CSC 3210 Computer Organization and programming

LAB 8

Debugging in VSCode/Venus

INSTRUCTIONS FOR LAB 8

Tasks to be done in this lab:

- Learn that there are several different errors that the assembler may point out, what the problems are, and how to fix them
- Learn that the assembler will not catch all errors, and that programs that assemble can still generate errors
- Learn that there is a "gdb" program for debugging executable programs
- Learn how to set a break-point, and step through a program
- Learn how to get information about registers, the PC, etc.

ASSIGNMENT FOR LAB 8

- There are a lot of things that can go wrong in programming, and assembly language is no different. This lab is about common problems, and using a debugger to help.
- There are 2 parts in this lab.
- Answer the questions that are mentioned in bold.

PART-1

First, we will look at a few possible problems that you might run into. Download the code at several_errors.S and rename it "lab8.S". Try to assemble it with Venus, and edit it to correct the errors noted by the assembler.

1. Starting off, we get the error message "AssemblerError: lab8.S:1: instruction with name ; not found ; This is an assembly language program written for the Venus simulator". The message is a bit strange, but examining line 1, you may notice that the first character is a semi-colon. If you make programs for another assembler, you might not see that this is a problem.

   For example, "NASM" uses comments that start with a semi-colon. The comments in Venus use the pound-sign. This insight allows us to figure out what is wrong and how to fix it: replace the semi-colons with pound-signs. You should do this for line 2, as well as line 6.

2. The next error is "AssemblerError: lab8.S:4: instruction with name section not found section .data". This is another problem that looks like it would be OK to someone used to programming for NASM. On top of that, your first instinct in seeing such an error is to think about how this is different from the code that we've seen before, and alter it accordingly.

   Since the data section has been after the code section, would moving the data section to the bottom solve the problem? Try it.

PART-1

3. If you tried it, you now have a very similar error message. The problem is not where the data section is, but how the sections are defined. Refer to programs that we have seen, such as "HelloWorld.S", and you should be able to figure out what to change (in two locations).

4. The next error is "AssemblerError: lab8.S:23: Instruction found outside text segment in line int1 .word 10". Note that the line number indicated will depend on how you changed the code above.

   The error is not really about having an instruction outside the text segment. The problem is trying to define our data values, and the next lines are to blame:

   ```
   int1 .word 10
   int2 .word 5
   sum  .doubleword 0
   ```

   The code looks OK, and may even be correct with some assemblers. The way to fix it is simple, to add a colon after the labels "int1", "int2", etc.

PART-1

5. The next problem is also a syntax error. The assembler says "AssemblerError: lab8.S:25: unknown assembler directive .doubleword sum: .doubleword 0". You will notice that the line above it is OK, which means that either "sum" or ".doubleword" is a problem. In some languages, "sum" may already be defined, e.g. MATLAB. However, that's not the problem here. Looking at a list of RISC-V assembler directives, you might conclude that ".doubleword" needs to be ".dword". But if you try that, you'll still get this error. One solution is simply to use ".word" instead, since this is a 32-bit value, and the simulator appears to be targeted to the 32-bit version of RISC-V. However, the simulator does allow us to use ".double". Since we do not actually use "sum" in this program, either way is OK.

6. This is not in the assembly file, however, another common sort of syntax error is to specify something that looks correct, but is not, such as the following.

   ```
   move a1, a0
   ```

   The problem is that "move" is not a mnemonic. Obviously, the programmer means "mv" here, but that is not obvious to the assembler.

7. Next, we get "AssemblerError: lab8.S:10: label to used but not defined li a1, to # Value to print". Careful inspection of the line reveals that "to" is specified, but this is not a register. Register "t0" was loaded with a value on the previous instruction, so this must be what is meant.

PART-1

8. Next, we get an error on the same line: "AssemblerError: lab8.S:10: label t0 used but not defined li a1, t0 # Value to print". A hint to the problem is the wording "label", and the question to ask here is "what is the programmer trying to accomplish here?" So far, the code loaded the word at label "int2" and put it into register t0. It makes sense that the next step is to copy the value from t0 to a1, since we use a1 with the ecall. But li means load immediate, and a register is not an immediate value. Instead, we need this command:

   ```
   mv  a1, t0
   ```

- Now let's examine something more difficult. The previous errors were all caught by the assembler, and it helpfully pointed out each line that was a problem. Now the program should run in the simulator. But it stops at the first ecall with the error "Error, invalid ecall id: 0 Exited with error code -1". Again, ask "what is the programmer trying to accomplish here?" Given the spacing, and the fact that we have a value in a1 leads us to expect that the code should print that value. Looking back at other code examples, we see that

  ```
  li   a0, 1
  ```

  corresponds to printing an integer value with Venus. Since this error occurs when the program runs, the assembler does not indicate that there will be a problem.

