Principia Softwarica: The ARM Assembler 5a

version 0.5

Yoann Padioleau
yoann.padioleau@gmail.com

with code from
Rob Pike

April 28, 2026

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

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

Contents

1 Introduction **7**
- 1.1 Motivations . . . 7
- 1.2 The Plan 9 ARM assembler: `5a` . . . 7
- 1.3 Other assemblers . . . 8
- 1.4 Getting started . . . 9
- 1.5 Requirements . . . 11
- 1.6 About this document . . . 11
- 1.7 Copyright . . . 11
- 1.8 Acknowledgments . . . 11

2 Overview **12**
- 2.1 Assembler principles . . . 12
  - 2.1.1 Assemblers and assembly language . . . 12
  - 2.1.2 Mnemonics, opcodes, and operands . . . 12
  - 2.1.3 Syntax families: Intel, AT&T, and Plan 9 . . . 12
  - 2.1.4 Symbolic addresses . . . 13
  - 2.1.5 The phases of assembly . . . 14
  - 2.1.6 Object code and the linker . . . 14
  - 2.1.7 The death of assembly that never happened . . . 14
- 2.2 `5a` command-line interface . . . 15
- 2.3 Assembling `helloworld.s` . . . 15
  - 2.3.1 Background . . . 15
  - 2.3.2 The program . . . 16
  - 2.3.3 Pseudo instructions . . . 17
  - 2.3.4 Labels . . . 17
  - 2.3.5 Assignments . . . 17
  - 2.3.6 Addressing modes . . . 18
  - 2.3.7 Pseudo registers . . . 18
  - 2.3.8 Call stack . . . 19
  - 2.3.9 System calls . . . 19
  - 2.3.10 Function calls . . . 21
  - 2.3.11 Virtual instructions . . . 21
  - 2.3.12 C calling conventions . . . 22
  - 2.3.13 Data layout . . . 23
- 2.4 The ARM architecture . . . 23
- 2.5 Input assembly language . . . 24
  - 2.5.1 Lexical elements . . . 24
  - 2.5.2 Syntactical elements . . . 25
  - 2.5.3 Opcodes . . . 25
  - 2.5.4 Operands . . . 25

2.5.5 Addressing modes . . . 25
2.5.6 Advanced features . . . 26
2.6 Output object format . . . 26
2.7 Code organization . . . 27
2.8 Software architecture . . . 28
2.9 Book structure . . . 30

3 Core Data Structures **31**
3.1 `Opcode` . . . 31
3.2 `Register` . . . 32
3.3 `Operand` . . . 33
3.4 Instructions . . . 34
3.5 Tokens and `itab` . . . 35
3.6 `Symbols` and `hash` table . . . 38
3.7 `Sym_kind` . . . 42

4 Main Functions **43**
4.1 `main()` . . . 43
4.2 `cinit()` . . . 44
4.3 `assemble()` . . . 45

5 Input **50**
5.1 `pinit()` . . . 50
5.2 File management: `iostack` . . . 50
5.3 Buffer management: `fi` . . . 53
5.4 `GETC()` . . . 53

6 Lexing **56**
6.1 `yylex()` . . . 56
6.2 Peek, seek, and look ahead . . . 57
6.3 Newlines (and semicolons) . . . 58
6.4 Comments . . . 58
6.5 Mnemonics, symbols, and labels . . . 59
6.6 Numbers . . . 60
6.6.1 Decimal numbers . . . 60
6.6.2 Hexadecimal and octal numbers . . . 61
6.6.3 Floating-point numbers . . . 62
6.7 Characters . . . 63
6.7.1 Escaped sequences . . . 63
6.7.2 Triple quotes . . . 65
6.7.3 `escchar()` . . . 65
6.7.4 Escaped newlines . . . 66
6.7.5 `getc()` . . . 67
6.8 Strings . . . 67

7 Preprocessing **69**
7.1 Directives dispatch: `mactab`, and `domacro()` . . . 69
7.2 `#include` . . . 72
7.2.1 Include search path . . . 72
7.2.2 `-I` . . . 73

7.2.3 System paths . . . 73
7.2.4 macinc() . . . 74
7.3 #line . . . 77
7.3.1 Motivations . . . 77
7.3.2 maclin() . . . 78

8 Parsing **81**
8.1 Overview . . . 81
8.2 Instructions . . . 82
8.2.1 Arithmetic and logic . . . 82
8.2.2 Memory . . . 85
8.2.3 Control flow . . . 87
8.2.4 Software interrupt . . . 91
8.3 Label definitions and pc . . . 91
8.4 Operands . . . 92
8.4.1 Registers . . . 93
8.4.2 Immediate constants . . . 94
8.4.3 ARM shifted registers . . . 95
8.4.4 Memory (de)references, pointers . . . 97
8.4.5 Named memory locations, symbols . . . 97
8.4.6 Code references, labels . . . 99
8.5 Pseudo instructions . . . 100
8.5.1 TEXT/GLOBL . . . 100
8.5.2 WORD/DATA . . . 101
8.5.3 END . . . 102
8.6 ARM conditional execution . . . 102
8.7 Special bits . . . 103
8.7.1 Arithmetic instructions and .S . . . 103
8.7.2 Moves and .P/.W . . . 103
8.8 yyparse() . . . 104

9 Object Code Generation **105**
9.1 Object file principles . . . 105
9.2 Object format . . . 106
9.3 Instruction output: outcode() . . . 108
9.4 Operand output: outopd() . . . 109
9.5 Object file symbol table: h and ANAME . . . 110

10 Debugging Support **114**
10.1 Line origin history: Hist . . . 114
10.2 Recording history: linehist() . . . 115
10.3 Displaying history: prfile() . . . 116
10.4 Saving history: outhist() and AHISTORY . . . 119

11 Advanced Topics TODO **121**
11.1 Other assembly language features . . . 121
11.1.1 Constant expressions . . . 121
11.1.2 Symbolic constants . . . 122
11.2 Other instructions and registers . . . 123
11.2.1 Floating-point numbers . . . 123

11.2.2 Multiplication and accumulation . . . 126
11.2.3 64-bits multiplication . . . 127
11.2.4 Moving multiple registers at the same time . . . 127
11.2.5 Program status register . . . 129
11.2.6 Mutual exclusion instructions . . . 129
11.2.7 Coprocessors . . . 129
11.3 Other pseudo opcodes . . . 131
11.3.1 Compiler-only opcodes . . . 131
11.3.2 Linker-only opcodes . . . 131
11.4 `TEXT` attributes . . . 131
11.5 Other preprocessing directives . . . 132
11.5.1 `#define` . . . 132
11.5.2 `#undef` . . . 139
11.5.3 `#ifdef` . . . 140
11.5.4 `#pragma` . . . 141
11.6 `#pragma lib` and automagic linking . . . 141
11.7 Processing multiple files . . . 142

12 Conclusion **144**
12.1 Patterns and techniques . . . 144
12.2 Connections to other books . . . 144
12.3 Beyond the Plan 9 assembler . . . 145

A Debugging **146**
A.1 Line information debugging: `5a -f` . . . 146
A.2 Macro debugging: `5a -m` . . . 146

B Error Management **148**

C Utilities **150**
C.1 Buffer management . . . 150
C.2 Memory management . . . 150

D Examples of Assembly Programs TODO **152**
D.1 `hello.s` and `pwrite.s` . . . 152
D.2 `memset.s` . . . 152
D.3 `div.s` . . . 152
D.4 `main9.s` . . . 153
D.5 `tas.s` . . . 153
D.6 `getcallerpc.s` . . . 153

E Extra Code **154**
E.1 `include/` . . . 154
E.1.1 `include/obj/common.out.h` . . . 154
E.1.2 `include/obj/5.out.h` . . . 154
E.2 `assemblers/misc/` . . . 155
E.2.1 `assemblers/misc/data2s.c` . . . 155
E.3 `assemblers/aa/` . . . 156
E.3.1 `assemblers/aa/aa.h` . . . 156
E.3.2 `assemblers/aa/utils.c` . . . 158
E.3.3 `assemblers/aa/globals.c` . . . 160

E.3.4 assemblers/aa/lookup.c . . . 160
E.3.5 assemblers/aa/error.c . . . 160
E.3.6 assemblers/aa/float.c . . . 161
E.3.7 assemblers/aa/hist.c . . . 161
E.3.8 assemblers/aa/lexbody.c . . . 161
E.3.9 assemblers/aa/macbody.c . . . 166
E.4 assemblers/5a/ . . . 167
E.4.1 assemblers/5a/a.h . . . 167
E.4.2 assemblers/5a/globals.c . . . 167
E.4.3 assemblers/5a/lex.c . . . 168
E.4.4 assemblers/5a/a.y . . . 168
E.4.5 assemblers/5a/obj.c . . . 168
E.4.6 assemblers/5a/main.c . . . 168
E.5 lib_core/libc/arm/ . . . 169
E.5.1 lib_core/libc/arm/div.s . . . 169

Glossary **172**

Indexes **173**

References **176**

Chapter 1

Introduction

The goal of this book is to explain with full details the source code of an assembler.

1.1 Motivations

Why an assembler? Because I think you are a better programmer if you fully understand how things work under the hood, and an assembler is an essential part of any development toolchain. Indeed, compilers for higher level languages such as C usually do not generate directly machine code but instead rely on an assembler (and linker) for their final code-generation step.

Even if most programmers very rarely write assembly code, understanding an assembly language, and so also what an assembler does, is essential to understand low-level fundamental concepts such as binary logic and arithmetic, signed and unsigned integers representation, memory operations, pointers, stack processing, frames, or software interrupts. Understanding assembly is also necessary to understand what a compiler generates. Moreover, most programmers will need at some point to look at generated assembly code (or disassembled code) to optimize code or fix bugs. Finally, some code, especially in the kernel, can not be written in C and has to be written in assembly.

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

- What is the list of all assembly instructions? What can a typical computer do?
- What are the essential features of an assembler? How do those features help compared to writing directly machine code?
- What are the most important assembly constructs? How can they be used to implement high-level constructs in languages such as C?
- How are function calls implemented? What is a "frame"? What is a "frame pointer"? How are recursive functions implemented? What are "calling conventions"?
- What does an object file contain? What are the differences between an object file and an executable?
- How do the assembler and linker work together?

1.2 The Plan 9 ARM assembler: `5a`

I will explain in this book the code of the Plan 9 ARM assembler `5a` [Pik93][1], which contains about 4200 lines of code (LOC). `5a` is written in C for the most part. The parser of `5a` is using also Yacc [Joh79].

[1] See `http://plan9.bell-labs.com/magic/man2html/1/8a` for the manual page of `5a`. Despite its name, this page covers also `5a`.

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' in 5a comes from the Plan 9's convention to name architectures with a single character ('0' is MIPS, '5' is ARM, '8' is x86, etc.) and the 'a' means assembler.

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 assembler 5a, and not for instance the x86 variant 8a, for reasons of simplicity. Indeed, RISC machines are far simpler than CISC machines. Moreover, the availability under Plan 9 of an ARM emulator (called 5i) helps to understand the semantics of the assembly instructions used in 5a.

Note that the Plan 9 assembler's output differs slightly from other traditional assemblers. The files generated by 5a, called the *object files*, are ARM-specific but they do not contain machine code. Instead, an object file is essentially the serialized form of the abstract syntax tree of an assembly source. The actual machine code generation is performed by the linker 5l[2]. I think this design leads to less code in total (when combining the code of 5a and 5l), and to simpler code.

1.3 Other assemblers

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

- The original UNIX assembler [Rit79], called as[3], was targetting the DEC PDP11 architecture. It was modeled after a proprietary assembler provided by DEC called PAL11R. as contains 3600 LOC, which is smaller than 5a, but it is written in assembly, which makes its codebase significantly harder to understand. Moreover, it targets an obsolete architecture (the PDP11).

- The GNU Assembler gas [EF00], part of the binutils package[4], is probably the most used open source assembler. It supports many architectures (ARM, x86, etc.) and can generate object files using different formats (ELF, COFF, etc.). It is called internally by gcc and so is indirectly used to assemble most open source programs. However, the code of gas is very big: 350 000 LOC, which is almost two orders of magnitude more code than 5a. The whole binutils package contains 3.4 million LOC (not including the code in the testsuite). Even the ARM-specific file gas/config/tc-arm.c has already 25 000 LOC.

- LLVM Machine Code[5] (MC) is a library part of the LLVM infrastructure that translates assembly into machine code. When used by the llvm-mc program, the library acts as a regular assembler. The library is fairly small, 25 000 LOC, but is part of a fairly large infrastructure, LLVM, with 1.3 million LOC.

- NASM [Dun00][6] is a popular x86 assembler using the Intel syntax as opposed to gas (and 5a) which uses the AT&T syntax. It is also fairly large with 50 000 LOC.

- AS86[7] is an historical assembler used to compile old versions of Minix and Linux. It is an x86 16-bit and 32-bit assembler part of Bruce Evans's C compiler (BCC). AS86 is also using the Intel syntax. Because it can generate 16-bit "real-mode" machine code, it is still used to compile programs such as boot loaders. It is fairly small: 12 500 LOC. This includes the machine code generation, a part that is not done by 5a (but done by 5l). However, because x86 is a rather complicated architecture — the 16-bit/32-bit as well as the different CPU modes (real-mode, protected-mode, virtual-mode) being just a testimony of this complexity — I prefer to present instead an ARM assembler.

[2] Readers interested in this topic should read instead the LINKER book [Pad15b].
[3] http://minnie.tuhs.org/cgi-bin/utree.pl?file=V7/usr/src/cmd/as
[4] https://www.gnu.org/software/binutils/
[5] http://blog.llvm.org/2010/04/intro-to-llvm-mc-project.html
[6] http://www.nasm.us/
[7] http://v3.sk/~lkundrak/dev86/

- MMIXAL [Knu99][8] is an assembler for the MMIX virtual machine. Both MMIXAL and MMIX were designed by Donald Knuth. MMIXAL is a small and very well documented program. Its (literate) source is about 3200 LOC, including the machine-code generation part. However, in Principia Softwarica I want to restrict myself to programs that can run on real machines, not on virtual machines such as the MMIX.

Figure 1.1 presents a timeline of the major assemblers. I think `5a` represents the best compromise for this book: it implements the essential features of an assembler, for an architecture that is still relevant today (the ARM), while still having a small and understandable codebase (4200 LOC).

`5a` is obviously not as used as `gas`, but it is still a production-quality assembler. It was used to assemble all Plan 9 programs at Bell Labs and it is still used in the toolchain of the Go programming language[9]. This is partly because one of the main designer of Go, Rob Pike, was also the author of `5a` (and Plan 9). Because Go was originally conceived and used at Google, some of Google's services are currently assembled by a Plan 9 assembler.

1.4 Getting started

To play with `5a`, 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[10], so you can cross-assemble ARM programs from your regular machine. Once installed, you can test `5a` under Plan 9 with the following commands:

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

The command in Line 2 assembles the very simple `helloworld.s` ARM assembly program and generates the `helloworld.5` ARM object file. Note that in Plan 9 object files do not use the `.o` filename extension. Instead, an object file for the ARM architecture uses the `.5` filename extension, hence the use of `helloworld.5` above. Line 3 then links the object file (see the LINKER book [Pad15b]) and generates the final ARM binary executable `hello`. Line 4 launches the program, assuming you are under an ARM machine (e.g., a Raspberry Pi[11]).

Note that it is easy under Plan 9 to cross compile from another architecture; you can use exactly the same commands (`5a`, `5l`, etc.). To play with `5a` under an x86 machine you just need after the linking step Line 3 to use instead the ARM emulator `5i`:

```
1   $ cd /tests/5a
2   $ 5a helloworld.s
3   $ 5l helloworld.5 -o hello
4   $ 5i hello
...
```

See the EMULATOR book [Pad15a] for more information on `5i`.

[8] https://www-cs-faculty.stanford.edu/~knuth/mmixware.html
[9] https://golang.org/doc/asm
[10] https://github.com/aryx/goken9cc
[11] https://www.raspberrypi.org/

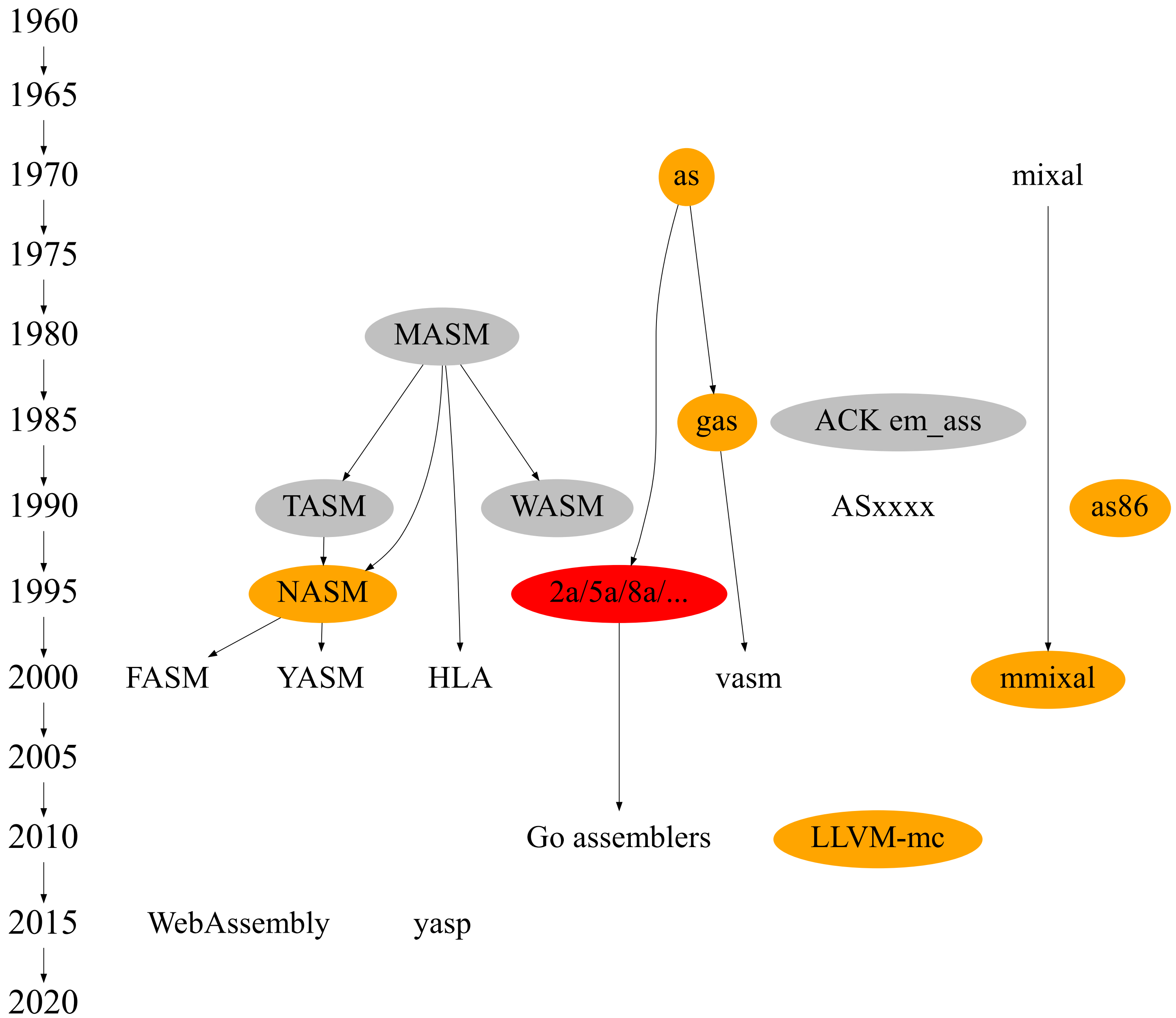


Figure 1.1: Assemblers timeline

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. To understand Chapter 8, you will also need to know Yacc [BLM92].

Note that this book is not an introduction to assembly programming. I assume you already know one assembly language, not necessarily the one used by `5a` though (e.g., NASM [Dun00]), and that you have a basic understanding of computer architecture (see [Tan88, PH13] for good books on the subject). I assume you are already familiar with concepts such as a register, a stack pointer, a program counter, memory move, jumps, labels, etc. However, I do not assume that you know how an assembler works. Even if in a few Principia Softwarica books such as the COMPILER book [Pad16b] or KERNEL book [Pad14] I assume a knowledge of the concepts and theory underlying those programs, this is not the case here. Indeed, there are very few books explaining how an assembler works (I can cite almost only *Assemblers and Loaders* [Sal93]), as opposed to a myriad of books on compilers and kernels.

It is not necessary to know the ARM architecture to understand this book, but I recommend to read the EMULATOR book [Pad15a] if you want to fully understand the semantics of the assembly instructions presented in this book. An alternative is to read the ARM edition [PH16] of the classic computer architecture book by Patterson and Hennessy [PH13].

If, while reading this book, you have specific questions on the assembly syntax used in this book or on the interface of `5a`, I suggest you to consult the man page of `5a` at `docs/man/1/8a`[12] in my Plan 9 repository. You can also consult the documentation of the Plan 9 assemblers [Pik93] (available also in `assemblers/docs/asm.pdf`).

1.6 About this document

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

1.7 Copyright

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

1.8 Acknowledgments

I would like to acknowledge of course the author of `5a`, Rob Pike, who wrote in some sense most of this book. Thanks also to Pascal Garcia for his comments on earlier versions of this book.

[12]Despite its name, this document covers also `5a`.

Chapter 2

Overview

Before showing the source code of `5a` in the following chapters, I first give an overview in this chapter of the general principles of an assembler, of the assembly language supported by `5a`, and of the format of the object files generated by `5a`. 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 Assembler principles

2.1.1 Assemblers and assembly language

An *assembler* is a program that translates source code written in an assembly programming language into machine code, or into an object code close to machine code. An *assembly language* is a low-level programming language mimicking closely the instructions of a machine, but using a textual format rather than the binary format used internally by computers. This textual format is far more convenient for the programmer. Note that each assembly language is specific to a computer architecture.

Before assemblers existed, programming meant encoding every instruction as a binary number by hand and keeping track of memory addresses on paper. The first assemblers—written for the EDSAC at Cambridge in 1949 by David Wheeler and for the IBM 701 in the early 1950s—were among the earliest programs ever written, and their contribution was immediate: symbolic names for instructions and addresses turned programming from a clerical nightmare into something resembling engineering. Assembly remained *the* programming language for systems work from 1949 until the mid-1970s, when C (Dennis Ritchie, 1972) showed that a high-level language could produce code efficient enough for an operating system kernel.

2.1.2 Mnemonics, opcodes, and operands

Assembly languages use *mnemonics* to denote low-level instructions. For instance, an assembly programmer can simply use the mnemonic `ADD` in his code instead of having to remember that `0000100` is the binary code to perform an addition in an ARM processor.

Typical instructions are made of an *opcode* and one or more *operands*. For instance, `ADD 15, R1, R4` is a complete assembly instruction telling the computer to add fifteen to the first register and to put the result in the fourth register.

2.1.3 Syntax families: Intel, AT&T, and Plan 9

Unlike C or Python, which have language standards agreed upon across implementations, *assembly languages are not standardized*. Each assembler's author chooses their own syntax, and two assemblers for the *same* processor often accept wildly different input. Reading assembly code from an unfamiliar toolchain can feel like learning a

new language even when the underlying machine instructions are identical. Three syntax families dominate in practice, and every low-level programmer eventually meets all three.

Intel syntax, used by Intel's own reference manuals and by the dominant Windows assemblers—MASM (Microsoft), TASM (Borland), NASM, FASM, YASM, and Intel's `icc`—writes the destination *first* and the source *second*. The instruction `mov eax, 42` means "store 42 into `eax`". Register names appear unadorned (`eax`, `ebx`), memory addressing uses square brackets (`[ebx`4]+), and operand sizes are noted with keywords like `byte ptr` and `dword ptr` when the context does not make them obvious. Hex constants use the `0x` prefix or the Intel-native `h` suffix (`0x2a` or `2ah`).

AT&T syntax, the default for the GNU assembler `gas` (and therefore for `gcc` output, the Linux kernel, and essentially everything else built with GNU tools), inverts the operand order: source first, destination last. It marks register names with a `%` sigil and constants with a `$` sigil, and it encodes operand size in the mnemonic itself as a single-letter suffix—`movb` for byte, `movw` for 16-bit word, `movl` for 32-bit long, `movq` for 64-bit quadword. So "store 42 into `eax`" becomes `movl $42, %eax`. Memory references use a baroque `offset(base, index, scale)` form, inherited directly from the original AT&T UNIX assembler written for the DEC PDP-11 in the early 1970s, from which the whole UNIX assembler lineage descends.

Plan 9 syntax, the form used by `5a` and every other Plan 9 assembler, inherits the source-first/destination-last order from its AT&T ancestor but makes several deliberate departures. It drops the `%` sigil, so register names are unadorned (`R0`, `AX`). It makes the mnemonic *generic* across operand sizes: `MOVW` means "move a 32-bit word" on any architecture, and the assembler or linker picks the correct underlying instruction from the operand types. It adds ARM-specific condition suffixes (`BEQ`, `ADDCS`) and addressing-mode suffixes (`.P` for post-increment, `.W` for write-back, `.S` for set-flags). It introduces four *pseudo-registers*—`SB` (static base), `FP` (frame pointer), `SP` (stack pointer), `PC`—that are separate concepts from the hardware registers of the same name. And a handful of *virtual instructions* like `RET` and `NOP` have no direct machine encoding at all and are expanded by the linker. Plan 9 syntax outlived Plan 9's commercial decline because Rob Pike brought it into the Go language toolchain at Google: Go's compiler emits Plan 9-style assembly and the Go runtime's architecture-specific code is written in it, which is why anyone reading modern Go internals has to learn the form.

Here is "store the literal value 42 into the first general-purpose register" in all three styles, on x86:

```
operation       Intel            AT&T                Plan 9
-------------   --------------   -----------------   ----------------
load imm 42     mov  eax, 42     movl $42, %eax      MOVL $42, AX
load from mem   mov  eax, [ebx]  movl (%ebx), %eax   MOVL (BX), AX
store to mem    mov  [ebx], eax  movl %eax, (%ebx)   MOVL AX, (BX)
add register    add  eax, ebx    addl %ebx, %eax     ADDL BX, AX
```

The three forms encode exactly the same underlying x86 instructions but look nothing like one another, and they trip up anyone who knows one syntax well and meets another cold. On ARM, Plan 9 syntax stays the same in spirit but uses ARM-specific register names (`R0` through `R15`); `MOVW $42, R0` is the ARM equivalent of `MOVL $42, AX` above. The rest of this book uses Plan 9 syntax exclusively, which from Section 2.3 onward I call *Asm5* for the ARM variant.

2.1.4 Symbolic addresses

A key feature of assemblers is to allow the use of *symbolic addresses* as operands, freeing the programmer from tedious manual calculations. Indeed, in assembly a programmer can define *symbols* designating certain memory areas (code area or data area) and he can then use those symbols as operands. For instance, the instruction `B foobar` allows to branch (jump) to the code following the `foobar` symbol. Without symbolic addresses, a programmer would have instead to write something like `B 1562` and make sure that he calculated correctly that 1562 was the address corresponding to the thing he wanted to jump to. This would require to know the size of each instruction, and each further modification of the program could entail the recalculation of all those addresses.

2.1.5 The phases of assembly

To summarize, the main functions of an assembler are typically the following:

1. *Parse* an input textual file
2. *Check* that the combinations of opcodes and operands form valid machine instructions
3. *Compute* the concrete values of symbolic addresses (this usually requires a two-pass algorithm as one can reference symbols defined later in the file)
4. *Generate* the binary machine code

2.1.6 Object code and the linker

In fact, most assemblers do not generate the final machine code but instead generate an *object code*, which is mostly machine code but with extra information about *unresolved symbols*. Indeed, even if for small programs the definitions and uses of symbolic addresses could be in the same single file, as programs grow larger, it becomes useful to separate the source in different files. In this case, you could want to reference in one assembly file a symbol defined in another file. This is why the object file must contain, in addition to machine code, enough information about the external symbols this assembly file is using (as well as the symbols it defines) so that another tool, the *linker*, can later fully *resolve* all the symbol references used in all the files. In essence, an object file is really the simplest form of a *module*; it packs code, data, and information about exported and imported entities.

A linker then essentially concatenates the code (and data) of the multiple object files together, resolves all the symbolic addresses (now that all the code and data is available and has been assigned a fixed memory area), patches all the incomplete instructions that were using unresolved symbols, and finally generates the binary executable.

2.1.7 The death of assembly that never happened

C (Dennis Ritchie, 1972) was supposed to make assembly obsolete, and for 99% of code it did: by the late 1970s most UNIX programs were written in C rather than assembly, and by the 1990s even operating system kernels were overwhelmingly C. But the remaining 1% never went away. Kernel entry and exit code (the trap vector, context switch, and register save/restore) must be written in assembly because it manipulates state the C compiler does not know about. Boot loaders run before the C runtime exists. Cryptographic code (AES-NI, constant-time algorithms to defeat side-channel attacks) needs instructions the compiler will not emit. SIMD inner loops for audio and video codecs squeeze the last few percent of performance that the auto-vectorizer misses. And sometimes the compiler simply generates suboptimal code for a critical hot path, and a few lines of hand-written assembly are the pragmatic fix. Moreover, the "last 1%" is not a static residual—it is a *recurring wave*. Every time Intel ships AVX-512, or ARM adds SVE2, or a new architecture like RISC-V appears, there is a period where the compiler does not yet know about the new instructions and someone has to write assembly to use them. The cycle repeats every few years, which is what keeps assemblers relevant decade after decade.

The consequence is that assemblers are alive and well in 2024—just with a different *customer*. In the 1960s the customer was the programmer, who wrote entire programs in assembly. Today the primary customer is the *compiler*: `5c` (or `gcc`, or `clang`) generates assembly, and `5a` (or `gas`, or the LLVM integrated assembler) translates it to object code. Human-written assembly is the exception, confined to the handful of files where the compiler cannot do the job. `5a`'s design reflects this reality: its input language (Asm5) is designed to be easy for a compiler to emit, not for a human to write from scratch, and its output is an object file that the linker will finish rather than a standalone executable. The rest of this book is what that compiler-serving tool looks like inside.

2.2 5a command-line interface

The command-line interface of the assembler `5a` is pretty simple:

```
$ 5a
usage: 5a [-options] file.s
$ 5a foo.s
$ 5l foo.5 bar.5 ...
```

Given an input assembly file `foo.s`, `5a` outputs an object file `foo.5`. You can change this default behaviour by using the `-o <outfile>` option. Other options are related to macroprocessing and debugging and will be described later.

Object files in other operating systems (e.g., Linux) usually end with the `.o` filename extension. However, because Plan 9 supports multiple architectures and makes it very easy to cross-assemble or cross-compile programs, it is more convenient to use the code of the architecture (here `5` for ARM) as the filename extension of object files. That way, you can have in the same directory the ARM object file `foo.5` and the x86 object file `foo.8` without any name conflict. For assembly programs, the need for different object filename extensions may not be obvious. Indeed, assembly files are architecture specific anyway. However, for C programs, which can be compiled by `5c` or `8c`, generating different object files from the same source is very useful.

2.3 Assembling `helloworld.s`

From now on, I call *Asm5* the ARM assembly language supported by `5a`. Because the different Plan 9 assemblers (e.g., `5a`, `8a`) are variations of a single program, the assembly languages they support are also variations of a single language I call *Asm9*. You can see Asm5 as a specialized version of Asm9 for the ARM processor. By understanding Asm5, you will understand also fairly well the assembly languages supported by the other Plan 9 assemblers (e.g., `8a`), because they have a lot in common.

In this section, I will show a simple Asm5 program, `helloworld.s`, which prints `Hello World` when executed. I will use this program as a tutorial for Asm5 (and more generally for Asm9).

2.3.1 Background

To understand `helloworld.s`, I must first show the equivalent program written in C to introduce some background on how to perform *system calls* in Plan 9.

Here is the simplest Plan 9 `hello world` program written in C:

⟨helloworld1.c 15a⟩≡
```
#include <u.h>
#include <libc.h>

void main() {
  print("hello world\n");
}
```

This code is using the `print()` function from the core C library (see the LIBCORE book [Pad16c]). If I expand the code of this function, and simplify things, I will get this C program:

⟨helloworld2.c 15b⟩≡
```
#include <u.h>
#include <libc.h>

void main() {
  pwrite(1, "hello world\n", 12, 0);
}
```

The pwrite() function is also defined in the C library, but it is written in assembly in lib_core/libc/9syscall/p pwrite() is a small wrapper around the ARM instruction SWI (for "software interrupt"), which performs a system call (also known as a *syscall*). Here is the prototype of pwrite() defined in include/core/libc.h:

```
⟨prototype pwrite in libc.h 16a⟩≡
  extern long pwrite(fdt, void*, long, vlong);
```

The pwrite() interface is documented in docs/man/2/read[1]. The parameters are a file descriptor (e.g., 1 for the standard output), a string pointer, the number of bytes to write, and finally a vlong seeking offset. Most of the parameters use 4 bytes on the ARM, except the vlong which uses 8 bytes on the ARM.

2.3.2 The program

You now have enough background to understand partially the helloworld.s program below:

```
⟨assemblers/5a/tests/helloworld.s 16b⟩≡
  1 TEXT _main(SB), $20
  2         B later
  3         B loop /* not reached */
  4 later:
  5         /* fill missing characters for hello */
  6         MOVW $hello(SB), R2
  7         MOVW $'W', R1
  8         MOVB R1, 6(R2)
  9         MOVW $'o', R1
 10         MOVB R1, 7(R2)
 11         MOVW $'r', R1
 12         MOVB R1, 8(R2)
 13         MOVW $'l', R1
 14         MOVB R1, 9(R2)
 15         MOVW $'d', R1
 16         MOVB R1, 10(R2)
 17         MOVW $'\n', R1
 18         MOVB R1, 11(R2)
 19         /* prepare the system call PWRITE(1,&hello,12,0LL) */
 20         MOVW $1, R1
 21         MOVW R1, 4(R13)
 22         MOVW $hello(SB), R1
 23         MOVW R1, 8(R13)
 24         MOVW $12, R1
 25         MOVW R1, 12(R13)
 26         MOVW $0, R1
 27         MOVW R1, 16(R13)
 28         MOVW R1, 20(R13)
 29         MOVW $9 /*PWRITE*/, R0
 30         /* system call */
 31         SWI $0
 32         BL exit(SB)
 33         RET /* not reached */
 34 loop:
 35         B loop
 36
 37
 38 TEXT exit(SB), $4
 39         /* prepare the system call EXITS(0) */
 40         MOVW $0, R1
 41         MOVW R1, 4(R13)
 42         MOVW $3 /*EXITS*/, R0
```

[1] Again, despite its name, this document covers also pwrite.

```
43        /* system call */
44        SWI $0
45        RET /* not reached */
46
47
48 GLOBL   hello(SB), $12
49 DATA    hello+0(SB)/6, $"Hello "
50
```

To assemble, link, and execute this program, simply do like in Section 1.4:

```
$ 5a helloworld.s
$ 5l helloworld.5 -o hello
$ ./hello
hello world
```

Appendix D contains more examples of assembly programs.

2.3.3 Pseudo instructions

The `helloworld.s` program above defines three *symbols*: two procedures, `_main()` and `exit()`, and one global, `hello`. The two procedures are introduced via the `TEXT` *pseudo instruction* Line 1 and Line 38, and the global via `GLOBL` Line 48. I say "pseudo" (or sometimes "virtual") because those instructions do not match directly a machine instruction. They are assembly-only constructs, also known as *assembly directives*. Indeed, the ARM processor has no notion of procedure names; it just manages numbers and concrete addresses.

The operands of the pseudo instructions `TEXT` and `GLOBL` are the name of the symbol it defines followed by `(SB)`, which I will explain later, and a *constant* value prefixed by a dollar. In Asm9, *all constants are prefixed by a dollar*. For `GLOBL`, the constant value represents the size, in number of bytes, this global will use. For `hello` Line 48 it is `12` (enough to hold the `"hello world\n"` string). For `TEXT`, the constant value represents the size, in number of bytes, this procedure will need for its *locals* in the stack. For `_main()` Line 1 it is `20` (the number of bytes needed to hold all the arguments in the stack to the `PWRITE` system call: 4 for the file descriptor integer, plus 4 for the string pointer, plus 4 for the size, plus 8 for the `vlong` offset).

As you will see in the LINKER book [Pad15b], the linker `5l` is looking for a procedure named `_main` for the entry point of the executable it generates, even though the entry point of C programs is `main`, not `_main`. This is because the core C library defines a `_main()` procedure, written in assembly, which does some core initializations and then calls `main()` (see the LIBCORE book [Pad16c]). In `helloworld.s`, I do not use and so do not link the C library, to simplify the code and the explanations, so I must define a `_main()` at Line 1.

2.3.4 Labels

The first instruction of `_main()`, Line 2, is a jump, known as a *branch* in ARM (hence the `B`). It is a jump to `later`, a *label* defined Line 4. In Asm9, *all label definitions are suffixed by a colon*. Labels are similar to symbols: They allow to give a symbolic name to a memory area. However, labels are restricted to code area and are locals to an assembly file. They are used for intra-procedural jumps.

Note that the syntax for comments Line 3 and Line 5 is the same than in C.

2.3.5 Assignments

Line 6 places the *address* of the `hello` global (suffixed again by `(SB)`, which I will explain later) in the register `R2`.

There are 16 ARM registers named `R0` to `R15`. Note that Asm9 uses a *left-to-right* assignment syntax[2]. Moreover, in Asm9 `MOV` instructions are suffixed with a letter corresponding to a size: `W` for word, `B` for byte, etc.

Line 7 places the character constant `'W'`, converted by the assembler in its integer ASCII value (87), into the register `R1`.

2.3.6 Addressing modes

Line 8 introduces a new *memory addressing mode*. It is the first instruction that writes into memory. Indeed, until now the code in `helloworld.s` was only modifying the content of registers. `MOVB R1, 6(R2)` at Line 8 stores the first byte (because of the `B` suffix) of register `R1` (which should contain 87) at the address denoted by `R2` plus `6`. The assembly instruction `MOVB N, O(B)` roughly corresponds to the following C statement `B[O] = N`, if `B` is a byte pointer. This instruction is also equivalent to this C statement `*(B+O) = N`. `O` is called the *offset*, and it is applied to a pointer `B` called the *base*. This addressing mode is called *indirect with offset*. The parenthesis around the register corresponds roughly to the C dereferencing operator `*`.

In fact, Line 6 introduced also an addressing mode. The syntax `hello(SB)` is reminiscent of the base and offset addressing mode we have just seen. `SB` stands for *static base* register. It refers to the beginning of the address space of the program. In Asm9, all references to globals and procedures are written as offsets to `SB` (for definition references see Lines 1, 38, 48, and 49; for use references see Lines 6, 22, and 32). The instruction `MOVW foo(SB), R1` will store the *content* at the address denoted by the symbol `foo` into `R1`. The instruction `MOVW $foo(SB), R1` will store the *address* denoted by the symbol `foo` in `R1`. The $ in that case corresponds roughly to the C address operator `&`.

2.3.7 Pseudo registers

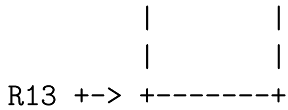
`SB` is one of the few *pseudo registers* of Asm9. `PC` is another one. It corresponds to the *virtual program counter*. Similar to pseudo instructions, pseudo registers do not correspond exactly to machine registers. Indeed, the ARM has already an hardware register called `R15` representing the program counter. Because the ARM has fixed-length instructions of 4 bytes, the value of `R15` is always a multiple of 4. The pseudo register `PC` instead counts instructions, not bytes of data. Thus, to branch to the second following instruction (to skip one instruction), you could use the following instruction: `B 2(PC)`. This instruction is equivalent to `B 8(R15)`.

Why should you use pseudo registers? The advantage of using `2(PC)` instead of `8(R15)` in Asm5 may look small. However, on some architectures the size of instructions is variable. For instance, it is not trivial on x86 to compute the number of bytes two arbitrary instructions are using. Just like with symbolic addresses, using pseudo registers allows the programmer to think in slightly higher-level terms (for `PC` in terms of instructions instead of bytes of data), and to delegate tedious tasks such as counting the size of instructions to the computer.

Asm9 defines four pseudo registers:

- `PC`, the (virtual) *program counter*, counts the number of instructions. For the ARM, `PC` is related to the (real) program counter register `R15`.
- `SB`, the *static base* register, refers to the beginning of the address space of the program. For the ARM, `SB` is related to the machine register `R12`.
- `SP`, the (virtual) *stack pointer*, can be used to access local variables in the stack. For the ARM, `SP` is related to the machine register `R13`.
- `FP`, the *frame pointer*, can be used to access the arguments of the procedure in the stack. `FP` is also related to `R13` for the ARM.

[2] This syntax is called the AT&T syntax, as opposed to the Intel syntax which is right-to-left.



```
        |        |
        |        |
R13 +-> +--------+
```

Figure 2.1: Stack content when `hello` started.

In this tutorial, I will avoid using those pseudo registers (except `SB` because it is mandatory). Indeed, they may simplify things in the long term but they add some extra complexities at the beginning.

Line 6 through 18 set characters in the `hello` global array of characters.

2.3.8 Call stack

I can now go through Line 20 to Line 29. Those instructions build the arguments for the system call performed Line 31 with the software interrupt instruction `SWI`. *Arguments*, in function calls or system calls, are by convention in Plan 9 hold primarily in the *stack*. You will see later that `R0` plays also a special role regarding arguments because of some C and kernel calling conventions. As I mentioned in the previous section, `R13` is by convention used to represent the stack pointer. Figure 2.1 contains a representation of the stack when the `hello` program is loaded in memory by the kernel. The stack grows downward, so high addresses are at the top in the diagram.

`R13` is initialized by the kernel before the kernel gives control to the entry point of the binary program. Its value is very high in the virtual memory address space. Here is a dump of the registers before the program starts by using the emulator/debugger `5i` [3]:

```
$ 5i hello
5i> $r
...
R0  #7ffff70 R1  #7ffffffc R2  #0       R3  #0
R4  #0       R5  #0        R6  #0       R7  #0
R8  #0       R9  #0        R10 #0       R11 #0
R12 #0       R13 #7ffff58  R14 #0       R15 #1020
```

Figure 2.2 contains a representation of the stack after Line 1, when the processor starts to execute the instruction Line 2. Remember that the second operand of the `TEXT` pseudo instruction is the number of bytes this procedure will need for its locals in the stack. Thus, when assembled, the `TEXT` pseudo instruction for `_main` Line 1 should generate a machine instruction that decrements `R13` by 20. In fact, the actual generated instruction decrements `R13` by 24; you will see later in Section 2.3.10 why `_main` uses an extra word in the stack before the arguments.

Figure 2.3 contains the same stack before Line 31, after the arguments to the system call have been set by Lines 20 through 28.

