Introduction to ARM Assembly

Anas Shrinah

November 26, 2021

We have learnt how a computer is built from the bottom up, how the logic can be implemented in hardware. We also tested simple assembly codes with our 4-bit CPU. Now is the time to move on to a more advanced CPU. For the next step of the unit, we will write assembly programs using the ARM instruction set. To test these codes, we need an ARM processor. If you have a computer with an ARM-based processor, then you can start coding in RAM assembly straightaway. However, if you do not have access to a machine with an ARM processor, you can use an ARM emulator.

Credits: The content of this worksheet is based on lab materials originally prepared and developed by previous lecturers of COMSM1302 at the University of Bristol.

Goals of this lab

In this lab:

- You learn how to use the GNU tool chain to execute and debug assembly programs.
- You will learn the main directives in ARM assembly codes.
- You will get the chance to practice what you have learnt in the lectures about the ARM instruction set.
- You will get the opportunity to look at your code with the eye of a verification engineer.
- In summary, this lab will equip you with the essential practical skills to write ARM assembly programs.

It is recommended that you work in small informal lab groups, e.g. with two to three students located close to you in the lab. Please note, however, that each of you needs to develop a good understanding of the material so that you can perform the tasks on your own once you have worked through the lab sheets.

Questions

If you have any questions, please ask. Make use of the TAs, but do not expect them to give you complete designs, they are there to guide you, not do the work for you. Please be patient. You may need several attempts to get something working.

When requesting help, TAs will expect you to show them what you have done so far (no matter how sketchy) and they will ask you to clearly explain your reasoning. This makes it easier for the TAs to help you. For this reason, priority will be given to students who can show and explain how far they have got.

1 Setting-up the lab environment

We will run code through an emulator called QEMU. It is an open-source processor emulator that implements lots of different architectures. This emulator is already installed on the lab machines, but if you prefer to run it on your computer, you should install it by yourself.

1.1 Setting-up the environment to run the ARM emulator using a lab computer - locally

In order to include the programs that we will need, you are going to have to load a module that will include the binaries that we need. Execute the following commands from the terminal.

```
module use /eda/cadence/modules
module load course/COMSM1302
```

You will have to load these modules every time you want to work on this material, as they are removed upon logging out. If you want the modules to be always included, you have to add them to your .bashrc file

```
echo 'module use /eda/cadence/modules' >> ~/.bashrc
echo 'module load course/COMSM1302' >> ~/.bashrc
```

Do not forget to reload the current environment.

```
source ~/.bashrc
```

1.2 Accessing the lab machines remotely

To remotely access a computer in the Linux lab, execute the following commands from your terminal.

```
ssh "your user id"@seis.bris.ac.uk
ssh rd-mvb-linuxlab.bristol.ac.uk
```

Replace the "your user-id" with your user id. Your university user id is of the format ab12345. Once you have accessed a lab machine, you have to the steps in Section 1.1 to set up the lab environment.

1.3 Installing ARM emulator on a personal computer

If you do not want to run the ARM emulator on your personal computer, then you can skip this step. If you want to run the code on your own computer, you have to set up the cross-development environment yourself. Below you can find instructions for how to do this. The instructions assume that you are using some form of the Linux operating system. As the computer that you are using is most likely not an ARM-based machine, you need to set up a different environment to be able to create and run executable files. For Debian derivatives (such as Ubuntu), execute the following command to install QEMU.

```
sudo apt-get install qemu
```

The next thing we need is a cross-compilation environment. What this means is that we want to use the local machine to generate executable binaries for a different architecture. What we need are the tools that support ARM. If you want to install these on your own machine, these packages should be available in your package manager.

```
sudo apt-get install binutils-arm-none-eabi gcc-arm-none-eabi
```

Now the way we are going to interact with the program is through GDB which is the GNU debugger. We need a version of GDB that understands ARM architecture. In Debian there is a specific package for ARM that can be installed, other flavours including Ubuntu have a gdb-multiarch that you can install.

```
sudo apt-get install gdb-arm-none-eabi # Debian
sudo apt-get install gdb-multiarch # Ubuntu
```

Now the way we are going to interact with the program is through GDB, which is the GNU debugger. We need a version of GDB that understands ARM architecture. In Debian, there is a specific package for ARM that can be installed. Other flavours, including Ubuntu, have a gdb-multiarch that you can install.

2 GNU Tool chain

In this lab, we will use the GNU tools to develop our programs. The GNU Toolchain contains development tools for a large range of different architectures and programming languages. You have already seen GCC in the Programming in C unit, and now we will introduce a couple of more tools. The great thing about the tools is that as they exist for many platforms and are open source. So, you can feel safe that anything that you learn here will be applicable elsewhere.

