# Project - Phase I: Assembler & Virtual Machine

## CSE 460: Operating Systems
## Winter 2019, Zemoudeh
## School of Computer Science and Engineering
## California State University, San Bernardino

Write a C++ program to simulate a simple 16-bit CPU (Virtual Machine). VM consists 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 (`clock`), an Arithmetic and Logic Unit (ALU), a 256 word Memory (`mem` with `base` and `limit` registers), and a Disk.

Represent General Purpose Registers with a vector of 4 integers, `mem` with a vector of 256 integers, `pc` with an integer, `ir` with an integer, ...

```
class VirtualMachine {
private:
    int msize;
    int rsize;
    int pc, ir, sr, sp, clock;
    vector<int> mem;
    vector<int> r;
    int base, limit;
public:
    VirtualMachine(): msize(256), rsize(4), clock(0)
    {
        mem = vector<int>(msize);
        r = vector<int>(rsize);
    }
...
};
```

Since this is a 16-bit machine, we only use the lower 16 bits of the variables. The least significant five bits of `sr` are reserved for OVERFLOW, LESS, EQUAL, GREATER, and CARRY in that order, the rest are "don't-care" (d):

| d | ... | d | V | L | E | G | C |
|---|---|---|---|---|---|---|---|
| 15 | | 5 | 4 | 3 | 2 | 1 | 0 |

ALU is part of the logic of your program, and disk is represented by a collection of files. `clock` could be alternatively represented by a class.

VM supports two instruction formats.

Format 1:

| OP | RD | I | RS | UNUSED |
|---|---|---|---|---|
| 15:11 | 10:9 | 8 | 7:6 | 5:0 |

Format 2:

| OP | RD | I | ADDR/CONST |
|---|---|---|---|

15:11 10:9 8 7:0

where OP (bits 11 to 15 from right to left) stands for opcode,
RD (bits 9 and 10) stands for register-destination,
I (bit 8) stands for immediate,
and RS (bits 6 and 7) 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 two's complement constant (CONST). This implies 0 <= ADDR < 256 and -128 <= CONST < 128.

`load` and `loadi` are special instructions, they both use format 2: when I = 0, we use ADDR, when I = 1, we use CONST.

If a field is unused, it is considered don't-care, and it can be set to any bit pattern, but in this project we will set don't-cares to all zeros.

To simplify writing programs for the VM, we need an assembly language and its corresponding assembler. The following table lists all instructions supported by the Assembler and in turn VM.

| **VM Instruction Set** | | | | |
|---|---|---|---|---|
| **OP** | **I** | **Instruction** | **Semantic in Pseudo C++ Syntax** | **Additional Action** |
| 00000 | 0 | load RD ADDR | r[RD] = mem[ADDR] | |
| 00000 | 1 | loadi RD CONST | r[RD] = CONST | |
| 00001 | 1 | store RD ADDR | mem[ADDR] = r[RD] | |
| 00010 | 0 | add RD RS | r[RD] = r[RD] + r[RS] | Set CARRY |
| 00010 | 1 | addi RD CONST | r[RD] = r[RD] + CONST | Set CARRY |
| 00011 | 0 | addc RD RS | r[RD] = r[RD] + r[RS] + CARRY | Set CARRY |
| 00011 | 1 | addci RD CONST | r[RD] = r[RD] + CONST + CARRY | Set CARRY |
| 00100 | 0 | sub RD RS | r[RD] = r[RD] - r[RS] | Set CARRY |
| 00100 | 1 | subi RD CONST | r[RD] = r[RD] - CONST | Set CARRY |
| 00101 | 0 | subc RD RS | r[RD] = r[RD] - r[RS] - CARRY | Set CARRY |
| 00101 | 1 | subci RD CONST | r[RD] = r[RD] - CONST - CARRY | Set 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 | d | compl RD | r[RD] = ~ r[RD] | |
| 01001 | d | shl RD | r[RD] = r[RD] << 1, shift-in-bit = 0 | Set CARRY |

| | | | | |
|---|---|---|---|---|
| 01010 | d | shla RD | shl arithmetic | Set CARRY & Sign Extend |
| 01011 | d | shr RD | r[RD] = r[RD] >> 1, shift-in-bit = 0 | Set CARRY |
| 01100 | d | shra RD | shr arithmetic | Set CARRY & Sign Extend |
| 01101 | 0 | compr RD RS | if r[RD] < r[RS] set LESS reset EQUAL and GREATER;<br>if r[RD] == r[RS] set EQUAL reset LESS and GREATER;<br>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;<br>if r[RD] == CONST set EQUAL reset LESS and GREATER;<br>if r[RD] > CONST set GREATER reset EQUAL and LESS | |
| 01110 | d | getstat RD | r[RD] = SR | |
| 01111 | d | putstat RD | SR = r[RD] | |
| 10000 | 1 | jump ADDR | pc = ADDR | |
| 10001 | 1 | jumpl ADDR | if LESS == 1, pc = ADDR, else do nothing | |
| 10010 | 1 | jumpe ADDR | if EQUAL == 1, pc = ADDR, else do nothing | |
| 10011 | 1 | jumpg ADDR | if GREATER == 1, pc = ADDR, else do nothing | |
| 10100 | 1 | call ADDR | push VM status; pc = ADDR | |
| 10101 | d | return | pop and restore VM status | |
| 10110 | d | read RD | read new content of r[RD] from .in file | |
| 10111 | d | write RD | write r[RD] into .out file | |
| 11000 | d | halt | halt execution | |
| 11001 | d | noop | no operation | |

Since `mem` consists of a set of integers (bits), any program written in the above assembly language (`*.s`) has to be translated to its equivalent object program (`*.o`) to be loaded in `mem` and run by the VM. Therefore, we must translate (assemble) each assembly instruction into an object code. The sequence of object codes is called an object program.

