

DAY 1

1)8-BIT ADDITION :-

LDA 8500

MOV B,A

LDA 8501

ADD B

STA 8502

RST 1

Input :-

ADDRESS(HEX)	ADDRESS	DATA
2134	8500	6
2135	8501	5

Output :-

ADDRESS(HEX)	ADDRESS	DATA
2136	8502	11

2) 8-BIT SUBTRACTION :-

LDA 8000

MOV B, A

LDA 8001

SUB B

STA 8002

RST 1

Input :-

ADDRESS(HEX)	ADDRESS	DATA
1F40	8000	10
1F41	8001	20

Output :-

ADDRESS(HEX)	ADDRESS	DATA
1F42	8002	10

3) 8-BIT MULTIPLICATION :-

```
LXI H,100
MOV B,M
MVI A,00
MOV C,A
INX H
CONT: ADD M
JNC SKIP
INR C
SKIP: DCR B
JNZ CONT
STA 102
MOV A,C
STA 103
HLT
```

Input :-

ADDRESS(HEX)	ADDRESS	DATA
0064	100	10
0065	101	5

Output :-

ADDRESS(HEX)	ADDRESS	DATA
0066	102	50

4) 8-BIT DIVISION :-

```
LDA 8501
MOV B,A
LDA 8500
MVI C,00
LOOP: CMP B
JC LOOP1
SUB B
INR C
JMP LOOP
LOOP1: STA 8502
MOV A,C
STA 8502
RST 1
```

Input :-

ADDRESS(HEX)	ADDRESS	DATA
2134	8500	20
2135	8501	10

Output :-

ADDRESS(HEX)	ADDRESS	DATA
2136	8502	2

5) 16-BIT ADDITION :-

```
LDA 3050
MOV B,A
LDA 3051
ADD B
STA 3052
LDA 3053
MOV B,A
LDA 3054
ADC B
STA 3055
HLT
```

Input :-

ADDRESS(HEX)	ADDRESS	DATA
0BEA	3050	10
0BEB	3051	15

Output :-

ADDRESS(HEX)	ADDRESS	DATA
0BEC	3052	25

6) 16-BIT SUBTRACTION :-

```
LHLD 2050
XCHG
```

LHLD 2052
MVI C,00
MOV A,E
SUB L
STA 2054
MOV A,D
SUB H
STA 2055
HLT

Input :-

ADDRESS(HEX)	ADDRESS	DATA
0802	2050	20
0804	2052	10

Output :-

ADDRESS(HEX)	ADDRESS	DATA
0805	2054	10

7) 16-BIT MULTIPLICATION :-

LHLD 2050
SPHL
LHLD 2052
XCHG
LXI H,0000H
LXI B,0000H
AGAIN: DAD SP
JNC START

```

INX B
START: DCX D
MOV A,E
ORA D
JNZ AGAIN
SHLD 2054
MOV L,C
MOV H,B
SHLD 2055
HLT

```

Input :-

ADDRESS(HEX)	ADDRESS	DATA
0802	2050	10
0804	2052	5

Output :-

ADDRESS(HEX)	ADDRESS	DATA
0806	2054	50

8) 16-BIT DIVISION :-

```

LDA 8501
MOV B,A
LDA 8500
MVI C,00
LOOP: CMP B
JC LOOP1
SUB B

```

INR C
JMP LOOP
STA 8503
DCR C
MOV A,C
LOOP1: STA 8502
RST 1

Input :-

ADDRESS(HEX)	ADDRESS	DATA
2134	8500	10
2135	8501	30

Output :-

ADDRESS(HEX)	ADDRESS	DATA
2136	8502	3

9) FACTORIAL OF A NUMBER :-

LDA 2001
MOV B,A
MVI C,#01
MVI E,#01
LOOP: MOV D,C
MVI A,00H
LP: ADD E
DCR D
JNZ LP
MOV E,A

INR C
DCR B
JNZ LOOP
MOV A,E
STA 2002
HLT

Input :-

ADDRESS(HEX)	ADDRESS	DATA
07D1	2001	5

Output :-

ADDRESS(HEX)	ADDRESS	DATA
07D2	2002	120

10) LARGEST NUMBER IN AN ARRAY :-

LXI H,2050
MOV C,M
DCR C
INX H
MOV A,M
LOOP1: INX H
CMP M
JNC LOOP
MOV A,M
LOOP: DCR C
JNZ LOOP1
STA 2058

HLT

Input :-

ADDRESS(HEX)	ADDRESS	DATA
2142	2050	7
2143	2051	6
2144	2052	8
2145	2053	2
2146	2054	4
2147	2055	9
2148	2056	11
2149	2057	15

