Compiling to Assembly

from Scratch

— Extract from the Draft —

Vladimir Keleshev

July 2020

Contents

Chapter 1

Introduction

> It is not the gods who make our pots
>
> *Ancient proverb*

Welcome, to the wonderful journey of writing your own compiler!

Having bought this book, you are probably already quite convinced that you want to understand how compilers work, and even want to write one. Nevertheless, here's a list of some of the reasons to do it:

- Writing a compiler is the ultimate step in understanding how computers work and how they execute our programs.
- By writing a small compiler, you can see that they are programs, just like others, and they are not magic made by gods.
- By understanding assembly and how compilers translate your programs to it, you can better grasp the performance of the programs you write.
- It will allow you to see the trade-offs of different language features more clearly, so you are better informed when to use them and how to use them effectively.
- Learning about parsing will help you deal with unstructured data, like scraping, or dealing with a data format for which you don't have a library.
- It will also prepare you for making your own domain-specific languages, when necessary, for the tasks at hand.

- It may be a first step into the field of compiler engineering, a lucrative and exciting job.
- And finally, it will allow you to create and experiment with a language of your making, and experience the fun and excitement of crafting your own language!

The topic of making compilers is the single most researched topic in computer science. Nothing else comes close. So there's a massive amount of useful techniques and algorithms in compiler literature. And it turns out, a lot of it is very applicable to our day-to-day programming, it's just that whatever we are working on today is not as well researched, at the moment. There's also a school of thought that, in the end, maybe all programs are compilers. Maybe we are not writing web apps, but compilers from DOM nodes to JSON and from JSON to SQL, who knows!

1.1 Structure of the book

The book describes the design and implementation of a compiler written in TypeScript, which compiles a small language to 32-bit ARM assembly.

The book consists of two parts.

Part I describes the design and development of a minimal *baseline compiler* in great detail. We call it a *baseline compiler* because it lays the foundation for developing more advanced features introduced in *Part II*. The *implementation language* of the compiler is TypeScript. But the compiler's *source* or *input* language is a *subset* (or a simplified version) of TypeScript. This subset consists of things common to any practical programming language, not specific to TypeScript: arithmetic and comparison operators, integer numbers, functions, conditional statements and loops, local variables, and assignments. We call this language the *baseline language*. It can express simple programs and functions, like this one, for example:

```
function factorial(n) {
  var result = 1;
  while (n != 1) {
    result = result * n;
    n = n - 1;
  }
  return result;
```

```
}
```

Part II builds upon the *baseline compiler* and describes various *compiler extensions* in lesser detail. Those extensions are often mutually exclusive (like static typing and dynamic typing), but they all use the baseline compiler as the foundation.

> **Note!**
>
> *Part II* is not available in this draft yet, but I am working on it, and it will be added to a next revision of the draft.

Appendix A describes how to run the ARM assembly code the compiler produces. You can skip this if you're developing your compiler on a computer which is based on an ARM processor with a 32-bit operating system like Raspberry Pi OS (formerly Raspbian). However, if you are running an x86-64 system like those from Intel and AMD, you need to see *Appendix A*.

Appendix B describes the differences between the two mainstream ARM assembly syntaxes: the GNU assembler (GAS) syntax, and the legacy ARMASM syntax.

1.2 Why ARM?

In many ways, the ARM instruction set is what makes this book possible.

Compared to Intel x86-64, the ARM instruction set is a work of art.

Intel x86-64 is the result of evolution from an 8-bit processor, to a 16-bit one, then to a 32-bit one, and finally to a 64-bit one. At each step of the evolution, it accumulated complexity and cruft. At each step, it tried to satisfy conflicting requirements.

- Intel x86-64 is based on *Complex Instruction Set Architecture* (CISC), which was initially optimized for writing assembly by hand.
- ARM, on the other hand, is based on *Reduced Instruction Set Architecture* (RISC), which is optimized for writing compilers.

