Minor Assignment Submission

Indian Institute of Technology Delhi

COL216: Computer Architecture

Name: Deepanshu Entry Number: 2019CS50427 Group: 04

1 Introduction

The assignment is about converting the MIPS interpreter developed in Assignment-3 into a MIPS simulator by adding two features viz adding main memory model and making it non-blocking.

2 Approach taken to solve the problem

Following is the step wise description of the approach taken to solve the problem.

2.1 Choosing proper data structures

As mentioned in the problem statement, we need to look for appropriate data structures for maintaining processor components such as Register File and Memory. I carried out the hunt for data structures as follows:

1. **Array:** Array is used for representing memory as expected. In addition to memory, it is also used to store the values of register and also the frequency of each operation performed (to show statistics in the end).
2. **Unordered Map:** It is used to give an integer token to each of the 10 commands so that instructions can be stored in memory.
   Also it maps appropriate register ($zero, $t7 etc.) to appropriate integer key.
3. **Vector:** It is used to process single line of input and store the command entered and various parameters required for that command. This vector is then used along with unordered list to load the instruction in the memory array.

2.2 Taking major design decisions

We need to take many design decision along the way. Some of them are as follows:

1. I took the name of registers as what is present in QTSpim. It required mapping all register name to appropriate index. I used unordered map for this which gives the key in constant time, so **efficiency is not worse-off.**
2. Using switch statement instead of nested if statements. **As mentioned by Prof. Panda in the lecture**, one should prefer using switch instead of nested if-else in cases when we have more number of comparisons. Hence, I used unordered map of C++ to give integer id to each operation and then use switch statement over these integer values.
3. I am giving 4 spaces in the array corresponding to each instruction. In case we have operations like j that take only one argument, we leave the other two values as zero so as to maintain the design principle of **simplicity favours regularity.**
4. We have chosen non-queue implementation for the sake of scalability and simplicity. The advantages and disadvantages are discussed briefly in the coming sections.
5. I chose to go with the option in which every instruction other than "add" and "addi" are also given. The code is capable of handling those cases and hence we got a richer set of test cases in the end for testing our code.

2.3 Choosing various functions for the program

We also need to think about various functions which are required to run this program. Some of them include:

1. void printRegisters()
   This will be used after every clock cycle to print the content of the register in hexadecimal format as specified in the problem statement.
2. printStats()
   This will be used to give the total statistics in the end of the program to give various statistics like total number of times each command is executed and total clock cycles taken to execute the code. This is again as specified in the problem statement.
3. decToHexa()
   This will be to convert decimal number to hexadecimal number of the register values.
4. dramIssueDeclaration()
   This will be called every time we issue a DRAM declaration.

2.4 Strengths of this approach

Following are the strong points for this approach of solving problem:

1. **No separate DRAM matrix**
   This approach is space efficient as does not require making extra 2-D DRAM array/matrix. The exact same thing is done considering out array to be row-major implementation of the matrix. i.e (i,j)th element of matrix is the (1024*i +j)th element in the memory array.

2. **Space Efficiency**
   This implementation does not uses queue and hence less space is taken. In cases with lots of loading and storing commands, the queue can take up a lot of space without increasing the efficient. Moreover, we need to enqueue dependent instructions like addi, add etc. also in the queue.

3. **Time Efficiency in certain cases:**
   This approach does not require searching for "safe" and "unsafe" registers in the queue corresponding to each of the instructions. In cases with lots of loading and storing commands, time taken will be more and overall efficiency will do down.

4. **Simpler implementation:**
   This implementation is a very basic code and anyone with basic knowledge of arrays and syntax of C++ will be able to understand it and even build upon it further.
   Other fancy methods including one made using a queue might require strong understanding of data structures and is more complex as it involves more number of corner cases.

5. **Building upon previous case:** I took the design decision to keep my code versatile so that it is capable of handling previous branching and setting commands too.

2.5 Weaknesses of this approach

Since most of the engineering problems deal with making a trade-off. Here also, for getting above advantages/strengths we have to trade-off and compromise on other features. Following is the detailed explanation of it:

1. Lesser efficiency in some cases. In certin cases when we have load or store commands followed by their independent terms after some time, the queue-implementation will be better.

2. Not close to actual implementation. **As professor Panda discussed in the lectures**, most of the actual task allocation done by operating systems have queue based backend.

3 Testing Strategy

3.1 Testing the C++ code

In this , we took the C++ code and run against it for different input files(i.e test cases) which has MIPS commands written on it and finally matched MIPS output for same commands with C++ output.

3.2 Error Handling

Following are the possible errors that can occur while running the code:

1. **Syntax Error :** Following are the various possibilities of syntax error:
   (a) Instructions which are not defined.
   For example : addu \$t1,\$t2,\$t3
   (b) Statement involving unknown characters:
   For example : add \$t1 ; \$t2 , \$t3
   (c) Lesser number of registers used for operation than required:
   For example : add \$t1 , \$t2
   (d) More number of operators used for operation than required :
   For example : add \$t1,\$t2,\$t3,\$t4,\$t5
2. **Memory Limited Exceed:** There is a possibility of memory limit exceed if total memory used for storing instruction and data exceed 2^{20} bytes .
3. **Segmentation Fault :** It may occur when we to excess the memory which is not available in the memory.
4. **Large offset used:** If the user tend to give the offset out of bound i.e larger than 2^{20} or less than 0. Even if it is not divisible by 4, we are flagging it as an error.

3.3 Choosing the test cases

All the test cases are provided in the zip file. We covered all the cases extensively ranging from error handling to looping. Some of the cases are as follows:

1. **Empty File :** Gives total clock cycle as 0 and displays all the relevant stats (all counts will be zero).
2. **Test case given by instructors:** Also tested against the test cases given by the instructors and matched the output with the expected output.
3. **Infinite Loop:** I tested the code against the code which runs infinitely. The simulator does the same in this case.

4. **Single input:** We gave a single instruction as input. There are two cases depending on whether the command is lw/sw or not.

5. **Testing against various standard programs**. Some of them are follows:

   (a) Finding factorial of a positive integer.

   (b) Given a positive integer n, find sum of first n positive integers.

   (c) Going in infinite loop.

6. **Cases according to load and store:** As mentioned in the strengths and weaknesses' column, there can be certain cases where number of load and store instructions are either extremely large or extremely small. So, I tested against those inputs as well.

7. **Testing for error handling:** I also tested the error handling capability by providing various possible wrong commands given by the user.