Principia Softwarica: The ARM Assembler `5a`

OCaml edition

version 0.1

Yoann Padioleau
yoann.padioleau@gmail.com

with code from
Rob Pike and Yoann Padioleau

March 10, 2026

Contents

1 Introduction **6**
- 1.1 Motivations . . . 6
- 1.2 The Plan 9 ARM assembler: `5a` . . . 6
- 1.3 Other assemblers . . . 7
- 1.4 Getting started . . . 8
- 1.5 Requirements . . . 9
- 1.6 About this document . . . 9
- 1.7 Copyright . . . 10
- 1.8 Acknowledgments . . . 10

2 Overview **11**
- 2.1 Assembler principles . . . 11
- 2.2 `5a` command-line interface . . . 12
- 2.3 `helloworld.s` . . . 12
  - 2.3.1 Background . . . 12
  - 2.3.2 The program . . . 13
  - 2.3.3 Pseudo instructions . . . 14
  - 2.3.4 Labels . . . 15
  - 2.3.5 Assignments . . . 15
  - 2.3.6 Addressing modes . . . 15
  - 2.3.7 Pseudo registers . . . 15
  - 2.3.8 Call stack . . . 16
  - 2.3.9 System calls . . . 17
  - 2.3.10 Function calls . . . 18
  - 2.3.11 Virtual instructions . . . 18
  - 2.3.12 C calling conventions . . . 19
  - 2.3.13 Data layout . . . 20
- 2.4 The ARM architecture . . . 21
- 2.5 Input assembly language . . . 21
  - 2.5.1 Lexical elements . . . 21
  - 2.5.2 Syntactical elements . . . 22
  - 2.5.3 Opcodes . . . 22
  - 2.5.4 Operands . . . 22
  - 2.5.5 Addressing modes . . . 22
  - 2.5.6 Advanced features . . . 23
- 2.6 Output object format . . . 23
- 2.7 Code organization . . . 23
- 2.8 Software architecture . . . 23
- 2.9 Book structure . . . 23

3 Core Data Structures **24**
3.1 Tokens . . . 24
3.1.1 General assembly tokens . . . 24
3.1.2 ARM assembly tokens . . . 24
3.1.3 Converting tokens . . . 26
3.2 Program AST . . . 26
3.2.1 Location and other basic types . . . 26
3.2.2 General assembly instructions . . . 27
3.2.3 Pseudo instructions . . . 27
3.2.4 Virtual instructions . . . 28
3.2.5 Labels and jumps . . . 28
3.2.6 Entities . . . 28
3.2.7 ARM assembly instructions . . . 29
3.2.8 Special registers . . . 29
3.2.9 Arithmetic instructions . . . 30
3.2.10 Memory instructions . . . 30
3.2.11 Branching instructions . . . 31
3.2.12 System instructions . . . 31
3.3 Object file . . . 31

4 Main Functions **32**
4.1 `main()` . . . 32
4.2 `assemble()` . . . 34
4.3 `parse()` . . . 35

5 Lexing **37**
5.1 Overview . . . 37
5.2 Spaces and comments . . . 39
5.3 Newlines and semicolons . . . 39
5.4 Punctuations and operators . . . 39
5.5 Mnemonics and labels . . . 40
5.6 Numbers . . . 41
5.6.1 Decimal numbers . . . 42
5.6.2 Hexadecimal and octal numbers . . . 42
5.6.3 Floating-point numbers . . . 42
5.7 Characters . . . 42
5.7.1 Escaped sequences . . . 43
5.7.2 Triple quotes . . . 43
5.7.3 Escaped newlines . . . 43
5.8 Strings . . . 43

6 Parsing **44**
6.1 Overview . . . 44
6.2 Grammar overview . . . 44
6.3 Program and lines . . . 47
6.4 Instructions . . . 47
6.4.1 Arithmetic and logic . . . 47
6.4.2 Memory . . . 47
6.4.3 Control flow . . . 47
6.4.4 Software interrupt . . . 48

6.5 Label definitions . . . 48
6.6 Operands . . . 48
6.6.1 Registers . . . 48
6.6.2 Immediate constants . . . 48
6.6.3 ARM shifted registers . . . 48
6.6.4 Memory (de)references, pointers . . . 49
6.6.5 Named memory locations, symbols . . . 49
6.6.6 Code references, labels . . . 50
6.7 Pseudo instructions . . . 50
6.7.1 `TEXT/GLOBL` . . . 51
6.7.2 `WORD/DATA` . . . 51
6.8 ARM conditional execution . . . 51
6.9 Special bits . . . 51
6.9.1 Arithmetic instructions and `.S` . . . 51
6.9.2 Moves and `.P/.W` . . . 51

7 Resolving **52**

8 Object Code Generation **54**
8.1 Object format . . . 54
8.2 Program output . . . 54
8.3 Object file symbol table . . . 54

9 Debugging Support **55**
9.1 Line origin history . . . 55
9.2 Recording history . . . 55
9.3 Displaying history . . . 55
9.4 Saving history . . . 55

10 Advanced Topics **56**
10.1 Other assembly language features . . . 56
10.1.1 Constant expressions . . . 56
10.2 Other instructions and registers . . . 56
10.2.1 Floating-point numbers . . . 56
10.2.2 Multiplication and accumulation . . . 57
10.2.3 64-bits multiplication . . . 57
10.2.4 Moving multiple registers at the same time . . . 57
10.2.5 Program status register . . . 57
10.2.6 Mutual exclusion instructions . . . 57
10.2.7 Coprocessors . . . 57
10.3 Other pseudo opcodes . . . 58
10.3.1 Compiler-only opcodes . . . 58
10.3.2 Linker-only opcodes . . . 58
10.4 `TEXT` attributes . . . 58

11 Conclusion **59**

A Debugging **60**
A.1 AST debugging: `5a -dump_ast` . . . 60
A.2 Line information debugging: `5a -f` . . . 60
A.3 Macro debugging: `5a -m` . . . 60

B Examples of Assembly Programs **61**
B.1 `hello.s` and `pwrite.s` . . . 61
B.2 `memset.s` . . . 61
B.3 `div.s` . . . 61
B.4 `main9.s` . . . 61
B.5 `tas.s` . . . 61
B.6 `getcallerpc.s` . . . 61

C Extra Code **62**
C.1 `objects/Ast_asm.ml` . . . 62
C.2 `objects/Ast_asm5.ml` . . . 65
C.3 `objects/Object_file.mli` . . . 67
C.4 `objects/Object_file.ml` . . . 67
C.5 `CLI.mli` . . . 68
C.6 `CLI.ml` . . . 68
C.7 `Lexer_asm.mll` . . . 69
C.8 `Main.ml` . . . 69
C.9 `Parser_asm.ml` . . . 69
C.10 `Parse_asm5.mli` . . . 70
C.11 `Parse_asm5.ml` . . . 70
C.12 `Parser_asm5.mly` . . . 71
C.13 `Resolve_labels.mli` . . . 71
C.14 `Resolve_labels.ml` . . . 71
C.15 `Token_asm.ml` . . . 71

Glossary **72**

Indexes **73**

References **73**

Chapter 1

Introduction

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

1.1 Motivations

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

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

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

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

1.2 The Plan 9 ARM assembler: `5a`

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

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

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

Like for the other Principia Softwarica books covering the development toolchain, I chose the ARM architecture [Sea01] variant, in this case of the Plan 9 assembler `5a`, and not for instance the x86 variant `8a`, for reasons of simplicity. Indeed, RISC machines are far simpler than CISC machines. Moreover, the availability under Plan 9 of an ARM emulator (called `5i`) helps to understand the semantics of the assembly instructions used in `5a`.

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

1.3 Other assemblers

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

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

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

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

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

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

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

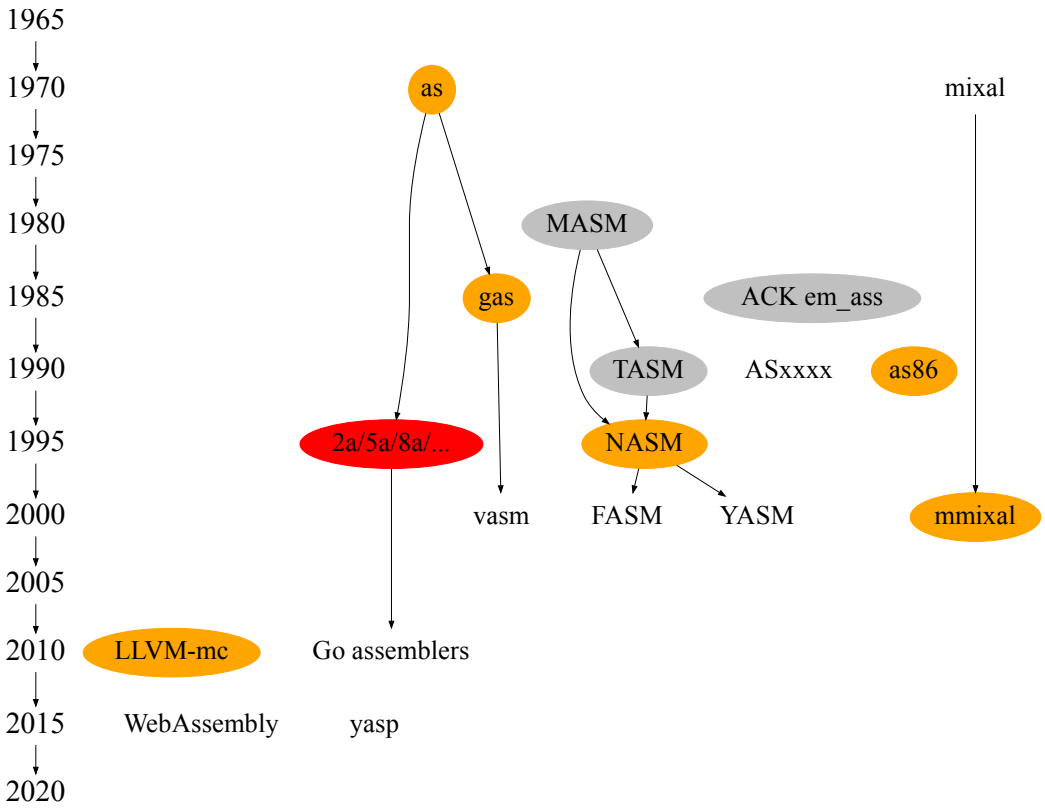

Figure 1.1: Assemblers timeline

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

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

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

1.4 Getting started

To play with `5a`, you will first need to install the Plan 9 fork used in Principia Softwarica (see http://www.principia-softwarica.org/wiki/Install). Once installed, you can test `5a` under Plan 9 with the following commands:

```
1   $ cd /tests/5a
2   $ 5a helloworld.s
```

[8] https://www-cs-faculty.stanford.edu/~knuth/mmixware.html
[9] https://golang.org/doc/asm

```
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[10]).

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 6, 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 [Pad16a] 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`[11] in my Plan 9 repository. You can also consult the documentation of the Plan 9 assemblers [Pik93] (available also in `assemblers/docs/asm.pdf`).

1.6 About this document

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

[10] https://www.raspberrypi.org/

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

1.7 Copyright

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 13a⟩≡
```
#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 [Pad16b]). If I expand the code of this function, and simplify things, I will get this C program:

⟨helloworld2.c 13b⟩≡
```
#include <u.h>
#include <libc.h>

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

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

⟨prototype pwrite in libc.h 13c⟩≡
```
extern long pwrite(fdt, void*, long, vlong);
```

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

2.3.2 The program

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

⟨assemblers/5a/tests/helloworld.s 13d⟩≡
```
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
```

[1] 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 B 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 [Pad16b]). In `helloworld.s`, I do not use and so do not link the C library, to simplify the code and the explanations, so I must define a `_main()` at Line 1.

2.3.4 Labels

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

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

2.3.5 Assignments

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

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

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

2.3.6 Addressing modes

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

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

2.3.7 Pseudo registers

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

Why should you use pseudo registers? The advantage of using `2(PC)` instead of `8(R15)` in Asm5 may look small. However, on some architectures the size of instructions is variable. For instance, it is not trivial on x86

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

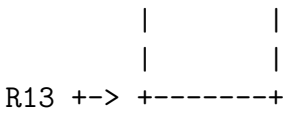

Figure 2.1: Stack content when `hello` started.

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` [3]:

