ECE 243S - Computer Organization
January 2026
Lab 1

Introduction to Assembly Language Programming
Using the Nios V Processor on the CPUlator Simulator
and the DE1-SoC Hardware

Learning Objectives

The goal of this lab is to introduce you to the basic concepts of Assembly Language programming of a processor, and to become familiar with two sets of software tools for doing the preparation and for running programs on the physical hardware. Assembly Language programming shows you what a computer is actually doing when it executes software that you have written, compiled and run. We will use the Nios V processor (which is Altera's version of the widely-used RISC V processor). This processor part of the *DE1-SoC Computer* that you will program into the FPGA on the DE1-SoC board in the physical lab. This computer system includes the processor, a memory for holding programs and data, and various input/output devices. It is described in the companion document provided with this lab, *DE1-SoC Computer System with Nios V*, which is provided along with this Lab.

In lab period itself, you will learn how to program this full computer system into the DE1-SoC board using the Windows computer that sits next to the DE1-SoC board in the Bahen lab rooms. You will do this using a command-line interface (a Powershell in Windows). You will also learn to compile your assembly language programs on the Windows computer, and to download the compiled program into the memory of the computer system. Using GDB (which you learned about in ECE 244) you will learn how to debug your software using single stepping and breakpoints, on the physical hardware system.

As a key part of the preparation, you will learn how to use an online simulator for this same system - called **CPULator** - which allows you to write and debug programs for this system in a single (very capable!) web page. An overview of the Nios V processor can be found in the tutorial document *Introduction to the Nios V Soft Processor*, which is also provided with this lab.

What To Submit

You should hand in the following files, to the Quercus assignment for Lab 1, by the end of your lab period. However, note that the grading takes place in-person, with your TA.

- Your group's answers to the questions 1, 2 and 3 in Part I, in a PDF file named `teamq.pdf`.
- Your program from Part III, labeled `part3.s`.

Part I - Choose and Get to Know Your Partner

All the labs and the project in this course are done together with a partner, and you'll be working closely together.

You should choose a partner who is scheduled into the same lab period as you. They do not have to be the identical lab section, just scheduled at the same time. Those time periods are:

1. Wednesdays 9am-12pm (PRA0105 & PRA0106)

2. Wednesdays 3pm-6pm (PRA0107 & PRA0108)
3. Thursdays 3pm-6pm (PRA0101 & PRA0102)
4. Fridays 9am-12pm (PRA0103 & PRA0104)

Each partner should consider and answer the questions below. Write up your answers (1 paragraph each per question - no particular word limit, but keep it short) in a file named `teamq.pdf`. Send your answers to your partner before the lab, and discuss them in a brief 5 minute conversation. That conversation should contemplate how compatible you and your partner are, with respect to these questions. If you have very different habits, discuss how you might deal with those differences. Each partner should submit their document to the Quercus Lab 1 assignment. Each partner should bring these documents to your lab period, and be prepared to discuss them with your TA.

1. How far in advance do you like to have finished your preparation for a lab? The night before, two days before, a week or more?
2. How do you like to interact with a partner - in person, online, or some mixture of the two?
3. What is your personal approach to resolving disagreements - for example, do your prefer to raise issues in person, or by email/message? Are you unlikely to bring up issues because you don't like conflict, or do you like to discuss issues as soon as they arise, or something in between?
4. Describe whether or not you and your partner are compatible, having discussed your alignment (or lack thereof) on these questions. If you are aligned, state how. For items that you are not aligned on, describe how you and your partner plan to deal with the differences.

Part II - Introduction to CPUlator - Preparation

CPUlator is a web-based software tool that simulates the behavior of the *DE1-SoC Computer*, including the Nios V processor, memory, and a number of Input/Output devices found on the board. You'll be familiar with some of those devices - the LEDs, the switches, the buttons - from your work in ECE 241 last term, as this is the same hardware! (The difference is that these devices will be controlled through software that is connected to that hardware - a key learning outcome in this course!) You will find that CPUlator is an excellent tool for developing and debugging Nios V programs on your home computer—it is easy to use and does not require a DE1-SoC board.

