CS 25000 Lab 07 – Due at the end of your lab session this week. Work in teams of two (or three if an odd number of students attend lab), **but upload individual versions of this document to Gradescope.**

- Delete nothing from this file.
- Edit this file to add your typewritten answers to each question.
- When your answer includes a diagram make sure that it is clear and large enough to read.
- Ensure that your answer fits on the same page as its question.
- If you change the pagination of this file or if your complete answer to a question does not fit on the page with that question, then you may receive a lower score.
- Export your completed Word file to PDF.
- Upload your PDF file to Gradescope.com. It is your responsibility to upload this assignment to its correct place in Gradescope. You may upload multiple times. Your final upload will be scored. Use the download capability to check your upload.
- Uploading will be blocked after the due time (plus grace period).
- Max score = 20 points; 2 points per question.
- The above directions apply for all assignments uploaded to Gradescope.

- Why should your answer be on the same page with its question?
  Answer: Gradescope has been programmed to expect it. This allows Gradescope to automatically display each answer for scoring.

Experiment 1: The C Preprocessor

Tell gcc to stop just after generating the hello.i file with the command

```
gcc -E -P hello.c -o hello.i
```

Use the command `more hello.i` to look at how typedef and extern definitions from stdio.h have been placed inline into your code. Notice also near the very end of hello.i that the function sum has been partially evaluated, substituting in the definition of SUM() and also the constant values for X and Y. A well-designed compiler will take an opportunity to replace a source code computation for which all operands are available at compile time with the result of that computation. The compiler is shifting this computation time out of the time needed to execute the program, incurred every time the program is run, and into the time needed to compile the program, which is often done fewer times than the program will be executed. This strategy saves time overall.

Question 1. Which lines of code in hello.c do not appear in hello.i? Which lines of code in hello.c appears in slightly different form in hello.i? Ignore blank hello.c lines.

The define statements, the comments, and the include statement do not appear in hello.i

```
  printf("SUM=%d\n", SUM(X,Y));
```

SUM(X,Y) is being replaced with (4 +2), this line is the one that appears slightly different because the sum is being partially evaluated

Experiment 2: The Compiler

Prove that hello.c is not actually a C program by trying to compile hello.c. To do this, first use the copy command, cp,

```
cp hello.c hello.i
```

to create a file containing the contents of hello.c but with the name hello.i, a name that will trick gcc into assuming that it has been given a file containing a C language program rather than a C source file. Then tell gcc to compile hello.i (containing the text of hello.c) using the command

```
gcc hello.i -o hello
```

Question 2. What output do you receive from gcc?

Errors, there are not allowed symbols in this file.

```
hello.i:3:1: error: stray '#' in program
    3 | #include <stdio.h>
      | ^
hello.i:3:10: error: expected '=', ',', ';', 'asm' or
'__attribute__' before '<' token
    3 | #include <stdio.h>
      |          ^
```

Repeat the command

`gcc -E -P hello.c -o hello.i`

to restore hello.i to containing a C program.

Use the command `gcc -S hello.i -o hello.s` to have gcc translate the C program into an assembly language program and stop. `hello.s` contains an assembly language program for the computer known as soji.cs.purdue.edu. Use the `arch` command to print the name of the ISA of soji.cs.purdue.edu.

Question 3. What company created this ISA? Hint: Search Wikipedia.org using the ISA name as the keyword.

Arm Holdings created Aarch64

Examine the assembly language program in `hello.s` using the `more` or `cat` commands. In an assembly language program, by tradition labels start at the beginning of a line and end with the colon character.

Question 4. Make a two-column table. Name the columns "Label" and "Origin". In the first column list all the labels that appear in hello.s in their order of appearance. In the second column name the origin of each label using only words that appear in Figure 4.6 (see above).