2.3.9 System calls

The Plan 9 *kernel calling conventions* impose to have all the arguments in the stack "above" `R13+4` and to use `R0` to hold the syscall "code". The magic constant value `9` at Line 29 is the code corresponding to the `PWRITE` system call (see the file `lib_core/libc/9syscall/sys.h`, which defines all those syscall codes). The `SWI` instruction Line 31 then performs the system call and jumps in the kernel. `SWI` has one operand in the ARM. This operand is normally used to specify which entry in the *interrupt table* to go to. However, the Plan 9 kernel uses `R0` instead for that purpose. Thus, the operand of `SWI` is not used under Plan 9. This is why I pass zero to `SWI` in `helloworld.s` Line 31.

[3] To fully understand the values of those registers, see the KERNEL book [Pad14] or the EMULATOR book [Pad15a], especially the code of `initmemory()` in `5i`.

```
        |        |
        |        |
    +24 +--------+ <-+ old value of R13
        |        |
    +20 +--------+
        |        |
    +16 +--------+
        |        |
    +12 +--------+
        |        |
     +8 +--------+
        |        |
     +4 +--------+ <- start of "locals"
        |        |
R13 +-> +--------+
```

Figure 2.2: Stack content before Line 2.

```
        |        |
        |        |
    +24 +--------+ <-+ old value of R13
        |  0     |
    +20 +--------+
        |  0     |
    +16 +--------+
        |  12    |
    +12 +--------+
        |&hello  |
     +8 +--------+
        |  1     |
     +4 +--------+ <- start of PWRITE arguments
        |        |
R13 +-> +--------+
```

Figure 2.3: Stack content before Line 31.

```
TEXT foo(SB), $0   MOVW R14, (R13)
                   SUB $4, R13, R13
 ...               ...                       ...
                   ADD $4, R13, R13
 RET               B -4(R13)                 B (R14)
----------------  -----------------------   ------------------
  assembly code    machine code non-leaf     machine code leaf
```

Figure 2.4: Machine code for the pseudo/virtual instructions TEXT and RET.

2.3.10 Function calls

In Asm5, regular function calls (e.g., the call to `exit` Line 32), use the `BL` instruction. You should use `SWI` only for system calls. `BL` stands for *branch and link*, which I will explain below.

Some processors such as the x86 have a `CALL` instruction, which when executed pushes on the stack the value of the program counter for the next instruction. This value corresponds to a *return address*. `CALL` then jumps (branches) to the code of the *callee*. A corresponding `RET` instruction in the callee will pop back in the program counter the value pushed by `CALL` to return back to the *caller*.

There is no `CALL` or `RET` instruction in the ARM. Instead, the `BL` instruction just saves the current value of the program counter (`R15`) plus 4 (the next instruction) in the special register `R14`, called the *link register*, and then jumps to the callee.

The use of a special register to hold a return address is an ARM optimization that avoids for certain calls to use the stack, and so the memory (which is slow). Indeed, when a function does not call other functions, in which case it is called a *leaf* function, the value of `R14` does not need to be saved. Returning to the caller from the leaf function can be done simply by setting the program counter to the value in the link register with `B (R14)`. However, if the function is not a leaf then the value in `R14` must be saved somewhere, in the stack, before the function calls another function via `BL` (which would overwrite `R14`).

Non-leaf functions are the reason the `TEXT` pseudo instruction allocates one more word than its second operand, for instance, 24 instead of 20 for `_main` Line 1. This extra word in Figure 2.2 can be used by functions called from `_main` to save the return address stored in `R14`.

2.3.11 Virtual instructions

The leaf detection is done (statically) by the linker `5l`. Indeed, `5l` is the program generating the machine code in Plan 9, and so the program that needs this information to optimize machine code. A leaf is simply any procedure that does not contain a `BL` instruction, for instance, `exit` in `helloworld.s`[4]. `_main` in `helloworld.s` on the opposite is not a leaf function because it is using `BL` Line 32.

The machine instructions generated by `5l` for the `TEXT` pseudo instruction depends on whether the function is a leaf. The same is true for the `RET` instruction, Line 33 and Line 45. `RET` is called a *virtual instruction*. Indeed, as I mentioned before, the ARM does not have any `RET` instruction. However, it is convenient for the programmer to use a `RET` assembly instruction in his program to return to the caller, whether the caller is a leaf or a non-leaf function.

Figure 2.4 describes roughly the machine code generated by `5l` for the pseudo and virtual instructions `TEXT` and `RET` for a simple procedure `foo`, depending on whether this procedure is a leaf or not. For more information, see the LINKER book [Pad15b].

The virtual instruction `RET`, just like the pseudo instruction `TEXT`, or the pseudo registers, allows the programmer to think in slightly higher-level terms and let the computer do the appropriate optimizations. In fact,

[4] `SWI` does not count as a function call; `R14` is not overwritten by a `SWI`.

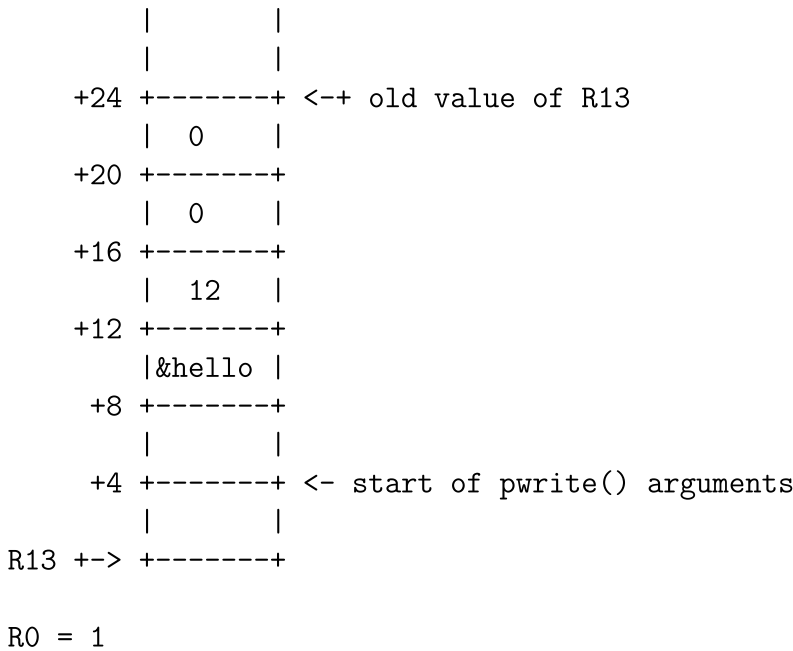


Figure 2.5: Stack and `R0` value for a C call to `pwrite()`.

`MOVW` used in Figure 2.4 is also a virtual instruction. It hides some architecture restrictions and peculiarities regarding memory accesses. in Asm5, you can use the very general `MOVW` even though the ARM processor supports only the more basic and separate `LDR` (load) and `STR` (store) instructions.

Instructions Line 40 though 44 are similar to the instructions we have seen before; they perform the system call `exits(0)`, which terminates the program. This is why I marked a few instructions with a comment indicating the instruction could not be reached. Indeed, the program will have exited already (and so will not perform the `RET`).

2.3.12 C calling conventions

Note that the way arguments are laid out in the stack in Figure 2.3, as well as the extra word before the arguments (to save the return address to the caller stored temporarily in `R14`), or the fact that all those elements are "above" the stack pointer (`R13`) at the entry of the function callee (or syscall interrupt handler), are just *conventions*. The ARM processor does not dictate any specific way to call procedures. In fact, it does not even have a notion of procedure; the ARM provides only the `BL` machine instruction, which just saves `R15` in `R14` and jumps to an address. The rest is a design choice.

The calling conventions I use in this tutorial come from the *C calling conventions* used by the C compiler `5c`. Indeed, the assembly code generated by `5c` assumes arguments are passed in a certain way and the stack pointer `R13` has a certain value. In fact, the use of `R13` for the stack pointer is also a convention; ARM has no notion of stack, it just manipulates memory.

I use conventions similar to the C calling conventions in this tutorial because the code I present performs a system call and the Plan 9 kernel calling conventions are derived from the C calling conventions. Indeed, the kernel is mostly written in C.

The C calling conventions are slightly different from the kernel calling conventions. In the C calling conventions, instead of using `R0` to store the syscall code, `R0` is used to store the first argument. In fact, `R0` is also used to store the return value of C functions. Using `R0` is an optimization similar to the one you have seen previously with the link register. For certain simple and small functions, everything can be done using just registers, without having to use the stack and so the (slow) memory.

Figure 2.5 shows how the stack would look like if we were calling a C function `pwrite()` instead of doing directly the system call to `PWRITE`. Note that because the first argument is stored in `R0`, the code generated by `5c` for a function call does not bother to store it also in the stack. However, similar to the link register optimization, an extra word in the stack is still allocated for the first argument in case the callee need to overwrite `R0` (in which case it can save its value in the stack).

The calling conventions used by `5c` could be completely different: the order of the arguments in the stack could be reversed, more registers could be used to store the first n arguments, the arguments to a function could be "below" the stack pointer, the return address to a function could be "below" the stack pointer, the responsibility to save the return address in the stack could be done by the caller instead of the callee, etc. The *C calling conventions* implemented by `5c` are a nice compromise between simplicity and speed.

Because most assembly code interoperates with C code, the C calling conventions are also the *assembly calling conventions* in Plan 9. Note that the only constraint imposed by Asm5 on the calling convention is the extra word in the stack above `R13` to store the return address to the caller (because of the code generated for `TEXT` and `RET` by `5l`).

2.3.13 Data layout

The last piece of assembly code I need to explain is Line 49, which initializes (partially) the content of the `hello` global:

⟨*assemblers/5a/tests/helloworld.s repeat* 23a⟩≡

```
...
48 GLOBL   hello(SB), $12
49 DATA    hello+0(SB)/6, $"Hello "
```

Even if Line 48 *declares* the `hello` global (with the `GLOBL` pseudo instruction), it is the `DATA` pseudo instruction Line 49 that *defines* the content of the global. The operands of `DATA` are in order:

1. the symbol of the global, possibly with an offset (`+0` at Line 49), and as always for references to globals the `(SB)` suffix,
2. a slash followed by an integer between 1 and 8 representing the size in number of bytes this `DATA` pseudo instruction is defining (here `/6`),
3. a constant, which can be an integer, a character, or a string of less than 8 bytes.

To define data that takes more than 8 bytes, you need to use multiple `DATA` pseudo instructions. For instance, to define completely `hello`, which would remove the need for Lines 6 to 18 in `helloworld.s` (added for educational purposes in this tutorial), you could write instead:

⟨*full definition of* `hello` *content* 23b⟩≡

```
GLOBL   hello(SB), $12
DATA    hello+0(SB)/8, $"hello wo"
DATA    hello+8(SB)/4, $"rld\n"
```

This concludes the tutorial of Asm5. I tried to present the main instructions and main assembly constructs of Asm5. I will present more instructions gradually in the rest of the book.

2.4 The ARM architecture

The three letters ARM represent different things: a family of RISC machines (Acorn RISC Machines), a family of *instruction set architectures* (ISAs), and finally a family of processors. Up until now, and for the rest of this document, when I use the term ARM I mean the *ARMv6* instruction set architecture. Confusingly, ARMv6 is

the instruction set used in the *ARM11* 32-bits processor family[5]. This family powers most smartphones, as well as the Raspberry Pi[6] (an extremely cheap machine popular among electronic hobbyists).

Because an assembly language mimics closely the instructions of a machine, most of the instructions in Asm5 are instructions of the ARM. The opcodes `B`, `BL`, `SWI`, or `ADD` you have seen in the Asm5 tutorial correspond to ARM opcodes. The same is true for the registers `R0` to `R15`. To fully understand those assembly instructions, I refer you to either the EMULATOR book [Pad15a], which describes the semantics of the corresponding ARM machine instructions, or the ARM reference manual [Sea01].

Asm5 introduces also some pseudo instructions, pseudo registers, and virtual instructions, as you have seen in section 2.3. Those instructions will be fully explained in this document; they do not correspond directly to ARM machine instructions.

As I said briefly in Section 1.2, `5a` does not generate machine code; the linker `5l` does. Thus, there is no need to know the binary format of ARM instructions to understand this document.

2.5 Input assembly language

I have covered already in Section 2.3 most of features of the assembly language Asm5, the language used by the input files of the assembler `5a`. In this section, I summarize those features and give a more systematic description of Asm5. This should help solidify your understanding of Asm5.

2.5.1 Lexical elements

The main lexical elements of Asm5 (and also of Asm9) are:

- *integers*, e.g., `42` (decimal), `0x12` (hexadecimal), or `0666` (octal)
- *characters*, e.g., `'W'`, `'\n'`, or `'\007'`
- *strings*, e.g., `"hello wo"`. Strings are limited to 8 characters or less.
- *floats*, e.g., `4.2`, `10e43`
- *predefined identifiers* in uppercase. Asm5 uses those predefined identifiers for the mnemonics of opcodes (e.g., `ADD`), registers (e.g., `R0`), pseudo instructions (e.g., `TEXT`), and finally pseudo registers (e.g., `PC`).
- *identifiers* in lowercase, used for symbols (e.g., `_main`) and labels (e.g., `later`)
- *comments*, e.g., `/* not reached */` or `// comment`. Comments in Asm5 use the same syntax than C comments. They are ignored by `5a`.
- *spaces* and *TABs*, which are also ignored by `5a`. By convention, it is common to indent with a TAB most instructions, except pseudo instructions such as `TEXT` or `GLOBL` that are kept in the first column.
- *newlines*, which are internally transformed in semicolons. Indeed, the semicolon is an *instruction terminator* in Asm9. By transforming newlines in semicolons, you you can then either write multiple instructions on different lines, or multiple instructions on the same line but separated by an explicit `;`.
- *operators* (e.g., $, `()`, `/`, `+`, `;`, `<>`). Operators can have different meanings depending on the context.

[5] http://www.arm.com/products/processors/classic/arm11/?tab=Specifications
[6] https://www.raspberrypi.org/

2.5.2 Syntactical elements

The main syntactical element of Asm5 is the *instruction*. An Asm5 file is made essentially of a list of instructions, usually one per line. An instruction is composed of an *opcode* followed possibly by 1, 2, or 3 *operands*. A line can also contain a *label* definition, which is an identifier followed by a colon (e.g., `later:`).

2.5.3 Opcodes

Asm5 opcodes can correspond to different kinds of instructions:

- *machine instructions*, e.g., `B`, `SWI`, or `ADD`.
- *pseudo instructions*, e.g., `TEXT`, `GLOBL`, also known as assembly directives.
- *virtual instructions*, e.g., `RET`, `MOVW`, or `NOP`. Virtual instructions are instructions without a one-to-one mapping to an ARM machine code instruction. They are convenient for the programmer (or compiler writer) because they hide some architecture restrictions or peculiarities, or relieve the programmer from tedious tasks such as checking whether a function is a leaf.

2.5.4 Operands

Depending on the opcode, Asm5 supports different kinds of operands:

- *constants*, which in Asm9 are always prefixed with a dollar, e.g., $42, $`'W'`
- *registers*, named `R0` to `R15` in Asm5
- *shifted registers*, which I will explain in Section 8.4.3
- *pseudo registers*, named `PC`, `SP`, `FP`, and `SB` in Asm9
- *label references*, e.g., `later`
- *memory references*, which I describe below

2.5.5 Addressing modes

Memory references appear mainly in memory moves (e.g., `MOVW`), branching instructions (e.g., `BL`), as well as in entity definitions (e.g., `TEXT`). There are 5 different ways to reference memory in Asm5, called *memory addressing modes*:

1. In *indirect* mode, you use parenthesis around a register or pseudo register (e.g., `(R1)`) to access the content at the address denoted by the register (here `R1`)
2. In *indirect with offset* mode, you combine an offset with a register (e.g., `4(R1)`), or pseudo register (e.g., `4(FP)`), to access the content at the address denoted by the sum of the register (here `R1` and `FP`) and the offset (here `4`). Note that you do not prefix offsets with a dollar.
3. In *symbol reference* mode, you combine a symbol with a pseudo register (e.g., `hello(SB)`) to access the content at the address denoted by the symbol.

4. In *symbol reference with offset* mode, you combine a symbol, an offset, and a pseudo register (e.g., `hello+4(SB)`), to access the content at the address denoted by the symbol plus the value in the offset. With `SB`, the symbol denotes a global or procedure, with `SP` a local variable, and with `FP` an argument.

   For locals and arguments, it is common to use a symbol as in `x+4(SP)` or `length+4(FP)`. In those cases, the symbol is really just a comment to give a name to the local or argument. However, this comment is kept in the symbol table of the object file, as you will see in Chapter 9, and can be used by debuggers, as you will see in Chapter 10.

5. In *symbol address* mode, you use a dollar sign before a symbol (e.g., `$hello(SB)`) to access the address of the symbol (here `hello`).

Note that Asm9 uses a left-to-right assignment syntax. For moves, the instruction `MOVW (R1), R2` means moving the content at the address denoted by `R1` into `R2`. The same is true for other instructions such as `ADD` where the destination register is the last operand and the sources the first two operands (as in `ADD R1, R2, R3`).

Note also that the ARM processor supports only the first two addressing modes. The remaining addressing modes, which involve symbols, are assembly-only constructs. Ultimately, an assembly instruction using a symbol reference (e.g., `hello(SB)`) will be converted by the linker in a machine instruction using a register and an offset (e.g., `4100(R12)`). Indeed, as you will see in the LINKER book [Pad15b], `5l` reserves the register `R12` to represent the pseudo register `SB` and converts symbol references in offsets to `R12`.

2.5.6 Advanced features

Asm9 has a few more features beyond mnemonics and symbolic addresses:

- *constant expressions* as operands, e.g., `$(1<<6|3)`, which are evaluated at assembling-time
- an embedded *macro-processor* similar to the C preprocessor `cpp`, with features such as `#include` and `#define`. `5a` is thus what people calls a *macro-assembler*. This, combined with the previous feature, allows to overcome some of the original limitations of `5a`. Indeed, even if `5a` does not support the mnemonics corresponding to the advanced coprocessor ARM instructions, you can simply define and use the macro `MCRR` below to encode the binary format of the instruction directly.
- *symbolic constants* definitions (e.g., `TMP = 11`), and uses (e.g.. `MOV R1, R(TMP)`). Those constants were called originally (and ironically) *variables*. They are redundant with the `#define` of the macro-processor; they were probably added before `5a` became a macro-assembler.

⟨*macro* `MCRR` 26⟩≡
```
#define MCRR(coproc, op, rd, rn, crm) \
    WORD $(0xec400000|(rn)<<16|(rd)<<12|(coproc)<<8| \
           (op)<<4|(crm))
```

The `WORD` pseudo instruction used above will be described in Section 8.5.2.

2.6 Output object format

As I said in Section 1.2, the object files generated by `5a` are ARM-specific but they do not contain machine code. Instead, an object file contains the serialized form of the abstract syntax tree of the assembly source. Because an Asm5 file is essentially a list of instructions, its serialized form is simply a concatenation of serialized instructions. The complete format of an object file is described precisely in Chapter 9 but Figure 2.6 describes roughly the content of an object file (remember that object files generated by `5a` use the `.5` filename extension).

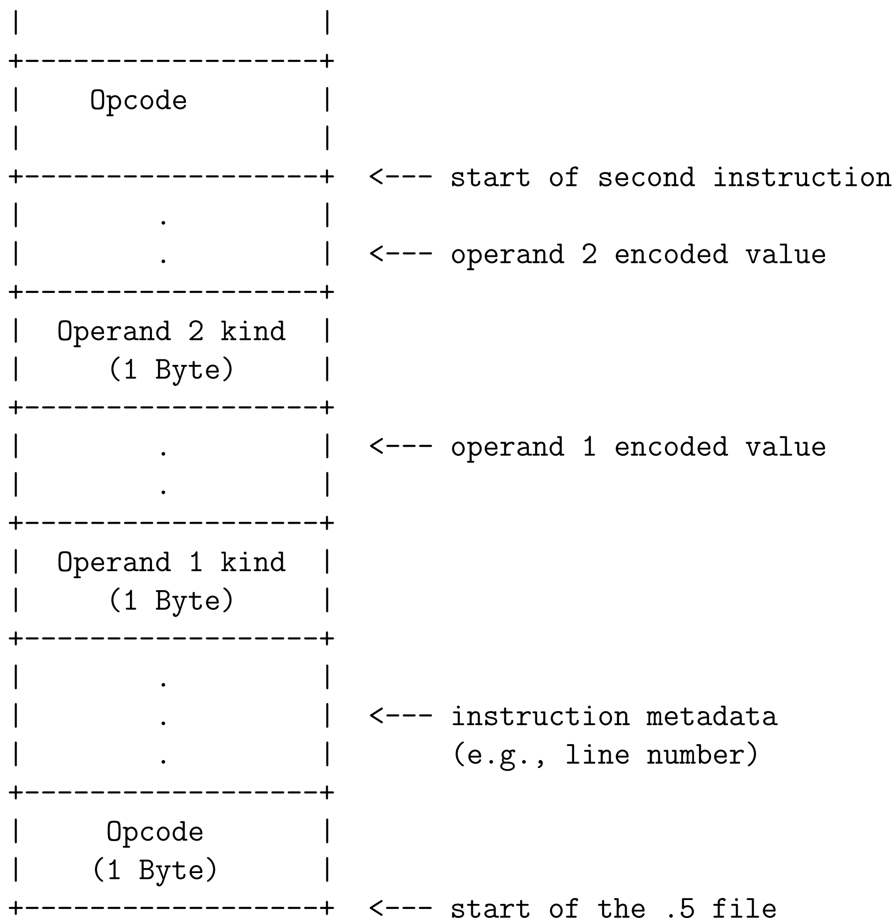


Figure 2.6: File format of a `.5` object file.

A serialized instruction has also a pretty simple format because all instructions, including the pseudo instructions, have the same syntax: an opcode followed by 0 to 3 operands. One byte is enough to encode an Asm5 opcode because the ARM is a RISC machine with far less than 256 ARM opcodes (and Asm5 introduces very few pseudo opcodes) [7].

After the opcode, `5a` uses a few bytes to store metadata about the instruction. After this metadata, another byte is used to encode the *kind* of the first operand (e.g., an integer constant, a register, a memory address, or nothing) followed by a series of bytes encoding the actual operand value (e.g., 4 bytes for an integer constant, 1 byte for the register number, etc.) as shown in Figure 2.6.

`5a` resolves the use of label addresses so `5a` does not need to serialize label definitions. Label references in branching code are transformed in absolute addresses.

2.7 Code organization

Table 2.1 presents short descriptions of the source files used by `5a`, 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.

There are multiple assemblers in Plan 9: one per supported architectures (ARM and x86 in the Plan 9 fork used in Principia Softwarica). The generic code has been factorized in the `assemblers/aa/` directory; the

[7] `8a`, the Plan 9 assembler for x86, needs two bytes to represent an x86 opcode. Indeed, x86 is a CISC machine with more than 400 opcodes.

ARM-specific code is in `assemblers/5a/` as well as in `include/obj/5.out.h`[154c8].

Function	Ch.	File	Entities	LOC
ARM opcodes	3	include/obj/5.out.h	Opcode[31] Operand_kind Register[32]	306
ARM operand	3	5a/a.h	Gen[33b]	70
ARM-specific globals	3	5a/globals.c	nullgen[34a]	7
general data structures	3	aa/aa.h	Sym[156b] Io[156b] Fi[53d]	283
general globals	3	aa/globals.c	hash[39a] symb[40a] outfile[44a] pass[45c] pc[92a]	107
symbol table lookup	3	aa/lookup.c	slookup()[39e] lookup()[40c]	62
main functions	4	5a/main.c	main()[155a] assemble()[45d]	207
IO and lexing utilities	5	aa/lexbody.c	pinit()[50a] newio()[51g] newfile()[52d]	540
lexer for Asm5	6	5a/lex.c	yylex()[56b] cinit()[44c]	590
macro processor	7	aa/macbody.c	mactab[69b] macinc()[74a] macdef()[133b]	846
grammar for Asm5	8	5a/a.y	inst()[83a] yyparse()[104c]	600
object file generation	9	5a/obj.c	outcode()[108]	278
location information	10	aa/hist.c	linehist()[115g] prfile()[118b]	95
float data structure	11	include/obj/common.out.h	Ieee	22
float utilities	11	aa/float.c	ieeedtod()[126b]	35
error management	B	aa/error.c	errorexit()[148a] yyerror()[148c]	50
utilities	C	aa/utils.c	alloc()[151b]	121
Total				4219

Table 2.1: Chapters and source files of 5a.

2.8 Software architecture

Figure 2.7 describes roughly the main control flow and main components of 5a, whereas Figure 2.8 describes the main data flow of 5a. 5a is an assembler, which is a kind of compiler. Thus, it has the classic software architecture of a compiler: a lexer, a preprocessor, a parser, and a code generator.

I will now explain briefly the control flow of 5a, starting from the top of Figure 2.7. After some basic command-line processing and initializations, the function `main()`[155a] calls `assemble()`[45d] with the name of the assembly file to process. `assemble()`, after further initializations, uses a two-pass algorithm and so calls `yyparse()`[104c], the function generated by Yacc [Joh79] from the Asm5 grammar, two times. It also modifies the global `pass`[45c]; for the first call to `yyparse()` the value of `pass` is 1, and for the second call the value of `pass` is 2.

The parsing function `yyparse()` internally calls the lexing function `yylex()`[56b] to get the next *token* from the input file. `yylex()` is usually generated by the program Lex [LS79] from a set of regular expressions. However, in the case of 5a, `yylex()` was handcoded instead. `yylex()` itself calls macro-processing functions such as `macinc()`[74a] (for "macroprocessing include"), as 5a is a macro-assembler supporting `cpp` directives such as `#include`.

From a set of tokens, `yyparse()` can parse using different *grammar rules* a constant, a register, an operand, an opcode, an instruction, or a set of instructions. The Yacc *actions*[9] associated with those grammar rules

[8] And for x86, the x86-specific code is in `assemblers/8a/` and `include/obj/8.out.h`.

[9] See [BLM92] for more information on Yacc's grammar rules and actions.

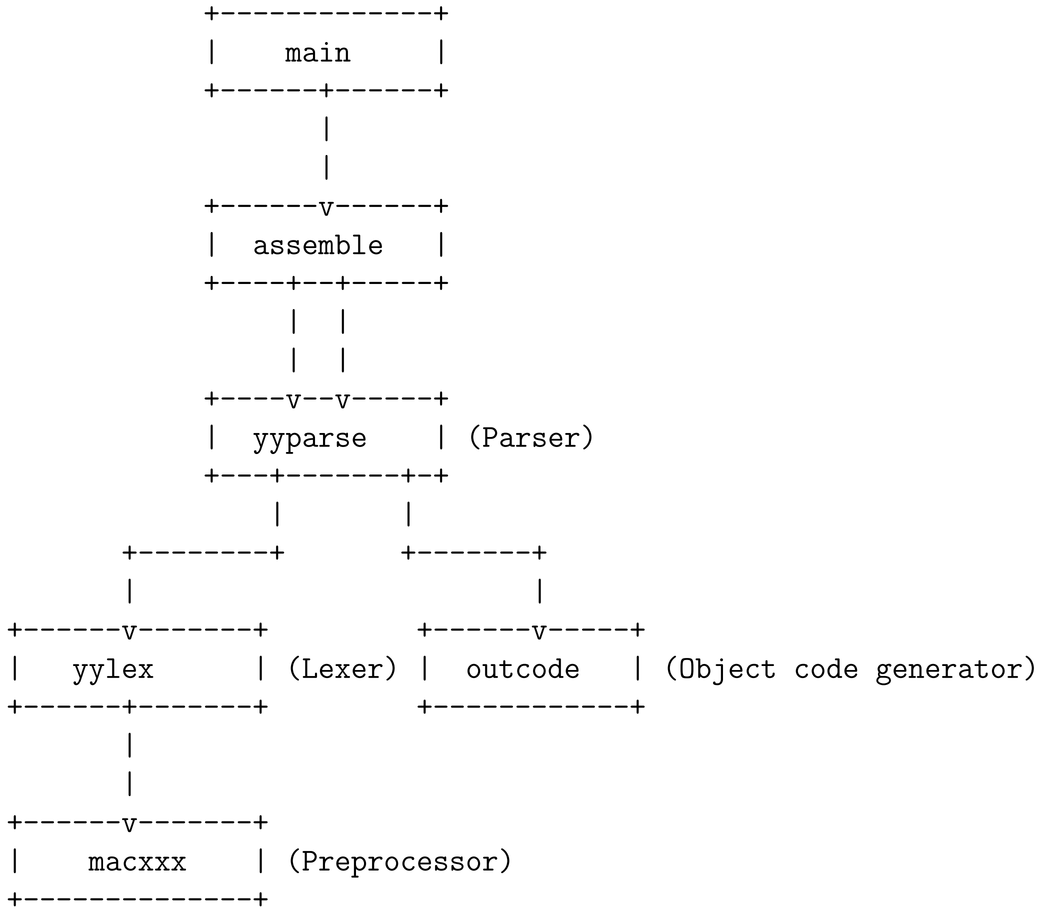


Figure 2.7: Control flow diagram of `5a`.

```
assembly file -> tokens -> instructions -> object file
                (use global symbol table)
```

Figure 2.8: Data flow diagram of `5a`.

gradually build and return abstract syntax trees (ASTs) corresponding to the different parts of an instruction. Once `yyparse()` parsed a full instruction, via the `inst`[83a] grammar rule, the Yacc action for `inst` is triggered. This action calls the function `outcode()`[108] with the AST of the parsed instruction as an argument.

`outcode()` outputs then the serialized form of the instruction's AST in the object file. However, it does so only when the global value of `pass` is 2. Whatever the value of `pass`, `outcode()` also increments each time `pc`[92a], a global representing the current value of the *virtual program counter*.

The lexing and parsing code internally uses and modifies a *symbol table* called `hash`[39a] that keeps track of the value of different kinds of identifiers, for instance, labels. During lexing, when `yylex()` encounters a new identifier, `yylex()` creates a new symbol in the symbol table `hash`. During parsing, once `yyparse()` recognizes an identifier as a label (because it is followed by a colon token), `yyparse()` changes its *type* in the symbol table to a label type and its value is set to be the current value of the global `pc`. During the second pass, `yyparse()` can resolve any reference to this label by simply looking at the value of the label in the symbol table. This is why `outcode()` outputs code only during the second pass, once `yyparse()` resolved the values of every labels in the first pass.

2.9 Book structure

You now have enough background to understand the source code of `5a`. The rest of the book is organized as follows. I will start by describing the core data structures of `5a` in Chapter 3. Then, I will use a top-down approach in Chapter 4, and, starting from `main()`[155a], I will present the code of the main functions of `5a` (e.g., `assemble()`[45d]). The following chapters will describe the main components of the assembling pipeline: Chapter 5 will present the input routines, Chapter 6 the lexer, Chapter 7 the preprocessor, Chapter 8 the parser, and finally Chapter 9 the object code generator. In Chapter 10 I will describe the code responsible for adding debugging support in `5a`. For example, this code adds line information in the object code; 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. Chapter 11 presents other assembly features that I did not present before to simplify the explanations, for instance, the support for floating-points. Those features tend to crosscut many components with extensions to the lexer, the parser, and the code generator. Finally, Chapter 12 concludes and gives pointers to other books in the Principia Softwarica series.

Some appendices present the code of non-functional properties: code to help debug `5a` itself in Appendix A, and code to manage errors in Appendix B. Appendix C contains the code of generic utilities used by `5a` but which are not specific to `5a`. Appendix D contains more examples of assembly programs to get a better feel of Asm5 and to see how it is used in practice.

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 `5a`. The first three sections contain the definitions of the different components of the abstract syntax tree (AST) of an instruction: the opcode, the register, which is a kind of operand, and the other kinds of operands. Then, you will see how tokens are represented. Finally, I will present the symbol table, which is a central data structure used by the lexer, the preprocessor, and the parser.

3.1 `Opcode`

All Asm5 instructions (machine-, pseudo-, and virtual-) have an opcode defined in `include/obj/5.out.h`[154c] by the following type:

⟨*enum* `Opcode`*(arm)* 31⟩≡ (154c)

```
// coupling: with 5c/enam.c
enum Opcode
{
    AXXX,

    ANOP, // VIRTUAL removed by linker
    // ------------------------------------------------------------
    // Arithmetic and logic opcodes
    // ------------------------------------------------------------
    ⟨Opcode cases, logic opcodes 82d⟩
    ⟨Opcode cases, add/sub opcodes 83b⟩
    ⟨Opcode cases, mul/div/mod opcodes 84b⟩
    ⟨Opcode cases, bitshift opcodes 84d⟩
    // ------------------------------------------------------------
    // Memory opcodes
    // ------------------------------------------------------------
    ⟨Opcode cases, mov opcodes 86a⟩
    ⟨Opcode cases, swap opcodes 87a⟩
    // ------------------------------------------------------------
    // Control flow opcodes
    // ------------------------------------------------------------
    ⟨Opcode cases, comparison opcodes 88b⟩
    ⟨Opcode cases, branching opcodes 87d⟩
    // ------------------------------------------------------------
    // System opcodes
```

```
    // ------------------------------------------------------------
    ⟨Opcode cases, interrupt opcodes 91c⟩
    // ------------------------------------------------------------
    // Float opcodes
    // ------------------------------------------------------------
    ⟨Opcode cases, float mov opcodes 123c⟩
    ⟨Opcode cases, float arithmetic opcodes 123e⟩
    // ------------------------------------------------------------
    // Pseudo opcodes
    // ------------------------------------------------------------
    ⟨Opcode cases, pseudo opcodes 100d⟩

    ALAST,
};
```

The opcodes follow closely the instruction set (ISA) of the ARM (see the EMULATOR book [Pad15a]) and are all prefixed by an `A` (for Assembly). For example, the `ASUBX` assembly opcode I will present later corresponds to the `SUB` ARM machine instruction. I have divided the opcodes above in different categories: arithmetic and logic, memory, control flow, system, float, and pseudo opcodes. I will present the different opcodes gradually in this book.

The first opcode above, `AXXXX`, represents the *invalid opcode*. It is in the first position of the enumeration above as a form of defensive programming.

The second opcode, `ANOPX`, is a virtual instruction removed by `5l`. It is occasionally used by the assembly generator in `5c` to represent an instruction without any effect (see the COMPILER book [Pad16b]).

Finally, the last opcode, `ALASTX`, is an *end marker* (a C idiom making it easy to loop over all the opcodes or to declare an array of opcodes).

3.2 Register

The ARM processor has 16 registers named `R0` to `R15`. They are represented simply by an integer in `5a`: 0 is `R0`, 1 is `R1`, etc. The following type introduces some convenient names for some of the special registers you have seen before:

⟨*enum* `Register`*(arm)* 32⟩≡ (154c)

```
enum Register {
    ⟨Register compiler conventions cases 131e⟩
    ⟨Register linker conventions cases 131g⟩

    // reserved by the linker
    REGSB =     12, // static base
    REGSP =     13, // stack pointer

    // reserved by the ARM processor
    REGLINK =   14,
    REGPC =     15,

    NREG = 16,
};
```

Again, `NREGX` is not a register but an end marker. With it, you can declare easily an array containing all the ARM registers with `int allregs[NREG];`.

Note than `REGSBX` and `REGSPX` are not special ARM registers. However, they are reserved by the assembler and linker for special uses.

There are few situations where something, e.g., an operand, can be a register or nothing. We can not use zero to represent nothing because zero is already used to encode `R0`. Instead, the following constant represents

nothing:

⟨*constant* R_NONE*(arm)* 33a⟩≡ (154c)
```
#define R_NONE 16
```

The ARM processor has also floating-point registers, coprocessor registers, and status registers, but I will introduce the constants representing those registers later in Chapter 11.

3.3 Operand

A register is only one kind of operand; constants (integers, floats, or strings) are another kind. The following type represents all kinds of operands:

⟨*struct* Gen*(arm)* 33b⟩≡ (167a)
```
struct  Gen
{
    // enum<Operand_kind>
    short   type;

    // switch on Gen.type
    union {
        long    offset; // offset or lval or ...
        double  dval;
        char    sval[NSNAME];
    };
    // option<enum<Register>> (None = R_NONE)
    short   reg; // abused also to store a size for DATA

    ⟨Gen other fields 42b⟩
};
```
Uses __anon_struct_2 33b.

The `Gen.type` field above encodes the *operand kind*. The type of `Gen.type` is a `short`, but `Gen.type` can take only values from the `Operand_kind` enumeration below:

⟨*enum* Operand_kind*(arm)* 33c⟩≡ (154c)
```
enum Operand_kind {
    D_NONE,

    D_CONST,
    D_SCONST,
    D_FCONST,

    D_REG,
    ⟨Operand_kind cases 34b⟩
};
```

The union in `Gen` represents the *value* of the operand. This value has to be interpreted in different ways depending on the kind of the operand: for integer constants (`D_CONSTX`), the `offset` field is used; for string constants (`D_SCONSTX`), the `sval` field is used; for float constants (`D_FCONSTX`), the `dval` field is used. Remember that characters are converted in integers by `5a` so there is no need for a `D_CHARCONST` constant and a `cval` field.

The `long` element of the union is called `offset` and not `lval` (which would be more consistent with the other union fields) because it is (ab)used to represent different things, not just integer (or character) constants, as you will see soon.

The `D_NONEX` case above is used to represent the *null operand*. Indeed, certain instructions have no operands (e.g., `RET`), but the instruction format in the object file (see Section 2.6) imposes the presence of two operands. For those instructions, the operand kind is set to `D_NONEX`.

nullgen below is a global you will often in the code of 5a. It represents the null operand. It is also used to initialize freshly allocated operands.

⟨*global* nullgen 34a⟩≡ (167b)
```
Gen nullgen;
```

D_REGX above is used to represent register operands. In that case, the Gen.reg field encodes the register number.

For indirect operands (e.g., (R1)), or indirect with offset operands (e.g., 4(R1)), the new operand kind D_OREG (for "offset register") is used:

⟨Operand_kind *cases* 34b⟩≡ (33c) 42e ▷
```
D_OREG,
```

In those cases, the register number is encoded also in the Gen.reg field, and the optional offset is stored in Gen.offset (hence the name of the field). Here is the C value of the operand mentioned above:

```
// operand value for: 4(R1)
{ .type = D_OREG;       // (...)
  .offset = 4;          // 4
  .reg    = 1;          // R1
}
```

Strings in Asm5 are limited to 8 characters, hence the use of the constant below for the Gen.sval field:

⟨*constant* NSNAME 34c⟩≡ (154c)
```
#define NSNAME      8
```

3.4 Instructions

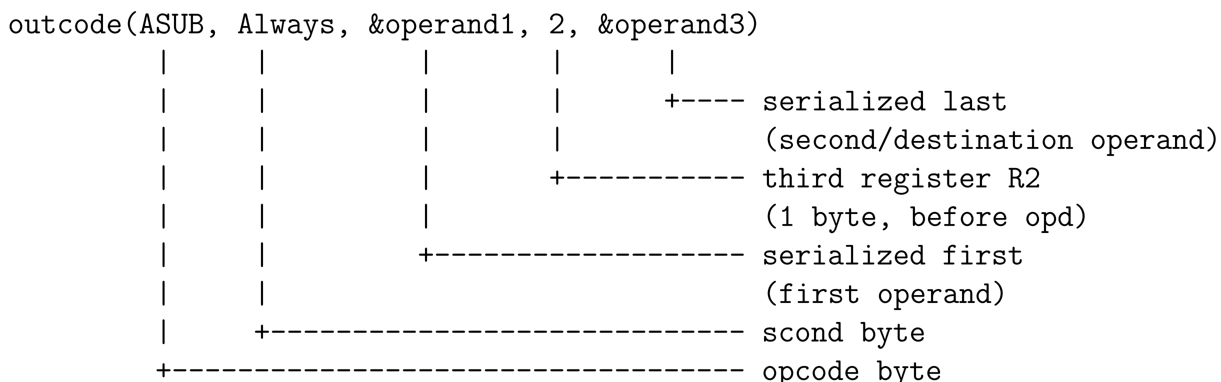
5a does not define a type to represent a full instruction. Instead, 5a uses multiple parameters of the outcode() [108] function below to represent the different components of an instruction. Here is the signature of outcode():

⟨*signature* outcode*(arm)* 34d⟩≡ (167a)
```
void    outcode(int opcode, int cond, Gen* opd1, int reg, Gen* opd3);
```

Here is an example of use of outcode() to output the serialized form of the instruction SUB $14, R2, R3:

```
Gen operand1 = { .type = D_CONST; .offset = 14; ...};
Gen operand3 = { .type = D_REG;   .reg = 3; ... };
outcode(ASUB, Always, &operand1, 2, &operand3);
```

Since 5a does not define a single "instruction record" type (no Prog-like struct as you will see in the LINKER book [Pad15b]), the five parameters of outcode() *are* the in-memory form of one instruction while a grammar action is running. The diagram below shows how the call outcode(ASUB, Always, &operand1, 2, &operand3) from the example above maps to the six slots of a serialized instruction described in Section 2.6:



```
+-----------+---------+------+---------+---------+---------+
|  opcode   | scond   | reg  | line#   | opd1    | opd3    |
| (1 byte)  | (1 byte)|(1 B) | (4 B)   |(variable|(variable|
+-----------+---------+------+---------+---------+---------+
  ASUB        Always    2      lineno    D_CONST   D_REG
                                         .off=14   .reg=3
```

The middle-operand-is-always-a-register asymmetry (why `reg` is an `int` while `opd1` and `opd3` are `Gen*`) lets `5a` serialize the three-address ARM format without allocating a third `Gen`[33b] just to hold one register number. That byte is simply written between the line number and the two variable-sized operand blobs; you will see the code in `outcode()` later in Section 9.2.

The `Always`[103b] constant in the second argument to `outcode()` is used to represent the *conditional execution* of an instruction. It is an ARM feature I will explain later in Section 8.6. The fourth parameter is an integer and not a `Gen*` because when an ARM instruction has three operands, the middle one is always a register, so an integer is enough. Depending on the number of operands, here are the constraints on the value of the arguments to `outcode()`:

- 0 operand: `opd1` and `opd3` are `&nullgen` and `reg` is `R_NONE`[33a],
- 1 operand: `opd3` is `&nullgen` and `reg` is `R_NONE`
- 2 operands: `reg` is `R_NONE`
- 3 operands: no constraint, all arguments are used.