Guess which one is an easier target for a compiler?

If this book targeted x86-64 instead of ARM, it would have been two times as long and — more likely — never written. Also, with

160 billion devices shipped, we better get used to the fact that ARM is the dominant instruction set architecture today.

In other words, ARM is a good start. After learning it, you will be better equipped for moving to x86-64 or the new ARM64.

1.3 Why TypeScript?

This book describes the design and development of a compiler written in TypeScript, which compiles a small language that also uses TypeScript syntax.

The compiler doesn't have to be written in TypeScript. It could be written in any language, but I had to pick. I have used a straightforward subset of TypeScript for the examples, to make it readable for anyone who knows one or more mainstream languages.

The next chapter, *TypeScript Basics* gives you a quick overview of the language.

1.4 How to read this book

Part I is structured linearly, with each chapter building upon the previous one. However, don't feel guilty skipping chapters, if you are already familiar with a topic.

If you plan to follow along and implement the compiler described in this book (or a similar one), I recommend first to read *Part I* without writing any code. Then you can go back to the beginning and start implementing the compiler while skimming *Part I* again.

The book is also sprinkled with the following notes, titled *Explore…*:

> **Explore…**
>
> These notes contain suggestions and ways to try out things on your own. You might find them useful for practicing and building your confidence, or you might find it more fitting to have a minimal working compiler first, and only then optionally come back to these.

You might also see some notes titled *Well, actually…*:

Well, actually…

These contain some pedantic notes which are beside the point, but the book would be incomplete without them.

We will also use *code folding* in the code snippets. We will use the ellipsis (…) to denote that some code in the snippet was omitted, usually because it was already shown before. Like this:

```
function factorial(n) {
  var result = 1;
  while (n != 1) {…}  // <- See here
  return result;
}
```

Part II is structured in mostly independent sections. Feel free to reach just for the parts you are interested in. No need to read both about *static typing* and *dynamic typing* if you want to focus only on one of these topics.

Some pages are omitted

Part I

Baseline Compiler

Chapter 3

High-level Compiler Overview

A *compiler* is a program that translates another program from one language to another.

In our case, it transforms from what we call a *baseline language* to ARM assembly language.

3.1 Types of compilers

Our compiler will be an *ahead-of-time* (AOT) compiler. Only once the compilation is finished the resulting program can be run.

There are also *just-in-time* (JIT) compilers that compile a program as it runs.

Think of AOT compilers as translator services for foreign languages: you might send them a few papers to translate from English to Japanese, and when they are done, they send the results back. On the other hand, JIT compilers are more like simultaneous translators at a business meeting: they translate participants as they speak.

Our compiler *targets* (or produces) an *assembly language*. The *assembly language* is a textual representation of the binary *machine language* that processors execute directly. It has a straight-forward translation to such binary. Such translation is called *assembling* and is much less sophisticated than what is found in a compiler. The program that performs this translation is called an *assembler*. In most cases on ARM, one assembly instruction is translated into one 32-bit binary integer. Think of assembly language as an API for directly accessing your processor's functionality.

Some compilers target binary *machine code* directly, but this is increasingly rare. Instead, most compilers compile to assembly and then call the assembler behind the scenes.

Some compilers target *byte code* instead of assembly. Byte code is similar to assembly: it consists of similar instructions. However, these do not target a real processor, but instead an *abstract machine*, which is a processor that is implemented in software. This could be done for portability reasons, or to add security features that are not available in hardware. Often byte code, in turn, is translated to machine code by a JIT compiler.

A possible compiler target could be another programming language. We call these compilers *source-to-source* compilers. For example, the TypeScript compiler is a source-to-source compiler that targets JavaScript.

3.2 Compiler passes and intermediate representations

Compilers are structured into *passes*. At the high-level, each pass is a function that takes one representation of the program and converts it to a different representation of the program. The first such representation is the source of the program. The last one is the compiled program in the target language. In between them, we have representations that are *internal* to the compiler. We call them *intermediate representations* or IR.

