Simple Stack Machine Manual
($Revision: 1.62 $)

Gary T. Leavens
Leavens@ucf.edu

August 30, 2024

Abstract

This document defines the machine code of the Simple Stack Machine VM for use in the Systems Software class (COP 3402) at UCF.

1 Overview

The Simple Stack Machine (SSM) is a virtual machine (VM) that is a hybrid of Montagne's VM/0 [3, Chapter 2] and the MIPS processor's ISA [2]. In particular, the SSM is a word-addressed stack machine, with 32-bit (4-byte) words. All instructions are also one word (32-bits) wide and there is no floating-point support, kernel mode support, or other features that would support an OS.

2 VM Operations

The VM has two basic modes of operation: execution of a program and (just) printing it.

The VM is passed a single file name on its command line as an argument; it may also be passed the option `-p` as a command line argument. The file name given to the VM must be the name of a (readable) binary object file containing a program.

When given the `-p` option, followed by a binary object file name, the VM loads the binary object file and prints the assembly language form of that program, see Section 2.3 for details on the assembly language.

2.1 Binary Object Files

Programs are given to the VM in binary object files, which are files that contain the binary form of a program.

2.2 Executing Programs

When VM is only given a binary object file to execute (and no `-p` flag) on the command line, the VM will execute the program contained in the named file. For example, if the VM's executable is named `vm` and the program it should run is contained in the file named `test.bof` (and both these files are in the current directory), then the VM should execute the program in the file `test.bof` by executing the following command in the Unix shell.

```
./vm test.bof
```

The format of a binary object file (BOF) is given in the header file `bof.h`, which is shown, in part, in Figure 1. A BOF starts with the header, then the instructions (also in binary form) follow, followed by the initial (binary) values of data. This layout of binary object files is shown in Figure 2.

The header of a binary object file starts with a 4-character field that contains the (ASCII codes for) the characters "`BO32`" (i.e., in hexadecimal: 424f3332); this kind of "magic number" is commonly used to identify files in Unix. The magic number is followed (in the BOF header) by the starting address[1] of the program's code and the length of the program's code (in words), which constitutes the "text" section of the binary object file. These are followed by the starting address of the data section[2] and its length (in words). The data section contains the global/static variables that the program uses. Finally, the header contains the initial value for the stack and frame pointers (i.e., the $fp and $sp registers, which start at the same value), which is the address (in words) of the bottom of the runtime stack.

```
/* $Id: bof.h,v 1.22 2024/08/20 00:46:41 leavens Exp $ */
// Binary Object File Format (for the SSM)
#ifndef _BOF_H
#define _BOF_H
#include <stdbool.h>
#include <stdio.h>
#include <stdint.h>
#include "machine_types.h"

#define MAGIC_BUFFER_SIZE 4

typedef struct { // Field magic should hold value of MAGIC (with no null char)
    char      magic[MAGIC_BUFFER_SIZE];
    word_type text_start_address;  // word address to start running (PC)
    word_type text_length;         // size of the text section in words
    word_type data_start_address;  // word address of static data (GP)
    word_type data_length;         // size of data section in words
    word_type stack_bottom_addr;   // word address of stack "bottom" (FP)
} BOFHeader;

// a type for Binary Output Files
typedef struct {
    FILE *fileptr;
    const char *filename;
} BOFFILE;
```

Figure 1: The `bof.h` header file that defines the format and operations for binary object files.

2.2.1 Initial/Default Values

The memory of the machine starts at all zero (0) values. Then the instructions specified by the given binary object file (as named on the command line) are loaded into memory, starting at address 0, making the contents of the first N words of memory be the same as the N words following the header itself in the binary object file; here N is the same as the header's value of the `text_length` field. Following those N words are the words of the data section. These are loaded into the memory starting at the data start address

[1] The starting address of the code is the initial value of the program counter (*PC*).

[2] The starting address of the data section is the address of the first element of the data section and the initial value of the $gp register.

