# CS330L Computer Architecture Lab
# Lab 02: Introduction to RISC-V assembly

CS Program
Habib University

Spring 2020

## 1 Introduction

In this lab we will learn:

1. a brief introduction the RISC-V assembly language
2. introduction to an online RISC-V simulator
3. use RISC-V arithmetic inscturctions
4. use RISC-V memory instructions

## 2 What is RISC-V assembly?

As you know assembly is a low level language used to write programs for processors. Being a low level language it is hardware dependent which means, that every processor, architecture actually, has its own assembly language with its own syntax. What this means in practice is that Intel x86 assembly is different from ARM assembly which, in turn is different from POWER assembly, etc. RISC-V therefore has its own assembly language.

Assembler is a software that transforms this human readable assembly language into the machine language (binary) that the processor understands and can execute.

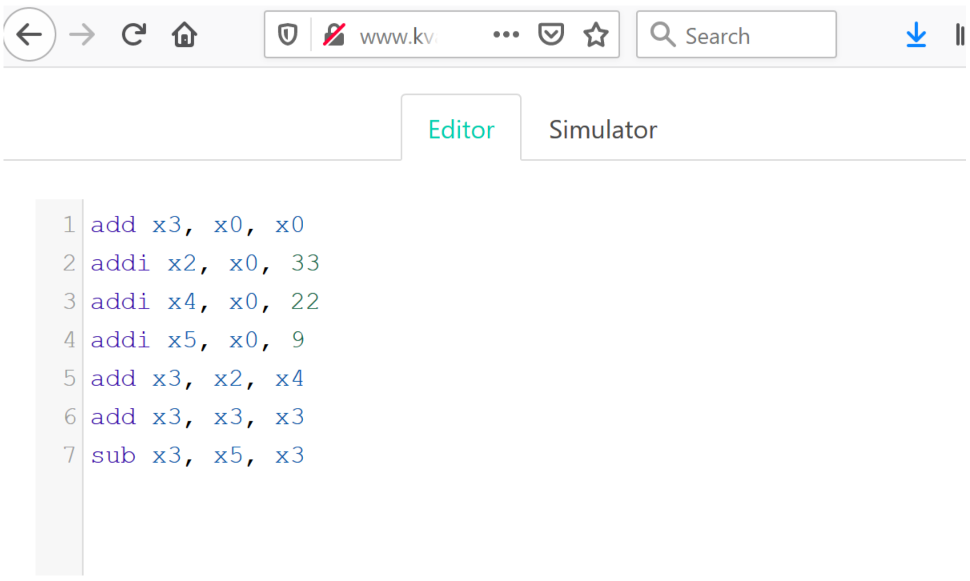
In this lab we will write some very basic code in RISC-V assembly language, convert it into binary and try to execute it on a RISC-V simulator.

## 3 What is a processor simulator?

Usually you need a processor, the hardware chip, to execute a program binary but there often there arise situations, like ours now, where access to the hardware processor is unavailable. To work around such situations, there exist softwares that mimic the behaviour of a hardware processor. Such softwares are called processor simulators.

Processor simulators behave exactly like the processors they simulate. They take the binary program as input, take the instructions in that program and execute them one by one. They maintain the internal state of a processor, i.e., its registers, it's pipeline and other circuits, in software and update this state in the light of the instructions being executed exactly as it would be updated by a real hardware processor chip.

If that didn't make any sense, let us see an example. The simulator that we will be using in this course is available online at the address: `http://www.kvakil.me/venus/`. Type this URL in your browser and you will see a view like the one shown in Figure 1.



