CSI 2372 - Final Exam - Cheat Sheet

Question 1-5 (4 points each)

The first five questions will be True/False, Multiple Choice, and/or Multiple Selection

Pointer: Variable that stores memory address as its value. Declaration: `int *ptr` **Address-of Operator** (`&`): Obtains the memory address of a variable. Declaration: `&variable` **Dereference Operator** (`*`): Accesses the value stored at a pointer's address. Declaration: `*ptr`

Ternary Operator

I don't think he explicitly taught this, but many people might not have been aware of the ternary operator.

The ? operator works as a primitive if-else statement under a certain formatting. It is as follows:

```
condition ? expression_if_true : expression_if_false;
```

```
#include <iostream>

int main() {
    int num1 = 10;
    int num2 = 20;

    // Using the ternary operator to find the smaller number
    int smallerNumber = (num1 < num2) ? num1 : num2;

    std::cout << "The smaller number is: " << smallerNumber << std::endl;

    return 0;
}
```

Increment/Decrement

The ++ and - - operators can be useful for incrementing and decrementing integers and array memory locations.

```
int x = 0;

x++; // x is now 1
++x; // x is now 2
x--; // x is now 1
--x; // x is now 0
```

Both of these formatting types work for their purpose, however when used in a function, it is very important to know which one may be needed.

For example, when this function is done:

```
int x = 1;
int y = 2;

std::cout << x++ + ++y;
```

The output of this would be 4. The code takes x and uses it in the addition **before** incrementing it. x is now 2, but was used as 1 in the operation. y had the ++ preceding it, so it was increased to 3 before being used in the operation, so the operation was 1 + 3. The same works for the - - operator.

Macros

Processed BEFORE the actual compilation of the code, enabling code reusability, conditional compilation, and platform-specific adaptations. They do not need to be in capital letters, but it is standard formatting to do so.

#define

Macros can work either as object-like or function-like. All macros, either object or function use the #define operator.

Object-Like Macros

They work like any standard variable.

```
#define PI 3.14
#define GREETING "Hello, World!"

double radius = 5.0;
double circumference = 2 * PI * radius;  // Results in circumference being approximately 31.4159

std::cout << GREETING; //prints "Hello, World!"
```

Function-Like Macros

Preprocessor macros can also define functions. The formatting is as shown below.

```
#define MACRO_NAME(parameter1, parameter2, ...) (operation)
```

Used in an example:

```
#define SUM(x, y) (x + y)

// You can then use this function as you would normally

int main(){
    int x = 2;
    int y = 24;
    std::cout << SUM(x,y) << std::endl;

}
```

#undef #ifdef #elif #else #endif

Checks if a macro is defined. Works like an if condition on a boolean case on if it is defined or not.

```
#include <iostream>

#define TEST 1

int main() {
  #ifdef TEST
    std::cout << "TEST defined\n";
  #else
    std::cout << "TEST undefined\n";
  #endif

  #undef TEST

  #ifdef TEST
    std::cout << "TEST defined\n";
  #else
    std::cout << "TEST undefined\n";
  #endif
}
```

Below is a simple test of if, else if,

Question 6 (6 points)

Assesses your memory management skills by requiring a C++ code realization based on a provided description.

Topic List

Dynamic Memory Allocation

```
#include <iostream>

int* pointVar; // declare int pointer (int *pointVar; works too)
pointvar = new int; // for initializing to a new variable.

int *differentVar = new int; // alternate way of declaring new.

int x = 420; //declare and initialize int
int *pointx = &x; // use & to declare pointer to a preexisting variable
```

```
int main() {
    // Initialize a matrix, 4 rows, 3 columns. Each element of the array of size 4 contains an array of size 3.
    int matrix[4][3] = { {11, 12, 13}, {14, 15, 16}, {17, 18, 19}, {110, 111, 112} };
 // Create a pointer to the second row of the matrix, index 1
 // This a pointer to an array of size 3
 // The brackets are needed () for array pointer declaration
```

```
 // pointer name is arraypoint
 int(*arraypoint)[3] = &matrix[1]; // Points to the second row (index 1)
 // Access the values at the addresses of individual elements in the second row
 int value_1 = (*arraypoint)[0]; // int address_1 = &(*ptr)[0]; for address of this pointer
 int value_2 = (*arraypoint)[1]; // int address_2 = &(*ptr)[1]; for address of this pointer
 int value_3 = (*arraypoint)[2]; // int address_3 = &(*ptr)[2]; for address of this pointer
 // Print the values
 std::cout << "Value at Matrix[1][0]: " << value_1 << std::endl;
 std::cout << "Value at Matrix[1][1]: " << value_2 << std::endl;
 std::cout << "Value at Matrix[1][2]: " << value_3 << std::endl;
}
```

