# HONOR CODE

- I have not used any online resources during the exam.
- I have not obtained any help either from anyone in the class or outside when completing this exam.
- No sharing of notes/slides/textbook between students.
- NO SMARTPHONES.
- **CANVAS ANSWERS WILL BE LOCKED AFTER 1ST TRY.**

# Questions Sheet.

Read all of the following information before starting the exam:

- For each question fill out the appropriate choice or write text on Canvas page. Also type clearly on in the exam on the appropriate text.
- IF THE MULTIPLE CHOICE ANSWER IS WRONG WE WILL MARK THE ANSWER WRONG. IF THE MULTIPLE-CHOICE ANSWER IS CORRECT, WE WILL READ THE WRITTEN PORTION.
- Show all work, clearly and in order, if you want to get full credit.
- We reserve the right to take off points if we cannot see how you logically got to the answer (even if your final answer is correct).
- Circle or otherwise indicate your final answers.

- Please keep your written answers brief; be clear and to the point.
- I will take points off for rambling and for incorrect or irrelevant statements. This test has seven problems.

# A. Easy. Lets C. [6 Points]

## Q1-6

Grayscale color values can be represented as an ascii value between 0---255. Consider square images of $N \times N$ pixels.
We can organize these 2D images into a 1D array of $N^2$ elements.
Each element is an 8 bit number.

```
char *img = malloc(sizeof(char) * 4);
img[0] = 0xA;
img[1] = 0xB;
img[2] = 0xC;
img[3] = 0xD;
```

Fill out the following function `tile` . It returns a new, larger image array, which is the same image tiled rep times in both the x and y direction. You may or may not need all of the lines. For a better idea of what must be accomplished, consider the above example:

```
char *t_img = tile(img, 2, 2);
// The contents of tiled_image
would then look like:
[0xA, 0xB, 0xA, 0xB
0xC,  0xD, 0xC, 0xD
0xA,  0xB, 0xA, 0xB
0xC,  0xD, 0xC, 0xD];
```

Now fill-in-the blanks for the code shown below on canvas.

```
char *tile(char*b, int n, int rep) {
  int w =_______Q1_____________________;
  char *t_img = malloc(__________Q2______);
  for (int j = 0; j < w; j++) {
        for (int i = 0; i < w; i++) {
            int x = ___Q3____;
            int y = ___Q4____;
            int loc = _Q5____;
            t_img[loc] = b[x + y*n];
            }
    }
    _______Q6 (could be multiple lines)______;
    return t_img;
}
```

Q1: n * rep
Q2: $w * w$ or $w * w * sizeof(char)$
Q3: i%n
Q4: j%n
Q5: $j * w$ + i
Q6: Blank or nothing

# B. Easy. RISCV Magic. [7 Points]

Assume we have two arrays input and output.

```
int *input = malloc(8*sizeof(int));
int *result = malloc(8, sizeof(int));
for (int i = 0; i < 8; i++) {
    input[i] = i;
}
```

Study the following RISC-V code shown below and answer the questions.
You can assume a0:input a1:result a2:8

To understand and fill in the blanks in the `tile` function, let's analyze what the function needs to do and how the given example corresponds to the expected output.

The `tile` function takes three arguments:

1. `char *b`: The original image data.
2. `int n`: The size (width/height) of the original image (assumed square).
3. `int rep`: The number of times the image should be repeated both horizontally and vertically.

The function's goal is to create a new image `t_img` by tiling the original image `rep` times horizontally and vertically. Let's fill in the blanks based on this understanding:

## Analyzing the Tiling Process

Given:

- Original image data: `0xA, 0xB, 0xC, 0xD`
- Tiling 2x2:

Output (tiled) image data:

```
[0xA, 0xB, 0xA, 0xB
 0xC, 0xD, 0xC, 0xD
 0xA, 0xB, 0xA, 0xB
 0xC, 0xD, 0xC, 0xD]
```

## Determining the Fill-in-the-Blanks

1. **Q1: int w = _____;**
    - The width of the new tiled image. Since the original image is `n x n` and is to be tiled `rep` times in both dimensions, the width of the new image is `n * rep`.

    ```
    int w = n * rep;
    ```

2. *Q2: char t_img = malloc(_____);*
    - Allocate enough memory for the tiled image. The size of the memory should be the total number of pixels, which is `w * w`.

