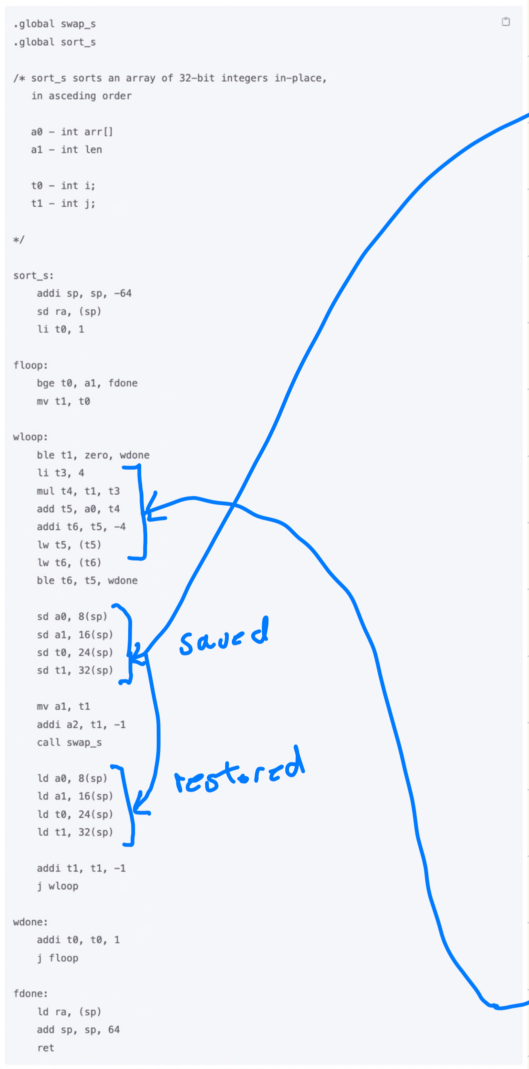
Question 1 - RISC-V Assembly

Consider the following RISC-V assembly code, then answer the following questions.



```
.global swap_s
.global sort_s

/* sort_s sorts an array of 32-bit integers in-place,
   in asceding order

   a0 - int arr[]
   a1 - int len

   t0 - int i;
   t1 - int j;

*/

sort_s:
    addi sp, sp, -64
    sd ra, (sp)
    li t0, 1

floop:
    bge t0, a1, fdone
    mv t1, t0

wloop:
    ble t1, zero, wdone
    li t3, 4
    mul t4, t1, t3
    add t5, a0, t4
    addi t6, t5, -4
    lw t5, (t5)
    lw t6, (t6)
    ble t6, t5, wdone

    sd a0, 8(sp)
    sd a1, 16(sp)
    sd t0, 24(sp)
    sd t1, 32(sp)

    mv a1, t1
    addi a2, t1, -1
    call swap_s

    ld a0, 8(sp)
    ld a1, 16(sp)
    ld t0, 24(sp)
    ld t1, 32(sp)

    addi t1, t1, -1
    j wloop

wdone:
    addi t0, t0, 1
    j floop

fdone:
    ld ra, (sp)
    add sp, sp, 64
    ret
```

saved

restored

Which caller-saved registers are preserved in this function, if any?

a0, a1, t0, t1

Which callee-saved registers are preserved in this function, if any?

Sp is a callee-saved register. However, we don't save it on the stack. Instead we subtract, then add back the same amount of bytes.

Are there caller-saved registers that are used but not preserved? If so, why is this okay?

Yes, t3, t4, t5, t6. These are used, but recomputed each time through the loop. No need to preserve.

