Lab 3: Subroutines and C

Description

The purpose of this lab is to introduce subroutines, by writing assembly language subroutines that call and are called by C subroutines.

The purpose of a subroutine is to decompose a larger program into relatively self-contained modules that can be easily *composed* together. Since functions are often written by different people (or compilers), functions must agree on a common set of rules on how they interact: how to pass parameters from caller to callee, how to pass a return value back to the caller, and what registers a function is allowed to modify ("clobber"). This set of rules is commonly known as an Application Binary Interface (ABI). Following an ABI allows functions to call other functions without needing to know the internal details of the functions. We will be using the Nios II ABI (mainly pages 2 to 8).

In this lab, you are provided with three functions in C (that obey the Nios II ABI). Each function takes one integer parameter and prints it out (on the Monitor program's Terminal) as an octal, hexadecimal, and decimal number, respectively:

```
void printOct ( int val );
void printHex ( int val );
void printDec ( int val );
```

For example, calling `printOct(10)` results in `12` being printed on the terminal.

Preparation: Do the questions from the Preparation section below before attempting to write the code.

Yes, really.

Part 1: Calling a function (0 marks)

Write an assembly program that prints out 10 in octal, hexadecimal, and decimal. The output should be the following:

```
12
A
10
```

- Since your project will include C files, set the project type to "C Program". Add lab3_print.c and your assembly code (lab3_part1_main.s) to your project. Make a global label `main` as the entry point to your program, since the C runtime expects your program to always begin at `main`.
- There will be some startup code from the C library that executes before `main` is called. When debugging, use a breakpoint to skip past the startup code
- Hint: The assembly program should need around 10 instructions.

Part 2: Being called by a function (2 marks)

In this part, you are to write an assembly function that is both called by a C function, *and* calls other C functions. You are given a `main` function in C (lab3_main.c, see below).

The program begins executing in the C `main` function. The `main` function calls a function called `printn`, which you are required to write *in assembly*. Some numbers are passed to `printn`, which should print out each number in either octal, hexadecimal, or decimal. Use the three C functions from Part 1 to do the printing by calling them from your assembly `printn` function.

The definition of the `printn` function, in C syntax, looks as follows:

```
void printn ( char *fmt, ... );
```

The `printn` function takes one string parameter (`fmt`) followed by zero or more integer parameters. "..." in C means a variable number of arguments, similar to `printf` or `scanf`. The `printn` function's purpose is to print out a variable number of integers, printing each integer in one of three number formats (decimal, hexadecimal, and octal). The first parameter (`fmt`) specifies a string (pointer to null-terminated array of chars) that defines two things:

- how many numbers are to be printed as determined by the size of the string (in C, the end of a string is indicated by a zero `'\0'` character)
- the format in which the numbers will be printed: `'O'` - octal, `'H'`- hexadecimal, or `'D'` - decimal.

The second and subsequent parameters are the integers you want to print. You can assume that the `printn` function will always be called so that the number of integers matches the size of the string specified with `fmt`.

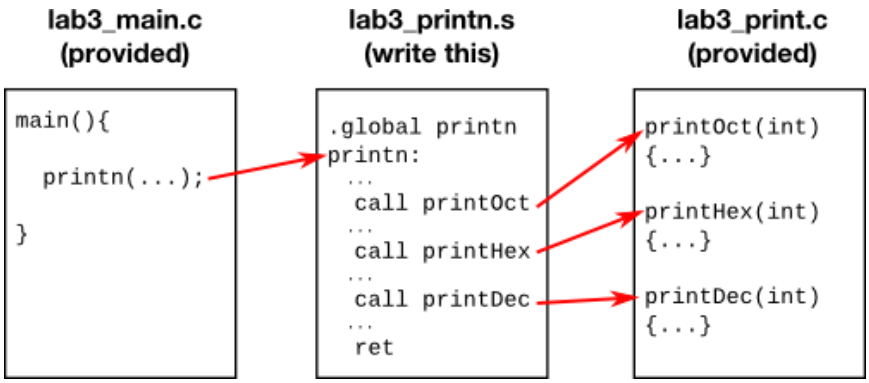
For example, let's use `printn` to print the number 10 in octal, hexadecimal, and decimal. The C function call would look like:

```
printn("OHD",10,10,10);
```

The output should then be:

```
12
A
10
```

In summary, the `main` program will be responsible for calling your `printn` function. Then your `printn` function will need to call a proper print function from one of `printOct`, `printHex`, or `printDec` with the appropriate parameter, as shown in the diagram below:



More on the Nios II Application Binary Interface

Since your assembly function will be called from C, and you will be calling functions that are written in C, you will have to determine how the registers and stack are used by the C compiler, and make your assembly code consistent with it. These conventions form part of the *Application Binary Interface* (ABI). Read the sections on "Register Usage" and "Stacks" starting on page 2 of the Nios II Processor Reference Handbook. Study the tables and diagrams there. Pay particular attention to the section on stack frames for functions with variable arguments (Pages 5 through 6).

