# **Microprocessors (662-133) NAME: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Lab #8 – Introduction to C

Description

This lab will detail how to create a simple C program in the Altera NIOS EDS. We will also examine some of the expanded debugging options available to us in C, and how to view the assembly code produced by the compiler.

Learning objectives

1. Create the proper file structure needed by Eclipse.
2. Use expanded debugging techniques available in C.
3. Download a C program.
4. Use the expression window to debug and evaluate program operation.
5. Observe function calls.
6. Use the disassembly to view the assembly program created by the complier.

**Procedure**

Creating a project:

Follow the usual steps of creating a new project (new Nios II Application and BSP from Template), and select the .sopcinfo file and change the location as desired. Continue to use the blank project. When the project is created, create a new source file, this time use the name lab8.c instead of .s

Replace the contents of the file with those provided in lab8.c

Do a Build All

Add a breakpoint on the line "printf(%d! = %d\n",x,y);" Then go ahead and setup the debug configuration as before. HOWEVER, this time take note that you can leave the 'break on main' setting checked without any ill effects, and it saves you always having to add a breakpoint at the start of your main function.

Debug the program. You will notice that even though we had the automatic breakpoint entered at main, the program actually stopped with the first line inside the while(1) loop selected. The reason is that this is the first real **statement** in the program, the variable declarations 'int x' and 'int y' do no actual work, and our while loop is a special kind that just loops forever (there is a constant value inside the parenthesis). Therefore there is no actual work to do there either. In C, you cannot actually break on most non-statement lines, because there is no actual associated assembly instruction for the processor to stop at. Again, this is because each part of the C language describes an idea or abstraction of data or behavior, NOT a specific assembly instruction.

Lets see what we can see about the state of the processor and our program:

Observe the Debug window at the top. You will see the line Thread [1] (Suspended: Breakpoint hit.) and below that the names of three functions start(), alt\_main(), and main(). The only one that is part of our program is main(), the others are the bootloader/initialization routines. If we called a subroutine from inside of main(), it would appear above main() in this list. This list is known as the call stack. You can always work your way down from the top and see all the function calls that resulted in your program reaching the current function that is running. You can even click on one of the other functions to open it's source file and see what it looks like, and what line each function is 'stopped' on as the function call occurs. When you are done looking at the other functions just close those files, **but be sure to click back on "3 main() lab8.c".** If you fail to do this, various debugging functions won't work right.

We can observe the CPU registers and system memory as normal. However, since this is a high level language, we don't really know which registers represent which variables, so keeping track of what is happening can be tricky. Along with Registers and Memory is the Expressions window. In here you can write C expressions and see how they currently evaluate.

Select the Expressions window and click on Add new Expression.

Add the following expressions:

x

&x

y

&y

z

num

fact

x and y are two variables in our program, the & symbol means 'address of' and lets us see where the variable is located in memory (even though the current value of the variable may be in a register, the address of a variable is its 'permanent' home, and where it will be saved if the compiler needs to use the registers for something else.) The other variables (z, num, fact) appear in some functions other than main(), so they will currently evaluate as an error message (the variables are not in scope), until the program enters the function.

The memory locations you will see for &x and &y are located on the **stack** (go check the sp register if you want to!). This is because the variables are local to our main() function, once the function ends, the stack pointer is moved back, and the variables cease to exist.

However, currently the variables have some random values, as you can see. This is because they are not yet initialized. X will not be assigned a value until the IORD\_ALTERA\_AVALON\_PIO\_DATA function runs, and y will not be assigned a value until the calc() function finishes.

Step through the program. Remember the kinds of stepping:

Step over: Goes past each line, all the actions of each statement are performed completely and the next line is highlighted.

Step into: As above, but if the line includes a function call, proceed into the function.

Step return: Run until you hit a return statement and go back to the previous function on the call stack.

If you accidentally step into a printf() statement, or other system function, you can always use Step Return to get back to your program.

To start with, if you step over the x = and y = line, you will see those values appear in x and y in the expressions window.

