Lab #8: Disassembler Implementation

Rishit D (CS21BTECH11053)

November 29, 2022

1 Functions Overview

The implementation of the diassembler has been done in `C++` with the help of `iostream`, `fstream`, `vector` and `string` libraries. The code consists of the following (group of) functions:

1. Instruction Printing Function: `printVec`
2. Generators:
   (a) Generates register names (if valid): `genReg`
   (b) Generates immediate values (I-type) in decimal: `genImm`
   (c) Generates immediate values (I-type) in hexadecimal: `genImmHex`
   (d) Generates offset for B-type instructions: `getBimm`
   (e) Generates offset for J-type instructions: `getJimm`
3. Helper Functions for Hexadecimal Parsing:
   (a) Checks if a character is from 'a' to 'f' or 'A' to 'F': `isHexAlpha`
   (b) Checks if a character is numeric ('0' to '9'): `isNum`
   (c) Removes white-spaces if any from the right-end (disregards white-spaces at the end of lines of input file): `rstrip`
   (d) Converts a hexadecimal string to an integer: `strHex`
4. Helper Functions for Instruction Parsing (Case-Switch for `funct3` and `funct7`):
   (a) Prints instruction for R-format: `printR`
   (b) Prints instruction for I-format: `printI`
   (c) Prints instruction for S-format: `printS`
   (d) Prints instruction for B-format: `printB`
   (e) Prints instruction for J-format: `printJ`
   (f) Prints instruction for U-format: `printU`

2 `main()` Overview

1. The `main()` function starts with taking `argv[1]` as the file name and attempts to open the file. If unable, it terminates the program and returns an error message.
2. The program now parses through the lines of the file and removes leading white-spaces to ensure uniformity and stores the hexadecimal strings in a vector.
3. Now, each string is converted to an integer if it is a valid hexstring. Otherwise, an error is reported and the program is terminated. A separate list for storing future labels is parallely created (each instruction having default value as 0).

4. The program now parses through the instructions and extracts corresponding opcodes and transfers control to the appropriate format via `enum` definitions supplemented with `case-switch` blocks. The offsets for the `B` and `J` format instructions are noted. If the opcode is wrong, an error is message is stored in the final vector (to printed at the end) instead of the usual instruction. If the offset of the `B`/`J` instruction either exceeds the program size or leads to a negative index, the program is terminated and an appriopriate error message is printed. If the offset is valid, the instruction to which the present instruction leads to is given a label (integer which auto increments). The instruction without the offset is stored in the final vector.

5. The program now appropriately suffixes/prefixes labels based on offsets and the table of labels for reference.

6. The program now prints the final vector and terminates the program. Any errors in individual lines (`opcode`, `funct3`, `funct7` mismatches) will be shown here.

3 Code Output (and Errors)

For the following discussion, we will consider the test-case files in the directory `TestCases` which consists of 5 files with hexadecimal coded instructions in separate lines. Note that the program executable will take the file-name as command line argument (`argv[1]`).

3.1 Wrong File

We will attempt to run the execxutable on a non-existent file (say `myLife` for example). The following is the output:

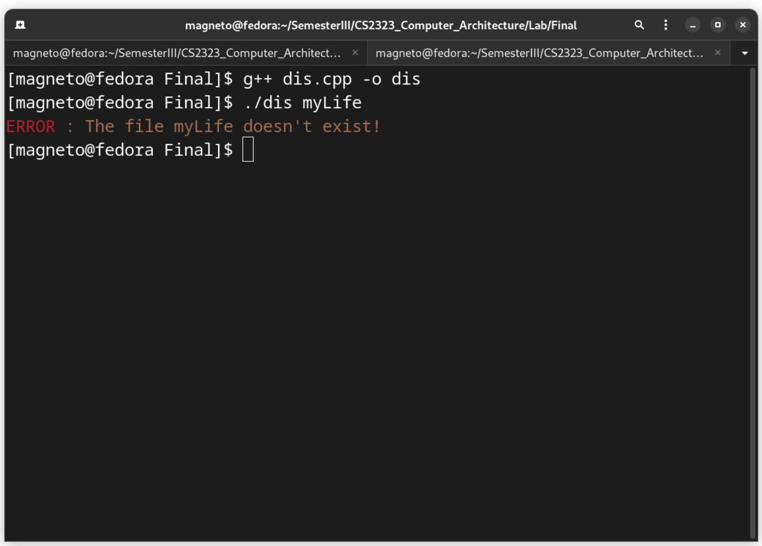

Figure 1: Wrong File

3.2 Invalid Hexadecimal String

Consider test-case `case1.s`. In this case, the 6^{th} line has the line $00a585x3$ has an invalid hexadecimal character x. Hence the output is as follows.