3.5 Tokens and `itab`

The Yacc-based parser of `5a`, called via `yyparse()`[104c], is taking as input a set of tokens. Those tokens are returned by the `yylex()`[56b] function but the type of those tokens is defined by a set of *Yacc directives* in the Yacc grammar `5a/a.y`[1]. Here are the directives to declare the names of those tokens as well as the type of value they hold:

⟨*token declarations(arm)* 35⟩≡ (81a)

```
/* opcodes */
%token  <lval>  LARITH LCMP LBRANCH LBCOND LMOV LSWAP LRET
%token  <lval>  LSWI LSYSTEM
%token  <lval>  LARITHFLOAT LCMPFLOAT LSQRTFLOAT  LMULL LMULA  LMOVM LMVN
%token  <lval>  LDEF LDATA LWORD LEND
%token  <lval>  LMISC
/* registers */
%token  <lval>  LPC LSP LFP LSB
%token  <lval>  LR LREG  LPSR
%token  <lval>  LF LFREG  LFCR
%token  <lval>  LC LCREG
/* constants */
%token  <lval>  LCONST
%token  <dval>  LFCONST
%token  <sval>  LSCONST
/* names */
%token  <sym>   LNAME LLAB
%token  <sym>   LVAR
/* bits */
%token  <lval>  LCOND
%token  <lval>  LS LAT
```

[1] Again, see [BLM92] for more information on Yacc.

```
                         token     token
                         code      value

 +------------------+     +---------+---------+
 |TEXT  -----------+---->|  LDEF   |  ATEXT  |
 |                  |     +---------+---------+
 |                  |     +---------+---------+
 |   SUB ... ------+---->| LARITH  |  ASUB   |
 |                  |     +---------+---------+
 |                  |     +---------+---------+
 |   ... 42  ------+---->| LCONST  |   42    |
 |                  |     +---------+---------+
 |                  |     +---------+---------+
 |   ... 3.7 ------+---->| LFCONST |   3.7   |
 |                  |     +---------+---------+
 |                  |     +---------+---------+
 |   ... "foo" ----+---->| LSCONST |  "foo"  |
 |                  |     +---------+---------+
 |                  |     +---------+---------+
 |   NOP ----------+---->|  LMISC  |  ANOP   |
 |                  |     +---------+---------+
 |                  |     +---------+---------+
 |   RET ----------+---->|  LRET   |  ARET   |
 |                  |     +---------+---------+
 |                  |     +---------+---------+
 |GLOBL ....-------+---->|  LDEF   | AGLOBL  |
 +------------------+     +---------+---------+

  Assembly file                 tokens
```

Figure 3.1: Examples of tokens.

Those directives are a bit complicated but things will be clearer when I will describe the lexer in Chapter 6. To understand those directives, it can be useful to look at the code generated by Yacc and to look at how tokens are internally represented. A Yacc token is a pair composed of the following two elements:

- The *token code* is an integer representing the kind of the token. Based on the directives above, Yacc will generate a `y.tab.h` header file where all the token names are assigned a code. See an excerpt of this file below. For example, 57346 represents all the arithmetic mnemonics tokens (e.g., `ADD`, `SUB`, etc.), whereas 57353 represents the token for the `RET` mnemonic, and 57381 represents all integer tokens.

- The *token value* is a union I will describe below. Depending on the kind of the token, the value can be an integer (e.g., for integer tokens), a float (for float tokens), or something else. For instance, for the arithmetic mnemonic tokens (represented by the code 57346 and the token name `LARITH`[35]), the value will be the opcode corresponding to the actual mnemonic (e.g., `ASUBX` for `SUB`), which is an enumeration case.

Figure 3.1 illustrates how `5a` converts certain strings in the assembly file in tokens.

Here is an excerpt of the `y.tab.h` file generated by Yacc:

⟨*5a/y.tab.h excerpt* 36⟩≡

```
...
```

```
#define LARITH 57346
...
#define LSWI 57352
#define LRET 57353
...
#define LCONST 57381
#define LFCONST 57382
...
```

The values of the integers in `y.tab.h` are high to not conflict with the values of regular *unicode* characters. Indeed, `yylex()` can also return single-character tokens, e.g., `';'`, in which case there is no need to give them a name.

Here is finally the union representing the different kinds of values a token can hold:

⟨*union declaration(arm)* 37a⟩≡ (81a)

```
%union {
 // long for LCONST
 // and enum<Opcode>  for LARITH/...
 // and enum<Register> for LREG/...
 // and enum<Cond>    for LCOND
 // and ...
 long   lval;    // for LCONST/LARITH/LREG/LCOND/...
 double dval;    // for LFCONST
 char   sval[NSNAME]; // for LSCONST

 ⟨union declaration other fields(arm) 42a⟩
}
```

Again, this directive is a bit complicated but things will be clearer later. The name of those fields (e.g., `lval`), are referenced in the `%token` directives above. For instance, `%token <lval> LCONST` indicates that the `LCONST`[35] tokens will use the `lval` field of the union to hold their values.

Another important data structure related to tokens is the `Itab` structure and the global `itab` below:

⟨*struct* `Itab`*(arm)* 37b⟩≡ (168a)

```
struct Itab
{
    char *name;

    //enum<token_code>
    ushort type;
    //enum<Opcode> | enum<Register> | ... | int
    ushort value;
};
```

⟨*global* `itab`*(arm)* 37c⟩≡ (168a)

```
// map<string, (token_code * token_value)>
struct Itab itab[] =
{
    "NOP",  LMISC, ANOP,
    ⟨itab elements 82e⟩
    0
};
```

Uses `Itab` 37b and `LMISC`.

`itab` maps the predefined identifiers of Asm5 (e.g., `NOP`, `PC`) to their corresponding tokens, which as you have seen before are pairs of a token code and a token value. I will reveal gradually elements of the `itab` array in this book, but the first entry is shown above for `NOP`.

The three data structures I have introduced in this section (the Yacc `%union`, the `y.tab.h` codes, and `itab`) click together only once you trace a single identifier through `5a` end-to-end. Below I sketch the path of the string `"SUB"` from the input file all the way to what `yylex()` returns to `yyparse()`:

```
source byte-stream           "...  SUB $1, R2, R3 ..."
                                    |
                                    v lexer reads word
                               symb = "SUB"
                                    |
                                    v lookup(): hash the string
                               hash[h] --> Sym{
                                             name  = "SUB"
                                             type  = LARITH     (set by cinit)
                                             value = ASUB       (set by cinit)
                                           }
                                    |
                                    v yylex() returns
                            (tokencode, tokenvalue)
                            (LARITH ,  ASUB     )
                                ^         ^
                                |         |
                         %token<lval>     |
                          directive       union field lval
                          picks LARITH    of %union{...}

itab[]   (static table, walked once by cinit)
+----------+-------+--------+
|  "NOP"   | LMISC | ANOP   |   --> slookup("NOP")->type/value
|  "SUB"   | LARITH| ASUB   |   --> slookup("SUB")->type/value
|  "R0"    | LREG  | 0      |   --> slookup("R0") ->type/value
|  ...     |  ...  | ...    |
+----------+-------+--------+
```

`itab` is thus just the *static seed* of the symbol table. At startup, `cinit()`[44c] loops over `itab`, calls `slookup()`[39e] for each entry, and copies `type` and `value` into the freshly-allocated `Sym`[156b]. After that, the lexer no longer needs `itab` at all: every predefined mnemonic or register is now a regular symbol that lives in `hash`[39a] side-by-side with the user's labels. The two fields of `Itab` are overloaded: for `LARITH` entries the `value` is an `Opcode`[31], for `LREG` it is a register number, for `LCOND` it is a condition code, and so on. The interpretation is dictated by `type`, which is exactly what the Yacc `%type` / `%union` machinery needs to know in order to decide which union field to use when an action accesses $\$n$. So `itab` is really a tiny untyped map and the Yacc directives bolt a type system back onto it.

3.6 Symbols and hash table

One of the main jobs of an assembler is to manage symbols. The *symbol table* is thus a central data structure in the code of `5a`. The structure below represents a symbol. It essentially associates a name to a value:

⟨*struct* `Sym` 38⟩≡ (156b)

```
struct  Sym
{
    // Sym is (ab)used to represent many things in the assembler:
    //   actual symbols, labels, but also macros,
    //   tokens for opcodes and registers, etc.

    // ---------------------------------------------------------
    // The "key"
```

```
    // -------------------------------------------------------------
    // ref_own<string>
    char    *name;

    // -------------------------------------------------------------
    // The generic "value"
    // -------------------------------------------------------------
    long    value;

    ⟨Sym identifier fields 112a⟩
    ⟨Sym macro fields 132f⟩
    ⟨Sym token fields 41e⟩

    // -------------------------------------------------------------
    // Extra
    // -------------------------------------------------------------
    ⟨Sym extra fields 39c⟩
};
```

As an example, the label `later` in the `helloworld.s` program in Section 2.3, once resolved, could be represented by the following `Sym`[156b]: `.name = "later"; .value = 2; ...` . Indeed, the label is defined after the second instruction in the `helloworld.s` program.

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 assembling pipeline (e.g., the lexer, the parser). 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` 39a⟩≡ (160a)

```
// hash<string, ref_own<Sym>> (next = Sym.link in bucket)
Sym* hash[NHASH];
```

Uses `NHASH` 39b.

⟨*constant* `NHASH` 39b⟩≡ (156b)

```
#define NHASH       503
```

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* 39c⟩≡ (38)

```
// list<ref<Sym>> (from = hash)
Sym*    link;
```

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

⟨*constant* `S` 39d⟩≡ (156b)

```
#define S   ((Sym*)nil)
```

The main interface to the symbol table are the functions `slookup()` and `lookup()`[40c] below. They both internally use the global `hash`. `slookup()`, given a name, returns the `Sym` in the symbol table `hash` associated with this name, or a new symbol if the name was not found:

⟨*function* `slookup` 39e⟩≡ (160b)

```
Sym*
slookup(char *s)
{

    strcpy(symb, s);
    return lookup();
}
```

Uses `lookup()` 40c and `symb` 40a.

`slookup()`[39e] modifies the global `symb` (used also by `yylex()`[56b] as you will see later) before calling `lookup()`.

⟨*global* `symb` 40a⟩≡ (160a)
```
char symb[NSYMB];
```
Uses `NSYMB` 40b.

⟨*constant* `NSYMB` 40b⟩≡ (156b)
```
#define NSYMB       500
```

Here is finally the code of `lookup()`:

⟨*function* `lookup` 40c⟩≡ (160b)
```
Sym*
lookup(void)
{
    Sym *sym;
    long h;
    int c;
    int len;
    ⟨lookup() other locals 41a⟩

    ⟨lookup() compute hash value h of symb 41b⟩

    // hash_lookup(symb, hash)
    c = symb[0];
    for(sym = hash[h]; sym != S; sym = sym->link) {
        // fast path
        if(sym->name[0] != c)
            continue;
        // slow path
        if(memcmp(sym->name, symb, len) == 0)
            return sym;
    }
    ⟨lookup() if symbol name not found 40d⟩
}
```
Uses `S` 39d, `hash` 39a, and `symb` 40a.

If the symbol is not found in the symbol table, a fresh symbol is created and added in the hash table:

⟨`lookup()` *if symbol name not found* 40d⟩≡ (40c)
```
sym = alloc(sizeof(Sym));
sym->name = alloc(len);
memmove(sym->name, symb, len);

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

syminit(sym);
return sym;
```
Uses `alloc()` 151b, `hash` 39a, `symb` 40a, and `syminit()` 40e.

⟨*function* `syminit` 40e⟩≡ (168a)
```
void
syminit(Sym *sym)
{
    sym->type = LNAME;
    sym->value = 0;
}
```
Uses `LNAME`.

The hashing code below iterates over the characters in `symb` until the end-of-string character (`0`). It also computes the length of the symbol in `len`:

⟨`lookup()` *other locals* 41a⟩≡ (40c)
```
  char *p;
```

⟨`lookup()` *compute hash value* `h` *of* `symb` 41b⟩≡ (40c)
```
  // h = hashcode(symb); len = strlen(symb)
  h = 0;
  for(p=symb; c = *p; p++)
      h = h+h+h + c;
  len = (p - symb) + 1;
  if(h < 0)
      h = ~h;
  h %= NHASH;
```
Uses `NHASH` 39b and `symb` 40a.

The symbol table `hash` is used (you could say abused) to represent and keep track of many different things in `5a`:

- *labels*, e.g., `later:`, in which case `Sym.valueX` contains eventually the value of the (virtual) program counter `pc`[92a] at the label definition,
- *symbols*, for procedures (e.g., `TEXT foo(SB)`) and globals (e.g., `GLOBL hello(SB)`), in which case `Sym.valueX` is not used because the addresses of global symbols are not resolved by `5a` (but but `5l`). However, another field of `Sym` is used to store metadata about the symbol. This metadata will be stored in the symbol table of the object file as you will see in Section 9.5.
- *macros*, e.g `#define FOO` (`5a` is a macro-assembler that embeds its own macroprocessor), in which case `Sym.valueX` is not used. Another field of `Sym` contains the text of the macro as you will see in Section 11.5.1.
- *symbolic constants*, e.g., `RTMP = 10`, in which case `Sym.valueX` contains the constant value.
- *predefined identifiers*, for opcodes (e.g., `MOVW`) and registers (e.g., `R1`), in which case `Sym.valueX` contains the appropriate enumeration value (e.g., `AMOVW`).

Predefined identifiers are stored in the symbol table for efficiency reason. Indeed, the symbol table is first a hash table, and so it can be (ab)used by the lexer to quickly check if an identifier corresponds to the mnemonic of an opcode or register. One of the initialization functions of `5a`, `cinit()`[44c], resets and then populates `hash` with all the array elements of `itab`[37c] you have seen in the previous section:

⟨`cinit()` *locals* 41c⟩≡ (44c)
```
  Sym *s;
  int i;
```

⟨`cinit()` *hash initialization from itab* 41d⟩≡ (44c)
```
  for(i=0; i<NHASH; i++)
      hash[i] = S;
  for(i=0; itab[i].name; i++) {
      s = slookup(itab[i].name);
      s->value = itab[i].value;
      s->type = itab[i].type;
  }
```
Uses `NHASH` 39b, `S` 39d, `hash` 39a, `itab` 37c, and `slookup()` 39e.

In the case of predefined identifiers, another field of `Sym` is used to store the token code of the identifier (in addition to the token value stored in `Sym.valueX`):

⟨`Sym` *token fields* 41e⟩≡ (38)
```
  //enum<token_code> (e.g., LLAB, LNAME, LVAR, LARITH)
  ushort  type;
```

`Sym` is part of the token union we have seen in Section 3.5. Indeed, the token value for symbols (`LNAME`[35]), labels (`LLAB`[35]), and symbolic constants (`LVAR`[35]) is a direct reference to its `Sym` in the symbol table:

⟨*union declaration other fields(arm)* 42a⟩≡ (37a) 92f ▷
```
Sym    *sym;    // for LNAME/LLAB/LVAR
```

`Sym` is also a possible part of an operand when the operand involves a symbol (or label):

⟨`Gen` *other fields* 42b⟩≡ (33b) 42c ▷
```
// option<ref<Sym>> (owner = hash)
Sym*    sym;
```

For example, in `hello+4(SB)`, `Gen.sym` will point to the symbol `hello` in the global `hash` table.

3.7 `Sym_kind`

For operands involving symbols, e.g., `hello+4(SB)`, an additional `Gen`[33b] field is used to represent the *kind* of the symbol:

⟨`Gen` *other fields* 42c⟩+≡ (33b) ◁ 42b
```
// option<enum<Sym_kind>> (None = N_NONE)
short   symkind;
```

⟨*enum* `Sym_kind`*(arm)* 42d⟩≡ (154c)
```
enum Sym_kind {
    N_NONE,

    N_EXTERN, // text/data/bss values (from SB)
    N_LOCAL,  // stack values (from SP)
    N_PARAM,  // parameter (from FP)
    ⟨Sym_kind cases 99b⟩
};
```

Thus, the C value of the operand in the example above is:

```
// operand value for: hello+4(SB)
{ .type = D_OREG;       // (...)
  .offset = 4;          // +4
  .sym = &<hello Sym>;  // hello
  .sym_kind = N_EXTERN; // SB
}
```

Finally, `5a` uses another operand kind, `D_ADDR`, to encode operands such as `$hello(SB)`.

⟨`Operand_kind` *cases* 42e⟩+≡ (33c) ◁ 34b 95d ▷
```
D_ADDR,
```

Here is an example of C value for such an operand:

```
// operand value for: $hello(SB)
{ .type = D_ADDR;       // $
  .offset = 0;
  .sym = &<hello Sym>;  // hello
  .sym_kind = N_EXTERN; // SB
}
```

Chapter 4

Main Functions

.There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.

C.A.R Hoare

I now switch from a bottom-up approach in Chapter 3, to a top-down approach in this chapter. Indeed, I will describe the main functions of `5a`, starting from `main()`[155a] down to `assemble()`[45d].

4.1 `main()`

Before showing the code of `main()`[155a], I first introduce a few globals set by `main()`.

Two globals common in Plan 9 code are `thechar` and `thestring`. They both represent the current architecture. As I said in Section 1.2, 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 `5a` for many things, for instance, for the filename extension of object files (e.g., `helloworld.5`). This character is stored in the following global:

⟨*global* `thechar` 43a⟩≡ (160a)
```
int thechar;
```

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

⟨*global* `thestring` 43b⟩≡ (160a)
```
char* thestring;
```

`thestring` is used by `5a` to find architecture-specific system header files (e.g., in `/arm/include/`). Indeed, `5a` is a macro-assembler and `#include` have an *include search path*.

I can now present the code of `main()`, the entry point of `5a`. The most important thing in the code below is the call to `assemble()`[45d] at the end, with the filename of the assembly file to process passed through a command-line argument:

⟨*function* `main`*(arm)* 43c⟩≡ (168c)
```
void
main(int argc, char *argv[])
{
    errorn err;
    ⟨main() locals 73a⟩

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

    cinit();
```

```
    ⟨main() remaining initializations 72f⟩

    ARGBEGIN {
    ⟨main() command line processing 44b⟩
    } ARGEND

    if(*argv == '\0') {
        print("usage: %ca [-options] file.s\n", thechar);
        errorexit();
    }

    ⟨main() multiple files handling 142b⟩
    // else

    err = assemble(argv[0]);
    if(err > 0)
        errorexit();
    else
        exits(nil);
}
```

Another important global, set possibly by `main()`, is `outfile`. `outfile` stores the name of the output object file, which can be modified by the `-o` option:

⟨*global* `outfile` 44a⟩≡ (160a)

```
char* outfile = nil;
```

Uses `outfile` 44a.

⟨`main()` *command line processing* 44b⟩≡ (43c) 73b ▷

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

The macros `ARGBEGIN`, `ARGEND`, and `ARGF`, used in `main()` above, are defined in the C library (see the LIBCORE book [Pad16c]). They allow to iterate over command-line arguments in a simple way.

4.2 `cinit()`

`cinit()`, called from `main()`[155a], contains the initializations of a few globals:

⟨*function* `cinit`*(arm)* 44c⟩≡ (168a)

```
/// main -> <>
void
cinit(void)
{
    ⟨cinit() locals 41c⟩

    ⟨cinit() nullgen initialization 44d⟩
    ⟨cinit() hash initialization from itab 41d⟩
    ⟨cinit() pathname initialization from cwd 45b⟩
}
```

I mentioned before the global `nullgen`[34a]. Here is the code initializing `nullgen`:

⟨`cinit()` *nullgen initialization* 44d⟩≡ (44c) ?? ▷

```
nullgen.type    = D_NONE; // no operand type set yet
nullgen.reg     = R_NONE;
nullgen.symkind = N_NONE;
nullgen.sym = S;
nullgen.offset = 0; // part of a union
```

Uses `S` 39d and `nullgen` 34a.

pathname below represents the *current working directory* (CWD). This global will be used when 5a generates debugging information in the object file (see Section 10.4).

⟨*global* pathname 45a⟩≡ (160a)
```
char* pathname;
```

⟨cinit() *pathname initialization from cwd* 45b⟩≡ (44c)
```
pathname = allocn(pathname, 0, 100);
if(getwd(pathname, 99) == 0) {
    pathname = allocn(pathname, 100, 900);
    if(getwd(pathname, 999) == 0)
        strcpy(pathname, "/???");
}
```
Uses allocn() 151c and pathname 45a.

4.3 assemble()

The most important function called from main()[155a] is assemble(). This is where resides most of the assembling logic. Because a programmer can reference labels defined later in an assembly file, a common approach for assembling is to use a *two-pass* algorithm. The first pass focuses on the *definitions* of labels; by the end of the file, the values of all labels are known. The second pass focuses on the *uses* of labels; the second pass leverages the information computed in the first pass. This is the strategy used by 5a and so an important global is pass, which stores in which pass we currently are (1 or 2):

⟨*global* pass 45c⟩≡ (160a)
```
// 1|2
int pass;
```

pass is set by assemble() below. It is then used in a few places in the grammar as well as in outcode()[108] as you will see later:

⟨*function* assemble 45d⟩≡ (168c)
```
errorn
assemble(char *infile)
{
    fdt of; // outfile
    ⟨assemble() locals 47c⟩

    ⟨assemble() set p to basename(infile) and adjust include 47e⟩
    if(outfile == nil) {
        ⟨assemble() set outfile to basename(infile).thechar 49⟩
    }
    ⟨assemble() setinclude("/thestring/include") or use INCLUDE 73h⟩

    of = mycreat(outfile, 0664);
    ⟨assemble() sanity check of 46a⟩
    Binit(&obuf, of, OWRITE);

    // Pass 1
    pass = 1;
    pinit(infile);
    ⟨assemble() init Dlist after pinit 132e⟩
    yyparse(); // calls outcode(), which does almost nothing when pass == 1
    ⟨assemble() sanity check nerrors 46b⟩

    // Pass 2
    pass = 2;
    outhist(); // output file/line history information in object file
```

```
    pinit(infile);
    ⟨assemble() init Dlist after pinit 132e⟩
    yyparse(); // calls outcode() which now does things

    cclean();
    return nerrors;
}
```
Uses cclean() 47b, mycreat() 159a, nerrors 148b, obuf 47a, outfile 44a, outhist() 119b, pass 45c, pinit() 50a, and yyparse().

⟨assemble() *sanity check* of 46a⟩≡ (45d)
```
if(of < 0) {
    yyerror("%ca: cannot create %s", thechar, outfile);
    errorexit();
}
```
Uses errorexit() 148a, outfile 44a, thechar 43a, and yyerror() 148c.

`assemble()`[45d] calls `yyparse()`[104c] two times, with the appropriate value set in `pass` first, and after having also initialized some globals related to the input file with `pinit()`[50a]. This means the input assembly file is parsed two times, but the actions in the grammar do different things the second time.

A tiny concrete example shows what each pass contributes. Suppose the input file contains a forward jump to a label that is only defined further down:

```
TEXT    main(SB), $0
        MOVW    $0, R0
        B       later     ; forward reference -- not yet known
        MOVW    $1, R1
later:
        RET

pass            | outcode() behaviour    | label table (Sym.value)
----------------+------------------------+------------------------
pass == 1       | returns immediately    | MOVW at virtual pc 0
(‘‘seek         | after pc++             | B    at virtual pc 1
 labels’’)      |                        | MOVW at virtual pc 2
                |                        | later: defined = 3
                |                        | RET  at virtual pc 3
----------------+------------------------+------------------------
pass == 2       | actually serializes    | every label Sym already
(‘‘emit         | each instruction to    | has its final .value,
 bytes’’)       | obuf                   | so B can resolve
                |                        | "later" to offset 3
```

Note that the grammar is the same in both passes—it is only the semantic actions that diverge. `outcode()` checks `if(pass == 1) goto out;` and bails out before writing any bytes in pass 1, but still increments `pc`. The label-definition rules in `a.y` always record the current `pc` into the `Sym` they are defining, so by the end of pass 1 the symbol table holds the final virtual address of every label. Pass 2 then emits bytes and, whenever it encounters a label use, looks up the (now-known) `Sym.value` to compute the branch offset.

`assemble()` returns the number of errors it found or zero if everything went fine. The functions called from `assemble()` increment the global `nerrors`[148b] to indicate an error, hence the check below:

⟨assemble() *sanity check nerrors* 46b⟩≡ (45d)
```
if(nerrors) {
    cclean();
    return nerrors;
}
```
Uses cclean() 47b and nerrors 148b.

See Appendix B for more information on the error management in 5a.

Another global initialized by assemble() is obuf, the output buffer that will contain the content of the object file generated by 5a:

⟨*global* obuf 47a⟩≡ (160a)
```
Biobuf obuf;
```

This global will be used notably by outcode(). The type of obuf is Biobuf (for "buffered input/ouput buffer"). Biobuf is part of the libbio Plan 9 library; its interface is quickly described in Appendix C.1.

The main output data flow in assemble() is to go from the name of the output file (outfile[44a]), to an output file descriptor (of), to finally an output buffer (obuf).

Note that 5a relies on many globals for outputting data (e.g., obuf), as well as for inputting data (as you will see in the code of pinit() later), which makes the code harder to read. This is because yyparse(), the function generated by Yacc, does not take any argument nor return anything, so the interface of yyparse() imposes to use globals.

The function below is called just before returning from assemble(). cclean() adds the last instruction in the object file. This instruction uses the special opcode AENDX. AENDX is a special marker convenient to have when dealing with libraries in 5l (see the Linker book [Pad15b]). Indeed, libraries are little more than object files concatenated together; the AENDX opcode represents a boundary between object files.

⟨*function* cclean*(arm)* 47b⟩≡ (168c)
```
/// main -> assemble -> <>
void
cclean(void)
{

    outcode(AEND, Always, &nullgen, R_NONE, &nullgen);
    Bflush(&obuf);
}
```
Uses Always 103b, nullgen 34a, obuf 47a, and outcode() 108.

The remaining code in assemble() consists mostly in filename manipulations and modifications of globals used for #include processing. Those filename manipulations use the temporary variable ofile:

⟨assemble() *locals* 47c⟩≡ (45d) 47d ▷
```
char ofile[100];
```

This temporary is then used to compute outfile. The value of outfile is derived automatically from the input file infile (unless you used the -o option of 5a). For example, given the input file /tests/5a/foo.s, 5a will generate the object file in the output file foo.5 in the current directory. Figure 4.1 illustrates the evolution of ofile, outfile, infile, and the local variable p while executing the code below when the input file is /tests/5a/foo.s.

⟨assemble() *locals* 47d⟩+≡ (45d) ◁ 47c 73g ▷
```
char *p;
```

⟨assemble() *set p to basename(infile) and adjust include* 47e⟩≡ (45d)
```
// p = basename(infile)
strcpy(ofile, infile);
p = utfrrune(ofile, '/');
if(p) {
    *p++ = '\0';
    ⟨assemble() adjust first entry in include with dirname infile 72g⟩
} else
    p = ofile;
```

```
           +--------------------+
  infile:  |/tests/5a/foo.s     |
(argv[0])  +--------------------+              start
           0                         100         of
           +--------------------------+       assemble
  ofile:   |                          |
           +--------------------------+
outfile:    nil
      p:    nil

                      p                         after
           0          |              100       setting
           +----------v---------------+         p to
  ofile:   |/tests/5a|foo.s           |       basename
           +--------------------------+


                   outfile
                      |  p
           0          |  |           100    just after
           +----------v--v------------+        call
  ofile:   |/tests/5a|foo.s           |         to
           +--------------------------+      utfrrune


                   outfile
                      |  p
           0          |  |            100
           +----------v--v------------+        end
  ofile:   |/tests/5a|foo.5           |         of
           +--------------------------+      assemble
```

Figure 4.1: Evolution of `ofile` and other variables in `assemble()`.

The function `utfrrune()` (for "UTF reverse rune search") called above comes from the C library and returns the *last* occurrence of a rune in a string. A *rune* is a Unicode character in Plan 9 terminology (see the LIBCORE book [Pad16c] for more information about Unicode and runes). `5a` accepts filename arguments using unicode characters, hence the use of `utfrrune()` above and below.

⟨`assemble()` *set outfile to basename(infile).thechar* 49⟩≡ (45d)

```
// outfile =  p =~ s/.s/.5/;
outfile = p;
if(outfile){
    p = utfrrune(outfile, '.');
    if(p)
        if(p[1] == 's' && p[2] == '\0')
            p[0] = '\0';
    p = utfrune(outfile, '\0');
    p[0] = '.';
    p[1] = thechar;
    p[2] = '\0';
} else
    outfile = "/dev/null";
```

Uses `outfile` 44a and `thechar` 43a.

Chapter 5

Input

Now that you have seen `assemble()`[45d], I can start explaining the different components in the assembling pipeline, starting in this chapter with the input functions.

The most important functions I will present in this chapter are `pinit()`[50a] and `GETC()`[53f]. I mentioned before `pinit()`; it initializes globals related to input. `GETC()`, which is called by the lexer `yylex()`[56b] (called itself by the parser `yyparse()`[104c]), uses those globals to return the next character from the input assembly file.

5.1 pinit()

`pinit()` initializes a few globals related to input. It is called at the beginning of each pass in `assemble()`[45d] (see the code of `assemble()`):

⟨*function* `pinit` 50a⟩≡ (161c)

```
/// main -> assemble -> { <> ; yyparse } x 2
void
pinit(char *f)
{
    ⟨pinit() locals ??⟩

    lineno = 1;

    newio(); // set ionext
    newfile(f, FD_NONE); // use ionext, set iostack, set fi

    ⟨pinit() initializations 57c⟩
}
```

Uses `FD_NONE` 51b, `lineno` 50b, `newfile()` 52d, and `newio()` 51g.

The `lineno` global, set above, tracks the current line number; `lineno` will be incremented by the lexer after each newline.

⟨*global* `lineno` 50b⟩≡ (160a)

```
long lineno;
```

`lineno` is used in the object code generator to remember from where each instruction comes from. This will be useful for debugging as explained in Chapter 10.

The calls to `newio()`[51g] and `newfile()`[52d]] in `pinit()`[50a] above will be explained in the following sections.

5.2 File management: iostack

`5a` is a macro-assembler. Moreover, `5a` embeds in its code its own `cpp`-like preprocessor [1]. Because `5a` supports

[1]Note that `5a` could instead rely on an external preprocessor. However, doing so would be slower because of the need to fork

`#include`, the input assembly file can include other files, which themselves can include other files, and so on. Thus, `5a` may have to open many files. Moreover, when a file has been fully processed, `5a` needs to go back to the file which was including it, the *includer*. This suggests the use of a *stack* data structure to represent the stack of opened input files.

The structure below represents an element of this stack. It is essentially a file descriptor referencing an opened file:

⟨*struct* Io 51a⟩≡ (156b)

```
struct  Io
{
    // option<fdt> (None = FD_NONE)
    fdt f;
    ⟨Io buffer fields 53a⟩
    // Extra
    ⟨Io extra fields 51d⟩
};
```

⟨*constant* FD_NONE 51b⟩≡ (156b)

```
#define FD_NONE (-1)
```

The global `iostack` below represents the top of the stack, that is the current file being scanned:

⟨*global* iostack 51c⟩≡ (160a)

```
// list<ref_own<Io> (next = Io.link)
Io* iostack = I;
```

Uses `I` 51e and `iostack` 51c.

`5a` uses a list data structure to represent the stack of opened files:

⟨Io *extra fields* 51d⟩≡ (51a)

```
// list<ref_own<Io>> (from = iostack or iofree)
Io* link;
```

⟨*constant* I 51e⟩≡ (156b)

```
#define I   ((Io*)nil)
```

The main job of `newio()`[51g], called from `pinit()`[50a], is to make the global `ionext` point to a newly allocated `Io`[156b] that can be used later by `newfile()`[52d] (also called from `pinit()`):

⟨*global* ionext 51f⟩≡ (160a)

```
// option<ref<Io>>
Io* ionext;
```

⟨*function* newio 51g⟩≡ (161c)

```
/// main -> assemble -> pinit -> <>; newfile
void
newio(void)
{
    Io *i;

    ⟨newio() allocate a new Io in i or find a free one 52b⟩
    ionext = i;
    i->f = FD_NONE;
    i->c = 0;
}
```

Uses `FD_NONE` 51b and `ionext` 51f.

---

another process in the assembling pipeline.

The `Io` allocator mentioned in the code above could be simply a call to `malloc()`. The matching call to `free()` should then be in the code that closes the opened file when the content is fully read. Instead, `5a` uses the common C technique of a *free list* to manage `Io`s. By using a free list, `5a` can "recycle" an `Io` for another file:

⟨*global* `iofree` 52a⟩≡ (160a)
```
// list<ref<Io>> (next = Io.link)
Io* iofree = I;
```
Uses `I` 51e and `iofree` 52a.

⟨`newio()` *allocate a new Io in* `i` *or find a free one* 52b⟩≡ (51g)
```
static int pushdepth = 0;

i = iofree;
if(i == I) {
    ⟨newio() sanity check depth of macro expansion 52c⟩
    i = alloc(sizeof(Io));
} else
    // pop(iofree)
    iofree = i->link;
```
Uses `I` 51e, `alloc()` 151b, and `iofree` 52a.

⟨`newio()` *sanity check depth of macro expansion* 52c⟩≡ (52b)
```
pushdepth++;
if(pushdepth > 1000) {
    yyerror("macro/io expansion too deep");
    errorexit();
}
```
Uses `errorexit()` 148a and `yyerror()` 148c.

`newfile()`, which assumes `ionext` has been set correctly in `newio()`, can finally open the input assembly file:

⟨*function* `newfile` 52d⟩≡ (161c)
```
/// main -> assemble -> pinit -> { newio; <> }
/// yylex -> macinc -> <>
void
newfile(char *s, fdt f)
{
    Io *i;

    // add_list(ionext, iostack)
    i = ionext;
    i->link = iostack;
    iostack = i;

    i->f = f;
    if(i->f == FD_NONE)
        i->f = open(s, OREAD);
    ⟨newfile() sanity check i->f 52e⟩
    fi.c = 0;
    ⟨newfile() call linehist 115f⟩
}
```
Uses `FD_NONE` 51b, `fi` 53c, `ionext` 51f, and `iostack` 51c.

⟨`newfile()` *sanity check* `i->f` 52e⟩≡ (52d)
```
if(i->f < 0) {
    yyerror("%ca: %r: %s", thechar, s);
    errorexit();
}
```
Uses `errorexit()` 148a, `thechar` 43a, and `yyerror()` 148c.

5.3 Buffer management: fi

Once the input file is opened (and accessible in iostack->f), the lexer needs to look at its content one character at a time. Instead of using many system calls such as read(..., 1) to read one character at a time, which would be slow, 5a instead calls read() once in a while to fill a large *input buffer*. 5a then uses auxiliary functions to move pointers in this buffer. This buffer is simply an array of characters stored in Io.b:

⟨Io *buffer fields* 53a⟩≡ (51a) 53e▷
```
char    b[BUFSIZ];
```

⟨*constant* BUFSIZ 53b⟩≡ (156b)
```
#define BUFSIZ      8192
```

Each opened file has its own input buffer. This buffer will be referenced by fields in the global below.

⟨*global* fi 53c⟩≡ (160a)
```
struct Fi fi;
```
Uses Fi 53d.

Indeed, the global fi contains pointers in the input buffer of the currently processed input file:

⟨*struct* Fi 53d⟩≡ (156b)
```
struct Fi
{
    // ref<char> (target = Io.b)
    char*   p;
    // remaining characters in Io.b to read
    int     c;
};
```

Once 5a processed an included file, 5a needs to go back to the includer file (which can be found easily thanks to iostack->link). However, 5a needs to remember in which position it was in the includer file, hence those extra fields:

⟨Io *buffer fields* 53e⟩+≡ (51a) ◁53a
```
// like Fi, saved pointers in Io.b
char*   p;
short   c;
```

5.4 GETC()

I can finally show the code of GETC(), which will use the globals iostack[51c] and fi[53c] I mentioned before. GETC() is a macro called many times in yylex()[56b] to get the next character from the input file:

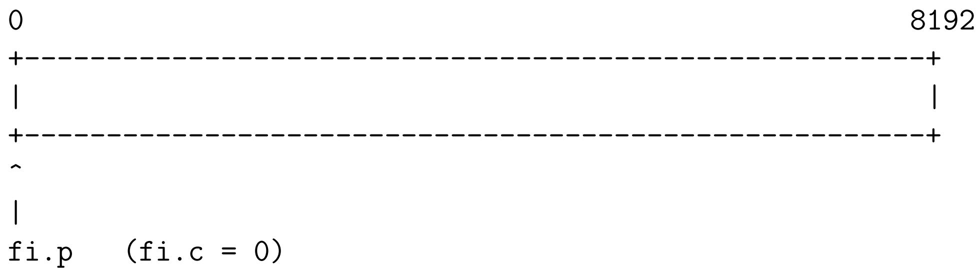
⟨*function* GETC 53f⟩≡ (156b)
```
/// main -> assemble -> yyparse -> yylex -> <>
#define GETC()      ((--fi.c < 0) ? filbuf() : *fi.p++ & 0xff)
```

GETC()[53f] is a complicated macro. Moreover, it relies also on the complicated function below:

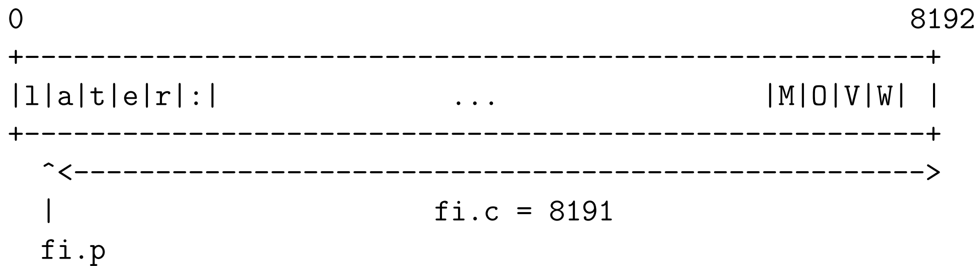
⟨*function* filbuf 53g⟩≡ (161c)
```
int
filbuf(void)
{
    Io *i;

loop:
    i = iostack;
    ⟨filbuf() if no more input files in the stack 55b⟩

    // system call! fill the buffer
```



(a) Empty buffer in iostack->b



(b) After the first call to GETC(). filbuf() returned 'l',
incremented fi.p, and decremented fi.c

Figure 5.1: Evolution of the input buffer after first call to `GETC()`.

```
    fi.c = read(i->f, i->b, BUFSIZ) - 1;
    ⟨filbuf() if no more character to read 54a⟩
    // else
    fi.p = i->b + 1;
    return i->b[0];

⟨filbuf() pop 54b⟩
}
```

Uses `BUFSIZ` 53b, `fi` 53c, and `iostack` 51c.

Figure 5.1 represents the evolution of the state of `fi` while the input buffer gets filled during a call to `GETC()`.

The `-1` applied after the call to `read()` above is needed because `Fi.c`[53d] represents the remaining characters to read. Thus, if `read()` returns only 1, meaning only one character was read, then this character will be returned by `filbuf()`[53g] (with the code `return i->b[0]`), so there is nothing else to read after. In that case, `Fi.c` should be 0, hence the `-1`.

When there is nothing more to read in the file, `5a` needs to "pop" the file from `iostack` and updates `fi` to point to the input buffer of the includer file:

⟨`filbuf()` *if no more character to read* 54a⟩≡ (53g)

```
if(fi.c < 0) {
    close(i->f);
    ⟨filbuf() when close file, call linehist 116b⟩
    goto pop;
}
```

Uses `fi` 53c.

⟨`filbuf()` *pop* 54b⟩≡ (53g)

```
pop:
    // pop(iostack)
```

```
    iostack = i->link;
    // push(i, iofree)
    i->link = iofree;
    iofree = i;

    // i = top(iostack), the fresh top of the stack input file
    i = iostack;
    ⟨filbuf() if no more input files in the stack 55b⟩
    // restore file pointers
    fi.p = i->p;
    fi.c = i->c;
    if(--fi.c < 0)
        goto loop;
    // else, return one character
    return *fi.p++;
```

Uses `fi` 53c, `iofree` 52a, and `iostack` 51c.

5a uses a few constants to represent special characters. They are negatives to not conflict with regular characters. `EOF` (for "end of file") below is one of those special characters:

⟨*constant* `EOF` 55a⟩≡ (156b)

```
#define EOF     (-1)
```

`filbuf()` (and then `GETC()` and `yylex()`) returns this constant when there is no more character to read:

⟨`filbuf()` *if no more input files in the stack* 55b⟩≡ (54b 53g)

```
if(i == I)
    return EOF;
```

Uses `EOF` 55a and `I` 51e.

`EOF` will be used by `yyparse()`[104c] to avoid calling `yylex()` another time when there is nothing more to read.

Chapter 6

Lexing

The next component in the assembling pipeline is the lexer function `yylex()`[56b], called by the parser function `yyparse()`[104c]. `yylex()` relies on `GETC()`[53f], which I described in Chapter 5, to get the next character in the currently processed input file.

6.1 yylex()

Even though `5a` is not using Lex [LS79], Yacc requires to name the lexer function `yylex()`[56b]. This function does not take any argument (hence the use of many globals), and must return one of the element below:

- A *token code* (see Section 3.5)
- `-1` (represented by EOF), when there is no more token to read
- A single character, if the token is simply a single character

Here is the signature of `yylex()`:

⟨*signature* `yylex` 56a⟩≡ (167a)

```
// unit -> (enum<token_code> | -1 (EOF) | char)
long    yylex(void);
```

`yylex()` can also modify the global `yylval`[104b] to store the *token value* of the token. The type of `yylval` is a union type I described in Section 3.5. This token value can then be retrieved via the $\$n$ notation in the Yacc actions, as you will see in Chapter 8.

`yylex()` is essentially an *automata* reading characters in a loop while using a big `switch` on the current character read to transition states until a full token is formed. To read characters, `yylex()` uses `GETC()`[53f], to loop it uses the label `l0:` below and some `goto l0` statements, and to transition state it uses more labels and `goto`s as you will see soon. Here is finally the skeleton of `yylex()`:

⟨*function* `yylex` 56b⟩≡ (168a)

```
/// main -> assemble -> yyparse -> <>
long
yylex(void)
{
    int c;
    ⟨yylex() locals 58a⟩

    ⟨yylex() peekc handling, starting part 57d⟩
l0:
    c = GETC();
l1:
    if(c == EOF) {
```

```
            return EOF;
        }

        if(isspace(c)) {
            ⟨yylex() if c is newline 58c⟩
            // ignore spaces
            goto l0;
        }
        // else

        ⟨yylex() before switch, if isxxx 59b⟩
        switch(c) {
    //XxX: missing?
    //    case '\n':
    //        lineno++;
    //        return ';';
        ⟨yylex() switch c cases 58d⟩
        default:
            return c;
        }
        ⟨yylex() peekc handling, ending part 58b⟩
        return c;
    }
```
Uses EOF 55a and GETC 53f.

In the next sections, I will show the code of the different cases in the `switch` above.

6.2 Peek, seek, and look ahead

When trying to identify a token, for instance, an identifier, it is frequent to read characters until the current character does not satisfy a certain criteria, for instance, being a letter. In that case, you have read too far in the input file and should go back one character. One technique is to use `seek()` to go back. Another technique is to store instead in a global this extra character. This is the strategy used by `5a` with the global `peekc`:

⟨*global* `peekc` 57a⟩≡ (160a)
```
    // option<char> (None = IGN)
    int peekc = IGN;
```
Uses IGN 57b and peekc 57a.

⟨*constant* `IGN` 57b⟩≡ (156b)
```
    #define IGN     (-2) // Ignore
```

⟨`pinit()` *initializations* 57c⟩≡ (50a) 92b ▷
```
    peekc = IGN;
```
Uses IGN 57b and peekc 57a.

Then, we need to make sure the function reading characters from the input file (`yylex()`[56b]) first looks in this global before trying to read any new character:

⟨`yylex()` *peekc handling, starting part* 57d⟩≡ (56b)
```
    c = peekc;
    if(c != IGN) {
        peekc = IGN; // consume the extra character saved in peekc
        goto l1; // skip the GETC(), we already have a character in c
    }
```
Uses IGN 57b and peekc 57a.

