Simulator and Assembler

The Assignment

You are provided with a description of a simple computer's Instruction Set Architecture (ISA), including its instruction set, instruction format, and register list. Your task is to write a simulator and assembler for the described architecture. After completing this assignment, it will be possible to write programs in assembly, process them with your assembler, and execute them on your simulator.

Educational Objectives

This assignment is designed to help build your understanding of simple processors and the fetch-decode-execute cycle as well as the role of assembly language in software development. It will also provide reinforcement in C Language Programming, especially in the use of the bit-wise operators and function pointers.

The Instruction Set Architecture (ISA)

The *Instruction Set Architecture (ISA)* provides a view of a processor's features as seen from the perspective of an assembly or machine language programmer. For our purposes, this means that the ISA describes the instructions that the processor understands, the way those instructions are presented to the processor, the register set, and the way memory is organized. A real-world processor's ISA would also include a few additional items, such as its interrupt and/or exception handling facilities, basic data types, and different modes of operation, e.g. supervisor vs. normal mode.

Registers are a special type of memory built into the processor. They basically serve as special variables accessible to the *Arithmetic and Logic Unit (ALU)*, the brains of the processor that actually executes most instructions. You'll find that most instructions operate on the values in the registers instead of operating directly on memory. As a result, you'll *load* values from memory into registers, operate on them, and then *store* them back into memory. This arrangement is called a *load-store* architecture.

There are six (6) *general purpose* registers that can be used for any purpose. Additionally, there is a *zero* register that always contains a constant 0 value to be used for initializing other registers to 0. The same register can be read and written within the same instruction, so, for example "A = A + 5" is legal as is "A = A + A".

The *program counter (PC)* is a special purpose register that keeps track of the current address in memory, the address that the processor is currently executing. Since instructions are 4-bytes wide, the PC moves forward by four bytes with each instruction cycle. The *instruction register (IR)* is a scratch register used to decode instructions. The PC is 24-bits wide. The IR is 32-bits wide.

The simulated machine has a 2-byte word size, so registers and immediate values are 2-bytes wide. Integers are signed using the high-bit. In other words, the highest bit is 0 if the number is positive and 1 otherwise. This bit is set correctly by the mathematical operations.

The simulated system also has two flags, which are set by various instructions: The *overflow* flag is set upon a mathematical operation to true, if an operation overflows (carries) outside of 16 bits, and to false otherwise. The *compare* flag is set upon a comparison operation to true if the comparison operation is true, and it is set to false otherwise. The flags cannot be set directly.

Ports are a mechanism for accessing input and output devices that are independent from main memory. Port #15 is a terminal device console used for output. Port #0 is a terminal device console used for input.

Each reads or writes one character at a time, translating from that character to the ASCII value. The terminal device has sufficient buffering to avoid dropping character in normal applications.

The system's main memory is byte addressable. In other words, bytes are addressed and addresses range from byte-0 through byte 2^{24}-1.

Ports

```
Purpose     Binary      Notes
-----------------------------------------
Input       0000 0000   Returns ASCII code of character read from terminal
Output      0000 1111   Writes ASCII code of character to terminal
```

Registers

```
Register    Number      Notes
-----------------------------------------
Z           000         Constant: Always zero (0)
A           001
B           010
C           011
D           100
E           101
F           110
G           111
PC                      Program Counter. Not addressable
IR                      Instruction register. 32-bits wide. Non addressable.
```

