Project
Computer Organization

Introduction

- This project, required us to design and implement a custom assembler and a custom simulator for a given ISA.
- We were not restricted to any programming language. However, our program had to read from stdin and write to stdout.
- We needed to write our code in the directories described in the README file.
- We also had to track our changes via Github.
- Committing our code to the repository periodically to prevent any loss of code due to system failures or any other issues was an important part of the project.

PARTS:

1. Designing and Implementing the assembler.
2. Designing and Implementing the simulator.

For 1 and 2:

1. Assembler: The test cases are divided into 3 sets:
    a. ErrorGen: These tests are supposed to generate errors
    b. simpleBin: These are simple test cases which are supposed to generate a binary.
    c. hardGen: These are hard test cases which are supposed to generate a binary.
2. Simulator: The test cases are divided into 2 sets:
    a. simpleBin: These are simple test cases which are supposed to generate a trace.
    b. hardGen: These are hard test cases which are supposed to generate a trace.

Description

ISA description:

Consider a 16 bit ISA with the following instructions and opcodes, along with the syntax of an assembly language which supports this ISA.

The ISA has 6 encoding types of instructions. The description of the types is given later.

Opcode	Instruction	Semantics	Syntax	Type
00000	Addition	Performs reg1 = reg2 + reg3. If the computation overflows, then the overflow flag is set	add reg1 reg2 reg3	A

00001	Subtraction	Performs reg1 = reg2 - reg3. In case reg3 > reg2, 0 is written to reg1 and overflow flag is set.	sub reg1 reg2 reg3	A
00010	Move Immediate	Performs reg1 = $Imm where Imm is a 8 bit value.	mov reg1 $Imm	B
00011	Move Register	Performs reg1 = reg2.	mov reg1 reg2	C
00100	Load	Loads data from mem_addr into reg1.	ld reg1 mem_addr	D
00101	Store	Stores data from reg1 to mem_addr.	st reg1 mem_addr	D
00110	Multiply	Performs reg1 = reg2 x reg3. If the computation overflows, then the overflow flag is set.	mul reg1 reg2 reg3	A
00111	Divide	Performs reg3/reg4. Stores the quotient in R0 and the remainder in R1.	div reg3 reg4	C

01000	Right Shift	Right shifts reg1 by $Imm, where $Imm is an 8 bit value.	rs reg1 $Imm	B
01001	Left Shift	Left shifts reg1 by $Imm, where $Imm is an 8 bit value.	ls reg1 $Imm	B

01010	Exclusive OR	Performs bitwise XOR of reg2 and reg3. Stores the result in reg1.	xor reg1 reg2 reg3	A
01011	Or	Performs bitwise OR of reg2 and reg3. Stores the result in reg1.	or reg1 reg2 reg3	A
01100	And	Performs bitwise AND of reg2 and reg3. Stores the result in reg1.	and reg1 reg2 reg3	A
01101	Invert	Performs bitwise NOT of reg2. Stores the result in reg1.	not reg1 reg2	C
01110	Compare	Compares reg1 and reg2 and sets up the FLAGS register.	cmp reg1 reg2	C
01111	Unconditional Jump	Jumps to mem_addr, where mem_addr is a memory address.	jmp mem_addr	E
10000	Jump If Less Than	Jump to mem_addr if the less than flag is set (less than flag = 1), where mem_addr is a memory address.	jlt mem_addr	E

10001	Jump If Greater Than	Jump to mem_addr if the greater than flag is set (greater than flag = 1), where mem_addr is a memory address.	jgt mem_addr	E
10010	Jump If Equal	Jump to mem_addr if the equal flag is set (equal flag = 1), where mem_addr is a memory address.	je mem_addr	E
10011	Halt	Stops the machine from executing until reset	hlt	F

where reg(x) denotes register, mem_addr is a memory address (must be an 8-bit binary number), and Imm denotes a constant value (must be an 8-bit binary number).

The ISA has 7 general purpose registers and 1 flag register. The ISA supports an address size of **8 bits**, which is **double byte addressable**. Therefore, each address fetches two bytes of data. This results in a total address space of 512 bytes. **This ISA only supports whole number arithmetic.** If the subtraction results in a negative number; for example "3 - 4", the reg value will be set to 0 and overflow bit will be set. All the representations of the number are hence unsigned.

