CS 2340 Term Project Report

Video link: <https://youtu.be/Oppd94DGePY>

Description:

The program has five module files and one main file. The functions of the modules are as follows: input file management, input conversion, macros needed for the disassembler, the disassembler, and output file management.

Input from a DAT file is read by the input file manager and placed into an array containing ASCII, then an array converts the ASCII into hexadecimal as well as placing each digit of the instruction into a single byte. Finally, every two decimal bytes are combined into one byte and reversed so each instruction takes up the space of one word and later programs can access the instruction’s first byte as the most significant 8 bits.

Various macros are loaded with functions such as isolating register codes, converting hexadecimal to decimal, and formatting numbers and strings to be stored at a label reserved for output. In the disassembler, all instructions are looped through, and checks for syscall, r-type, j-type, and i-type formats are used. When a type is identified, a branch occurs to handle instruction disassembly and output to the reserved output label, which is continually updated after each stored value to contain the address of the next available byte.

The output file manager creates and writes to ‘output.asm’ using the label after the loop ends.

Challenges:

Attempting to isolate different parts of an encoded instruction initially proved to be difficult. My first guess was to load specific bytes and use both shift left logical and shift right logical, but this felt too inefficient. I realized that after loading a word and using shift left logical, a logical AND immediate instruction could remove unwanted leading bits.

Calculating the target address for j-type instructions was moderately difficult because there were gaps between my conceptual knowledge of j-type address encoding and functional knowledge of using it. However, after implementing a loop counter in the disassembler to represent line numbers, I was able to figure it out.

I encountered minor problems with converting hexadecimal values to decimal, as I could only view the result of my code in the MIPS data segment display in hexadecimal, which made for confusion in trying to confirm a successful conversion. An online Hex-to-Decimal and Decimal-to-Hex-tool greatly helped with this.

The most problematic part of the project for me was trying to interpret the base and order of bytes in general. Halfway through, I realized my code for isolating the opcode and funct code was not working because load word loaded bytes similar to a little-Endian system, while I was expecting a big-Endian format. For example, the byte needed for the opcode in the value 0x01242140 is 0x01, however the byte I was loading was 0x40. I later reformatted by input conversion process to store the value as 0x40212401, so the first loaded byte would be 0x01.

What I Learned:

I learned how instructions are encoded in MIPS more in-depth, the practical use of logical AND, OR, SLL, and SRL, and how MIPS stores and loads data in bytes. I also learned how to use labels, branches, and jumps to achieve functional loops, if-else, and case-switch statements. Finally, I learned the basics of using modules and macros in MIPS.

Discussion of Algorithms and Techniques Used:

The process of converting ASCII input to usable hexadecimal values contains three loops.

To fluidly chain included modules together after the first file executing sequentially, I placed a jump to the bottom of the later modules and used jump-and-link to jump to a loop at the top, which would return to the bottom after its completion and drop off to the next module.

I used macros to avoid repeating code in the disassembler. I understood that because I did not want to complicate the program by saving register values, I had to avoid using certain registers. The registers I did use could potentially be different from their values before the macro, so I maintained caution in using registers to store important values between macro calls.

Hardcoded arrays of opcode strings (e.g. “add”) and their decimal codes (e.g. “add” maps to 32) were used to match isolated opcodes and funct codes and save their matching string to the output label. Hardcoded branches were used to output the encoded instructions for syscall and jump register (jr).

I also used temporary spaces for converting isolated instruction codes to readable ASCII values. The hexadecimal value’s digits were split into bytes, converted into their corresponding ASCII values, and stored to the output label. For instructions that required a decimal value to be stored instead of hexadecimal, such as shift, branch, load, or store instructions, their isolated immediate value or target address was converted to decimal, then used the same steps as before.