ARM Assembly

Carl Henrik Ek

Week 9 & 10

Abstract

In the previous lab we introduced how the cross development environment worked using `qemu` and `gdb`. The aim was for you to get a good grasp on how the execution of code works at a low-level inside the computer. In this lab we are now moving this one step further and aim to create a few more complete programs and practice our assembler programming skills.

During the lectures before the reading week we went through a few programs. Make sure that you understand each of these programs as we will now make use of all the different parts that we introduced. The most important part is to make sure that you understand how conditional execution works, how we can execute instructions that sets flags and then proceed differently dependent on what these flags are. Before you start with the lab make sure you go through the following programs in the repository,

`strcmp.s` compare two strings to each other

`fibonacci.s` computes the fibonacci sequence

`largest.s` finds the largest number in a sequence

`factorial_recursive.s` computes the factorial sequence using recursion

Once you feel that you have understood how these programs work it is time for us to move forward and build a slightly bigger program. But before we do this we need to introduce the concept of `fixed point arithmetics`.

1 Fixed-Point Math

So far we have only used integers for our computations, this can be very limiting as we often have operations which should work on fractions of integers. The simplest way to do so is if you have a dedicated hardware that implements floating point operations. However, there is another approach that allows to use the same integer hardware but still gives us fractions. The idea is very simple, rather than working on the numbers directly we pre-multiply each number by a value and work on this shifted representation instead. As an example, say that we would want to compute $\frac{5}{2} = 2.5$ what we would instead do is to compute $\frac{50}{2} = 25$ giving us an integer result. The nice thing about this representation is that additions and subtractions works exactly the same while we have to take a little bit more care when it comes to multiplication and division. In the example above we multiplied by 10 as the numbers where represented in base 10. As we will work with binary numbers we will instead multiply by something in base 2. Importantly to do so we do not need to use a an expensive multiplication we are going to use `shift` operations instead. There are two different shift operations in the `ARM`, *logical* and *arithmetic*. The first shift does not respect the signed bit while the second does so. Try the code below and make sure that you understand the differenc between the two operations.

Code

```
_start:
        mov         r0, #42
        mvn         r1, #42

        lsl  r0, #8
        asr         r0, #8
        lsl         r0, #8
        lsr         r0, #8

        lsl  r1, #8
        asr         r1, #8
        lsl         r1, #8
        lsr         r1, #8
```

As you can see there is only 3 different shift instructions, the logical shift in both directions while the arithmetic shift only exists in the right direction. Think about why that is, why don't we need the `asl` instructions?

As the registers in the `ARM` is `32bits` we will use the upper `24` for the integer part and the bottom `8` for the fraction. Now the only thing that we need to keep track of is which representation a register is in. If we multiply two fixed point registers in this format together we now have a result that has lost integer precision as the new number will have `16` bits to represent the fractional part. Look at the code below and make sure that you understand the results.

Code

```
_start:
        mov         r0, #42
        mov         r0, r0, lsl #8          @ 24.8
        mov         r2, #1
        mov         r2, r2, lsl #6          @ 24.8 (0.25)
        add         r0, r0, r2      @ 24.8 (42.25)

        mov         r1, #4
        mov         r1, r1, lsl #8          @ 24.8 (4)

        mul         r2, r0, r1          @ 16.16
        mov         r0, r2, asr #16          @ 16.0

        @ You should now have the result 169 in r0
```

In the implementation above I've merged the shift command and made it a part of the `mov` this is a feature of the ARM processor as it allows you to have a shift for free. In the code above there is no benefit but I am sure you will find ways of making use of this.

The same thing is true for division but we will now loose precision in the other directions. If we divide two numbers that are in `24.8` representation we will have a result that has no fractional term. In order to avoid this we first have to scale the nominator prior to performing the division. Look at the code below and make sure that you follow the results.