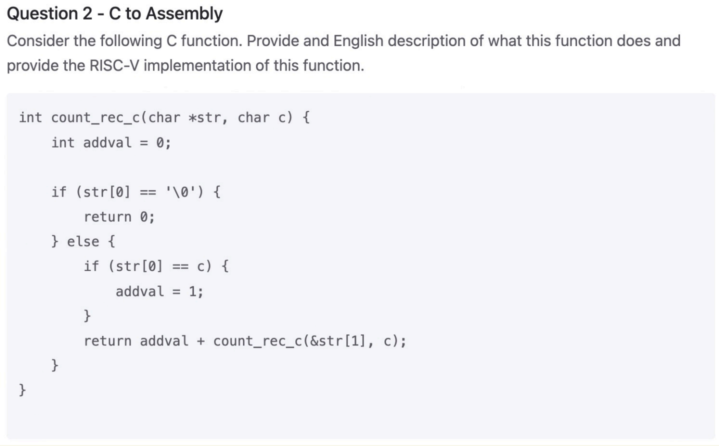
How many bytes of the stack are actually used by this function?

40 bytes out of 64 are used.

we save ra, a0, a1, t0, t1 on the stack. This is 5x8 bytes = 40 bytes.

Does this function use pointer-based array access or indexed-based array access?

It uses indexed-based array access because we compute the address of an array element using t0(i).



Question 2 - C to Assembly

Consider the following C function. Provide and English description of what this function does and provide the RISC-V implementation of this function.

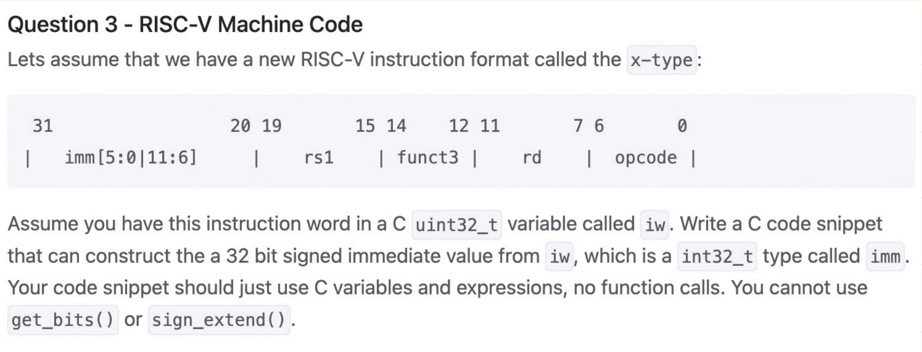
```
int count_rec_c(char *str, char c) {
    int addval = 0;

    if (str[0] == '\0') {
        return 0;
    } else {
        if (str[0] == c) {
            addval = 1;
        }
        return addval + count_rec_c(&str[1], c);
    }
}
```

This function counts the number of occurences of char c in the string str. It computes the count recursively.

```
.global count_rec_s

count_rec_s:
      addi sp, sp, -16
      sd   ra, (sp)
      li   t0, (a0)              # t0(addval) = 0
      lb   t1, (a0)
      beq  t1, zero, done        # t1(str[0]) == '\0' ?
      bne  t1, a1, recstep
      li t0, 1
recstep:
      sd t0, 8(sp)               # preserve t0
      addi a0, a0, 1             # &str[i]
      call count_rec_s
      ld t0, 8(sp)               # restore t0
      add t0, t0, a0             # t0(addval) += a0 (retval)
done:
      mv a0, t0                  # a0 = t0(addval)
      ld ra, (sp)
      addi sp, sp, 16
      ret
```



Question 3 - RISC-V Machine Code

Lets assume that we have a new RISC-V instruction format called the `x-type`:

```
 31                  20 19        15 14     12 11        7 6        0
|   imm[5:0|11:6]      |     rs1     | funct3 |     rd     |  opcode |
```

Assume you have this instruction word in a C `uint32_t` variable called `iw`. Write a C code snippet that can construct the a 32 bit signed immediate value from `iw`, which is a `int32_t` type called `imm`. Your code snippet should just use C variables and expressions, no function calls. You cannot use `get_bits()` or `sign_extend()`.

```
uint32_t iw;
uint32_t imm_5_0 = (iw >> 26) & 0b111111;
uint32_t imm_11_6 = (iw >> 20) & 0b111111;
int32_t imm = (imm_11_6 << 6) | imm_5_0;
imm = (imm << 20) >> 20; // sign extend
```