We need an assembler to perform the above translation. For example, when the Assembler encounters

```
        loadi 2 71
```

it translates the instruction to

```
        0000010101000111
```

where from left to right
00000 is the opcode for loadi or load
10 represents r[2]
1 represent immediate addressing (I == 1) and therefore loadi is the opcode
and 01000111 is CONST 71.

1351 is the object code for this instruction, since $0000010101000111_2 = 1351_{10}$.

As an example, your assembler should produce the object program on the right from 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 9 | ! r[1] = mem[9] | 00521 |
| load 2 9 | ! r[2] = mem[9] | 01033 |
| loadi 2 -123 | ! r[2] = -123 (set reg 2) | 01413 |
| loadi 2 71 | | 01351 |
| add 0 3 | ! r[0] += r[3] | 04288 |
| addi 0 -56 | ! r[0] += -56 | 04552 |
| jump 10 | ! pc = 10 (runtime error) | 33034 |
| store 2 20 | ! m[20] = r[2] (runtime error) | 03348 |
| halt | | 49152 |
| noop | | 51200 |

The Assembler reads an assembly program and outputs its corresponding object program. An assembly program must have a `.s` suffix, and its corresponding object program must have the same name with a `.o` suffix. Assembler creates a `.o` file. VM reads in this `.o` file, stores it in memory, and starts executing it. Assembler should catch any out-of-range error for ADDR and CONST and stop producing object codes. 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 above VM Instruction Table is illegal. The Assembler should be designed and implemented as a C++ class.

Design and implement a C++ class for the virtual machine (`VirtualMachine`) to interpret object programs. Store the object program 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 object program. VM then enters into instruction fetch-execute cycle (an infinite loop):

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

VM initializes the `clock` to 0 after loading the object program in memory.
Each of load, store, call, and return instructions take 4 clock ticks to execute.
Each of read and write instructions take 28 clock ticks to execute.
The rest of the instructions take 1 clock tick each to execute.
Note that `loadi`, which is the set instruction and uses an immediate operand, takes 1 clock tick and not 4 ticks. This is because `loadi` does not access memory.
**Print the final value of clock in `.out` file.**

Be careful when 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 extension 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.

Since VM is a 16-bit machine, it's best to always zero out the high-order 16 bits of variables that represent the registers in VM and just work with the low-order 16 bits. For example, after an operation on register 0 that might result in "spill over" in high-order bits, perform the following operation:
`r[0] &= 0xffff;`

`call` and `return` instructions need special attention. As part of the execution of `call` instruction the status of VM must be pushed on to stack. Status of VM consists of `pc, r[0]-r[3],` and `sr.` The stack grows from the bottom of memory up, therefore initially `sp = 256.` After a call, `sp` is decremented by 6 as the values of `pc, r[0]-r[3],` and `sr` in the VM are pushed on to stack. When a `return` instruction executes, `sp` is incremented by 6 as values of `pc, r[0]-r[3],` and `sr` are popped from stack and restored in VM registers. When `sp >= 256` stack is empty, and when `sp < limit+6` stack is full.

`noop` instruction can be used as a place holder in memory to store a temporary value and later retrieve it.

Write your Assembler, VM, and OS in an object oriented and extensible fashion! This is specially the case as new requirements are added in the next two phases. Use separate compilation for this (large) project. This means class `Assembler` must be defined in files:

```
        Assembler.h
        Assembler.cpp
```

and class `VirtualMachine` must be defined in files:

```
        VirtualMachine.h
        VirtualMachine.cpp
```

Compile `Assembler.cpp` and `VirtualMachine.cpp` separately using the `-c` option:

```
        $ g++ -c Assembler.cpp
        $ g++ -c VirtualMachine.cpp
```

These two commands produce `Assembler.o` and `VirtualMachine.o`.
`os.cpp` includes:

```
        #include "Assembler.h"
        #include "VirtualMachine.h"
        main()
```

where `main()` declares 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 in this phase!):

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

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

```
        $ os prog.s
```

which assembles `prog.s` into `prog.o`, loads `prog.o` into memory, and finally invokes VM to run the program.
Make sure that your program works correctly for `test.s` program:

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

for `add5.s` program:

```
        read 0
        call 5
        load 0 8
        write 0
        halt
        addi 0 5       ! add5 function
        store 0 8
        return
        noop           ! location for return value
```

and for `fact.s` program:

```
! 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
os.cpp
```

and demonstrate your program for the above assembly programs. Your grade will be based on:

- 40% correctness and efficiency
- 20% clarity and conciseness
- 20% documentation and proper indentation
- 20% "object oriented-ness" and extensibility