Code

```
_start:
        mov     r0, #42
        lsl     r0, r0, #8              @ 24.8 (42)
        mov     r2, #1
        lsl     r2, r2, #6              @ 24.8 (0.25)
        add     r0, r0, r2              @ 24.8 (42.25)

        mov     r1, #4
        lsl     r1, r1, #8              @ 24.8 (4)

        lsl     r0, r0, #8              @ 16.16 (42.25)
        sdiv     r2, r0, r1              @ 16.8
        mov     r3, #1
        add     r2, r3, lsl #7
        asr     r2, r2, #8

        @ You should now have the result 11 in r2
```

Now as you probably see there is no rounding involved when we shift, we will simply cut the fractional part of the numbers. In order to get the rounding to work, what we need to do is to shift the rounding from the mid-point 0.5 to be at an integer value instead. We can do this by simply adding 0.5 and then cut the fraction. This is what we do in the `add r2, r3, lsl #7` where we have the value `1` in `r3` that we shift by `7` meaning that it is equal to `0.5`.

Make sure that you got the idea of how fixed-point arithmetics works, try to implement something that does a few operations in a row and make sure that you get the result the way it should be. We are now ready to move onto the big program that we are going to write.

2 Gaussian Elimination

We are now going to put all our knowledge together to implement a very useful routine, namely a general method to solve a system of linear equations. In specific, we are given a problem specified on the following form,

$$\underbrace{\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}}_{\mathbf{A}} \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}}_{\mathbf{x}} = \underbrace{\begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}}_{\mathbf{b}}, \tag{1}$$

where $\mathbf{A}$ and $\mathbf{b}$ is known and we want to infer what $\mathbf{x}$ is. The idea with Gaussian elimination is to re-write this on a form such that the matrix of coefficients is upper-triangular.

$$\underbrace{\begin{bmatrix} y_{11} & y_{12} & y_{13} & y_{14} \\ 0 & y_{22} & a_{23} & y_{24} \\ 0 & 0 & y_{33} & y_{34} \\ 0 & 0 & 0 & y_{44} \end{bmatrix}}_{\mathbf{A}'} \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}}_{\mathbf{x}'} = \underbrace{\begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}}_{\mathbf{b}'} \tag{2}$$

Given the form above it is easy to infer what the values of $\mathbf{x}$ is by first inferring $x_4 = \frac{b_4}{y_4 4}$ and so forth. The nice thing is that we can write down an algorithm that solves the problem above and this is what we will now try to implement.

You should now write a program that converts the coefficient matrix to an upper-diagonal matrix using elementary row operations. Below is a naive algorithm that will generate an upper-diagonal matrix, it is by no means complete and not optimal in any sense.

Algorithm 1 Naïve pseudocode for Gauss elimination

Data: A pointer to a Square Matrix **A** a column vector **b** and the size N in File matrix.s
Result: The upper triangular matrix written nicely on the terminal

Arrange the matrix and the vector so that it is on the form in memory $M = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1N} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2N} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{N1} & a_{N2} & \cdots & a_{NN} & b_N \end{bmatrix}$

for *i = 1 ... N+1* **do**
  for *j = i+1 ... N* **do**
    for *k = 1 ... N+1* **do**
      $p_k = -\frac{M_{ik}}{M_{ii}} M_{ji}$
    end
    for *k = 1 ... N+1* **do**
      $M_{jk} = p_k + M_{jk}$
    end
  end
end

In order to make sure that you got everything correct I suggest that you do this first on paper with a simple equation system. As a second line of attack you can implement this in a high-level language such as `Python` or `C` first just to make sure that you got it right.

Now the first tricky bit that we will have to work around is that we do not have any floating-point registers and operations. We want to implement this completely in terms of integers. To do so we will use a fixed point representation to represent fractions. As a suggested precision use 24 bits for the integer part and 8 bits for the fraction. The ARM implements a arithmetic shift instructions that will keep track of the signed bit for you, this will be very useful when working with fixed point.

Rather than writing everything in a single file I recommend that you place the matrix and vector in a separate file called `matrix.s` you can then simply `link` this file as you do not have to reassemble it each time you alter the code. This will also allow you to test different matrices efficiently.

3 Summary

If you have reached the end of this lab you have done stellar work, this is not an easy task to implement as it includes a lot of different elements. Importantly though, assembler programming doesn't become much harder than this, its not about knowing the "language" as well there is none, its about understanding the hardware that we program. In this unit we are only using the CPU but there are lots of more fun to be had when program things such as the graphics display etc.