**A 16-bit Operating System**

**Systems Specification**

**Mentor:**

Dr. Kay Zemoudeh

***Programmed by:***

Alexis Lopez

Edwin Navar

Date: March 2019

Last Modified: October 2020

**Table of Contents**

1. **Assembler & Virtual Machine**

1.1 Operating System Design 1

1.2 Instruction Format 1

1.3 Virtual Machine Instruction Set 2

1.4 Instruction Fetch-Execute Cycle 4

1.5 Call and Return Instructions 6

1.6 Run and Compile 6

1.7 Test Programs 7

1. **Process Management**

2.1 To be continued … 9

1. **Memory Management**

**1 Assembler & Virtual Machine**

**1.1 Operating System Design**

This is a basic 16-bit Operating System (OS) written on C++ as the main programming language. The program simulates a 16-bit CPU (Virtual Machine). VM 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 (clock), an Arithmetic and Logic Unit (ALU), a 256 word Memory (mem with base and limit registers), and a Disk.

To represent our virtual machine, we create a class to represent the four general purpose registers with 4 integers, *mem* with a vector of 256 integers, *pc* with an integer, *ir* with an integer and so on.



**1.2 Instruction Format**

Due to the nature of this 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

The ALU is part of the logic of the program, and disk is represented by a collection of files (\*.s, \*.in, \*.out) files. The \*.s files are the assembly code that the OS take as instructions to perform. The \*.in is the input value that are loaded into a register for later use. The \*.out file is the output of the file. The \*.out file is automatically generated by the OS, but the user must create the \*.s and \*.in files.

Since we want our OS to be able to work with addresses and immediate values, we need two types of format so we can distinguish between them when assembling our program. The VM supports two instruction formats.

**Format 1:** Source and Register

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **OP** | **RD** | **I** | **RS** | **UNUSED** |

15:11 10:9 8 7:6 5:0

**Format 2:** Immediate address

|  |  |  |  |
| --- | --- | --- | --- |
| **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

RS (bits 6 and 7) stands for register-source

When the immediate bit (I) is set to 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.

**1.3 Virtual Machine Instruction Set**

To avoid writing on machine language we write in assembly language. To simplify the VM we need the assembler to convert the assembly language to machine language.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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; 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; if r[RD] == CONST set EQUAL reset LESS and GREATER; 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 |  |

*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 to any patter, but for redundancy we set don’t care to all zeros.

Since *mem* consists of a set of integers (bits), any program written in the 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 and object code. The sequence of object codes is called an object program. The assembler encounters the following.

*loadi 2 71*

it translates the instruction to

0000010101000111

Where from left to right 000002 is the opcode for *load* or *loadi*

102 represents r[2]

12 represents immediate addressing (I == 1) and therefore *loadi* is the opcode and

010001112 is CONST 7110.

1351 is the produced object code for the instruction, since 00000101010001112 = 135110.

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.

**1.4 Instruction Fetch-Execute Cycle**

To encompass the Virtual Machine a C++ class is implemented (*Virtual Machine)* to interpret the object programs. The program stores the object program to be run at the top of memory, this implies setting *pc* and *base* register to 0 and *limit* register to the size of object program. VM enter the 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 |

**Mem**

1. Start Stack Pointer.
2. Start mem loc.

|  |
| --- |
|  |
|  |
|  |
|  |
|  |

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 = 111111002 = -410, then to store it in some r[RD] register, it must be sign extended to 11111111111111002 (still -410). 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 is 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;*

**1.5 Call and Return Instructions**

*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.

**1.6 Run and Compile**

Since, the program is meant to be modular and being able to expand in the future we defined the Assembler and Virtual Machine class in files.

Class Assembler is defined as:

*Assembler.h*

*Assembler.cpp*

Class Virtual Machine is defined as:

*VirtualMachine.h*

*VirtualMachine.cpp*

Compile *Assembler.cpp* and *VitualMachine.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* include:

**

Compile and link to make the rudimentary OS (rudimentary only on this section!)

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

and run *prog.s* in your environment:

*$ os prog.s*

Which assembles *prog.s* into *prog.o*, loads *prog.o* into memory, and finally invokes VM to run the program.

**1.7 Test Programs**

Some test program to run on the OS.



prog.s program

This program shows how the structure of the assembler works and how the instructions are formulated and translated to machine code.



fact.s program

This is a program that test most if not all the instructions on the OS and every call creates a stack that grows downward from mem[256] to limit location. The program object code starts from mem[0] to limit. Where limit is the size of the program.

**2 Process Management**

**2.1 To be continued…**