COMP15111 Lab 1 – Introduction to Bennett and RISC-V

1.1 Aims

To introduce the RISC-V Instruction Set Architecture and Bennett, our RISC-V simulator.

Bennett provides a comprehensive environment for developing and debugging programs that run on a variety of processors. You will use Bennett over multiple course units in the first two years. In COMP15111, we will only target the RISC-V processor and a small subset of Bennett's capabilities.

1.2 Learning Outcomes

On successful completion of this exercise, you will:

- be able to use Bennett to load, assemble and run a RISC-V program
- be able to use Bennett to modify and debug a simple RISC-V program
- have written your first RISC-V program

1.3 Summary

You will first use Bennett to assemble and run a simple RISC-V program we have written for you. You will use Bennett's debugging facilities to watch what happens when the program runs and to modify the program.

1.4 Deadline

Each lab exercise has the usual deadline which is the Friday of the week after the current one. All of our deadlines are listed on the unit's landing page on Blackboard. *We encourage you to engage with this assignment long before the deadline*: a) this will allow you to use the lab sessions productively, b) you will be able to manage your time better (e.g. not having 2-3 deadlines at the same time), and c) you will be able to accommodate unexpected delays. You can submit earlier if you are ready.

Remember that you must tag your commits in the usual way to show that you completed your work by the deadline. The tag for this assignment is **Lab-1-Marking**

On Blackboard, there are specific instructions on how to use git and Bennett.

1.5 Description

A source-level-debugging system such as Bennett allows the user to:

- choose particular source files (files which contain the program).
- assemble or compile a source file to RISC-V machine code (object code).
- run the program under controlled conditions – step through instructions one at a time.
- examine areas of memory and display the contents of memory locations in various formats, including the instructions' format (disassembling).
- examine and change processor resources, such as the contents of memory and registers.

1.6 Getting started

Run Bennett

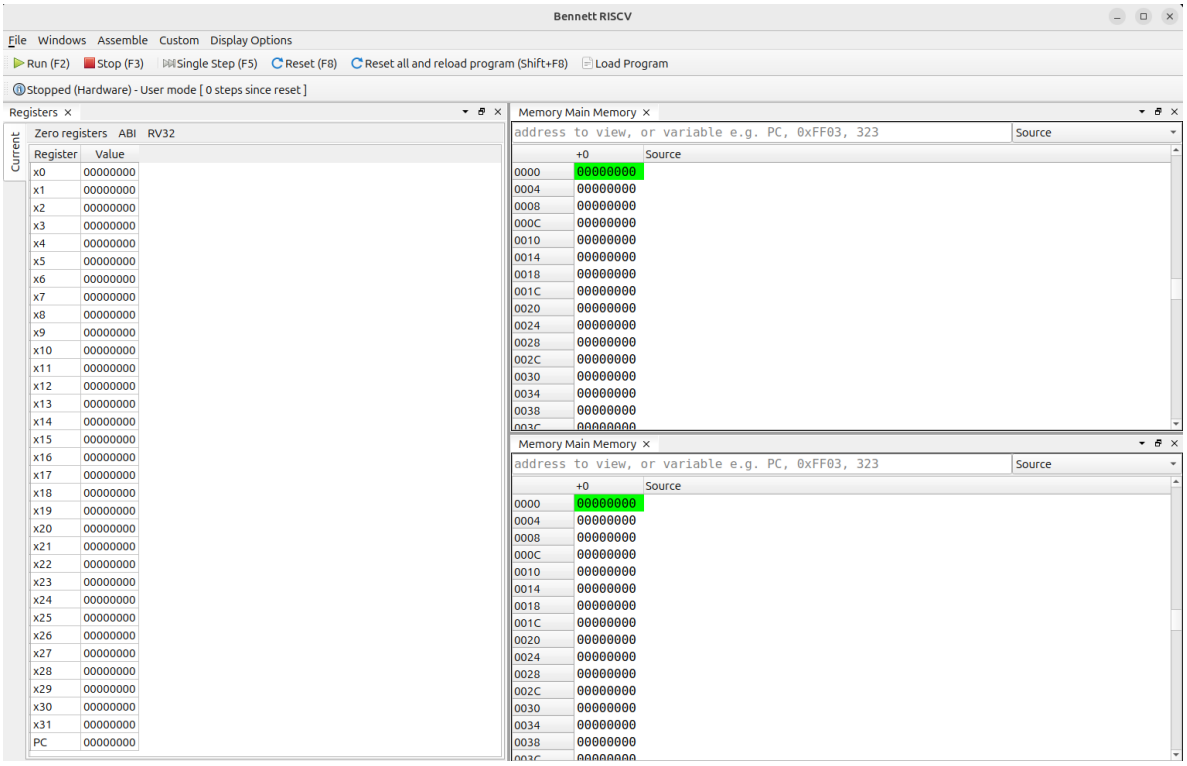
Bennett is already installed on the lab's Linux workstations. If you want to run Bennett on your personal Linux computer please follow the instructions here:

https://wiki.cs.manchester.ac.uk/engineering/index.php/Bennett

In either case, to run Bennett, execute the following command from a terminal / command line:

```
start_bennett_151
```

Bennett should now be running, and you should see something similar to the screenshot below:



Bennett Overview

Across the top of the window there are various menus and control buttons that can be used to load, assemble and run programs. We will look at these in more detail below as start using them.

On the left hand side is a register pane displaying the 32 general purpose RISC-V registers, as well as the PC. Selecting a register allows its contents to be changed. Registers are named by default with their RV32 names (x0-x31). There are three button towards the top of the pane:

1. "Zero registers" resets all registers to the value zero.
2. "ABI" changes the register names into the ABI ones (zero, ra, t0, etc). We will discuss these names in a few weeks.
3. "RV32" changes the register names back to the familiar x0-x31 names.

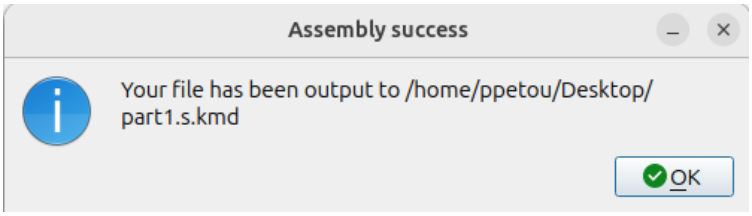
To the right of the register pane there are two main panes: the upper Memory Pane and the bottom

Memory. Each line represents one memory word. The address of the word is in the leftmost column, next is the binary value it contains, and then it's the source code mapping to that address.

Compiling (Assembling) a file:

For the first exercise, a program has already been created for you. Clone your comp15111 repository (follow the instructions on Blackboard). The program is `part1.s` in your lab1 repository.

