Programming Assignment 0 (PA0) - Welcome to CSE 30!

Due: **Wednesday night, April 11th @ 11:59pm**

Assignment Overview

The purpose of this assignment is to introduce you to ARM assembly language instructions, vim (a text editor), gdb (the GNU debugger), Git (a distributed revision control and source code management system), using make and Makefiles to compile your C and assembly language source files, and the turn-in facility to turn in your programs. This assignment is not worth very much of your grade, so make all the mistakes now! :) The source files will be provided for you (more info below in the Getting Started section). Note that the source files do NOT compile. There are a total of 8 bugs in the given files which cause 3 compilation errors, 1 compilation warning, and 5 runtime errors. **Make sure you keep track of them in your README and fix them to get the desired output.**

Grading

- **README**: 10 points - See README File section
- **Compiling**: 5 points - Using our Makefile; no warnings. If what you turn in does not compile with the given Makefile, you will receive 0 points for this assignment. **NO EXCEPTIONS!**
- **Style**: 20 points - See Style Requirements here
    - http://cseweb.ucsd.edu/~ricko/CSE30StyleGuidelines.pdf
    - **NOTE:** you must carefully read and follow the style requirements, as we will strictly follow them when we grade your style. You will not be granted a regrade for style if you didn't follow the requirements and get points deducted.
- **Correctness**: 65 points
    - Includes both abnormal and normal output, both to the correct destination (stderr vs stdout).
- **Wrong Language:** You will lose 10 points for each module in the wrong language, C vs. Assembly or vice versa.

NOTE: If what you turn in does not compile with given Makefile, you will receive 0 points for this assignment.

Tutorials and Readings

Here are some useful tutorials and readings that will help you become familiar with the vi/vim editor and Unix/Linux command line shell environment:

Unix Tutorial http://www.ee.surrey.ac.uk/Teaching/Unix/
Vim Tutorial https://blog.interlinked.org/tutorials/vim_tutorial.html
Getting Started-Git Basics http://git-scm.com/book/en/Getting-Started-Git-Basics
Git Basics Chapter http://git-scm.com/book/en/Git-Basics
Learn C https://www.learn-c.org/
C for Java Programmers http://www.cs.columbia.edu/~hgs/teaching/ap/slides/CforJavaProgrammers.ppt
Debugging with gdb http://www.delorie.com/gnu/docs/gdb/gdb_toc.html
CSE 30 Debugging Tips https://cseweb.ucsd.edu/~ricko/CSE30/Debugging.Tips.html

Getting Started

Follow these steps to acquire the starter files and prepare your Git repository.

Log in using your cs30x course specific class account. ALL of your work needs to be done on the **pi-cluster.ucsd.edu server**. Here is how you should do it:

First, ssh into the **pi-cluster** (pi-cluster nodes will only accept access from a ucsd.edu IP address). If you are on campus, and connected to campus wireless, you should be able to access the **pi-cluster** directly (whether on your own computer or one of the workstations in the labs). If coming from a non-UCSD IP address, you will need to ssh to your cs30x account on **ieng6** first, then ssh to your cs30x account on **pi-cluster**:

```
$ ssh cs30xyz@pi-cluster.ucsd.edu
```

pi-cluster.ucsd.edu is a front-end address that will land you on one of the individual Raspberry Pi nodes in the cluster. All of the cs30x pi-cluster accounts share the same home directory and file system structure as the cs30x accounts on ieng6.

Do the following once you are logged in to pi-cluster.

Configuring .vimrc:

We highly recommend setting up a `.vimrc` file to make using vim/gvim on the pi-cluster easier. You can get a `.vimrc` file created by a former tutor by typing the following command in your home directory:

```
cs30xyz@pi-cluster-001:~$ cp ~/../public/sample_vimrc ~/.vimrc
```

If you type `ls -a`, you can see there's a file called `.vimrc`. Now whenever you open a file in vim/gvim, it will first run all the commands listed in the `.vimrc`. This will make it behave a little friendlier (automatically use spaces in C files and tabs in assembly, lets you use backspace better, turns on the ruler, etc.). Feel free to play around with some of the settings and add your own if you like!