Output :-

ADDRESS(HEX)	ADDRESS	DATA
2150	2058	15

DAY 2

1) INTEGER ARITHMETIC ADDITION :-

```
#include <stdio.h>
```

```
int main() {
```

```
int num1, num2;
```

```
int sum;
```

```
    printf("Enter two numbers: ");
```

```
    scanf("%d %d", &num1, &num2);
```

```
    sum = num1 + num2;
```

```
    printf("Sum: %d\n", sum);
```

```
    return 0;
```

```
}
```

Input :-

Enter two numbers: 5 6

Output :-

Sum: 11

2) INTEGER ARITHMETIC SUBTRACTION :-

```
#include <stdio.h>
```

```
int main() {
```

```
    int num1, num2;
```

```
    int diff;
```

```
    printf("Enter two numbers: ");
```

```
    scanf("%d %d", &num1, &num2);
```

```
    diff = num1 - num2;
```

```
    printf("diff: %d\n", diff);
```

```
    return 0;
```

```
}
```

Input :-

Enter two numbers: 6 8

Output :-

diff: -2

3) INTEGER ARITHMETIC MULTIPLICATION :-

```
#include <stdio.h>
```

```
int main() {
```

```
int num1, num2;

int mul;

printf("Enter two numbers: ");

scanf("%d %d", &num1, &num2);

mul = num1 * num2;

printf("mul: %d\n", mul);

return 0;

}
```

Input :-

Enter two numbers: 8 9

Output :-

mul: 72

4) INTEGER ARITHMETIC DIVISION :-

```
#include <stdio.h>

int main() {

    int num1, num2;

    int division;

    printf("Enter two numbers: ");

    scanf("%d %d", &num1, &num2);

    division = num1 / num2;

    printf("quotient: %d\n", division);

    return 0;

}
```

Input :-

Enter two numbers: 6 4

Output :-

quotient: 1

5) FLOATING POINT OF ADDITION :-

```
#include <stdio.h>
```

```
int main() {
```

```
    float num1, num2;
```

```
    float sum;
```

```
    printf("Enter two floating-point numbers: ");
```

```
    scanf("%f %f", &num1, &num2);
```

```
    sum = num1 + num2;
```

```
    printf("Sum: %.2f\n", sum);
```

```
    return 0;
```

```
}
```

Input :-

Enter two floating-point numbers: 9 6

Output :-

Sum: 15.00

6) FLOATING POINT OF SUBTRACTION :-

```
#include <stdio.h>
```

```
int main() {
```

```
    float num1, num2;
```

```
    float sub;
```

```
printf("Enter two floating-point numbers: ");  
  
scanf("%f %f", &num1, &num2);  
  
sub = num1 - num2;  
  
printf("Difference: %.2f\n", sub);  
  
return 0;  
  
}
```

Input :-

Enter two floating-point numbers: 6 4

Output :-

Difference: 2.00

7) FLOATING POINT OF MULTIPLICATION :-

```
#include <stdio.h>  
  
int main() {  
  
    float num1, num2;  
  
    float mult;  
  
    printf("Enter two floating-point numbers: ");  
  
    scanf("%f %f", &num1, &num2);  
  
    mult = num1 * num2;  
  
    printf("Product: %.2f\n", mult);  
  
    return 0;  
  
}
```

Input:

Enter two floating-point numbers: 12.6 45.3

Output:

Product: 570.78

8) FLOATING POINT OF DIVISION :-

```
#include <stdio.h>

int main() {
    float num1, num2;
    float div;
    printf("Enter two floating-point numbers: ");
    scanf("%f %f", &num1, &num2);
    div = num1 / num2;
    printf("Quotient: %.2f\n", div);
    return 0;
}
```

Input:

Enter two floating-point numbers: 1.3 5.6

Output:

