Programming Assignment 1 (PA1) - drawS

Milestone Due: **Wednesday, April 13 @ 11:59pm**
Final Due: **Tuesday, April 19 @ 11:59 pm**

Assignment Overview

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

You will be writing a program that takes 4 inputs from the command line:

```
[cs30xzzz@ieng9]:pa1$ ./pa1 sWidth sChar fillerChar borderChar
```

Explanation of command line arguments:

- `sWidth` -- Width of the 'S' (not including the borders)
- `sChar` -- ASCII value of the character making up the 'S'
- `fillerChar` -- ASCII value of the character filling the space between the 'S' and the border
- `borderChar` -- ASCII value of the character making up the rectangular border around the 'S'

These inputs are parameters for displaying a large 'S' character which will be printed to `stdout`. See `man ascii` for a map of the ASCII character set. ASCII values are used to help you get used to the ASCII character set which will appear on future tests. Appropriate error checking and reporting is required. Note that in most cases, we will display all the related errors instead of stopping after we encounter the first error.

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 -s3c 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 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 section
- **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 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 ieng9 via ssh (replace cs30xzzz with YOUR cs30 account).

```
      $ ssh cs30xzzz@ieng9.ucsd.edu
```

Create and enter the pa1 working directory.

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

Copy the starter files from the public directory.

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

<u>Starter files provided:</u>

`pa1.h`	`Makefile`	`test.h`
`pa1Strings.h`	`testisInRange.c`	

<u>Preparing Git Repository:</u>
You are required to use Git with this and all future programming assignments. Look at the PA0 writeup for additional information on Git.

<u>Setting up a local repository:</u>
Navigate to your pa1 directory and initialize a local git repository.

```
      $ git init
```

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

```
      $ git config --global user.name "John Doe"
      $ git config --global user.email  "johndoe@ucsd.edu"
```

<u>Adding and committing files:</u>
As you're developing, you can see the status of the files in your directory with the following command.

```
      $ git status
```

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

```
      $ git add filename
      $ 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.

```
      $ git add file1
      $ git add file2
      $ git add file3
      $ 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.

Ignoring files with .gitignore:
You may notice as you're developing your program that Git really wants to keep track of `.o` files, `.ln` files, and `.swp` files, 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.

```
      $ vim .gitignore
```

And add the following lines to it.

```
.gitignore
*.o
*.ln
*.sw*
*~
pa1
core
```

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

Sample Output

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
```

If there is a discrepancy between the sample output in this document and the `pa1test` output, follow the `pa1test` output.

Below are some brief example outputs of this program. Make sure you experiment with the public executable to further understand the program behavior. Bolded text is what you type in the terminal.

1. Command-line Parsing Errors

1.1. Too many arguments (extra operand).

```
[cs30xzzz@ieng9]:pa1$ ./pa1 4 32 32 32 32

Usage: ./pa1 sWidth sChar fillerChar borderChar
    sWidth      (must be within the range of [4 - 1000])
                (must be even)
    sChar       (must be an ASCII value within the range [32 - 126])
    fillerChar  (must be an ASCII value within the range [32 - 126])
                (must be different than sChar)
    borderChar  (must be an ASCII value within the range [32 - 126])
                (must be different than sChar)
```

1.2. No arguments.
```
[cs30xzzz@ieng9]:pa1$ ./pa1

Usage: ./pa1 sWidth sChar fillerChar borderChar
    sWidth      (must be within the range of [4 - 1000])
                (must be even)
    sChar       (must be an ASCII value within the range [32 - 126])
    fillerChar  (must be an ASCII value within the range [32 - 126])
                (must be different than sChar)
    borderChar  (must be an ASCII value within the range [32 - 126])
                (must be different than sChar)
```

1.3 Invalid `sWidth` strtol conversion (no errno)
```
[cs30xzzz@ieng9]:pa1$ ./pa1 10a 33 34 34

        "10a" is not an integer
```

1.4 Errno set in strtol conversion
```
[cs30xzzz@ieng9]:pa1$ ./pa1 999999999999999999 33 34 34

