()` 248b, `arfree()` 259a, `arstream()` 257c, `dupfound` 239b, and `rl()` 251b.

⟨*function* `rl` 251b⟩≡ (266d)

```
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++;
    sprint(a.date, "%-12ld", time(0));
    sprint(a.uid, "%-6d", 0);
    sprint(a.gid, "%-6d", 0);
    sprint(a.mode, "%-8lo", 0644L);
    sprint(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 238a, aend 266d, amiddle 266d, astart 266d, symdef 238g, symdefsize 239a, wrerr() 248c, and wrsym() 252a.

⟨*function* wrsym 252a⟩≡ (266d)

```
/*
 * 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() 248c.

⟨*function* match 252b⟩≡ (266d)

```
/*
```

```
 * Check if the archive member matches an entry on the command line.
 */
int
match(int count, char **files)
{
    int i;
    char name[ARNAMESIZE+1];

    for(i=0; i<count; i++) {
        if(files[i] == 0)
            continue;
        trim(files[i], name, ARNAMESIZE);
        if(strncmp(name, file, ARNAMESIZE) == 0) {
            file = files[i];
            files[i] = 0;
            return 1;
        }
    }
    return 0;
}
```

Uses ARNAMESIZE-19 239d, file 239f, and trim() 254a.

⟨*function* bamatch 253a⟩≡ (266d)

```
/*
 * 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, ARNAMESIZE) == 0)
                state = 1;
        } else if (bflag) {
            if (strncmp(file, pivot, ARNAMESIZE) == 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 ARNAMESIZE-19 239d, aflag 238h, and bflag-18 238i.

⟨*function* mesg 253b⟩≡ (266d)

```
/*
 * 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 239g and vflag 238m.

⟨*function* trim 254a⟩≡ (266d)

```
/*
 * 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 254b⟩≡ (266d)

```
/*
 * utilities for printing long form of 't' command
 */
#define SUID 04000
```

⟨*constant* SGID 254c⟩≡ (266d)

```
#define SGID 02000
```

⟨*constant* ROWN 254d⟩≡ (266d)

```
#define ROWN 0400
```

⟨*constant* WOWN 254e⟩≡ (266d)

```
#define WOWN 0200
```

⟨*constant* XOWN 254f⟩≡ (266d)

```
#define XOWN 0100
```

⟨*constant* RGRP 254g⟩≡ (266d)

```
#define RGRP 040
```

⟨*constant* WGRP 254h⟩≡ (266d)

```
#define WGRP 020
```

⟨*constant* XGRP 254i⟩≡ (266d)

```
#define XGRP 010
```

⟨*constant* ROTH 254j⟩≡ (266d)

```
#define ROTH 04
```

⟨*constant* WOTH 255a⟩≡ (266d)
```
#define WOTH 02
```

⟨*constant* XOTH 255b⟩≡ (266d)
```
#define XOTH 01
```

⟨*constant* STXT 255c⟩≡ (266d)
```
#define STXT 01000
```

⟨*function* longt 255d⟩≡ (266d)
```
void
longt(Armember *bp)
{
    char *cp;

    pmode(strtoul(bp->hdr.mode, 0, 8));
    Bprint(&bout, "%3ld/%1ld", strtol(bp->hdr.uid, 0, 0), strtol(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 239g and pmode() 256b.

⟨*global* m1 255e⟩≡ (266d)
```
int m1[] = { 1, ROWN, 'r', '-' };
```
Uses ROWN-23 254d.

⟨*global* m2 255f⟩≡ (266d)
```
int m2[] = { 1, WOWN, 'w', '-' };
```
Uses WOWN-24 254e.

⟨*global* m3 255g⟩≡ (266d)
```
int m3[] = { 2, SUID, 's', XOWN, 'x', '-' };
```
Uses SUID-21 254b and XOWN-25 254f.

⟨*global* m4 255h⟩≡ (266d)
```
int m4[] = { 1, RGRP, 'r', '-' };
```
Uses RGRP-26 254g.

⟨*global* m5 255i⟩≡ (266d)
```
int m5[] = { 1, WGRP, 'w', '-' };
```
Uses WGRP-27 254h.

⟨*global* m6 255j⟩≡ (266d)
```
int m6[] = { 2, SGID, 's', XGRP, 'x', '-' };
```
Uses SGID-22 254c and XGRP-28 254i.

⟨*global* m7 255k⟩≡ (266d)
```
int m7[] = { 1, ROTH, 'r', '-' };
```
Uses ROTH-29 254j.

