

# Principia Softwarica: The ARM Assembler 5a

version 0.5

Yoann Padioleau  
`yoann.padioleau@gmail.com`

with code from  
Rob Pike

March 24, 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

<b>1</b>	<b>Introduction</b>	<b>7</b>
1.1	Motivations	7
1.2	The Plan 9 ARM assembler: <code>5a</code>	7
1.3	Other assemblers	8
1.4	Getting started	9
1.5	Requirements	10
1.6	About this document	11
1.7	Copyright	11
1.8	Acknowledgments	11
<b>2</b>	<b>Overview</b>	<b>12</b>
2.1	Assembler principles	12
2.2	<code>5a</code> command-line interface	13
2.3	<code>helloworld.s</code>	13
2.3.1	Background	13
2.3.2	The program	14
2.3.3	Pseudo instructions	15
2.3.4	Labels	16
2.3.5	Assignments	16
2.3.6	Addressing modes	16
2.3.7	Pseudo registers	16
2.3.8	Call stack	17
2.3.9	System calls	18
2.3.10	Function calls	19
2.3.11	Virtual instructions	19
2.3.12	C calling conventions	20
2.3.13	Data layout	21
2.4	The ARM architecture	22
2.5	Input assembly language	22
2.5.1	Lexical elements	22
2.5.2	Syntactical elements	23
2.5.3	Opcodes	23
2.5.4	Operands	23
2.5.5	Addressing modes	23
2.5.6	Advanced features	24
2.6	Output object format	24
2.7	Code organization	25
2.8	Software architecture	26
2.9	Book structure	28

<b>3</b>	<b>Core Data Structures</b>	<b>29</b>
3.1	Opcode	29
3.2	Register	30
3.3	Operand	31
3.4	Instructions	32
3.5	Tokens and itab	33
3.6	Symbols and hash table	35
3.7	Sym_kind	39
<b>4</b>	<b>Main Functions</b>	<b>41</b>
4.1	main()	41
4.2	cinit()	42
4.3	assemble()	43
<b>5</b>	<b>Input</b>	<b>47</b>
5.1	pinit()	47
5.2	File management: iostack	47
5.3	Buffer management: fi	50
5.4	GETC()	50
<b>6</b>	<b>Lexing</b>	<b>53</b>
6.1	yylex()	53
6.2	Peek, seek, and look ahead	54
6.3	Newlines (and semicolons)	55
6.4	Comments	55
6.5	Mnemonics, symbols, and labels	56
6.6	Numbers	57
6.6.1	Decimal numbers	57
6.6.2	Hexadecimal and octal numbers	58
6.6.3	Floating-point numbers	59
6.7	Characters	60
6.7.1	Escaped sequences	60
6.7.2	Triple quotes	62
6.7.3	escchar()	62
6.7.4	Escaped newlines	63
6.7.5	getc()	64
6.8	Strings	64
<b>7</b>	<b>Preprocessing</b>	<b>66</b>
7.1	Directives dispatch: mactab, and domacro()	66
7.2	#include	69
7.2.1	Include search path	69
7.2.2	-I	70
7.2.3	System paths	70
7.2.4	macinc()	71
7.3	#line	74
7.3.1	Motivations	74
7.3.2	maclin()	75

<b>8</b>	<b>Parsing</b>	<b>78</b>
8.1	Overview	78
8.2	Instructions	79
8.2.1	Arithmetic and logic	79
8.2.2	Memory	82
8.2.3	Control flow	84
8.2.4	Software interrupt	88
8.3	Label definitions and pc	88
8.4	Operands	89
8.4.1	Registers	90
8.4.2	Immediate constants	91
8.4.3	ARM shifted registers	92
8.4.4	Memory (de)references, pointers	93
8.4.5	Named memory locations, symbols	94
8.4.6	Code references, labels	96
8.5	Pseudo instructions	97
8.5.1	TEXT/GLOBL	97
8.5.2	WORD/DATA	98
8.5.3	END	99
8.6	ARM conditional execution	99
8.7	Special bits	100
8.7.1	Arithmetic instructions and .S	100
8.7.2	Moves and .P/.W	100
8.8	yyparse()	101
<b>9</b>	<b>Object Code Generation</b>	<b>102</b>
9.1	Object format	102
9.2	Instruction output: outcode()	104
9.3	Operand output: outopd()	105
9.4	Object file symbol table: h and ANAME	106
<b>10</b>	<b>Debugging Support</b>	<b>109</b>
10.1	Line origin history: Hist	109
10.2	Recording history: linehist()	110
10.3	Displaying history: prfile()	111
10.4	Saving history: outhist() and AHISTORY	114
<b>11</b>	<b>Advanced Topics TODO</b>	<b>116</b>
11.1	Other assembly language features	116
11.1.1	Constant expressions	116
11.1.2	Symbolic constants	117
11.2	Other instructions and registers	117
11.2.1	Floating-point numbers	117
11.2.2	Multiplication and accumulation	121
11.2.3	64-bits multiplication	121
11.2.4	Moving multiple registers at the same time	122
11.2.5	Program status register	123
11.2.6	Mutual exclusion instructions	123
11.2.7	Coprocessors	123
11.3	Other pseudo opcodes	125

11.3.1	Compiler-only opcodes	125
11.3.2	Linker-only opcodes	125
11.4	TEXT attributes	125
11.5	Other preprocessing directives	126
11.5.1	#define	126
11.5.2	#undef	133
11.5.3	#ifdef	134
11.5.4	#pragma	135
11.6	#pragma lib and automagic linking	135
11.7	Processing multiple files	136
<b>12</b>	<b>Conclusion</b>	<b>138</b>
12.1	Patterns and techniques	138
12.2	Connections to other books	138
12.3	Beyond the Plan 9 assembler	139
<b>A</b>	<b>Debugging</b>	<b>140</b>
A.1	Line information debugging: 5a -f	140
A.2	Macro debugging: 5a -m	140
<b>B</b>	<b>Error Management</b>	<b>142</b>
<b>C</b>	<b>Utilities</b>	<b>144</b>
C.1	Buffer management	144
C.2	Memory management	144
<b>D</b>	<b>Examples of Assembly Programs TODO</b>	<b>146</b>
D.1	hello.s and pwrite.s	146
D.2	memset.s	146
D.3	div.s	146
D.4	main9.s	147
D.5	tas.s	147
D.6	getcallerpc.s	147
<b>E</b>	<b>Extra Code</b>	<b>148</b>
E.1	include/	148
E.1.1	include/obj/common.out.h	148
E.1.2	include/obj/5.out.h	148
E.2	assemblers/misc/	149
E.2.1	assemblers/misc/data2s.c	149
E.3	assemblers/aa/	150
E.3.1	assemblers/aa/aa.h	150
E.3.2	assemblers/aa/utils.c	152
E.3.3	assemblers/aa/globals.c	154
E.3.4	assemblers/aa/lookup.c	154
E.3.5	assemblers/aa/error.c	154
E.3.6	assemblers/aa/float.c	155
E.3.7	assemblers/aa/hist.c	155
E.3.8	assemblers/aa/lexbody.c	155
E.3.9	assemblers/aa/macbody.c	160
E.4	assemblers/5a/	161

E.4.1	assemblers/5a/a.h	161
E.4.2	assemblers/5a/globals.c	161
E.4.3	assemblers/5a/lex.c	162
E.4.4	assemblers/5a/a.y	162
E.4.5	assemblers/5a/obj.c	162
E.4.6	assemblers/5a/main.c	162
E.5	lib_core/libc/arm/	163
E.5.1	lib_core/libc/arm/div.s	163

<b>Glossary</b>	<b>166</b>
-----------------	------------

<b>Indexes</b>	<b>167</b>
----------------	------------

<b>References</b>	<b>170</b>
-------------------	------------

# 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]<sup>1</sup>, 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].

---

<sup>1</sup>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<sup>2</sup>. 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`<sup>3</sup>, was targeting 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<sup>4</sup>, 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<sup>5</sup> (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]<sup>6</sup> 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<sup>7</sup> 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.

---

<sup>2</sup>Readers interested in this topic should read instead the LINKER book [Pad15b].

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

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

<sup>5</sup><http://blog.llvm.org/2010/04/intro-to-llvm-mc-project.html>

<sup>6</sup><http://www.nasm.us/>

<sup>7</sup><http://v3.sk/~lkundrak/dev86/>

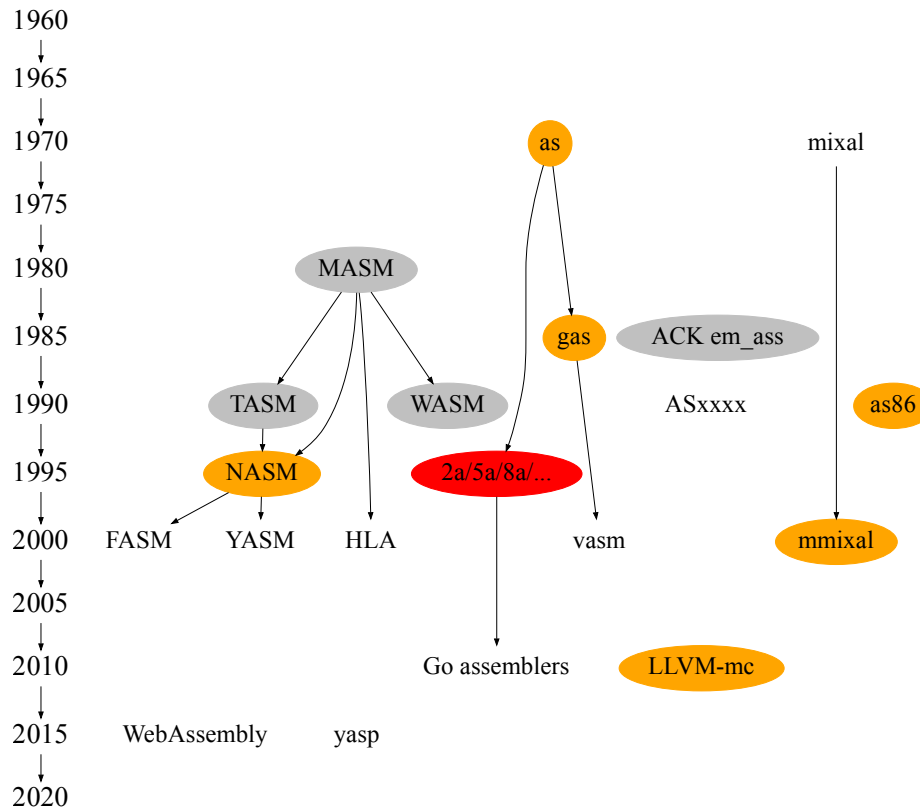


Figure 1.1: Assemblers timeline

- MMIXAL [Knu99]<sup>8</sup> 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 (iterate) 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 `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<sup>9</sup>. 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<sup>10</sup>, so you can cross-assemble ARM programs from your regular machine. Once installed, you can test 5a under Plan 9 with the following commands:

<sup>8</sup><https://www-cs-faculty.stanford.edu/~knuth/mmixware.html>

<sup>9</sup><https://golang.org/doc/asm>

<sup>10</sup><https://github.com/aryx/goken9cc>

```
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<sup>11</sup>).

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.

## 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`<sup>12</sup> in my Plan 9 repository. You can also consult the documentation of the Plan 9 assemblers [Pik93] (available also in `assemblers/docs/asm.pdf`).

---

<sup>11</sup><https://www.raspberrypi.org/>

<sup>12</sup>Despite its name, this document covers also 5a.

## 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.

# 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

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.

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.

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.

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

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.

## 2.2 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 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 14a>≡
#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 14b>≡
#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/pwrite()`. `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 14c>≡
extern long pwrite(fdt, void*, long, vlong);
```

The `pwrite()` interface is documented in `docs/man/2/read`<sup>1</sup>. 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 14d>≡
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
```

---

<sup>1</sup>Again, despite its name, this document covers also `pwrite`.

```

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
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<sup>2</sup>. 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, 0(B)` roughly corresponds to the following C statement `B[0] = N`, if B is a byte pointer. This instruction is also equivalent to this C statement `*(B+0) = N`. 0 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

---

<sup>2</sup>This syntax is called the AT&T syntax, as opposed to the Intel syntax which is right-to-left.

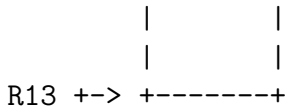


Figure 2.1: Stack content when `hello` started.

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.

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`<sup>3</sup>:

```
$ 5i hello
5i> $r
...
R0 #7ffffff70 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 #7ffffff58 R14 #0         R15 #1020
```

<sup>3</sup>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`.

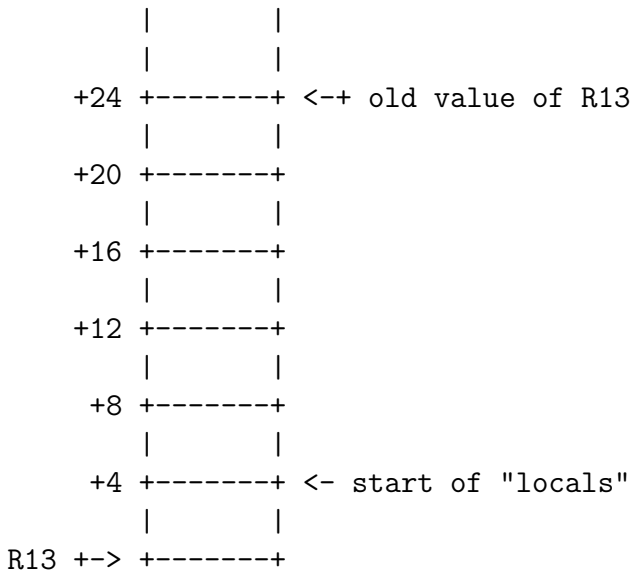


Figure 2.2: Stack content before Line 2.

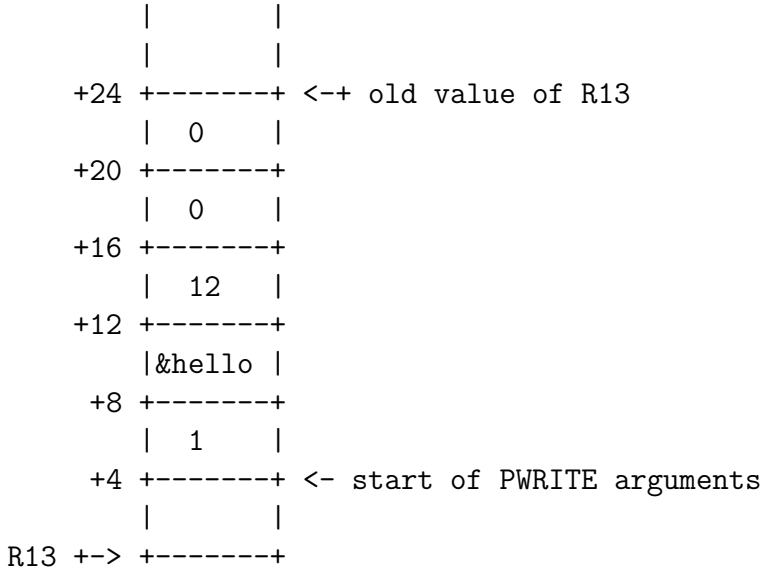


Figure 2.3: Stack content before Line 31.

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.

### 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 51. Indeed, 51 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`<sup>4</sup>. `_main` in `helloworld.s` on the opposite is not a leaf function because it is using BL Line 32.

The machine instructions generated by 51 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 51 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, 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 `exit(0)`, which terminates the program. This is why I marked a few instructions with a comment indicating

---

<sup>4</sup>SWI does not count as a function call; R14 is not overwritten by a SWI.

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.

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 51).

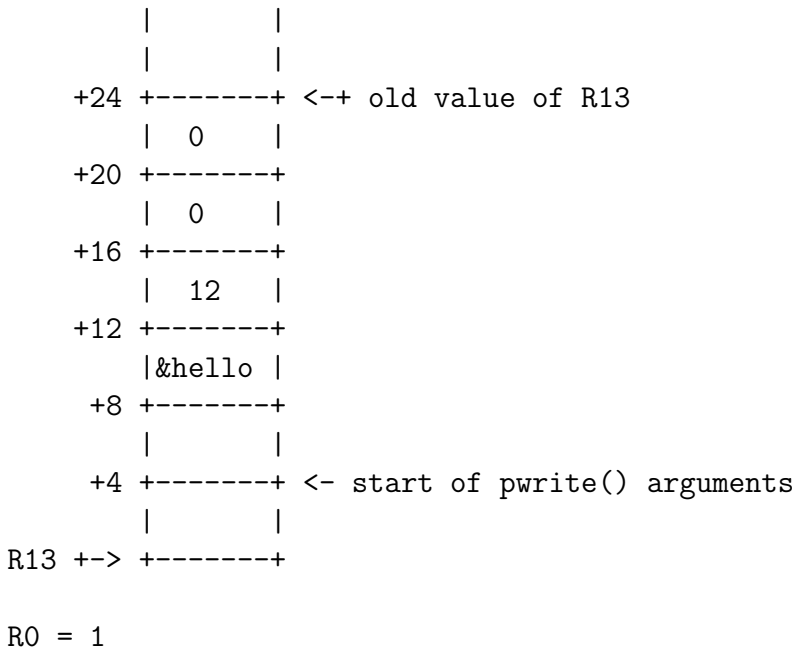


Figure 2.5: Stack and R0 value for a C call to `pwrite()`.

