Lab 2
RISC-V: Arithmetic, Logical and Shift Instructions

Module ID : CX-204

Modern Computer Architecture

Instructor : Dr. Muhammad Imran

Version 1.1

Document History

The changes and versions of the document are outlined below:

Version	State / Changes	Date	Author(2)
1.0	Initial Draft	Dec, 2024	Qamar Moavia, Dr. Muhammad Imran

Table of Contents

Objectives

By the end of this lab, students will

- Install and get familiar with the Venus simulator.
- Understand what exactly is assembly language and how it's different from High Level Languages like C/C++, Java etc.
- Understand the RISC-V Arithmetic, Logical, Shift and Data Transfer instructions.
- Write basic programs using these Arithmetic, Logical and Shift, and Data Transfer instructions.

Deliverables

- Assembly Code Files for each task
- Output screenshots.
- Git Repository Link.

1. Assembly Language

Assembly language is a low-level language that helps to communicate directly with computer hardware. It uses mnemonics to represent the operations that a processor must do.

It is an intermediate language between high-level languages like C++ and the machine language.

In an assembly language program, each instruction represents a single operation that the computer's CPU can perform. These can include simple arithmetic and logical operations, such as adding and subtracting values, as well as more complex operations that involve manipulating data stored in the computer's memory. Assembly language programs are typically written in a text editor and then assembled using a specialized software tool called an **assembler.** The assembler converts this assembly language code into machine code that the computer's CPU can really understand.

Let's say that we have this very simple c code that adds two number 5 and 10:

```
int main() {
    int a = 5, b = 10;
    int sum = a + b;
    return 0;
}
```

You may write it in this high level programming language but computers don't understand this language. To ask a computer's CPU for this task we

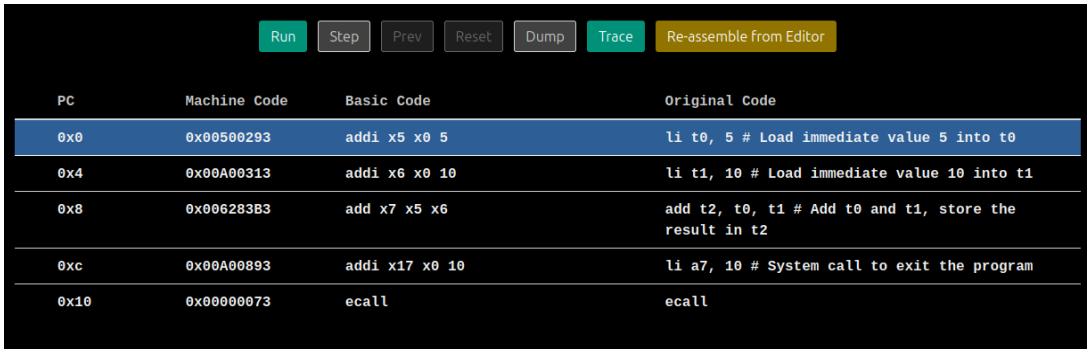
need to talk to the CPU in the language that he can understand. So, these instructions need to be translated into machine code (Only 1's and 0's). But for that we will first convert that into the assembly language that is the bridge between human and CPU.

Here is the assembly language code that is equivalent to the above C-code.

```
li t0, 5          # Load immediate value 5 into t0
li t1, 10         # Load immediate value 10 into t1
add t2, t0, t1    # Add t0 and t1, store the result in t2
# Program ends
li a7, 10         # System call to exit the program
ecall
```

This assembly code will be assembled (converted into the machine language) by the assembler. The machine code for the above assembly code will be something like this:

```
0x00500293
0x00A00313
0x006283B3
0x00A00893
0x00000073
```



2. Introduction to Venus Simulator

For the next few labs, we will work with several RISC-V assembly files, each of which has a ".s" file extension. To run these, we will be using Venus, an academic RISC-V assembler and simulator. For these labs we will use Venus website for simplicity but we recommend you to go through these instructions for Venus simulator, Venus reference.