In the figure you can see an example of a three-pass compiler diagram.

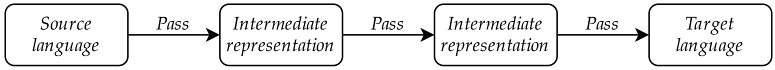

Figure 3.1: An example of a three-pass compiler

Intermediate representations of the program are data structures convenient for us to manipulate at different stages of the compiler. For one stage, we might want to use a tree-like representation. For another, we might pick a graph-like one. For some, a linear array-like representation is due.

To convert from one IR to another one, each pass needs to traverse it once (or iterate through it). That's why it's called a *pass*.

The number of passes in a compiler ranges wildly, from single-pass compilers to multiple-pass compilers with dozens of passes (those called nano-pass compilers).

The number of compiler passes presents a trade-off. On the one hand, we want to write many small passes that do one thing and are maintainable and testable in isolation. We also want to write more passes that do sophisticated analysis to improve the resulting program's performance. On the other hand, we want to minimize the number of traversals to improve our compiler's performance: how fast it compiles the programs.

Our baseline compiler is a two-pass compiler. The first pass converts the source into an IR called *abstract syntax tree* or AST. This process is called *parsing*. The second pass converts from AST to assembly. It is called *emitting code* or *code generation*.

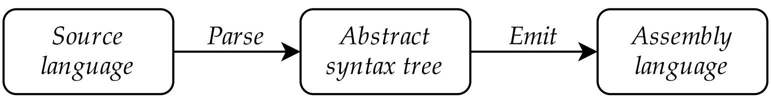

Figure 3.2: Baseline compiler structure

In *Part II* of the book, we will introduce some more passes.

Abstract syntax trees are the most common type of intermediate representations. Let's talk about them in detail.

Some pages are omitted

Chapter 7

Introduction to ARM Assembly Programming

By the end of this chapter, you will learn enough ARM assembly programming to implement the rest of the compiler.

We will be using the GNU Compiler Collection (GCC) toolchain, most notably, the GNU Assembler (GAS). For differences with ARMASM assembler, see *Appendix B*.

First, we'll take a bird-eye view of a simple hello-world program to get a taste of assembly programming. After this rough initial overview, we will dive into details.

7.1 A taste of assembly

This is not one of these *how to draw an owl* tutorials. I will assume that you have never done any assembly programming and walk your way through it. However, in the beginning, I wanted to start with a small, complete program to get a taste of assembly programming. After that, we'll cover each part in much more detail.

So here it is, our first program:

```
/* Hello-world program.
   Prints "Hello, assembly!" and exits with code 42. */
```

```
.data                                   /* 1 */
  hello:
    .string "Hello, assembly!"

.text                                   /* 2 */
  .global main
  main:
    push {ip, lr}

    ldr r0, =hello                      /* 3 */
    bl printf

    mov r0, #41                         /* 4 */
    add r0, r0, #1  // Increment

    pop {ip, lr}                        /* 5 */
    bx lr
```

What this program does is it prints `Hello, assembly!` to the console and exits with error code 42.

A quick overview:

1. The program starts with a `.data` directive. Under this directive, there are definitions of our global data, potentially mutable (or read-write). There we have only one definition, a byte string defined with a `.string` directive. It has a *label* named `hello:` which stands for the memory address of this string, which we can refer to.

2. The data section ends, and the `.text` directive starts the *code* section. This section is for immutable (read, no-write) data. It is used for constants, as well as for the actual assembly instructions. The only definition in the `.text` section is a function called `main` defined with the label `main:`. It is declared "public" using the `.global` directive. The function starts with an instruction `push` that saves some necessary registers on the stack.

3. The first thing our function does is it loads the address of the string, we defined earlier by referring to the `hello` label. The instruction `ldr` *loads* the address into *register* `r0`. The following instruction `bl printf` is the *call* instruction that calls the `printf` function to print the string. The register `r0` is used to

pass a parameter to `printf`.