```
$ 5i hello
5i> $r
...
R0  #7fffff70 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 #7fffff58 R14 #0        R15 #1020
```

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

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

Figure 2.2: Stack content before Line 2.

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

Figure 2.3: Stack content before Line 31.

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

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

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

The virtual instruction RET, just like the pseudo instruction TEXT, or the pseudo registers, allows the programmer to think in slightly higher-level terms and let the computer do the appropriate optimizations. In fact, MOVW used in Figure 2.4 is also a virtual instruction. It hides some architecture restrictions and peculiarities regarding memory accesses. in Asm5, you can use the very general MOVW even though the ARM processor supports only the more basic and separate LDR (load) and STR (store) instructions.

Instructions Line 40 though 44 are similar to the instructions we have seen before; they perform the system call exits(0), which terminates the program. This is why I marked a few instructions with a comment indicating

[4] 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 `5l`).

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

R0 = 1
```

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* 20a⟩≡

```
  ...
  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* 20b⟩≡

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

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

2.4 The ARM architecture

The three letters ARM represent different things: a family of RISC machines (Acorn RISC Machines), a family of *instruction set architectures* (ISAs), and finally a family of processors. Up until now, and for the rest of this document, when I use the term ARM I mean the *ARMv6* instruction set architecture. Confusingly, ARMv6 is the instruction set used in the *ARM11* 32-bits processor family[5]. This family powers most smartphones, as well as the Raspberry Pi[6] (an extremely cheap machine popular among electronic hobbyists).

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

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

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

2.5 Input assembly language

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

2.5.1 Lexical elements

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

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

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

2.5.2 Syntactical elements

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

2.5.3 Opcodes

Asm5 opcodes can correspond to different kinds of instructions:

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

2.5.4 Operands

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

- *constants*, which in Asm9 are always prefixed with a dollar, e.g., $42, `$'W'`
- *registers*, named `R0` to `R15` in Asm5
- *shifted registers*, which I will explain in Section 6.6.3
- *pseudo registers*, named `PC`, `SP`, `FP`, and `SB` in Asm9
- *label references*, e.g., `later`
- *memory references*, which I describe below

2.5.5 Addressing modes

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

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

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

   For locals and arguments, it is common to use a symbol as in `x+4(SP)` or `length+4(FP)`. In those cases, the symbol is really just a comment to give a name to the local or argument. However, this comment is kept in the symbol table of the object file, as you will see in Chapter 8, and can be used by debuggers, as you will see in Chapter 9.

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

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

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

2.5.6 Advanced features

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

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

⟨*macro* `MCRR` 23⟩≡
```
#define MCRR(coproc, op, rd, rn, crm) \
   WORD $(0xec400000|(rn)<<16|(rd)<<12|(coproc)<<8| \
          (op)<<4|(crm))
```

The `WORD` pseudo instruction used above will be described in Section 6.7.2.

2.6 Output object format

2.7 Code organization

2.8 Software architecture

2.9 Book structure

Chapter 3

Core Data Structures

3.1 Tokens

3.1.1 General assembly tokens

⟨*type* `Token_asm.t` 24a⟩≡ (71c)

```
type t =
  (* pseudo *)
  | TTEXT | TGLOBL
  | TWORD | TDATA
  (* virtual *)
  | TRET | TNOP
  (* registers *)
  | TR | TF
  | TPC | TSB | TFP | TSP
  | TRx of Ast_asm.register
  | TFx of Ast_asm.fregister
  (* immediate *)
  | TINT of int
  | TFLOAT of float
  | TSTRING of string
  (* names *)
  | TIDENT of string
  (* punctuation *)
  | TSEMICOLON of int (* global line number *)
  | TCOLON | TDOT | TCOMMA | TDOLLAR
  | TOPAR | TCPAR
  (* operators *)
  | TPLUS | TMINUS
  | TMUL | TSLASH | TMOD
  (* for cpp; see also Parse_cpp.token_category *)
  | TSharp
  | EOF
```

⟨*type* `Ast_asm.register` 24b⟩≡ (62b)

```
type register = R of int (* between 0 and 15 on ARM *)
```

3.1.2 ARM assembly tokens

⟨*Parser tokens* 24c⟩≡ (44b)

```
/*(*----------------------------------------*)*/
/*(*2 opcodes *)*/
/*(*----------------------------------------*)*/
```

```
%token <Ast_asm5.arith_opcode> TARITH
%token <Ast_asm5.arithf_opcode * Ast_asm.floatp_precision> TARITHF
%token <Ast_asm.floatp_precision> TCMPF
%token TMVN
%token <Ast_asm.move_size> TMOV TSWAP
%token TB  TBL
%token <Ast_asm5.cmp_opcode> TCMP
%token <Ast_asm5.condition> TBx TCOND
%token TSWI TRFE
/*(* virtual opcodes *)*/
%token TRET TNOP
/*(* pseudo opcodes *)*/
%token TTEXT TGLOBL
%token TDATA TWORD

/*(*----------------------------------------*)*/
/*(*2 registers *)*/
/*(*----------------------------------------*)*/

%token <Ast_asm.register> TRx
%token <Ast_asm.fregister> TFx
%token TR TF
/*(* pseudo registers *)*/
%token TPC TSB TFP TSP

/*(* ARM specific registers *)*/
%token TC
%token <Ast_asm5.creg> TCx

/*(*----------------------------------------*)*/
/*(*2 Constants *)*/
/*(*----------------------------------------*)*/

%token <int> TINT
%token <float> TFLOAT
%token <string> TSTRING

/*(*----------------------------------------*)*/
/*(*2 Names *)*/
/*(*----------------------------------------*)*/
%token <string> TIDENT

/*(*----------------------------------------*)*/
/*(*2 Punctuation *)*/
/*(*----------------------------------------*)*/

/*(* the int is for the (global) line number *)*/
%token <int> TSEMICOLON

%token TCOLON TDOT TDOLLAR
%token TOPAR TCPAR

/*(*----------------------------------------*)*/
/*(*2 Operators *)*/
/*(*----------------------------------------*)*/

%token TSHL TSHR   TSHMINUS TSHAT
%token TPLUS TMINUS TTILDE TMUL TMOD
%token TSLASH
```

```
/*(*----------------------------------------*)*/
/*(*2 Misc *)*/
/*(*----------------------------------------*)*/
%token TSharp
%token EOF
```

3.1.3 Converting tokens

⟨*function* `Parse_asm5.token` 26a⟩≡ (70b)

```
let token (lexbuf : Lexing.lexbuf) : Parser_asm5.token =
  let tok = Lexer_asm.token lexbuf in
  match tok with
  | T.TTEXT -> TTEXT
  | T.TGLOBL -> TGLOBL
  | T.TDATA -> TDATA
  | T.TWORD -> TWORD
  | T.TRET -> TRET
  | T.TNOP -> TNOP
  | T.TR -> TR
  | T.TF -> TF
  | T.TPC -> TPC
  | T.TSB -> TSB
  | T.TFP -> TFP
  | T.TSP -> TSP
  | T.TINT i -> TINT i
  | T.TFLOAT f -> TFLOAT f
  | T.TSTRING s -> TSTRING s
  | T.TSEMICOLON i -> TSEMICOLON i
  | T.TCOLON -> TCOLON
  | T.TDOT -> TDOT
  | T.TCOMMA-> TC
  | T.TDOLLAR-> TDOLLAR
  | T.TOPAR-> TOPAR
  | T.TCPAR-> TCPAR
  | T.TPLUS-> TPLUS
  | T.TMINUS-> TMINUS
  | T.TMUL-> TMUL
  | T.TSLASH-> TSLASH
  | T.TMOD-> TMOD
  | T.TSharp-> TSharp
  | T.EOF-> EOF
  ⟨Parse_asm5.token match tok other cases 40d⟩
```

3.2 Program AST

3.2.1 Location and other basic types

⟨*type* `Ast_asm.loc` 26b⟩≡ (62b)

```
(* (global) line# *)
type loc = Location_cpp.loc
```

⟨*type* `Ast_asm.integer` 26c⟩≡ (62b)

```
type integer = int
```

3.2.2 General assembly instructions

⟨*type* Ast_asm.program 27a⟩≡ (62b)
```
(* The location_history allows to convert a loc to the right place *)
type 'instr program = 'instr lines * Location_cpp.location_history list
```

⟨*type* Ast_asm.lines 27b⟩≡ (62b)
```
type 'instr lines = ('instr line * loc) list
```

⟨*type* Ast_asm.line 27c⟩≡ (62b)
```
type 'instr line =
  | Pseudo of pseudo_instr
  | Virtual of virtual_instr
  | Instr of 'instr

  (* disappear after resolve *)
  | LabelDef of label
  (* less: PragmaLibDirective of string *)
```

3.2.3 Pseudo instructions

⟨*type* Ast_asm.pseudo_instr 27d⟩≡ (62b)
```
type pseudo_instr =
  (* stricter: we allow only SB for TEXT and GLOBL, and no offset *)
  | TEXT of global * attributes * int (* size locals, (multiple of 4 on ARM) *)
  | GLOBL of global (* can have offset? *) * attributes * int (* size *)

  | DATA of global * offset * int (* size, should be in [1..8] *) * ximm
  (* any ximm? even String? And Float? for float should have DWORD? *)
  | WORD of ximm
```

⟨*type* Ast_asm.global 27e⟩≡ (62b)
```
type global = {
  name: symbol;
  ⟨Asm_asm.global other fields 27i⟩
}
```

⟨*type* Ast_asm.symbol 27f⟩≡ (62b)
```
type symbol = string
```

⟨*type* Ast_asm.offset 27g⟩≡ (62b)
```
(* can be 0, negative, or positive *)
type offset = int
```

⟨*type* Ast_asm.ximm 27h⟩≡ (62b)
```
type ximm =
  | Int of integer
  | String of string (* limited to 8 characters *)
  ⟨Ast_asm.ximm other cases 28g⟩
```

⟨Asm_asm.global *other fields* 27i⟩≡ (27e) 28a▷
```
(* 'Some _' when entity is a private symbol (aka static symbol).
 * mutable (ugly?) modifed by linker in naming phase.
 *)
mutable priv: int option;
```

⟨*function* Ast_asm.s_of_global 27j⟩≡ (62b)
```
let s_of_global (x : global) : string =
  x.name ^ (match x.priv with None -> "" | Some _ -> "<>")
```

⟨Asm_asm.global *other fields* 28a⟩+≡ (27e) ◁27i
```
  (* for safe linking (generated only by 5c, not 5a) *)
  signature: int option;
```

3.2.4 Virtual instructions

⟨*type* Ast_asm.virtual_instr 28b⟩≡ (62b)
```
  type virtual_instr =
    | RET
    | NOP (* removed by linker *)
    (* TODO? out MOV here with sizes and sign/unsigned *)
```

3.2.5 Labels and jumps

⟨*type* Ast_asm.label 28c⟩≡ (62b)
```
  (* jmp labels *)
  type label = string
```

⟨*type* Ast_asm.virt_pc 28d⟩≡ (62b)
```
  (* increments by unit of 1 *)
  type virt_pc = int
```

⟨*type* Ast_asm.branch_operand 28e⟩≡ (62b)
```
  type branch_operand = branch_operand2 ref
```

⟨*type* Ast_asm.branch_operand2 28f⟩≡ (62b)
```
  and branch_operand2 =

    (* resolved by assembler *)
    (* relative to PC, in units of virtual_code_address *)
    | Relative of int
    (* we could transform labels in symbols early-on, but nice to resolve ASAP *)
    | LabelUse of label * offset (* useful to have offset? *)

    (* resolved by linker *)
    | SymbolJump of global (* no offset (it would not be used by 5l anyway) *)

    (* after resolution *)
    | Absolute of virt_pc

    (* resolved dynamically by the machine (e.g., B (R14)) *)
    | IndirectJump of register
```

3.2.6 Entities

⟨Ast_asm.ximm *other cases* 28g⟩≡ (27h) 56e▷
```
  (* I used to disallow address of FP or SP, and offset to SB, but
   * 5c needs this feature, so you can take the address of a local.
   * old: Address of global.
   *)
  | Address of entity
```

⟨*type* Ast_asm.entity 28h⟩≡ (62b)
```
  type entity =
    | Param of symbol option * offset (* FP *)
    | Local of symbol option * offset (* SP *)
    (* stricter: we disallow anonymous offsets to SB *)
    | Global of global * offset (* SB *)
```

3.2.7 ARM assembly instructions

⟨*type* Ast_asm5.program 29a⟩≡ (65c)
```
(* On the ARM every instructions can be prefixed with a condition.
 * Note that cond should be AL (Always) for B/Bxx instructions.
*)
type program = instr_with_cond A.program
```

⟨*type* Ast_asm5.instr_with_cond 29b⟩≡ (65c)
```
type instr_with_cond = instr * condition
```

⟨*type* Ast_asm5.condition 29c⟩≡ (65c)
```
  and condition =
    (* equal, not equal *)
    | EQ | NE
    (* greater than, less than, greater or equal, less or equal *)
    | GT of sign | LT of sign | GE of sign | LE of sign
    (* minus/negative, plus/positive *)
    | MI | PL
    (* overflow set/clear *)
    | VS | VC
    (* always/never *)
    | AL | NV
```

⟨*type* Ast_asm.sign 29d⟩≡ (62b)
```
type sign = S (* Signed *) | U (* Unsigned *)
```

⟨*type* Ast_asm5.instr 29e⟩≡ (65c)
```
type instr =
  (* Arithmetic *)
  ⟨Ast_asm5.instr arithmetic instructions cases 30a⟩