Additionally, you can hover your mouse over the variables in the main program window and a small tooltip will appear showing you the current value of the variable.

If you step around the end of the loop, you might notice that the program will stop on the end of the code block at the } symbol. If you recall above, we said the program cannot break unless there is an associated assembly instruction at that point in the code. So what assembly instruction might appear at the end of an unconditional loop?

You can hopefully guess, but to see it, click on the Disassembly window. Indeed, it is a br statement to return to the top of the while loop. Turn on instruction-stepping mode (the i-> symbol near the single-stepping buttons). Now press step into or step over, and the code will go back to the top of the while loop.

You can now step through the loop assembly-instruction by assembly-instruction if you so choose. Remember if you press 'step into' on a call instruction, you will go to the function being called (like printf), so generally you want to be sure to step OVER those call instructions.

When you're done looking at the assembly turn off instruction-stepping mode (click the i-> again). If you fail to do so, step over and step into will appear to sit over a particular C instruction as you are actually stepping over individual assembly, even if you aren't looking at the Disassembly window.

Step around the loop again, but this time when "y = calc(x)" is selected, press Step Into. It will step to the first line of the calc() function. You will notice two things as well: A new entry appears in the call stack, 4 calc() lab8.c appears above 3 main() lab8.c, this indicates there was a function call from inside of main() to calc(). Also in the expression window, the values for x and y disappear, because they do not exist outside of the main() function. You can see that the variable z appears now, this the the name of the function parameter. The value that was passed into calc, via the line y = calc(x) is copied into the parameter variable. So in this case the value of x in main has been copied into z.

Notice that the value of z is changed after temp is calculated. Press step return and observe the value of x. It did not change because parameters in C are copied by VALUE, that is, the value of the variable x was copied into variable z. This is a ONE WAY operation that only occurs when the call is made. So changing z inside the function has no effect.

Step back into the y = calc(x) function, and this time press step into right away, to step into the factorial() function. Once again, you will notice in the expression window that z disappears, but 'num' and 'fact' now have a value. You will also see the fifth function appear at the top of the call stack. Try clicking on calc() or main() in the call stack. The highlighted line returns back to the line where the function calls appeared, and you can see in the expression window the various values appear/dissapear again. This can let you trace back, if you for instance place a breakpoint in a function, you can go back 'up' the call stack and see each call and what the values of the various variables were right as the call was made.

Click back on the factorial() function at the top of the call stack. In this function appears a for() loop. If you use Step Into, you can see that the first element (i = 2) executes before the loop runs, then the second expression is checked before each time through the loop (i <= num), and finally after each time through the loop, the last statement (i = i + 1) is executed. This ordering should be similar to what you did in assembly when construction a loop with an index variable. Initialize the counter, check the counter, and increment the counter (though for ease of use with assembly, you may have put the check at the 'end' of the loop, essentially it is the same structure).

You can feel free to explore or modify the program as you might like, but try the following:

Explain how the twoRaisedtopower function calculates its value and what isi the maximum value it can return.

Modify the expression:

int temp = factorial(z);

to include the twoRaisedToPower() function in some way. (Remember that the function needs a parameter! You can pass it z, or some expression of z (like z + 2) or just some constant (like "5").

If you make a typo or do something incorrect, the C compiler will give you an error or warning, and then highlight the line. Hover your mouse over the small yellow/red icon (appears on the left where you would place breakpoints.)

Examine the line printf("%d! plus two to the nth = %d\n",x,y);

The part between the quotes is the text that will be displayed in the console. The **%d** is a special control code that says 'this will be replaced by an integer parameter'. In this case you can see two such codes appear, and will be replaced by the value of x and y, respectively. The **\n** at the end simply causes a new line to appear (printf doesn't do this by default). Otherwise all the text will run together. Try changing the text around, but be sure to leave the two %d symbols, unless you change the parameters as well. Especially if you changed the calc() function, try to change the printed formula to match.

Try adding a printf() statement to one of the other functions. Use the %d code inside the string, and print out one of the variables in the other functions (like temp, num, or fact).