Stack Frames

When a function is called, space is allocated on the stack for use by the function. The space is used for incoming parameters for the function, saving callee-saved registers clobbered by the function (including `ra`), local variables, and outgoing parameters for nested function calls. All of this space that is used by one instance of a function is called a **stack frame**. Figure 3 of the Nios II ABI Reference illustrates the contents of one stack frame. According to this definition, adjacent stack frames share the space for function parameters: Outgoing function parameters in one function are incoming function parameters for a function it calls.

Every time a function is called, a new stack frame is pushed on the stack (by the function prologue). The stack frame is used by the function until it is popped off the stack just before the function returns (by the function epilogue). A call stack consists of stack frames, one frame for each function call that is in progress and has yet to return.

A frame pointer (also called base pointer) is used to point to a fixed location near the top of each stack frame (In Nios II, `fp` points to the location where the previous `fp` is saved on the stack). It is used to make it easier for debuggers to get a complete call stack trace, and for dynamic allocation of variables on the stack. We won't be using this. See Call Stack on Wikipedia for more details on call stacks.

To help you understand the Nios II ABI, you can view the output of the compiler in the disassembly window. Even with an empty `printn` function, you can disassemble the program and view the compiler's output for the `main` function. A sample output is shown below:

C Function

```
#define  TEXT   "DDOOHH"

int main ( )
{
  char* text = TEXT;
  printn ( text, 16,17,18,19,20,21 );
  return 0;
}
```

Generated Assembly

```
main:
addi    sp,sp,-16      # Allocate space on stack for 4 words
stw     ra,12(sp)      # Save return address to stack

movi    r2,19
stw     r2,0(sp)       # 5th parameter on stack
movi    r2,20
stw     r2,4(sp)       # 6th parameter on stack
movi    r2,21
stw     r2,8(sp)       # 7th parameter on stack

movhi   r4,0
addi    r4,r4,1080     # r4 = address of char array (i.e., string)
movi    r5,16
movi    r6,17
movi    r7,18          # 1st through 4th arguments in r4 through r7

call    printn         # Call printn function

mov     r2,zero        # main's return value (in r2) is 0
ldw     ra,12(sp)      # Restore return address
addi    sp,sp,16       # Deallocate stack space

ret
```

Note that a string is really a char array, so using a string as the first function parameter results in the address of the string being passed in r4. Examine how the arguments are pushed on the stack and in what order.

Don't forget to declare the `printn` label as a global symbol since it must be used by another file.

Files

- Part 1 and 2: lab3_print.c: Three print functions
- Part 1: lab3_part1_main.s: Skeleton code for Part 1
- Part 2: lab3_main.c: Main function for part 2
- Part 2: lab3_printn.s: Skeleton code for Part 2

Preparation (2 marks)

1. Study the sections "Register Usage" and "Stacks" on pages 2 through 7 of the Nios II Processor Reference Handbook. Answer the following questions:
    1. Which registers are callee-saved?
    2. Which registers may change (i.e., clobbered) after calling (and executing and returning from) a function?
    3. What is the sp (or r27) register used for?
    4. Does the stack pointer point to the first unused spot, or to the last value pushed onto the stack?
    5. What is the ra (or r31) register used for? Who saves it?
    6. Which register(s) are used to pass parameters (arguments) to a function?
    7. Which register(s) are used to return the results of a function?

8. Figure 3 of the ABI reference suggests copying r4 through r7 onto the stack in the prologue of functions with variable arguments. Draw the contents of the child (callee) function's stack frame (including all incoming stack arguments) immediately after r4 through r7 are pushed onto the stack. How can doing this simplify your algorithm?

2. Assuming `printn` was called as follows:

```
printn("OOOHHHDDD",8,9,10,11,12,13,14,15,16);
```

Fill in the contents of the register and stack locations below as seen by the first instruction in your `printn` function. Also indicate where the current stack frame ends, beyond which you no longer know (nor care) what is on the stack. Assume that the string `"OOOHHHDDD"` is stored at the memory location `0x2000`.

r4
r5
r6
r7
ra

0(sp)
4(sp)
8(sp)
12(sp)
16(sp)
20(sp)
24(sp)
28(sp)
32(sp)
36(sp)
...(sp)

3. Write the assembly code for Part 1 and Part 2. In the Altera Monitor Program, create your project using program type "C program". Add `lab3_print.c` to your project for Part 1 and both `lab3_print.c` and `lab3_main.c` for Part 2. Compile the code and fix any compilation errors.

In Lab (2 marks)

Demonstrate your working programs on the DE1-SoC Computer. To run the program, load it in the Monitor, and click Continue. You can view the printed output in the Terminal window of the Monitor Program. Note: main() will also check if any registers were clobbered, indicated by a red LED being on. Make sure that at the end of main all red LEDs are off.

Package all of your source code for part 2 into code.zip (all .s and .c files), and **submit it on Blackboard.**

Quiz (1 mark)

Be prepared to answer any questions about the lab and your code, function calls, passing parameters, and usage of the stack.