Module 1 – The virtual machine

This module discusses the first steps of implementing a virtual machine that meets the project's requirements. It will need to handle a variety of instructions as well as memory management. This module's memory management will be simple but will lay the groundwork for later virtual and paged memory implementation.

The CPU

Overview

The foundation of a virtual machine is the CPU.

The CPU will have the following specifications:

Feature	Notes
Opcodes consisting of a 1-byte opcode and two optional integer (4-byte) operands	See below for details
Fourteen integer registers.	Registers are addressed 1-14. Registers 11 and above are reserved. #11 – Instruction Pointer #12 – Current Process Id #13 – Stack Pointer #14 – Global Memory start
A stack pointer	Points to the stack top and moves downward. Stored in register 13
A system clock measured in ticks. Each instruction takes one tick.	The CPU updates this after each instruction
An interrupt system to allow custom code upon an interrupt	This is a table of functions that get called in different circumstances. We will set up the table and its infrastructure for use later. It contains pointers to functions such as “QuantumExpired”, “MemoryNotPresent”, “ProgramTerminated”, “ProcessPaused” and others. The CPU triggers these to handle outside events.
An instruction pointer (IP)	The instruction pointer register is special and is 32-bits wide. • It is not accessible to the programmer. You can’t read or assign to it . • CPU modifies it as the program executes. • It must be saved by the operating system on function calls and context switches. • Stored in register 11
A memory manager	An abstraction used to manage memory access. Initially, it will provide direct access, but we encapsulate it to allow it to evolve later.
Two flag registers: • Sign Flag • Zero Flag These flags can be implemented as Booleans. You don’t need to use bit manipulation.	Sign Flag: Used when comparing two quantities results in a non-zero result. X < y is equivalent to x-y >0 and x-y <0. The sign flag is set as a result, indicating whether a number has a negative sign. Zero Flag: This is set when comparing 2 quantities results in a zero (both quantities are equal).

Instructions

Opcodes

Since this project intends to teach Operating system design and implementation, the machine is incomplete because it does not provide an entire repertoire of instructions. Each opcode is 1-byte long bytes long and is followed by two 4-byte arguments (even if they are unused) of the following format:

- Note – 4 bytes for each opcode lets us simplify parsing. In a real OS this would be different.

Opcode Memory Format

- <opcode> <argument 1> <argument 2>
    - Example: incr r1 -> This translates to 1 0001 0000 in memory
- All instructions are multiple bytes.
- All instructions take exactly one clock cycle to execute, independent of the actual operation of the instruction.
- Context switches only happen on instruction boundaries.
- Constants are indicated by the operator $ in front of them.
- Registers are indicated by r1 … r10.
- Arguments can be positive or negative. Precede negative numbers with a minus sign
- The stack pointer (register 10) is often called SP

Opcode File Format

Opcodes are represented in files in assembly language, which you will need to translate into machine language. Here are the rules:

- <OpCodeTerm> <Param1>, <Param2> ; single line comment
    - OpCodeTerm – An opcode descriptor like incr or movi. These are case insensitive.
    - Param1 and Param 2 – These may contain one of 4 types of values:
        - Address – a numeric address in memory. 1234
        - Numeric constant - $1 – the constant number 1
        - Character constant - @a – the character a. Represented in memory as its ASCII value stored in a 4-byte area.
        - Register identifier – r1 – register 1 - 10.
    - Example: Incr r1 ; increment the contents of register 1.

Notation:

rx	Contents of rx
[rx]	Contents of memory at the address stored in rx (indirection)
#x	Numeric constant – can be used anywhere $a can be used

$a	Character constant - can be used anywhere #x can be used
1111	Address

Opcode	Example	Meaning
Incr	Incr r1	rx ⇐ rx + 1.
Addi	Addi r1, #3	rx ⇐ rx + c
Addr	Addr r1, r2	r1 ⇐ r1 + r2.
Pushr	Pushr r2	Push rx onto the stack. Decrement SP by 4.
Pushi	Pushi #3	Pushes the constant x onto the stack. Decrement SP by 4
Movi	Movi r2, #4	rx ⇐ n
Movr	Movr r2, r3	rx ⇐ ry
Movmr	Movmr r2,r3	rx ⇐ [ry] [] indicates contents stored at the address in ry
Movrm	Movrm r3,r2	[rx] ⇐ ry
Movmm	Movmm r2,r3	[rx] ⇐ [ry] – this is a memory-to-memory move
Printr	Printr r5	Display contents of register 1
Printm	Printm r5	Display memory contents at the address in register x.
Printcr	Printcr r1	Displays contents of register 1 as a character
Printcm	Printcm r5	Display memory contents at the address in register x as a character
Jmp	Jmp r5	Transfer control to the instruction whose address is rx bytes relative to the current instruction. Rx may be negative. For example: If r1 = 10, jmp r1 will add 10 bytes to the instruction pointer.
Jmpi	Jmpi #36	Transfer control to the instruction with offset x from the current IP
Jmpa	Jmpa #1234	Transfer control to the absolute address x.
Cmpi	Cmpi r4, #5	Subtract y from register rx. If rx < y, set the sign flag. If rx > y, clear the sign flag. If rx == y, set zero flag.
Cmpr	Cmpr r4, r5	Like cmpi, except now both operands are registers.
Jlt	Jlt r5	If the sign flag is set, jump to the instruction whose offset is rx bytes from the current instruction.
Jlti	Jlti #10	If the sign flag is set, jump to the instruction whose offset is x bytes from the current instruction.
Jlta	Jlta #1234	If the sign flag is set, jump to address x.