  (* Memory *)
  ⟨Ast_asm5.instr memory instructions cases 30e⟩

  (* Control flow *)
  ⟨Ast_asm5.instr control-flow instructions cases 31a⟩

  (* System *)
  ⟨Ast_asm5.instr system instructions cases 31c⟩
```

3.2.8 Special registers

⟨*type* Ast_asm5.reg 29f⟩≡ (65c)
```
type reg = A.register (* between 0 and 15 *)
```

⟨*constant* Ast_asm5.rLINK 29g⟩≡ (65c)
```
let rLINK = R 14
```

⟨*constant* Ast_asm5.rPC 29h⟩≡ (65c)
```
let rPC   = R 15
```

⟨*constant* Ast_asm5.nb_registers 29i⟩≡ (65c)
```
let nb_registers = 16
```

⟨*constant* Ast_asm5.rSB 29j⟩≡ (65c)
```
let rSB  = R 12
```

⟨*constant* Ast_asm5.rSP 29k⟩≡ (65c)
```
let rSP  = R 13
```

3.2.9 Arithmetic instructions

⟨Ast_asm5.instr *arithmetic instructions cases* 30a⟩≡ (29e) 56f▷
```
  | Arith of arith_opcode * arith_cond option *
      arith_operand (* src *) * reg option * reg (* dst *)
```

⟨*type* Ast_asm5.arith_opcode 30b⟩≡ (65c)
```
    and arith_opcode =
      (* logic *)
      | AND | ORR | EOR
      (* arithmetic *)
      | ADD | SUB   | MUL   | DIV | MOD (* DIV and MOD are virtual *)
      (* bit shifting; immediate operand can only be between 0 and 31 *)
      | SLL | SRL | SRA (* virtual, sugar for bitshift register *)
      (* less useful *)
      | BIC  | ADC | SBC  | RSB | RSC
      (* middle operand always empty (could lift up and put special type) *)
      | MOV | MVN (* MOV has no reading syntax in 5a, MOVE is used *)
```

⟨*type* Ast_asm5.arith_operand 30c⟩≡ (65c)
```
  type arith_operand =
    | Imm of A.integer (* characters are converted to integers *)
    | Reg of reg
    (* can not be used with shift opcodes (SLL/SRL/SRA) *)
    | Shift of reg * shift_reg_op *
               (reg, int (* between 0 and 31 *)) Either_.t
```

⟨*type* Ast_asm5.shift_reg_op 30d⟩≡ (65c)
```
    and shift_reg_op =
      | Sh_logic_left | Sh_logic_right
      | Sh_arith_right | Sh_rotate_right
```

3.2.10 Memory instructions

⟨Ast_asm5.instr *memory instructions cases* 30e⟩≡ (29e)
```
  | MOVE of A.move_size * move_option *
      mov_operand (* src *) * mov_operand (* dst *) (* virtual *)
  | SWAP of A.move_size (* actually only (Byte x) *) *
       reg (* indirect *) * reg * reg option
```

⟨*type* Ast_asm.move_size 30f⟩≡ (62b)
```
  type move_size = Word | HalfWord of sign | Byte of sign
```

⟨*type* Ast_asm5.mov_operand 30g⟩≡ (65c)
```
  type mov_operand =
    (* Immediate shift register *)
    | Imsr of arith_operand
    (* eXtended immediate.
     * (Ximm (Int x) is converted in Imsr (Imm x) in the parser
     *)
    | Ximm of ximm

    | Indirect of reg * A.offset
    (* another form of Indirect *)
    | Entity of A.entity
```

⟨*type* Ast_asm5.move_option 30h⟩≡ (65c)
```
     and move_option = move_cond option
```

3.2.11 Branching instructions

⟨Ast_asm5.instr *control-flow instructions cases* 31a⟩≡ (29e) 57c ▷
```
  | B  of A.branch_operand (* branch *)
  | BL of A.branch_operand (* branch and link *)
  | Cmp of cmp_opcode * arith_operand * reg
  (* just Relative or LabelUse here for branch_operand *)
  | Bxx of condition * A.branch_operand (* virtual, sugar for B.XX *)
```

⟨*type* Ast_asm5.cmp_opcode 31b⟩≡ (65c)
```
    and cmp_opcode =
      | CMP
      (* less useful *)
      | TST | TEQ | CMN
```

3.2.12 System instructions

⟨Ast_asm5.instr *system instructions cases* 31c⟩≡ (29e)
```
  | SWI of int (* value actually unused in Plan 9 and Linux *)
  | RFE (* virtual, sugar for MOVM *)
```

3.3 Object file

⟨*type* Object_file.t 31d⟩≡ (67)
```
  (* An object file (.o) in Plan 9 is really just a serialized assembly AST *)
  type 'instr t = {
    prog:  'instr Ast_asm.program;
    arch: Arch.t
  }
```

Chapter 4

Main Functions

4.1 main()

⟨*toplevel* Main._1 32a⟩≡ (69a)
```
let _ =
  Cap.main (fun (caps : Cap.all_caps) ->
     let argv = CapSys.argv caps in
     Exit.exit caps
        (Exit.catch (fun () ->
            CLI.main caps argv))
  )
```

⟨*type* CLI.caps 32b⟩≡ (68)
```
(* Need:
 * - open_in: for argv derived input file but also for #include'd files
 *   because 5a/va/... are macroassemblers
 * - open_out for -o object file or argv[0].5
 * - env: for INCLUDE (for cpp)
 *)
type caps = < Cap.open_in; Cap.open_out; Cap.env >
```

⟨*signature* CLI.main 32c⟩≡ (68a)
```
(* entry point (can also raise Exit.ExitCode) *)
val main: <caps; ..> ->
  string array -> Exit.t
```

⟨*function* CLI.main 32d⟩≡ (68c)
```
let main (caps: <caps; ..>) (argv: string array) : Exit.t =
  let arch =
    (* alt: use Arch.arch_of_char argv.(0).(1) *)
    match Filename.basename argv.(0) with
    | "o5a" -> Arch.Arm
    | "ova" -> Arch.Mips
    | s -> failwith (spf "arch could not be detected from argv0 %s" s)
  in

  let thechar = Arch.thechar arch in
  let thestring = Arch.thestring arch in

  let usage =
    spf "usage: %s [-options] file.s" argv.(0)
  in

  (* alt: Fpath.t option ref *)
  let infile  = ref "" in
```

```
let outfile = ref "" in

let level = ref (Some Logs.Warning) in
(* for debugging *)
let backtrace = ref false in

(* for cpp *)
let include_paths : Fpath.t list ref = ref [] in
let macro_defs = ref [] in

let options = [
  "-o", Arg.Set_string outfile,
  " <file> output file";

  (* dup: same in compiler/CLI.ml *)
  "-D", Arg.String (fun s ->
    let (var, val_) =
      if s =~ "\\(.*\\)=\\(.*\\)"
      then Regexp_.matched2 s
      else (s, "1")
    in
    macro_defs := (var, val_)::!macro_defs
  ), " <name=def> (or just <name>) define the name to the preprocessor";
  "-I", Arg.String (fun s ->
    include_paths := Fpath.v s::!include_paths
  ), " <dir> add dir as a path to look for '#include <file>' files";

  "-v", Arg.Unit (fun () -> level := Some Logs.Info),
   " verbose mode";
  "-verbose", Arg.Unit (fun () -> level := Some Logs.Info),
  " verbose mode";
  "-debug", Arg.Unit (fun () -> level := Some Logs.Debug),
  " guess what";
  "-quiet", Arg.Unit (fun () -> level := None),
  " ";

  (* pad: I added that *)
  "-dump_ast", Arg.Set dump_ast,
  " dump the parsed AST";
  (* pad: I added that *)
  "-backtrace", Arg.Set backtrace,
  " dump the backtrace after an error";
] |> Arg.align
in
(try
  Arg.parse_argv argv options (fun f ->
   if !infile <> ""
   then failwith "already specified an input file";
   infile := f;
 ) usage;
with
| Arg.Bad msg -> UConsole.eprint msg; raise (Exit.ExitCode 2)
| Arg.Help msg -> UConsole.print msg; raise (Exit.ExitCode 0)
);
Logs_.setup !level ();
Logs.info (fun m -> m "assembler ran from %s with arch %s"
      (Sys.getcwd()) thestring);

if !infile = ""
then begin Arg.usage options usage; raise (Exit.ExitCode 1); end;
```

```
    let outfile : Fpath.t =
      (if !outfile = ""
      then
        let b = Filename.basename !infile in
        if b =~ "\\(.*\\)\\.s"
        then Regexp_.matched1 b ^ (spf ".o%c" thechar)
        else (b ^ (spf ".o%c" thechar))
      else !outfile
      ) |> Fpath.v
    in

    (* dup: same in compiler/CLI.ml *)
    let system_paths : Fpath.t list =
      (try CapSys.getenv caps "INCLUDE" |> Str.split (Str.regexp "[ \t]+")
       with Not_found ->
         [spf "/%s/include" thestring; "/sys/include";]
      ) |> Fpath_.of_strings
    in
    let conf = Preprocessor.{
      defs = !macro_defs;
      (* this order? *)
      paths = system_paths @ List.rev !include_paths;
      dir_source_file = Fpath.v (Filename.dirname !infile);
    }
    in

    try
      (* main call *)
      (* can't create chan from infile to avoid passing Cap.open_in because
       * with cpp we need to open other files.
       *)
      outfile |> FS.with_open_out caps (fun chan ->
          assemble caps conf arch (Fpath.v !infile) chan
      );
      Exit.OK
    with exn ->
      if !backtrace
      then raise exn
      else
        (match exn with
        | Location_cpp.Error (s, loc) ->
            (* less: could use final_loc_and_includers_of_loc loc *)
            let (file, line) = Location_cpp.final_loc_of_loc loc in
            Logs.err (fun m -> m "%s:%d %s" !!file line s);
            Exit.Code 1
        | _ -> raise exn
        )
```

⟨*signature* CLI.assemble 34a⟩≡ (68a)

```
  (* main algorithm; works by side effect on outfile *)
  val assemble: <Cap.open_in; .. > ->
    Preprocessor.conf -> Arch.t -> Fpath.t (* infile *) -> Chan.o (* outfile *) ->
    unit
```

4.2 assemble()

⟨*function* CLI.assemble 34b⟩≡ (68c)

```
  (* Will modify chan as a side effect *)
  let assemble (caps: < Cap.open_in; .. >) (conf : Preprocessor.conf) (arch: Arch.t) (infile : Fpath.t) (chan : C
    match arch with
    | Arch.Arm ->
        let prog = assemble5 caps conf infile in
        Object_file.save arch prog chan
    | Arch.Mips ->
        let prog = assemblev caps conf infile in
        Object_file.save arch prog chan
    | _ ->
     failwith (spf "TODO: arch not supported yet: %s" (Arch.thestring arch))
```

⟨*signature* Object_file.save 35a⟩≡ (67a)
```
  val save: Arch.t -> 'instr Ast_asm.program -> Chan.o (* obj file *) -> unit
```

⟨*function* CLI.assemble5 35b⟩≡ (68c)
```
  let assemble5 (caps: < Cap.open_in; .. >) (conf : Preprocessor.conf) (infile : Fpath.t) : Ast_asm5.program =
    let prog = Parse_asm5.parse caps conf infile in
    let prog = Resolve_labels.resolve Ast_asm5.branch_opd_of_instr prog in
    if !dump_ast
    then prog |> Meta_ast_asm5.vof_program |> OCaml.string_of_v |> (fun s ->
          Logs.app (fun m -> m "AST = %s" s));
    prog
```

⟨*signature* Parse_asm5.parse 35c⟩≡ (70a)
```
  (* will preprocess the code internally first *)
  val parse:
    < Cap.open_in; .. > -> Preprocessor.conf -> Fpath.t -> Ast_asm5.program
```

⟨*signature* Resolve_labels.resolve 35d⟩≡ (71a)
```
  (* The final program has neither labels (defs and uses) nor
   * relative jumps in branching instructions.
   * Those are converted in absolute jumps.
   * !!Actually works by side effect on input program so take care!!
   *)
  val resolve:
    ('instr -> Ast_asm.branch_operand option) ->
    'instr Ast_asm.program -> 'instr Ast_asm.program
```

4.3 parse()

⟨*signature* Parse_asm5.parse_no_cpp 35e⟩≡ (70a)
```
  val parse_no_cpp: Chan.i -> Ast_asm5.program
```

⟨*function* Parse_asm5.parse_no_cpp 35f⟩≡ (70b)
```
  (* Simpler code path; possibly useful in tests *)
  let parse_no_cpp (chan : Chan.i) : Ast_asm5.program =
    L.line := 1;
    let lexbuf = Lexing.from_channel chan.ic in
    try
      Parser_asm5.program token lexbuf, []
    with Parsing.Parse_error ->
        failwith (spf "Syntax error: line %d" !L.line)
```

⟨*function* Parse_asm5.parse 36⟩≡ (70b)

