Name:

Q1: /20

Q2: /30

Q3: /24

Q4: /26

Total: /100
For functional call related questions, let’s assume the following convention is used when implementing function calls (which is exactly the same as we discussed in class):

1. The stack grows downwards unless specified otherwise.
2. The stack contains the following items, in the order that they are allocated on the stack:
   - Return address
   - Frame Pointer
   - Input Parameters
   - Return Value
   - Local Variables of the Callee
3. All parameters are passed via the stack (none are passed via registers)
4. Caller only allocates part of the activation frame for the callee before calling 'jal' - it only allocates until the return value (including return value) and updates the stack pointer accordingly (so it points to the return value).
5. Caller pushes input parameters onto the stack in the reverse order as the parameters are declared.
6. Callee is responsible of populating the rest of the activation frame (just the local variables).
7. Callee is responsible of deallocating the local variable part of the activation frame when it returns
8. Caller is responsible of deallocating the rest of the callee’s activation frame.

In addition to the above, we also have the following:

9. The local variables are allocated in the same order as they were declared.
10. Stack pointer points to the last occupied slot on the stack.
Question 1 stack (20 pts)

!! This question does not use the convention defined in page 2 !!

In our class, we learned that we will use the stack to pass input parameters into functions, track return addresses, collect return values, and hold local variables. In the classroom examples, we always assume that the stack grows towards the lower end of address spaces. However, it is perfectly fine to design a system where the stack grows upwards.

Suppose you are given the following system:

1. The stack grows in one direction, but we do not know which direction yet.
2. You can only program in C.
3. You cannot operate on the stack directly (i.e. only function calls in your program can change content of the stack, you cannot push/pop the stack directly).
4. The activation frame contains the following items:
   i. Return address
   ii. Frame Pointer
   iii. Input Parameters
   iv. Return Value
   v. Local Variables of the Callee
5. However, we do not know anything about the order that items within an activation frame are placed, e.g., the return address may be at the bottom of the activation frame, or input parameters may be pushed in a random order – they may even be placed after the local variables, or the local variables may be allocated not in program order, etc.

Write a short C function to detect whether the stack grows downwards or upwards. The function needs to return 0 when it’s downwards, and 1 otherwise.
**Question 2 30 points**
The following variables are allocated in the memory beginning at address 2000 on a **64-bit** system. Note that the allocation goes towards the **upper end** of the address space.

Part A: (6 pts) We know the following data type sizes already:

- **char**: 1 byte
- **float**: 4 bytes
- **int**: 4 bytes

What are the sizes of the following data types?

- **char** *
- **float** *
- **int** *

Part B (16 pts): Complete the memory map below with the variable names at each word or byte in memory. Do not include the variable's value. For **B[4]**, you need to specify the mapping for each of the 4 elements in array B.

![Memory Map](image)

The variables are allocated in the order that they are defined. The allocation scheme tries not to leave gaps in the memory unless necessary (e.g. no gaps between char data types). **Watch for alignment**.

Part C (8 pts): using the values in part B, give the value of each expression.

- **p1**
- ***p1**
- **&p1**
- **p2**
- ***p2**
- **&p2**
- **p2[2]**
- **s[3]**
Question 3: 24 points

Part A: (6 points) Given the following C code:

```c
#include <stdio.h>
#include <stdlib.h>
int foo(int x) {
    x = x + 1;
    return 0;
}

int main() {
    int i = 100;
    foo(i);
    printf("%d\n", i);
    return 0;
}
```

What will `main()` print?

Part B: (10 points) Given the following C code

```c
#include <stdio.h>
#include <stdlib.h>
int foo(int *x) {
    *x = *x + 1;
    return 0;
}

int main() {
    int i = 100;
    foo(&i);
    printf("%d\n", i);
    return 0;
}
```

Will the code compile? If yes, what will `main()` print? If not, Why?
Part C: (8 points) Given the following C code

```c
#include <stdio.h>
#include <stdlib.h>
int foo(char *a) {
    char b = 'Y';
    *a = 'X';
    attention:          // a = &b;
    printf("%c", *a);
    return 0;
}

int main() {
    char str[] = "ABCDE";
    foo(str);
    printf("%c", str[0]);
}
```

The line labeled “attention” changes the value of a in foo(), does this change the value of str in main()?

What will printf() in foo() print? What will printf() in main() print?
Question 4: 26 points

Given the following C function:

```c
int Foo (int n) {
    int a;
    a = Bar(0);
    return a;
}
```

Implement the function Foo() in MIPS. You need to follow the comment exactly. Add labels as necessary. You need to use the convention on the first page.

Also, it is caller’s responsibility to save its frame pointer onto the stack before calling a function, and to set the frame pointer register for the callee. However, it is the callee’s responsibility to restore caller’s frame pointer when the callee returns.

<table>
<thead>
<tr>
<th>Label</th>
<th>Instruction</th>
<th>comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foo</td>
<td></td>
<td># reserve return address&lt;br&gt;# allocate local variable a on the stack</td>
</tr>
<tr>
<td></td>
<td></td>
<td># Allocate the activation frame for Bar (caller’s part)&lt;br&gt;# Save Foo’s Frame Pointer&lt;br&gt;# Set Frame Pointer for Bar</td>
</tr>
<tr>
<td></td>
<td></td>
<td># Call Bar</td>
</tr>
<tr>
<td></td>
<td></td>
<td># read return value of bar()&lt;br&gt;# save the return value into a, i.e. a=bar();</td>
</tr>
<tr>
<td></td>
<td></td>
<td># deallocate the activation frame for Bar (caller’s part)</td>
</tr>
<tr>
<td></td>
<td></td>
<td># save a as the return value</td>
</tr>
<tr>
<td></td>
<td></td>
<td># restore Foo’s caller’s frame pointer&lt;br&gt;# restore the return address&lt;br&gt;# return</td>
</tr>
</tbody>
</table>