Programming Assignment 1 (PA1) - Cool S

Milestone Due: **Wednesday, April 17 @ 11:59 pm**
Final Due: **Wednesday, April 24 @ 11:59 pm**

Assignment Overview	Getting Started	Example Input	Detailed Overview	Milestone Functions
Post-Milestone Functions	Unit Testing	README File	Extra Credit	Turnin Summary

Assignment Overview

The purpose of this programming assignment is to build your knowledge of the ARM assembly language, especially branching and looping logic, calling assembly routines from within a C program, calling C functions from within assembly routines, allocating space for local variables, passing parameters and returning values, using Unix command line arguments, and learning some useful Standard C Library routines.

In this assignment you are writing a program that takes in 2 inputs from the command line and prints a Cool S design to `stdout`. Check out the Wikipedia page for info on Cool S. See `man ascii` for a map of the ASCII character set. This assignment will require appropriate error checking and reporting (as documented below).

Start early! **Remember that you can and should use `man` in order to lookup information on specific C functions**. For example, if you would like to know what type of parameters `strtol()` takes, or what `strtol()` does to `errno`, type `man -s3 strtol`. Also, take advantage of the tutors in the lab. They are there to help you learn more on your own and help you get through the course!

Grading

- **README: 10 points** - See README Requirements here and questions below
  - http://cseweb.ucsd.edu/~ricko/CSE30READMEGuidelines.pdf
- **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
  - **Note:** Character literals are considered as magic numbers as well (**except** '\0' which has ASCII value of 0)
  - http://cseweb.ucsd.edu/~ricko/CSE30StyleGuidelines.pdf
- **Correctness: 65 points**
  - **Milestone (15 points)** - To be distributed across the Milestone functions (see below)
  - Make sure you have all files tracked in Git.
- **Extra Credit: 5 points** - View Extra Credit section for more information.
- **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 the given Makefile, you will receive 0 points for this assignment.

Getting Started

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

Gathering Starter Files:
The first step is to gather all the appropriate files for this assignment. Connect to pi-cluster via ssh.

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

Create and enter the pa1 working directory.

```
$ mkdir ~/pa1
$ cd ~/pa1
```

Copy the starter files from the public directory.

```
$ cp ~/../public/pa1StarterFiles/* ~/pa1/
```

Starter files provided:

`pa1.h`	`pa1Strings.h`	`Makefile`
`test.h`	`testintervalContains.c`	`drawCrissCross.c`

Preparing Git Repository:
You are required to use Git with this and all future programming assignments. Refer to the PA0 writeup for how to set up your local git repository.

Example Input

A sample stripped executable provided for you to try and compare your output against is available in the public directory. Note that you cannot copy it to your own directory; you can only run it using the following command (where you will also pass in the command line arguments):

```
$ ~/../public/pa1test
```

NOTE:

1. The output of your program MUST match exactly as it appears in the `pa1test` output. You need to pay attention to everything in the output, from the order of the error messages to the small things like extra newlines at the end (or beginning, or middle, or everywhere)!
2. **We are not giving you any sample outputs. Instead, you are provided some example inputs. You are responsible for testing all functionalities of the program; the list of example inputs is not exhaustive or complete. It is important that you fully understand how the program works and test your final solution thoroughly against the executable.**

Note: In the following examples, you will notice that special characters such as `$`, `&`, `*`, `!` and `@` are surrounded in single quotes. This is because some special characters are bash meta-characters, which means they will be interpreted as commands to bash, and not command line arguments to your program. We use single quotes to tell bash that these characters need to be interpreted as command line arguments for your executable. You do not need to do anything special in your code to handle this--bash will read the single quotes and will only pass string inside the quotes to your program as a command line argument.

On a related note, we use `$` and surround escape sequences with single quotes (e.g. `$'\n'`) in order to use them for our arguments. This lets us insert usually non-printable characters. For example, we can use the syntax to write `$'\x01'`, which is the SOH "Start of Heading" character). If we were to only surround the literal in single quotes, it would not properly escape the escape sequence (e.g. the escape sequence '\n' would be interpreted as a literal backslash and then a literal 'n' character).