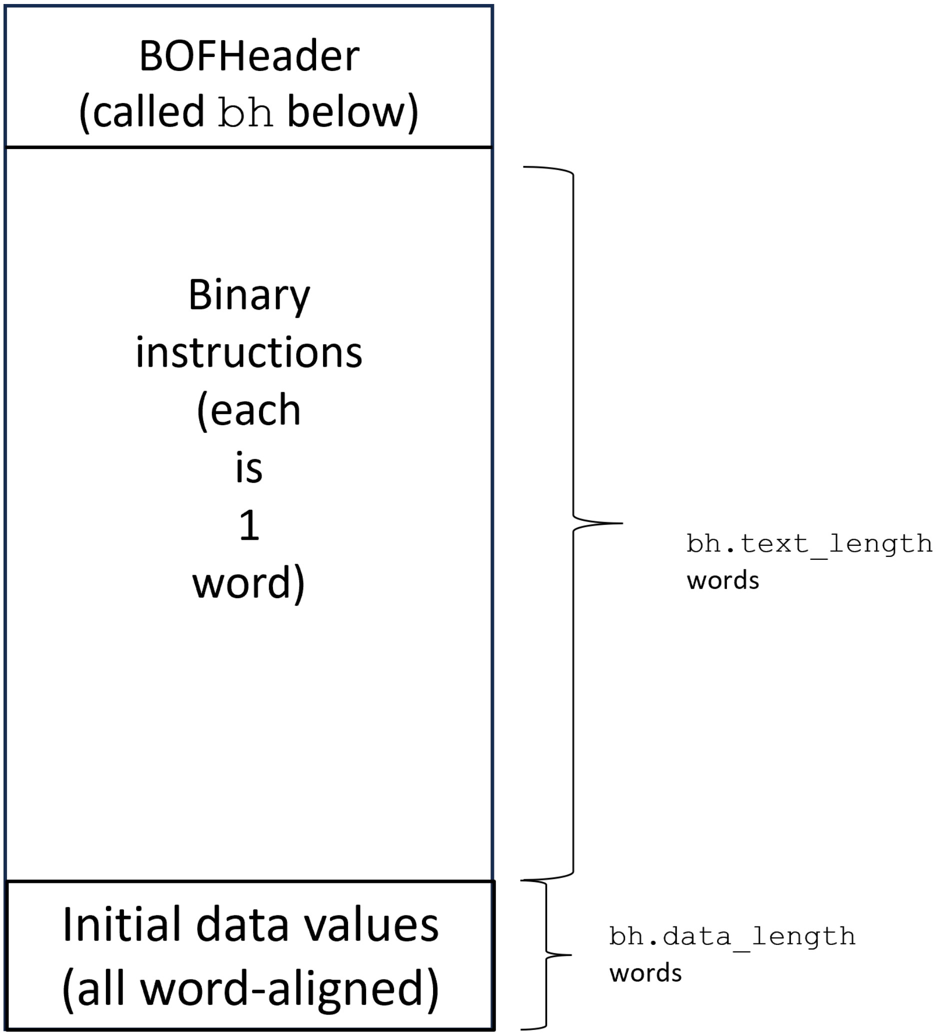
BOFHeader
(called bh below)
Binary instructions (each is 1 word)
bh.text_length words
Initial data values (all word-aligned)
bh.data_length words

Figure 2: The layout of a binary object file.

given in the header; thus any initial values are copied from the data section of the binary object file into VM's memory.

When the program starts executing:

- The register $gp is set to the (word) address of the start of the data section given in the header.
- The registers $fp and $sp are both is set to the stack bottom's address, which is given in the header. This address must be strictly greater than the start address of the data section.
- The program counter *PC* is set to the starting address of the text section, which is given in the header. This address must be strictly less than the data section's start address.

2.2.2 The Running Program's Input and Output

When the program executes instructions to read or write characters, these are read from standard input (`stdin`) and written to standard output (`stdout`).

However, note that if you want the program to read a character, typing a single character (say `x`) into the terminal (i.e., to the shell) while the program is running will not send that character immediately to the program, as standard input (on Unix) is buffered by default. Thus, in the Unix shell, to send characters to the program it is best to use a pipe or file redirection; for example, to send the two characters `x` and `y` (followed by a newline character) to the VM running the program `progfile.bof` one could use the following command at the Unix shell:

```
echo xy | ./vm progfile.bof
```

Another Unix command that would accomplish the same thing would be to put the characters to be input into a file (using a text editor), say `xy-input.txt`, and then to use the following Unix command.

```
./vm progfile.bof < xy-input.txt
```

2.2.3 Tracing Output

By default, the VM produces output on `stdout` that traces the VM's execution. This tracing output can be turned off by executing a `NOTR` system call instruction and can be turned on by executing a `STRA` system call instruction.[3] (See Section A.5 for more information about these system call instructions.)

The tracing output shows the initial state of the VM, then for each instruction executed, it shows the word address of that instruction and then the assembly language form of that instruction.

The state of the machine shown in tracing output shows: the values in the PC, the (signed) values of the HI and LO registers (if those are not zero), the (signed) values in all the general purpose registers, then the memory starting at the address in GPR[$gp] with locations containing zeros only indicated by "`...`", and then the data in the current activation record (i.e., the memory between the addresses GPR[$sp] and GPR[$fp] inclusive). When showing the memory, repeated locations containing zeros are only indicated with "`...`". Note that when printing, all numbers as printed as signed decimal **int**s, so the C print routines will sign-extend them (and thus 0x0000ffff will print as -1). For example, when the binary object file that is assembled from the assembly code shown in Figure 3 is executed, which is found in the file `vm_test0.bof`, it produces the output shown in Figure 4.

[3] The `STRA` instruction has no effect if the VM is already producing tracing output. Similarly, the NOTR instruction has no effect if the VM is not already producing tracing output.

```
        # $Id: vm_test0.asm,v 1.3 2024/07/26 12:44:46 leavens Exp $
        .text start
start:  STRA
        ADDI $sp, -1, 2
        EXIT 0
        .data 1024
        .stack 4096
        .end
```

Figure 3: The SSM assembly language file `vm_test0.asm`.

```
      PC: 0
GPR[$gp]: 1024  GPR[$sp]: 4096  GPR[$fp]: 4096  GPR[$r3]: 0     GPR[$r4]: 0
GPR[$r5]: 0     GPR[$r6]: 0     GPR[$ra]: 0
    1024: 0             ...
    4096: 0

==>      0: STRA
      PC: 1
GPR[$gp]: 1024  GPR[$sp]: 4096  GPR[$fp]: 4096  GPR[$r3]: 0     GPR[$r4]: 0
GPR[$r5]: 0     GPR[$r6]: 0     GPR[$ra]: 0
    1024: 0             ...
    4096: 0

==>      1: ADDI $sp, -1, 2
      PC: 2
GPR[$gp]: 1024  GPR[$sp]: 4096  GPR[$fp]: 4096  GPR[$r3]: 0     GPR[$r4]: 0
GPR[$r5]: 0     GPR[$r6]: 0     GPR[$ra]: 0
    1024: 0             ...         4095: 2
    4096: 0

==>      2: EXIT 0
```

Figure 4: The output of running the VM on file `vm_test0.bof` (which is the result of using the assembler on `vm_test0.asm`), with the command `./vm vm_test0.bof`, as would be printed on standard output.

