

ECE 243S - Computer Organization
January 2024
Lab 1

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

Due date/time: During your scheduled lab period in the week of January 15th, 2024. If you have a valid reason for missing a lab, you must petition it using the online petition portal. If your petition is validated, then your TA will arrange a subsequent time for your work to be evaluated.

Learning Objectives

The goal of this lab is to introduce you to the basic concepts of Assembly Language programming of a processor; this shows you what a computer is actually doing when it executes software that you have created. We will make use of the Nios II processor which is part of the *DE1-SoC Computer*. This computer system is designed for the DE1-SoC board and includes the processor, memory for holding programs and data, and various I/O ports, and is described in the companion document provided with this lab, *DE1-SoC Computer System with Nios II*, which is provided on Quercus along with Lab 1. You will learn how to use the online simulator, **CPULator**, to work on programs for this system, and the in-lab hardware itself, using the **Monitor Program**.

An overview of the Nios II processor can be found in the tutorial document *Introduction to the Nios II Soft Processor*, which is also provided with Lab 1.

What To Submit

You should hand in the following files prior to 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. Mondays 9am-12pm (PRA0103 & PRA0104)
2. Mondays 12pm-3pm (PRA0101 & PRA0102)
3. Mondays 3-6pm (PRA0107 & PRA0108)
4. Tuesdays 9am-12pm (PRA0105 & PRA0106)

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 - particularly to see if there how compatible you are. 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, or more?
2. How do you like to interact with a partner - in person, online, or a mixture?
3. What is your personal approach to resolving disagreements - for example, do you 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?

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 II 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. You will find that the CPULATOR is an excellent tool for developing and debugging Nios II 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 (really, the correct term is 'assemble') and debug software code for the Nios II processor, in both assembly language and the C language. It was written by a former TA in this course – Dr. Henry Wong, who maintains, 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 II 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* web-site for Nios II DE1-Soc here: <https://cpulator.01xz.net/?sys=nios-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 register 8 and 9)
and put the result into register r10 */

```

```

.global _start
_start:

