CSE220 Fall 2014 - Homework 6

Due Friday 10/31/2014 @ 6pm

The short version of the Unix philosophy is that we should develop small programs which can do one thing, and do that one thing well. We can then use these small and simple tools together to complete more complicated tasks.

In this assignment we will create a tool which accepts hexadecimal values representing MIPS instructions on stdin and produce some information about these instructions on stdout. The output should be formatted in such a way that we can then feed it to tools such as grep, sort, etc. This program will also accept command line arguments which can be used to adjust the type of information that is written to stdout. If your tool fails to operate correctly it should return an exit status. Two basic C exit status codes are defined in `stdlib.h`: `EXIT_SUCCESS` & `EXIT_FAILURE`.

The goal of this assignment is to have you write a basic, but useful C program. It should show you how a typical command line tool is usually designed, and get you familiar with using your program with other command line tools to complete more intricate and complex operations.

Corrections

- Changes from 10/26/2014 are in **purple**.
- Changes from 10/29/2014 are in **green**.

Detailed program operation

Your program will have 3 different output modes that you will have to implement. It also will need to support an option that displays a human readable header and a help option which will display to the user how to operate this program. To denote these output modes, we will use command line flags (similar to the flags used in HW5 for gcc).

- When given the command line flag `-h` your program should display the usage of this program.

> The name of the program in the help menu should be the name of the executable. You can find this by using the value of `argv[0]` in your main function. If the `-h` is provided it should always show the help menu and then exit no matter what other flags are provided. When `-h` is used, the program should return `EXIT_SUCCESS`.

Example output:

```
$ ./a.out -h
Usage: ./a.out [-himru]
    -h         Displays this help menu.
    -i         Displays statistics about instruction type usages.
    -m         Displays all the immediate values used in I-Type and J-Type
instructions.
    -r         Displays information about the registers.
    -u         Displays human readable headers for the different information
displayed.
               Should only be used with the -i, -r, and -m flags.
$ echo $?
0
```

```
$ ./a.out
Usage: ./a.out [-himru]
    -h         Displays this help menu.
    -i         Displays statistics about instruction type usages.
    -m         Displays all the immediate values used in I-Type and J-Type
instructions.
    -r         Displays information about the registers.
    -u         Displays human readable headers for the different information
displayed.
               Should only be used with the -i, -r, and -m flags.
$ echo $?
4
```

To display the exit code for the last command we use the tool `echo` to output text to the terminal. The shell variable `$?` contains the exit code of the last program ran.

> `$?` is an environment variable in bash which holds the error code. Most *nix systems, cygwin, and OSX default to bash. Unfortunately on Sparky the shell defaults to `csh` (you can switch by simply typing `bash`). You can see the return code in `csh` by using `echo $status`. If you are using the regular windows command prompt (not cygwin) you can view the return code by using `echo %errorlevel%`. If you are using another shell, its up to you to figure out where this return code is stored if you want it.

If no flags are provided the help option should also be displayed and with an error code of

`ERROR_FLAGS`.

If an invalid combination of flags is given (eg. `-rim`) display the `-h` help menu and exit with `ERROR_FLAGS`.

- When given the command line flag `-i` your program will produce some simple statistics in the following format. Each line will contain the instruction type, the count of this instruction type given in the input, and the percentage of this instruction type (based on the total number of instructions provided in the input). Upon successful completion the program should return `EXIT_SUCCESS`. If it fails due to invalid input your program should print nothing and return `ERROR_INSTR` (see Fail Conditions section).

Example output:

```
$ cat instructions.txt | ./a.out -i
R-type      56      46.7%
I-type      42      35.0%
J-type      22      18.3%
$ echo $?
0
```

- When given the command line flag `-r` your program should list out the statistics of each register which includes how many times it was used in total, how many r-type instructions used it, how many i-type instructions used it, how many j-type instructions used it, and the percentage of times the register was being used based on all the other registers. Upon successful completion the program it should return `EXIT_SUCCESS`. If it fails due to invalid input your program should print nothing and return `ERROR_REG` (see Fail Conditions section).

