Dalin Williams

Project One: MIPS and other assembly solution creation and write-up

Project 1 Write Up

MIPS, Accumulator, and Single Register

Table of Contents

[Algorithm Description 1](#_Toc414153352)

[MIPS 1](#_Toc414153353)

[General Components 2](#_Toc414153354)

[Single-Register 2](#_Toc414153355)

[Accumulator 2](#_Toc414153356)

[Data Collected and Comparison 3](#_Toc414153357)

[Conclusion 4](#_Toc414153358)

[Results and Reasoning 4](#_Toc414153359)

[Expectations 4](#_Toc414153360)

[Figures 5](#_Toc414153361)

# Algorithm Description

Here, we will describe the algorithms and their behavior. All figures referenced are present in the Figures section of this paper, as to avoid the bread-up and orphaning of information. Please note that since all of the algorithms are single cycle, we have equated the instruction count to the execution cycles.

## MIPS

The MIPS algorithm created followed in line with the specifications of the algorithm provided as the description. To gather the information pertaining to the running time, we used in-built statistics within the MARS simulator to gather the information shows in the Data Collection and Comparison section. For more information concerning the behavior of the simulator used, please refer to the following [link](http://courses.missouristate.edu/kenvollmar/mars/papers.htm). The general formation of the application followed the pseudo code presented in the project description, so we will document all further changes we had to implement on the respective models.

The MIPS Assembly began with the .data field which designates the staging area for the array to be sorted. The .data is followed by the declaration of the array, which we set to .word to hold 2 bytes of integer data for each of the one-thousand integers in the array. We then initialize the .text field to designate the instructions section of the application, and then specify the main function as beginning to the application.

The program starts in main and initializes the array’s address into a register, and then adds three-thousand nine-hundred ninety-six to the address to point to the end of the read array. The application then continues to the outer loop which initializes the isComplete flag, and re-reads the address of the array.

The next component, the inner loop, loads array element n-1 and n for comparison. It then checks if n-1 > n, and sets an indicator if it is. If the indicator is zero, the application then will end, as the item is in its right place, otherwise we move to swap the items in the swap function.

The next component is the continue tag, which increments the address pointer of Array, and determines the looping functionality of the application. If we are not at the end of the array, we continue to the inner loop, and if the indicator is not zero, we continue to the outer loop. If all of these are false, we exit the application.

The swap function sets the indicator checked in the continue function. This function also stores the first and second items in the opposite original locations in memory, thereby swapping them. The function then exits and returns to the previous location in memory.

The application continues through all one-thousand elements in the array until the array is sorted.

## General Components

The shared components include the memory list, general application interface, and the file reader.

Upon the application’s start, the user is prompted with several questions pertaining to the desired method of sorting the five available arrays. Once the user selects an array to be used in a method provided, the application then transforms this data into a ‘memory’ list and launches the requested parser. Upon the parser’s conclusion, the user is presented with the execution time in milliseconds, along with the option to continue sorting arrays. Once the user selects exit, the application closes.

For a visual aid to how the entirety of our solution was set up, please refer to Figure 1: The general execution of our application please refer to Figure 1 and Figure 2 in the like-named section.

## Single-Register

For the Single Register simulator, we split the functionality of the application into several components. With several components shared between the simulators, we will first cover the design and description of the single register simulator followed by these shared components.

The Single Register simulator takes the incoming line of text from the application file (the formatted ‘MIPS’) code and interprets it into the anticipated result. Our register values are stored as a dictionary with each key being a MIPS register. We then use this dictionary to manipulate the data computed by the parser. The main body of our application uses a case statement to call the functions corresponding with the MIPS operation (e.g. ADD, LOAD, SLT). This data is then processed, and stored in our pseudo registers.

For the shared components, please refer to General Components.

The major changes to the architecture of the ‘original’ algorithm was on the instruction level, as there were a few areas where we found that the functions provided were slightly limiting. In the swap function, we had an extra load and store word due to the limitation of managing one register at a time. We also had to zero out all used registers in the outer loop as not to muddle the results or have errors due to the flag checking. Finally, we loaded in all environment variables in a pre-amble to the application, as this was the way we chose to simulate the introduction of the array’s address.

## Accumulator

For the Accumulator Simulator, we split the functionality of the application into several components. With several components shared between the simulators, we will first cover the design and description of the single register simulator followed by these shared components.

The Accumulator takes the incoming lines of text from the application file (the formatted ‘MIPS’ code) in a fashion similar to the Single Register architecture, with the exception of the usage of multiple registers. Instead, the Accumulator uses a single register and memory to manipulate the data.

For the shared components, please refer to General Components.

The changes made to the architecture of the ‘original’ algorithm were about the same as the single cycle, with the exception of the addition of several push and pop operations, as there is only one register. This greatly affected our algorithm statistics, as you can see later in this report.