    movi r8,2      /* r8 <- 2 */
    movi r9,3      /* r9 <- 3 */

    add r10, r8, r9 /* r10 <- r8 + r9 */

done: br done      /* infinite loop */

```

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

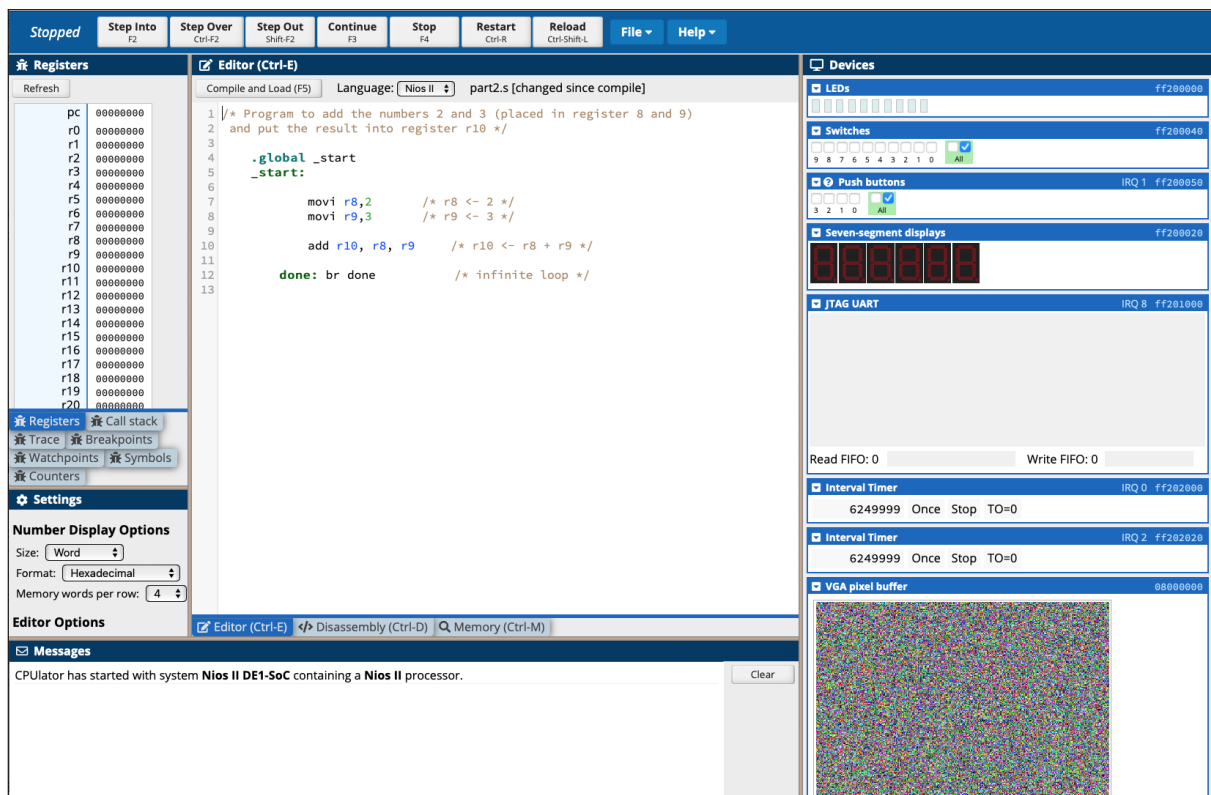


Figure 2: The Edit Pane in CPULator.

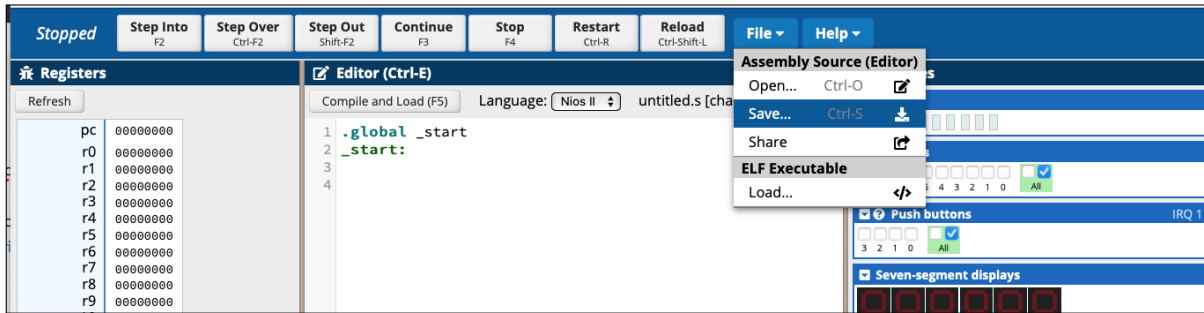


Figure 3: The File menu in CPULator.

- Click on the `Compile and Load` button, as shown in Figure 4, to *assemble* your program 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 inadvertently introduced an error in the program code; fix any such errors and recompile.
- 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 `movi r8, 2`, which is the number 02000084. This number is in the hexadecimal base (base 16), and is the numeric machine code of the assembly instruction `movi r8, 2`. As mentioned in class, the code that is actually executed by the process are these numbers. That is, the process of compiling/assembly converts the assembly language instructions into these numbers. Later in the course you will learn more on how this translation happens.

Finally, these codes for these instructions exist at memory locations in the computer system's memory. For example, the instruction `movi r9, 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 II memory, as discussed in class.

- Select the `Continue` command near the top of the CPULator window. Doing this launches the simulated execution of the program on the Nios II processor and system that is being simulated. As illustrated in 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 to halt the program's execution. Note that the largest number

Editor (Ctrl-E)

Compile and Load (F5) Language: Nios II part2.s