### 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 21a⟩≡
...
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 21b⟩≡
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<sup>5</sup>. This family powers most smartphones, as well as the Raspberry Pi<sup>6</sup> (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.

<sup>5</sup><http://www.arm.com/products/processors/classic/arm11/?tab=Specifications>

<sup>6</sup><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.

- 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.
- 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 `MCCR` 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 MCCR 24>≡
#define MCCR(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).

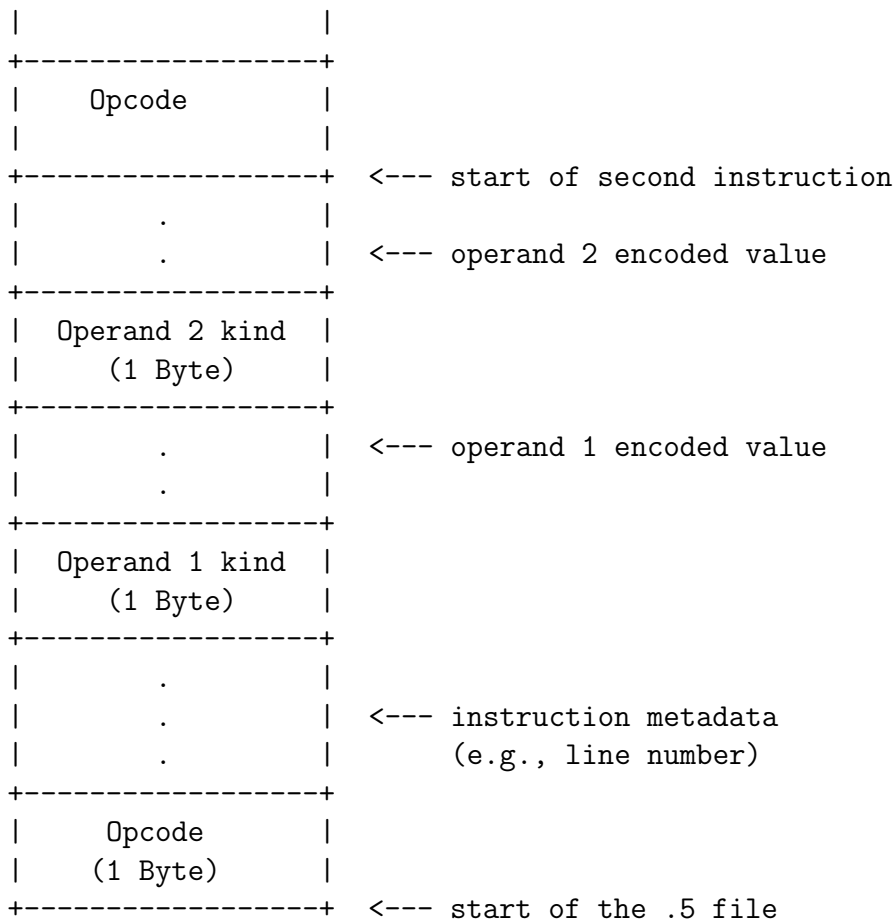


Figure 2.6: File format of a .5 object file.

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) <sup>7</sup>.

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

<sup>7</sup>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`<sup>148c8</sup>.

Function	Ch.	File	Entities	LOC
ARM opcodes	3	<code>include/obj/5.out.h</code>	Opcode <sup>29</sup> Operand_kind Register <sup>30</sup>	306
ARM operand	3	<code>5a/a.h</code>	Gen <sup>31b</sup>	70
ARM-specific globals	3	<code>5a/globals.c</code>	nullgen <sup>32a</sup>	7
general data structures	3	<code>aa/aa.h</code>	Sym <sup>150b</sup> Io <sup>150b</sup> Fi <sup>50d</sup>	283
general globals	3	<code>aa/globals.c</code>	hash <sup>36a</sup> symb <sup>37a</sup> outfile <sup>42a</sup> pass <sup>43c</sup> pc <sup>89a</sup>	107
symbol table lookup	3	<code>aa/lookup.c</code>	slookup() <sup>36e</sup> lookup() <sup>37c</sup>	62
main functions	4	<code>5a/main.c</code>	main() <sup>149a</sup> assemble() <sup>43d</sup>	207
IO and lexing utilities	5	<code>aa/lexbody.c</code>	pinit() <sup>47a</sup> newio() <sup>48g</sup> newfile() <sup>49d</sup>	540
lexer for Asm5	6	<code>5a/lex.c</code>	yylex() <sup>53b</sup> cinit() <sup>42c</sup>	590
macro processor	7	<code>aa/macbody.c</code>	mactab <sup>66b</sup> macinc() <sup>71a</sup> macdef() <sup>127b</sup>	846
grammar for Asm5	8	<code>5a/a.y</code>	inst() <sup>80a</sup> yyparse() <sup>101b</sup>	600
object file generation	9	<code>5a/obj.c</code>	outcode() <sup>104a</sup>	278
location information	10	<code>aa/hist.c</code>	linehist() <sup>110g</sup> prfile() <sup>113b</sup>	95
float data structure	11	<code>include/obj/common.out.h</code>	ieee	22
float utilities	11	<code>aa/float.c</code>	ieeedtod() <sup>120d</sup>	35
error management	B	<code>aa/error.c</code>	errorexit() <sup>142a</sup> yyerror() <sup>142c</sup>	50
utilities	C	<code>aa/utills.c</code>	alloc() <sup>145b</sup>	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()`<sup>149a</sup> calls `assemble()`<sup>43d</sup> with the name of the assembly file to process. `assemble()`, after further initializations, uses a two-pass algorithm and so calls `yyparse()`<sup>101b</sup>, the function generated by Yacc [Joh79] from the Asm5 grammar, two times. It also modifies the global `pass`<sup>43c</sup>; 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()`<sup>53b</sup> 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()`<sup>71a</sup> (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*<sup>9</sup> associated with those grammar rules

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

<sup>9</sup>See [BLM92] for more information on Yacc’s grammar rules and actions.

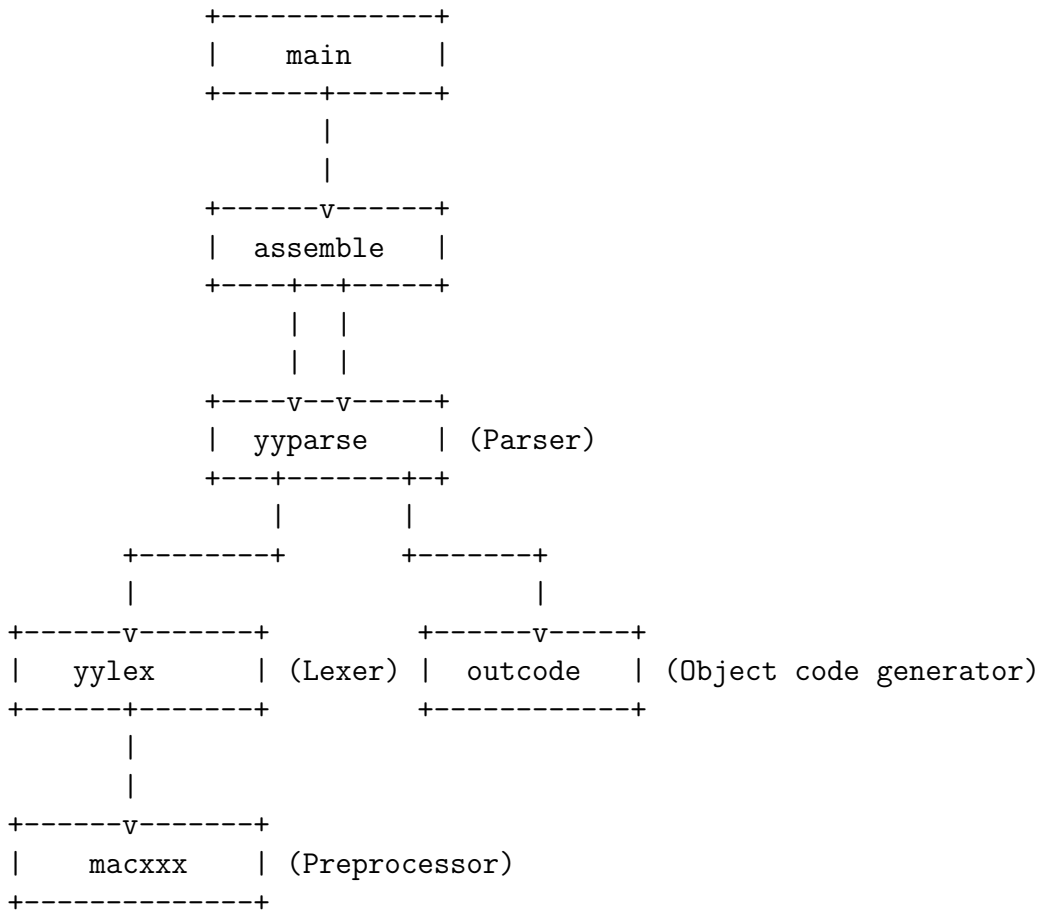


Figure 2.7: Control flow diagram of 5a.

```

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`<sup>80a</sup> grammar rule, the Yacc action for `inst` is triggered. This action calls the function `outcode()`<sup>104a</sup> 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`<sup>89a</sup>, 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`<sup>36a</sup> 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()`<sup>149a</sup>, I will present the code of the main functions of 5a (e.g., `assemble()`<sup>43d</sup>). 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`<sup>148c</sup> by the following type:

```
<enum Opcode(arm) 29>≡ (148c)
// coupling: with 5c/enam.c
enum Opcode
{
    AXXX,

    ANOP, // VIRTUAL removed by linker
    // -----
    // Arithmetic and logic opcodes
    // -----
    <Opcode cases, logic opcodes 79d>
    <Opcode cases, add/sub opcodes 80b>
    <Opcode cases, mul/div/mod opcodes 81b>
    <Opcode cases, bitshift opcodes 81d>
    // -----
    // Memory opcodes
    // -----
    <Opcode cases, mov opcodes 83a>
    <Opcode cases, swap opcodes 84a>
    // -----
    // Control flow opcodes
    // -----
    <Opcode cases, comparison opcodes 85b>
    <Opcode cases, branching opcodes 84d>
    // -----
    // System opcodes
```

```

// -----
⟨Opcode cases, interrupt opcodes 88c⟩
// -----
// Float opcodes
// -----
⟨Opcode cases, float mov opcodes 117e⟩
⟨Opcode cases, float arithmetic opcodes 118a⟩
// -----
// Pseudo opcodes
// -----
⟨Opcode cases, pseudo opcodes 97c⟩

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 51. 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) 30⟩≡ (148c)
enum Register {
    ⟨Register compiler conventions cases 125e⟩
    ⟨Register linker conventions cases 125g⟩

    // 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 that 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) 31a>≡ (148c)
#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) 31b>≡ (161a)
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 39d>
};
```

Uses `__anon_struct_2 31b`.

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) 31c>≡ (148c)
enum Operand_kind {
    D_NONE,

    D_CONST,
    D_SCONST,
    D_FCONST,

    D_REG,
    <Operand_kind cases 32b>
};
```

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 find in the code of 5a. It represents the null operand. It is also used to initialize freshly allocated operands.

```
<global nullgen 32a>≡ (161b)
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 32b>≡ (31c) 39g▷
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 32c>≡ (148c)
#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()`<sup>104a</sup> function below to represent the different components of an instruction. Here is the signature of `outcode()`:

```
<signature outcode(arm) 32d>≡ (161a)
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);
```

The `Always`<sup>100a</sup> 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`<sup>31a</sup>,
- 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()`<sup>101b</sup>, is taking as input a set of tokens. Those tokens are returned by the `yylex()`<sup>53b</sup> function but the type of those tokens is defined by a set of *Yacc directives* in the Yacc grammar 5a/a.y<sup>1</sup>. Here are the directives to declare the names of those tokens as well as the type of value they hold:

```
(token declarations(arm) 33a)≡ (78a)
/* 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
```

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`<sup>33a</sup>), 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 33b)≡
...
#define LARITH 57346
...
#define LSWI 57352
#define LRET 57353
...
#define LCONST 57381
#define LFCONST 57382
...
```

---

<sup>1</sup>Again, see [BLM92] for more information on Yacc.

	token code	token 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.

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) 35a⟩≡ (78a)
%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 LCONST
    char sval[NSNAME]; // for LCONST

    ⟨union declaration other fields(arm) 39c⟩
}

```

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 `LCONST33a` 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) 35b⟩≡ (162a)
struct Itab
{
    char *name;

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

⟨global itab(arm) 35c⟩≡ (162a)
// map<string, (token_code * token_value)>
struct Itab itab[] =
{
    "NOP", LMISC, ANOP,
    ⟨itab elements 79e⟩
    0
};

```

Uses `Itab 35b` 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`.

## 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 35d⟩≡ (150b)
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 107f>
<Sym macro fields 126f>
<Sym token fields 39b>

// -----
// Extra
// -----
<Sym extra fields 36c>
};

```

As an example, the label `later` in the `helloworld.s` program in Section 2.3, once resolved, could be represented by the following `Sym`<sup>150b</sup>: `.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 36a>≡ (154a)
// hash<string, ref_own<Sym>> (next = Sym.link in bucket)
Sym* hash[NHASH];

```

Uses `NHASH` 36b.

```

<constant NHASH 36b>≡ (150b)
#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 36c>≡ (35d)
// list<ref<Sym>> (from = hash)
Sym* link;

```

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

```

<constant S 36d>≡ (150b)
#define S      ((Sym*)nil)

```

The main interface to the symbol table are the functions `slookup()` and `lookup()`<sup>37c</sup> 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 36e>≡ (154b)
Sym*
slookup(char *s)
{

```

```

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

```

Uses `lookup()` 37c and `symb` 37a.

`slookup()` 36e modifies the global `symb` (used also by `yylex()` 53b as you will see later) before calling `lookup()`.

```

⟨global symb 37a⟩≡ (154a)
char symb[NSYMB];

```

Uses `NSYMB` 37b.

```

⟨constant NSYMB 37b⟩≡ (150b)
#define NSYMB      500

```

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

```

⟨function lookup 37c⟩≡ (154b)
Sym*
lookup(void)
{
    Sym *sym;
    long h;
    int c;
    int len;
    ⟨lookup() other locals 38b⟩

    ⟨lookup() compute hash value h of symb 38c⟩

    // 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 37d⟩
}

```

Uses `S` 36d, `hash` 36a, and `symb` 37a.

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 37d⟩≡ (37c)
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()` 145b, `hash` 36a, `symb` 37a, and `syminit()` 38a.

```

⟨function syminit 38a⟩≡ (162a)
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 38b⟩≡ (37c)
char *p;

```

```

⟨lookup() compute hash value h of symb 38c⟩≡ (37c)
// 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 36b and symb 37a.

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`<sup>89a</sup> 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 51). 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 9.4.
- *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()`<sup>42c</sup>, resets and then populates `hash` with all the array elements of `itab`<sup>35c</sup> you have seen in the previous section:

```

⟨cinit() locals 38d⟩≡ (42c)
Sym *s;
int i;

```

```

<cinit() hash initialization from itab 39a>≡ (42c)
    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 36b, S 36d, hash 36a, itab 35c, and slookup() 36e.

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 39b>≡ (35d)
    //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`<sup>33a</sup>), labels (`LLAB`<sup>33a</sup>), and symbolic constants (`LVAR`<sup>33a</sup>) is a direct reference to its `Sym` in the symbol table:

```

<union declaration other fields(arm) 39c>≡ (35a) 89f>
    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 39d>≡ (31b) 39e>
    // 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`<sup>31b</sup> field is used to represent the *kind* of the symbol:

```

<Gen other fields 39e>+≡ (31b) <39d
    // option<enum<Sym_kind>> (None = N_NONE)
    short  symkind;

```

```

<enum Sym_kind(arm) 39f>≡ (148c)
    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 96a>
    };

```

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 39g>+≡ (31c) <32b 92d>
    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()`<sup>149a</sup> down to `assemble()`<sup>43d</sup>.

### 4.1 `main()`

Before showing the code of `main()`<sup>149a</sup>, 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 41a>≡ (154a)
int thechar;
```

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

```
<global thestring 41b>≡ (154a)
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()`<sup>43d</sup> at the end, with the filename of the assembly file to process passed through a command-line argument:

```
<function main(arm) 41c>≡ (162c)
void
main(int argc, char *argv[])
{
    errorn err;
    <main() locals 70a>

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

    cinit();
```

```

⟨main() remaining initializations 69f⟩

ARGBEGIN {
⟨main() command line processing 42b⟩
} ARGEND

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

⟨main() multiple files handling 136b⟩
// else

err = assemble(argv[0]);
if(err > 0)
    errexit();
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 42a⟩≡ (154a)
char* outfile = nil;

```

Uses `outfile 42a`.

```

⟨main() command line processing 42b⟩≡ (41c) 70b▷
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()`<sup>149a</sup>, contains the initializations of a few globals:

```

⟨function cinit(arm) 42c⟩≡ (162a)
/// main -> <>
void
cinit(void)
{
    ⟨cinit() locals 38d⟩

    ⟨cinit() nullgen initialization 42d⟩
    ⟨cinit() hash initialization from itab 39a⟩
    ⟨cinit() pathname initialization from cwd 43b⟩
}

```

I mentioned before the global `nullgen`<sup>32a</sup>. Here is the code initializing `nullgen`:

```

⟨cinit() nullgen initialization 42d⟩≡ (42c) ??▷
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 36d` and `nullgen 32a`.

