Problem 1: Short Answer *[10 points]*

Scribe: **[Scribe's name here]**

1. While waiting in office hours, you overhear your peer mention that, while they can work through caller-callee save register problems in the homework, they don't understand the motivation in the first place. In your own words (about 30 words or fewer), explain why saving/restoring register values with function calls is necessary, and what would happen if we didn't do it.

   There are a limited number of registers and these registers are shared between different function calls, so values need to be saved / restored to memory to ensure correctness.

2. The student thanks you for engaging in such wonderful peer instruction. They follow up with a question on linking, asking why accesses to global variables defined in a file (with no "extern" keyword used) need to be placed in the relocation table. They don't seem to really get what the relocation table is used for. Explain why this is needed, and a possible problem that might arise if we did not place such instructions in the table.

   After linking, each corresponding data/text/global sections are combined. This means that the original memory address of the global variable, even though defined in the same file, will likely be moved to a different memory location.

Problem 2: Link Like You've Never Linked Before! *[20 Points]*

Scribe: **[Scribe's name here]**

Read the following two files and fill in the rest of the symbol and relocation tables for each (on the next page). **Note that not all entry spaces in the tables provided need to be used.** We have completed one entry in each symbol and relocation table to show the format of each entry. Hint: see the Symbol and Relocation table guide on the website.

main.c

```
#include <string.h>
#include <stdlib.h>
#include <dog.h>

#define DOG_CNT 500 // Hint: see link

int tricks = 0;
int pet();
extern int bark(int dog);
extern int total_barks;
extern int dog_happiness [500];


int main(){
   for(int i = 0; i<DOG_CNT; i++){
      dog_happiness[i] = 1;
      pet(i);
   }
   char out [50];

}

void pet(int i){
   static int petted_dogs = 0;

      dog_happiness[i] *= 2;
      petted_dogs += 1;

      if (rand() & 1) {
        bark(i);
      }

}
```

dog.c

```
#include <stdlib.h>


extern int tricks;
int total_barks = 10;


void bark(int dog){
   total_barks += 1;


}
```

main.o symbol table		*dog.o symbol table*	
Symbol	***Type (T/D/U)***	***Symbol***	***Type (T/D/U)***
tricks	*D*	tricks	U
pet	T	total_barks	D
bark	U	bark	T
total_barks	U		
dog_happiness	U		
main	T		
rand	U		
static petted_dogs	D		

main.o relocation table			*dog.o relocation table*		
Line	***Instruction (LD, ST, BL)***	***Symbol***	***Line***	***Instruction (LD, ST, BL)***	***Symbol***
16	*STUR*	*dog_happiness*	9	STUR	total_barks
17	BL	pet	9	LDUR	total_barks
26	STUR	dog_happiness			
26	LDUR	dog_happiness			
27	STUR	petted_dogs			
27	LDUR	petted_dogs			
29	BL	rand			
30	BL	bark			

Entries in blue are not required. Statics don't have to be in the symbol table, but if they are they must be **explicitly** marked as static to avoid global naming conflicts. BLs to functions in the same file don't need to be relocated if the branches in the ISA jump to PC-relative offsets, like beq in LC2K or BL in ARM. jalr would be an example of a branch that uses an absolute address.

Underline indicates extern symbols that are unused; the corresponding symbol table entries may or may not be included (depending on the implementation of symbol table). Since there is no clear definition, no points are taken if not included.

Problem 3: LC2K ABI *[20 points]*
Scribe: **[Scribe's name here]**

Fill in the blanks of the LC2K assembly code to match the provided C code. Some of the blanks are already filled in for you—purple denotes where registers are being saved and red denotes the function call to `mystery`. The assembly should follow the LC2K ABI shown below:

- r0: always 0
- r1 - r2: function arguments, caller-save
- r3 - r4, r6: caller-save
- r5: stack pointer
- r7: return address, caller-save

Hint: Assume the stack grows "up." If you want to store a new variable on the stack, you should write it to an address offset by r5, and then increment r5 by the appropriate amount. Ensure you decrement r5 back to its original value before returning from the function.

```
void start() {
      int a = -1, b = 2, c = 3;
      for (int i = 3; i != 0; --i) {
            mystery(i, a);
            c = i + b;
      }
}
```

These questions will help you fill out the below *[5 points]:*

- Which variables are live across the function call to `mystery`?

  a, b, and i (c is not live)

- Read the hint above. Why does r5 need to be decremented back to its original value?

  So the calling function can access its variables locally.

Fill in the blanks below *[15 points]*:

```
Start lw    0     2     neg1  //a
      lw    0     3     one   //inc
      add   3     3     4     //b
      add   3     4     6     //c
      add   0     6     1     //i
loop  beq   1     0     end
      lw    0     3     one   //inc
      sw    5     7     Stack //start saving variables to stack
      add   5     3     5     //increment stack pointer
      sw    5     1     Stack
      add   5     3     5
      sw    5     2     Stack
      add   5     3     5
      sw    5     4     Stack
      add   5     3     5
      lw    0     6     fcall
      jalr  6     7           // jump to Mystery function
      lw    0     3     neg1
      add   5     3     5
      lw    5     4     Stack
      add   5     3     5
      lw    5     2     Stack
      add   5     3     5
      lw    5     1     Stack
      add   5     3     5
      lw    5     7     Stack
      add   1     4     6
      add   1     3     1
      beq   0     0     loop
end   jalr  7     6
fcall .fill Mystery
one   .fill 1
neg1  .fill -1
```