

# ARM Instruction Set Architecture

Profs. John McLeod & Arash Reyhani

ECE3375, Winter 2022

This lesson briefly introduces the hardware of an ARM process, and covers the load-modify-store paradigm of Assembly language programming. The differences between machine code and Assembly mnemonics is described, and we start to discuss Assembly language in detail.

## ARM Processors

There are many different microprocessors architectures, and in this course we focus on the ARM Cortex-A9. The textbook discusses the ARM Cortex-M3/M4, which is very similar.

- ARM processors are widely used in personal electronics as well as specialist embedded systems. Over 200 billion ARM chips have been produced as of 2022.
- “ARM” originally stood for “Acorn RISC Machine”, and was first used as the processor in the Acorn Computer in 1985.
- The company later changed the meaning of their name to “*Advanced* RISC Machine”.<sup>1</sup>
- Then the company changed the meaning of their name again and now “ARM” doesn’t stand for anything.
- “RISC” stands for *reduced instruction set computer*, meaning the machine code emphasizes an efficient and minimalist set of instructions.

<sup>1</sup> A classic example of submitting to the anti-oak agenda.

There are three lines of ARM Cortex CPUs: The A series, the R series, and the M series.

1. The **ARM Cortex-A** family are for “applications”. These are high performance processors designed for smartphones and other consumer electronics.
2. The **ARM Cortex-R** family are for “real-time processing”. These are designed to be highly reliable, and are used in specialist high performance, mission-critical situations.

3. The **ARM Cortex-M** family are for “microcontrollers”. These are designed to be cost-effective and energy-efficient. These are often used in embedded systems.

There are also three families of instruction sets for ARM Cortex processors.

1. The **ARM** set of instructions is 32 bits and is the most powerful and complete instruction set. This instruction set is available on Cortex-A processors — the Cortex-A9 uses the **ARMv7** instruction set.
2. The **Thumb** set of instructions is only 16 bits. This is a minimalist instruction set designed for energy efficiency. It is largely obsolete.
3. The **Thumb-2** set of instructions is a mix of 16 bit and 32 bit instructions. This instruction set extends the **Thumb** set by adding some higher performance 32 bit instructions. This instruction set is used by the Cortex-M series.

It is possible to configure the Cortex-A9 to use either Thumb or Thumb-2 instructions, but we won't bother doing that. It is important to note that since the textbook uses the Cortex-M3/M4, it uses Thumb-2 Assembly language. This looks very similar to the ARMv7 language we use in this course, but there are a few differences.

## Processor Architecture

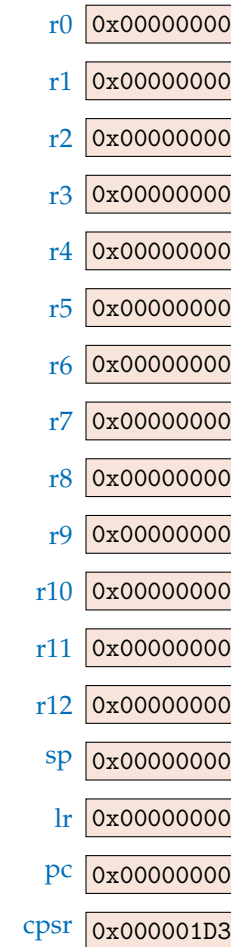
The huge logic circuit that forms a CPU is obviously extremely complicated. We don't need to worry about that. All we need to understand, in this course, is what parts of the processor are available for us to work with. We have already discussed the concept of a **register**; a piece of hardware in the processor that is used to work with data.

- Every processor has several **general-purpose registers** for handling program data.
- The processor also has a **program counter** register for keeping track of the next instruction in the the program.

Sometimes you can write data directly to the **program counter** — i.e., treat it like a **general-purpose register**, but that is often a dumb thing to do because it will completely destroy the flow of your program.

Processors also often have several other special registers. For the ARM Cortex-A9, two useful special registers are the **current program status register** `cpsr`, the **stack pointer**, and the **link register** `lr`.

- The **current program status register** contains all of the **flags** (or “control bits”) describing the current state of the system. In particular, the **carry** and **overflow** flags for arithmetic operations are in this register.
- The **stack pointer** contains the address of the top of the **stack**. The **stack** is a flexible memory structure used to store tempo-

A vertical list of 17 registers for the ARM Cortex-A9 processor. Each register is represented by a label on the left and a value in a box on the right. The registers are r0 through r12, sp, lr, pc, and cpsr. r0 through r12 all contain 0x00000000. sp, lr, and pc also contain 0x00000000. cpsr contains 0x000001D3.

r0	0x00000000
r1	0x00000000
r2	0x00000000
r3	0x00000000
r4	0x00000000
r5	0x00000000
r6	0x00000000
r7	0x00000000
r8	0x00000000
r9	0x00000000
r10	0x00000000
r11	0x00000000
r12	0x00000000
sp	0x00000000
lr	0x00000000
pc	0x00000000
cpsr	0x000001D3

Figure 1: Main registers in the ARM Cortex-A9 processor. There are some others that are less-important.

rary data.

