Lab 5: Introduction to RISC-V Assembly

Digital Design and Computer Architecture: RISC-V Edition (Harris & Harris, Elsevier © 2021)

Objective
In this lab, you will write and simulate simple RISC-V assembly code snippets to introduce you to the RISC-V architecture and assembly language programming.

1. RISC-V Assembly Programs
Write the following RISC-V assembly programs:

Program 1: Divisible by 9?
Write a RISC-V assembly program that determines if the value in register a0 is divisible by 9. If it is, the register a0 should be 1 at the end of the program, otherwise a0 should be 0. Run, test, and debug your program using the Venus simulator. Test enough different values of a0 to convince yourself that your assembly code works. Be sure to comment each line of code (using #). **You may use only the following subset of instructions:** add, addi, sub, beq, blt, bge, bne, and j. Be sure to comment each line of code (using #). Note: assume that a0 contains only positive numbers. Name your text file **div9.s**.

Hint: think of the method of repeated subtraction to determine if a number is divisible by 9. Try your algorithm out by hand (with several non-trivial but simple examples) before coding it.

Program 2: Big-endian to little-endian conversion
Write a RISC-V assembly language code snippet that converts 8 words of memory starting at address 0x300 from big-endian to little-endian. Name your text file **big2little.s**.

> **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 big-endian at memory address 0 would have the following byte addresses and data:
> Address 0 holds 0x12
> Address 1 holds 0x34
> Address 2 holds 0xAB
> Address 3 holds 0xCD
>
> To convert this to little-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 0x12
> Address 2 holds 0x34
> Address 1 holds 0xAB
> Address 0 holds 0xCD

Program 3: Bubble sort

Write a bubble sorting algorithm that sorts a 10-element array called sortarray from smallest to largest. The array starts at address 0x400. The bubble sorting algorithm starts at element 0 and continues to the end of the array and swaps adjacent elements if one is greater than the other. For example, if sortarray[0] = 10 and sortarray[1] = 7, then it would swap the elements resulting in: sortarray[0] = 7 and sortarray[1] = 10. The algorithm then continues to compare the next two elements of the array: sortarray[1] and sortarray[2]. And so on, until it compares the last two elements of the array: sortarray[8] and sortarray[9]. The algorithm repeats this process until it processes the entire array without performing a swap. **Sketch high-level code (C code)** that uses a for loop to perform this algorithm. Name that program **bubblesort.c**. You need not test your C code, and you may use any text editor to write it. Then convert the high-level code to **RISC-V assembly**. Have your program hold any program variables (i.e., int x, etc.) in s0 – s11. Other values can be held in t0 – t6. Name your text file **bubblesort.s**. After you have written the assembly code, show the results of the code sorting the 10-element array {89, 63, -55, -107, 42, 98, -425, 203, 0, 303}. Remember that sortarray starts at address 0x400.

2. Simulation

Run and simulate each of your RISC-V assembly programs by copy/pasting your program code into the Venus simulator. To test your programs, you need to add variable / array initialization code to your programs (i.e., to your .s files). For example, in Program 1, you need to initialize a0. In Program 2, you need to initialize (i.e., put values in) memory from addresses 0x300 to 0x31c. In Program 3, you need to initialize (i.e., put initial values in) the array, sortarray.

Some hints are provided at the end of the lab about viewing memory in the Venus simulator.

What to Turn In

1. Please indicate how many hours you spent on this lab. This will be helpful for calibrating the workload for next time the course is taught.

2. Please submit each of your code screenshot: div9.s, big2little.s, and bubblesort.s.

Please indicate any bugs you found in this lab manual, or any suggestions you would have to improve the lab.

RISC-V Venus Simulator Tutorial

Below is a brief tutorial on how to use the Venus simulator.