2.2.4 Error Outputs

All error messages (e.g., for division by zero) are sent to standard error output (`stderr`).

2.2.5 Exit Code

When the machine halts normally, it exits with a zero error code (which indicates success on Unix). However, when the machine encounters an error it halts and exits with a non-zero exit code (which indicates failure on Unix).

2.3 Printing the Program

When the VM is given the argument `-p` option followed by a binary object file name, it first loads the instructions and data from the given binary object file, then it prints what was loaded in assembly language format, and then the VM exits *without* running the program. This printing of the program can be helpful for understanding what a compiler is producing. An example of this output is shown in Figure 5.

```
Address Instruction
     0: STRA
     1: ADDI $sp, -1, 2
     2: EXIT 0
    1024: 0             ...
```

Figure 5: The output of running `./vm -p vm_test0.bof`. (where the file `vm_test0.bof` is the result of assembling the file `vm_test0.asm` that is shown in Figure 3).

3 Architecture

The SSM is a stack machine that is word-addressed. (The SSM's words are 32 bits; i.e., 4 bytes.) The instructions sometimes interpret these words as integers (in arithmetic instructions), as bits (in logical instructions), or as (word) addresses (in other instructions such as jump instructions).

3.1 Registers

In a computer, a register is a piece of hardware that can store and retrieve data very quickly; typically a register holds a word of data.

In a VM, what we call a *register* is simply an important variable that simulates a hardware register. In a VM program, these registers could be implemented as variables or as an array.

3.1.1 General Purpose Registers and Their Names

The SSM has 8 general purpose registers, numbered 0 to 7 (inclusive). These are all 32-bit registers. Since there are 8 registers, instructions use 3 bits to specify them.

Conventions (partly from the MIPS architecture [2]) are followed for these registers and their names, as shown in Table 1. The names shown in Table 1 are used in this document and in the SSM's assembly language, and thus in printing instructions.

There are also some special purpose registers described next.

Table 1: SSM Register Numbers, Use, and Names

Number	Use	Name
0	globals pointer	$gp
1	stack pointer	$sp
2	frame pointer	$fp
3 – 6	registers (for other uses)	$r3 – $r6
7	return address	$ra

3.1.2 Special Purpose Registers

SSM also has a few special registers. The registers are named:

- *PC*, the program counter which holds the (word) address of the next instruction to execute,
- *HI*, the high part (i.e., most significant bits) of the result of a multiplication or the remainder in a division,
- *LO*, the low part (i.e., least significant bits) of the result of a multiplication or the quotient in a division.

The *PC* register is manipulated by various branch and jump instructions, as well as the CALL, CSI, and RTN instructions. The *HI* and *LO* registers are written by instructions that perform multiplication and division, and are read by instructions that move their contents into memory.

3.1.3 Calling Convention

The calling convention on the SSM is designed to be as simple as possible, but also efficient.

Saving and restoring registers is the caller's responsibility. That is, the caller must save (somewhere in memory) all registers that will be needed after a call, and in such cases the caller must restore the saved values back into those registers.

Note that the Call (`CALL`) and Call Subroutine Indirect (`CSI`) instructions do not save any registers except the *PC*; they both save the *PC* by putting it into register 7 ($ra). Similarly, the return (`RTN`) instruction does not restore any registers, it only moves the saved address from the $ra register back into the *PC*.

Arguments and results are passed on the runtime stack. For example, if a function f takes 3 arguments, say a_1, a_2, and a_3, then the values of those arguments (in a call-by-value language) are pushed on the stack in their order of evaluation (e.g., a_1 first, then a_2, then a_3) so that when f's code starts executing, it finds the last argument (the value of a_3) in memory[GPR[$sp]], the next-to-last argument's (i.e., a_2's) value in memory[GPR[$sp] + 1], and the first argument's (i.e., a_1's) value in memory[GPR[$sp] + 2]. If f computes a result, then f's code will allocate a word on top of the stack and will ensure that the result's value is on top of the runtime stack (i.e., in memory[GPR[$sp]]) when it returns.

3.2 Binary Instruction Format

In object code, all instructions are one word (32-bits) long and start with a 4-bit opcode. However, instructions may have one of several formats, with the format depending on the opcode (called "op" below) and the function field (called "func" below). The fields of each instruction format shown in Table 2 are followed by their width in bits; for example the func field is 4 bits wide.

The list of instructions and details on their execution appears in Appendix A.

