CS 240 – Computer Architecture

Lab 3:
RISC-V Assembler
&
Assembly Programming

Scope of the Lab

In this lab assignment, you will develop an assembler for the RISC-V ISA. This assembler will take a program written in RISC-V assembly and will convert into corresponding machine code. In the previous lab assignment, you have developed an instruction-level simulator for RISC-V and test it out with some test programs provided with the assignment. For this assignment, you will use the output of your RISC-V assembler as an input to RISC-V instruction-level simulator you have developed during the previous lab assignment.

Your RISC-V assembler should accept an input file that contains the program written in RISC-V assembly and should generate an output file that contains the corresponding RISC-V machine code. The machine code will be similar to ones that you used in the previous lab assignment (each line represents an instruction and encoded as 32-bit hexadecimal value). As an example, the following C code excerpt:

```
int sum = 0;
for(i = 0; i < 10; i++){
        sum += i;
}
```

can be written in RISC-V ISA as:

```
[address of ins]    instruction
-------------------------------------------------------------------------
[0x00010000]        addi t0, zero, 0x0      # set sum initially to 0
[0x00010004]        addi t1, zero, 0xA      # upper bound for i is 10
[0x00010008]        addi t2, zero, 0x0      # i is initially 0
[0x0001000c]        add t0, t0, t2          # update sum
[0x00010010]        addi t2, t2, 0x1        # increment i
[0x00010014]        bne t1, t2, -8          # if true, then jump back to loop at 0x0001000c
[0x00010018]        addi a7, zero, 0x5d     # set a7 to 0x5d to exit
[0x0001001c]        ecall
```

Of course, you can write a different assembly program for the same C code; however, this is just for illustration. Also, the address of each instruction is provided to help you understand the code better. Therefore, you don't need to write the addresses of instructions in your RISC-V assembly code.

Your RISC-V assembler should convert the above assembly program to the following machine code:

```
00000293
00a00313
00000393
007282b3
00138393
fe731ce3
05d00893
00000073
```

Your assembler should be able to encode (i.e., generate binary representations) the following instructions:

R-type Instructions: ADD, SUB, AND, OR, XOR, SLL, SRL, SLT
I-type Instructions: ADDI, ANDI, ORI, XORI, SLLI, SRLI, SLTI
Load Instructions: LW, LB, LH
Store Instructions: SW, SB, SH
S-type Instructions: BEQ, BNE, BLT, BGE
UJ-type Instructions: JAL, JALR
U-type Instructions: LUI, AUIPC
System Call: ECALL

and any other instruction that you used when solving the problems given below.

Resources

On github, you are given a skeleton of an assembler (Lab3-Assembler) that can read an input file and then write the encoded instructions in .hex and .bin files. The **assembler.c** contains your skeleton code. A **loop.s** file that contains the assembly code written above and a Makefile to compile your program.

- https://classroom.github.com/a/85-2B8vj

To compile and run the assembler, you should run the following commands in your terminal.

```
make clean
make
./assembler loop.s
```

This would generate two output files:
- "loop.hex" which is a text file and should have the hex representation of the instructions. This file can be opened with any text editor, so you can actually see the hexadecimal representation of the instructions you encoded.
- "loop.bin" which is a binary file and should have the binary representation of the instructions you encoded. Since this file is binary, it cannot be opened with a regular text editor. You may use programs, such as xxd to read the content of the binary file. This binary file is essential, since it will be used to generate an executable that you can run it with Spike simulator. Note that the instructions are stored in little-endian format (least significant byte first) in RISC-V; therefore an instruction whose hexadecimal representation is "fe731ce3" should be written into binary file (i.e., .bin file) in the following order "e31c73fe" (note that it should be written as binary, not as string. "e31c73fe" is shown here to indicate the order of bytes to be stored in .bin file). For more clarification, please look at the comments in assembler.c)

Exercises

Once you complete the implementation of the RISC-V assembler, you should write the RISC-V program for the following problems and save them with .s extension (e.g., problem1.s; and problem2.s). Then, you should use the assembler that you wrote to convert the assembly instructions in .s file into machine code.
You should use a separate input file for each problem. For example, for the firs problem, you should have:

```
./assembler problem1.s
```

The above command will generate two output files, as mentioned earlier.

- problem1.hex
- problem1.bin

Once you have obtained these output files, you can create an executable and debug the program with Spike. You can look at the content of problem1.hex to see hexadecimal representation of the instructions. However, to generate executable, you should use the "problem1.bin" file and follow the steps below.
- you are given "problem1.asm" file that you would need to generate executable from the "problem1.bin" file. First, make sure the content of "problem1.asm" is correct. it should have a line in which your binary instructions are included (e.g, .incbin "problem1.bin")
- then, you should execute the following commands to create executable:

```
riscv32-unknown-elf-as -mabi=ilp32 problem1.asm -o problem1.o

riscv32-unknown-elf-ld -T myLinkScript.ld -o problem1 problem1.o @ldoptions.txt
```

For Mac Users:

```
riscv64-unknown-elf-as -march=rv32ima -mabi=ilp32 problem1.asm -o problem1.o
riscv64-unknown-elf-gcc -T myLinkScript.ld -march=rv32i -mabi=ilp32 -o problem1 problem1.o
```

The first command above creates an object file for your program whose instructions are included from the "problem1.bin" file. The second command links necessary object files and generates executable program. Upon completion of these commands you should have an executable (in this this case, with the name of "problem1").

After obtaining executable program, you can run Spike in debug mode (with -d flag) and locate the instructions that you wrote for the program and check if they are running correctly.

Note that to find the location of your instructions in the executable, you can use the following command:

```
riscv32-unknown-elf-objdump -d --disassemble=main problem1
```

The above command should print out the instructions in the main() function (which includes your assembly code you wrote) along with their addresses in the memory.

To launch the Spike in debug mode with your executable run the following command.

```
spike --isa=RV32IMA -d pk problem1
```

Once you know the address of the beginning of your program, you can fast-forward to this address in Spike with the following command (when you are in debug mode in Spike):

```
(spike) untiln pc 0 <address of the first instruction of your code>
```

Below are the problems for which you need to write an assembly program.

Problem 1: Big-endian to little-endian conversion

Write a RISC-V assembly program that converts 8 words of memory starting at address 0x10000000 from little-endian to big-endian.

> **Some hints:** Start by doing an example by hand for a word starting at memory address 0. Be sure to write out the byte addresses below each byte (for both big- and little-endian). For example, the word 0x1234ABCD stored in little-endian at memory address 0 would have the following byte addresses and data:

Address 0 holds 0xCD
Address 1 holds 0xAB
Address 2 holds 0x34
Address 3 holds 0x12

To convert this to big-endian, you would move the bytes to the following addresses (i.e., by first loading the bytes into registers using load byte: `lb`, and then storing them into different address locations using store byte: `sb`):

Address 3 holds 0xCD
Address 2 holds 0xAB
Address 1 holds 0x34
Address 0 holds 0x12

Problem 2: Fibonacci Number

Write a RISC-V program that finds the Fibonacci number of a given value. Calculate the Fibonacci number of 10 to test your RISC-V program.

For grading, submit the code for your completed assembler along with the RISC-V programs you've written for the problems above. During the grading session, you will demonstrate the running Spike simulations for these programs, using executables generated from the binary files produced by your assembler.