Example input that has error output:

```
cs30xyz@pi-cluster-001:~/pa1$ ./pa1
cs30xyz@pi-cluster-001:~/pa1$ ./pa1 str
cs30xyz@pi-cluster-001:~/pa1$ ./pa1 8a '$'
cs30xyz@pi-cluster-001:~/pa1$ ./pa1 9999999999999999999 a
cs30xyz@pi-cluster-001:~/pa1$ ./pa1 7 $'\n'
cs30xyz@pi-cluster-001:~/pa1$ ./pa1 8 a a
```

Example input that has normal output:

```
cs30xyz@pi-cluster-001:pa1$ ./pa1 23 '!'
cs30xyz@pi-cluster-001:pa1$ ./pa1 15 m
```

Detailed Overview

The function prototypes for the various C and Assembly functions are as follows.

C routines:

```
void drawCoolS( int size, char fillChar );
void outputCharNTimes( char ch, int n );
int main( int argc, char * argv[] );
```

Assembly routines:

```
int myRem( int dividend, int divisor );
int isDividable( int dividend, int divisor );
int intervalContains( int start, int end, int value );
void outputChar( char ch );
void drawCap( int size, char fillChar, int direction );
```

For the Milestone, you will need to complete:

```
myRem.s          isDividable.s          intervalContains.s
outputChar.s     outputCharNTimes.c
```

Process Overview:

The following is an explanation of the main tasks of the assignment, and how the individual functions work together to form the whole program.

This program takes 2 command line arguments:

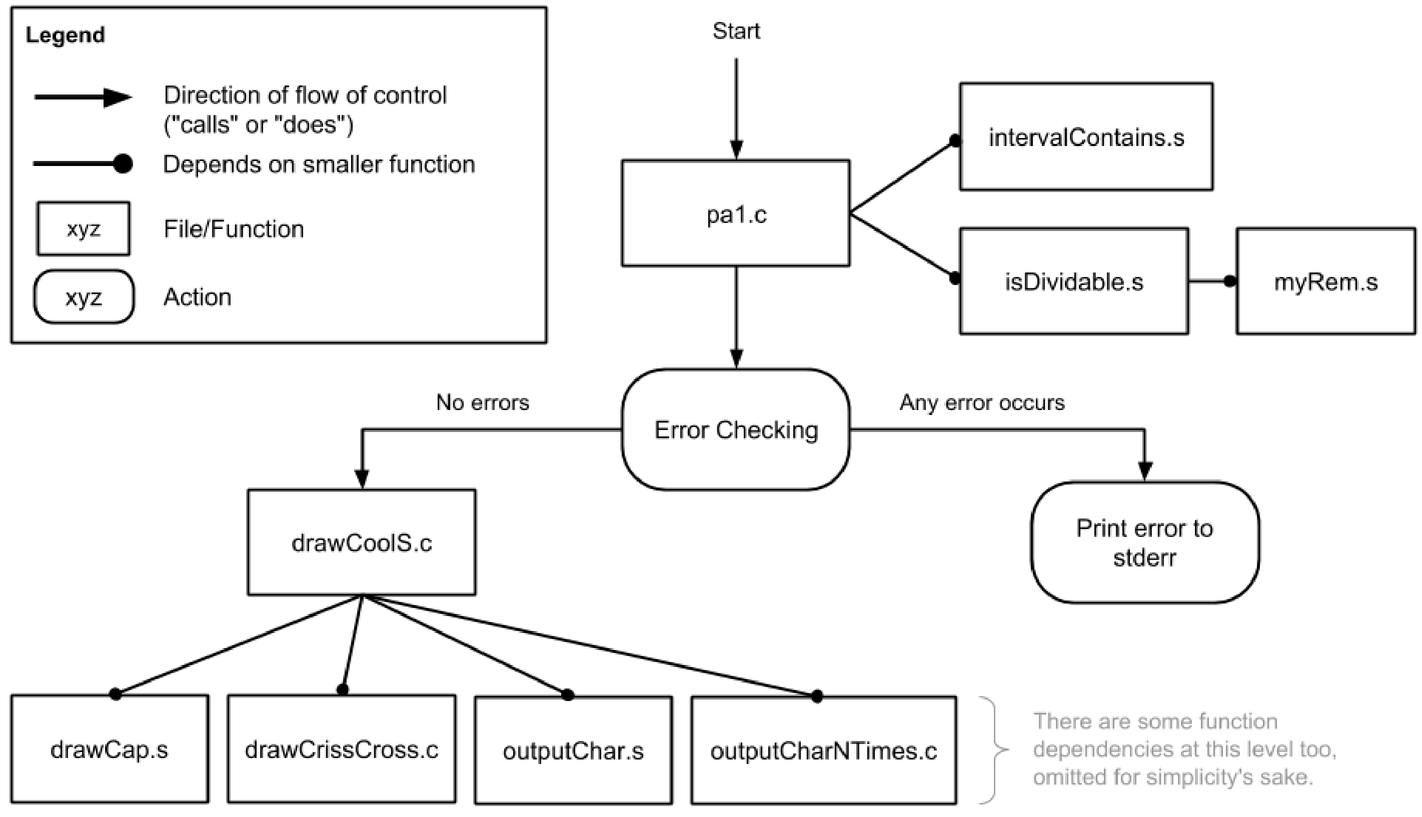
```
$ ./pa1 size fillChar
```

Explanation of the Command Line Arguments:

<table>
<tr><td>size</td><td>maximum width of the Cool S</td><td rowspan="2">size
  ^
 /a\
 /aaa\
/aaaaa\
|aa|aa|
|aa|aa|
\aa\aa/
 \aa\/
 /\aa\
/aa\aa\
|aa|aa|
|aa|aa|
\aaaaa/
 \aaa/
 \a/
  v
fillChar</td></tr>
<tr><td>fillChar</td><td>the character used to fill in the Cool S</td></tr>
</table>

Drawing the Cool S consists of the following steps:

1. Parse command line arguments in `main()`, where `intervalContains()`, `myRem()` and `isDividable()` help with error checking.
   a. If there is an error, print the appropriate error message to **stderr** and exit right away.
   b. If there is no error, proceed to print out the Cool S pattern.
2. Draw the Cool S using `drawCoolS()`, where `outputChar()`, `outputCharNTimes()`, `drawCrissCross()` and `drawCap()` are used to print out each individual character that makes up the different sections of the pattern.



Milestone Functions to be Written

Listed below are the modules to be written for the milestone.

---

myRem.s

```
int myRem( int dividend, int divisor );
```

Calculates the remainder when dividing `dividend` by `divisor`. This should have the same behavior as the `%` operator in C.

Reasons for error:

- `divisor` is zero → result is undefined (we will not be checking for divide by 0)

Return Value: The remainder.

---

isDividable.s

```
int isDividable( int dividend, int divisor );
```

Tests if the `dividend` is evenly dividable by the `divisor`. Return 1 if the `dividend` is evenly dividable by the `divisor`, returns 0 otherwise.

Reasons for error:

- If the `divisor` is 0, return -1. Note that negative numbers are allowed.

Return Value: -1 if error, 1 if dividable, 0 if not dividable.

---

intervalContains.s

`int intervalContains( int start, int end, int value );`

Determines whether or not `value` is inside the interval [`start`, `end`]. This interval is inclusive on both ends.

Reasons for error:

- If `start` is greater than `end`, return -1 for error.

Return Value: -1 if the interval is invalid, 1 if `value` is contained in the interval, 0 if `value` is not contained in the interval.

---

outputChar.s

`void outputChar( char ch );`

This assembly module prints the character `ch` to stdout (see `man -s3 printf`). This is very similar to the example assembly module given below. The main difference is that `outputChar()` just prints a single character (so think about how that might affect the format string `fmt`).

Example: This example assembly module takes in a string that represents a class, and prints out a message saying that that class is your favorite class.

`printFavoriteClass("CSE 30")` → `"My favorite class is CSE 30"`

```
        .cpu    cortex-a53
        .syntax unified

        .equ    FP_OFFSET, 4

        .section        .rodata

fmt:    .asciz "My favorite class is %s\n"

        .global printFavoriteClass

        .text
        .align  2

printFavoriteClass:
        push    {fp, lr}                @ Saves registers on the stack.
        add     fp, sp, FP_OFFSET       @ Sets fp to point at bottom of saved regs.
                                        @   Uses 4, from (#_of_regs_saved - 1) * 4.
        mov     r1, r0                  @ Moves string param to r1 as arg to printf.
        ldr     r0, =fmt                @ Gets address of format string.
        bl      printf                  @ Calls printf.

        sub     sp, fp, FP_OFFSET       @ Reset sp to point to top of saved regs.
        pop     {fp, pc}                @ Returns from function.
```

outputCharNTimes.c

```
void outputCharNTimes( char ch, int n );
```

Prints the character `ch` to stdout `n` times. Please note that you may NOT use printf to implement this function.

Hint: Is there a helper function that you could use?

