Jaci Reichenberger, A20131719

Stephen Potter, A11341625

ECEN 4243: Computer Architecture

1 March 2021

Lab 2: Instruction-Level ARM Simulator

Section 1: Introduction

The purpose of this lab was to write a C program that will be an instruction-level simulator for a limited subset of the ARM instruction set. This program will allow users to run ARM programs and see the outputs. The objectives of this lab include introducing the software and process for running code, compiling in C, and introducing the ARM ISA. Several input files are given to test the code that is written. The shell and the simulation routine are the two main sections for the simulator. The goal is to write code for and implement the simulation routine, as the shell code is already given. Code was written within the sim.c and isa.h for the data processes, branching, and memory. The data processing and branching instructions were written in full, but the memory coding was not due to lack of knowledge for the lab, rather, lack of time to completely implement what this lab required. However, the results of the code that was written were able to be compiled and used with the inputs, which resulted in a working ARM Simulator. Thus, this lab taught students how to design an ARM Architecture Simulator based in the C language.

Section 2: Baseline Design

The baseline design for this lab was creating code for sim.c and isa.h in order to implement the ARM Architecture. This code was broken into three main sections: the data processing, the branching processes, and the memory processes.

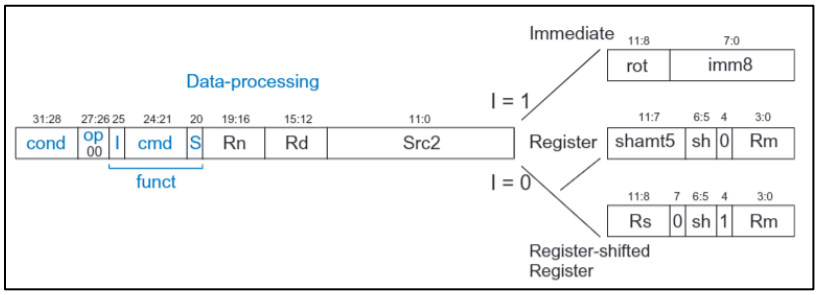

Figure 1: Data Processing Instructions

As seen in Figure 1, the data processing commands used this basic template in order to differentiate between them. Each command had their own distinctive cmd, which allows the code to call on each separate function as needed in the sim.c in order for it to execute in the isa.h. In the isa.h, the commands are further broken down into Immediate, Register, or Register-shifted Register. It is Immediate if I = 1, a Register if I = 0 and bit4 = 0, and a Register-shifted Register if I = 0 and bit4 = 1.

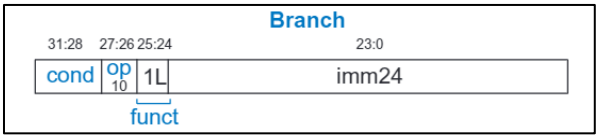

Figure 2: Branch Instructions

Figure 2 depicts the basic structure for branch instructions. The branch instructions were similar to the data instructions in that L was used to differentiate between branch and branch with link. Then, it was determined if the branch was equal or not based on if the Zero flag was high or low.

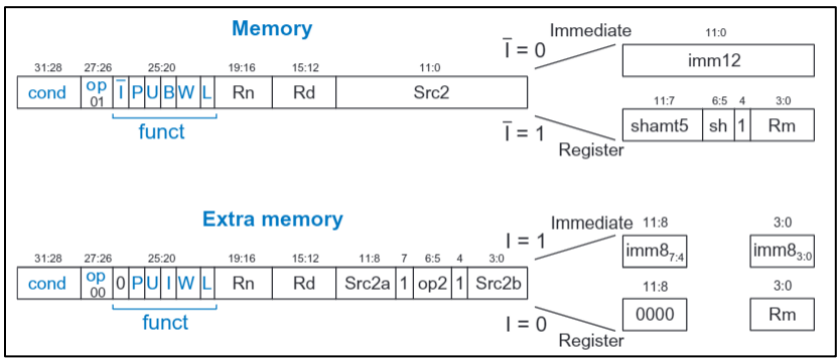

Figure 3: Memory Instructions

Figure 3 shows both the Memory instructions and extra memory structure. Again, this is similar to the data processing instructions and the branch instructions. However, the memory instructions have the I, P, U, B, W, and L bits. These bits determine what type of memory command will occur. The I is for whether or not it is immediate and the L distinguishes between a load and a store, for L = 1 and L = 0, respectively. The other bits are explained in detail later on in the report.