        Converting "999999999999999999" base "10": Result too large
```

2. Other Errors

2.1. Multiple errors: `sWidth` odd and out of range, `fillerChar` and `borderChar` same as `sChar`
```
[cs30xzzz@ieng9]:pa1$ ./pa1 3 32 32 32

        sWidth(3) must be within the range of [4 - 1000]

        sWidth(3) must be even

        sChar(32) and fillerChar(32) must be different

        sChar(32) and borderChar(32) must be different
```

2.2. `fillerChar` and `borderChar` out of ASCII range
```
[cs30xzzz@ieng9]:pa1$ ./pa1 38 82 30 140

        fillerChar(30) must be an ASCII code in the range [32 - 126]

        borderChar(140) must be an ASCII code in the range [32 - 126]
```

3. <u>Valid Output</u>

3.1. `sWidth` = 10, `sChar` = ' ', `fillerChar` = `X`, `borderChar` = `|`

```
[cs30xzzz@ieng9]:pa1$ ./pa1 10 32 88 124
||||||||||||||
||||||||||||||
||X        X||
|| XXXXXXXXX||
|| XXXXXXXXX||
|| XXXXXXXXX||
|| XXXXXXXXX||
|| XXXXXXXXX||
|| XXXXXXXXX||
|| XXXXXXXXX||
|| XXXXXXXXX||
||X        X||
||XXXXXXXXX ||
||XXXXXXXXX ||
||XXXXXXXXX ||
||XXXXXXXXX ||
||XXXXXXXXX ||
||XXXXXXXXX ||
||XXXXXXXXX ||
||XXXXXXXXX ||
||X        X||
||||||||||||||
||||||||||||||
```

3.2. `sWidth` = 6, `sChar` = `^`, `fillerChar` = ‘ ‘, `borderChar` = `z`

```
[cs30xzzz@ieng9]:pa1$ ./pa1 6 94 32 122
zzzzzzzz
z ^^^^ z
z^     z
z^     z
z^     z
z^     z
z ^^^^ z
z     ^z
z     ^z
z     ^z
z     ^z
z ^^^^ z
zzzzzzzz
```

3.3. `sWidth` = 8, `sChar` = `-`, `fillerChar` = `?`, `borderChar` = `#`

```
[cs30xzzz@ieng9]:pa1$ ./pa1 8 45 63 35
##########
#?------?#
#-???????#
#-???????#
#-???????#
#-???????#
#-???????#
#-???????#
#?------?#
#???????-#
#???????-#
#???????-#
#???????-#
#???????-#
#???????-#
#?------?#
##########
```

Detailed Overview

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

C routines (to be written):

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

Assembly routines:

```
void drawS( long sWidth, long sChar, long fillerChar, long borderChar );
int isEven( long value );
int isInRange( long minRange, long maxRange, long value, long exclusive );
int numOfDigits( long num, int base );
void printChar( char ch );
void printSegment( long character, long amount );
```

For the Milestone, you will need to complete:

`isEven.s` `isInRange.s` `numOfDigits.s`

Process Overview:

The following is an explanation of the main tasks of the assignment, broken into 3 parts.

1. Parse command-line arguments in `pa1.c`

There are 4 expected user inputs: `sWidth`, `sChar`, `fillerChar`, and `borderChar`. Within your main function in `pa1.c`, you will be parsing each command-line argument and checking if it is a valid input. Each argument must be converted from a string to a long (see `man strtol`), must be within a certain range, and the `sWidth` must be even.

a) Check for potential error conditions (invalid input, out of range, etc.). You will be using the functions `isInRange()` and `isEven()` for this error checking.

There are 6 possible errors that could be encountered:

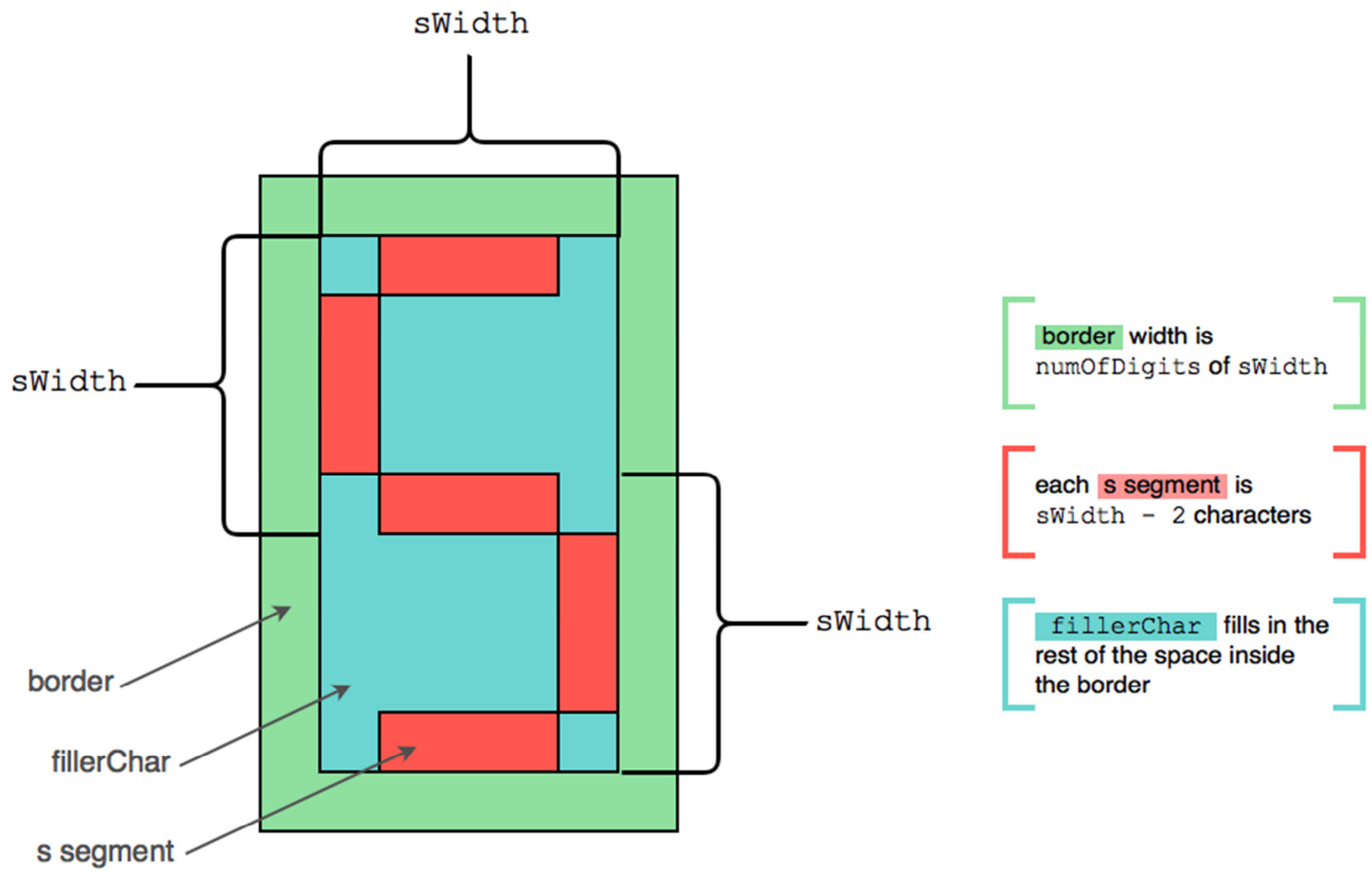
- `sWidth` is NOT even
- `sWidth` is not inclusively within the range of `S_WIDTH_MIN` and `S_WIDTH_MAX`
- `sChar, fillerChar, and borderChar` are not inclusively within the range of `ASCII_MIN` and `ASCII_MAX.`
- `sChar` is same value as `borderChar` or `fillerChar.`
- There was an error (`errno` was set) when converting `sWidth`, `sChar`, `fillerChar`, or `borderChar` to a long.
- The user input for `sWidth`, `sChar`, `fillerChar`, and `borderChar` is NOT an integer value (this is determined by checking `endptr`).

b) If any of the above errors are caught, print the appropriate error messages (located in `pa1Strings.h`) as the errors are encountered (see `pa1.c` below) and return. Otherwise, move on to step 2.

2. <u>Make the call to `drawS()`</u>

The `drawS()` function is the main driver of the program. It will provide all of the functionality for printing the 'S' character.

- `sWidth` refers to the total width of 'S' (the number of characters beginning from the leftmost side of 'S' to the rightmost side), not including the border.
- `sChar` is the ASCII value of the character that will represent the 'S'.
- `fillerChar` is the ASCII value of the character that will fill in the empty space inside the border.
- `borderChar` is the ASCII value of the character that will make up the rectangular border of the 'S'. The width of the border will be equal to the number of digits in `sWidth`.


sWidth
sWidth
sWidth
border
fillerChar
s segment
border width is numOfDigits of sWidth
each s segment is sWidth - 2 characters
fillerChar fills in the rest of the space inside the border

3. Logic in `drawS()`

- You will be given the C code equivalent of this function that can be translated directly into Assembly code in order to display the ‘S’ character. You will be writing several looping constructs to print the appropriate characters within each line.

a) Draw the top border using `borderChar` where the width of the border is the number of digits in `sWidth`. You will be making calls to `printSegment()` throughout your `drawS()` function in order to print a character a specific number of times.

b) To print each of the 3 horizontal segments of the S, you will first print the segment of `borderChar`. Next, the `fillerChar` will be inserted as the corners of the ‘S’ by making a call to `printChar()`. Then you will print `sChar` a total of `sWidth-2` times by calling `printSegment`(). Finally, the `fillerChar` and the `borderChar`s will be added in the same way as previously described.

