High Level Languages

Carl Henrik Ek

Week 11

Abstract

Today our aims are rather bold, we are going to try and tie together the low-level programming we have done over the last three weeks with the high-level programming that you have been doing with Neill Campbell for a large proportion of the term. Our goal is that as we now know a fair bit about how computer hardware works, we want to treat the computer with respect and write nice and pretty code that it will enjoy to execute. But the problem is, when we write in an abstract language that is separated from the hardware, i.e. anything but assembler, we do not actually know what the computer actually does. Today we are going to try and figure this one out and look at what assembly code a piece of C-code actually generates.

Over the last few weeks we have been working on implementing Gaussian elimination in Assembler, what we will do now is to also implement the same thing in `C`. As we can create an executable that will run on your machine this means that somewhere inside that executable resides assembly code, today we are going to use a set of tools and extract this so that we can have a look at the code that the GNU C compiler generates and compare it to the one that you carefully crafted by hand.

1 Gaussian Elimination in C

Let us begin by implementing this in `C`. Now I am sure that you can do a much better job than I have and write this in a nicer way but below is a quick implementation that will do the job for you.

Code

```
#include <stdio.h>
#include <stdlib.h>

void printmatrix(float M[3][4])
{
  for(int i=0;i<3;i++)
    {
      for(int j=0;j<4;j++)
        {
          fprintf(stdout, "%f,", M[i][j]);
        }
      fprintf(stdout,"\n");
    }
}


int main(void)
{
  float M[3][4] = {1,2,3,4,5,6,7,8,9,10,11,12};
  float *p = (float *)malloc(sizeof(float)*4);

  printmatrix(M);

  for(int i=0;i<4;i++)
    {
      for(int j=i+1;j<3;j++)
        {
          for(int k=0;k<5;k++)
            {
              *(p+k) = -M[i][k]/M[i][i]*M[j][i];
            }
          for(int k=0;k<5;k++)
            {
              M[j][k] = *(p+k) + M[j][k];
            }
        }
    }
  printmatrix(M);

  free(p);
  return 0;
}
```

Having written the code, make sure that it does the job that it is supposed to by testing the executable. You should get a second matrix which is upper-diagonal and you can now solve the linear equation system easily. Now what we would want to do is to run this also in the arm environment. Sadly we cannot do this out of the box as this code uses additional system routines for printing the matrix. Furthermore, we use `malloc` this is a system routine so we also need to replace this with something else. Therefore after you have verified for yourself that the code works strip the printing and the malloc away from the code so that we only have the logic left. Now we are ready to move this to the `ARM` computer.

Lets first compile this for the `ARM` architecture rather than the `x86` machine that you are currently using.

We can do this by using the same tools as previously. Assuming your code is called `gauss.c` this should do the trick for you.

Code

```
> arm-none-eabi-gcc --specs=nosys.specs -g gauss.c
```

Now we have the executable named `a.out` that we are going to continue working with. Now we are going to have a look at the executable that we have and extract the underlying `assembly` code that `GCC` have generated. We can do so by using the program `obj-dump`.

Code

```
> arm-none-eabi-objdum -S a.out
```

In order to extract the source from an executable we give the program the argument `S`. As you can probably see the terminal now got flooded with text. This is because `obj-dump` normally outputs to `stdout` which is actually very nice as this means that we can very easily `pipe` this output to a file. You can do this as follows.

Code

```
> arm-none-eabi-objdum -S a.out > gauss.s
```

Now when we have the assembly code lets open this and have a look at it. Open the file `gauss.s` in your favourite editor[1]. You will see that the file is actually rather big, mine ended up being over `1000` lines of assembly code with a few lines of debug output. As you can see the file is nicely organised using columns, the first column specifies the `address` in memory where this is located, the second the machine code and the last the assembly code. Having programmed the ARM over the last few weeks you know that at the start the program counter sits at address `0x8000` so thats a good place to start looking. What you will probably have as the first set of instructions is the code below,

Code

```
mov        ip, sp
push        {r3, r4, r5, r6, r7, r8, r9, sl, fp, ip, lr, pc}
sub        fp, ip, #4
sub        sp, fp, #40        ; 0x28
ldm        sp, {r4, r5, r6, r7, r8, r9, sl, fp, sp, lr}
bx        lr
```