1. Go to https://www.kvakil.me/venus/
2. Type or copy/paste RISC-V assembly code into the Venus Editor, as shown in Figure 1.
   The screenshot below shows the following RISC-V assembly code.

   addi s3, zero, -5   # s3 = -5 (0xFFFFFFFB)
   addi s4, zero 22   # s4 = 22 (0x16)
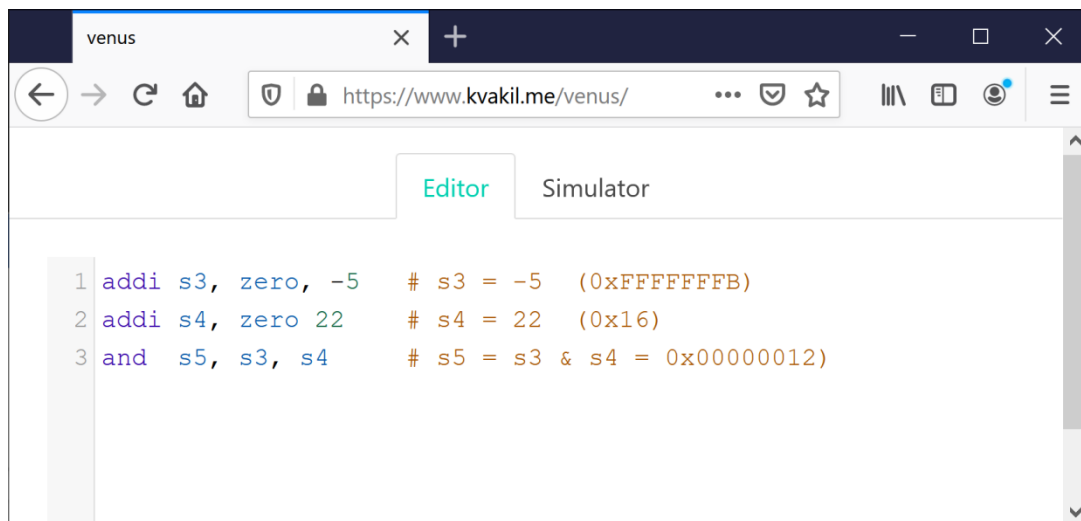   and s5, s3, s4   # s5 = s3 & s4 = 0x00000013)

Figure 1. Enter RISC-V assembly program into Editor

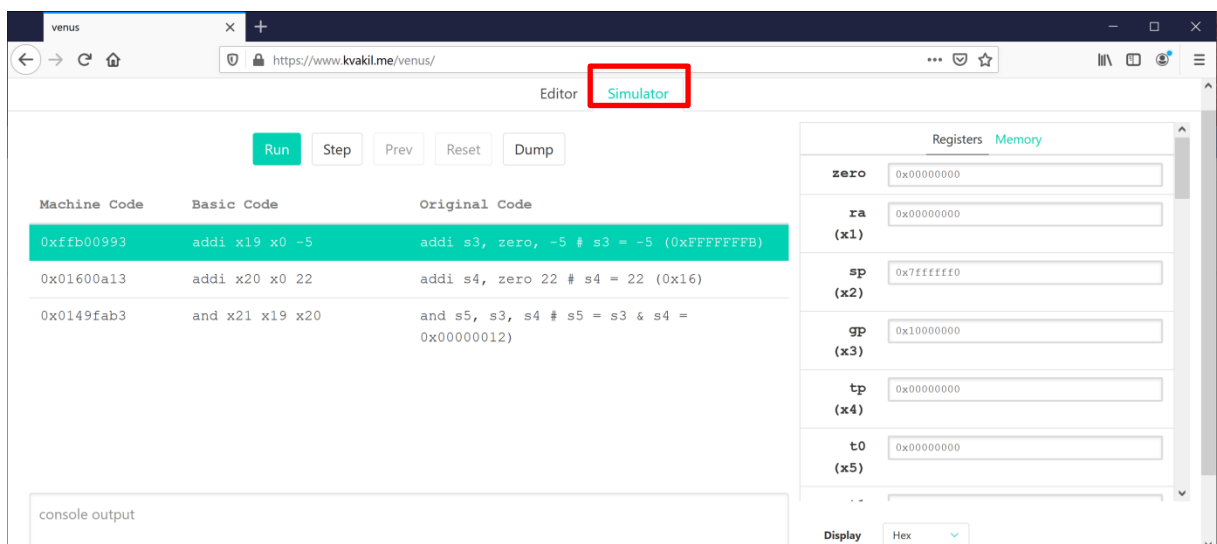3. Now click on the Simulator tab, as shown in Figure 2.