4. Here, we set up the exit code. First, we use the `mov` instruction to *move*, or copy a number `41` into register `r0`. Then the `add` instruction increments `r0` by one, resulting in `42`.

5. What follows is the return sequence for our `main` function. The registers that we saved, in the beginning, are now restored with the `pop` instructions, and then we return from the function with the `bx lr` instruction, assuming that the return value is in `r0`, which should be `42`.

As you probably noticed, the familiar single- and multi-line comments are supported.

7.2 Running an assembly program

Here's how you get this simple program running.

> **Note**
>
> The instructions below assume that you are running the commands on an ARM-based computer (like Raspberry Pi) with a 32-bit operating system (like Raspberry Pi OS, formerly Raspbian). If this is not the case for you, check out *Appendix A* on how to adapt these instructions to other environments.

Save the previous program into a file called `hello.s` using a text editor. Then type the following command into the console:

```
$ gcc hello.s -o hello
```

This will instruct GCC to assemble and link our program producing a `hello` executable. By default, GCC will link our assembly with a `libc` library, which provides us with basic functions such as `printf` that we used here.

You see, the operating system kernel doesn't provide such functions directly. For example, printing to the console is implemented in libraries like `libc` on top of (operating) *system calls* like `writev`. Without these basics, we would be stuck without even being able to print to the console. However, in *Part II*, we will have a quick overview of how to make *system calls* directly.

Now, you can run the program as usual:

```
$ ./hello
Hello, assembly!
```

Oh, hello there!

We can check the exit code by printing the `$?` shell variable.

```
$ echo $?
42
```

Now that we have a template to run our programs and a rough overview, we will dive deep into the details.

We will start with the basic data structure of assembly programming, a *machine word*, then followed by an overview of how memory and registers work, and finally proceed to cover the different kinds of instructions that manipulate registers and memory.

7.3 Machine word

ARM is a 32-bit instruction set. That means that most operations work with a 32-bit data structure called the *machine word*.

In ARM, a word consists of 32 bits. Each bit is binary `0` or `1`.

Another way to look at it, a word consists of *four* bytes, where each byte is 8 bits.

There are also half-words and double-words. The names speak for themselves. Operations on them are not as common.

Let's look at the following word.

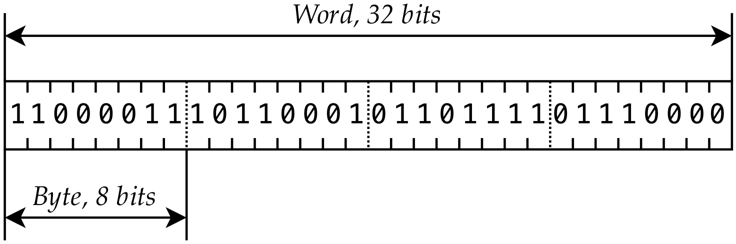

Figure 7.1: An example machine word

What does it mean? What does it stand for? Well, the processor doesn't care. It doesn't have a type system. It doesn't have any information attached to words to help us distinguish what a word stands for in isolation.

If interpreted as an unsigned integer, this word could stand for `3_283_185_520`. If interpreted as a signed integer, then it's `-1_011_781_776`. It could be a byte array of four bytes, `[195, 177, 111, 112]`. It could be an array of bits, where each bit is a single flag. Or it could be a UTF-8—encoded string `"ñop"`, where `ñ` is encoded using as two bytes `195, 177`, while `o` and `p` are encoded as single-byte ASCII characters with codes `111` and `112`. It could also be an encoding of an ARM instruction with mnemonic `rsb`. Or it could be an address pointing to some data location in memory.

It is up to us (programmers and compiler writers) to assign meaning to each word and to keep track of what they stand for.

> **Note**
>
> Throughout this book, we'll use (non-overlapping) solid boxes of different shapes and sizes to refer to 32-bit words. Like here, we'll use dashed lines to delimit individual bytes, where it adds clarity.

