ECE 212: Microprocessors, Hardware, Software, and Interfacing

Final Report: Understanding the ARM V7 and Cortex A9 Processor

Karysse Hay
College of Engineering
University of Miami
Coral Gables, United States
kdh85@miami.edu

***Abstract*— This report aims to explain the process of how we learned to successfully code in ARM through the multiple labs mentioned. It will also touch on why these skills are important and how they can be used in future endeavors. It will begin by giving a detailed explanation of the basics of the ARM processor and how we got started with the DE1-SoC board. Learning ARM assembly allows for the opportunity to completely understand the systems that enable high level computer actions. In the end, I can safely say that I have a much better understanding of ARM than I did before I enrolled in this class. With this knowledge, there is a better comprehensive understanding of the ARM architecture and microprocessor. In addition to this, the skills learnt may prove to be useful further in the computer engineering field.**

Keywords—ARM, DE1-SoC board, comprehensive, architecture, microprocessors

I. Introduction

The importance of this report is to holistically capture the useful insight gained from this course and the five labs assigned throughout the semester. This will allow for the main discoveries and results to be recorded in one document for easy reference. It will also attempt to explain my general thought process as I completed this course

Throughout this semester, we learned multiple aspects of the ARM V7 and the Cortex A9. ARM V7 is the basis for all ARM Cortex Processors including Cortex- A9. In Electrical and Computer Engineering. The ARM architecture plays a huge role in any computing or communication market. For example, it can be found in smartphones, personal computers (PCs) etc.

There are many benefits to learning the code for the ARM Processor. Some of the benefits explained to us in the lecture include writing assembly code generator for a compiler, interrupts and booting the computer etc. It is also extremely beneficial to anyone interested in working with computers, whether it be in software or hardware. Learning skills such as optimizing your code, debugging and programming complex tasks are very useful for this field.

.

II. Methods/ Approach

The program introduced to us was the Altera/ Intel Monitor Program [2]. This program can be used to teach computer organization and embedded systems without requiring knowledge of hardware design. Before beginning any of the labs, I looked at two sources of information, the DE1-SoC User Manual and the lecture slides, to better grasp the concepts that will be used in the labs [3]. This was done for each lab, and I can safely say that it assisted tremendously in the flow of my thought process when writing the code.

One of the first things that was taught to us was understanding how to use the software interface of Altera/ Intel Monitor Program and setup the programming environment. In order to do this, the AMP was launched, and a new project was created. The directory and the name of the project was selected and named. The architecture was also selected to be ARM Cortex-A9 as this was the architecture that would be explored further in this course. The DE1-SoC Computer system was selected, and the program type was set to Assembly Language. This was done whenever the lab required us to download it to the DE1-SoC board [3].

Understanding how to set up the program for the labs played a very important role in everything that we did after that. However, for one or two labs, the online emulator was used which did not require any of this as it had a 'built in' DE1-SoC board. This reduced the setup time substantially.

The first lab allowed us to explore the developmental board and the processing capabilities, and the software required to operate the board that will be used throughout the semester. The sample code given was copy and pasted into a text file so that it can be compiled and downloaded to the ARM Cortex A9 Processor. At the end of this first lab, the memory address and registers were available for viewing. This was our first introduction to it and came in handy for what came after this lab. The memory address and registers were further studied in other labs.

The next important aspect of the ARM Processor/s architecture is its Memory and Addressing section. The memory is a high-level description of how the processor should interact with the address region. ARM V7 is a memory-mapped architecture, and this assists us in many labs as it allows us to access and store numerous pieces of information.

In class, the Professor began by teaching the definition of a bit, a byte, a word and a half word. There are 8 bits in a byte and four bytes make up a word. A halfword was considered to be 2 bytes. It was later learnt that these definitions would be crucial building blocks in everything that we did. This taught us that aligning our messages by byte/ word/ half-word could determine how they would be stored and displayed in the memory canvas [4].