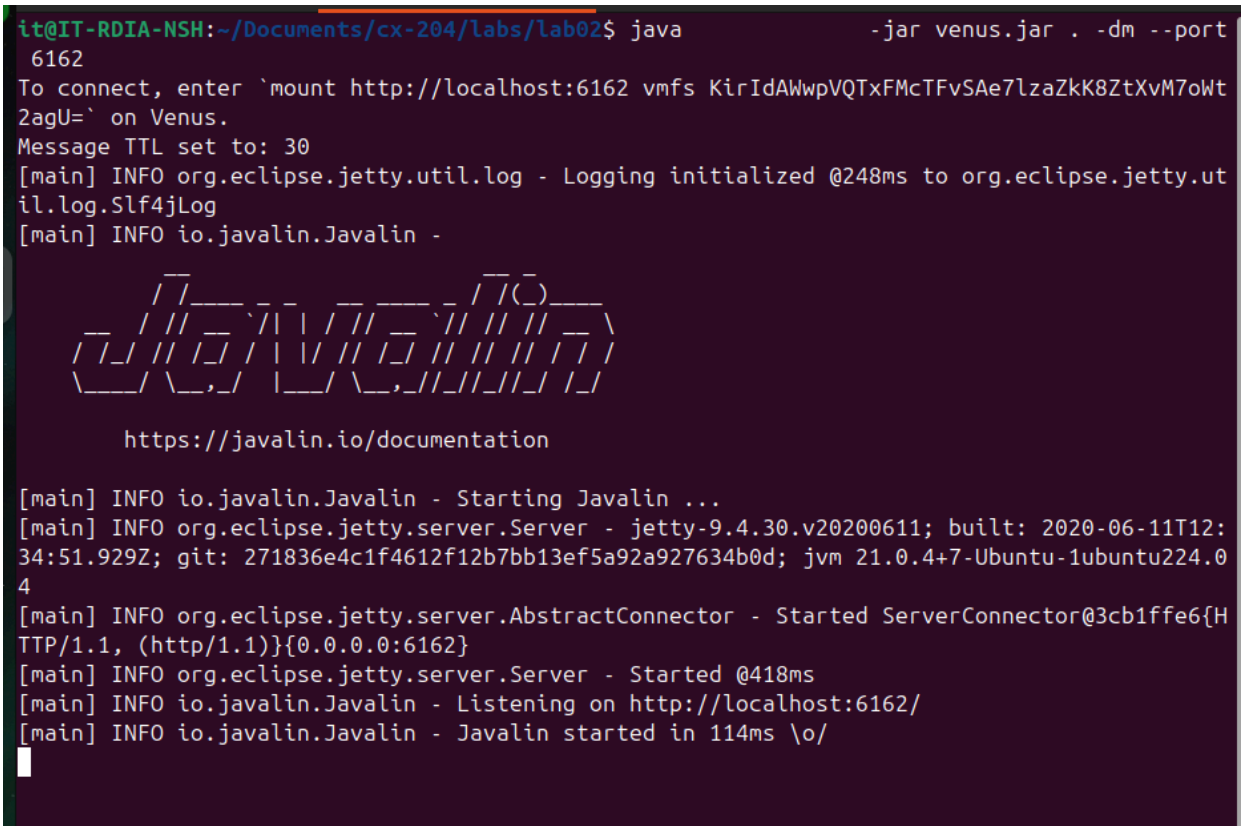
TASK 1 : Setting Up & Getting Familiar with Venus

You can "**mount**" a folder from your local device onto Venus's web frontend, so that edits you make within the browser Venus editor are reflected in your local file system, and vice versa. If you don't do this step, files created and edited in Venus will be lost each time you close the tab, unless you copy/paste them to a local file.

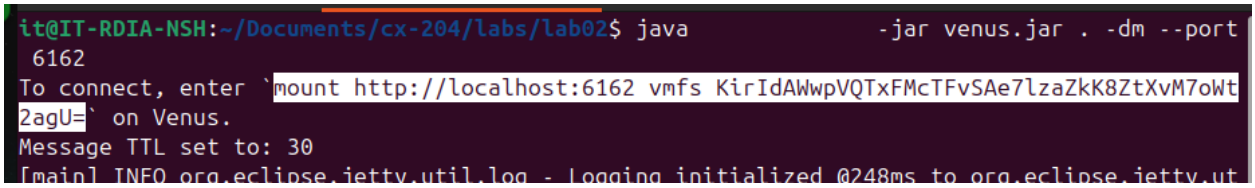
This exercise will walk you through the process of connecting your file system to Venus, which should save you a lot of trouble copying/pasting files between your local drive and the Venus editor.

1. Firstly, you are required to download the **lab2** that is uploaded on the lms.
2. Unzip the lab2 that you just downloaded from the lms. Switch to the lab2 directory. You will be working in this directory for this whole lab.
3. In your labs folder, **run** java -jar venus.jar -dm .
   This will expose your lab directory to Venus on a network port.
   - You should see a big "**Javalin**" logo.
   - **If the above command does not work**, then you can specify another port number explicitly by appending --port PORT_NUM to the command (for example, java -jar venus.jar . -dm --port 6162 will expose the file system on port **6162**).

