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 TJ 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 constraining. As a group, we decided to include the full integer MIPS instruction set.

Github was chosen as the platform to collaborate on the code.

We decided to lay out this program looking forward to the pipelining implementation next week. We decided to put the helper functions of all of pipeline stages at the end of the code to help with scrolling in the program. The code in in one file to help with merging with github (probably not the best design, but this worked for us). For our simulator, we wanted to have ALL of the real integer instructions (taken from Wikipedia). Note - this means that pseudoinstructions such as mul (3 register multiply) or li (load immediate value) are not included, nor are any floating point instructions. As such, any MIPS assembly code which does not use floating points can be simulated. The layout below is as follows with the flow of the code:

----------------------------------------------------------------------

headers

- Initializes all of the variables, array, and functions that will be used in the code.

- We created struct types 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.

- Matthew has added comments with his struc headers to indicate how the pointers operate and function with the design of the program.

main

- The main takes in the assembly file and runs the file through our program.

- We used the main for lots of testing as well.

readmips

- The function has the appropriate flags to check to see if the instruction is valid.

- The function reads in the MIPS instructions and accounts for the differing amount of elements an instruction can have.

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 (4 bytes each). 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.

- The instructions are also sent to the appropriate pipelining stage that it needs to go to next.

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 make the correct values within the structs to delineate in pieces what the instruction wants.

- Creates either 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.

- alu() is positioned betweeen execute and memory

- All of the if/else statements that call the alu operations.

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 the mfhi, mflo, and mtcZ

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.

- This is only the instructions and jumplocations arrays for now.