⟨*global* m8 255l⟩≡ (266d)
```
int m8[] = { 1, WOTH, 'w', '-' };
```
Uses WOTH-30 255a.

⟨*global* m9 255m⟩≡ (266d)
```
int m9[] = { 2, STXT, 't', XOTH, 'x', '-' };
```
Uses STXT-32 255c and XOTH-31 255b.

⟨*global* m 256a⟩≡ (266d)
```
int *m[] = { m1, m2, m3, m4, m5, m6, m7, m8, m9};
```
Uses m1 255e, m2 255f, m3 255g, m4 255h, m5 255i, m6 255j, m7 255k, m8 255l, and m9 255m.

⟨*function* pmode 256b⟩≡ (266d)
```
void
pmode(long mode)
{
    int **mp;

    for(mp = &m[0]; mp < &m[9];)
        select(*mp++, mode);
}
```
Uses m 256a and select() 256c.

⟨*function* select 256c⟩≡ (266d)
```
void
select(int *ap, long mode)
{
    int n;

    n = *ap++;
    while(--n>=0 && (mode & (*ap++))==0)
        ap++;
    Bputc(&bout, *ap);
}
```
Uses bout 239g.

⟨*function* newtempfile 256d⟩≡ (266d)
```
/*
 * 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() 259b.

⟨*function* newmember 256e⟩≡ (266d)
```
Armember *
newmember(void)   /* allocate a member buffer */
{
    return (Armember *)armalloc(sizeof(Armember));
}
```
Uses armalloc() 259b.

⟨*function* `aread` 257a⟩≡ (266d)

```
void
aread(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()` 259b, `free()` 226b, and `rderr()` 249a.

⟨*function* `arinsert` 257b⟩≡ (266d)

```
/*
 * 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` 257c⟩≡ (266d)

```
/*
 * 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() 258a, rderr() 249a, and wrerr() 248c.

⟨*function* arwrite 258a⟩≡ (266d)

```
/*
 * 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 238a.

⟨*function* page 258b⟩≡ (266d)

```
/*
 * 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) {
            fprint(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() 258a and free() 226b.

⟨*function* getspace 258c⟩≡ (266d)

```
/*
 * 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(amiddle))
            return 1;
    if (aend && aend->head && page(aend))
            return 1;
    return 0;
}
```

Uses aend 266d, amiddle 266d, astart 266d, and page() 258b.

⟨*function* arfree 259a⟩≡ (266d)

```
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() 226b.

⟨*function* armalloc 259b⟩≡ (266d)

```
/*
 * 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());
    fprint(2, "ar: out of memory\n");
    exits("malloc");
    return 0;
}
```

Uses getspace() 258c and malloc() 226a.

E.5 strip

⟨*function error (linkers/misc/strip.c)* 259c⟩≡ (271b)

```
void
error(char* fmt, ...)
{
    va_list arg;
    char *e, s[256];

    va_start(arg, fmt);
    e = seprint(s, s+sizeof(s), "%s: ", argv0);
    e = vseprint(e, s+sizeof(s), fmt, arg);
    e = seprint(e, s+sizeof(s), "\n");
    va_end(arg);

    write(2, s, e-s);
}
```

⟨*function usage (linkers/misc/strip.c)* 260a⟩≡ (271b)

```
static void
usage(void)
{
    error("usage: %s -o ofile file\n\t%s file ...\n", argv0, argv0);
    exits("usage");
}
```

⟨*function* strip 260b⟩≡ (271b)

```
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, 0LL, 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() 226b and malloc() 226a.

⟨*function main (linkers/misc/strip.c)* 262⟩≡ (271b)

```
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 263a⟩≡ (263i)
```
#define _MAGIC(f, b) ((f)|((((4*(b))+0)*(b))+7))
```

⟨*constant* I_MAGIC 263b⟩≡ (263i)
```
#define I_MAGIC  _MAGIC(0, 11)  /* intel 386 */
```

⟨*constant* E_MAGIC 263c⟩≡ (263i)
```
#define E_MAGIC  _MAGIC(0, 20)  /* arm */
```

⟨*constant* HDR_MAGIC 263d⟩≡ (263i)
```
#define HDR_MAGIC 0x00008000  /* header expansion */
```

⟨*constant* MIN_MAGIC 263e⟩≡ (263i)
```
#define MIN_MAGIC 11
```

⟨*constant* MAX_MAGIC 263f⟩≡ (263i)
```
#define MAX_MAGIC 20   /* <= 90 */
```

⟨*constant* DYN_MAGIC 263g⟩≡ (263i)
```
#define DYN_MAGIC 0x80000000  /* dlm */
```

⟨struct Sym a.out.h 263h⟩≡ (263i)
```
struct Sym
{
    vlong value;
    uint sig;
    char type;
    char *name;
};
```

⟨include/a.out.h 263i⟩≡
```
typedef struct Exec Exec;
typedef struct Sym Sym;