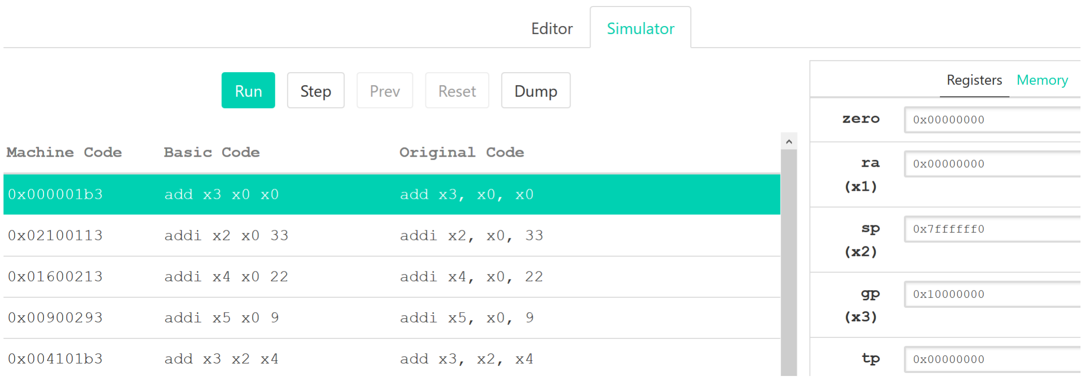
```
add x3, x0, x0
addi x2, x0, 33
addi x4, x0, 22
addi x5, x0, 9
add x3, x2, x4
add x3, x3, x3
sub x3, x5, x3
```

Figure 1: A RISC-V simulator

There are two main tabs:

- Editor: where we write our assembly code. It is the one open in the figure. You can see some code as well.
- Simulator: where we execute/simulate our code.

Click on the **Simulator** tab and you will see the view shown in Figure 2.



| Machine Code | Basic Code | Original Code |
|---|---|---|
| 0x000001b3 | add x3 x0 x0 | add x3, x0, x0 |
| 0x02100113 | addi x2 x0 33 | addi x2, x0, 33 |
| 0x01600213 | addi x4 x0 22 | addi x4, x0, 22 |
| 0x00900293 | addi x5 x0 9 | addi x5, x0, 9 |
| 0x004101b3 | add x3 x2 x4 | add x3, x2, x4 |

| | |
|---|---|
| zero | 0x00000000 |
| ra (x1) | 0x00000000 |
| sp (x2) | 0x7ffffff0 |
| gp (x3) | 0x10000000 |
| tp | 0x00000000 |

Figure 2: The simulation tab

This is where most of the action will happen so let us take some time to explore its composants

## 3.1 The simulation tab

### 3.1.1 The code window

The major screen area is being occupied by the screen showing our code. It shows us three columns. From right to left they are:

- Original code. This shows the code in the form that we wrote in the Editor tab. The assembler gives different names to many registers for user friendliness, e.g. it names x0 as zero, x2 as sp, etc. These names can be used in the Editor window instead of x0-x31. If we had used those names we would have seem them here.
- Basic code. This shows the code where the assembler user friendly register names have been translated into their original names. Since we used the original names from the start, therefore both columns are identical for us. However, if we had used sp instead of x2, then in the right most column it would have appeared as sp and in the middle column as x2.
- Machine code. As the name suggests this column shows the final binary generated for each of our assembly instructions when the assembler translates them. Count the number of bits in each binary instruction. What can you infer?

### 3.1.2 The register/memory window

To our right we can see a vertical window containing two tabs: Registers and Memory. The Register tab is shown in the Figure 2.

- Registers: This is where the simulator shows us the internal processor state that it is maintaining, i.e., the registers. All 32 RISC-V registers, from x0-x31, are visible here each showing the value it contains.
  For ease of programming, the assembler give different names to these register which we can use in the assembly language. We can see from Figure **??** that x2 has another alias called sp. Similarly we can see aliases for all other registers.
- Memory: If you click on te memory tab, you will see a view as shown in Figure 3. This shows us a view of memory that the simulator is maintaining. Memory as we know is made up of bytes and each byte has an address. We observe on the left "Address" column the addresses starting from 0x00 at the bottom and increasing progressively upwards.

