Project 3: Functions and Conventions

This project is due on **March 27, 2026** at **6 p.m.** and counts for 6.25% of your course grade.

The code and other answers you submit must be entirely your own work, and you are bound by the Honor Code. You may consult with other students about the conceptualization of the project and the meaning of the questions, but you may not look at any part of someone else's solution or collaborate with others to develop a solution. You may consult published references, provided that you appropriately cite them (e.g., with program comments), as you would in an academic paper.

There is starter code on **Github Classroom** (`https://classroom.github.com/a/HvI10nWT`), which is also where you will turn in the project by committing and pushing to the repository you create when you accept the assignment.

If you receive a permission error while accepting the assignment, check your email associated with your Github account for an invitation link to your repository.

Github Classroom also provides autograding for you. Click on the `Actions` tab in your repository, and select a commit to see the autograding report for that version of your repository. Autograding is run each time you commit.

Late days Each student is given three late days for the semester for use on homeworks or projects. If you wish to use one or more of your late days on this assignment, fill out this form: `https://forms.gle/JfQJ3jQPYiXTwKYA7`

Introduction

In this project, you will use the Nios II application binary interface to make function calls in assembly and C.

Objectives:

- Learn the importance of calling conventions / ABIs in assembly
- Use callee and caller saved registers and the stack
- Understand buffer overflows and their consequences

Part 1. Application Binary Interface

In this part, you'll write *three* ABI-compliant functions in Nios II assembly.

Part 1.1 Calling Convention

In Nios II assembly, write an ABI-compliant function that takes 2 arguments, and returns the sum of them. You should use signed arithmatic, and your function should have the type signature:
`int sum_two(int a, int b)`
(Note: this is just the function prototype; you'll write your function entirely in assembly).

Part 1.2 Saving Registers

We've defined a new mathmatical operation, $\circ$, and written an ABI-complaint function that implements it. The operation is binary: it takes two inputs, and produces one output. We won't tell you what the operation is (it's a mystery!), but we can say that $\circ$ is a commutative and associative operation. This means that $a \circ b = b \circ a$, and that $(a \circ b) \circ c = a \circ (b \circ c)$. Our function is named `op_two`, and takes two 32-bit inputs a and b, and returns a single 32-bit value $a \circ b$.

Your task is to implement an ABI-compliant `op_three` function in Nios II assembly, that inputs *three* 32-bit inputs, a, b, and c, and returns $a \circ b \circ c$.

The type signature of the given function is `int op_two(int a, int b)`, and your function's type signature should be `int op_three(int a, int b, int c)`.

Note that you don't know what operation $\circ$ is, so you'll have to rely on calling `op_two` to perform that for you! Your function should work for any operation we implement in `op_two`. At the very least, you should test it with your above `sum_two` function as the `op_two` function, but we encourage you to try it with other commutative/associative functions, such as multiplication, minimum integer, bitwise xor, etc.

Make sure you don't leave a `op_two` function in your submission! We'll define one, and use it to test your program.

Part 1.3 Recursvie functions

In Nios II assembly, write an ABI-compliant *recursive* function that calculates the Fibonacci sequence for a given input index number. The Fibonacci sequence is defined as the sum of the previous two numbers in the sequence, with the first two numbers being 0 and 1 (i.e. fibonacci(n) = fibonacci(n-1) + fibonacci(n-2), with fibonacci(0) = 0 and fibonacci(1) = 1).

Thus, for example, if given index 0, your function should return 0. Input index 1 should return 1, and input index 7 should return 13.

Your function's type signature should look like `int fibonacci(int n)`.

What to submit Three separate files, one for each function:

- `sum_two.s` — your function from Part 1.1
- `op_three.s` — your function from Part 1.2
- `fib.s` — your function from Part 1.3

None of the files should define a `_start` label, and `op_three.s` should *not* define an `op_two` function.
For example, `sum_two.s` should look something like:

```
.text
.global sum_two
sum_two:
    # implement Part 1.1 here
```

You may want to test your program (we recommend you do!!). You can do this by testing each file individually in CPUlator: add a `_start` label to call your function and verify the result. For `op_three.s`, also define an `op_two` function in the same file for testing. Remove the `_start` (and `op_two`) before submitting.

Part 2. Buffer Overflows

In this section, you'll provide different data to two programs we've written to determine your grade.

Part 2.1

In the first part, we've written a simple program in C that reads your name over the JTAG UART port, and then assigns a grade, writing the result back over the UART terminal.

The C source is provided in your repository as `part2_1.c`, and the compiled assembly is in `part2_1.s`. You can load `part2_1.s` directly into the CPUlator: `https://cpulator.01xz.net/?sys=nios-de10-lite&maxmem=67108864`.

You'll note that the code appears to assign an F- for everyone. Your job is to provide a specific name that will make your grade an A+ instead. It doesn't matter what's printed in the name field, but it should print `Your grade is:  A+`. You'll provide your input name in `part2_1.txt` in your repository.

Please note: we will grade your assignment with compiler optimizations turned *off* (in your project's Program Settings, "Additional compiler flags" should not have `-O1` in it).

Part 2.2

In the second part, we've learned better than to store grades on the stack.

Instead, we have a hardcoded assembly program that assigns the grades, and the name is provided with its length as a global variable `STUDENT_NAME`.

The assembly is provided in your repository as `part2_2.s`.

Your task is to change the values of `STUDENT_NAME` and `STUDENT_NAME_LEN` to improve your grade. Hint: there is a `print_good_grade` function in the program, but nothing calls it. How can you make sure that code gets used?

You may *NOT* change the value of any other data or code in the program, besides `STUDENT_NAME` and `STUDENT_NAME_LEN`.
You may use the CPUlator or your DE-10 for this part.

What to submit Two plaintext files:

- `part2_1.txt` — your answer to Part 2.1: the name string you provide (on its own line)
- `part2_2.txt` — your answer to Part 2.2, with the following template:

```
STUDENT_NAME_LEN:     .word   <YOUR VALUE HERE>
STUDENT_NAME:         .ascii "<YOUR VALUE HERE>"
```

Submission Checklist

1. `sum_two.s` containing your function from Part 1.1
2. `op_three.s` containing your function from Part 1.2
3. `fib.s` containing your function from Part 1.3
4. `part2_1.txt` containing your name string from Part 2.1
5. `part2_2.txt` containing your two data lines for Part 2.2

Commit and push all five files to your Github Classroom repository. Check the Actions tab for the autograding output.