2.1 Our first ARM assembly program

Fire up your favourite editor, and we will write our first program. Let's not worry too much about what this actually does. We just need something to show how each of the steps from code to running an executable works.

```
.section .text
.align 2
.global _start
_start: mov r0,#-1
        mov r1,#1
```

First, we will look at a few directives that we can pass to the assembler. Directives are instructions that the assembler and linker will use when it creates the binary. In the code above there are three directives, .section, .align, and .global. These directives have the following interpretation:

- .section tells the assembler that what follows is a .text for code or .data for data. Each section can be placed in different locations in memory.
- .align 2 indicates that the following section should be aligned on $2^2 = 4$ byte boundaries in memory.
- .global indicates that the label _start should be visible outside this file. This is a directive that is used when the linker combines several different objective files in order to create a single binary.

We will see more directives in the next week. Save the code as tst.s. Now we will convert the code to machine language using the assembler.

2.2 Assembler and Linker

The assembler included in the GNU toolchain is referred to as "as". In order to create executable files, we will use a two-stage process.

- First, we will generate the machine code corresponding to the assembler instructions we have written. The output of this will be an object file.
- Once we have the object file, we will link this file using the GNU linker "ld" to create the executable. The linker is a very powerful command. The linker allows you to combine several different programs together into a single executable.

Simply invoking the command "as" from the terminal will cause the assembler to assemble your code for the hardware in your local machine. As we want to assemble for a different architecture, we need to use the following commands instead.

```
arm-none-eabi-as -o tst.o -g tst.s # create the object file tst.o
arm-none-eabi-ld -o tst tst.o # create the executable tst
```

We give the assembler the flag "g" so that we attach debug information that we will use later.

2.3 QEMU

Now we have an ARM executable, and it's time to run the code inside the emulator. The whole purpose of this lab is to focus on and understand the execution of the code, so we are going to execute it in a special manner. We, therefore, want to start the executable but make it controlled from the outside. We do so by starting qemu-arm, which is the emulator. We give it the commands singlestep and g with parameter 1234. The last parameter of this command indicates that we will later attach a debugger to port 1234, where qemu will output information.

```
qemu-arm -singlestep -g 1234 tst &
```

By providing the & sign to the terminal, we tell the process that we are starting to fork from the current terminal. The process is still running in the background, which you can see by executing ps and filtering out the relevant processes using grep.

```
ps -e | grep qemu
```

Now the program is running, and it's time for us to investigate what it is actually doing.

2.4 GDB

In order to follow the execution of the program, we are going to use GDB, which is the debugger in the GNU toolchain. A debugger is basically a program that allows you to watch the execution of codes. We will now start gdb and make it listen to the port that qemu outputs over. We can do this by executing the following commands.

```
arm-none-eabi-gdb
# you will now get a prompt for GDB
(gdb)
```

Use the command "file" followed by the name of the ARM executable file "tst".

```
(gdb) file tst # tell GBD which file we are going to use
```

Then execute this command:

```
(gdb) target remote localhost:1234 # tell GDB where the program is running
```

Now we can interact with the program that is running inside qemu. We can use the command print to view what the value of a register is, for example, "print $r7" should print the value of the register r7.

```
(gdb) print $r7
$1 = 0
```

The result might be something different as we have not started the program, so it is not guaranteed what the value of this register actually is. We can also look at what the value of the program counter is by writing:

```
(gbd) print $pc
$2 = (void (*)()) 0x8000 <_start>
```

This means that our program counter is currently on address 0x8000, which is a memory address we have given the label _start. We will now move the program forward one step in memory. We can do so by writing "si", which stands for step instruction. This instruction will move the program counter forward one step.

```
(gdb) si
```

To verify the effect of this command, print the value of the program counter again.

```
(gdb) print $pc
$3 = (void (*)()) 0x8004 <_start+4>
```

Now we are four addresses further in our code and have executed line number 5. The reason that the program counter has stepped 4 addresses further in memory is that each address is a single byte in memory and each instruction in ARM is 32 bits long, i.e. 4 bytes. So after executing line 5, the CPU will increase the program counter by 4, and be ready to execute line 6. We can also display all the registers by writing info registers. Now execute the command "c", which stands for continue.

```
(gdb) c
    Continuing.
Program recieved signal SIGILL, Illegal instruction,
0x00008054 in ?? ()
(gdb)
```

By executing the command "c", we just let the computer run and does its loop of fetch, decode and execute. Now, as we have no idea of what is in the memory after the instruction mov r1, #1, we do not actually know what the program will do. It is simply reading random stuff in memory, interpreting them as instructions and executing them. Eventually, it will find a bit pattern that doesn't correspond to an instruction leading it to break. We can fix this by adding a simple infinite loop after our program using this bit of code.

```
.section text
.align 2
.global _start
_start: mov r0,#-1
        mov r1,#1
_end:   b _end
```

Add a label _end in the line after the last instruction and at that place, add an unconditional branch instruction that will set the program counter to the address of _end when it reaches it. Therefore, creating an infinite loop. Try to assemble the program above and execute command "c" in gdb to make sure that you no longer end with an illegal instruction. The program will now just be stuck in an infinite loop, and you will need to send it a kill signal through GDB using Ctrl-C.