    ```
    char *t_img = malloc(w * w * sizeof(char));
    ```

3. **Q3: int x = _____;**
    - The horizontal position in the original image. To determine this, we use the modulus operation on the current position `i` in the new image by `n` to wrap around.

```
int x = i % n;
```

4. **Q4: int y = _____;**
    - The vertical position in the original image. To determine this, we use the modulus operation on the current position `j` in the new image by `n`.

```
int y = j % n;
```

5. **Q5: int loc = _____;**
    - The position in the new tiled image array. This can be calculated using `j * w + i`.

```
int loc = j * w + i;
```

6. **Q6 (could be multiple lines):**
    - Since the new image is created and filled, there might be additional operations, such as freeing allocated memory or other post-processing, but in this case, we only need to return the newly created `t_img`.

```
// No additional operations needed here, return the pointer to the new
image.
```

## Final Code

Here's the filled-out `tile` function:

```
char *tile(char *b, int n, int rep) {
    int w = n * rep;
    char *t_img = malloc(w * w * sizeof(char));
    for (int j = 0; j < w; j++) {
        for (int i = 0; i < w; i++) {
            int x = i % n;
            int y = j % n;
            int loc = j * w + i;
            t_img[loc] = b[x + y * n];
        }
    }
    return t_img;
}
```

Explanation:

- `int w = n * rep;`: Calculates the width of the new tiled image.
- `char *t_img = malloc(w * w * sizeof(char));`: Allocates memory for the new tiled image.
- `int x = i % n;` and `int y = j % n;`: Calculate the position in the original image to fetch the data from, based on the current position in the new tiled image.
- `int loc = j * w + i;`: Calculates the location in the new image where the pixel value should be placed.
- `t_img[loc] = b[x + y * n];`: Assigns the value from the original image to the new tiled image.
- **Returns** `t_img`, the pointer to the tiled image data.

```
1  main:
2    .....
3    addi a2, zero, 8
4  #  Q7 What registers need to be stored onto the stack?
5    jal ra, MAGIC   # a0 holds input, a1 holds result a2 holds 8.
6    # CHECK  finished calling BLACKBOX...
7  .... # Other code and function calls.
8  exit:
9    addi a0, x0, 10
10   add  a1, x0, x0
11   ecall            # Terminate ecall
12
13 BLACKBOX:
14 # Q8 What registers need to be stored onto the stack?
15   mv s0, zero
16   mv s1,a1
17   mv t0, zero
18 loop:
19   beq t0, a2, done
20   lw t1, 0(a0)
21   add s0, s0, t1
22   slli t2, t0, 2
23   add t2, t2, s1
24   sw s0, 0(t2)
25   addi t0, t0, 1
26   addi a0, a0, 4
27   jal x0, loop
28 done:
29   mv a0, s0
30 # Q9 TODO: epilogue.  What registers need to be restored?
31 jr ra
```

## 7. What is the minimum set of registers need to be stored onto the stack at this point: line 4. ? [1]

t0, t1, t2, a0,a1,a2

## 8. What is the minmum set of registers need to be stored onto the stack at this point: line 14. ? [1]

s0 s1,ra

## 9. What is the minmum set of registers need to be restored from the stack at this point: line 30 ? [1]

s0 s1,ra

### 10. Assume you have the prologue and epilogue correctly coded. You set a breakpoint at `line 6: CHECK". What does result contain when your program pauses at the breakpoint? [4]

0 1 3 6 10 15 21 28

# C. RISC-V Instructions Encoding [5 points]

Consider the standard RISC-V encoding below. Standard 32 bit instructions. Answer questions below

```
1  main:
2          mv s1, a0
3          addi t2, t2 4
4  Start:
5          beq s1, x0, End
6      lw a0, 0(s1)
7          addi a0,a0,4
8          add s1, t2, s1
9      lw s1, 0(s1)
10     jal x0, Start
11 End:
12         addi a0,a0,10
13   ecall
```

### 11. For the instruction line 7: `beq s1, x0, end` . What is the immediate field

24

### 12. What is actual opcode, rs1 and rs2 (not pseudo-names) ?

opcode: 0x63
rs1: 01001 (x9)
rs2: 00000 (x0)

### 13. What is funct7 and funct3 ?

f3 - 0 f7 N/A

### 14. What is the instruction corresponding to `0x0004A503` ?

```
lw x10 0(x9) or
```

lw a0, 0(s1) or
line 6

# D. RISC-V Custom Opcodes [6]

Prof. Shriraman is designing a new CPU with fewer operations. He decides to adapt and rethink the design of RISC-V instruction. He only needs to support 9 different operations: ADD, MUL, XOR, LD, SW, LUI, ADDI, MULI and BLT. He decides that each instruction should be 19 bits wide.