c) The two middle parts of the ‘S’ character will then be printed within a loop that runs `sWidth-2` number of times. For the first middle part, `sChar` will be printed once using `printChar()`and the rest of the line will be filled with `fillerChar` using `printSegment()`. For the second middle part, the line will be filled with `fillerChar` using `printSegment()`and the last character `sChar` will be printed once using `printChar()`. (The border segment will similarly be printed at the beginning and end of each line like in part b).

d) Finally, the bottom border will be printed in the same way the top border was printed.

C Functions to be Written

Listed below are the modules to be written in C.

---

pa1.c

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

This function will drive the rest of the program. It will first perform input checking. If all input is valid, it will call `drawS()`. If any of the input checks fail, print the corresponding error strings which are located in the `pa1Strings.h` file. Keep in mind that all the error strings have format specifiers, so be sure to add the appropriate arguments when making the calls to `fprintf()`.

We will first check that the number of command line arguments entered by the user matches the `EXPECTED_ARGS`. If the user didn’t enter the correct number of arguments, print `STR_USAGE` to `stderr` and return `EXIT_FAILURE`.

Remember: Before parsing each command line argument, be sure to set the global variable `errno` equal to 0.

First, we convert the `sWidth` from `argv` to a long by using `strtol()`.

1. If `errno` has been set, use `snprintf` with the corresponding error message, `STR_CONVERTING`. Call `perror` with the error string that was filled by `snprintf` and continue to the next conversion.
2. If `errno` has not been set, check if `strtol` has completed successfully by checking what is stored in `endptr`. If there was an error, print `STR_NOTINT` to `stderr` and continue to the next conversion.
3. If there was not a parsing error, check if the width is in range. If the width is not within the min and max values specified in `pa1.h` (inclusive), print `STR_ERR_S_WIDTH_RANGE` and continue to step 4.
4. Check if `sWidth` is even, otherwise print the error message `STR_ERR_S_WIDTH_EVEN` to `stderr` and continue to the next conversion.

Second, we convert the `sChar` from `argv` to a long by taking the same steps 1-3 as above, but this time we use the `ASCII_MIN` and `ASCII_MAX` values to check if it is within range (inclusive). If any errors were encountered, print the appropriate error messages and continue to the next conversion.

Third, we convert the `fillerChar` from `argv` by taking the same steps we used to convert `sChar`, using the correct error strings when any errors occur. In addition, we need to check if `fillerChar` is the same as `sChar`. If they are the same, print the corresponding error string and continue to the next conversion.

Fourth, we convert `borderChar` by taking the same steps we used to convert `sChar`. In addition, we need to check if `borderChar` is the same as `sChar`. If they are the same, print the correct error string.

If any errors were encountered, return `EXIT_FAILURE`. Otherwise, call `drawS` using the converted values and return `EXIT_SUCCESS`.

Reasons for error:

- The number of command line arguments is not correct
- `errno` is set when parsing the `argv` values
- `endptr` points to a character other than the null character after converting a value
- `sWidth` is even or out of range
- `sChar`, `fillerChar`, or `borderChar` are out of range
- `sChar` is the same as `fillerChar` or `borderChar`

Return Value: If errors were encountered, return `EXIT_FAILURE`. Otherwise, return `EXIT_SUCCESS`.

Assembly Functions to be Written

Listed below are the modules to be written in Assembly.

---

drawS.s