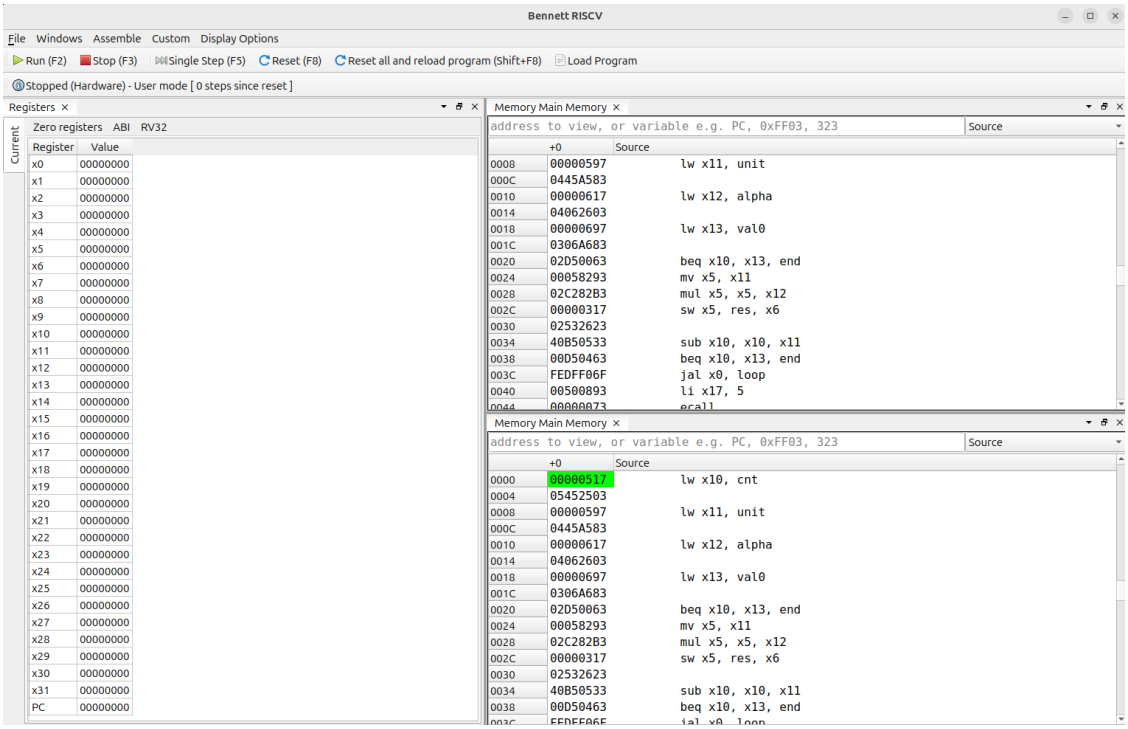
First we need to select the program and then we need to compile it. Click on the top the "Assemble → Assemble File" button and select your file to load. After you have selected the file, click the "Assemble" button in order for Bennett to assemble your program (i.e., translate it from the mnemonic form of the .s file to machine code). If everything went smooth, you should see in your terminal a success message like the one below:



As we see, Bennett assembled our program successfully and generated a .kmd file. The .kmd file contains your machine code which you can now load to Bennett and execute.

Loading a compiled/assembled file

To load a program, click on the "Load Program" button and select the .kmd file. After successfully loading the program, you should see the contents of the memory and instructions:



Running the code

Note the current values of registers x5-x6, x10-x13, and PC. To run the program press "Reset", the "Zero registers" button and then "Run". Not much appears to have happened, however some of the register values should have changed. To see why, we will step through the instructions one at a time.

Stepping through the code

First we need to be able to rerun the program. Press "Reset all and reload program" to get the program ready to run again. Clicking on the "Single-Step" button causes a single instruction to be executed. Step through the program examining the effect of each instruction on the relevant registers: in particular, note the effect on the PC. In each step, the next instruction to be executed is highlighted in the Main Memory panes. (You should stop when you reach the `ecall` instruction.)

Modifying registers

Click on a register name in the register pane: the name and value of the register will be entered into the entry boxes allowing the user to modify the contents of the register (remember to press ↵ to complete the modification). With the program halted at the **`mul`** instruction, modify the contents of registers **`x10`** or **`x11`** and then continue single stepping through the program. What's the effect on the result?

Modifying memory

The contents of memory locations can be examined (and modified). In the memory pane, the memory is visible as plain data and as RISC-V instructions. The default view lists the instructions in your source file. You can also ask Bennett to display the disassembled contents of the memory: the operation that the stored binary value encodes, whether it's an instruction or not. This is what the RISC-V processor would do if it executed that location. There is less information here than in your source code because the various labels have been replaced by simple addresses - that's all the machine needs. You may also notice some significant differences between the source and the disassembly; this occurs when pseudo-operations are substituted.

Reset the program, change the contents of the memory location for "**`cnt`**" and run the program again. To change the memory contents, click on the line to select it, edit the box next to the address, and press return. The disassembly pane contents should have changed– do you understand what has happened?

1.7 Tasks Of Lab1

Task1: Open your copy of the file "answers.txt" that exists in your repository and answer these questions.

Important: Do not edit the answers.txt file directly on the browser. Instead, use an editor on your machine and commit your file following the instructions.

All values used in any of the answers below must be in hexadecimal.