In the sections below, the local `c1` will be used to store the *look ahead* character when the lexer needs to read an extra character in addition to `c` to decide what to do:

⟨`yylex()` *locals* 58a⟩≡ (56b) 59a ▷
```
int c1;
```

This extra character is considered by default the peek character for the next call to `yylex()` (unless `yylex()` returns before):

⟨`yylex()` *peekc handling, ending part* 58b⟩≡ (56b)
```
peekc = c1;
```
Uses `peekc` 57a.

6.3 Newlines (and semicolons)

As I mentioned in Section 2.5.1, `5a` transforms internally newlines in semicolons. This transformation is done in `yylex()`[56b]:

⟨`yylex()` *if c is newline* 58c⟩≡ (56b)
```
if(c == '\n') {
    lineno++;
    return ';'; // newline transformed in fake ';'
}
```
Uses `lineno` 50b.

The grammar of Asm5 then uses those semicolons as instruction terminators (see Section 8.1).

Note that because the semicolon token is a single character, it can be returned directly by `yylex()`. There is no need to introduce a `LSEMICOLON`[35] token code.

As we will see in Section 8.1, the grammar of Asm5 allows empty instructions, so empty lines with just a newline, or lines with only comments, will not pose any problem.

6.4 Comments

Asm5 comments use the same syntax than C comments (e.g., `/* foo */`, `// foo`). They are skipped by the lexer, hence the `goto` back to `l0` (or `l1`) below:

⟨`yylex()` *switch c cases* 58d⟩≡ (56b) 59c ▷
```
case '/':
    c1 = GETC();
    if(c1 == '/') {
        // '/''/' read; skip everything until next '\n'
        for(;;) {
            c = GETC();
            if(c == '\n')
                goto l1; // which will convert the \n in c in a ';'
            if(c == EOF) {
                yyerror("eof in comment");
                errorexit();
            }
        }
    }
    if(c1 == '*') {
        // '/''*' read; skip everything until next '*''/'
        for(;;) {
            c = GETC();
            while(c == '*') { // not an if! to handle /** not finished */
                c = GETC();
```

```
            if(c == '/')
                goto l0;
        }
        if(c == EOF) {
            yyerror("eof in comment");
            errorexit();
        }
        if(c == '\n')
            lineno++;
    }
}
break;
```

Uses EOF 55a, GETC 53f, errorexit() 148a, lineno 50b, and yyerror() 148c.

6.5 Mnemonics, symbols, and labels

Until now, `yylex()`[56b] was mostly skipping characters (spaces and comments), or returning a single character (a semicolon). For identifiers, `yylex()` needs to accumulate a set of characters and look if those characters correspond to a know mnemonic, symbol, or label. The local `cp` below will point in the `symb`[40a] global buffer I introduced before. Remember that the global `symb` is used by `lookup()`[40c] to look for predefined identifiers, labels, or symbols, in the global symbol table `hash`[39a].

⟨yylex() *locals* 59a⟩+≡ (56b) ◁58a 61a▷

```
// ref<char> (target = symb)
char *cp;
// ref<Symbol> (owner = hash)
Sym *s;
```

Here is the code that uses `cp`, `symb`, and the local variable `s` to recognize an identifier:

⟨yylex() *before switch, if isxxx* 59b⟩≡ (56b) 60b▷

```
if(isalpha(c))
    goto talph;
```

⟨yylex() *switch c cases* 59c⟩+≡ (56b) ◁58d 60c▷

```
case '_':
case '@':
// case 'a'..'z' 'A'..'Z': (isalpha())
// XxX: case '.' too
talph:
    cp = symb;

aloop:
    *cp++ = c;
    c = GETC();
    if(isalpha(c) || isdigit(c) || c == '_' || c == '$')
        goto aloop;
    // went too far
    peekc = c;

    *cp = '\0';
    s = lookup(); // uses symb global (referenced by cp)
    ⟨yylex() if macro symbol 136c⟩
    //XxX?
    //if(s->type == 0)
    //    s->type = LNAME;

    ⟨yylex() in identifier case, set yylval 60a⟩
```

```
    return s->type;
```
Uses GETC 53f, lookup() 40c, peekc 57a, and symb 40a.

Remember that yylex() must return to yyparse()[104c] a token code when the token is not a single character. Sym.typeX, used in the expression s->type above, contains the token code of the recognized symbol (e.g., LARITH[35] if the identifier was an arithmetic opcode like ADD).

The global yylval[104b] must then contain the token value. For labels and symbols, the token value is a pointer to a Sym[156b]; for predefined identifiers, it is the value of an enumeration (e.g., AADDX).

⟨yylex() *in identifier case, set yylval* 60a⟩≡ (59c)
```
  if(s->type == LNAME || s->type == LLAB || s->type == LVAR) {
      yylval.sym = s;
  } else {
      yylval.lval = s->value;
  }
```
Uses LLAB, LNAME, LVAR, and yylval.

6.6 Numbers

Numbers are a bit more complicated to parse than what we have seen until now. Indeed, Asm5 allows to use different kinds of numbers with different syntax. Just like in C, in Asm5 you can use decimals (12), hexadecimals (0xa12), octals (0777), or floats (0.2, 1e3) in your assembly program. The code below use different labels and gotos to transition to different states where each state recognizes a different kind of number:

⟨yylex() *before switch, if isxxx* 60b⟩+≡ (56b) ◁59b
```
  if(isdigit(c))
      goto tnum;
```

⟨yylex() *switch c cases* 60c⟩+≡ (56b) ◁59c 63a▷
```
  // case '0'..'9': (isdigit())
  tnum:
      cp = symb;
      if(c != '0')
          goto dc;
      // else, read a '0', maybe the start of an hexadecimal number
      ⟨yylex() in number case, 0xxx handling 61b⟩

  ⟨yylex() in number case, decimal dc label handling 60d⟩
  ⟨yylex() in number case, float labels handling 62c⟩
```
Uses symb 40a.

Note that the code above also (ab)uses cp and symb[40a] to accumulate characters (but this time there will be no call to lookup()[40c]).

I will explain the chunks referenced in the code above in the following sections.

6.6.1 Decimal numbers

The first label (and state) used to recognize numbers is dc (for "decimal"). yylex()[56b] jumps to this label when the first digit is not zero. yylex() then reads more digits until the next character is not a digit anymore:

⟨yylex() *in number case, decimal dc label handling* 60d⟩≡ (60c)
```
  dc:
      for(;;) {
          if(!isdigit(c))
              break;
          *cp++ = c;
          c = GETC();
```

```
    }
    ⟨yylex() in number case, in decimal case, float handling 62b⟩
    *cp = '\0';
    yylval.lval = strtol(symb, nil, 10);

    peekc = c;
    return LCONST;
```
Uses GETC 53f, LCONST, peekc 57a, symb 40a, and yylval.

Most of the magic to convert a set of accumulated (digit) characters to an integer is done by the `strtol()` function from the core C library (see the LIBCORE book [Pad16c]).

6.6.2 Hexadecimal and octal numbers

If a number starts with a zero, it is maybe the start of an hexadecimal or octal number. In that case, similar to the code to handle decimals, `yylex()`[56b] accumulates also characters in `symb`[40a] (via `cp`). However, this time `yylex()` does not rely on `strtol()` to convert the characters in an integer[1]. Instead, `yylex()` gradually updates `yylval.lval`:

⟨yylex() *locals* 61a⟩+≡ (56b) ◁59a 67e▷
```
int baselog2;
```

⟨yylex() *in number case, 0xxx handling* 61b⟩≡ (60c)
```
*cp++ = c;
c = GETC();
baselog2 = 3; // 2^3, for octal
if(c == 'x' || c == 'X') {
    baselog2 = 4; // 2^4, for hexadecimal
    c = GETC();
}
else if(c < '0' || c > '7')
    goto dc;

yylval.lval = 0;
for(;;) {
    if(c >= '0' && c <= '9') {
        if(c > '7' && baselog2 == 3)
            break;
        yylval.lval <<= baselog2;
        yylval.lval += c - '0';
        c = GETC();
        continue;
    }
    // else
    if(baselog2 == 3)
        break;
    // else
    ⟨yylex() in number case, 0xxx handling, normalize letters 62a⟩
    if(c >= 'a' && c <= 'f') {
        yylval.lval <<= baselog2;
        yylval.lval += c - 'a' + 10;
        c = GETC();
        continue;
    }
    break;
}
//XxX: goto ncu;
```

[1]It could because `strtol()` handles those cases.

```
  peekc = c;
  return LCONST;
```

Uses GETC 53f, LCONST, peekc 57a, and yylval.

⟨yylex() *in number case, 0xxx handling, normalize letters* 62a⟩≡ (61b)

```
  if(c >= 'A' && c <= 'F')
      // c = lowercase(c)
      c += 'a' - 'A';
```

6.6.3 Floating-point numbers

yylex()[56b] uses a set of labels to recognize floating-point numbers:

⟨yylex() *in number case, in decimal case, float handling* 62b⟩≡ (60d)

```
  if(c == '.')
      goto casedot;
  if(c == 'e' || c == 'E')
      goto casee;
   //XxX:         *cp = 0;
   //XxX:         if(sizeof(yylval.lval) == sizeof(vlong))
   //XxX:             yylval.lval = strtoll(symb, nil, 10);
   //XxX:         else
   //XxX:             yylval.lval = strtol(symb, nil, 10);
   //XxX:
   //XxX:     ncu:
   //XxX:         while(c == 'U' || c == 'u' || c == 'l' || c == 'L')
   //XxX:             c = GETC();
```

⟨yylex() *in number case, float labels handling* 62c⟩≡ (60c)

```
  casedot:
      for(;;) {
          *cp++ = c;
          c = GETC();
          if(!isdigit(c))
              break;
      }
      if(c == 'e' || c == 'E')
          goto casee;
      goto caseout;

  casee:
      *cp++ = 'e';
      c = GETC();
      if(c == '+' || c == '-') {
          *cp++ = c;
          c = GETC();
      }
      while(isdigit(c)) {
          *cp++ = c;
          c = GETC();
      }

  caseout:
      *cp = '\0';
      peekc = c;
      if(FPCHIP) {
          yylval.dval = atof(symb);
          return LFCONST;
```

```
    } else {
        yyerror("assembler cannot interpret fp constants");
        yylval.lval = 1L;
        return LCONST;
    }
```

Uses FPCHIP 123a, GETC 53f, LCONST, LFCONST, peekc 57a, symb 40a, yyerror() 148c, and yylval.

Most of the magic to convert a set of characters in a floating-point number is done by the `atof()` function from the core C library (see the LIBCORE book [Pad16c]).

In Asm5, a dot without any number before can be the start of a float, the start of an identifier, or simply the dot character, hence the code below:

⟨yylex() *switch c cases* 63a⟩+≡ (56b) ◁60c 63b▷

```
  case '.':
    c = GETC();
    if(isalpha(c)) { // an identifier
        cp = symb;
        *cp++ = '.';
        goto aloop;
    }
    if(isdigit(c)) { // a float
        cp = symb;
        *cp++ = '.';
        goto casedot;
    }
    // else
    peekc = c;
    return '.'; // a single '.'
```

Uses GETC 53f, peekc 57a, and symb 40a.

6.7 Characters

In Asm5, like in C, characters start and end with a single quote character (e.g., `'a'`). However, because the quote character is used to mark the end of a character, how do you represent the quote character itself? Moreover, how do you represent ASCII characters that can not be put between quotes such as the backspace character?

6.7.1 Escaped sequences

Asm5, like C, allows to use escape sequences to represent special characters. An *escape sequence* starts with the antislash character and is followed by a character or a series of numbers. For example, `'\n'` represents the newline character, `'\007'` the character with the ASCII value 7, and `'\''` the quote character itself.

The function `escchar()`[65b] used below is a wrapper around `GETC()`[53f]. Like `GETC()`, `escchar()` reads a character from the input file and returns it unless (1) the character read is the start of an escape sequence, or (2) the quote character itself. In the first case, `escchar()` reads more characters from the input file and returns the value of the escaped character. In the second case, it returns `EOF`[55a] to indicate there was no character between the two quotes. Figure 6.1 illustrates a few examples of use of `escchar()`.

Here is finally the code to recognize characters in Asm5:

⟨yylex() *switch c cases* 63b⟩+≡ (56b) ◁63a 68▷

```
  case '\'':
    c = escchar('\'');
    ⟨yylex() in character case, if c is EOF 65a⟩
    if(escchar('\'') != EOF)
        yyerror("missing '");
```

```
             cursor
               |
 input  +-------v---------+     before
buffer  |. . . ' a ' . . .|    escchar()
        +-----------------+
               cursor
                 |
 input  +---------v-------+       after
buffer  |. . . ' a ' . . .|   escchar('\'')
        +-----------------+

    return value: 97 (ASCII code of 'a')


             cursor
               |
 input  +-------v---------+     before
buffer  |. . . ' \ n ' . .|    escchar()
        +-----------------+
                 cursor
                   |
 input  +-----------v-----+       after
buffer  |. . . ' \ n ' . .|   escchar('\'')
        +-----------------+

    return value: 10 (ASCII code of '\n')


             cursor
                |
 input  +---------v-------+     before
buffer  |. . . ' a ' . . .|    escchar()
        +-----------------+
                cursor
                  |
 input  +-----------v-----+       after
buffer  |. . . ' a ' . . .|   escchar('\'')
        +-----------------+

        return value: EOF
```

Figure 6.1: Examples of use of `escchar()`.

```
    yylval.lval = c;
    return LCONST;
```
Uses `EOF` 55a, `LCONST`, `escchar()` 65b, `yyerror()` 148c, and `yylval`.

As I said before, `5a` converts characters in integers, hence the use of the `LCONST`[35] token code above used also for numbers (see Section 6.6.1).

There are two calls to `escchar()` above. The first consumes the character after the first quote (e.g., `a` in `'a'`, as shown at the top of Figure 6.1) The second consumes the ending quote itself (as illustrated at the bottom of Figure 6.1). Indeed, `escchar()` returns the special code `EOF` if the character read is the same than the character passed as an argument (here a single quote).

6.7.2 Triple quotes

Note that the C code above uses itself the escape sequence `'\''` to represent the quote character. Asm5 uses another technique to represent the quote character: the programmer can *double* the quote character to represent a single quote character. Indeed, `5a` interprets the sequence of characters `'''` as the single quote. Here is the code to handle `'''`:

⟨`yylex()` *in character case, if c is EOF* 65a⟩≡ (63b)
```
  if(c == EOF)
      c = '\'';
```
Uses `EOF` 55a.

`5a` uses this same doubling technique to represent the escaping character itself; the sequence `'\\'` represents the antislash character.

6.7.3 `escchar()`

The argument to `escchar()` is a character used to mark the end of an "entity". In this section, the mark is a single quote (`'`) because we are parsing characters. In the next section, the argument to `escchar()` will be the double quote character (`"`) because we will parse strings.

Here is finally the code of `escchar()`:

⟨*function* `escchar` 65b⟩≡ (161c)
```
  int
  escchar(int e)
  {
      int c;
      ⟨escchar() other locals 66c⟩

  loop:
      c = getc();
      ⟨escchar() sanity check if newline 66e⟩
      if(c != '\\') {
          if(c == e)
              return EOF;
          return c;
      }
      // else c is '\\'
      c = getc();
      ⟨escchar() if octal character 66d⟩
      switch(c)
      {
      ⟨escchar() switch cases 66a⟩
      }
      // else (e.g., '\'', '\\')
      return c;
```

```
}
```

Uses EOF 55a and getc() 67c.

⟨escchar() *switch cases* 66a⟩≡ (65b) 66b ▷

```
case 'n': return '\n';
case 't': return '\t';
case 'b': return '\b';
case 'r': return '\r';
case 'f': return '\f';
```

Asm5 allows a few more escape sequences compared to C:

⟨escchar() *switch cases* 66b⟩+≡ (65b) ◁ 66a 67b ▷

```
case 'a': return 0x07;
case 'v': return 0x0b;
case 'z': return 0x00;
```

In Asm5, a series of numbers after an escape character is interpreted as a series of octal numbers. Here is the code to handle those escaped sequences (e.g., `'\007'`):

⟨escchar() *other locals* 66c⟩≡ (65b)

```
int l;
```

⟨escchar() *if octal character* 66d⟩≡ (65b)

```
if(c >= '0' && c <= '7') {
    l = c - '0';
    c = getc();
    if(c >= '0' && c <= '7') {
        l = l*8 + c-'0';
        c = getc();
        if(c >= '0' && c <= '7') {
            l = l*8 + c-'0';
            return l;
        }
    }
    peekc = c;
    return l;
}
```

Uses getc() 67c and peekc 57a.

6.7.4 Escaped newlines

5a forbids the use of newlines in the middle of a character or string:

⟨escchar() *sanity check if newline* 66e⟩≡ (65b)

```
if(c == '\n') {
    yyerror("newline in character or string");
    return EOF;
}
```

Uses EOF 55a and yyerror() 148c.

To represent a character or string containing a newline, use the escape sequence `'\n'` as in `"hello world\n"`. 5a could allow to use directly a newline in a character or string as in:

⟨*not valid Asm5 example* 66f⟩≡

```
  MOVW $'
', R1
```

However, this would destroy the indentation of the program, which is why `'\n'` is a more convenient way for the programmer to represent a newline character.

In some cases, it is useful to split a long string on multiple lines. Asm5, like C, allows to *escape a newline* as in the following code:

⟨*example of escaped newline* 67a⟩≡
```
DATA foo(SB), "this is\
a long string"
```

Here is the code to handle escaped newlines:

⟨`escchar()` *switch cases* 67b⟩+≡ (65b) ◁66b
```
case '\n': goto loop; // multi line strings
```

Note that the escaped character and the following newline are skipped by `escchar()`[65b].

Escaped newlines are not really useful in Asm5 because Asm5 limits strings to less than 8 characters. They are supported by `5a` because the function `escchar()` was reused from the code of the C compiler `5c`.

6.7.5 `getc()`

`getc()`, called from `escchar()`[65b], is a small wrapper around `GETC()`[53f] that takes care of `lineno`[50b]:

⟨*function* `getc` 67c⟩≡ (161c)
```
int
getc(void)
{
    int c;

    ⟨getc() peekc handling 67d⟩
    c = GETC();

    if(c == '\n')
        lineno++;
    if(c == EOF) {
        yyerror("End of file");
        errorexit();
    }
    return c;
}
```
Uses `EOF` 55a, `GETC` 53f, `errorexit()` 148a, `lineno` 50b, and `yyerror()` 148c.

⟨`getc()` *peekc handling* 67d⟩≡ (67c)
```
c = peekc;
if(c != IGN) {
    peekc = IGN;
    return c;
}
```
Uses `IGN` 57b and `peekc` 57a.

6.8 Strings

Asm5 supports the use of strings, for instance, in `DATA` pseudo instructions. The code below relies again on the function `escchar()`[65b] to handle escaped sequences in strings.

⟨`yylex()` *locals* 67e⟩+≡ (56b) ◁61a
```
int i;
```

⟨yylex() *switch c cases* 68⟩+≡ (56b) ◁63b 69a▷

```
  case '"':
      memcpy(yylval.sval, nullgen.sval, sizeof(yylval.sval));
      cp = yylval.sval;
      i = 0;
      for(;;) {
          c = escchar('"');
          if(c == EOF)
              break;
          if(i < sizeof(yylval.sval))
              *cp++ = c;
          i++;
      }
      if(i > sizeof(yylval.sval))
          yyerror("string constant too long");
      return LSCONST;
```

Uses `EOF` 55a, `LSCONST`, `escchar()` 65b, `nullgen` 34a, `yyerror()` 148c, and `yylval`.

Chapter 7

Preprocessing

The next component in the assembling pipeline is the preprocessor. Indeed, as I mentioned in Section 2.5.6, `5a` is a macro-assembler.

`5a`, like the C compiler `5c`, does not rely on an external program (e.g., `/bin/cpp`) to preprocess code. Instead, `5a` *embeds* in its own code a macroprocessor. By doing so, `5a` avoids to fork and to communicate with an external program, which would slow down the assembling process.

In this chapter, I will not present all the code of this embedded macroprocessor. Indeed, the macroprocessor of `5a` uses the same syntax than the C preprocessor, so most of this code is identical to the code of the C preprocessor I present in the COMPILER book [Pad16b]. However, I will present the code to handle the `#include` and `#line` directives because those directives are either strongly related to code I presented before (`iostack`[51c] in Section 5.2 for `#include`), or related to important code I will present later (debugging support in Section 10.1 for `#line`). For the code to handle `#define`, `#ifdef`, `#undef`, and `#pragma`, see Section 11.5 in the Advanced Topics chapter.

7.1 Directives dispatch: `mactab`, and `domacro()`

One of the two entry points in the macroprocessor embedded in `5a` is the call to `domacro()`[70a] in `yylex()`[56b] below to handle all preprocessing directives[1]:

⟨`yylex()` *switch c cases* 69a⟩+≡ (56b) ◁68

```
case '#':
    domacro();
    goto l0;
```

Uses `domacro()` 70a.

`domacro()` will read all the characters on the line and modify some globals (e.g., `iostack`[51c] and `fi`[53c] when processing a `#include`). Note that after the call to `domacro()` above, `yylex()` jumps back to `l0`. Indeed, a preprocessing directive is not a meaningful token for the parser. Once a directive line (e.g., a `#include`) has been preprocessed, `yylex()` must go back and read more characters (this time from the included file).

As you will see soon, `domacro()` relies internally on the structure and global below. `mactab` below maps a directive string to a *callback* responsible for parsing and processing the directive.

⟨*global* `mactab` 69b⟩≡ (166)

```
struct
{
    char    *macname;
    void    (*macf)(void);
} mactab[] =
{
    "ifdef",    nil,    /* macif(0) */
```

[1] The other entry point is the call to `macexpand()`[137] to expand macro when `yylex()` recognizes an identifier (see Section 11.5.1).

```
    "ifndef",   nil,    /* macif(1) */
    "else",     nil,    /* macif(2) */
    "endif",    macend,

    "include",  macinc,
    "line",     maclin,
    "define",   macdef,
    "undef",    macund,

    "pragma",   macprag,
    0
};
```

Uses __anon_struct_1 69b, macdef() 133b, macend() 72b, macinc() 74a, maclin() 78a, macprag() 141b, and macund() 139.

In this chapter, I will only describe the code of `macinc()`[74a] and `maclin()`[78a]. For the other callbacks, see Section 11.5.

Here is finally the code of `domacro()`:

⟨*function* domacro 70a⟩≡ (166)

```
/// main -> assemble -> yyparse -> yylex -> <>
void
domacro(void)
{
    int i;
    Sym *s;

    s = getsym();
    ⟨domacro() set s to endif symbol if no symbol 141a⟩

    for(i=0; mactab[i].macname; i++)
        if(strcmp(s->name, mactab[i].macname) == 0) {
            // dispatch!
            if(mactab[i].macf)
                (*mactab[i].macf)();
            else
                macif(i);
            return;
        }
    // else
    ⟨domacro() handle unknown directives 72a⟩
}
```

Uses getsym() 70b, macif() 140, and mactab 69b.

`domacro()` reads the name of the directive with `getsym()` and then iterates over `mactab` to dispatch the appropriate callback.

The code of `getsym()` below is similar to the code in `yylex()` to recognize identifiers (see Section 6.5). The code also accumulates characters in the global `symb`[40a] and calls `lookup()`[40c].

⟨*function* getsym 70b⟩≡ (166)

```
Sym*
getsym(void)
{
    int c;
    char *cp;

    c = getnsc();
    if(!isalpha(c) && c != '_' && c < Runeself) {
        unget(c);
        return S;
    }
```

```
    for(cp = symb;;) {
        if(cp <= symb+NSYMB-4)
            *cp++ = c;
        c = getc();
        if(isalnum(c) || c == '_' || c >= Runeself)
            continue;
        // else
        unget(c);
        break;
    }
    *cp = '\0';
    ⟨getsym() sanity check cp 71a⟩
    return lookup();
}
```

Uses `NSYMB` 40b, `S` 39d, `getc()` 67c, `getnsc()` 71b, `lookup()` 40c, `symb` 40a, and `unget()` 71d.

⟨`getsym()` *sanity check cp* 71a⟩≡ (70b)

```
if(cp > symb+NSYMB-4)
    yyerror("symbol too large: %s", symb);
```

Uses `NSYMB` 40b, `symb` 40a, and `yyerror()` 148c.

`getsym()`[70b] calls first `getnsc()` (for "get first non-space character") below:

⟨*function* `getnsc` 71b⟩≡ (161c)

```
int
getnsc(void)
{
    int c;

    for(;;) {
        c = getc();
        if(!isspace(c) || c == '\n')
            return c;
    }
}
```

Uses `getc()` 67c.

Indeed, macroprocessor directives can contain spaces between the # and the identifier, as in the following example:

⟨*example of use of ifdef* 71c⟩≡

```
#ifdef FOO
#   ifdef BAR
...
#   endif //BAR
#endif //FOO
```

`getsym()` must read an extra character to find the end of an identifier. Doing so, it goes too far in the input buffer. Similar to code in `yylex()`, the function `unget()` below, called from `getsym()` above, allows to go back in the input buffer by storing the extra character in the global `peekc`[57a].

⟨*function* `unget` 71d⟩≡ (161c)

```
void
unget(int c)
{
    peekc = c;
    if(c == '\n')
        lineno--;
}
```

Uses `lineno` 50b and `peekc` 57a.

If the symbol read by `getsym()` does not correspond to a known directive, `5a` reports an error and skips the line:

⟨`domacro()` *handle unknown directives* 72a⟩≡ (70a)
```
yyerror("unknown #: %s", s->name);
macend();
```
Uses `macend()` 72b and `yyerror()` 148c.

⟨*function* `macend` 72b⟩≡ (166)
```
void
macend(void)
{
    int c;

    for(;;) {
        c = getnsc();
        if(c < 0 || c == '\n')
            return;
    }
}
```
Uses `getnsc()` 71b.

7.2 #include

I can now describe the first callback `macinc()`[74a], which handles `#include` directives.

7.2.1 Include search path

Before showing the code of `macinc()`[74a], here are a few globals used by `macinc()`:

⟨*global* `include` 72c⟩≡ (160a)
```
// array<option<string>> (size = ninclude)
char* include[NINCLUDE];
```
Uses `NINCLUDE` 72d.

⟨*constant* `NINCLUDE` 72d⟩≡ (156b)
```
#define NINCLUDE    10
```

⟨*global* `ninclude` 72e⟩≡ (160a)
```
int ninclude;
```

`include` above contains a set of directories to search for header files, for instance, `["."; "/usr/include"; "/arm/include"]`. This set is called the *include search path*. Indeed, in Asm5, like in C, you do not have to specify the full path of a header file to include it. You can just write in your program `#include <u.h>` and `5a` will try to find this header file automatically in one of the directory in the include search path.

By convention, the first entry in `include` corresponds to the directory containing the input assembly file passed on the command-line to `5a`. Thus, if you run the command `5a /tests/foo.s` from any directory, the `foo.s` program can still include the header file `/tests/foo.h` by just using `#include "foo.h"`.

`main()`[155a] sets first a default value for this first entry:

⟨`main()` *remaining initializations* 72f⟩≡ (43c) 146b ▷
```
include[ninclude++] = ".";
```

This value is then refined in `assemble()`[45d] during the processing of the input filename (see Figure 4.1):

⟨`assemble()` *adjust first entry in include with dirname infile* 72g⟩≡ (47e)
```
include[0] = ofile;
```
Uses `include` 72c.

7.2.2 -I

You can also add directories in the include search path by using the command-line flag `-I` as in `5a -I /arm/include -I /sys/include foo.s`. Here is the code to manage those flags:

⟨`main()` *locals* 73a⟩≡ (43c) 142a▷
```
char *p;
```

⟨`main()` *command line processing* 73b⟩+≡ (43c) ◁44b 132a▷
```
case 'I':
    p = ARGF();
    setinclude(p);
    break;
```

⟨*function* `setinclude` 73c⟩≡ (161c)
```
void
setinclude(char *p)
{
    int i;

    ⟨setinclude() sanity check p 73d⟩
    ⟨setinclude() check if include already added 73e⟩
    ⟨setinclude() check if too many entries in include 73f⟩
    // else
    include[ninclude++] = p;
}
```
Uses `include` 72c and `ninclude` 72e.

⟨`setinclude()` *sanity check p* 73d⟩≡ (73c)
```
if(p == nil)
    return;
```

⟨`setinclude()` *check if include already added* 73e⟩≡ (73c)
```
for(i=1; i < ninclude; i++)
    if(strcmp(p, include[i]) == 0)
        return;
```
Uses `include` 72c and `ninclude` 72e.

⟨`setinclude()` *check if too many entries in include* 73f⟩≡ (73c)
```
if(ninclude >= nelem(include)) {
    yyerror("ninclude too small %d", nelem(include));
    exits("ninclude");
}
```
Uses `include` 72c, `ninclude` 72e, and `yyerror()` 148c.

7.2.3 System paths

Finally, `5a` automatically adds in the include search path directories contained in the `INCLUDE` environment variable if this variable is set, or the architecture-specific directory `/arm/include` otherwise:

⟨`assemble()` *locals* 73g⟩+≡ (45d) ◁47d 132d▷
```
char incfile[20];
```

⟨`assemble()` *setinclude("/thestring/include") or use INCLUDE* 73h⟩≡ (45d)
```
p = getenv("INCLUDE");
if(p) {
    setinclude(p);
} else {
     sprint(incfile,"/%s/include", thestring);
     setinclude(strdup(incfile));
}
```
Uses `setinclude()` 73c and `thestring` 43b.

7.2.4 macinc()

Here is finally the code of `macinc()`:

⟨*function* `macinc` 74a⟩≡ (166)

```
void
macinc(void)
{
    char *hp;
    int n;
    fdt f = -1;
    ⟨macinc() other locals 75c⟩

    // lexing

    ⟨macinc() lexing the included filename 75e⟩

    // action

    ⟨macinc() find and store the included filename full path in symb 76a⟩

    n = strlen(symb) + 1;

    while(n & 3)
        n++;
    hp = malloc(n);
    memcpy(hp, symb, n);

    newio();
    pushio();
    newfile(hp, f);

    return;

⟨macinc() bad 75f⟩
}
```

Uses `newfile()` 52d, `newio()` 51g, `pushio()` 74b, and `symb` 40a.

I will describe later the code to lex the `#include` line to extract the included filename, as well as the code to find the full path of this included file in the include search path. But before, note the call to `newio()`[51g] and `newfile()`[52d] above. I introduced both functions in Chapter 5. Those functions modify the globals `iostack`[51c] and `fi`[53c]. Thus, when `yylex()`[56b] will `goto` back to `l0` (after having processed the `#include` line via `domacro()`[70a] and `macinc()`[74a]), further calls to `GETC()`[53f] will operate on an empty buffer `fi`. This will trigger a call to `filbuf()`[53g], which will use the new `iostack` pointing now to a new file.

Before the call to `newfile()`, `macinc()` calls `pushio()` below to save the current value in `fi`. That way, when `5a` will finish to process the included file, it will be able to go back to where it was in the includer file.

⟨*function* `pushio` 74b⟩≡ (161c)

```
void
pushio(void)
{
    Io *i;

    i = iostack;
    ⟨pushio() sanity check i 75a⟩
    // save current position in includer
    i->p = fi.p;
    i->c = fi.c;
}
```

Uses `fi` 53c and `iostack` 51c.

⟨pushio() *sanity check i* 75a⟩≡ (74b)
```
if(i == I) {
    yyerror("botch in pushio");
    errorexit();
}
```
Uses I 51e, errorexit() 148a, and yyerror() 148c.

The code to parse an #include must handle two forms of #include as shown in the example below:

⟨*example of use of include* 75b⟩≡
```
#include <system_file.h>
#include "other_file.h"
```

⟨macinc() *other locals* 75c⟩≡ (74a) 75g ▷
```
char str[STRINGSZ];
int c, cend;
```
Uses STRINGSZ 75d.

⟨*constant* STRINGSZ 75d⟩≡ (156b)
```
#define STRINGSZ    200
```

⟨macinc() *lexing the included filename* 75e⟩≡ (74a)
```
cend = getnsc();
if(cend != '"') {
    if(cend != '<') {
        c = cend;
        goto bad;
    }
    cend = '>';
}
// cend = '"' or '>'
hp = str;
for(;;) {
    c = getc();
    if(c == cend)
        break;
    if(c == '\n')
        goto bad;
    *hp++ = c;
}
*hp = '\0';

⟨macinc() finish parsing the line 76d⟩
```
Uses getc() 67c and getnsc() 71b.

⟨macinc() *bad* 75f⟩≡ (74a)
```
bad:
    unget(c);
    yyerror("syntax in #include");
    macend();
```
Uses macend() 72b, unget() 71d, and yyerror() 148c.

Once macinc() filled its local variable str with the characters of the included filename (through the hp pointer), macinc() tries to find the full path of the included filename by iterating over the include search path:

⟨macinc() *other locals* 75g⟩+≡ (74a) ◁ 75c
```
int i;
```

⟨macinc() *find and store the included filename full path in* symb 76a⟩≡ (74a)
```
  for(i=0; i<ninclude; i++) {
      ⟨macinc() skipped first entry for system headers 76b⟩
      strcpy(symb, include[i]);
      strcat(symb, "/");
      ⟨macinc() normalize path 76c⟩
      strcat(symb, str);

      f = open(symb, OREAD);
      if(f >= 0)
          break;
      // else, try another directory
  }
  // could not find a directory, maybe it was an absolute path
  if(f < 0)
      strcpy(symb, str);
```
Uses include 72c, ninclude 72e, and symb 40a.

The first entry of `include`[72c], which corresponds by convention to the directory of the input assembly file, is not looked for with system headers:

⟨macinc() *skipped first entry for system headers* 76b⟩≡ (76a)
```
  if(i == 0 && cend == '>') // do not look in '.' for system headers
      continue;
```

Thus, you can use in your own project a header file with the same name than a system header (e.g., `stdlib.h`) and select which file to include by using either `<>` or `""` around the filename.

⟨macinc() *normalize path* 76c⟩≡ (76a)
```
  if(strcmp(symb, "./") == 0)
      symb[0] = '\0';
```
Uses symb 40a.

The macroprocessor allows only spaces or comments after the directive and before the newline:

⟨macinc() *finish parsing the line* 76d⟩≡ (75e)
```
  c = getcom();
  if(c != '\n')
      goto bad;
```
Uses getcom() 76e.

The code of `getcom()` below is similar to the code of `yylex()` to handle comments in Section 6.4, except newlines are not allowed here:

⟨*function* getcom 76e⟩≡ (166)
```
  int
  getcom(void)
  {
      int c;

      for(;;) {
          c = getnsc(); // skip whitespaces

          if(c != '/')
              break;
          c = getc();
          if(c == '/') {
              while(c != '\n')
                  c = getc();
              break;
          }
          if(c != '*')
```

```
            break;
        c = getc();
        for(;;) {
            if(c == '*') {
                c = getc();
                if(c != '/')
                    continue;
                c = getc();
                break;
            }
            if(c == '\n') {
                yyerror("comment across newline");
                break;
            }
            c = getc();
        }
        if(c == '\n')
            break;
    }
    return c;
}
```
Uses `getc()` 67c, `getnsc()` 71b, and `yyerror()` 148c.

7.3 #line

The other (and last) callback I will present in this chapter is `maclin()`[78a], which handles `#line` directives.

7.3.1 Motivations

`#line` directives are extremly useful for debugging purposes, as I will fully explain in Chapter 10. They are usually not written by the programmer but instead generated by programs. For example, if you process an assembly file with the `/bin/cpp` program (instead of relying on the embedded macroprocessor in `5a`), `cpp` will expand all directives (e.g., `#include`, `#ifdef`, `#define`), but it will also generate in the output many `#line` directives. That way, subsequent programs processing this output can know from which file and which line a line in the output comes from, which is useful for precise error reporting and debugging. Here is an example of output of `cpp` on a toy program:

```
$ cd /tests/cpp/
$ cpp foo.s
#line 1 "/tests/cpp/foo.s"



#line 1 "/tests/cpp/./foo.h"
#line 1 "/tests/cpp/./foo1.h"
...
$
```

Because `5a` embeds its own macroprocessor, in theory `5a` does not need to handle `#line` directives; `5a` can keep track internally of the origin of lines when processing `#include` and macros. However, some assembly files may also be generated by programs. For example, a C compiler could generate an assembly file from a C file and add `#line` directives in the generated code so you could see to which line in the C file certain assembly instructions correspond to. Debuggers can also leverage such information as explained in Chapter 10. Thus, it is important for `5a` to also handle the `#line` directive, even if it embeds its own macroprocessor.

Note that most compilers for most programming languages accept in their input a `#line` directive. In fact, it is usually the only directive they support. Indeed, programmers use the C preprocessor with many programming languages (e.g., Haskell, OCaml, LambdaProlog), not just C, because it is a versatile tool. Moreover, most programming languages have in their environment tools similar to Lex and Yacc but specific to the programming language (e.g., `ocamllex` and `ocamlyacc` for the OCaml programming language). Those tools will also generate source files containing `#line` directives so the programming language compiler can report errors correctly. Thus, `#line` directives are extremely useful in many contexts.

7.3.2 `maclin()`

Here is finally the code of `maclin()`:

⟨*function* `maclin` 78a⟩≡ (166)

```
void
maclin(void)
{
    char *cp;
    long n;
    int size;
    ⟨maclin() other locals 78b⟩

    // lexing

    ⟨maclin() lexing the line n and filename in symb 78c⟩

    // action
nn:
    size = strlen(symb) + 1;

    while(size & 3)
        size++;
    cp = malloc(size);

    memcpy(cp, symb, size);

    ⟨maclin() call linehist 116a⟩
    return;

⟨maclin() bad 79c⟩
}
```

Uses `symb` 40a.

`maclin()`[78a], like `newfile()`[52d] (called from `macinc()`[74a] and `pinit()`[50a]), calls the function `linehist()`[115g] above. It is this function that will keep track of the origin of lines as you will see in Chapter 10.

The code to parse a `#line` is similar to the code in `macinc()`. The code also accumulates characters in `symb`[40a] through a pointer `cp`. However, the function `getnsn()`[79e] below recognizes first the integer before the filename.

⟨`maclin()` *other locals* 78b⟩≡ (78a)

```
int c;
```

⟨`maclin()` *lexing the line* `n` *and filename in* `symb` 78c⟩≡ (78a)

```
// the line number

n = getnsn();

// the (optional) filename
```

```
  c = getc();
  if(n < 0)
      goto bad;

  for(;;) {
      ⟨maclin() skipping whitespaces 79b⟩
      if(c == '"')
          break;
      ⟨maclin() if no filename 79a⟩
      // else
      goto bad;
  }
  cp = symb;
  for(;;) {
      c = getc();
      if(c == '"')
          break;
      *cp++ = c;
  }
  *cp = '\0';

  ⟨maclin() finish parsing the line 79d⟩
```

Uses `getc()` 67c, `getnsn()` 79e, and `symb` 40a.

⟨`maclin()` *if no filename* 79a⟩≡ (78c)

```
  if(c == '\n') {
      strcpy(symb, "<noname>");
      goto nn;
  }
```

Uses `symb` 40a.

⟨`maclin()` *skipping whitespaces* 79b⟩≡ (78c)

```
  if(c == ' ' || c == '\t') {
      c = getc();
      continue;
  }
```

Uses `getc()` 67c.

⟨`maclin()` *bad* 79c⟩≡ (78a)

```
  bad:
      unget(c);
      yyerror("syntax in #line");
      macend();
```

Uses `macend()` 72b, `unget()` 71d, and `yyerror()` 148c.

⟨`maclin()` *finish parsing the line* 79d⟩≡ (78c)

```
  c = getcom();
  if(c != '\n')
      goto bad;
```

Uses `getcom()` 76e.

⟨*function* `getnsn` 79e⟩≡ (166)

```
  long
  getnsn(void)
  {
      long n;
      int c;

      c = getnsc();
```

```
    if(c < '0' || c > '9')
        return -1;
    n = 0;
    while(c >= '0' && c <= '9') {
        n = n*10 + c-'0';
        c = getc();
    }
    unget(c);
    return n;
}
```
Uses `getc()` 67c, `getnsc()` 71b, and `unget()` 71d.

Chapter 8

Parsing

In this chapter we present the Yacc *grammar* of Asm5, the ARM assembly language supported by `5a`. Yacc generates from this grammar the `yyparse()` [104c] parsing function which is called by `assemble()` [45d] and which internally calls `yylex()` [56b]. We assume you know how to read a Yacc grammar, otherwise read [BLM92][1].

8.1 Overview

Here is the outline of the Yacc grammar file for Asm5:

⟨*5a/a.y* 81a⟩≡
```
%{
#include "a.h"
%}
⟨union declaration(arm) 37a⟩
⟨priority and associativity declarations 121c⟩
⟨token declarations(arm) 35⟩
⟨type declarations(arm) 92e⟩
%%
⟨grammar(arm) 81b⟩
```

The `%union` and `%token` directives have been already described in Section 3.5. The `%type` declarations will be described gradually in this chapter. The priority declarations are only used for the constant expressions (an advanced Asm9 feature) and will be described in Section 11.1.1.

The grammar part at the end is the most important one. It contains the different parsing *rules* which will be presented gradually in this chapter:

⟨*grammar(arm)* 81b⟩≡ (81a)
```
⟨prog rule(arm) 82a⟩
⟨line rule(arm) 82b⟩
⟨inst rule(arm) 83a⟩
⟨operand rules(arm) 92d⟩
⟨cond rule(arm) 102h⟩

⟨advanced topics rules ??⟩

⟨helper rules(arm) 96b⟩
```

Essentially, an assembly program is made of a set lines containing instructions or labels. Instructions are made of an opcode followed possibly by a few operands. Each operand can be a register, a constant, a symbol, or certain combinations of the previous elements (e.g., a register and constant offset forming an address).

[1] You can also read the original Yacc paper at `generators/docs/yacc.pdf` in our Plan 9 repository, or read our Compiler-Generator book [Pad16a] to fully understand how Yacc works.

More formally, an Asm5 `program` is either an empty file or a set of `lines`:

⟨*prog rule(arm)* 82a⟩≡ (81b)
```
prog:
  /* empty */
| prog line
```

A `line` can be an `inst`[83a]ruction followed by a semicolon (remember that the lexer transforms newlines in semicolons as explained in Section 6.3):

⟨*line rule(arm)* 82b⟩≡ (81b) 82c ▷
```
line:
  inst ';'
```

It can also be empty, or contain the special `error` Yacc token used for error recovery (see [BLM92]):

⟨*line rule(arm)* 82c⟩+≡ (81b) ◁ 82b 91h ▷
```
| ';'
| error ';'
```

In the following section we will focus on the `instruction` rule and see the major opcodes of Asm5. In Section 8.3 we will see that a line can also contain label definitions. Then, in Section 8.4 we will focus on the major operands of Asm5. We will then explore a few advanced features of Asm5. While presenting the syntax of the different features of Asm5, we will also try in this chapter to discuss how those features can be used to implement higher-level constructs of languages such as C. We will conclude this chapter by discussing the generated code for `yyparse()`[104c] by Yacc.