Example output:

```
$ cat instructions.txt | ./a.out -r
$0     67      53      14     0        14.3%
$1     12       8       4     0        4.5%
... Removed rest of output for brevity
$31     4       4       0     0        1.2%
$ echo $?
0
```

- When given the command line flag `-m` your program should list the immediate values used in the i-type instructions and the address values used in the j-type instructions as hex numbers. Immediate values and address values will be intermixed. Printout each immediate/address values as you process each instruction. If an invalid input is found, stop printing and return `ERROR_IMMEDIATE` (see Fail Conditions section). Upon successful completion the program should return `EXIT_SUCCESS`.

Example output:

```
$ cat instructions.txt | ./a.out -m
0x123456
0x2a
0xe92c
...
$ echo $?
0
```

- When given the command line flag `-u` the printout will display a header labeling what the columns for the output mean. This flag can ONLY be used in conjunction with another flag. **Note:** this is only useful for humans, the `-u` flag should not be provided when giving your output to another program. If `-u` is provided but `-i`, `-m`, or `-r` is not provided your program should display the help menu and exit with the return code ~~`ERROR_HUMAN`~~ **`ERROR_FLAGS`** (see Fail Conditions section). **Flags can come in any order combined or separate.**

Example output:

```
$ cat instructions.txt | ./a.out -i -u
  TYPE   COUNT    PERCENT
R-type      56      46.7%
...
$ echo $?
0
```

Another example with the flags combined:

```
$ cat instructions.txt | ./a.out -ru
REG   USE  R-TYPE  I-TYPE J-TYPE     PERCENT
$0     67      53      14      0       14.3%
...
$ echo $?
0
```

```
$ cat instructions.txt | ./a.out -um
IMMEDIATE-VALUE
0x123456
0x2A
0xE92C
...
$ echo $?
0
```

Example of incorrect usage:

```
$ cat instructions.txt | ./a.out -u
Usage: ./a.out [-himru]
    -h         Displays this help menu
...
$ echo $?
5
```

> Your output should match the sample output formatting as closely as possible. Use the printf format specifiers to define the widths of the columns. All columns should be separated by spaces or tabs only.

Command line arguments

It is typical for command line programs to use what is called short ops (this is the `-r`, `-m`, `-u`, etc.). You will need to use the `argc` and `argv` variables passed into the main program to determine which options should be used. There is a helper library which exists that you can use called getopt. You do not have to use this library, but if you do it is up to you to figure out how to use it. It will handle all the complex command line argument positions though, which may make it worth your time.

Fail conditions

Your program should fail if an invalid input was provided in the input file: an invalid hexadecimal number Eg. `0xzbq3`, the file provided to your program is empty, the `-u` flag is provided without the `-r`, `-i`, or `-m` flags, or the program was executed without any flags provided.

The `-r`, `-i`, and `-m` flags should not be used together. If they are your program should also terminate with a `ERROR_FLAGS` value.

Clarifications from piazza

- When a program fails due to a a illegal combination of flags, your program should print out nothing unless stated otherwise and return `ERROR_FLAG`.
- When a program fails due to an invalid hex instruction you should return the ERROR flag for whichever mode you are currently in (`-r`, `-i`, or `-m`)and print nothing.
- When the user provides an invalid flag you should return `ERROR_FLAG`.
- You do not have to check if the value of an instruction is itself a correct instruction **BUT** all valid MIPS instructions are 32-bits. So if you parse an instruction that is less than 32-bits or more than

32-bits this is an invalid input.

Getting started

- Create a new c program file.
- At the top of your program in comments put your name and id.

```
/*
Homework #6
name: MY_NAME
sbuid: MY_SBU_ID
*/
```

> See Piazza post @366 for an example C program to help you get started. This is an example program, but the main program and headers will help you get started if you are unsure of what to do.

Define Error Codes

Define the following error codes as preprocessor constants in your program. You should use these constants as the return values.

1 - ERROR_REG
2 - ERROR_INSTR
3 - ERROR_IMMEDIATE
4 - ERROR_FLAGS
~~5 - ERROR_HUMAN~~ This should be replaced by **ERROR_FLAGS** anywhere in your code.

