Electrical Engineering
& Computer Science
University of Missouri

ECE 4270/7270: Computer Architecture
Spring 2023
LAB 1: RISC-V Simulator

Scope

In this lab assignment, you will develop an instruction-level simulator for 32-bit RISC-V instruction set architecture (ISA). RISC-V (pronounced “risk-five”) is a new ISA that was originally designed to support computer architecture research and education, but now also become a standard free and open architecture for industry implementations. RISC-V is based on load/store architecture that is the memory can be accessed only through load and store instructions (data can be transferred between main memory and registers) and ALU instructions can only operate data residing on registers (not in memory). Don't worry about all the specifications of RISC-V ISA at this point, as we will cover them during our lectures in the upcoming weeks. For now, you should be fine with the given information and guidelines in this document.

The instruction-level simulator that you will develop should mimic the functional behavior of RISC-V instructions for a given input program, and the simulator should generate the desired output. A set of input programs will be provided for you to test your simulator. You will be asked to elaborate on your simulator and add more functionalities as we go through the labs, so you should take some time to think about how you should model and build your simulator.

Resources

You are given a skeleton of the instruction-level simulator that you will develop. You will find attached to this file in Canvas, mu-risc-v-v1.tar.gz. When you extract the file, you should have two directories called src/ and input/. In src/ directory, you will find mu-riscv.c and mu-riscv.h files. These are two files that contain the skeleton of the instruction-level simulator. To compile the simulator code, you can use provided Makefile.

You are expected to implement the following functions in mu-riscv.c whose declarations are provided in mu-riscv.h.

```
void handle_instruction();
void print_program();
```

handle_instruction() is called every cycle for a new instruction and it should simulate the current instruction. To do so, you should identify the type of instructions, as well as its operands. Depending on the instruction type and its operands, you should update registers/memory and other relevant architectural states, accordingly. For this assignment, your implementation should simulate the following RISC-V instructions according to the RISC-V Instruction Set Manual (https://riscv.org/wp-

content/uploads/2017/05/riscv-spec-v2.2.pdf)

ALU Instructions: ADD, ADDU, ADDI, ADDIU, SUB, SUBU, MULT, MULTU, DIV, DIVU, AND, ANDI, OR, ORI, XOR, XORI, NOR, SLT, SLTI, SLL, SRL, SRA
Load/Store Instructions: LW, LB, LH, LUI, SW, SB, SH, MFHI, MFLO, MTHI, MTLO
Control Flow Instructions: BEQ, BNE, BLEZ, BLTZ, BGEZ, BGTZ, J, JR, JAL, JALR
System Call: SYSCALL (you should implement it to exit the program. To exit the program, the value of 10 (0xA in hex) should be in $v0 when SYSCALL is executed.

On the other hand, the print_program() should print out the program loaded into memory. Note that the given input program is in hexadecimal format, and print_program() should translate hexadecimal format into the RISC-V assembly language.