As the labs progressed, and our knowledge deepened, we were then able to manipulate strings in memory. For this, we first had to fully understand the 'ldr' and 'str' instructions

likewise with the ‘str’ instruction. These were important instructions and were used in almost all the labs. We then had to familiarize ourselves with other instructions like the ‘b’ and ‘cmp’ instructions. The ‘b’ instruction, when used, will load the pc with a new address. Whereas ‘cmp’ will perform subtraction and set the CPSR flags. With this, we would be able to compare bits to unique characters from either the ASCII table or other sources and therefore store the message in memory different ways using the outputs from the comparisons made [6].

After memory and other software features of the processor had been explored, we then moved onto how to interface with peripherals that are connected. This included the LEDs, switches and push buttons. For this lab, the ‘.equ’ directive proved itself to be useful. This was used to equate the base addresses of the LEDs, switches and push buttons to a simplified name so that it can be referenced in the code easily. The basic instructions such as ‘ldr’ and ‘str’ were continued to be used in all labs as this was the only way, that was exposed to us, to gain access to anything [4].

For the last lab regarding interrupts, the lab report was set up very differently to anything we had seen before. This lab required us to fill in the blanks in some code but first we had to fully understand what the code was doing. After going through the lecture slides on interrupts and help from our TA, we were able to pull apart the code and distinguish what each part was trying to accomplish. With this, we were able to fill in the blanks correctly.

As mentioned before, with the knowledge from lectures and the given slides, it was easier to approach the questions in the lab. In addition to this, our TA would draw images to illustrate what should take place in the labs so that we can better visualize what needed to be done. This helped a lot as majority of people are visual learners.

III. Results and Analysis

This section will go into detail on what each lab resulted in and how they connect with one another.

For reference, there are eleven general-purpose registers. These registers span from R0 to R10 and the remaining registers are referred to as special registers that have pre-determined tasks. The special registers are as follows, R11 is the frame pointer, which tracks stack frame, R12 is the instruction register, R13 is called the stack pointer which holds the address where the stack ends, R14 is the linked register that holds a return address for a subroutine if there is a branch instruction being used and the final special register, R15 is the program counter. This hold the address of the next instruction that is to be executed [8]. The registers can be seen in Figure 1.

Since the first lab was nothing more but an introduction to the software interface of the DE1-SoC board, all of the registers mentioned in the paragraph before were made visible to the user. The memory addresses were also visible to me. This allowed me to have a better understanding of what I will be working with in the upcoming labs.

The next part in my ARM education journey involved the loading of characters into memory. After having seen what the memory tab looked like in Lab 1, this was very confusing to me. This is because what I had first been introduced to was multiple lines of addresses, not actual words. I later learnt that I had to right click inside the memory canvas and select “Show ASCII equivalent characters” and also select “view as byte” [8]. Only then was I able to view the message that was my name in a more desirable fashion. This can be seen in Figure 2.

Also, when storing a message of characters as different number of bits, such as a byte, a word and half-word, it was obvious that the size allocated to the message affected the way it would display in memory. It might appear on multiple lines or begin in the middle of a line. This is because memory was grouped into words and each line increased in intervals of 4. At first, this was not known as it was still our first experiences with the memory section therefore the message did appear on two lines even though it could have easily fit on one.

In class, the “.align” directive was taught to assist in the lining up of words. This made it possible for the message to be byte, word or half word aligned [8]. This code was played with for a bit to figure out the most efficient way to align the message with the least number of instructions. The output of this can be seen in Figure 3 where each set of characters began on a different line and at the beginning of that line. At this point, the memory of ARM became a more familiar to us. We were able to write messages, view them in memory and align them just the way we wanted it to be.

