G51PRG Exercise Five: DisARMing ELF

Steven R. Bagley

1. Introduction

This exercise involves writing an ARM disassembler. That is, a program that can take an ARM instruction and decode it back to its mnemonic. For instance, you may be given the integer values `0xEF000002`, `0xE35500FF`, or `0xD282102A` and your program should decode these back to `SWI 2`, `CMP R5, #255`, and `ADDLE R1, R2, #42` respectively. For those doing G51CSA, this is similar to what you have done by hand in coursework two. For those who are not doing G51CSA, or would like a refresher then there are some recommended reading in the resource section.

In addition, this program should be a full 'disassembler' in that it takes an ELF (*Executable and Linking Format*) executable and decodes all the instructions in the program. ELF is a binary file format that contains both the ARM instructions, and a series of headers that specify where to load these in.

Finally, rather than getting you to write the program *ab initio*, we are providing a skeleton file containing most of the code required to process an ELF executable binary, although you will need to make some modifications to this (as described in section 4). You are to add your code to this skeleton to make it print out the ARM mnemonics for each instruction as well as the hex value.

There are two main tasks for this coursework:

- Implement the decoding of ARM instructions
- Modify the ELF loader so it processes all program headers rather than just the first.

2. Getting Started

To get started, you will need to download the skeleton file, and some sample ELF ARM executables from the G51PRG website. These can be found in the `cswk5-support.zip` file.

Once extracted, you will find a directory containing some sample test files (for each `.elf` file, the `.s` file used to generate it is also provided, which you can compare your decoded output against), a C header file `elf.h`, and the skeleton `disarm.c`. The header file, `elf.h`, contains definitions of various structures used in the ELF format and is `#include`d into `disarm.c`. It is important that the two files are kept in the same directory, *otherwise it will not compile…*

As it stands, if you compile and run `disarm.c` and feed it one of the sample ELF executables, it will read in the first program section of the ELF file and output the hex values for each instruction (and also any embedded data, which we'll treat as if it were code).

The code to load the ELF file can be found in `main()`. An ELF binary begins with a header described in the `struct _elfHeader` (although note it uses `typedef` to simplify the C code). Since this is stored on disk in binary, we can read it directly into memory using `fread()`. This header contains various fields that tell us about the instructions stored in the file. We are interested in the location of the ELF program headers, which can be found in the field `phdrpos` (with the number of headers stored in the file present in `phdrcnt`).

The program then uses `fseek()` to move within the file to the program header and reads it into a block of memory allocated using `malloc()`. The program header is another structure (`struct _elfProgHeader`), that contains the location and number of ARM instructions within the file. With this knowledge, we can `malloc()` enough memory to read them into. Note that

the program currently only loads in the first program header, but, within the ELF file, a program is usually split into multiple sections so you will need to modify this code to read in all the program headers (see section 4).

Having read in the ARM assembly instructions from the file (again using `fseek()`, `fread()` and `malloc()` based on the values stored in the program header fields, `offset` and `filesize`), it then calls the function `Disassemble()` which actually starts to disassemble the file. This function takes a pointer to the loaded instructions, and a count of the number of instructions (`Disassemble()` treats this as an array), as well as the '`startAddress`' of where these instructions should be loaded in memory. This is used to print the correct memory location of each instruction as well as its hex code. It then passes each instruction to `DecodeInstruction()` to print out the ARM mnemonic.

3. Decoding Instructions

As described above, the `DecodeInstruction()` routine is called from `Disassemble()` for each instruction, passing the 32-bits as an `unsigned int` parameter. You will need to implement this function so that it prints out the correct mnemonic for the instruction passed in. How you do this is up to you, you may choose to build up a string in memory and then print it in one go, or to print out the separate parts as you go…

To do this, you will need to follow the same procedure you would by hand to decode an ARM instruction: isolating certain bits to find out what type of instruction it is, and then using that knowledge to process the other bits in the instruction. You can do this in C by using the bitwise operations (and, or, and xor) and bit shifting.

As an example, if we wanted to isolate the condition codes for an instruction (stored in the most significant four bits), we could use a bitwise-AND to clear all but the top four bits and a shift right to move them from bits `31`– `28`, to bits `3`– `0`. So with, the instruction in the `unsigned int instr`, the following code would give the conditional execution code in `ccode`:

```
ccode = (instr & 0xF0000000) >> 28;
```

We suggest that you implement the instructions gradually, starting with the simpler ones and once they are working progress to the trickier ones. Our suggestion would be start by processing `SWI`s, then move to branch instructions, before moving to the data processing instructions and then onto other instruction sets (load/stores, multiply etc). Remember to test your program works properly after you've implemented each set of instructions!

Any coursework fully implementing `SWI`s, branches and data-processing instructions will be awarded the equivalent of a mark in the 2:1 range (60%-70%). To obtain the equivalent of a first-class mark (>70%) then you will have needed to also (at least) have implemented the load/store instructions as well.

Note that decoding some of the instructions (branches, data processing, for instance) will require you to sign-extend and rotate bits which C does not provide support for. We've provided some simple functions that perform these operations (see section five for details of how they work).

Note Remember that you don't want to print an '`S`' on a `CMP` insturction, and that there's no need to print a conditional executional code for 14 '(`AL`').

4. Modifying ELF Loader

The other part of this exercise is to modify the ELF loader so that it processes every `_elfProgHeader` in the file, rather than just the first one. This will ensure that all the instructions are printed out.

All the `_elfProgHeaders` are stored one after the other in the file (the number is specified by the field `phdrcnt` in the `_elfHeader` structure). You will need to modify the loading code so that it allocates enough memory for `phdrcnt` headers and reads all of them into the memory (effectively creating an array of them). Finally, you will need to put a loop into the code so that it processes each program header and block of instructions from the file, and calls `Disassemble()`

on that block of code.

Some of the sample files are prefixed '`MP`' to signify that they contain multiple program headers.

5. Helper functions

There are two helper functions provided: `SignExtend()`, and `Rotate()`. The first takes a two's-complement number in an arbitary number of bits and converts it to an `int`. The second rotates an `int` a specified number of bits to the right.

5.1. Sign extension

Some ARM instructions (e.g. the branches) encode a two's-complement number in less than 32-bits. This means that once extracted from the ARM instruction C won't treat it as negative number (when approrpriate). This function:

```
int SignExtend(unsigned int x, int bits)
```

Extends the two's complement number `x` to fill all 32-bits of an `int`. To use it, extract the bits from the instruction and pass them as the parameter `x`. Also, pass the number of `bits` used to store the number, e.g. for a branch, 26. The return value would be a signed `int` that can be used elsewhere in your program, (e.g. to pass to `printf()`).

5.2. Rotate

Some instructions (e.g. the data-processing instructions) encode a number as a set of 8 bits and a number of bits to rotate them to the right. This function:

```
int Rotate(int rotatee, int amount)
```

will rotate the parameter `rotatee` to the right by `amount` bits and return the results.

6. Submission

Submission will be by cw submit in the usual fashion. You should submit both your modified `disarm.c` and also `elf.h`, so that we can compile and run your file.

7. Resources

This PDF contains a breakdown of how each instruction is assigned into the instructions bits:

```
http://www.cs.nott.ac.uk/~dfb/G51CSA/resources_files/layouts-3.pdf
```

Also, you may find it instructive to look at DFB's G51CSA lecture notes as well:

```
http://g51csa.cs.nott.ac.uk/
```