⟨struct Exec 21⟩

⟨constant HDR_MAGIC 263d⟩

⟨function _MAGIC 263a⟩
```

⟨*constant* I_MAGIC 263b⟩
⟨*constant* E_MAGIC 263c⟩

⟨*constant* MIN_MAGIC 263e⟩
⟨*constant* MAX_MAGIC 263f⟩

⟨*constant* DYN_MAGIC 263g⟩

⟨*struct Sym a.out.h* 263h⟩

F.1.2 include/exec/elf.h

⟨*enum ElfConstants* 264⟩≡ (266b)

```
/* 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 */

    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 265a⟩≡ (266b)

```
/*
 * 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 265b⟩≡ (266b)

```
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 265c⟩≡ (266b)

```
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 266a⟩≡ (266b)
```
#define ELF_MAG  ((0x7f<<24) | ('E'<<16) | ('L'<<8) | 'F')
```

⟨include/exec/elf.h 266b⟩≡

⟨*enum ElfConstants* 264⟩

```
typedef struct Ehdr Ehdr;
typedef struct Phdr Phdr;
typedef struct Shdr Shdr;
```

⟨*struct* Ehdr 265a⟩

⟨*struct* Phdr 265b⟩

⟨*struct* Shdr 265c⟩

⟨*constant* ELF_MAG 266a⟩

F.1.3 include/obj/ar.h

⟨include/ar.h 266c⟩≡
⟨*constant* ARMAG 65a⟩
⟨*constant* SARMAG 65b⟩

⟨*constant* ARFMAG 65c⟩
⟨*constant* SARNAME 65d⟩

⟨*struct* ar_hdr 65e⟩
⟨*constant* SAR_HDR 66a⟩

F.2 linkers/misc/

F.2.1 linkers/misc/ar.c

⟨linkers/misc/ar.c 266d⟩≡
```
/*
 * 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` 237a⟩

⟨*struct* `Armember` 237b⟩

⟨*struct* `Arfile` 237c⟩

⟨*struct* `Hashchain` 237d⟩

⟨*constant* `NHASH` 237e⟩

⟨*function* `HEADER_IO` 238a⟩

```
        /* constants and flags */
```
⟨*global* `man` 238b⟩
⟨*global* `opt` 238c⟩
⟨*global* `artemp` 238d⟩
⟨*global* `movtemp` 238e⟩
⟨*global* `tailtemp` 238f⟩
⟨*global* `symdef` 238g⟩

⟨*global* `aflag` 238h⟩
⟨*global* `bflag` 238i⟩
⟨*global* `cflag` 238j⟩
⟨*global* `oflag` 238k⟩
⟨*global* `uflag` 238l⟩
⟨*global* `vflag` 238m⟩

```
Arfile *astart, *amiddle, *aend; /* Temp file control block pointers */
```

⟨*global* `allobj` 238n⟩
⟨*global* `symdefsize` 239a⟩
⟨*global* `dupfound` 239b⟩
⟨*global* `hash` 239c⟩