Smart Pointers

Unique Pointer

Type of smart pointer in C++ that represents exclusive ownership of a dynamically allocated object. Unlike shared pointers, only one unique pointer can own the object, and when the unique pointer goes out of scope, the associated memory is automatically deallocated.

```
#include <memory>

int main() {
    std::unique_ptr<int> uniqueInt = std::make_unique<int>(42);
    // uniqueInt cannot be copied, only moved.

    // Attempting to assign it to another unique pointer will result in a compilation error
    // std::unique_ptr<int> uniquePtr2 = uniquePtr1;  // Error: copy constructor is deleted

    // Move ownership to another unique pointer
    std::unique_ptr<int> uniqueInt2 = std::move(uniqueInt1);

    // Attempting to access the original unique pointer after the move
    // This will result in undefined behavior, as the ownership has been transferred
    // std::cout << *uniquePtr1 << std::endl;  // Undefined behavior

    // Accessing the new unique pointer
    std::cout << *uniquePtr2 << std::endl;  // Valid

    return 0;  // uniqueInt is automatically deallocated when it goes out of scope
}
```

Shared Pointer

Type of smart pointer in C++ that allows multiple shared pointers to share ownership of the same dynamically allocated object. Memory is automatically deallocated when the last shared pointer that owns it goes out of scope, preventing premature deallocation.

```
#include <memory>

int main() {
    std::shared_ptr<int> sharedInt = std::make_shared<int>(42); //1 Instance
    // sharedInt can be copied, and all copies share ownership
```

```
    {
        std::shared_ptr<int> sharedInt2 = sharedInt; //2 Instances
    } //1 Instance (sharedInt)
    return 0;  // sharedInt is automatically deallocated when all of them go out of scope
}//0 Instances
```

Weak Pointer

Type of smart pointer in C++ that allows access to the be shared from a shared_ptr, but doesn't count towards the number of instances that need to go out of scope for the pointer to deallocate. In other words, it is a shared pointer that does not have any ownership of the memory.

```
#include <memory>

int main() {
    std::shared_ptr<int> sharedInt = std::make_shared<int>(42); //1 Instance
    // sharedInt can be copied, and all copies share ownership

    {
        std::weak_ptr<int> weakInt = sharedInt; //1 Instances (weak pointer doesn't add to instances)
    } //1 Instance
    return 0;  // sharedInt is automatically deallocated when all of them go out of scope
}//0 Instances
```

Memory Deallocation

```
// Dynamic array deallocation
int* dynamicArray = new int[5];
delete[] dynamicArray;
```

Memory Pool

Memory pools are a form of memory management where a large block of memory is allocated, and then smaller blocks are assigned from within that block as needed.

```
// MemoryPool class definition
class MemoryPool {
private:
    char* block;        // Pointer to the allocated memory block
    std::size_t nextIndex;  // Index to track the next available position in the block

public:
    // Constructor: Allocates memory block of given size
    MemoryPool(std::size_t block_size) : nextIndex(0) {
        block = new char[block_size];
    }

    // Destructor: Frees the allocated memory block
    ~MemoryPool() {
        delete[] block;
    }
```

```
    // Allocate function: Returns a pointer to the next available position in the block
    void* allocate(std::size_t size) {
        void* ptr = block + nextIndex;  // Calculate the pointer to the next position
        nextIndex += size;              // Move the nextIndex to the next available position
        return ptr;
    }

    // DeallocateAll function: Resets the nextIndex to deallocate all memory
    void deallocateAll() {
        nextIndex = 0;
    }
};

int main() {
    // Create a MemoryPool with a block size of 64 bytes
    MemoryPool memoryPool(64);

    // Allocate memory for an integer and set its value
    int* intPtr = static_cast<int*>(memoryPool.allocate(sizeof(int)));
    *intPtr = 42;

    // Allocate memory for a character and set its value
    char* charPtr = static_cast<char*>(memoryPool.allocate(sizeof(char)));
    *charPtr = 'A';

    // Deallocate all memory obtained from the pool
    memoryPool.deallocateAll();

    return 0;
}
```

Question 7 (12 points)

evaluates your ability to provide access to reading and writing files with some specifications.