The registers in assembly are named as R0, R1, R2, ... , R6 and FLAGS. Each register is 16 bits.

Note: "`mov reg $Imm`": This instruction copies the Imm(8bit) value in the register's lower 8 bits. The upper 8 bits are zeroed out.

Example:

Suppose R0 has `1110_1010_1000_1110` stored, and **`mov R0 $13`** is executed.

The final value of R0 will be `0000_0000_0000_1101`.

FLAGS semantics

The semantics of the flags register are:

- Overflow (V): This flag is set by `add`, `sub` and `mul`, when the result of the operation overflows. This shows the overflow status for the last executed instruction.
- Less than (L): This flag is set by the "`cmp reg1 reg2`" instruction if reg1 < reg2
- Greater than (G): This flag is set by the "`cmp reg1 reg2`" instruction if the value of reg1 > reg2
- Equal (E): This flag is set by the "`cmp reg1 reg2`" instruction if reg1 = reg2

The default state of the FLAGS register is all zeros. If an instruction does not affect the FLAGS register, then the state of the FLAGS register is reset to 0 upon the execution.

The structure of the FLAGS register is as follows:

Unused 12 bits	V	L	G	E
15	3	2	1	0

The only operation allowed in the FLAGS register is "`mov reg1 FLAGS`", where reg1 can be any of the registers from R0 to R6. This instruction reads FLAGS register and writes the data into reg1. All other operations on the FLAGS register are prohibited.

The `cmp` instruction can implicitly write to the FLAGS register. Similarly, conditional jump instructions can implicitly read the FLAGS register.

Example:
R0 has 5, R1 has 10

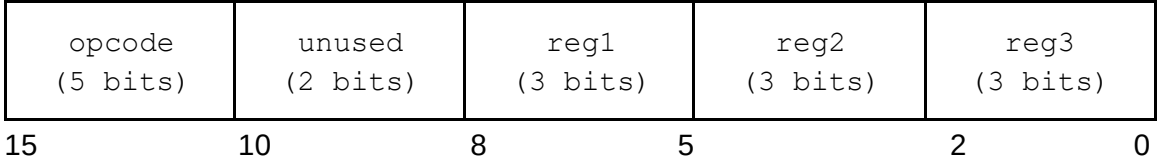
Implicit write: **`cmp R0 R1`** will set the L (less than) flag in the FLAGS register.

Implicit read: **`jlt 10001001`** will read the FLAGS register and figure out that the L flag was set, and then jump to address `10001001`.

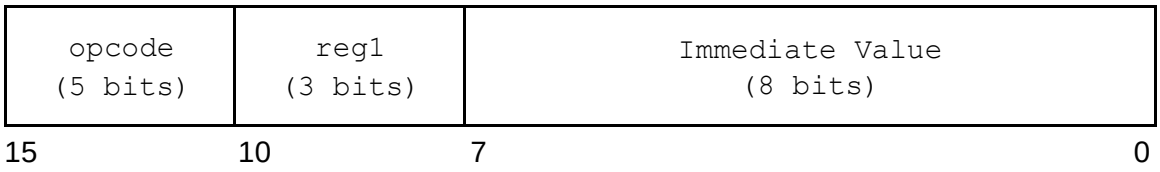
Binary Encoding

The ISA has 6 types of instructions with distinct encoding styles. However, each instruction is of 16 bits, regardless of the type.

