Introduction to ARM Programming - Laboratory #1 Report

ECSE 324 - Computer Organization

October 13th, 2018

Sarah Swanson - 260660847

Bryan Jay - 260738764

Group 19

Introduction

The purpose of the first laboratory in Computer Organization was to learn how to work with an ARM processor as well as understand the basics of ARM assembly programming. We used the DE1-SoC Computer System which contains an ARM Cortex-A9 processor, and peripheral components on the FPGA on the DE1-SoC board. The FPGA Monitor Program 16.1 was used for the IDE. After the initial setup of the system, we were given the task to write four common routines in ARM assembly language: finding the largest integer value, standard deviation computation, centering an array, and sorting.

Part 1 - Finding the Largest Integer Value

The purpose of part 1 of the laboratory was to learn the initial basics by copying and testing the provided code within the lab document. This program finds the largest integer value by taking an input of a set of numbers in a list, the length of the list, and shuffles through the list to find the largest integer value in the list.

The code given to us for the program was very specific. There was a maximum register that was initialized to the start of the list, having the purpose of storing the largest number in the list. There is then a comparison with the next consecutive value in the list and the current maximum number. If this next value in the list of integers is larger than the current maximum number, this value is placed within the register and is now the current maximum number.

There were no challenges coding this section because the code was given to us. We took the time to understand the code with the help of the teaching assistant. We learned how to use the Altera Monitor Program with the debugging mode.

Part 2 - Some Programming Challenges

After learning the basics of ARM assembly programming with an example, we were given the task to write our own programs. The details of the programs are explained below.

Fast Standard Deviation Computation

There are two ways to program standard deviation in ARM assembly code. One involves multiplication, division and square root operations which is not available as instructions on all of the processors.Therefore, the task to write a program for the fast standard deviation computation uses a simpler, more hardware-friendly approach called

the “range rule”. This rule is a close approximation to the original standard deviation formula and is given by the formula below.

$\sigma \approx \frac{(\chi max - \chi min)}{4}$ where χmax is the maximum value and χmin is the minimum value of the signal

In order to accomplish this task, we used a loop to iterate through the array of numbers. The loop pointed to the next number in the list and loaded it into a register. This number was compared to the maximum to see if it was greater. If it was greater, update to make it the current max. If it was not greater, branch to another loop called “MIN_LOOP” to check if it was smaller than the minimum number. If smaller, update the minimum number. Branch back to LOOP after either case to keep decrementing through loop counter until it reaches zero and your desired minimum and maximum numbers are found. The next step is to compute the standard deviation by subtracting the minimum from the maximum number and then doing a logical shift right by 2^k (k = 2) bits. Store the maximum number into the result.

The biggest challenge when coding this program was the task of dividing the subtracted numbers by four. We ended up using a logical shift right by 4 bits. An improvement could have been to use one loop instead of two, which would cause the runtime of the program to be faster.

Centering an Array

The next task to code was centering an array. The purpose of centering an array is to ensure that a signal is “centered”, meaning it’s average is zero. In order to center a signal, you can calculate the average value of the signal and subtract the age from every sample. We were also asked to store the resulting centered signal ‘in place’, meaning store it in the same memory location that the input signal was passed in at. For the simplicity of this task, signal lengths could only be powers of two. The program should accept the signal length as an input parameter.

The way we approached this task was done in steps. The first step was to add all the values of the array to have a total value of the numbers. We stored the total in a register. In order to get the total, we created a loop that added each number one by one into into the register. The loop is stopped when there are no more values in the array. The next step was to calculate the average of the array. This was done by dividing the amount of numbers by 2 until you hit zero, then dividing one more time. This tells us how many times you have to divide the total number by 2. Then divide the total number by 2 that many times. Now that we have the average, we make a loop to subtract the average from every number in the list. Once the counter hit zero, the numbers have been cycled through and a centered array was created.

A challenge that we faced was a lot of testing to get the right loop counters. Since there was dividing of the number of array numbers, this counter was not known at first. Another challenge was making the code so nothing was fixed to a certain value because the array could be any length of powers of two. An improvement could have been to create an actual divide function so that any number of values could be in the array, not just powers of two.

Sorting

The final task to code was sorting an array in ascending order and store it 'in place'. We decided to use the bubble sort algorithm provided in the laboratory document. The approach we took was quite simple, using two main steps to sort the array.

The way we implemented the sorting algorithm in ascending order was to create two different functions: SORT and SWAP. The first step was to SORT, done by a loop that decremented the total number of items in the array until all of the items had been cycled through. Two numbers (starting with the first two values and with each decrement, taking the next two values in the array) from the array were placed in two registers and assigned addresses each loop cycle. These two numbers were compared to see if the first number was bigger than the second. If this were true, we went to the SWAP function. The SWAP function would store the first number in the second register and the second number in the first register, essentially swapping the numbers. Then we would assign the next two numbers in the array to addresses and into the corresponding registers. If this were false, the first number was not bigger than the second, the SORT function would continue and assign the next two numbers in the array to addresses and into the corresponding registers. The last step was to recall the SORT function until we had decremented through all the number of items in the array and the array was fully sorted in ascending order and stored into place.

A challenge that we faced implementing the sorting algorithm in ascending order was figuring out how to swap the two numbers. We decided to use two extra registers to hold the two address from the array and load the values from those addresses the two registers which were being swapped. An improvement that could have been helpful would have been to use a sorting algorithm with a better runtime (like merge sort). Bubble sort has a runtime of $O(n^2)$ and merge sort has a runtime of $O(nlogn)$.