CS24 Assignment 2

The files for this assignment are provided in the archive **`cs24hw2.tgz`** on the course website. The archive contains a top-level directory **`cs24hw2`**, in which all other files are contained. Download and unpack the archive, then complete your work in this **`cs24hw2`** directory, being careful to put your answers in the locations specified in the assignment.

When you have finished the assignment, you can re-archive this directory into a tarball:

*(From the directory containing **`cs24hw2`**; replace **`username`** with your username.)*

```
tar -czvf cs24hw2-username.tgz cs24hw2
```

Then, submit the resulting file through the course website.

As mentioned before, failure to follow the specified filenames, or submitting an otherwise wonky tarball, will result in point deductions. Always make things easy for your grader. ☺

Note: All section references and problem numbers are from the second edition of CS:APP (typically indicated as CS:APP2e). If the first edition is not materially different, a reference will usually be included. **However, all book problems are from the 2nd edition of the book!**

Part 1: IA32 Assembly Language (45 points)

1. What does the following assembly code do? For example, if `8(%ebp)` is *x*, what does the code compute? ***(5 points)***

```
movl    8(%ebp), %eax
imull   $3, %eax
addl    $12, %eax
imull   8(%ebp), %eax
subl    $17, %eax
```

   Put your answer in: **`cs24hw2/prob1.txt`**

2. Compile the code below (provided as **`example.c`**) to symbolic assembly code. Identify where each of the three C-level operations (+, *, -) is performed in the resulting assembly code. ***(5 points)***

   - To compile a file to assembly, use: **`gcc -O2 -S <file>`**
     The **`-O2`** (capital-o) argument applies some optimizations to the generated code. (If you compile on 64-bit Linux (vs. 32-bit Linux), add **`-m32`** as an argument.)
   - Copy **`example.s`** to **`example.annotated.s`**, and make your annotations in that file.
   - CS:APP2e §3.2 also reviews various ways to produce assembly code from C code.

`example.c`:

```
int ex(int a, int b, int c, int d) {
    int res;
    res = a * (b - c) + d;
    return res;
}
```

3. Read CS:APP2e section 3.5.1, then do Practice Problem 3.6, p.178. ***(5 points)***

   Because this is a practice problem, the answer is in the back of the chapter, but *do not look at the answer until you have worked through the entire question.* If you want to check your answers when you are finished, this is fine, but you need to include all your work to get full credit for the problem.

   Make sure to do the version of the problem from the US version of CS:APP2e, as there may be variations in the problem. A PDF scan of the problem will be provided.

   Put your answers and work in the file **`leal.txt`**

4. Read CS:APP2e section 3.6.7 (CS:APP section 3.6.6), then do Problem 3.59, pp.298-299. Make sure to explain the process by which you come to your answer. ***(10 points)***

   Make sure to do the version of the problem from the US version of CS:APP2e, as there may be variations in the problem. A PDF scan of the problem will be provided.

   Put your answers and work in the file **`prob3.59.txt`**

5. The file **`math.s`**, included in the tarball, contains the implementation of three basic, very common math operations. ***(20 points)***

   Copy **`math.s`** to **`math.annotated.s`**, and complete the following:

   - In the comment-header of each function, explain what common math operation the function performs.
   - Annotate the function's assembly code, explaining exactly how the operation is performed.

   Finally, in the comment-block at the top of **`math.annotated.s`**, answer this question:

   > What is the "common theme" of all three implementations? Specifically, consider a naïve implementation of each of these operations, and note what kinds of operations all three of these implementations avoid. Why would the compiler want to avoid simply using a naïve implementations of these functions?

   Hint: You will find it particularly helpful to read CS:APP2e section 3.6.6.

Part 2: IA32 Subroutine Calls (55 points)

1. We provide the function **`fact.s`** in assembly. Call **`fact.s`** from a C program. We provide a **`Makefile`** to build a top-level executable **`factmain`**. ***(10 points)***

   - Provide a file **`factmain.c`** from which you call the **`fact.s`** assembly function and print the result. The program should take a single integer command-line argument, to pass to the factorial function.
   - Make sure your **`factmain.c`** program performs basic verification of its command-line. Specifically:
     - Verify that the program receives one command-line argument
     - Verify that the numeric argument is nonnegative
     - You don't have to verify that the argument is in fact an integer; you can assume this. (If you want to be extra clever, use **`strtol()`** instead of **`atoi()`** to verify that the argument is an integer. This is not required.)

     Report a suitably descriptive error message to the console if the program receives invalid input.

   See the appendix of this assignment if you are unfamiliar with how to parse command-line arguments in C.

2. Write up the contents of the stack from the computation of **`fact(3)`**. ***(15 points)***

   Assume that the stack pointer is at 0x1000 when you enter the **`fact`** function the first time (i.e. immediately after the first **`call`** to **`fact`** has been completed). Your answer should represent the contents of the stack through to the point where the argument to **`fact`** is 0, and execution of **`fact(0)`** has reached the **`fact_return`** label.

   Show the stack frame from 0x1008 to as far down as it is defined for the deepest call you have executed in this sequence. For each stack element, indicate its address, its value, and its logical role (e.g. which argument, stack pointer, base of frame pointer, etc.). Assume the code addresses: **`fact`**=0x2000, **`fact_continue`**=0x2020, **`fact_resume`**=0x2030, **`fact_return`**=0x2040.

   There will be some values on the stack that you do not know. You can leave these slots of the stack empty, but make sure to annotate their purpose.

   Provide your resulting stack outline in **`stackframe.txt`**

3. Write a recursive Greatest Common Divisor (GCD) routine based on Euclid's algorithm in assembly and a C function to call it and print the result. ***(30 points)***

   Your implementation must be recursive. (Yes, an iterative solution would be faster, but the whole point is to practice cdecl in the context of recursion.)

You can find a review of Euclid's algorithm in SICP[1]:

http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-11.html#%25_sec_1.2.5

- Provide a file **`gcd.s`** with your recursive GCD routine. Make sure your routine is commented so it is easy to understand.

- Provide a file **`gcdmain.c`** with your C main routine, which takes two integer command line arguments, passes these values to your assembly GCD routine in **`gcd.s`**, and prints the result. The program should print a suitably descriptive error message if it doesn't receive the correct number of arguments, or if any argument is ≤ 0. It is not required to report an error if the arguments are not valid integers (i.e. it may assume that it will be given integers.)

Your GCD routine need only work for non-negative arguments. Your GCD assembly code may assume that the arguments are in a specified order (e.g. largest argument first), if you wish. Your C code should then include code that switches the arguments, if necessary.

Appendix: C Command-Line Arguments

Given a C program with **`main`** function declared like this:

```
int main(int argc, char **argv) {
    ...
}
```

The command-line arguments are available in the **`argc`** and **`argv`** arguments. The **`argc`** parameter is the number of arguments, and is always at least 1. The **`argv`** argument is an array of **`char*`** values (zero-terminated strings), each of which is an argument. The value **`argv[0]`** is always the name used to invoke the program; thus, the actual arguments to the program start with **`argv[1]`** and so forth. Finally, **`argv[argc]`** is always set to **`NULL`**.

A simple function to convert a string to an integer is the **`int atoi(char *nptr)`** function. This function doesn't indicate whether the input was a valid integer, so you don't need to worry about detecting when inputs are an invalid format. (If you wish to be more clever, you should look at the **`strtol()`** function, but its use is not required in HW2.)

[1] Structure and Interpretation of Computer Programs by Abelson and Sussman