Input to your program

Your program will take as input via `stdin` the hexadecimal representation of MIPS instructions (one instruction per line).

Input Example File:

```
0x3c011001
0x3424024f
0x24020004
0x0000000c
0x24020005
0x0000000c
0x24080001
0x24090002
0x240a0003
0x1048000f
0x1049000e
0x104a017e
0x00404025
0x3c011001
```

> ℹ You should use the scanf format `%x` to convert the hex strings into integers.
>
> ℹ You can create your own input file with your own MIPS instructions using MARS. Take any Mips program, and assemble it. After assembling press the command `ctrl-d`. This should pop up a menu about dumping a section of your program. For the **memory segment** select the *.text* section and for the **dump format** select *Hexadecimal Text*. Finally select *Dump to File…* which will ask you were you want to save your instruction dump.
>
> ℹ You will have to use C bitwise operations after parsing the hex values to figure out what type of instructions they are.

Your program will then parse the instructions to gather the statistical data required to create the data tables shown with each flag option.

To provide the input to your program, you will need to redirect the instructions from a file to the stdin of your program. This can be done like: `$ ./a.out -r < instructions.txt`

Alternatively you can use the pipe operator to send the output of another program (such as `cat`) to the stdin of your program. This can be done like: `$ cat instructions.txt | ./a.out -r`

MIPS Instruction Format Reference

As previously discussed, the MIPS instruction formats are as follows:

R-type:

OPCODE	RS	RT	RD	SHAMT	FUNC
000000	5-BITS	5-BITS	5-BITS	5-BITS	6-BITS

I-type:

OPCODE	RS	RT	IMMEDIATE
6-BITS	5-BITS	5-BITS	16-BITS

> Opcodes for Immediate instructions are any values other than 0, 2, or 3, which are used for the other instruction formats.

J-type:

OPCODE	ADDRESS
000010	26-BITS
000011	26-BITS

> For the `-m` flag, you should include the ADDRESS field in the J-type instructions as well as the IMMEDIATE field in the I-type.
>
> Registers can be specified in the RS, RT and RD fields of the instructions. You will need to mask out the values of each of these fields to collect the statistics about register usage.
>
> A sheet containing all the MIPS reference data has been provided for you on piazza in the resources section.

Combining your program with other UNIX tools

We will run your program along with a number of other UNIX tools and check the results.

wc
grep
sort
head
tail

Sample usage of these tools in combination with your program are shown below:

Assume that your program produces the following output:

```
$ ./a.out -i < instructions.txt
I-type      42      35.0%
J-type      22      18.3%
R-type      56      46.7%
```

Now lets chain this together with the `sort` program. We will sort the rows based on the **numeric** values in the second column.

```
$ ./a.out -i < instructions.txt |  sort -k2n
J-type      22      18.3%
I-type      42      35.0%
R-type      56      46.7%
```

Now lets use `grep` to search for a value. We will sort the rows based on the **numeric** values in the second column in descending order and then use grep to look for any row that contains the value 2.

```
$ ./a.out -i < instructions.txt |  sort -k2nr | grep 2
I-type      42      35.0%
J-type      22      18.3%
```

Now lets use `wc` to count how many results we have:

```
$ ./a.out -i < instructions.txt |  sort -k2nr | grep 2 | wc -l
2
```

> ⚠ Just because we showed these commands, it does not mean this is the only way we will test your program. This is just a good way to gauge that your program is working correctly but it is **NOT** the end all be all. Test your program appropriately.

Hand-in instructions

Before submitting your assignment make sure your program will compile using `gcc` with no warnings and errors using the `-Wall` and `-Werror` flags.

See Sparky Submission Instructions on piazza for hand-in instructions.

> When writing your program try to comment as much as possible. Try to stay consistent with your formatting. It is much easier for your TA and the professor to help you if we can figure out what your code does quickly. If your program doesn't work correctly the grader may be able to give you partial points, assuming he can read the code that you have written.