CSCE 2303 – Computer Organization and Assembly Language Programming
Summer 2020

Project 1: RISC-V Disassembler

Report

[Group 12]

Farida Tarek Yousry 900171942

Marwan Awad 900172585

1. APPLICATION OVERVIEW

1.1 DESIGN RATIONALE

The design of the disassembler applications is based on dissecting the inputted binary code in a file into individual instructions consisting of 32 bits (4 bytes) each, in order to begin identifying the instruction and its parameters. The program first analyzes the operation code (opcode) to identify the instruction type, and hence pinpointing what and where its parameters are located within the instruction itself. Afterwards, through matching each segment of the instruction against the look-up table, the program outputs its equivalent in assembly code.

1.2 IMPLEMENTATION & PARTICIPATION

It was more efficient to implement the application using C++ seeing as both group members are proficient in it and to utilize the provided skeleton.

The program mainly consists of one key function :

```
void instDecExec(unsigned int instWord),
```

which takes as a parameter a 4-byte binary instruction and is responsible for parsing, identifying its type, and translating it into its equivalent RISCV32I assembly instruction. It does this by assigning the first 7 bits to the int variable `opcode`, through bit-masking and anding, as it is the common denominator of all instruction types. The next 5 bits are assigned to the destination register `rd`, and the next 3 bits to the `funct3`, while the next 5 bits are assigned to the source 1 register `rs1`, the next 5 bits for the source 2 register `rs2`, the next 5 for `funct7`. While the immediate values for types other than the R-type overlap with the source registers and other bit segments, they also get assigned values from the

instruction as follows, and according to the opcode, each variable is utilized accordingly, in order to save processing time.

```
opcode = instWord & 0x0000007F;            //to get rightmost 7 bits (OP Code)

    rd = (instWord >> 7) & 0x0000001F;      //to delete op code and get next 5
bits for rd (dest register)

    funct3 = (instWord >> 12) & 0x00000007;     //to delete 12 right most bits
and get 3 bits for funct3

    rs1 = (instWord >> 15) & 0x0000001F;    //delete rightmost 15 bits and get
5 bits for rs1

    rs2 = (instWord >> 20) & 0x0000001F;    //delete rightmost 20 bits and get
5 bits for rs2

    funct7 = (instWord >> 25) & 0x0000007F;

    I_imm = (instWord >> 20) & 0x00000FFF;

    S_imm = ((instWord >> 7) & 0x0000001F) + ((instWord >> 25) & 0x0000007F);

    U_imm = (instWord >> 12) & 0x000FFFFF;
```

After deriving the opcode, funct3, rd, rs1, rs2 and the immediates by shifting right and masking by the appropriate number of bits for each variable. The masking was to ensure that we extracted exactly the number of bits required for each variable, and the masking consisted of the repetition of ones x the number of bits needed to be extracted.

The first thing to be checked was the opcode and after that, depending on the instruction type, was funct3. After that, also depending on the instruction type, we checked funct3 and in 2 of the I type instructions we checked the immediates for the type as both instructions shared the same opcode and funct3.

Once the opcode is identified, the application then determines the exact instruction through inspecting `funct3` and `funct7`, in the case that more than 1 instruction share the same `funct3`.

The application then proceeds to display the instruction to the console if identified, else an error message “Unknown []-Type instruction” is displayed, and moves onto processing the next instruction.

For negative immediate values, we created a variable to store the most significant bit, if its 1 then the immediate is negative and if its 0 then the immediate is positive. If found negative, we take the immediate value and xor it with 111111111111 or 0xFFF then added 1(mimicking 2s complement) to produce the negative value then printing a ‘-’ beside the new immediate.

Test Cases:

To ensure system reliability and efficiency, we have utilized both the provided test cases and generated our own test cases through simple programs such GCD in order to test all 47+ecall instructions in the RISCV32I Instruction Set Architecture (ISA). They can be found under the directory “Samples”, with each file including both the binary and assembly equivalents.

Gcd: gcd using euclidean algorithm

while(true)

```
{
if (a = b) go to exit
else (b > a?) 1;0
        If so, b - a
else (if b < a)
        a - b
}

exit:
        return a
```

---

sum0-9: add the numbers from 0 to 9

```
int sum = 0;
int i;
for (i = 0; i != 10; i += 1)
        sum += i;

//s0 = i, s1 = sum
```

PowersOf2Sum(1-100): add the powers of 2 in the range 1-100

```
int sum = 0;
int i;
for (i = 1; i < 101; i *=2)
        sum + i;
```

s0 = i, s1 = sum

ALL-samples: tests all RISCV32I instructions simultaneously

Contains a line for each instruction syntax to ensure testing of full RISCV32I instruction set.

Participation:

Marwan initialized and set the values for the immediates for each instruction type, along with writing the test cases for the B, J, S, and U type instructions as well as ecall and ebreak.

While Farida wrote the test cases for the I and R type instructions and set up the GCC compiler to fetch the file from the command line to run and execute. In addition to generating 3 folders of sample code to test all 47 instructions ecall, along with the ALL-samples file which tests all the instructions at one go.

Both members contributed to this report.

2. SYSTEM OVERVIEW

2.1 CHALLENGES

-Bit-parsing and assigning bits to different variables was difficult due to novel familiarity with the different instruction formats. Moreover, the unordered immediate bits in some instruction types made it difficult to concatenate them orderly.

-Generating sufficient test cases for the whole instruction set proved challenging and was quite time consuming.

-Due to the time constraint, the application scope is not as wide as we would of liked which caused the following **Limitations (2.2)**

-It was our first time using command line invocation, therefore it took some time to download and become familiarized with GCC compiler.

2.2 DESIGN LIMITATIONS

-Pseudoinstructions are not considered in the design, where only native RISCV32I instructions are disassembled.

-The program accommodates all negative immediates except for 2 instructions however we struggled to convert the immedates for the SW and SH from 2’s complement to decimal.

-Program only accepts files in certain format; without ‘.text’ and ‘.data’ labels