Lab 2 Report

Matthew Crump A01841001
Nicholas Williams A02057223

Objectives

Demonstrate a functional compiler for DLX assembly and generate a MIF file.
Understand all supported DLX instructions and the bit mapping for reach instruction.

Procedure

We looked at all 49 instructions and the 5 general instruction types to get a feel for what we needed to do. Since we had a similar Python program (designed to take pseudo assembly code, 8 instructions, and generate Verilog), we used that as a general jumping off point for this lab to get a headstart. Only moderate modifications were required. This Python program had 3 main stages: tokenizer, parser and the assembler.

The tokenizer would take the text file and read it character by character. Spaces, tabs and similar characters were skipped and everything else was combined to make a token. A support class was created to keep track of what index we were at in the text file and also noted the start and end index for each token. This was done to make error reporting easier. At this stage, only illegal characters would cause an error, no grammar was checked at this time. Each token had a type associated with it and could be one of the following: VARIABLE, LABEL, INSTRUCTION, INTEGER, REGISTER, INVALID, or SEGMENT.

The parser then takes the list of tokens and groups them together and checks some grammar rules. So, if the first token was an instruction and depending on that it would match up the correct number and type of operands. If the operand types were incorrect, an error message is generated pointing to the line in the file. The parser would also check if it was in the data segment or text segment. While in the data segment, the parser would verify that only variable declarations were present and each variable had the correct number of values. The parser would then verify the text segment to ensure that no variables were declared and all operators had the correct number of operands.

Finally, the assembler would then preprocess both the data tokens and text tokens. The data tokens simply received a sequential address to live at and the name of the variable was saved into a lookup table to easily find its address later. If any variables were

declared twice an error was produced. Then the text section was analyzed to assign addresses for labels which was then handed off for final assembly. With addresses of both variables and labels resolved, all instructions were converted to a binary string to make the bit ordering easier. Then that binary string was converted into a hex string and the corresponding code was appended as a comment. If a label or variable was referenced but not declared an error message was thrown.

```
Invalid Register: Expected a Register from 0-31 but got ->: R111
File .\examples\example3.dlx, line 17

SW    a(R12), R111
                ^^^^
```

```
Invalid Instruction: Expected a valid DLX instruction but got ->: ADDG
File .\examples\example3.dlx, line 14

ADDG  R1, R5, R0
 ^^^^
```

```
Undeclared Label Reference: The following label has been referenced but not
declared ->: loop_done
File .\examples\example3.dlx, line 34

BEQZ  R9, loop_done
              ^^^^^^^^^
```

Figure 1. Example Error Outputs

The program was also split into 3 files to make it easier to maintain and search for specific constants, error handling and then the assembler itself. As discussed earlier, the main program has 3 main stages and consists of 5 different classes. This should make it easy to maintain and extend its functionality later on.

Results

We tested our program as we built it. We first tested our tokenizer by printing out to the console every token or label it found in the order it found it to make sure that we were not losing any information from the input file. We then wrote the parser section to make sure that the input file was correct. For example an add command required that 3 registers be listed afterward. The parser section will be useful in the future as we write more of our own DLX files. The parser section even found a few errors in the example

files that we were given for this assignment. The actual assembler section worked right away. We first convert the assembly instructions to binary and then to hexadecimal.

After we had our assembler working for all the example files we wrote the DLX file for computing a factorial. We decided to just keep the logic simple and only use addition to multiply the values together instead of trying to perform logical bit shifts. To make sure the code was working we performed 5 factorial and walked through each line to make sure the end result stored the proper value into the memory location “f”.

Conclusion

We successfully wrote a DLX assembler in python that reads in a single DLX file and outputs separate, assembled data and code files. This assignment helped us better understand how DLX instructions are assembled to binary segments. We also gained experience in writing a DLX file. Writing a DLX file was a great opportunity to practice using assembly language and understanding code at a register transfer level.