EECS 373 Cortex M3 Simulator

Questions? Issues? E-Mail `373F11class@umich.edu` with [proj1] in Subject

Phase 1 – Assigned: Sep 15, **Due: Sep 30, 10:30 AM(class)**

The goal of this project is to teach everyone a little about their little corner of the Cortex M3, as well as a little bit about the challenges of building and designing a system where everyone needs to work together, and everyone's work depends on everyone else.

You should read this whole document before trying anything else, it will save you a lot of pain (and isn't that long – promise).

1 General Project Information

This project is likely unlike any project you have ever done for an EECS class before. While each of you have your own tasks you are required to implement, the entire class is working on the *same* project. This has some very interesting dependency implications; notably that everyone is depending on everyone else. If you write the perfect `mov` instruction, but the person responsible for the `add` or `sub` instructions hasn't written them yet, the simulator probably won't get very far. In addition, this model will make "debugging by trial" very hard. You will have to think hard about every line of code you write to ensure it is correct.

1.1 Subversion

To collaborate, we will be using subversion. If you have never used subversion before, take a few minutes to check out this primer at `www.eecs.umich.edu/~pmchen/eecs482/svn.pdf`. The repository is located at

```
svn co svn+ssh://UNIQNAME@loginlinux.engin.umich.edu/afs/umich.edu/user/p/p/ppannuto/Public/373F11 proj1
```

and should be accessible from any CAEN machine or via ssh to `loginlinux.engin.umich.edu`.

It is in everyone's best interest to make commit messages with useful information – particularly since they will be relied heavily upon for grading. The first line of your commit message should be a short (60ish characters) summary of the commit. Every time someone makes a commit, an email will be sent to `373F11class@umich.edu` with subject `[svn] --First-line-of-the-commit-message`.

A quick reminder, recall that everyone is sharing this repository, so try to keep expletives or other such frustrations to a minimum. Also, don't clutter the repository by adding .o or other compiled files to the repository. Stub .c files are supplied for you already.

1.2 Grading

Since everyone is working on the same project, we will be relying on subversion to attribute credit for work done on the project. There is not a fixed grading rubric, as the requirements vary somewhat depending what portion of the simulator you have been assigned. The expectation is that the simulator will reach a point where it is working 100% accurately. We will be actively monitoring the class's progress to help ensure we meet that end. Please take careful note of the deadlines; if you slip a deadline, we will write your portion of the project for you, and if that happens you should expect your grade to drop drastically.

1.3 Deadlines

The *entire* simulator is due Sep 30, 10:30 AM. We will have one checkpoint, due Sep 23, 10:30 AM. The purpose of the checkpoint is to ensure that a few procrastinators do not cause the entire class to fall behind on the project. By the checkpoint you are required to have made a serious, appreciable effort towards the completion of your portion of the project. This is a subjective measure, and we will send warning emails out 24 hours before the deadline if it looks like you aren't going to make it. You are very strongly advised, however, that a haphazard commit Thursday morning most definitely does not qualify.

2 Building the Simulator

This section deals with code in the root directory and the `operations/` directory.

2.1 Simulator Architecture

You should be familiar with the Cortex M3 from lecture and other resources. This section explains the architecture of the M3 simulator. In general, everyone has one or two functions they will have to fill in the body for. In writing your functions, you will have to use accessor functions. These are all sorted and prototyped in `cortex_m3.h`.

YOU SHOULD NOT HAVE TO MODIFY THE cortex_m3.h FILE

Our simulated chip has Power-On Reset, so the first function called will be the `reset` function. Recall we are simulating the processor, so this function must do everything a real Cortex M3 would do when the reset pin is asserted.

You will notice there is no `main()` function anywhere - the simulator program entry (main function) is in the provided `simulator.o` file. The simulator architecture will call your functions as necessary to correctly simulate a Cortex M3. To draw example from the previous paragraph, when the simulator begins running, the first function it will call is `reset()`.

Every cycle the simulator will (like a real processor) read the value of the `PC` register and fetch that instruction. The simulator will then check if it knows how to execute that instruction and either proceed or quit.

2.2 Registering Opcodes, or "Teaching the Simulator"

Say for example the simulator fetches the instruction `0xb083`. (This is `sub sp, #12`). The simulator will scan the list of all registered opcode masks to see if it knows how to execute this instruction. A pair of opcode masks are made up of the "`ones_mask`" and "`zeros_mask`". These masks represent the bits that *must* be one and *must* be zero respectively. For example the masks `(0xb0c2,0x4f40)` would match `0xb083` while `(0xb080,0x4f80)` would not. It is worth noting that some instructions may require multiple masks.

Exercise: Why do you need two masks?

To register an opcode mask is to "claim" that instruction as one that you know how to execute. The implementation of the `sub` instruction is then done something like this:

`operations/sub.c:`

```
#include "../cortex_m3.h"

int sub4(uint32_t inst) {
        uint32_t new_sp_val;
        // ...
        CORE_reg_write(SP_REG, new_sp_val);
        return SUCCESS;
}

void register_opcodes_sub(void) {
        // This instruction is a 16-bit instruction, so we require
        // the top halfword of the instruction to be all 0's
        register_opcode_mask(0xb0c2, 0xffff4f40, sub4);
}
```

n.b.: The given examples are **not** the correct masks for sub4

The implementation for all of the Thumb instructions can be found in the operations directory. Since this is a shared project, the implementation is broken out into different files, where the filenames are taken from the boxes of the ARM Quick Reference Card.