`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 43a>≡ (154a)
char* pathname;
```

```
<cinit() pathname initialization from cwd 43b>≡ (42c)
pathname = allocn(pathname, 0, 100);
if(getwd(pathname, 99) == 0) {
    pathname = allocn(pathname, 100, 900);
    if(getwd(pathname, 999) == 0)
        strcpy(pathname, "///");
}
```

Uses `allocn()` 145c and `pathname` 43a.

## 4.3 assemble()

The most important function called from `main()` <sup>149a</sup> 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 43c>≡ (154a)
// 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()` <sup>104a</sup> as you will see later:

```
<function assemble 43d>≡ (162c)
errorn
assemble(char *infile)
{
    fdt of; // outfile
    <assemble() locals 45a>

    <assemble() set p to basename(infile) and adjust include 45c>
    if(outfile == nil) {
        <assemble() set outfile to basename(infile).thechar 46>
    }
    <assemble() setinclude("/thestring/include") or use INCLUDE 70h>

    of = mycreat(outfile, 0664);
    <assemble() sanity check of 44a>
    Binit(&obuf, of, OWRITE);

    // Pass 1
    pass = 1;
    pinit(infile);
    <assemble() init Dlist after pinit 126e>
    yyparse(); // calls outcode(), which does almost nothing when pass == 1
    <assemble() sanity check errors 44b>

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

```

pinit(infile);
⟨assemble() init Dlist after pinit 126e⟩
yyvsparse(); // calls outcode() which now does things

cclean();
return nerrors;
}

```

Uses `cclean()` 44d, `mycreat()` 153a, `nerrors` 142b, `obuf` 44c, `outfile` 42a, `outhist()` 114b, `pass` 43c, `pinit()` 47a, and `yyvsparse()`.

```

⟨assemble() sanity check of 44a⟩≡ (43d)
if(of < 0) {
    yyerror("%ca: cannot create %s", thechar, outfile);
    errexit();
}

```

Uses `errexit()` 142a, `outfile` 42a, `thechar` 41a, and `yyerror()` 142c.

`assemble()` 43d calls `yyvsparse()` 101b two times, with the appropriate value set in `pass` first, and after having also initialized some globals related to the input file with `pinit()` 47a. This means the input assembly file is parsed two times, but the actions in the grammar do different things the second time.

`assemble()` returns the number of errors it found or zero if everything went fine. The functions called from `assemble()` increment the global `nerrors` 142b to indicate an error, hence the check below:

```

⟨assemble() sanity check nerrors 44b⟩≡ (43d)
if(nerrors) {
    cclean();
    return nerrors;
}

```

Uses `cclean()` 44d and `nerrors` 142b.

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 44c⟩≡ (154a)
Biobuf obuf;

```

This global will be used notably by `outcode()`. The type of `obuf` is `Biobuf` (for “buffered input/output 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` 42a), 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 `yyvsparse()`, the function generated by Yacc, does not take any argument nor return anything, so the interface of `yyvsparse()` 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) 44d⟩≡ (162c)
/// main -> assemble -> <>
void
cclean(void)
{

    outcode(AEND, Always, &nullgen, R_NONE, &nullgen);
    Bflush(&obuf);
}

```

Uses `Always` 100a, `nullgen` 32a, `obuf` 44c, and `outcode()` 104a.

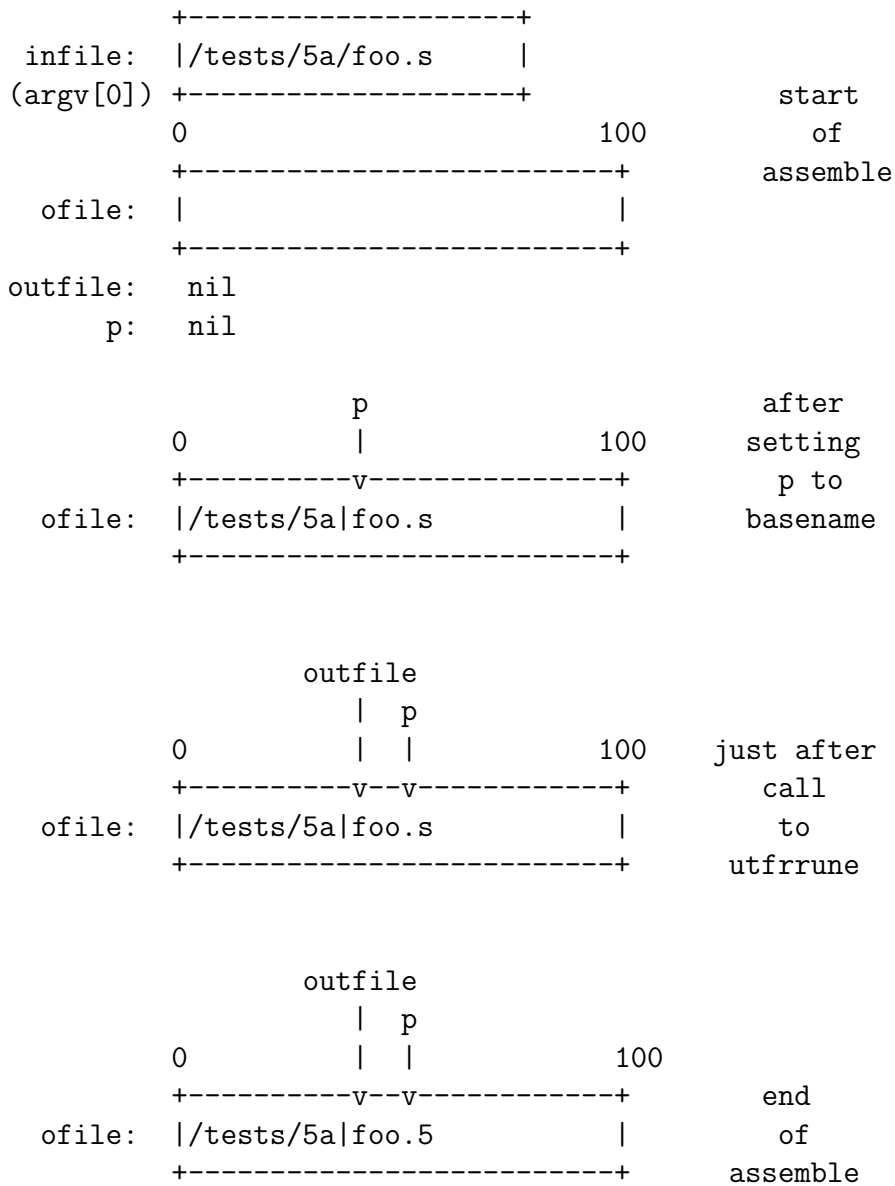


Figure 4.1: Evolution of `ofile` and other variables in `assemble()`.

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 45a⟩≡ (43d) 45b▷
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 45b⟩+≡ (43d) <45a 70g▷
char *p;

```

```

⟨assemble() set p to basename(infile) and adjust include 45c⟩≡ (43d)
// p = basename(infile)
strcpy(ofile, infile);

```

```

p = utfrune(ofile, '/');
if(p) {
    *p++ = '\0';
    <assemble() adjust first entry in include with dirname infile 69g>
} else
    p = ofile;

```

The function `utfrune()` (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 `utfrune()` above and below.

```

<assemble() set outfile to basename(infile).thechar 46>≡ (43d)
// outfile = p =~ s/.s/.5/;
outfile = p;
if(outfile){
    p = utfrune(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 42a` and `thechar 41a`.

# Chapter 5

## Input

Now that you have seen `assemble()`<sup>43d</sup>, 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()`<sup>47a</sup> and `GETC()`<sup>50f</sup>. I mentioned before `pinit()`; it initializes globals related to input. `GETC()`, which is called by the lexer `yylex()`<sup>53b</sup> (called itself by the parser `yyparse()`<sup>101b</sup>), 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()`<sup>43d</sup> (see the code of `assemble()`):

```
<function pinit 47a>≡ (155c)
  /// 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 54c>
  }
```

Uses `FD_NONE` 48b, `lineno` 47b, `newfile()` 49d, and `newio()` 48g.

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

```
<global lineno 47b>≡ (154a)
  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()`<sup>48g</sup> and `newfile()`<sup>49d</sup> in `pinit()`<sup>47a</sup> 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<sup>1</sup>. Because 5a supports

---

<sup>1</sup>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 48a>≡ (150b)
struct Io
{
    // option<fdt> (None = FD_NONE)
    fdt f;
    <Io buffer fields 50a>
    // Extra
    <Io extra fields 48d>
};
```

```
<constant FD_NONE 48b>≡ (150b)
#define FD_NONE (-1)
```

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

```
<global iostack 48c>≡ (154a)
// list<ref_own<Io> (next = Io.link)
Io* iostack = I;
```

Uses I 48e and `iostack` 48c.

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

```
<Io extra fields 48d>≡ (48a)
// list<ref_own<Io>> (from = iostack or iofree)
Io* link;
```

```
<constant I 48e>≡ (150b)
#define I ((Io*)nil)
```

The main job of `newio()`<sup>48g</sup>, called from `pinit()`<sup>47a</sup>, is to make the global `ionext` point to a newly allocated `Io`<sup>150b</sup> that can be used later by `newfile()`<sup>49d</sup> (also called from `pinit()`):

```
<global ionext 48f>≡ (154a)
// option<ref<Io>>
Io* ionext;
```

```
<function newio 48g>≡ (155c)
/// main -> assemble -> pinit -> <>; newfile
void
newio(void)
{
    Io *i;

    <newio() allocate a new Io in i or find a free one 49b>
    ionext = i;
    i->f = FD_NONE;
    i->c = 0;
}
```

Uses `FD_NONE` 48b and `ionext` 48f.

---

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 Ios. By using a free list, 5a can “recycle” an Io for another file:

```
<global iofree 49a>≡ (154a)
```

```
// list<ref<Io>> (next = Io.link)
Io* iofree = I;
```

Uses I 48e and iofree 49a.

```
<newio() allocate a new Io in i or find a free one 49b>≡ (48g)
```

```
static int pushdepth = 0;

i = iofree;
if(i == I) {
    <newio() sanity check depth of macro expansion 49c>
    i = alloc(sizeof(Io));
} else
    // pop(iofree)
    iofree = i->link;
```

Uses I 48e, alloc() 145b, and iofree 49a.

```
<newio() sanity check depth of macro expansion 49c>≡ (49b)
```

```
pushdepth++;
if(pushdepth > 1000) {
    yyerror("macro/io expansion too deep");
    errexit();
}
```

Uses errexit() 142a and yyerror() 142c.

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

```
<function newfile 49d>≡ (155c)
```

```
/// 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 49e>
    fi.c = 0;
    <newfile() call linehist 110f>
}
```

Uses FD\_NONE 48b, fi 50c, ionext 48f, and iostack 48c.

```
<newfile() sanity check i->f 49e>≡ (49d)
```

```
if(i->f < 0) {
    yyerror("%ca: %r: %s", thechar, s);
    errexit();
}
```

Uses errexit() 142a, thechar 41a, and yyerror() 142c.

## 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 50a>≡ (48a) 50e▷  
char b[BUFSIZ];
```

```
<constant BUFSIZ 50b>≡ (150b)  
#define BUFSIZ 8192
```

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

```
<global fi 50c>≡ (154a)  
struct Fi fi;
```

Uses `Fi 50d`.

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

```
<struct Fi 50d>≡ (150b)  
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 50e>+≡ (48a) <50a  
// like Fi, saved pointers in Io.b  
char* p;  
short c;
```

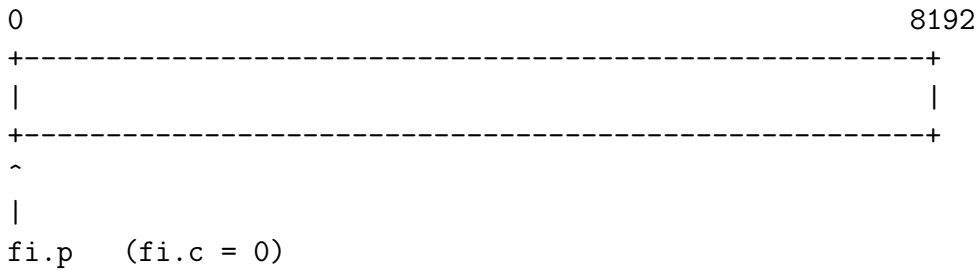
## 5.4 GETC()

I can finally show the code of `GETC()`, which will use the globals `iostack48c` and `fi50c` I mentioned before. `GETC()` is a macro called many times in `yylex()53b` to get the next character from the input file:

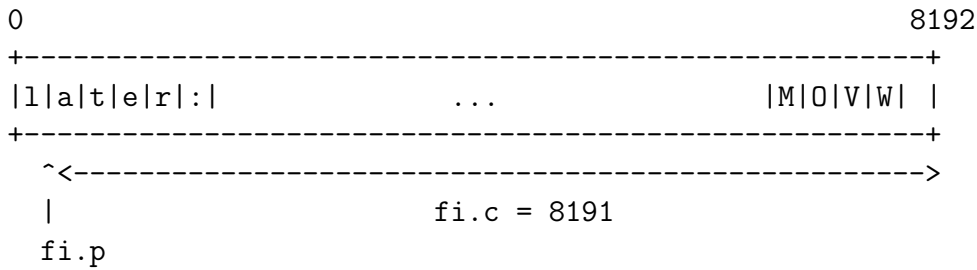
```
<function GETC 50f>≡ (150b)  
// main -> assemble -> yyparse -> yylex -> <>  
#define GETC() ((--fi.c < 0) ? filbuf() : *fi.p++ & 0xff)
```

`GETC()50f` is a complicated macro. Moreover, it relies also on the complicated function below:

```
<function filbuf 50g>≡ (155c)  
int  
filbuf(void)  
{  
    Io *i;  
  
loop:  
    i = iostack;  
    <filbuf() if no more input files in the stack 52b>  
  
    // 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 51a>
// else
fi.p = i->b + 1;
return i->b[0];

```

```

<filbuf() pop 51b>
}

```

Uses BUFSIZ 50b, fi 50c, and iostack 48c.

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<sup>50d</sup> 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()<sup>50g</sup> (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 51a>≡ (50g)
if(fi.c < 0) {
    close(i->f);
    <filbuf() when close file, call linehist 111b>
    goto pop;
}

```

Uses fi 50c.

```

<filbuf() pop 51b>≡ (50g)
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 52b⟩
// 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` 50c, `iofree` 49a, and `iostack` 48c.

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 52a⟩≡ (150b)
#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 52b⟩≡ (51b 50g)
if(i == I)
    return EOF;

```

Uses `EOF` 52a and `I` 48e.

EOF will be used by `yyparse()`<sup>101b</sup> 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()`<sup>53b</sup>, called by the parser function `yyparse()`<sup>101b</sup>. `yylex()` relies on `GETC()`<sup>50f</sup>, 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()`<sup>53b</sup>. 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 53a>≡ (161a)
// unit -> (enum<token_code> | -1 (EOF) | char)
long yylex(void);
```

`yylex()` can also modify the global `yylval`<sup>101a</sup> 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()`<sup>50f</sup>, to loop it uses the label 10: below and some `goto 10` statements, and to transition state it uses more labels and `gotos` as you will see soon. Here is finally the skeleton of `yylex()`:

```
<function yylex 53b>≡ (162a)
/// main -> assemble -> yyparse -> <>
long
yylex(void)
{
    int c;
    <yylex() locals 55a>

    <yylex() peekc handling, starting part 54d>
10:
    c = GETC();
11:
    if(c == EOF) {
```

```

    return EOF;
}

if(isspace(c)) {
    <yylex() if c is newline 55c>
    // ignore spaces
    goto 10;
}
// else

<yylex() before switch, if isxxx 56b>
switch(c) {
//XxX: missing?
//    case '\n':
//        lineno++;
//        return ',';
<yylex() switch c cases 55d>
default:
    return c;
}
<yylex() peekc handling, ending part 55b>
return c;
}

```

Uses EOF 52a and GETC 50f.

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 54a>≡ (154a)
// option<char> (None = IGN)
int peekc = IGN;

```

Uses IGN 54b and peekc 54a.

```

<constant IGN 54b>≡ (150b)
#define IGN    (-2) // Ignore

```

```

<pinit() initializations 54c>≡ (47a) 89b▷
peekc = IGN;

```

Uses IGN 54b and peekc 54a.

Then, we need to make sure the function reading characters from the input file (`yylex()`<sup>53b</sup>) first looks in this global before trying to read any new character:

```

<yylex() peekc handling, starting part 54d>≡ (53b)
c = peekc;
if(c != IGN) {
    peekc = IGN; // consume the extra character saved in peekc
    goto 11; // skip the GETC(), we already have a character in c
}

```

Uses IGN 54b and peekc 54a.

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 55a>≡ (53b) 56a▷  
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 55b>≡ (53b)  
peekc = c1;
```

Uses `peekc 54a`.

## 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()` <sup>53b</sup>:

```
<yylex() if c is newline 55c>≡ (53b)  
if(c == '\n') {  
    lineno++;  
    return ';' ; // newline transformed in fake ';' ;  
}
```

Uses `lineno 47b`.

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` <sup>33a</sup> 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 55d>≡ (53b) 56c▷  
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");  
                errexit();  
            }  
        }  
    }  
    if(c1 == '*') {  
        // '/' '*' read; skip everything until next '*' '/'  
        for(;;) {  
            c = GETC();  
            while(c == '*') { // not an if! to handle /** not finished */  
                c = GETC();  
            }  
        }  
    }  
}
```

```

        if(c == '/')
            goto 10;
    }
    if(c == EOF) {
        yyerror("eof in comment");
        errexit();
    }
    if(c == '\n')
        lineno++;
}
}
break;

```

Uses EOF 52a, GETC 50f, errexit() 142a, lineno 47b, and yyerror() 142c.

