ARM7TDMI Simulator Report

Computer Science 310: Microprocessor Architecture

Rhys Fuller

December 9th, 2023

Table of Contents

Introduction

This assignment was to build a functioning ARM7TDMI microprocessor through software. It is capable of loading and storing memory with a simulated shell, decoding actual ARM7TDMI encoded instructions, working with simulated flags and branches, and completing the fetch-decode-execute cycle for entire executables written in C and compiled for the ARM7TDMI microprocessor. In order to properly decode ARM7TDMI syntax, the ARM DDI 0100E ARM Architecture Reference Manual, the exhaustive guide to the ARM7TDMI syntax, was consulted.

Features

In short, the simulator is capable of fetching, decoding, and executing basic instructions from the ARM7TDMI instruction set. The specific instructions supported are as follows:

- MUL (Multiply)
- AND (Binary AND two numbers)
- EOR (Exclusive OR two numbers)
- SUB (Subtract two numbers)
- RSB (Reverse Subtract two numbers)
- ADD (Add two numbers)
- ORR (Inclusive OR two numbers)
- MOV (Move a number into a register
- BIC (Bit clears a register with a mask)
- MVN (Moves the logical one's complement of a number to a register)
- LDR (Calculates a memory address and then loads the value at the address to a register)
- LDM (Loads multiple values from a calculated memory address to multiple registers)
- STR (Calculates a memory address and then stores a value at the address)
- STM (Stores multiple values from a calculated memory address to multiple addresses)
- SWI (Software Interrupt to notify the shell)
- CMP (Compare two numbers and set flags)
- B (Branch to a calculated address)
- BL (Branch and link to a calculated address)
- BX (Branch and exchange to a calculated address)

The processor is capable of running a set amount of cycles, or running until it hits a SWI 0 instruction or an instruction not yet implemented. The processor is also capable of performing accurate trace logs, and is capable of simulating the Program Counter (PC) at the ARM7TDMI's specifications of being 8 ahead of the address.

In addition, this program makes use of multiple coding languages. The shell, which was provided as part of the assignment, is written entirely in Python, which interfaces with a C program to hold the state of a register, which interfaces with a Rust program to complete the decode-execute portion of the cycle. The multiple interface design is an academic exercise to demonstrate the ability to work in complex software environments and use different languages for their strengths.

As an extra credit item for this assignment, I taught myself the Rust programming language, with my instructor providing guidance with issues I came upon, and making use of the books *Rust Programming for Beginners* by Nathan Metzler and *Programming Rust* by Blandy & Orendorff, and the online Rust documentation. This is why much of the actual functionality is implanted in Rust, to demonstrate a basic level of proficiency in the language in multiple low-level work areas.

Software Prerequisites

In order to run this software, one must have installed Python 3, Rust, and C. This software was ran and tested in Windows Subsystem for Linux Ubuntu, using the Python 3 that comes with WSL Ubuntu, the GCC compiler for C, and the Rust compiler that can be installed using the apt-get installer. CMake is necessary to create the makefiles and determine what the appropriate compilers to use is.

Specifically, Python 3.10.12, gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0, rustc 1.72.0 (5680fa18f 2023-08-23), and Ubuntu 22.04.3 LTS versions of the respective softwares were used.

Build and Test

In order to build and test the program, one can use the following commands in order on an Ubuntu command line. This assumes that Python 3 is installed on Ubuntu, as is the default.

- **sudo apt-get install rustc**
- **sudo apt-get install cargo**
- **sudo apt-get install cmake**
- **git clone https://github.com/RFuller25/cps310-simulator.git**
- **chmod -R +x ./cps310-simulator**
- **cd ./cps310-simulator**
- **cmake .**
- **make**
- **./run_tests.sh**

The final command will run the program on eleven different test programs and output the results. The first seven tests validate the functionality of various data manipulation instructions such as ADD and MOV, loading and storing instructions such as LDR and STM, and correct value, address, and program counter calculations for all of those commands.

The next two tests test the functionality of the compare instruction and branching instructions, and the final two are C programs compiled in ARM7TDMI. The first deals with local variables and makes significant use of the stack, testing the load multiple and store multiple and branching functionality, and the second deals with pointers, making significant use of address calculations.

