ÖZYEĞİN UNIVERSITY

CS 240 – Computer Architecture

Lab 2:
RISC-V Disassembler
&
Instruction-Level Simulation

Scope of the Lab

In this lab assignment, you will develop a disassembler (that decodes the hex representation of instructions) and instruction-level simulator for RISC-V ISA. RISC-V is a RISC (Reduced Instruction Set Computer) ISA, meaning that it has small set of simple, yet general instructions. 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 reside on registers (not in memory). You should not worry about the specifications of all of the RISC-V ISA at this point. 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 program files (represented as hex code) are provided for you to test your simulator.

Resources

You are given a skeleton of the disassembler and instruction-level simulator that you will develop. You can clone the skeleton code from the GitHub (called ozu-riscv32-v1).

- https://github.com/demirhansevim/ozu-riscv32-v1

Once you clone the code into your computer, you should have three directories called *src/, input/ and test/*. In src/ directory, you will find the following files:

- *ozu-riscv32.c*
- *ozu-riscv32.h*

These are the files that contain the skeleton of the instruction-level simulator. To compile the simulator code, you may use the provided Makefile. Once you build the code, it will create an executable called “ozu-riscv32”. You may use this executable with the input files provided in the *input/* directory.

You can run the following commands in your terminal while you are inside the *src/* folder to compile and execute your program.

```
make
./ozu-riscv32 ../input/test1.hex
```

Notice that, the given code is not complete, so it will not generate any output at the beginning. **You are expected to implement the following functions in ozu-riscv32.c** whose declarations are provided in ozu-riscv32.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 read the instruction from a memory, identify the type of instruction and all of its operands. Depending on the instruction type and its operands, you should update registers/memory and other relevant architectural states, accordingly. For this lab, your implementation should simulate the following RISC-V instructions below according to the descriptions given in RISC-V specification file (you can find RISC-V ISA specification document on LMS and on the web).

R-type Instructions: AND, OR, XOR, ADD, SUB, SLL, SRL, SRA, SLT, MUL, DIV, DIVU

I-type Instructions: ANDI, ORI, XORI, ADDI, SLLI, SRLI, SRAI, SLTI,
I-type Load Instructions: LW, LB, LH, LBU, LHU
S-type Instructions: SW, SB, SH
SB-type Instructions: BEQ, BNE, BLT, BGE, BLTU, BGEU,
UJ-type Instructions: JAL, JALR
U-type Instructions: LUI, AUIPC
System Call: ECALL (you should implement it to exit the program. To exit the program, the value of 93 (0x5D in hex) should be in register a7 (x17) when ECALL 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 hexademical format, and *print_program()* should translate hexadecimal format into the RISC-V assembly language.

