Major Class Project Final Report

By Matthew Jagielski, Jacob Bieker, and Theodore LaGrow

CIS 314 Computer Organization

Fall 2015

Preface: Before Dr. Kinsy sent out the project assignment, Jacob, Matthew, and Theodore formed their group with the common interest in studying and implementing something with parallel program (potentially parallel programing with MIPS, if that was even possible). After receiving the actual assignment, Jacob, Matthew, and Theodore enthusiastically jumped into the project.

Week 1: The first week was the most conceptually difficult and time consuming. As a group, we decided to include the full real integer MIPS instruction set to have a more complete simulation (instead of just implementing the instructions used in the test functions). This means the pseudo-instructions, such as mul (a three register multiply) or li (a load immediate value), were not included in the simulation. The list was pulled from Wikipedia. The Floating Point Instruction Set were also not used in this simulation.

GitHub was the chosen platform to collaborate on the code. As for the code layout, the premeditated decision was made to layout the program to adhere for the pipelining implementation in Week 2. The decision was also made to keep all of the work in one file to help keep the code modular and less convoluted when pushing and pulling from GitHub with three people in the group. This might not have been the best set up, however the single file and GitHub repository worked well for our group.

The code is organized in mainly the five CPU stages. Give or take the necessary ‘header/registers/main’ sections at the beginning or ‘helper’ section at the end, the program is laid out modularly with Instruction Fetch, Instruction Decode, Execute, Memory, and Write Back. The helper functions were placed at the end of the code to help with scrolling while writing the code. The next paragraph will outline the function used in the Week 1 Single-Cycle Simulator.

**Headers**: initializes all of the variables, array, and functions that will be used in the code. Struct types were created for all of the different types of instructions, an indexed register (to help with parsing), and for both execute-memory registers and memory-writeback registers. **Main**: the main takes in the assembly file and runs the file through our program. Testing was also conducted here. **Readmips**: the function reads in the MIPS instructions and accounts for the differing amount of elements an instruction can have. The function has the appropriate flags to check to see if the instruction is valid. **Control logic**: The printstate() function helps with the front end of the project to actually see what is being run by the program. It prints out the pc, then the registers with their names, and finally the first 70 ints in memory. Also printed are instructions and the values in the pipeline registers. The control logic runs each function through the pipeline. PC is updated which keeps the program advancing. **Fetch**: getinstruction() gets the instruction from the instruction[ ] array for control logic to operate on. **Decode:** the decodeinstruction() is taking in the instruction after fetch and (depending on its type) will parse through and set the correct values within the structs to delineate in pieces what the instruction wants. The function creates an r-type, i-type, or j-type struct depending on instruction and passes on the pointer. **Execute**: the pipeline stage consists of three excutes for the three types of instruction type. The execute functions will create an appropriate exmem struct that will create a destination where the values can be store so the memory pipeline stage can use those values. Each function will return a stuct that will take in it's type from the decode stage. The *execute\_r()* passes all of the r-type instructions, including basics like add or the shift operations, and also includes the mult and div instructions, mfhi and mflo, and jr. The *execute\_i()* handles the lui instruction, all of the standard memory stores, all of the standard memory loads, the bne instruction, beq instruction, and then the immediate alu instructions (addi, etc.). The *execute\_j()* function accounts for the j instruction, and the jal instruction. **Memory:** the memory\_rw() function takes in the specific instruction from decode, and performs the correct memory operations. The function will then create an appropriate memwb struct to pass on important values to writeback. **Writeback:** the writeback() function writes values into appropriate registers. This function also takes into account mfhi, mflo, and mtcZ. **Helpers:** *Fetch Helpers*: none - fetch is pretty simple in our implementation. *Decode Helpers*: the functions make\_r\_type(), make\_i\_type(), and make\_j\_type() parse instructions, as well as some others like nextint(), nextregister(), etc. *Execute Helpers*: all of the Arithmetic Logical functions and shifting functions are here. *Memory Helpers:* all of the Data Transfer load functions are here. *Writeback Helpers*: all of the Data Transfer store functions are here. *America*: this function uses the free() method to release the memory we have malloc'ed with the pointers and structs.

Week 2: The program is relatively the same as Week 1. By creating structs and functions to implement the five processing stages, the program was set up well for pipelining. Pthreading was decided as the module to conduct the multithreading. The differences from Week 1 to Week 2 are detailed below in the following paragraphs.

