Introduction to ARM Assembly

Carl Henrik Ek

Week 7

Abstract

We have now learnt how a computer is built up from the bottom up, how the logic can be implemented in hardware. Now for the next step of the unit we will provide instructions (program) to the hardware. Importantly we will do so in what is called bare metal manner which means that we have no operating system to work around, its just us and the logic. To do so we will use Raspberry Pi computers and mainly we will look at the CPU of these which are from the ARM family. As we do not have the possibility to connect external machines in the lab we will work with them in emulated form. Importantly if you have a RPi or are interested in buying one you can run the code on the real hardware.

We will first describe how you can set-up the environment that is needed in order to run the code that we will develop. 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.

Code

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

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

1 Personal Computer

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 on how to do this. The instructions assumes that you are using some form of *nix operating system.

As the computer that you are most likely using is not a `ARM` based machine we need to set-up a different environment to be able to create and run executable files. We will run code through an emulator called QEMU. It is an open-source processor emulator that implements lots of different architectures. This should already be installed on the lab machines but if you want to run it on your laptop it should be contained in most package managers. For Debian derivatives (such as Ubuntu) a simple,

Code

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

should do the trick to get it installed.

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 supports ARM. If you want to install these on your own machine these packages should be available in your package manager.

Code

```
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. Its a fantastic piece of software that will make your life as a programmer a lot easier. Now 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.

Code

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

Another resource is to compile the package yourself. It doesn't have a UI so the requirements are not too complicated. If you go to URL you can download the latest GDB and follow the instructions how to compile it.

Now when we have everything set-up, or if you are using the lab machines we should be able to write our first program, assemble it and then run it. We will walk through each of the steps one by one quite quickly initially just to make sure you can get a program running.

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 exists for many platforms and are open source you can feel safe that anything that you learn here will be applicable elsewhere.

2.1 Our First program

Fire up your favourite editor[1] and we will write our first program. Lets not worry too much about what this actually does. Its just that we need something to show how each of the steps from code to running an executable works.

Code

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

Save the code as tst.s. Now we will convert the code to machine language using the assembler.

2.2 The Assembler

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 that allows you to combine several different programs together into a single executable. Simply invoking as will use the assembler for the hardware in your local machine. As we want to assemble for a different architecture we therefore need to use the following commands instead.

[1] emacs

Code

```
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 executble tst
```

We give the assembler the flag `g` so that we attache debug information that we will use later.

2.3 QEMU

Now we an ARM executable and its time to run the code inside the emulator. Now 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 indicates that we will later attach a debugger to port `1234` where `qemu` will output information.

Code

```
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 filter out the relevant processes using `grep`.

Code

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

Now the program is running and its 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. Its an incredibly useful program and its very well worth learning how to use this properly. The nice thing with it is that it is well documented and has an intuitive help. A debugger is basically a program that allows you to watch the execution of a program, it allows you to poke into the internal of a program while it is running removing the need for unreliable print statements scattered around your code in order to understand its functionality. In this case we are running on a completely Bare Metal Machine so we have no means of printing etc. so this is our only way of following the execution of the code. We will now start `gdb` and make it listen to the port that `qemu` outputs over. We can do this as follows.

Code

```
arm-none-eabi-gdb
# you will now get a prompt for GDB
(gdb) file tst                     # tell GBD which file we are going to use
(gdb) target remote localhost:1234 # tell GDB where the program is running
```

Now we can interact with the program that are running inside `qemu`. The first thing we can do is to write `list` this will print out the program. We can use the command `print` to view what the value of a register is, for example

Code

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

should print the value of the register `r7`. The result might be something different as we have not started the program so its 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,

Code

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

This is means that our program counter is currently on address `0x8000` which is a memory address we have given 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 will move the program counter forward one step. To verify this print the value of the program counter again,

Code

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

Now we are `4` addresses further in our code and have executed the line `5`. The reason that the program counter have 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.

Code

```
(gdb) c
Continuing.

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

By executing `c` we just let the computer run and do its loop of `fetc`, `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.

Code

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

_end:
        b         _end
```

We add a label `_end` to the address 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 thereby creating an infinite loop. Try to assemble the program above and write `c` to make sure that you no longer

end with an illegal instruction. The program will no just be stuck in an infinite loop and you will need to send it a kill signal through GDB using `Ctrl-C`.

