Memory Access and Studying C in ARM Assembly

Anas Shrinah

November 29, 2021

In the last lab, we have learnt how to use the GNU toolchain to execute and debug assembly programs. You also wrote and tested some ARM assembly programs. In this lab, we will extend these codes to make them able to load variables from memory and store calculation result into memory. We also will investigate the assembly codes produced by the GCC compiler for the main C control structures.

Credits: Some of 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 will put to practice some of the theoretical information about ARM memory access instructions.
- You will practice how to perform function calls in assembly.
- You will learn how to examine the assembly codes that are generated by the GCC compiler for C programs.
- You will better understand how the main control structures are coded in assembly.
- You will see how assembly can help you to understand programming in general.
- In summary, this lab will equip you with the required practical skills to write advanced 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.

2 GNU Toolchain

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 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.

2.2 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.

2.3 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
```

Use "si"command, which stands for step instruction, to progress the execution one instruction.

```
(gdb) si
```

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
```

2.3.1 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.

3 Programs

In this lab, we are going to extend the determinant, 3n+1, and the Great Common Divisor programs from the previous lab. Your task will be to enable your programs to load from and store to memory.

3.1 The determinant of a 3 x 3 matrix

3.1.1 Load elements of the matrix from the memory

In the previous lab, you wrote a program to compute the determinant of a 2×2 matrix. You were asked to load the registers r0, r1,r2, and r3 with the values a,b,c, and d with immediate values. In this lab, we will improve our program to load the values of a,b,c, and d from memory.

Use the following template to write a code to calculate the determinant of the matrix saved in the data section.

```
.section .text
.align 2
.global _start:
_start:

@ your code goes here

.section .data
_matrix:
.word 1,2,3,4 @ a,b,c, and d values
```

Now you have to load the address of the first element (a) to the register r0. Then load the value of the elements of the matrix using ldr instruction with pre-indexed addressing mode.

Remember you can calculate the determinant of a 2 x 2 matrix using the following equation:

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

3.1.2 Use the code of calculating the determinant as a function

Now you have a program to load the elements of a 2 x 2 matrix from memory and calculate the determinant of this matrix. Let us try to extend this program a bit and make it calculate the determinant of 3 × 3 matrices. The computation of a 3 × 3 determinant can be written up as a function of three 2 × 2 determinants as below.

$$\begin{vmatrix} \overset{0}{a} & \overset{1}{b} & \overset{2}{c} \\ \overset{3}{d} & \overset{4}{e} & \overset{5}{f} \\ \overset{6}{g} & \overset{7}{h} & \overset{8}{i} \end{vmatrix} = \underline{\overset{0}{a} * \begin{vmatrix} \overset{4}{e} & \overset{5}{f} \\ \overset{7}{h} & \overset{8}{i} \end{vmatrix}} - \underline{\overset{1}{b} * \begin{vmatrix} \overset{3}{d} & \overset{5}{f} \\ \overset{6}{g} & \overset{8}{i} \end{vmatrix}} + \underline{\overset{2}{c} * \begin{vmatrix} \overset{3}{d} & \overset{4}{e} \\ \overset{6}{g} & \overset{7}{h} \end{vmatrix}}$$

brute force

What we will do now is to create a function that we can call to compute the 2 × 2 determinant. Let's name this function `_det_2x2`. To call this function use the `bl` instruction: `bl _det_2x2` This function will take four arguments, the four elements of the matrix and will return a scalar that contains the determinant.

Use r1,r2,r3, and r4 to pass the four elements of the 2 x 2 submatrix. The function `_det_2x2` should calculate the determinant and store it in register r7 before the instruction `move pc ,lr` is executed to resume the execution at the main code.