7.4 Numeric notation

As we already did here, we'll use JavaScript notation to refer to different interpretations of data. Modern JavaScript is quite good at that. We could refer to the above word using binary *(base-2)* notation, with `0b` prefix:

```
0b11000011101100010110111101110000
```

JavaScript allows us to add underscores for readability, for example, to distinguish bit patterns of individual bytes:

```
0b11000011_10110001_01101111_01110000
```

We can use good old decimal *(base-10)* notation:

```
3_283_185_520
```

We can also use hexadecimal *(base-16)* notation, with `0x` prefix;

```
0xC3_B1_6F_70
```

How do we decide which notation to use? Why would we ever use hexadecimal?

Binary notation is very straightforward: you can see how individual bits are set, and you can visually split a word into bytes, but it is very verbose!

Decimal notation is much terser, you get a good understanding of the magnitude of the number, but it is hard to reason about the values of individual bytes and bits.

Hexadecimal notation is terse, *and* it is easy to split a word into bytes visually. Each hexadecimal digit maps to four bits, no matter the position in a number, so two hexadecimal digits always map to a byte. All you need to remember is bit patterns of the 16 hexadecimal digits:

Hexadecimal	Binary
`0x0`	`0b0000`
`0x1`	`0b0001`
`0x2`	`0b0010`
`0x3`	`0b0011`
`0x4`	`0b0100`
`0x5`	`0b0101`
`0x5`	`0b0110`
`0x7`	`0b0111`
`0x8`	`0b1000`
`0x9`	`0b1001`
`0xA`	`0b1010`
`0xB`	`0b1011`
`0xC`	`0b1100`
`0xD`	`0b1101`
`0xE`	`0b1110`
`0xF`	`0b1111`

This way we can easily translate from hexadecimal to binary and back. Take `0xC3_B1_6F_70`, as an example:

- `0xC3` is `0b1100_0011`
- `0xB1` is `0x1011_0001`
- `0x6F` is `0b0110_1111`
- `0x70` is `0b0111_0000`

Thus we can conclude that `0xC3_B1_6F_70` is the same as:

`0b11000011_10110001_01101111_01110000`

Can't do this with decimal notation!

7.5 Memory

Think of memory as a large continuous byte array. It contains our program instructions encoded as binary words. It contains the data that our program works with: data segment, code segment, stack, and heap (more on these later).

Like a byte array, you can access a single byte from memory given an index into this array. We call this index, a *memory address*. Memory addresses are 32-bit on ARM (how it all aligns, eh?).

However, not only can you access single bytes from memory, you can also access whole 32-bit words. But there is a restriction: you can only access *aligned* words. In this case, *aligned* means non-overlapping words or words which address is divisible by *four*. One word contains four bytes, and each byte has its own address, but we address words only by the address of the first byte in the word.

> **Well, actually…**
>
> Newer ARM processors support unaligned access, but not for all relevant instructions, and it incurs a performance penalty. In this book, we avoid it.

In the following figure, you can see a stretch of memory starting from address `0x00` that shows how individual bytes and their addresses map to aligned words and their addresses.

From here on, we won't need this much detail when talking about memory so that we will use a simplified (but still as precise) word-level diagrams. Like the next one that describes the same stretch of memory.

Sometimes we store a memory address in a memory word. We call that a *pointer*. In our diagrams, we will use arrows to show where a memory word is pointing. We will mostly omit the actual memory addresses in our diagrams since the exact value is not important. The important part is where it points to, not the value itself.