Post-Milestone Functions to be Written

Listed below are the modules to be written after the milestone functions are complete.

drawCap.s

```
void drawCap( int size, char fillChar, int direction );
```

This assembly module will print out individual characters (via calls to `outputChar()` and `outputCharNTimes()`) such that the Cool S caps will be displayed based on the user-supplied values.
NOTE: You will have to allocate space on the stack for local variables and formal parameters.

To help you with this module, you may use these offsets (on the right) for local variables and parameters. Material in lecture will cover what these offsets are used for in depth, and in future assignments you will need to calculate these offsets yourself. Remember that you will need to load and store from these locations.

```
@ Constants for local variables
        .equ LOCAL_VAR_SPACE, 32
        .equ TIP_CHAR_OFFSET, -8
        .equ LEFT_SLASH_CHAR_OFFSET, -12
        .equ RIGHT_SLASH_CHAR_OFFSET, -16
        .equ START_ITER_OFFSET, -20
        .equ END_ITER_OFFSET, -24
        .equ INCR_OFFSET, -28
        .equ CAP_SIZE_OFFSET, -32
        .equ I_OFFSET, -36

@ Constants for parameters
        .equ PARAM_SPACE, 16
        .equ SIZE_OFFSET, -40
        .equ FILL_CHAR_OFFSET, -44
        .equ DIRECTION_OFFSET, -48
```

Here's the equivalent C version of drawCap.c:

```
#define HALF_DIVISOR 2 // Used for dividing variable size in half
#define DOUBLE       2 // Used for doubling values

// Cool S cap directions
#define DIR_UP   0
#define DIR_DOWN 1

// Cool S whitespace, tip, and border characters
#define SPACE_CHAR         ' '
#define NEWLINE_CHAR       '\n'
#define FORWARD_SLASH_CHAR '/'
#define BACK_SLASH_CHAR    '\\'
#define CARAT_CHAR         '^'
#define V_CHAR             'v'

void drawCap(int size, char fillChar, int direction) {
  // The characters for drawing the edges of the cap; dependent on direction
  int tipChar;
  int leftSlashChar;
  int rightSlashChar;

  // Loop iteration counters
  int startIter;
  int endIter;
  int incr;

  int capSize = size / HALF_DIVISOR;

  int i;

  // Drawing the top cap
  if (direction == DIR_UP) {
    tipChar = CARAT_CHAR;
    leftSlashChar = FORWARD_SLASH_CHAR;
    rightSlashChar = BACK_SLASH_CHAR;

    startIter = 0;
    endIter = capSize + 1;
    incr = 1;
  // Drawing the bottom cap
  } else {
    tipChar = V_CHAR;
    leftSlashChar = BACK_SLASH_CHAR;
    rightSlashChar = FORWARD_SLASH_CHAR;

    startIter = capSize;
    endIter = -1;
    incr = -1;
  }

  // Start drawing the cap
  i = startIter;
  while (i != endIter) {
    // Draw the leading whitespace
    outputCharNTimes(SPACE_CHAR, capSize - i);

    // Draw the actual cap content, conditionally the tip
    // if it's the first/last iteration
    if (i == 0) {
      outputChar(tipChar);
    } else {
      outputChar(leftSlashChar);
      outputCharNTimes(fillChar, DOUBLE * i - 1);
      outputChar(rightSlashChar);
    }

    // Draw the trailing whitespace
    outputCharNTimes(SPACE_CHAR, capSize - i);
    outputChar(NEWLINE_CHAR);

    // Same as i = i + incr
    i += incr;
  }
}
```

This will be a translation task for you. All of the assembly constructs you will use will have been covered in lecture and can be referenced in your notes. You are not limited to using the algorithm provided in the link above, but part of the purpose of this programming assignment is to learn how to write looping/conditional constructs (branches), to use the simple `bl` instruction to branch to subroutines with parameter passing, and to perform simple arithmetic instructions (`add`/`sub`, `sdiv`) in assembly.

We would encourage you to use the linked algorithm for these reasons, however we do not want to suppress creative thinking - alternative solutions are welcome. However, you must use the "preferred" style of coding loops, backwards branching logic, as detailed in class: set up an opposite logic branch to jump over the loop body and a positive logic branch to jump backwards to the loop body. Points will be taken off for not using backwards branching logic.

As always, you must define constants to avoid using magic numbers. A helpful list of #defines at the top of the drawCap C version has been provided for you. You must translate these into corresponding .equ directives in your assembly translation, as #defined constants in C are not (easily) accessible from assembly files.

