Simplified RISC Machine Manual
($Revision: 1.62 $)

Gary T. Leavens
Leavens@ucf.edu

November 15, 2023

Abstract

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

1 Overview

The Simplified RISC Machine (SRM) processor's instruction set architecture (ISA) is simplified from the MIPS processor's ISA [2]. In particular, SRM is a little-endian machine with 32-bit (4-byte) words. All instructions are also 32-bits wide and there is no floating-point support, kernel mode support, or other advanced features.

1.1 Inputs and Outputs

1.1.1 Binary Object Files

The VM is passed a single file name on its command line as an argument. Optionally the VM be passed a single command line option, either `-p` or `-t`.

When given a `-p` argument followed by a binary object file name, the VM loads the binary object file and prints the assembly language form of the program, see Section 1.4 details.

When given a `-t` argument followed by a binary object file name, the VM runs the binary object file with tracing output enabled. By default, no tracing happens, but users can turn on tracing under program control by executing the `STRA` system call.

The remainder of this section is concerned with what the VM does when it only is given a binary object file name on its command line as an argument.

The file name given to the VM must be the name of a (readable) binary object file containing the program that the VM should execute. 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 the header, then the instructions (also in binary form) follow, followed by the initial 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 characters "`BOF`" (and a null character); 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 of the program's code and the length of the program's code (in bytes), which constitutes the "text" section of the binary object file. These are followed by the starting address of the data section and its length (in bytes). 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, which is the address (in bytes) of the bottom of the runtime stack.

```
/* $Id: bof.h,v 1.13 2023/11/13 18:26:52 leavens Exp $ */
