KERNEL CODE WITH SOLID PRINCIPLE:

- ❖ **Single Responsibility Principle (SRP)**

Principle: A class should have only one reason to change. This means that a class should have only one job or responsibility

To achieve this principle: The code is organized into functions, each with a single responsibility. For example:

device_open handles the opening of the device.

device_release handles the closing of the device.

device_ioctl handles ioctl commands and performing arithmetic operations based on the command received.

This modular approach ensures each function has a clear, single responsibility, making the code easier to maintain and understand.

```
static int device_open(struct inode *inode, struct file *file)
static int device_release(struct inode *inode, struct file *file)
static long device_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
```

- ❖ **Open/Closed Principle (OCP)**

Principle:Software entities should be open for extension but closed for modification.

To achieve this principle: New arithmetic operations can be added by defining new ioctl commands and extending the switch statement in device_ioctl. This allows the module to be extended with new functionality without modifying existing code

```
 switch (cmd) {

 case IOCTL_ADD: values[2] = values[0] + values[1];

 printk(KERN_INFO "Add %d + %d = %d\n", values[0], values[1], values[2]);

Break;
```

The IOCTL commands (IOCTL_ADD, IOCTL_SUB, IOCTL_MUL, IOCTL_DIV, IOCTL_MOD) allow for adding new arithmetic operations without modifying the existing code.

- ❖ **Liskov Substitution Principle (LSP)**

Principle: Objects of a superclass should be replaceable with objects of a subclass without affecting the functionality of the program.

To achieve this principle:Each command (IOCTL_ADD, IOCTL_SUB, etc.) can be considered a subclass of the generic IOCTL_BASIC_ARITH or IOCTL_ADVANCED_ARITH. The device_ioctl function handles each command appropriately, ensuring that any valid command can be processed without changing the device_ioctl logic.

```
// Basic Arithmetic IOCTLs #define IOCTL_ADD _IOWR(IOCTL_BASIC_ARITH, 1, int[3])
#define IOCTL_SUB _IOWR(IOCTL_BASIC_ARITH, 2, int[3])
static long device_ioctl(struct file *file, unsigned int cmd, unsigned long arg) {
    // Code for performing arithmetic operations based on IOCTL commands
}
```

The device_ioctl function handles different IOCTL commands interchangeably, adhering to the LSP. It performs the appropriate operation based on the command received.

- ❖ **Interface Segregation Principle (ISP)**

Principle: Software entities should not be forced to depend on interfaces they do not use.

To achieve this principle:The file operations structure (fops) provides specific hooks for open, release, and ioctl operations. Each operation (open, release, ioctl) is handled by its specific function. This ensures that the interface is minimal and that each function only depends on what it actually needs.

```
static struct file_operations fops = {
    .owner = THIS_MODULE,
    .open = device_open,
    .release = device_release,
    .unlocked_ioctl = device_ioctl,
};
```

The kernel code segregates operations into distinct functions (device_open, device_release, device_ioctl) based on their responsibilities.

- **Dependency Inversion Principle (DIP)**

Principle: High-level modules should not depend on low-level modules. Both should depend on abstractions (interfaces or abstract classes). Abstractions should not depend on details. Details should depend on abstractions.

```
// Basic Arithmetic IOCTLs
 #define IOCTL_ADD _IOWR(IOCTL_BASIC_ARITH, 1, int[3])
 #define IOCTL_SUB _IOWR(IOCTL_BASIC_ARITH, 2, int[3]) ‘
// Advanced Arithmetic IOCTLs
 #define IOCTL_MUL _IOWR(IOCTL_ADVANCED_ARITH, 1, int[3])
 #define IOCTL_DIV _IOWR(IOCTL_ADVANCED_ARITH, 2, int[3])
#define IOCTL_MOD _IOWR(IOCTL_ADVANCED_ARITH, 3, int[3])
```

USER SPACE CODE WITH SOLID PRINCIPLE

- **Single Responsibility Principle (SRP)**

Principle: A class should have one, and only one, reason to change.

To achieve this principle: In the user-space program,we separate the code for input handling, IOCTL command handling, and output display into different functions.

```
void handle_input(char *operation, int *operand1, int *operand2)

void perform_operation(int fd, unsigned long cmd, int operand1, int operand2, int *result)

void display_result(int result)
```

Each function has a single responsibility:

handle_input: Takes user input for the operation and operands.
perform_operation: Communicates with the device driver and executes the operation.
display_result: Displays the result to the user.

- **Open/Closed Principle (OCP)**

Principle: Software entities should be open for extension, but closed for modification.

To achieve this principle:Extend the IOCTL commands without modifying existing code by adding new IOCTL for each operations like add,subtract,multiply,divide.

```
void add_operation(int fd, int operand1, int operand2, int *result)
void subtract_operation(int fd, int operand1, int operand2, int *result)}
```

// Other operation functions...
New operations (e.g., multiplication, division) can be added easily by creating new functions without modifying existing ones.

- **Liskov Substitution Principle (LSP)**

Principle: Objects of a superclass should be replaceable with objects of a subclass without affecting the functionality of the program.

To achieve this principle: operation_func function pointer is used. Any function that matches the operation_func signature can be passed to execute_operation. This flexibility ensures that the program can substitute one operation function for another without Affecting the execute_operation function's implementation.

```
typedef void (*operation_func)(int fd, int operand1, int operand2, int *result);

void execute_operation(operation_func op, int fd, int operand1, int operand2, int *result) {
   op(fd, operand1, operand2, result);
}
```
The operation_func function pointer allows for interchangeable operation functions with the same signature.

- **Interface Segregation Principle (ISP)**

Principle: A client should not be forced to depend on interfaces it does not use.

To achieve this principle:The program avoids forcing functions to depend on unnecessary parameters or functionalities. Each operation function (add_operation, subtract_operation, etc.) comply to a simple and specific interface (operation_func), which takes only the necessary arguments. This keeps the interfaces small and focused.

```
// Each operation function has a specific responsibility and interface
void add_operation(int fd, int operand1, int operand2, int *result) { ... }
void subtract_operation(int fd, int operand1, int operand2, int *result) { ... }
// Other operation functions...
```
Operations are segregated into distinct functions based on their responsibilities, preventing clients from depending on unnecessary operations.

- **Dependency Inversion Principle (DIP)**

Principle: High-level modules should not depend on low-level modules. Both should depend on abstractions.

To achieve this principle:The high-level logic in main does not depend on the low-level details of how each operation is performed. Instead, it uses function pointers (operation_func) to call the appropriate operation function.

```
void execute_operation(operation_func op, int fd, int operand1, int operand2, int *result) {
    op(fd, operand1, operand2, result);
```

The execute_operation function depends on an abstraction (operation_func function pointer) rather than concrete implementations, allowing for easy extension and modification without directly relying on specific implementations.