---

drawCoolS.c

```
void drawCoolS( int size, char fillChar );
```

This function will print out the entire Cool S design using the given `size` and `fillChar` specified by the user. To recreate the Cool S, follow the steps below and use your helper functions (such as `drawCap()`) to draw each section. Each output snippet in this description assumes that the `size` is `7` and the `fillChar` is `'a'`.

1. Draw the top cap. This is just the pyramid-like structure on the top of the S.

```
   ^
  /a\
 /aaa\
/aaaaa\
```

2. Draw the first straight section. This refers to the section where there are `fillChar`s partitioned into two sections by single pipe (`|`) characters.
    - There is no leading or trailing whitespace.
    - The number of times `fillChar` repeats in between the pipes is equal to `size / 2 - 1`.
    - The number of lines of the straight portion also equals `size / 2 - 1`.

   As you can see in the example, only 2 `a`'s are printed in between each pair of pipes, and there are only 2 lines in the example output, since (`7 / 2 - 1) == 2`.

```
|aa|aa|
|aa|aa|
```

3. Draw the criss-cross section. The criss-cross section is where the `a`'s cross over, forming a shape resembling an `X` character.

```
\aa\aa/
 \aa\/
 /\aa\
/aa\aa\
```

4. Draw the second straight section. This should be identical to the first one.

```
|aa|aa|
|aa|aa|
```

5. Draw the bottom cap. This is like the top cap, but flipped upside-down.

```
\aaaaa/
 \aaa/
  \a/
   v
```

---

pa1.c

```
int main( int argc, char * argv[] );
```

The main function will drive the rest of the program. It will first perform input checking by parsing the command-line arguments and checking for errors. If all inputs are valid, it will call `drawCoolS()`. Otherwise, it will print the corresponding error message and return right away. Remember that many of the error strings have format specifiers, so be sure to add the appropriate arguments when printing error messages.

First: check that the user entered a valid number of command line arguments. If they didn't, print the `COOL_S_USAGE` to stderr and return `EXIT_FAILURE` right away.

Now we can parse the command line arguments in the following steps. For all cases, if you encounter an error condition, print the corresponding error message to stderr and return `EXIT_FAILURE` immediately:

1. **size:** set the global variable `errno` to 0 (see `man -s3 errno`), then use `strtol()` to convert the Cool S size to an int (see `man -s3 strtol`). Check for the following errors in the order they appear below.

Error	How to Handle
Error converting to int (`errno` was set by `strtol()`)	Use `snprintf()` to build the error string using `SIZE_CONVERT_ERR` (make sure the string is null-terminated). Call `perror()` to print the string, passing the string as a parameter. Call `fprintf()` to print a newline after the error message.
size contains non-numerical characters (check `endptr`)	Use `fprintf()` to print the `SIZE_NOT_INT_ERR` error message.
size is not in bounds (not within `[MIN_SIZE, MAX_SIZE]`)	Use `fprintf()` to print the `SIZE_RANGE_ERR` error message.
size is not in the format of 4n + 3 (use `isDividable()`)	Use `fprintf()` to print the `SIZE_FORMAT_ERR` error message.

2. **fillChar:** extract the first character from the fillChar argument. Check for the following errors in the order they appear below.

Error	How to Handle
Not a single character (hint: `man -s3 strlen`)	Use `fprintf()` to print the `SINGLE_CHAR_ERR` error message.
char not in bounds (not within `[MIN_CHAR`, `MAX_CHAR]`)	Use `fprintf()` to print the `CHAR_RANGE_ERR` error message.

3. If no errors were encountered, draw the Cool S by calling the `drawCoolS()` function, passing in the appropriate arguments.

4. Return `EXIT_SUCCESS`.

Return Value: If an error is encountered, return `EXIT_FAILURE`. Otherwise, return `EXIT_SUCCESS`.

Unit Testing

You are provided with only one basic unit test file for the Milestone function `intervalContains()`. This file only has minimal test cases and is only meant to give you an idea of how to write your own unit test files. **You must write unit test files for each of the Milestone functions, as well as add several of your own thorough test cases to all of the unit test files. You need to add as many unit tests as necessary to cover all possible use cases of each function. You will lose points if you don't do this!** You are responsible for making sure you thoroughly test your functions. Make sure you think about boundary cases, special cases, general cases, extreme limits, error cases, etc. as appropriate for each function. The Makefile includes the rules for compiling these tests. Keep in mind that your unit tests will not build until all required files for that specific unit test have been written. These test files **will be collected with the Milestone**, they must be complete before you submit your Milestone.