Instructions

```
---------------Control------------
Instruction       -Op-  ----Address---
HLT               0000 0000 0000 0000 0000 0000 0000 0000   Stop simulation
JMP               0001 0000 AAAA AAAA AAAA AAAA AAAA AAAA   Jump
CJMP              0010 0000 AAAA AAAA AAAA AAAA AAAA AAAA   Jump if true
OJMP              0011 0000 AAAA AAAA AAAA AAAA AAAA AAAA   Jump if overflow

------------Load/Store------------
Instruction       -Op- Reg0 ------------Value------------
LOAD              0100 0RRR AAAA AAAA AAAA AAAA AAAA AAAA   Load
STORE             0101 0RRR AAAA AAAA AAAA AAAA AAAA AAAA   Store
LOADI             0110 0RRR 0000 0000 IIII IIII IIII IIII   Load Immediate
NOP               0111 0000 0000 0000 0000 0000 0000 0000   Store Immediate

-----------------Math--------------
Instruction       -Op- Reg0 Reg1 Reg2 0000 0000 0000 0000
ADD               1000 0RRR 0RRR 0RRR 0000 0000 0000 0000   Reg0 = Reg1+Reg2
SUB               1001 0RRR 0RRR 0RRR 0000 0000 0000 0000   Reg0 = Reg1-Reg2

-----------Device/IO--------------
Instruction       -Op- Reg0 0000 0000 0000 0000 --Port --
IN                1010 0RRR 0000 0000 0000 0000 PPPP PPPP   Read port->Reg0
OUT               1011 0RRR 0000 0000 0000 0000 PPPP PPPP   Write Reg0->Port

------------Comparison-------------
Instruction       -Op- Reg0 Reg1
EQU               1100 0RRR 0RRR 0000 0000 0000 0000 0000   Cflg=(Reg0==Reg1)
LT                1101 0RRR 0RRR 0000 0000 0000 0000 0000   Cflg=(Reg0<Reg1)
LTE               1110 0RRR 0RRR 0000 0000 0000 0000 0000   Cflg=(Reg0<=Reg1)
NOT               1111 0000 0000 0000 0000 0000 0000 0000   Cflg=(!Cflg)
```

Writing and Assembling a Program By Hand

A program is a text file with one instruction per line. Each line should be a very simple space-delimited line. It can include comments, which begin with a #. When you first write out the program by hand, number the lines, ignoring comment lines. Use the line numbers in place of addresses for jumps.

```
# This program gets two numbers, A and B, from the user
# Then prints out the numbers A through B
0. LOADI    A     1           # Get the number 1 into A register
1. LOADI    B     48          # 48 is int value of 'A', pseudo-constant
2. IN       C     0000        # Get starting point in ASCII
3. SUB      D     C     B     # Get integer value of input character
4. IN       C     0000        # Get ending point in ASCII
5. SUB      E     C     B     # Convert ending from ASCII to int val

# Starting value is D, ending value is E
6. LTE      D     E           # (D <= E)
7. NOT                        # !(D <= E) -> (D > E)
8. CJMP     {Line D}          # If (D > E) from above, exit loop
9. ADD      C     D     B     # Convert D as int into ASCII
A. OUT      C     1111        # Print out the number
B. ADD      D     D     A     # Increment D
C. JMP      {Line 6}          # Go back to the top of the loop
D. HLT
```

Once you are done writing out the program, multiple each line number by 4. This will give you the address of that line of code within memory. This is because each instruction is 4 bytes long. Rewrite the program replacing the line numbers with addresses in hexadecimal.

```
# This program gets two numbers, A and B, from the user
# Then prints out the numbers A through B
0. LOADI    A     1           # Get the number 1 into A register
4. LOADI    B     48          # 48 is int value of 'A', pseudo-constant
8. IN       C     0000        # Get starting point in ASCII
C. SUB      D     C     B     # Get integer value of input character
10. IN      C     0000        # Get ending point in ASCII
14. SUB     E     C     B     # Convert ending from ASCII to int val

# Starting value is D, ending value is E
18. LTE     D     E           # (D <= E)
1C. NOT                       # !(D <= E) -> (D > E)
20. CJMP    34                # If (D > E) from above, exit loop
24. ADD     C     D     B     # Convert D as int into ASCII
28. OUT     C     1111        # Print out the number
2C. ADD     D     D     A     # Increment D
30. JMP     18                # Go back to the top of the loop
34. HLT
```