Some times GDB does not terminate QEMU. In this cases use the following command to kill the qemu process.

```
kill -9 "qemu-process-id"
```

Where "qemu-process-id" is the qemu process id we can get by executing this command:

```
ps -e | grep qemu
```

2.4.1 GDB commands

Here are a few useful commands that are worth knowing.

1. **until <linenum>** runs the code until the PC reaches the address in memory corresponding to the line number specified by <linenum>.
2. **break <linenum>** sets a breakpoint at the line number specified by <linenum>. This breakpoint will make the execution stop when it reaches this line.
3. **break <label>** sets a breakpoint at the label. For example, you might want to stop the code when it reaches _end in the code above.

4. **step <N>** steps N instructions forward.

To display the internals of the hardware these commands are useful to know:

1. **print <registers>** will print the content of the register.
2. **info reg** will print the content of all integer registers.
3. **list** will show 10 lines of code close to what we are currently executing.
4. **list <linenum>** will list the code around the line number.
5. **x <address>** will display the content of memory address.

Now we have the framework up and running, and it's time for us to move on to actually write some more interesting codes. Do make sure that you understand how the procedure works as it is a bit tricky. Remember that we are running codes on a machine that has no operating system. The emulator starts executing codes from the address pointed to by the program counter, and that is it.

3 Programs

Now we have gotten the hang of how the execution works and can run a program. Next, we are going to write assembly codes to solve the following problems.

3.1 The determinant of a 2 * 2 matrix

Write a program to compute the determinant of a 2×2 matrix.

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = a * d - b * c$$

1. Load the registers r0,r1,r2, and r3 with the values of a,b,c, and d respectively.
2. Calculate $a * d$
3. Calculate $b * c$
4. Calculate $a * d - b * c$
5. Add an infinite loop at the end of your code.

Suggest some 2 * 2 matrices to test your code. It is usually a good idea to test your code with border cases.
To determine the border cases of your code answer the following questions:

1. Are we using signed or unsigned representation?
2. What is the range of the values of the variables a,b,c, and d?
3. What is the range of the determinate that we can calculate?
4. What is the range of the register that will hold the result of the intermediate calculation $a * d$?
5. Can we guarantee the result of $a * d$ or $b * c$ can fit in 32-bit registers? Why?
6. Give an example of two inputs of a and b that would cause $a * d$ to not fit in a 32-bit register.

In this example, we assume it is the user responsibility to understand and use our code properly. In the extra task, you will be asked to automate the process of checking the correctness of the result of your code.

3.2 The 3n+1 problem

The 3n+1 problem is a mathematical open problem. Given a positive integer n, if you repeatedly do the following:

1. If n is even, replace it by n/2.
2. If n is odd, replace it by 3n+1.

For every positive integer tried so far, this algorithm eventually reaches n = 1, at which point it cycles 1 - 4 - 2 - 1 forever. The open problem is whether this property holds for all positive integers. We will not be able to answer this question, but your assignment is the following:

Write an ARM assembly program to runs the 3n+1 procedure above. Your program should halt if the value 1 is reached. You can assume that we will be working with small enough values that there will be no integer overflow.

Please note that you do not need to use division or multiplication instructions for this code.

3.3 Great Common Divisor - Euclidian Algorithm

The greatest common divisor of two positive integers a and b is the largest divisor common to a and b. For example, GSD(3,7) = 1, GSD(2,60) = 2, and GSD(60,75) = 15.

A nice and algorithmic way of calculating GCD is the Euclidian Algorithm. For more information about this algorithm check this video https://www.youtube.com/watch?v=JUzYl1TYMcU.

Your task is to research this method and write an assembly code to calculate the GCD for two positive numbers. First, move the value of the first number to r0 and the value of the second number to r1. The code should store the GCD of r0 and r1 in r2 before it goes to an infinite loop (halt).

4 Extra task

Please only attempt these tasks when you have done all other sections.

4.1 Improve the determinant of a 2 * 2 matrix code to make it sound

In Section 3.1, we assumed it is the responsibility of the end-user to use our program wisely. But the truth is, as programmers, it is our responsibility to specify the behaviour of our code. We have two options here.

- List down all the ranges and combinations of the values for which we guarantee our code to produce correct results, or
- give a warning to the user when we are not sure that our program will return the correct answer for the given inputs.

Think about the ranges and limits where the calculation $a * d$, $b * c$ and $a * d - b * c$ can go wrong because of using 32-bit registers.

Update your code from Section 3.1 to give a warning to the user when the inputs might cause the result to be invalid. Use r8 as a warning flag. Put 1 in r8 if the result of the calculation is correct. Put -1 in r8 if the result is wrong.

Congratulations **you are officially an ARM assembly programmer now!**