CPUlator is also a full-featured software development environment that allows you to compile (or 'assemble' if the program is in assembly language) software code for the Nios V processor, in both assembly language and the C language. It was written by a former TA in this course – Dr. Henry Wong, who maintains CPUlator, as a volunteer effort on his own personal time. Thank you Henry!!

In this part of the Lab, as preparation, you will explore some features of *CPUlator* by working with a very simple Nios V assembly language program. Consider the program given in Figure 1, which places two numbers into two registers, and then adds them to and puts the result into a third register.

Make sure that you understand the program in Figure 1 and the meaning of each instruction in it. Note the extensive use of comments in the program. You should always include meaningful comments in programs that you will write!

Do the following:

1. Open the *CPUlator* website for Nios V DE1-Soc here: https://cpulator.01xz.net/?sys=rv32-de1soc.
2. Type the program given in Figure 1 into the window labeled "Editor", as shown in Figure 2. You do not need to include the comments.

3. Click on the `File` command near the top of the *CPUlator* window as shown in Figure 3, and then select `Save...`. This will save a copy of your text file in your Downloads folder, so that you can re-use it later. (You can use the `Load...` under File to load a file into the editor; you should rename any file that is downloaded to an appropriate name).

4. You can make changes to your code in the Editor pane if needed by simply selecting text with your mouse and making edits using your keyboard.

```
/* Program to add the numbers 2 and 3 (placed in temporary registers 0 and 1)
and put the result into saved register 0  */

.global _start
_start:

li t0, 2 /* t0 <- 2 */
li t1, 3 /* t1 <- 3 */

add s0, t0, t1 /* s0 <- t0 + t1 */

done: j done
```

Figure 1: Assembly-language program to set and add two numbers

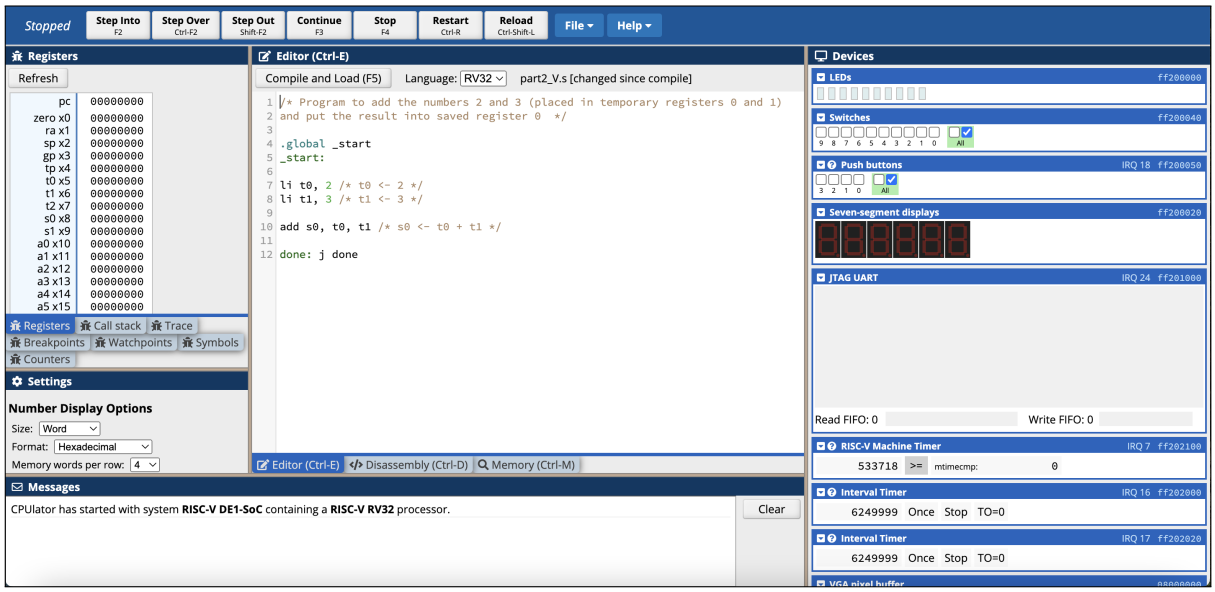

Figure 2: The `Edit Pane` in *CPUlator*.

