The American University in Cairo

School of Sciences and Engineering

CSCE230301 - Comp Org.& Assembly Lang Prog

Summer 2021

Dr. Mohamed Shalaan

Project 1 / Disassembler

Basant Elhussein Abdelaal / 900192802

Hashem Khaled Abdelfatah / 900192812

July 5, 2021

Table of Contents

Implementation Details

First: Instruction decoding:

Our logic for decoding the instructions goes the same way as they encoded.

- ➔ We check whether the instruction is 16 or 32-bit by checking the first two bits of the opcode (32: 11 , 16: otherwise)
- ➔ Accordingly, we will use the relative function to decode the instruction.

```
if(encode[0] == '1' && encode[1] == '1') get32(encode);
else get16(encode);
```

get32, get16:

```
// Get 16-bit instructions
void get16(string& code)
{
    if(code[0] == '0' && code[1] == '0') print16Zero(code);
    else if(code[0] == '1' && code[1] == '0') print16One(code);
    else if(code[0] == '0' && code[1] == '1') print16Two(code);
    else output << "Unknown Instruction\n";
}
```

```
// Get 32-bit instructions
void get32(string& code)
{
    int op = orBits(code, 0, 6);
    cout<<op<<endl;
    if(op == 51) printRIns(code, op);
    else if(op == 19) printI1Ins(code, op);
    else if(op == 3 || op == 103) printI2Ins(code, op);
    else if(op == 35) printSIns(code, op);
    else if(op == 111) printJIns(code, op);
    else if(op == 99) printBIns(code, op);
    else if(op == 115) output<< "ecall\n";
    //else if(op == 15) printFIns(code, op);
    else output<< "Unknown Instruction\n";
}
```

The two functions are used to decode the 32-bit instructions and the compressed 16-bit instructions.

The logic to get the instruction name and format follows a very similar logic to how they are encoded. We first check the opcode, and then if a more than 1 instruction has the same opcode, we check func3 and so on.

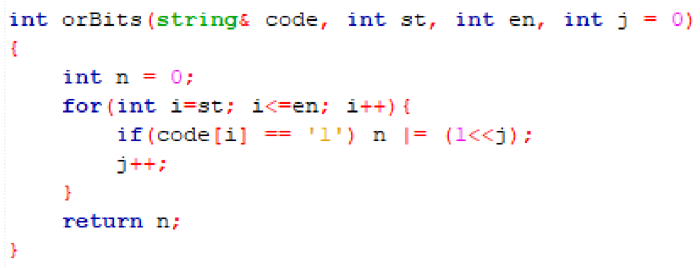
To do this, we used the map data structure to save the instruction name according to the mentioned logic along with if statements to differentiate between instruction formats.

```
// 32 bits map
map<int,  string> opcodes;  // opcode  ,  type/instruction "r" --> R , "i" --> I , "s" --> S , "b" --> B , "j" --> jalr , "f" --> fence
map<pair<int, string>, string> func3;  // <func3,  type>  ,  sub-type/instruction
map<pair<int, string>, string> func7;    //  <func7,  sub-type>  ,  instruction

// 16 bits map
map<int, string> zero16;
```

The maps are initialized using createMaps() function that reads the data of the 32-bit instructions from files (opcodes.txt, func3.txt, func7.txt)

Second: Loading registers and immediate values:



```
int orBits(string& code, int st, int en, int j = 0)
{
    int n = 0;
    for(int i=st; i<=en; i++){
        if(code[i] == '1') n |= (1<<j);
        j++;
    }
    return n;
}
```

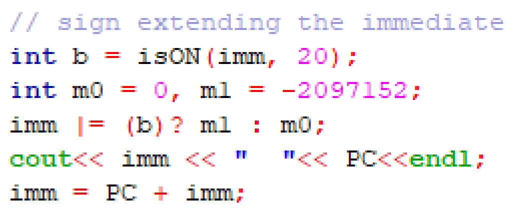
We relied on function: orBits() to get the data from the instruction word/half word. This function basically ORs an integer (initialized by zero) with the instruction word/half word passed from a specific start bit to an end bit in the instruction, and you may initialize the start bit of the number as well.

This allowed us to get the register numbers. For the immediate values that are placed in nonsequential bits, we kept ORing with the desired bits until we get the value.