All of the logs of the test are then compared to the answer key logs, and the percentage and ratio of their correctness is outputted on the screen.

User Guide

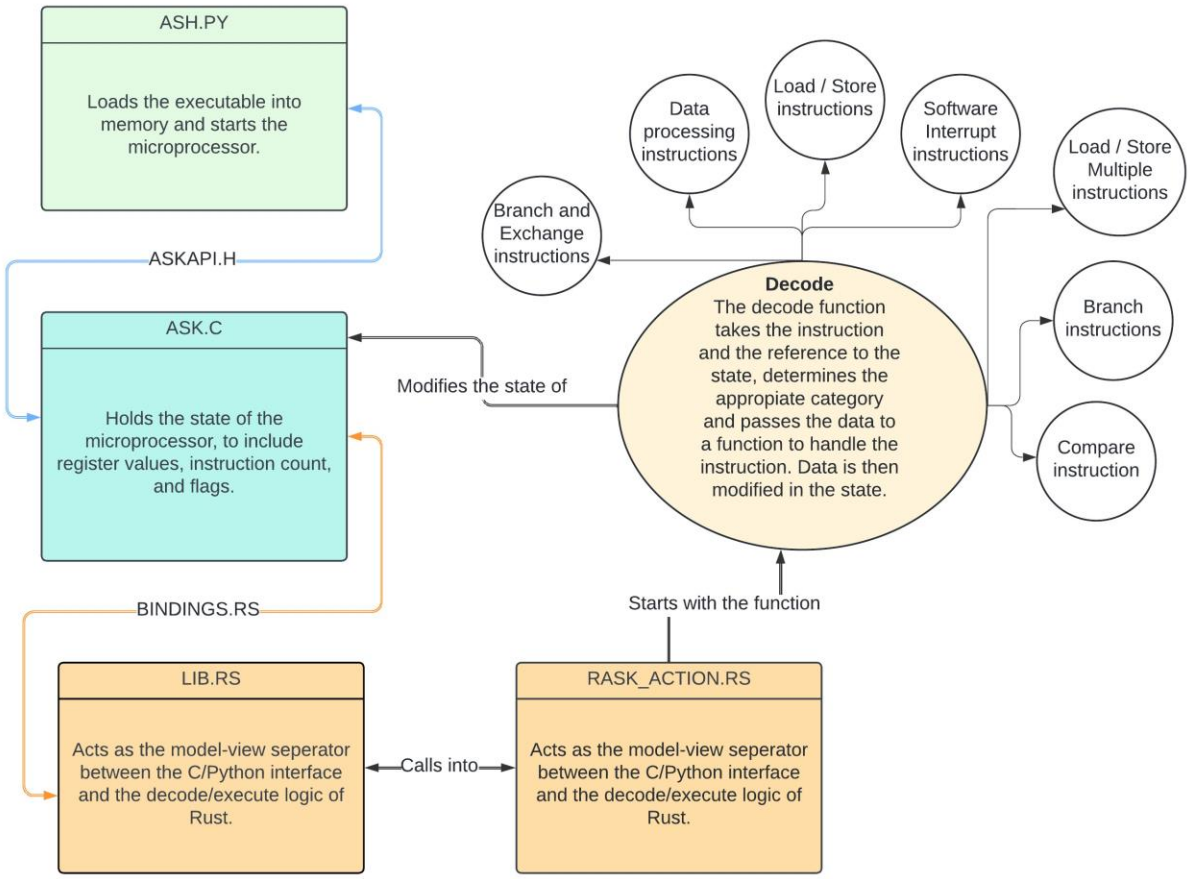
In order to run the program with a given executable, one can use the following syntax:

python3 ash.py -t LOG_FILE_NAME -c ./libask.so EXE_NAME

This will run an ARM7TDMI compatible executable through ash.py, which acts as the shell to load the executable into memory, and then runs the program using the program as the microprocessor.

Compatible executables can be compiled using the arm-none-eabi-gcc compiler to compile C programs or assembly programs.

Software Architecture Diagram



Software Architecture Description

The file **ask.c** acts as the entry point for commands from **ash.py**. **ask.c** calls into **lib.rs** to interface with the Rust files. The C implementation **ask.c** sends requests and I/O over to **lib.rs**, which sends it to **rask_action.rs**. **rask_action.rs** starts with the decode function, which determines the appropriate command and then passes it to the category function to execute the logic of the command and modify state appropriately.