// Binary Object File Format (for the SRM)
#ifndef _BOF_H
#define _BOF_H
#include <stdio.h>
#include <stdint.h>
#include "machine_types.h"

#define MAGIC_BUFFER_SIZE 4

typedef struct { // Field magic should be "BOF" (with the null char)
    char     magic[MAGIC_BUFFER_SIZE];
    address_type text_start_address;  // byte address to start running (PC)
    address_type text_length;         // size of the text section in bytes
    address_type data_start_address;  // byte address of static data (GP)
    address_type data_length;         // size of data section in bytes
    address_type stack_bottom_addr;   // byte address of stack "bottom" (FP)
} BOFHeader;

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

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

1.1.2 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 bytes (where N is divisible by 4) of memory be the same as the N bytes 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 bytes are the bytes of the data section. These are loaded into the memory starting at the data start address 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 start address of the data section given in the header, which must be divisible by 4,
- the registers $fp and $sp are both is set to the stack bottom address given in the header, which must be divisible by 4 and strictly greater than the start address of the data section,

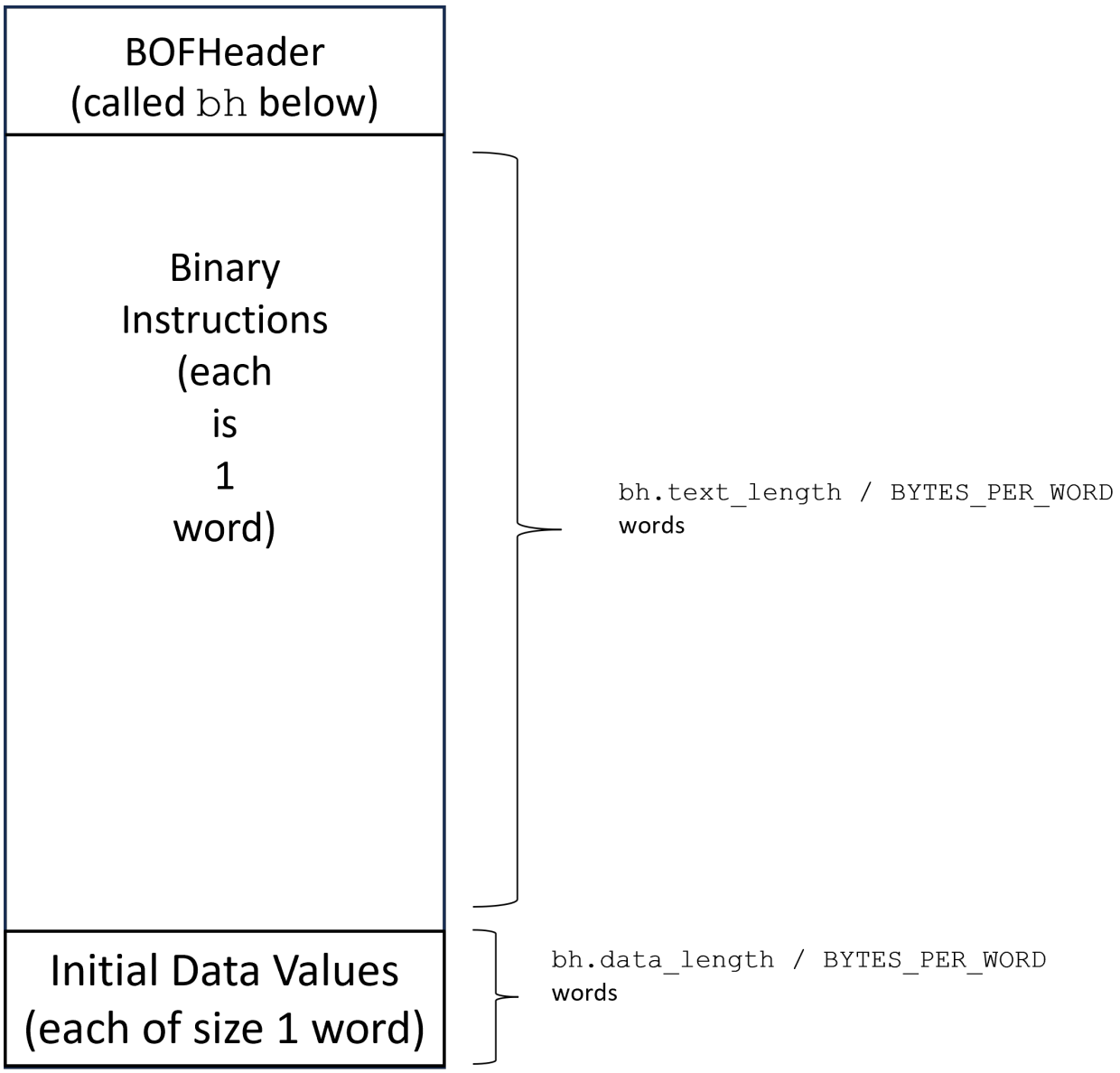

Figure 2: The layout of a binary object file.

- to simulate a call of the program (by the OS) the $a0 register is set to the same value as $fp, and
- the program counter *PC* is set to the text section's start address, which must be divisible by 4 and strictly less than the data section's start address.

1.2 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 is buffered by default. To send characters to the program it is best to use a pipe or file redirection in the Unix shell; 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 command that would accomplish the same thing is 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
```