4. You will see something like this on your terminal upon success:



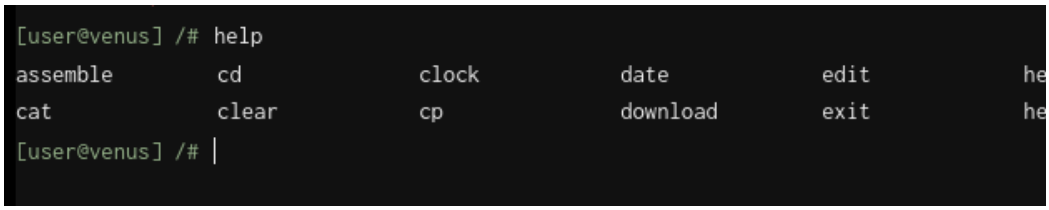
5. You will see this highlighted command on your terminal that include a key to link venus to your file system:



6. **Copy this command** and then **Open** https://venus.cs61c.org in your web browser (**Chrome or Firefox are recommended**).

   Note: Change **vmfs** to **cx-204-labs** in the copied command.

7. In the Venus web terminal, **paste** the copied command. This connects Venus to your file system.
8. **Go to** the "Files" tab. You should now be able to see your cx-204-labs directory under the current folder. In case you haven't modified vmfs with cx-204-labs that will be vmfs.
9. Now go to the terminal window of the venus again and enter the help command to see the different commands supported by venus.



10. You can use basic commands to create a directory ("mkdir"), or to list the files present in the current working directory ("ls") etc.
11. Switch to cx-204-labs directory and make sure it works by creating a new file ex1_hello.s. You can do this by running the following command in the terminal window of venus: edit ex1_hello.s.
    - You should see the ex1_hello.s created and opened in the editor. This editor behaves like most other text editors without many of the fancier features.
12. Copy the following RISC-V Assembly code to ex1_hello.s:

```
.text
   main:
        li t0, 5   # Load immediate value 5 into register t0
        li t1, 10  # Load immediate value 10 into register
t1
        add a1, t0, t1  # a1 <= t0 + t1

        # Print the result using a system call
        li a0, 1       # System call for printing an integer
        ecall

        # Exit the program
```

```
li a0, 10      # System call for exit
ecall
```

13. To assemble this assembly program that is opened in the editor, **click** the "Simulator" tab, and **click** "Assemble & Simulate from Editor".
    - In the future, if you already have a program open in the simulator, click "Re-assemble from Editor" instead. Note that this will overwrite everything you have in the simulator tab, such as an existing debugging session.
14. To run the program, **click** "Run".
    - You can see other buttons like "Step", "Prev", and "Reset". You will use these buttons later on in the labs to debug your program or to run your assembly program line by line.
15. Go back to the editor tab, and **edit** ex1_hello.s so that the output prints the difference of the same two numbers 5 and 10.
    - Hint: If you want to display an integer number, make sure that number is stored in the register a1. Then store that value "**1**" in the register a0. Then running the system call will result in printing the number that is stored in register a1. What ecall does exactly is out of scope for this class though.

Note: Please do read about RISC-V environmental-calls from the following link: https://github.com/61c-teach/venus/wiki/Environmental-Calls

16. **Save** the changes you just made by hitting Cmd + s on macOS and Ctrl + s on Windows/Linux. This will update your local copy of the file.
17. **Open** the file on your local machine using the text editor of your choice to check and make sure it matches what you have in the web editor.

    - **Note:** If you make any changes to a file in your local machine using a text editor, if you had the same file open in the Venus editor, you'll need to reopen it from the "Files" menu to get the new changes **or** You can reopen the file by just running the following command in the terminal window in the venus: edit ex1_hello.s.

18. **Run** the program again, you should now see **-5** if you modified the source file correctly.

3. Classification of Instructions

The RISC-V instructions can be simply classified into these categories:

1. Arithmetic Instructions
2. Logical and Shift Instructions
3. Data Transfer Instructions
4. Program Control/Branch Instructions
5. System Instructions

Note: In today's lab we will only work on the Arithmetic, logical and shift instructions. Here is the table for the Arithmetic, Logical and Shift instructions for your reference.

Instruction	Description	Operation
`add rd, rs1, rs2`	add	`rd = rs1 + rs2`
`sub rd, rs1, rs2`	sub	`rd = rs1 − rs2`
`addi rd, rs1, imm`	add immediate	`rd = rs1 + SignExt(imm)`
`or rd, rs1, rs2`	or	`rd = rs1 | rs2`
`and rd, rs1, rs2`	and	`rd = rs1 & rs2`
`xor rd, rs1, rs2`	xor	`rd = rs1 ^ rs2`
`ori rd, rs1, imm`	or immediate	`rd = rs1 | SignExt(imm)`
`andi rd, rs1, imm`	and immediate	`rd = rs1 & SignExt(imm)`
`xori rd, rs1, imm`	xor immediate	`rd = rs1 ^ SignExt(imm)`
`sll rd, rs1, rs2`	shift left logical	`rd = rs1 << rs2`$_{4:0}$
`srl rd, rs1, rs2`	shift right logical	`rd = rs1 >>` `rs2`$_{4:0}$
`sra rd, rs1, rs2`	shift right arithmetic	`rd = rs1 >>>` `rs2`$_{4:0}$
`slli rd, rs1, uimm`	shift left logical immediate	`rd = rs1 << uimm`
`srli rd, rs1, uimm`	shift right logical immediate	`rd = rs1 >> uimm`
`srai rd, rs1, uimm`	shift right arithmetic imm.	`rd = rs1 >>> uimm`