Reading From Files

Write a program in C++ that reads data from a file called "input.txt" and writes it to another file called "output.txt". Use stream input/output to read and write the data, and handle any errors that may occur during the process.

```
#include <iostream>
#include <fstream>
#include <string>

int main() {
    // Open a file for reading
    std::ifstream inputFile("example.txt");

    // Check if the file is successfully opened
    if (!inputFile.is_open()) {
        std::cerr << "Unable to open the file." << std::endl;
        return 1; // Return an error code
    }

    // Read content from the file line by line
```

```
    std::string line;
    while (std::getline(inputFile, line)) {
        std::cout << line << std::endl; // Display each line
    }

    // Close the file
    inputFile.close();

    return 0;
}
```

Input Mode

```
std::ifstream inputFile("example.txt", std::ios::in);
```

Binary Mode

```
struct MyStruct {
    int data;
    char name[20];
};

MyStruct myData;
inputFile.read(reinterpret_cast<char*>(&myData), sizeof(MyStruct));
```

Writing From Files

```
#include <iostream>
#include <fstream>

int main() {
    // Open a file for writing
    std::ofstream outputFile("example_output.txt");

    // Check if the file is successfully opened
    if (!outputFile.is_open()) {
        std::cerr << "Unable to open the file for writing." << std::endl;
        return 1; // Return an error code
    }

    // Write data to the file
    outputFile << "Hello, World!" << std::endl;
    outputFile << "This is a new line.";

    // Close the file
    outputFile.close();

    std::cout << "Data has been written to the file." << std::endl;

    return 0;
```

```
}
```

File Security

File Permission Constants:

- S_IRUSR (user read)
- S_IWUSR (user write)
- S_IXUSR (user execute)
- S_IRGRP (group read)
- S_IWGRP (group write)
- S_IXGRP (group execute)
- S_IROTH (others read)
- S_IWOTH (others write)
- S_IXOTH (others execute)

Streams

Table 6.1: Modes for Opening Files

Mode	Description
in	Open for reading (default for ifstream).
out	Open for writing (default for ofstream).
app	start reading or writing at the end of the file (append).
ate	erase file before reading or writing (truncate).
nocreate	error when opening if file does not already exist.
noreplace	error when opening for output if file already exists, unless ate or app is set.
binary	open file in binary (not text) mode.

mode & ~std::ios_base::ate						std::fopen access mode	Action if file already exists	Action if file does not exist
binary	in	out	trunc	app	noreplace (since C++23)			
-	+	-	-	-	-	"r"	Read from start	Failure to open
+	+	-	-	-	-	"rb"		
-	+	+	-	-	-	"r+"		Error
+	+	+	-	-	-	"r+b"		
-	-	+	-	-	-	"w"	Destroy contents	Create new
-	-	+	+	-	-			
+	-	+	-	-	-	"wb"		
+	-	+	+	-	-			
-	+	+	+	-	-	"w+"		
+	+	+	+	-	-	"w+b"		
-	-	+	-	-	+	"wx"	Failure to open	Create new
-	-	+	+	-	+			
+	-	+	-	-	+	"wbx"		
+	-	+	+	-	+			
-	+	+	+	-	+	"w+x"		
+	+	+	+	-	+	"w+bx"		
-	-	+	-	+	-	"a"	Write to end	Create new
-	-	-	-	+	-			
+	-	+	-	+	-	"ab"		
+	-	-	-	+	-			
-	+	+	-	+	-	"a+"		
-	+	-	-	+	-			
+	+	+	-	+	-	"a+b"		
+	+	-	-	+	-			

Cursor Manipulation

seekg

```
inputFile.seekg(10, std::ios::beg);
// Move 10 characters from the beginning of the file
```

End Of File

```
while (!inputFile.eof()) {
    // Read from the file
}
```

Ignoring Characters

```
inputFile.ignore(100, '\n'); // Ignore up to 100 characters or until a newline
```

tellg

```
    std::streampos currentPosition = inputFile.tellg();  // Get the current position of the get pointer
```

tellp

```
    // Get and print the current position of the put pointer
    std::streampos currentPosition = outputFile.tellp();
```

Question 8.1-8.3 + 8.4 bonus (6 points each + 6 point bonus)

Code related to links and nodes, divided into three segments, each worth 6 points. There's a bonus for those who complete the entire code, in addition to the required three segments. The main method, crucial for the bonus, must yield specified outputs (but it is optional). Completing only the three segments earns 18 points, and the bonus carries a weight of 6 points.

