Workshop 10: ARM Assembly

Manchester Raspberry Jam

Contents

0 Introduction

This workshop takes a very quick look into the ARM assembly language, how programming in assembly differs to more typical programming languages, and how the assembly code we write is relating to the Raspberry Pi hardware.

This workshop is recommended for people with experience in a regular programming language such as Python.

These booklets were created using LaTeX, an advanced typesetting system used for several sorts of books, academic reports and letters. Thanks to the Fribourg Open Source Community for their ARM language definition for Latex's listings package.

What you'll need

Not much! programming in assembly requires no additional software, everything we need is bundled in with Raspbian. Here's what we'll be using:

- A Text editor
  At the Jam, we'll be using VIM, but any will do. If you'd like to try using VIM, install by typing the following command into the terminal:

```
$ sudo apt-get install vim vim-gnome
```

- An Assembler
  We'll be using the GNU portable assembler. This is pre-installed with Raspbian.
- The files for this workshop from our Google Drive:
  Go to `bit.ly/mcrraspjam`, then navigate to 'Workshops > 010: ARM Assembly'.

Code listings & asides

When you need to make changes to your code, they'll be presented in boxes like the following:

```
1      .global _start
2  _start:
3      @set a register
4      LDR R0, =string
```

You might not need to copy everything, so check the line numbers to make sure you're not copying something twice.

Terminal commands are listed as such, copy everything *after* the dollar sign. Lines without dollar signs are example outputs, and do not need to be copied.

```
$ ./addition.s
$ echo $?
102
```

Further Reading

Parts of this workshop are based on *"Raspberry Pi Assembly Language Raspbian" [Third Edition]* by Bruce Smith.

Questions?

We're just covering the basics of ARM assembly today. If you enjoy it, and want more ARM workshops, let me know!

Send any corrections, suggestions or questions to: `jam@jackjkelly.com`

1 Quick Start

We won't be starting with "Hello, World!" today, as when we're programming at a low level, we're more interested in numbers than text.

Printing text takes an extra step, which we'll try later. For now, let's start by moving some numbers about, to see how a very simple program works.

1_register.s

If you copied the workshop files to your Raspberry Pi, **open** the text file titled *'1_register.s'*. Otherwise, copy out the program below:

```
    .global _start
_start:
```

These two lines tell the computer to start below the line '_start: '. This will be the start of every program we write.

Below this line, we will write a basic program. **Copy** out the code below. Indentation and capitalisation will help you read your code, but aren't necessary.

```
    @set a register
    MOV R0, #65

    @exit
    MOV R7, #1
    SWI 0
```

Let's run the program, then we can take a look at what we just created.

Assembling your program

Navigate to your .s file from within a terminal. (use *ls* and *cd* commands) Once found, assemble your program by **typing** the following command

```
$ as -o 1_register.o 1_register.s
$ ld -o 1_register 1_register.o
```

This assembles and 'links' the program we just wrote, and creates a binary executable. To run this, we can **type**:

```
$ ./1_register
```

It would seem that nothing has happened. In fact, because there are no 'print' statements in this program, it has completed without providing us any output. we can get our number 65 back, by **typing** in the console

```
$ echo $?
65
```

This is an OS short cut, which prints the last value of R0, which should be the 65 from our program. Let's now figure out what our program just did.

2 ARM assembly basics

Let's break down the first line we wrote:

```
    MOV R0, #65
```

The operations we perform in assembly programs are all done using **instructions**. Apart from a couple of exceptions, instructions directly use one of the mathematical or logical functions of the Raspberry Pi CPU.

Each instruction consists of a 3 to 5-letter **mnemonic**, as well as a number of **registers**, the fast memory locations we use in assembly programming.

MOV is an instruction that is used to move values between registers. A value is moved from the second register to the first.

In this case, we moved the number 65 into register 0. We could have used another register in place of 65, but in this case the # symbol denotes that we are using a numeric value, and we call this a **literal**.