The next lab was the turning point in my opinion. This lab had the most new content which at first was scary but after our TA explained the goal of the lab in detail, it did not seem that bad. We were required to copy a message into memory but to omit the spaces. To do this, we had to, in a sense, create a pointer that would run through the message and compare each character with the ASCII value for a space. When there was no space found, the character would be copied over whereas when there was, the pointer would just increment by one thus skipping over the space. This was done using ‘cmp’ and ‘b’ instructions. The CPSR also played a huge role in this lab. The CPSR is an abbreviation for the Current Program Status Register. This allows for the program to access information about the current program and/or result of the previous operation. In other words, using add-ons such as ‘eq’ and ‘ne’ to branch instructions, allowed for the program to branch to different instructions granted that the flags for ‘eq’ and/or ‘ne’ were set [6].

At this point, we were fairly familiar with load and store instructions and also using the CPSR to our advantage. With these specific lab requirements, we also had a fair amount of time experimenting with loops in ARM. This was a very harsh transition from me, especially coming from coding languages like C++. C++ is the first and only programming language that I have been exposed to. Therefore, being able to construct loops outside of the C++ coding structure was a new experience for me. Learning these loops was indescribably important for the next ask.

For the next part of this particular lab, extreme manipulation of messages in memory was asked of us. We had to print the string either forward or backward depending on

what the first letter was. This was very intimidating at the beginning because what the lab report asked seemed almost impossible. After collecting all the ASCII values for each of the comparisons, the code was drafted [6]. This code needed six loops. At the time, that seemed like a lot. Now, I feel like the organization of my code could have probably been better and the code could have been optimized more. The storing of the string "eI am a Cane" backwards can be seen in Figure 4. It was stored backwards as I inserted a lower case 'e' to the front of the original message.

For the second to last lab, instead of interacting with the memory of the DE1-SoC board, we would be directly interacting with the peripheral components, namely the LEDs, switches and pushbuttons. Using the knowledge that we have already gained from previous labs, this lab seemed like one of the easier ones. The emulator was also used which limited the set-up process.

Here, we were tasked with conducting operations and displaying the inputs and outputs on the LEDs whilst using the switches. We read two values from the switches and stored them into two registers. After we mastered this, we then had to read a third number that would determine the operations. The operations already had numbers assigned to them as mentioned in the lab instructions. These operations included: addition, bitwise AND, multiplication and logical shifting left [6]. When the third register was stored, the operation would be read, and the code would branch to its specific instruction. The result would then be displayed on the LEDs and if the result was zero, all LEDs would have turned on.

In this lab, the biggest takeaways were how to read a value from the switches and display it using the LEDs. The operations did not seem too difficult as I am also enrolled in ECE 315, in which some of these concepts were already exposed to me. The hardest part of this part was understanding the flow of each operation and the storing of the different values without overwriting anything. Because of this, the lab also required us to create a flowchart displaying the flow of logic for the code. After completing this chart, it was apparent to me that it should have been done before the code as the flow of thinking would have been clearer and more straightforward. This would have saved time in the coding process. This flow chart is visible in Figure 6.

Finally, our exposure to interrupts was solidified after completing the last lab. We were given many files with code. Some lines of code were omitted, and the task was to complete it in order for the interrupt to run successfully. The lines missing required simple coding. However, the most difficult part was understanding what the entire file was trying to accomplish. The code had to be looked at a few times to figure this out but after we fully understood what the code needed; it was very easy to complete the missing parts.

Additionally, we were required to write a bit of code for the Key interrupts. With the hints given in the lab instructions, this did not take up too much time. The Keys were compared to binary values 1 and 2 and if they were equal, the code would branch to the Key interrupt telling the program that one of the buttons were pushed [5]. The output of the code written can be seen in Figure 7 and 8.