The fields in each instruction are listed below (no funct3 and funct7)

- R-type: rs2,rs1,rd,opcode
- I-type and Loads: imm,rs1,rd,opcode
- S-type: imm,rs2,rs1,opcode
- B-type: imm,rs2,rs1,opcode
- U-type: imm,rd,opcode
- UJ-type: imm,rd,opcode

### 15. What is the minimum bits would be required for the opcode field? [1]

4 bits

### 16. If the opcode was encoded 5 bits and we would like to support the usual R-type instructions, 2 source and 1 destination. What is the maximum number of registers we can use? [1]

4 bits. 16 registers.

### 17. Given the opcode is 5 bits wide. Each register field is 3 bits. What is the offset in terms of bytes that the branch instruction can jump forward. Note that instructions are 19 bits wide. [2]

19 - (3 + 3 + 5) = 8 bits for imm
$(2^7 - 1)$ - max positive offset OR if you assume 19 bit instructions are (3 byte aligned) then multiply by 3.

### 18. What is the max negative offset that an I instruction can use ? Opcode bits is same as Q15. Assume that register width is same as Q16. [2]

Opcode: 4 bits
Register: 4 bits. AddI requires 1 rd and 1rs.
Imm= 19 - $4 + 4 + 4$= 7 bits. $-2^6$

# E. Easy Floating Point [5 points]

Let's break down the given answers step-by-step based on the modified RISC-V instruction set with 19-bit instructions and the specified fields:

## Question 15: Opcode Bits

**Question:** What is the minimum number of bits required for the opcode field?

**Answer:** 4 bits

**Explanation:** To represent 9 different operations, we need at least (\lceil \log_2(9) \rceil = 4) bits, as (2^3 = 8) is not sufficient, and (2^4 = 16) provides enough room for 9 operations. Hence, the opcode field requires a minimum of 4 bits.

## Question 16: Maximum Number of Registers

**Question:** If the opcode is encoded with 5 bits and we would like to support the usual R-type instructions (2 source registers and 1 destination register), what is the maximum number of registers we can use?

**Answer:** 4 bits for registers, 16 registers total.

**Explanation:** Given that each instruction is 19 bits wide and we assume a 5-bit opcode, we can calculate the number of bits available for registers:

- Total instruction bits: 19
- Bits for opcode: 5

Remaining bits for register fields in an R-type instruction (which has 3 registers: `rs1`, `rs2`, and `rd`):

- (19 - 5 = 14) bits remaining

To distribute these among 3 registers:

- Each register field gets (\lfloor 14/3 \rfloor = 4) bits.

Thus, each register field can have 4 bits, allowing for (2^4 = 16) registers.

## Question 17: Offset for Branch Instruction

**Question:** Given a 5-bit opcode and 3 bits per register, what is the offset in terms of bytes that the branch instruction can jump forward?

**Answer:** 254 bytes

**Explanation:** For a B-type instruction:

- Opcode: 5 bits
- `rs1`: 3 bits
- `rs2`: 3 bits

Total bits for fixed fields:

- (5 + 3 + 3 = 11) bits

Remaining bits for immediate (imm):

- (19 - 11 = 8) bits

Since the immediate field can be a signed number (and thus can represent both positive and negative values), the maximum positive value for an 8-bit field is (2^7 - 1 = 127). However, since instructions are 19 bits wide, we assume the CPU uses 3-byte (24-bit) aligned addressing for jumps, allowing jumps to only even addresses.

Thus, the maximum offset is:

- (127 \times 3) bytes (since each increment in the imm field represents 3 bytes due to the alignment requirement).

## Question 18: Maximum Negative Offset for I-Type Instruction

**Question:** What is the maximum negative offset that an I-type instruction can use? Assume the same opcode bit width and register width as in the previous questions.

**Answer:** -26

**Explanation:** For an I-type instruction:

- Opcode: 4 bits (from Q15)
- `rd`: 4 bits (destination register, from Q16)
- `rs1`: 4 bits (source register, from Q16)
- Immediate (imm): remaining bits

Total bits used by opcode and registers:

- (4 + 4 + 4 = 12) bits

Remaining bits for immediate field:

- (19 - 12 = 7) bits

The immediate field in I-type instructions is a signed number, so it can represent both positive and negative values. The range for a signed 7-bit number is from -64 to 63.

The maximum negative offset, therefore, is:

- (-64 + 1 = -63) (since -64 is the most negative value representable, but typically the range is from -63 to 63 when accounting for signed representation in immediate fields).

.

The TAs get tired of having to convert floating-point values into 32 bits. As a result they propose the following smaller floating-point representation which is useful in a number of machine learning applications. It consists of a total of 10 bits as show below.
Exponent is biased similar to conventional floating point.

| Sign | Exponent | Mantissa |
|---|---|---|
| 1 bit | 5 bits. | 4 bits. |

- Numbers are rounded to the closest representable value. Any numbers that have 2 equidistant representations are rounded down towards zero.