There are several other ways to control the execution inside `GDB` here are a few useful commands that is worth knowing.

`until <linenum>` runs the code until the `PC` have reached the address in memory corresponding to the

`break <linenum>` sets a breakpoint at `linenum` making the execution stop when it reaches this point

`break <label>`:: sets a breakpoint at `label`, for example you might want to stop the code when it reaches `_end` in the code above.

`step <N>` steps `N` instructions forward

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

`print $<registers>` will print the content of the register

`info registers` will print the content of all integer registers

`list` will show `10` lines of code close to what we are currently executing

`list <linenum>` will list the code around `linenum`

`x <address>` will display the content of memory address

GDB is an incredibly powerful program that is really useful not only for `assembler` programming but for anything that you really do and its worthwhile learning a bit more about how you can use it. GDB has a very useful help included that you can access by simply typing `help` at the prompt.

Now we have the framework up and running and its time for us to move on to actually write some more interesting code. Do make sure that you understand how the procedure works as it is a bit tricky. What you have to remember is that we are running code on a machine that has no operating system, it just starts executing code from the point where the `program counter` is point at and that is it.

3 Program

Now when we have gotten the hang of how the execution works and we are able to run a program we are going to look at writing some more interesting code. What we will first look at is a few additional *directives* that we can pass to the `assembler`. Directives are instructions that the assembler and linker will use when it creates the binary. You have already seen a few of these in the code we have written. Lets look add a few things to our program including some additional directives.

Code

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

_end:
        b         _end
        .section        .data
_text:
        .asciz        "Hello world\n"
_data:
        .word        0,1,2,3,4,5,6,7,8,9
```

In the code above there are 4 directives, `.section`, `.align`, `.global`, `.asciz` which 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` indicates that the following section should be aligned on 2-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

`.asciz` decides that the information that follows should be interpreted as a text and converted using the `ascii` table.

`.word` indicates that the data that follows should be interpreted as a sequence of `32bit` elements

Now let us write our first program. lets make a program that computes the determinant of a 2×2 matrix. The algorithm for this is quite simple,

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} = a \cdot d - b \cdot c \tag{1}$$

Now we will include the matrix directly directly in the file using the `.word` directive. We are also going to need a few new instructions in order to write this code. As we are now programming the hardware directly the place to read about how to code is the reference manual for the hardware. The manual for the `ARM` is very well written but it is very exhaustive. What I prefer to use is the Quick Reference Card that simply outlines the instructions. Another place to get information from is the excellent guide written by Dave Thomas. In order to implement the determinant calculation we need a few more instructions. These are the ones that we will need for the rest of this worksheet.

Data transfer `<operand>{cond}{size} Rd, <address>`

`ldr` LoaD value `<address>` to `Rd`

`str` SToRe value in `Rd` in `<address>`

Movement `<operation>{cond}{S} Rd, Operand2`

`mov` MOVe value in `Operand2` to `Rd`

`mov r0, r1` move the value in `r1` to `r0`

`mov r0, #3` move the value `3` to `r0`

Arithmetic `<operation>{cond}{S} Rd, Rn, Operand2`

`add` ADD `Rn` and `Operand2` and store the result in `Rd`

`sub` SUBtract `Rn` and `Operand2` and store the result in `Rd`

Multipluy `<operation>{cond}{S} Rd, Rm, Rs, {, Rn}`

mul MULtiply `Rm` and `Rs` and store the result in `Rd`

Branch `<operation>{cond} <address>`

b Branch to `<address>`

The data transfer instructions are the ones that allows us to interact with memory. There are several different ways we can give the form of the address. Below are some useful modes that we will need to understand the code that computes the determinant.

`ldr r0,[r1,#4]` Load word addressed by R1+4.

`str r0,[r1],#4` Store R0 to word addressed by R1. Increment R1 by 4.

`ldr r0,[r1,#4]!` Load word addressed by R1+4. Increment R1 by 4.

Now we have everything that we need and it is time for us to implement the determinant code. Write up the code below and make sure that you understand the execution. Try to change the values of the matrix and see if you get the correct result for other matrices.

Code