Q1) After executing the `lw x10, cnt` instruction, the value of **PC** is ... and of **x10** is … 0.5 Point

Q2) After executing the `lw x11, unit` instruction, the value of **PC** is ... and of **x11** is … 0.5 Point

Q3) After executing the `lw x12, alpha` instruction, the value of **PC** is ... and of **x12** is … 0.5 Point

*The lw instructions above are **pseudo-instructions**, so in machine code they are replaced with **two instructions**. "After executing the lw…" in this context means after executing both machine instructions corresponding to the lw.*

Try typing **PC** or **PC-8** (followed by pressing the <Enter> key, also known as "return" or "carriage return") in the box just at the top of the memory pane, between "Memory Main Window" and the memory contents. Try stepping through the program again – what happens now?

For Q4, Q5 we are asking about the 'working' registers, not **PC**):

Q4) The first time the `mul` instruction is obeyed, the value of register ... changes from ... to … 0.5 Point

Q5) The first time the `sub` instruction is obeyed, the value of register ... changes from ... to ... 0.5 Point

---

As a result of running the program from the beginning until it stops at the `ecall` instruction:

Q6) The **beq** *in the loop* (address 0x38) is executed ... times, but branches to **end** only ... times 0.5 Points

Q7) Register ... counts down from ... to … 1 Point

Q8) Which arithmetic operation does the program implement? Write down the equation in terms of the variables, val0, unit, alpha, cnt, and res 1 Point

---

Task2: Open your copy of the file "**solution.s**" in your repository and answer these questions by directly modifying the file.

You are now going to make a set of small changes to the program. Load "**solution.s**" into your favourite plain text editor (e.g., gedit, nedit, notepad) and start experimenting with the code. "solution.s" is a copy of "part1.s". We ask you to modify "solution.s" and not "part1.s" so that you are able to revisit Q1-Q8 even after you've changed the code for Q9-Q11.

Answer the following question(s) by directly editing your "solution.s". Make all your changes in the same file. Make sure your solution is still syntactically and functionally correct.

Q9) The program can be made more efficient by eliminating any use of `val0`. Can you change the code to achieve this while maintaining the original functionality? 2 Points

Q10) At least one other variable could be replaced with an immediate, *reducing the number of registers needed*. Think about what kind of values we can replace with immediates and what kind of instructions take an immediate argument? Replace one variable using immediates by modifying the code appropriately. 2 Points

Q11) Optimise the two control flow instructions *inside the loop* (.i.e., don't replace the first branch before the loop) by replacing them with one instruction. You must maintain the original functionality of the whole program. Test your output for different input variable values. 1 Points

---

Task3: *Stretch goal*. you can still get a perfect 10 without answering this. Since it requires a rewrite of the solution, we will make the changes in a different file, "**solution_stretch.s**".

Q12) Write a solution that achieves the same result while using *six or less* assembly (pseudo or real) instructions (four or less instructions excluding the two ecall-related ones). You can assume that `alpha` *and* `unit` *are constant* and are always equal to 2 and 1 respectively. But `cnt` *and* `res` *are not constant*, i.e. you have to use whatever value of cnt happens to be in memory and you have to store the result back in res. 1 Point

1.8 Completion, Feedback and Marking Process

The total mark for Lab 1 is 10 points. Your submission is composed of three files:

- You need to edit the "answers.txt" file with your answers (replace the (?) with your answers) for Q1-Q8.
- For Q9-11, you need to apply the three sets of changes to the "solution.s" file.
- For Q12, edit/rewrite "solution_stretch.s".

As soon as you have completed the exercise, you need to git commit and push to your repository your three files. Please consult the git instructions on Blackboard in case you have not done already.

We will mark all submissions within 2 weeks of the deadline. We will provide detailed feedback, which you will receive by email when we mark your submission. We will also discuss this assignment in a live session after the extended submission deadline.

File(s) to be committed to Gitlab

1. Updated version of file "answers.txt".
2. Updated version of "solution.s"
3. (Optional) Updated version of "solution_stretch.s"