```
let parse (caps : < Cap.open_in; .. >) (conf : Preprocessor.conf) (file : Fpath.t) : Ast_asm5.program =
  let hooks = Parse_cpp.{
     lexer = token;
     parser = Parser_asm5.program;
     category = (fun t ->
       match t with
       | Parser_asm5.EOF -> Parse_cpp.Eof
       | Parser_asm5.TSharp -> Parse_cpp.Sharp
       | Parser_asm5.TIDENT s -> Parse_cpp.Ident s
        (* stricter: I forbid to have macros overwrite keywords *)
       | _ -> Parse_cpp.Other
     );
     eof = Parser_asm5.EOF;
  }
  in
  Parse_cpp.parse caps hooks conf file
```

Chapter 5

Lexing

5.1 Overview

⟨Lexer_asm.mll 37⟩≡
```
{
(* Copyright 2015, 2016, 2025 Yoann Padioleau, see copyright.txt *)
open Common

open Token_asm
module L = Location_cpp

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* Common parts of the different Plan 9 assembly lexers.
 *
 * Limitations compared to 5a/va/...:
 *  - no unicode support
 *  - no uU lL suffix
 *    (but was skipped by 5a anyway)
 *  - no '=' used for constant definition
 *    (but can use cpp #define for that)
 *)

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)
⟨function Lexer_asm.error 38b⟩
⟨function Lexer_asm.code_of_escape_char 43a⟩
⟨function Lexer_asm.string_of_ascii 43d⟩
⟨function Lexer_asm.char_ 39a⟩

(*****************************************************************************)
(* Regexps aliases *)
(*****************************************************************************)
⟨constant Lexer_asm.letter 40a⟩
⟨constant Lexer_asm.space 39b⟩
⟨constant Lexer_asm.digit 40b⟩
⟨constant Lexer_asm.hex 42b⟩
⟨constant Lexer_asm.oct 42d⟩

(*****************************************************************************)
(* Main rule *)
(*****************************************************************************)
⟨rule Lexer_asm.token 38a⟩
```

```
  (*****************************************************************************)
  (* Rule char *)
  (*****************************************************************************)
  ⟨rule Lexer_asm.char 42h⟩

  (*****************************************************************************)
  (* Rule string *)
  (*****************************************************************************)
  ⟨rule Lexer_asm.string 43c⟩

  (*****************************************************************************)
  (* Rule comment *)
  (*****************************************************************************)
  ⟨rule Lexer_asm.comment 39f⟩
```

⟨rule Lexer_asm.token 38a⟩≡ (37)

```
  rule token = parse
    (* ---------------------------------------------------------------------- *)
    (* Spacing/comments *)
    (* ---------------------------------------------------------------------- *)
    ⟨Lexer_asm.token space/comment cases 39c⟩

    (* ---------------------------------------------------------------------- *)
    (* Symbols *)
    (* ---------------------------------------------------------------------- *)
    ⟨Lexer_asm.token symbol cases 39h⟩

    (* ---------------------------------------------------------------------- *)
    (* Mnemonics and identifiers *)
    (* ---------------------------------------------------------------------- *)
    ⟨Lexer_asm.token mnemonics/identifiers cases 40c⟩

    (* ---------------------------------------------------------------------- *)
    (* Numbers *)
    (* ---------------------------------------------------------------------- *)
    ⟨Lexer_asm.token numbers cases 41b⟩

    (* ---------------------------------------------------------------------- *)
    (* Chars/Strings *)
    (* ---------------------------------------------------------------------- *)
    ⟨Lexer_asm.token chars/strings cases 42g⟩

    (* ---------------------------------------------------------------------- *)
    (* CPP *)
    (* ---------------------------------------------------------------------- *)
    ⟨Lexer_asm.token cpp cases 38c⟩

    (* ---------------------------------------------------------------------- *)
    (* less: maybe return a fake semicolon the first time? *)
    | eof { EOF }
    | _ (*as c*)   { let c = char_ lexbuf in
                     error (spf "unrecognized character: '%c'" c) }
```

⟨function Lexer_asm.error 38b⟩≡ (37)

```
  let error s =
    raise (L.Error (spf "Lexical error: %s" s, !L.line))
```

⟨Lexer_asm.token cpp cases 38c⟩≡ (38a)

```
  (* See ../macroprocessor/Lexer_cpp.mll (called from Parse_asmX.ml) *)
  | "#" { TSharp }
```

⟨*function* `Lexer_asm.char_` 39a⟩≡ (37)
```
(* needed only because of ocamllex limitations in ocaml-light
 * which does not support the 'as' feature.
 *)
let char_ lexbuf =
  let s = Lexing.lexeme lexbuf in
  String.get s 0
}
```

5.2 Spaces and comments

⟨*constant* `Lexer_asm.space` 39b⟩≡ (37)
```
let space = [' ''\t']
```

⟨`Lexer_asm.token` *space/comment cases* 39c⟩≡ (38a) 39d▷
```
| space+ { token lexbuf }
```

⟨`Lexer_asm.token` *space/comment cases* 39d⟩+≡ (38a) ◁39c 39e▷
```
| "//" [^'\n']* { token lexbuf }
```

⟨`Lexer_asm.token` *space/comment cases* 39e⟩+≡ (38a) ◁39d 39g▷
```
| "/*"          { comment lexbuf }
```

⟨*rule* `Lexer_asm.comment` 39f⟩≡ (37)
```
and comment = parse
  | "*/"          { token lexbuf }
  | [^ '*' '\n']+ { comment lexbuf }
  | '*'           { comment lexbuf }
  | '\n'          { incr L.line; comment lexbuf }
  | eof           { error "end of file in comment" }
```

5.3 Newlines and semicolons

⟨`Lexer_asm.token` *space/comment cases* 39g⟩+≡ (38a) ◁39e
```
(* newlines are converted in fake semicolons for the grammar *)
| '\n' { let old = !L.line in incr L.line; TSEMICOLON old }
```

⟨`Lexer_asm.token` *symbol cases* 39h⟩≡ (38a) 39i▷
```
| ';' { TSEMICOLON !L.line }
```

5.4 Punctuations and operators

⟨`Lexer_asm.token` *symbol cases* 39i⟩+≡ (38a) ◁39h
```
| ':' { TCOLON } | ',' { TCOMMA }
| '(' { TOPAR } | ')' { TCPAR }
| '$' { TDOLLAR }

| '+' { TPLUS } | '-' { TMINUS }
(* '/' is used for division and for DATA too *)
| '*' { TMUL }  | '/' { TSLASH } | '%' { TMOD }

(* has to be before the rule for identifiers *)
| '.' { TDOT }
```

5.5 Mnemonics and labels

⟨*constant* `Lexer_asm.letter` 40a⟩≡ (37)
```
let letter = ['a'-'z''A'-'Z']
```

⟨*constant* `Lexer_asm.digit` 40b⟩≡ (37)
```
let digit = ['0'-'9']
```

⟨`Lexer_asm.token` *mnemonics/identifiers cases* 40c⟩≡ (38a) 40e▷
```
| "R" (digit+ (*as s*))
    { let s = Lexing.lexeme lexbuf |> String_.drop_prefix 1 in
      let i = int_of_string s in
      (* the range check is arch-specific and must be done in Parse_asmX.ml *)
      TRx (Ast_asm.R i)
    }
| "F" (digit+ (*as s*))
    { let s = Lexing.lexeme lexbuf |> String_.drop_prefix 1 in
      let i = int_of_string s in
      (* the range check is arch-specific and must be done in Parse_asmX.ml *)
      TFx (Ast_asm.FR i)
    }
```

⟨`Parse_asm5.token` *match* `tok` *other cases* 40d⟩≡ (26a) 41a▷
```
| T.TRx ((A.R i) as x) ->
    if i <= 15 && i >=0
    then TRx x
    else Lexer_asm.error ("register number not valid")
| T.TFx ((A.FR i) as x) ->
    if i <= 15 && i >=0
    then TFx x
    else Lexer_asm.error ("register number not valid")
```

⟨`Lexer_asm.token` *mnemonics/identifiers cases* 40e⟩+≡ (38a) ◁40c
```
(* looser: actually for '.' 5a imposes to have an isalpha() after *)
| (letter | '_' | '@' | '.') (letter | digit | '_' | '$' )* {
    let s = Lexing.lexeme lexbuf in
    (* alt: use Hashtbl.t *)
    match s with
    (* pseudo instructions *)
    | "TEXT" -> TTEXT | "GLOBL" -> TGLOBL
    | "WORD" -> TWORD | "DATA" -> TDATA

    (* virtual instructions *)
    | "RET" -> TRET
    | "NOP" -> TNOP

    (* registers (see also the special rule above for R digit+) *)
    | "R" -> TR
    | "F" -> TF

    (* pseudo registers *)
    | "PC" -> TPC | "SB" -> TSB | "SP" -> TSP | "FP" -> TFP

    (* each Parse_asmX.ml will refine and convert some TIDENT *)
    | _ -> TIDENT s
  }
```

⟨Parse_asm5.token *match* tok *other cases* 41a⟩+≡ (26a) ◁40d

```
  | T.TIDENT s ->
      (match s with
      (* instructions *)
      | "AND" -> TARITH AND | "ORR" -> TARITH ORR | "EOR" -> TARITH EOR

      | "ADD" -> TARITH ADD | "SUB" -> TARITH SUB
      | "MUL" -> TARITH MUL | "DIV" -> TARITH DIV | "MOD" -> TARITH MOD
      | "SLL" -> TARITH SLL | "SRL" -> TARITH SRL | "SRA" -> TARITH SRA

      | "BIC" -> TARITH BIC
      | "ADC" -> TARITH ADC | "SBC" -> TARITH SBC
      | "RSB" -> TARITH RSB | "RSC" -> TARITH RSC

      | "MVN" -> TMVN

      (* could move to Lexer_asm.mll and Ast_asm.virtual_instr *)
      | "MOVW" -> TMOV A.Word
      | "MOVB" -> TMOV (A.Byte     A.S) | "MOVBU" -> TMOV (A.Byte     A.U)
      | "MOVH" -> TMOV (A.HalfWord A.S) | "MOVHU" -> TMOV (A.HalfWord A.U)

      | "B" -> TB | "BL" -> TBL
      | "CMP" -> TCMP CMP
      | "TST" -> TCMP TST | "TEQ" -> TCMP TEQ | "CMN" -> TCMP CMN

      | "BEQ" -> TBx EQ | "BNE" -> TBx NE
      | "BGT" -> TBx (GT A.S) | "BLT" -> TBx (LT A.S)
      | "BGE" -> TBx (GE A.S) | "BLE" -> TBx (LE A.S)
      | "BHI" -> TBx (GT A.U) | "BLO" -> TBx (LT A.U)
      | "BHS" -> TBx (GE A.U) | "BLS" -> TBx (LE A.U)
      | "BMI" -> TBx MI | "BPL" -> TBx PL
      | "BVS" -> TBx VS | "BVC" -> TBx VC

      | "SWI" -> TSWI
      | "RFE" -> TRFE

      (* conditions *)
      | ".EQ" -> TCOND EQ | ".NE" -> TCOND NE
      | ".GT" -> TCOND (GT A.S)   | ".LT" -> TCOND (LT A.S)
      | ".GE" -> TCOND (GE A.S)   | ".LE" -> TCOND (LE A.S)
      | ".HI" -> TCOND (GT A.U) | ".LO" -> TCOND (LT A.U)
      | ".HS" -> TCOND (GE A.U) | ".LS" -> TCOND (LE A.U)
      | ".MI" -> TCOND MI | ".PL" -> TCOND PL
      | ".VS" -> TCOND VS | ".VC" -> TCOND VC