## 6.5 Mnemonics, symbols, and labels

Until now, `yylex()`<sup>53b</sup> 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 known mnemonic, symbol, or label. The local `cp` below will point in the `symb`<sup>37a</sup> global buffer I introduced before. Remember that the global `symb` is used by `lookup()`<sup>37c</sup> to look for predefined identifiers, labels, or symbols, in the global symbol table `hash`<sup>36a</sup>.

```

<yylex() locals 56a>+≡ (53b) <55a 58a>
// 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 56b>≡ (53b) 57b▷
if(isalpha(c))
    goto talph;

```

```

<yylex() switch c cases 56c>+≡ (53b) <55d 57c>
case '_' :
case '@' :
// case 'a'..'z' 'A'..'Z': (isalpha())
// XxX: case '.' too
talph:
    cp = symb;

```

```

alooop:
    *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 130c>
    //XxX?
    //if(s->type == 0)
    //    s->type = LNAME;

```

```

<yylex() in identifier case, set ylval 57a>

```

```
return s->type;
```

Uses GETC 50f, lookup() 37c, peekc 54a, and symb 37a.

Remember that `yylex()` must return to `yyparse()`<sup>101b</sup> 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`<sup>33a</sup> if the identifier was an arithmetic opcode like `ADD`).

The global `yylval`<sup>101a</sup> must then contain the token value. For labels and symbols, the token value is a pointer to a `Sym`<sup>150b</sup>; for predefined identifiers, it is the value of an enumeration (e.g., `AADDX`).

```
<yylex() in identifier case, set yylval 57a>≡ (56c)
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 57b>+≡ (53b) <56b
if(isdigit(c))
    goto tnum;
```

```
<yylex() switch c cases 57c>+≡ (53b) <56c 60a>
// 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 58b>
```

```
<yylex() in number case, decimal dc label handling 57d>
```

```
<yylex() in number case, float labels handling 59c>
```

Uses symb 37a.

Note that the code above also (ab)uses `cp` and `symb`<sup>37a</sup> to accumulate characters (but this time there will be no call to `lookup()`<sup>37c</sup>).

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()`<sup>53b</sup> 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 57d>≡ (57c)
dc:
for(;;) {
    if(!isdigit(c))
        break;
    *cp++ = c;
    c = GETC();
}
```

```

}
⟨yylex() in number case, in decimal case, float handling 59b⟩
*cp = '\0';
yylval.lval = strtol(symb, nil, 10);

peekc = c;
return LCONST;

```

Uses GETC 50f, LCONST, peekc 54a, symb 37a, 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()`<sup>53b</sup> accumulates also characters in `symb`<sup>37a</sup> (via `cp`). However, this time `yylex()` does not rely on `strtol()` to convert the characters in an integer<sup>1</sup>. Instead, `yylex()` gradually updates `yylval.lval`:

```

⟨yylex() locals 58a⟩+≡ (53b) <56a 64e>
    int baselog2;

```

```

⟨yylex() in number case, 0xxx handling 58b⟩≡ (57c)

```

```

*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 59a⟩
    if(c >= 'a' && c <= 'f') {
        yylval.lval <<= baselog2;
        yylval.lval += c - 'a' + 10;
        c = GETC();
        continue;
    }
    break;
}
//XxX: goto ncu;

```

---

<sup>1</sup>It could because `strtol()` handles those cases.

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

Uses GETC 50f, LCONST, peekc 54a, and yylval.

```
<yylex() in number case, 0xxx handling, normalize letters 59a>≡ (58b)
if(c >= 'A' && c <= 'F')
    // c = lowercase(c)
    c += 'a' - 'A';
```

### 6.6.3 Floating-point numbers

yylex() <sup>53b</sup> uses a set of labels to recognize floating-point numbers:

```
<yylex() in number case, in decimal case, float handling 59b>≡ (57d)
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 59c>≡ (57c)
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 LFCNST;
```

```

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

```

Uses FPCHIP 117c, GETC 50f, LCONST, LFCONST, peekc 54a, symb 37a, yyerror() 142c, 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 60a>+≡ (53b) <57c 60b>
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 50f, peekc 54a, and symb 37a.

## 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()`<sup>62b</sup> used below is a wrapper around `GETC()`<sup>50f</sup>. 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`<sup>52a</sup> 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 60b>+≡ (53b) <60a 65>
case '\\':
    c = escchar('\\');
    <yylex() in character case, if c is EOF 62a>
    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 52a, LCONST, `escchar()` 62b, `yyerror()` 142c, and `yylval`.

As I said before, 5a converts characters in integers, hence the use of the LCONST<sup>33a</sup> 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 62a⟩≡ (60b)
    if(c == EOF)
        c = '\'';

```

Uses EOF 52a.

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 62b⟩≡ (155c)
    int
    escchar(int e)
    {
        int c;
        ⟨escchar() other locals 63c⟩

    loop:
        c = getc();
        ⟨escchar() sanity check if newline 63e⟩
        if(c != '\\') {
            if(c == e)
                return EOF;
            return c;
        }
        // else c is '\\'
        c = getc();
        ⟨escchar() if octal character 63d⟩
        switch(c)
        {
            ⟨escchar() switch cases 63a⟩
        }
        // else (e.g., '\'', '\\')
        return c;
    }

```

```
}
```

Uses EOF 52a and getc() 64c.

```

<escchar() switch cases 63a>≡
case 'n': return '\n';
case 't': return '\t';
case 'b': return '\b';
case 'r': return '\r';
case 'f': return '\f';

```

(62b) 63b▷

Asm5 allows a few more escape sequences compared to C:

```

<escchar() switch cases 63b>+≡
case 'a': return 0x07;
case 'v': return 0x0b;
case 'z': return 0x00;

```

(62b) <63a 64b>

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 63c>≡
int l;

```

(62b)

```

<escchar() if octal character 63d>≡
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;
}

```

(62b)

Uses getc() 64c and peekc 54a.

## 6.7.4 Escaped newlines

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

```

<escchar() sanity check if newline 63e>≡
if(c == '\n') {
    yyerror("newline in character or string");
    return EOF;
}

```

(62b)

Uses EOF 52a and yyerror() 142c.

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 63f>≡
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 64a>≡  
DATA foo(SB), "this is\  
a long string"
```

Here is the code to handle escaped newlines:

```
<escchar() switch cases 64b>+≡ (62b) ◀63b  
case '\n': goto loop; // multi line strings
```

Note that the escaped character and the following newline are skipped by `escchar()`<sup>62b</sup>.

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()`<sup>62b</sup>, is a small wrapper around `GETC()`<sup>50f</sup> that takes care of `lineno`<sup>47b</sup>:

```
<function getc 64c>≡ (155c)  
int  
getc(void)  
{  
    int c;  
  
    <getc() peekc handling 64d>  
    c = GETC();  
  
    if(c == '\n')  
        lineno++;  
    if(c == EOF) {  
        yyerror("End of file");  
        errexit();  
    }  
    return c;  
}
```

Uses EOF 52a, GETC 50f, errexit() 142a, lineno 47b, and yyerror() 142c.

```
<getc() peekc handling 64d>≡ (64c)  
c = peekc;  
if(c != IGN) {  
    peekc = IGN;  
    return c;  
}
```

Uses IGN 54b and peekc 54a.

## 6.8 Strings

Asm5 supports the use of strings, for instance, in `DATA` pseudo instructions. The code below relies again on the function `escchar()`<sup>62b</sup> to handle escaped sequences in strings.

```
<yylex() locals 64e>+≡ (53b) ◀58a  
int i;
```

`<yylex() switch c cases 65>+≡`

`(53b) <60b 66a>`

```
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 52a, LSCONST, escchar() 62b, nullgen 32a, yyerror() 142c, 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`<sup>48c</sup> 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()`<sup>67a</sup> in `yylex()`<sup>53b</sup> below to handle all preprocessing directives<sup>1</sup>:

```
<yylex() switch c cases 66a>+≡ (53b) <65
  case '#':
    domacro();
    goto 10;
```

Uses `domacro()` 67a.

`domacro()` will read all the characters on the line and modify some globals (e.g., `iostack`<sup>48c</sup> and `fi`<sup>50c</sup> when processing a `#include`). Note that after the call to `domacro()` above, `yylex()` jumps back to 10. 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 66b>≡ (160)
struct
{
  char    *macname;
  void    (*macf)(void);
} mactab[] =
{
  "ifdef",    nil,    /* macif(0) */
```

<sup>1</sup>The other entry point is the call to `macexpand()`<sup>131</sup> 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` 66b, `macdef()` 127b, `macend()` 69b, `macinc()` 71a, `maclin()` 75a, `macprag()` 135b, and `macund()` 133.

In this chapter, I will only describe the code of `macinc()`<sup>71a</sup> and `maclin()`<sup>75a</sup>. For the other callbacks, see Section 11.5.

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

```

⟨function domacro 67a⟩≡ (160)
  /// main -> assemble -> yyparse -> yylex -> <>
  void
  domacro(void)
  {
    int i;
    Sym *s;

    s = getsym();
    ⟨domacro() set s to endif symbol if no symbol 135a⟩

    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 69a⟩
  }

```

Uses `getsym()` 67b, `macif()` 134, and `mactab` 66b.

`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`<sup>37a</sup> and calls `lookup()`<sup>37c</sup>.

```

⟨function getsym 67b⟩≡ (160)
  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
    ungetc(c);
    break;
}
*cp = '\0';
⟨getsym() sanity check cp 68a⟩
return lookup();
}

```

Uses NSYMB 37b, S 36d, getc() 64c, getnsc() 68b, lookup() 37c, symb 37a, and ungetc() 68d.

```

⟨getsym() sanity check cp 68a⟩≡
if(cp > symb+NSYMB-4)
    yyerror("symbol too large: %s", symb);

```

(67b)

Uses NSYMB 37b, symb 37a, and yyerror() 142c.

getsym()<sup>67b</sup> calls first getnsc() (for “get first non-space character”) below:

```

⟨function getnsc 68b⟩≡
int
getnsc(void)
{
    int c;

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

```

(155c)

Uses getc() 64c.

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

```

⟨example of use of ifdef 68c⟩≡
#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 ungetc() below, called from getsym() above, allows to go back in the input buffer by storing the extra character in the global peekc<sup>54a</sup>.

```

⟨function ungetc 68d⟩≡
void
ungetc(int c)
{
    peekc = c;
    if(c == '\n')
        lineno--;
}

```

(155c)

Uses lineno 47b and peekc 54a.

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 69a>≡ (67a)
yyerror("unknown #: %s", s->name);
macend();
```

Uses `macend()` 69b and `yyerror()` 142c.

```
<function macend 69b>≡ (160)
void
macend(void)
{
    int c;

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

Uses `getnsc()` 68b.

## 7.2 #include

I can now describe the first callback `macinc()`<sup>71a</sup>, which handles `#include` directives.

### 7.2.1 Include search path

Before showing the code of `macinc()`<sup>71a</sup>, here are a few globals used by `macinc()`:

```
<global include 69c>≡ (154a)
// array<option<string>> (size = ninclude)
char* include[NINCLUDE];
```

Uses `NINCLUDE` 69d.

```
<constant NINCLUDE 69d>≡ (150b)
#define NINCLUDE 10
```

```
<global ninclude 69e>≡ (154a)
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()`<sup>149a</sup> sets first a default value for this first entry:

```
<main() remaining initializations 69f>≡ (41c) 140b▷
include[nininclude++] = ".";
```

This value is then refined in `assemble()`<sup>43d</sup> during the processing of the input filename (see Figure 4.1):

```
<assemble() adjust first entry in include with dirname infile 69g>≡ (45c)
include[0] = ofile;
```

Uses `include` 69c.

## 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 70a>≡ (41c) 136a▷  
char *p;
```

```
<main() command line processing 70b>+≡ (41c) <42b 126a▷  
case 'I':  
    p = ARGF();  
    setinclude(p);  
    break;
```

```
<function setinclude 70c>≡ (155c)  
void  
setinclude(char *p)  
{  
    int i;  
  
    <setinclude() sanity check p 70d>  
    <setinclude() check if include already added 70e>  
    <setinclude() check if too many entries in include 70f>  
    // else  
    include[ninclude++] = p;  
}
```

Uses include 69c and ninclude 69e.

```
<setinclude() sanity check p 70d>≡ (70c)  
if(p == nil)  
    return;
```

```
<setinclude() check if include already added 70e>≡ (70c)  
for(i=1; i < ninclude; i++)  
    if(strcmp(p, include[i]) == 0)  
        return;
```

Uses include 69c and ninclude 69e.

```
<setinclude() check if too many entries in include 70f>≡ (70c)  
if(ninclude >= nelem(include)) {  
    yyerror("ninclude too small %d", nelem(include));  
    exits("ninclude");  
}
```

Uses include 69c, ninclude 69e, and yyerror() 142c.

## 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 70g>+≡ (43d) <45b 126d▷  
char incfile[20];
```

```
<assemble() setinclude("/thestring/include") or use INCLUDE 70h>≡ (43d)  
p = getenv("INCLUDE");  
if(p) {  
    setinclude(p);  
} else {  
    sprintf(incfile, "%s/include", thestring);  
    setinclude(strdup(incfile));  
}
```

Uses setinclude() 70c and thestring 41b.

## 7.2.4 macinc()

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

```
<function macinc 71a>≡ (160)
void
macinc(void)
{
    char *hp;
    int n;
    fdt f = -1;
    <macinc() other locals 72c>

    // lexing

    <macinc() lexing the included filename 72e>

    // action

    <macinc() find and store the included filename full path in symb 73a>

    n = strlen(symb) + 1;

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

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

    return;

<macinc() bad 72f>
}
```

Uses `newfile()` 49d, `newio()` 48g, `pushio()` 71b, and `symb` 37a.

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()` 48g and `newfile()` 49d above. I introduced both functions in Chapter 5. Those functions modify the globals `iostack` 48c and `fi` 50c. Thus, when `yylex()` 53b will goto back to 10 (after having processed the `#include` line via `domacro()` 67a and `macinc()` 71a), further calls to `GETC()` 50f will operate on an empty buffer `fi`. This will trigger a call to `filbuf()` 50g, 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 71b>≡ (155c)
void
pushio(void)
{
    Io *i;

    i = iostack;
    <pushio() sanity check i 72a>
    // save current position in includer
    i->p = fi.p;
    i->c = fi.c;
}
```

Uses `fi` 50c and `iostack` 48c.

```

⟨pushio() sanity check i 72a)≡ (71b)
    if(i == I) {
        yyerror("botch in pushio");
        errexit();
    }

```

Uses I 48e, errexit() 142a, and yyerror() 142c.

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

```

⟨example of use of include 72b)≡
    #include <system_file.h>
    #include "other_file.h"

```

```

⟨macinc() other locals 72c)≡ (71a) 72g▷
    char str[STRINGSZ];
    int c, cend;

```

Uses STRINGSZ 72d.

```

⟨constant STRINGSZ 72d)≡ (150b)
    #define STRINGSZ    200

```

```

⟨macinc() lexing the included filename 72e)≡ (71a)
    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 73d)

Uses getc() 64c and getnsc() 68b.

```

⟨macinc() bad 72f)≡ (71a)
    bad:
        unget(c);
        yyerror("syntax in #include");
        macend();

```

Uses macend() 69b, unget() 68d, and yyerror() 142c.

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 72g)+≡ (71a) ◁72c
    int i;

```

```

<macinc() find and store the included filename full path in symb 73a>≡ (71a)
for(i=0; i<ninclude; i++) {
    <macinc() skipped first entry for system headers 73b>
    strcpy(symb, include[i]);
    strcat(symb, "/");
    <macinc() normalize path 73c>
    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 69c, ninclude 69e, and symb 37a.

The first entry of include<sup>69c</sup>, 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 73b>≡ (73a)
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 73c>≡ (73a)
if(strcmp(symb, "./") == 0)
    symb[0] = '\\0';

```

Uses symb 37a.

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

```

<macinc() finish parsing the line 73d>≡ (72e)
c = getcom();
if(c != '\\n')
    goto bad;

```

Uses getcom() 73e.

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 73e>≡ (160)
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()` 64c, `getnsc()` 68b, and `yyerror()` 142c.

## 7.3 #line

The other (and last) callback I will present in this chapter is `maclin()`<sup>75a</sup>, which handles `#line` directives.

### 7.3.1 Motivations

`#line` directives are extremely 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 75a>≡ (160)
void
maclin(void)
{
    char *cp;
    long n;
    int size;
    <maclin() other locals 75b>

    // lexing

    <maclin() lexing the line n and filename in symb 75c>

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

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

    memcpy(cp, symb, size);

    <maclin() call linehist 111a>
    return;

    <maclin() bad 76c>
}

```

Uses `symb` 37a.

`maclin()`<sup>75a</sup>, like `newfile()`<sup>49d</sup> (called from `macinc()`<sup>71a</sup> and `pinit()`<sup>47a</sup>), calls the function `linehist()`<sup>110g</sup> 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`<sup>37a</sup> through a pointer `cp`. However, the function `getnsn()`<sup>76e</sup> below recognizes first the integer before the filename.