3.3 Invalid `funct3` Values

Consider test-case `case2.s`. Here, the 4^{th} line has the line $0081D023$. Note that the `funct3` are the bits $[14:12]$ which in this case is $0x5$. This does not correspond to any valid instruction from the S-type instruction (opcode $= 0x23 = 35$). Hence we get the following output.

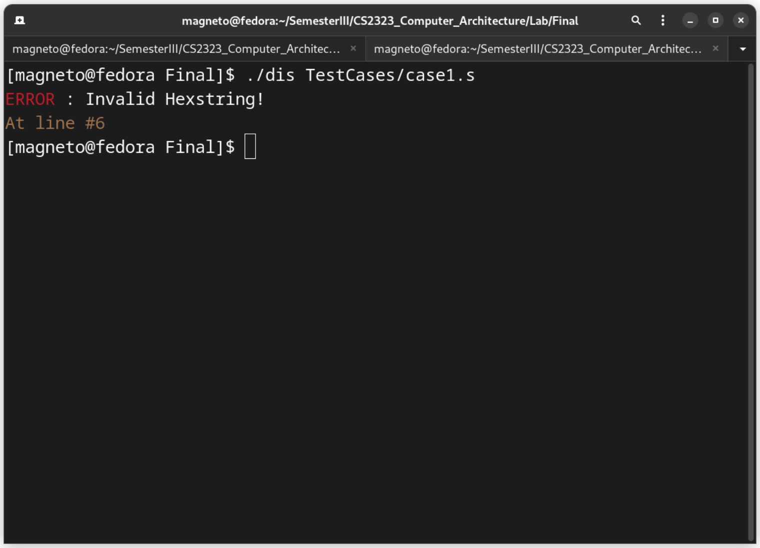

Figure 2: Wrong Hexadecimal Instruction

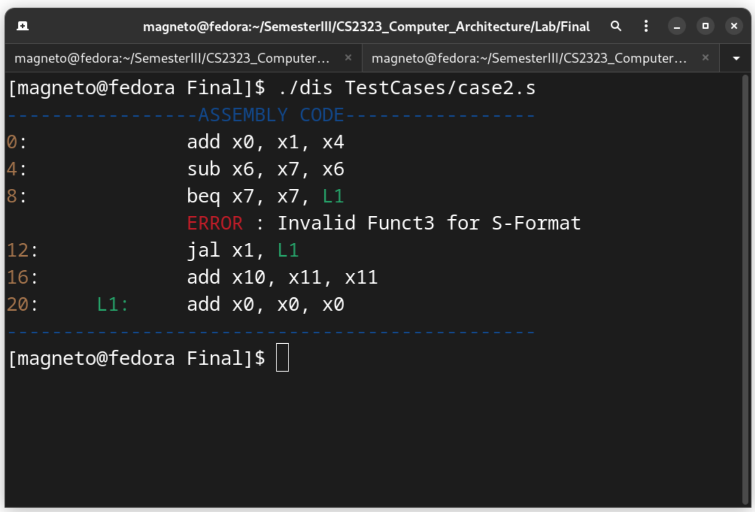

Figure 3: Invalid `funct3`

3.4 Invalid `funct7` Values

Consider test-case `case3.s` which is code to multiply two matrices. This code uses `mult` whose `funct7` code is not recognized. Hence we get the following output.

3.5 Out of Bounds

Consider test-case `case4.s`. This program consists of a branch statement that has an offset of -40 at line 4, which results in the "goto" instruction as $12 - 40 = -28$. Hence we get the following output.

3.6 Sample Input

Consider test-case `case5.s` which is the test-case in the problem statement. The following is the output:

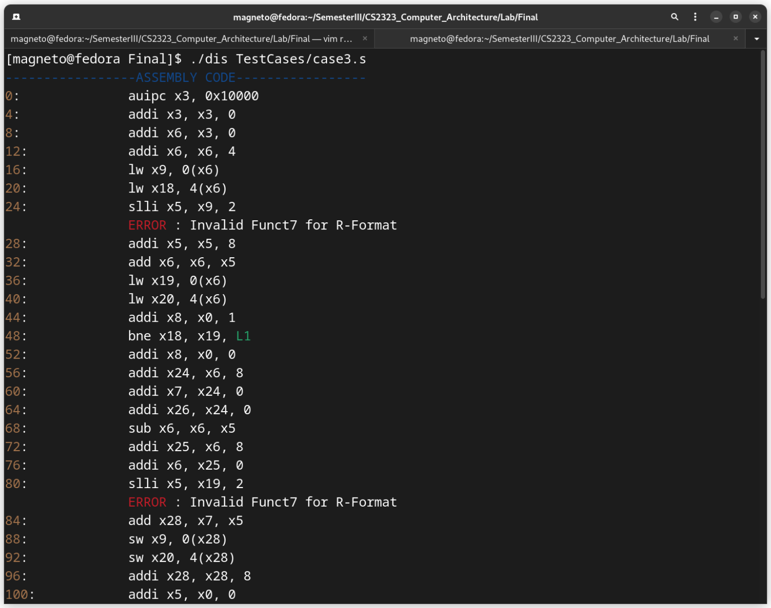

Figure 4: Invalid `funct7`

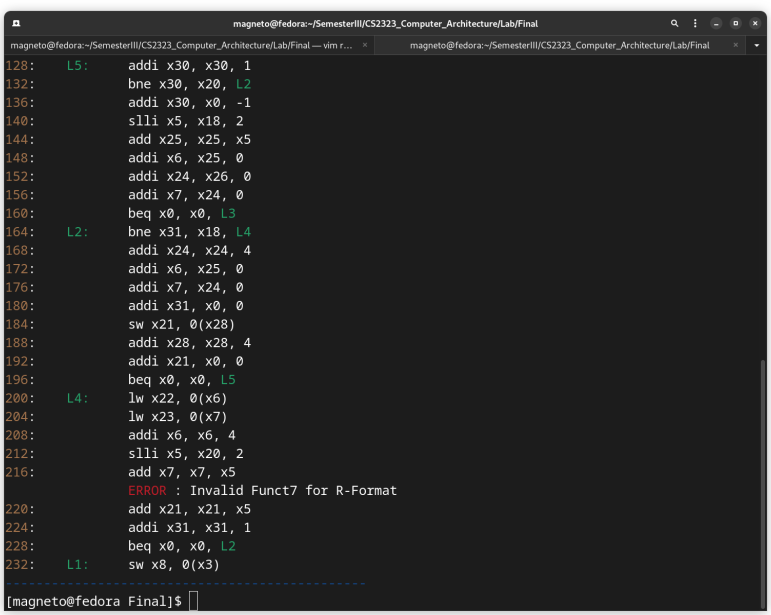

Figure 5: Invalid `funct7`

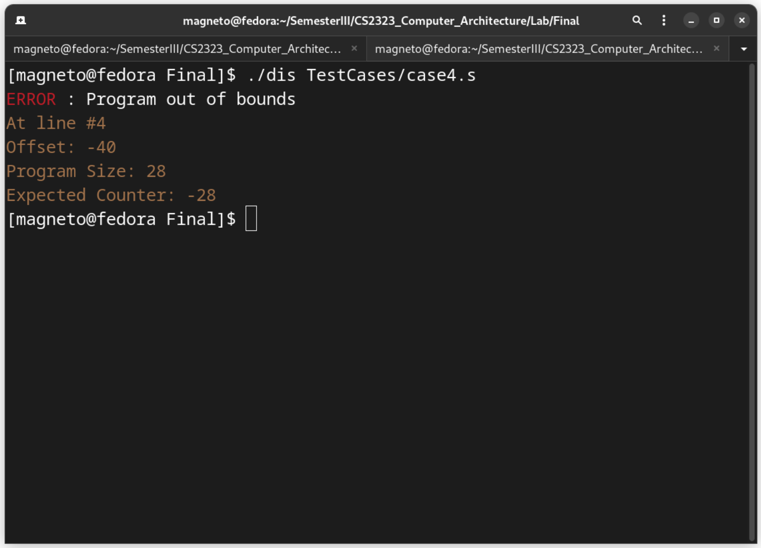

Figure 6: Out of Bounds

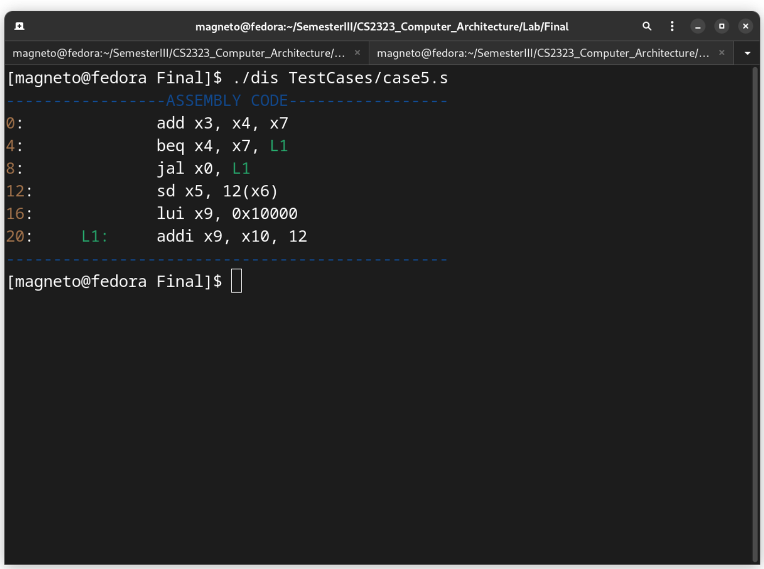

Figure 7: Program from Problem Statement