**Richter, Evan -- Lab 4 - C - "Etch-a-Sketch and Pong"**

**Prelab**

**Data types**

Go to page 76 of the C Compiler User's Guide to complete the following table. For the type, fill in data type that produces a variable of the given size. For max/min values, write in the maximum and minimum values that can be represented with the data type in that row. Two examples have been given.

| **Size** | **Signed/Unsigned** | **Type** | **Min value** | **Max value** |
| --- | --- | --- | --- | --- |
| 8-bit | unsigned | unsigned char | 0 | 255 |
| 8-bit | signed | signed char | -128 | 127 |
| 16-bit | unsigned | unsigned short | 0 | 65 535 |
| 16-bit | signed | int | -32 768 | 32 767 |
| 32-bit | unsigned | unsigned long | 0 | 4 294 967 295 |
| 32-bit | signed | signed long | -2 147 483 648 | 2 147 483 647 |
| 64-bit | unsigned | unsigned long long | 0 | 18 446 744 073 709 551 615 |
| 64-bit | signed | signed long long | -9 223 372 036 854 775 808 | 9 223 372 036 854 775 807 |

When writing embedded C code, it is always a good idea to separate your code from the architecture as much as possible because to make the code easier to change. This is why it is better to:

* use the peripheral register names in your code (e.g. P2IN) rather than their address (e.g. 0x28).
* use peripheral register field names in your code

Because space is limited on microcontrollers, it is a common practice to use variables with a range suitable for the task at hand. Unfortunately, there is no standard among C compilers between the basic data types like char, short, long and the number of bits in the underlying data representation. Furthermore, when writing and reading code, it is not readily apparent how many bits are in a short or long variable. Consequently, we will write our programs using typed definitions that provide an obvious connection between the data type and the number of bits in the representation.

Start by consulting the [Typedef Wikipedia page](http://en.wikipedia.org/wiki/Typedef). Next, fill in the following chart with the appropriate C code definitions.

| **Type** | **Meaning** | **C typedef declaration** |
| --- | --- | --- |
| int8 | unsigned 8-bit value | typedef unsigned char int8; |
| sint8 | signed 8-bit value | typedef signed char sint8; |
| int16 | unsigned 16-bit value | typedef unsigned short int16; |
| sint16 | signed 16-bit value | typedef signed short sint16; |
| int32 | unsigned 32-bit value | typedef unsigned long int32; |
| sint32 | signed 32-bit value | typedef signed long sint32; |
| int64 | unsigned 64-bit value | typedef unsigned long long int64; |
| sint64 | signed 64-bit value | typedef signed long long sint64; |

**Calling/Return Convention**

Make a project around simpleLab4.c. While the functioning of the program is not really that important, let's first take some time to understand what is going on in this program before we look at the underlying assembly language. Use CCS to step through the program and examine the a, b, c, d, e variables in main, just after the call to the function func in line 16.

| **Iteration** | **a** | **b** | **c** | **d** | **e** |
| --- | --- | --- | --- | --- | --- |
| 1st | 1 | 2 | 3 | 4 | 2 |
| 2nd | 10 | 9 | 8 | 7 | 8 |
| 3rd | 16 | 15 | 14 | 13 | 14 |
| 4th | 22 | 21 | 20 | 19 | 20 |
| 5th | 28 | 27 | 26 | 25 | 26 |

Now examine the assembly code generated by the compiler by selecting the View -> Disassembly menu item. You should see the disassembly window as a selectable tab in the subwindow where your registers are displayed. To fill in the following table with the appropriate values, you have a few tasks:

* Firstly, find the code for the function func and write down the starting and ending address in the table below.
* Next, identify which registers are used to pass the input parameters from main to the function. Write their identities below. If it is not clear which register holds which input parameter, test it out! Go ahead and change the code, so thatfunc only has one input parameter, recompile the code, and then examine the assembly.
* Finally, determine which register is used to return the value from func to main.

| **Parameter** | **Value Sought** |
| --- | --- |
| Starting address of func | xC2C8 |
| Ending address of func | xC326 |
| Register holding w | r12 |
| Register holding x | r13 |
| Register holding y | r14 |
| Register holding z | r15 |
| Register holding return value | r12 |

**Cross language build constructs**

What is the role of the extern directive in a .c file? Hint: check out the [external variable](http://en.wikipedia.org/wiki/External_variable) Wikipedia page.

*It describes a variable that has been allocated to memory in another file.*

What is the role of the .global directive in an .asm file (used in lines 28-32)? Hint: reference section 2.6.2 in the MSP 430 Assembly Language Tools v4.3 User's Guide.

.global*can either define or reference a variable that is used by multiple files. The compiler chooses where the variable is actually allocated.*

## Grading

| **Item** | **Grade** | **Points** | **Out of** | **Date** | **Due** |
| --- | --- | --- | --- | --- | --- |
| Prelab | **On-Time:** 0 ---- Check Minus ---- Check ---- Check Plus |  | 10 |  | BOC L23 |
| Required Functionality | **On-Time** --------------------------- **Late:** 1Day ---- 2Days ---- 3Days ---- 4+Days |  | 30 |  | COB L24 |
| B Functionality | **On-Time** --------------------------- **Late:** 1Day ---- 2Days ---- 3Days ---- 4+Days |  | 15 |  | COB L24 |
| A Functionality | **On-Time** --------------------------- **Late:** 1Day ---- 2Days ---- 3Days ---- 4+Days |  | 10 |  | COB L24 |
| Bonus Functionality | **On-Time** --------------------------- **Late:** 1Day ---- 2Days ---- 3Days ---- 4+Days |  | 5 each for circle, fine, and inverted |  | COB L24 |
| Lab Notebook | **On-Time:** 0 ---- Check Minus ---- Check ---- Check Plus ----- **Late:** 1Day ---- 2Days ---- 3Days ---- 4+Days |  | 35 |  | COB L25 |
| **Total** |  |  | **100** |  |  |