```
void drawS( long sWidth, long sChar, long fillerChar, long borderChar );
```

This assembly module will perform the actual outputting of individual characters (via calls to `printSegment()` and `printChar()`) such that the 'S' character is displayed with the user-supplied values.

Here is the equivalent C version :

http://ieng9.ucsd.edu/~cs30x/pa1/drawS_template

All of the assembly constructs you will be using will have been covered in lecture and can be referenced in Chapter 2 of the textbook. You are not limited to using the above algorithm, but one of the purposes of this programming assignment is to learn to write looping/conditional constructs (branches), use the simple `.mul`/`.div`/`.rem` subroutines with parameter passing and return values, and perform simple arithmetic instructions (`inc`, `add`/`sub`) 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. 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.

Return Value: None

---

isEven.s
`int isEven( long value );`

This function determines whether or not the `value` is an even number.

Return Value: If `value` is even, return 1. If `value` is odd, return 0.

---

isInRange.s
`int isInRange( long minRange, long maxRange, long value, long exclusive );`

This assembly module will check if the `value` is within `minRange` and `maxRange`. If `exclusive` is 0, check the inclusive range; if `exclusive` is non-0, check the exclusive range. Return -1 if there is an error (see below), 1 if true, 0 otherwise. Remember that in C, 0 represents false and any other value represents true.

Error conditions:
- `minRange` is larger than `maxRange` → return -1

Return Value: If `minRange` is larger than `maxRange`, return -1. Otherwise, return 1 to represent true, 0 to represent false.

---

numOfDigits.s
`int numOfDigits( long num, int base );`

This function counts the number of `base` digits in `num`.
- First check to make sure the `base` is within the range [2, 36], inclusive by calling `isInRange()`. If `base` is not within this range, return -1.
- If `num` is 0, return 0
- Otherwise return the number of digits in `num`.

For example: If `base` is 10, the number -2693 has 4 digits.
If `base` is 2, the number 357 (101100101) has 9 digits.

Error conditions:
- `base` is not within the range [2, 36] inclusive → return -1

Return Value: If `base` is outside the defined range, return -1. If `num` is 0, return 0. Otherwise, return the number of digits in `num`.

---

printChar.s
`void printChar( char ch );`

This assembly module prints the character argument to stdout. This is very similar to the assembly module `printWelcome.s` given as part of PA0 -- use `printf()`. The difference is that `printChar()` just prints a single character (so think about how that might affect the format string).

Return Value: None.

---

printSegment.s
`void printSegment( long character, long amount );`

This assembly module utilizes `printChar()` to print `character` (to stdout) `amount` number of times.

Return Value: None.

Unit Testing

You are provided with only one basic unit test file for the Milestone function, `isInRange()`. 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 3 unit test files. 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 and running these tests. Keep in mind that your unit tests will not build until all required files for that specific unit test has been written. These test files are not being collected for the Milestone and will only be collected for the final turnin (however, they should already be written by the time you turn in the Milestone because you should be using them to test your Milestone functions).

Unit tests files you need to complete:

```
testisEven.c
testisInRange.c
testnumOfDigits.c
```

To compile:

```
$ make testisInRange
```

To run:

```
$ ./testisInRange
```

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

README File

Your README file for this and all assignments should contain:

- High level description of what your program does.
- How to compile it (be more specific than: just typing "make"--i.e., what directory should you be in?, where should the source files be?, etc.).
- How to run it (give an example).
- An example of normal output and where that normal output goes (stdout or a file or ???).
- An example of abnormal/error output and where that error output goes (stderr usually).

- How you tested your program (be more specific than diff'ing your output with the solution output--i.e., what are some specific test cases you tried?, what different types of cases did you test?, etc.).
- Anything else that you would want/need to communicate with someone who has not read the assignment write-up but may want to compile and run your program.
- Answers to questions (if there are any).

Questions to Answer in the README

Start gdb with your pa1 executable, then set a breakpoint at strtol.

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

```
run 9InchNails 999999999999999 55 123
```

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

1. How do you print the value of the string that is the 1st arg in strtol? (The value should be "9InchNails")
2. How do you print the decimal value of the base that is the 3rd arg in strtol? (The value should be 10)
3. How do you print the hex value of `&endptr` that is the 2nd arg in strtol? (The value should be something like 0xffbe---- (a high stack address - will vary))

Go to the next high level source instruction in main. This should be the next C instruction in main after the function call to strtol. Type `next`

4. How do you print the value returned by strtol? (The value should be 9) 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
5. How do you print the character `endptr` is pointing to? (Should be the character 'I')
6. How do you print the entire null-terminated string `endptr` is pointing to? (Should be "InchNails")
7. 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`

You should see the error message `"9InchNails" is not an integer` displayed by your program.

You should be back at the strtol breakpoint. Go to the next source instruction in main. Type `next`

It should be the source-level instruction after the call to strtol passing in 999999999999999 to convert to an int. 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 page for `errno` and section 2 intro. You can continue or quit.

The following questions pertain to Git:

8. What is the Git command to show the current state of your working tree?
9. What is the Git command to discard any changes made to a file since its last commit?
10. What is the Git command to display the differences between the local version of a file and the version last committed?

Academic Integrity:

11. 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 Total, 0.5 for each nop]** Eliminating nops in the sample assembly file.
  (1 nop cannot be removed - see the comment in `nops.s`)

Getting Started

Copy over `mainEC.c` and `nops.s` from the public directory:

```
[cs30xzzz@ieng9]:pa1$ cp ~/../public/mainEC.c ~/pa1
[cs30xzzz@ieng9]:pa1$ cp ~/../public/nops.s ~/pa1
```

Overview

You will be removing up to six nops in `nops.s` to perform assembly optimization--do NOT modify any other files. Note that there are seven nops total in the file, and there is a comment specifying which one should not be removed. Be sure to move the instruction you intend to fill the delay slot with into where the nop instruction was, do NOT just remove the nop.

In order to do this extra credit, you need a working version of `isEven.s`. A reference executable will not be provided, so you should use the non-optimized version of `nops.s` to test against your optimized version.

Compiling

Once you have a working version, you can compile the extra credit program:

```
[cs30xzzz@ieng9]:pa1$ gcc --o mainEC mainEC.c nops.s isEven.s
```

Example output

This program takes in a single positive integer as input, and prints a secret number based on the input.