Quotient: 0.23

9) RESTORING AND NON RESTORING :-

```
#include <stdio.h>
#include <stdlib.h>

void restoringDivision(int dividend, int divisor) {
    int quotient = 0;
    int remainder = 0;
    dividend = abs(dividend);
    divisor = abs(divisor);
    while (dividend >= divisor) {
        dividend -= divisor;
    }
}
```

```

        quotient++;
    }
    remainder = dividend;
    printf("Restoring Division:\n");
    printf("Quotient: %d\n", quotient);
    printf("Remainder: %d\n", remainder);
}

void nonRestoringDivision(int dividend, int divisor) {
    int quotient = 0;
    int remainder = 0;
    dividend = abs(dividend);
    divisor = abs(divisor);
    while (dividend >= divisor) {
        dividend -= divisor;
        quotient++;
    }
    if (dividend < 0) {
        dividend += divisor;
        quotient--;
    }
    remainder = dividend;
    printf("\nNon-Restoring Division:\n");
    printf("Quotient: %d\n", quotient);
    printf("Remainder: %d\n", remainder);
}

int main() {
    int dividend, divisor;
    printf("Enter dividend: ");
    scanf("%d", &dividend);

```

```
printf("Enter divisor: ");
scanf("%d", &divisor);
restoringDivision(dividend, divisor);
nonRestoringDivision(dividend, divisor);
return 0;
}
```

Input:

Enter dividend: 250

Enter divisor: 50

Output:

Restoring Division:

Quotient: 5

Remainder: 0

Non-Restoring Division:

Quotient: 5

Remainder: 0

10) BOOTH ALGORITHM:

```
#include <stdio.h>

void printBinary(int num, int width) {
    int i;
    for (i = width - 1; i >= 0; i--) {
        printf("%d", (num >> i) & 1);
    }
    printf("\n");
}
```



```

int getSign(int num) {
    if (num < 0) {
        return -1;
    } else if (num > 0) {
        return 1;
    } else {
        return 0;
    }
}

int boothsAlgorithm(int a, int b) {
    int signA = getSign(a);
    int signB = getSign(b);
    int product = 0;
    int multiplier = 0;
    int multiplicand = abs(a);
    int shift = 0;
    while (multiplicand != 0) {
        int bit = (multiplicand >> (sizeof(int) * 8 - 1)) & 1;
        int operation = 0;
        if (bit == 1) {
            operation = signB;
        } else if (multiplier == 1 && bit == 0) {
            operation = -signA;
        }
        product += operation * (1 << shift);
        multiplier = (multiplier << 1) | bit;
        multiplicand <<= 1;
        shift++;
    }
}

```

```

    if (signA == -1 && signB == -1) {
        product = -product;
    } else if (signA == -1) {
        product = -product;
    } else if (signB == -1) {
        product = -product;
    }
    return product;
}

int main() {
    int a = -16;
    int b = 5;
    printf("a = ");
    printBinary(a, 32);
    printf("b = ");
    printBinary(b, 32);
    int product = boothsAlgorithm(a, b);
    printf("Product = %d\n", product);
    printf("Product in binary = ");
    printBinary(product, 32);
    return 0;
}

```

Input :

```

a = 11111111111111111111111111110000
b = 0000000000000000000000000000101

```

Output :

```

Product = -134217728
Product in binary = 11111000000000000000000000000000

```

12) SINGLE PRECISION :-

```
#include <stdio.h>

typedef union {
    float f;
    unsigned int i;
} float_int_union;

void printFloat(float f) {
    float_int_union u;
    u.f = f;
    unsigned int sign = u.i >> 31;
    unsigned int exponent = (u.i >> 23) & 0xFF;
    unsigned int mantissa = u.i & 0x7FFFFFFF;
    if (exponent == 0 && mantissa == 0) {
        printf("0.0\n");
    } else if (exponent == 0xFF && mantissa == 0) {
        printf("NaN\n");
    } else if (exponent == 0xFF && mantissa != 0) {
        printf("Inf\n");
    } else {
        printf("%.2f\n", f);
    }
    printf("Sign: %d\n", sign);
    printf("Exponent: %d\n", exponent - 127);
    printf("Mantissa: ");
    for (int i = 22; i >= 0; i--) {
        printf("%d", (mantissa >> i) & 1);
    }
    printf("\n");
}
```

```

}

int main() {

    float f = 3.14159265358979323846;

    printFloat(f);

    return 0;

}