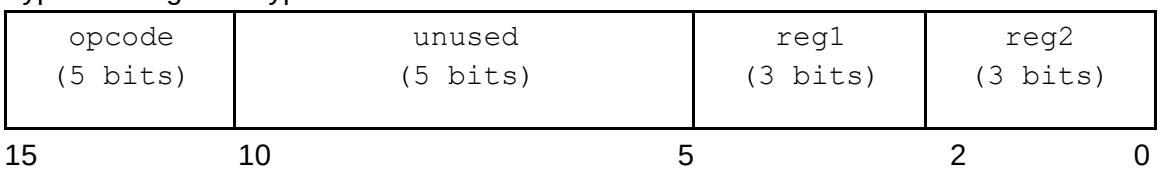
- Type A: 3 register type



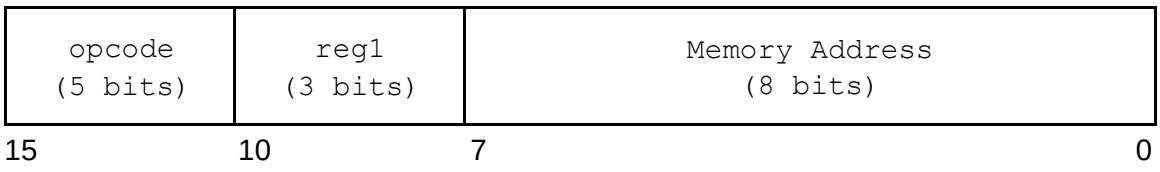
- Type B: register and immediate type



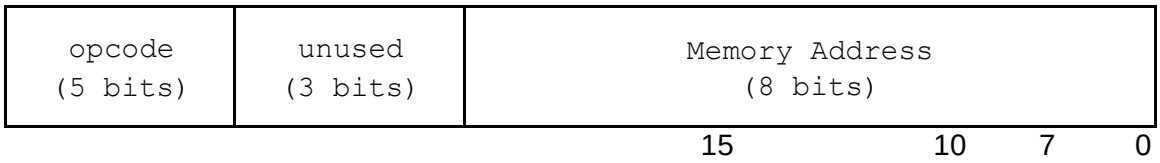
- Type C: 2 registers type



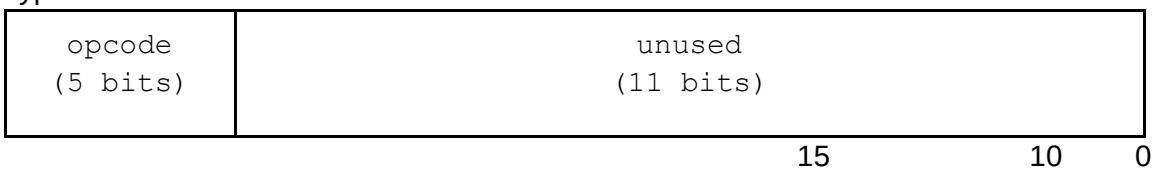
- Type D: register and memory address type



- Type E: memory address type



- Type F: halt



Binary representation for the register are given as follows:-

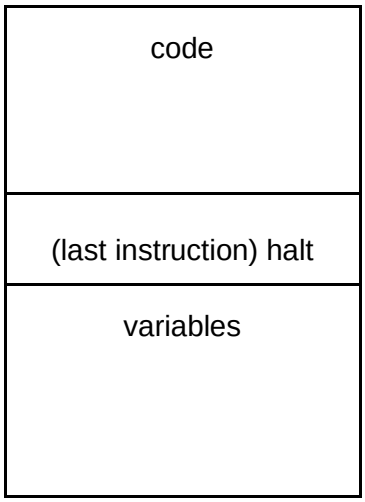
Register	**Address**
R0	000
R1	001
R2	010
R3	011
R4	100

R5	101
R6	110
FLAGS	111

Executable binary syntax

The machine exposed by the ISA starts executing the code provided to it in the following format, until it reaches `hlt` instruction. There can only be one `hlt` instruction in the whole program, and it must be the last instruction. The execution starts from the 0^{th} address. The ISA follows von-neumann architecture with a unified code and data memory.

The variables must be allocated in the binary in the program order.



Parts:

1: Assembler:

Program an assembler for the aforementioned ISA and assembly. The input to the assembler is a text file containing the assembly instructions. Each line of the text file may be of one of 3 types:

- Empty line: Ignore these lines
- A label followed by an instruction
- An instruction
- A variable definition

Each of these entities have the following grammar:

- The syntax of all the supported instructions is given above. The fields of an instruction are whitespace separated. The instruction itself might also have whitespace before it. An instruction can be one of the following:
    - The opcode must be one of the supported mnemonic.
    - A register can be one of R0, R1, … R6, and FLAGS.

