ECE/CS 250: A Guide to Debugging MIPS

The QtSPIM control bar has the following controls on it:

From left to right, they are:

- Load file
- Reinitialize and load file
- Save logs
- Print logs
- Clear registers
- Reinitialize simulator
- Run/continue program
- Pause
- Stop
- Step through the program line-by-line
- Help

Common issues:

- Make sure you always use the "Reinitialize and load file" button (unless it's the first program you're loading since you've opened QtSPIM), so that residual values from previous executions don't interfere with your new execution.
- Although loading ASCII-immediates using li (e.g. `li $reg, "\n"`) is allowed in the GUI-version of QtSPIM, the command line version of SPIM (that the autograder uses) doesn't allow that. You need to use the actual ASCII value as such: `li $reg, 10`.
- If you see a message along the lines of "Instruction references undefined symbol at ... jal main" that means you forgot to load your program, or you didn't define a main in your program.
- **Make sure you're following calling conventions**, especially for $ra, and the $s registers. Even though calling conventions aren't "rules" that are required to be followed in real life (we will require you to follow them for certain questions, as stated in the HW document), they exist for a reason and following them will eliminate a lot of potential bugs. There've been cases before where students' programs would execute just fine, but never quit, because they didn't save the $ra register on the stack. These are difficult to spot and to reason about and can be avoided by simply following calling conventions.
- If you open a stack frame at the beginning of a function, **make sure you close the stack frame at every point of exit** from that function. For example, if a recursive function `foo` has two base cases and a general case, you would've opened the stack frame just once (at the beginning of

the function), but you need to close it at all the three possible exit points (the two base cases and the general case).

- Use **`la`** only for loading addresses of variables declared in the **`.data`** segment of your program. Using **`la`** like: **`la $reg2, offset($reg1)`** is technically correct, but it does something very different from what you're probably expecting. It simply loads the effective address, rather than load the actual value from that address. In other words, it executes the following operation: **`$reg2 = (value in $reg1) + offset`**, and not: **`$reg2 = value in ($reg1 + offset)`** as you might expect. You need to use **`lw`** for that.
- **If you have taken care of all these common mistakes and bugs still persist, you need to debug your program by manually stepping through the code line-by-line.**

Debugging Line-by-Line

Since MIPS assembly is a low-level language that is directly assembled into bytecode that is executed by the processor (or in our case, a simulator such as SPIM), there aren't any specialized tools such as GDB to debug assembly level programs. The good news, however, is that since QtSPIM is a simulator that shows you real time contents of the registers and the entire memory (as well as the execution stream of code), you don't need any more tools than QtSPIM to debug any MIPS program. SPIM does have a command line interface with some debugging capabilities but we recommend using QtSPIM.

Setting a breakpoint: A breakpoint is a point in your code up to which the program will execute in one go. At that point, you will gain control and you can step through it line by line. Right click on a line and then choose "Set breakpoint" or "Clear breakpoint" to set or clear a breakpoint.

Real-time register values: The two tabs on the left show you the contents of all the integer and floating-point registers (in separate tabs) in real time.

Real-time execution stream: If you step through your program line-by-line, QtSPIM will highlight the line of code (in the "Text" tab) that is about to be executed. A single line in the text tab looks like this:

```
[00400010] 00c23021  addu $6, $6, $2        ; 187: addu $a2 $a2 $v0
```

It is interpreted as follows:

- **`[00400010]`** - The address (in memory) of the current instruction is **`0x00400010`**, in hexadecimal notation**.** This is also the value of the PC (program counter).
- **`00c23021`** - The 32-bit representation of the assembly instruction, written in hexadecimal notation.
- **`addu $6, $6, $2`** - The raw interpreted assembly command that will be executed.
- **`187:`** The line number in the source file that contains this command
- **`addu $a2 $a2 $v0`** - The command as written by the human in the source file.

Real-time memory values: You can inspect values across the entire address space in real time by clicking on the "Data" tab (next to the "Text" tab). A single line in the data tab looks like this:

```
[90000180]    90000024  90000033  9000003b  90000043   $ . . . 3 . . . ; . . . C . . .
```

It is interpreted as follows:

- **[90000180]** - The address (in hexadecimal notation) of the first byte being displayed on this line. Every line in this screen displays 16 bytes of information. (Which means the next line will have the address **[90000190]** because that's **90000180 + 10,** and 10 is 16 in hexadecimal.)
- **90000024 90000033 9000003b 90000043** - The spaces exist for your convenience, but it basically means the 16 addresses starting at (and including) **90000180,** have the following 16 values in them: **90000024 90000033 9000003b 90000043.** This string is also in hexadecimal, which means every pair of digits in this string corresponds to a byte of information. As you can see, there are 16 bytes of information on this line.
- **IMPORTANT:** Note the endianness of the values described above. In the first chunk containing the first 4 bytes (**90000024**), the first address has the byte 0x24, the second and third addresses have bytes 0x00, and the fourth address has byte 0x90.
- **$ . . . 3 . . . ; . . . C . . .** - Any information following the 16 bytes of information described in the previous bullet is just the same information but printed out in a human readable form. In other words, if any of the 16 bytes correspond to an ASCII value, that ASCII value is printed out in this section. If the value contained in a byte doesn't correspond to a known, displayable, standard ASCII character it's displayed as a dot.