8.2 Instructions

Most instructions of Asm5 correspond to machine instructions of the ARM described in the Emulator book [Pad15a], and can be grouped in the same categories: arithmetic and logic, memory, control flow, and software interrupt. To fully understand the semantics of those Asm5 instructions read the Emulator book [Pad15a].

8.2.1 Arithmetic and logic

And/or/xor

Here are the bitwise boolean logic opcodes of Asm5, which are enumeration cases of the `Opcode`[31] type (overviewed in Section 3.1):

⟨`Opcode` *cases, logic opcodes* 82d⟩≡ (31) 84f ▷
```
AAND,
AORR,
AEOR, // a.k.a. XOR
```

Their “reading syntax” are described via entries in `itab`[37c] presented in Section 3.5 (those entries populate the symbol table `hash`[39a] as explained in Section 3.6):

⟨`itab` *elements* 82e⟩≡ (37c) 84a ▷
```
"AND",  LARITH, AAND,
"ORR",  LARITH, AORR,
"EOR",  LARITH, AEOR,
```
Uses `LARITH`.

The logic opcodes take either 2 or 3 operands, mostly registers. Usually the first two operands are combined and the result stored in the third operand. Here are some examples:

```
AND $0x0f, R1, R2 // R2 = R1 & 0x0f;
ORR $0xff, R2     // <=> ORR $0xff, R2, R2
```

Logic instructions have the same syntax than arithmetic instructions in Asm5 and so share the same grammar rule:

⟨*inst rule(arm)* 83a⟩≡ (81b) 85f▷
```
inst:
/*
 * AND/ORR/ADD/SUB/...
 */
  LARITH cond imsr ',' regi ',' reg { outcode($1, $2, &$3, $5, &$7); }
| LARITH cond imsr ',' reg          { outcode($1, $2, &$3, R_NONE, &$5); }
```

The $n Yacc notation in the *actions* between braces gives access to the *value* of the nth terminal or non-terminal in the rule. For terminals this value is derived from the global `yylval`[104b] set by the lexer. Here are the different values for the different elements in the first rule of `inst`[83a] above:

1. `LARITH`[35] is the token name for all arithmetic and logic mnemonics. $1 contains the enumeration value of the corresponding opcode, for instance `AANDX` if the lexed token was `"AND"` (see the `itab` entries above)
2. `cond`[102h] represents the *conditional execution* of an instruction. It is an ARM feature we will explain later in Section 8.6. Every Asm5 instruction rules have a non-terminal `cond` after the opcode.
3. `imsr`[92d] is a shorthand for: `i`mmediate (constant) or `s`hifted-register or `r`egister. Those are operands we will describe in Section 8.4. $3 is of type `Gen`[33b], the generalized form of operand I described before.
4. the first comma, a single-character token without any value
5. `regi`[93f] is a register. $5 is an integer representing the register, e.g., 10 for `R10`.
6. the second comma
7. `reg`[93d] is also a register but wrapped in a `Gen`. We use a `Gen` for $7 and an integer for $5 because of the signature of `outcode()`[108] described in Section 3.4. Indeed, when an ARM instruction has 3 operands, the middle one is always a register, so an integer is enough for the fourth argument to `outcode()`.

Note that almost all operands above are registers, except the first one which can also be a constant or a shifted register. There is no memory reference. Indeed, the ARM is a RISC machine where memory references are restricted to only the LOAD and STORE operations (unified in the `MOV` virtual instruction in Asm5), which we will see later.

The three opcodes in this section have a direct correspondence with the following C operators: `&`, `|`, and `^`. They can also be used to encode boolean expressions involving `&&` and `||`. Those C operators are useful in many contexts: complex conditions in `if` and `while` statements (`if(e1 && e2)`), boolean variables (`bool b = b1 || b2;`), bitsets operations (`set1 | set2`), bit extraction (`x & 0x4`), bitmask (`y & 0xff`), etc. Those operators are heavily used in the Emulator book [Pad15a]. Indeed, managing the binary format of ARM instructions requires many bit manipulations.

Boolean logic is at the foundation of mathematics and so it is also logic (no pun intended) to have boolean opcodes as fundamental opcodes of the computer [2].

Add/sub

Basic arithmetic is also fundamental to have in a computer:

⟨`Opcode` *cases, add/sub opcodes* 83b⟩≡ (31) 85b▷
```
AADD,
ASUB,
```

[2] In fact, a whole computer can be made using just the Nand logic gate [NS05].

⟨itab *elements* 84a⟩+≡ (37c) ◁82e 84c▷
```
  "ADD",  LARITH, AADD,
  "SUB",  LARITH, ASUB,
```
Uses LARITH.

As said in the previous section, arithmetic instructions have the same syntax than logic instructions in Asm5 and so use the same grammar rule involving the LARITH[35] token code.

Again, The two opcodes above have a direct correspondence with C operators: + and -.

Mul/div/mod

The three opcodes below are in direct correspondence with the following C operators: *, /, %.

⟨Opcode *cases, mul/div/mod opcodes* 84b⟩≡ (31) 126c▷
```
  AMUL,
  ADIV, // VIRTUAL, transformed in call to _div
  AMOD, // VIRTUAL, transformed in call to _mod
```

⟨itab *elements* 84c⟩+≡ (37c) ◁84a 84e▷
```
  "MUL",  LARITH, AMUL,
  "DIV",  LARITH, ADIV,
  "MOD",  LARITH, AMOD,
```
Uses LARITH.

The ARM actually has no DIV or MOD instruction. Those virtual instructions are converted by 5l in calls to assembly functions of the core C library _div() and _mod() which implement the division and modulo algorithm in software. See Appendix D for the code of those assembly functions.

Sll/srl/sra

The opcodes below are in direct correspondence with the following C operators: <<, and >>:

⟨Opcode *cases, bitshift opcodes* 84d⟩≡ (31)
```
  ASLL, // Shift Left  Logic, VIRTUAL transformed in bitshifted registers
  ASRL, // Shift Right Logic, VIRTUAL transformed in bitshifted registers
  ASRA, // Shift Right Arithmetic, VIRTUAL transformed in bitshifted registers
```

⟨itab *elements* 84e⟩+≡ (37c) ◁84c 85a▷
```
  "SLL",  LARITH, ASLL,
  "SRL",  LARITH, ASRL,
  "SRA",  LARITH, ASRA,
```
Uses LARITH.

The ARM does not really have those opcodes though; it does not use those mnemonics for bit shift operations. Instead, the ARM has a more general approach to bit shifting: operands of many operations (AND, SUB, etc.) can be registers with *shift annotations* as explained later in Section 8.4.3. The opcodes above are thus translated by 5l in the MOV ARM instruction (which confusingly as nothing to do with the Asm5 virtual instruction MOV we will see later) with bitshifted register operands.

Bit shift operations are also heavily used in the Emulator book [Pad15a] because of the binary format of ARM instructions.

Other arithmetic and logic opcodes

There are a few additional Asm5 opcodes related to arithmetic and logic mimicking ARM instructions, but they are less useful:

⟨Opcode *cases, logic opcodes* 84f⟩+≡ (31) ◁82d
```
  ABIC, // ??
```

⟨itab *elements* 85a⟩+≡ (37c) ◁84e 85c▷
```
  "BIC",  LARITH, ABIC,
```
Uses LARITH.

⟨Opcode *cases, add/sub opcodes* 85b⟩+≡ (31) ◁83b 85d▷
```
  ARSB, // ??
  AADC, // Add and carry?
  ASBC, // Sub and carry?
  ARSC, // ??
```

⟨itab *elements* 85c⟩+≡ (37c) ◁85a 85e▷
```
  "RSB",  LARITH, ARSB,
  "ADC",  LARITH, AADC,
  "SBC",  LARITH, ASBC,
  "RSC",  LARITH, ARSC,
```
Uses LARITH.

⟨Opcode *cases, add/sub opcodes* 85d⟩+≡ (31) ◁85b
```
  AMVN, // MOV negative, but nothing to do with MOVW
```

⟨itab *elements* 85e⟩+≡ (37c) ◁85c 86b▷
```
  "MVN",  LMVN, AMVN, /* op2 ignored */
```
Uses LMVN.

⟨*inst rule(arm)* 85f⟩+≡ (81b) ◁83a 86c▷
```
  /*
   * MVN
   */
  | LMVN cond imsr ',' reg { outcode($1, $2, &$3, R_NONE, &$5); }
```

Many of those opcodes are actually never used in the assembly code generated by the C compiler 5c. See the EMULATOR book [Pad15a] to learn about their semantics.

8.2.2 Memory

Because the ARM is a RISC machine, all memory references in the ARM are restricted to mostly two machine instructions: LDR which *loads* some data from memory into a register, and STR which *stores* the content of a register into memory. In Asm5 those two instructions are unified in the single virtual instruction MOV. Another memory instruction SWP allows to *swap* the content of a register with the content at a memory address.

Moves

MOVs are converted by 5l in the appropriate machine instruction depending on the operands:

```
MOVW R1, (R2)  // This is a store
MOVW (R2), R1  // This is a load
```

MOVs come in different variants, depending on the size of the memory moved:

- MOVW for moving words (4 bytes)
- MOVB for moving bytes (1 byte)
- MOVH for moving half words (2 bytes)
- MOVBU for moving unsigned bytes (1 byte)

- MOVHU for moving unsigned half words (2 bytes)

Here are the corresponding Asm5 opcodes:

⟨Opcode *cases, mov opcodes* 86a⟩≡ (31) 127i ▷
```
AMOVW, // VIRTUAL, transformed in load and store instructions
AMOVB,
AMOVBU,
AMOVH,
AMOVHU,
```

⟨itab *elements* 86b⟩+≡ (37c) ◁85e 87b ▷
```
"MOVW",  LMOV, AMOVW,
"MOVB",  LMOV, AMOVB,
"MOVBU", LMOV, AMOVBU,
"MOVH",  LMOV, AMOVH,
"MOVHU", LMOV, AMOVHU,
```
Uses LMOV.

MOVs take a new kind of operand, gen[93b], which is very general (hence the name). gen accepts (as we will see later in Section 8.4) registers, constants, but also many forms of memory references such as the indirect with offset addressing mode in 4(R1):

⟨*inst rule(arm)* 86c⟩+≡ (81b) ◁85f 87c ▷
```
/*
 * MOVW
 */
| LMOV cond gen ',' gen { outcode($1, $2, &$3, R_NONE, &$5); }
```

Note that even if the grammar rule above is very general, allowing to write things like MOVW (R13), 6(R2), the linker 5l will actually not accept those instructions and report an error at linking time. Indeed, the linker will force the programmer to decompose the preceding instruction in two simpler instructions, which are then closer to the LDR and STR machine instructions of the ARM:

```
MOVW (R13), R1 // =~ LDR
MOVW R1, 6(R2) // =~ STR
```

This is why in the helloworld.s program in Section 2.3.2 we wrote code like:

```
MOVW $'W', R1
MOVB R1, 6(R2) // can not write MOVB $'W', 6(R2) directly
MOVW $'o', R1
MOVB R1, 7(R2) // can not write MOVB $'o', 7(R2) directly
```

Because MOVW is anyway a virtual instruction, you can wonder why the linker, instead of reporting an error, does not generate automatically the multiple LDR and STR instructions which are needed. After all, 5l transforms already certain Asm5 virtual instructions such as DIV or MOD in multiple machine instructions as we will see in the LINKER book [Pad15b]. My guess is that even though virtual instructions allow to abstract certain peculiarities (separate LDR and STR instructions), hide certain optimizations (code generated for leaf functions for TEXT and RET), or overcome certain limitations (no DIV and MOD), an assembly language should still try to mimic as closely as possible the instructions of a machine. By being forced to decompose and so to write two instructions in the example above, it is arguably easier for the programmer then to evaluate the number of cycles a procedure will take, or the number of memory references the procedure performs, by just counting lines.

Swaps

Memory/register swaps come also in different variants depending on the size of the memory swapped:

⟨Opcode *cases, swap opcodes* 87a⟩≡ (31)
```
  ASWPW,
  ASWPBU,
```

⟨itab *elements* 87b⟩+≡ (37c) ◁86b 87e▷
```
  "SWPW",  LSWAP, ASWPW,
  "SWPBU", LSWAP, ASWPBU,
```
Uses LSWAP.

⟨*inst rule(arm)* 87c⟩+≡ (81b) ◁86c 88a▷
```
  /*
   * SWAP
   */
  | LSWAP cond reg ',' ireg         { outcode($1, $2, &$5, $3.reg, &$3); }
  | LSWAP cond ireg ',' reg         { outcode($1, $2, &$3, $5.reg, &$5); }
  | LSWAP cond reg ',' ireg ',' reg { outcode($1, $2, &$5, $3.reg, &$7); }
```

The operand `ireg`[97d] stands for indirect register and will be described later. Here are a few examples of swapping instructions:

```
SWPW  R1, (R2)
SWPW (R1), R2
SWPW R1, (R2), R3
```

The point of `SWPW` may not be obvious. Why do we need such an instruction? `SWPW` is very useful though because the instruction is guaranteed to be *atomic*. `SWPW` is the foundation to build concurrency primitives. For instance the *test-and-set* function in Plan 9 in `lib_core/libc/arm/tas.s` is using `SWPW`.

8.2.3 Control flow

The control flow constructs of Asm5 are in direct correspondence with certain C constructs: unconditional jump with `goto`, conditional jump with `if`, branch and link with function call `x()` and `return`.

Jump (unconditional)

The most basic control flow instruction of Asm5 is the direct jump, called *branch* in the ARM, hence the `B` below:

⟨Opcode *cases, branching opcodes* 87d⟩≡ (31) 88e▷
```
  AB,  // =~ JMP
```

⟨itab *elements* 87e⟩+≡ (37c) ◁87b 88c▷
```
  "B",  LBRANCH, AB,
```
Uses LBRANCH.

The operand of a branch can be one of the element below:

- An offset to the PC pseudo register (e.g., `B 4(PC)`)
- A label (e.g., `B later`)
- A symbol (e.g., `B foo(SB)`)
- An indirect register (e.g., `B (R3)`)

The first two cases are covered by the the `rel`[99f] (register or label) non-terminal below, the last two cases by `nireg`[99d] (name or indirect register).

⟨*inst rule(arm)* 88a⟩+≡ (81b) ◁87c 88d▷
```
/*
 * B/BL
 */
| LBRANCH cond rel   { outcode($1, $2, &nullgen, R_NONE, &$3); }
| LBRANCH cond nireg { outcode($1, $2, &nullgen, R_NONE, &$3); }
```

It is easy to implement loops using `B`. The `helloworld.s` program in Section 2.3.2 contains one (stupid) loop. To not loop infinitely though one needs a way to escape the loop and jump elsewhere *if* a certain criteria is true.

Conditional jump and comparisons

The `helloworld.s` program in Section 2.3.2 did not contain any conditional jump. Indeed, printing `"hello world"` is too simple; there was no need for such an instruction. In the same way, there was also no need for an `if` in the corresponding `helloworld1.c` C program. The conditional jump though is a fundamental instruction. Without it programs would be just long sequences of instructions or infinite loops.

The conditional jump instruction is usually preceded by a *comparison instruction*. The following opcode allows to "compare" two operands in Asm5:

⟨`Opcode` *cases, comparison opcodes* 88b⟩≡ (31) 90a▷
```
ACMP,
```

⟨`itab` *elements* 88c⟩+≡ (37c) ◁87e 89a▷
```
"CMP",  LCMP, ACMP,
```
Uses LCMP.

⟨*inst rule(arm)* 88d⟩+≡ (81b) ◁88a 89b▷
```
/*
 * CMP
 */
| LCMP cond imsr ',' regi { outcode($1, $2, &$3, $5, &nullgen); }
```

Note that despite its name, this instruction does not really compare its two operands. Instead, it loads those two operands in some internal registers of the processor. Those internal registers can then be used by the instruction coming just after, for instance a conditional jump, to actually compare using different *relational operators* those registers and jump accordingly:

⟨`Opcode` *cases, branching opcodes* 88e⟩+≡ (31) ◁87d 90c▷
```
/*
 * Do not reorder or fragment the conditional branch
 * opcodes, or the predication code will break
 */
// VIRTUAL, AB derivatives with condition code, see 5i/
ABEQ, // ==
ABNE, // !=
ABHS, // >= unsigned (higher or same)
ABLO, // <  unsigned (lower)
ABMI, // minus/negative
ABPL, // plus/positive
ABVS, // overvlow set
ABVC, // overflow clear
ABHI, // >  unsigned
ABLS, // <= unsigned
ABGE, // >=
ABLT, // <
```

```
  ABGT, // >
  ABLE, // <=
  //ABAL (always) done via AB
  //ABNV (never) done via ANOP (see bcode[])
```

⟨itab *elements* 89a⟩+≡ (37c) ◁88c 90b▷

```
  "BEQ",  LBCOND, ABEQ,
  "BNE",  LBCOND, ABNE,
  "BHS",  LBCOND, ABHS,
  "BLO",  LBCOND, ABLO,
  "BMI",  LBCOND, ABMI,
  "BPL",  LBCOND, ABPL,
  "BVS",  LBCOND, ABVS,
  "BVC",  LBCOND, ABVC,
  "BHI",  LBCOND, ABHI,
  "BLS",  LBCOND, ABLS,
  "BGE",  LBCOND, ABGE,
  "BLT",  LBCOND, ABLT,
  "BGT",  LBCOND, ABGT,
  "BLE",  LBCOND, ABLE,
```

Uses LBCOND.

⟨*inst rule(arm)* 89b⟩+≡ (81b) ◁88d 91b▷

```
  /*
   * BEQ/...
   */
  | LBCOND rel { outcode($1, Always, &nullgen, R_NONE, &$2); }
```

Here is an example of a comparison and conditional jump:

```
    CMP $0, R1
    BEQ r1_is_zero
    B r1_is_nonzero
r1_is_zero:
    // R1 == 0
    ...
    B after
r1_is_nonzero:
    // else
    ...
    // fallthrough
after:
    ...
```

All the control flow constructs of C (`if`, `while`, `for`, etc.) can be translated in assembly code using simply labels, branches, and conditional jumps.

The B*xx* Asm5 instructions above are actually virtual instructions. Similar to the bitshift operations vs shifted-registers, the ARM provides a more general approach to conditional jump where *every instruction can be conditionally executed* as explained later in Section 8.6. The opcodes above are thus translated by 5l in the B instruction with special bits set to mark the instruction for conditional execution. Similar to the bitshift operations, Asm5 provides those specialized virtual instructions jumps because they correspond more to what assembly programmers (or compiler writers) expect from an assembly language.

There are a few additional Asm5 comparison opcodes similar to `CMP` mimicking ARM instructions, but they are less useful. Some of them, again, are not even used by `5c`:

⟨`Opcode` *cases, comparison opcodes* 90a⟩+≡ (31) ◁88b
```
  ATST,
  ATEQ,
  ACMN, // CMP negative
```

⟨`itab` *elements* 90b⟩+≡ (37c) ◁89a 90d▷
```
  "TST",  LCMP, ATST,
  "TEQ",  LCMP, ATEQ,
  "CMN",  LCMP, ACMN,
```
Uses `LCMP`.

Function call and return

The `B`, `CMP`, and `B`*xx* instructions are enough to have a Turing-complete machine. There is no need for a function call instruction; it would not add any expressivity to the computer. In fact, there is no `CALL` or `RET` instruction in the ARM, but instead a very basic `BL` instruction. As explained in Section 2.3.10, `BL` stands for *branch and link* because the instruction just saves the current value of the program counter (`R15`) in a special *link register* (`R14`), and then branch/jump:

⟨`Opcode` *cases, branching opcodes* 90c⟩+≡ (31) ◁88e 90e▷
```
  ABL, // =~ CALL, Branch and Link
```

⟨`itab` *elements* 90d⟩+≡ (37c) ◁90b 91a▷
```
  "BL",  LBRANCH, ABL,
```
Uses `LBRANCH`.

The `BL` instruction uses the same syntax than `B` and so use the same grammar rule involving the `LBRANCH`[35] token name. Indeed, `BL` is really just a slightly different `B`. But, `BL` performs in one instruction something which can be used as a building block for organizing code into functions. Indeed, by using `BL`, the link register `R14`, the stack pointer `R13`, and symbols defined by `TEXT`, you can easily simulate functions, function calls, and `return` as explained in Sections 2.3.10 and 2.3.11.

From a theoretical computer science point of view, functions do not add any expressivity to the machine language, but from a software engineering perspective they are tremendously useful. Indeed, with functions you can decompose a big program in multiple independent parts which can be programmed and understood in *isolation*. Because those functions though must use and *share* the same machine registers, one needs to setup some *conventions* on how to use and save those registers so that code of one function can be programmed independently of the code of another function. In Plan 9, the convention is as follows:

> "a subroutine is responsible for saving its own registers, and therefore is free to use any registers without saving them".

This convention is also known as "caller saves". This means that if the body of a function `foo` uses the register `R1`, and then must call another function `bar` (with `BL bar(SB)`), then `foo` must save the register `R1` somewhere if it plans to use its value after the call to `bar`, because `bar` could have overwritten the value in `R1`. The stack is usually used to save the values of those registers.

We need also to setup *calling conventions* on how to pass arguments and use parameter in functions. In Plan 9 the stack and `R0` are used to pass the arguments as explained in Section 2.3.8. By using the stack and not fixed memory locations, you can call functions recursively.

Asm5 provides the virtual instruction `RET` to make it easier for the programmer to manage the stack when returning from a function. Indeed, `RET` abstracts away the differences between leaf and non-leaf functions (as explained in Section 2.3.11):

⟨`Opcode` *cases, branching opcodes* 90e⟩+≡ (31) ◁90c
```
  ARET, // VIRTUAL, transformed in B (R14) or MOV xxx(R13), R15
```

⟨itab *elements* 91a⟩+≡ (37c) ◁90d 91d▷
```
  "RET",  LRET, ARET,
```
Uses LRET.

⟨*inst rule(arm)* 91b⟩+≡ (81b) ◁89b 91e▷
```
  /*
   * RET
   */
  | LRET cond { outcode($1, $2, &nullgen, R_NONE, &nullgen); }
```

8.2.4 Software interrupt

The software interrupt instruction SWI is really a form of function call but to a fixed set of special routines setup by the kernel. Normally the argument to SWI specifies an entry in an interrupt table but under Plan 9 this argument is actually not used and instead R0 is used to store the syscall code.

⟨Opcode *cases, interrupt opcodes* 91c⟩≡ (31) 91f▷
```
  ASWI, // syscall
```

⟨itab *elements* 91d⟩+≡ (37c) ◁91a 91g▷
```
  "SWI",  LSWI, ASWI,
```
Uses LSWI.

⟨*inst rule(arm)* 91e⟩+≡ (81b) ◁91b 100f▷
```
  /*
   * SWI
   */
  | LSWI cond gen { outcode($1, $2, &nullgen, R_NONE, &$3); }
```

SWI is the fundamental building block for having a kernel program clearly separated from user programs. It provides a safe way to go from user program to kernel code via the system call API bridge.

Asm5 provides also the virtual instruction RFE to return from kernel code to a user program. RFE is actually just a short alias to the more cryptic MOVM instruction we will describe in Section 11.2.4.

⟨Opcode *cases, interrupt opcodes* 91f⟩+≡ (31) ◁91c
```
  ARFE, // VIRTUAL, return from exception/interrupt, MOVM.IA.S.W (R13), [R15]
```

⟨itab *elements* 91g⟩+≡ (37c) ◁91d 93g▷
```
  "RFE",  LRET, ARFE,
```
Uses LRET.

8.3 Label definitions and pc

Labels allow to give symbolic names to code addresses. They are a fundamental feature of any assembly language and makes assembly code significantly easier to understand and maintain. Indeed, by using label names such as _loop1, if_zero, or _else, assembly code using jumps (conditional or unconditional) can become closer to C code using statements such as for or if. Labels in Asm5 are defined on a line by using an identifier followed by a colon:

⟨*line rule(arm)* 91h⟩+≡ (81b) ◁82c 92c▷
```
  | LNAME ':'
   {
    $1->type = LLAB;
    $1->value = pc;
   }
    line
```

Even though you could have multiple label definitions on a line, or even labels and instructions on the same line, it is a common practice to put the label definition alone on its own line, in the first column.

The value of a label is the value of the *virtual program counter* at the label definition. This counter is stored in the global `pc`, initialized to 0 before each pass and incremented by `outcode()`[108] after each code instruction:

⟨*global* `pc` 92a⟩≡ (160a)
```
long pc;
```

⟨`pinit()` *initializations* 92b⟩+≡ (50a) ◁57c 111b▷
```
pc = 0;
```
Uses `pc` 92a.

Once a label has been defined, the code of its token in the symbol table is changed to `LLAB`[35] [3] as shown in the rule above. Further use of this label in the rest of the program or in the next assembling pass will trigger then this rule:

⟨*line rule(arm)* 92c⟩+≡ (81b) ◁91h 122a▷
```
| LLAB ':'
 {
  if($1->value != pc) {
   yyerror("redeclaration of %s", $1->name);
   $1->value = pc;
  }
 }
  line
```

Note that there is no use of `outcode()` here because labels can be fully resolved by the assembler. There is no need to keep them in the object file.

8.4 Operands

Now that we have seen the major opcodes and instructions of Asm5, we can switch to explain the operands of those instructions. The first non-terminal corresponding to an operand we saw in this chapter was `imsr`, the first operand of opcodes such as `AADDX` or `AORRX`:

⟨*operand rules(arm)* 92d⟩≡ (81b) ??▷
```
imsr:
  imm
| shift
| reg
```

The next three subsections will each cover one of the `imsr`[92d] alternatives.

Rules which return values (in $$) must have their type declared in Yacc via a `%type` directive. Most operand rules have the `genval` type:

⟨*type* `declarations`*(arm)* 92e⟩≡ (81a) 93a▷
```
%type   <genval> imsr
%type   <genval> imm shift reg
```

In fact, `genval` is not really a type but the name of one of the field of the Yacc `%union` declaration (described partially in Section 3.5):

⟨*union declaration other fields(arm)* 92f⟩+≡ (37a) ◁42a
```
Gen    genval;
```

[3]This means Yacc actions actually have a side effect on how the lexer behaves. Asm5 is thus not a context free grammar. This is similar to C where typedefs declarations change the token code of certain identifiers.

The type of genval[92f] is Gen[33b], the generalized form of operand I described before. Thanks to those %type and %union Yacc directives, the generated code by Yacc for the $\$n$ referring to non-terminals will access the appropriate field of the union. In the same way, the %token and %union Yacc directives will guide Yacc to generate the right code for the $\$n$ referring to terminals (the tokens).

The gen operand rule, specifying the operands of MOVW, has the same type:

⟨*type* declarations*(arm)* 93a⟩+≡ (81a) ◁92e 93c▷
```
%type   <genval>   gen
```

⟨*gen rule* 93b⟩≡ (? 0—1)
```
gen:
  ximm
| shift
| reg
```
⟨*more gen rule* 97a⟩

For now gen[93b] is almost identical to imsr, except for the ximm[95a] (extended immediate) vs imm[94c] which I will explain soon, but we will see later many extensions to the gen rule.

⟨*type* declarations*(arm)* 93c⟩+≡ (81a) ◁93a 93e▷
```
%type   <genval>   ximm
```

8.4.1 Registers

Most Asm5 instructions can take one, two, or even three registers as operands. The reg rule below allows to wrap in a Gen[33b] the integer representing the register number. It uses the D_REGX *operand kind* I described before:

⟨*operand rules(arm)* 93d⟩+≡ (81b) ◁?? 94c▷
```
reg:
 regi
 {
  $$ = nullgen;
  $$.type = D_REG;
  $$.reg = $1;
 }
```

regi (register integer) returns directly the register number:

⟨*type* declarations*(arm)* 93e⟩+≡ (81a) ◁93c 96a▷
```
%type   <lval>  regi
```

⟨*regi rule(arm)* 93f⟩≡ (? 0—1) 94b▷
```
regi:
  LREG
```

⟨itab *elements* 93g⟩+≡ (37c) ◁91g 94a▷
```
"R0",  LREG, 0,
"R1",  LREG, 1,
"R2",  LREG, 2,
"R3",  LREG, 3,
"R4",  LREG, 4,
"R5",  LREG, 5,
"R6",  LREG, 6,
"R7",  LREG, 7,
"R8",  LREG, 8,
"R9",  LREG, 9,
"R10",  LREG, 10,
"R11",  LREG, 11,
"R12",  LREG, 12,
```

```
  "R13",  LREG, 13,
  "R14",  LREG, 14,
  "R15",  LREG, 15,
```
Uses LREG.

Asm5 allows also the R(xx) syntax to specify a register:

⟨itab *elements* 94a⟩+≡ (37c) ◁93g 96c▷
```
  "R",  LR, 0,
```
Uses LR.

⟨*regi rule(arm)* 94b⟩+≡ (? 0—1) ◁93f
```
  | LR '(' expr ')'
   {
    if($3 < 0 || $3 >= NREG)
        print("register value out of range\n");
    $$ = $3;
   }
```

This feature can be useful when combined with the symbolic constant feature of Asm5, as shown in Appendix D.3. You can achieve a similar effect though by using simply macros.

8.4.2 Immediate constants

Many instructions can take a register or an (immediate) constant as their first operand. This is pretty fundamental of course. All constants in Asm9 are prefixed with a dollar:

⟨*operand rules(arm)* 94c⟩+≡ (81b) ◁93d 95c▷
```
  imm: '$' con
   {
    $$ = nullgen;
    $$.type = D_CONST;
    $$.offset = $2;
   }
```

⟨*con rule* 94d⟩≡ (121d) 121b▷
```
  con:
    LCONST
  | '-' con      { $$ = -$2; }
  | '+' con      { $$ = $2; }
  | '~' con      { $$ = ~$2; }
```

Note that the ARM uses fixed-length instructions of 32 bits. Because a few of those 32 bits are used to encode the opcode and a few more to encode the other operands, the range for ARM immediate constants is limited. For arithmetic instructions, only 12 bits of the instruction can be used for the constant, so in theory only constants between 0 and 4096 could be represented. But, the ARM uses a clever trick and those 12 bits are actually divided in two parts: 8 bits for a constant, and 4 bits for a *rotation* of this constant. 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 or bitmasks. See the EMULATOR book [Pad15a] and LINKER book [Pad15b] for more information or http://alisdair.mcdiarmid.org/arm-immediate-value-encoding/ which is an excellent tutorial on the topic. In any case, Asm5 allows any 32 bits constants in operands. If the constant does not fit the immediate constant constraint of the ARM, the linker 5l will use multiple instructions to encode the constant.

MOVs instructions, which use the gen[93b] non-terminal as operand (which derives ximm) can also take a constant as an operand:

⟨*ximm rule* 95a⟩≡ (? 0—1) 95b▷
```
ximm:
  '$' con
 {
  $$ = nullgen;
  $$.type = D_CONST;
  $$.offset = $2;
 }
```

In fact, even string constants are possible operands of MOVs:

⟨*ximm rule* 95b⟩+≡ (? 0—1) ◁95a 98e▷
```
| '$' LSCONST
 {
  $$ = nullgen;
  $$.type = D_SCONST;
  memcpy($$.sval, $2, sizeof($$.sval));
 }
```

For those cases the constant itself can not be encoded in 32 bits though. The address of the string, which will become an integer constant after linking, will be used instead in the generated ARM instruction.

8.4.3 ARM shifted registers

As mentioned in Section 8.2.1, in addition to the (virtual) instructions SLL, SRL, and SRA, Asm5 gives access to the ARM generalized approach to bitshift by allowing to use *shifted registers* as the first operand in all arithmetic instructions:

⟨*operand rules(arm)* 95c⟩+≡ (81b) ◁94c 97c▷
```
shift:
 regi '<' '<' rcon
 {
  $$ = nullgen;
  $$.type = D_SHIFT;
  $$.offset = $1 | $4 | (0 << 5);
 }
| regi '>' '>' rcon
 {
  $$ = nullgen;
  $$.type = D_SHIFT;
  $$.offset = $1 | $4 | (1 << 5);
 }
| regi '-' '>' rcon
 {
  $$ = nullgen;
  $$.type = D_SHIFT;
  $$.offset = $1 | $4 | (2 << 5);
 }
| regi LAT '>' rcon
 {
  $$ = nullgen;
  $$.type = D_SHIFT;
  $$.offset = $1 | $4 | (3 << 5);
 }
```

A shifted-register combines a register (a D_REGX) and an immediate constant (a D_CONSTX) in a new kind of operand:

⟨Operand_kind *cases* 95d⟩+≡ (33c) ◁42e 99g▷
```
D_SHIFT,
```

You can then use an opcode like ADD while at the same time doing a shift operation on one of the register in one single instruction (which opens many optimizations opportunities in the C compiler 5c):

```
ADD R1 >> 8, R2, R3
```

Note that Gen.offset is used to encode the whole operand value [4]: 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 via rcon (register or constant) below.

⟨*type* declarations*(arm)* 96a⟩+≡ (81a) ◁93e 97b▷
```
  %type   <lval>  rcon
```

⟨*helper rules(arm)* 96b⟩≡ (81b) 98a▷
```
  rcon:
    regi
   {
    if($1 < 0 || $1 >= NREG)
        print("register value out of range\n");
    $$ = (($1&15) << 8) | (1 << 4);
   }
  | con
   {
    if($1 < 0 || $1 >= 32)
        print("shift value out of range\n");
    $$ = ($1&31) << 7;
   }
```

Putting the four fields together, here is what Gen.offset actually looks like for a single shifted-register operand:

```
  bit:  11  10   9   8 | 7 | 6   5 | 4 | 3   2   1   0
       +---+---+---+---+---+---+---+---+---+---+---+---+
       |  shift amt / reg  | kind  |isr|  base register |
       +---+---+---+---+---+---+---+---+---+---+---+---+

  base register (bits 0..3):  R0 .. R15
  isr           (bit  4)   :  0 = shift amount is a constant
                              1 = shift amount is a register
  kind          (bits 5..6):  00 = LSL (<<)    10 = ASR (->)
                              01 = LSR (>>)    11 = ROR (@>)
  shift amount  :  if isr = 0, bits 7..11 hold a 5-bit constant
                   if isr = 1, bits 8..11 hold a register number
                               (bit 7 stays 0)
```

For example, `R1 << 4` is encoded as `1 | (4<<7) | (0<<5) = 0x201`: register 1 in bits 0..3, the 5-bit constant 4 in bits 7..11, shift kind LSL (zero) in bits 5..6, and isr zero. Each action in the shift and rcon rules above contributes one slot of this picture.

The tricky Gen.offset encoding for bitshifted registers reflects how such operands are actually encoded in the binary format of ARM instructions.

In addition to the << and >> logical shift operators, Asm5 provides the -> operator for right shift arithmetic and @> for right rotate. Again, see the EMULATOR book [Pad15a] for more information. Because @ can be part of an identifier in Asm9, a special entry in itab[37c] had to be used to represent the single character:

⟨itab *elements* 96c⟩+≡ (37c) ◁94a 98b▷
```
  "@",  LAT, 0,
```
Uses LAT.

[4] We could use Gen.reg and Gen.offset instead.

8.4.4 Memory (de)references, pointers

We have seen the three components of the `imsr`[92d] operand rule used in arithmetic instructions: the immediate constant, the shifted register, and the register. We will now focus on the `MOVW` instruction and its `gen`[93b] operand which is even more general (hence the name). Section 2.5.5 listed the different ways to reference memory in Asm5 (the different memory addressing modes), for instance "indirect with offset" in `4(R1)`.

This is where we start to deviate from `imsr` in `gen` (in addition to the `ximm`[95a] vs `imm`[94c] difference):

⟨*more gen rule* 97a⟩≡ (93b) 97e▷
```
 | ioreg
```

`ireg` (indirect register) allows to dereference a pointer stored in a register. `ioreg` (indirect offset register) adds an offset to the memory address in the register. Both are using the `D_OREGX` operand kind I introduced before:

⟨*type* `declarations`*(arm)* 97b⟩+≡ (81a) ◁96a 97f▷
```
 %type   <genval> ioreg ireg
```

⟨*operand rules(arm)* 97c⟩+≡ (81b) ◁95c 97d▷
```
 ioreg:
   ireg
 | con '(' regi ')'
  {
   $$ = nullgen;
   $$.type = D_OREG;
   $$.reg = $3;
   $$.offset = $1;
  }
```

⟨*operand rules(arm)* 97d⟩+≡ (81b) ◁97c 99d▷
```
 ireg:
  '(' regi ')'
  {
   $$ = nullgen;
   $$.type = D_OREG;
   $$.reg = $2;
   $$.offset = 0;
  }
```

8.4.5 Named memory locations, symbols

You can also use symbols to reference memory in Asm5:

⟨*more gen rule* 97e⟩+≡ (93b) ◁97a 98d▷
```
 | name
```

⟨*type* `declarations`*(arm)* 97f⟩+≡ (81a) ◁97b 99c▷
```
 %type   <genval>   name
 %type   <lval>  offset
 %type   <lval>  pointer
```

⟨*name rule* 97g⟩≡ (? 0—1) 99a▷
```
 name:
   LNAME offset '(' pointer ')'
  {
   $$ = nullgen;
   $$.type = D_OREG;
   $$.sym = $1;
   $$.symkind = $4;
   $$.offset = $2;
  }
```

Note that symbol operands use the same operand kind `D_OREGX` than the indirect register with offset we have seen above. This is because a name can also be seen as a sort of "offset" to one of the (pseudo) register we have seen in Section 2.3.7 (except `PC`):

⟨*helper rules(arm)* 98a⟩+≡ (81b) ◁96b 98c▷

```
pointer:
  LSB
| LSP
| LFP
```

⟨`itab` *elements* 98b⟩+≡ (37c) ◁96c 100c▷

```
"SB",  LSB, N_EXTERN,
"SP",  LSP, N_LOCAL,
"FP",  LFP, N_PARAM,
```

Uses `LFP`, `LSB`, and `LSP`.

The name (which itself is seen as an offset to a pseudo register) can also have an additional offset applied to it:

⟨*helper rules(arm)* 98c⟩+≡ (81b) ◁98a

```
offset:
 /* empty */ { $$ = 0; }
| '+' con    { $$ = $2; }
| '-' con    { $$ = -$2; }
```

Locals and parameters, `SP` and `FP`

Note that for locals and parameters the symbol name is not really necessary, which is why one can also reference parameters only with an offset, as in `MOVW 4(FP), R1`:

⟨*more gen rule* 98d⟩+≡ (93b) ◁97e 125b▷

```
| con '(' pointer ')'
 {
  $$ = nullgen;
  $$.type = D_OREG;
  $$.sym = S;
  $$.symkind = $3;
  $$.offset = $1;
 }
```

The use of symbols for locals and parameters can be useful though as comments, as in `MOVW count+4(FP), R1`. The symbol name is also kept in the symbol table of the object file as we will see in Chapter 9 which can be leveraged later by tools like debuggers as we will see in Chapter 10.

Symbol addresses

Using a `name`[99a] as part of a `MOVW` implicitly means a pointer dereference to access the value at the address denoted by the symbol, e.g., `MOVW foo(SB), R1`. To get the address of the symbol itself one needs to prefix the name with a dollar, e.g., `MOVW $foo(SB), R1`. This is where `ximm`[95a] deviates from `imm`[94c]:

⟨*ximm rule* 98e⟩+≡ (? 0—1) ◁95b 125c▷

```
| '$' name
 {
  $$ = $2;
  $$.type = D_ADDR;
 }
```

Note that the `name` non-terminal can also be the operand of the `TEXT` pseudo instruction as we will see soon, in which case it defines the symbol.

Private symbols

Asm5 provides also a way to use *private* (a.k.a static) symbols for globals and procedures in a file by suffixing the name with angles:

⟨*name rule* 99a⟩+≡ (? 0—1) ◁97g

```
  | LNAME '<' '>' offset '(' LSB ')'
   {
    $$ = nullgen;
    $$.type = D_OREG;
    $$.sym = $1;
    $$.symkind = N_INTERN;
    $$.offset = $4;
   }
```

⟨Sym_kind *cases* 99b⟩≡ (42d) 120b▷

```
  N_INTERN, // private (aka static) entities (from SB)
```

Those names will not be visible outside the file, but they will also not conflict with identical names in other files.

Branching symbols

Branching instructions (e.g., BL) can also jump to symbols via the nireg operand rule below. They can also jump to the address contained in a register:

⟨*type* declarations*(arm)* 99c⟩+≡ (81a) ◁97f 99e▷

```
  %type   <genval>   nireg
```

⟨*operand rules(arm)* 99d⟩+≡ (81b) ◁97d ??▷

```
  nireg:
    name
  | ireg
```

8.4.6 Code references, labels

The last operand rule we have to see is one of the operand rule of branching instructions (B, BL) called rel (register or label):

⟨*type* declarations*(arm)* 99e⟩+≡ (81a) ◁99c 102g▷

```
  %type   <genval>   rel
```

⟨*rel rule* 99f⟩≡ (? 0—1) 100a▷

```
  rel:
    LLAB offset
   {
    $$ = nullgen;
    $$.type = D_BRANCH;
    $$.sym = $1;
    $$.offset = $1->value + $2;
   }
```

This operand uses a new kind of operand:

⟨Operand_kind *cases* 99g⟩+≡ (33c) ◁95d 124e▷

```
  D_BRANCH,
```

The value for this kind of operands is the resolved absolute code address of the label. It is stored in `Gen.offset`.

When a label is declared later in a file, the first mention of this label in an operand in the first pass will trigger the rule below. Indeed, the symbol table does not know yet that this identifier is really a `LLAB`[35] not an `LNAME`[35]:

⟨*rel rule* 100a⟩+≡ (? 0—1) ◁99f 100b▷
```
 | LNAME offset
  {
   $$ = nullgen;
   if(pass == 2)
       yyerror("undefined label: %s", $1->name);
  }
```

Another way to reference code is to add an offset to the virtual program counter `PC`, allowing to make *relative jumps* (even though this is transformed in an absolute code address thanks to `pc`[92a]):

⟨*rel rule* 100b⟩+≡ (? 0—1) ◁100a
```
 | con '(' LPC ')'
  {
   $$ = nullgen;
   $$.type = D_BRANCH;
   $$.offset = $1 + pc;
  }
```

⟨`itab` *elements* 100c⟩+≡ (37c) ◁98b 100e▷
```
 "PC",  LPC, D_BRANCH,
```
Uses `LPC`.

8.5 Pseudo instructions

Pseudo instructions have an opcode and operands, just like the other instructions we have seen before, but they do not correspond to any machine instruction. This is also true for virtual instructions like `MOVW` or `RET` but virtual instructions eventually become machine instructions after linking. Pseudo instructions on the opposite are assembly-only constructs, also know as *assembly directives*. It would be more accurate to call them *linker directives* though because they declare things which will be used by the linker and which will eventually disappear after linking.

8.5.1 TEXT/GLOBL

Symbols, for procedures or globals, are declared respectively by the `TEXT` and `GLOBL` pseudo instructions (as explained in Section 2.3.3):

⟨`Opcode` *cases, pseudo opcodes* 100d⟩≡ (31) 101a▷
```
 ATEXT,
 AGLOBL,
```

⟨`itab` *elements* 100e⟩+≡ (37c) ◁100c 101b▷
```
 "TEXT",  LDEF, ATEXT,
 "GLOBL", LDEF, AGLOBL,
```
Uses `LDEF`.

