# Lab 6 Worksheet

1. Stack Pointer Warm-up

Answer the following questions with the following assembly code in mind. You can assume that the label "helper" is defined and accessible to your code.

```
foo:
addi sp, sp, ____
sd ra, ____(sp)
sd ____, 0(sp)

mv s1, a0
addi t0, s1, 4
call helper

ld ____, 0(sp)
ld ra, ____(sp)
addi sp, sp, ____
ret
```

1a. Fill in the blanks above. Remember RISC-V mandates a 16-byte alignment for the stack pointer.

1b. Place a box around the prologue and a circle around the epilogue in the assembly above.

1c. Fill in the blank: In 64-bit RISC-V, one saved register occupies _____ bytes on the stack.

1d. Which register is not saved on the stack? Why?

1e. Why does the value in *ra* need to be preserved on the stack in the prologue and then restored back to *ra* in the epilogue?

2. Calling Conventions Practice

2a. Fill In the Blanks

Given this C function, implement the corresponding assembly code for RV64 architecture. Your solution must use the course's specified calling conventions.

Note: ints are 32 bits, but signed integer overflow is undefined, so you can assume that never happens.

```
#include <stdio.h>

int multiply(int num1, int num2) {
    int product = num1 * num2;
    return product;
}

int dot_product(int *vec1, int *vec2, int length) {
    int sum = 0;
    for (int i = 0; i < length; i++) {
        sum += multiply(vec1[i], vec2[i]);
    }
    return sum;
}

int main() {
    int vec1[] = {1,2,3};
    int vec2[] = {4,5,6};
    int length = sizeof(vec1) /sizeof(vec1[0]);

    int dot_prod = dot_product(vec1, vec2, length);
    printf("The dot product is %d\n", dot_prod);
}
```

```
addi sp, sp, __          # allocate the stack frame
sd   ra,  __(sp)
sd   s1, __(sp)
sd   s2, __(sp)
sd   s3, __(sp)
sd   s4, __(sp)

mv   _______             # s2 = vec1
mv   _______             # s3 = vec2
mv   _______             # s4 is initialized to length

_____________            # initialize the sum variable

for_loop:
_____________            # loop guard – can be done in any appropriate location
# Prepare to pass in a[i] and b[i] to multiply
lw   _________
lw   _________

call  multiply
_____________            # update the sum variable with the result of the multiply function

# prepare for next loop iteration
_____________
_____________
_____________
_____________

epilogue:
# place return value in appropriate register
_____________

ld   ra,  __(sp)
ld   s1, __(sp)
ld   s2, __(sp)
ld   s3, __(sp)
ld   s4, __(sp)

# deallocate the stack frame and return from the function

addi sp, sp __
_____________
```

2b. Why was it better (more favorable) to use callee-saved registers vs caller-saved registers in certain areas of the dot_product function's assembly? What would happen if we used caller-saved registers instead?