## 19. What is the smallest non-zero positive value that can be represented? [1]

$2^-18$ = Denormalized form. exponent is $2^-14$ and mantissa is $2^-4$

## 20. How do you represent the number 4.5 ? [1]

0x112

0 10001 0010

exppnent: 17 - 15 = 2
$2^2$ * 1.125 = 4.5

## 21. How do you represent $-2^{-9}$ [1]

0x260

1 00110 0000

## 22. How many numbers can this 10 bit floating point represent in the range 1 ≤ f < 8). Hint: Write does the floating point expressions for 1 and 8 and the answer should be apparent. [1]

- 1 is 0x70 . 0 0111 0000 = $1 \times 2^0$
- 8 is 0xa0 . 0 1010 0000 = $1 \times 2^3$

There are 16 numbers, 0111 0000
There are 16 numbers, 1000 0000
There are 16 numbers, 1001 0000

Total - 48

Let's go through the questions one by one based on the given 10-bit floating point format, which has 1 sign bit, 5 exponent bits, and 4 mantissa bits.

**Question:** What is the smallest non-zero positive value that can be represented?

**Answer:** The smallest non-zero positive value is represented in the denormalized form, where the exponent is zero and the mantissa has a leading 1.

**Explanation:**

- **Sign bit (s):** 0 (indicating positive)
- **Exponent bits:** 00000 (which denotes the denormalized form)
- **Mantissa bits:** 0001 (smallest non-zero mantissa)

For denormalized numbers:

- The exponent is effectively ( 1 - \text{bias} ), where bias is ( 2^{(5-1)} - 1 = 15 ).
- In this case, the exponent is ( 1 - 15 = -14 ).
- The significand (or mantissa) for denormalized numbers is calculated as ( 0.\text{mantissa_bits} ), so for the given mantissa bits (0001), the significand is ( 0.0001_2 = 1 \times 2^{-4} = 2^{-4} ).

Thus, the smallest non-zero positive value is: [ 2^{-14} \times 2^{-4} = 2^{-18} ]

## Question 20: Representation of the Number 4.5

**Answer:** 0x112 or 0 10001 0010.
**Explanation:**

- **Sign bit (s):** 0 (positive number)
- **Exponent calculation:** The number 4.5 in binary is ( 100.1_2 ). This can be written as ( 1.001 \times 2^2 ).
- **Exponent bits:**
  - The exponent is 2. To represent this, we use the formula ( \text{exponent} = E + \text{bias} ).
  - Here, bias is 15, so ( E = 2 ), and the exponent value in bits is ( 2 + 15 = 17 ).
  - The exponent bits in binary are 10001.
- **Mantissa bits:** The significant part after removing the leading 1 (implicit) is 0010.

Thus, the representation of 4.5 in this floating point format is:

- Binary: 0 10001 0010
- Hexadecimal: 0x112

**Question:** How do you represent the number with the hex code 0x260?

**Answer:** 1 00110 0000.

**Explanation:**

- **Sign bit (s):** 1 (indicating a negative number)
- **Exponent bits:** 00110
- **Mantissa bits:** 0000

To convert 0x260 into binary:

- 0x260 in binary is 1 00110 0000.

**Question:** How many numbers can this 10-bit floating point represent in the range ( 1 \le f < 8 )?

**Answer:** There are 48 numbers.
**Explanation:** To determine the representable numbers within a range, we need to look at the exponents and the corresponding mantissas.

**Representation of 1**

For the smallest normalized value ( f = 1 ):

- **Binary Representation:** 1.0000 (implicit leading 1)
- **Exponent:** ( 0 ) in real value, represented as ( 0 + 15 = 15 ).
- **Exponent bits:** 01111

So, the binary representation is 0 01111 0000 (in hex: 0x70).

**Representation of 8**

For the value ( f = 8 ):

- **Binary Representation:** 1.0000 (implicit leading 1)
- **Exponent:** ( 3 ) in real value, represented as ( 3 + 15 = 18 ).
- **Exponent bits:** 10010

So, the binary representation is 0 10010 0000 (in hex: 0xa0).

**Counting the Representable Numbers**

For each exponent value, the mantissa can vary, providing different numbers. The range covers exponent values from:

- 15 (0x70, for ( f = 1 )) to
- 17 (0x90) but not including 18 (0xa0, for ( f = 8 )).