We also recommend entering the following commands for improved syntax highlighting in assembly files:

```
cs30xyz@pi-cluster-001:~$ mkdir -p ~/.vim
cs30xyz@pi-cluster-001:~$ cp -r ~/../public/vimfiles/syntax ~/.vim
```

Gathering Starter Files:

Type `mkdir pa0` to create a directory named pa0 in your current working directory.

Type `ls` to list out the files and directories in the current working directory. That is an “LS”, not “1S”; now is a good time to get familiar with a courier 1 vs. a courier l. Here is the actual output:

```
cs30xyz@pi-cluster-001:~$ mkdir pa0
cs30xyz@pi-cluster-001:~$ ls
pa0
```

`cd` (“change directory”) to directory pa0.

```
cs30xyz@pi-cluster-001:~$ cd pa0
```

Copy the starter files from the public directory.

```
$ cp ~/../public/pa0StarterFiles/* ~/pa0/
```

Back to Top

Starter files provided:		
`initArray.c`	`pa0.c`	`pa0.h`
`printInOrder.c`	`printReversed.c`	`Makefile`

You are also responsible for creating your own copy of the file `sum3.s`, which should be identical to the file pictured below. Please type up this file in vim **EXACTLY** as it appears here (remember to indent instructions with tabs!). The point of this exercise is to get you familiar with vim as well as the Raspberry Pi directives that we will use at the start of every assembly file. **You must replace all TODO's in `sum3.s`** and the other source files with the appropriate information.

```
/*
 * Filename: sum3.s
 * Author: TODO: Enter your name
 * Userid: TODO: Enter your cse30 login id
 * Description: Sum three numbers and return the result
 * Date: TODO: Enter the date you first modified this file
 * Sources of Help: TODO: List all the people, books, websites, etc. that you
 *                  used to help you write the code in this source file.
 */

@ Raspberry Pi directives
        .cpu    cortex-a53      @ Version of our Pis
        .syntax unified         @ Modern ARM syntax

        .equ    FP_OFFSET, 4    @ Offset to set fp to base of saved regs

        .global sum3            @ Specify sum3 as a global symbol

        .text                   @ Switch to Text segment
        .align 2                @ Align on evenly divisible by 4 byte address;
                                @  .align n where 2^n determines alignment
/*
 * Function Name: sum3()
 * Function Prototype: int sum3( int a, int b, int c );
 * Description: Returns the sum of the three formal parameters
 * Parameters: a - the first value to sum
 *             b - the second value to sum
 *             c - the third value to sum
 * Side Effects: None
 * Error Conditions: None
 * Return Value: The sum of the three formal parameters
 *
 * Registers used:
 *     r0 - arg 1 -- a
 *     r1 - arg 2 -- b
 *     r2 - arg 3 -- c
 *     r3 - local var -- holds the intermediate and final sum of a, b, and c
 */
sum3:
        @ Standard prologue
        push    {fp, lr}                @ Save registers: fp, lr on the stack
        add     fp, sp, FP_OFFSET       @ Set fp to base of saved registers

        @ Incoming parameters in r0, r1, r2
        add     r3, r0, r1              @ Add param1 and param2; result into r3
        add     r3, r3, r2              @ Add previous result with param3;
                                        @   result into r3

        mov     r3, r0                  @ Put return value in r0

        @ Standard epilogue
        sub     sp, fp, FP_OFFSET       @ Set sp to top of saved registers
        pop     {fp, pc}                @ Restore fp; restore lr into pc for
                                        @   return
```

Back to Top

Preparing Git Repository:

You are required to use Git with this and ALL future programming assignments. You may not appreciate Git now, but at least once a quarter a student will accidentally delete all of their files on the day the assignment is due. Git can help save you from this terrible fate! Check out the Tutorials and Readings section above for more info on Git.