# Data Collected and Comparison

Here we show the data gathered from the execution of the simulator and our analysis of said data. This data is to be read linearly, with each resulting section building on the previous. Please note that all data is averaged on three runs, as to limit the variance between operations. Arrays sorted consisted of five arrays of integers with ranges from -100 to 100, with duplicates and zeros present.

|  |  |  |  |
| --- | --- | --- | --- |
| **MIPS** | **Total Execution Cycles** | **Memory References** | **Lines of Code** |
| *Array 1:* | 6,983,638 | 2,406,696 | 26 (withholding comments) |
| *Array 2:* | 6,966,090 | 2,404,090 | 26 (withholding comments) |
| *Array 3:* | 6,753,661 | 2,332,358 | 26 (withholding comments) |
| *Array 4:* | 7,015,422 | 2,420,212 | 26 (withholding comments) |
| *Array 5:* | 7,048,412 | 2,433,408 | 26 (withholding comments) |

As the above chart shows, the MIPS architecture preforms the fastest of the three, preforming with a total execution cycles 3 million less than single register, and 7 million faster than accumulator. The memory references are understandably lower as well, although not as extreme when compared to single register. This is because the underlying algorithm architecture is the same, thereby encoring similar running time only varying by 1 million when in comparison to the single register.

|  |  |  |  |
| --- | --- | --- | --- |
| **Single-Register** | **Total Execution Cycles** | **Memory References** | **Lines of Code** |
| *Array 1:* | 9,094,622 | 3,831,454 | 35 |
| *Array 2:* | 9,078,728 | 3,834,864 | 35 |
| *Array 3:* | 9,004,916 | 3,813,668 | 35 |
| *Array 4:* | 9,136,226 | 3,859,190 | 35 |
| *Array 5:* | 9,095,954 | 3,832,342 | 35 |

As you can see, the single-register architecture executed 3 million less than the MIPS architecture, but 12 million less than the accumulator. As for the memory references, there is a large jump from the previous model. The jump in memory references is due to the lack of registers present in the single register model. This makes since; MIPS can work with 2-3 registers at a time, while the single-register can only work with on at a time, with temporary reprieve in the MOVE statement. This effect of increased references is reflected in the increase in the total execution cycles, as more code must be used to compensate for the lack of registers. However, since the single cycle model has 32 registers, memory will not have to be access as much when compared to the accumulator model.

|  |  |  |  |
| --- | --- | --- | --- |
| **Accumulator** | **Total Execution Cycles** | **Memory References** | **Lines of Code** |
| *Array 1:* | 14,593,152 | 10,529,955 | 50 |
| *Array 2:* | 14,563,341 | 10,518,781 | 50 |
| *Array 3:* | 14,440,683 | 10,440,401 | 50 |
| *Array 4:* | 14,655,558 | 10,585,427 | 50 |
| *Array 5:* | 14,595,150 | 10,531,731 | 50 |

As you can see from the above table, the total execution cycles are by far the worst here, over double of the cycles for the single register model, and 7 more than the MIPS model! The memory references are also staggering, over 12 million more than the single register and 7 million more than the MIPS memory references. As large as these differences are, this actually makes some since. The accumulator runs via one register, whereas the single register model has 1 but gets away with 2 via MOVE, and the MIPS 2-3. This limitation on registers forces the accumulator to write to memory MUCH more often than the other models, hence the extreme near-exponential jump in memory references. As for the execution cycles, the shier jump in lines of code the algorithm must traverse substantially contributes to this increase to the total execution cycles. For example, the swap function in the MIPS, single cycle, and accumulator have 3, 5, and 8 lines (excluding jumps) respectively. Since the jump is called quasi equally on these three models, we can say the swap is called an average of several million times. This ‘small’ increment in lines of code in the swap alone will GREATLY contribute to the execution cycles.

# Conclusion

We will close with a few finial remarks on the overall performance of the application via what we thought of the results of this application, what we expected, and why we believe the program behaved in the manor it did.

## Results and Reasoning

As we saw in the Data Collected section, we can see that the MIPS implementation of quicksort is by far the least costly, followed by the Single Register simulator, and finally by the Accumulator. This is because the methods used to implement bubble sort on each respective simulator took more memory accesses in order to correctly sort the array that its predecessor. These memory references are vividly shown in the execution of the accumulator. Also, the shier increase in lines of code also had a large play in this as well, forcing an exponential increase in execution cycles, dependent on the model of course.

## Expectations

While creating this project, we anticipated this outcome, as, again, and the more said application references to memory, the slower it becomes. Therefore our initial exceptions generally matched the Results.

## Figures



Figure : The general execution of our application



Figure : The class design of our solution