Figure 2. Venus Simulator window

Notice the following parts of the Simulator window:

1. **Code:** On the left side of the Simulator window is the code you typed in the Editor tab.
   - The **Basic Code** column shows the instruction using register numbers (instead of register names).
   - The **Original Code** column shows the instructions you typed into the Editor window.
   - The **Machine Code** column shows the instructions translated into 1's and 0's – that is, the binary encoding of the instruction.
2. **Registers/Memory:** The right side of the Simulator window shows the contents of the RISC-V registers or memory. By default, that column shows the contents of the Registers. You can also view the contents of memory by clicking on the Memory tab. At the bottom of the Registers/Memory pane, you can choose what representation to view the contents: hex (hexadecimal), decimal (2's complement), unsigned, or ASCII. We will use hexadecimal and decimal most often.

4. Now run the RISC-V assembly code snippet by clicking on the Step button at the top of the window. This will step through the code one instruction at a time. (The Run button, on the other hand, runs the entire code snippet.) Figure 3 shows the Simulator window after pressing the Step button once. At this point, the first instruction (addi s3, zero, -5) has executed, and we can see that register s3 (x19) now contains the value -5 (i.e., 0xFFFFFFFB).
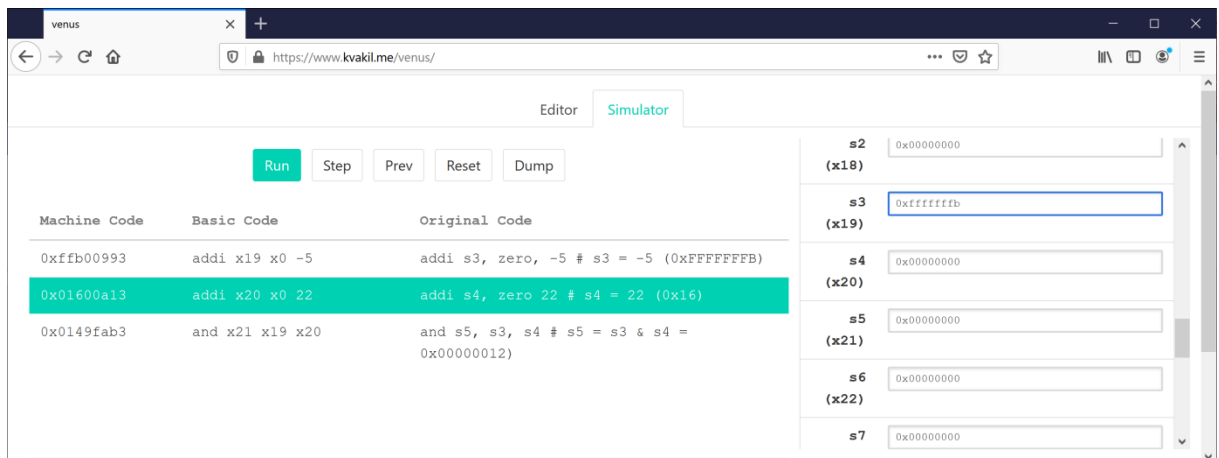