Table 2: SSM Instruction Formats

- Computation type instruction format, with opcode 0:

op:4	rt:3	ot:9	rs:3	os:9	func:4

- Other Computation type instruction format, with opcode 1:

op:4	reg:3	offset:9	arg:12	func:4

- System call instructions, whose format is a variant of the other computational type instruction format, with opcode 1 and a func field of 15:

op:4	reg:3	offset:9	code:12	func:4

- Immediate operand type instruction format, with a register and an immediate value:

op:4	reg:3	offset:9	immed:16

- Jump type instruction format:

op:4	addr:28

3.3 Machine Cycles

The SSM instruction cycle conceptually does the following for each instruction:

1. Let *IR* be the instruction at the (word) address that *PC* indicates. (Note that *IR* could be considered to be the contents of a hardware register.)

2. The *PC* is made to point to the next instruction (i.e., it is set to the (word) address $PC + 1$).

3. Then the instruction in *IR* is executed. The op component of this instruction (*IR*.op) indicates the operation to be executed. For example, if *IR*.op encodes the instruction JMP, then the machine jumps to the specified address in the instruction, by setting the *PC* register to that address.

Note that the execution of an instruction (step 3) happens *after* the *PC* has already been made to point to the next instruction (in step 2).

3.4 Size Limits

The following constants, defined in the provided file `machine.h`, define the size of the memory for the VM.

```
// a size for the memory (2^16 = 32K words)
#define MEMORY_SIZE_IN_WORDS 32768
```

3.5 Invariants

The VM enforces the following invariants and must halt with an error message (written to stderr) if one of them is violated:

- $0 \leq$ GPR[$gp],

- GPR[$gp] $<$ GPR[$sp],

- GPR[$sp] $\leq$ GPR[$fp],
- GPR[$fp] $<$ `MEMORY_SIZE_IN_WORDS`,
- $0 \leq PC$, and
- $PC <$ `MEMORY_SIZE_IN_WORDS`.

A Appendix A

In the following tables, italicized names (such as r, t, and s) are meta-variables that refer to integers. If an instruction's field is notated as -, then its value does not matter (and for such fields, we use 0 as a placeholder for values in examples).

All numbers appearing in the following tables are in decimal (base 10) notation.

A.1 Computational Format Instructions

The computational instructions with an opcode of 0, which are found in Table 3, have their function specified by the func field. In the explanations, the contents of the general purpose register r is notated as GPR[r]. All numbers in the table are in decimal notation.

All arithmetic and logical operations are performed as for C `int` (integer) values. (That is, the SSM uses two's complement 32-bit arithmetic.)

The operation "formOffset" does a sign extension. For example if o is -2, (i.e., 1FE in hexadecimal), then formOffset(o) is also -2.

The notation memory[e] represents the signed word which is found in the VM's memory at the word address e Thus, the expression memory[GPR[s] + formOffset(o)] denotes the signed word of the VM's memory at the (word) address GPR[s] + formOffset(o), which is the sum of the contents of the general purpose register s and the offset formOffset(o). The notation umemory[e] represents an unsigned view of the VM's memory at the word address e; this view is used for bit manipulation instructions, logical shifts, and the JMP instruction.

A.2 Other Computational Instructions

The other computational instructions, as shown in Table 4, all have an op code of 1 and a specific function specified in the function field.

Note that the right shift works on the contents of the register GPR[t] in a logical manner, as if it were an **unsigned short**, so it should shift in zeros from the left.

Also note that the jump-relative (JREL) instruction jumps relative to the instruction's own address, which by the time it executes is $PC - 1$. (The idea is to help a person writing assembly language or a compiler understand the target address for a jump more easily.)

Table 3: Computational Format Instructions with opcode 0

Name	op	t	offset	s	offset	func	Effect
NOP	0	-	-	-	-	0	Does nothing (no-operation)
ADD	0	t	ot	s	os	1	Add: memory[GPR[t] $+$ formOffset(ot)] $\leftarrow$ memory[GPR[$sp]] $+$ (memory[GPR[s] $+$ formOffset(os)])
SUB	0	t	ot	s	os	2	Subtract: memory[GPR[t] $+$ formOffset(ot)] $\leftarrow$ memory[GPR[$sp]] $-$ (memory[GPR[s] $+$ formOffset(os)])
CPW	0	t	ot	s	os	3	Copy Word: memory[GPR[t] $+$ formOffset(ot)] $\leftarrow$ memory[GPR[s] $+$ formOffset(os)]
AND	0	t	ot	s	os	5	Bitwise And: umemory[GPR[t] $+$ formOffset(ot)]; $\leftarrow$ umemory[GPR[$sp]] $\wedge$ (umemory[GPR[s] $+$ formOffset(os)]);
BOR	0	t	ot	s	os	6	Bitwise Or: umemory[GPR[t] $+$ formOffset(ot)]; $\leftarrow$ umemory[GPR[$sp]] $\vee$ (umemory[GPR[s] $+$ formOffset(os)]);
NOR	0	t	ot	s	os	7	Bitwise Not-Or: umemory[GPR[t] $+$ formOffset(ot)]; $\leftarrow$ $\neg$(umemory[GPR[$sp]] $\vee$ (umemory[GPR[s] $+$ formOffset(os)]));

XOR	0	t	ot	s	os	8	Bitwise Exclusive-Or:
							umemory[GPR[t] $+$ formOffset(ot)];
							$\leftarrow$ umemory[GPR[$sp]]
							xor (umemory[GPR[s] $+$ formOffset(os)]);