⟨*constant* `ARNAMESIZE` 239d⟩

⟨*global* `poname` 239e⟩
⟨*global* `file` 239f⟩
⟨*global* `bout` 239g⟩
⟨*global* `bar` 239h⟩

```
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 239i⟩

⟨*function* main 239j⟩
⟨*function* setcom 240a⟩
⟨*function* rcmd 240b⟩

⟨*function* dcmd 242⟩

⟨*function* xcmd 243a⟩
⟨*function* pcmd 243b⟩
⟨*function* mcmd 244⟩
⟨*function* tcmd 245a⟩
⟨*function* qcmd 245b⟩

⟨*function* scanobj 246⟩

⟨*function* objsym 247a⟩

⟨*function* duplicate 247b⟩

⟨*function* openar 248a⟩

⟨*function* arcreate 248b⟩

⟨*function* wrerr 248c⟩

⟨*function* `rderr` 249a⟩

⟨*function* `phaseerr` 249b⟩

⟨*function* `usage` 249c⟩

⟨*function* `getdir` 249d⟩

⟨*function* `armove` 250a⟩

⟨*function* `arcopy` 250b⟩

⟨*function* `skip` 250c⟩

⟨*function* `install` 251a⟩

⟨*function* `rl` 251b⟩

⟨*function* `wrsym` 252a⟩

⟨*function* `match` 252b⟩

⟨*function* `bamatch` 253a⟩

⟨*function* `mesg` 253b⟩

⟨*function* `trim` 254a⟩

⟨*constant* `SUID` 254b⟩
⟨*constant* `SGID` 254c⟩
⟨*constant* `ROWN` 254d⟩
⟨*constant* `WOWN` 254e⟩
⟨*constant* `XOWN` 254f⟩
⟨*constant* `RGRP` 254g⟩
⟨*constant* `WGRP` 254h⟩
⟨*constant* `XGRP` 254i⟩
⟨*constant* `ROTH` 254j⟩
⟨*constant* `WOTH` 255a⟩
⟨*constant* `XOTH` 255b⟩
⟨*constant* `STXT` 255c⟩

⟨*function* `longt` 255d⟩

⟨*global* `m1` 255e⟩
⟨*global* `m2` 255f⟩
⟨*global* `m3` 255g⟩
⟨*global* `m4` 255h⟩
⟨*global* `m5` 255i⟩
⟨*global* `m6` 255j⟩
⟨*global* `m7` 255k⟩
⟨*global* `m8` 255l⟩
⟨*global* `m9` 255m⟩

⟨*global* `m` 256a⟩

⟨*function* `pmode` 256b⟩

⟨*function* `select` 256c⟩

⟨*function* `newtempfile` 256d⟩

⟨*function* `newmember` 256e⟩

⟨*function* `arread` 257a⟩

⟨*function* `arinsert` 257b⟩

⟨*function* `arstream` 257c⟩

⟨*function* `arwrite` 258a⟩

⟨*function* `page` 258b⟩

⟨*function* `getspace` 258c⟩

⟨*function* `arfree` 259a⟩

⟨*function* `armalloc` 259b⟩

F.2.2 `linkers/misc/nm.c`

⟨`linkers/misc/nm.c` 270⟩≡

```
/*
 * nm.c -- drive nm
 */
#include <u.h>
#include <libc.h>
#include <bio.h>

#include <mach.h>
#include <ar.h>
```

⟨*enum NmConstants* 231a⟩

⟨*global* `errs` 231b⟩
⟨*global* `filename` 231c⟩
⟨*global* `symname` 231d⟩
⟨*global* `multifile` 231e⟩
⟨*global aflag (linkers/misc/nm.c)* 231f⟩
⟨*global* `gflag` 231g⟩
⟨*global* `hflag` 231h⟩
⟨*global* `nflag` 231i⟩
⟨*global* `sflag` 231j⟩
⟨*global uflag (linkers/misc/nm.c)* 231k⟩
⟨*global* `Tflag` 231l⟩

⟨*global* `fnames` 231m⟩
⟨*global* `symptr` 231n⟩
⟨*global* `nsym` 232a⟩
⟨*global bout (linkers/misc/nm.c)* 232b⟩

```
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)* 232c⟩

⟨*function main (linkers/misc/nm.c)* 232d⟩

⟨*function* doar 232e⟩

⟨*function* dofile 233a⟩

⟨*function* cmp_symbol 233b⟩
⟨*function* zenter 234a⟩

⟨*function* execsyms 234b⟩

⟨*function* psym 235⟩

⟨*function* printsyms 236a⟩

⟨*function* error 236b⟩

F.2.3 linkers/misc/size.c

⟨linkers/misc/size.c 271a⟩≡
```
#include <u.h>
#include <libc.h>
#include <bio.h>

#include <mach.h>
```

⟨*function* size 230a⟩

⟨*function main (linkers/misc/size.c)* 230b⟩

F.2.4 linkers/misc/strip.c

⟨linkers/misc/strip.c 271b⟩≡
```
#include <u.h>
#include <libc.h>
#include <bio.h>

#include <mach.h>
```