A linked list is a collection of nodes. Each node has two parts: data and a reference to the next node. Unlike arrays, linked lists don't require contiguous memory allocation. They can easily grow or shrink during runtime. Common types of linked lists include: ➢ Singly linked lists (each node points to the next), ➢ Doubly linked lists (each node points to both the next and previous), and ➢ Circular linked lists (the last node points back to the first). ➢ Trees, Possibly

Potential Topics

- Add
- Remove
- Check for Element

Singly Linked Lists

Insertion

```
#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* next;
};

// this is a weird function. usually you should insert at the end.
Node* insertAtBeginning (Node* head, int data) {
    Node* newNode = new Node;
    newNode->data = data;
    newNode->next = head; // Set the new node's next to the current head return newNode;
    return newNode; // Update the head to the new node
}

void traverse (Node* head) {
    Node* current = head;
```

```
    while (current != nullptr) {
        cout<< current->data << " ";
        current = current->next;
    }
    cout << endl;
}

Node *insertAtEnd (Node* head, int data) {
    Node* newNode = new Node; // create new node
    newNode->data = data; // set new node data
    if (head == nullptr) { // if list is empty, new node becomes the head
        head = newNode;
    }
    Node* current = head;
    while (current->next != nullptr) { //traverse the list until we reach the last node
        current = current->next;
    }
    current->next = newNode; // set the next element of the last node to be our new node
    return newNode; // update the list
}
```

Insertion at beginning Steps:

1. Create a New Node
2. Set the new Node's next to the head of the list
3. Return the New Node as the new head

Insertion at end Steps:

1. Create new node
2. If list is empty, set new node to head
3. Else, traverse the list till you reach the last node, and set its next node to be the new node created in Step 1

Traversal Steps:

1. Set current node to head
2. While the current node is not null, print its data, and set current to current->next

Deletion

```
#include <iostream>

// Define a Node structure
struct Node {
    int data;
    Node* next;

    // Constructor to initialize data and next pointer
    Node(int value) : data(value), next(nullptr) {}
};

// Function to delete the entire linked list
void deleteList(Node** head_ref) {
    Node* current = *head_ref;
    Node* next;
```

```
    while (current != nullptr) {
        next = current->next;
        delete current;
        current = next;
    }

    *head_ref = nullptr; // Set the head to null in the caller
}

// Function to insert a new node at the beginning of the list
void push(Node** head_ref, int new_data) {
    // Allocate a new node
    Node* new_node = new Node(new_data);

    // Link the old list to the new node
    new_node->next = *head_ref;

    // Move the head to point to the new node
    *head_ref = new_node;
}

// Driver code
int main() {
    // Start with an empty list
    Node* head = nullptr;

    // Use push() to construct the list: 1 -> 12 -> 1 -> 4 -> 1
    push(&head, 1);
    push(&head, 4);
    push(&head, 1);
    push(&head, 12);
    push(&head, 1);

    std::cout << "Deleting linked list\n";
    deleteList(&head);
    std::cout << "Linked list deleted\n";

    return 0;
}
```

Searching

Same for Doubly and Circular

```
// Search a node
bool searchNode(struct Node** head_ref, int key){
    struct Node* current = *head_ref;

    while (current != NULL) {

    if (current->data == key){
        return true;
    }

    current = current->next;

    }
```

```
    return false;
}
```

Doubly Linked Lists

Insertation at Head

```
void push(Node** head_ref, int new_data) {
}
// 1. allocate node
Node* new_node = new Node();
// 2. put in the data
new_node->data = new_data;
// 3. Make next of new node as head and previous as NULL
new_node->next = (*head_ref);
new_node->prev = NULL;
// 4. change prev of head node to new node
if ((*head_ref) != NULL)
    (*head_ref)->prev = new_node;
// 5. move the head to point to the new node
(*head_ref) = new_node;
```

Insertation at the End

```
void append(Node** head_ref, int new_data) {
    // 1. allocate node
    Node* new_node = new Node();
    Node* last *head_ref; /* used in step 5*/
    // 2. put in the data
    new_node->data = new_data;
    // 3. This new node is going to be the last node, so make next     of it as NULL
    new_node->next = NULL;
    // 4. If the Linked List is empty, then make the new
    // node as head
    if (*head_ref== NULL) {
        new_node->prev = NULL;
        *head_ref = new_node;
        return;
    }
    // 5. Else traverse till the last node
    while (last->next != NULL)
        last = last->next;
        // 6. Change the next of last node
        last->next = new_node;
        // 7. Make last node as previous of new node
    new_node->prev = last
    return;
}
```

Deletion

```
/* Function to delete a node in a Doubly Linked List head_ref --> pointer to head node pointer. del --> pointer to node to be deleted. */

void deleteNode (Node** head_ref, Node* del)
    /* base case */
    if (*head_ref == NULL || del == NULL)
        return;

    /* If node to be deleted is head node */
    if (*head _ref == del)
        *head_ref = del->next;
    /* Change next only if node to be deleted is NOT the last node */
    if (del->next != NULL)
        del->next->prev = del->prev;
    /* Change prev only if node to be deleted is NOT the first node */
    if (del->prev != NULL)
        del->prev->next = del->next;
    /* Finally, free the memory occupied by del*/ free(del);
```

