Name (Last, First):

This exam consists of 5 questions on 7 pages. Be sure you have the entire exam before starting. The point value of each question is indicated at its beginning; the entire exam is worth 100 points. Individual parts of a multi-part question are generally assigned approximately the same point value; exceptions are noted. For this exam you are allowed to bring a single page of notes (front and back). You may NOT share material with another student during the exam. Use of electronic devices is not allowed.

Be concise and clearly indicate your answers. Presentation and simplicity of your answers may affect your grade. Answer each question in the space following the question. If you find it necessary to continue an answer elsewhere, clearly indicate the location of its continuation and label its continuation with the question number and subpart if appropriate.

You should read through all the questions first, then pace yourself.

Problem	Possible	Score
1	20	
2	20	
3	20	
4	20	
5	20	
Total	100	

1 (20 points)

Short answer questions

(a) Consider the following C code snippet:

```
int *q = (int *) 0x3CDE2218;
q = q + 1;
```

What is the value of q after executing this snippet? Give the value in hexadecimal.

(b) Consider the following C code snippet:

```
uint32_t r = ((0x11223344) >> 8) & 0xFF;
```

What is the value of r in hexadecimal after executing this statement?

(c) Consider this 8-bit 2's complement binary value: `0b11101100`. What is this value in decimal? Show your work.

(d) Assume you have a Direct Mapped Cache with 64 words and block size of 2 words. How many total slots does this cache have? How many slot index bits are there in a 64 bit address?

2 (20 points)

RISC-V Assembly Part 1

Consider the following RISC-V assembly code snippets and initial register values. Show the values of each register used next to each instruction. For example:

```
li  t0, 1      # r0 <- 1
add t0, t0, t0 # r0 <- 1 + 1 = 2
```

You must show your work to get credit.

(a) Assume `a0 = 4, a1 = 3, a2 = 7`. What is the value of `a0` after executing this snippet?

```
add a0, a1, a2
mul a0, a0, a1
srai a0, a0, 1
```

(b) Assume `a0 = 3, a1 = 2`. What is the value of `a0` after executing this snippet?

```
    beq a0, a1, foo
    addi a0, a0, 22
    j boo
foo:
    addi a0, a0, 1
    j goo
boo:
    bne a0, a1, goo
    addi a0, a0, 33
goo:
    addi a0, a0, 1
```

(c) Assume `a0 = 10, a1 = 22`. What is the value of `a0` after executing this snippet?

```
addi sp, sp, -16
sw a1, (sp)
addi a1, a1, 1
sw a1, 4(sp)
mv a0, sp
lw a0, (a0)
addi sp, sp, 16
```

(d) Assume `a0 = 5, a1 = 1, a2 = 3`. What is the value of `a0` after executing this snippet? How many instructions are executed in this snippet? Labels do not count as instructions, but branches count if they are taken or not.

```
loop:
    beq a2, a1, loopend
    addi a0, a0, 2
    addi a2, a2, -1
    j loop
loopend:
    addi a0, a0, 3
```

3 (20 points)

RISC-V Assembly Part 2

Consider the following RISC-V assembly function. Answer the questions below.

```
.global func_s
.global swap
func_s:
    addi sp, sp, -48
    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, 48
    ret
```

(a) What does this function do? Give a possible C prototype for this function.

(b) What caller-saved registers are preserved if any?

(b) What callee-saved registers are preserved if any?

(c) How much of the allocated stack space is used by this function?

4 (20 points)

Recursive Sum Array

Recall in class we developed a RISC-V program called `sumarr_s.s` that sums the elements of an integer array. Here is a recursive version of the `sum_arr_c` function:

```
int sumarr_rec_c(int *arr, int start, int end) {
    int r;
    if (start == end) {
        r = arr[start];
    } else {
        r = arr[start] + sumarr_rec_c(arr, start + 1, end);
    }
    return r;
}
```

Write an equivalent RISC-V Assembly version of this program called `sumarr_rec_s`. Your implementation must be recursive, must follow the logic in the C version, and must follow the RISC-V calling conventions. In particular, you must implement indexed-based array access like the C code. You can write your program in two columns to make it fit on this page.

5 (20 points)

RISC-V Machine Code Encoding

For this problem we are going to build instruction encoders. That is, we are going to write functions that construct 32-bit instruction words from indivdual field values. Think of this as the opposite of the decoding we did in Project04. First, consider the R-type instruction format:

```
|31               25|24     20|19      15|14    12|11              7|6      0|
|       funct7       |   rs2   |    rs1   | funct3 |       rd        | opcode |
```

Here is an encoder function for the R-type:

```
uint32_t encode_r_type(uint32_t funct7, uint32_t rs2, uint32_t rs1,
                       uint32_t funct3, uint32_t rd, uint32_t opcode) {

    return (funct7 << 25) | (rs2 << 20) | (rs1 << 15) | (funct3 << 12) | (rd << 7) | opcode;

}
```

Now, consider the S-type instruction word format:

```
|31               25|24     20|19      15|14    12|11              7|6      0|
|     imm[11:5]     |   rs2   |    rs1   | funct3 |    imm[4:0]     | opcode |
```

Implement the encoder function for the S-type format. You can only use C variables and operators, you cannot assume you have helper functions like `get_bits()`. Hint: you will need to take apart the `imm` value to get the different parts to put in the final instruction word.

```
uint32_t encode_s_type(int32_t imm, uint32_t rs2, uint32_t rs1,
                       uint32_t funct3, uint32_t opcode) {
```

Continue your answers here if necessary.