⟨*function error (linkers/misc/strip.c)* 259c⟩

⟨*function usage (linkers/misc/strip.c)* 260a⟩

⟨*function* strip 260b⟩

⟨*function main (linkers/misc/strip.c)* 262⟩

F.3 linkers/5l/

F.3.1 linkers/5l/l.h

⟨*enum* misc_constant*(arm)* 271c⟩≡ (272)
```
enum misc_constants {
```

```
    ⟨constant BIG 176d⟩

    ⟨constant STRINGSZ 212c⟩
    ⟨constant NHASH linker 28b⟩
    ⟨constant NHUNK linker 225d⟩

    ⟨constant MINSIZ 177a⟩
    ⟨constant MAXIO 227a⟩
    ⟨constant MAXHIST 150e⟩
};

⟨linkers/5l/l.h 272⟩≡
  #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) 30a⟩

  ⟨struct Prog(arm) 31a⟩
  ⟨constant P 32f⟩

  ⟨struct Sym 27⟩
  ⟨constant S 28d⟩

  ⟨enum Section(arm) 29d⟩

  ⟨enum Mark(arm) 32c⟩

  ⟨enum headtype(arm) 36c⟩

  ⟨struct Auto(arm) 143b⟩

  ⟨enum rxxx 169c⟩
  ⟨enum misc_constant(arm) 271c⟩

  ⟨constant SIGNINTERN(arm) 197g⟩

  ⟨constant LIBNAMELEN 69b⟩

  ⟨struct Buf 226d⟩

  //------------------------------------------------------------------------
  // 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    lcsize;

extern  char*   noname;
```
⟨*constant* TNAME*(arm)* 33e⟩

```
// 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 169f⟩

```
// debugging
extern  bool    debug[128];
extern  char*   anames[];

// utils (for statistics)
extern  long    thunk;
extern long nsymbol;
```

⟨*pragmas varargck type* 212a⟩
⟨*pragmas varargck argpos* 222c⟩

```
//------------------------------------------------------------------------
// 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  ieeedtod(Ieee*);
long    ieeedtof(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  errorexit(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 218a⟩

```
// fmt.c (dumpers)
void listinit(void);
void    prasm(Prog*);
```

Uses Adr 272, Auto 272, Prog 272, and Sym 272.

F.3.2 linkers/5l/m.h

⟨linkers/5l/m.h 276a⟩≡

```
typedef struct  Optab   Optab;
typedef struct  Oprange Oprange;

⟨enum Operand_class(arm) 95c⟩

⟨struct Optab(arm) 93b⟩

⟨struct Oprange(arm) 95a⟩

⟨enum Optab_flag(arm) 135d⟩

// globals
extern  Optab   optab[];
extern  long    instoffset;

// span.c
Optab*  oplook(Prog*);

// codegen.c
void    asmout(Prog*, Optab*);
```

Uses Oprange 95a and Optab 93b.

F.3.3 linkers/5l/globals.c

⟨linkers/5l/globals.c 276b⟩≡

```
#include "l.h"
#include "m.h"

⟨global thechar 34a⟩
⟨global thestring 34b⟩

⟨global HEADR 36d⟩
⟨global HEADTYPE 36a⟩
⟨global INITDAT 36g⟩
⟨global INITRND 36f⟩
⟨global INITTEXT 36e⟩
⟨global INITTEXTP 182d⟩
⟨global INITENTRY 38a⟩

⟨global outfile 34c⟩
⟨global cout 34d⟩
⟨global bso 211c⟩

⟨global curauto 144c⟩
⟨global curhist 149a⟩
```

⟨*global* `curp` 33c⟩
⟨*global* `curtext` 33b⟩

⟨*global* `autosize`*(arm)* 60d⟩
⟨*global* `instoffset`*(arm)* 107d⟩

⟨*global* `datap` 33a⟩
⟨*global* `etextp` 145e⟩
⟨*global* `firstp` 32d⟩
⟨*global* `lastp` 32g⟩
⟨*global* `textp` 145d⟩

⟨*global* `debug` 211a⟩

⟨*global* `textsize` 89a⟩
⟨*global* `datsize` 87a⟩
⟨*global* `bsssize` 87b⟩
⟨*global* `symsize` 141b⟩
⟨*global* `lcsize` 153d⟩