Circular Linked Lists

Insertation at the Beginning

```
struct Node* addBegin(struct Node* last, int data) {
    if (last == NULL)
        return addToEmpty(last, data);
    // Creating a node dynamically.
    struct Node* temp = (struct Node*)malloc(sizeof(struct Node));
    // Assigning the data.
    temp->data = data;
    // Adjusting the links. temp->next = last->next;
    last->next = temp;
    return last;
}
```

Insertation at the End

```
struct Node* addEnd (struct Node* last, int data) {
    if (last == NULL)
        return addToEmpty(last, data);
    // Creating a node dynamically.
    struct Node* temp = (struct node *)malloc(sizeof(struct Node));     // Assigning the data.
    temp->data = data;
    // Adjusting the links.
    temp->next = last->next;
    last->next = temp;
    last = temp;
    return last;
}
```

Deletion

Question 9 (18 points)

three segments focusing on adding and removing observers for different events. Your ability to complete tasks with specific constructors and classes will be evaluated.

Observer Pattern

```
#include <iostream>
#include <vector>
#include <algorithm>

class Observer{
public:
    virtual void update(const std::string& message) = 0;
};

class Subject {
private:
    std::vector<Observer*> observers;
public:
    void addObserver(Observer* observer){
        observers.push_back(observer);
    }
    void removeObserver(Observer* observer){
        observers.erase(std::remove(observers.begin(),observers.end(),observer),observers.end());
    }

    void notifyObservers(const std::string& message){
        for(Observer* observer : observers){
            observer->update(message);
        }
    }
};

class ConcreteObserver : public Observer{
private:
    std::string name;
public:
    ConcreteObserver(const std::string& name) : name(name){}

    void update(const std::string& message) override{
        std::cout << name << " received message: " << message << std::endl;
    }
};

int main(){
    Subject subject;

    //Create observers
    ConcreteObserver observer1("Observer 1");
    ConcreteObserver observer2("Observer 2");

    //Add observers to the subject
    subject.addObserver(&observer1);
    subject.addObserver(&observer2);

    //Notify observers of an event
```

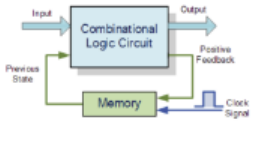
```
    subject.notifyObservers("Event occurred!");

    //Remove an observer
    subject.removeObserver(&observer1);

    //Notify remaining observers
    subject.notifyObservers("Another event occurred!");

    return 0;
}
```

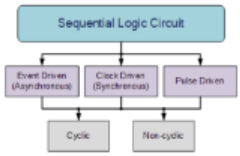
```
OUTPUT:

Observer 1 received message: Event occurred!
Observer 2 received message: Event occurred!
Observer 2 received message: Another event occurred!
```

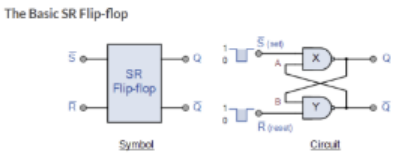
Question 10.1-10.2 (6 points + 14 points)

covers the logic of a JK flip-flop in two segments: the first involves implementation and logic/schematic (6 points), while the second assesses your ability to write a C++ code for an 8-bit binary counter using a JK flip-flop. Be prepared for either synchronous (iteration of 0 to 255 binary) or asynchronous (event-driven) scenarios. This segment is worth 14 points.

SR Flip Flop

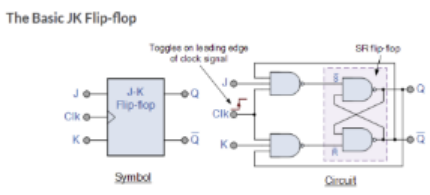


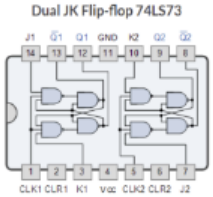




State	S	R	Q	$\overline{Q}$	Description
Set	1	0	0	1	Set $\overline{Q}=1$
	1	1	0	1	no change
Reset	0	1	1	0	Reset $\overline{Q}=0$
	1	1	1	0	no change
Invalid	0	0	1	1	Invalid Condition

JK Flip Flop



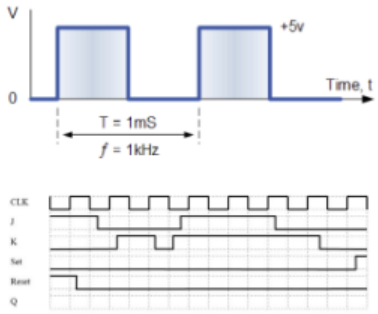


	Clock	Input		Output		Description
	Clk	J	K	Q	$\overline{Q}$	
same as for the SR Latch	X	0	0	1	0	Memory no change
	X	0	0	0	1	
	$\overline{\downarrow}$	0	1	1	0	Reset Q = 0
	X	0	1	0	1	
	$\overline{\downarrow}$	1	0	0	1	Set Q = 1
	X	1	0	1	0	
toggle action	$\overline{\downarrow}$	1	1	0	1	Toggle
	$\overline{\downarrow}$	1	1	1	0	

