Background

Programs run on computers by having the hardware (or system software) execute basic operations called instructions. Many languages (such as Java) represent the program to be executed as byte codes which are very similar to machine instructions. In this assignment, you will build an interpreter for a simple machine language.

At their lowest level, computers operate by manipulating information stored in registers and memory. Registers and memory are like variables in C or Java; in fact inside the computer that is how variables get stored. In addition to data, memory also stores the instructions to execute. The basic operation of a computer is to read an instruction from memory, execute it and then move to the instruction stored in the next memory location. Typical instructions will read one or more values (called operands) from memory and produce a result into another register or memory location. For example, an instruction might add the values stored in two registers, R3 and R4 and then store the result in the register R5. Other instructions might just move data from one place to another (between registers or between a register and a memory location). A final type of instruction, called a branch instruction, is used to change what instruction is executed next (to allow executing if and looping statements).

The Simulated Computer

The simulated computer has a memory that contains $2^{16}$ (65,536) words of memory, each 32 bits long.

In addition to memory, the computer has 16 registers that can be used to hold values. Two of the registers are special. R0 is hardwired to 0 and writing to it is legal, but doesn’t change it, but reading from it returns 0. R1 is the “Program Counter” and always contains the address of the next instruction to execute. R1 can be read like a normal register, but can only be modified using special instructions (Bal and Beq), any other attempt to modify it is an Invalid Instruction (including using it as the register1 value of Branch) R2-R15 are General Purpose Registers, and can be read or written.

Description of instructions

\textbf{lw} <register\textsubscript{1}> <register\textsubscript{2}> <memory>

Copies the value stored in the memory location \( <memory> + <\text{register}\textsubscript{2}> \) into the register location \(<\text{register}\textsubscript{1}>\). For example if \( \text{register}\textsubscript{2} = \text{R0} \) and \( \text{memory} = 42 \), the data stored in location 42 is loaded into the register. However, if the second operand was \( \text{R4} \) (rather than \( \text{R0} \)) and \( \text{R4} \) contained 22, the values would be loaded from memory location \( 42 + 22 = 64 \). (opcode 0)
li <register₁> <number>
Copies the supplied number #<number> into the register location <register₁> (opcode 1).

mv <register₁> <register₂>
Copies the value stored in <register₂> into <register₁> (opcode 2)

sw <register₁> <register₂> <memory>
Copies the value stored in <register₁> into memory location <memory> + the contents of <register₂> (opcode 3)

add <register₁> <register₂> <register₃>
Adds the value stored in <register₁> to the value stored in <register₂> and stores the result into <register₃> (opcode 4)

neg <register₁>
Negates the value stored in <register₁> (i.e. 1 becomes –1), (opcode 5)

beq <register₁> <register₂> <memory>
If the value stored in <register₁> is equal to the value stored in <register₂>, change the program counter (next instruction to execute) to execute the instruction stored in <memory> next. If they are not equal the instruction has no effect (opcode 6)

bal <register₁> <register₂> <memory>
Stores the current value of R1 (program counter) into <register₁>, Sets the value of R1 (program counter) to the contents of <register₂> plus the value of the <memory> field. (opcode 7)

read <register₁>
Read an integer from standard input, and store the value in <register₁> (opcode 8)

write <register₁>
Write the integer value stored in <register₁> to standard output. The instruction should print a newline after the integer. (opcode 9)

halt
Terminates execution of the machine. No operands are used (opcode 10)

**Instruction Format**
In the computer, instructions are stored in memory locations just like data. However, each bit of the memory location is used to identify different parts of the instruction. The first 4 bits indicate which instruction is stored (called an opcode). For example, a load instruction is op code 1 so 0001₂ would be stored in the 4 bits of the opcode. The next three parts of the instruction store the number of the registers to be used. For example, if
a load instruction was trying to load something into R5, it would have $0101_2$ stored in Register1 field. The final part of the instruction is the Memory location field. This field is 16 bits long and can describe any of the 65,536 memory locations in the computer. Not all instructions use all of the fields. For example, the load instruction does not use Register2 or Register3. The following figure shows the layout of a memory location storing an instruction.

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Opcode</th>
<th>Register1</th>
<th>Register2</th>
<th>Register3</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size (bits)</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>16</td>
</tr>
</tbody>
</table>

```c
typedef struct {
    unsigned int opcode:4;
    unsigned int r1:4;
    unsigned int r2:4;
    unsigned int r3:4;
    unsigned int address:16;
} instruction;
```

Consider the following assembly language instructions and how they would be stored:

<table>
<thead>
<tr>
<th>li R5 12345</th>
<th>add R6 R7 R8</th>
<th>halt</th>
</tr>
</thead>
<tbody>
<tr>
<td>instruction example1; example1.opcode = 1; example1.reg1 = 5; example1.address = 12345;</td>
<td>instruction example2; example2.opcode = 4; example2.reg1 = 6; example2.reg2 = 7; example2.reg3 = 8;</td>
<td>instruction example3; example3.opcode = 10;</td>
</tr>
</tbody>
</table>

**The Assignment**

In this assignment, you will write a collection of functions to manipulate instructions. Here are the functions you will write:

```c
void printInsn(instruction insn);
```

This function is passed an instruction and prints out the instruction in a format that is easier for humans to read. You will use the switch statement to figure out what instruction to print and you will use the printf function to do the printing. You should not print a newline character. The operands should be printed in lower case as shown earlier in the assignment.

The output format should look like the definitions of the instructions given above. So for example if the instruction contained:

```
0000 0011 0000 0000 0000 0000 0000 1110
```

<table>
<thead>
<tr>
<th>opCode</th>
<th>r1</th>
<th>r2</th>
<th>r3</th>
<th>address</th>
</tr>
</thead>
</table>

```c
```
you would print `lw R3 14`. Notice there is exactly one space and no comma or other punctuation between the instruction and operands. Only those operands that are used by a given instruction should be printed. For example the second and third registers are not used by `lw`, so they were not printed above. If the instruction is invalid for any reason, print “Invalid Instruction”.

void disassemble(memoryLocation mem[], int limit);

You will also write a second function that will take an array of memory as a parameter and print out the instructions from 0 up to passed limit of limit.

The disassemble function prints the address of each memory location followed by a colon and a space “: ” then the output of your printInsn function. So if the sample instruction above is in memory location 0, the first line of output for the function would be:

```
0: li R3 14
```

bool validInstruction(instruction insn);

Returns true if the instruction is valid, and false if the instruction is invalid. Instructions can be invalid if their op code is invalid (not 0-10), if the destination register writes to one of the special registers (R0 and R1).

We will supply a main program for this assignment. We will also supply several header files. The file machine.h contains a definition of the machine instructions and memory. disassemble.h contains the prototypes for the functions you will write. All of your code should be placed in the file named disassemble.c that we supply. You will obtain the files by typing the UNIX command line:

```
cp ~/212files/p1.tar.gz
zunzip p1.tar.gz
tar xvf p1.tar
```