(1) Setting up a local repository

Navigate to your pa0 directory and initialize a local git repository:

```
cs30xyz@pi-cluster-001:~/pa0$ git init
```

If you haven't already set your global git user info, go ahead and do that now:

```
cs30xyz@pi-cluster-001:~/pa0$ git config --global user.name "John Doe"
cs30xyz@pi-cluster-001:~/pa0$ git config --global user.email "johndoe@ucsd.edu"
```

(2) Adding and committing files

As you're developing, you can see the status of the files in your directory with the following command:

```
cs30xyz@pi-cluster-001:~/pa0$ git status
```

After you edit a file with meaningful changes*, you should add and commit it to the repository:

```
cs30xyz@pi-cluster-001:~/pa0$ git add filename
cs30xyz@pi-cluster-001:~/pa0$ git commit -m "Some message describing the changes"
```

Note: You can commit multiple files at the same time by git adding several files before calling commit:

```
cs30xyz@pi-cluster-001:~/pa0$ git add file1
cs30xyz@pi-cluster-001:~/pa0$ git add file2
cs30xyz@pi-cluster-001:~/pa0$ git add file3
cs30xyz@pi-cluster-001:~/pa0$ git commit -m "Changed things in three files"
```

NOTE: You must do a `git add` on each file you wish to commit before every git commit! It is not enough to do a `git add` once at the beginning. `git commit` will only collect files that have been `git add`'ed since the last commit.

* "Meaningful change" is a subjective term. Essentially, whenever you make a code change that results in a stable version that you want to keep track of, you should commit those changes.

There are other ways to do this, including some that make it substantially quicker for simple projects like those in this class, so it is worth looking at a few Git tutorials and/or references. From this point forward it is up to you when to add and commit to take your Git snapshots. There are other things that can be done with Git (quite a lot of things) and those are also up to your discretion.

(3) Ignoring files with .gitignore

You may notice as you're developing your program that Git really wants to keep track of `.o` files, `.swp` files, executables, etc. which you don't really need to track. You can get it to stop bugging you about them by creating a `.gitignore` file. Simply open `.gitignore` in vim/gvim:

Back to Top

```
cs30xyz@pi-cluster-001:~/pa0$ vim .gitignore
```

And add the following lines to it:

```
.gitignore
*.o
*.sw*
*~
a.out
core
```

Now when you do `git status`, those pesky files won't show up in the list of untracked files.

Editing Files:
Use the vi/vim editor to edit C and assembly source files (for this assignment and all future assignments). Make sure you read all the comments in the provided code as they will help you fix the errors you are required to fix in this assignment.

Compiling

Now you have the source files (**remember to create `sum3.s`!**) and the Makefile in your pa0 directory. You need to run `make` to compile these files. Type `make` at the prompt. This will create the executable necessary to run the program. By default, the target executable will be named `a.out`.

Again, the source files we provided do **NOT** compile. There are 3 compilation errors and 1 compilation warnings (note that one of the bugs causes 1 compilation error and 1 compilation warning). Find these bugs and fix them. Keep track of the bugs as you fix them and take note of the file that contains the error, the line number, and your fix for each error in your README file.

Here is an example output of some compilation errors:

```
cs30xyz@pi-cluster-001:~/pa0$ make
Compiling each C source file separately ...
/usr/bin/gcc -c -g -W -Wall -D__EXTENSIONS__ -std=c99 initArray.c

Compiling each C source file separately ...
/usr/bin/gcc -c -g -W -Wall -D__EXTENSIONS__ -std=c99 pa0.c
pa0.c: In function 'main':
pa0.c:47:3: error: expected ';' before '}' token
   }
   ^
pa0.c:66:3: error: expected declaration or statement at end of input
   return EXIT_SUCCESS;
   ^
Makefile:36: recipe for target 'pa0.o' failed
make: *** [pa0.o] Error 1
cs30xyz@pi-cluster-001:~/pa0$
```

Back to Top

The output specifies which file the errors are coming from. In this case, they are from pa0.c. The output also specifies where the errors are located in the file. The second line of the error message gives it away:
`pa0.c:47:3: error: expected ';' before '}' token`

Open the file and look closely at line 47 (if you type `vim pa0.c +47` at the prompt, this will open pa0.c in vim with the cursor already on line 47--this is very useful for debugging). Now that you've found the first error, go ahead and fix it, and then run `make` again. Keep fixing each of the errors/warnings one at a time, always starting with the first error from the list of compilation errors, and then recompiling after each fix (in general you should do this for all of your assignments when you run into compiler errors).

