ECE 243S - Computer Organization
January 2026
Lab 2

Accessing Memory, Loops, Conditional Branches

Due date/time: During your scheduled lab period in the week of January 19th, 2026.

Learning Objectives

The goals of this lab are to:

- continue to learn Assembly language programming and debugging skills
- understand how memory is accessed, the size of elements accessed, the addressing, and a few memory addressing modes of the load and store instructions.
- how to view memory in both CPUlator and the hardware DE1-SoC board using the `gdb` debugger.
- understand the distinction between an *executable* assembly statement, and an *assembler directive*, and to understand how the compiler/assembler translates the assembly language program and places it into memory.

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:

- The commented codes for Part 1, in the files `part1_big.s`, and `part1_lowest.s`.
- The commented code for Part 2, in the files `part2_1.s` and `part2_2.s`.

Part I

Review the code in the file `FindLargest.s`, provided with this lab, and make sure you understand how it works. The purpose of the code is to determine which of a set of numbers initialized into memory is the largest. For preparation, for this part, do the following:

- Run the code in CPUlator, using single stepping, and check that your understanding of each instruction is correct by checking what happens to registers and memory.
- Create two new versions of the the program that change the function of the program:
  1. Have the program search through 15 numbers (rather than 7), in which you choose those 15 numbers, and they should be unique - i.e. none of them are the same). The numbers should be small enough that they can be represented in 10 or fewer bits, as we will want to display them on the LEDs on the DE1-SoC. Call this code `part1_big.s` and submit it.
  2. Change the program so that it finds the lowest number, rather than the highest. Run it and debug it using the same numbers that you chose above. Call this code `part1_lowest.s` and submit it as part of your lab.

  In the lab period, load and run both of these programs on the DE1-SoC board. Before doing so, make the output appear on the 10 LEDs, as you did in Lab 1, and check that it is correct. In doing so, make sure that

you can convert base 10 numbers to base 2 and back again. Show your TA that it works on the DE1-SoC board in the lab.

Part II

You are to write a Nios V assembly language program to "look up" someone's grade in a course, given their student number. The program begins with a specific student number, and searches through a list of student numbers, which are stored in consecutive words (32 bits/4 bytes each). The last element of the list will have an entry of 0, so your program can know when to stop searching through the list.

The assembly code shown in Figure 1 gives a skeleton of the code: First, the student number being searched for is placed into register `s0`, using the `li` instruction. After that you would place your code. This is followed by anAssembler Directive (`.word`) that places the set of student numbers in a list, the first address of which is given by the label `Snumbers`. Those students' grades are placed in the another list (following `Snumbers`) beginning with the label `Grades`.

There is a direct correspondence in the order of the two lists - the first student's grade (student number `10392584`) received the grade that is the first number in the `Grades` list, `99`. The second student number received `68` and so on.

Your program should determine the 'index' of the student number, and use that to determine the grade from the `Grades` list. However, if the student number isn't in the `Snumbers` list, then the result should be a grade of '-1.' You should store that result - the grade or -1 - into the memory location given by the label `result` in the `part2_skeleton.s` code.

The student number being searched for is placed into register `s0` at the beginning of the program.

The file `part2_skeleton.s` provided with this assignment contains the code shown in Figure 1.

```
.global _start
_start:

/* s3 should contain the grade
of the person with the student number, -1 if it was not found */
# s0 has the student number being searched

    li s0, 718293


# Your code goes below this line and above iloop

iloop: j iloop

/* result should hold the grade of the student number put into s0, or
-1 if the student number isn't found */

result: .word 0

/* Snumbers is the "array," terminated by a zero of the student numbers  */
Snumbers: .word 10392584, 423195, 644370, 496059, 296800
        .word 265133, 68943, 718293, 315950, 785519
        .word 982966, 345018, 220809, 369328, 935042
        .word 467872, 887795, 681936, 0

/* Grades is the corresponding "array" with the grades, in the same order*/
Grades: .word 99, 68, 90, 85, 91, 67, 80
        .word 66, 95, 91, 91, 99, 76, 68
        .word 69, 93, 90, 72
```

Figure 1: Skeleton Code and Placing Lists into Memory

For preparation, do the following:

1. Write, run and debug your code for the above problem on CPUlator. Submit your code to this Quercus assignment in a file named `part2_1.s`.
2. Make a new version of your code in which the Grades are stored as 8-bit bytes, rather than full words. That is, change the `.word` assembler directives for the Grades list to be `.byte`, as well as the directive for the `result`. **Do not** change the initialization of the student numbers (Snumbers) in this way, they won't fit inside a byte! [even though CPUlator won't give you an error if the number is too big for a byte :-(].

   This will cause several issues that you'll need to address - relating to the fact that words take up 4 bytes, and the addressing of the words in memory versus bytes. There are several ways to deal with this, which are left to you to figure out. Submit your code to this Quercus assignment in a file named `part2_2.s`

In the laboratory, run code for both programs `part2_1.s` and `part2_2.s` on the DE1-SoC board and show that they also works there. Make the low-order ten bits of the final output displayed on the LEDs. Be prepared to answer questions about your code and the underlying concepts.