A. *Figures and Tables*

a) Preparation

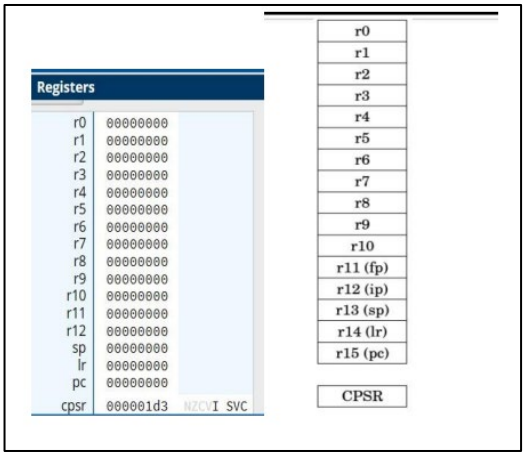

Figure 1: The Registers in the Altera/ Intel Monitor Program

In the Figure above on the left, the registers can be seen as it would look like in the program. R0 t R11 are general registers and R11-R15 are the special registers. The abbreviations for the special registers can be seen in both photos.

b) Lab 2

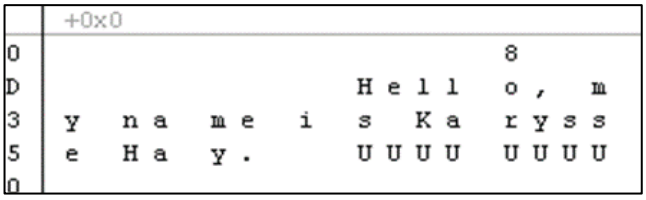

Figure 2: Message visible after changing the settings in the Memory Canvas

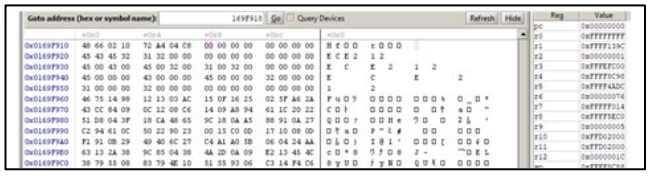

Figure 3: After the charcaters ECE212 were alligned according to their respective sizes

c) Lab 3

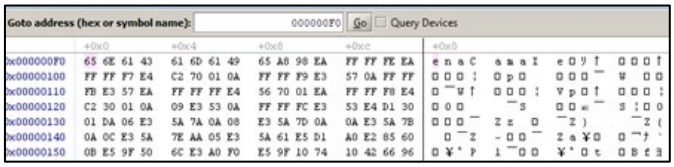

Figure 4: The storing of sting "I am a Cane" backwards as the first character was a lower case letter

As mentioned, the message was changed from "I am a Cane" to "eI am a Cane" so that the program could have been checked if it worked. The Figure above shows the message printed backwards at the same destination address. The code was successful in checking if the first character was in the lower-case range of 61-7A and stored the message backwards thus showing "enaCamaIe" at the destination address chosen.

d) Lab 4

Figure 5: Addition performed on the Emulator using the switches

The result on the LEDs is 0000001010 in binary. This was because the first two numbers inputted was bx0000000010 and bx0000000011 which is 2 and 3 in decimal. This can be seen in the registers 3 and 4. The operation selected was '00' so the value of X can also be seen in R6 which is 0x5