      ⟨Parse_asm5.token in TIDENT case, other cases 56a⟩

      | _ -> TIDENT s
      )
```

5.6 Numbers

⟨Lexer_asm.token *numbers cases* 41b⟩≡ (38a)

```
  ⟨Lexer_asm.token octal case 42e⟩
  ⟨Lexer_asm.token hexadecimal case 42c⟩
  ⟨Lexer_asm.token decimal case 42a⟩
  ⟨Lexer_asm.token float case 42f⟩
```

5.6.1 Decimal numbers

⟨Lexer_asm.token *decimal case* 42a⟩≡ (41b)
```
  | digit+           { TINT (int_of_string (Lexing.lexeme lexbuf)) }
```

5.6.2 Hexadecimal and octal numbers

⟨*constant* Lexer_asm.hex 42b⟩≡ (37)
```
  let hex = (digit | ['A'-'F''a'-'f'])
```

⟨Lexer_asm.token *hexadecimal case* 42c⟩≡ (41b)
```
  | "0x" hex+        { TINT (int_of_string (Lexing.lexeme lexbuf)) }
```

⟨*constant* Lexer_asm.oct 42d⟩≡ (37)
```
  let oct = ['0'-'7']
```

⟨Lexer_asm.token *octal case* 42e⟩≡ (41b)
```
  | "0"  (oct+ (*as s*))
      { let s = Lexing.lexeme lexbuf |> String_.drop_prefix 1 in
        TINT (int_of_string ("0o" ^ s))  }
```

5.6.3 Floating-point numbers

⟨Lexer_asm.token *float case* 42f⟩≡ (41b)
```
  (* stricter: I impose some digit+ after '.' and after 'e' *)
  | (digit+ | digit* '.' digit+) (['e''E'] ('+' | '-')? digit+)?
     { TFLOAT (float_of_string (Lexing.lexeme lexbuf)) }
```

5.7 Characters

⟨Lexer_asm.token *chars/strings cases* 42g⟩≡ (38a) 43b ▷
```
  | "'" { TINT (char lexbuf) }
```

⟨*rule* Lexer_asm.char 42h⟩≡ (37)
```
  and char = parse
    | "'''                          { Char.code '\'' }
    | "\\" ((oct oct? oct?) (*as s*)) "'"
        { let s = Lexing.lexeme lexbuf |>
             String_.drop_prefix 1 |> String_.drop_suffix 1 in
          int_of_string ("0o" ^ s) }
    | "\\" (['a'-'z'] (*as c*)) "'"
       { let c = String.get (Lexing.lexeme lexbuf) 1 in
         code_of_escape_char c }
    | [^ '\\' '\'' '\n'] (*as c*)  "'"
        { let c = String.get (Lexing.lexeme lexbuf) 0 in
          Char.code c }
    | '\n' { error "newline in character" }
    | eof  { error "end of file in character" }
    | _    { error "missing '" }
```

5.7.1 Escaped sequences

⟨*function* `Lexer_asm.code_of_escape_char` 43a⟩≡ (37)
```
let code_of_escape_char c =
  match c with
  | 'n' -> Char.code '\n' | 'r' -> Char.code '\r'
  | 't' -> Char.code '\t' | 'b' -> Char.code '\b'

  | 'f' -> error "unknown \\f"
  (* could be removed, special 5a escape char *)
  | 'a' -> 0x07 | 'v' -> 0x0b
  (* useful to generate C-compatible strings with special end marker *)
  | 'z' -> 0

  (* stricter: we disallow \ with unknown character *)
  | _ -> error "unknown escape sequence"
```

5.7.2 Triple quotes

5.7.3 Escaped newlines

5.8 Strings

⟨`Lexer_asm.token` *chars/strings cases* 43b⟩+≡ (38a) ◁42g
```
| '"'
    { let s = string lexbuf in
      (* less: why this limit though? *)
      if String.length s > 8
      then error ("string constant too long")
      else TSTRING s
    }
```

⟨*rule* `Lexer_asm.string` 43c⟩≡ (37)
```
and string = parse
  | '"' { "" }
  | "\\" ((oct oct oct) (*as s*))
      { let s = Lexing.lexeme lexbuf |> String_.drop_prefix 1 in
        let i = int_of_string ("0o" ^ s) in string_of_ascii i ^ string lexbuf }
  | "\\" (['a'-'z'] (*as c*))
      { let c = String.get (Lexing.lexeme lexbuf) 1 in
        let i = code_of_escape_char c in string_of_ascii i ^ string lexbuf  }
  | [^ '\\' '"' '\n']+
      { let x = Lexing.lexeme lexbuf in x ^ string lexbuf }
  | '\n' { error "newline in string" }
  | eof  { error "end of file in string" }
  | _    { error "undefined character in string" }
```

⟨*function* `Lexer_asm.string_of_ascii` 43d⟩≡ (37)
```
let string_of_ascii i =
  String.make 1 (Char.chr i)
```

Chapter 6

Parsing

6.1 Overview

⟨*function* `Parser_asm.error` 44a⟩≡ (69b)
```
let error s =
  raise (L.Error (spf "Syntax error: %s" s, !L.line))
```

6.2 Grammar overview

⟨`Parser_asm5.mly` 44b⟩≡
```
%{
(* Copyright 2015, 2016, 2025 Yoann Padioleau, see copyright.txt *)
open Common
open Either

open Ast_asm
open Parser_asm
open Ast_asm5
module L = Location_cpp

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* The 5a ARM assembly grammar.
 *
 * Limitations compared to 5a:
 *  - just imm for SWI
 *
 * todo:
 *  - special bits
 *  - lots of advanced instructions (float, mulm, ...)
 *)

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)
(* See Parser_asm.ml *)

%}

/*(*************************************************************************)*/
/*(*1 Tokens *)*/
/*(*************************************************************************)*/
```

⟨*Parser tokens* 24c⟩

```
/*(*************************************************************************)*/
/*(*1 Priorities *)*/
/*(*************************************************************************)*/
```
⟨*Parser tokens priorities* 45a⟩

```
/*(*************************************************************************)*/
/*(*1 Rules type declaration *)*/
/*(*************************************************************************)*/
```
⟨*Parser type declarations* 46⟩
```
%start program

%%
```
⟨*Parser grammar* 45b⟩

⟨*Parser tokens priorities* 45a⟩≡ (44b)
```
%left TOR
%left TXOR
%left TAND
%left TLT TGT
%left TPLUS TMINUS
%left TMUL TSLASH TMOD
```

⟨*Parser grammar* 45b⟩≡ (44b)
```
/*(*************************************************************************)*/
/*(*1 Program (arch independent) *)*/
/*(*************************************************************************)*/
```

⟨*grammar rule* `program` 47a⟩
⟨*grammar rule* `lines` 47b⟩
⟨*grammar rule* `line` 47c⟩

⟨*grammar rule* `label_def` 48b⟩

```
/*(*************************************************************************)*/
/*(*1 Pseudo instructions (arch independent) *)*/
/*(*************************************************************************)*/
/*(* I can't factorize in attr_opt; shift/reduce conflict with TC *)*/
```
⟨*grammar rule* `pseudo_instr` 50e⟩

```
/*(* stricter: I introduced those intermediate rules *)*/
```
⟨*grammar rule* `global` 50f⟩
⟨*grammar rule* `global_and_offset` 51a⟩

```
/*(*************************************************************************)*/
/*(*1 Virtual instructions (arch independent) *)*/
/*(*************************************************************************)*/
```

⟨*grammar rule* `virtual_instr` 47f⟩

```
/*(*************************************************************************)*/
/*(*1 Instructions, arch specific!! *)*/
/*(*************************************************************************)*/
```

⟨*grammar rule* `instr` 47d⟩

```
/*(*************************************************************************)*/
/*(*1 Operands *)*/
/*(*************************************************************************)*/
```

⟨*grammar rule* `imsr` 48c⟩

⟨*grammar rule* `imm` 48e⟩
⟨*grammar rule* `reg` 48d⟩
⟨*grammar rule* `shift` 48h⟩

⟨*grammar rule* `rcon` 49a⟩

⟨*grammar rule* `gen` 49b⟩

⟨*grammar rule* `ximm` 49c⟩

⟨*grammar rule* `ioreg` 49d⟩
⟨*grammar rule* `ireg` 49e⟩

⟨*grammar rule* `branch` 50b⟩
⟨*grammar rule* `rel` 50c⟩

```
/*(*----------------------------------------*)*/
/*(*2 name and offset (arch independent)  *)*/
/*(*----------------------------------------*)*/
```

⟨*grammar rule* `name` 49f⟩
⟨*grammar rule* `pointer` 49h⟩

⟨*grammar rule* `offset` 50a⟩

```
/*(*----------------------------------------*)*/
/*(*2 float *)*/
/*(*----------------------------------------*)*/
```

⟨*grammar rule* `freg` 57g⟩
⟨*grammar rule* `frcon` 57h⟩

```
/*(*----------------------------------------*)*/
/*(*2 number constants and expressions (arch independent)  *)*/
/*(*----------------------------------------*)*/
```

⟨*grammar rule* `con` 48f⟩

⟨*grammar rule* `fcon` 57i⟩

⟨*grammar rule* `expr` 56c⟩

```
/*(*************************************************************************)*/
/*(*1 Misc *)*/
/*(*************************************************************************)*/
```

```
/*(* todo: special bits or inline in previous rule? *)*/
```
⟨*grammar rule* `cond` 51b⟩

⟨*Parser type declarations* 46⟩≡ (44b)
```
%type <Ast_asm5.instr_with_cond Ast_asm.lines> program
```

6.3 Program and lines

⟨*grammar rule* `program` 47a⟩≡ (45b)
```
program: lines EOF { $1 }
```

⟨*grammar rule* `lines` 47b⟩≡ (45b)
```
lines:
 | /*empty*/  { [] }
 | line lines { $1 @ $2 }
```

⟨*grammar rule* `line` 47c⟩≡ (45b)
```
line:
 |              TSEMICOLON { [] }
 | instr        TSEMICOLON { [(Instr (fst $1, snd $1), $2)] }
 ⟨line other cases 47e⟩
```

6.4 Instructions

⟨*grammar rule* `instr` 47d⟩≡ (45b)
```
instr:
 | TARITH cond  imsr TC reg TC reg
     { (Arith ($1, None, $3, Some $5, $7), $2) }
 | TARITH cond  imsr TC reg  { (Arith ($1,  None, $3, None, $5), $2) }
 | TMVN   cond  imsr TC reg  { (Arith (MVN, None, $3, None, $5), $2) }

 | TARITHF cond frcon         TC freg  { (ArithF ($1, $3, None, $5), $2) }
 | TARITHF cond frcon TC freg TC freg  { (ArithF ($1, $3, Some $5, $7), $2) }
 | TCMPF cond freg TC freg             { (CmpF ($1, $3, $5), $2) }

 | TMOV   cond  gen  TC gen     { (MOVE ($1, None, $3, $5), $2) }

 | TSWAP  cond  reg  TC ireg    { (SWAP ($1, $5, $3, None), $2) }
 | TSWAP  cond  ireg TC reg     { (SWAP ($1, $3, $5, None), $2) }
 | TSWAP  cond  reg  TC ireg TC reg
     { (SWAP ($1, $5, $3, Some $7), $2) }