- The last thing is more of an observation than a problem. The ret command should stop the program, or at least this is what we would expect. In this simulator, it does something else. (See the related question below.)

PART-1

QUESTIONS:

1. **What is the effect of ret in the simulator in this program?**
2. **What should ret be replaced with?**
3. **You may have noticed that to get the program working, we had to figure out what the code was supposed to do. What could the previous programmer have done to make this clear?**
4. **When we have multiple instances of semi-colons, why would it be a bad idea to use a global find/replace operation on the code?**

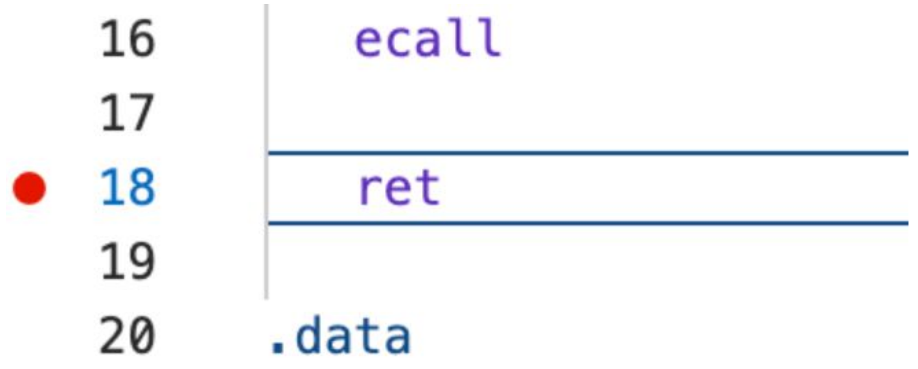
In the above, you should have altered the assembly language program to fix all of the errors noted by the assembler. Make sure to show the new version of the program, and that it runs. Now we will look at how you might debug errors that occurs at run-time.

PART-2

Note: the "gdb" program is the GNU debugger, and if you needed to debug a program on SNOWBALL, you would use that. You can run it from the command line by typing

```
gdb several_errors
```

- From the "(gdb)" prompt, you can type "quit" to exit the program. Another good command to know is "help", such as "help running". This provides documentation on commands related to running a program in the debugger. Since we are not using NASM, you do not need to use gdb. However, you should know that it exists in case you need it.
- Using Venus, we will set a breakpoint. This is a place where the execution will stop, allowing us to inspect the state of the computer. Assuming that ret is still in your program, click to the left of the line number for that line. It should look like this:



```
16      ecall
17
18      ret
19
20  .data
```

This sets a break-point at this line. You could set multiple break-points, if desired.

PART-2

- Now run the program by clicking on the "Continue" icon. The computer appears to jumo straight to that line, but only because it executes the lines before it so quickly. You can verify this by examining the output: you will see that it printed a number and newline. (You could click on the "Continue" icon again, if desired.)
- Now you can step through the code with one of the "step over" or "step into" icons. This executes the next instruction. You can repeat this to step through as many instructions as you want. Clicking on "continue" causes it to keep going until the next break-point.
- The problem with stepping through the code again and again is that you will lose track of what it is doing. We have a way to see the progress on the left, by watching "PC" change as we step through the program.
- You should inspect the contents of the registers. Click on "Integer" on the left-hand side, and you can view the registers change their values as you step through the code. Stop the program, then click on "run and debug" and watch how the register values change as you step through the code.

PART-2

QUESTIONS:

1. **Try changing the line**
   ```
   int2: .word 5
   ```
   to
   ```
   int2: .double 5
   ```
   then stepping through the program. What value is printed? What value does t0 get, and why?
2. **How can you get rid of the breakpoints?**
3. **What registers changed their values?**
4. **Did any of the registers change unexpectedly, and if so, why do think that happened?**
5. **What are the minimum and maximum values that PC has for this program?**
6. **As we advance from one instruction to the next, how much does PC's value change? Why?**
7. **Is there a direct relationship to line number and PC? Why or why not?**

IMPORTANT NOTE:

Remember that we will grade your lab report so it is vital to turn that in. The other files (your code, a text version of any log file, etc.) are to document your work in case we need more information.