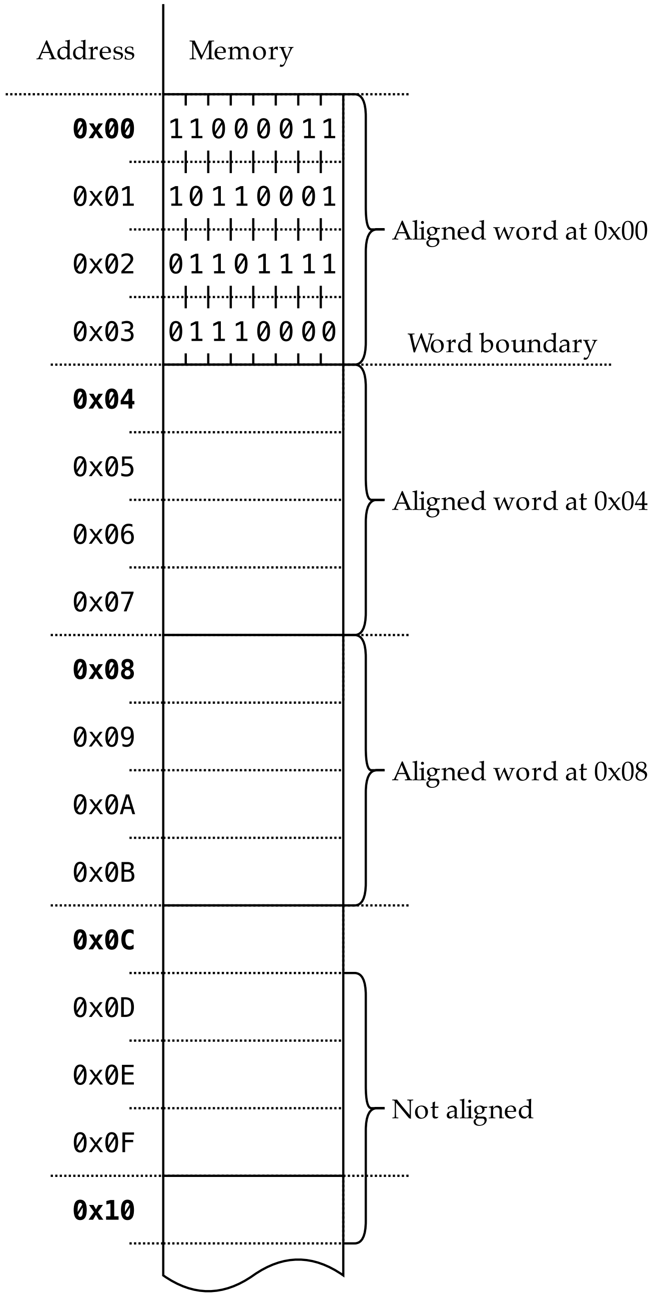

Figure 7.2: An example stretch of memory on byte-level

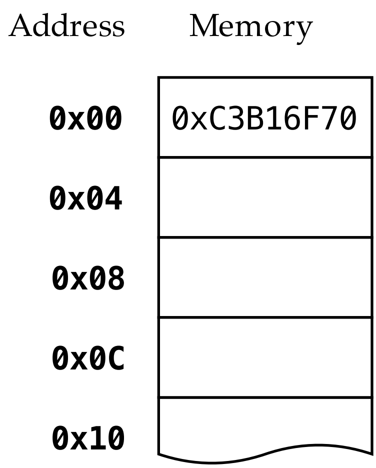

Figure 7.3: Same stretch of memory on word-level

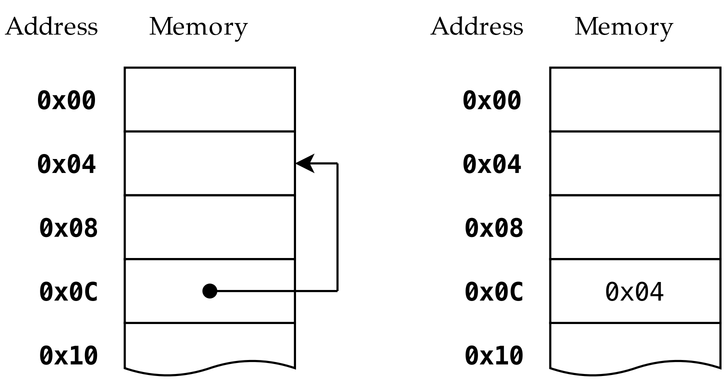

Figure 7.4: Pointer notation: left—with arrows, right—with actual values

As we already mentioned, the memory contains our data segment, code segment, stack, and heap. However, what it does *not* contain (on most architectures, anyway) is *registers*.