Now most likely you have not encountered the register `ip` before. This is just a renaming of the register `r12` it is used as a *scratch* register for the compiler, something that you use but you do not really need to take care of. The same thing is `fp` this is what is called a *frame pointer* and is an additional registers which is used when we have multiple processes running at the same time. Now let us move forward in the code. Because we have added the argument `-g` when we compiled the executable we actually know what parts of our C code generate which part of the assembly code, therefore we can search for main as this is where our logic is. The first location in my code where I find this looks something like this,

Code

```
8228:        eb000007        bl        824c <main>
822c:        eb0001f3        bl        8a00 <exit>
```

[1] emacs

This is a simple `branch link` to address `0x824c`. If we go to this address we will see the actual logic that we have implemented. Spend some time looking through this and see if you can understand what each part of the code actually does.

After looking at the code for a while I am sure that you are surprised of how long it is, the actual mine is between address `0x824c` and `0x8420` that is `468` bytes of memory, with each instruction taking `4` bytes of space this means that we have `117` lines of code. I assume that a lot of you who have done the implementation yourselfs actually ended up with something quite significantly shorter. This is due to two things, first and foremost because you are excellent programmers but also because writing a compiler is really really hard work.

2 Other Code

Now when we have the framework for dissassembling code and actually looking what it does we can actually try to see what GCC does with other things, for example what you can do is to try to alter the flags and see how the output changes. Lets write this silly program.

Code

```
int main(void)
{
  int j = 0;
  for(int i=0; i<200000; i++)
    {
      j += i;
    }

  return 0;
}
```

Now first create the executable in exactly the same manner as you did previously and look at the assembly code. I hope that the code makes sense. But now, lets look a bit at what we have actually done, this piece of code is ridiculous, this program will always return 0, the loop is just something that doesn't actually do something relevant. Now try to compile the program by altering the flags slightly and turn on the optimiser.

Code

```
> arm-none-eabi-gcc --specs=nosys.specs -g -O3 silly.c
> arm-none-eabi-objdum -S a.out > silly.s
```

That is quite a difference right, now it simply places `0` in `r0` and returns. In this case the optimiser have done a huge difference, removing an enormous amount of loops for no good reason, this loop has `200 000` `cmp`, `mov`, `bne` and `add` which have been replace by one single `mov`. Quite a bit of speed-up. In order for the optimiser part of the compiler to be able to do things like this it effectively needs to try and understand what the code does without actually running it. That is quite a challenging thing but in many cases it is possible. Now you can try to test different things that you write in C and look what the generate. As a first thing what would happen if you would change the code such that the return statement was `return j` instead? You can also try this code

Code

```
int main(void)
{

  int a = 4;
  a = a/2;

  return 0;
}
```

The important part of the assembly code should look something like this, do you understand what the code actually does?

Code

```
mov        r3, #4
str        r3, [fp, #-8]
ldr        r3, [fp, #-8]
lsr        r2, r3, #31
add        r3, r2, r3
asr        r3, r3, #1
str        r3, [fp, #-8]
```

3 Summary

Now when you have reached this lab we hope that you feel that you can tie together everything that we have done during the term. We started of with the real lowest level discussing how the logic is implemented inside a computer. We then moved to the part where we saw how programs can be executed inside a computer, and importantly you directly wrote code that executed on one of these bare-metal machines. At the same time you have been learning how to write C for the duration of the term. Looking over your shoulders in the labs I've seen quite some remarkable stuff being created so I know that you are excellent C programmers. Now the hardware doesn't speak C, it speaks machine code, so how does it execute. Well, machine code is just assembler written in numbers, so each executable has to be in terms of this. What we did here was that we now use a dissassembler to look at the code that actually comes out of a compiler, and because you know assembly you can read it.

So why is this at all important. I am going to leave with a simple comment that I got from a friend who is one of the key developers of Electronic Arts game engine. "A good programmer thinks in assembly, we might not write it directly, but we think about what code is going to be generated by the compiler, and we know the compiler well enough to know the assembler it spits out".