Remember to keep track of all the compilation errors (filenames and line numbers) and how you fixed them - you will need this for your README file.

Running

Once you have successfully compiled using make, type `./a.out` at the prompt to execute your code.

```
cs30xyz@pi-cluster-001:~/pa0$ ./a.out
```

What happened? It looks like the program isn't responding anymore, and could possibly be stuck in an infinite loop! Press `ctrl-C` to end execution. You should now see your command prompt again.

```
cs30xyz@pi-cluster-001:~/pa0$ ./a.out
^C
cs30xyz@pi-cluster-001:~/pa0$
```

We are going to use gdb to find out why this is happening. Run your executable in gdb by typing `gdb a.out` at the prompt. Once the debugger is loaded, set a breakpoint in main:

```
(gdb) break main
Breakpoint 1 at 0x1050c: file pa0.c, line 35.
```

Now you can run the program within gdb by typing `run` (note: the **`/* bolded comments */`** below are only there to explain to you what is going on, and should not be typed in along with the following gdb commands. If at any point you are confused by what the gdb commands are doing, type `help` in gdb followed by the confusing command, and gdb will tell you what the command does and how to use it):

```
(gdb) run /* run the program */
Starting program: /home/linux/ieng6/cs30x/cs30xyz/pa0/a.out

Breakpoint 1, main () at pa0.c:35
35        int v1 = FIRST_NUM_TO_SUM;
```

The first thing our program does is initialize local variables. Let's step through these lines of code to move on to something more interesting.

```
(gdb) next /* execute the next line of code */
36        int v2 = SECOND_NUM_TO_SUM;
(gdb) next
```

Back to Top

```
37        int v3 = THIRD_NUM_TO_SUM;
(gdb) next
42        initArray( intArray, SIZE );
```

The next thing our program does is call initArray to fill our array with values. Lets step into the initArray function and see what's going on:

```
(gdb) step /* step into the function */
initArray (array=0x7efffac8, length=15) at initArray.c:26
26        int i = 0;
```

Let's view the next few lines of code that will be executed in initArray:

```
(gdb) list /* show the next few lines of code */
21    * Side Effects: Initializes the values of the array
22    * Error Conditions: None
23    * Return Value: None
24    */
25    void initArray( int array[], int length ) {
26      int i = 0;
27
28      while ( i < length ) {
29        array[i] = ODD_MULTIPLIER * i + 1;
30      }
(gdb)
```

As we can see, initArray contains a loop responsible for assigning a value to each element in the array, but it's possible that this loop is not doing what we expect it to do. Now, follow these steps in gdb to get more information about our first runtime error:

```
(gdb) next /* execute the next line of code */
28        while ( i < length ) {
(gdb) print i  /* check the loop index value */
$1 = 0
(gdb) print length /* check the length we're comparing with */
$2 = 15
(gdb) next
29          array[i] = ODD_MULTIPLIER * i + 1;
(gdb) next
28        while ( i < length ) {
(gdb) print array[i]  /* check that the first element is set correctly */
$3 = 1
(gdb) next
29          array[i] = ODD_MULTIPLIER * i + 1;
(gdb) print i  /* check the loop index value */
$4 = 0
(gdb)
```