⟨*inst rule(arm)* 100f⟩+≡ (81b) ◁91e 101c▷
```
 /*
  * TEXT/GLOBL
  */
 | LDEF name ',' imm         { outcode($1, Always, &$2, R_NONE, &$4); }
 | LDEF name ',' con ',' imm { outcode($1, Always, &$2, $4, &$6); }
```

The same non-terminal `name`[99a] we have seen in Section 8.4.5 to reference memory via a symbol is used also here to define a symbol. This is why one needs also to add the `(SB)` suffix in symbol definitions as in `TEXT _main(SB), $20`.The second operand is an immediate constant which represents a size in bytes for the locals of the procedure or for the global as explained in Section 2.3.3. The `TEXT` and `GLOBL` can also take three operands as indicated by the second alternative in the grammar rule above, in which case the middle operand is used to encode *attributes* of the entity defined as explained in Section 11.4. Note that pseudo instructions have the same binary format than other Asm5 instructions in the object file, hence the similar use of `outcode()`[108] above in the action.

`TEXT` and `GLOBL` are pseudo instructions because they give names to procedures and globals while a machine has no notion of procedure, global, or name. Indeed, those names eventually become concrete addresses after linking, which are really just concrete numbers. Those numbers, not names, are then used as arguments of machine instructions such as `LDR` (to access globals), or `BL` (to call procedures).

Labels are similar to `TEXT` symbols but they are not public and do not have a size. This is because labels are used for intra-procedural jumps, while `TEXT` symbols are used for public procedures with parameters. Because labels are private to a file, all the references to a label must be in the same file. All those references can thus be resolved by the assembler and converted in absolute (virtual) code addresses in the object file. `TEXT` symbols on the opposite are public and may be referenced by other files which is why they have to be kept in the object file, to be resolved later by the linker.

`TEXT` pseudo instructions are used by the linker to put the code of procedures in the *text section* of the executable. `GLOBL` are used by the linker to allocate some space in the *data section* of the executable.

Note that for procedures, the constant used to represent the size of the locals can surprisingly also be `$-4`. Indeed, as explained in Section 2.3.8, each time a procedure is entered an extra word in the stack is by default allocated before the locals to store the return address stored in `R14`. This is true even when the procedure does not declare any local as in `TEXT foo(SB), $0`, if `foo()` is not a leaf procedure. In some situations though the programmer would like to change this default behavior and not allocate anything at all in the stack in which case he must use `$-4`

8.5.2 WORD/DATA

`GLOBL` tells the linker to allocate some space in the data section of the executable but it is the `DATA` pseudo instruction which tells the linker how to fill this space (as explained in Section 2.3.13):

⟨`Opcode` *cases, pseudo opcodes* 101a⟩+≡ (31) ◁100d 102a▷

```
  ADATA,
```

⟨`itab` *elements* 101b⟩+≡ (37c) ◁100e 102b▷

```
  "DATA",  LDATA, ADATA,
```

Uses `LDATA`.

⟨*inst rule(arm)* 101c⟩+≡ (81b) ◁100f 102c▷

```
  /*
   * DATA
   */
  | LDATA name '/' con ',' ximm { outcode($1, Always, &$2, $4, &$6); }
```

Note that the `name`[99a] can contain an offset as in `hello+8(SB)`. The `con`[94d] specifies the size in bytes this `DATA` pseudo instruction is defining. Note also the use of `ximm`[95a] which allows to use string constants or even addresses of other globals as in `$otherdata(SB)`).Here are some examples of use of `DATA`:

```
DATA    hello+0(SB)/8, $"hello wo"
DATA    hello+8(SB)/4, $"rld\n"
...
DATA boot_CONF_outlen+0(SB)/4, $67523
```

```
DATA boot_CONF_outcode+0(SB)/8, $"\z\z\1\353\z\z\213\33"
...
```

DATA allows to put any value in the data section of the executable. A similar pseudo instruction, WORD, does the same for the text section:

⟨Opcode *cases, pseudo opcodes* 102a⟩+≡ (31) ◁101a 102d▷
```
  AWORD,
```

⟨itab *elements* 102b⟩+≡ (37c) ◁101b 102e▷
```
  "WORD",  LWORD, AWORD,
```
Uses LWORD.

⟨*inst rule(arm)* 102c⟩+≡ (81b) ◁101c 102f▷
```
  /*
   * WORD
   */
  | LWORD ximm { outcode($1, Always, &nullgen, R_NONE, &$2); }
```

The main use of WORD is to overcome some of the limitations of the assembler. Indeed, for 5a, even if certain system ARM instructions do not have yet an Asm5 mnemonics, you can still use WORD to specify this instruction if one knows the exact binary encoding of this instruction. You can even use WORD in a macro as shown in Section 2.5.6.

8.5.3 END

The END pseudo instruction is a marker used to indicate the end of the instructions in the object file:

⟨Opcode *cases, pseudo opcodes* 102d⟩+≡ (31) ◁102a 111d▷
```
  AEND,
```

⟨itab *elements* 102e⟩+≡ (37c) ◁102b 103a▷
```
  "END",  LEND, AEND,
```
Uses LEND.

⟨*inst rule(arm)* 102f⟩+≡ (81b) ◁102c 124d▷
```
  /*
   * END
   */
  | LEND { outcode($1, Always, &nullgen, R_NONE, &nullgen); }
```

The need for such an instruction in the assembly file is not clear. Having such a marker in the object file can be useful though. Indeed, when many object files are concatenated together in a library, the AENDX marker can be used to indicate object boundaries. Note that 5a automatically adds such a marker at the end of the generated object file via cclean()[47b] (see the code of cclean()).

8.6 ARM conditional execution

In the ARM, *every instruction can be conditionally executed*, not just branching instructions. This is why every Asm5 instructions we have seen before uses the cond non-terminal in their grammar rule. In Asm5, you can suffix any opcode with a *condition* as in ADD.EQ R1, R2, R3:

⟨*type* declarations*(arm)* 102g⟩+≡ (81a) ◁99e 121a▷
```
  %type   <lval>  cond
```

⟨*cond rule(arm)* 102h⟩≡ (81b) 103d▷
```
  cond:
    /* empty */ { $$ = Always; }
  | cond LCOND  { $$ = ($1 & ~C_SCOND) | $2; }
```

⟨itab *elements* 103a⟩+≡ (37c) ◁102e 103e▷

```
".EQ",  LCOND, 0,
".NE",  LCOND, 1,
".HS",  LCOND, 2,
".LO",  LCOND, 3,
".MI",  LCOND, 4,
".PL",  LCOND, 5,
".VS",  LCOND, 6,
".VC",  LCOND, 7,
".HI",  LCOND, 8,
".LS",  LCOND, 9,
".GE",  LCOND, 10,
".LT",  LCOND, 11,
".GT",  LCOND, 12,
".LE",  LCOND, 13,
".AL",  LCOND, Always,
```

Uses `Always` 103b and `LCOND`.

⟨*constant* `Always`*(arm)* 103b⟩≡ (167a)

```
#define    Always 14
```

So, the branching instruction `BEQ` we saw in Section 8.2.3 is really just an alias for `B.EQ`. The need to conditionally execute non-branching instructions (e.g., an addition) may not be obvious. This feature was originally designed to compensate for the lack of branch predicator in ARM CPUs by having code using less branches. See `https://en.wikipedia.org/wiki/ARM_architecture#Conditional_execution` or the Emulator book [Pad15a] for more information .

There are bit manipulations involving `C_SCOND` above because the `cond`[102h] non-terminal in addition to the conditional execution also stores special ARM instruction bits (as explained in Section 8.7):

⟨*constant* `C_SCOND`*(arm)* 103c⟩≡ (154c)

```
/* scond byte */
#define C_SCOND ((1<<4)-1)
```

8.7 Special bits

8.7.1 Arithmetic instructions and `.S`

⟨*cond rule(arm)* 103d⟩+≡ (81b) ◁102h

```
| cond LS    { $$ = $1 | $2; }
```

⟨itab *elements* 103e⟩+≡ (37c) ◁103a 103g▷

```
".S",  LS, C_SBIT,
```

Uses `LS`.

⟨*constant* `C_SBIT`*(arm)* 103f⟩≡ (154c)

```
#define C_SBIT  (1<<4)
```

8.7.2 Moves and `.P`/`.W`

⟨itab *elements* 103g⟩+≡ (37c) ◁103e 123d▷

```
".P",  LS, C_PBIT,
".W",  LS, C_WBIT,
```

Uses `LS`.

⟨*constant* `C_PBIT`*(arm)* 103h⟩≡ (154c)

```
#define C_PBIT  (1<<5)
```

⟨*constant* `C_WBIT`*(arm)* 104a⟩≡ (154c)
```
#define C_WBIT  (1<<6)
```

8.8 yyparse()

Based on the grammar of Asm5 we have seen in this chapter, Yacc will generate:

- a `y.tab.h` header file containing the token codes and the union for `yylval`[104b], as explained in Section 3.5. Both the token codes and `yylval` are used by `yylex()`[56b].
- a `y.tab.c` C file containing the function `yyparse()`[104c]. This function will internally call `yylex()` (which is why we call our lexing function `yylex()` in Chapter 6) to get tokens until the end of the file token (`EOF`[55a]), and will parse those tokens according to the first grammar rule declared in the file: `prog`.

Here are excerpts of those generated files:

⟨`y.tab.h` *generated file excerpt* 104b⟩≡
```
typedef union  {
 long   lval;
 double dval;
 char   sval[NSNAME];
 Sym    *sym;
 Gen    genval;
} YYSTYPE;
extern YYSTYPE yylval;
#define LARITH 57346
#define LCMP 57347
...
```

⟨`y.tab.c` *generated file excerpt* 104c⟩≡
```
typedef union  {
 long   lval;
 double dval;
 char   sval[NSNAME];
 Sym    *sym;
 Gen    genval;
} YYSTYPE;
extern int yyerrflag;
...
YYSTYPE yylval;
...
#define LARITH 57346
#define LCMP 57347
...
int
yyparse(struct Yyarg *yyarg)
{
...
}
```

Chapter 9

Object Code Generation

We are now ready to present the last component in the assembling pipeline: the object code generation. As mentioned before, `5a` does not actually generate machine code (the linker does), which simplifies things. Indeed, the only thing `5a` needs to do is to serialize the Asm5 instructions and to record in the object file the set of symbols used in the assembly file, and where those symbols are used, so that they can be resolved later by the linker. Note that labels are resolved during parsing by `5a` and so their definitions do not need to be stored in the object file. The LINKER book [Pad15b] will describe then the reverse operation which is to read an object file. It will also describe which machine code is generated from the object files and how those object files are combined to form an executable.

9.1 Object file principles

An *object file* is the assembler's output—a binary artifact containing the translated machine code plus enough bookkeeping for the linker to combine it with other object files into a final executable. The essential question an object format has to answer is: *what information does the linker need to do its job?*. At minimum the linker needs the machine code itself (so it has something to relocate and concatenate), a list of *symbols defined* in this file (so other files can reference them), a list of *symbols used* but not defined in this file (so the linker can resolve them against other files' symbol tables), and a list of *relocation entries* telling the linker where in the code to patch addresses once the final layout is known. Every serious object format then piles on optional extras: multiple code/data sections, debug information, per-function exception-handling metadata, constructor/destructor tables, version information, link-time optimization bitcode, and so on.

Object formats have evolved in a handful of distinct lineages. *a.out* ("assembler output") was the original UNIX format, introduced with Ken Thompson's first UNIX on the PDP-11 in 1971—a flat header plus text, data, bss, symbol table, and relocations, with no notion of multiple sections. COFF (*Common Object File Format*) was introduced with AT&T UNIX System V in 1983 to add arbitrary named sections, becoming the model for every modern format to follow. ELF (*Executable and Linkable Format*), shipped with SVR4 in 1988, is now the Linux, BSD, and Solaris standard; it extended COFF with a cleaner section header table, a program header table for runtime loading, and explicit dynamic-linking slots. *Mach-O* is NEXTstep's format (derived from the Mach microkernel work at Carnegie Mellon), inherited by macOS, iOS, and the other Apple platforms. PE (*Portable Executable*) is Microsoft's UNIX-free variant of COFF, wrapped in an MS-DOS "MZ" header for compatibility and used by every Windows executable and DLL since NT. Plan 9, going against the grain, stayed with a revised *a.out*: it keeps the 1971 design's simplicity while dropping the parts that did not survive, adding only what is strictly necessary for a working toolchain. ELF won on UNIX and Linux because its design was technically cleaner than a.out's: arbitrary named sections (not just the fixed text/data/bss triple), explicit support for dynamic linking and shared libraries, and adoption as the SVR4 standard gave it an installed base the older formats could not match.

Regardless of lineage, every object format has to store four kinds of information. *Sections* divide the object

into coarse regions with different properties: `.text` (executable code), `.data` (initialized data), `.bss` (zero-initialized, takes no file space), `.rodata` (read-only data), plus debug and metadata sections that the runtime ignores. *Symbol tables* list every name the file defines or references, each tagged with its section, its offset, its size, its scope (local, global, weak), and its type (function, object, section). *Relocation entries* tell the linker "at this offset in `.text` there is a reference to symbol `foo` that needs to be patched once `foo` gets its final address", along with a relocation *type* that encodes how to do the patch—absolute 32-bit, PC-relative, upper 16 bits of a 32-bit constant for ARM, and so on, with dozens of type codes per architecture. *Debug information* lets debuggers map machine addresses back to source file and line, stack frames to local-variable names, and instructions to expression trees; the historical format was STABS (AT&T, 1980s, lines of text embedded in the symbol table), replaced by DWARF (DWARF 1 in 1992, now at version 5—a relatively compact binary tree of *debugging information entries*), with Microsoft's *CodeView* and PDB as the Windows equivalents.

Plan 9's object format is striking mainly for what it *omits*. There are no named sections: the object file contains TEXT, DATA, and BSS as fixed parts, not as a general section list the way COFF and ELF have. There are no weak symbols, no version symbols, no constructor/destructor tables, no thread-local storage section, no DWARF, no exception-handling metadata, no dynamic-loader relocations. Debug information is reduced to a compact *line-origin history* that records which source file each range of instructions came from (`Hist`, covered in Chapter 10); local-variable debug info is recovered from the compiler's symbol emission by `acid` rather than from DWARF expressions. The entire format fits in a few hundred lines of C in both `5a` and `5l`, a fraction of what any modern UNIX toolchain devotes to object-format handling. The rest of this chapter walks through the Plan 9 format concretely: what `5a` writes for each Asm5 instruction (`outcode`), how operands are serialized (`outopd`), and how the symbol table is emitted (the `ANAME` table and `h` hash). The corresponding reader side lives in the linker `5l` and is covered in LINKER book [Pad15b].

9.2 Object format

We described roughly the format of the object files at the beginning of this book in Section 2.6. We can now describe more precisely this format in Figure 9.1 since we now have a better understanding of Asm5.

The format of an object file is pretty regular, essentially a list of instructions where each instruction is encoded as follows:

- An opcode (1 byte), e.g., the value of `ASUBX`
- A conditional execution (1 byte), e.g., the value of `Always`[103b]
- A register number (1 byte), e.g., 4 for `R4`, when an instruction has 3 operands or the value of `R_NONE`[33a]
- A line number (4 bytes) encoded in an architecture independent way by having the bytes representing the lower part of the number stored first in the object file[1].
- The first operand
- The second operand

The operands themselves are encoded as:

- The kind of the operand (1 byte), e.g., the value of `D_REGX`. Note that for opcodes with 0 or 1 operand, `D_NONEX` is used to represent the null operand.
- A register number (1 byte), e.g., 5 for `R5` when the operand involves a register (as in `(R5)`) or `R_NONE`.

[1] Also known as the *little-endian* format

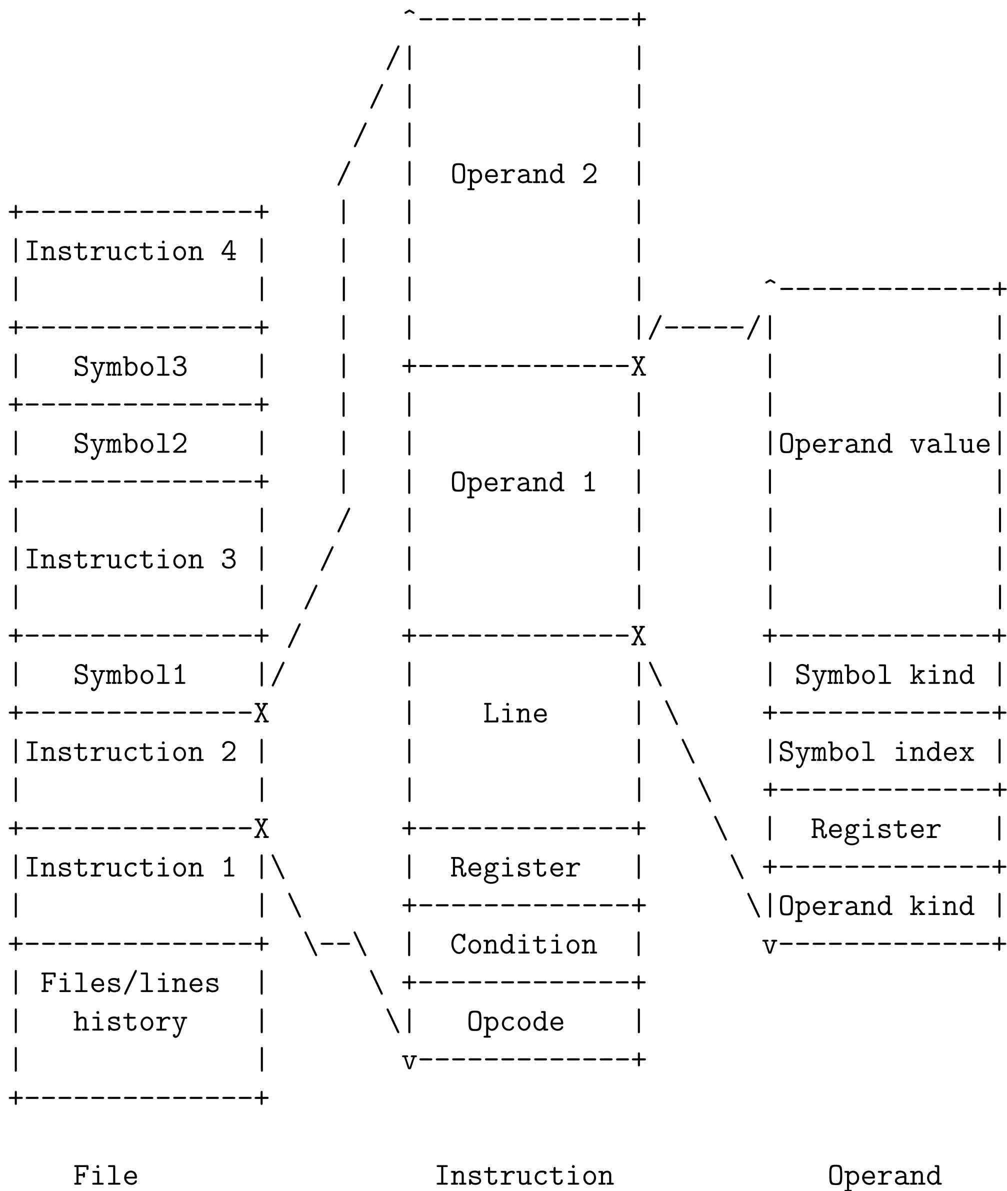


Figure 9.1: Complete format of a `.5` object file.

- An *index* (1 byte) in the *object file symbol table*, when the operand involves a symbol (as in `hello(SB)`) or 0. The encoding scheme for this symbol table will be described later in Section 9.5.
- The kind of the symbol (1 byte), e.g., the value of `N_EXTERNX`, when the operand involves a symbol or `N_NONEX`.
- The value of the operand, which has a variable size e.g., 4 bytes for the integer value of a `D_CONSTX` but 0 byte for `D_REGX` since the register number is stored already above.

A symbol reference in an operand is represented as an index in a symbol table, which saves space since the same symbol could be referenced many times. This symbol table could be stored at the beginning or end of the file. Instead, `5a` uses a rather sophisticated scheme explained in Section 9.5 where the symbol table is "spread" in the object file along with the instruction (represented by the `Symbol` cells in Figure 9.1). The beginning of an object file contains files and lines information which will be explained in Section 10.4.

9.3 Instruction output: `outcode()`

The job of `outcode()`, introduced in Section 3.4, and used many times in the grammar, is mostly to serialize one instruction (machine, pseudo, or virtual) according to the format described in the middle of Figure 9.1:

⟨*function* `outcode`*(arm)* 108⟩≡ (168b)

```
/// main -> assemble -> yyparse -> <>
void
outcode(int opcode, int scond,  Gen *g1, int reg, Gen *g2)
{
    ⟨outcode() locals 111e⟩

    ⟨outcode() adjust opcode and scond when opcode is AB 109a⟩

    if(pass == 1)
        goto out;

    ⟨outcode() st and sf computation, and possible calls to zname 112b⟩

    Bputc(&obuf, opcode);
    Bputc(&obuf, scond);
    Bputc(&obuf, reg);
    Bputc(&obuf, lineno);
    Bputc(&obuf, lineno>>8);
    Bputc(&obuf, lineno>>16);
    Bputc(&obuf, lineno>>24);
    outopd(g1, sf);
    outopd(g2, st);

out:
    if(opcode != AGLOBL && opcode != ADATA)
        pc++;
}
```

Uses `lineno` 50b, `obuf` 47a, `outopd()` 109c, `pass` 45c, and `pc` 92a.

It also increments the virtual program counter `pc`[92a] (used to resolve labels) for all instructions except the one related to the data section of the executable. Indeed, `DATA` or `GLOBL` instructions can be mixed with `TEXT` instructions; they do not have to be at the end of the assembly file (even though this is a common practice).

Note the use of `goto` in the code above where an `if` would have been arguably clearer. This is perhaps the sign that the person who wrote this code likes too much assembly.

outcode()[108] also *normalizes* the code:

⟨outcode() *adjust opcode and scond when opcode is AB* 109a⟩≡ (108)
```
/* hack to make B.NE etc. work: turn it into the corresponding conditional*/
if(opcode == AB){
    opcode = bcode[scond&0xf];
    scond = (scond & ~0xf) | Always;
}
```
Uses Always 103b and bcode-56 109b.

⟨*global* bcode*(arm)* 109b⟩≡ (168b)
```
static int bcode[] =
{
    ABEQ,
    ABNE,
    ABHS,
    ABLO,
    ABMI,
    ABPL,
    ABVS,
    ABVC,
    ABHI,
    ABLS,
    ABGE,
    ABLT,
    ABGT,
    ABLE,
    AB,
    ANOP,
};
```

The bit manipulation with scond above is because scond in addition to store the conditional execution also stores special ARM instruction bits as explained in Section 8.7.

9.4 Operand output: outopd()

Assuming a symbol index parameter symidx computed by the caller (and explained in Section 9.5), outopd() outputs an operand according to the format described in the right of Figure 9.1:

⟨*function* outopd*(arm)* 109c⟩≡ (168b)
```
/// main -> assemble -> yyparse -> outcode -> <>
void
outopd(Gen *a, int symidx)
{
    ⟨outopd() locals(arm) ??⟩

    Bputc(&obuf, a->type);
    Bputc(&obuf, a->reg);
    // idx in symbol table, 0 if no symbol involved in the operand
    Bputc(&obuf, symidx);
    // symkind of the symbol, or N_NONE
    Bputc(&obuf, a->symkind);

    switch(a->type) {
    ⟨outopd() cases(arm) 110a⟩
    default:
        print("unknown type %d\n", a->type);
        exits("arg");
```

```
    }
}
```
Uses obuf 47a.

The operand value has a variable size depending on the kind of the operand:

⟨outopd() *cases(arm)* 110a⟩≡ (109c) 125f▷
```
case D_NONE:
    break;

case D_REG:
    break;

case D_CONST:
case D_ADDR:
case D_OREG:
case D_BRANCH:
case D_SHIFT:
    l = a->offset;
    Bputc(&obuf, l);
    Bputc(&obuf, l>>8);
    Bputc(&obuf, l>>16);
    Bputc(&obuf, l>>24);
    break;

case D_SCONST:
    n = a->sval;
    for(i=0; i<NSNAME; i++) {
        Bputc(&obuf, *n);
        n++;
    }
    break;
```
Uses obuf 47a.

9.5 Object file symbol table: h and ANAME

Operands can mention symbols which are important to keep track in the object file since those symbol references must be resolved later by the linker. Because only one byte is used in the format of the operand to store the index of the symbol involved (see Section 9.2), you could think Asm5 allows only 256 different symbols in an assembly file. But, the *object file symbol table* we will describe in this section, in addition to be *spread* in the object file, is also a *circular* table with 50 elements. This clever scheme allows any number of symbols in an object file as we will see.

The global h below mimics in memory this circular table:

⟨*global* h 110b⟩≡ (160a)
```
// array<Htab>
struct Htab h[NSYM];
```
Uses Htab 110d.

⟨*constant* NSYM 110c⟩≡ (154a)
```
#define NSYM 50
```

⟨*struct* Htab 110d⟩≡ (156b)
```
struct Htab
{
    // option<ref<Sym>>
    Sym*    sym;
```

```
    //enum<Sym_kind>
    short   symkind;
};
```

The object file symbol table `h` is different from the symbol table `hash`[39a] I described before. Indeed `hash` is an hash table which keeps track of macros, predefined identifiers, labels, and symbols. The object file symbol table stores only symbols (and labels but only for debugging purpose) and is just an array.

The first entry in `h` is unused since 0 represents the absence of symbol in the operand format. The next free entry is stored in the following global:

⟨*global* `symcounter` 111a⟩≡ (160a)

```
int symcounter;
```

⟨`pinit()` *initializations* 111b⟩+≡ (50a) ◁92b 132g▷

```
symcounter = 1;
for(i=0; i<NSYM; i++) {
    h[i].symkind = 0; // N_NONE
    h[i].sym = S;
}
```

Uses `S` 39d, `h` 110b, and `symcounter` 111a.

As we will output instructions in `outcode()`[108], new symbols will be referenced in operands and new entries in `h` will be created as we will see soon. To output in the object file a new entry, the following function is used:

⟨*function* `zname`*(arm)* 111c⟩≡ (168b)

```
/// outcode -> <>
void
zname(char *n, int symkind, int symidx)
{

    Bputc(&obuf, ANAME);
    Bputc(&obuf, symkind); /* type */
    Bputc(&obuf, symidx); /* sym */
    while(*n) {
        Bputc(&obuf, *n);
        n++;
    }
    Bputc(&obuf, '\0');
}
```

Uses `obuf` 47a.

A new pseudo opcode is used:

⟨`Opcode` *cases, pseudo opcodes* 111d⟩+≡ (31) ◁102d 119c▷

```
ANAME,
```

This constant must be part of the `Opcode`[31] enum so that it is different from any other opcode. This makes the format of an object file unambiguous. Indeed, to read an object file, we can just read one byte and decide if the further bytes represent an instruction or an element of the symbol table.

We can finally see the code in `outcode()` which maintains `h`, manages the symbol indexes, and calls `zname()`[111c] when a new entry is needed. The locals below are set by this code. `sf` and `st` are passed to `outopd()`[109c] (look at the code of `outcode()`):

⟨`outcode()` *locals* 111e⟩≡ (108)

```
// symbol from, index in h[]
int sf;
// symbol to, index in h[]
int st;
int oldsymcounter;
```

An additional field in `Sym`[156b] is needed to store the index of symbols already present in `h`:

⟨`Sym` *identifier fields* 112a⟩≡ (38)
```
// index in h when the Sym is a symbol, 0 otherwise
int symidx;
```

The code below to compute `sf` and `st` is rather subtle:

⟨`outcode()` *st and sf computation, and possible calls to zname* 112b⟩≡ (108)
```
jackpot:
oldsymcounter = symcounter;
sf = symidx_of_symopt(g1->sym, g1->symkind);
st = symidx_of_symopt(g2->sym, g2->symkind);
⟨outcode() if jackpot condition goto jackpot 112e⟩
```
Uses `symcounter` 111a and `symidx_of_symopt()` 112c.

⟨*function* `symidx_of_symopt` 112c⟩≡ (168b)
```
int
symidx_of_symopt(Sym *sym, int symkind)
{
    int idx = 0;

    if(sym != S) {
        idx = sym->symidx;
        ⟨symidx_of_symopt() sanity check idx 112d⟩

        // already generated an ANAME for this symbol reference?
        if((h[idx].symkind != symkind || h[idx].sym != sym)) {
            sym->symidx = symcounter;
            h[symcounter].sym = sym;
            h[symcounter].symkind = symkind;
            idx = symcounter;
            zname(sym->name, symkind, symcounter);

            symcounter++;
            if(symcounter >= NSYM)
                // circular array
                symcounter = 1;
        }
    }
    return idx;
}
```
Uses `S` 39d, `h` 110b, `symcounter` 111a, and `zname()` 111c.

⟨`symidx_of_symopt()` *sanity check idx* 112d⟩≡ (112c)
```
if(idx < 0 || idx >= NSYM)
    idx = 0;
```

There is a rather complex condition which can force us to recompute `sf` and `st`:

⟨`outcode()` *if jackpot condition goto jackpot* 112e⟩≡ (112b)
```
if (sf == st && sf != 0 && symcounter != oldsymcounter)
   goto jackpot;
```
Uses `symcounter` 111a.

This situation can happen for instance when the first operand involves a symbol `foo` with an index of 10 (`sf == 10`) and when the second operand involves another symbol `bar` not yet seen. Because of the circular nature of `h`, `symcounter` could have already done one full round and be back to the value of 10. This means the next entry is 10 and `h[10].sym` will be overwritten to point now to the symbol for `bar`. In this case `sf == st == 10` but `sf` now points to a wrong entry. In that case we want to recompute `sf` hence the `goto`. The extra condition about `oldsymcounter` is to avoid an infinite loop in `outcode()` with code such as `MOVW bar(SB), bar(SB)`.

The circular buffer `h` is easier to picture if I walk through a small source that deliberately exercises the wrap-around. Imagine `NSYM` was just 4 instead of 50 and the assembly file was:

```
    MOVW    a(SB), R0
    MOVW    b(SB), R0
    MOVW    c(SB), R0
    MOVW    a(SB), R0           ; "a" has been evicted by now
```

The time-lapse of `h` and `symcounter` is then:

```
  step        h[0]   h[1]   h[2]   h[3]   symcounter    emits ANAME?
  ----        ----   ----   ----   ----   ----------    ------------
  init          -      -      -      -         1              -
  a (sf=1)      -      a      -      -         2          ANAME "a" idx=1
  b (st=2)      -      a      b      -         3          ANAME "b" idx=2
  c (st=3)      -      a      b      c         1  wrap    ANAME "c" idx=3
  a (sf=1)      -      a*     b      c         2          ANAME "a" idx=1 (!)
```

The last row shows the trick: the first time `a` was seen, `symidx_of_symopt()`[112c] set `a->symidx = 1` and `h[1].sym = a`. A few instructions later, the same slot was needed for `a` again, so `5a` must re-emit an `ANAME` entry with `symidx = 1` to reassign the meaning of index 1 in the linker. The linker rebuilds its own copy of `h` on the fly while reading the object file, so whenever it encounters an operand with `symidx = 1`, it uses whatever symbol currently lives in slot 1 of its reconstruction.

The `jackpot` condition guards one extra wrinkle. Suppose an instruction has the same fresh symbol in both operands, as in `MOVW foo(SB), foo(SB)` with `foo` not yet seen. The first call to `symidx_of_symopt()` for `g1` writes `foo` into slot `symcounter` and increments `symcounter`; the second call for `g2` sees `foo->symidx` already set to the same value and returns immediately. So `sf == st` and we are fine. But now change the example to `MOVW bar(SB), foo(SB)` with a tiny `NSYM` and a buffer that has gone all the way around: `bar` lands at index 10 (say), `foo` also lands at index 10 because `symcounter` has just wrapped and re-overwritten the slot `bar` used one nanosecond ago. Now `sf == st == 10` but `sf` points at `foo`, not `bar`. The test `sf == st \&\& sf != 0 \&\& symcounter != oldsymcounter` detects exactly this case and jumps back to `jackpot` to recompute `sf`. The `symcounter != oldsymcounter` guard stops the degenerate `MOVW bar(SB), bar(SB)` case from looping forever, since on the second attempt no new `ANAME` is emitted and the counters stay equal.

Chapter 10

Debugging Support

The Plan 9 debugger `db`, which we will describe in the Debugger book [Pad16d], 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 machine code comes from. It also knows the name of the function containing this piece of machine code, the names of the parameters of this function, as well as the name of the caller to this function. Finally it knows 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 as we have seen in the previous chapter. This is why Asm5 programmers write code like `MOVW count+4(FP), R1` even though `count` is not used to generate any machine code. Indeed, this symbol is actually kept in the object file and then transmitted in the executable and can then be used by debuggers and tracers. Labels are resolved during parsing by `5a` but the names of those labels are also kept in the object file symbol table. Their references in operands are kept too, which again helps to debug programs.

In this chapter we will focus on the file and line information as the information about symbols has already been described in the previous chapter.

10.1 Line origin history: `Hist`

Because `5a` is a macro-assembler, the file and line origin of an instruction in the object file are not necessarily the assembly filename passed on the command line to `5a` and a line number in this file. Indeed, this file could have included via `#include` other files containing also assembly instructions. Moreover, some assembly files are derived from C programs in which case one would rather have the location metadata refers to line numbers in the original C programs (thanks to the `#line` directive we saw in Section 7.3) than line numbers in the generated assembly programs. So, recording only a line number in the format of an instruction as described in Section 9.2 seems insufficient; a filename should also be recorded.

The global `lineno`[50b] I introduced before, and which is used by `outcode()`[108] to store the line number of an instruction in the object file, is incremented after each newlines by the lexer (see for instance Section 6.3). But, this is done while pre-processing the file which means `lineno` is actually more a *global line number* than a line number in one file. What we need then is a way to go from a global line number to a filename and a *local line number*.

The following structure will help to keep track of the #include and #line directives used while pre-processing the input file. This will allow us then to compute the file and local line number of a global line number.

⟨*struct* Hist 115a⟩≡ (156b)
```
struct  Hist
{
    // option<ref_own<string> (None = nil = a ''pop'')
    char*   filename;

    // global line of this Hist
    long    global_line;
    // 0 for #include, +n for #line, -1 for #pragma lib (ugly)
    long    local_line;

    // Extra
    ⟨Hist extra fields 115c⟩
};
```

The global hist below maintains a list of those Hist[156b]. The first element of this list will be the input assembly file of 5a passed on the command line.

⟨*global* hist 115b⟩≡ (160a)
```
// list<ref_own<Hist>> (next = Hist.link)
Hist* hist;
```

⟨Hist *extra fields* 115c⟩≡ (115a)
```
// list<ref<Hist>> (from = hist)
Hist*   link;
```

⟨*constant* H 115d⟩≡ (156b)
```
#define H   ((Hist*)nil)
```

We will need also a fast access to the tail of the list:

⟨*global* ehist 115e⟩≡ (160a)
```
// ref<Hist> (end from = hist)
Hist* ehist;
```

10.2 Recording history: linehist()

The only function modifying hist[115b] is linehist(). It is called notably by newfile()[52d] which itself is called at the beginning in pinit()[50a] with the name of the input assembly file and later for each included file:

⟨newfile() *call linehist* 115f⟩≡ (52d)
```
linehist(s, 0);
```
Uses linehist() 115g.

⟨*function* linehist 115g⟩≡ (161b)
```
/// (pinit | macinc -> newfile) | (GETC -> filbuf) | maclin -> <>
void
linehist(char *f, int local_line)
{
    Hist *h;

    ⟨linehist() debug 146d⟩

    h = alloc(sizeof(Hist));
    h->filename = f;
    h->global_line = lineno;
    h->local_line = local_line;
```

```
    //add_list(hist, ehist, h)
    h->link = H;
    if(ehist == H) {
        hist = h;
        ehist = h;
        return;
    }
    ehist->link = h;
    ehist = h;
}
```

Uses `H` 115d, `alloc()` 151b, `ehist` 115e, `hist` 115b, and `lineno` 50b.

It is also called for `#line` directives such as `#line 10 "foo"`:

⟨`maclin()` *call linehist* 116a⟩≡ (78a)

```
linehist(cp, n);
```

Uses `linehist()` 115g.

Note that the included files form a tree. We use a list data structure though for `hist` which seems insufficient. But, the special value `nil` in a `Hist`[156b] encodes the end of a file which is enough to reconstruct the tree:

⟨`filbuf()` *when close file, call linehist* 116b⟩≡ (54a)

```
linehist(nil, 0);
```

Uses `linehist()` 115g.

Figure 10.1 illustrates the evolution of the global `lineno`[50b] on the content of `/tests/cpp/foo.s` (which includes other files) as well as the value of `hist` and `ehist`[115e] after having pre-processed the entire file.

One of the option of `5a` described in Section A.1 helps to debug the debugging support code itself by logging the value of `lineno` and `Hist` elements in the calls to `linehist()`[115g]:

```
$ cd /tests/cpp
$ 5a -f foo.s
   1: foo.s
   4: foo.h
   5: foo1.h
   6: <pop>
   8: foo2.h
   9: <pop>
  11: <pop>
  13: bar.s
  15: <pop>
  18: foobar.c (#line 10)
  20: <pop>
...
```

10.3 Displaying history: `prfile()`

By keeping the full history of the `#include` directives, `5a` can also give better error message such as:

```
$ 5a foo.s
foo.s:5 bar.s:2 redeclaration of lbl1
```

```
                                              hist
                                       +-------------------+
     foo.s--------------------------+ >|"foo.s", G1, L0     |
 1   |L1                            |  +-------------------+
 2   |L2                            |
 3   |L3 #include "foo.h"           |  +-------------------+
     |   foo.h------------------+ - + >|"foo.h", G4, L0     |
 4   |   |L1 #include "foo1.h"  |   |  +-------------------+
     |   |   foo1.h----------+ -|- -|->|"foo1.h", G5, L0    |
 5   |   |   |L1            |  |   |  +-------------------+
     |   |   +---------------+ -|- -|->|nil, G5, L0        |
 6   |   |L2                    |   |  +-------------------+
 7   |   |L3 #include "foo2.h"  |   |  +-------------------+
     |   |   foo2.h----------+ -|- -|->|"foo2.h", G8, L0    |
 8   |   |   |L1            |  |   |  +-------------------+
     |   |   +---------------+ -|- -|->|nil, G8, L0        |
 9   |   |L4                    |   |  +-------------------+
 10  |   |L5                    |   |  +-------------------+
     |   +----------------------+ - + >|nil, G10, L0       |
 11  |L4                            |  +-------------------+
 12  |L5 #include "bar.s"           |  +-------------------+
     |   bar.s------------------+ - + >|"bar.s", G13, L0    |
 13  |   |L1                    |   |  +-------------------+
 14  |   |L2                    |   |  +-------------------+
     |   +----------------------+ - + >|nil, G14, L0       |
 15  | L6                           |  +-------------------+
 16  | L7                           |  +-------------------+
 17  | L8#line 10 "foobar.c"        |  |"foobar.c", G18, L10|
 18  | L9                           |  +-------------------+
 19  |L10                           |  +-------------------+
     +------------------------------+ >|nil, G19, L0       |
                                       +-------------------+
lineno         foo.s and                     ehist
(global)     included files
```

Figure 10.1: Values of `lineno` and `hist` for `/tests/cpp/foo.s`.

This means the error is in the file `bar.s` at line 2 and this file was actually included from `foo.s`. We are now ready to understand `prfile()` which is used when reporting error in `5a`. It converts a global line number `l` into a series of filenames and line numbers representing the history of this global line.

The size of this series is limited to 20 elements:

⟨*constant* `HISTSZ` 118a⟩≡ (156b)
```
#define HISTSZ      20
```

For instance with the data on Figure 10.1, `prfile(14)` should print:

```
foo.s:5 bar.s:2
```

⟨*function* `prfile` 118b⟩≡ (161b)
```
/// yyerror -> <>
void
prfile(long l)
{
    Hist a[HISTSZ];
    int n = 0;
    Hist *h;
    int i;
    long d;

    ⟨prfile() compute a and n 118c⟩
    if(n > HISTSZ)
        n = HISTSZ;
    for(i=0; i<n; i++)
        print("%s:%ld ", a[i].filename,
                         (long)(l - a[i].global_line + a[i].local_line + 1));
}
```
Uses `HISTSZ` 118a.

The `+1` above is to have line numbers starting at 1.

⟨`prfile()` *compute a and n* 118c⟩≡ (118b)
```
for(h = hist; h != H; h = h->link) {
    if(l >= h->global_line) {
        if(h->filename) {
            // a #include
            if(h->local_line == 0) {
                if(n >= 0 && n < HISTSZ)
                    a[n] = *h;
                n++;
            }
            ⟨prfile() compute a and n, when line directive 119a⟩
        }
        // a pop
        else {
            n--;
            ⟨prfile() compute a and n, when pop, adjust parents 118d⟩
        }
    }
}
```
Uses `H` 115d, `HISTSZ` 118a, and `hist` 115b.

⟨`prfile()` *compute a and n, when pop, adjust parents* 118d⟩≡ (118c)
```
if(n >= 0 && n < HISTSZ) {
    d = h->global_line - a[n].global_line;
    for(i=0; i<n; i++)
        a[i].global_line += d;
}
```
Uses `HISTSZ` 118a.

⟨prfile() *compute a and n, when line directive* 119a⟩≡ (118c)
```
// a #line
else {
    if(n > 0 && n < HISTSZ)
        // previous was a #include
        if(a[n-1].local_line == 0) {
            a[n] = *h;
            n++;
        } else
            a[n-1] = *h; // overwrite previous #line
}
```
Uses HISTSZ 118a.

10.4 Saving history: outhist() and AHISTORY

We can now describe the only missing piece in the object file format in Figure 9.1: the `files/lines history` at the beginning of the object file. The function below is called by `assemble()`[45d] before the many calls to `outcode()`[108] and essentially serializes `hist`[115b]. To do so it is reusing (some would say abusing) the instruction format and the symbol table format to respectively store the global/local line numbers in a `Hist`[156b] and its filename:

⟨*function* outhist*(arm)* 119b⟩≡ (168b)
```
/// main -> assemble -> <> (at beginning of pass 2)
void
outhist(void)
{
    Gen g;
    Hist *h;
    char *p;
    ⟨outhist() locals(arm) ??⟩

    g = nullgen;
    for(h = hist; h != H; h = h->link) {
        p = h->filename;

        ⟨outhist() adjust p and op if p is relative filename 120d⟩
        ⟨outhist() output each path component as an ANAME 120a⟩

        Bputc(&obuf, AHISTORY);
        Bputc(&obuf, Always);
        Bputc(&obuf, 0); // reg, but could be R_NONE too
        Bputc(&obuf, h->global_line);
        Bputc(&obuf, h->global_line>>8);
        Bputc(&obuf, h->global_line>>16);
        Bputc(&obuf, h->global_line>>24);
        outopd(&nullgen, 0);
        g.offset = h->local_line;
        outopd(&g, 0);
    }
}
```
Uses Always 103b, H 115d, hist 115b, nullgen 34a, obuf 47a, and outopd() 109c.