 /*(*stricter: no cond here, use Bxx form, so normalized AST *)*/
 | TB        branch           { (B $2, AL) }
 | TBx       rel              { (Bxx ($1, $2), AL) }
 | TBL  cond branch           { (BL $3, $2)}
 | TCMP cond imsr TC reg  { (Cmp ($1, $3, $5), $2) }

 | TSWI cond imm { (SWI $3, $2) }
 | TRFE cond     { (RFE, $2) }
```

6.4.1 Arithmetic and logic

6.4.2 Memory

6.4.3 Control flow

⟨`line` *other cases* 47e⟩≡ (47c) 48a▷
```
 | virtual_instr TSEMICOLON { [(Virtual $1, $2)] }
```

⟨*grammar rule* `virtual_instr` 47f⟩≡ (45b)
```
virtual_instr:
 /*(* was in instr before. stricter: no cond (nor comma) *)*/
 | TRET                  { RET }
```

6.4.4 Software interrupt

6.5 Label definitions

⟨line *other cases* 48a⟩+≡ (47c) ◁47e 50d▷
```
  | label_def line           { $1::$2 }
```

⟨*grammar rule* label_def 48b⟩≡ (45b)
```
  label_def: TIDENT TCOLON    { (LabelDef $1, !L.line) }
```

6.6 Operands

⟨*grammar rule* imsr 48c⟩≡ (45b)
```
  imsr:
   | imm   { Imm $1 }
   | shift { $1 }
   | reg   { Reg $1 }
```

6.6.1 Registers

⟨*grammar rule* reg 48d⟩≡ (45b)
```
  reg:
   | TRx                { $1 }
   /*(* stricter? could remove, redundant with cpp *)*/
   | TR TOPAR expr TCPAR
       { if $3 <= 15 && $3 >= 0
         then R $3
         else error "register value out of range"
       }
```

6.6.2 Immediate constants

⟨*grammar rule* imm 48e⟩≡ (45b)
```
  imm: TDOLLAR con      { $2 }
```

⟨*grammar rule* con 48f⟩≡ (45b)
```
  con:
   | TINT { $1 }
   ⟨con other cases 48g⟩
```

⟨con *other cases* 48g⟩≡ (48f) 56b▷
```
  | TMINUS con { - $2 }
  | TPLUS  con { $2 }
  | TTILDE con { lnot $2 }
```

6.6.3 ARM shifted registers

⟨*grammar rule* shift 48h⟩≡ (45b)
```
  /*(* ARM specific *)*/
  shift:
   | reg TSHL rcon     { Shift ($1, Sh_logic_left, $3)  }
   | reg TSHR rcon     { Shift ($1, Sh_logic_right, $3)  }
   | reg TSHMINUS rcon { Shift ($1, Sh_arith_right, $3)  }
   | reg TSHAT rcon    { Shift ($1, Sh_rotate_right, $3)  }
```

⟨*grammar rule* rcon 49a⟩≡ (45b)
```
rcon:
 | reg { Left $1 }
 | con { if ($1 >= 0 && $1 <= 31)
         then Right $1
         else error "shift value out of range"
       }
```

6.6.4 Memory (de)references, pointers

⟨*grammar rule* gen 49b⟩≡ (45b)
```
gen:
 | ximm  { match $1 with Int x -> Imsr (Imm x) | x -> Ximm x }
 | shift { Imsr ($1) }
 | reg   { Imsr (Reg $1) }

 | ioreg { $1 }
 | name                    { Entity $1 }
 | con TOPAR pointer TCPAR { Entity ($3 None $1) }
```

⟨*grammar rule* ximm 49c⟩≡ (45b)
```
ximm:
 | imm             { Int $1 }
 | fcon            { Float $1 }
 | TDOLLAR TSTRING { String $2 }
 | TDOLLAR name    { Address $2 }
```

⟨*grammar rule* ioreg 49d⟩≡ (45b)
```
ioreg:
 | ireg     { Indirect ($1, 0) }
 | con ireg { Indirect ($2, $1) }
```

⟨*grammar rule* ireg 49e⟩≡ (45b)
```
ireg: TOPAR reg TCPAR { $2 }
```

6.6.5 Named memory locations, symbols

⟨*grammar rule* name 49f⟩≡ (45b)
```
name:
 | TIDENT offset         TOPAR pointer TCPAR { $4 (Some (mk_e $1 false)) $2 }
 | TIDENT TLT TGT offset TOPAR TSB     TCPAR { Global (mk_e $1 true, $4) }
```

⟨*function* Parser_asm.mk_e 49g⟩≡ (69b)
```
let mk_e name static =
  { name; priv = if static then Some (-1) else None; signature = None; }
```

⟨*grammar rule* pointer 49h⟩≡ (45b)
```
pointer:
 | TSB  { (fun name_opt offset ->
           match name_opt with
           | None -> error "identifier expected"
           | Some e -> Global (e, offset)
         )
        }
 | TSP  { (fun name_opt offset ->
           match name_opt with
           | None -> Param (None, offset)
           | Some ({name = s; priv = _false; signature = _}) -> Param (Some s, offset)
```

```
          )
        }
  | TFP  { (fun name_opt offset ->
            match name_opt with
            | None -> Local (None, offset)
            | Some ({name = s; priv = _false; signature = _}) -> Local (Some s, offset)
            )
          }
```

⟨*grammar rule* `offset` 50a⟩≡ (45b)

```
offset:
  | /* empty */ { 0 }
  | TPLUS  con  { $2 }
  | TMINUS con  { - $2 }
```

6.6.6 Code references, labels

⟨*grammar rule* `branch` 50b⟩≡ (45b)

```
branch:
  | rel               { $1 }
  | global            { ref (SymbolJump $1) }
  | ireg              { ref (IndirectJump $1) }
```

⟨*grammar rule* `rel` 50c⟩≡ (45b)

```
rel:
  | TIDENT offset        { ref (LabelUse ($1, $2)) }
  | con TOPAR TPC TCPAR  { ref (Relative $1) }
```

6.7 Pseudo instructions

⟨`line` *other cases* 50d⟩+≡ (47c) ◁48a

```
  | pseudo_instr  TSEMICOLON { [(Pseudo $1, $2)] }
```

⟨*grammar rule* `pseudo_instr` 50e⟩≡ (45b)

```
pseudo_instr:
  | TTEXT  global TC imm
      { TEXT  ($2, noattr, $4) }
  | TGLOBL global TC imm
      { GLOBL ($2, noattr, $4) }

  /*(* less: would be better to have mnemonics for attributes too *)*/
  | TTEXT global TC con TC imm
      { TEXT ($2, attributes_of_int $4, $6) }
  | TGLOBL global TC con TC imm
      { GLOBL ($2, attributes_of_int $4, $6) }

  | TDATA global_and_offset TSLASH con TC ximm
      { DATA (fst $2, snd $2, $4, $6) }
  | TWORD ximm
      { WORD $2 }
```

⟨*grammar rule* `global` 50f⟩≡ (45b)

```
global: name
   { match $1 with
     | Global (e, 0) -> e
     | _ -> error "global (without any offset) expected"
   }
```

⟨*grammar rule* `global_and_offset` 51a⟩≡ (45b)
```
global_and_offset: name
  { match $1 with
    | Global (e, n) -> (e, n)
    | _ -> error "global with offset expected"
  }
```

6.7.1 TEXT/GLOBL

6.7.2 WORD/DATA

6.8 ARM conditional execution

⟨*grammar rule* `cond` 51b⟩≡ (45b)
```
cond:
 | /* empty */ { AL }
 | TCOND  { $1 }
```

6.9 Special bits

6.9.1 Arithmetic instructions and .S

⟨*type* `Ast_asm5.arith_cond` 51c⟩≡ (65c)
```
    and arith_cond = Set_condition (* .S *)
```

6.9.2 Moves and .P/.W

⟨*type* `Ast_asm5.move_cond` 51d⟩≡ (65c)
```
      and move_cond = WriteAddressBase (* .W *) | PostOffsetWrite (* .P *)
```

Chapter 7

Resolving

⟨*function* Resolve_labels.error 52a⟩≡ (71b)
```
let error (s : string) (line, _locs) =
  (* TODO: use Location_cpp.Error instead or use locs at least! *)
  failwith (spf "%s at line %d" s line)
```

⟨*function* Resolve_labels.resolve 52b⟩≡ (71b)
```
(* ocaml: see how little processing 5a actually does :) *)
let resolve branch_opd_of_instr (prog : 'instr program) : 'instr program =
  let pc : virt_pc ref = ref 0 in
  let h = Hashtbl.create 101 in

  let (ps, locs) = prog in

  (* first pass, process the label definitions, populate h *)
  ps |> List.iter (fun (p, line) ->
    (match p with
    | LabelDef lbl ->
        (* better to check duplicate here than via lexing tricks *)
        if Hashtbl.mem h lbl
        then error (spf "redeclaration of %s" lbl) (line, locs);

        Hashtbl.add h lbl !pc;
        (* no incr pc; share pc if multiple labels at same place *)
    (* coupling: must be consistent with second pass below and incr pc
     * for the same cases!
     *)
    | Instr _ | Pseudo (TEXT _ | WORD _) | Virtual (RET | NOP) ->
        incr pc
    | Pseudo (DATA _ | GLOBL _ ) -> ()
    )
  );

  (* second pass, process the label uses, resolve some branch operands *)
  pc := 0;
  (* filter but modify also by side effect p *)
  let ps' = ps |> List.filter (fun (p, line) ->
    (match p with
    (* no need to keep the labels in the object file *)
    | LabelDef _ -> false

    | Pseudo (TEXT _ | WORD _) ->
        incr pc;
        true
    | Pseudo (DATA _ | GLOBL _) ->
        (* no pc increment here *)
```

```
        true
    | Virtual (RET | NOP) ->
        incr pc;
        true
    | Instr instr ->

        (* TODO? move nested function out? *)
        let resolve_branch_operand (opd : branch_operand2 ref) : unit =
          match !opd with
          | SymbolJump _ | IndirectJump _ -> ()
          (* Relative and LabelUse -> Absolute *)
          | Relative i ->
              opd := Absolute (!pc + i)
          | LabelUse (lbl, i) ->
              (try
                 let pc = Hashtbl.find h lbl in
                 (* less: could keep label info also for debugging purpose? *)
                 opd := Absolute (pc + i)
               with Not_found ->
                 error (spf "undefined label: %s" lbl) (line, locs)
               )
          | Absolute _ ->
              raise (Impossible "Absolute can't be made via assembly syntax")
        in
        branch_opd_of_instr instr |> Option.iter resolve_branch_operand;

        incr pc;
        true
    )
  )
  in
  ps', locs
```

⟨*function* Ast_asm5.branch_opd_of_instr 53⟩≡ (65c)

```
let branch_opd_of_instr (instr : instr_with_cond) : A.branch_operand option =
  (* less: could issue warning if cond <> AL when B or Bxx, or normalize? *)
  match fst instr with
  (* ocaml-light: | B opd | BL opd | Bxx (_, opd) -> *)
  | B opd -> Some opd
  | BL opd -> Some opd
  | Bxx (_cond, opd) -> Some opd
  | Arith _ | ArithF _ | MOVE _ | SWAP _ | Cmp _ | CmpF _ | SWI _ | RFE -> None
```

Chapter 8

Object Code Generation

8.1 Object format

⟨*signature* Object_file.version 54a⟩≡ (67a)
```
(* used also in Library_file.ml *)
val version : int
```

⟨*constant* Object_file.version 54b⟩≡ (67b)
```
let version = 6
```

⟨*signature* Object_file.is_obj_filename 54c⟩≡ (67a)
```
(* look whether the filename finishes in .o[5vi] *)
val is_obj_filename : Fpath.t -> bool
```

⟨*function* Object_file.is_obj_filename 54d⟩≡ (67b)
```
let is_obj_filename (file : Fpath.t) : bool =
  !!file =~ ".*\\.o[5v]$"
```

8.2 Program output

⟨*function* Object_file.save 54e⟩≡ (67b)
```
let save (arch : Arch.t) (obj : 'instr Ast_asm.program) (chan : Chan.o) : unit =
  Logs.info (fun m -> m "Saving %s object in %s" (Arch.thestring arch)
      (Chan.destination chan));
  output_value chan.oc (version, { prog = obj; arch })
```

⟨*exception* Object_file.WrongVersion 54f⟩≡ (67)
```
exception WrongVersion
```

⟨*signature* Object_file.load 54g⟩≡ (67a)
```
(* may raise WrongVersion *)
val load: Chan.i (* obj file *) -> 'instr t
```

⟨*function* Object_file.load 54h⟩≡ (67b)
```
let load (chan : Chan.i) : 'instr t =
  Logs.info (fun m -> m "Loading object %s" (Chan.origin chan));
  let (ver, obj) = input_value chan.ic in
  if ver <> version
  then raise WrongVersion
  else obj
```

8.3 Object file symbol table

Chapter 9

Debugging Support

9.1 Line origin history

9.2 Recording history

9.3 Displaying history

9.4 Saving history

Chapter 10

Advanced Topics

10.1 Other assembly language features

⟨Parse_asm5.token *in* TIDENT *case, other cases* 56a⟩≡ (41a) 57f▷
```
(* less: special bits *)
```

10.1.1 Constant expressions

⟨con *other cases* 56b⟩+≡ (48f) ◁48g
```
| TOPAR expr TCPAR { $2 }
```

⟨*grammar rule* expr 56c⟩≡ (45b)
```
expr:
 | con { $1 }

 | expr TPLUS expr  { $1 + $3 }
 | expr TMINUS expr { $1 - $3 }
 | expr TMUL expr   { $1 * $3 }
 | expr TSLASH expr { $1 / $3 }
 | expr TMOD expr   { $1 mod $3 }