Unit tests you need to complete:

```
testmyRem.c
testisDividable.c
testintervalContains.c
testoutputChar.c
testoutputCharNTimes.c
```

To compile:

```
$ make testintervalContains
```

To run:

```
$ ./testintervalContains
```

(Replace "`testintervalContains`" with the appropriate file names to compile and run the other unit tests)

We also supply you an extra set of "optimized" Makefile targets for your tester files. These targets will compile your tester file with all optimizations turned on, similar to how your Milestone functions will be compiled for grading. Sometimes you will get (un)lucky where your program appears to work even when there is something wrong (like incorrect memory layout or modifying a register you shouldn't). Compiling with optimizations will expose some of these hidden problems. Again, this is how the milestone will be tested during grading, so use the optimization targets to try to expose these issues.

However, compiling this way prevents you from using gdb to debug, so make sure to compile the regular way when debugging and try the optimized targets after your tests pass.

To compile with optimizations on:

```
$ make testintervalContains-opt
```

To run:

```
$ ./testintervalContains-opt
```

README File

Remember to follow all of the guidelines outlined in the README Guidelines. If you did the extra credit, write a program description for it in the README file as well.

Questions to Answer in your README:

Vimdiff Questions

1. Copy the directory that contains 3 files from public directory:

   ```
   $ cp -r ~/../public/pa1README ~/pa1README
   ```

   Go to the directory we just copied the files to:

   ```
   $ cd ~/pa1README
   ```

   Make your terminal window wide enough to view two files side-by-side, then compare average3V1.s and average3V2.s with the command:

   ```
   $ vimdiff average3V1.s average3V2.s
   ```

   The following should show up:

   ```
   + +-- 35 lines: Filename: average3.s----|+ +-- 35 lines: Filename: average3.s---
      *     r2 - arg 3 -- c                |   *     r2 - arg 3 -- c
      *     r3 - holds the intermediate sum|   *     r3 - holds the intermediate su
      */                                   |   */
     average3:                             |  average3:
     @ Standard prologue                   |  @ Standard prologue
             push    {fp, lr}              |          push    {fp, lr}
             add     fp, sp, FP_OFFSET     |  ------------------------------------
                                           |
             @ Incoming parameters in r0, r|          @ Incoming parameters in r0,
             add     r3, r0, r1            |          add     r3, r0, r1
                                           |
             add     r3, r3, r2            |          add     r3, r3, r2
                                           |
   + +--  2 lines: @ Compute the average of|+ +--  2 lines: @ Compute the average o
             mov     r1, NUM_ELEMENTS      |          mov     r1, NUM_ELEMENTS
                                           |
             sdiv    r0, r3, r1            |          sdiv    r0, r3, r1
                                           |
     @ Standard epilogue                   |  @ Standard epilogue
             sub     sp, fp, FP_OFFSET     |          sub     sp, fp, FP_OFFSET
             pop     {fp, pc}              |          pop     {pc, fp}
   average3V1.s              1,1        Top average3V2.s             1,1           Top
   "average3V2.s" 57L, 1897C
   ```

   As you can see, there are 2 differences. The first one is at line 42. average3V2.s does not have the line to add the correct offset to sp to set fp. The second one is at line 57 for averageV1.s and line 56 for average3V2.s. The order of pc and fp are switched in average3V2.s, which is highlighted in red.

Spot the 3 differences between average3V1.s and average3V3.s using vimdiff and document them as the answer to this README question. Be clear about the location of the differences.

C Questions

2. Here is a sample C program that prints out a number:

```
1   #include <stdio.h>
2
3   void foo(int * x) {
4     ___________;
5   }
6
7   int main() {
8     int a = 3;
9     int * b = &a;
10    foo(b);
11    printf("%d\n", a);
12  }
```

   What line of code needs to be added on line 4 to make the program print `9` ?

3. We have an executable called `s`. If it is run with the following command, what will the value of `argc` in `main()` be? Why?

```
$ ./s html is the best programming language
```

Vim Questions

4. (a) What command do you use to delete a single line at the current cursor? (b) What about n number of lines (including the line at the current cursor)?
5. (a) Which command would you use to copy a line at the current cursor? (b) Then which command to “paste” the copied line to after the cursor?
6. (a) Which command would you use to show line numbers? (b) Which command to hide line numbers?
7. (a) Which command would you use to move the cursor to line x? (b) Which command to go to the first line of the file? (c) Which command to go to the last line of the file?
8. (a) Which command would you use to jump the cursor 10 lines down? (b) Which command to jump 10 lines up?