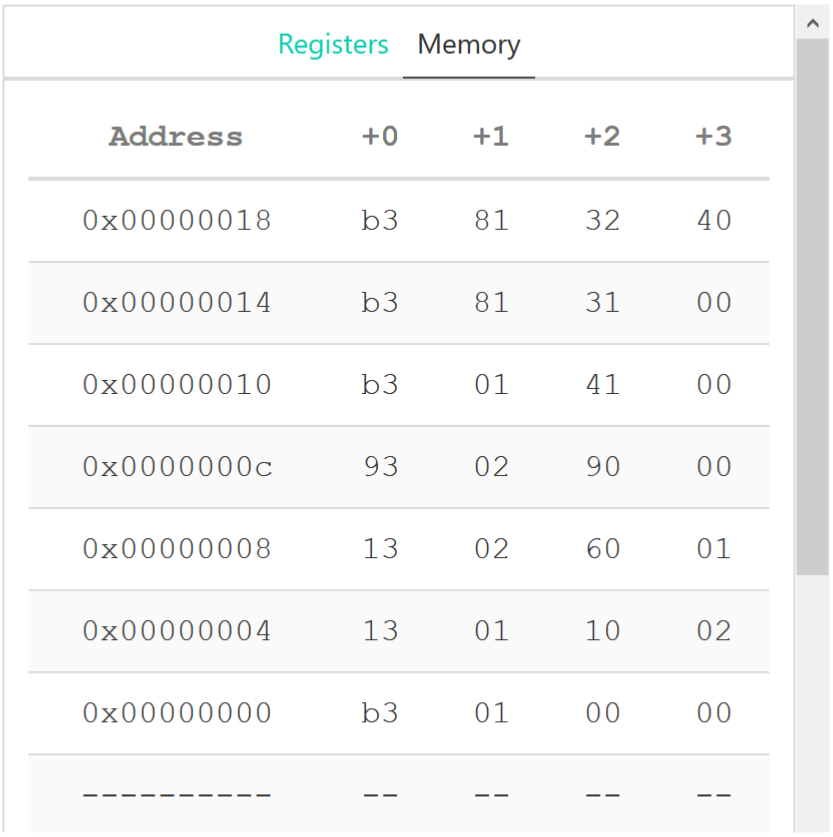
  To the right we have the memory locations pointed to by these addresses and the values contained in those memory locations. Each location is a byte and the column title +0, +1, +2, +3 gives the offset of that particular memory byte from the address shown in the right most column. Using the Address column with these offsets you can locate each byte of memory and the value it contains.
  Let's try an example: Locate byte number 22 (this number is in decimal) and find out what value it contains.

  If you scroll the memory tab to the bottom you will see the view shown in Figure 4.
  First observe the drop down menu "Display Setting" with which you can control whether the values in memory and registers should be shown in a hexadecimal base or some other. By default they are in base 16.
  The "Up" and "Down" buttons would take you up and down in the simulated memory.

### 3.1.3 The execution buttons

Just above the code window we see five buttons as shown in the Figure 5. Initially when we arrive at the simulator tab from the Editor tab, the code has not started executing yet. The highlighted instruction is ready to execute next. These buttons help us execute our code:

- Run: This button if pressed would execute our code till the end in one go.
- Step: This button would just execute one instruction, the highlighted one, and then stop.
- Prev: This button would take you one step back in code.



| Address | +0 | +1 | +2 | +3 |
|---|---|---|---|---|
| 0x00000018 | b3 | 81 | 32 | 40 |
| 0x00000014 | b3 | 81 | 31 | 00 |
| 0x00000010 | b3 | 01 | 41 | 00 |
| 0x0000000c | 93 | 02 | 90 | 00 |
| 0x00000008 | 13 | 02 | 60 | 01 |
| 0x00000004 | 13 | 01 | 10 | 02 |
| 0x00000000 | b3 | 01 | 00 | 00 |
| ---------- | -- | -- | -- | -- |

Figure 3: The memory tab

- Reset: Where ever you are in your execution, pressing this button would take you to the start of the code. A restart.
- Dump: Pressing this button will copy your binary machine code to the clip board so that you can paste it somewhere else.

