1. Assume we have an array in memory that contains **`int* arr = {1,2,3,4,5,6,0}`**. Let the values of arr be a multiple of 4 and stored in register s0. What do the snippets of RISC-V code do? Assume that all the instructions are run one after the other in the same context.

a)
```
lw t0, 12(s0) -->  Sets t0 equal to arr[3]
```
b)
```
slli t1, t0, 2
add t2, s0, t1
lw t3, 0(t2) -->  Increments arr[t0] by 1
addi t3, t3, 1
sw t3, 0(t2)
```
c)
```
lw t0, 0(s0)
xori t0, t0, 0xFFF -->  Sets t0 to -1 * arr[0]
addi t0, t0, 1
```
d)

[inst]	[destination register] [argument register 1] [argument register 2]
add	Adds the two argument registers and stores in destination register
xor	Exclusive or's the two argument registers and stores in destination register
mul	Multiplies the two argument registers and stores in destination register
sll	Logical left shifts AR1 by AR2 and stores in DR
srl	Logical right shifts AR1 by AR2 and stores in DR
sra	Arithmetic right shifts AR1 by AR2 and stores in DR
slt/u	If AR1 < AR2, stores 1 in DR, otherwise stores 0, u does unsigned comparison

[inst]	[register] [offset]([register with base address])
sw	Stores the contents of the register to the address+offset in memory
lw	Takes the contents of address+offset in memory and stores in the register

[inst]	[argument register 1] [argument register 2] [label]
beq	If AR1 == AR2, moves to label
bne	If AR1 != AR2, moves to label

[inst]	[destination register] [label]
jal	Stores the current instruction's address into DR and moves to label

You may also see that there is an "i" at the end of certain instructions, such as addi, slli, etc. This means that AR2 becomes an "immediate" or an integer instead of using a register.

While only using the instructions (and their "i" forms) given above, how can we branch on the following conditions:

`s0 < s1`
```
slt t0, s0, s1
bne t0, zero, label
```

`s0 ≥ s1`
```
slt t0, s0, s1
beq t0, zero, label
```

`s0 > 1`
```
sltiu t0, s0, 2
beq t0, zero, label
```

2. Write a function quadruple in RISC-V that, when given an integer x, returns 4x.

```
quadruple: add a0, a0, a0
           add a0, a0, a0
           jr ra
```

3. Write a function power in RISC-V that takes in two numbers x and n, and returns x^n. You may assume that n≥0 and that multiplication will always result in a 32-bit number.

```
power: li t0, 0
       addi t1, a0, 0
loop:  bge t0, a1, end
       mul a0, a0, t1
       addi t0, t0, 1
       jal x0, loop
end: jr ra
```

4. Translate between the C and RISC-V verbatim

C	RISC-V

`// computes s1 = 2^30`
`s1 = 1;`
`for(s0=0;s0<30;s++) {`
`  s1 *= 2;`
`}`
	`  addi s0, x0, 0`
	`  addi s1, x0, 1`
	`  addi t0, x0, 30`
	`loop:`
	`  beq s0, t0, exit`
	`  add s1, s1, s1`
	`  addi s0, s0, 1`
	`  jal x0, loop`
	`exit:`

`// s0 -> int * p = intArr;`
`// s1 -> a;`
`*p = 0;`
`int a = 2;`
`p[1] = p[a] = a;`
	`sw x0, 0(s0)`
	`addi s1, x0, 2`
	`sw s1, 4(s0)`
	`slli t0, s1, 2`
	`add t0, t0, s0`
	`sw s1, 0(t0)`

`// s0 -> a, s1 -> b`
`int a = 5, b = 10;`
`if(a + a == b) {`
`  a = 0;`
`} else {`
`  b = a - 1;`
`}`
	`  addi s0, x0, 5`
	`  addi s1, x0, 10`
	`  add t0, s0, s0`
	`  bne t0, s1, else`
	`  xor s0, x0, x0`
	`  jal x0, exit`
	`else:`
	`  addi s1, s0, -1`
	`exit:`

`// s0 -> a, s1 -> b`
`// s2 -> c, s3 -> z`
`int a = 4, b = 5, c = 6, z;`
`z = a + b + c + 10;`
	`addi s0, x0, 4`
	`addi s1, x0, 5`
	`addi s2, x0, 6`
	`add s3, s0, s1`
	`add s3, s3, s2`
	`addi s3, s3, 10`

`// s0 -> n, s1 -> sum`
`// assume n > 0 to start`
`for(int sum = 0; n > 0; n--) {`
`  sum += n;`
`}`
	`  addi s1, s1, 0`
	`loop:`
	`  beq s0, x0, exit`
	`  add s1, s1, s0`
	`  add s0, s0, -1`
	`  jal x0, loop`
	`exit:`

5. Computing a Fibonacci Number. The Fibonacci number F_n is recursively defined as

```
F(n) = F(n - 1) + F(n - 2);
```

where `F(1) = 1 and F(2) = 1. So, F(3) = F(2) + F(1) = 1 + 1 = 2`, and so on. Write the RISC-V assembly for the fib(n) function, which computes the Fibonacci number F(n):

```
int fib(int n)
{
    int a = 0;
    int b = 1;
    int c = a + b;
    while (n > 1) {
        c = a + b;
        a = b;
        b = c;
        n--;
    }
return c;
}
```

```
// s0 -> n, s1 -> c
// t0 -> a, t1 -> b

addi t0, x0, 0
addi t1, x0, 1
add  s1, t0, t1
addi t2, x0, 1
Loop: beq  s0, t2, RetF
      add  s1, t0, t1
      addi t1, t0, 0
      addi t0, s1, 0
      addi s0, s0, -1
      jal  x0, Loop
RetF: add  a0, x0, s1
```

6. We covered the following addressing modes in class:

- Absolute
- Register indirect
- Based (base + displacement)
- Scale indexed (base + index _ constant)
- Memory indirect
- Auto increment/decrement (by 1 byte)

Consider the following high-level programs:

- ```
  uint8_t a[150]; // a is allocated in memory
  for (i = 0; i < 150; i++) {
      a[i] = 5;
  }
  ```
- ```
  int a[150]; // a is allocated in memory
  for (i = 0; i < 150; i++) {
      a[i] = 5;
  }
  ```
- ```
  int *p; // *p is allocated in memory
  *p = 150;
  ```
  ```

- ```
  int **p; // *p and **p are allocated in memory
  **p = 150;
  ```

Assume that in the first two programs, a register contains the address of the start of the array, and in the last two programs, a register contains the value of p. For each of the above three programs, which of the addressing modes, do you think, would lead to the minimum number of instructions? (Note that no addressing mode fits perfectly. You might require other instructions for address computation.)

(a) Auto increment
(b) Scale indexed
(c) Register indirect
(d) Memory indirect
  ```