1. Arithmetic Instructions

Arithmetic instructions perform basic mathematical operations such as addition, subtraction, multiplication, and division. These operations are essential for numerical calculations in programs.

a. Example 1: add t2, t0, t1 → Adds t0 and t1, storing the result in t2.
b. Example 2: sub t2, t0, t1 → Subtracts t1 from t0, storing the result in t2.

TASK 2 : Arithmetic operations

Do, the following mini tasks on arithmetic operations using RISC-V assembly language:

1. Create a separate task2.s file for this task and open that file in the venus.
2. Using RISC-V add / addi / sub instructions, implement 1's complement (NOT) of a value in x5 register.

   i.e., if x5 = 0xFFFFFFFF, after 1's complement x5 = 0x00000000

3. Print the contents of register x5 register.

   Note: If you want to print some integer value:

   a. Copy that integer value into the register a1.
   b. Then load the integer value 1 into a0.
   c. Then run the ecall instruction.

   Here, this example will print integer value 42:

```
addi a1, x0, 42        # a1 <= 42

addi a0, x0, 1         # print_int ecall

ecall
```

4. Write a RISC-V assembly code that creates a 32-bit constant value 0x12345678 and places it in x5, print the content of x5 register at the end.
5. Write a RISC-V code that tests whether a number in x5 register is even or odd. The result is indicated in x10 such that x10 is 0 if the number is even and 1 if the number is odd.

2. Logical and Shift Instructions

Logical and shift instructions manipulate data at the bit level. Logical instructions perform bitwise operations, while shift instructions move bits to the left or right.

- Example 1: and t2, t0, t1 → Performs bitwise AND between t0 and t1, storing the result in t2.
- Example 2: sll t2, t0, 1 → Shifts the bits in t0 left by 1 position, storing the result in t2.

TASK 3 : Logical and Shift operations

Following are the mini-sub tasks that are required to be performed using RISC-V assembly language:

1. Write a RISC-V code that tests whether a number in x5 register is even or odd. The result is indicated in x10 such that x10 is 0 if the number is even and 1 if the number is odd.
2. Write a RISC-V assembly code that toggles a specific bit of x5 with the bit number to be toggled being specified in x10.
3. Write a RISC-V assembly code that checks if there are an even number of 1's in value specified in register x5 or an odd number of 1's. If there are even numbers of 1's in x5, set x10 to 0 otherwise set x10 to 1.
4. Write a RISC-V assembly code that selects one of the 4 bytes from a register x5 and places the resulting byte (with 0 extension) in x10. The byte number (0,1,2 or 3) is specified in x6.
5. Write a RISC-V assembly code that selects one of the 4 bytes from a register x5 and places the resulting byte (with sign extension) in x10. The byte number (0,1, 2 or 3) is specified in x6.
6. Write a RISC-V assembly code that compares x5 and x6 and sets x10 to 1 if x5 is greater (signed comparison) than x6.
    a. Do this without using slt (set less than) instruction which achieves the same thing!

TASK 4 : Some More Practice

Following are the mini-sub tasks that are required to be performed using RISC-V assembly language:

1. Write a RISC-V code that sorts the data in register x5 and x6 in ascending order. e.g if we have x5 = 10, x6 = 4 then it should swap x6 and x5. Do this using the minimum number of registers being used in your code.
2. Once sorting is done then x5 contains the minimum value and x6 contains the maximum value.
3. Now identify the bit positions in the 32-bit binary representation of x5 that are set (their binary value is 1'b1). Use these positions to clear the corresponding bits in x6 and store the final output in x7.

   Let's say the bit positions 0, 1, 3 and 4 are set as given blow

   ```
   x5 = 0b00000000000000000000000000011011
   ```

   and the value of x6 register is this:

   ```
   x6 = 0b00000000000000000111010101110111
   ```

   Then bit number 0, 1, 3 and 4 should be cleared in x6, and the final value of x7 will be something like this:

   ```
   x6 = 0b00000000000000000111010101100100
   ```

4. Now again use the same positions to take one's complement of those bits in x6 that are set in x5 register. The final result should be stored in the x28 register.

Good Luck 🙂