1.3 Tracing Output

When used with the `-t` option or after the `STRA` system call is executed, the VM produces output that traces the its execution on `stdout`. (This tracing output can be turned off under program control by executing a `NOTR` system call instruction.) (See Section A.4 for more information about the `STRA` and `NOTR` system call instructions.)

The tracing output shows the initial state of the VM, then for each instruction executed, it shows the address (in bytes) 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 values of the HI and LO registers (if those are not zero) and the values in all the general purpose registers, then the memory starting at the address in GPR[$gp] with locations containing zeros only indicated with "`...`". and then the data between the addresses between GPR[$sp] and the stack bottom address specified in the binary object file. The idea is that those addresses should include the runtime stack. When showing the memory, locations containing zeros are only indicated with "`...`". 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.

1.4 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 can be helpful for understanding what a program is doing. This output is shown in Figure 5.

1.5 Error Outputs

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

```
        # $Id: vm_test0.asm,v 1.1 2023/09/18 03:32:18 leavens Exp $
        .text start
start:  STRA
        ADDI $0, $t0, 1
        EXIT
        .data 1024
        .stack 4096
        .end
```

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

```
      PC: 4
GPR[$0 ]: 0     GPR[$at]: 0     GPR[$v0]: 0     GPR[$v1]: 0     GPR[$a0]: 4096  GPR[$a1]: 0
GPR[$a2]: 0     GPR[$a3]: 0     GPR[$t0]: 0     GPR[$t1]: 0     GPR[$t2]: 0     GPR[$t3]: 0
GPR[$t4]: 0     GPR[$t5]: 0     GPR[$t6]: 0     GPR[$t7]: 0     GPR[$s0]: 0     GPR[$s1]: 0
GPR[$s2]: 0     GPR[$s3]: 0     GPR[$s4]: 0     GPR[$s5]: 0     GPR[$s6]: 0     GPR[$s7]: 0
GPR[$t8]: 0     GPR[$t9]: 0     GPR[$k0]: 0     GPR[$k1]: 0     GPR[$gp]: 1024  GPR[$sp]: 4096
GPR[$fp]: 4096  GPR[$ra]: 0
    1024: 0     ...
    4096: 0     ...
==> addr:    4 ADDI $0, $t0, 1
      PC: 8
GPR[$0 ]: 0     GPR[$at]: 0     GPR[$v0]: 0     GPR[$v1]: 0     GPR[$a0]: 4096  GPR[$a1]: 0
GPR[$a2]: 0     GPR[$a3]: 0     GPR[$t0]: 1     GPR[$t1]: 0     GPR[$t2]: 0     GPR[$t3]: 0
GPR[$t4]: 0     GPR[$t5]: 0     GPR[$t6]: 0     GPR[$t7]: 0     GPR[$s0]: 0     GPR[$s1]: 0
GPR[$s2]: 0     GPR[$s3]: 0     GPR[$s4]: 0     GPR[$s5]: 0     GPR[$s6]: 0     GPR[$s7]: 0
GPR[$t8]: 0     GPR[$t9]: 0     GPR[$k0]: 0     GPR[$k1]: 0     GPR[$gp]: 1024  GPR[$sp]: 4096
GPR[$fp]: 4096  GPR[$ra]: 0
    1024: 0     ...
    4096: 0     ...
==> addr:    8 EXIT
```

Figure 4: The tracing output of running the VM (with the `-t` option) on the 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.

```
Addr  Instruction
   0 STRA
   4 ADDI $0, $t0, 1
   8 EXIT
    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).

1.6 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 Architecture

In SRM, words are 32 bits (4 bytes). These bits can be interpreted as an integer or as an address. The machine is byte addressed but instructions must always be at an address that is on a word boundary (i.e., whose address is evenly divisible by 4).

2.1 Registers

2.1.1 General Purpose Registers and Their Names

The SRM is a register machine with 32 general purpose registers[1], numbered 0 to 31 (inclusive). These are all 32-bit registers. Since there are 32 registers, instructions use 5 bits to specify them.

Register 0 cannot be written to, and when read its value is always 0.

Conventions (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 conventional ones.

Table 1: SRM Register Numbers, Use, and Names

Number	Use	Name
0	always 0 (can't write to this register!)	
1	assembler temporary	$at
$2-3$	function results	$v0, $v1
$4-7$	function arguments	$a0–$a3
$8-15$	temporaries	$t0–$t7
$16-23$	temporaries	$s0–$s7
$24-25$	temporaries	$t8, $t9
$26-27$	reserved for use by OS (don't use!)	
28	globals pointer	$gp
29	stack pointer	$sp
30	frame pointer	$fp
31	return address	$ra

2.1.2 Special Purpose Registers

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

- *PC*, the program counter which holds the address of the next instruction to execute,

[1] What we call "registers" in this document are simply important concepts that simulate what would be registers in a hardware implementation of the virtual machine. For the VM program, these registers would be implemented as variables.

- *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 jump instructions. The *HI* and *LO* registers are read by instructions that move their contents into another register.

2.1.3 Calling Convention

The calling convention on the SRM follows the calling convention on the MIPS processor.

That is, the caller must save registers $1 - 15$, and $24 - 25$ if they will be needed after a call (and then restores them when needed).

The callee saves (and restores before it returns) registers $16 - 23$ and $29 - 31$, if it uses (writes) them.

(Furthermore, register 0 cannot be changed and registers 1 and 28 should not be changed by a hand-written routine. Registers $26 - 27$ should not be changed by user code.)

Note that the jump-and-link (`JAL`) instruction does not save any registers except the *PC*, and it will save that in register 31.

2.2 Binary Instruction Format

In object code, all instructions are one word long and start with a 6-bit opcode. However, instructions may have one of several formats, with the format depending on the opcode (called "op" below). The fields of each instruction format shown in Table 2 are followed by their width in bits; for example the op field is 6 bits wide.

Table 2: SRM Instruction Formats

- Register/computational type instruction format:

op:6	rs:5	rt:5	rd:5	shift:5	func:6

- System call instructions, whose format is a variant of the register type instruction format, but with a func field value of 12:

op:6	code:20	func:6

- Immediate operand type instruction format:

op:6	rs:5	rt:5	immed:16

- Jump type instruction format:

op:6	addr:26

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

2.3 Machine Cycles

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

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

2. The *PC* is made to point to the next instruction (i.e., it is set to the address $PC + 4$).

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 JR, then the machine jumps to the specified address by setting the *PC* register (to the contents of the given register).

2.4 Size Limits

The following constant defines the size of the memory for the VM.