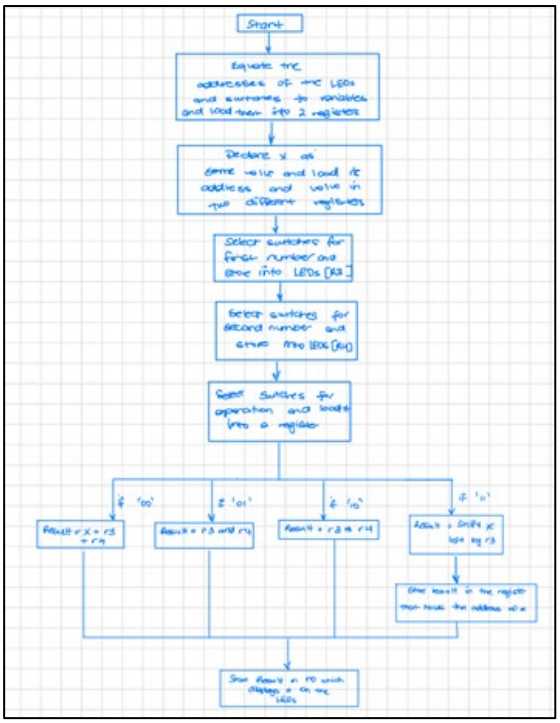

Figure 6: Flow chart depicting the code for Lab 4

e) Lab 5

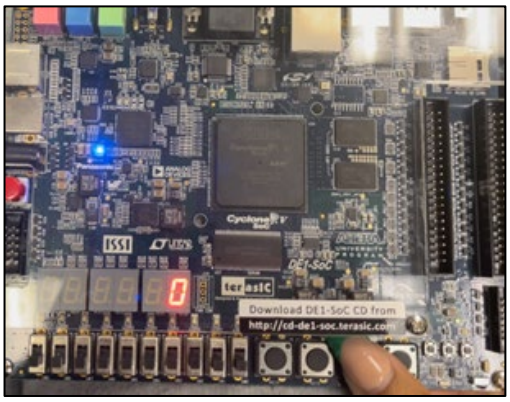

Figure 7: Digit zero displayed on HEX 0 display

In the Figure above, the digit zero is displayed when KEY0 is pressed and blank when it is pressed again.

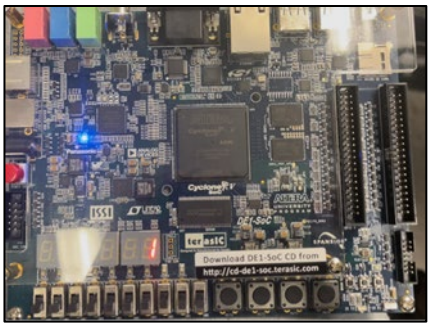

Figure 8: Digit one displayed on HEX 0 display

In the Figure above, the digit one is displayed when KEY1 is pressed and blank when it is pressed again.

IV. Conclusion and Future Work

The objective of this course was to gain a holistic, comprehensive understanding of the ARM architecture and to gain insight on solving complex tasks using assembly language. The ordering of the labs went hand in hand with the topics taught in the lecture. This assisted in the preparation for each lab.

The Altera/ Intel Monitor Program was sufficiently easy to use and to follow. Our TA also played a part in fully grasping the concepts at hand. They were very helpful and responsive. If a problem were to occur, I would simply ask the TA to explain it to me further or I would refer to my class notes and/ or lecture notes. The slides from class proved to be very useful, not only when writing this report, but during the labs as well. They contained useful and relevant information that pertained to the labs and any question I may have. The relevant textbook also assisted when studying for exams and/or quizzes [1].

Success in this class was not defined by finishing the labs on time but instead, understanding each concept efficiently and effectively. For me, drawing pictures and flow charts proved to be most beneficial as it helped me to better visualize what needed to be done and how to do it.

With this new gained knowledge, I have a better comprehensive understanding of the ARM architecture and assembly language. I feel empowered in my ability to approach problems in assembly. This process of going through the labs one by one has left me with the capacity to debug and code with better efficacy. Now if I was ever required to work on a project that required Assembly knowledge or encounter a question pertaining to assembly in an interview setting, I will be fully equipped with knowledge and skills learnt from this course.

References

[1] Pyeatt, Larry D. Modern Assembly Programming with the ARM Processor. Newnes, 2016.

[2] “Intel FPGA Monitor Program Tutorial for ARM”, Intel FPGA, November 2016.

[3] “DE1-SoC User Manuel”, Terasic Technology, August 5th 2015.

[4] H. Asafrjalani, “Basic ARM architecture and Assembly Instuctions”, 2021 pp. 1–25,

[5] H. Asafrjalani, “Interrupt Handling”, 2022

[6] H. Alsafrialani, “Integer Math”, 2022

[7] Wikipedia, “Assembly Language”, Edited May 2022.

[8] H. Asafrjalani, “GNU Assembly Syntax”, 2022