LWR	0	t	-	s	o	9	Load Word into Register: GPR[t] $\leftarrow$ memory[GPR[s] $+$ formOffset(os)]
SWR	0	t	ot	s	-	10	Store Word from Register: memory[GPR[t] $+$ formOffset(ot)] $\leftarrow$ GPR[s]
SCA	0	t	ot	s	os	11	Store Computed address: memory[GPR[t] $+$ formOffset(ot)] $\leftarrow$ (GPR[s] $+$ formOffset(os))
LWI	0	t	ot	s	os	12	Load Word Indirect: memory[GPR[t] $+$ formOffset(ot)] $\leftarrow$ memory[memory[GPR[s] $+$ formOffset(os)]]
NEG	0	t	ot	s	os	13	Negate: memory[GPR[t] $+$ formOffset(ot)] $\leftarrow$ $-$memory[GPR[s] $+$ formOffset(os)]

Table 4: Other Computational Instructions, with opcode 1

Name	op	reg	offset	arg	func	Effect
LIT	1	t	o	i	1	Literal (load): $\text{memory}[\text{GPR}[t] + \text{formOffset}(o)] \leftarrow \text{sgnExt}(i)$
ARI	1	r	-	i	2	Add register immediate: $\text{GPR}[r] \leftarrow (\text{GPR}[r] + \text{sgnExt}(i))$
SRI	1	r	-	i	3	Subtract register immediate: $\text{GPR}[r] \leftarrow (\text{GPR}[r]-\text{sgnExt}(i))$
MUL	1	s	o	-	4	Multiply stack top by $\text{memory}[\text{GPR}[s] + \text{formOffset}(o)]$, putting the most significant bits in *HI* and the least significant bits in *LO*. $(\mathit{HI}, \mathit{LO})$ $\leftarrow \text{memory}[\text{GPR}[\mathsf{\$sp}]]$ $\times\ (\text{memory}[\text{GPR}[s] + \text{formOffset}(o)])$
DIV	1	s	o	-	5	Divide (remainder in *HI*, quotient in *LO*): $\mathit{HI} \leftarrow \text{memory}[\text{GPR}[\mathsf{\$sp}]]$ $\texttt{\%}\ (\text{memory}[\text{GPR}[s] + \text{formOffset}(o)]);$ $\mathit{LO} \leftarrow \text{memory}[\text{GPR}[\mathsf{\$sp}]]$ $/\ (\text{memory}[\text{GPR}[s] + \text{formOffset}(o)])$
CFHI	1	t	o	-	6	Copy from HI: $\text{memory}[\text{GPR}[t] + \text{formOffset}(o)] \leftarrow \mathit{HI}$
CFLO	1	t	o	-	7	Copy from LO: $\text{memory}[\text{GPR}[t] + \text{formOffset}(o)] \leftarrow \mathit{LO}$
SLL	1	t	o	h	8	Shift Left Logical: $\text{umemory}[\text{GPR}[t] + \text{formOffset}(o)]$ $\leftarrow \text{umemory}[\text{GPR}[\mathsf{\$sp}]]$ « h
SRL	1	t	o	h	9	Shift Right Logical: $\text{umemory}[\text{GPR}[t] + \text{formOffset}(o)]$ $\leftarrow \text{umemory}[\text{GPR}[\mathsf{\$sp}]]$ » h
JMP	1	s	o	-	10	Jump: $\mathit{PC} \leftarrow \text{umemory}[\text{GPR}[s] + \text{formOffset}(o)];$
CSI	1	s	o	-	11	Call Subroutine Indirectly: $\text{GPR}[\mathsf{\$ra}] \leftarrow \mathit{PC};$ $\mathit{PC} \leftarrow \text{memory}[\text{GPR}[s] + \text{formOffset}(o)]$
JREL	1	-	-	o	12	Jump Relative to address: $\mathit{PC} \leftarrow ((\mathit{PC}-1)+\text{formOffset}(o));$
	1				15	System Call: (see Table 7)

A.3 Immediate Type Instructions

The instructions in Table 5, all have an immediate operand, which is a 16-bit value.

For arithmetic operations, the immediate value is sign-extended (to an **int** value), using the function "sgnExt."

However, for logical operations, the immediate value is zero-extended, which is written in the explanations using the function "zeroExt." For example, suppose i is -2, which again as a 16-bit hex number is FFFE; then zeroExt(i) is 0000FFFE in hexadecimal notation.

For the branches, the immediate value, i is interpreted as a (word) offset from the instruction's (word) address; thus before being added to the (word) address $PC - 1$, it is sign extended, which is written as "formOffset" in the table. Note that the resulting offset is added to the (word) address of the instruction itself; this is why $PC - 1$ is used, since the PC has already been advanced by 1, when the instruction is being executed.

A.4 Jump Format Instructions

The instructions in Table 6 have a 28-bit field "addr" which is used to form the address to jump to. Forming this address is done by concatenating the (4) high bits of the instruction's (word) address with those bits to form a 32-bit (word) address. (This would allow hardware implementations to have a memory of size 2^{32} words.) This is written "formAddress$(PC - 1, a,)$" in the table. For example if a is AFEED3E in hexadecimal notation, and $PC - 1$ is F0000000 in hexadecimal notation, then formAddress$(PC - 1, a)$ is FAFEED3E in hexadecimal notation.

The CALL instruction and the CSI instruction do subroutine calls. As noted above, these do not save any registers, nor does the RTN instruction restore any registers. Saving registers (including $ra) before a call is the responsibility of the programmer or compiler. Restoring any saved registers after the return is the responsibility of the programmer or compiler.