7.6 Registers

Registers are special memory cells that are *outside* of the main memory. They are used for intermediate values, sort of like temporary variables. There's a limited number of these—usually 8, 16, or 32.

ARM has 16 main registers and a special *status* register (CPSR). The main registers are called `r0` to `r15`, but some of them have alternative names. See the next figure for more details.

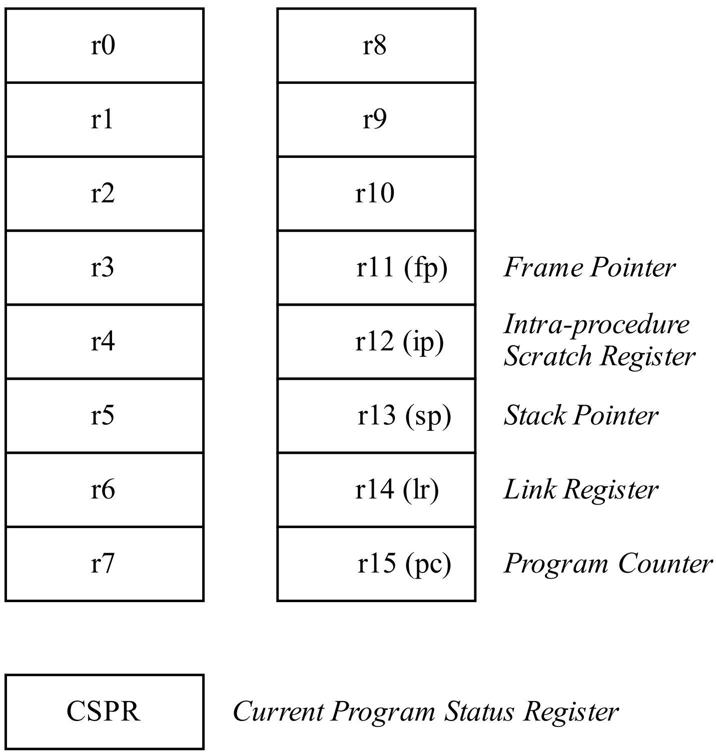

Figure 7.5: Figure about registers

First, why do we need registers? Couldn't instructions work directly with memory? They could, and there are other architectures such as accumulator-based and stack-based architectures that need only one register or no registers at all. However, ARM is a *load-store* architecture.

With *load-store* architecture, the basic workflow is as follows:

- data is loaded from memory into registers, then
- operations are performed on registers, and finally
- the data is stored back into memory.

It turns out this workflow is quite efficient, and most modern architectures follow it.

Most ARM registers are general-purpose and can be used for any intermediate values. We'll cover the more *special-purpose* registers like `fp`, `ip`, `sp`, `lr`, `pc`, and CPSR as we discover instructions that work with them.

You have maybe heard that *registers are fast*. What does that even mean? Why couldn't we use the same technology for memory?

Two reasons.

There are only 16 registers. That means you can encode a register using only 4 bits. At the same time, memory addresses are 32-bit. So you need fewer bytes (and instructions) to encode an operation on three registers, rather than an operation on three memory addresses. And a processor can decode fewer instructions faster.

Second, computer memory has several *levels* of caches, usually referred to as *L1–L3*. And even if the fastest cache uses the same technology as registers, there could still be a *cache miss*. But such a cache miss can never happen in the case of registers.