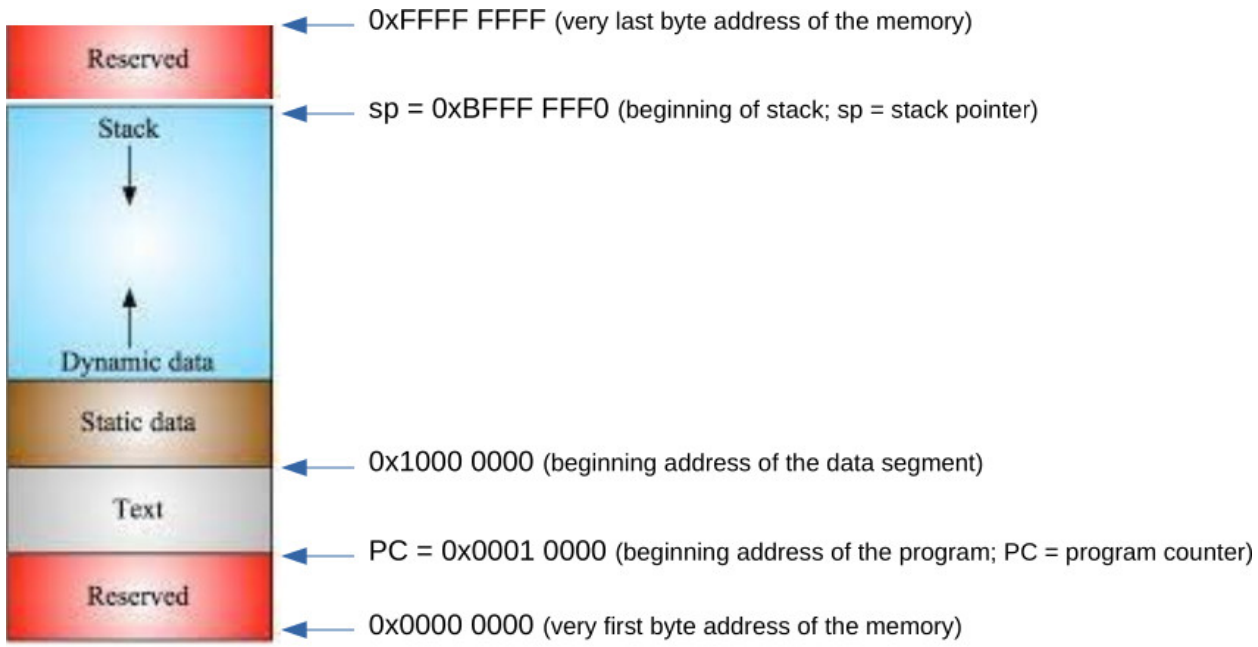
Technicalities

RISC-V address space can be considered as an array of 2^{32} bytes. Each byte has an address. The addresses of RISC-V main memory range from 0x00000000 to 0xFFFFFFFF (so, each address is 32-bit long). However, user programs cannot use all these 2^{32} bytes. Some parts of the address space is used by the operating system and reserved for kernel operations. The user address space that is accessible to a user program are divided into following segments.

Text Segment (.text): This is where the input program is loaded. It should start from the address 0x0001 0000 (a program is loaded into memory starting from this address, so the value of PC should be set to 0x0001 0000 at the beginning of the program).

Data Segment: This segment holds the data that the program works on. It has two parts: static data segment (.data) and dynamic data segment. 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 needed during the program execution.

Stack Segment (.stack): 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 (or functions) are activated and deactivated.



RISC-V has 32 general purpose registers to store result of operations. Below, you can find the details of the general purpose registers.

Registers	Symbolic names	Description
x0	zero	Hardwired zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5-x7	t0-t2	Temporary registers
x8-x9	s0-s1	Saved registers
x10-x11	a0-a1	Function arguments and return values
x12-x17	a2-a7	Function arguments
x18-x27	s2-s11	Saved registers
x28-x31	t3-t6	Temporary registers

System calls are called as executive calls (ecall) in RISC-V. To exit the program properly, system call "sys_exit" should be called at the end of the programs. Therefore, to exit the program via "sys_exit", the value of 93 (0x5D in hex) should be in register a7 when ecall instruction is executed.

When you run the ***print_program()*** should print out RISC-V instructions loaded into memory. You may use the register names as x0, x1, x2, ..., x31 or their symbolic names (as shown in the table above) when printing the operands of instructions.

Exercise 1: Disassembling Instructions and Simulating Them

During this lab, you will implement the ***handle_instruction()***, and ***print_program()*** functions for the skeleton code provided. The specification of these functions are given above.

Once you implemented these functions, you need to test them by using the input files given in the *input/* directory. You should be able to print the content of the registers, content of the specific address range of the memory, and assembly code of the program, properly.

Note that you need to decode each instruction (this is **disassembling** part) and perform the necessary changes on registers and memory (this is **simulation** part).

Once you complete the implementation of these functions, our TAs may evaluate your implementation with a separate input program, so you should implement the decode logic for **all instructions** mentioned above in the Resources section.

Exercise 2: Comparing the Simulation Results with Spike

Once you complete the implementation of the simulator, you should debug and compare the results of the simulator you implemented and the results of the Spike. To run the Spike for the program that you disassembled, you should put your disassembled program into the “testAssembly.s” file which will be in the *test/* directory.

To generate the executable from the testAssembly.s file you should assemble and the link it with the following commands (the content of ldoptions.txt file may be edited based on your system):

riscv32-unknown-elf-as -mabi=ilp32 testAssembly.s -o testAssembly.o
riscv32-unknown-elf-ld -T myLinkScript.ld testAssembly.o -o testAssembly @ldoptions.txt

For Mac Users:

- *riscv64-unknown-elf-as -march=rv32ima -mabi=ilp32 testAssembly.s -o testAssembly.o*
- *riscv64-unknown-elf-gcc -T myLinkScript.ld -march=rv32i -mabi=ilp32 testAssembly.o -o testAssembly*

Then, you can run Spike in debug mode (with -d flag) and compare Spike results with the results of your simulator implementation.

spike --isa=RV32IMA -d pk testAssembly

Note that the executable includes other instructions that allows to program to boot up, so you should able to locate the beginning of your main() function (*myLinkScript.ld* forces the linker to put main function starting at 0x00010000 address). So, when you are in debug mode of Spike, you should skip the startup code and stop at address 0x00010000. To do so you can run the following command (when you are in debug mode):

(spike) untiln pc 0 00010000

Then, you should be able to go over each instruction one by one and show the content of the registers and memory addresses. For the debug commands, you may use “help” command in debug mode, which will list the available commands and show how to use these commands.

Please show the running Spike simulations for the programs that you disassembled in Exercise 1 to our TAs, and report any mismatches (if there is any).