Figure 3. Step through code

5. Continue stepping through the code and viewing the contents of the registers. Figure 4 shows the contents of the registers after the code has completed executing. As shown, register s5 contains 0x12, as expected. Registers s3 and s4 contain 0xFFFFFFFB and 0x16, respectively, also as expected.
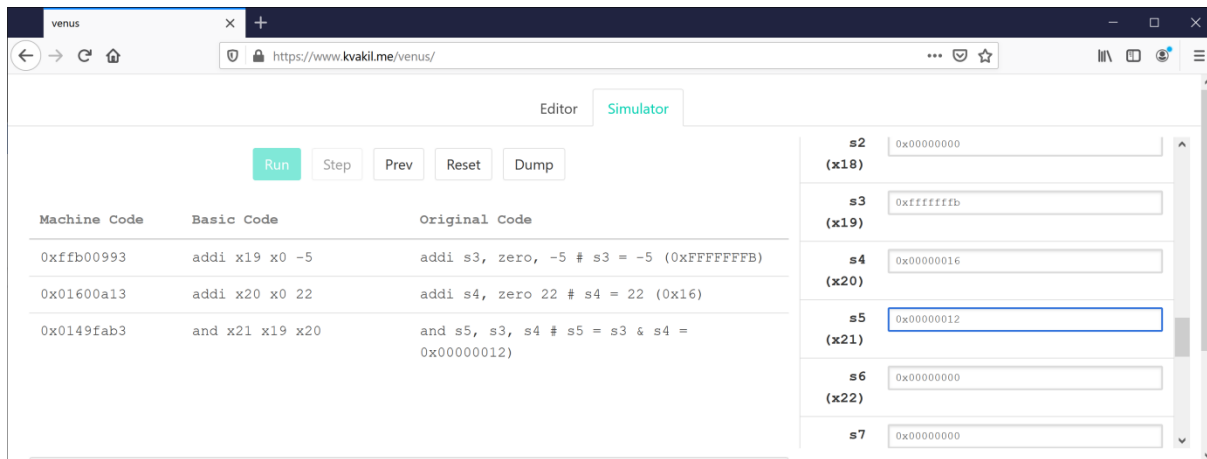
Figure 4. Contents of registers after the code has finished simulating.

You can include breakpoints in the Venus RISC-V simulator by clicking on a line of assembly code in the Simulator tab. Then click Run, and the program will run until it reaches a line set as a breakpoint (or multiple lines if you set multiple breakpoints). This makes it so that you don't have to tediously step through code that you already know works.

To view the memory in the Venus RISC-V simulator, click on Memory in the right pane and then viewing the value at address 0x300, 0x301, etc. – see figure below.

To help you understand the contents of memory correctly, run the following code in the Venus RISC-V simulator.

```
lui  s0, 0xABCDE     # s0 = 0xABCDE000
addi s0, s0, 0x123   # s0 = 0xABCDE123
sw   s0, 0x300(zero) # memory[300] = 0xABCDE123
```

What byte is stored at byte address 0x301? **Hint:** write out the word by hand and then write the byte addresses above the word for each type of memory system (big- and little-endian). Then compare this with the memory addresses of each byte in the Venus simulator.

For example, as shown in the figure below, address 0x302 holds the data value 0xcd.
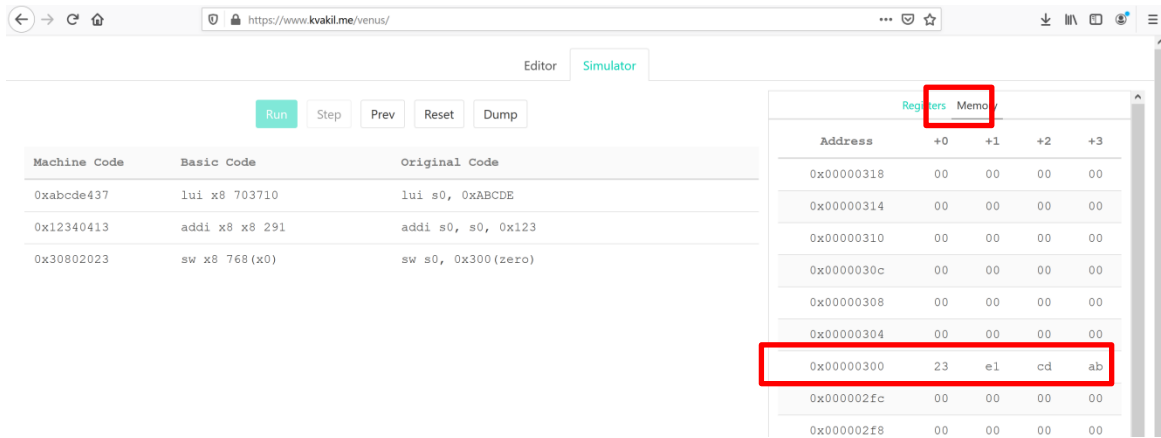
Figure 5. Viewing memory in the Venus simulator

The Venus RISC-V simulator uses little-endian memory, but it displays the bytes from left-to-right starting with the smallest address (i.e., in this case 0x300, then 0x301, then 0x302, then 0x303), so the word value looks weird.