```

<maclin() other locals 75b>≡ (75a)
    int c;

```

```

<maclin() lexing the line n and filename in symb 75c>≡ (75a)
    // the line number

    n = getnsn();

    // the (optional) filename

```

```

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

for(;;) {
    <maclin() skipping whitespaces 76b>
    if(c == ' ')
        break;
    <maclin() if no filename 76a>
    // else
    goto bad;
}
cp = symb;
for(;;) {
    c = getc();
    if(c == ' ')
        break;
    *cp++ = c;
}
*cp = '\0';

```

<maclin() finish parsing the line 76d>

Uses getc() 64c, getnsn() 76e, and symb 37a.

```

<maclin() if no filename 76a>≡ (75c)
if(c == '\n') {
    strcpy(symb, "<noname>");
    goto nn;
}

```

Uses symb 37a.

```

<maclin() skipping whitespaces 76b>≡ (75c)
if(c == ' ' || c == '\t') {
    c = getc();
    continue;
}

```

Uses getc() 64c.

```

<maclin() bad 76c>≡ (75a)
bad:
    ungetc(c);
    yyerror("syntax in #line");
    macend();

```

Uses macend() 69b, ungetc() 68d, and yyerror() 142c.

```

<maclin() finish parsing the line 76d>≡ (75c)
c = getcom();
if(c != '\n')
    goto bad;

```

Uses getcom() 73e.

```

<function getnsn 76e>≡ (160)
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();
}
ungetc(c);
return n;
}
```

Uses `getc()` 64c, `getnsc()` 68b, and `ungetc()` 68d.

# 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()`<sup>101b</sup> parsing function which is called by `assemble()`<sup>43d</sup> and which internally calls `yylex()`<sup>53b</sup>. We assume you know how to read a Yacc grammar, otherwise read [BLM92]<sup>1</sup>.

### 8.1 Overview

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

```
<5a/a.y 78a>≡
%{
#include "a.h"
%}
<union declaration(arm) 35a>
<priority and associativity declarations 116c>
<token declarations(arm) 33a>
<type declarations(arm) 89e>
%%
<grammar(arm) 78b>
```

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) 78b>≡ (78a)
<prog rule(arm) 79a>
<line rule(arm) 79b>
<inst rule(arm) 80a>
<operand rules(arm) 89d>
<cond rule(arm) 99f>

<advanced topics rules ??>

<helper rules(arm) 93b>
```

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).

---

<sup>1</sup>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) 79a>≡ (78b)
prog:
/* empty */
| prog line

```

A line can be an `inst`<sup>80a</sup> instruction followed by a semicolon (remember that the lexer transforms newlines in semicolons as explained in Section 6.3):

```

<line rule(arm) 79b>≡ (78b) 79c▷
line:
inst ';'

```

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

```

<line rule(arm) 79c>+≡ (78b) <79b 88h▷
| ';'
| 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()`<sup>101b</sup> 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`<sup>29</sup> type (overviewed in Section 3.1):

```

<Opcode cases, logic opcodes 79d>≡ (29) 81f▷
AAND,
AORR,
AEOR, // a.k.a. XOR

```

Their “reading syntax” are described via entries in `itab`<sup>35c</sup> presented in Section 3.5 (those entries populate the symbol table `hash`<sup>36a</sup> as explained in Section 3.6):

```

<itab elements 79e>≡ (35c) 81a▷
"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) 80a>≡ (78b) 82f▷
inst:
/*
 * AND/ORR/ADD/SUB/...
 */
LARITH cond imsr ',' 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  $n$ th terminal or non-terminal in the rule. For terminals this value is derived from the global `yy1val`<sup>101a</sup> set by the lexer. Here are the different values for the different elements in the first rule of `inst`<sup>80a</sup> above:

1. `LARITH`<sup>33a</sup> 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`<sup>99f</sup> 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`<sup>89d</sup> is a shorthand for: `immediate` (constant) or `shifted-register` or `register`. Those are operands we will describe in Section 8.4.  $\$3$  is of type `Gen`<sup>31b</sup>, the generalized form of operand I described before.
4. the first comma, a single-character token without any value
5. `reg`<sup>90f</sup> is a register.  $\$5$  is an integer representing the register, e.g., 10 for R10.
6. the second comma
7. `reg`<sup>90d</sup> 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()`<sup>104a</sup> 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 <sup>2</sup>.

## Add/sub

Basic arithmetic is also fundamental to have in a computer:

```

<Opcode cases, add/sub opcodes 80b>≡ (29) 82b▷
AADD,
ASUB,

```

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

`<itab elements 81a>+≡` (35c) <79e 81c>

```
"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<sup>33a</sup> 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 81b>≡` (29) 121a>

```
AMUL,  
ADIV, // VIRTUAL, transformed in call to _div  
AMOD, // VIRTUAL, transformed in call to _mod
```

`<itab elements 81c>+≡` (35c) <81a 81e>

```
"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 51 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 81d>≡` (29)

```
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 81e>+≡` (35c) <81c 82a>

```
"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 51 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 81f>+≡` (29) <79d

```
ABIC, // ??
```

`<itab elements 82a>+≡` (35c) <81e 82c>

```
"BIC", LARITH, ABIC,
```

Uses LARITH.

`<Opcode cases, add/sub opcodes 82b>+≡` (29) <80b 82d>

```
ARSB, // ??
```

```
AADC, // Add and carry?
```

```
ASBC, // Sub and carry?
```

```
ARSC, // ??
```

`<itab elements 82c>+≡` (35c) <82a 82e>

```
"RSB", LARITH, ARSB,
```

```
"ADC", LARITH, AADC,
```

```
"SBC", LARITH, ASBC,
```

```
"RSC", LARITH, ARSC,
```

Uses LARITH.

`<Opcode cases, add/sub opcodes 82d>+≡` (29) <82b>

```
AMVN, // MOV negative, but nothing to do with MOVW
```

`<itab elements 82e>+≡` (35c) <82c 83b>

```
"MVN", LMVN, AMVN, /* op2 ignored */
```

Uses LMVN.

`<inst rule(arm) 82f>+≡` (78b) <80a 83c>

```
/*
```

```
* 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 51 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 83a>≡ (29) 122a▷
AMOVW, // VIRTUAL, transformed in load and store instructions
AMOVB,
AMOVBU,
AMOVH,
AMOVHU,
```

```
<itab elements 83b>+≡ (35c) <82e 84b▷
"MOVW", LMOV, AMOVW,
"MOVB", LMOV, AMOVB,
"MOVBU", LMOV, AMOVBU,
"MOVH", LMOV, AMOVH,
"MOVHU", LMOV, AMOVHU,
```

Uses LMOV.

MOVs take a new kind of operand, `gen90b`, 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) 83c>+≡ (78b) <82f 84c▷
/*
 * 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 51 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, 51 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 84a>≡ (29)
  ASWPW,
  ASWPBU,
```

```
<itab elements 84b>+≡ (35c) <83b 84e>
  "SWPW", LSWAP, ASWPW,
  "SWPBU", LSWAP, ASWPBU,
```

Uses LSWAP.

```
<inst rule(arm) 84c>+≡ (78b) <83c 85a>
/*
 * 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`<sup>94b</sup> 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 84d>≡ (29) 85e>
  AB, // =~ JMP
```

```
<itab elements 84e>+≡ (35c) <84b 85c>
  "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`<sup>96e</sup> (register or label) non-terminal below, the last two cases by `nireg`<sup>96c</sup> (name or indirect register).

```
<inst rule(arm) 85a>+≡ (78b) <84c 85d>
/*
 * 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 85b>≡ (29) 87a>
ACMP,
```

```
<itab elements 85c>+≡ (35c) <84e 86a>
"CMP", LCMP, ACMP,
Uses LCMP.
```

```
<inst rule(arm) 85d>+≡ (78b) <85a 86b>
/*
 * 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 85e>+≡ (29) <84d 87c>
/*
 * 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, // overflow 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 86a>+≡` (35c) <85c 87b>

```

"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) 86b>+≡` (78b) <85d 88b>

```

/*
 * 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 `Bxx` 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 51 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 87a>+≡ (29) <85b
  ATST,
  ATEQ,
  ACMN, // CMP negative
```

```
<itab elements 87b>+≡ (35c) <86a 87d>
  "TST",  LCMP, ATST,
  "TEQ",  LCMP, ATEQ,
  "CMN",  LCMP, ACMN,
Uses LCMP.
```

## Function call and return

The `B`, `CMP`, and `Bxx` 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 87c>+≡ (29) <85e 87e>
  ABL, // =~ CALL, Branch and Link
```

```
<itab elements 87d>+≡ (35c) <87b 88a>
  "BL",  LBRANCH, ABL,
Uses LBRANCH.
```

The `BL` instruction uses the same syntax than `B` and so use the same grammar rule involving the `LBRANCH`<sup>33a</sup> 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 87e>+≡ (29) <87c>
  ARET, // VIRTUAL, transformed in B (R14) or MOV xxx(R13), R15
```

```

<itab elements 88a>+≡ (35c) <87d 88d>
    "RET", LRET, ARET,
Uses LRET.

```

```

<inst rule(arm) 88b>+≡ (78b) <86b 88e>
/*
 * 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 88c>≡ (29) 88f>
    ASWI, // syscall

```

```

<itab elements 88d>+≡ (35c) <88a 88g>
    "SWI", LSWI, ASWI,
Uses LSWI.

```

```

<inst rule(arm) 88e>+≡ (78b) <88b 97e>
/*
 * 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 MOVW instruction we will describe in Section 11.2.4.

```

<Opcode cases, interrupt opcodes 88f>+≡ (29) <88c
    ARFE, // VIRTUAL, return from exception/interrupt, MOVW.IA.S.W (R13), [R15]

```

```

<itab elements 88g>+≡ (35c) <88d 90g>
    "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) 88h>+≡ (78b) <79c 89c>
| 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()`<sup>104a</sup> after each code instruction:

```
<global pc 89a>≡ (154a)
long pc;
```

```
<pinit() initializations 89b>+≡ (47a) <54c 107b>
pc = 0;
```

Uses `pc 89a`.

Once a label has been defined, the code of its token in the symbol table is changed to `LLAB`<sup>33a 3</sup> 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) 89c>+≡ (78b) <88h 117a>
| 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) 89d>≡ (78b) ??>
imsr:
  imm
| shift
| reg
```

The next three subsections will each cover one of the `imsr`<sup>89d</sup> 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) 89e>≡ (78a) 90a>
%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) 89f>+≡ (35a) <39c>
Gen    genval;
```

---

<sup>3</sup>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`<sup>89f</sup> is `Gen`<sup>31b</sup>, 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) 90a>+≡ (78a) <89e 90c>
%type <genval> gen
```

```
<gen rule 90b>≡ ( ? 0—1)
gen:
  ximm
  | shift
  | reg
  <more gen rule 93d>
```

For now `gen`<sup>90b</sup> is almost identical to `imsr`, except for the `ximm`<sup>92a</sup> (extended immediate) vs `imm`<sup>91c</sup> which I will explain soon, but we will see later many extensions to the `gen` rule.

```
<type declarations(arm) 90c>+≡ (78a) <90a 90e>
%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`<sup>31b</sup> the integer representing the register number. It uses the `D_REGX` *operand kind* I described before:

```
<operand rules(arm) 90d>+≡ (78b) <?? 91c>
reg:
  regi
  {
    $$ = nullgen;
    $$ .type = D_REG;
    $$ .reg = $1;
  }
```

`regi` (register integer) returns directly the register number:

```
<type declarations(arm) 90e>+≡ (78a) <90c 93a>
%type <lval> regi
```

```
<regi rule(arm) 90f>≡ ( ? 0—1) 91b>
regi:
  LREG
```

```
<itab elements 90g>+≡ (35c) <88g 91a>
"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 91a>+≡ (35c) <90g 93c>
"R", LR, 0,
```

Uses LR.

```
<regi rule(arm) 91b>+≡ (? 0—1) <90f
| 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) 91c>+≡ (78b) <90d 92c>
imm: '$' con
{
  $$ = nullgen;
  $$ .type = D_CONST;
  $$ .offset = $2;
}
```

```
<con rule 91d>≡ (116d) 116b>
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 51 will use multiple instructions to encode the constant.

MOVs instructions, which use the `gen90b` non-terminal as operand (which derives `ximm`) can also take a constant as an operand:

```
<ximm rule 92a>≡ ( ? 0—1 ) 92b▷
ximm:
  '$' con
  {
    $$ = nullgen;
    $$ .type = D_CONST;
    $$ .offset = $2;
  }
```

In fact, even string constants are possible operands of MOVs:

```
<ximm rule 92b>+≡ ( ? 0—1 ) <92a 95c▷
| '$' 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) 92c>+≡ (78b) <91c 94a▷
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 92d>+≡ (31c) <39g 96f▷
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<sup>4</sup>: 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) 93a>+≡ (78a) <90e 93e>
%type <lval> rcon
```

```
<helper rules(arm) 93b>≡ (78b) 94f>
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;
  }
}
```

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`<sup>35c</sup> had to be used to represent the single character:

```
<itab elements 93c>+≡ (35c) <91a 94g>
"@", LAT, 0,
Uses LAT.
```

## 8.4.4 Memory (de)references, pointers

We have seen the three components of the `imsr`<sup>89d</sup> 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`<sup>90b</sup> 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`<sup>92a</sup> vs `imm`<sup>91c</sup> difference):

```
<more gen rule 93d>≡ (90b) 94c>
| 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) 93e>+≡ (78a) <93a 94d>
%type <genval> ioreg ireg
```

---

<sup>4</sup>We could use `Gen.reg` and `Gen.offset` instead.

```

<operand rules(arm) 94a>+≡ (78b) <92c 94b>
ioreg:
  ireg
| con '(' regi ')',
{
  $$ = nullgen;
  $$ .type = D_OREG;
  $$ .reg = $3;
  $$ .offset = $1;
}

```

```

<operand rules(arm) 94b>+≡ (78b) <94a 96c>
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 94c>+≡ (90b) <93d 95b>
| name

```

```

<type declarations(arm) 94d>+≡ (78a) <93e 96b>
%type <genval> name
%type <lval> offset
%type <lval> pointer

```

```

<name rule 94e>≡ (? 0—1) 95d>
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) 94f>+≡ (78b) <93b 95a>
pointer:
  LSB
| LSP
| LFP

```

```

<itab elements 94g>+≡ (35c) <93c 97b>
"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) 95a>+≡ (78b) <94f  
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 95b>+≡ (90b) <94c 119d>  
| 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 `name95d` 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 `ximm92a` deviates from `imm91c`:

```
<ximm rule 95c>+≡ (? 0—1) <92b 119e>  
| '$' 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 95d>+≡ (? 0—1) <94e  
| LNAME '<' '>' offset '(' LSB ')'  
{  
  $$ = nullgen;  
  $$ .type = D_OREG;  
  $$ .sym = $1;  
  $$ .symkind = N_INTERN;  
  $$ .offset = $4;  
}
```

```

<Sym_kind cases 96a>≡ (39f) 115b▷
  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) 96b>+≡ (78a) <94d 96d▷
  %type <genval> nireg

```

```

<operand rules(arm) 96c>+≡ (78b) <94b ??▷
  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) 96d>+≡ (78a) <96b 99e▷
  %type <genval> rel

```

```

<rel rule 96e>≡ (? 0—1) 96g▷
  rel:
    LLAB offset
    {
      $$ = nullgen;
      $$ .type = D_BRANCH;
      $$ .sym = $1;
      $$ .offset = $1->value + $2;
    }

```

This operand uses a new kind of operand:

```

<Operand_kind cases 96f>+≡ (31c) <92d 119b▷
  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`<sup>33a</sup> not an `LNAME`<sup>33a</sup>:

```

<rel rule 96g>+≡ (? 0—1) <96e 97a▷
  | 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 `pc89a`):

```
<rel rule 97a>+≡ ( ? 0—1 ) <96g
| con '( LPC )'
{
  $$ = nullgen;
  $$ .type = D_BRANCH;
  $$ .offset = $1 + pc;
}
```

```
<itab elements 97b>+≡ (35c) <94g 97d>
"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 known 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 97c>≡ (29) 98a>
ATEXT,
AGLOBL,
```

```
<itab elements 97d>+≡ (35c) <97b 98b>
"TEXT", LDEF, ATEXT,
"GLOBL", LDEF, AGLOBL,
```

Uses LDEF.

```
<inst rule(arm) 97e>+≡ (78b) <88e 98c>
/*
 * 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 `name95d` 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()`<sup>104a</sup> 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 98a>+≡ (29) <97c 98d>
  ADATA,
```

```
<itab elements 98b>+≡ (35c) <97d 98e>
  "DATA", LDATA, ADATA,
Uses LDATA.
```

```
<inst rule(arm) 98c>+≡ (78b) <97e 99a>
/*
 * DATA
 */
| LDATA name '/' con ',,' ximm { outcode($1, Always, &$2, $4, &$6); }
```

Note that the name<sup>95d</sup> can contain an offset as in hello+8(SB). The con<sup>91d</sup> specifies the size in bytes this DATA pseudo instruction is defining. Note also the use of ximm<sup>92a</sup> 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 98d>+≡ (29) <98a 99b>
  AWORD,
```

```
<itab elements 98e>+≡ (35c) <98b 99c>
  "WORD", LWORD, AWORD,
Uses LWORD.
```

```

<inst rule(arm) 99a>+≡ (78b) <98c 99d>
/*
 * 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 99b>+≡ (29) <98d 107d>
AEND,

```

```

<itab elements 99c>+≡ (35c) <98e 99g>
"END", LEND, AEND,
Uses LEND.