By changing the execute stage from having three functions that accounted for all three types of MIPS instructions to having one central execute function helped with the control flow of the stage itself. This way the pthreading only had to deal with one function instead of three with execute.

There was an error that arose when the execute function was combined that outputted zeros for all of our values. Matthew found the fix. The space being malloc'd was also trying to be written on and the output was zero.

The decision was made to create central staged function (getinstructions, decodeinstruction, execute, memory\_rw, writeback) a void function and make each function return a void\*. This way the functions fit into the parameters of the pthreading to operate pthreading.

5 pthread\_creates were added for each stage in our control logic function. 5 pthread\_join were created for each stage in control logic. This will make sure each thread will wait before starting the next increment cycle so no function is called multiple (or incorrect) amount of times. 5 structs were created to take in account for when a stage completes and needs to write to the next cycle but the next cycle is still running. We called these the intermediary structs.

A Makefile was created to help with compiling the program (there is an extra piece that needs to be added for the pthreading when compiling [-lpthread]). The Makefile is also set up to run all of the tests to iterate through each test.

To account for the three hazards that arise with pipelining, the decision was made to implement a stall method instead of a forwarding method. Locks were used to control the flow in control logic.

A major issue that arose on Week 2 was an issue with looping and registers not being allocated to correct memory. The code was running continuously until the incorrect pointer was found.

Week 3: This week’s big challenge was how to go about the Direct Mapped Cache in the current program’s layout. Below is what was decided.

Structs were created for both the cache and the cache\_line. Since the each stage of the CPU is not outputting huge amounts of data it was decided that the caches should be 64 bytes long and not 128 bytes. A check\_cache() Boolean function was added to delineate when the cache had a hit or had a miss. The function is marked as void to account for the hits and misses, conducting the correct operation thereafter for each output. For the memory stage of the CPU, the structure was changed to adhere to the structure of a cache line (tag, slot, and offset) for both the store memory functions and the load memory functions. A write\_cache\_line function was added to write the current cache line to the memory when needed.

A huge problem arose Week 3 with the branching implementation. Jal was a slight issue in Week 2 and bled over into Week 3. Debugging this branch issue proved problematic. It was discovered that Jal was not the issue and thought that Jr was the issue, however it turned out to be a problem with pointer issues and locking issues. This bug took up a majority of the time in Week 3.

Conclusion: This was a fun project that really got our juices flowing! Lots of late nights in the library and Deschutes. We realized late night that we build a von Neumann Architecture simulation. Our memory and instructions are in separate locations and we all got a laugh out of the fact that other knowledge of the creped into the project inadvertently.

We realize that our program is nowhere near optimal; there are several ways to speed up the code. For example, if we did the program again, we would not have wrote the three structs for the different types of instructions in execute. This is a mess and convoluted way of writing this stage. It could be messy for an outsider to understand our writing. We could have also done forwarding during the pipelining stage of the project instead of stalling. The print statements are terrible and could use some serious work. The output is not as pleasing as it could be. The control logic is hacked together but works for what we need to do it; it could be done in a more elegant way. We don’t lock registers in use the way locks are intended this is a little sloppy. Variable names can be more precise (some of the variables are squished names). Error checking can be improved. We feel we did a good job in testing our program, but we’re sure we missed certain cases. We used too many struct pointers, which is sort of a hack in memory stage where instead of writing to a new struct, we wrote to the struct memory allocation. We could have also written the code in Python in several hundred lines shorter (but that was not the point!). And, clearly, there are not enough ASCII characters. But in general, we feel that most of our code can be done in a more elegant way, potentially using hashmaping to solve some of these issues.

Looking forward, we could take this code and decide to add a different caching system depending on the type of MIPS code being simulated. We could also add more instructions like the floating point instructions or even the pseudo-instructions (especially mul and li). However, this project was a great introduction to the scope of what Computer Architecture could be. Thank you for the opportunity!

Resources: If you would like to see the GitHub repository, the link is below. However, we all do not agree with the stats and commits. The commits and amount of lines written are not valued correctly. For example, Matthew did not register his email until halfway through the project and his line count is significantly low comparatively to what was actually written. Conversely, Theodore accidentally pushed a full duplicate of the project Week 2 then deleted that copy which skewed his lines written and deleted. In the same effect, Jacob committed as often as he could (including when the code wasn’t compiling) whereas Matthew would write for hours, and then commit a huge chunk of code. So the stats vary, however the repository can be used to see history of the trajectory of the code.

Link: https://github.com/jacobbieker/CIS\_314\_Final\_Project