Back to Top

What's happening here? From our first print statement, we can see that `i = 0` when we first enter the loop. This is what we expect based on what we can see in the code. However, it also looks like `i = 0` the second time the condition check of our while loop is executed. If `i` never changes, will our loop ever end? Probably not. Let's keep using `next` to make the loop execute a few more times and then check the values in our array:

```
(gdb) next
28        while ( i < length ) {
(gdb) n  /* n also works as next */
29          array[i] = ODD_MULTIPLIER * i + 1;
(gdb)   /* we can also just hit enter to repeat the last command */
28        while ( i < length ) {
(gdb)
29          array[i] = ODD_MULTIPLIER * i + 1;
(gdb)
28        while ( i < length ) {
(gdb)
29          array[i] = ODD_MULTIPLIER * i + 1;
(gdb)
28        while ( i < length ) {
(gdb) x/15 array  /* examine the memory of our array, which has 15 elements */
0x7efffac8:     1       0       0       1996484548
0x7efffad8:     4       1994898408      0       1994897912
0x7efffae8:     2130705184      1996330904      0       0
0x7efffaf8:     66668   67396   1996188576
(gdb)
```

As we suspected, the loop doesn't appear to be moving forward and the array is not being set up properly (keep printing the value of `i` in each iteration if you're not feeling convinced). We would expect the elements of the array to be 1, 3, 5, and so on, but instead the values are mostly garbage. Type `q` to quit gdb, then fix the runtime error causing this and make a note of where it occurred.

Once you've done that, recompile and trying running your program again. You should see something like this:

```
cs30xyz@pi-cluster-001:~/pa0$ ./a.out
1
3
5
7
9
11
13
15
17
19
21
23
25
27
29
```

Back to Top

```
29
67324
32
15
5
15
0
1993982612
1995182080
.
.
.
(Garbage values continued)
.
.
.
778121006
7632239
0
Segmentation fault
cs30xyz@pi-cluster-001:~/pa0$
```

In addition to all the garbage being printed, it looks like we have a segmentation fault. We can use GDB (remember to start gdb with `gdb a.out`) to find out where this is happening:

```
(gdb) run
Starting program: /home/linux/ieng6/cs30x/cs30xyz/pa0/a.out
1
3
5
7
9
11
13
15
17
19
21
23
25
27
29

29
67324
32
15
5
```

Back to Top

```
15
0
1993982612
1995182080
.
.
.
(Garbage values continued)
.
.
.
778121006
7632239
0

Program received signal SIGSEGV, Segmentation fault.
0x000106ac in printReversed (array=0x7efffac8, length=15) at printReversed.c:27
27          printf( "%d\n", array[i] );
(gdb)
```

As we can see, gdb can tell us exactly where our segfault occurred: `at printReversed.c:27` (line 27 of printReversed.c). Note that we didn't have to set a breakpoint to find the segfault because, lucky us, the segfault itself is a "point" where our program "breaks" and stops execution (and therefore acts just like a breakpoint, so we don't need to manually set one--remember this when debugging segfaults in future assignments).

So why did this happen? Let's print out the current value of `i` and see if it's reasonable:

```
(gdb) p i  /* p also works as print */
$1 = 334
(gdb)
```

In my case, `i` was 334! This is probably outside the bounds of our array, which has a size of just 15 defined in pa0.h. Trying to access array elements outside of the array bounds can cause unexpected and problematic behaviors like segmentation faults (because you are trying to access memory that you don't have access to). Note that your output may be different due to the somewhat random nature of memory. Take a look at printReversed.c and try to figure out why this is happening (use gdb if you'd like!).

Once you've fixed this runtime error, keep testing your code to see (and fix) other runtime errors until your output looks like the following (remember to take notes on all the errors you find, where you found them, and how you fixed them):

```
cs30xyz@pi-cluster-001:~/pa0$ ./a.out
1
3
5
7
9
11
13
```

Back to Top

```
15
17
19
21
23
25
27
29

29
27
25
23
21
19
17
15
13
11
9
7
5
3
1

1
3
5
7
9
11
13
15
17
19
21
23
25
27
29

The sum of 5 + 15 + 32 = 5
```

We are almost done, but clearly the sum of 5, 15, and 32 is not 5, so we still have some work to do. Since sum3 is the function that gets called to calculate this sum, we should probably look there (and yes, we'll use our best friend, gdb).