```

```

<inst rule(arm) 99d>+≡ (78b) <99a 119a>
/*
 * 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()`<sup>44d</sup> (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) 99e>+≡ (78a) <96d 116a>
%type <lval> cond

```

```

<cond rule(arm) 99f>≡ (78b) 100c>
cond:
/* empty */ { $$ = Always; }
| cond LCOND { $$ = ($1 & ~C_SCOND) | $2; }

```

```

<itab elements 99g>+≡ (35c) <99c 100d>
".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 [100a](#) and LCOND.

```

⟨constant Always(arm) 100a⟩≡ (161a)
#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 predictor in ARM CPUs by having code using less branches. See [https://en.wikipedia.org/wiki/ARM\\_architecture#Conditional\\_execution](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 `cond99f` non-terminal in addition to the conditional execution also stores special ARM instruction bits (as explained in Section [8.7](#)):

```

⟨constant C_SCOND(arm) 100b⟩≡ (148c)
/* scond byte */
#define C_SCOND ((1<<4)-1)

```

## 8.7 Special bits

### 8.7.1 Arithmetic instructions and .S

```

⟨cond rule(arm) 100c⟩+≡ (78b) <99f
| cond LS { $$ = $1 | $2; }

```

```

⟨itab elements 100d⟩+≡ (35c) <99g 100f>
".S", LS, C_SBIT,

```

Uses LS.

```

⟨constant C_SBIT(arm) 100e⟩≡ (148c)
#define C_SBIT (1<<4)

```

### 8.7.2 Moves and .P/.W

```

⟨itab elements 100f⟩+≡ (35c) <100d 117f>
".P", LS, C_PBIT,
".W", LS, C_WBIT,

```

Uses LS.

```

⟨constant C_PBIT(arm) 100g⟩≡ (148c)
#define C_PBIT (1<<5)

```

```

⟨constant C_WBIT(arm) 100h⟩≡ (148c)
#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`<sup>101a</sup>, as explained in Section 3.5. Both the token codes and `yylval` are used by `yylex()`<sup>53b</sup>.
- a `y.tab.c` C file containing the function `yyparse()`<sup>101b</sup>. 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<sup>52a</sup>), 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 101a>≡
typedef union {
    long    lval;
    double  dval;
    char    sval[NSNAME];
    Sym     *sym;
    Gen     genval;
} YYSTYPE;
extern YYSTYPE yyval;
#define LARITH 57346
#define LCMP 57347
...
```

```
<y.tab.c generated file excerpt 101b>≡
typedef union {
    long    lval;
    double  dval;
    char    sval[NSNAME];
    Sym     *sym;
    Gen     genval;
} YYSTYPE;
extern int yyerrflag;
...
YYSTYPE yyval;
...
#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 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`<sup>100a</sup>
- A register number (1 byte), e.g., 4 for `R4`, when an instruction has 3 operands or the value of `R_NONE`<sup>31a</sup>
- 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<sup>1</sup>.
- 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`.
- 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 9.4.

---

<sup>1</sup>Also known as the *little-endian* format



- 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 9.4 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.

## 9.2 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) 104a>≡ (162b)
  // main -> assemble -> yyparse -> <>
  void
  outcode(int opcode, int scond, Gen *g1, int reg, Gen *g2)
  {
    <outcode() locals 107e>

    <outcode() adjust opcode and scond when opcode is AB 104b>

    if(pass == 1)
      goto out;

    <outcode() st and sf computation, and possible calls to zname 108a>

    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` 47b, `obuf` 44c, `outopd()` 105b, `pass` 43c, and `pc` 89a.

It also increments the virtual program counter `pc`<sup>89a</sup> (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()`<sup>104a</sup> also *normalizes* the code:

```

<outcode() adjust opcode and scond when opcode is AB 104b>≡ (104a)
  /* 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 100a and bcode-56 105a.

```

⟨global bcode(arm) 105a⟩≡ (162b)
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.

### 9.3 Operand output: `outopd()`

Assuming a symbol index parameter `symidx` computed by the caller (and explained in Section 9.4), `outopd()` outputs an operand according to the format described in the right of Figure 9.1:

```

⟨function outopd(arm) 105b⟩≡ (162b)
/// 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) 106a⟩
    default:
        print("unknown type %d\n", a->type);
        exits("arg");
    }
}

```

Uses `obuf` 44c.

The operand value has a variable size depending on the kind of the operand:

```
<outopd() cases(arm) 106a>≡ (105b) 120b▷
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 44c.

## 9.4 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 9.1), 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 106b>≡ (154a)
// array<Htab>
struct Htab h[NSYM];
```

Uses Htab 106d.

```
<constant NSYM 106c>≡ (148a)
#define NSYM 50
```

```
<struct Htab 106d>≡ (150b)
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`<sup>36a</sup> 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 107a>≡ (154a)
int symcounter;
```

```
<pinit() initializations 107b>+≡ (47a) <89b 126g>
symcounter = 1;
for(i=0; i<NSYM; i++) {
    h[i].symkind = 0; // N_NONE
    h[i].sym = S;
}
```

Uses `S` 36d, `h` 106b, and `symcounter` 107a.

As we will output instructions in `outcode()`<sup>104a</sup>, 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) 107c>≡ (162b)
/// 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` 44c.

A new pseudo opcode is used:

```
<Opcode cases, pseudo opcodes 107d>+≡ (29) <99b 114c>
ANAME,
```

This constant must be part of the `Opcode`<sup>29</sup> 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()`<sup>107c</sup> when a new entry is needed. The locals below are set by this code. `sf` and `st` are passed to `outopd()`<sup>105b</sup> (look at the code of `outcode()`):

```
<outcode() locals 107e>≡ (104a)
// symbol from, index in h[]
int sf;
// symbol to, index in h[]
int st;
int oldsymcounter;
```

An additional field in `Sym`<sup>150b</sup> is needed to store the index of symbols already present in `h`:

```
<Sym identifier fields 107f>≡ (35d)
// 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 108a>≡ (104a)
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 108d>
```

Uses `symcounter` 107a and `symidx_of_symopt()` 108b.

```
<function symidx_of_symopt 108b>≡ (162b)
int
symidx_of_symopt(Sym *sym, int symkind)
{
    int idx = 0;

    if(sym != S) {
        idx = sym->symidx;
        <symidx_of_symopt() sanity check idx 108c>

        // 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` 36d, `h` 106b, `symcounter` 107a, and `zname()` 107c.

```
<symidx_of_symopt() sanity check idx 108c>≡ (108b)
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 108d>≡ (108a)
if (sf == st && sf != 0 && symcounter != oldsymcounter)
    goto jackpot;
```

Uses `symcounter` 107a.

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)`.

# 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 9.1 seems insufficient; a filename should also be recorded.

The global `lineno`<sup>47b</sup> I introduced before, and which is used by `outcode()`<sup>104a</sup> 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 110a>≡ (150b)
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 110c>
};

```

The global `hist` below maintains a list of those `Hist`<sup>150b</sup>. The first element of this list will be the input assembly file of 5a passed on the command line.

```

<global hist 110b>≡ (154a)
// list<ref_own<Hist>> (next = Hist.link)
Hist* hist;

```

```

<Hist extra fields 110c>≡ (110a)
// list<ref<Hist>> (from = hist)
Hist* link;

```

```

<constant H 110d>≡ (150b)
#define H ((Hist*)nil)

```

We will need also a fast access to the tail of the list:

```

<global ehist 110e>≡ (154a)
// ref<Hist> (end from = hist)
Hist* ehist;

```

## 10.2 Recording history: `linehist()`

The only function modifying `hist`<sup>110b</sup> is `linehist()`. It is called notably by `newfile()`<sup>49d</sup> which itself is called at the beginning in `pinit()`<sup>47a</sup> with the name of the input assembly file and later for each included file:

```

<newfile() call linehist 110f>≡ (49d)
linehist(s, 0);

```

Uses `linehist()` 110g.

```

<function linehist 110g>≡ (155b)
/// (pinit | macinc -> newfile) | (GETC -> filbuf) | maclin -> <>
void
linehist(char *f, int local_line)
{
    Hist *h;

    <linehist() debug 140d>

    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 110d, alloc() 145b, ehist 110e, hist 110b, and lineno 47b.

It is also called for #line directives such as #line 10 "foo":

```

<maclin() call linehist 111a>≡ (75a)
    linehist(cp, n);

```

Uses linehist() 110g.

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`<sup>150b</sup> encodes the end of a file which is enough to reconstruct the tree:

```

<filbuf() when close file, call linehist 111b>≡ (51a)
    linehist(nil, 0);

```

Uses linehist() 110g.

Figure 10.1 illustrates the evolution of the global `lineno`<sup>47b</sup> on the content of `/tests/cpp/foo.s` (which includes other files) as well as the value of `hist` and `ehist`<sup>110e</sup> 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()`<sup>110g</sup>:

```

$ 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
(global)    included files

                                ehist

```

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 113a>≡ (150b)
#define HISTSZ      20
```

For instance with the data on Figure 10.1, `prfile(14)` should print:

```
foo.s:5 bar.s:2
```

```
<function prfile 113b>≡ (155b)
/// yyerror -> <>
void
prfile(long l)
{
    Hist a[HISTSZ];
    int n = 0;
    Hist *h;
    int i;
    long d;

    <prfile() compute a and n 113c>
    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 113a.

The +1 above is to have line numbers starting at 1.

```
<prfile() compute a and n 113c>≡ (113b)
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 114a>
        }
        // a pop
        else {
            n--;
            <prfile() compute a and n, when pop, adjust parents 113d>
        }
    }
}
```

Uses H 110d, HISTSZ 113a, and hist 110b.

```
<prfile() compute a and n, when pop, adjust parents 113d>≡ (113c)
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 113a.

```

⟨prfile() compute a and n, when line directive 114a⟩≡ (113c)
// 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 113a.

```

## 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()`<sup>43d</sup> before the many calls to `outcode()`<sup>104a</sup> and essentially serializes `hist`<sup>110b</sup>. 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`<sup>150b</sup> and its filename:

```

⟨function outhist(arm) 114b⟩≡ (162b)
/// 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 115d⟩
        ⟨outhist() output each path component as an ANAME 115a⟩

        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 100a, H 110d, hist 110b, nullgen 32a, obuf 44c, and outopd() 105b.

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 114c⟩+≡ (29) <107d 125b>
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 115a>≡ (114b)
// =~ 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 115e>
}

```

Uses `obuf 44c`.

Note the use of the same symbol index above, 1, which normally is incremented after each symbol is added in Section 9.4. 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 115b>+≡ (39f) <96a 125i>
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 `pathname43a` below is the current working directory set in `cinit()42c`.

```

<outhist() locals(arm) 115c>+≡ (114b) <??>
char *op;

<outhist() adjust p and op if p is relative filename 115d>≡ (114b)
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 43a`.

```

<outhist() adjust p and op if p was a relative filename 115e>≡ (115a)
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) 116a>+≡ (78a) <99e 119c>*  
%type <lval> con expr

*<con rule 116b>+≡ (116d) <91d 117b>*  
| '(' expr ')' { \$\$ = \$2; }

*<priority and associativity declarations 116c>≡ (78a)*  
%left '|'   
%left '^'   
%left '&'   
%left '<' '>'   
%left '+' '-'   
%left '\*' '/' '%'

*<constant expression rules 116d>≡ (? 0—1)*  
*<con rule 91d>*  
*<expr rule 116e>*

*<expr rule 116e>≡ (116d)*  
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; }

## 11.1.2 Symbolic constants

```
<line rule(arm) 117a>+≡ (78b) <89c
| 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 117b>+≡ (116d) <116b
| LVAR      { $$ = $1->value; }
```

## 11.2 Other instructions and registers

### 11.2.1 Floating-point numbers

```
<constant FPCHIP(arm) 117c>≡ (161a)
#define FPCHIP      true
```

```
<struct ieee 117d>≡ (148a)
/*
 * 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 117e>≡ (29)
AMOVWD,
AMOVWF,
AMOVDW,
AMOVFW,
AMOVFD,
AMOVDF,
AMOVF,
AMOVD,
```

```
<itab elements 117f>+≡ (35c) <100f 118b>
"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 118a>≡`

`(29)`

```
ACMPF,  
ACMPD,  
AADDf,  
AADDd,  
ASUBf,  
ASUBd,  
AMULf,  
AMULd,  
ADIVf,  
ADIVd,  
ASQRTf,  
ASQRTd,
```

`<itab elements 118b>+≡`

`(35c) <117f 118c>`

```
"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 118c>+≡`

`(35c) <118b 118d>`

```
"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 118d>+≡`

`(35c) <118c 121b>`

```
"FPSR", LFCR, 0,  
"FPCR", LFCR, 1,
```

Uses LFCR.

`<enum Fregister(arm) 118e>≡`

`(148c)`

```
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) 119a>+≡ (78b) <99d 121c>
/*
 * 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 119b>+≡ (31c) <96f 121d>
D_FREG,
D_FPCR,

<type declarations(arm) 119c>+≡ (78a) <116a 121i>
%type <genval> freg fcon frcon

<more gen rule 119d>+≡ (90b) <95b 120a>
| freg

<ximm rule 119e>+≡ (? 0—1) <95c>
| fcon

<float rules 119f>≡ (? 0—1)
freg:
    LFREG
    {
        $$ = nullgen;
        $$ .type = D_FREG;
        $$ .reg = $1;
    }
| LF '(' con ')'
    {
        $$ = nullgen;
        $$ .type = D_FREG;
        $$ .reg = $3;
    }

fcon:
    '$' LFCNST
    {
        $$ = nullgen;
        $$ .type = D_FCONST;
        $$ .dval = $2;
    }
| '$' '-' LFCNST
    {
        $$ = nullgen;
        $$ .type = D_FCONST;
        $$ .dval = -$3;
    }

frcon:
    freg
| fcon

```

```

<more gen rule 120a>+≡ (90b) <119d 123f>
| LFCR
{
  $$ = nullgen;
  $$ .type = D_FPCR;
  $$ .reg = $1;
}

```

```

<outopd() cases(arm) 120b>+≡ (105b) <106a 120c>
case D_FREG:
case D_FPCR:
  break;

```

```

<outopd() cases(arm) 120c>+≡ (105b) <120b 121e>
case D_FCONST:
  ieeeedtod(&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()` 120d and `obuf` 44c.

```

<function ieedtod 120d>≡ (155a)
/// ?? -> <>
void
ieeedtod(Ieee *ieee, double native)
{
  double fr, ho, f;
  int exp;

  if(native < 0) {
    ieedtod(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()` 120d.

## 11.2.2 Multiplication and accumulation

*<Opcode cases, mul/div/mod opcodes 121a>+≡* (29) *<81b 121f>*  
AMULA,

*<itab elements 121b>+≡* (35c) *<118d 121g>*  
"MULA", LMULA, AMULA,

Uses LMULA.

*<inst rule(arm) 121c>+≡* (78b) *<119a 121h>*  
/\*  
\* 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 121d>+≡* (31c) *<119b 123d>*  
D\_REGREG,

*<outopd() cases(arm) 121e>+≡* (105b) *<120c 123e>*  
case D\_REGREG:  
Bputc(&obuf, a->offset);  
break;

Uses obuf 44c.

## 11.2.3 64-bits multiplication

*<Opcode cases, mul/div/mod opcodes 121f>+≡* (29) *<121a 125c>*  
AMULL,  
AMULAL,  
AMULLU,  
AMULALU,

*<itab elements 121g>+≡* (35c) *<121b 122b>*  
"MULL", LMULL, AMULL,  
"MULAL", LMULL, AMULAL,  
"MULLU", LMULL, AMULLU,  
"MULALU", LMULL, AMULALU,

Uses LMULL.

*<inst rule(arm) 121h>+≡* (78b) *<121c 122c>*  
/\*  
\* MULL hi,lo,r1,r2  
\*/  
| LMULL cond reg ',' reg ',' regreg { outcode(\$1, \$2, &\$3, \$5.reg, &\$7); }

*<type declarations(arm) 121i>+≡* (78a) *<119c 122d>*  
%type <genval> regreg

*<operand rules(arm) 121j>+≡* (78b) *<??>*  
/\* 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 122a>+≡ (29) <83a*  
AMOVN,

*<itab elements 122b>+≡ (35c) <121g 122f>*  
"MOVN", LMOVN, AMOVN,

Uses LMOVN.

*<inst rule(arm) 122c>+≡ (78b) <121h 124a>*

```
/*
 * MOVN
 */
| LMOVN cond ioreg ',' '[' reglist ']'
{
    Gen g;

    g = nullgen;
    g.type = D_CONST;
    g.offset = $6;
    outcode($1, $2, &$3, R_NONE, &g);
}
| LMOVN cond '[' reglist ']' ',' ioreg
{
    Gen g;

    g = nullgen;
    g.type = D_CONST;
    g.offset = $4;
    outcode($1, $2, &g, R_NONE, &$7);
}
```

*<type declarations(arm) 122d>+≡ (78a) <121i 124b>*  
%type <lval> reglist

*<reglist rule 122e>≡ (? 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 122f>+≡ (35c) <122b 122h>*  
".U", LS, C\_UBIT,

Uses LS.