```
        .section        .text
        .align                2
        .global                _start
_start:
        ldr         r0, =matrix

        ldr         r1, [r0]          @ a
        ldr         r2, [r0, #12]        @ d
        mul         r7, r2,r1         @ a*d

        ldr         r1, [r0, #4]        @ b
        ldr         r2, [r0, #8]        @ c
        mul         r6, r2, r1         @ b*c

        subs         r7, r6                @ ad-bc
_end:
        b         _end

        .section        .data
matrix:
        .word         1,2,3,4
```

Now lets try to extend the program a bit and make it calculate 3×3 matrices instead as this will allow us to test out some branching and flow control in our program. The computation of a 3×3 determinant can be written up as a function of three 2×2 determinants as below,

$$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a \begin{vmatrix} e & f \\ g & h \end{vmatrix} - b \begin{vmatrix} d & f \\ g & i \end{vmatrix} + c \begin{vmatrix} d & e \\ g & h \end{vmatrix}. \tag{2}$$

What we will do now is to create a "function" that we can call to compute the 2×2 determinant. This function will take four arguments, the four elements of the matrix and will return a scalar that contains the determinant. When creating functions we often want to make sure that we do not "destroy" the value of the registers that we are working with internally. What we do to avoid this is as soon as we enter the function we put all the registers that we will use on the stack using the instruction `stmdb` then we use the registers and before returning from the function we use the instruction `ldmia` to reload the original values from the stack. The second thing we need to know a little bit more about is branching. We have so far seen that the instruction `b` simply takes the operands value and writes it to the `PC`. Now however, we want to return from the subroutine that we have created so we need to remember the address where we jumped from. There is a really nice instruction in `ARM` that can do this called `bl` which stands for `Branch Link` what it will do is before it executes the branch it will copy the address of `PC+4` i.e. the instruction after the branch to register `r14`. `r14` is a special register called the *link register* that we can use for branching. Now when we have come to the end of the subroutine we can now copy the value of `r14` to `PC` and then we will automatically return to the place in the code we where before we executed the branch.

Code

```
        ....
        bl        _sub    @ call _sub
        ....        @ will return here

_sub:
        stmdb       sp!, {r2-r3,r7}

        ldmia       sp!, {r2-r3,r7}
        mov        pc, r14
```

Now your task for the rest of the lab is to implement the 3×3 determinant using a subroutine that computes the determinant for the 2×2.

4 Additional Programs

Now if you have managed to get the set-up working and got a small determinant program working you are where we want you to be. Now you can try out a couple of additional programs,

4.1 Fibonacci

The fibonacci sequence is a famous number sequence that appears in lots of places in nature. The sequence can be formulated as follows.

$$f(n) = f(n-1) + f(n-2) \tag{3}$$
$$f(1) = f(2) = 1 \tag{4}$$
$$\tag{5}$$

Create a program which takes the element in the sequence that you want to compute in `r0` and returns from the function with the value of the sequence in `r0`. Make sure that your subroutine does not "trash" any registers.

4.2 Reverse a string

Take the string from a label and write the value in reverse order at a different memory location. In the code below the `.asciz` directive indicates that there will be a string of ascii characters followed by a terminating 0. Below we are using an additional directive called `rept` this will repeat the code between `rept` and `endr` 10 times.

Code

```
string:
        .asciz       "helloworld\n"
stringreverse:
        .rept       10
        .byte       0
        .endr
```

4.3 Bubble sort

Take a sequence on numbers and perform a bubble sort on them.

5 Summary

This lab probably was quite a lot to take in, you have seen programming from a different perspective, where you do not try to understand a programming language but where you actually try to understand the computer. The aim here is not to make you expert `ARM` programmers but rather to try and give you an idea of how it works programming on really low-level. We will do one more lab in week `9` where we continue and go a bit deeper into low-level programming. If you want to try out things till then why don't you have a look in the reference manual and see if there are other instructions that you fancy giving a try. Come up with some more simple programs that you can write, see if you can write them in more clever manner than the verbose way we did here.