Again, just like for `ANAMEX`, a new pseudo opcode is used, different from any other opcode, so that the linker can just read one byte in the object file and decide if the further bytes represent an instruction or an element of the symbol table or a line history element:

⟨Opcode *cases, pseudo opcodes* 119c⟩+≡ (31) ◁111d 131b▷
```
AHISTORY,
```

The `Hist.filenameX` is stored in the object file symbol table with `ANAMEX`. The full filename is actually split according to its directory components and each component is stored separately with its own entry in the object file symbol table.

⟨`outhist()` *output each path component as an ANAME* 120a⟩≡ (119b)
```
// =~ split("/", p) ...
while(p) {
    q = strchr(p, '/');
    if(q) {
        n = q-p;
        if(n == 0){
            n = 1; /* leading "/" */
            *p = '/'; // redundant?
        }
        q++;
    } else {
        n = strlen(p);
        q = nil;
    }

    if(n) {
        Bputc(&obuf, ANAME);
        Bputc(&obuf, N_FILE); /* type */ // symkind
        Bputc(&obuf, 1);     /* sym */  // symidx
        Bputc(&obuf, '<');
        Bwrite(&obuf, p, n);
        Bputc(&obuf, '\0');
    }
    p = q;
    ⟨outhist() adjust p and op if p was a relative filename 120e⟩
}
```
Uses `obuf` 47a.

Note the use of the same symbol index above, `1`, which normally is incremented after each symbol is added in Section 9.5. This is because the code in the linker which loads the object file treats specially those entries in the object file symbol table. A new kind of symbol is also used:

⟨`Sym_kind` *cases* 120b⟩+≡ (42d) ◁99b 131i▷
```
N_FILE,
```

The code below is to convert relative filenames into absolute paths. Indeed, The object file stores absolute filenames, which is better for `db` so that wherever `db` is run from, it can find the source files referenced in the executable. Remember that `pathname`[45a] below is the current working directory set in `cinit()`[44c].

⟨`outhist()` *locals(arm)* 120c⟩+≡ (119b) ◁??
```
char *op;
```

⟨`outhist()` *adjust p and op if p is relative filename* 120d⟩≡ (119b)
```
if(p && p[0] != '/' && h->local_line == 0 && pathname && pathname[0] == '/') {
    op = p; // save p
    p = pathname; // start with cwd
} else {
    op = nil; // start directly with p
}
```
Uses `pathname` 45a.

⟨`outhist()` *adjust p and op if p was a relative filename* 120e⟩≡ (120a)
```
if(p == nil && op) {
    p = op;
    op = nil;
}
```

Chapter 11

Advanced Topics TODO

I have described in the previous chapters all the main features of Asm5 and `5a`. You can write many assembly programs with only the instructions I described until now, and you can have a working assembler with only the code I presented before. In this chapter, I will describe features of Asm5 and `5a` that I did not present before to simplify the explanations. Those features are used more rarely by the assembly programmer.

11.1 Other assembly language features

11.1.1 Constant expressions

⟨*type* `declarations`*(arm)* 121a⟩+≡ (81a) ◁102g 125a▷

```
%type   <lval>  con expr
```

⟨*con rule* 121b⟩+≡ (121d) ◁94d 122b▷

```
| '(' expr ')' { $$ = $2; }
```

⟨*priority and associativity declarations* 121c⟩≡ (81a)

```
%left   '|'
%left   '^'
%left   '&'
%left   '<' '>'
%left   '+' '-'
%left   '*' '/' '%'
```

⟨*constant expression rules* 121d⟩≡ (? 0—1)

  ⟨*con rule* 94d⟩
  ⟨*expr rule* 121e⟩

⟨*expr rule* 121e⟩≡ (121d)

```
expr:
  con
| expr '+' expr     { $$ = $1 + $3; }
| expr '-' expr     { $$ = $1 - $3; }
| expr '*' expr     { $$ = $1 * $3; }
| expr '/' expr     { $$ = $1 / $3; }
| expr '%' expr     { $$ = $1 % $3; }

| expr '<' '<' expr { $$ = $1 << $4; }
| expr '>' '>' expr { $$ = $1 >> $4; }

| expr '&' expr     { $$ = $1 & $3; }
| expr '^' expr     { $$ = $1 ^ $3; }
| expr '|' expr     { $$ = $1 | $3; }
```

Notice that the type of `expr` is just `lval`, a `long`. So the actions above are not building an expression tree; each reduce *immediately evaluates* the sub-expression and propagates a plain integer. Asm5 therefore has no notion of a symbolic constant expression in its AST—by the time an operand reaches `outcode()`[108], the expression has already collapsed to a single number. This fold-at-reduce is easiest to see on a concrete input. Consider `$(4+3)*8` which appears inside a `MOVW` operand: Yacc drives the reductions bottom-up, and each action writes back one `long`:

```
source:                   $ ( 4 + 3 ) * 8

tokens:                     '(' LCONST '+' LCONST ')' '*' LCONST
                                 4           3              8

stack after each reduce:
  1. con: LCONST            (      4                    )          -- $$ = 4
  2. expr: con              expr(4)                                -- $$ = 4
  3. con: LCONST                   con(3)                          -- $$ = 3
  4. expr: con                     expr(3)                         -- $$ = 3
  5. expr: expr '+' expr    expr(7)                                -- $$ = 4+3 = 7
  6. con: '(' expr ')'      con(7)                                 -- $$ = 7
  7. expr: con              expr(7)                                -- $$ = 7
  8. con: LCONST                              con(8)               -- $$ = 8
  9. expr: con                                expr(8)              -- $$ = 8
 10. expr: expr '*' expr    expr(56)                               -- $$ = 7*8 = 56

action in imm rule:       $$.type = D_CONST; $$.offset = 56;
```

Every intermediate `$$` is a `long` that lives in Yacc's value stack; no `Expr` node is ever allocated. Compared to `5c` or `5l`, which do build expression trees for much the same operators, `5a` gets away with folding everything inline because Asm5 has no variables that could be unknown at parse time—`LVAR` symbolic constants are resolved through `Sym.valueX`, which is itself a `long`. The concrete syntax `$bits>>24` compiles to a single `D_CONST` operand in exactly the same way as `$0x01000000` would; the linker never sees a trace of the shift.

11.1.2 Symbolic constants

⟨*line rule(arm)* 122a⟩+≡ (81b) ◁92c

```
  | LNAME '=' expr ';'
   {
    $1->type = LVAR;
    $1->value = $3;
   }
  | LVAR '=' expr ';'
   {
    if($1->value != $3)
     yyerror("redeclaration of %s", $1->name);
    $1->value = $3;
   }
```

⟨*con rule* 122b⟩+≡ (121d) ◁121b

```
  | LVAR        { $$ = $1->value; }
```

11.2 Other instructions and registers

11.2.1 Floating-point numbers

⟨*constant* FPCHIP *(arm)* 123a⟩≡ (167a)
```
#define FPCHIP      true
```

⟨*struct* ieee 123b⟩≡ (154a)
```
/*
 * this is the simulated IEEE floating point
 */
struct ieee
{
    long l; /* contains ls-man 0xffffffff */
    long h; /* contains sign 0x80000000
                exp  0x7ff00000
                ms-man 0x000fffff */
};
```

⟨Opcode *cases, float mov opcodes* 123c⟩≡ (31)
```
AMOVWD,
AMOVWF,
AMOVDW,
AMOVFW,
AMOVFD,
AMOVDF,
AMOVF,
AMOVD,
```

⟨itab *elements* 123d⟩+≡ (37c) ◁103g 123f▷
```
"MOVD",  LMOV, AMOVD,
"MOVDF", LMOV, AMOVDF,
"MOVDW", LMOV, AMOVDW,
"MOVF",  LMOV, AMOVF,
"MOVFD", LMOV, AMOVFD,
"MOVFW", LMOV, AMOVFW,
"MOVWD", LMOV, AMOVWD,
"MOVWF", LMOV, AMOVWF,
```
Uses LMOV.

⟨Opcode *cases, float arithmetic opcodes* 123e⟩≡ (31)
```
ACMPF,
ACMPD,
AADDF,
AADDD,
ASUBF,
ASUBD,
AMULF,
AMULD,
ADIVF,
ADIVD,
ASQRTF,
ASQRTD,
```

⟨itab *elements* 123f⟩+≡ (37c) ◁123d 124a▷
```
"CMPF",  LCMPFLOAT, ACMPF,
"CMPD",  LCMPFLOAT, ACMPD,
"ADDF",  LARITHFLOAT, AADDF,
"ADDD",  LARITHFLOAT, AADDD,
"SUBF",  LARITHFLOAT, ASUBF,
```

```
  "SUBD",  LARITHFLOAT, ASUBD,
  "MULF",  LARITHFLOAT, AMULF,
  "MULD",  LARITHFLOAT, AMULD,
  "DIVF",  LARITHFLOAT, ADIVF,
  "DIVD",  LARITHFLOAT, ADIVD,
  "SQRTF", LSQRTFLOAT, ASQRTF,
  "SQRTD", LSQRTFLOAT, ASQRTD,
```

Uses LARITHFLOAT, LCMPFLOAT, and LSQRTFLOAT.

⟨itab *elements* 124a⟩+≡ (37c) ◁123f 124b▷

```
  "F",  LF, 0,

  "F0",  LFREG, 0,
  "F1",  LFREG, 1,
  "F2",  LFREG, 2,
  "F3",  LFREG, 3,
  "F4",  LFREG, 4,
  "F5",  LFREG, 5,
  "F6",  LFREG, 6,
  "F7",  LFREG, 7,
  "F8",  LFREG, 8,
  "F9",  LFREG, 9,
  "F10",  LFREG, 10,
  "F11",  LFREG, 11,
  "F12",  LFREG, 12,
  "F13",  LFREG, 13,
  "F14",  LFREG, 14,
  "F15",  LFREG, 15,
```

Uses LF and LFREG.

⟨itab *elements* 124b⟩+≡ (37c) ◁124a 126d▷

```
  "FPSR",  LFCR, 0,
  "FPCR",  LFCR, 1,
```

Uses LFCR.

⟨*enum* Fregister*(arm)* 124c⟩≡ (154c)

```
  enum Fregister {
      FREGRET = 0,
      /* compiler allocates register variables F0 up */
      FREGEXT = 7,
      /* compiler allocates external registers F7 down */
      FREGTMP = 15, // ??

      NFREG = 8,
  };
```

⟨*inst rule(arm)* 124d⟩+≡ (81b) ◁102f 127a▷

```
  /*
   * floating-point coprocessor
   */
  | LARITHFLOAT cond frcon ',' freg { outcode($1, $2, &$3, R_NONE, &$5); }
  | LARITHFLOAT cond frcon ',' LFREG ',' freg { outcode($1, $2, &$3, $5, &$7); }
  | LCMPFLOAT  cond freg ',' freg { outcode($1, $2, &$3, $5.reg, &nullgen); }
  | LSQRTFLOAT cond freg ',' freg { outcode($1, $2, &$3, R_NONE, &$5); }
```

⟨Operand_kind *cases* 124e⟩+≡ (33c) ◁99g 127b▷

```
  D_FREG,
  D_FPCR,
```

⟨*type* declarations*(arm)* 125a⟩+≡ (81a) ◁121a 127g▷
```
%type <genval> freg fcon frcon
```

⟨*more gen rule* 125b⟩+≡ (93b) ◁98d 125e▷
```
| freg
```

⟨*ximm rule* 125c⟩+≡ (? 0—1) ◁98e
```
| fcon
```

⟨*float rules* 125d⟩≡ (? 0—1)
```
freg:
  LFREG
 {
  $$ = nullgen;
  $$.type = D_FREG;
  $$.reg = $1;
 }
| LF '(' con ')'
 {
  $$ = nullgen;
  $$.type = D_FREG;
  $$.reg = $3;
 }

fcon:
 '$' LFCONST
 {
  $$ = nullgen;
  $$.type = D_FCONST;
  $$.dval = $2;
 }
| '$' '-' LFCONST
 {
  $$ = nullgen;
  $$.type = D_FCONST;
  $$.dval = -$3;
 }

frcon:
  freg
| fcon
```

⟨*more gen rule* 125e⟩+≡ (93b) ◁125b 129f▷
```
| LFCR
 {
  $$ = nullgen;
  $$.type = D_FPCR;
  $$.reg = $1;
 }
```

⟨outopd() *cases(arm)* 125f⟩+≡ (109c) ◁110a 126a▷
```
case D_FREG:
case D_FPCR:
    break;
```

⟨outopd() *cases(arm)* 126a⟩+≡ (109c) ◁125f 127c▷

```
  case D_FCONST:
      ieeedtod(&e, a->dval);
      Bputc(&obuf, e.l);
      Bputc(&obuf, e.l>>8);
      Bputc(&obuf, e.l>>16);
      Bputc(&obuf, e.l>>24);
      Bputc(&obuf, e.h);
      Bputc(&obuf, e.h>>8);
      Bputc(&obuf, e.h>>16);
      Bputc(&obuf, e.h>>24);
      break;
```

Uses ieeedtod() 126b and obuf 47a.

⟨*function* ieeedtod 126b⟩≡ (161a)

```
  /// ?? -> <>
  void
  ieeedtod(Ieee *ieee, double native)
  {
      double fr, ho, f;
      int exp;

      if(native < 0) {
          ieeedtod(ieee, -native);
          ieee->h |= 0x80000000L;
          return;
      }
      if(native == 0) {
          ieee->l = 0;
          ieee->h = 0;
          return;
      }
      fr = frexp(native, &exp);
      f = 2097152L;  /* shouldnt use fp constants here */
      fr = modf(fr*f, &ho);
      ieee->h = ho;
      ieee->h &= 0xfffffL;
      ieee->h |= (exp+1022L) << 20;
      f = 65536L;
      fr = modf(fr*f, &ho);
      ieee->l = ho;
      ieee->l <<= 16;
      ieee->l |= (long)(fr*f);
  }
```

Uses ieeedtod() 126b.

11.2.2 Multiplication and accumulation

⟨Opcode *cases, mul/div/mod opcodes* 126c⟩+≡ (31) ◁84b 127d▷

```
  AMULA,
```

⟨itab *elements* 126d⟩+≡ (37c) ◁124b 127e▷

```
  "MULA",  LMULA, AMULA,
```

Uses LMULA.

⟨*inst rule(arm)* 127a⟩+≡ (81b) ◁124d 127f▷
```
/*
 * MULA hi,lo,r1,r2
 */
| LMULA cond reg ',' reg ',' reg ',' regi
 {
  $7.type = D_REGREG;
  $7.offset = $9;
  outcode($1, $2, &$3, $5.reg, &$7);
 }
```

⟨Operand_kind *cases* 127b⟩+≡ (33c) ◁124e 129d▷
```
D_REGREG,
```

⟨outopd() *cases(arm)* 127c⟩+≡ (109c) ◁126a 129e▷
```
case D_REGREG:
    Bputc(&obuf, a->offset);
    break;
```
Uses obuf 47a.

11.2.3 64-bits multiplication

⟨Opcode *cases, mul/div/mod opcodes* 127d⟩+≡ (31) ◁126c 131c▷
```
AMULL,
AMULAL,
AMULLU,
AMULALU,
```

⟨itab *elements* 127e⟩+≡ (37c) ◁126d 128a▷
```
"MULL",  LMULL, AMULL,
"MULAL", LMULL, AMULAL,
"MULLU", LMULL, AMULLU,
"MULALU", LMULL, AMULALU,
```
Uses LMULL.

⟨*inst rule(arm)* 127f⟩+≡ (81b) ◁127a 128b▷
```
/*
 * MULL hi,lo,r1,r2
 */
| LMULL cond reg ',' reg ',' regreg { outcode($1, $2, &$3, $5.reg, &$7); }
```

⟨*type* declarations*(arm)* 127g⟩+≡ (81a) ◁125a 128c▷
```
%type   <genval>   regreg
```

⟨*operand rules(arm)* 127h⟩+≡ (81b) ◁??
```
/* for MULL */
regreg:
 '(' regi ',' regi ')'
 {
  $$ = nullgen;
  $$.type = D_REGREG;
  $$.reg = $2;
  $$.offset = $4;
 }
```

11.2.4 Moving multiple registers at the same time

⟨Opcode *cases, mov opcodes* 127i⟩+≡ (31) ◁86a
```
AMOVM,
```

⟨itab *elements* 128a⟩+≡ (37c) ◁127e 128e▷
```
  "MOVM",  LMOVM, AMOVM,
```
Uses LMOVM.

⟨*inst rule(arm)* 128b⟩+≡ (81b) ◁127f 130a▷
```
  /*
   * MOVM
   */
  | LMOVM cond ioreg ',' '[' reglist ']'
   {
    Gen g;

    g = nullgen;
    g.type = D_CONST;
    g.offset = $6;
    outcode($1, $2, &$3, R_NONE, &g);
   }
  | LMOVM cond '[' reglist ']' ',' ioreg
   {
    Gen g;

    g = nullgen;
    g.type = D_CONST;
    g.offset = $4;
    outcode($1, $2, &g, R_NONE, &$7);
   }
```

⟨*type* declarations*(arm)* 128c⟩+≡ (81a) ◁127g 130b▷
```
  %type   <lval>  reglist
```

⟨*reglist rule* 128d⟩≡ (? 0—1)
```
  reglist:
    regi          { $$ = 1 << $1; }
  | regi '-' regi
   {
    int i;
    $$=0;
    for(i=$1; i<=$3; i++)
        $$ |= 1<<i;
    for(i=$3; i<=$1; i++)
        $$ |= 1<<i;
   }
  | regi ',' reglist { $$ = (1<<$1) | $3; }
```

⟨itab *elements* 128e⟩+≡ (37c) ◁128a 128g▷
```
  ".U",  LS, C_UBIT,
```
Uses LS.

⟨*constant* C_UBIT*(arm)* 128f⟩≡ (154c)
```
  #define C_UBIT  (1<<7)  /* up bit */
```

⟨itab *elements* 128g⟩+≡ (37c) ◁128e 129a▷
```
  ".IB",  LS, C_PBIT|C_UBIT,
  ".IA",  LS, C_UBIT,
  ".DB",  LS, C_PBIT,
  ".DA",  LS, 0,
```
Uses LS.

⟨itab *elements* 129a⟩+≡ (37c) ◁128g 129b▷
```
  ".PW",  LS, C_WBIT|C_PBIT,
  ".WP",  LS, C_WBIT|C_PBIT,
```
Uses LS.

⟨itab *elements* 129b⟩+≡ (37c) ◁129a 129c▷
```
  ".IBW",  LS, C_WBIT|C_PBIT|C_UBIT,
  ".IAW",  LS, C_WBIT|C_UBIT,
  ".DBW",  LS, C_WBIT|C_PBIT,
  ".DAW",  LS, C_WBIT,
```
Uses LS.

11.2.5 Program status register

⟨itab *elements* 129c⟩+≡ (37c) ◁129b 129h▷
```
  "CPSR",  LPSR, 0,
  "SPSR",  LPSR, 1,
```
Uses LPSR.

⟨Operand_kind *cases* 129d⟩+≡ (33c) ◁127b
```
  D_PSR,
```

⟨outopd() *cases(arm)* 129e⟩+≡ (109c) ◁127c
```
  case D_PSR:
      break;
```

⟨*more gen rule* 129f⟩+≡ (93b) ◁125e
```
  | LPSR
   {
    $$ = nullgen;
    $$.type = D_PSR;
    $$.reg = $1;
   }
```

⟨*constant* C_FBIT*(arm)* 129g⟩≡ (154c)
```
  #define C_FBIT  (1<<7)  /* psr flags-only */
```

⟨itab *elements* 129h⟩+≡ (37c) ◁129c 129i▷
```
  ".F",  LS, C_FBIT,
```
Uses LS.

11.2.6 Mutual exclusion instructions

11.2.7 Coprocessors

⟨itab *elements* 129i⟩+≡ (37c) ◁129h 130d▷
```
  "MCR",  LSYSTEM, 0,
  "MRC",  LSYSTEM, 1,
```
Uses LSYSTEM.

⟨*inst rule(arm)* 130a⟩+≡ (81b) ◁128b

```
/*
 * MCR MRC
 */
| LSYSTEM cond con ',' expr ',' regi ',' creg ',' creg oexpr
 {
  Gen g;

  g = nullgen;
  g.type = D_CONST;
  g.offset =
   (0xe << 24) |    /* opcode */
   ($1 << 20) |     /* MCR/MRC */
   ($2 << 28) |     /* scond */
   (($3 & 15) << 8) |   /* coprocessor number */
   (($5 & 7) << 21) |   /* coprocessor operation */
   (($7 & 15) << 12) |  /* arm register */
   (($9 & 15) << 16) |  /* Crn */
   (($11 & 15) << 0) |  /* Crm */
   (($12 & 7) << 5) |   /* coprocessor information */
   (1<<4);          /* must be set */ // opcode component
  outcode(AWORD, Always, &nullgen, R_NONE, &g);
 }
```

⟨*type* `declarations`*(arm)* 130b⟩+≡ (81a) ◁128c 130e▷

```
%type   <lval>  creg
```

⟨*creg rule* 130c⟩≡ (? 0—1)

```
creg:
  LCREG
| LC '(' expr ')'
 {
  if($3 < 0 || $3 >= NREG)
      print("register value out of range\n");
  $$ = $3;
 }
```

⟨`itab` *elements* 130d⟩+≡ (37c) ◁129i

```
"C",  LC, 0,

"C0",  LCREG, 0,
"C1",  LCREG, 1,
"C2",  LCREG, 2,
"C3",  LCREG, 3,
"C4",  LCREG, 4,
"C5",  LCREG, 5,
"C6",  LCREG, 6,
"C7",  LCREG, 7,
"C8",  LCREG, 8,
"C9",  LCREG, 9,
"C10",  LCREG, 10,
"C11",  LCREG, 11,
"C12",  LCREG, 12,
"C13",  LCREG, 13,
"C14",  LCREG, 14,
"C15",  LCREG, 15,
```

Uses `LC` and `LCREG`.

⟨*type* `declarations`*(arm)* 130e⟩+≡ (81a) ◁130b

```
%type   <lval>  oexpr
```

⟨*oexpr rule* 131a⟩≡ (? 0—1)
```
/* for MCR */
oexpr:
  /* empty */ { $$ = 0; }
| ',' expr    { $$ = $2; }
```

11.3 Other pseudo opcodes

11.3.1 Compiler-only opcodes

⟨Opcode *cases, pseudo opcodes* 131b⟩+≡ (31) ◁119c 131d▷
```
ACASE,
ABCASE,
```

⟨Opcode *cases, mul/div/mod opcodes* 131c⟩+≡ (31) ◁127d
```
AMULU,
ADIVU, // VIRTUAL, transformed to call to _divu
AMODU, // VIRTUAL, transformed to call to _modu
```

⟨Opcode *cases, pseudo opcodes* 131d⟩+≡ (31) ◁131b 131h▷
```
ASIGNAME,
```

⟨Register *compiler conventions cases* 131e⟩≡ (32) 131f▷
```
// reserved by compiler, calling conventions
REGRET =    0,
REGARG =    0,
```

⟨Register *compiler conventions cases* 131f⟩+≡ (32) ◁131e
```
/* compiler allocates R1 up as temps */
/* compiler allocates register variables R2 up */
REGMIN =    2,
REGMAX =    8,
/* compiler allocates external registers R10 down */
REGEXT =    10, // R9/R10 possible 'extern register xxx;'
```

⟨Register *linker conventions cases* 131g⟩≡ (32)
```
// reserved by linker, long address/offset loading
REGTMP =    11,
```

11.3.2 Linker-only opcodes

⟨Opcode *cases, pseudo opcodes* 131h⟩+≡ (31) ◁131d
```
AGOK,
```

⟨Sym_kind *cases* 131i⟩+≡ (42d) ◁120b
```
N_LINE, // used by linker only
```

11.4 TEXT attributes

⟨*constant* DUPOK *(arm)* 131j⟩≡ (154c)
```
#define DUPOK       (1<<1)
```

⟨*constant* NOPROF *(arm)* 131k⟩≡ (154c)
```
#define NOPROF      (1<<0)
```

11.5 Other preprocessing directives

11.5.1 #define

-D

⟨main() *command line processing* 132a⟩+≡ (43c) ◁73b 146c▷

```
case 'D':
    p = ARGF();
    if(p)
        Dlist[nDlist++] = p;
    break;
```

⟨*global* Dlist 132b⟩≡ (160a)

```
char* Dlist[30];
```

⟨*global* nDlist 132c⟩≡ (160a)

```
int nDlist;
```

⟨assemble() *locals* 132d⟩+≡ (45d) ◁73g

```
int i;
```

⟨assemble() *init Dlist after pinit* 132e⟩≡ (45)

```
for(i=0; i<nDlist; i++)
        dodefine(Dlist[i]);
```

Uses Dlist 132b, dodefine() 132h, and nDlist 132c.

⟨Sym *macro fields* 132f⟩≡ (38)

```
//option<string>  for '#define FOO xxx' expansion
char*   macro;
```

⟨pinit() *initializations* 132g⟩+≡ (50a) ◁111b

```
for(i=0; i<NHASH; i++)
    for(s = hash[i]; s != S; s = s->link)
        s->macro = nil;
```

Uses NHASH 39b, S 39d, and hash 39a.

⟨*function* dodefine 132h⟩≡ (166)

```
void
dodefine(char *cp)
{
    Sym *s;
    char *p;
    long l;
    char *x;

    strcpy(symb, cp);
    p = strchr(symb, '=');
    if(p) {
        *p++ = '\0';
        s = lookup();
        l = strlen(p) + 2; /* +1 null, +1 nargs */

        while(l & 3)
            l++;
        x = malloc(l);

        *x = '\0';
        strcpy(x+1, p);
        s->macro = x;
```

```
    } else {
        s = lookup();
        s->macro = "\0001"; /* \000 is nargs */
    }
    ⟨dodefine() debug 146e⟩
}
```

Uses `lookup()` 40c and `symb` 40a.

#define

⟨*constant* NARG 133a⟩≡ (166)

```
#define NARG 25
```

⟨*function* macdef 133b⟩≡ (166)

```
void
macdef(void)
{
    Sym *s, *a;
    char *args[NARG], *np, *base;
    int n, i, c, len;
    bool dots;
    int ischr;

    s = getsym();
    if(s == S)
        goto bad;
    if(s->macro)
        yyerror("macro redefined: %s", s->name);
    c = getc();
    n = -1;
    dots = 0;
    if(c == '(') {
        n++;
        c = getnsc();
        if(c != ')') {
            unget(c);
            for(;;) {
                a = getsymdots(&dots);
                if(a == S)
                    goto bad;
                if(n >= NARG) {
                    yyerror("too many arguments in #define: %s", s->name);
                    goto bad;
                }
                args[n++] = a->name;
                c = getnsc();
                if(c == ')')
                    break;
                if(c != ',' || dots)
                    goto bad;
            }
        }
        c = getc();
    }
    if(isspace(c))
        if(c != '\n')
            c = getnsc();

    base = hunk;
```

```
len = 1;
ischr = 0;
for(;;) {
    if(isalpha(c) || c == '_') {
        np = symb; // jarod: overflow?
        *np++ = c;
        c = getc();
        while(isalnum(c) || c == '_') {
            *np++ = c;
            c = getc();
        }
        *np = '\0';
        for(i=0; i<n; i++)
            if(strcmp(symb, args[i]) == 0)
                break;
        if(i >= n) {
            i = strlen(symb);
            base = allocn(base, len, i);
            memcpy(base+len, symb, i);
            len += i;
            continue;
        }
        base = allocn(base, len, 2);
        base[len++] = '#';
        base[len++] = 'a' + i;
        continue;
    }
    if(ischr){
        if(c == '\\'){
            base = allocn(base, len, 1);
            base[len++] = c;
            c = getc();
        }else if(c == ischr)
            ischr = 0;
    }else{
        if(c == '"' || c == '\''){
            base = allocn(base, len, 1);
            base[len++] = c;
            ischr = c;
            c = getc();
            continue;
        }
        if(c == '/') {
            c = getc();
            if(c == '/'){
                c = getc();
                for(;;) {
                    if(c == '\n')
                        break;
                    c = getc();
                }
                continue;
            }
            if(c == '*'){
                c = getc();
                for(;;) {
                    if(c == '*') {
                        c = getc();
                        if(c != '/')
```

```
                            continue;
                        c = getc();
                        break;
                    }
                    if(c == '\n') {
                        yyerror("comment and newline in define: %s", s->name);
                        break;
                    }
                    c = getc();
                }
                continue;
            }
            base = allocn(base, len, 1);
            base[len++] = '/';
            continue;
        }
    }
    if(c == '\\') {
        c = getc();
        if(c == '\n') {
            c = getc();
            continue;
        }
        else if(c == '\r') {
            c = getc();
            if(c == '\n') {
                c = getc();
                continue;
            }
        }
        base = allocn(base, len, 1);
        base[len++] = '\\';
        continue;
    }
    if(c == '\n')
        break;
    if(c == '#')
    if(n > 0) {
        base = allocn(base, len, 1);
        base[len++] = c;
    }
    base = allocn(base, len, 1);
    base[len++] = c;
    c = ((--fi.c < 0)? filbuf(): (*fi.p++ & 0xff)); //jarod: GETC
    if(c == '\n')
        lineno++;
    if(c == -1) { // jarod: EOF
        yyerror("eof in a macro: %s", s->name);
        break;
    }
}
do {
    base = allocn(base, len, 1);
    base[len++] = 0;
} while(len & 3);

*base = n+1;
if(dots)
    *base |= VARMAC;
s->macro = base;
```

```
    ⟨macdef() debug 146f⟩
    return;

bad:
    if(s == S)
        yyerror("syntax in #define");
    else
        yyerror("syntax in #define: %s", s->name);
    macend();
}
```

Uses NARG-2 133a, S 39d, VARMAC-1 136a, allocn() 151c, fi 53c, filbuf() 53g, getc() 67c, getnsc() 71b, getsym() 70b, getsymdots() 136b, hunk 150a, lineno 50b, macend() 72b, symb 40a, unget() 71d, and yyerror() 148c.

⟨*constant* VARMAC 136a⟩≡ (166)

```
#define VARMAC 0x80
```

⟨*function* getsymdots 136b⟩≡ (166)

```
Sym*
getsymdots(bool *dots)
{
    int c;
    Sym *s;

    s = getsym();
    if(s != S)
        return s;

    c = getnsc();
    if(c != '.'){
        unget(c);
        return S;
    }
    if(getc() != '.' || getc() != '.')
        yyerror("bad dots in macro");
    *dots = true;
    return slookup("__VA_ARGS__");
}
```

Uses S 39d, getc() 67c, getnsc() 71b, getsym() 70b, slookup() 39e, unget() 71d, and yyerror() 148c.

Macro expansion

⟨yylex() *if macro symbol* 136c⟩≡ (59c)

```
if(s->macro) {
    newio();
    cp = ionext->b;
    macexpand(s, cp);
    pushio();

    ionext->link = iostack;
    iostack = ionext;

    fi.p = cp;
    fi.c = strlen(cp);
    if(peekc != IGN) {
        cp[fi.c++] = peekc;
        cp[fi.c] = 0;
        peekc = IGN;
    }
    goto l0;
```

```
}
```
Uses IGN 57b, fi 53c, ionext 51f, iostack 51c, macexpand() 137, newio() 51g, peekc 57a, and pushio() 74b.

⟨*function* macexpand 137⟩≡ (166)
```
void
macexpand(Sym *s, char *b)
{
    char buf[2000];
    int n, l, c, nargs;
    char *arg[NARG], *cp, *ob, *ecp, dots;

    ob = b;
    if(*s->macro == 0) {
        strcpy(b, s->macro+1);
        ⟨macexpand() debug part1 147a⟩
        return;
    }

    nargs = (char)(*s->macro & ~VARMAC) - 1;
    dots = *s->macro & VARMAC;

    c = getnsc();
    if(c != '(')
        goto bad;
    n = 0;
    c = getc();
    if(c != ')') {
        unget(c);
        l = 0;
        cp = buf;
        ecp = cp + sizeof(buf)-4;
        arg[n++] = cp;
        for(;;) {
            if(cp >= ecp)
                goto toobig;
            c = getc();
            if(c == '"')
                for(;;) {
                    if(cp >= ecp)
                        goto toobig;
                    *cp++ = c;
                    c = getc();
                    if(c == '\\') {
                        *cp++ = c;
                        c = getc();
                        continue;
                    }
                    if(c == '\n') //jarod: how can have this in a macro def?
                        goto bad;
                    if(c == '"')
                        break;
                }
            if(c == '\'')
                for(;;) {
                    if(cp >= ecp)
                        goto toobig;
                    *cp++ = c;
                    c = getc();
                    if(c == '\\') {
                        *cp++ = c;
```

```
                    c = getc();
                    continue;
                }
                if(c == '\n')
                    goto bad;
                if(c == '\'')
                    break;
            }
        if(c == '/') {
            c = getc();
            switch(c) {
            case '*':
                for(;;) {
                    c = getc();
                    if(c == '*') {
                        c = getc();
                        if(c == '/')
                            break;
                    }
                }
                *cp++ = ' ';
                continue;
            case '/':
                while((c = getc()) != '\n')
                    ;
                break;
            default:
                unget(c);
                c = '/';
            }
        }
        if(l == 0) {
            if(c == ',') {
                if(n == nargs && dots) {
                    *cp++ = ',';
                    continue;
                }
                *cp++ = 0;
                arg[n++] = cp;
                if(n > nargs)
                    break;
                continue;
            }
            if(c == ')')
                break;
        }
        if(c == '\n')
            c = ' ';
        *cp++ = c;
        if(c == '(')
            l++;
        if(c == ')')
            l--;
    }
    *cp = 0;
}
if(n != nargs) {
    yyerror("argument mismatch expanding: %s", s->name);
    *b = 0;
    return;
```

```
	}
	cp = s->macro+1;
	for(;;) {
		c = *cp++;
		if(c == '\n')
			c = ' ';
		if(c != '#') {
			*b++ = c;
			if(c == 0)
				break;
			continue;
		}
		c = *cp++;
		if(c == 0)
			goto bad;
		if(c == '#') {
			*b++ = c;
			continue;
		}
		c -= 'a';
		if(c < 0 || c >= n)
			continue;
		strcpy(b, arg[c]);
		b += strlen(arg[c]);
	}
	*b = 0;
	⟨macexpand() debug part2 147b⟩
	return;

bad:
	yyerror("syntax in macro expansion: %s", s->name);
	*b = 0;
	return;

toobig:
	yyerror("too much text in macro expansion: %s", s->name);
	*b = 0;
}
```

Uses `NARG-2` 133a, `VARMAC-1` 136a, `getc()` 67c, `getnsc()` 71b, `unget()` 71d, and `yyerror()` 148c.

11.5.2 #undef

⟨*function* `macund` 139⟩≡ (166)

```
void
macund(void)
{
	Sym *s;

	s = getsym();
	macend();
	if(s == S) {
		yyerror("syntax in #undef");
		return;
	}
	s->macro = nil;
}
```

Uses `S` 39d, `getsym()` 70b, `macend()` 72b, and `yyerror()` 148c.

11.5.3 #ifdef

⟨*function* macif 140⟩≡ (166)

```
void
macif(int f)
{
    int c, l;
    bool bol;
    Sym *s;

    if(f == 2)
        goto skip;
    s = getsym();
    if(s == S)
        goto bad;
    if(getcom() != '\n')
        goto bad;
    if((s->macro != nil) ^ f)
        return;

skip:
    bol = true;
    l = 0;
    for(;;) {
        c = getc();
        if(c != '#') {
            if(!isspace(c))
                bol = false;
            if(c == '\n')
                bol = true;
            continue;
        }
        if(!bol)
            continue;
        s = getsym();
        if(s == S)
            continue;
        if(strcmp(s->name, "endif") == 0) {
            if(l) {
                l--;
                continue;
            }
            macend();
            return;
        }
        if(strcmp(s->name, "ifdef") == 0 || strcmp(s->name, "ifndef") == 0) {
            l++;
            continue;
        }
        if(l == 0 && f != 2 && strcmp(s->name, "else") == 0) {
            macend();
            return;
        }
    }

bad:
    yyerror("syntax in #if(n)def");
    macend();
}
```

Uses `S` 39d, `getc()` 67c, `getcom()` 76e, `getsym()` 70b, `macend()` 72b, and `yyerror()` 148c.

⟨domacro() *set s to endif symbol if no symbol* 141a⟩≡ (70a)

```
if(s == S)
    s = slookup("endif");
```

Uses S 39d and slookup() 39e.

11.5.4 #pragma

⟨*function* macprag 141b⟩≡ (166)

```
void
macprag(void)
{
    Sym *s;
    ⟨macprag() locals 141c⟩

    s = getsym();

    ⟨macprag() if pragma lib 141d⟩
    else {
        while(getnsc() != '\n')
            ;
        return;
    }
}
```

Uses getnsc() 71b and getsym() 70b.

11.6 #pragma lib and automagic linking

⟨macprag() *locals* 141c⟩≡ (141b)

```
Hist *h;
char *hp;
int c0, c;
```

⟨macprag() *if pragma lib* 141d⟩≡ (141b)

```
if(s && strcmp(s->name, "lib") == 0) {
    c0 = getnsc();
    if(c0 != '"') {
        c = c0;
        if(c0 != '<')
            goto bad;
        c0 = '>';
    }
    for(hp = symb;;) {
        c = getc();
        if(c == c0)
            break;
        if(c == '\n')
            goto bad;
        *hp++ = c;
    }
    *hp = '\0';
    c = getcom();
    if(c != '\n')
        goto bad;

    /*
     * put pragma-line in as a funny history
     */
```

```
    c = strlen(symb) + 1;
    while(c & 3)
        c++;

    hp = malloc(c);
    memcpy(hp, symb, c);

    h = alloc(sizeof(Hist));
    h->filename = hp;
    h->global_line = lineno;
    h->local_line = -1; // ugly, special mark for #pragma lib in Hist

    h->link = H;
    if(ehist == H) {
        hist = h;
        ehist = h;
        return;
    }
    ehist->link = h;
    ehist = h;

    return;

bad:
    unget(c);
    yyerror("syntax in #pragma lib");
    macend();

}
```

Uses `H` 115d, `alloc()` 151b, `ehist` 115e, `getc()` 67c, `getcom()` 76e, `getnsc()` 71b, `hist` 115b, `lineno` 50b, `macend()` 72b, `symb` 40a, `unget()` 71d, and `yyerror()` 148c.

11.7 Processing multiple files

⟨`main()` *locals* 142a⟩+≡ (43c) ◁73a

```
int nout, nproc, status;
int i, c;
```

⟨`main()` *multiple files handling* 142b⟩≡ (43c)

```
if(argc > 1) {
    nproc = 1;
    if(p = getenv("NPROC"))
        nproc = atol(p); /* */
    c = 0;
    nout = 0;
    for(;;) {
        while(nout < nproc && argc > 0) {
            i = fork();
            if(i < 0) {
                i = mywait(&status);
                if(i < 0)
                    errorexit();
                if(status)
                    c++;
                nout--;
                continue;
            }
            if(i == 0) {
```

```
				print("%s:\n", *argv);
				if(assemble(*argv))
					errorexit();
				exits(nil);
			}
			nout++;
			argc--;
			argv++;
		}
		i = mywait(&status);
		if(i < 0) {
			if(c)
				errorexit();
			exits(nil);
		}
		if(status)
			c++;
		nout--;
	}
}
```

Chapter 12

Conclusion

You now know how the Plan 9 ARM assembler `5a` works, to the smallest details, and more generally how many assemblers work.

Despite being a complete assembler—with a lexer, preprocessor, parser, and object code generator—`5a` fits in roughly 5000 lines of C. This compactness comes from two key design choices: the AST-based object format (deferring machine code encoding to the linker `5l`), and the uniform instruction syntax where every instruction is simply an opcode followed by up to three operands. Along the way, you have seen many patterns common to language processors: a hand-written lexer with a peek buffer, a `Sym` hash table with multiple namespaces, a recursive-descent parser driven by opcode classes, and a two-pass architecture for resolving forward references.

12.1 Patterns and techniques

These techniques apply far beyond assembly language:

- *Two-pass processing*: scan once to learn (collect symbols and sizes), scan again to act (generate code). LaTeX runs twice for the same reason—resolving forward references requires knowing what comes later.
- *Hand-written lexer with peek*: the lexer uses a one-character `peek` buffer for lookahead, avoiding the complexity of a generated scanner. This is the simplest viable tokenizer architecture, and the same approach is used by JSON parsers, HTTP header parsers, and configuration file readers—any format simple enough that a state machine with one character of lookahead suffices.
- *Classification-based dispatch*: grouping hundreds of opcodes into a handful of classes (data-processing, load/store, branch) and dispatching on the class. The same technique—collapsing many cases into few categories—is used in Unicode processing (character categories) and network protocol handling (message types).
- *Deferred encoding*: writing an abstract representation (`Prog` nodes) and letting the linker handle binary encoding. This separation of "what" from "how to encode it" is the idea behind LLVM IR, Java bytecode, and Protocol Buffers: defer the final representation to a later stage that has more context.

12.2 Connections to other books

The assembler sits at the center of the toolchain, connecting to nearly every other book in Principia Softwarica:

- Linker book [Pad15b]: the linker and assembler are strongly connected in any system (for instance `ld` and `gas` are part of the same `binutils` package in the GNU system). The Linker book [Pad15b] is the next logical step after this book—it reads the AST-based object files that `5a` produces and performs the actual machine code generation.

- EMULATOR book [Pad15a]: fully describes the semantics of the machine instructions you have seen in this book.
- DEBUGGER book [Pad16d]: contains code to *disassemble* machine code, the reverse of what `5a` does.
- COMPILER book [Pad16b]: reading it will now be easier because you can understand the assembly code generated by the C compiler `5c`.
- KERNEL book [Pad14]: the kernel is assembled by `5a`, and you have seen how pseudo instructions like `GLOBL` and `DATA` are used to set up the kernel's entry point and data sections.

12.3 Beyond the Plan 9 assembler

Most programmers today never write assembly directly, but assemblers remain a critical part of every toolchain. Here are some features found in other assemblers:

- *Machine code generation*: unlike `5a`, most assemblers (GNU `as`, LLVM's integrated assembler, NASM) generate machine code directly, producing relocatable object files in ELF or Mach-O format. Plan 9's choice to defer encoding to the linker is unusual but keeps the assembler simpler and avoids duplicating instruction encoding logic.
- *Macro systems*: GNU `as` has a basic macro facility, but assemblers like NASM and MASM provide powerful macro systems with conditional assembly, string operations, and macro libraries. `5a`'s preprocessor is inherited from the C preprocessor (`#define`, `#include`, `#ifdef`) which is less specialized but familiar to C programmers.
- *Multiple architectures*: LLVM's integrated assembler and GNU `as` support dozens of architectures from a single codebase. Plan 9 takes the opposite approach: one assembler per architecture (`5a` for ARM, `8a` for x86, `6a` for AMD64), each small and self-contained.
- *Integrated assemblers*: modern compilers like GCC (with `as`) and especially LLVM/Clang integrate the assembler into the compiler itself, going directly from IR to machine code without emitting assembly text. This is faster and avoids parsing assembly, but loses the ability to inspect the intermediate assembly output easily.
- *Directives*: `5a` has relatively few pseudo instructions. Production assemblers like GNU `as` support a large set of directives for alignment, section control, debug information (DWARF), exception handling tables, and linker hints—reflecting the complexity of modern executable formats.