Git Questions

9. What is the Git command to display differences between the local version of a file and the version last committed?

GDB Questions

Start gdb with your pa1 executable, then set a breakpoint at `strtol`. Breakpoints are set with the `break` command (abbreviated `b`):

```
(gdb) break strtol
```

Note that you may see this message:

```
Function "strtol" not defined.
Make breakpoint pending on future shared library load? (y or [n])
```

In this case just hit `y` to continue.

Run the program (in gdb) with the following command line args:

```
(gdb) run 19Spring '#'
```

It's okay if you see an error like "`strtol.c: No such file or directory`".

Now you should be at the entry point of the Std C Lib routine strtol called from main.

10. How do you print the value of the string that is the 1st arg in strtol? (The value should be "19Spring")

11. How do you print the *hex* value of `&endptr` that is the 2nd arg in strtol? (The value should be something like 0x7efff - a high stack address - will vary)

Go to the next high level source instruction in main by entering `next` until you see a C source line from your main.c file pop up (about 5 times). This should be the next C instruction in main after the function call to strtol. Again, if you encountered the `"strtol" not defined` error earlier or "`in strtol.c`" messages, you may have to enter `next` a few times until the code returns to main.

12. How do you print the value returned by strtol? (The value should be 19) Show two ways:
    a. Using the name of the local variable you use to hold the return value
    b. Displaying the value in the register used to return the value

13. How do you print the character `endptr` is pointing to? (Should be the character 'S')

14. How do you print the entire null-terminated string `endptr` is pointing to? (Should be "Spring")

15. How do you print the decimal value of the global variable `errno` at this point? (The value should be 0)

Continue the execution of your pa1 in gdb. Type `continue,` which means resuming program execution up to the next breakpoint or until termination if no breakpoints are encountered. In this case, the following error message should be printed and the program will exit with code 01:

```
Error: size must be an integer in base 10
```

Run the program again (in gdb) with the following command line args:

```
(gdb) run 11111111111111 '#'
```

You should be at the entry point of the Std C Lib routine strtol called from main. Go to the next source instruction in main using the `next` command like before. Again, if you encountered the `"strtol" not defined` error or "`in strtol.c`", you may have to enter `next` a few times until the code returns to main.

It should be the source-level instruction after the call to strtol passing in 11111111111111 to convert to an integer. Print the decimal value of `errno` at this point. The value of `errno` should be 34 now which is the value of `ERANGE`. See the man pages `man -s3 errno` and `man -s2 intro` for more information about `errno`. Now feel free to `continue` or `quit` from gdb.

GDB Text User Interface
This can be helpful for debugging.

You can start your program with `gdb -tui <executable_name>`

Use command `layout asm` to show the assembly code
Use command `layout reg` to show all the registers
Use command `layout src` to show the source code
If the window corrupts during execution, use command `refresh` or press Ctrl+L to refresh the window.

For example, when we run this on pa0, it will show the following:

```
┌─Register group: general─────────────────────────────────────────────────────────┐
│r0             0x1      1                     r1             0x3      3           │
│r2             0x5      5                     r3             0x0      0           │
│r4             0x0      0                     r5             0x0      0           │
│r6             0x1040c  66572                 r7             0x0      0           │
│r8             0x0      0                     r9             0x0      0           │
│r10            0x76fff000        1996484608   r11            0x7efffb4c        2130705228
   ┌───────────────────────────────────────────────────────────────────────────────┐
   │0x10944 <average3>       push   {r11, lr}                                       │
B+>│0x10948 <average3+4>     add    r11, sp, #4                                     │
   │0x1094c <average3+8>     add    r3, r0, r1                                      │
   │0x10950 <average3+12>    add    r3, r3, r2                                      │
   │0x10954 <average3+16>    mov    r1, #3                                          │
   │0x10958 <average3+20>    sdiv   r0, r3, r1                                      │
   │0x1095c <average3+24>    sub    sp, r11, #4                                     │
   └───────────────────────────────────────────────────────────────────────────────┘
child process 4936 In: average3                                   Line: 42   PC: 0x10948
(gdb) run 1 3 5
Starting program: /home/linux/ieng6/cs30x/cs30x2/pa0/pa0-solution/a.out 1 3 5

Breakpoint 1, average3 () at average3.s:42
(gdb) layout asm
(gdb) layout reg
(gdb) refresh
(gdb)
```

Academic Integrity Question

16. What was your process for completing this assignment with integrity?

Extra Credit

There are 5 points total for extra credit on this assignment.