 | expr TLT TLT expr { $1 lsl $4 }
 | expr TGT TGT expr { $1 asr $4 }

 | expr TAND expr    { $1 land $3 }
 | expr TOR expr     { $1 lor $3 }
 | expr TXOR expr    { $1 lxor $3 }
```

10.2 Other instructions and registers

10.2.1 Floating-point numbers

⟨*type* Ast_asm.floatp 56d⟩≡ (62b)
```
type floatp = float
```

⟨Ast_asm.ximm *other cases* 56e⟩+≡ (27h) ◁28g
```
| Float of floatp
```

⟨Ast_asm5.instr *arithmetic instructions cases* 56f⟩+≡ (29e) ◁30a
```
| ArithF of (arithf_opcode * A.floatp_precision) *
    (A.floatp, freg) Either_.t * freg option * freg
```

⟨*type* `Ast_asm.fregister` 57a⟩≡ (62b)
```
type fregister = FR of int (* between 0 and 15 on ARM *)
```

⟨*type* `Ast_asm5.freg` 57b⟩≡ (65c)
```
type freg = A.fregister (* between 0 and 15 *)
```

⟨`Ast_asm5.instr` *control-flow instructions cases* 57c⟩+≡ (29e) ◁31a
```
| CmpF of A.floatp_precision * freg * freg
```

⟨*type* `Ast_asm5.arithf_opcode` 57d⟩≡ (65c)
```
    and arithf_opcode =
      | ADD_ | SUB_ | MUL_ | DIV_
```

⟨*type* `Ast_asm.floatp_precision` 57e⟩≡ (62b)
```
type floatp_precision = F (* Float *) | D (* Double *)
```

⟨`Parse_asm5.token` *in* `TIDENT` *case, other cases* 57f⟩+≡ (41a) ◁56a 58a▷
```
(* float, MUL, ... *)
| "ADDF" -> TARITHF (ADD_, A.F)
| "SUBF" -> TARITHF (SUB_, A.F)
| "MULF" -> TARITHF (MUL_, A.F)
| "DIVF" -> TARITHF (DIV_, A.F)
| "ADDD" -> TARITHF (ADD_, A.D)
| "SUBD" -> TARITHF (SUB_, A.D)
| "MULD" -> TARITHF (MUL_, A.D)
| "DIVD" -> TARITHF (DIV_, A.D)

| "CMPF" -> TCMPF A.F
| "CMPD" -> TCMPF A.D
```

⟨*grammar rule* `freg` 57g⟩≡ (45b)
```
freg:
 | TFx                { $1 }
 | TF TOPAR con TCPAR
     { if $3 <= 15 && $3 >= 0
       then FR $3
       else error "register value out of range"
     }
```

⟨*grammar rule* `frcon` 57h⟩≡ (45b)
```
frcon:
  | freg { Right $1 }
  | fcon { Left $1 }
```

⟨*grammar rule* `fcon` 57i⟩≡ (45b)
```
fcon:
 | TDOLLAR TFLOAT { $2 }
 | TDOLLAR TMINUS TFLOAT { -. $3 }
```

10.2.2 Multiplication and accumulation

10.2.3 64-bits multiplication

10.2.4 Moving multiple registers at the same time

10.2.5 Program status register

10.2.6 Mutual exclusion instructions

10.2.7 Coprocessors

⟨*type* `Ast_asm5.creg` 57j⟩≡ (65c)

```
  type creg = C of int (* between 0 and 15 *)
```

⟨Parse_asm5.token *in* TIDENT *case, other cases* 58a⟩+≡ (41a) ◁57f
```
  (* advanced *)
  | "C" -> TC
  | _ when s =~ "^C\\([0-9]+\\)$" ->
        let i = int_of_string (Regexp_.matched1 s) in
        if i >= 0 && i <= 15
        then TCx (C i)
        else Lexer_asm.error ("register number not valid")
```

10.3 Other pseudo opcodes

10.3.1 Compiler-only opcodes

10.3.2 Linker-only opcodes

10.4 TEXT attributes

⟨*type* Ast_asm.attributes 58b⟩≡ (62b)
```
    and attributes = { dupok: bool; prof: bool }
```

⟨*constant* Parser_asm.noattr 58c⟩≡ (69b)
```
  let noattr = { dupok = false; prof = true }
```

⟨*function* Parser_asm.attributes_of_int 58d⟩≡ (69b)
```
  (* less: should use keywords in Asm5 instead of abusing integers
   * alt: anyway one can also use cpp to define NOPROF/DUPOK macros
   *)
  let attributes_of_int i =
     match i with
     | 0 -> noattr
     (* NOPROF *)
     | 1 -> { dupok = false; prof = false }
     (* DUPOK *)
     | 2 -> { dupok = true; prof = true }
     (* both DUPOK and NOPROF *)
     | 3 -> { dupok = true; prof = false }

     | _ -> error (spf "unknown attribute or attribute combination: %d" i)
```

Chapter 11

Conclusion

Appendix A

Debugging

A.1 AST debugging: `5a -dump_ast`

⟨*constant* `CLI.dump_ast` 60⟩≡ (68c)

```
let dump_ast = ref false
```

A.2 Line information debugging: `5a -f`

A.3 Macro debugging: `5a -m`

Appendix B

Examples of Assembly Programs

B.1 `hello.s` and `pwrite.s`

B.2 `memset.s`

B.3 `div.s`

B.4 `main9.s`

B.5 `tas.s`

B.6 `getcallerpc.s`

Appendix C

Extra Code

C.1 objects/Ast_asm.ml

⟨*function* Ast_asm.visit_globals_program 62a⟩≡ (62b)
```
  (* Visit globals (e.g., to populate symbol table with wanted symbols in linker).
   * This is more complicated than in 51 because we can not rely
   * on an ANAME or Operand.sym.
   * less: boilerplate which could be auto generated by ocamltarzan in
   *  a visitor_asm.ml
   *)
  let rec visit_globals_program visit_instr (f : global -> unit) (prog : 'instr program) : unit =
    let (xs, _locs) = prog in
    xs |> List.iter (fun (x, _line) ->
      match x with
      | Pseudo y ->
        (match y with
        | TEXT (ent, _, _) -> f ent
        | GLOBL (ent, _, _) -> f ent
        | DATA (ent, _, _, ix) -> f ent; visit_globals_ximm f ix
        | WORD (ix) -> visit_globals_ximm f ix
        )
      | Virtual (RET | NOP) | LabelDef _ -> ()
      | Instr instr ->
        visit_instr f instr
    )

  and visit_globals_ximm f x =
    match x with
    | Address (Global (x, _)) -> f x
    | Address (Param _ | Local _) | String _ | Int _ | Float _ -> ()
  and visit_globals_branch_operand f x =
    match !x with
    | SymbolJump ent -> f ent
    | IndirectJump _ | Relative _ | LabelUse _ | Absolute _ -> ()
```

⟨objects/Ast_asm.ml 62b⟩≡
```
  (* Copyright 2025 Yoann Padioleau, see copyright.txt *)
  open Common

  (*****************************************************************************)
  (* Prelude *)
  (*****************************************************************************)
  (* Types common to the different Plan 9 assembler ASTs.
   *
   * Note that in Plan 9 object files are mostly the serialized form of
```

```
 * the assembly AST, which is why this file is in this directory.
 *
 * !!! If you modify this file please increment Object_file.version !!!
 *)

(*****************************************************************************)
(*  AST related types *)
(*****************************************************************************)

(* ------------------------------------------------------------------------ *)
(* Location *)
(* ------------------------------------------------------------------------ *)
⟨type Ast_asm.loc 26b⟩
[@@deriving show]

(* ------------------------------------------------------------------------ *)
(* Numbers and Strings *)
(* ------------------------------------------------------------------------ *)
⟨type Ast_asm.virt_pc 28d⟩
[@@deriving show]

(* 'int' enough for ARM 32 bits? on 64 bits machine it is enough :)
 * TODO: use Int64.t so sure it's enough for every arch.
 * alt: have separate type in each Ast_asmxxx.ml, with more precise size
 * but not worth it because it prevents more generalization in this file
 * like imm_or_ximm, which in turn allow to generalize pseudo_instr,
 * and anyway Plan 9 asms are not a direct match of the machine assembly.
*)
⟨type Ast_asm.integer 26c⟩
[@@deriving show]

(* floating point number.
 * TODO: float enough too? can contain 'double' C?
 * alt? use explicit float record { Int32.t; Int32.t mantisse/exp } ?
*)
⟨type Ast_asm.floatp 56d⟩
[@@deriving show]

⟨type Ast_asm.offset 27g⟩
[@@deriving show]

⟨type Ast_asm.label 28c⟩
[@@deriving show]

⟨type Ast_asm.symbol 27f⟩
[@@deriving show]

⟨type Ast_asm.global 27e⟩
[@@deriving show]

(* ------------------------------------------------------------------------ *)
(* Operands *)
(* ------------------------------------------------------------------------ *)
⟨type Ast_asm.register 24b⟩
[@@deriving show]

⟨type Ast_asm.fregister 57a⟩
[@@deriving show]

⟨type Ast_asm.entity 28h⟩
```

```
[@@deriving show]

(* extended immediate *)
⟨type Ast_asm.ximm 27h⟩
[@@deriving show]

(* I use a ref below so the code that resolves branches is shorter.
 * The ref is modified by the assembler and then by the linker.
 *)
⟨type Ast_asm.branch_operand 28e⟩
⟨type Ast_asm.branch_operand2 28f⟩
[@@deriving show]

(* In a MOVE, sign is relevant only for a load operation *)
⟨type Ast_asm.sign 29d⟩
[@@deriving show]

(* TODO: add Dword? use shorter W_ | H_ | B_ | D_ ?  *)
⟨type Ast_asm.move_size 30f⟩
[@@deriving show]

⟨type Ast_asm.floatp_precision 57e⟩
[@@deriving show]

(* ------------------------------------------------------------------------ *)
(* Instructions *)
(* ------------------------------------------------------------------------ *)

⟨type Ast_asm.pseudo_instr 27d⟩

⟨type Ast_asm.attributes 58b⟩
[@@deriving show {with_path = false}]

(* alt: move in arch-specific Ast_asmx.instr
 * alt: merge with pseudo_instr
*)
⟨type Ast_asm.virtual_instr 28b⟩
[@@deriving show]

(* ------------------------------------------------------------------------ *)
(* Program *)
(* ------------------------------------------------------------------------ *)

⟨type Ast_asm.line 27c⟩
[@@deriving show {with_path = false}]

⟨type Ast_asm.lines 27b⟩
[@@deriving show]

⟨type Ast_asm.program 27a⟩
[@@deriving show]

(*****************************************************************************)
(*  Extractors/visitors *)
(*****************************************************************************)

⟨function Ast_asm.s_of_global 27j⟩

⟨function Ast_asm.visit_globals_program 62a⟩
```

C.2 objects/Ast_asm5.ml

⟨*constant* Ast_asm5.rTMP 65a⟩≡ (65c)
```
let rTMP = R 11
```

⟨*function* Ast_asm5.visit_globals_instr 65b⟩≡ (65c)
```
let visit_globals_instr (f : global -> unit) (i : instr_with_cond) : unit =
  let mov_operand x =
    match x with
    | Entity (A.Global (x, _)) -> f x
    | Entity (A.Param _ | A.Local _) -> ()
    | Ximm x -> A.visit_globals_ximm f x
    | Imsr _ | Indirect _ -> ()
  in
  match fst i with
  | MOVE (_, _, m1, m2) -> mov_operand m1; mov_operand m2
  (* ocaml-light: | B b | BL b | Bxx (_, b) -> branch_operand b *)
  | B b -> A.visit_globals_branch_operand f b
  | BL b -> A.visit_globals_branch_operand f b
  | Bxx (_, b) -> A.visit_globals_branch_operand f b
  | Arith _ | ArithF _ | SWAP _ | Cmp _ | CmpF _ | SWI _ | RFE -> ()
```

⟨objects/Ast_asm5.ml 65c⟩≡
```
(* Copyright 2015, 2016 Yoann Padioleau, see copyright.txt *)
open Common

module A = Ast_asm
open Ast_asm

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* Abstract Syntax Tree (AST) for the assembly language supported by 5a.
 * I call this language Asm5.
 *
 * Note that many types are now defined in Ast_asm.ml instead because they are
 * mostly arch independent and can be reused in other plan9 assemblers
 * (e.g. va, ia, 7a).
 *
 * !!! If you modify this file please increment Object_file.version !!!
 *
 * TODO:
 *  - 5c-only opcodes? CASE, BCASE, MULU/DIVU/MODU (or better in Ast_asm.ml too?)
 *  - MULA, MULL,
 *  - MOVM (and his special bits .IA/...),
 *  - PSR, MCR/MRC,
 *  - handle the instructions used in the kernel
 *)