The fundamentals are the same everywhere: read mnemonics, look them up in an opcode table, encode operands, emit relocations for unresolved symbols. `5a` presents these ideas in their clearest form, free of the accumulated complexity of supporting dozens of architectures and executable formats.

Appendix A

Debugging

`5a` has a simple debug flag mechanism: the `debug` array is indexed by character, so `5a -dm` enables debug output for the `'m'` (macro) category. Most of the debugging output relates to the preprocessor, which is the trickiest part of `5a` to get right.

⟨*global* `debug` 146a⟩≡ (160a)

```
bool debug[256];
```

⟨`main()` *remaining initializations* 146b⟩+≡ (43c) ◁72f

```
memset(debug, false, sizeof(debug));
```

⟨`main()` *command line processing* 146c⟩+≡ (43c) ◁132a

```
default:
    c = ARGC();
    if(c >= 0 || c < sizeof(debug))
        debug[c] = true;
    break;
```

A.1 Line information debugging: `5a -f`

⟨`linehist()` *debug* 146d⟩≡ (115g)

```
if(debug['f'])
    if(f) {
        if(local_line)
            print("%4ld: %s (#line %d)\n", lineno, f, local_line);
        else
            print("%4ld: %s\n", lineno, f);
    } else
        print("%4ld: <pop>\n", lineno);
```

Uses `debug` 146a and `lineno` 50b.

A.2 Macro debugging: `5a -m`

⟨`dodefine()` *debug* 146e⟩≡ (132h)

```
if(debug['m'])
    print("#define (-D) %s %s\n", s->name, s->macro+1);
```

Uses `debug` 146a.

⟨`macdef()` *debug* 146f⟩≡ (133b)

```
if(debug['m'])
    print("#define %s %s\n", s->name, s->macro+1);
```

Uses `debug` 146a.

⟨`macexpand()` *debug part1* 147a⟩≡ (137)

```
  if(debug['m'])
      print("#expand %s %s\n", s->name, ob);
```

Uses `debug` 146a.

⟨`macexpand()` *debug part2* 147b⟩≡ (137)

```
  if(debug['m'])
      print("#expand %s %s\n", s->name, ob);
```

Uses `debug` 146a.

Appendix B

Error Management

When 5a encounters an error (syntax error, unknown opcode, bad operand), it reports the error with the current file and line number but continues processing to find further errors. On exit, `errorexit()` removes the partial output file so the linker does not accidentally use a corrupt object file.

⟨*function* `errorexit` 148a⟩≡ (160c)

```
/// main | assemble | yyerror | ... -> <>
void
errorexit(void)
{

    if(outfile)
        remove(outfile);
    exits("error");
}
```

Uses `outfile` 44a.

⟨*global* `nerrors` 148b⟩≡ (160a)

```
int nerrors = 0;
```

Uses `nerrors` 148b.

⟨*function* `yyerror` 148c⟩≡ (160c)

```
/// assemble | yylex | yyparse | ... -> <>
void
yyerror(char *a, ...)
{
    char buf[200];
    va_list arg;

    ⟨yyerror() when called from yyparse 149⟩

    prfile(lineno);

    va_start(arg, a);
    vseprint(buf, buf+sizeof(buf), a, arg);
    va_end(arg);

    print("%s\n", buf);

    nerrors++;
    if(nerrors > 10) {
        print("too many errors\n");
        errorexit();
    }
}
```

Uses `errorexit()` 148a, `lineno` 50b, `nerrors` 148b, and `prfile()` 118b.

⟨`yyerror()` *when called from yyparse* 149⟩≡ (148c)

```
/*
 * hack to intercept message from yaccpar
 */
if(strcmp(a, "syntax error") == 0) {
    yyerror("syntax error, last name: %s", symb);
    return;
}
```

Uses `symb` 40a and `yyerror()` 148c.

Appendix C

Utilities

This appendix collects utility code used by 5a: buffered I/O (libbio) and 5a's custom memory allocator, which allocates from a large arena in fixed-size chunks rather than calling malloc() for each small allocation.

C.1 Buffer management

C.2 Memory management

⟨*global* hunk 150a⟩≡ (160a)
```
char* hunk;
```

⟨*global* nhunk 150b⟩≡ (160a)
```
long nhunk = 0;
```
Uses nhunk 150b.

⟨*global* thunk 150c⟩≡ (160a)
```
long thunk;
```

⟨*constant* NHUNK 150d⟩≡ (156b)
```
#define NHUNK       10000
```

⟨*function* gethunk 150e⟩≡ (158)
```
void
gethunk(void)
{
    char *h;
    long nh;

    nh = NHUNK;
    if(thunk >= 10L*NHUNK)
        nh = 10L*NHUNK;

    h = (char*)sbrk(nh);
    if(h == (char*)-1) {
        yyerror("out of memory");
        errorexit();
    }
    hunk = h;
    nhunk = nh;
    thunk += nh;
}
```
Uses NHUNK 150d, errorexit() 148a, hunk 150a, nhunk 150b, thunk 150c, and yyerror() 148c.

⟨*constant* `MAXALIGN` 151a⟩≡ (156b)

```
#define MAXALIGN 7
```

⟨*function* `alloc` 151b⟩≡ (158)

```
/*
 * real allocs
 */
void*
alloc(long n)
{
    void *p;

    while((uintptr)hunk & MAXALIGN) {
        hunk++;
        nhunk--;
    }

    while(nhunk < n)
        gethunk();
    p = hunk;
    nhunk -= n;
    hunk += n;

    return p;
}
```

Uses `MAXALIGN` 151a, `gethunk()` 150e, `hunk` 150a, and `nhunk` 150b.

⟨*function* `allocn` 151c⟩≡ (158)

```
void*
allocn(void *p, long on, long n)
{
    void *q;

    q = (uchar*)p + on;
    if(q != hunk || nhunk < n) {

        while(nhunk < on+n)
            gethunk();
        memmove(hunk, p, on);
        p = hunk;
        hunk += on;
        nhunk -= on;

    }
    hunk += n;
    nhunk -= n;
    return p;
}
```

Uses `gethunk()` 150e, `hunk` 150a, and `nhunk` 150b.

Appendix D

Examples of Assembly Programs TODO

D.1 hello.s and pwrite.s

D.2 memset.s

D.3 div.s

The _div function in lib_core/libc/arm/div.s implements in software the integer division. Indeed, the ARM has no DIV instruction.

⟨*function* _div 152a⟩≡
```
TEXT _div(SB), 1, $16
 BL save<>(SB)
 CMP $0, R(Q)
 BGE d1
 RSB $0, R(Q), R(Q)
 CMP $0, R(D)
 BGE d2
 RSB $0, R(D), R(D)
d0:
 BL div<>(SB)  /* none/both neg */
 MOVW R(Q), R(TMP)
 B out
d1:
 CMP $0, R(D)
 BGE d0
 RSB $0, R(D), R(D)
    // Fallthrough
d2:
 BL div<>(SB)  /* one neg */
 RSB $0, R(Q), R(TMP)
 B out

out:
 BL rest<>(SB)
 B out
```

The _mod function in lib_core/libc/arm/div.s implements in software the integer modulo since the ARM has no MOD instruction.

⟨*function* _mod 152b⟩≡
```
TEXT _mod(SB), 1, $16
 BL save<>(SB)
 CMP $0, R(D)
```

```
  RSB.LT $0, R(D), R(D)
  CMP $0, R(Q)
  BGE m1
  RSB $0, R(Q), R(Q)
  BL div<>(SB)  /* neg numerator */
  RSB $0, R(N), R(TMP)
  B out
m1:
  BL div<>(SB)  /* pos numerator */
  MOVW R(N), R(TMP)
  B out
```

D.4 main9.s

D.5 tas.s

D.6 getcallerpc.s

Appendix E

Extra Code

E.1 include/

E.1.1 include/obj/common.out.h

⟨include/obj/common.out.h 154a⟩≡

```
// The entities below were originally in some xxx/y.out.h, but were always
// the same in all architecture, hence the factorization below.

⟨constant NSYM 110c⟩

⟨struct ieee 123b⟩
typedef struct ieee Ieee;
```

E.1.2 include/obj/5.out.h

⟨constant SYMDEF(arm) 154b⟩≡ (154c)

```
/*
 * this is the ranlib header
 */
#define SYMDEF  "__.SYMDEF"
```

⟨include/obj/5.out.h 154c⟩≡

```
// Many of the types below are serialized in the .5 object files,
// so take care when changing those types to not alter the order
// (or to recompile carefully everything).

⟨enum Register(arm) 32⟩
⟨constant R_NONE(arm) 33a⟩

⟨enum Fregister(arm) 124c⟩

⟨enum Opcode(arm) 31⟩

⟨enum Operand_kind(arm) 33c⟩

⟨enum Sym_kind(arm) 42d⟩

⟨constant NSNAME 34c⟩

// Attributes
```

```
⟨constant NOPROF (arm) 131k⟩
⟨constant DUPOK (arm) 131j⟩

⟨constant C_SCOND (arm) 103c⟩
⟨constant C_SBIT (arm) 103f⟩
⟨constant C_PBIT (arm) 103h⟩
⟨constant C_WBIT (arm) 104a⟩
⟨constant C_FBIT (arm) 129g⟩
⟨constant C_UBIT (arm) 128f⟩

#define COND_ALWAYS 14

⟨constant SYMDEF (arm) 154b⟩
```

E.2 assemblers/misc/

E.2.1 assemblers/misc/data2s.c

⟨*function* main 155a⟩≡ (155b)

```
void
main(int argc, char *argv[])
{
    Biobuf bin, bout;
    long len, slen;
    int c;

    if(argc != 2){
        fprint(2, "usage: data2s name\n");
        exits("usage");
    }
    Binit(&bin, 0, OREAD);
    Binit(&bout, 1, OWRITE);
    for(len=0; (c=Bgetc(&bin))!=Beof; len++){
        if((len&7) == 0)
            Bprint(&bout, "DATA %scode+%ld(SB)/8, $\"", argv[1], len);
        if(c)
            Bprint(&bout, "\\%uo", c);
        else
            Bprint(&bout, "\\z");
        if((len&7) == 7)
            Bprint(&bout, "\"\n");
    }
    slen = len;
    if(len & 7){
        while(len & 7){
            Bprint(&bout, "\\z");
            len++;
        }
        Bprint(&bout, "\"\n");
    }
    Bprint(&bout, "GLOBL %scode+0(SB), $%ld\n", argv[1], len);
    Bprint(&bout, "GLOBL %slen+0(SB), $4\n", argv[1]);
    Bprint(&bout, "DATA %slen+0(SB)/4, $%ld\n", argv[1], slen);
    exits(0);
}
```

⟨assemblers/misc/data2s.c 155b⟩≡

```
#include <u.h>
```

```
#include <libc.h>
#include <bio.h>

⟨function main 155a⟩
```

E.3 assemblers/aa/

E.3.1 assemblers/aa/aa.h

⟨*enum platform* 156a⟩≡ (156b)

```
/*
 *  system-dependent stuff from ../cc/compat.c
 */
enum    /* keep in synch with ../cc/cc.h */
{
    Plan9   = 1<<0,
    Unix    = 1<<1,
};
```

⟨assemblers/aa/aa.h 156b⟩≡

```
#include <u.h>
#include <libc.h>
#include <bio.h>
#include <ctype.h>

// The content of this file was originally copy pasted in 8a/a.h, 5a/a.h, etc.
// It was almost always the same in all archi, so I factorized the generic
// part in aa.h.; for the specifics do #include Y.out.h in Ya/a.h as followed:
//#include "8.out.h" in 8a/a.h
//#include "5.out.h" in 5a/a.h

#include <common.out.h>

#pragma lib "../aa/aa.a$O"

//----------------------------------------------------------------------------
// Data structures and constants
//----------------------------------------------------------------------------

typedef struct  Sym Sym;
typedef struct  Io Io;
typedef struct  Hist Hist;

⟨constant MAXALIGN 151a⟩
⟨constant NSYMB 40b⟩
⟨constant BUFSIZ 53b⟩
⟨constant HISTSZ 118a⟩
⟨constant NINCLUDE 72d⟩
⟨constant NHUNK 150d⟩
⟨constant NHASH 39b⟩
⟨constant STRINGSZ 75d⟩

⟨constant EOF 55a⟩
⟨constant IGN 57b⟩
⟨function GETC 53f⟩


⟨struct Sym 38⟩
```

```
⟨constant S 39d⟩

⟨struct Fi 53d⟩

⟨struct Io 51a⟩
⟨constant I 51e⟩

⟨constant FD_NONE 51b⟩

⟨struct Htab 110d⟩

⟨struct Hist 115a⟩
⟨constant H 115d⟩

//----------------------------------------------------------------------------
// Globals
//----------------------------------------------------------------------------

// core algorithm
extern  Sym*  hash[NHASH];
extern  int   pass;
extern  long  pc;

extern  char* outfile;
extern  Biobuf obuf;
extern  char* pathname;
extern struct Fi fi;

extern struct Htab h[NSYM];
extern  int   symcounter;

extern  int   thechar;
extern  char* thestring;

// input files (used by lex.c)
extern  Io* iostack;
extern  Io* iofree;
extern  Io* ionext;

// cpp
extern  char* include[NINCLUDE];
extern  int   ninclude;
extern  char* Dlist[30];
extern  int   nDlist;

// lexer and lookup
extern  char symb[NSYMB];
extern  int  peekc;

// debugging support
extern  Hist*   hist;
extern  Hist*   ehist;
extern  long    lineno;

// debugging
extern  bool    debug[256];

// error management
extern  int nerrors;
```

```
// utils (used by mac.c)
extern  char*   hunk;
extern  long    nhunk;
extern  long    thunk;

//----------------------------------------------------------------------------
// Functions
//----------------------------------------------------------------------------

// lookup.c
Sym*    slookup(char*);
Sym*    lookup(void);
// this actually must be defined in lex.c; it depends on LNAME
void    syminit(Sym*);

// lexbody.c (used by lex.c and macbody.c)
void  pinit(char*);
void  pushio(void);
void  newio(void);
void  newfile(char*, int);
void  setinclude(char*);
int   escchar(int);
int  filbuf(void); // used by GETC()
int  getc(void);
int  getnsc(void);
void unget(int);

// macbody.c (used by lex.c and main.c)
void dodefine(char*);
void domacro(void);
void macexpand(Sym*, char*);

// hist.c
void linehist(char*, int);
void prfile(long l);

// float.c
void ieeedtod(Ieee *ieee, double native);

// error.c
void yyerror(char*, ...);
void  errorexit(void);

// utils.c
void* alloc(long n);
void* allocn(void *p, long on, long n);
int systemtype(int);
int pathchar(void);
int mywait(int*);
int mycreat(char*, int);


⟨enum platform 156a⟩
```
Uses `Hist` 156b, `Io` 156b, and `Sym` 156b.

E.3.2 assemblers/aa/utils.c

⟨assemblers/aa/utils.c 158⟩≡
```
#include "aa.h"
```

⟨*function* gethunk 150e⟩

⟨*function* alloc 151b⟩

⟨*function* allocn 151c⟩

⟨*function* mycreat 159a⟩

⟨*function* mywait 159b⟩

⟨*function* systemtype 159c⟩

⟨*function* pathchar 159d⟩

⟨*function* mycreat 159a⟩≡ (158)

```
int
mycreat(char *n, int p)
{

    return create(n, 1, p);
}
```

⟨*function* mywait 159b⟩≡ (158)

```
int
mywait(int *s)
{
    int p;
    Waitmsg *w;

    if((w = wait()) == nil)
        return -1;
    else{
        p = w->pid;
        *s = 0;
        if(w->msg[0])
            *s = 1;
        free(w);
        return p;
    }
}
```

⟨*function* systemtype 159c⟩≡ (158)

```
int
systemtype(int sys)
{
    return sys & Plan9;
}
```

Uses Plan9 156a.

⟨*function* pathchar 159d⟩≡ (158)

```
int
pathchar(void)
{
    return '/';
}
```

E.3.3 assemblers/aa/globals.c

```
⟨assemblers/aa/globals.c 160a⟩≡
  #include "aa.h"

  ⟨global hash 39a⟩
  ⟨global pc 92a⟩
  ⟨global outfile 44a⟩
  ⟨global obuf 47a⟩
  ⟨global pass 45c⟩
  ⟨global pathname 45a⟩

  ⟨global thechar 43a⟩
  ⟨global thestring 43b⟩

  ⟨global iostack 51c⟩
  ⟨global iofree 52a⟩
  ⟨global ionext 51f⟩
  ⟨global fi 53c⟩

  ⟨global symb 40a⟩
  ⟨global peekc 57a⟩

  ⟨global include 72c⟩
  ⟨global ninclude 72e⟩
  ⟨global Dlist 132b⟩
  ⟨global nDlist 132c⟩

  ⟨global h 110b⟩
  ⟨global symcounter 111a⟩

  ⟨global lineno 50b⟩
  ⟨global hist 115b⟩
  ⟨global ehist 115e⟩

  ⟨global debug 146a⟩

  ⟨global nerrors 148b⟩

  ⟨global hunk 150a⟩
  ⟨global nhunk 150b⟩
  ⟨global thunk 150c⟩
```

E.3.4 assemblers/aa/lookup.c

```
⟨assemblers/aa/lookup.c 160b⟩≡
  #include "aa.h"

  // syminit() in lookup() depends on LNAME token defined in a.y,
  // so it can't be defined here.

  ⟨function slookup 39e⟩

  ⟨function lookup 40c⟩
```

E.3.5 assemblers/aa/error.c

```
⟨assemblers/aa/error.c 160c⟩≡
```

```
#include "aa.h"

⟨function errorexit 148a⟩

⟨function yyerror 148c⟩
```

E.3.6 assemblers/aa/float.c

```
⟨assemblers/aa/float.c 161a⟩≡
  #include "aa.h"

  ⟨function ieeedtod 126b⟩
```

E.3.7 assemblers/aa/hist.c

```
⟨assemblers/aa/hist.c 161b⟩≡
  #include "aa.h"

  ⟨function linehist 115g⟩

  ⟨function prfile 118b⟩
```

E.3.8 assemblers/aa/lexbody.c

```
⟨assemblers/aa/lexbody.c 161c⟩≡
  #include "aa.h"

  /*
   * common code for all the assemblers
   */

  ⟨function setinclude 73c⟩

  ⟨function pushio 74b⟩

  ⟨function newio 51g⟩

  ⟨function newfile 52d⟩

  // this was hard to factorize in aa/, so this is copy pasted
  // in each assembler (8a/, va/, etc)
  //long
  //yylex(void)
  //{
  // int c, c1;
  // char *cp;
  // Sym *s;
  //
  // c = peekc;
  // if(c != IGN) {
  //  peekc = IGN;
  //  goto l1;
  // }
  //l0:
```

```
// c = GETC();
//
//l1:
// if(c == EOF) {
//  peekc = EOF;
//  return -1;
// }
// if(isspace(c)) {
//  if(c == '\n') {
//   lineno++;
//   return ';';
//  }
//  goto l0;
// }
// if(isalpha(c))
//  goto talph;
// if(isdigit(c))
//  goto tnum;
// switch(c)
// {
// case '\n':
//  lineno++;
//  return ';';
//
// case '#':
//  domacro();
//  goto l0;
//
// case '.':
//  c = GETC();
//  if(isalpha(c)) {
//   cp = symb;
//   *cp++ = '.';
//   goto aloop;
//  }
//  if(isdigit(c)) {
//   cp = symb;
//   *cp++ = '.';
//   goto casedot;
//  }
//  peekc = c;
//  return '.';
//
// talph:
// case '_':
// case '@':
//  cp = symb;
//
// aloop:
//  *cp++ = c;
//  c = GETC();
//  if(isalpha(c) || isdigit(c) || c == '_' || c == '$')
//   goto aloop;
//  *cp = 0;
//  peekc = c;
//  s = lookup();
//  if(s->macro) {
//   newio();
//   cp = ionext->b;
//   macexpand(s, cp);
```

```
//   pushio();
//   ionext->link = iostack;
//   iostack = ionext;
//   fi.p = cp;
//   fi.c = strlen(cp);
//   if(peekc != IGN) {
//    cp[fi.c++] = peekc;
//    cp[fi.c] = 0;
//    peekc = IGN;
//   }
//   goto l0;
//  }
//  if(s->type == 0)
//   s->type = LNAME;
//  if(s->type == LNAME ||
//     s->type == LVAR ||
//     s->type == LLAB) {
//   yylval.sym = s;
//   return s->type;
//  }
//  yylval.lval = s->value;
//  return s->type;
//
// tnum:
//  cp = symb;
//  if(c != '0')
//   goto dc;
//  *cp++ = c;
//  c = GETC();
//  c1 = 3;
//  if(c == 'x' || c == 'X') {
//   c1 = 4;
//   c = GETC();
//  } else
//  if(c < '0' || c > '7')
//   goto dc;
//  yylval.lval = 0;
//  for(;;) {
//   if(c >= '0' && c <= '9') {
//    if(c > '7' && c1 == 3)
//     break;
//    yylval.lval <<= c1;
//    yylval.lval += c - '0';
//    c = GETC();
//    continue;
//   }
//   if(c1 == 3)
//    break;
//   if(c >= 'A' && c <= 'F')
//    c += 'a' - 'A';
//   if(c >= 'a' && c <= 'f') {
//    yylval.lval <<= c1;
//    yylval.lval += c - 'a' + 10;
//    c = GETC();
//    continue;
//   }
//   break;
//  }
//  goto ncu;
//
```

```
// dc:
//  for(;;) {
//   if(!isdigit(c))
//    break;
//   *cp++ = c;
//   c = GETC();
//  }
//  if(c == '.')
//   goto casedot;
//  if(c == 'e' || c == 'E')
//   goto casee;
//  *cp = 0;
//  if(sizeof(yylval.lval) == sizeof(vlong))
//   yylval.lval = strtoll(symb, nil, 10);
//  else
//   yylval.lval = strtol(symb, nil, 10);
//
// ncu:
//  while(c == 'U' || c == 'u' || c == 'l' || c == 'L')
//   c = GETC();
//  peekc = c;
//  return LCONST;
//
// casedot:
//  for(;;) {
//   *cp++ = c;
//   c = GETC();
//   if(!isdigit(c))
//    break;
//  }
//  if(c == 'e' || c == 'E')
//   goto casee;
//  goto caseout;
//
// casee:
//  *cp++ = 'e';
//  c = GETC();
//  if(c == '+' || c == '-') {
//   *cp++ = c;
//   c = GETC();
//  }
//  while(isdigit(c)) {
//   *cp++ = c;
//   c = GETC();
//  }
//
// caseout:
//  *cp = 0;
//  peekc = c;
//  if(FPCHIP) {
//   yylval.dval = atof(symb);
//   return LFCONST;
//  }
//  yyerror("assembler cannot interpret fp constants");
//  yylval.lval = 1L;
//  return LCONST;
//
// case '"':
//  memcpy(yylval.sval, nullgen.sval, sizeof(yylval.sval));
//  cp = yylval.sval;
```

```
//  c1 = 0;
//  for(;;) {
//   c = escchar('"');
//   if(c == EOF)
//    break;
//   if(c1 < sizeof(yylval.sval))
//    *cp++ = c;
//   c1++;
//  }
//  if(c1 > sizeof(yylval.sval))
//   yyerror("string constant too long");
//  return LSCONST;
//
// case '\'':
//  c = escchar('\'');
//  if(c == EOF)
//   c = '\'';
//  if(escchar('\'') != EOF)
//   yyerror("missing '");
//  yylval.lval = c;
//  return LCONST;
//
// case '/':
//  c1 = GETC();
//  if(c1 == '/') {
//   for(;;) {
//    c = GETC();
//    if(c == '\n')
//     goto l1;
//    if(c == EOF) {
//     yyerror("eof in comment");
//     errorexit();
//    }
//   }
//  }
//  if(c1 == '*') {
//   for(;;) {
//    c = GETC();
//    while(c == '*') {
//     c = GETC();
//     if(c == '/')
//      goto l0;
//    }
//    if(c == EOF) {
//     yyerror("eof in comment");
//     errorexit();
//    }
//    if(c == '\n')
//     lineno++;
//   }
//  }
//  break;
//
// default:
//  return c;
// }
// peekc = c1;
// return c;
//}
```

```
⟨function getc 67c⟩

⟨function getnsc 71b⟩

⟨function unget 71d⟩

⟨function escchar 65b⟩

⟨function pinit 50a⟩

⟨function filbuf 53g⟩
```

E.3.9 assemblers/aa/macbody.c

```
⟨assemblers/aa/macbody.c 166⟩≡
  #include "aa.h"

  // forward decls
  void macund(void);
  void macdef(void);
  void macinc(void);
  void macprag(void);
  void maclin(void);
  void macif(int);
  void macend(void);

  ⟨constant VARMAC 136a⟩

  ⟨function getnsn 79e⟩

  ⟨function getsym 70b⟩

  ⟨function getsymdots 136b⟩

  ⟨function getcom 76e⟩

  ⟨function dodefine 132h⟩

  ⟨global mactab 69b⟩

  ⟨function domacro 70a⟩

  ⟨function macund 139⟩

  ⟨constant NARG 133a⟩
  ⟨function macdef 133b⟩

  ⟨function macexpand 137⟩

  ⟨function macinc 74a⟩

  ⟨function maclin 78a⟩

  ⟨function macif 140⟩

  ⟨function macprag 141b⟩
```

⟨*function* macend 72b⟩

E.4 assemblers/5a/

E.4.1 assemblers/5a/a.h

⟨assemblers/5a/a.h 167a⟩≡
```
#include "../aa/aa.h"
#include <5.out.h>

//----------------------------------------------------------------------------
// Data structures and constants
//----------------------------------------------------------------------------

⟨constant FPCHIP(arm) 123a⟩

⟨constant Always(arm) 103b⟩

⟨struct Gen(arm) 33b⟩
typedef struct  Gen Gen;

//----------------------------------------------------------------------------
// Globals
//----------------------------------------------------------------------------

// globals.c
extern  Gen nullgen;

//----------------------------------------------------------------------------
// Functions
//----------------------------------------------------------------------------

// lex.c (for y.tab.c, main.c)
⟨signature yylex 56a⟩
void    cinit(void);

// y.tab.c from a.y (for main.c)
int yyparse(void);

// obj.c (for main.c)
⟨signature outcode(arm) 34d⟩
void    outhist(void);
```

Uses Gen 33b.

E.4.2 assemblers/5a/globals.c

⟨assemblers/5a/globals.c 167b⟩≡
```
#include "a.h"

⟨global nullgen 34a⟩
```

E.4.3 assemblers/5a/lex.c

```
⟨assemblers/5a/lex.c 168a⟩≡
  #include "a.h"
  #include "y.tab.h"

  ⟨struct Itab(arm) 37b⟩

  ⟨global itab(arm) 37c⟩

  ⟨function cinit(arm) 44c⟩

  ⟨function syminit 40e⟩

  // now use aa.a8
  //#include "../cc/lexbody"
  //#include "../cc/compat"

  // used to be in ../cc/lexbody and factorized between assemblers by
  // using #include, but ugly, so I copy pasted the function for now
  ⟨function yylex 56b⟩

  // #include "../cc/macbody"
```

E.4.4 assemblers/5a/a.y

E.4.5 assemblers/5a/obj.c

```
⟨assemblers/5a/obj.c 168b⟩≡
  #include "a.h"

  ⟨function zname(arm) 111c⟩

  ⟨function outopd(arm) 109c⟩

  ⟨global bcode(arm) 109b⟩

  ⟨function symidx_of_symopt 112c⟩

  ⟨function outcode(arm) 108⟩

  ⟨function outhist(arm) 119b⟩
```

E.4.6 assemblers/5a/main.c

```
⟨assemblers/5a/main.c 168c⟩≡
  #include "a.h"

  // forward decls
  int  assemble(char*);
  void cclean(void);

  ⟨function main(arm) 43c⟩

  ⟨function assemble 45d⟩

  ⟨function cclean(arm) 47b⟩
```

E.5 lib_core/libc/arm/

E.5.1 lib_core/libc/arm/div.s

⟨lib_core/libc/arm/div.s 169⟩≡

```
Q = 0
N = 1
D = 2
CC = 3
TMP = 11

TEXT save<>(SB), 1, $0
 MOVW R(Q), 0(FP)
 MOVW R(N), 4(FP)
 MOVW R(D), 8(FP)
 MOVW R(CC), 12(FP)

 MOVW R(TMP), R(Q)  /* numerator */
 MOVW 20(FP), R(D)  /* denominator */
 CMP $0, R(D)
 BNE s1
 MOVW -1(R(D)), R(TMP) /* divide by zero fault */
s1: RET

TEXT rest<>(SB), 1, $0
 MOVW 0(FP), R(Q)
 MOVW 4(FP), R(N)
 MOVW 8(FP), R(D)
 MOVW 12(FP), R(CC)
/*
 * return to caller
 * of rest<>
 */
 MOVW 0(R13), R14
 ADD $20, R13
 B (R14)

TEXT div<>(SB), 1, $0
 MOVW $32, R(CC)
/*
 * skip zeros 8-at-a-time
 */
e1:
 AND.S $(0xff<<24),R(Q), R(N)
 BNE e2
 SLL $8, R(Q)
 SUB.S $8, R(CC)
 BNE e1
 RET
e2:
 MOVW $0, R(N)

loop:
/*
 * shift R(N||Q) left one
 */
```

```
 SLL $1, R(N)
 CMP $0, R(Q)
 ORR.LT $1, R(N)
 SLL $1, R(Q)

/*
 * compare numerator to denominator
 * if less, subtract and set quotent bit
 */
 CMP R(D), R(N)
 ORR.HS $1, R(Q)
 SUB.HS R(D), R(N)
 SUB.S $1, R(CC)
 BNE loop
 RET

TEXT _div(SB), 1, $16
 BL save<>(SB)
 CMP $0, R(Q)
 BGE d1
 RSB $0, R(Q), R(Q)
 CMP $0, R(D)
 BGE d2
 RSB $0, R(D), R(D)
d0:
 BL div<>(SB)  /* none/both neg */
 MOVW R(Q), R(TMP)
 B out
d1:
 CMP $0, R(D)
 BGE d0
 RSB $0, R(D), R(D)
d2:
 BL div<>(SB)  /* one neg */
 RSB $0, R(Q), R(TMP)
 B out

TEXT _mod(SB), 1, $16
 BL save<>(SB)
 CMP $0, R(D)
 RSB.LT $0, R(D), R(D)
 CMP $0, R(Q)
 BGE m1
 RSB $0, R(Q), R(Q)
 BL div<>(SB)  /* neg numerator */
 RSB $0, R(N), R(TMP)
 B out
m1:
 BL div<>(SB)  /* pos numerator */
 MOVW R(N), R(TMP)
 B out

TEXT _divu(SB), 1, $16
 BL save<>(SB)
 BL div<>(SB)
 MOVW R(Q), R(TMP)
 B out

TEXT _modu(SB), 1, $16
 BL save<>(SB)
```

```
 BL div<>(SB)
 MOVW R(N), R(TMP)
 B out

out:
 BL rest<>(SB)
 B out
```

Glossary

LOC = Lines Of Code
ISA = Instruction Set Architecture
AST = Abstract Syntax Tree
PC = Program Counter (register)
SP = Stack Pointer (register)
FP = Frame Pointer (register)
SB = Static Base (register)
BL = Branch and Link
RISC = Reduced Instruction Set Computer
CISC = Complex Instruction Set Computer
ARM = Acorn Risc Machines
CWD = Current Working Directory
EOF = End Of File
IGN = Ignore

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.

alloc(): 40d, 52b, 115g, 141d, 151b
allocn(): 45b, 133b, 151c
Always: 47b, 103a, 103b, 109a, 119b
assemble(): 45d
bcode-56: 109a, 109b
BUFSIZ: 53b, 53g
cclean(): 45d, 46b, 47b
cinit(): 44c
debug: 146a, 146d, 146e, 146f, 147a, 147b
Dlist: 132b, 132e
dodefine(): 132e, 132h
domacro(): 69a, 70a
ehist: 115e, 115g, 141d
EOF: 55a, 55b, 56b, 58d, 63b, 65a, 65b, 66e, 67c, 68
errorexit(): 46a, 52c, 52e, 58d, 67c, 75a, 148a, 148c, 150e
escchar(): 63b, 65b, 68
FD_NONE: 50a, 51b, 51g, 52d
Fi: 53c, 53d
fi: 52d, 53c, 53g, 54a, 54b, 74b, 133b, 136c
Fi.c: 53d
Fi.p: 53d
filbuf(): 53g, 133b
FPCHIP: 62c, 123a
Gen: 33b, 167a
Gen.reg: 33b
Gen.sym: 42b
Gen.symkind: 42c
Gen.type: 33b
Gen (typedef): 167a
GETC: 53f, 56b, 58d, 59c, 60d, 61b, 62c, 63a, 67c
getc(): 65b, 66d, 67c, 70b, 71b, 75e, 76e, 78c, 79b, 79e, 133b, 136b, 137, 140, 141d
getcom(): 76d, 76e, 79d, 140, 141d
gethunk(): 150e, 151b, 151c
getnsc(): 70b, 71b, 72b, 75e, 76e, 79e, 133b, 136b, 137, 141b, 141d
getnsn(): 78c, 79e
getsym(): 70a, 70b, 133b, 136b, 139, 140, 141b
getsymdots(): 133b, 136b
H: 115d, 115g, 118c, 119b, 141d

h: 110b, 111b, 112c
hash: 39a, 40c, 40d, 41d, 132g
Hist: 156b, 156b
hist: 115b, 115g, 118c, 119b, 141d
Hist (typedef): 156b
HISTSZ: 118a, 118b, 118c, 118d, 119a
Htab: 110b, 110d
Htab.sym: 110d
Htab.symkind: 110d
hunk: 133b, 150a, 150e, 151b, 151c
I: 51c, 51e, 52a, 52b, 55b, 75a
ieeedtod(): 126a, 126b, 126b
IGN: 57a, 57b, 57c, 57d, 67d, 136c
include: 72c, 72g, 73c, 73e, 73f, 76a
Io: 156b, 156b
Io (typedef): 156b
iofree: 52a, 52a, 52b, 54b
ionext: 51f, 51g, 52d, 136c
iostack: 51c, 51c, 52d, 53g, 54b, 74b, 136c
Itab: 37b, 37c
itab: 37c, 41d
Itab.name: 37b
Itab.type: 37b
Itab.value: 37b
LARITH: 82e, 84a, 84c, 84e, 85a, 85c
LARITHFLOAT: 123f
LAT: 96c
LBCOND: 89a
LBRANCH: 87e, 90d
LC: 130d
LCMP: 88c, 90b
LCMPFLOAT: 123f
LCOND: 103a
LCONST: 60d, 61b, 62c, 63b
LCREG: 130d
LDATA: 101b
LDEF: 100e
LEND: 102e
LF: 124a
LFCONST: 62c
LFCR: 124b
LFP: 98b
LFREG: 124a
linehist(): 115f, 115g, 116a, 116b
lineno: 50a, 50b, 58c, 58d, 67c, 71d, 108, 115g, 133b, 141d, 146d, 148c
LLAB: 60a
LMISC: 37c
LMOV: 86b, 123d
LMOVM: 128a

LMULA: 126d
LMULL: 127e
LMVN: 85e
LNAME: 40e, 60a
lookup(): 39e, 40c, 59c, 70b, 132h
LPC: 100c
LPSR: 129c
LR: 94a
LREG: 93g
LRET: 91a, 91g
LS: 103e, 103g, 128e, 128g, 129a, 129b, 129h
LSB: 98b
LSCONST: 68
LSP: 98b
LSQRTFLOAT: 123f
LSWAP: 87b
LSWI: 91d
LSYSTEM: 129i
LVAR: 60a
LWORD: 102b
macdef(): 69b, 133b
macend(): 69b, 72a, 72b, 75f, 79c, 133b, 139, 140, 141d
macexpand(): 136c, 137
macif(): 70a, 140
macinc(): 69b, 74a
maclin(): 69b, 78a
macprag(): 69b, 141b
mactab: 69b, 70a
macund(): 69b, 139
main-57(): 155a
MAXALIGN: 151a, 151b
mycreat(): 45d, 159a
mywait(): 159b
NARG-2: 133a, 133b, 137
nDlist: 132c, 132e
nerrors: 45d, 46b, 148b, 148b, 148c
newfile(): 50a, 52d, 74a
newio(): 50a, 51g, 74a, 136c
NHASH: 39a, 39b, 41b, 41d, 132g
NHUNK: 150d, 150e
nhunk: 150b, 150b, 150e, 151b, 151c
NINCLUDE: 72c, 72d
ninclude: 72e, 73c, 73e, 73f, 76a
NSYMB: 40a, 40b, 70b, 71a
nullgen: 34a, 44d, 47b, 68, 119b
obuf: 45d, 47a, 47b, 108, 109c, 110a, 111c, 119b, 120a, 126a, 127c
outcode(): 47b, 108
outfile: 44a, 44a, 45d, 46a, 49, 148a
outhist(): 45d, 119b

outopd(): 108, 109c, 119b
pass: 45c, 45d, 108
pathchar(): 159d
pathname: 45a, 45b, 120d
pc: 92a, 92b, 108
peekc: 57a, 57a, 57c, 57d, 58b, 59c, 60d, 61b, 62c, 63a, 66d, 67d, 71d, 136c
pinit(): 45d, 50a
Plan9: 156a, 159c
prfile(): 118b, 148c
pushio(): 74a, 74b, 136c
S: 39d, 40c, 41d, 44d, 70b, 111b, 112c, 132g, 133b, 136b, 139, 140, 141a
setinclude(): 73c, 73h
slookup(): 39e, 41d, 136b, 141a
STRINGSZ: 75c, 75d
Sym: 156b, 156b
Sym (typedef): 156b
symb: 39e, 40a, 40c, 40d, 41b, 59c, 60c, 60d, 62c, 63a, 70b, 71a, 74a, 76a, 76c, 78a, 78c, 79a, 132h, 133b, 141d, 149
symcounter: 111a, 111b, 112b, 112c, 112e
symidx_of_symopt(): 112b, 112c
syminit(): 40d, 40e
systemtype(): 159c
thechar: 43a, 46a, 49, 52e
thestring: 43b, 73h
thunk: 150c, 150e
unget(): 70b, 71d, 75f, 79c, 79e, 133b, 136b, 137, 141d
Unix: 156a
VARMAC-1: 133b, 136a, 137
yyerror(): 46a, 52c, 52e, 58d, 62c, 63b, 66e, 67c, 68, 71a, 72a, 73f, 75a, 75f, 76e, 79c, 133b, 136b, 137, 139, 140, 141d, 148c, 149, 150e
yylex(): 56b
yylval: 60a, 60d, 61b, 62c, 63b, 68
yyparse(): 45d
zname(): 111c, 112c
__anon_enum_1: 156a
__anon_struct_1.macf: 69b
__anon_struct_1.macname: 69b
__anon_struct_1: 69b, 69b
__anon_struct_2.dval: 33b
__anon_struct_2.offset: 33b
__anon_struct_2.sval: 33b
__anon_struct_2: 33b, 33b

Bibliography

[BLM92] Doug Brown, John Levine, and Tony Mason. *Lex and Yacc*. O'Reilly, 1992. cited page(s) 11, 28, 35, 81, 82

[Dun00] Jeff Duntemann. *Assembly Language Step-by-step: Programming with DOS and Linux*. Wiley, 2000. cited page(s) 8, 11

[EF00] Dean Elsner and Jay Fenlason. *Using as, The GNU Assembler*. Free Software Foundation, 2000. Available at https://sourceware.org/binutils/docs-2.30/as/index.html/. cited page(s) 8

[Joh79] Stephen C. Johnson. Yacc: Yet another compiler-compiler. In *Unix Programmer's Manual Vol 2b*, 1979. Also available at generators/docs/yacc.pdf. cited page(s) 7, 28

[Knu92] Donald E. Knuth. *Literate Programming*. Center for the Study of Language and Information, 1992. cited page(s) 11

[Knu99] Donald E. Knuth. *MMIXWare, a RISC Computer for the Third Millenium*. Springer, 1999. cited page(s) 8

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988. cited page(s) 11

[LS79] M. E. Lesk and E. Schmidt. Lex — a lexical analyzer generator. In *Unix Programmer's Manual Vol 2b*, 1979. Also available at generators/docs/lex.pdf. cited page(s) 28, 56

[NS05] Noam Nisan and Shimon Shocken. *The Elements of Computing Systems*. MIT Press, 2005. cited page(s) 83

[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) 11, 19, 145

[Pad15a] Yoann Padioleau. *Principia Softwarica: The ARM Emulator 5i*. 2015. cited page(s) 9, 11, 19, 24, 32, 82, 83, 84, 85, 94, 96, 103, 145

[Pad15b] Yoann Padioleau. *Principia Softwarica: The ARM Linker 5l*. 2015. cited page(s) 8, 9, 17, 21, 26, 34, 47, 86, 94, 105, 106, 144

[Pad16a] Yoann Padioleau. *Principia Softwarica: (OCaml)Lex and (OCaml)Yacc*. 2016. cited page(s) 81

[Pad16b] Yoann Padioleau. *Principia Softwarica: The Plan 9 C Compiler 5c*. 2016. cited page(s) 11, 32, 69, 145

[Pad16c] Yoann Padioleau. *Principia Softwarica: The Plan 9 Core Libraries*. 2016. cited page(s) 15, 17, 44, 49, 61, 63

[Pad16d] Yoann Padioleau. *Principia Softwarica: The Plan 9 Debuggers and Tracers.* 2016. cited page(s) 114, 145

[PH13] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface.* Morgan Kaufmann, 2013. cited page(s) 11

[PH16] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface (ARM Edition).* Morgan Kaufmann, 2016. cited page(s) 11

[Pik93] Rob Pike. A manual for the plan 9 assembler. Technical report, Bell Labs, 1993. Also available at `assemblers/docs/asm.pdf`. cited page(s) 7, 11

[Rit79] Dennis M. Ritchie. Assembler reference manual. In *Unix Programmer's Manual Vol 2b*, 1979. cited page(s) 8

[Sal93] David Salomon. *Assemblers and Loaders.* Ellis Horwood Ltd, 1993. Out of print but available at `http://www.davidsalomon.name/assem.advertis/AssemAd.html`. cited page(s) 11

[Sea01] David Seal. *ARM, Architecture Reference Manual.* Addison-Wesley, 2001. cited page(s) 8, 24

[Tan88] Andrew S Tanenbaum. *Structured Computer Organization.* Prentice Hall, 1988. cited page(s) 11