For example to calculate the determinant of $\begin{vmatrix} e & f \\ h & i \end{vmatrix}$ make sure you load r1,r2,r3, and r4 with the values of the elements e, f, h, and i from the memory. Assume the elements of the 3 x 3 matrix are stored in the data section as follows:

use @ to comment

```
.section .data
_matrix:
.word 6,1,1,4,-2,5,2,8,7 @ a,b,c,d,e,f,g,h, and i values
```

result : -306 ✓

matrix_3y3 : 0xfffffece

3.2 The 3n+1 problem

In the previous lab, you wrote a code to run the 3n+1 procedure.

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. Your assignment now is extend your 3n+1 program to do the following:

1. Load the number n from a memory location.
2. At the end of each loop, store the new value of n into memory. Start storing the new values of n right after the location of the original value of n.
3. Your program should halt if the sequence 1 - 4 - 2 appeared number of times equal to a value stored in a memory location.

1 is always followed by 4-2, so we sub repet ,#1 at every n+1=1 and b-end

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

1 - 4 - 2

1 - 4 - 2 ← when rept = 0 and n+1=2
cmp cmpeq

3.3 Great Common Divisor - Euclidian Algorithm

In the previous lab, you wrote a code to calculate the Great Common Divisor (GCD) for two positive numbers using the Euclidian Algorithm.

Your assignment now is to extend your GCD program to do the following:

1. Load the first and second numbers from memory locations.
2. At the end of each iteration, store the remainder into memory. Start storing the remainders right after the locations of the two input values.

4 Studying C in Assembly

In C, a typical program will use some of the following control structures:

- If statement
- If...else statement
- For loop
- While loop
- Do while loop

This task is about studying how these control structures translate into ARM assembly. Your task is to experiment with small sample C programs showing each of these features one at a time and study the disassembly to discover how the compiler translates it into assembly language.

Use the following command to compile `c_code.c` and produces an object file `c_code.o`. Change the `c_code.c` to the name of your file.

```
arm-none-eabi-gcc -g -O1 -c c_code.c
```

Then use this command to shows the ARM assembly and machine code in an object file, with the C source lines as headings.

```
arm-none-eabi-objdump -S c_code.o > c_code.s
```

Or you can use the Compiler Explorer https://gcc.godbolt.org website to get the assembly codes for the C codes of the following tasks. Just make sure you chose the C programming language, ARM gcc 8.5 (linux), and use `-O1` for the compiler options.

4.1 If statement

Copy the following codes to separate C files and disassemble them, on at a time, using the instructions in Section 4.

4.1.1 If statement - example 1

```
int if_int(int x, int y){
    if(x ==y){x += y;}
    return x;
}
```

4.1.2 If statement - example 2

```
int if_char(char x, char y){
    if(x ==y){x = 'a';}
    return x;
}
```

4.2 If...else statement

Copy the following code to a C file and disassemble it using the instructions in Section 4.

```
char if_else (char a ,char b) {

if( a == 'a' ) {
      a++;
   } else {
     b++;
   }
return a+b;
}
```

4.3 For loop

Copy the following code to a C file and disassemble it using the instructions in Section 4.

```
int for_loop (int a ,int b) {

int c =0;
for( int i = a; i < b; i++ ){
      c++;
   }
return c;
}
```

4.4 While loop

Copy the following codes to separate C files and disassemble them, on at a time, using the instructions in Section 4.

Notice the difference between `b = a++ + b;` in the first code and `b = ++a + b;` in the second code.

4.5 While loop - example 1

```
int While_a_plus_plus (int a ,int b ,int c) {
   while( a < c) {
      b = a++ + b;
   }
   return b;

}
```

4.6 While loop - example 2

```
int While_plus_plus_a (int a ,int b,int c) {

   while( a < c) {
      b = ++a + b;
   }
   return b;

}
```

4.7 Do while loop

Copy the following code to a C file and disassemble it using the instructions in Section 4.

```
int do_while (int a ,int b) {

  do {
      a = a + 1;
   }while( a < b );
   return a;
```

Congratulations, you have completed the ARM assembly language labs!