As you execute the code, whether by step-by-step or in one go, you will see that the updates in the processor state are reflected in the registers as well as the memory. Thus the simulator reproduces the behaviour of the actual hardware processor with the execution of each instruction.

## 4 Task 1: Using Arithmetic Instructions

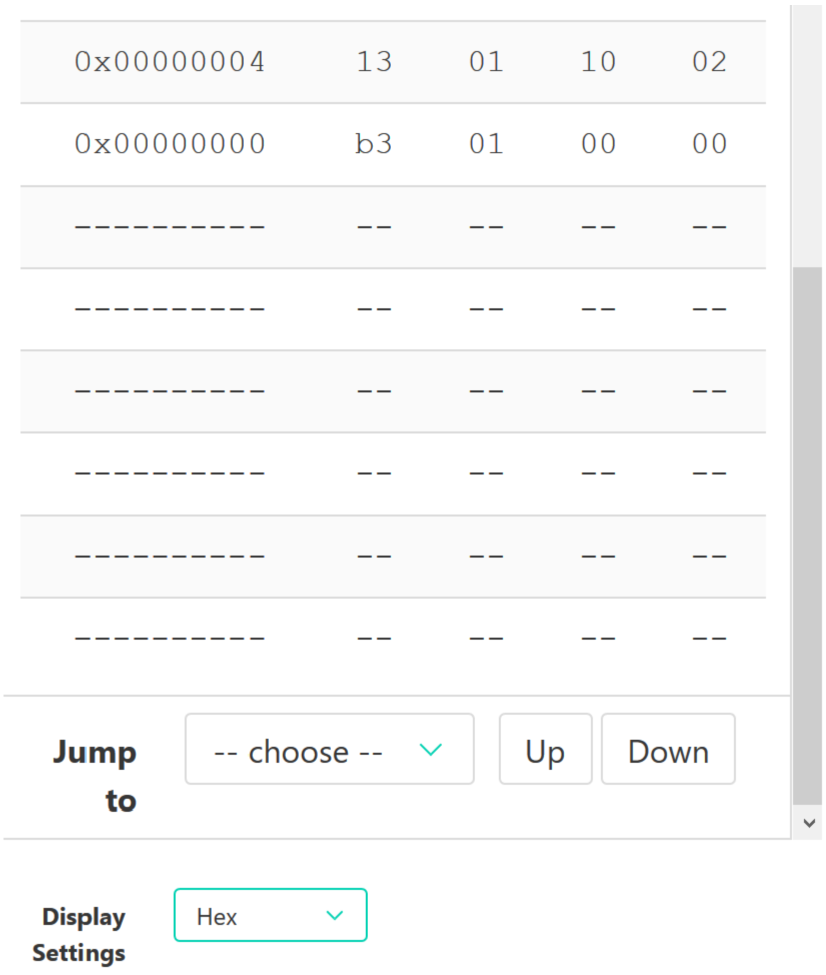
In this task we will learn the use of RISC-V arithmetic instrucitons.

### 4.1 Important Note!!

It is important to note that RISC-V instructions assume 64-bit operand sizes by default. This particular simulator does not support 64-bit words but rather use 32-bit operands. The 32-bit instructions contain "w" at the end of their names, i.e. instead of **add** you will have to write **addw**.