```

1  /* Program to add the numbers 2 and 3 (placed in register 8 and 9)
2  and put the result into register r10 */
3
4  .global _start
5  _start:
6
7      movi r8,2      /* r8 <- 2 */
8      movi r9,3      /* r9 <- 3 */
9
10     add r10, r8, r9 /* r10 <- r8 + r9 */
11
12  done: br done      /* infinite loop - statement 'branches to itself' */

```

Figure 4: Compiling and loading the program.

Messages

CPUlator has started with system **Nios II DE1-SoC** containing a **Nios II** processor.

Compiling...

Code and data loaded from ELF executable into memory. Total size is 16 bytes.

Assemble: nios2-elf-as --gdwarf2 -o work/asmSPUuP1.s.o work/asmSPUuP1.s

Link: nios2-elf-ld --section-start .reset=0 --script build.ld -e _start -u _start -o work/asmSPUuP1.s.elf work/asmSPUuP1.s.

Compile succeeded.

Figure 5: The Messages pane.

found in the sample list is 8 as indicated by the contents of register r0. This result is also stored in memory at the label RESULT.

</> Disassembly (Ctrl-D)

Go to address, label, or register: 00000000 Refresh

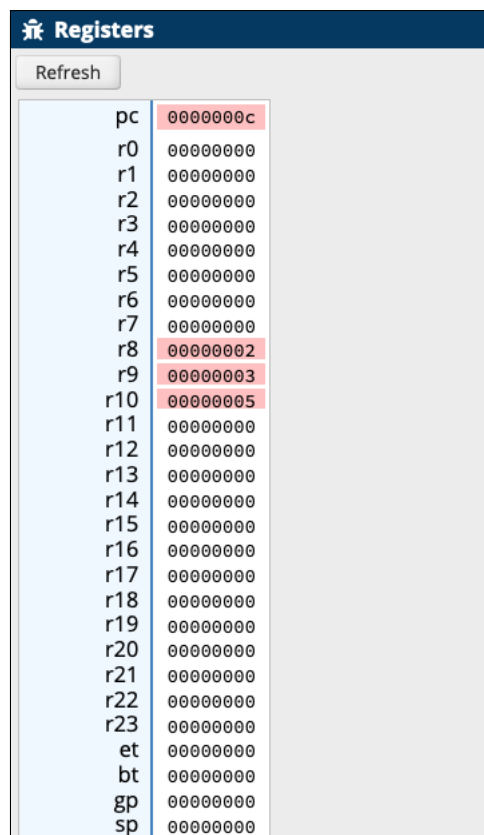
Address	Opcode	Disassembly
		5 _start:
		7 movi r8,2 /* r8 <- 2 */
		_start:
00000000	02000084	movi r8, 2
		8 movi r9,3 /* r9 <- 3 */
00000004	024000c4	movi r9, 3
		10 add r10, r8, r9 /* r10 <- r8 + r9 */
00000008	4255883a	add r10, r8, r9
		done:
0000000c	003fff06	br 0xc (0xc: done)
		_data:
		_end:
00000010	aaaaaaaa	?

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

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 - r8=2, r9=3 and r10=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 (13 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 this location, but cannot, so displays a question mark. Sometimes, more random instructions will appear even though you didn't program them.



Register	Value
pc	0000000c
r0	00000000
r1	00000000
r2	00000000
r3	00000000
r4	00000000
r5	00000000
r6	00000000
r7	00000000
r8	00000002
r9	00000003
r10	00000005
r11	00000000
r12	00000000
r13	00000000
r14	00000000
r15	00000000
r16	00000000
r17	00000000
r18	00000000
r19	00000000
r20	00000000
r21	00000000
r22	00000000
r23	00000000
et	00000000
bt	00000000
gp	00000000
sp	00000000

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 browser window, and load that saved file back into the editor pane using the `File->Load` 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] Compile and Load the program again, as above. Next, *single-step* through the program by (repeatedly) selecting the `Step Into` command. Observe these two things:
 - (a) How the `movi` instructions and `add` instruction change the register values with each step.
 - (b) Also observe how the register labelled `lpc` (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 restart the program. You can use this facility to monitor the progress of loops, to be covered later.

Address	Opcode	Disassembly
ffffffe8	fffffffe	?
ffffffec	fffffffe	?
fffffff0	fffffffe	?
fffffff4	fffffffe	?
fffffff8	fffffffe	?
fffffffc	fffffffe	?
2 and put the result into register r10 */		
4 .global _start		
5 _start:		
7 movi r8,2 /* r8 <- 2 */		
00000000	02000084	_start: movi r8, 2
8 movi r9,3 /* r9 <- 3 */		
00000004	024000c4	movi r9, 3
10 add r10, r8, r9 /* r10 <- r8 + r9 */		
00000008	4255883a	add r10, r8, r9
done:		
0000000c	003fff06	br 0xc (0xc: done)

Figure 8: Setting a breakpoint.

Part III - Writing and Testing Your First Assembly Language Program

Write a Nios II 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 `r12` 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 doesn't happen, STOP RIGHT THERE. Try to figure out what was wrong - first by looking at the whatever taught you the expectation in the first place, and checking that it was right, and then by asking a question of 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 engineering, 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 into an engineer!

This notion of 'having and expectation' and then checking if it is correct, and learning what was wrong if it wasn't is the core of the learning in this course.

Put comments in your code to explain it. Submit your code on Quercus, in a file named `part2.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 II 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 Intel Monitor program which compiles and downloads software into the hardware of the DE1-SoC.

As preparation read quickly through pages 1-22 of the document *MONITOR PROGRAM TUTORIAL FOR NIOS II* that is provided with this lab, in file `IntelDocs/Monitor_Program_Nios_II.pdf`. You do not need to install anything (Section 2 of the document) on your own home computer unless you have a DE1-SoC board and wish to use it there.

Note that the example program at the bottom of page 10 uses some concepts we have yet to cover in this course. You could, however, type it into CPULator and watch it run - its purpose is to connect the switches on the DE1-SoC to the LEDs (like you may have done in hardware in ECE 241) in a continuous loop. You can turn the switches on and off in the right hand top pane of CPULator.

In the lab period, login into the digital systems lab Windows computer that you would have used in ECE 241. Follow the full tutorial from pages 1 through 22, 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 in the following three lines of code just after the computation is complete (making sure that you don't use register `r25` in your code):

```
.equ    LEDs, 0xFF200000
movia r25, LEDs
stwio r12, (r25)
```

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.