- The **link register** is used to hold the next *sequential* instruction's address when the program is **branching** to somewhere else in the code. This is used to *return from a subroutine*.

We will discuss all of these in greater detail later.

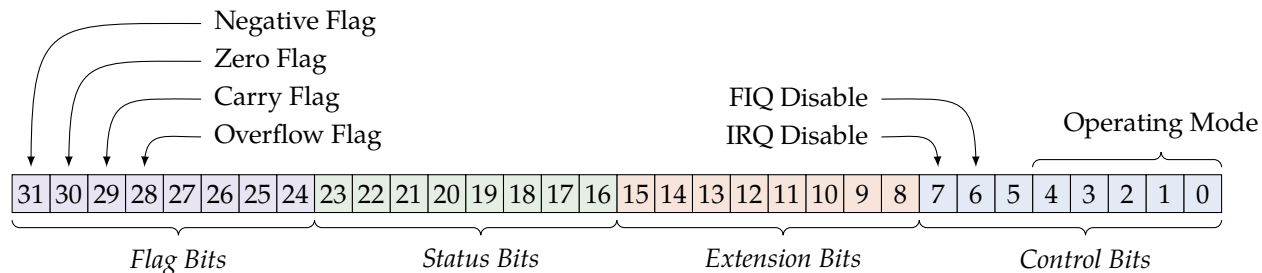


Figure 2: Structure of the ARM®Cortex-A **cpsr** and **spsr**, with some notable bits highlighted.

## Load-Modify-Store

Manipulating data is really what programming is all about. In higher level programming languages we are used to writing code to directly interact with the data, but on a hardware level that is not what is really happening.

- The CPU can only manipulate data stored in a register.
- The CPU has a limited number of registers.

In higher level programming languages we often declare new variables whenever we need a new “container” for data; but we can’t

declare a new register! The registers that already exist in the CPU are all we'll ever have access to.

This is why the **load-modify-store** paradigm is an important concept to understand in Assembly language programming. Any time we want to manipulate data, we have to perform three steps:

1. **Load** the data from memory into a suitable register.
2. **Modify** that data using whatever arithmetic or logic operations are appropriate.
3. **Store** that data back into memory at a suitable address.

The last step is very important, because it *frees up a register* for other uses.

- Beginner programmers in Assembly language often start to *run out of registers*, because **r0** is a loop counter, and **r1** is a hardware address, and ...
- If a register contains information that is important, but *not immediately useful*, that information should be written to memory so the register can be used for something else.

## Machine Code and Mnemonics

The **opcodes** that a microprocessor understands are a set of pre-defined binary codes that often occupy the most-significant bits of a **word**, while the least-significant bits are used for context-dependent information (such as addresses or data values).

- For example, to initialize a register in the microprocessor with a small number, the opcode 0xE3A01005 tells the microprocessor to *initialize register **r1** with the value 5* (in hexadecimal).
- The first four hex digits (0xE3A0) identify the instruction (“initialize a register”). There is a hard-coded table of these instruction codes in the logic circuit of the processor.
- The next hex digit (0x1) identifies the destination register (in this case,  $r_1$ ). The connection between these register identifiers and the physical registers on the chip is hardwired.
- The final three hex digits (0x005) identify the data to be written to the register (in this case, the number 5). This obviously cannot be hard-coded, but it is part of the opcode.
- In this sense, the number 5 is a constant, so it arguably does not count as data, and when the microcontroller is programmed this entire opcode will be written into ROM.

Programming with opcodes directly can be done, but is extremely difficult. Instead, we use the slightly-less-difficult method of coding in **mnemonics**.

*Definition:* A **mnemonic** is a “human-readable-esque” sequence of letters that has a one-to-one mapping with an opcode.