Back to Top

Using gdb, we can set a breakpoint in sum3 and then use the command `i r`, which will list all of the integer registers and display their contents. Here is an example of using `i r` if we set a breakpoint at the beginning of sum3:

```
Breakpoint 1, sum3 () at sum3.s:42
42              add     fp, sp, FP_OFFSET   @ Set fp to base of saved registers
(gdb) i r
r0             0x5      5
r1             0xf      15
r2             0x20     32
r3             0x0      0
r4             0x0      0
r5             0x0      0
r6             0x1037c  66428
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0x76fff000       1996484608
r11            0x7efffb1c       2130705180
r12            0xa      10
sp             0x7efffab8       0x7efffab8
lr             0x105c8  67016
pc             0x106e4  0x106e4 <sum3+4>
cpsr           0x60000010       1610612752
(gdb)
```

Note that the leftmost column lists the names of the registers, and the rightmost column lists the contents of the registers, displayed in decimal (base 10). Using what you've learned about gdb, try stepping through the code in sum3 and use `i r` to view the contents of the registers as the code is executed. This should help you debug this final issue.

Checking Your Output with the Reference Solution

If you see something different from the expected output, please correct your code. To be absolutely sure, we have provided a reference solution `pa0test`, which you can run to view the correct output with the following command:

```
cs30xyz@pi-cluster-001:~/pa0$ ~/../public/pa0test
```

You can easily check if your solution matches our solution by running both programs, redirecting both standard out and standard error to a file, and then comparing them. You can do this by typing the following from your pa0 directory. An example:

```
cs30xyz@pi-cluster-001:~/pa0$ ./a.out >& MYSOL
cs30xyz@pi-cluster-001:~/pa0$ ~/../public/pa0test >& REFSOL
cs30xyz@pi-cluster-001:~/pa0$ diff -c MYSOL REFSOL
cs30xyz@pi-cluster-001:~/pa0$
```

If you see some output after running diff, then that means your solution does not match ours. **IF THERE IS A DIFFERENCE, YOU MUST CORRECT THIS!** Part of your project is graded automatically based on your

Back to Top

solution exactly matching our solution. You can manually inspect the output files yourself using the following, or, open them up in vi/vim/gvim:

```
cs30xyz@pi-cluster-001:~/pa0$ cat MYSOL
cs30xyz@pi-cluster-001:~/pa0$ cat REFSOL
```

If the output looks the same, but diff is showing something, be sure to check for extra newline, tab, and trailing space characters.

README File

See the README guidelines here (http://cseweb.ucsd.edu/~ricko/CSE30READMEGuidelines.pdf).

Here are two sample READMEs for PA0 that you can take a look at:

- Sample 1: http://cseweb.ucsd.edu/~ricko/CSE30/README_Sample_A.txt
- Sample 2: http://cseweb.ucsd.edu/~ricko/CSE30/README_Sample_B.txt

Feel free to use these as a template for your README for this and future assignments. As you can see, READMEs don't all have to look the same, but make sure you have everything listed in the guidelines linked above.

Questions to Answer in the README

0. Why is it considered an integrity violation if a student submits code copied from someone/somewhere else?

1. List the 3 compilation errors and 1 compilation warning you found in the source files. Please include the name of the file that consists the error, the line number, and your fix for the error. Note that one of the fixes will resolve both a compiler error and the 1 compiler warning.

2. Why is the program not printing the correct output when you first run it after successful compilation? How did you fix it? Please include all 5 runtime errors. For each error, give the name of the file containing the error, the line number, and your fix for the error.

Complete the steps below to answer the following questions. Some questions have multiple parts. If you see a question mark, you should have an answer in your README for that question.

At any time while you are in gdb, you can use the `help` command to get more info on a particular command. For example, `help next` or just `h next` as a shortcut. For a more complete information on gdb, see the online gdb manual.