```
#define MEMORY_SIZE_IN_BYTES (65536 - BYTES_PER_WORD)
```

You might need to copy this definition into your program.
Note that `BYTES_PER_WORD` is defined to be 4 in `machine_types.h`, see Figure 7.

2.5 Invariants

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

- *PC* % `BYTES_PER_WORD` $= 0$,
- GPR[$gp] % `BYTES_PER_WORD` $= 0$,
- GPR[$sp] % `BYTES_PER_WORD` $= 0$,
- GPR[$fp] % `BYTES_PER_WORD` $= 0$,
- $0 \leq$ GPR[$gp],
- GPR[$gp] $<$ GPR[$sp],
- GPR[$sp] $\leq$ GPR[$fp],
- GPR[$fp] $<$ `MEMORY_SIZE_IN_BYTES`,
- $0 \leq$ *PC*,
- *PC* $<$ `MEMORY_SIZE_IN_BYTES`, and
- GPR[0] $= 0$.

A Appendix A

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

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

A.1 Register/Computational Instructions

Note that all of the instructions in table Table 3 have an opcode of 0, and a function specified by the func field. They each also have 3 register arguments: rs, rt, and rd. The contents of the general purpose register r is notated as GPR[r] in the table. All numbers in the table are in decimal notation.

All arithmetic and logical operations are performed as for C **int** values. However, the right shift works on the contents of the register GPR[t] in a logical manner, as if it were an **unsigned int**, so it should shift in zeros from the left.

Table 3: Register Format Instructions

Name	op	rs	rt	rd	shift	func	Comment (Explanation)
ADD	0	s	t	d	-	33	Add: GPR[d] $\leftarrow$ GPR[s] + GPR[t]
SUB	0	s	t	d	-	35	Subtract: GPR[d] $\leftarrow$ GPR[s] $-$ GPR[t]
MUL	0	s	t	-	-	25	Multiply: Multiply GPR[s] and GPR[t], putting the least significant bits in *LO* and the most significant bits in *HI*. (HI, LO) $\leftarrow$ GPR[s] $\times$ GPR[t]
DIV	0	s	t	-	-	27	Divide (remainder in *HI*, quotient in *LO*): *HI* $\leftarrow$ GPR[s] % GPR[t] *LO* $\leftarrow$ GPR[s]/GPR[t])
MFHI	0	-	-	d	-	16	Move from HI: GPR[d] $\leftarrow$ *HI*
MFLO	0	-	-	d	-	18	Move from LO: GPR[d] $\leftarrow$ *LO*
AND	0	s	t	d	-	36	Bitwise And: GPR[d] $\leftarrow$ GPR[s] $\wedge$ GPR[t]
BOR	0	s	t	d	-	37	Bitwise Or: GPR[d] $\leftarrow$ GPR[s] $\vee$ GPR[t]
NOR	0	s	t	d	-	39	Bitwise Not-Or: GPR[d] $\leftarrow \neg$(GPR[s] $\vee$ GPR[t])
XOR	0	s	t	d	-	38	Bitwise Exclusive-Or: GPR[d] $\leftarrow$ GPR[s] xor GPR[t]
SLL	0	-	t	d	h	0	Shift Left Logical: GPR[d] $\leftarrow$ GPR[t] « h
SRL	0	-	t	d	h	3	Shift Right Logical: GPR[d] $\leftarrow$ GPR[t] » h
JR	0	s	0	0	0	8	Jump Register: *PC* $\leftarrow$ GPR[s]
SYSCALL	0	-	-	-	-	12	System Call: (see Table 6)

A.2 Immediate Type Instructions

The instructions in Table 4 may have up to 2 register arguments, and all have an immediate operand, which is a 16 bit value.

For arithmetic operations, the immediate value is sign-extended (to an **int** value), which is written in the explanations using the function "sgnExt." For example, suppose i is -1, which is FFFF in hexadecimal notation; then sgnExt(i) is FFFFFFFF in hexadecimal, which still represents -1.

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 -1, which is FFFF in hexadecimal notation; then zeroExt(i) is 0000FFFF in hexadecimal notation.

For the branches, the immediate value, o is first shifted left 2 bits (multiplied by 4) and then sign-extended, which is written as "formOffset" in the table. (Thus formOffset(o) = sgnExt$(4 \times o)$.) Note that the resulting address is added to the address of the instruction following the currently executing instruction, not the address of the instruction itself, since the *PC* has already been advanced. (As a simplification, the opcode for the BLTZ instruction is different from that found in the MIPS architecture [2].)

For loads and stores, memory$[a]$ denotes the contents of the machine's memory at the byte address a.

Table 4: Immediate format instructions:

Name	op	rs	rt	immed	Comment (Explanation)
ADDI	9	s	t	i	Add immediate: GPR[t] $\leftarrow$ GPR[s] + sgnExt(i)
ANDI	12	s	t	i	Bitwise And immediate: GPR[t] $\leftarrow$ GPR[s] $\wedge$ zeroExt(i)