Now, convert this program into binary, by translating each mnemonic into the binary equivalent shown in the "Instructions" section. Do the same with each value.

```
# This program gets two numbers, A and B, from the user
# Then prints out the numbers A through B

# Get the number 1 into A register
# 0. LOADI  A     1
     0110   0001  0000 0000 0000 0000 0000 0001
```

```
# 4. LOADI  B     48          # Subtract 48: ascii char → int value
     0110   0010  0000 0000 0000 0000 0011 0000

# Get starting point in ASCII
# 8. IN     C     0000
     1010   0011  0000 0000 0000 0000 0000 0000

# Get integer value of input character
# C. SUB    D     C     B
     1001   0100  0011  0010 0000 0000 0000 0000

# Get ending point in ASCII
# 10. IN    C     0000
     1010   0011  0000 0000 0000 0000 0000 0000

# Convert ending from ASCII to int val
# 14. SUB   E     C     B
     1001   0101  0011 0010 0000 0000 0000 0000

# Starting value is D, ending value is E
# (D <= E)
# 18. LTE   D     E
     1110   0100  0101 0000 0000 0000 0000 0000

# !(D > E) -> (D > E)
# 1C. NOT
     1111   0000 0000 0000 0000 0000 0000 0000

# If (D > E) from above, exit loop
# 20. CJMP  34
     0010   0000 0000 0000 0000 0000 0010 0010

# Convert D as int into ASCII
# 24. ADD   C     D     B
     1000   0011  0100  0010 0000 0000 0000 0000

# Print out the number
# 28. OUT   C     1111
      1011  0011  1111 0000 0000 0000 0000 0000

# Increment the number
# 2C. ADD   D     D     A
      1000  0100  0100  0001 0000 0000 0000 0000

# Go back to the top of the loop
# 30. JMP  18
      0001 0000 0000 0000 0000 0000 0001 0010

# 34.HLT
      0000 0000 0000 0000 0000 0000 0000
```

Lastly, convert the binary representation into hexadecimal. This is a fully assembled program.
Each line represents a single 4-byte instruction. The first line resides at address 0, the second line resides at address 4, the third at address 8, and so on.

Although real-world computers use an actual binary representation without new lines, we think you'll appreciate this format which captures the same information in a more parseable, more human-readable way.

```
0x61000001
0x62000030
0xA3000000
0x94320000
0xA3000000
0x95320000
0xE4500000
0xF0000000
0x20000022
0x83420000
0xB3F00000
0x84410000
0x10000012
0x00000000
```

The Assembler (Look, Ma': No Hands!)

Your assembler is called "sas". It accepts an assembly source file, parses it, and translates it into an executable object file. It uses exactly the same process as illustrated previously. In other words, the program parses each line of the source file and translates it into hexadecimal. Each op code is then recognized, looked up in a table, and outputted. Then, each operand is recognized, translated, and outputted.

The name of the input and output files are specified at the command line:

```
sas input.s output.o
```

The Simulator

The simulator models the processor, the main memory, and the described terminal devices via ports. When run, it loads a compiled program into memory and simulates its execution until it halts.

Memory

Memory can be simulated as simply an array of unsigned chars of this size. This provides a byte-indexed memory. In order to interpret the lower addresses that contain the program text, you can assign an "unsigned int" pointer to the same array, or make use of a union. This way, when accessing word-oriented instructions, you can use the "unsigned int *", giving you a whole word at a time – just remember to divide the desired address by 4 because of pointer arithmetic.

Since relatively few programs will require a simulation of the entire physical memory, the program should accept the size of physical memory as a command line argument. It need only simulate the requested amount of memory.

Loading A Program Into Simulated Memory

To load the program, read each input line into memory. We suggest that you first do this by reading it into a temporary variable, an unsigned int, and then copying this into the unsigned char array simulating memory. The code below illustrates the idiom:

```
unsigned int instruction;
fscanf ("0x%x", &instruction);
memcpy (memory + address, &instruction, sizeof(instruction));
```

You are, of course, free to take a different approach. But, we strongly suggest the technique above – it dodges some potentially complicating issues. If you insist on taking a different approach, ask a friend in 15-213 about Endian-ness first. You're probably in for more of a ride than you expect. Really.