(*****************************************************************************)
(* The AST related types *)
(*****************************************************************************)

(* ------------------------------------------------------------------------ *)
(* Numbers and Strings *)
(* ------------------------------------------------------------------------ *)
(* see Ast_asm.ml *)

(* ------------------------------------------------------------------------ *)
(* Operands *)
```

```
(* ------------------------------------------------------------------------ *)
⟨type Ast_asm5.reg 29f⟩
[@@deriving show]

⟨type Ast_asm5.freg 57b⟩
[@@deriving show]

(* ?? *)
⟨type Ast_asm5.creg 57j⟩

(* reserved by linker *)
⟨constant Ast_asm5.rTMP 65a⟩
⟨constant Ast_asm5.rSB 29j⟩
⟨constant Ast_asm5.rSP 29k⟩
(* reserved by hardware *)
⟨constant Ast_asm5.rLINK 29g⟩
⟨constant Ast_asm5.rPC 29h⟩

⟨constant Ast_asm5.nb_registers 29i⟩

⟨type Ast_asm5.arith_operand 30c⟩

⟨type Ast_asm5.shift_reg_op 30d⟩
[@@deriving show]

(* alt: could almost be moved to Ast_asm.ml but Shift above of arith_operand
 * seems arm-specific
 *)
⟨type Ast_asm5.mov_operand 30g⟩

[@@deriving show]

(* ------------------------------------------------------------------------ *)
(* Instructions *)
(* ------------------------------------------------------------------------ *)

(* less: could probably factorize things and move stuff in Ast_asm.ml *)
⟨type Ast_asm5.instr 29e⟩

⟨type Ast_asm5.arith_opcode 30b⟩
⟨type Ast_asm5.arith_cond 51c⟩

⟨type Ast_asm5.arithf_opcode 57d⟩

⟨type Ast_asm5.cmp_opcode 31b⟩

⟨type Ast_asm5.condition 29c⟩

⟨type Ast_asm5.move_option 30h⟩
     (* this is used only with a MOV with an indirect with offset operand *)
⟨type Ast_asm5.move_cond 51d⟩

[@@deriving show]

(* ------------------------------------------------------------------------ *)
(* Program *)
(* ------------------------------------------------------------------------ *)

⟨type Ast_asm5.instr_with_cond 29b⟩
[@@deriving show]
```

```
⟨type Ast_asm5.program 29a⟩
[@@deriving show]

(*****************************************************************************)
(* Extractors/Visitors *)
(*****************************************************************************)

⟨function Ast_asm5.branch_opd_of_instr 53⟩

⟨function Ast_asm5.visit_globals_instr 65b⟩
```

C.3 objects/Object_file.mli

```
⟨objects/Object_file.mli 67a⟩≡

  ⟨type Object_file.t 31d⟩

  ⟨exception Object_file.WrongVersion 54f⟩
  ⟨signature Object_file.version 54a⟩

  ⟨signature Object_file.load 54g⟩

  ⟨signature Object_file.save 35a⟩

  ⟨signature Object_file.is_obj_filename 54c⟩
```

C.4 objects/Object_file.ml

```
⟨objects/Object_file.ml 67b⟩≡
  (* Copyright 2015, 2016 Yoann Padioleau, see copyright.txt *)
  open Common
  open Regexp_.Operators
  open Fpath_.Operators

  (*****************************************************************************)
  (* Prelude *)
  (*****************************************************************************)

  (*****************************************************************************)
  (* Types and constants *)
  (*****************************************************************************)
  (* TODO? could also add file origin? *)
  ⟨type Object_file.t 31d⟩

  (* less: could be sha1 of ast_asmxxx.ml for even safer marshalling *)
  ⟨constant Object_file.version 54b⟩

  (*****************************************************************************)
  (* API *)
  (*****************************************************************************)
  (* TODO? can normalize before? or check every invariants?
   * TODO? pass and save arch name string? like in goken? and
   * check for it at loading time?
   *)
  ⟨function Object_file.save 54e⟩
```

```
(* for slightly safer marshalling
 * TODO: exception WrongArch ?
 *)
⟨exception Object_file.WrongVersion 54f⟩

⟨function Object_file.load 54h⟩

⟨function Object_file.is_obj_filename 54d⟩
```

C.5 CLI.mli

⟨CLI.mli 68a⟩≡
```
⟨type CLI.caps 32b⟩

⟨signature CLI.main 32c⟩

⟨signature CLI.assemble 34a⟩
```

C.6 CLI.ml

⟨function CLI.assemblev 68b⟩≡ (68c)
```
let assemblev (caps: < Cap.open_in; .. >) (conf : Preprocessor.conf) (infile : Fpath.t) : Ast_asmv.program =
  let prog = Parse_asmv.parse caps conf infile in
  let prog = Resolve_labels.resolve Ast_asmv.branch_opd_of_instr prog in
  if !dump_ast
  then Logs.app (fun m -> m "AST = %s" (Ast_asmv.show_program prog));
  prog
```

⟨CLI.ml 68c⟩≡
```
(* Copyright 2015, 2016, 2025 Yoann Padioleau, see copyright.txt *)
open Common
open Fpath_.Operators
open Regexp_.Operators

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
(* An OCaml port of 5a/va, the Plan 9 ARM/MIPS assemblers.
 *
 * Main limitations compared to 5a/va/...:
 *  - no multiple files processing in parallel
 *    (not the place, use xargs)
 *  - no unicode support? or can ocamllex in ocaml-light do unicode?
 *
 * better than 5a/va/...:
 *  - far greater code reuse across all assemblers thanks to:
 *    * Lexer_asm.mll factorization
 *    * Ast_asm.ml and 'instr polymorphic type
 *    * use of simple marshalling instead of adhoc object format
 *      (which were slightly different in each arch)
 *    * separate AST generation from resolving which allows to factorize
 *      checks of redefinition and the resolution of labels
 *    * generalize more code in macroprocessor/
 *    * use simple marshal again for cpp line history
 *
```

```
 * todo:
 *  - port remaining assembler: ia, 7a, 8a, 6a (increasing difficulty)
 * later:
 *  - look at the 5a Go sources in the Golang source, maybe ideas to steal?
 *  - follow the new design by Rob Pike of Go assembler to factorize things
 *    (see https://www.youtube.com/watch?v=KINIAgRpkDA&feature=youtu.be )
 *    (=~ 2 tables, register string -> code, and opcode string -> code)
 *)

(*****************************************************************************)
(* Types, constants, and globals *)
(*****************************************************************************)
⟨type CLI.caps 32b⟩

⟨constant CLI.dump_ast 60⟩

(*****************************************************************************)
(* Main algorithm *)
(*****************************************************************************)

⟨function CLI.assemble5 35b⟩
⟨function CLI.assemblev 68b⟩

⟨function CLI.assemble 34b⟩

(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)
⟨function CLI.main 32d⟩
```

C.7 Lexer_asm.mll

C.8 Main.ml

⟨Main.ml 69a⟩≡

```
(* Copyright 2025 Yoann Padioleau, see copyright.txt *)
(* open Xix_assembler *)

(*****************************************************************************)
(* Entry point *)
(*****************************************************************************)

⟨toplevel Main._1 32a⟩
```

C.9 Parser_asm.ml

⟨Parser_asm.ml 69b⟩≡

```
open Common

open Ast_asm
module L = Location_cpp

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)
```

```
(* Helpers used in the different arch-specific parsers.
 *
 * General limitations compared to 5a/va/...:
 *  - no support for 'NAME=expr;' constant definition, nor use of it
 *    in expressions, which allows to evaluate constant at parsing time
 *    and avoid the need to build an expr AST (as we want to separate AST
 *    generation from checks/resolution/eval)
 *    (but can use cpp for constant definition so not a big loss)
 *  - does not allow SP and PC in a few places with 'spreg' and 'sreg' original
 *    grammar rule
 *    (but can use directly R13 and R15)
 *  - no END instruction
 *)

(*****************************************************************************)
(* Helpers *)
(*****************************************************************************)

⟨function Parser_asm.error 44a⟩

⟨constant Parser_asm.noattr 58c⟩
⟨function Parser_asm.attributes_of_int 58d⟩

⟨function Parser_asm.mk_e 49g⟩
```

C.10 Parse_asm5.mli

⟨Parse_asm5.mli 70a⟩≡

```
⟨signature Parse_asm5.parse 35c⟩

⟨signature Parse_asm5.parse_no_cpp 35e⟩
```

C.11 Parse_asm5.ml

⟨Parse_asm5.ml 70b⟩≡

```
(* Copyright 2015, 2016 Yoann Padioleau, see copyright.txt *)
open Common
open Regexp_.Operators

module L = Location_cpp
module T = Token_asm
module A = Ast_asm
open Parser_asm5
open Ast_asm5

(*****************************************************************************)
(* Prelude *)
(*****************************************************************************)

(*****************************************************************************)
(* Lexer *)
(*****************************************************************************)
⟨function Parse_asm5.token 26a⟩

(*****************************************************************************)
```

```
  (* Entry points *)
  (*****************************************************************************)
```
⟨*function* Parse_asm5.parse 36⟩

⟨*function* Parse_asm5.parse_no_cpp 35f⟩

C.12 Parser_asm5.mly

C.13 Resolve_labels.mli

⟨Resolve_labels.mli 71a⟩≡

⟨*signature* Resolve_labels.resolve 35d⟩

C.14 Resolve_labels.ml

⟨Resolve_labels.ml 71b⟩≡
```
  (* Copyright 2015, 2016, 2025 Yoann Padioleau, see copyright.txt *)
  open Common

  open Ast_asm

  (*****************************************************************************)
  (* Prelude *)
  (*****************************************************************************)
  (* Resolve and remove the LabelDef/LabelUse constructs as well as
   * the branch_operand Relative (turned into Absolute like for LabelUse).
   *
   * As you can see, the Plan 9 assembler does not do much. Most of the work
   * is done in the linker really.
   *)

  (*****************************************************************************)
  (* Error management *)
  (*****************************************************************************)
```
⟨*function* Resolve_labels.error 52a⟩
```
  (*****************************************************************************)
  (* Entry point *)
  (*****************************************************************************)
```
⟨*function* Resolve_labels.resolve 52b⟩

C.15 Token_asm.ml

⟨Token_asm.ml 71c⟩≡

⟨*type* Token_asm.t 24a⟩

Glossary

LOC = Lines Of Code

Indexes

Here is a list of the identifiers used, and where they appear. Underlined entries indicate the place of definition. This index is generated automatically.

Bibliography

[BLM92] Doug Brown, John Levine, and Tony Mason. *Lex and Yacc*. O'Reilly, 1992. cited page(s) 9

[Dun00] Jeff Duntemann. *Assembly Language Step-by-step: Programming with DOS and Linux*. Wiley, 2000. cited page(s) 7, 9

[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) 7

[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) 6

[Knu92] Donald E. Knuth. *Literate Programming*. Center for the Study of Language and Information, 1992. cited page(s) 9

[Knu99] Donald E. Knuth. *MMIXWare, a RISC Computer for the Third Millenium*. Springer, 1999. cited page(s) 8

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988. cited page(s) 9

[Pad09] Yoann Padioleau. Syncweb, literate programming meets unison, 2009. https://github.com/aryx/syncweb. cited page(s) 9

[Pad14] Yoann Padioleau. *Principia Softwarica: The Plan 9 Kernel 9*. 2014. cited page(s) 9, 16

[Pad15a] Yoann Padioleau. *Principia Softwarica: The ARM Emulator 5i*. 2015. cited page(s) 9, 16, 21

[Pad15b] Yoann Padioleau. *Principia Softwarica: The ARM Linker 5l*. 2015. cited page(s) 7, 9, 14, 18, 23

[Pad16a] Yoann Padioleau. *Principia Softwarica: The Plan 9 C Compiler 5c*. 2016. cited page(s) 9

[Pad16b] Yoann Padioleau. *Principia Softwarica: The Plan 9 Core Libraries*. 2016. cited page(s) 13, 15

[PH13] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 2013. cited page(s) 9

[PH16] David A. Patterson and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface (ARM Edition)*. Morgan Kaufmann, 2016. cited page(s) 9

[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) 6, 9

[Rit79] Dennis M. Ritchie. Assembler reference manual. In *Unix Programmer's Manual Vol 2b*, 1979. cited page(s) 7

[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) 9

[Sea01] David Seal. *ARM, Architecture Reference Manual*. Addison-Wesley, 2001. cited page(s) 7, 21

[Tan88] Andrew S Tanenbaum. *Structured Computer Organization*. Prentice Hall, 1988. cited page(s) 9