- Early turnin: **[2 Points]** 48 hours before regular due date and time
  [1 Point] 24 hours before regular due date and time
  (it's one or the other, not both)
- **[3 Points]** Chained Cool S

For extra credit, you'll be making a program to display a chain of Cool S's.

Getting Started:
Copy and rename the following file in your pa1 directory.

```
$ cp ~/pa1/drawCoolS.c ~/pa1/drawCoolSEC.c
```

You will also be creating a new assembly module called **sumOfDigitsEC.s** (more on this later).

Important: All of your original files must remain unchanged. You will need to turn in all of your original files plus two new files for the extra credit. Your original program must perform exactly as described in the main section of the writeup, and should be unaffected by the extra credit. Extra credit will be run as a separate executable.

Extra Credit Functions To Be Written:

Note that while the names of files to be written are `drawCoolSEC.c` and `sumOfDigitsEC.s`, the names of the functions should be `drawCoolS()` and `sumOfDigits()`, respectively.

---

drawCoolSEC.c

```
void drawCoolS( int size, char fillChar );
```

This function will print out the entire Cool S design using the given `size` and `fillChar` specified by the user, with an additional feature that the criss-cross and straight design will repeat `i` times, where `i` is the sum of the digits in `size`.

---

sumOfDigitsEC.s

```
int sumOfDigits( int size, int base );
```

Calculates the sum of all the digits in `size` with the given `base`. The sum is based on the individual digits in the `size`, ignoring the sign of the `size` (e.g. -123 in base 10 has a digit sum of 6). If the base is invalid (less than 2), then the digit values are undefined, so the sum of digits should just be 0.

Return Value: 0 if the value of `base` is invalid, else return the sum of digits in `size`

Compiling:

You can compile the extra credit program using the following command.

```
$ make pa1EC
```

Sample Executable:

A sample test executable is provided for the EC like the sample executable provided for the regular portion of the assignment. You can run the sample executable using the following command:

```
$ ~/../public/pa1ECtest
```

Sample Output:

```
cs30xyz@pi-cluster-001:~/pa1$ ./pa1EC 11 a
     ^
    /a\
   /aaa\
  /aaaaa\
 /aaaaaaa\
/aaaaaaaaa\
|aaaa|aaaa|
|aaaa|aaaa|
|aaaa|aaaa|
|aaaa|aaaa|
```

```
\aaaa\aaaa/
 \aaaa\aa/
  \aaaa\/
  /\aaaa\
 /aa\aaaa\
/aaaa\aaaa\
|aaaa|aaaa|
|aaaa|aaaa|
|aaaa|aaaa|
|aaaa|aaaa|
\aaaa\aaaa/
 \aaaa\aa/
  \aaaa\/
  /\aaaa\
 /aa\aaaa\
/aaaa\aaaa\
|aaaa|aaaa|
|aaaa|aaaa|
|aaaa|aaaa|
|aaaa|aaaa|
\aaaaaaaaa/
 \aaaaaaa/
  \aaaaa/
   \aaa/
    \a/
     v
```

Here, the criss-cross and bottom pattern repeats twice, as `sum` is 11 and 1 + 1 = 2.

Turnin Summary

See the turnin instructions here. Your file names must match the below *exactly* otherwise our Makefile will not find your files.

Milestone Turnin:
Due: Wednesday night, April 17 @ 11:59 pm

Files required for the Milestone:

myRem.s	isDividable.s	intervalContains.s
outputChar.s	outputCharNTimes.c	
testmyRem.c	testisDividable.c	testintervalContains.c
testoutputChar.c	testoutputCharNTimes.c	

Final Turnin:

Due: Wednesday night, April 24 @ 11:59 pm

Files required for the Final Turn-in:

drawCap.s	drawCoolS.c	pa1.h
intervalContains.s	drawCrissCross.c	pa1Strings.h
isDividable.s	outputCharNTimes.c	test.h
myRem.s	pa1.c	Makefile
outputChar.s	testintervalContains.c	README
	testisDividable.c	
	testmyRem.c	
	testoutputChar.c	
	testoutputCharNTimes.c	

Extra Credit Files:

sumOfDigitsEC.s	drawCoolSEC.c

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.