The Register File

Since general purpose registers are 16-bits wide, they can be implemented as an array of unsigned shorts. This way the register number can serve as the index. Since the special purpose registers are larger, they should be implemented using unsigned ints, a 32-bit type.

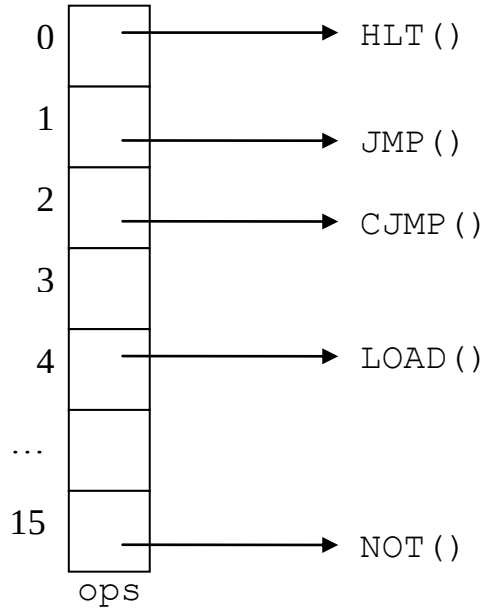
The Processor and Execution

Assuming that all of the instructions have been loaded into memory, the processor can be simulated using a fetch-decode-execute work loop. During execution, the processor fetches the next instruction by loading the instruction referenced by the PC from memory into the *instruction register (IR)*. This is simply a scratch register used by the processor to decode the instruction.

Once this is done, the PC is incremented to prepare for the eventual next processor cycle. The next step is to decode the op code, the 4-bit number associated with the operation. This can be done by shifting right to eliminate the other bits. Careful here: unless you are using unsigned ints, you'll get bitten while shifting because of the sign bit.

At this point, you are able to dispatch the instruction. Because one goal of this assignment is to reinforce your understanding of function pointers, you are asked to use an array of function pointers for this purpose.

For each instruction, you should create a function. Then, each of these functions should be mapped into an array of function pointers, where each function's index is its op code. This makes the dispatch very easy. For example, if your array is called *ops*, the dispatch is as easy as **(ops[opcode])()*.



Once within the instruction, you'll need to decode the operands and execute. Before considering the implementation, take a second look at the instructions. Notice that there are six different instruction formats. Write macros that use bit masks to decode the operands present in each of these formats:

```
REG0
REG1
REG2
IMMEDIATE
ADDRESS
```

For example:

```
#define REG0 ((IR >> 24) & 0x0F)
```

Once the operands have been decoded, you are free to implement the logic of the instruction. Perform any needed computation, then write back the values to the registers. If the instruction is a jump you'll need to update the PC. The simulation ends when the HLT instruction is called. This instruction should exit rather than returning.

Once the function implementing the operation returns, the simulation can loop back to the beginning of the fetch-decode-execute loop and repeat.

The Terminal Devices via Ports

The terminal devices are simulated only within the IN and OUT instructions. Implementing the OUT instruction is as simple as a write(1, …). The IN instruction can similarly be implemented using a read(0,…).

For those interested, this does mean that the processor is polling the port until it is able to supply or accept a character – and this isn't a very good use of a processor. But, since this is really straight-forward to implement with read() and write(), it is a really good use of a programmer.

The Simulator

The simulator should be called "ssim". It should actually load and then execute a correct, compiled program. It should be implemented in accordance with the model described above.

The physical memory size and executable file name should be specified as command-line arguments:

- argv[1] provides the size of your physical memory in bytes.
- argv[2] provides the name of the executable program

For example:

```
ssim 0x1000 sampleprogram
```

The simulator does not include any model exception handling facility. As a consequence, it cannot handle error states, such as invalid executables, bad memory accesses, and the like. Should any of these circumstances arise, it simply terminates with an informative error message.