Code

```
#include <iostream>

class JkFlipFlop {
private: bool Q; bool notQ;
public:
    JkFlipFlop() : Q(false), notQ(true) {}
    void operate(bool J, bool K) {
        if (J && K) {
            Q = !Q;
            notQ = !notQ;
        } else if (J) {
            Q = true;
            notQ = false;
        } else if (K) {
            Q = false;
            notQ = false;
        }

    }
    bool getState() const {
        return Q;
    }
};

int main() {

    const int numFlipFlops = 8;
    JkFlipFlop flipFlops[numFlipFlops];
    bool data[numFlipFlops];
    for (int i = 0; i < numFlipFlops; ++i) {
        data[i] = false;
    }
    auto updateFlipFlop = [&](int input) {
        for (int i = 0; i < numFlipFlops; ++i) {
            bool J = (input >> i) & 1;
            bool K = true;
            flipFlops[i].operate(J, K);
            data[i] = flipFlops[i].getState();
        }
    };
    for (int count = 0; count < 256; ++count) {
        updateFlipFlop(count);
        std::cout << "Count: ";
        for (int i = numFlipFlops - 1; i >= 0; --i) {
            std::cout << data[i];}
        std::cout << std::endl;
    }

    return 0;
}
```

```
#include <iostream>
#include <fstream>
#include <chrono>
#include <thread>
using namespace std;
int main() {
    // Open text files for writing
    ofstream clockFile("clock.txt");
    ofstream dataFile("random_data.txt");
    // Run the code for 10 seconds
    auto startTime = chrono::high_resolution_clock::now();
    while (chrono::duration_cast<chrono::seconds>(chrono::high_resolution_clock::now() -
startTime).count() < 10) {
        // Generate clock pulses
        for (int i = 0; i < 8; ++i) {
            // Assuming clock is represented by '1' during the pulse
            clockFile << "1";
            // Add a newline for better readability in the text file
            clockFile << endl;}
        for (int i = 0; i < 2; ++i) {
            // Assuming clock is represented by '0' during the off period
            clockFile << "0";
            // Add a newline for better readability in the text file
            clockFile << endl;}
        // Generate random binary data for each clock pulse
        for (int i = 0; i < 10; ++i) {
            int randomBit = rand() % 2;
            dataFile << randomBit;
            // Add a newline for better readability in the text file
            dataFile << endl;}}
    clockFile.close();// Close the files
    dataFile.close();
    cout << "Data generation and storage complete." << endl;
    return 0;}
```



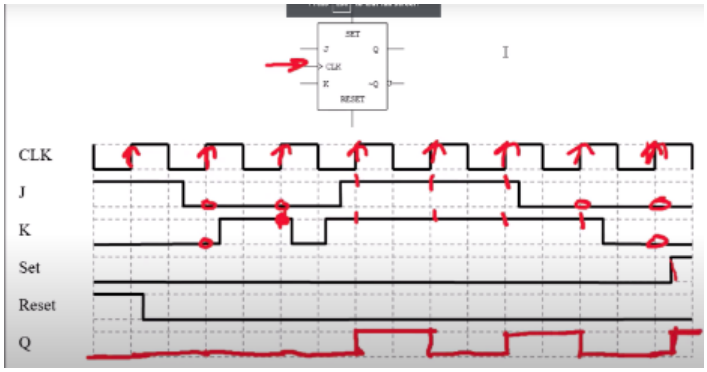

```cpp
#include <iostream>
#include <fstream>
#include <chrono>
#include <thread>

using namespace std;

int main(){
    // Open text files for writing
    ofstream clockFile("clock.txt");
    ofstream dataFile("random_data.txt");

    // Run the oce for 10 seconds
    auto startTime = chrono::high_resolution_clock::now();
    while(chrono::duration_cast<chrono::seconds>(chrono::high_resolution_clock::now() - startTime).count() < 10) {

        //Generate clock pulses
        for (int i = 0; i < 8; ++i) {
            // Assuming clock is represented by '1' during the pulse
            clockFile << "1";

            // Add a newline for better readability in the text file
            clockFile << endl;
        }

        for (int i = 0; i < 2; ++i) {
            // Assuming clock is represented by '0' during the off period
            clockFile << "0";

            // Add a newline for better readability in the text file
            clockFile << endl;
        }

        //Generate random binary data for each clock pulse
        for(int i = 0; i < 10; ++i){
            int randomBit = rand() % 2;
            dataFile << randomBit;

            //Add a newline for better readability in the text file
            dataFile << endl;
        }
    }
```

```
    clockFile.close(); // close the files
    dataFile.close();
    cout << "Data generation and storage complete." << endl;
    return 0;
}
```

The sequential operation of the JK flip flop is exactly the same as for the previous SR flip-flop with the same "Set" and "Reset" inputs. The difference this time is that the "JK flip flop" has no invalid or forbidden input states of the SR Latch even when S and R are both at logic "1".