Adding more instructions

let's add some more instructions to our program. the instruction ADD will perform an addition of two register locations. the instruction is used as follows:

ADD <Output>, <Input 1>, <Input 2>

Let's modify our program so that there are two registers with numbers held in them.

```
    @set two registers
    MOV R0, #65
    MOV R1, #37
```

now that we have our two inputs, we can add our addition instruction to the code.

```

    @store addition in R0
    ADD R0, R0, R1
```

Assemble and link the program, as we did before, then re-run the program. When echoed, the output should read 102, the sum of our R0 and R1 values.

```
$ echo $?
102
```

3 The Raspberry Pi Hardware

Registers

It was mentioned previously that registers are fast memory locations that we perform operations on. There are only a few registers, which we need, because the computer's RAM is far too slow to keep up with the CPUs operation speed.

There are 17 registers, each of which stores a numeric value. R0-R12 are available to be modified by our programs, but R13, R14, R15 are reserved for special purposes.

To ensure the registers are running at fast speeds, they are contained within the CPU hardware.

The CPU

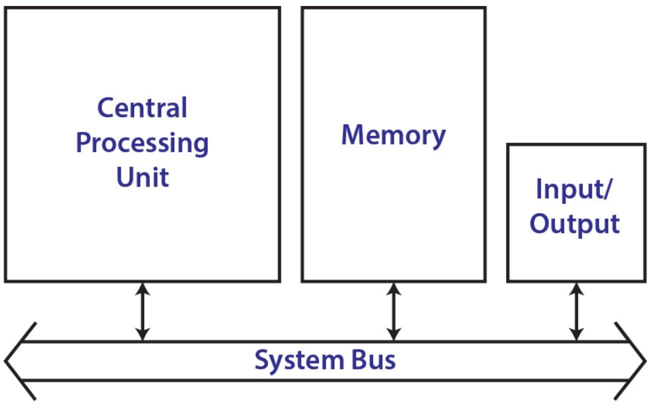

Figure 1: Single Bus Model of Computer Architecture

Figure 1 shows the traditional model of computer hardware. Data is moved from memory or external I/O devices into the CPU register bank, where it is operated upon, and later returned.

In Assembly, we're mostly concerned with the CPU, with little attention on the memory or I/O devices. Let's fill in some detail into the CPU part of this diagram.

There are 3 main components to a CPU:

1. The **register bank**, that holds data between CPU cycles.
2. The hardware that performs mathematical data on our register values. We call this collected hardware the Arithmetic Logic Unit (**ALU**)
3. The **control** logic that connects these components. For each instruction, the control sends the data to different parts of the ALU, then redirects it back to the register bank.

These components could be illustrated as shown in figure 2 below.

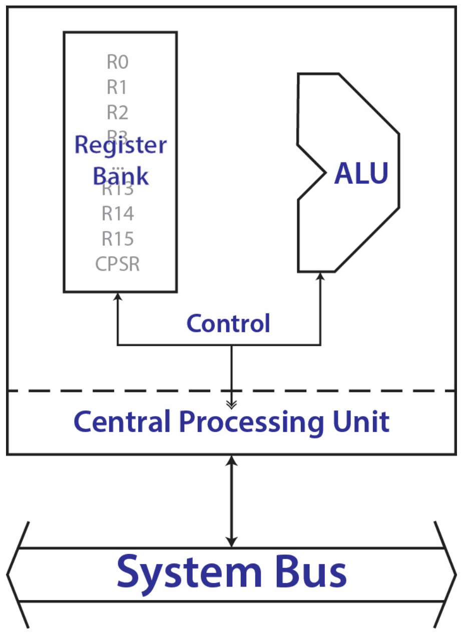

Figure 2: The basic internal components of a CPU, at a very high level.

Executing Assembly Instructions

The majority of assembly instructions are directly interpreted by the control hardware in the CPU, and each one performs a different ALU action on a number of register inputs.

An assembler simply translates this instructions from textual code into their binary instruction equivalents. The assembler will also perform some more advanced functions on some non-instruction code, such as labels and psuedo instructions.