A.5 System Calls

System calls are used to provide operating system services. System calls are made by instructions with op 1 and func 15 (neither of which is shown in Table 7). The code field is used to specify the (OS) service requested.

System calls include exiting a program and various kinds of printing and reading of characters. These are described in Table 7, using C library equivalents. In the table, an entry of "-" means that the contents of the argument is not specified (we use 0 for concrete examples in such cases). All printing done by these instructions is to the VM's standard output (stdout) and reading is from the VM's standard input (stdin). Strings use the null termination convention as in C, but double-byte characters are not supported; that is strings are null-terminated sequences of (single byte) characters.

The offset part of the `EXIT` instruction is used in that system call to specify the exit code of the VM as an immediate value, in other cases the offset is used to form an address.

In the PSTR instruction, the C standard library function `printf` will expect a C pointer to characters as its argument; this should be the address of those character's representations in the VM program's memory starting at the VM's address given by the contents of the memory at the location pointed to by $fp. (Since the VM's memory is word-aligned, a cast will be needed in an implementation in C to avoid a warning.)

Table 5: Immediate format instructions:

Name	op	reg	offset	immed	Effect
ADDI	2	r	o	i	Add immediate: $\text{memory}[\text{GPR}[r] + \text{formOffset}(o)]$ $\leftarrow (\text{memory}[\text{GPR}[r] + \text{formOffset}(o)]) + \text{sgnExt}(i)$
ANDI	3	r	o	i	Bitwise And immediate: $\text{umemory}[\text{GPR}[r] + \text{formOffset}(o)]$ $\leftarrow (\text{umemory}[\text{GPR}[r] + \text{formOffset}(o)]) \wedge \text{zeroExt}(i)$
BORI	4	r	o	i	Bitwise Or immediate: $\text{umemory}[\text{GPR}[r] + \text{formOffset}(o)]$ $\leftarrow (\text{umemory}[\text{GPR}[r] + \text{formOffset}(o)]) \vee \text{zeroExt}(i)$
NORI	5	r	o	i	Bitwise Not-Or immediate: $\text{umemory}[\text{GPR}[r] + \text{formOffset}(o)]$ $\leftarrow \neg(\text{umemory}[\text{GPR}[r] + \text{formOffset}(o)]) \vee \text{zeroExt}(i))$
XORI	6	r	o	i	Bitwise Exclusive-Or immediate: $\text{umemory}[\text{GPR}[r] + \text{formOffset}(o)]$ $\leftarrow (\text{umemory}[\text{GPR}[r] + \text{formOffset}(o)])$ xor $\text{zeroExt}(i)$
BEQ	7	r	o	i	Branch on Equal: **if** $\text{memory}[\text{GPR}[\texttt{\$sp}]] = \text{memory}[\text{GPR}[r] + \text{formOffset}(o)]$ **then** $PC \leftarrow (PC - 1) + \text{formOffset}(i)$
BGEZ	8	r	o	i	Branch ≥ 0: **if** $\text{memory}[\text{GPR}[r] + \text{formOffset}(o)] \geq 0$ **then** $PC \leftarrow (PC - 1) + \text{formOffset}(i)$
BGTZ	9	r	o	i	Branch > 0: **if** $\text{memory}[\text{GPR}[r] + \text{formOffset}(o)] > 0$ **then** $PC \leftarrow (PC - 1) + \text{formOffset}(i)$
BLEZ	10	r	o	i	Branch ≤ 0: **if** $\text{memory}[\text{GPR}[r] + \text{formOffset}(o)] \leq 0$ **then** $PC \leftarrow (PC - 1) + \text{formOffset}(i)$
BLTZ	11	r	o	i	Branch < 0: **if** $\text{memory}[\text{GPR}[r] + \text{formOffset}(o)] < 0$ **then** $PC \leftarrow (PC - 1) + \text{formOffset}(i)$
BNE	12	r	o	i	Branch Not Equal: **if** $\text{memory}[\text{GPR}[\texttt{\$sp}]] \neq \text{memory}[\text{GPR}[r] + \text{formOffset}(o)]$ **then** $PC \leftarrow (PC - 1) + \text{formOffset}(i)$

Table 6: Jump Format Instructions

			Comment
Name	op	addr	(Explanation)
JMPA	13	a	Jump to given address: $PC \leftarrow \text{formAddress}(PC - 1, a)$
CALL	14	a	Call subroutine: $\text{GPR}[\texttt{\$ra}] \leftarrow PC; PC \leftarrow \text{formAddress}(PC - 1, a)$
RTN	15	-	Return from subroutine: $PC \leftarrow \text{GPR}[\texttt{\$ra}]$

Table 7: System Calls

name	reg.	offset	code	Effect (in terms of C std. library)
EXIT	-	o	1	exit(sgnExt(o)) // halt with the given exit code
PSTR	s	o	2	$\text{memory}[\text{GPR}[\$\text{sp}]]$ $\leftarrow$ printf("%s",&memory[GPR[s] $+$ formOffset(o)])
PCH	s	o	4	$\text{memory}[\text{GPR}[\$\text{sp}]]$ $\leftarrow$ fputc(memory[GPR[s] $+$ formOffset(o)], stdout)
RCH	t	o	5	memory[GPR[t] $+$ formOffset(o)] $\leftarrow$ getc(stdin)
STRA	-	-	2046	start VM tracing output
NOTR	-	-	2047	no VM tracing; stop the tracing output