    - A mem_addr in jump instructions must be a label.
    - A Imm must be a whole number <= 255 and >= 0.
    - A mem_addr in load and store must be a variable.
- A label marks a location in the code and must be followed by a colon (:). No spaces are allowed between label name and colon(:). A label name consists of alphanumeric characters and underscores. A label followed by the instruction may looks like:

```
mylabel: add R1 R2 R3
```

- A variable definition is of the following format: `var xyz`

  which declares a 16 bit variable called xyz. This variable name can be used in place of mem_addr fields in load and store instructions. **All variables must be defined at the very beginning of the assembly program.** A variable name consists of alphanumeric characters and underscores.
- Each line may be preceded by whitespace.

The assembler should be capable of:

1. Handling all supported instructions
2. Handling labels
3. Handling variables
4. Making sure that any illegal instruction (any instruction (or instruction usage) which is not supported) results in a syntax error. In particular you must handle:
    a. Typos in instruction name or register name
    b. Use of undefined variables
    c. Use of undefined labels
    d. Illegal use of FLAGS register
    e. Illegal Immediate values (less than 0 or more than 255)
    f. Misuse of labels as variables or vice-versa
    g. Variables not declared at the beginning
    h. Missing `hlt` instruction
    i. `hlt` not being used as the last instruction
    j. Wrong syntax used for instructions (For example, `add` instruction being used as a type B instruction)

    You need to generate distinct readable errors for all these conditions. If you find any other illegal usage, you are required to generate a “General Syntax Error”. **The assembler must print out all these errors.**

    If the code is error free, then the corresponding binary is generated. The binary file is a text file in which each line is a 16bit binary number written using 0s and 1s in ASCII. The assembler can write less than or equal to 256 lines.

Input/Output format:

- The assembler must read the assembly program as an input text file (stdin).

- The assembler must generate the binary (if there are no errors) as an output text file (stdout).
- The assembler must generate the error notifications along with line number on which the error was encountered (if there are errors) as the output text file (stdout). **In case of multiple errors, the assembler may print any one of the errors.**

Example of an assembly program

```
var X mov R1
$10 mov R2
$100 mul R3
R1 R2 st R3
X hlt
```

The above program will be converted into the following machine code

```
0001000100001010
0001001001100100
0011000011001010
0010101100000101
1001100000000000
```

2: Simulator:

You need to write a simulator for the given ISA. The input to the simulator is a binary file (the format is the same as the format of the binary file generated by the assembler in Q1). The simulator should load the binary in the system memory at the beginning, and then start executing the code at address 0. The code is executed until `hlt` is reached. After execution of each instruction, the simulator should output one line containing an 8 bit number denoting the program counter. This should be followed by 8 space separated 16 bit binary numbers denoting the values of the registers (R0, R1, … R6 and FLAGS).

<PC (8 bits)><space><R0 (16 bits)><space>...<R6 (16 bits)><space><FLAGS (16 bits)>. The output must be written to stdout. Similarly, the input must be read from stdin. After the program is halted, print the memory dump of the whole memory. This should be 256 lines, each having a 16 bit value

<16 bit data>
<16 bit data>
…..
<16 bit data>

Your simulator must have the following distinct components:

1. Memory (MEM): MEM takes in an 8 bit address and returns a 16 bit value as the data. The MEM stores 512bytes, initialized to 0s.
2. Program Counter (PC): The PC is an 8 bit register which points to the current instruction.
3. Register File (RF): The RF takes in the register name (R0, R1, … R6 or FLAGS) and returns the value stored at that register.

4. Execution Engine (EE): The EE takes the address of the instruction from the PC, uses it to get the stored instruction from MEM, and executes the instruction by updating the RF and PC.

The simulator should follow roughly the following pseudocode:

```
initialize(MEM);           // Load memory from stdin
PC = 0; halted             // Start from the first instruction
= false;
white(not halted)
{
       Instruction = MEM.getData(PC);                // Get current instruction
       halted, new_PC = EE.execute(Instruction);     // Update RF compute new_PC
       PC.dump();                                    // Print PC
       RF.dump();                                    // Print RF state
       PC.update(new_PC);                            // Update PC
}
MEM.dump()                                           // Print memory state
```