Opcode	Example	Meaning
Jgt	Jgt r5	if the sign flag is clear, jump to the instruction whose offset is rx bytes from the current instruction
Jgti	Jgti #10	if the sign flag is clear, jump to the instruction whose offset is x bytes from the current instruction
Jgta	Jgta #1234	If the sign flag is clear, jump to address x.
Je	Je r5	if the zero flag is clear, jump to the instruction whose offset is rx bytes from the current instruction.
Jei	Jei #10	if the zero flag is clear, jump to the instruction whose offset is x bytes from the current instruction.
Jea	Jea #10	if the zero flag is clear, jump to address x
Call	Call r2	Call the subroutine at offset r2 bytes from the current instruction. The address of the next instruction to execute after a return is pushed on the stack.
Callm	Callm r1	Call the procedure at offset [rx] bytes from the current instruction. The address of the next instruction to execute after a return is pushed on the stack.
Ret	Ret	Pop the return address from the stack and transfer control to this instruction.
Exit	Exit	Exit. This opcode is executed by a process to exit and be unloaded. Another process or the idle process must now be scheduled.
Popr	Popr r3	Pop the contents at the top of the stack into register rx. The Stack pointer is decremented by 4.
Popm	Popm rx	Pop the contents at the top of the stack into [rx]. The Stack pointer is decremented by 4.
Sleep	Sleep r1	Sleep the # of clock cycles as indicated in rx. If the time to sleep is 0, the process sleeps indefinitely. There will be additional rules added to this in later steps
Input	Input r1	Read the next 64-bit value into rx from the keyboard.
Inputc	Inputc r1	Read the next character from the keyboard and store it in rx as its ASCII value
SetPriority	SetPriority rx	Sets the priority of the current process to [rx]
SetPriorityI	SetpriorityI #x	Sets the priority of the current process to x

Architectural Notes:

If you don't use classes well, this will be a more complicated assignment. Some ideas:

1. An operating systems class to drive the whole program.
2. A CPU class that implements the CPU and runs programs.
3. A memory manager class
4. An enumeration containing all the opcodes.
5. A program class representing the static program read from disk and converted into bytes.
    a. Read in files formatted like this:

```
movi r1, $1    ; store 1 in register 1
incr r1        ; increment r1
printr r1      ; print r1 – should print 2
```

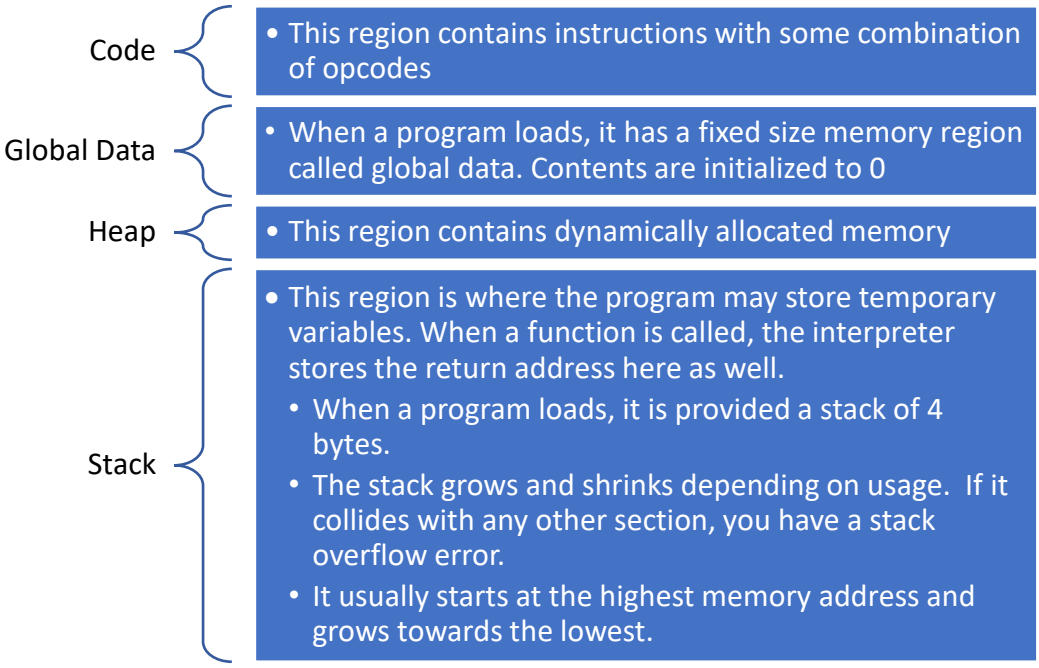
    b. This would be stored in memory (represented here as hex) as:
        i. 00 00 00 07 00 00 00 01 00 00 00 01
        ii. 00 00 00 01 00 00 00 01 00 00 00 00
        iii. 00 00 00 0B 00 00 00 01 00 00 00 00

All memory access should be performed through a memory management unit class because we will abstract this into virtual memory later.

It would be wise to have the memory management unit have a separate class for physical memory, as later, we will need to differentiate it from virtual memory.

Setting up a program in memory

In an OS, a program is divided into multiple areas:



The code area will be whatever size the program requires. Fix the global data, heap, and stack size. Using a configuration file would be a great idea here.