B Appendix B: Hints

B.1 Overall Structure of the Code

To implement the SSM VM, the first thing to do is to decide on some data structures to represent the VM's state: especially the memory and registers. You may want to represent the VM's memory using a definition like that in Figure 6. With this definition, the VM's memory is represented as 3 views of the same array of data (you can also think of this as 3 arrays that share the same storage): memory.words, memory.uwords, and memory.instrs. For example, the word at address 9 can be accessed as:

```
#include "machine_types.h"
#include "instruction.h"
// a size for the memory (2^16 words = 32K words)
#define MEMORY_SIZE_IN_WORDS 32768

static union mem_u {
    word_type words[MEMORY_SIZE_IN_WORDS];
    uword_type uwords[MEMORY_SIZE_IN_WORDS];
    bin_instr_t instrs[MEMORY_SIZE_IN_WORDS];
} memory;
```

Figure 6: A possible way to represent the memory of the VM, which allows access to the same storage as signed or unsigned words or as binary instructions.

- a word_type (i.e., as a signed **int**), using memory.words[9], which corresponds to the notation memory[9] in this document,
- a uword_type (i.e., as an **unsigned int**), using memory.uwords[9], which corresponds to the notation umemory[9] in this document, or
- an instruction, using memory.instrs[9].

This union definition allows VM's code to decide what view it wants of the storage at each point in the implementation, and whatever is changed in that view is seen by all the other views. Using a union in this way avoids casting and bit manipulation. Note that the C compiler considers memory.words to have the type word_type[], memory.uwords to have the type uword_type[], and memory.instrs to have the type bin_instr_t[].

The registers can also be an array or several variables; each register should be the size of a word (32 bits). However, it does not matter much if the registers or signed or unsigned, as they are not assigned to during the execution of arithmetic instructions. Thus the type of each register can be either word_type or uword_type (both of these are declared in the header file machine_types.h). However, when assigning a value from memory to a register, be sure to use the same type's view of the memory, to avoid arithmetic conversions.

Once the representation for memory and the registers is settled, implement the loading process and get the -p option to work. It may be helpful to use instruction_assembly_form function from the instruction module when printing instructions. See the code in the disassembler (disasm.c in particular) for a model of how to use it.

To read from the binary object files (BOFs) you should use the functions in the bof module. In particular, use bof_read_open to open such a file, use bof_read_header to read and return the header from a BOF, then read each of the instructions using instruction_read (from the instruction module), and bof_read_word to read words in the data section of the BOF.

After getting the -p option to work, you need to implement the basic fetch-execute cycle for the VM (without the -p option). We suggest that you write a function that executes a single instruction and handles the tracing, which calls a function that just executes each instruction. The function that executes and traces an instruction can then be called in a loop.

To implement the function that executes a single instruction, have that function decide between the possible instructions (using the functions from the instruction module) and in each case carry out the effect of the instruction as described this "Simplified Stack Machine Manual", which is available on Webcourses in the Files section. You can see the provided code in the instruction_assembly_form function in the instruction module as an example of how to decide what an instruction is.

B.2 Writing Your Own Tests

It is often helpful to write your own tests to execute just one or two instructions during testing of the VM. To write your own tests you can use the provided assembler (the asm program), since the VM only takes binary object files as inputs.

When you write your own tests, be sure to include an EXIT instruction in your test to stop your program's execution!

The SSM assembler can be compiled using the provided Makefile and the following command.

```
make asm
```

We also provide documentation for the assembly language in the course Files folder on Webcourses; see the file named "ssm-asm.pdf".

B.3 Disassembler

We also provide a disassembler, that does something similar to running the virtual machine with the -p option; this program can be built using the Makefile by issuing the command make disasm. The way that the disassembler does its output, using the instruction module, can be helpful in writing the code for the -p option of the VM.

However, the disassembler is mostly useful when writing the assembler, so once you have the -p option of the VM implemented, you may not need it.

B.4 Provided Files

We provide all the source files used to build the assembler and the disassembler. Many of these can be helpful in writing your VM implementation. The following describes some of these.

B.4.1 Makefile

The provided `Makefile` describes how to compile and link programs and run tests. This `Makefile` tells the GNU program `make` [1] about dependencies between files that `make` uses to decide when targets need to be built.

You should edit the Makefile's definition of `VM_OBJECTS`; change that to be a complete list of the `.o` files that are needed to build your virtual machine. The list present in the Makefile for `VM_OBJECTS` is what the course staff used, but you might, for example, combine the main function, which was in our files `machine_main.c` with what we put in the file `machine.c` (neither file is provided to you) into a single file named `vm.c`, in which case you would replace our `machine_main.o` and `machine.o` with your `vm.o` in the definition of `VM_OBJECTS`. On the other hand, you will need to leave all the names of the object files that your VM uses in `VM_OBJECTS`, so that the linker can find them. For example, leave `bof.o`, `instruction.o`, and `utilities.o` in the list `VM_OBJECTS`.

If you receive an error message from the Unix linker/loader (`ld`) about an "undefined reference" to a function or a piece of data, then the solution is likely to include the relevant object file in the Makefile's list of `VM_OBJECTS`.