The data processing instruction encodings that were written included Bitwise AND (AND), Bitwise XOR (EOR), Subtract (SUB), Subtract with Carry (SBC), Add with Carry (ADC), Test (TST), Test Equivalence (TEQ, Compare (CMP), Compare Negative (CMN), Bitwise OR (ORR), Move (MOV), Logical Shift Left (LSL), Logical Shift Right (LSR), Arithmetic Shift Right (ASR), Rotate Right (ROR), Bitwise Clear (BIC), and Bitwise Not (MVN). The Add (ADD) code was already given in the code provided. The branch instructions that were included were Branch (B), and Branch with Link (BL). Finally, the memory instructions that were included were the Store Register (STR), Load Register (LDR), Store Byte (STRB), and Load Byte (LDRB). The Software Interrupt (SWI) was already given in the code provided. All of these commands were implemented as specified in the ARM Architecture Reference Manual. For each command, the condition flags needed to be checked, as well as both the immediate and register operations had to be implemented. The framework for this code was given, as well as inputs to test said code. After the code for all the commands were written, it was compiled until all errors were fixed. Then, using the inputs and arm2hex, the code was tested to see if the code worked as intended. Then, the code was refined and commented on once it was believed that code was working as expected.

Section 3: Design

The design for the data processing instructions was implemented by observing the code provided and parsing through the instructions to understand how the instructions would be handled. Taking into account the structure of the code, the data processing functions were written to follow the data path and provide the function they were designed for. The data processing instructions were designed after examples that were provided, and from there molded to all the other processes that were needed in the isa.h file. The sim.c file provided the structure for the simulation of the hex files that had been converted from ARM assembly instructions to hex files to be passed into the simulation executable. Starting at the process instruction section of the sim.c file, the program would get a set of instructions from a hex file and it would be read into memory and then passed to another function named decode_and_execute. This function then parsed the instruction further to determine whether it was a branch, multiply, data processing, data transfer, or software interrupt instruction. From there, the simulation file would take those hex files and parse them to figure out which function to use and how to display them. The details of this operation are then handled by the functions that are written in the isa.h header file.

In the isa.h file, the functions for all of the data processing, branch, multiply, and memory instructions were written out and implemented. These functions took a 32-bit instruction and passed it into several variables that told the program what operations to do and how to handle the data depending on whether certain bits and flags were activated. With the I flag set to 1, the corresponding functions would execute immediate instructions. When the S flag was set high, the program knew that the instruction was signed, and therefore checked for four particular flags. These flags were Negative (N), Zero (Z), Overflow (V), and Carry (C). The N flag is set by an instruction if the result is negative, the Z flag is set if the result is zero, the V flag is set if the signed result overflows the 32-bit result register, and the C flag is set if the unsigned result overflows the 32-bit register.

The basic layout for the data processing instructions in the isa.h file were to first figure out whether it was immediate or not. If it was immediate, then the immediate instructions were initiated. If it was not immediate, then bit 4 was used to find if the instructions were a register or a register-shifted register for bit4 equal to zero or one, respectively. Then the code for the registers were initiated. Finally, the flags were checked for these cases and set as needed. The biggest variation to this layout lies in the shifting instructions, as MOV, LSL, LSR, ASR, and ROR are differentiated with the sh bits and whether or not I is a one or zero.

The basic layout for the branching instructions is to implement the offset. After that, the instructions were broken down to whether or not the branch was equal or not, based on if the zero flag was set or not, then the branching instructions were implemented. Likewise, the basic layout of the memory instructions was set for its various bits. First, it checked to see if it was immediate or not, then it checked for P. P = 0 means that it will use post-indexed addressing, and P = 1 indicates the use of offset addressing, or pre-indexed addressing. For P = 0, if W = 0, the instruction is LDR, LDRB, STR, or STRB and a normal memory access is performed. However, if W = 1, the instruction is LDRBT, LDRT, STRBT, or STRT, and an unprivileged memory access is performed, though this is not required for this lab. For P = 1, if W = 0, the base register is not updated for offset addressing. If W = 1, the calculated memory address is written back to the base register for pre-indexed addressing. Then the U will determine if the offset is added to the base or subtracted from the base, for U = 1 and U = 0, respectively.