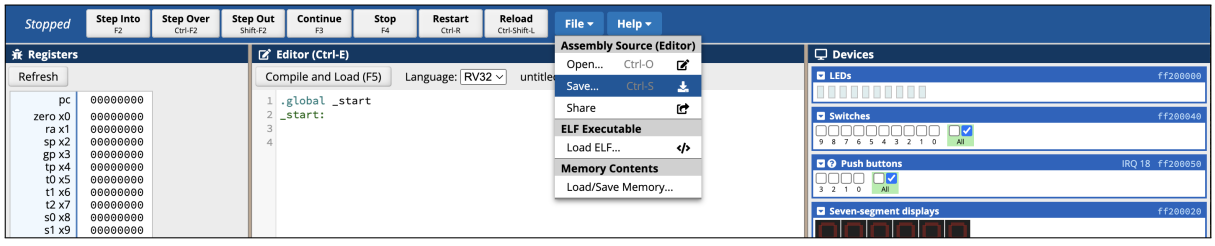

Figure 3: The `File` menu in *CPUlator*.

5. Click on the `Compile and Load` button, as shown in Figure 4, to *assemble* your program (into the numerical codes that the processor actually executes, also known as the *machine* or *object* code) and *load* it into the *memory* of the simulated computer system. You should see the message displayed in Figure 5, in the `Messages` pane, which reports a successful compilation result. If not, then you may have accidentally introduced an error in the program code; fix any such errors and recompile.

6. Once the compilation is successful, the *CPUlator* window automatically displays the `Disassembly` pane, shown in Figure 6, replacing the editor pane. (You can go back and forth from the editor pane and the disassembly pane by clicking on the tabs displayed at the bottom of the panes).

   The `Disassembly` pane lets you see the machine code for the program and gives the address in the *memory* of each machine-code word. Notice that it shows each instruction in the program twice: once using the original source code and a second time using the actual instruction found by *disassembling* (i.e. reversing the assembly process) the machine code. This is done because the *implementation* of an instruction may differ, in some cases, from the *specification* of that instruction in the source code. Examples where such differences happen will be shown in class.

   Note: you can change the way that code is displayed in the CPUlator by using its `Settings` menu, which is on the left hand side of the CPUlator window (for example, changing `Disassembly Options` from *Some* to *None* will ensure that each instruction is displayed only once).

   There is another thing to notice from Figure 6: under the column `OpCode`, notice the number to the right of the instruction `li t0, 2`, which is the number `00200293`. This number is in the hexadecimal base (base 16), and is the numeric machine code of the assembly instruction `li t0, 2`. As mentioned in class, the code that is actually executed by the processor are these numbers. That is, the process of compling/assembly converts the assembly language instructions into the these numbers. Later in the course you will learn more on how this translation happens, and how a processor interprets these numbers to know what to 'do' or 'execute.'

   Finally, the codes for these instructions exist at memory locations in the computer system's memory. For example, the instruction `li t1, 3` resides at memory location `0x0000004`. Notice also that the 32-bit instruction takes up four memory locations, because each byte (8 bits) has a separate address in the Nios V memory, as discussed in class.