You should not need to edit anything in the bottom half of the Makefile, which is the "developer's section". That section should only be used by the course staff.

There are several useful targets in the Makefile that can be used with the `make` command. A *target* is a name given on the left side of a rule in the makefile, and can also be given on the command line to `make`; for example `vm` is a target, and running the command `make vm` should compile and link the code needed to build your VM and create an executable program in the file `vm` (or `vm.exe` on Windows). The following is a list of the targets in the Makefile that may be useful.

`file.o` compiles `file.c` if it (or `file.h`) is newer than `file.o`, producing a new copy of `file.o`. This works for any file name not just "`file`", as the Makefile has a general rule to compile `.c` files into `.o` files.

`vm` compiles (if necessary) all the .o files named in the macro `VM_OBJECTS` and links them together into an executable named `vm` (or `vm.exe` on Windows).

`vm_test1.myo` runs your virtual machine program (`./vm`) on the input `vm_test1.bof` and sends the (standard) output (and standard error output) to `vm_test1.myo`. This works for any test file, not just "`vm_test1.bof`" as the Makefile has a general rule for this.

`vm_test1.myp` runs your virtual machine program (`./vm`) with the `-p` option on the input `vm_test1.bof` and sends the (standard) output (and standard error output) to `vm_test1.myp`. This works for any test file, not just "`vm_test1.bof`" as the Makefile has a general rule for this.

`check-vm-outputs` runs all of the provided tests in the `.bof` files and produces the corresponding `.myo` files using your VM (in `./vm`), and compares each one to the expected output in the corresponding `.out` file using the `diff` command. Each such test passes if no differences are detected.

`check-lst-outputs` runs all of the provided tests in the `.bof` files and produces the corresponding `.myp` files using your VM (as `./vm -p`), and compares each one to the expected output in the corresponding `.lst` file using the `diff` command. Each such test passes if no differences are detected.

`check-outputs` runs all of the provided tracing and listing tests (using the targets `check-lst-outputs` and `check-vm-outputs`).

`submission.zip` runs all of the provided tests (using the `check-outputs` target) and creates a zip file that can be submitted for the assignment including your code and outputs from the tests.

`clean` removes all the compiled object files (`*.o`) and testing output files (`*.myo` and `*.myp`) as well as the executable VM (files named `vm` and `vm.exe`) and the submission zip file (`submission.zip`). This allows you to start over from scratch, forcing `make` to build the do the work specified for the targets given, instead of thinking that they are up to date. (This is especially useful if some dependencies are not captured in the Makefile.)

B.4.2 C Typedefs for SSM Machine Types

We provide a module, `machine_types` (the header `machine_types.h` is shown in Figure 7), which defines some C equivalents for important types of data in the SSM.

B.4.3 Other modules provided

The following gives a brief summary of the other provided code modules, each of which consists of a `.c` file and a `.h` file.

- `bof`, which describes binary object files,
- `file_location`, which groups file names and line numbers,
- `instruction`, which describes machine instructions and provides several useful utilities for creating and printing instructions,
- `regname`, which provides access to the symbolic names of the SSM's general purpose registers,
- `utilities`, which describes several functions for error output, including `bail_with_error`.

In addition, we provide code to build the assembler (including many of the above and also the files `asm_main.c`, `asm.y` (which is the context-free grammar in Bison), `asm_lexer.l` (which is the lexical grammar in Flex), `asm.tab.h` (the definitions that Bison generates), `asm_unparser.[ch]` (that is both the `.c` and `.h` files), `ast.[ch]`, `lexer.[ch]`, `pass1.[ch]`, `assemble.[ch]`, and the header file `id_attrs_assoc.h`) as well as the disassembler (including many mentioned above and also `disasm_main.c`, `disasm.[ch]`).

You can use any of these provided files in your solution.

References

[1] Free Software Foundation. *GNU Make Manual*, Feb 2023. `https://www.gnu.org/software/make/manual/`.

[2] Gerry Kane and Joe Heinrich. *MIPS RISC architectures*. Prentice-Hall, Inc., 1992.

[3] Euripides Montagne. *Systems Software: Essential Concepts*. Cognella Academic Publishing, 2021.

```
// $Id: machine_types.h,v 1.29 2024/08/29 21:58:39 leavens Exp $
// Machine Types for the Simplified Stack Machine (SSM)
#ifndef _MACHINE_TYPES_H
#define _MACHINE_TYPES_H

#define BYTES_PER_WORD 4

// operation codes
typedef unsigned short opcode_type;

// registers encoded in instructions
typedef unsigned short reg_num_type;

// offset values (which are signed) encoded in instructions
typedef short offset_type;

// function codes in computational instructions
typedef unsigned short func_type;

// argument values (signed) encoded in instructions
typedef short arg_type;

// shift values encoded in other computational instructions
typedef unsigned short shift_type;

// immediate operands encoded in instructions
typedef int immediate_type;

// unsigned immediate operands in instructions
typedef unsigned int uimmed_type;

// addresses, assuming ints are 32 bits
typedef unsigned int address_type;

// machine words, assuming ints are 32 bits
typedef int word_type;

// machine words, unsigned
typedef unsigned int uword_type;

// bytes, assuming chars are 8 bits
typedef unsigned char byte_type;
// ...
#endif
```

Figure 7: Macros and type definitions from the header file of the `machine_types` module, which provides some basic definitions for the VM.