*<constant C\_UBIT(arm) 122g>≡ (148c)*  
#define C\_UBIT (1<<7) /\* up bit \*/

*<itab elements 122h>+≡ (35c) <122f 123a>*

```
".IB", LS, C_PBIT|C_UBIT,
".IA", LS, C_UBIT,
".DB", LS, C_PBIT,
".DA", LS, 0,
```

Uses LS.

`<itab elements 123a>+≡` (35c) <122h 123b>  
`"PW", LS, C_WBIT|C_PBIT,`  
`"WP", LS, C_WBIT|C_PBIT,`  
 Uses LS.

`<itab elements 123b>+≡` (35c) <123a 123c>  
`".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 123c>+≡` (35c) <123b 123h>  
`"CPSR", LPSR, 0,`  
`"SPSR", LPSR, 1,`  
 Uses LPSR.

`<Operand_kind cases 123d>+≡` (31c) <121d  
`D_PSR,`

`<outopd() cases(arm) 123e>+≡` (105b) <121e  
`case D_PSR:`  
`break;`

`<more gen rule 123f>+≡` (90b) <120a  
`| LPSR`  
`{`  
`$$ = nullgen;`  
`$$ .type = D_PSR;`  
`$$ .reg = $1;`  
`}`

`<constant C_FBIT(arm) 123g>≡` (148c)  
`#define C_FBIT (1<<7) /* psr flags-only */`

`<itab elements 123h>+≡` (35c) <123c 123i>  
`".F", LS, C_FBIT,`  
 Uses LS.

## 11.2.6 Mutual exclusion instructions

## 11.2.7 Coprocessors

`<itab elements 123i>+≡` (35c) <123h 124d>  
`"MCR", LSYSTEM, 0,`  
`"MRC", LSYSTEM, 1,`  
 Uses LSYSTEM.

*<inst rule(arm) 124a>+≡* (78b) <122c

```
/*
 * 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) 124b>+≡* (78a) <122d 124e>

```
%type <lval> creg
```

*<creg rule 124c>≡* (? 0—1)

```
creg:
  LCREG
| LC '(' expr ')'
```

```
{
  if($3 < 0 || $3 >= NREG)
    print("register value out of range\n");
  $$ = $3;
}
```

*<itab elements 124d>+≡* (35c) <123i

```
"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) 124e>+≡* (78a) <124b

```
%type <lval> oexpr
```

```

<oeexpr rule 125a>≡ (29) <0—1>
/* for MCR */
oeexpr:
/* empty */ { $$ = 0; }
| ', ' expr { $$ = $2; }

```

## 11.3 Other pseudo opcodes

### 11.3.1 Compiler-only opcodes

```

<Opcode cases, pseudo opcodes 125b>+≡ (29) <114c 125d>
ACASE,
ABCASE,

```

```

<Opcode cases, mul/div/mod opcodes 125c>+≡ (29) <121f>
AMULU,
ADIVU, // VIRTUAL, transformed to call to _divu
AMODU, // VIRTUAL, transformed to call to _modu

```

```

<Opcode cases, pseudo opcodes 125d>+≡ (29) <125b 125h>
ASIGNAME,

```

```

<Register compiler conventions cases 125e>≡ (30) 125f>
// reserved by compiler, calling conventions
REGRET = 0,
REGARG = 0,

```

```

<Register compiler conventions cases 125f>+≡ (30) <125e>
/* 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 125g>≡ (30)
// reserved by linker, long address/offset loading
REGTMP = 11,

```

### 11.3.2 Linker-only opcodes

```

<Opcode cases, pseudo opcodes 125h>+≡ (29) <125d>
AGOK,

```

```

<Sym_kind cases 125i>+≡ (39f) <115b>
N_LINE, // used by linker only

```

## 11.4 TEXT attributes

```

<constant DUPOK(arm) 125j>≡ (148c)
#define DUPOK (1<<1)

```

```

<constant NOPROF(arm) 125k>≡ (148c)
#define NOPROF (1<<0)

```

## 11.5 Other preprocessing directives

### 11.5.1 #define

-D

```
<main() command line processing 126a>+≡ (41c) <70b 140c>
case 'D':
    p = ARGV();
    if(p)
        Dlist[nDlist++] = p;
    break;
```

```
<global Dlist 126b>≡ (154a)
char* Dlist[30];
```

```
<global nDlist 126c>≡ (154a)
int nDlist;
```

```
<assemble() locals 126d>+≡ (43d) <70g
int i;
```

```
<assemble() init Dlist after pinit 126e>≡ (43)
for(i=0; i<nDlist; i++)
    dodefine(Dlist[i]);
```

Uses Dlist 126b, dodefine() 126h, and nDlist 126c.

```
<Sym macro fields 126f>≡ (35d)
//option<string> for '#define F00 xxx' expansion
char* macro;
```

```
<pinit() initializations 126g>+≡ (47a) <107b
for(i=0; i<NHASH; i++)
    for(s = hash[i]; s != S; s = s->link)
        s->macro = nil;
```

Uses NHASH 36b, S 36d, and hash 36a.

```
<function dodefine 126h>≡ (160)
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 140e⟩
}

```

Uses lookup() 37c and symb 37a.

**#define**

```

⟨constant NARG 127a⟩≡ (160)
#define NARG 25

```

```

⟨function macdef 127b⟩≡ (160)

```

```

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 140f>
return;

```

```

bad:
  if(s == S)
    yyerror("syntax in #define");
  else
    yyerror("syntax in #define: %s", s->name);
  macend();
}

```

Uses NARG-2 127a, S 36d, VARMAC-1 130a, allocn() 145c, fi 50c, filbuf() 50g, getc() 64c, getnsc() 68b, getsym() 67b, getsymdots() 130b, hunk 144a, lineno 47b, macend() 69b, symb 37a, unget() 68d, and yyerror() 142c.

```

<constant VARMAC 130a>≡ (160)
#define VARMAC 0x80

```

```

<function getsymdots 130b>≡ (160)
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 36d, getc() 64c, getnsc() 68b, getsym() 67b, slookup() 36e, unget() 68d, and yyerror() 142c.

## Macro expansion

```

<yylex() if macro symbol 130c>≡ (56c)
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 54b](#), [fi 50c](#), [ionext 48f](#), [iostack 48c](#), [macexpand\(\) 131](#), [newio\(\) 48g](#), [peekc 54a](#), and [pushio\(\) 71b](#).

*<function macexpand 131>*≡

(160)

```
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 141a>
        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();
                    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;
            }
        }
}
```

```

        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:
        ungetc(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 141b⟩
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 127a, VARMAC-1 130a, getc() 64c, getnsc() 68b, unget() 68d, and yyerror() 142c.

## 11.5.2 #undef

```

⟨function macund 133⟩≡
void
macund(void)
{
    Sym *s;

    s = getsym();
    macend();
    if(s == S) {
        yyerror("syntax in #undef");
        return;
    }
    s->macro = nil;
}

```

Uses S 36d, getsym() 67b, macend() 69b, and yyerror() 142c.

(160)

### 11.5.3 #ifdef

*(function macif 134)*≡

(160)

```
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 36d, getc() 64c, getcom() 73e, getsym() 67b, macend() 69b, and yyerror() 142c.

`<domacro() set s to endif symbol if no symbol 135a>≡` (67a)

```
if(s == S)
    s = slookup("endif");
```

Uses S 36d and slookup() 36e.

## 11.5.4 #pragma

`<function macprag 135b>≡` (160)

```
void
macprag(void)
{
    Sym *s;
    <macprag() locals 135c>

    s = getsym();

    <macprag() if pragma lib 135d>
    else {
        while(getnsc() != '\n')
            ;
        return;
    }
}
```

Uses getnsc() 68b and getsym() 67b.

## 11.6 #pragma lib and automagic linking

`<macprag() locals 135c>≡` (135b)

```
Hist *h;
char *hp;
int c0, c;
```

`<macprag() if pragma lib 135d>≡` (135b)

```
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 110d, alloc() 145b, ehist 110e, getc() 64c, getcom() 73e, getnsc() 68b, hist 110b, lineno 47b, macend() 69b, symb 37a, unget() 68d, and yyerror() 142c.

## 11.7 Processing multiple files

```

<main() locals 136a>+≡
    int nout, nproc, status;
    int i, c;

```

(41c) <70a

```

<main() multiple files handling 136b>≡
    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)
                        errexit();
                    if(status)
                        c++;
                    nout--;
                    continue;
                }
                if(i == 0) {

```

(41c)

```
        print("%s:\n", *argv);
        if(assemble(*argv)
            errexit();
            exits(nil);
    }
    nout++;
    argc--;
    argv++;
}
i = mywait(&status);
if(i < 0) {
    if(c)
        errexit();
        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 140a>≡ (154a)
    bool debug[256];
```

```
<main() remaining initializations 140b>+≡ (41c) <69f
    memset(debug, false, sizeof(debug));
```

```
<main() command line processing 140c>+≡ (41c) <126a
    default:
        c = ARGV();
        if(c >= 0 || c < sizeof(debug))
            debug[c] = true;
        break;
```

### A.1 Line information debugging: 5a -f

```
<linehist() debug 140d>≡ (110g)
    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 140a and lineno 47b.

### A.2 Macro debugging: 5a -m

```
<dodefine() debug 140e>≡ (126h)
    if(debug['m'])
        print("#define (-D) %s %s\n", s->name, s->macro+1);
```

Uses debug 140a.

```
<macdef() debug 140f>≡ (127b)
    if(debug['m'])
        print("#define %s %s\n", s->name, s->macro+1);
```

Uses debug 140a.

`<macexpand() debug part1 141a>≡` (131)

```
if(debug['m'])  
    print("#expand %s %s\n", s->name, ob);
```

Uses debug 140a.

`<macexpand() debug part2 141b>≡` (131)

```
if(debug['m'])  
    print("#expand %s %s\n", s->name, ob);
```

Uses debug 140a.

# 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 errexit 142a>≡ (154c)
  /// main | assemble | yyerror | ... -> <>
  void
  errexit(void)
  {
```

```
    if(outfile)
        remove(outfile);
    exits("error");
  }
```

Uses `outfile 42a`.

```
<global nerrors 142b>≡ (154a)
  int nerrors = 0;
```

Uses `nerrors 142b`.

```
<function yyerror 142c>≡ (154c)
  /// assemble | yylex | yyparse | ... -> <>
  void
  yyerror(char *a, ...)
  {
```

```
    char buf[200];
    va_list arg;

    <yyerror() when called from yyparse 143>

    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");
        errexit();
    }
  }
```

Uses `errexit() 142a`, `lineno 47b`, `nerrors 142b`, and `prfile() 113b`.

`<yyerror() when called from yyparse 143>`≡

(142c)

```
/*  
 * hack to intercept message from yaccpar  
 */  
if(strcmp(a, "syntax error") == 0) {  
    yyerror("syntax error, last name: %s", symb);  
    return;  
}
```

Uses `symb 37a` and `yyerror() 142c`.

# 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 144a>*≡ (154a)  
`char* hunk;`

*<global nhunk 144b>*≡ (154a)  
`long nhunk = 0;`

Uses `nhunk 144b`.

*<global thunk 144c>*≡ (154a)  
`long thunk;`

*<constant NHUNK 144d>*≡ (150b)  
`#define NHUNK 10000`

*<function gethunk 144e>*≡ (152)  
`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");  
 errexit();  
 }  
 hunk = h;  
 nhunk = nh;  
 thunk += nh;  
}`

Uses `NHUNK 144d`, `errexit() 142a`, `hunk 144a`, `nhunk 144b`, `thunk 144c`, and `yyerror() 142c`.

*<constant MAXALIGN 145a>*≡ (150b)  
#define MAXALIGN 7

*<function alloc 145b>*≡ (152)  
/\*  
 \* 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 145a, gethunk() 144e, hunk 144a, and nhunk 144b.

*<function allocn 145c>*≡ (152)  
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() 144e, hunk 144a, and nhunk 144b.

# 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 146a>≡
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 146b>≡
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 148a>≡`

```
// 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 106c>`

`<struct ieee 117d>`

```
typedef struct ieee Ieee;
```

#### E.1.2 include/obj/5.out.h

`<constant SYMDEF(arm) 148b>≡`

```
/*
 * this is the ranlib header
 */
#define SYMDEF "__SYMDEF"

l
```

(148c)

`<include/obj/5.out.h 148c>≡`

```
// 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) 30>`

`<constant R_NONE(arm) 31a>`

`<enum Fregister(arm) 118e>`

`<enum Opcode(arm) 29>`

`<enum Operand_kind(arm) 31c>`

`<enum Sym_kind(arm) 39f>`

`<constant NSNAME 32c>`

```
// Attributes
```

*<constant NOPROF(arm) 125k>*

*<constant DUPOK(arm) 125j>*

*<constant C\_SCOND(arm) 100b>*

*<constant C\_SBIT(arm) 100e>*

*<constant C\_PBIT(arm) 100g>*

*<constant C\_WBIT(arm) 100h>*

*<constant C\_FBIT(arm) 123g>*

*<constant C\_UBIT(arm) 122g>*

`#define COND_ALWAYS 14`

*<constant SYMDEF(arm) 148b>*

## E.2 assemblers/misc/

### E.2.1 assemblers/misc/data2s.c

*<function main 149a>*≡

```
void
main(int argc, char *argv[])
{
    Biobuf bin, bout;
    long len, slen;
    int c;

    if(argc != 2){
        fprintf(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);
}
```

(149b)

*<assemblers/misc/data2s.c 149b>*≡

`#include <u.h>`

```
#include <libc.h>
#include <bio.h>

<function main 149a>
```

## E.3 assemblers/aa/

### E.3.1 assemblers/aa/aa.h

```
<enum platform 150a>≡ (150b)
/*
 * 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 150b>≡
#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$0"

//-----
// Data structures and constants
//-----

typedef struct  Sym Sym;
typedef struct  Io Io;
typedef struct  Hist Hist;

<constant MAXALIGN 145a>
<constant NSYMB 37b>
<constant BUFSIZ 50b>
<constant HISTSZ 113a>
<constant NINCLUDE 69d>
<constant NHUNK 144d>
<constant NHASH 36b>
<constant STRINGSZ 72d>

<constant EOF 52a>
<constant IGN 54b>
<function GETC 50f>

<struct Sym 35d>
```

*<constant S 36d>*

*<struct Fi 50d>*

*<struct Io 48a>*

*<constant I 48e>*

*<constant FD\_NONE 48b>*

*<struct Htab 106d>*

*<struct Hist 110a>*

*<constant H 110d>*

```
//-----  
// 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  ieeeedtod(Ieee *ieee, double native);

// error.c
void  yyerror(char*, ...);
void  errexit(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 150a>*

Uses Hist 150b, Io 150b, and Sym 150b.

## E.3.2 assemblers/aa/utils.c

```

<assemblers/aa/utils.c 152>≡
#include "aa.h"

```

*<function gethunk 144e>*

*<function alloc 145b>*

*<function allocn 145c>*

*<function mycreat 153a>*

*<function mywait 153b>*

*<function systemtype 153c>*

*<function pathchar 153d>*

```
<function mycreat 153a>≡ (152)
int
mycreat(char *n, int p)
{
    return create(n, 1, p);
}
```

```
<function mywait 153b>≡ (152)
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 153c>≡ (152)
int
systemtype(int sys)
{
    return sys & Plan9;
}
```

Uses Plan9 150a.

```
<function pathchar 153d>≡ (152)
int
pathchar(void)
{
    return '/';
}
```

### E.3.3 assemblers/aa/globals.c

```
<assemblers/aa/globals.c 154a>≡  
#include "aa.h"  
  
<global hash 36a>  
<global pc 89a>  
<global outfile 42a>  
<global obuf 44c>  
<global pass 43c>  
<global pathname 43a>  
  
<global thechar 41a>  
<global thestring 41b>  
  
<global iostack 48c>  
<global iofree 49a>  
<global ionext 48f>  
<global fi 50c>  
  
<global symb 37a>  
<global peekc 54a>  
  
<global include 69c>  
<global ninclude 69e>  
<global Dlist 126b>  
<global nDlist 126c>  
  
<global h 106b>  
<global symcounter 107a>  
  
<global lineno 47b>  
<global hist 110b>  
<global ehist 110e>  
  
<global debug 140a>  
  
<global nerrors 142b>  
  
<global hunk 144a>  
<global nhunk 144b>  
<global thunk 144c>
```

### E.3.4 assemblers/aa/lookup.c

```
<assemblers/aa/lookup.c 154b>≡  
#include "aa.h"  
  
// syminit() in lookup() depends on LNAME token defined in a.y,  
// so it can't be defined here.  
  
<function slookup 36e>  
  
<function lookup 37c>
```

### E.3.5 assemblers/aa/error.c

```
<assemblers/aa/error.c 154c>≡
```

```
#include "aa.h"

⟨function errexit 142a⟩

⟨function yyerror 142c⟩
```

### E.3.6 assemblers/aa/float.c

```
⟨assemblers/aa/float.c 155a⟩≡
#include "aa.h"

⟨function ieeeedtod 120d⟩
```

### E.3.7 assemblers/aa/hist.c

```
⟨assemblers/aa/hist.c 155b⟩≡
#include "aa.h"

⟨function linehist 110g⟩

⟨function prfile 113b⟩
```

### E.3.8 assemblers/aa/lexbody.c

```
⟨assemblers/aa/lexbody.c 155c⟩≡
#include "aa.h"

/*
 * common code for all the assemblers
 */

⟨function setinclude 70c⟩

⟨function pushio 71b⟩

⟨function newio 48g⟩

⟨function newfile 49d⟩

// 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') {
// lineneno++;
// return ',';
// }
// goto l0;
// }
// if(isalpha(c))
// goto talph;
// if(isdigit(c))
// goto tnum;
// switch(c)
// {
// case '\n':
// lineneno++;
// 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 LFCNST;
// }
// 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");
//         errexit();
//       }
//     }
//   }
//   if(c1 == '*') {
//     for(;;) {
//       c = GETC();
//       while(c == '*') {
//         c = GETC();
//         if(c == '/')
//           goto l0;
//       }
//       if(c == EOF) {
//         yyerror("eof in comment");
//         errexit();
//       }
//       if(c == '\n')
//         lineno++;
//     }
//   }
//   break;
//
// default:
//   return c;
// }
// peekc = c1;
// return c;
//}

```

*<function getc 64c>*  
*<function getnsc 68b>*  
*<function unget 68d>*  
*<function escchar 62b>*  
*<function pinit 47a>*  
*<function filbuf 50g>*

### E.3.9 assemblers/aa/macbody.c

*<assemblers/aa/macbody.c 160>*≡  
#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 130a>*  
*<function getnsn 76e>*  
*<function getsym 67b>*  
*<function getsymdots 130b>*  
*<function getcom 73e>*  
*<function dodefine 126h>*  
*<global mactab 66b>*  
*<function domacro 67a>*  
*<function macund 133>*  
*<constant NARG 127a>*  
*<function macdef 127b>*  
*<function macexpand 131>*  
*<function macinc 71a>*  
*<function maclin 75a>*  
*<function macif 134>*  
*<function macprag 135b>*

*<function macend 69b>*

## E.4 assemblers/5a/

### E.4.1 assemblers/5a/a.h

```
<assemblers/5a/a.h 161a>≡
#include "../aa/aa.h"
#include <5.out.h>

//-----
// Data structures and constants
//-----

<constant FPCHIP(arm) 117c>

<constant Always(arm) 100a>

<struct Gen(arm) 31b>
typedef struct Gen Gen;

//-----
// Globals
//-----

// globals.c
extern Gen nullgen;

//-----
// Functions
//-----

// lex.c (for y.tab.c, main.c)
<signature yylex 53a>
void cinit(void);

// y.tab.c from a.y (for main.c)
int yyparse(void);

// obj.c (for main.c)
<signature outcode(arm) 32d>
void outhist(void);
```

Uses Gen *31b*.