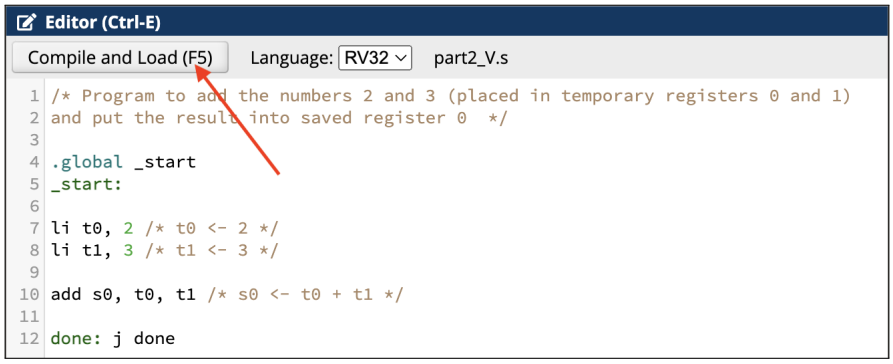

Figure 4: Compiling and loading the program.

7. Select the `Continue` command near the top of the *CPUlator* window. Doing this launches the simulated execution of the program on the Nios V processor and system that is being simulated. As illustrated in

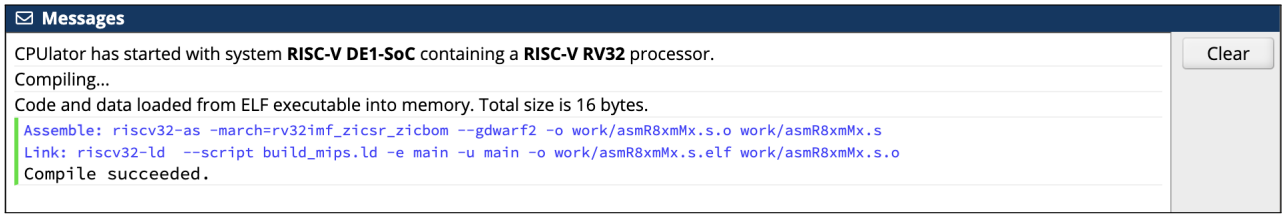

Figure 5: The `Messages` pane.

Figure 6, the program runs to the line of code labeled `done`, at memory address `0xC`, where it remains in an endless loop. Select the `Stop` command (a button at the top of the page) to stop the program's execution.

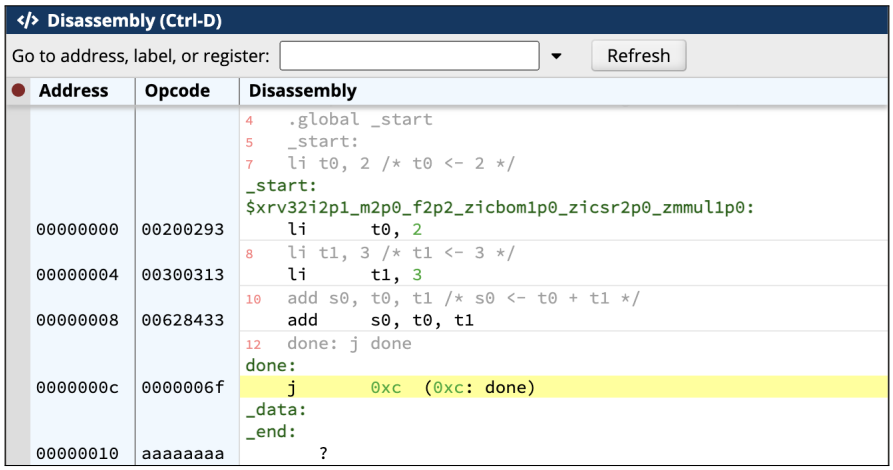

Figure 6: The Dissassembly Pane, and end of execution of program

Now take a look at *CPUlator's* `Register` pane, located on the left hand side of the web page, as illustrated in Figure 7.

You can see the values of all of the registers in Figure 7, and that they are what you would expect - t0=2, t1=3 and s0=5. One thing to note is that the numbers there are displayed in base 16 (hexadecimal, but 2, 3 and 5 are the same in decimal and hexadecimal), and you will need to become used to working in hexadecimal in this course. The numbers in CPUlator can be set to display in decimal. You can see how to do this in the pane just below the register pane.