⟨*global hash linker* 28a⟩
⟨*global* `pc` 49a⟩
⟨*global* `zprg` 39b⟩

⟨*global* `histfrog` 150d⟩
⟨*global* `histfrogp` 150g⟩
⟨*global* `histgen` 150c⟩

⟨*global* `xrefresolv` 71f⟩

⟨*global* `thunk` 225c⟩
⟨*global nsymbol linker* 224b⟩

⟨*global* `armv4`*(arm)* 202a⟩
⟨*global* `vfp`*(arm)* 185b⟩

⟨*global* `doexp` 164c⟩
⟨*global* `dlm` 164a⟩

⟨*global* `EXPTAB` 165a⟩
⟨*global* `undefp` 169e⟩

F.3.4 `linkers/5l/optab.c`

⟨`linkers/5l/optab.c` 277a⟩≡
```
#include "l.h"
#include "m.h"
```

⟨*global optab (linkers/5l/optab.c)(arm)* 94a⟩

F.3.5 `linkers/5l/utils.c`

⟨`linkers/5l/utils.c` 277b⟩≡

```
#include "l.h"

⟨function log 218b⟩

⟨constructor prg 39a⟩

⟨function lookup 28e⟩

⟨function atolwhex 228c⟩

⟨function rnd 229a⟩

⟨function fileexists 228b⟩

⟨global hunk 225a⟩
⟨global nhunk 225b⟩
// thunk defined in globals.c because also used by main.c for profiling report

⟨function gethunk 225e⟩

⟨function malloc 226a⟩

⟨function free 226b⟩

⟨function setmalloctag 226c⟩
```

F.3.6 linkers/5l/error.c

⟨linkers/5l/error.c 278a⟩≡

```
#include "l.h"

⟨global nerrors 222a⟩

⟨function errorexit 222b⟩

⟨function diag 222d⟩
```

F.3.7 linkers/5l/fmt.c

⟨linkers/5l/fmt.c 278b⟩≡

```
#include "l.h"

⟨function Pconv(arm) 212d⟩

⟨function Aconv(arm) 213a⟩

⟨global strcond(arm) 216a⟩

⟨function Cconv(arm) 216b⟩

⟨function Dconv(arm) 213b⟩

⟨function Nconv(arm) 215⟩

⟨function Sconv(arm) 217⟩
```

⟨*function* `listinit`*(arm)* 211e⟩

⟨*function* `prasm`*(arm)* 212b⟩

Uses `Sconv()` 217.

F.3.8 linkers/5l/layout.c

⟨`linkers/5l/layout.c` 279a⟩≡

```
#include "l.h"
#include "m.h"
```

⟨*global* `pool`*(arm)* 138b⟩

⟨*global* `blitrl`*(arm)* 135a⟩
⟨*global* `elitrl`*(arm)* 135b⟩

```
void checkpool(Prog*);
void  flushpool(Prog*, int);
void    addpool(Prog*, Adr*);
```

⟨*function* `xdefine`*(arm)* 92a⟩

⟨*function* `dodata`*(arm)* 87c⟩

⟨*function* `span`*(arm)* 89b⟩

⟨*function* `checkpool`*(arm)* 138a⟩

⟨*function* `flushpool`*(arm)* 137b⟩

⟨*function* `addpool`*(arm)* 136a⟩

F.3.9 linkers/5l/pass.c

⟨`linkers/5l/pass.c` 279b⟩≡

```
#include "l.h"

// forward decls
void    xfol(Prog*);
```

⟨*function* `brchain`*(arm)* 179b⟩

⟨*function* `relinv`*(arm)* 179c⟩

⟨*function* `follow` 178a⟩

⟨*function* `xfol`*(arm)* 178c⟩

⟨*constant* `LOG` 80b⟩
⟨*function* `mkfwd` 80c⟩

⟨*function* `brloop`*(arm)* 177e⟩

⟨*function* `patch`*(arm)* 77⟩

F.3.10 linkers/5l/datagen.c

⟨`linkers/5l/datagen.c` 280a⟩≡
```
#include "l.h"
```

⟨*constant* `Dbufslop` 44c⟩

⟨*global* `inuxi1` 46b⟩
⟨*global* `inuxi2` 46c⟩
⟨*global* `inuxi4` 46d⟩

⟨*global* `fnuxi4` 187d⟩
⟨*global* `fnuxi8` 187e⟩

⟨*function* `find1` 47a⟩

⟨*function* `nuxiinit`*(arm)* 46f⟩

⟨*function* `datblk`*(arm)* 44d⟩

F.3.11 linkers/5l/dynamic.c

⟨`linkers/5l/dynamic.c` 280b⟩≡
```
#include "l.h"

// forward decls
typedef struct Reloc Reloc;
```