```
$ ./mainEC
Usage:  ./mainEC num
        num - integer to calculate secret number for

$ ./mainEC 0
The secret number for 0 is 0

$ ./mainEC 1
   1: 0
The secret number for 1 is 0

$ ./mainEC 2
   1: 0
   2: 0
The secret number for 2 is 0

$ ./mainEC 3
   1: 0
   2: 0
   3: 2
```

```
The secret number for 3 is 2

$ ./mainEC 4
   1: 0
   2: 0
   3: 2
   4: 3
The secret number for 4 is 3

$ ./mainEC 5
   1: 0
   2: 0
   3: 2
   4: 3
   5: 7
The secret number for 5 is 7

$ ./mainEC 6
   1: 0
   2: 0
   3: 2
   4: 3
   5: 7
   6: 17
The secret number for 6 is 17

$ ./mainEC 17
   1: 0
   2: 0
   3: 2
   4: 3
   5: 7
   6: 17
   7: 23
   8: 80
   9: 88
  10: 396
  11: 406
  12: 2233
  13: 2245
  14: 14592
  15: 14606
  16: 109545
  17: 109561
The secret number for 17 is 109561
```

Milestone Turn-in Instructions

Milestone Turn-in - due Wednesday night, April 13 @ 11:59 pm [15 points of Correctness Section]

Before final and complete turnin of your assignment, you are required to turnin several modules for the Milestone check.

Files required for the Milestone:		
`isEven.s`	`isInRange.s`	`numOfDigits.s`

Each module must pass all of our unit tests in order to receive full credit.

A working Makefile with all the appropriate targets and any required header files must be turned in as well. All Makefile test cases for the milestone functions must compile successfully via the commands `make test***`.

In order for your files to be graded for the Milestone Check, you must use the milestone specific turnin script.

```
      $ cd ~/pa1
      $ cse30_pa1milestone_turnin
```

To verify your turn-in:

```
      $ cse30verify pa1milestone
```

Final Turn-in Instructions

Final Turn-in - due Tuesday night, April 19 @ 11:59 pm

Once you have checked your output, compiled, executed your code, and finished your README file (see above), you are ready to turn it in. Before you turn in your assignment, you should do `make clean` in order to remove all the object files, lint files, core dumps, and executables.

Files required for the Final Turn-in:

drawS.s	pa1.c	printSegment.s
isEven.s	pa1.h	Makefile
isInRange.s	pa1Strings.h	README
numOfDigits.s	printChar.s	
testisEven.c	testisInRange.c	testnumOfDigits.c
Extra Credit Files:		
mainEC.c	nops.s	

Use the above names *exactly* otherwise our Makefiles will not find your files.

How to Turn in an Assignment

Use the following turnin script to submit your full assignment before the due date as follows:

```
      $ cse30turnin pa1
```

To verify your turn-in:

```
      $ cse30verify pa1
```

Up until the due date, you can re-submit your assignment via the scripts above. Note, if you turned in the assignment early for extra credit and then turned it in again later (after the extra credit cutoff), you will no longer receive early turn-in credit.

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.

Style Requirements

You will be graded on style for all the programming assignments. The requirements are listed below. Read carefully, and if any of them need clarification do not hesitate to ask.

- Use reasonable comments to make your code clear and readable.
- Use file headers and function header blocks to describe the purpose of your programs and functions. Sample file/function headers are provided with PA0.
- Explicitly comment all the various registers that you use in your assembly code.
- In the assembly routines, you will have to give high level comments for the synthetic instructions, specifying what the instruction does.
- You should test your program to take care of invalid inputs like non-integers, strings, no inputs, etc. This is very important. Points will be taken off if your code doesn't handle exceptional cases of inputs.
- Use reasonable variable names.
- Error output goes to stderr. Normal output goes to stdout.
- Use #defines and assembly constants to make your code as general as possible.
- Use a local header file to hold common #defines, function prototypes, type definitions, etc., but not variable definitions.
- Judicious use of blank spaces around logical chunks of code makes your code easier to read and debug.
- Keep all lines less than 80 characters, split long lines if necessary.
- Use 2-4 spaces for each level of indenting in your C source code (do not use tab). Be consistent. Make sure all levels of indenting line up with the other lines at that level of indenting.
- Do use tabs in your Assembly source code.
- Always recompile and execute your program right before turning it in just in case you commented out some code by mistake.
- Do #include only the header files that you need and nothing more.
- Always macro guard your header files (#ifndef ... #endif).
- Never have hard-coded magic numbers (any number other than -1, 0, or 1 is a magic number). This means we shouldn't see magic constants sitting in your code. Use a #define if you must instead.