To start the GNU debugger, at the prompt type:

```
gdb executablename
```

In our case it will be:

```
gdb a.out
```

Once the debugger is loaded, type the following:

```
display/i $pc
```

Back to Top

```
break main
break sum3
run
```

3. What line of C code do you see printed to the screen?

4. What happens if you type `nexti` at this point? Why?

Type `list`. This should show you about 10 lines from your main program (you can type list at any point during the debugging process and it will show you the "C" code that is around the line you are executing).

5. Type `continue`. Which function are you in now?

Do `nexti` twice until you see something similar to the following line:

```
46              add   r3, r3, r2            @ Add previous result with param3;
1: x/i $pc
=> 0x106ec <sum3+12>:   add      r3, r3, r2
```

6. Type `p $r3`**.** What number does it show?

7. Type `nexti`, then again type `p $r3`. What number does it show now? The line displayed in gdb is the line about to be executed.

8. Do `nexti` until you see something similar to the following line:

```
52              sub   sp, fp, FP_OFFSET     @ Set sp to top of saved registers
1: x/i $pc
=> 0x106f4 <sum3+20>:   sub      sp, r11, #4
```

Using one of the commands discussed earlier, what is the value of r1? List the value and **two** different commands you can use to determine the value.

Type `disassemble`. This should show you about 10 lines from your sum3.s file. (You can type disassemble at any point during the debugging process and it will show you the "assembly" code that is around the line you are executing.)

To finish running the executable, type `continue` and the program should run to completion.

Type `q` to quit the debugger and return to the shell prompt.

If you need to debug a program after getting a core dump, start gdb as:
`gdb executable_name core`

Now that we've warmed up with some easy gdb let's try playing around with examining values on the stack. Start gdb with the argument a.out and set a breakpoint in main, right after the call to initArray, somewhere around line 45.

```
(gdb) b pa0.c:45
Breakpoint 1 at 0x10540: file pa0.c, line 45.
```

Back to Top

```
(gdb)
```

By setting a breakpoint here, we are ensuring that all the local variables in main, including the array, have been initialized and should contain some recognizable values. Now, let's let the program run and then take a look at what values are currently on the stack. We will use x to examine memory, and will specify that it should print out 24 4-byte blocks of memory.

```
(gdb) run
Starting program: /home/linux/ieng6/cs30x/cs30xyz/pa0/a.out

Breakpoint 1, main () at pa0.c:45
45      for ( i = 0; i < SIZE; i++ ) {
(gdb) x/24x $sp
0x7efffad0:     0x00000000     0x00000000     0x00000001     0x00000003
0x7efffae0:     0x00000005     0x00000007     0x00000009     0x0000000b
0x7efffaf0:     0x0000000d     0x0000000f     0x00000011     0x00000013
0x7efffb00:     0x00000015     0x00000017     0x00000019     0x0000001b
0x7efffb10:     0x0000001d     0x00010700     0x00000020     0x0000000f
0x7efffb20:     0x00000005     0x00000000     0x00000000     0x76e90294
(gdb)
```

What are we looking at here? It's the local variables for the function main that have been allocated on the stack! Let's confirm this by printing out the address and the value of the variable `v1` from main and seeing if it matches what we see above.

```
(gdb) p/x &v1
$3 = 0x7efffb20
(gdb) p/x v1
$4 = 0x5
(gdb)
```

As you can see, the address of `v1` is `0x7efffb20`, which as seen in our print-out of the stack above contains the value `0x00000005`, or 5 in decimal. Something worth noting is that local variables in a function are stored in a very specific order in the stack. Remember that in pa0.c our variables are declared like this:

```
int i;
int v1 = FIRST_NUM_TO_SUM;
int v2 = SECOND_NUM_TO_SUM;
int v3 = THIRD_NUM_TO_SUM;
int sum;
int intArray[SIZE];
```

The first variable we declared, `i`, is stored at the highest memory address, in this case `0x7efffb24`, while the last variable we declare, `intArray`, starts at the lowest memory address, in this case `0x7efffad8`. You can confirm this by looking for the values of `v1`, `v2`, and `v3` in the stack we printed above.