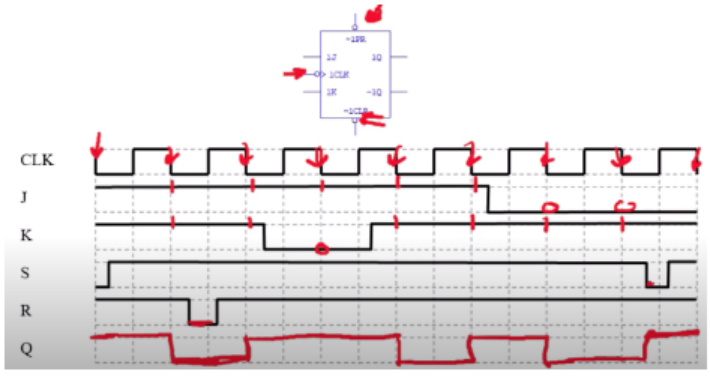
The **JK flip flop** is basically a gated SR flip-flop with the addition of a clock input circuitry that prevents the illegal or invalid output condition that can occur when both inputs S and R are equal to logic level "1". Due to this additional clocked input, a JK flip-flop has four possible input combinations, "logic 1", "logic 0", "no change" and "toggle". The symbol for a JK flip flop is similar to that of an *SR Bistable Latch* as seen in the previous tutorial except for the addition of a clock input.

Timing Diagram



Positively Edged Triggered (So output can only change when the clock moves from low to high) When Set is High, Output is High Priority) When Reset is High, Output is Low Priority

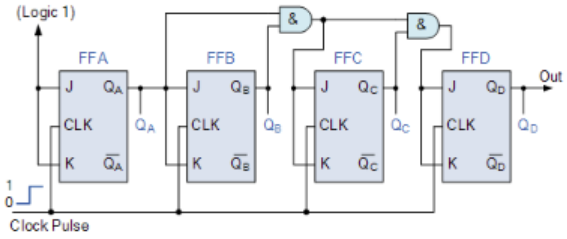
When J and K are High = Toggle the Output When J and K are Low = No Change When J is High and K is Low Q = 1 When K is High and J is Low Q = 0



Negatively Edge Triggered

8 Bit Counter

An 8-bit counter implemented using JK flip-flops is a circuit that counts in binary from 00000000 to 11111111 (0 to 255 in decimal). To create an 8-bit counter, we need eight JK flip-flops, one for each bit. The JK flip-flop is a type of flip-flop that can toggle its output based on the inputs.



JK Flip Flop for a 4 Bit Counter

JK Flip Flop 8 Bit Counter

```
#include <iostream>

class JKFlipFlop {
private:
    bool Q;  // Output
    bool J, K, CLK;

public:
    JKFlipFlop() : Q(false), J(false), K(false), CLK(false) {}

    void setInputs(bool j, bool k, bool clk) {
        J = j;
        K = k;
        CLK = clk;
        process();
    }

    bool getOutput() const {
        return Q;
    }

private:
    void process() {
        if (CLK) {
            if (J && K) {
                Q = !Q;  // Toggle
            } else if (J) {
                Q = true;
            } else if (K) {
                Q = false;
            }
            // Do nothing if both J and K are 0 (no change)
        }
    }
};

class EightBitCounter {
private:
    JKFlipFlop flipFlops[8];

public:
    EightBitCounter() {}
```

```
    void clockCycle() {

        bool done = true;

        while (done) {
            for (int i = 0; i < 8; ++i) {
                if (flipFlops[i].getOutput() == 1) {
                    flipFlops[i].setInputs(true,true,true);
                }

                else {
                    flipFlops[i].setInputs(true,true,true);
                    done = false;
                    break;
                }
            }
        }
    }

    void displayCounterState() const {
        int counterValue = 0;
        for (int i = 7; i >= 0; --i) {
            counterValue = (counterValue << 1) | flipFlops[i].getOutput();
        }
        std::cout << "Counter Value: " << counterValue << std::endl;
    }
};

int main() {
    EightBitCounter counter;

    // Simulate clock cycles
    for (int i = 0; i < 10; ++i) {
        counter.displayCounterState();
        counter.clockCycle();
    }

    return 0;
}
```

Question 11 (6 points)

draw a schematic of a hardware controller based on given logic and fundamentals from the lecture notes.