Model-view separation is enforced with function calls from ash.py being routed through **ask.c** and **lib.rs** to the execute logic. The model is **lib.rs** as it calls functions from **rask_action.rs**, and it works independently of any other file. The view has no way of modifying state or doing work itself, but instead takes input and output and routes it to the model, receiving updates through state modification.

The Bindgen library was used to interface between C and Rust. This library looks through C types used in an C header file, in this case **askapi.h**, and creates Rust types appropriately, in this case **bindings.rs**.

This uses a chain-of-responsibility design pattern. **ask.c** is responsible for holding the state, and then passes commands from **ash.py** to the rest of the program and returns data to **ash.py** as appropriate. **lib.rs** runs the fetch portion of the cycle and ensures the cycle continues as appropriate, and then passes multiple commands to **rask_action.rs** to perform the decode and execute portions of each cycle, even when multiple cycles are needed from one command from **ask.c**.

Bug Report

- **Prototype Test One –** No differences from test data.
- **Prototype Test Two –** No differences from test data.
- **Prototype Test Three –** No differences from test data.
- **Prototype Test Four –** No differences from test data.
- **Prototype Test Five –** No differences from test data.
- **Prototype Test Six –** No differences from test data.
- **Prototype Test Seven –** No differences from test data.
- **Baseline Test One (branch.s) –** No differences from test data.
- **Baseline Test Two (cmp.s)** – No differences from test data.
- **Baseline Test Three (locals.c)** – No differences from test data.
- **Baseline Test Four (pointers.c)** – No differences from test data.

No omissions for all the attempted features, and no known bugs.

Appendix (Journal)

PHASE	TIME SPENT
ROUGH DESIGN	14 hours
REFINED DESIGN	20.75 hours
PROTOTYPE	14 hours
FINAL SIMULATOR	32 hours
TOTAL	**80.75 hours**

Digest for Rough Draft (September 25th)

I think the easiest parts will be fetching from memory and storing the registers and flags.
I think the hardest parts will be decoding instructions efficiently and abstracting parts in logical manner.
I estimate it will take me over sixty hours to complete this project.

Experimentation Notes (September 25th)

I think generally my design logic of abstracting parts will be optimal, and it seemed to work well for implementing the one instruction. Test program passed, and creating debug messages along the way I found that the system worked exactly as I intended it to (after some debugging and fixing) which was encouraging. The hardest part will be an efficient method of if/else / case statements to digest instructions, but I think with experience and further learning this will be more intuitive.

Rust and the Refined Design (October 10th)

After Dr. Jueckstock's help in getting the Rust and C to talk to each other, I think I am at the place where I am able to accomplish the CPU in rust, or at least make a good college try at it. I bought the book "Rust Programming For Beginners" by Nathan Metzler on sale, and I have been consulting it heavily. In addition, I have been making heavy usage of the [doc.rustlang.org](https://doc.rust-lang.org) (Rust Online Documentation.) Dr. Jueckstock is also letting me borrow “Programming Rust” by Jim Blandy and Jason Orendorff, and I am beginning to consult that book as well.

For specific problems that neither of those can help me with, I've been using stack overflow forums:

- https://stackoverflow.com/questions/56485167/how-to-format-a-byte-into-a-2-digit-hex-string-in-rust

- https://stackoverflow.com/questions/51571066/what-are-the-exact-semantics-of-rusts-shift-operators

- https://stackoverflow.com/questions/65261859/why-cant-i-index-a-u32-with-a-u32

I briefly consulted ChatGPT in an effort to get C and Rust to interface, and while it was able to help me with what I asked it for, I simply did not know enough about what I needed to do in order to ask the right questions. A transcript of the conversation and a brief commentary is available in the PDF file “chatgptrust.pdf”.

Estimated Timeline (September 25th)

- Rough Design - 9/25

    - +7 hours
    - Minimum working example
    - Focus on basics of design, namely, seperation of files

- Refined Design - 10/06

    - +10-15 hours
    - Basics implemented of well-thought out implementation to digest commands
    - Tests built and passing

- Prototype - 11/1

    - +10-15 hours
    - Using foundations already built, build it to pass all official tests.
    - Decide on EC options

- Simulator - 11/18

    - +20 hours
    - All tests are passing.
    - EC options are completed and functional
    - Self-made test that shows of EC and full functionality is built and functioning well.