⟨*enum SpanConstants(arm)* 173a⟩

⟨*global* `modemap` 172b⟩

⟨*struct* `Reloc` 171c⟩

⟨*global* `rels` 171d⟩

⟨*global* `imports` 165f⟩
⟨*global* `nimports` 165d⟩
⟨*global* `exports` 165g⟩
⟨*global* `nexports` 165e⟩

⟨*function* `zerosig` 165c⟩

⟨*function* `readundefs` 166a⟩

⟨*function* `undefsym` 170b⟩

⟨*function* `import`*(arm)* 169b⟩

⟨*function* `ckoff` 170d⟩

⟨*function* `newdata`*(arm)* 168a⟩

⟨*function* `export`*(arm)* 166b⟩

⟨*function* `grow` 173c⟩

⟨*function* `dynreloc`*(arm)* 173b⟩

⟨*function* `sput` 172a⟩

⟨*function* `asmdyn` 171f⟩

Uses `Reloc` 171c.

F.3.12 `linkers/5l/codegen.c`

⟨`linkers/5l/codegen.c` 281⟩≡

```
#include "l.h"
#include "m.h"
```

⟨*function* `oprrr`*(arm)* 114d⟩

⟨*function* `opvfprrr`*(arm)* 194b⟩

⟨*function* `opbra`*(arm)* 123a⟩

⟨*function* `olr`*(arm)* 126b⟩

⟨*function* `olhr`*(arm)* 205a⟩

⟨*function* `osr`*(arm)* 128b⟩

⟨*function* `oshr`*(arm)* 205b⟩

⟨*function* `olrr`*(arm)* 127e⟩

⟨*function* `olhrr`*(arm)* 205d⟩

⟨*function* `osrr`*(arm)* 128e⟩

⟨*function* `oshrr`*(arm)* 205c⟩

⟨*function* `ovfpmem`*(arm)* 192f⟩

⟨*function* `ofsr`*(arm)* 189c⟩

⟨*function* `omvl`*(arm)* 120b⟩

⟨*function* `asmout`*(arm)* 102e⟩

F.3.13 linkers/5l/io.c

⟨linkers/5l/io.c 282a⟩≡
```
#include "l.h"

⟨global buf 227b⟩

⟨global cbc 104b⟩
⟨global cbp 104a⟩


⟨function readsome 54d⟩


⟨function strnput(arm) 228a⟩

⟨function cput(arm) 227c⟩

⟨function wput(arm) 227e⟩

⟨function wputl(arm) 227d⟩

⟨function lput(arm) 104f⟩

⟨function lputl(arm) 104d⟩

⟨function cflush 104e⟩
```

F.3.14 linkers/5l/asm.c

⟨linkers/5l/asm.c 282b⟩≡
```
#include "l.h"
#include "m.h"

⟨function entryvalue(arm) 42b⟩

⟨function asmb(arm) 41a⟩
```

F.3.15 linkers/5l/span.c

⟨linkers/5l/span.c 282c⟩≡
```
#include "l.h"
#include "m.h"

⟨global oprange(arm) 95b⟩
⟨global xcmp(arm) 102a⟩

⟨function cmp(arm) 98a⟩

⟨function ocmp(arm) 99b⟩

⟨function buildop(arm) 98d⟩
```

⟨*function* `regoff`*(arm)* 189a⟩

⟨*function* `immrot`*(arm)* 108b⟩

⟨*function* `immaddr`*(arm)* 110a⟩

⟨*function* `immfloat`*(arm)* 184c⟩

⟨*function* `immhalf`*(arm)* 203c⟩

⟨*function* `aclass`*(arm)* 96a⟩

⟨*function* `oplook`*(arm)* 97a⟩

Uses `C_GOK` 95c.

F.3.16 linkers/5l/obj.c

⟨`linkers/5l/obj.c` 283a⟩≡
```
#include "l.h"
#include <ar.h>
```

⟨*global noname linker* 33f⟩
⟨*global symname linker* 66b⟩

⟨*global* `version` 59a⟩

⟨*global* `literal`*(arm)* 185f⟩

⟨*function* `isobjfile` 164e⟩

⟨*function* `inopd`*(arm)* 52d⟩

⟨*function* `collapsefrog` 152a⟩

⟨*function* `nopout` 181f⟩

⟨*function* `ldobj`*(arm)* 51⟩

⟨*function* `objfile` 40a⟩

Uses `diag()` 222d, `histfrog` 150d, `histfrogp` 150g, `symname` 283a, and `version` 283a.

