CprE381 HW9

1.
    a. Included .asm file
    b. Included .asm file
    c. https://github.com/madhurimarawat/Data-structure-using-C/blob/main/Sorting/Merge-Sort-code.c?raw=true
    The above file demonstrates both spatial and temporal data locality during execution.
    Lines 76 to 77 (in the merge function) show **spatial locality** because they access array elements sequentially (a[k], a[k+1]...). When the CPU fetches a[k] into the cache, it likely also brings a[k+1] and other elements. This clearly shows the use of spatial locality since the CPU brings instructions close to the first fetched instruction into the cache.
    Lines 21 to 27 (in the mergesort function) show **temporal locality** since the variables a, b, low, and pivot are used repeatedly across the three function calls. Reusing the same variables back to back clearly shows temporal locality since they are being accessed quickly, which means they likely remain in the cache block between the accesses.

2. Row Major Ordering:
    Run 1 – 0.965488 seconds
    Run 2 – 0.965639 seconds
    Run 3 – 0.965135 seconds
    Run 4 – 0.960522 seconds
    Run 5 – 0.969345 seconds
    Average – 0.9652258 seconds

    Column Major Ordering:
    Run 1 – 0.994050 seconds
    Run 2 – 0.985649 seconds
    Run 3 – 1.002257 seconds
    Run 4 – 0.985245 seconds
    Run 5 – 0.989142 seconds
    Average – 0.9953312 seconds

The above indicates that row-major ordering is faster on my computer (Apple M2 chip). It is faster than column-major ordering because arrays are stored in row-major ordering in C, meaning that the elements of each row are laid out next to each other in memory. When my program accesses elements using row-major ordering, it benefits from C's contiguous memory access, which results in better cache locality. On the other hand, accessing elements in column-major order causes jumps across memory locations, which leads to poor cache usage and frequent cache misses.

3.

a. Direct Mapped Cache (16 one-word blocks) – No Offset

Reference	Binary Address	Tag (28 bits)	Index (4 bits)
0x03 (Miss)	0000 0000 0000 0000 0000 0000 0000 0011	0x0000_000	0b0011
0xb4 (Miss)	0000 0000 0000 0000 0000 0000 1011 0100	0x0000_00b	0b0100
0x2b (Miss)	0000 0000 0000 0000 0000 0000 0010 1011	0x0000_002	0b1011
0x02 (Miss)	0000 0000 0000 0000 0000 0000 0000 0010	0x0000_000	0b0010
0xbf (Miss)	0000 0000 0000 0000 0000 0000 1011 1111	0x0000_00b	0b1111
0x58 (Miss)	0000 0000 0000 0000 0000 0000 0101 1000	0x0000_005	0b1000
0xbe (Miss)	0000 0000 0000 0000 0000 0000 1011 1110	0x0000_00b	0b1110
0x0e (Miss)	0000 0000 0000 0000 0000 0000 0000 1110	0x0000_000	0b1110
0xb5 (Miss)	0000 0000 0000 0000 0000 0000 1011 0101	0x0000_00b	0b0101
0x2c (Miss)	0000 0000 0000 0000 0000 0000 0010 1100	0x0000_002	0b1100
0xba (Miss)	0000 0000 0000 0000 0000 0000 1011 1010	0x0000_00b	0b1010
0xfd (Miss)	0000 0000 0000 0000 0000 0000 1111 1101	0x0000_00f	0b1101

b. Direct Mapped Cache (8 two-word blocks) – 1 Bit Offset

Reference	Binary Address	Tag (29 bits)	Index (3 bits)
0x03 (Miss)	0000 0000 0000 0000 0000 0000 0000 0011	0x0000_000	0b001, Offset: 1
0xb4 (Miss)	0000 0000 0000 0000 0000 0000 1011 0100	0x0000_00b	0b010, Offset: 0
0x2b (Miss)	0000 0000 0000 0000 0000 0000 0010 1011	0x0000_002	0b101, Offset: 1
0x02 (Hit)	0000 0000 0000 0000 0000 0000 0000 0010	0x0000_000	0b001, Offset: 0
0xbf (Miss)	0000 0000 0000 0000 0000 0000 1011 1111	0x0000_00b	0b111, Offset: 1
0x58 (Miss)	0000 0000 0000 0000 0000 0000 0101 1000	0x0000_005	0b100, Offset: 0
0xbe (Hit)	0000 0000 0000 0000 0000 0000 1011 1110	0x0000_00b	0b111, Offset: 0
0x0e (Miss)	0000 0000 0000 0000 0000 0000 0000 1110	0x0000_000	0b111, Offset: 0
0xb5 (Hit)	0000 0000 0000 0000 0000 0000 1011 0101	0x0000_00b	0b010, Offset: 1
0x2c (Miss)	0000 0000 0000 0000 0000 0000 0010 1100	0x0000_002	0b110, Offset: 0
0xba (Miss)	0000 0000 0000 0000 0000 0000 1011 1010	0x0000_00b	0b101, Offset: 0
0xfd (Miss)	0000 0000 0000 0000 0000 0000 1111 1101	0x0000_00f	0b110, Offset: 1

c. C1 has a miss rate of 12/12 (100%), C2 has a miss rate of 10/12 (83.3%), and C3 has a miss rate of 11/12 (91.6%). Comparing all of them, C2 has the best cache design regarding miss rate.

C1:
Given the scenario above, 12 * 27 = **324** is the worst-case cycle count.

C2:
Given the scenario above, (10 * 28) + (2 * 3) = **286** is the worst-case cycle count.

C3:
Given the scenario above, (11 * 30) + (1 * 5) = **335** is the worst-case cycle count.

Based on the above calculations, C2 has the best cache design since it has the lowest worst-case cycle time.

d. Yes, we can use this to index a direct-mapped cache. However, since it is only using Block address[31:27] XOR Block address[26:22], it will produce a 5-bit index value (only valid for a cache with 32 blocks, not 1024).
This means that if we wanted to implement this logic instead of mod, we would need to yield 10 bits instead of 5, which means using more bits from the block addressing.