For each exponent value, the mantissa can vary from 0000 to 1111 (16 possible values). There are 3 exponent values (15, 16, and 17), so the total number of representable numbers is: [ 3 x 16 = 48]
Thus, 48 numbers can be represented in the range ( 1 \leq f < 8 ).

### 23. Sort the following numbers 0x300, 0x100, 0x104, 0x328, 0x12c smallest to largest [1]

Shown are the hex representations.

0x300 (-2), 0x100 (2), 0x104 (2.5), 0x328 (-12), 0x12c (14)

# F. Unsigned/Signed Numbers [5]

Suppose that we define a new number format, **NOT**.
`Negative numbers are represented by the binary NOT ∼ of the binary representations of their corresponding positive numbers.` Like 2's complement most significant bit of NOT denotes the number's sign (0 for positive, 1 for negative). Answer the following questions.

NOT (~)

| Input | Output |
|---|---|
| 1 | 0 |
| 0 | 1 |

### 24. Represent -133 as a 8-bit NOT number. [1]

N/A

### 25. Represent -124 as 8-bit NOT number. [1]

124 - 0111 1100
-124 - 1000 0011

### 26. What is the range of numbers represented by 8 bit NOT8. [1]

-127 to +127

### 27. What is the representation that requires fewest number of bits needed to cover the given range 0 to 10. [1]

Unsigned. 4 bits

### 28. What is the representation that requires fewest number of bits needed to cover the given range -1 to 4. [1]

Two's complement 3.

# G. Bitgames. Write down single line expressions that calculate the following. [4]

### 29. NegativeFloat(x) - Return bit-level equivalent of expression -f for floating point argument f. Assume f is N bits

N | 0x1 << $(31)$

### 30. is_float_power_of_2(float x, int e, int m).

Normalized number : 1.m * $(2^e - 1 - bias)$
((x & (2^e-1) << m) != (2^e -1)) && (x & (2^m -1)) == 0

# H. Assembler Linker Compiler [6]

```
.data
str: string "The sum of 1..100 is %d \n"

.text
main:
  add sp,sp -4
  sw ra, 0 (sp)
  mv a1, zero
  li x9, 100
  j check

loop:
  mul s2,x9,x9
  add a1,a1,s2
  add x9,x9,-1

chk:
  bnez x9, loop
  la a0, str
  jal printf
  mv a0,zero
  lw ra, 0(sp)
  addi sp,sp,4
  ret
```

### 31. How many pseudo instructions and registers ? [2]

8 instructions, 7 registers (sp,ra,a2,zero,s2,a1,a0)

mv,li,j,bnez,la,jal,ret

Assume data segment starts at 0x80. This was mentioned in class

### 32. What is the symbol and relocation table ? [2]

| | |
|---|---|
| main | 0x0 |
| loop | 0x14 |
| chk | 0x20 |
| str | 0x80 |

Relocation

| Inst | Addr | Dep |
|---|---|---|
| la | 0x24 | str |
| jal printf | 0x28 | printf |

### 33. Replace the labels of PC-relative targets with their immediate values. What is the offset value of bnez at address 0x20? Write your answer in decimal. [2]

-12. Note that bnez is located at 8 instructions from start; code always starts at 0x0. Each instruction is 4 bytes. Thus bnez is located at 32 bytes from start (or 0x20 in hex). its target is 3 instructions before or -12.

# G. RISCV

### 34. What is the value of s2 at the end of the program ? Write in hex (e.g., 0xFFFF) [3]

Initial values: s0 = 0xC, s1 = 0xA, a0 = 3, a1 = 0, a2 = 2

```
.text
  la a3, start
start:
        beq a1,a0,End
  lw  a4,20(a3)
  slli a5,a2,12
  add a4,a4,a5
  sw a4, 20(a3)
  add s2,s1,s0
  addi a1,a1,1
  j start

End:
    addi a0,zero,10
   ecall
```

Answer: 0xE

Start is address of beq a1,a0,end We are doing start+20 and moving it into a4. Question is what is 20 bytes from start

Assuming start is at 0 (you can really assume any address). Each instruction is 4 bytes 0 : beq 4: lw 8: slli 12: add 16: sw 20: add s2,s1,s0

So you are loading the word at address 20 bytes from start. Which is the encoded add s2,s1,s0 (whatever the 32 bit number is)

a4: Instruction encoding of add s2,s1,s0 (0x00848933)

a5: modifies lines up 2 in the funct 3 position. line 7 repeatedly modifies funct3 and increments it by 2. First

iteration its 0x**2** : **slt** operation . Next iteration its 0x**4**: xor operation. Finally 0x4: or operation.

Loop only runs 3 times. So final iteration the value in x2 is 0xC | 0xA = 0xE