### E.4.2 assemblers/5a/globals.c

```
<assemblers/5a/globals.c 161b>≡
#include "a.h"

<global nullgen 32a>
```

### E.4.3 assemblers/5a/lex.c

```
<assemblers/5a/lex.c 162a>≡
#include "a.h"
#include "y.tab.h"

<struct Itab(arm) 35b>

<global itab(arm) 35c>

<function cinit(arm) 42c>

<function syminit 38a>

// 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 53b>

// #include "../cc/macbody"
```

### E.4.4 assemblers/5a/a.y

### E.4.5 assemblers/5a/obj.c

```
<assemblers/5a/obj.c 162b>≡
#include "a.h"

<function zname(arm) 107c>

<function outopd(arm) 105b>

<global bcode(arm) 105a>

<function symidx_of_symopt 108b>

<function outcode(arm) 104a>

<function outhist(arm) 114b>
```

### E.4.6 assemblers/5a/main.c

```
<assemblers/5a/main.c 162c>≡
#include "a.h"

// forward decls
int assemble(char*);
void cclean(void);

<function main(arm) 41c>

<function assemble 43d>

<function cclean(arm) 44d>
```

## E.5 lib\_core/libc/arm/

### E.5.1 lib\_core/libc/arm/div.s

`<lib_core/libc/arm/div.s 163>`≡

```
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 quotient 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()`: [37d](#), [49b](#), [110g](#), [135d](#), [145b](#)  
`alloca()`: [43b](#), [127b](#), [145c](#)  
Always: [44d](#), [99g](#), [100a](#), [104b](#), [114b](#)  
`assemble()`: [43d](#)  
`bcode-56`: [104b](#), [105a](#)  
`BUFSIZ`: [50b](#), [50g](#)  
`cclean()`: [43d](#), [44b](#), [44d](#)  
`cinit()`: [42c](#)  
`debug`: [140a](#), [140d](#), [140e](#), [140f](#), [141a](#), [141b](#)  
`Dlist`: [126b](#), [126e](#)  
`dodefine()`: [126e](#), [126h](#)  
`domacro()`: [66a](#), [67a](#)  
`ehist`: [110e](#), [110g](#), [135d](#)  
`EOF`: [52a](#), [52b](#), [53b](#), [55d](#), [60b](#), [62a](#), [62b](#), [63e](#), [64c](#), [65](#)  
`errorexit()`: [44a](#), [49c](#), [49e](#), [55d](#), [64c](#), [72a](#), [142a](#), [142c](#), [144e](#)  
`escchar()`: [60b](#), [62b](#), [65](#)  
`FD_NONE`: [47a](#), [48b](#), [48g](#), [49d](#)  
`Fi`: [50c](#), [50d](#)  
`fi`: [49d](#), [50c](#), [50g](#), [51a](#), [51b](#), [71b](#), [127b](#), [130c](#)  
`Fi.c`: [50d](#)  
`Fi.p`: [50d](#)  
`filbuf()`: [50g](#), [127b](#)  
`FPCHIP`: [59c](#), [117c](#)  
`Gen`: [31b](#), [161a](#)  
`Gen.reg`: [31b](#)  
`Gen.sym`: [39d](#)  
`Gen.symkind`: [39e](#)  
`Gen.type`: [31b](#)  
`Gen (typedef)`: [161a](#)  
`GETC`: [50f](#), [53b](#), [55d](#), [56c](#), [57d](#), [58b](#), [59c](#), [60a](#), [64c](#)  
`getc()`: [62b](#), [63d](#), [64c](#), [67b](#), [68b](#), [72e](#), [73e](#), [75c](#), [76b](#), [76e](#), [127b](#), [130b](#), [131](#), [134](#), [135d](#)  
`getcom()`: [73d](#), [73e](#), [76d](#), [134](#), [135d](#)  
`gethunk()`: [144e](#), [145b](#), [145c](#)  
`getnsc()`: [67b](#), [68b](#), [69b](#), [72e](#), [73e](#), [76e](#), [127b](#), [130b](#), [131](#), [135b](#), [135d](#)  
`getnsn()`: [75c](#), [76e](#)  
`getsym()`: [67a](#), [67b](#), [127b](#), [130b](#), [133](#), [134](#), [135b](#)  
`getsymdots()`: [127b](#), [130b](#)  
`H`: [110d](#), [110g](#), [113c](#), [114b](#), [135d](#)

h: [106b](#), [107b](#), [108b](#)  
hash: [36a](#), [37c](#), [37d](#), [39a](#), [126g](#)  
Hist: [150b](#), [150b](#)  
hist: [110b](#), [110g](#), [113c](#), [114b](#), [135d](#)  
Hist (typedef): [150b](#)  
HISTSZ: [113a](#), [113b](#), [113c](#), [113d](#), [114a](#)  
Htab: [106b](#), [106d](#)  
Htab.sym: [106d](#)  
Htab.symkind: [106d](#)  
hunk: [127b](#), [144a](#), [144e](#), [145b](#), [145c](#)  
I: [48c](#), [48e](#), [49a](#), [49b](#), [52b](#), [72a](#)  
ieeedtod(): [120c](#), [120d](#), [120d](#)  
IGN: [54a](#), [54b](#), [54c](#), [54d](#), [64d](#), [130c](#)  
include: [69c](#), [69g](#), [70c](#), [70e](#), [70f](#), [73a](#)  
Io: [150b](#), [150b](#)  
Io (typedef): [150b](#)  
iofree: [49a](#), [49a](#), [49b](#), [51b](#)  
ionext: [48f](#), [48g](#), [49d](#), [130c](#)  
iostack: [48c](#), [48c](#), [49d](#), [50g](#), [51b](#), [71b](#), [130c](#)  
Itab: [35b](#), [35c](#)  
itab: [35c](#), [39a](#)  
Itab.name: [35b](#)  
Itab.type: [35b](#)  
Itab.value: [35b](#)  
LARITH: [79e](#), [81a](#), [81c](#), [81e](#), [82a](#), [82c](#)  
LARITHFLOAT: [118b](#)  
LAT: [93c](#)  
LBCOND: [86a](#)  
LBRANCH: [84e](#), [87d](#)  
LC: [124d](#)  
LCMP: [85c](#), [87b](#)  
LCMPFLOAT: [118b](#)  
LCOND: [99g](#)  
LCONST: [57d](#), [58b](#), [59c](#), [60b](#)  
LCREG: [124d](#)  
LDATA: [98b](#)  
LDEF: [97d](#)  
LEND: [99c](#)  
LF: [118c](#)  
LFCONST: [59c](#)  
LFCR: [118d](#)  
LFP: [94g](#)  
LFREG: [118c](#)  
linehist(): [110f](#), [110g](#), [111a](#), [111b](#)  
lineno: [47a](#), [47b](#), [55c](#), [55d](#), [64c](#), [68d](#), [104a](#), [110g](#), [127b](#), [135d](#), [140d](#), [142c](#)  
LLAB: [57a](#)  
LMISC: [35c](#)  
LMOV: [83b](#), [117f](#)  
LMOVM: [122b](#)

LMULA: [121b](#)  
LMULL: [121g](#)  
LMVN: [82e](#)  
LNAME: [38a](#), [57a](#)  
lookup(): [36e](#), [37c](#), [56c](#), [67b](#), [126h](#)  
LPC: [97b](#)  
LPSR: [123c](#)  
LR: [91a](#)  
LREG: [90g](#)  
LRET: [88a](#), [88g](#)  
LS: [100d](#), [100f](#), [122f](#), [122h](#), [123a](#), [123b](#), [123h](#)  
LSB: [94g](#)  
LSCONST: [65](#)  
LSP: [94g](#)  
LSQRTFLOAT: [118b](#)  
LSWAP: [84b](#)  
LSWI: [88d](#)  
LSYSTEM: [123i](#)  
LVAR: [57a](#)  
LWORD: [98e](#)  
macdef(): [66b](#), [127b](#)  
macend(): [66b](#), [69a](#), [69b](#), [72f](#), [76c](#), [127b](#), [133](#), [134](#), [135d](#)  
macexpand(): [130c](#), [131](#)  
macif(): [67a](#), [134](#)  
macinc(): [66b](#), [71a](#)  
maclin(): [66b](#), [75a](#)  
macprag(): [66b](#), [135b](#)  
mactab: [66b](#), [67a](#)  
macund(): [66b](#), [133](#)  
main-57(): [149a](#)  
MAXALIGN: [145a](#), [145b](#)  
mycreat(): [43d](#), [153a](#)  
mywait(): [153b](#)  
NARG-2: [127a](#), [127b](#), [131](#)  
nDlist: [126c](#), [126e](#)  
nerrors: [43d](#), [44b](#), [142b](#), [142b](#), [142c](#)  
newfile(): [47a](#), [49d](#), [71a](#)  
newio(): [47a](#), [48g](#), [71a](#), [130c](#)  
NHASH: [36a](#), [36b](#), [38c](#), [39a](#), [126g](#)  
NHUNK: [144d](#), [144e](#)  
nhunk: [144b](#), [144b](#), [144e](#), [145b](#), [145c](#)  
NINCLUDE: [69c](#), [69d](#)  
ninclude: [69e](#), [70c](#), [70e](#), [70f](#), [73a](#)  
NSYMB: [37a](#), [37b](#), [67b](#), [68a](#)  
nullgen: [32a](#), [42d](#), [44d](#), [65](#), [114b](#)  
obuf: [43d](#), [44c](#), [44d](#), [104a](#), [105b](#), [106a](#), [107c](#), [114b](#), [115a](#), [120c](#), [121e](#)  
outcode(): [44d](#), [104a](#)  
outfile: [42a](#), [42a](#), [43d](#), [44a](#), [46](#), [142a](#)  
outhist(): [43d](#), [114b](#)

outopd(): [104a](#), [105b](#), [114b](#)  
pass: [43c](#), [43d](#), [104a](#)  
pathchar(): [153d](#)  
pathname: [43a](#), [43b](#), [115d](#)  
pc: [89a](#), [89b](#), [104a](#)  
peekc: [54a](#), [54a](#), [54c](#), [54d](#), [55b](#), [56c](#), [57d](#), [58b](#), [59c](#), [60a](#), [63d](#), [64d](#), [68d](#), [130c](#)  
pinit(): [43d](#), [47a](#)  
Plan9: [150a](#), [153c](#)  
prfile(): [113b](#), [142c](#)  
pushio(): [71a](#), [71b](#), [130c](#)  
S: [36d](#), [37c](#), [39a](#), [42d](#), [67b](#), [107b](#), [108b](#), [126g](#), [127b](#), [130b](#), [133](#), [134](#), [135a](#)  
setinclude(): [70c](#), [70h](#)  
slookup(): [36e](#), [39a](#), [130b](#), [135a](#)  
STRINGSZ: [72c](#), [72d](#)  
Sym: [150b](#), [150b](#)  
Sym (typedef): [150b](#)  
sym: [36e](#), [37a](#), [37c](#), [37d](#), [38c](#), [56c](#), [57c](#), [57d](#), [59c](#), [60a](#), [67b](#), [68a](#), [71a](#), [73a](#), [73c](#), [75a](#), [75c](#), [76a](#), [126h](#), [127b](#), [135d](#),  
[143](#)  
symcounter: [107a](#), [107b](#), [108a](#), [108b](#), [108d](#)  
symidx\_of\_symopt(): [108a](#), [108b](#)  
syminit(): [37d](#), [38a](#)  
systemtype(): [153c](#)  
thechar: [41a](#), [44a](#), [46](#), [49e](#)  
thestring: [41b](#), [70h](#)  
thunk: [144c](#), [144e](#)  
unget(): [67b](#), [68d](#), [72f](#), [76c](#), [76e](#), [127b](#), [130b](#), [131](#), [135d](#)  
Unix: [150a](#)  
VARMAC-1: [127b](#), [130a](#), [131](#)  
yyerror(): [44a](#), [49c](#), [49e](#), [55d](#), [59c](#), [60b](#), [63e](#), [64c](#), [65](#), [68a](#), [69a](#), [70f](#), [72a](#), [72f](#), [73e](#), [76c](#), [127b](#), [130b](#), [131](#), [133](#),  
[134](#), [135d](#), [142c](#), [143](#), [144e](#)  
yylex(): [53b](#)  
yylval: [57a](#), [57d](#), [58b](#), [59c](#), [60b](#), [65](#)  
yyparse(): [43d](#)  
zname(): [107c](#), [108b](#)  
\_\_anon\_enum\_1: [150a](#)  
\_\_anon\_struct\_1.macf: [66b](#)  
\_\_anon\_struct\_1.macname: [66b](#)  
\_\_anon\_struct\_1: [66b](#), [66b](#)  
\_\_anon\_struct\_2.dval: [31b](#)  
\_\_anon\_struct\_2.offset: [31b](#)  
\_\_anon\_struct\_2.sval: [31b](#)  
\_\_anon\_struct\_2: [31b](#), [31b](#)

# Bibliography

- [BLM92] Doug Brown, John Levine, and Tony Mason. *Lex and Yacc*. O'Reilly, 1992. cited page(s) 10, 26, 33, 78, 79
- [Dun00] Jeff Duntemann. *Assembly Language Step-by-step: Programming with DOS and Linux*. Wiley, 2000. cited page(s) 8, 10
- [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, 26
- [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) 9
- [KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988. cited page(s) 10
- [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) 26, 53
- [NS05] Noam Nisan and Shimon Shocken. *The Elements of Computing Systems*. MIT Press, 2005. cited page(s) 80
- [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) 10, 17, 139
- [Pad15a] Yoann Padioleau. *Principia Softwarica: The ARM Emulator 5i*. 2015. cited page(s) 10, 17, 22, 30, 79, 80, 81, 82, 91, 93, 100, 139
- [Pad15b] Yoann Padioleau. *Principia Softwarica: The ARM Linker 5l*. 2015. cited page(s) 8, 10, 15, 19, 24, 44, 83, 91, 102, 138
- [Pad16a] Yoann Padioleau. *Principia Softwarica: (OCaml)Lex and (OCaml)Yacc*. 2016. cited page(s) 78
- [Pad16b] Yoann Padioleau. *Principia Softwarica: The Plan 9 C Compiler 5c*. 2016. cited page(s) 10, 30, 66, 139
- [Pad16c] Yoann Padioleau. *Principia Softwarica: The Plan 9 Core Libraries*. 2016. cited page(s) 14, 16, 42, 46, 58, 60

- [Pad16d] Yoann Padioleau. *Principia Softwarica: The Plan 9 Debuggers and Tracers*. 2016. cited page(s) 109, 139
- [PH13] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 2013. cited page(s) 10
- [PH16] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface (ARM Edition)*. Morgan Kaufmann, 2016. cited page(s) 10
- [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, 10
- [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) 10
- [Sea01] David Seal. *ARM, Architecture Reference Manual*. Addison-Wesley, 2001. cited page(s) 8, 22
- [Tan88] Andrew S Tanenbaum. *Structured Computer Organization*. Prentice Hall, 1988. cited page(s) 10