The address of the label `done` for this program is `0x0000000C`, in hexadecimal (12 in base 10), which can be seen near the bottom of Figure 6. Also, you may notice that the `Disassembly` pane attempts to figure out the machine code at the next location, `0x00000010`, but cannot, and so it displays a question mark.

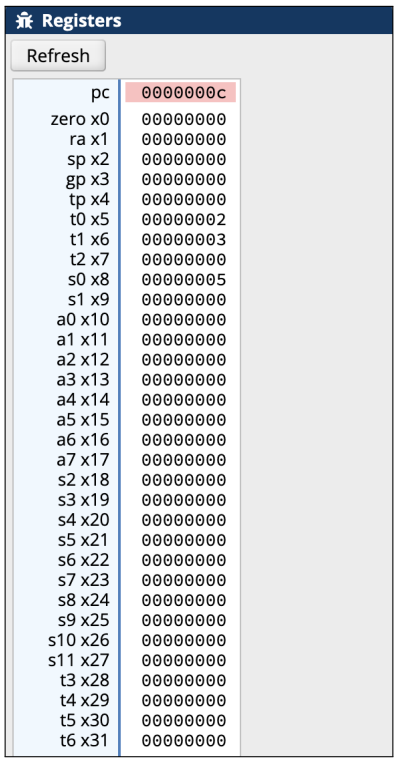

Figure 7: The `Register` pane showing contents of registers

8. Make sure that you have saved a copy of the working program as described above. Then, open a new CPUlator browswer window, and load that saved file back into the editor pane using the `File->Open` command. We are doing this as one way to set the register values back to 0. [You can also just click on the register values and set them back to 0]. Compile and Load the program again, as above. Next, *single-step* through the program by (repeatedly) selecting the `Step Into` button at the top. Observe these two things:

   (a) How the `li` instructions and `add` instrution change the register values with each step.

   (b) Also observe how the register labelled `pc` (the program counter register), which begins at value 0, advances by 4 each time you click `Step Into`. The program counter always gives the address of the next instruction to be executed.

9. Double-click on the `pc` register in the *CPUlator* and then change the value of the program counter to 0. This action has the same effect as selecting the `Restart` command.

10. Now set a breakpoint at address `0x00000008` by clicking on the gray bar to the left of this address, as illustrated in Figure 8. Select the `Continue` command to run the program again and observe the how execution stops at that location. Use `Continue` to cause the program to continue from that point. You can use this facility to monitor the progress of loops. You may already be familiar with such breakpoints from other debugging tools.

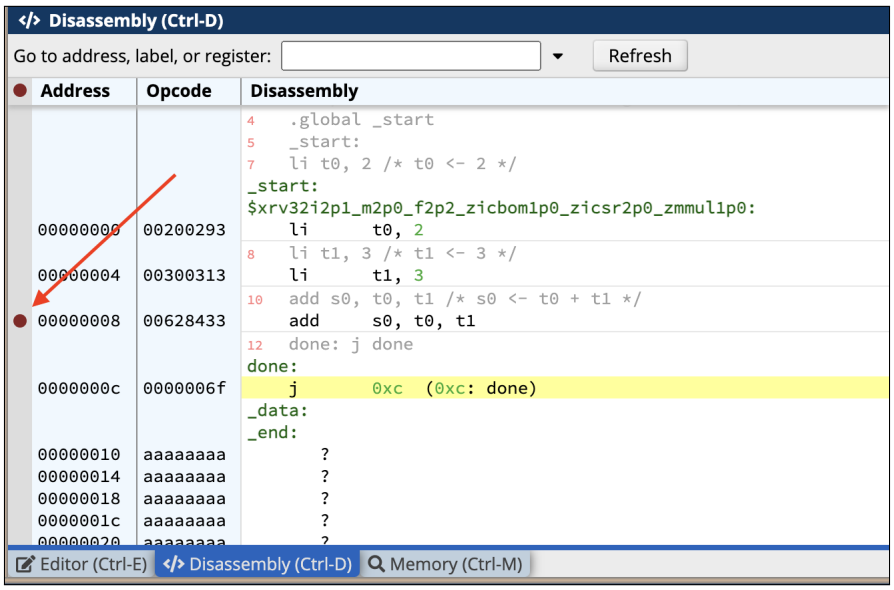

Figure 8: Setting a breakpoint.