F.3.17 linkers/5l/lib.c

⟨`linkers/5l/lib.c` 283b⟩≡
```
#include "l.h"
```

⟨*global* `library` 71a⟩
⟨*global* `libraryobj` 71c⟩
⟨*global* `libraryp` 71b⟩

⟨*global* `libdir` 68c⟩
⟨*global* `nlibdir` 68d⟩
⟨*global* `maxlibdir` 68e⟩

⟨*function* `addlibpath` 69f⟩

⟨*function* `findlib` 70b⟩

⟨*function* `loadlib` 71e⟩

⟨*function* `addlib` 72b⟩

F.3.18 linkers/5l/noop.c

⟨`linkers/5l/noop.c` 284a⟩≡
```
#include "l.h"
```

⟨*global* `sym_div`*(arm)* 197c⟩
⟨*global* `sym_divu`*(arm)* 197d⟩
⟨*global* `sym_mod`*(arm)* 197e⟩
⟨*global* `sym_modu`*(arm)* 197f⟩

⟨*global* `prog_div`*(arm)* 196a⟩
⟨*global* `prog_divu`*(arm)* 196b⟩
⟨*global* `prog_mod`*(arm)* 197a⟩
⟨*global* `prog_modu`*(arm)* 197b⟩

⟨*function* `noops`*(arm)* 82⟩

⟨*function* `sigdiv`*(arm)* 197h⟩

⟨*function* `divsig`*(arm)* 197i⟩

⟨*function* `sdiv`*(arm)* 198a⟩

⟨*function* `initdiv`*(arm)* 198b⟩

⟨*function* `nocache`*(arm)* 101f⟩

F.3.19 linkers/5l/float.c

⟨`linkers/5l/float.c` 284b⟩≡
```
#include "l.h"
```

⟨*function* `ieeedtof` 183b⟩

⟨*function* `ieeedtod` 183c⟩

⟨*global* `chipfloats`*(arm)* 186a⟩

⟨*function* `chipfloat`*(arm)* 186b⟩

F.3.20 linkers/5l/profile.c

⟨`linkers/5l/profile.c` 284c⟩≡
```
#include "l.h"
```

⟨*function* `doprof1`*(arm)* 157⟩

⟨*global* `brcond`*(arm)* 162b⟩

⟨*function* `doprof2`*(arm)* 160a⟩

F.3.21 linkers/5l/debugging.c

⟨`linkers/5l/debugging.c` 285a⟩≡

```
#include "l.h"
```

⟨*function* `putsymb` 141c⟩

⟨*function* `asmsym`*(arm)* 142⟩

⟨*constant* `MINLC`*(arm)* 153e⟩
⟨*function* `asmlc` 154⟩

F.3.22 linkers/5l/hist.c

⟨`linkers/5l/hist.c` 285b⟩≡

```
#include "l.h"
```

⟨*function* `addhist` 148b⟩

⟨*function* `histtoauto` 149d⟩

F.3.23 linkers/5l/main.c

⟨`linkers/5l/main.c` 285c⟩≡

```
#include "l.h"
```

⟨*function usage, linker* 35a⟩

⟨*function* `undef` 48b⟩

⟨*function* `main`*(arm)* 34e⟩

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.

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-Kauffman, 1999. Available at http://www.iecc.com/linker/. cited page(s) 11, 15

[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) 12, 15, 37, 92, 210

[Pad15a] Yoann Padioleau. *Principia Softwarica: The ARM Assembler 5a*. 2015. cited page(s) 11, 14, 16, 18, 20, 23, 25, 27, 30, 31, 49, 52, 56, 60, 61, 74, 76, 83, 84, 85, 94, 107, 113, 114, 115, 116, 120, 126, 140, 146, 148, 163, 209

[Pad15b] Yoann Padioleau. *Principia Softwarica: The ARM Emulator 5i*. 2015. cited page(s) 11, 31, 114, 115, 123, 124

[Pad16a] Yoann Padioleau. *Principia Softwarica: The Plan 9 C Compiler 5c*. 2016. cited page(s) 64, 86, 209

[Pad16b] Yoann Padioleau. *Principia Softwarica: The Plan 9 Core Libraries*. 2016. cited page(s) 92, 211

[Pad16c] Yoann Padioleau. *Principia Softwarica: The Plan 9 Debuggers and Tracers*. 2016. cited page(s) 21, 140, 146, 210

[Pad26] Yoann Padioleau. *Principia Softwarica: The The Plan 9 Profilers*. 2026. cited page(s) 156, 159, 210

[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