4 Loading from memory

Let's learn how to load data from memory. We'll need to learn how to do this in order to complete our "Hello, World!" program in the next section. For now, we'll just load a single number.

3_memory.s

We'll start with another blank file. Find and **open** *3_memory.s*

```
    .global _start
_start:
```

We can store a value in memory by adding it to our assembly file, like written below.

In a regular programming language, this is like a variable called 'value'. 'word' states that this is a 32-bit number, which is the size of each register location. **Copy** this, leaving it at the end of the file.

```
value:
    .word 72
```

Remember that memory is like a long list of values. Each memory location contains a value, as well as a unique address that allows that location to be found and accessed.

To load from memory, we will need to first get the address. The instruction **ADR** will find the address of our memory location:

```
    @load the memory address
    ADR R1, value
```

We can now load the number from memory. The **LDR** instruction loads a memory value from a given address; the square brackets around R1 show that this register contains an address, not a number.

```

    @load the memory value
    LDR R0, [R1]
```

Our number should now be in R0. We will end this program as before.

```

    @exit
    MOV R7, #1
    SWI 0
```

Assemble, link and run the program as before. The output from echo should be:

```
$ echo $?
72
```

5 "Hello, World!"

printing *value*

Sticking with the memory program for now, let's try outputting our loaded memory value as text in the terminal.

We'll be using an instruction called 'software interrupt' (**SWI**) to print text. This instruction will call to an external program within the operating system, which will print text for us.

Because of the complexity of the external program, we need to give it parameters by setting the following registers

Register	Purpose	Value Used
R7	Program to Run	4 (print)
R0	Output Stream	1 (terminal)
R2	String Length	1
R1	String Address	value

Figure 3: Printing parameters

As R0 is used by SWI, we will load our value to R1 instead of R0. We will set all of these values as such:

```
    @load the character address
    ADR R1, value

    @print
    MOV R7, #4
    MOV R0, #1
    MOV R2, #1
    SWI 0
```

note that we no longer need to load our value, as we only require the address for the SWI call.

Try assembling, linking and running the program again. The output is a little odd, can you think why this is?

```
$ ./4_print.s
H $
```

Printing "Hello, World!"

We could load each letter in turn, like our current program, but thankfully the assembler contains some useful short cuts for writing strings.

Let's remove value at the end of our file, and add our hello world string instead.

```
string:
    .ascii "Hello, World!\n"
```

When your program is assembled, this string will be turned into a succession of ASCII numbers, just as we need them.

Now we just need to change a couple of things. Make sure ADR is loading string, not value, and make sure that, as stated in figure 3, the length of the string is correct. (14 for me. Spaces count, \n is 1 ASCII character)

Try assembling, linking and running the program again. With any luck, you've finally reached "Hello, World!" in assembly.

```
$ ./5_helloworld.s
Hello, World!
$
```

6 Completed Code Listings

1_register.s

```
    .global _start
_start:
    @set a register
    MOV R0, #65

    @exit
    MOV R7, #1
    SWI 0
```

2_addition.s

```
    .global _start
_start:
    @set two registers
    MOV R0, #65
    MOV R1, #37

    @store addition in R0
    ADD R0, R0, R1

    @exit
    MOV R7, #1
    SWI 0
```

3_memory.s

```
    .global _start
_start:
    @load the memory address
    ADR R1, value

    @load the memory value
    LDR R0, [R1]

    @exit
    MOV R7, #1
    SWI 0

value:
    .word 72
```

4_print.s

```
    .global _start
_start:
    @load the character address
    ADR R1, value

    @print
    MOV R7, #4
    MOV R0, #1
    MOV R2, #1
    SWI 0


    @exit
    MOV R7, #1
    SWI 0

value:
    .word 72
```

5_helloworld.s

```
    .global _start
_start:
    @load the character address
    ADR R1, string

    @print
    MOV R7, #4
    MOV R0, #1
    MOV R2, #14
    SWI 0


    @exit
    MOV R7, #1
    SWI 0

string:
    .ascii "Hello, World!\n"
```