LABEL	ORIGIN
.LC0	compiler
.LC1	compiler
main	source code
.LFB0	compiler
.LFE0	compiler

In hello.s character strings that begin with the period character and do not end with the colon character are directives to the assembler program, the next step in producing an object file. Find the line in hello.s with the .section .rodata pair of directives. .section tells the assembler to create a segment of memory in the object file to hold .rodata. .rodata tells the assembler that what follows up to the next .text directive is the read-only data of hello.s. Read-only data is inherently constant data. The gcc compiler creates a unique label to point to each item of constant data by generating a sequence of text strings of the form ".L" followed by "C0" for the first, the zeroth, constant, then "C1" for the next constant, and so on through "Cj" for as many integers j as needed to name all of the read-only constants. The .string directive tells the assembler that what follows is to be placed in memory as a null-terminated character string.

Question 5. What items comprise the read-only data of hello.s?

.LC0 and .LC1 comprise the read-only data of hello.s, because they are format strings that do not get changed

Question 6. Find the first `bl` assembly language instruction in the text section. `bl` stands for Branch with Link and is the assembly language instruction to call C library functions. The first `bl` causes "Hello world" to be printed to the screen. Does gcc use printf() to carry out this action? If not, what C library function is used? Place each answer on a separate line to make it easy for the TAs to read your answers.

No it does not.

puts() is used instead of printf()

Question 7. Find the second `bl` in the text section. What function does it call? Could puts() be used instead? Why? If puts() is a faster library routine than is printf(), what is the compiler doing substituting puts() for printf() in the first instance of printf() in hello.c? Place each answer on a separate line.

It calls printf()
No, because puts() can only take one argument, and the SUM=%d\n string needs more arguments

It is optimizing the runtime because puts() works as well as printf() and faster.

Experiment 3: The Assembler

The assembler translates the assembly instructions in ASCII in hello.s into an object file named hello.o that has the assembly instructions in binary. The object file is not yet executable because it has references to functions that are not defined in the object file, such as calls to printf().

The assembler is invoked by gcc when compiling.

To generate an object file, type the following command:

gcc -c hello.c

This command generates the file hello.o. You may examine the contents with the command nm. Type the command:

nm hello.o

The nm command prints the symbol table, a table of pointers, for hello.o. Each symbol (abstract pointer) in the table

1. may have a value (an address) at this time (after assembly but before linking) which is given in hexadecimal notation
2. has a symbol type character that indicates whether the symbol is undefined (U) or part of the text section of the program (T),
3. and has a name.

Most undefined symbols will become defined by the linker. The value of a symbol is the number of bytes from the beginning of its program section where the label will be found.

Question 8. Correctly place a copy of the output from the command nm hello.o in the table here. Add comments to the right of each line delimited by a ";" character and saying the origin of the symbol. The origin will either be one of the family of "hello" files created up to this point or a word appearing in Figure 4.6.
Answer:

Value (Address)	Symbol type	Symbol name	; Comment on origin
0000000000000000	T	main	; hello.c
	U	printf	; hello.c
	U	puts	; compiler

Experiment 4: The Linker

The linker is a program that takes object files and static libraries as input and combines them together into an object file. The linker assigns starting addresses to each object file, and then assigns an address to each defined symbol. The linker then stores the address of a defined symbol in the machine instructions of the object files that have this symbol as undefined.

To generate an executable, use the command:

```
gcc hello.c -o hello
```

You can use the nm command to find the symbols that are defined and undefined in `hello`.

Question 9. What is the value of the symbol main now? In decimal, how many bytes of information are there in the object file hello before the start of the hello.c program at symbol main?

```
00000000000007d8 T main
```

2008 bytes of information

The addresses in memory of shared library routines are not known by the linker. The loader makes these final connections at the time the object file hello is loaded into memory when we command it to be run with the shell command `./hello`.

Question 10. What functions does nm -n hello reveal to be shared library functions?
w _ITM_deregisterTMCloneTable

```
U __libc_start_main@GLIBC_2.34
                 U abort@GLIBC_2.17
                 U printf@GLIBC_2.17
                 U puts@GLIBC_2.17
```