Third: Sign extending the immediate values:

Since we are saving the immediate values in integers, we need to sign extend them to express the negative value in instructions like addi, jal, etc.

To do so we followed this logic:



```
// sign extending the immediate
int b = isON(imm, 20);
int m0 = 0, m1 = -2097152;
imm |= (b)? m1 : m0;
cout<< imm << "  "<< PC<<endl;
imm = PC + imm;
```

b → represents the sign bit.

m0 → mask with all 0s

m1 → mask with all 1s starting from the bit after the sign bit (last bit the in the immediate)

if b is 1, then we OR the immediate with m1, otherwise we OR the immediate with m0.

Fourth: Program Counter:

a program counter is created and initialized by 0. It is incremented according to the instruction size (32 bits → increment by 4 bytes, 16 bits → increment by 2 bytes)

Fifth: Creating Labels for instructions that use labels (jal, branch, c.j, …) BONUS
To do so, we first scanned all the instructions used and whenever we find any of these, we saved the targeted address (Program Counter + Immediate) in a map called **labels**. This map maps the address to a string ("label" + "number of the label")

Then, the program counter is returned back to 0 and we start decoding the instructions in the output file, and whenever the PC reaches a saved address in the **labels**, it prints the label first. And whenever an instruction that uses labels is decoded, the name of the label is included beside the address.

Limitations

The only limitations in our program exist for the compressed instructions (C extension) because we were not able to spot the exact format of compressed instructions from RISC_V specifications document, moreover we tried to search for these formats, but some of them was hard to be found. So, we decided to make assumptions for theses formats by the way we used to in other instructions, following the same pattern. For example: the instruction format for **c.add** instruction would be **c.add rd, rs2**

The format instructions are divided into 3 groups according to the opcode (2 bits) of the 16 bit instructions into (00 opcode instructions), (01 opcode instructions) and (10 opcode instructions).

There are the formats that the compressed instructions are decoded to:

Group 0: (00 op)

c.add14spn rd, nzuimm[2:9]

c.fld rd, rs1, unimm[3:7]

c.lw rd, rs1, unimm[2:6]

c.flw rd, rs1, unimm[2:6]

c.fsd rs1, rs2, unimm[3:7]

c.fdw rs1, rs2, unimm[2:6]

Group 1: (01 op)

c.nop

c.addi rd, nzimm[0:5]

c.jal imm[1:11]

c.li rd, imm[0:5]

c.addi16sp nzimm[4:9]

c.lui rd, nzimm[12:17]

c.srli rd, nzuimm[0:5]

c.srai rd, nzuinn[0:5]

c.srai64 rd

c.andi rd, imm[0:5]

c.sub rd, rs2

c.xor rd, rs2

c.and rd, rs2

c.subw rd, rs2

c.addw rd, rs2

c.j imm[1:11]

c.beqz rs1, imm[1:8]

c.bnez rs1, imm[1:8]

Group 2: (10 op)

c.slli rd, nzuimm[0:5]

c.fldsp rd, uimm[3:8]

c.lwsp rd, uimm[4:9]

c.flwsp rd, uimm[2:7]

c.jr rs1

c.mv rd

c.ebreak

c.jalr rs1

c.add rd, rs2

c.fsdsp rs2, uimm[3:8]

c.swsp rs2, uimm[2:7]

c.fswsp rs2, uimm[2:7]

Known Issues

Our program is working properly without any issues, we tested it using all the sample tests provided and using another manually made sample for the compressed instructions and it outputs the right assembly code corresponding to the machine codes in the test cases.

Contributions of Each Team Member

Member 1 (Basant):

- Developing the get32 function and its corresponding maps.
- Loading registers and immediate values.
- Creating the orBits function that eases the excluding of the required bits to be encoded.
- Handling the program counter and creating labels map to perform the bonus part (Using labels for branch and Jal instructions).
- Getting arguments using cmd.

Member 2 (Hashem):

- Developing the get16 function and its corresponding maps.
- Creating files (opcodes.txt, func3.txt, func7.txt).
- Sign extending the immediate values.
- Developing formats for the compressed instructions.
- Handling the program counter and creating labels map to perform the bonus part (Using labels for branch and Jal instructions).

General Note:

We were generally working together all the time at the same place. So, we contributed in each part of the project together.