CSci 460: Operating Systems **Spring 2001, Zemoudeh**
Handout 2: Phase I – VM and Assembler **Due Wednesday April 25**

California State University, San Bernardino
Computer Science Department

Write a C++ program to simulate a simple 16 bit CPU or Virtual Machine (VM) which consist of 4 general purpose registers (`r[0]-r[3]`), a program counter (`pc`), an instruction register (`ir`), a status register (`sr`), a stack pointer (`sp`), a `clock`, an arithmetic and logic unit (ALU), a 256 word memory (`mem`) with `base` and `limit` registers, and a disk.
The registers are represented by a vector of 4 integers:

```
vector<int> r(4);
```

`mem` is represented by a vector of 256 integers:

```
vector<int> mem(256);
```

The least significant four bits of `sr` are reserved for LESS, EQUAL, GREATER, and CARRY status in that order:

···	L	E	G	C

ALU is part of the logic of your program, disk can be represented by a collection of files, and the rest of the components can be represented by simple variables in your program. We only use the lower 16 bits of the variables. The VM supports one of two general formats for the instructions:

OP	RD	I	RS	unused
15:11	10:9	8	7:6	5:0

OP	RD	I	unsigned memory ADDR or signed CONST
			7:0

where OP stands for opcode, RD stands for (register) destination, I stands for immediate, and RS stands for (register) source. When I is 0, the next 2 bits specify the source register and the next 6 bits are unused. When I is 1, immediate address mode is in effect: depending on the instruction, the next 8 bits are treated as either an unsigned 8 bit address (ADDR), or an 8 bit twos complement constant (CONST). This implies $0 \leq \text{ADDR} < 256$ and and $-128 \leq \text{CONST} < 128$. If a field is unused, it is considered "don't care" and it can be set to any bit pattern. We will set don't cares to all zeros. In case of load instruction (I == 0), the least significant 8 bits are the ADDR field. The following table lists all the instructions which must be supported by the assembler and the VM.

VM Instructions

OP I	Instruction	Semantic in Pseudo C++ Syntax	
00000 0	load RD ADDR	r[RD] = mem[ADDR]	
00000 1	loadi RD CONST	r[RD] = CONST	
00001 0	store RD ADDR	mem[ADDR] = r[RD]	
00010 0	add RD RS	r[RD] = r[RD] + r[RS]	(sets CARRY)
00010 1	addi RD CONST	r[RD] = r[RD] + CONST	(sets CARRY)
00011 0	addc RD RS	r[RD] = r[RD] + r[RS] + CARRY	(sets CARRY)
00011 1	addci RD CONST	r[RD] = r[RD] + CONST + CARRY	(sets CARRY)
00100 0	sub RD RS	r[RD] = r[RD] − r[RS]	(sets CARRY)
00100 1	subi RD CONST	r[RD] = r[RD] − CONST	(sets CARRY)
00101 0	subc RD RS	r[RD] = r[RD] − r[RS] − CARRY	(sets CARRY)
00101 1	subci RD CONST	r[RD] = r[RD] − CONST − CARRY	(sets CARRY)
00110 0	and RD RS	r[RD] = r[RD] `&` r[RS]	
00110 1	andi RD CONST	r[RD] = r[RD] `&` CONST	
00111 0	xor RD RS	r[RD] = r[RD] `^` r[RS]	
00111 1	xori RD CONST	r[RD] = r[RD] `^` CONST	
01000 0	compl RD	r[RD] = `~` r[RD]	
01001 0	shl RD	r[RD] = r[RD] << 1, shift-in-bit = 0	(sets CARRY)
01010 0	shla RD	shl arithmetic	(sign extends)
01011 0	shr RD	r[RD] = r[RD] >> 1, shift-in-bit = 0	(sets CARRY)
01100 0	shra RD	shr arithmetic	(sign extends)
01101 0	compr RD RS	if r[RD] < r[RS] set LESS reset EQUAL and GREATER;	
		if r[RD] == r[RS] set EQUAL reset LESS and GREATER;	
		if r[RD] > r[RS] set GREATER reset EQUAL and LESS	
01101 1	compri RD CONST	if r[RD] < CONST set LESS reset EQUAL and GREATER; ...	
01110 0	getstat RD	r[RD] = SR	
01111 0	putstat RD	SR = r[RD]	
10000 0	jump ADDR	`pc` = ADDR	
10001 0	jumpl ADDR	if LESS == 1, `pc` = ADDR, else do nothing	
10010 0	jumpe ADDR	if EQUAL == 1, `pc` = ADDR, else do nothing	
10011 0	jumpg ADDR	if GREATER == 1, `pc` = ADDR, else do nothing	
10100 0	call ADDR	push VM status; `pc` = ADDR	
10101 0	return	pop and restore VM status	
10110 0	read RD	read new content of r[RD] from `.in` file	
10111 0	write RD	write r[RD] in `.out` file	
11000 0	halt	halt execution	
11001 0	noop	no operation	

Since `mem` consist of a set of integers (bits), any program written in the above language has to be translated to its equivalent object code to be loaded in `mem` and run by the VM. Write an assembler for the above assembly language. For example, when the assembler encounters

loadi 2 71

it translates the instruction into

0000010101000111

where from left to right 00000 represents loadi or load, 10 represents r[2], I == 1 (therefore loadi is the opcode), and 01000111 is the CONST. 1351 is the object code for this instruction, since

$0000010101000111_2 = 1351_{10}$

As an example, your assembler should produce the object code on the right for the assembly program on the left. This program does not perform anything meaningful! It is intended to compare some related instructions. Note you may comment the rest of a line using an exclamation point (!). Your assembler should ignore comments.

