CSCI-GA.3033-016

Virtual Machines: Concepts and Applications

Homework Assignment 2
(Programming Assignment)

In this lab you will implement a process VM that uses only interpretation. The VM must fetch instructions from a text file that contains a simplified version of the x86 assembly and then execute it. So the output, to the stdout, is the execution of each instruction. So basically you are implementing a process VM that uses fetch & dispatch emulation to emulate a hypothetical machine on a x86 machine (i.e. on your CIMS account).

To make your life easier, we are giving you a reference VM (refvm) with which you can compare the output.

Starting from next page, you will find the syntax of the input language that you have to emulate.

Submission steps:

1. Name your source code: vm.c. If you wrote it with a different language you need to include a README file telling us how to compile it. The main restriction is that it has to work on our Linux CIMS machine because this is where we are going to grade it.
2. Your VM must accept a *.vm file from the command line and the output must match the output of the refvm program.
3. Add all your file (the source code, README, and any other file you think is needed) to a zip file named: lastname.firstname.zip where lastname is your last name and first name is your first name.
4. Upload your zip file to NYU classes in assignment called *HW 2*

Simplified x86 assembly syntax

VALUES

Values can be specified in decimal, hexadecimal, or binary. By default, values without a base specifier are assumed to be in decimal. Any value prepended with "0x" is assumed to be in hexadecimal.

Values can also be specified using base identifiers. To specify the value "32", for example:

- Just specifying 32 means it is in decimal
- 0x20 for hexadecimal
- 20|h also for hexadecimal
- 100000|b for binary

REGISTERS

This hypothetical machine has 17 registers. *Register names are written lower-case*.

(EAX - EDX, General Purpose)
EAX
EBX
ECX
EDX

ESI
EDI

ESP - Stack pointer, points to the top of the stack
EBP - Base pointer, points to the base of the stack

EIP - Instruction pointer, this is modified with the jump commands, never directly

R08 - R15, General Purpose

MEMORY

Memory addresses are specified using brackets, *in units of four bytes*. It is important to note that certain areas of memory are used for vital program function, such as the memory used by the stack. Overwriting such parts of memory will likely cause the program to crash.

To specify the 256th word in the address space, you can use [256], [100|h], [0x100], or [100000000|b]. Any syntax that's valid when specifying a value is valid when specifying an address.

So [0] is the address of the first 4-byte word. [1] is the address of the following 4-byte word, not the second byte of the memory.

LABELS

Labels are specified by appending a colon to an identifier. Labels must be specified at the beginning of a line or on their own line.
Examples
start:
loop:
end:

INSTRUCTIONS

In many instructions, one of the operands can be a constant, or a memory address.

I. Memory

mov arg0, arg1
Moves value specified from arg1 to arg0

II. Stack

push arg
Pushes arg onto the stack

pop arg
Pops a value from the stack, storing it in arg

pushf
Pushes the FLAGS register to the stack

popf arg
Pops the flag register to arg

III. Calling Conventions

call label
Push the current address to the stack and jump to the subroutine specified

ret
Pop the previous address from the stack to the instruction pointer to return control to the caller

IV. Arithmetic Operators

inc arg
Increments arg

dec arg
Decrements arg

add arg0, arg1
Adds arg1 to arg0, storing the result in arg0 (i.e. arg0 = arg0 + arg1)

sub arg0, arg1
Subtracts arg1 from arg0, storing the result in arg0 (i.e. arg0 = arg0 - arg1)

mul arg0, arg1
Multiplies arg1 and arg0, storing the result in arg0 (i.e. arg0 = arg0 * arg1)

div arg0, arg1
Divides arg0 by arg1, storing the quotient in arg0 and the remainder in a remainder register.

mod arg0, arg1
Same as the '%' (modulus) operator in C. Calculates arg0 mod arg1 and stores the result in the remainder register.

rem arg
Retrieves the value stored in the remainder register, storing it in arg
This is the only way to read the remainder register, as it is not a general purpose.

V. Binary (i.e. bitwise) Operators

not arg
Calculates the binary NOT of arg, storing it in arg

xor arg0, arg1
Calculates the binary XOR of arg0 and arg1, storing the result in arg0

or arg0, arg1
Calculates the binary OR of arg0 and arg1, storing the result in arg0

and arg0, arg1
Calculates the binary AND of arg0 and arg1, storing the result in arg0

shl arg0, arg1
Shift arg0 left by arg1 places

shr arg0, arg1
Shifts arg0 right by arg1 places

VI. Comparison

cmp arg0, arg1
Compares arg0 and arg1, storing the result in the FLAGS register

VII. Control Flow Manipulation

jmp address
Jumps to an address or label

je address
Jump if equal

jne address
Jump if not equal

jg address
Jump if greater

jge address
Jump if equal or greater

jl address
Jump if lesser

jle address]
Jump if lesser or equal

For example:
cmp eax, ebx
jle here
means if (eax <= ebx) then jump to here

VIII. Input / Output

prn arg
Print an integer