Pray

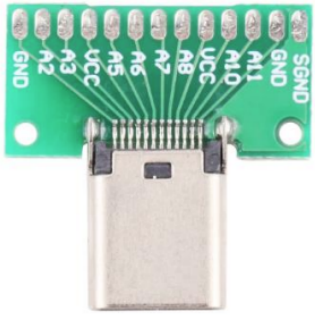
Moradi Moment

USB

USB (Universal Serial Bus):

Definition: USB is a widely used interface for connecting peripherals to computers and other devices. It supports plug-and-play functionality and provides a standardized connection for devices like keyboards, mice, printers, and external storage.

Key Points: High data transfer rates.
Power delivery for connected devices.
Multiple device support through hubs.

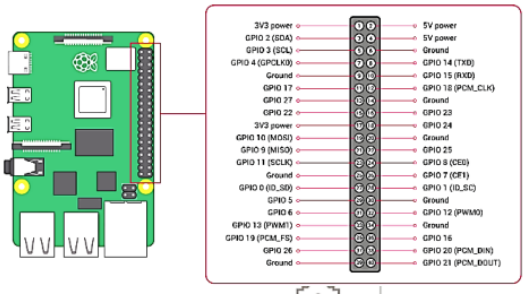


GPIO

GPIO (General Purpose Input/Output):

Definition: GPIO is a generic interface that allows digital signals to be both input and output. It is commonly used in microcontrollers to interact with external digital components.

Key Points: Configurable pins for input or output.
Used for interfacing with sensors, LEDs, buttons, and more.
Simple and versatile for digital signaling.

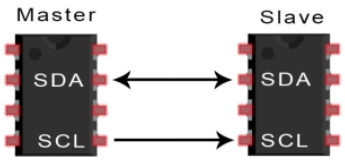


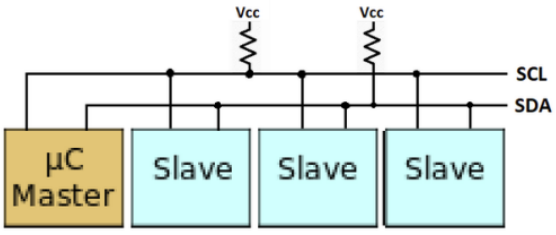
I2S

I2C (Inter-Integrated Circuit):

Definition: I2C is a serial communication protocol that allows multiple devices to communicate on the same bus using a master-slave architecture. It is commonly used for connecting sensors and other low-speed peripherals.

Key Points: Two-wire communication (serial clock 'SCL' and serial data 'SDA').
Supports multiple devices on the same bus.
Allows for easy hardware integration.





SPI

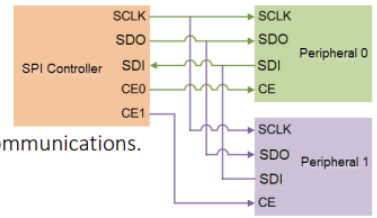
SPI (Serial Peripheral Interface):

Definition: SPI is a synchronous serial communication protocol commonly used for communication between microcontrollers and peripheral devices. It utilizes master-slave communication with multiple devices connected to the same bus.

Key Points: Three or four-wire communication (clock, data in, data out, and optional slave select).

Higher data transfer rates compared to I2C.

Commonly used for connecting sensors, displays, and memory devices.



GPIO 11 (SPI0_SCLK) outputs a serial clock signal to synchronize communications.
GPIO 10 (SPI0_SDO) outputs data to the SPI peripheral device.
GPIO 9 (SPI0_SDI) receives data from the SPI peripheral device.
GPIO 8 (SPI0_CE0) enables one SPI peripheral device.
GPIO 7 (SPI0_CE1) enables the other SPI peripheral device

UART

UART (Universal Asynchronous Receiver/Transmitter):

Definition: UART is a serial communication protocol that uses two wires for asynchronous communication (without use of clock signal – transmission speed are configurable) between devices. It is commonly used for communication between microcontrollers and other peripherals.

Key Points: Asynchronous communication with start and stop bits. UARTs are designed to transmit bytes of data by sending individual bits sequentially, and at the receiving end, a second UART reassembles the bits into complete bytes

Simplex or full-duplex communication.

UART has one connection pin for transmitted data, usually called TX, and another for received data, called RX. These connections are cross-coupled between a transmitter and a receiver. So the TX on one device is connected to the RX on the remote device and vice versa. GND stands for ground.