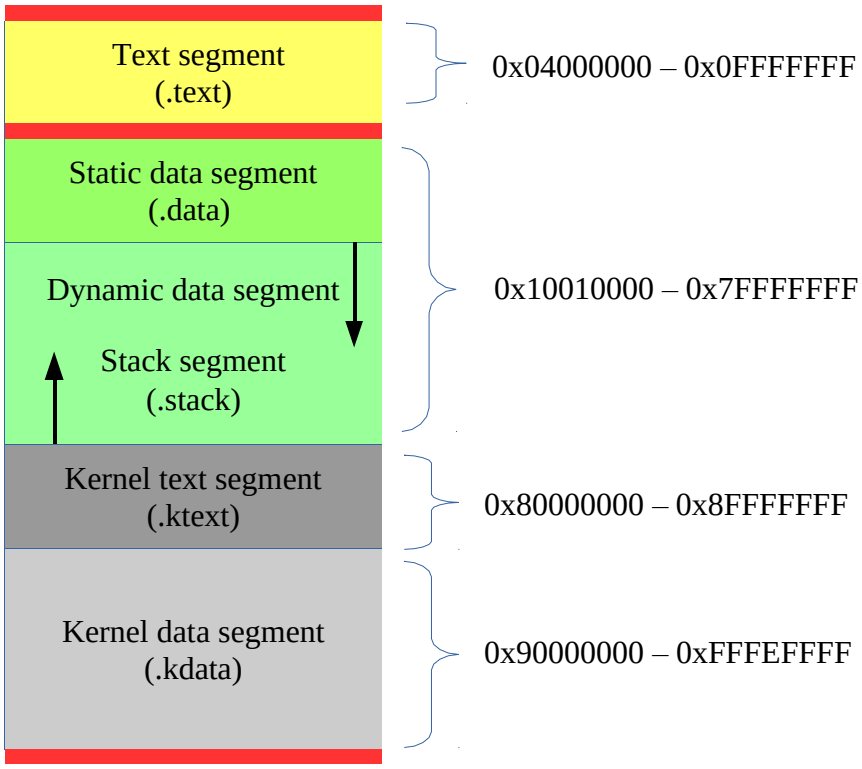
Technicalities

RISC-V address space is an array of 2^{32} bytes. Each byte has a 32-bit address. The addresses of RISC-V main memory range from 0x00000000 to 0xFFFFFFFF. However, user programs and data are restricted to the first 2^{31} bytes. The last half of the address space is used by the operating system. The user address space that is accessible to a user program are divided into following segments.

Text Segment: This is where the input program is loaded.

Data Segment: This segment holds the data that the program works on. It has two parts: static and dynamic. The size of static data does not change during the lifetime of the program, so it can be allocated by the assembler before the execution starts. The dynamic data, on the other hand, is allocated and deallocated as the program executes.

Stack Segment: It is the same address space with data segment; however, its start address is the top of user address space. It is used for local variables and parameters that are dynamically allocated and deallocated as procedures are activated and deactivated.



RISC-V has 32 general purpose registers and special HI/LO registers to store result of multiplication and division operations. Below, you can find the details of the general purpose registers.

Register Number	**Conventional Name**	**Usage**
$0	$zero	Hard-wired to 0
$1	$at	Reserved for pseudo-instructions
$2 - $3	$v0, $v1	Return values from functions
$4 - $7	$a0 - $a3	Arguments to functions - not preserved by subprograms
$8 - $15	$t0 - $t7	Temporary data, not preserved by subprograms
$16 - $23	$s0 - $s7	Saved registers, preserved by subprograms
$24 - $25	$t8 - $t9	More temporary registers, not preserved by subprograms
$26 - $27	$k0 - $k1	Reserved for kernel. Do not use.
$28	$gp	Global Area Pointer (base of global data segment)
$29	$sp	Stack Pointer
$30	$fp	Frame Pointer
$31	$ra	Return Address
$f0 - $f3	-	Floating point return values
$f4 - $f10	-	Temporary registers, not preserved by subprograms
$f12 - $f14	-	First two arguments to subprograms, not preserved by subprograms
$f16 - $f18	-	More temporary registers, not preserved by subprograms
$f20 - $f31	-	Saved registers, preserved by subprograms

System calls operates based on the values stored in $v0. Below you can find the expected behaviour of syscall based on the code stored in $v0. Notice that, to exit the program the value of 10 (0xA in hex) should be in $v0 when SYSCALL is executed.

Service	**Code in $v0**	**Arguments**	**Results**
print_int	1	$a0 = integer to be printed	
print_float	2	$f12 = float to be printed	
print_double	3	$f12 = double to be printed	
print_string	4	$a0 = address of string in memory	
read_int	5		integer returned in $v0
read_float	6		float returned in $v0
read_double	7		double returned in $v0
read_string	8	$a0 = memory address of string input buffer $a1 = length of string buffer (n)	
sbrk	9	$a0 = amount	address in $v0
exit	10		

Grading Rubric

Code: 75 points
Report: 25 points

Code (75 points):
Implementation of handle_instruction() 50 points. It should update the architectural states based on the instruction currently being simulated. Program should terminate properly (i.e. syscall with exit code should be implemented, as well).

Implementation of print_program() 25 points. It should print out the program loaded into memory.

Lab report (25 points):
Your report should give details about the work distribution within the group (who did what), milestones in your work and your implementation decisions (why did you choose the way you did it, and/or how did you do that).

Submission

You should submit the lab report along with the simulator code that you worked on. One member per group can submit both report and the code. Please, generate a pdf file for your report and name it lab1_report_groupX.pdf, then place it into the folder called lab1_groupX (where X is your group number—this number will be assigned to you during the lab session). The folder lab1_groupX should also contain the src/ and input/ folders that contain your code and given input program, respectively. Then, please compress the lab1_groupX folder as lab1_groupX.tar.gz and submit it through the Canvas.

Due Date

Your lab is due on: 02/17/2023