Each of the functions have a particular operation they implemented in the program which allows for data processing, memory manipulation, branching, and multiplication when implemented. The multiply instructions were not required at this time, so they were not implemented. The memory instructions LDR, LDRB, STR, STRB were also not implemented due to the complexity of the code and the setbacks from weather and lack of understanding until too late in the process. There is code that has been introduced into those sections that was attempted; however, it is still incomplete and does not work.

Section 4: Testing Strategy

The testing strategy for this program was to use the ‘make’ command, which compiles the files pointed to in the makefile. When it compiles these, the first time the src code is built into an application and if any subsequent changes are made to the target files then the make command only compiles the files that have been altered. Then, the arm simulation is tested by running the application in a terminal with hex files passed in as instructions for the program to follow. The shell.c file helps to format those instructions and asks the user to input the number of cycles to run and to quit the shell when finished checking the implementation. The user can then check the value of the registers with the rdump command and observe the assembly files to check whether the simulator is performing the operations correctly. This is repeated until each of the test conditions are checked and verified. After each case has been verified, the testing is concluded, and the results are recorded. When the tests were performed, the results were read by checking the registers with the rdump command. The results seen in the registers prove that the instructions were implemented as intended. All of these tests were performed with the exception of the memory test as the memory functions were incomplete.

Below is an outline of the tests included in the verification of the simulator:

- Addfirst.s - This test implements an MOV instruction and then an ADD instruction.
- Addiu.s - This test implements several MOV and ADD instructions.
- Andor.s - This test implements the MOV, ADD, ORR, CMP, B, and EOR instructions.
- Bltest.s - This test implements the branch with link instruction.
- Brtest0.s - This test implements the branch instruction.
- Memtest0.s - This test implements the memory manipulation instructions such as LDR, STR, LDRB, and STRB.

Section 5: Evaluation

The results from this code can be seen by running the code with the inputs provided. All in all, the results of the completed code work as intended. The results from the inputs supported this conclusion. Observing the first test, it is revealed that the simulation program properly moved the number 5 to the first register, then proceeded to add 300 to the value in register 1, and then stored that result into register 2. After completing those instructions, the simulator received an interrupt instruction and stopped the processing. By this behavior, the program shows that it is behaving properly. This is likewise true for the rest of the simulations stated above.

There were several setbacks while working on this lab. The first and foremost problem was a complete lack of understanding of what this lab was about. Of the three weeks working on the lab, it took until the last week to fully understand what this lab was wanting. It did not help that snow days, as well as virtual labs and power outages severely limited working on and attaining help for this lab. Another major problem for this lab was that Cygwin was not downloading correctly on either computer, nor was it on the lab computer to test the designs. Luckily, this issue was resolved, and one of the personal computers was finally able to download it, but very late in the process. Due to these problems, not everything could be completed. Luckily, it was only one section of the code: the memory instructions. The coding for these instructions were implemented in sim.c, but not initialized in isa.h, as the code for these could not get compiled to be tested.

This lab taught many things. Firstly, that understanding the premise of the lab is paramount to being able to write the code and complete the lab. Secondly, how various instructions for ARM are created and implemented in order to simulate ARM Instructions. Thirdly, an interesting lesson that came from this lab was the levels of detail required to create all the different parts of the software that make up processing instructions. It showed the many layers that an instruction has to go through in order to be parsed into usable instructions and information. Building this architecture shows the software side of creating a working ARM architecture system. All in all, this lab was a steep learning curve, but students were able to learn how to design an ARM Architecture Simulator written in C language.

References

Harris, D. (2007). *Appendix B*. *Digital Design and Computer Architecture: ARM Edition* (535-540). Massachusetts: Morgan Kaufmann.

ARM Architecture Reference Manual. (2005). ARM Limited. Retrieved from: https://canvas.okstate.edu/courses/98722/files/9628839?module_item_id=2877589.