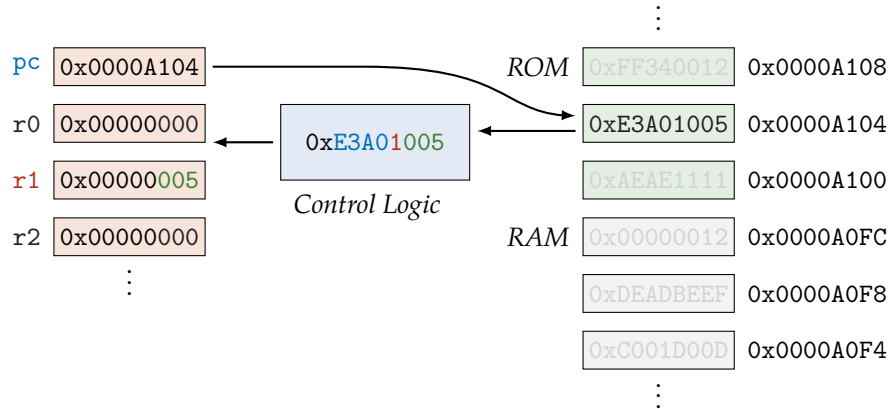


Figure 3: A sketch of the process within a microprocessor when the opcode `0xE3A01005` is executed. A special register called the *program counter* keeps track of where to find the next instruction. This is fetched from ROM and parsed through the control logic, where the opcode is interpreted and the value `0x5` is placed in register `r1`. The program counter will then increment and the next opcode (apparently, `0xFF340012`) will be fetched. Note that the memory addresses shown here increment by 4 (bytes), as each is schematically shown as containing a 32-bit number. In the actual device, each 32-bit number would be spread across 4 adjacent 1-byte memory cells.

So instead of writing `0xE3A01005` in order to perform the operation of *initializing register `r1` with the value 5*, we would write:

```
mov r1, #5
```

This mnemonic parses exactly the same way as the opcode: the compiler interprets `mov` as `0xE3A0`, the register name `r1` as `0x1`, and `#5` as `0x005`.<sup>2</sup> This process is schematically shown in Figure 3.

<sup>2</sup> Although it is not always the case that a mnemonic has a *sequential* mapping like this example — but it is always the case that a mnemonic has *some* direct mapping to a single opcode.



## Assembly Language and Microcontrollers

Assembly language is intimately related to the opcodes of a microprocessor, and consequently (and unfortunately for students!) there are many variants of assembly language, each unique to a particular system: the opcode 0xE3A01005 for an ARMv7-type microprocessor (like the ARM®Cortex-A9) won't have the same effect on an Intel x86\_64 or a Motorola 68k system.

- There is usually some common ground with the mnemonics: although `mov r1, #5` won't work on an Intel x86\_64 system, it is very close to the correct code.<sup>3</sup>
- Still, you can't expect code written for one microprocessor architecture to successfully compile and run on a different architecture.

Although each microcontroller has its own particular set of attributes — they are unique snowflakes, just like you<sup>4</sup> — they all have common components. Every microprocessor has:

1. A set of **registers** for holding data and addresses that are being manipulated by the program. The ARM®Cortex-A9 has thirteen 32-bit general-purpose registers, called `r0`, `r1`, ..., `r12`.
2. A special **program counter** register that contains the address in ROM of the next instruction. This is called `pc` in the ARM®Cortex-A9.
3. One or more special **program status registers** that contain the control bits returned from various operations. These generally cannot be accessed directly by user-written code. These are

<sup>3</sup> In fact, just `mov r1, 5` should work — drop the “#” in front of the decimal literal.

<sup>4</sup> And using a C-to-Assembler compiler to club these unique microprocessors into running the same standard program is just like how the education system forces conformity on you! (Insert unhinged rant about modern society here...)

called `spsr` and `cpsr` in the ARM®Cortex-A9.

4. A **link register** that contains an address in ROM. This is used for coding simple branch-and-return loops. This is called `lr` in the The ARM®Cortex-A9.
5. A **stack pointer** register that holds the address of the stack (in RAM). This is used for more complex or nested loops. This is called `sp` in the ARM®Cortex-A9.

Assembly language generally consists of lines of code of the form:

*label*: *mnemonic* *operand1* [, *operand2*, *operand3*] [*@ comment*]

The concept of a mnemonic was already discussed, and most of the remaining code is probably self-explanatory. But anyway:

- Inline comments can be written as *@ comment*, or over several lines as */\* long, multiline comment \*/*. Of course comments are always optional, but often a good idea — especially as assembly is so hard to read.
- Labels are used for the assembler equivalent of GOTO commands. While this type of coding is discouraged in higher-level programming,<sup>5</sup> it is common in assembly. Labels can be alphanumeric, and are optional.
- Operands are generally registers, or literal numbers (in decimal or hex) or addresses (always in hex). For some commands a label may also be used as an operand. At least one operand is needed for every mnemonic, but some require more.

<sup>5</sup> Or even made impossible by not having a GOTO command.

Assembly language programs also using include special code called

*directives*. These are instructions for the compiler, rather than the microprocessor.

- A commonly-used directive in ARMv7, for example, is *.data*, used to define a section of the program that provides a set of numbers that should be pre-loaded into the microcontroller's memory.

With this in mind, we will now look at the types of instructions used in assembly language, and provide specifics for ARMv7 assembly language. The basic categories of assembly language instructions are:

1. Data movement instructions.
2. Data manipulation instructions.
3. Conditionals and test instructions.
4. Branch instructions.
5. Subroutine instructions.
6. Interrupt instructions.

The last three categories will be examined in a future lesson.