Part III - Writing and Testing Your First Assembly Language Program

Write a Nios V Assembly language program that computes the sum of the numbers from 1 to 30, in a loop that explicitly computes the sum (i.e. don't use a formula to compute the sum, use a loop). The sum must be placed into register s1 when the program is finished, and the program should go into an infinite loop when it is done, as shown in Figure 1 at the bottom. Test your program by compiling and running it on CPUlator.

Very important: Do not write your entire program at once. Write only 2 or 3 assembly language statements, and type them into CPULator. Single-step through your program and for each statement, proceed in the following way:

- Before clicking 'step into' in the single step, ask yourself: 'What do I expect this instruction to do?' That expectation should come from the material in the lectures and/or the online text.
- **Next**, click the 'step into' button, and *look for evidence that your expectation was correct, or that it was incorrect*. That is, if you thought a register value was going to change, look at the value displayed in the register pane (shown in Figure 7). If you thought that execution would transfer to a different memory address/label, check that is true.
- If you are satisfied that your expectation was correct - the register changed (or later on, you see a memory location value change), then it is OK to move on to the next instruction, and so on through the program.
- If, however, your expectation was incorrect, and the thing you thought would happen does not happen, STOP RIGHT THERE. Try to figure out what was wrong - first by looking at whatever gave you the expectation in the first place, and checking that it was right, and then by asking a question to someone or google or chatGPT, to see what was incorrect about the expectation. It is crucial that you do this - figuring things out this way **is the job of an engineer!** Learning how to fix code/hardware that you design that isn't correct is how you become an engineer, and we expect you to do that in this course (and ECE 297/295).
- Also, you may be tempted to become sad or disappointed that your expectation was wrong. That is natural, but please remember the following: everyone learns by understanding their errors/mistakes of comprehension. That is your job here as a student, and the skills you uncover to do that learning is what will make you an engineer!

This notion of 'having an expectation' and then checking to see if it is correct, and learning what was wrong if it

wasn't is the core of the learning in this course. It may be the core learning of the entire program!!

Put comments in your code to explain it. Submit your code on Quercus, in a file named `part3.s`.

Part IV - Some Prep but Mainly to be done in lab

The previous section shows you how to run a small program in simulation on CPUlator, which is simulating a Nios V Processor in a DE1-SoC system. In this section, to be mainly done in your lab period, you'll learn how to do the same thing, except this time on actual hardware. You'll learn how to use the Windows computer in the lab to both **program** the computer system into the DE1-SoC board and FPGA, *and* to download the compiled/assembled program into the memory of that computer system.

As preparation read quickly through sections 1 and 2.1 to 2.5 of the document *Using GDB with Nios® V* document that is provided with this lab. You do not need to install anything on your own home computer unless you have a DE1-SoC board and wish to use it there. If you want to do that be sure to follow the installation instructions given here: https://github.com/fpgacademy/Tutorials/releases/download/v21.1/Using_GDB_Nios_V.pdf.

Section 2 of that document uses an example program that it is not part of the lab but will teach you how to use the monitor program better. You can find that code in the files given with this lab. In the lab period, login into the digital systems lab Windows computer that you would have used in ECE 241. Follow this tutorial, and show your TA that you've successfully completed it. The TA will ask you some questions about this experience.

Part V - in lab hardware testing of Your Program From Part III

Take your program from part III, and add the following three lines of code just after the computation is complete:

```
.equ LEDs, 0xFF200000
la s2, LEDs
sw s1,(s2)
```

This code will display the result of the computation, in binary, on the LEDs on the board.

Run your program from part III on the DE1-SoC system that you learned how to use in Part IV. Does it work? If not, debug it!

Show your TA that you have this program working.