Assembly Prog		Object Prog
load 1 69		581
load 2 69		1093
loadi 2 −123	! set register 2	1413
loadi 2 71		1351
add 0 3		4288
addi 0 −56		4552
jump 10	! produces runtime error	32778
halt		49152

The assembler reads and inputs an assembly program and outputs its corresponding object code. Any assembly program must have a `.s` suffix. Its corresponding object code must have the same name with a `.o` suffix. The assembler should catch any out-of-range error for ADDR and CONST and stop producing object code. Also any value other than 0, 1, 2, or 3 for RD or RS is illegal; and any opcode other than the ones listed in the VM Instruction table is illegal. The assembler should be designed and implemented as a C++ class.

Next, design and implement a C++ class called VM to interpret the object codes. Store the object code to be run in the top of the memory, this implies setting `pc` and `base` registers to 0 and `limit` register to the size of the object program. The VM then enters an infinite loop of instruction fetch-execute cycle:

```
top:    instruction fetch (ir ← mem[pc])
        pc++
        set OP, RD, I, RS, ADDR, CONST from ir
        execute the instruction specified by OP and I
        go to top
```

This loop terminates when a halt instruction is executed or some unexpected error occurs. Following the above file suffix convention, when executing a `.o` program and a read instruction is encountered, the input is read from a `.in` file with the same name. In case of a write instruction the output is printed into a `.out` file.

The `clock` is initialized to 0, load and store instructions take 4 clock ticks each, read and write instructions take 64 clock ticks each, and the rest of the instructions take 1 clock tick each.

Be careful in handling sign extension. For example, if in loadi instruction CONST = $11111100_2 = -4_{10}$, then to store it in some r[RD] register, it must be sign extended to 1111111111111100_2 (still -4_{10}). Sign extention occurs every time a short constant (in this case 8 bits) is assigned to a longer register (in this case 16 bits); look for this every time negative numbers are involved.

call and return instructions need special attention. As part of the execution of the call instruction the status of the VM must be pushed on to the stack. The status of the VM consist of `pc, r[0]-r[3],` and `sr`. The stack grows from bottom of (high) memory up, therefore initially `sp = 256`. After the first call it is deccremented by 6 as the current values of `pc, r[0]-r[3],` and `sr` are pushed on to the stack. When the return instruction is executed, `sp` is incremented by 6 as the new values of `pc, r[0]-r[3],` and `sr` are popped and restored from the stack in reverse order.

noop instruction can be used as a place holder in the memory, for example, to store a temporary value in memory and later retrieve it. Use noop instructions either at the beginning or at the end of the program; do not use them in the middle unless there is a very good reason.

You should write your assembler, VM, and the Operating System (OS) in as much as possible in an object oriented and flexible fashion! This is specially the case as new requirements are added from one phase to the next. Use separate compilation for this (large) project. This means the class `Assembler` must be defined in files `Assembler.h` and `Assembler.cpp` and the class `VirtualMachine` must be defined in files `VirtualMachine.h` and `VirtualMachine.cpp`. Compile `Assembler.cpp` and `VirtualMachine.cpp` separately using the `-c` option. `os.cpp` includes `main()`, `#include "Assembler.h"` and `#include "VirtualMachine.h"`. `main()` declare instances of `Assembler` and `VirtualMachine` and makes the proper calls:

```
...
#include "Assembler.h"
#include "VirtualMachine.h"
main(int argc, char *argv[])
{
        Assembler as;
        VirtualMachine vm;
        ...
} // main
```

Compile and link to make your rudimentary OS (rudimentary only at this phase!):

```
g++ -o os os.cpp Assembler.o VirtualMachine.o
```

and run `prog.s` in your OS environment:

```
os prog
```

Make sure that your program works correctly for `test.s`:

```
read  0
loadi 1 -2
add   0 1      ! subtract 2 from value read
write 0
halt
```

and also for `fact.s`:

```
! main for factorial program
        loadi  0 1     ! line 0, R0 = fact(R1)
        read   1       ! input R1
        call   6       ! call fact
        load   0 33    ! receive result of fact
        write  0
        halt
! fact function
        compri 1 1     ! line 6
        jumpe  14      ! jump over the recursive call to fact if
        jumpl  14      ! R1 is less than or equal 1
        call   16      ! call mult (R0 = R0 * R1)
        load   0 34    ! receive result of mult
        subi   1 1     ! decrement multiplier (R1) and multiply again
        call   6       ! call fact
        load   0 33
```

```
        store  0 33    ! line 14, return R0 (result of fact)
        return
! mult function
        loadi  2 8     ! line 16, init R2 (counter)
        loadi  3 0     ! init R3 (result of mult)
        shr    1       ! line 18 (loop), shift right multiplier set CARRY
        store  2 35    ! save counter
        getstat 2      ! to find CARRY's value
        andi   2 1
        compri 2 1
        jumpe  25      ! if CARRY==1 add
        jump   26      ! otherwise do nothing
        add    3 0
        shl    0       ! make multiplicand ready for next add
        load   2 35    ! restore counter
        subi   2 1     ! decrement counter
        compri 2 0     ! if counter > 0 jump to loop
        jumpg  18
        store  3 34    ! return R3 (result of mult)
        return
        noop           ! line 33, fact return value
        noop           ! line 34, mult return value
        noop           ! line 35, mult counter
```

On the due date hand in print outs of `Assembler.h`, `Assembler.cpp`, `VirtualMachine.h`, `VirtualMachine.cpp`, and `os.cpp` and demonstrate the program for the above assembly programs and a third program given close to the due date. Your grade will be based on 40% correctness, 20% clarity and conciseness, 20% documentation and proper indentation, and 20% "object orientedness" and flexibility.