7.7 The add instruction

Let's get to our first instruction, `add`:

```
add r1, r2, r3     /* r1 = r2 + r3; */
```

It consists of a mnemonic name `add` as well as three register *operands*, `r1`, `r2`, and `r3`. In this case, *operand 1* is `r2`, *operand 2* is `r3`, and `r1` is the result, also called the *destination* operand. This kind of instruction is called three-operand instruction.

As a comment, we provided a pseudo-code that describes the effect of the instruction. Note that the order of operands in the instructions is the same as in the pseudo-code. ARM assembly was designed such that this is always the case.

All ARM instructions are encoded into single 32-bit words in memory. In this figure, you can see how this particular instruction is encoded into binary form. We've left the meaning of some of the bits unexplained.

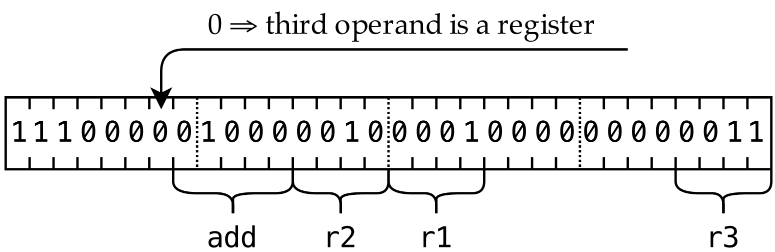

Figure 7.6: The encoding of: `add r1, r2, r3`

> *Well, actually…*
>
> Most ARM processors support several instruction sets. They have been historically called ARM, Thumb, and ARM64, but recently renamed to A32, T32, and A64.
>
> T32 (or Thumb 2), for example, is a variable-length instruction set with both 16-bit and 32-bit instructions.

7.8 Immediate operand

The `add` instruction has a second form, where the *last* operand is a small number encoded directly into the instruction. It is called an *immediate* operand, and the notation uses a # sign:

```
add r1, r2, #64000     /* r1 = r2 + 64000; */
```

GNU Assembler allows familiar syntax for hexadecimal values with `0x` prefix and binary values with `0b` prefix. However, it doesn't allow underscores in them. So, the previous instruction can be rewritten as:

```
add r1, r2, #0xFA00     /* r1 = r2 + 0xFA00; */
```

In the following figure you can see how this instruction is encoded.

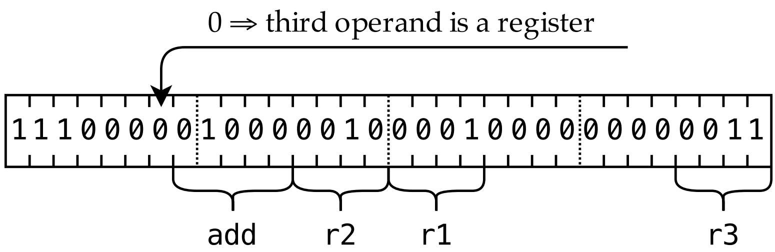

Figure 7.7: The encoding of: `add r1, r2, #0xFA00`

From the figure, you can see that there are 8 bits dedicated to the immediate operand, so you might conclude that it can represent any single byte value. But—wait!—we just used `64000` in our example! It turns out, there are four more bits in the instruction that encode how many *even* number of bits the immediate should be shifted. This way we can represent `0xFA`, or `0xFA0`, or `0xFA00`, or `0xFA000` and so on.

This is an ingenious way to encode a vast amount of interesting constants in a very tight space!

7.9 Signed, unsigned, two's complement

What are we adding with the `add` instruction? Unsigned integers? Signed integers?

It turns out that it works correctly both when all the operands are treated as unsigned integers, and in the case where they are all treated as signed integers. This is thanks to the signed number representation that most computers use, called *two's complement*. It was specifically designed for this trick: to use the same hardware adder for both signed and unsigned numbers.

> **Note**
>
> Even though `add` and most ARM instructions work on 32-bit words, in this section we'll show examples using signed and unsigned 8-bit bytes, to make them more manageable.

Some pages are omitted