9. Using the values from the stack printed above, what variable appears to have the value `0x00010700`? (if you are stuck try printing out the values of each variable in gdb using p/x).

10. Using the values from the stack printed above, what is the address of the last element of the array? What is the decimal value of the hexadecimal number stored at that address?

Back to Top

Now that we've looked at local variables on the stack, let's take a look at how we set up the frame pointer in our assembly file, sum3.s. Restart gdb with a.out, set a breakpoint in sum3.s, and run it.

```
(gdb) b sum3
Breakpoint 1 at 0x106e8: file sum3.s, line 42.
(gdb) run
Starting program: /home/linux/ieng6/cs30x/cs30xyz/pa0/a.out
.
.
.
Breakpoint 1, sum3 () at sum3.s:42
42              add     fp, sp, FP_OFFSET     @ Set fp to base of saved registers
```

Now, use `nexti` to execute the add instruction that sets the value of fp.

```
(gdb) ni
45              add     r3, r0, r1            @ Add param1 and param2; result
into r3
```

Now, use x to examine the stack and show the first two values on the stack.

```
(gdb) x/2x $sp
0x7efffad8:     0x7efffb3c      0x000105cc
(gdb)
```

11. What are the names of these two values? (check your notes)

12. What part of the program does the value `0x00105cc` point to? (try investigating with x)

13. What is a breakpoint? How do you set one? (You did this earlier).

14. What function are you debugging if gdb displays the following?

```
<foobar+32>:    ldr r1, [r0]
```

15. What is the difference between `step` and `next`? What is the difference between `step/next` and `stepi/nexti`?

16. What are `$r0`, `$r1`, etc, referring to?

This ends the GDB introduction part of the README. The next few README questions refer to things you can do when programming in C:

17. Given an integer number `n`, how do you use it as an ASCII number and print out its corresponding character in the ASCII chart using `printf`?

18. Given any integer `n` (in decimal form), how do you use `printf` to print out its corresponding hexadecimal and octal form without any leading "0x" or "0"? How do you print out its hexadecimal form *with* a leading "0x"? octal form *with* a leading "0"?

19. Assume you have these variables defined in your C code:

Back to Top

```
int value = 1234;
float thirty = 3.0;
char * rocks = "CSE30ROCKS!";
char plus = '+';
```

Give one line of C code **using these variables** that would print the following line to `stderr`:

```
CSE30ROCKS! 1234 + 3.0
```

20. Give the C code (can be more than one line) that would print the following to `stdout`:

```
The size of char is: ###
The size of short is: ###
The size of int is: ###
The size of long is: ###
The size of float is: ###
```

Where `###` is the decimal value of the size of each of the types. (Note: you must use one of the operators listed on your ANSI C Card handed out in class--you will receive zero points if you hardcode the sizes).

Turnin Instructions

Once you have checked your output, compiled, executed your code, finished your README file (see above), and double-checked your style (see style guidelines), you are ready to turn it in. Use the following names *exactly* otherwise our Makefile will not find your files.

Files required:

`pa0.c`	`initArray.c`	`sum3.s`
`pa0.h`	`printInOrder.c`	`Makefile`
	`printReversed.c`	`README`

How to Turn in an Assignment

Before turning in, run `make clean` and then `make` to double check for any compiler errors/warnings. Then use the following turnin script from the **raspberry pi** to submit your full assignment before the due date as follows:

```
$ ~/../public/bin/cse30turnin pa0
```

How to Verify your Submitted Files

Use the following script to view the time and date of your most recent submission:

```
$ ~/../public/bin/cse30verify pa0
```

The governing time will be the one which appears on that file (the system time). The system time may be obtained by typing `date`.

Up until the due date, you can re-submit your assignment via the scripts above.

Failure to follow the procedures outlined here will result in your assignment not being collected properly and will result in a loss of points. **Late assignments WILL NOT be accepted.**

If there is anything in these procedures which needs clarifying, please feel free to ask any tutor, the instructor, or post on the Piazza Discussion Board.

Back to Top