Log of work

Date	Description	Time Spent
9/16	Worked on implementing rust for 4 hours, realized that I was not going to get it done in time, so I implemented the mockup in C instead, completed it in 2 hours	6 hours
9/23	Worked on Design, built journal.	1 hour
9/24	Finished design, set up Github, started working on rough draf	4 hours
9/25	Polished design (apparently not finished), implemented one instruction	3 hours
9/30	Worked on trying to get Rust to interface properly, realized it was a bit complicated	2 hours
10/02	Consulted multiple websites trying to figure out how to get rust to send a char star star. Discovered the wonders of bindgen, and tried to get it to help me.	2 hours
10/03	Tried to find alternative ways to passing it, including CString. Got a char * working, and was experimenting with muts and mut muts to try to get that to work.	2 hours
10/04	After hitting many brick walls, decided to consult ChatGPT and ask it to help me find a way around interfacing.	2 hours
10/05	Worked with what ChatGPT gave me, tried to get it to work and coerce the code into getting something close, was unable to.	2 hours
10/06	Tried to start the C version, realized that was not going to happen by deadline	1 hour
10/09	Tried to start the C version, was too sick. Dr. J mentioned he would help with Rust.	15 minutes
10/10	Got the Rust code, worked on it but had to prioritize other homework.	30 minutes
10/12	Read through the rust code to try to get to a point where I understood it well. Got the code up to the rough design portion.	2 hours
10/13	Got the code to the point it is currently. Implemented the other MOV instructions and the LDR, went through the design instructions.	5 hours
10/27	Belatedly began work, due to conflicts with other obligations. Kept running into issues with the BorrowMutHandler not being able to be passed multiple times, once it was unwrapped it was over.	2 hours
10/30	Did some more work on it, tried putting the BorrowMutHandler into a Box(), but that was not working. Decided to ask Dr. Jueckstock for help getting the Rust to cooperate.	2 hours
10/31	Rust was cooperating well, I moved to implementation. Initially implemented all of the code but ran out of time to figure out how to get the program to actually start working, due to a shift for the Vintage where I had to work a different 3 hours	7 hours
11/1	Dr. Jueckstock helped me figure out the issue with the mut handle that was causing me to be unable to properly run the program. Once I had that finished, I was able to debug and work on getting everything ready and accounted for. Unfortunately, due to more time needed for the Vintage, I was able to get all of the work needed done and had to start working on the report with multiple bugs left	5 hours
11/20	Worked on debugging some of the bugs left from the prototype.	1 hour

11/21	Further work on debugging some of the bugs. Built the run_tests.sh file to aid with testing.	1 hour
12/4	Began further work on debugging, reacquainting myself with the code. Unfortunately, not much time was available due to everything being done for the end of the year.	1 hour
12/6	Having completed all of my other work for other classes, began work in earnest. One by one I debugged the failing prototype tests and isolated the incorrect places. In the instance of the load/store logic, this involved removing all of the logic and starting from scratch.	8 hours
12/7	Started on the branch instruction set. This involved rehauling the way PC was calculated. Instead of working with a PC that was reported to be constantly ahead 8 bits, I refactored it to actually be ahead by 8 bits, and instead simply report the last used PC, which would show correct trace logs while making branch instructions possible.	5 hours
12/8	Completed the compare instruction set. This was largely difficult due to failing tests due to Rust's checked arithmetic. Since overflow is not possible in Rust without an error, I consulted with Dr. Jueckstock to find ways to perform unsafe arithmetic in Rust. After this was figured out, I was able to finish the compare instruction set, which allowed me to finish debugging the branch instruction set. Both branch.s and cmp.s were finished debugging at this point. Later that night, I was able to finish debugging locals.c.	10 hours
12/9	Finished debugging pointers.c. The issue with pointers.c was another impossible overflow issue, that required adding more unsafe operations in order to solve. Removed all debugging messages from the code, formatted the code, and uploaded the code to the repository. Wrote this report.	6 hours