Lab 2 Report: Cameron Young, Matthew Kellermann
cky39v, mtkmc6

Work Distribution:

Code:

MIPS Code Struct- Cameron Young
Input String Read- Cameron Young
Immediate Address Handling- Cameron Young, Matthew Kellermann
Instruction Recognizing- Cameron Young
Register to Binary Conversion- Matthew Kellermann, Cameron Young
Binary to Hex Conversion- Cameron Young
Bugfixing- Cameron Young

Mips Problem Code:

Matthew Kellermann

Lab Report:

Cameron Young, Matthew Kellermann

Milestones:

MIPS Code Struct

Our MIPS code struct allowed for each line of MIPS code to have an individual object which contained character strings of each binary string needed to parse the instruction(rs, rt, rd, opcode etc.). The struct also indicated which type of instruction each line represented. This was very helpful in simplifying and organizing our code design, which avoided redundant code.

Instruction Recognizing:

In order to identify the type of instruction for each line of MIPS code, we used a function that would find the first word of the string, using “strcmp”, and would define in the line’s MIPS code object, which type of instruction it was.

Handling Immediate Addresses:

In the case of an immediate address in the MIPS line, we had to be able to parse that address first in binary to create the 32 bit string. For this reason, the code had to recognize if the address was Decimal or Hexadecimal, and then do the appropriate steps to convert the address to a binary string that could be concatenated to the 32 bit string.

Implementation Decisions:

Binary to Hex:

We decided that the easiest way to convert the MIPS code to Hexadecimal, was to first convert the MIPS code to binary, then convert to Hex. It would be nearly impossible to go from MIPS to Hex directly since instructions in MIPS have different binary bit lengths.

Combining Instruction types:

In the last lab, we had different functions for each instruction type. In this lab, we found that it was easier to handle all of the instruction types in a single function, by checking the MIPS struct. This allowed us to have less redundancy in our code when creating functions.

Using Case Statements:

We discovered that using Case Statements was an effective way of converting a specific instruction to binary Opcode. We had one function that would recognize the specific instruction, and return a unique integer. Another function would continuously call the instruction recognizing function in a loop, and use a case statement that would translate the unique integer into binary Opcode.