Please be sure to implement functions in the correct files, if in doubt – please ask (or consult `operations/MISSING.txt`).

2.3 Running The Simulator

The supplied Makefile will generate an executable called simulator. The simulator does not require any arguments, but has many options that should make testing and debugging at least a little less painful (`./simulator --help`).

To compile the simulator, simply type "`make`" in the root directory. You may alternatively choose "`make debug=1`" or "`make debug=2`" to enable debugging output. In general, try to keep "Level 1" debugging statements to things that do not print often (i.e. won't flood the screen when run). See `cortex_m3.h` for more details on the debugging controls.

To run the simulator, simply execute `./simulator`. The simulator will attempt to load the file `flash.mem` into ROM and then begin executing.

Until you are capable of generating valid programs, you can use the `--usetestflash` flag to use an internal copy of the supplied `echo` program.

2.4 Simulator Peripherals

Our simulator has two peripherals, some LEDs and a UART. For now, these peripherals are simply registers in memory. Documentation on reading and writing these devices can be found in `memmap.txt`.

2.4.1 LEDs

In general, toggling LEDs does not produce any output. You can enable notification of LED toggles via the `--showledtoggles` flag.

2.4.2 UARTs

To communicate with the processor's UART we will use the `netcat` program as the other peripheral. Once the simulator is running, in another terminal (on the same machine – careful if you're using SSH) run `nc -4 localhost 4100` to send and receive bytes.

3 Programming The Simulator

This section deals with code in the `programs/` directory.

3.1 Peripheral Support Libraries

Each of the peripherals (LEDs and UART) have support libraries found in `programs/lib/`. The files `led.h` and `uart.h` are supplied for you, but you are responsible for implementing `led.s` and `uart.c`.

To make the LED peripheral more challenging, you are required to write the function `LED_set` *in assembly.* In addition, your `LED_set` implementation cannot use any branch instructions (except bx to return). Lastly, you may only use 16-bit instructions for this section. This may seem challenging at first, but once you "get it" you'll really be thinking ARM!

Code for the UART peripheral should be written in C.

3.2 vector.s and memmap

These files are both located in the `programs/` directory, and will need to be completed before you can generate code.

Once these files are complete, you should be able to type `make` in the `programs/` directory and compile our test programs.

3.3 Test Programs

The simulator should run any valid thumb2 assembly. We have supplied three test programs, `blink.c`, `echo.c`, and `echo_str.c`. In addition to these programs you are required to write test programs to verify the validity of the simulator.

Your test programs should exit with return code 0 (that is from `main` it should `return 0;`) if the test passes or any non-zero return code. Your tests must contain legal ARM code and should not throw any exceptions.

4 General Hints

4.1 First Steps

1. Install the ARM toolchain (Confused? See 4.4)
2. Enter the `programs` directory and type `make` to build a copy of all the programs designed to run on the simulator. Only the `basic` program will work at first

   The `Makefile` generated four different file types: .o .elf .list .bin. The .bin file is a binary image, something you could flash onto real hardware, or load into the simulator via `--flash`. What are the other files?
3. Back in the root directory, run `./reference --flash programs/basic.bin --dumpallcycles`. What happened?
4. Now try `arm-[...]-eabi-objdump -Sd programs/basic.elf`. What is this output? How does it relate to what happened in the previous step?
5. Type `make` in the root directory to build your own copy of the simulator. Try the same command `./simulator --flash programs/basic.bin --dumpallcycles`. What happened this time?
6. You've got the basics! Now start reading through the code and try to figure out what's going on.

4.2 Debugging

Debugging this project will be challenging (so start early!). Check out all the options you get with `./simulator --help`. Try to eliminate as many variables as you can. If you're trying to debug the simulator, take advantage of the `--usetestflash` option so you know you're running "good" code. On the flip side if you're debugging programs or libraries you've written, use the `reference` simulator to execute it.

4.3 TTTA

Sprinkled throughout the code are "`TTTA`" comments (Things To Think About). These will either help with your implementation of the simulator or your understanding of the M3 - or both! Be sure you can answer all the `TTTA`'s by the time you finish the project... (even, or especially, those from other people's sections)

Hint: `grep -nR TTTA *.s *.c *.h` *to find all of them*

4.4 Missing Information?

At times, you may feel like you don't have enough information. *You probably don't.* A key part of this project is learning how to find resources and how to extract the information you need from them. As an example, missing from this document is exactly how to decode instructions (such as `0xb083` to `sub sp, #12`).

Use the resources available to you to try to figure out missing information. Also do not hesitate to use the mailing list, both to ask questions and to share resources you think others may find useful.

5 One Last Thought...

This project is a bit of an experiment. We hope things go well, but please be understanding if things are a bit fluid. To that end, email frequently, ask questions!

6 Document Revision History

- Revision 1.2

  Added section 4.1: First Steps

  Added section 4.2: Debugging

- Revision 1.1

  Add svn email notice

  Added explicit 16-bit instruction requirement to led.s

  Dropped the 16-bit instruction only limitation

  Clarified svn checkout

  Cleaned up document spacing

- Revision 1.0a

  Fixed typo in svn+ssh command

- Revision 1.0

  Initial Release