BORI	13	s	t	i	Bitwise Or immediate: GPR[t] $\leftarrow$ GPR[s] $\vee$ zeroExt(i)
XORI	14	s	t	i	Bitwise Xor immediate: GPR[t] $\leftarrow$ GPR[s] xor zeroExt(i)
BEQ	4	s	t	o	Branch on Equal: **if** GPR[s] = GPR[t] **then** *PC* $\leftarrow$ *PC* + formOffset(o)
BGEZ	1	s	1	o	Branch ≥ 0: **if** GPR[s] ≥ 0 **then** *PC* $\leftarrow$ *PC* + formOffset(o)
BGTZ	7	s	0	o	Branch > 0: **if** GPR[s] > 0 **then** *PC* $\leftarrow$ *PC* + formOffset(o)
BLEZ	6	s	0	o	Branch ≤ 0: **if** GPR[s] ≤ 0 **then** *PC* $\leftarrow$ *PC* + formOffset(o)
BLTZ	8	s	0	o	Branch < 0: **if** GPR[s] < 0 **then** *PC* $\leftarrow$ *PC* + formOffset(o)
BNE	5	s	t	o	Branch Not Equal: **if** GPR[s] $\neq$ GPR[t] **then** *PC* $\leftarrow$ *PC* + formOffset(o)
LBU	36	b	t	o	Load Byte Unsigned: GPR[t] $\leftarrow$ zeroExt(memory[GPR[b] + formOffset(o)])
LW	35	b	t	o	Load Word (4 bytes): GPR[t] $\leftarrow$ memory[GPR[b] + formOffset(o)]
SB	40	b	t	o	Store Byte (least significant byte of GPR[t]): memory[GPR[b] + formOffset(o)] $\leftarrow$ GPR[t]
SW	43	b	t	o	Store Word (4 bytes): memory[GPR[b] + formOffset(o)] $\leftarrow$ GPR[t]

A.3 Jump Type Instructions

The instructions in Table 5 have a 26-bit field "addr" which is used to form the address to jump to. Forming this address is done by left-shifting the given "addr" field, a, by 2 bits, and then concatenating the (4) high bits of the *PC* with those $26 + 2$ bits to form a 32-bit address. This is written "formAddress$(PC, a,)$" in the table. For example if a is DECADE in hexadecimal notation, and *PC* is FFFACADE in hexadecimal notation, then formAddress(PC, a) is F37B2B78 in hexadecimal notation. (Note: if the high-order 4 bits of *PC* are 0, then formAddress(PC, a) is equivalent to left-shifting a by 2 bits.)

Table 5: Jump Type Instructions

Name	op	addr	Comment (Explanation)
JMP	2	a	Jump: $PC \leftarrow \text{formAddress}(PC, a)$
JAL	3	a	Jump and Link: GPR[$ra] $\leftarrow PC$; $PC \leftarrow \text{formAddress}(PC, a)$

The Jump and Link (JAL) instruction does a subroutine call. It does not explicitly manipulate the runtime stack.

A.4 System Calls

System calls are used to provide operating system services. System calls are made by instructions with op 0 and func 12 having the following format (with code field made of the 20 bits of what would be the rs, rt, rd, and shift fields of a register type instruction, all combined). The code field is used to specify the service requested.

System calls include exiting a program and various kinds of printing and reading of character data (bytes). These are described in Table 6, using C library equivalents. In the table, an entry of $-$ means that the contents of argument registers is not specified. Otherwise, the contents of particular argument registers are used to pass actual arguments to the system calls (the program must load the actual argument values into those registers before making the call). (Recall that the correspondence between named registers and register numbers is given in Table 1.) All printing done by these instructions is to the VM's standard output (stdout) and reading is from the VM's standard input (stdin).

Table 6: System Calls

code	name	arg. reg.	Effect (in terms of C std. library)
10	EXIT	-	`exit(0) // halt`
4	PSTR	$a0	GPR[$v0] $\leftarrow$ `printf("%s",&`memory[GPR[$a0]]`)`
5	PINT	$a0	GPR[$v0] $\leftarrow$ `printf("%d",`GPR[$a0]`)`
11	PCH	$a0	GPR[$v0] $\leftarrow$`fputc(`GPR[$a0]`,stdout)`
12	RCH	-	GPR[$v0] $\leftarrow$ `getc(stdin)`
256	STRA	-	start VM tracing; start tracing output
257	NOTR	-	no VM tracing; stop the tracing output

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 memory starting at the VM address given by the contents of GPR[$a0].

B Appendix B: Hints

B.1 Overall Structure of the Code

To implement the SRM 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 arrays that share the same storage: `memory.bytes`, `memory.words`, and `memory.instrs`. For example, the 4 bytes at byte address 36 can be accessed as `memory.bytes[36]` or as `memory.words[9]` or as