You can find a summary of RISC-V instruction set in the RISC-V green card availabel here: `http://inst.eecs.berkeley.edu/~cs61c/fa17/img/riscvcard.pdf`

Assuming that the variables a, b, c, d, e, are associated with the registers x27-31 respectively:



```
0x00000004    13    01    10    02
0x00000000    b3    01    00    00
----------    --    --    --    --
----------    --    --    --    --
----------    --    --    --    --
----------    --    --    --    --
----------    --    --    --    --
----------    --    --    --    --
```

Figure 4: The bottom of the memory tab

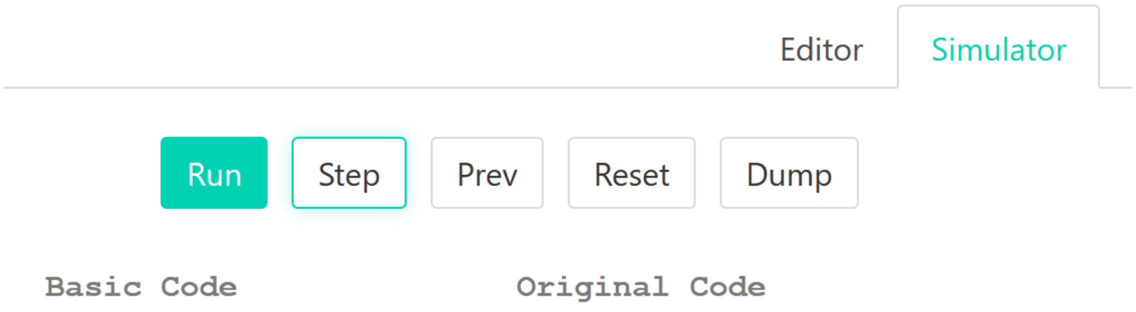


Figure 5: The execution buttons

### 4.2 Task 1a:

Covert following code in Listing ?? into its equivalent RISC-V assembly:

```
b = 0+0;
a = b + 32;
c = a + b;
d = c - 5;
```

```c
e = a + b + c + d;
```

Listing 1: task 1a.c

### 4.3 Task 1b:

Covert following code in Listing 2 into its equivalent RISC-V assembly:

```c
int a = 5;
int b = 33;
int c = 55;
int d = 6;
int e = (((a-d)+(b-a))+d)-c;
```

Listing 2: task 1b.c

## 5 Task 2: Using memory instructions

RISC-V provides you a variety of instructions to load data from RAM into the registers as well as to store data from the registers into the memory. You will choose an instruction based on the type (whether it is signed or unsigned) and the size (whether it is a char, short, int, or long) of your data. See the RISC-V green card for a summary of memory instructions.

Initialize the register x10 and x11 with values 0x78786464, 0xA8A8B9B9, respectively manually.

### 5.1 Task 2a

Write the RISC-V assembly code for each item below. Try guessing the result in each destination before executing the instruction and corroborate it after execution:

- Store x10 as unsigned integer at address 0x100.
- Store x11 as unsigned integer at address 0x1F0.
- Load an unsigned short integer (two bytes) from address 0x100 in x12.
- Load a short integer from address 0x1F0 in register x13.
- Load a singed character from address 0x1F0 in register 0x13.

### 5.2 Task 2b

Assume there are three character arrays a, b, and c located at addresses 0x100, 0x200, 0x300 respectively.

```c
for(int i=0; i<4; i++)
  c[i]=a[i]+b[i];
```

Listing 3: task 1b.c

- write equivalent RISC-V code Listing 3. You have not yet studied loops but the above code is manageable without loop instructions!
- Repeat the same exercise but this time assume that A, B, and C are integer arrays instead of character arrays.
- Repeat the same exercise but this time assume that A is a character array, B is a short array, and C is an unsigned integer array.

## 6 Rubric

StudnetID ________________ Student Name ________________

| Rubric | | | |
|---|---|---|---|
| Task No. | LR2 Code/Design (50) | LR5 Result (20) | LR7 Viva (30) |
| Task 1a | 10 | 05 | 05 |
| Task 1b | 15 | 05 | 10 |
| Task 2a | 10 | 05 | 05 |
| Task 2b | 15 | 05 | 10 |
| Total | 100 | | |

| Marks obtained | | | |
|---|---|---|---|
| Task No. | LR2 Code/Design (50) | LR5 Result (20) | LR7 Viva (30) |
| Task 1a | | | |
| Task 1b | | | |
| Task 2a | | | |
| Task 2b | | | |
| Total | | | |