```
#include "machine_types.h"
#include "instruction.h"
// a size for the memory (2^16 bytes = 64K)
#define MEMORY_SIZE_IN_BYTES (65536 - BYTES_PER_WORD)
#define MEMORY_SIZE_IN_WORDS (MEMORY_SIZE_IN_BYTES / BYTES_PER_WORD)

static union mem_u {
     byte_type bytes[MEMORY_SIZE_IN_BYTES];
     word_type words[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 bytes, words, or binary instructions.

`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 lots of casting and bit manipulation. Note that the C compiler considers `memory.bytes` to have the type `byte_type[]`, and `memory.words` to have the type `word_type[]`, and `memory.instrs` to have the type `bin_instr_t[]`.

The registers can be `word_type` variables (where `word_type` is defined in `machine_types.h`), or an array of them.

Once the representation for memory and the registers is settled, implement the loading process and get the `-p` option to work. There is code in the disassembler that can be used as a model of what to do.

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.

You will find the function `instruction_assembly_form` from the instruction module helpful in doing the printing of instructions. See the code in the disassembler (`disasm.c` in particular) for a model of how to use it.

After getting the `-p` option to work, you need to implement the basic fetch-execute cycle for the VM (without the `-p` option): make a function that executes a single instruction and handles the tracing and call the function to execute each instruction 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 in the "Simplified RISC Machine Manual", which is available on Webcourses. 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. However, be sure to include an `EXIT` instruction in your test to stop your program's execution!

To write your own tests you can use the provided assembler, since the VM only takes binary object files as inputs.

The SRM 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 on webcourses; see the file named "`srm-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.

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` and `machine.c` (none of these are provided to you) into a single file named `vm.c`, in which case would replace our `machine_main.o` and `machine.o` with your `vm.o` On the other hand, you will need to leave the object file names in the list that your code uses, such as `bof.o` and `utilities.o`. 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 list of `VM_OBJECTS`.

You should not need to edit anything in the bottom half of the Makefile, which is the "developer's section" 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 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 SRM 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 SRM.

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 SRM'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, asm_lexer.l, asm_unparser.[ch], ast.[ch], lexer.[ch], pass1.[ch], assemble.[ch], and id_attrs) and the disassembler (including many mentioned above and also disasm_main.c, disasm.[ch]).

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

```
// $Id: machine_types.h,v 1.8 2023/09/18 02:24:31 leavens Exp $
// Machine Types for the Simplified Risc Machine (SRM)
#ifndef _MACHINE_TYPES_H
#define _MACHINE_TYPES_H

// registers encoded in instructions
typedef unsigned short reg_num_type;

// type of shift values encoded in immediate instructions
typedef unsigned short shift_type;

// type of functions encoded in instructions
typedef unsigned short func_type;

// type of immediate operands encoded in instructions
typedef unsigned short immediate_type;

// type of addresses
typedef unsigned int address_type;

// type of bytes
typedef unsigned char byte_type;

// type of machine words
typedef int word_type;

#define BYTES_PER_WORD 4

// Return the sign extended equivalent of i
extern int machine_types_sgnExt(immediate_type i);

// Return the zero extended equivalent of i
extern unsigned int machine_types_zeroExt(immediate_type i);

// Return the offset given by o, which is the sign extension of (o times 4)
extern int machine_types_formOffset(immediate_type o);

// Return an address formed by shifting a left by 2 bits
// and concatenating that with the high-order 4 bits of PC.
extern address_type machine_types_formAddress(address_type PC, address_type a);

#endif
```

Figure 7: The header file of the `machine_types` module, which provides some basic definitions for the VM.

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.