```

Input :-

3.14

Output :-

Sign: 0

Exponent: 1

Mantissa: 10010010000111111011011

13) DOUBLE PRECISION :-

```

#include <stdio.h>

typedef union {

    double d;

    unsigned long long int i;

} double_int_union;

void printDouble(double d) {

    double_int_union u;

    u.d = d;

    unsigned long long int sign = u.i >> 63;

    unsigned long long int exponent = (u.i >> 52) & 0x7FF;

    unsigned long long int mantissa = u.i & 0xFFFFFFFFFFFFFFF;

    if (exponent == 0 && mantissa == 0) {

        printf("0.0\n");

    } else if (exponent == 0x7FF && mantissa == 0) {

```

```

        printf("NaN\n");
    } else if (exponent == 0x7FF && mantissa != 0) {
        printf("Inf\n");
    } else {
        printf("%.16f\n", d);
    }
    printf("Sign: %d\n", sign);
    printf("Exponent: %d\n", exponent - 1023);
    printf("Mantissa: ");
    for (int i = 51; i >= 0; i--) {
        printf("%d", (mantissa >> i) & 1);
    }
    printf("\n");
}

int main() {
    double d = 3.14159265358979323846;
    printDouble(d);
    return 0;
}

```

Input :

3.1415926535897931

Output :

Sign: 0

Exponent: 1

Mantissa: 1001001000011111101101010100010001000010110100011000

DAY 3

1) Register transferring:-

```
#include <stdio.h>

int main() {

    // Define two variables

    int a = 5;

    int b;


    // Transfer data from variable 'a' to variable 'b' using a register

    b = a;


    // Print the result

    printf("Value of b after register transfer: %d\n", b);


    return 0;
}
```

OUTPUT:

Value of b after register transfer: 5

2) logic operation:-

```
#include <stdio.h>

int main() {

    // Logic operations

    int a = 10; // Binary: 1010

    int b = 6; // Binary: 0110
```

```

// AND operation

int result_and = a & b; // Binary: 0010 (2 in decimal)

printf("AND Operation: %d & %d = %d\n", a, b, result_and);


// OR operation

int result_or = a | b; // Binary: 1110 (14 in decimal)

printf("OR Operation: %d | %d = %d\n", a, b, result_or);


// XOR operation

int result_xor = a ^ b; // Binary: 1100 (12 in decimal)

printf("XOR Operation: %d ^ %d = %d\n", a, b, result_xor);


// NOT operation

int result_not_a = ~a; // Binary: 0101 (Two's complement of 1010 is 0101)
int result_not_b = ~b; // Binary: 1001 (Two's complement of 0110 is 1001)

printf("NOT Operation: ~%d = %d, ~%d = %d\n", a, result_not_a, b, result_not_b);


return 0;
}

```

OUTPUT:

AND Operation: 10 & 6 = 2

OR Operation: 10 | 6 = 14

XOR Operation: 10 ^ 6 = 12

NOT Operation: ~10 = -11, ~6 = -7

3) Half Adder, Full Adder:-

```
#include <stdio.h>
```

```
// Function to perform half adder operation
```

```
void half_adder(int a, int b, int *sum, int *carry) {
```

```
    *sum = a ^ b; // XOR operation gives sum
```

```
    *carry = a & b; // AND operation gives carry
```

```
}
```

```
// Function to perform full adder operation
```

```
void full_adder(int a, int b, int carry_in, int *sum, int *carry_out) {
```

```
    int sum1, sum2, carry1, carry2;
```

```
    // First half adder operation
```

```
    half_adder(a, b, &sum1, &carry1);
```

```
    // Second half adder operation with previous carry
```

```
    half_adder(sum1, carry_in, &sum2, &carry2);
```

```
    // Final sum is XOR of the two half adders
```

```
    *sum = sum2;
```

```
    // Final carry out is OR of the two carries
```

```
    *carry_out = carry1 | carry2;
```

```
}
```

```
int main() {
```

```
    // Inputs
```

```
    int a = 1;
```

```
    int b = 1;
```



```

int carry_in = 0;

// Outputs

int sum, carry_out;

// Perform half adder operation
half_adder(a, b, &sum, &carry_out);
printf("Half Adder: Sum = %d, Carry = %d\n", sum, carry_out);

// Perform full adder operation
full_adder(a, b, carry_in, &sum, &carry_out);
printf("Full Adder: Sum = %d, Carry = %d\n", sum, carry_out);

return 0;
}

```

Output:

Half Adder: Sum = 0, Carry = 1

Full Adder: Sum = 0, Carry = 1

4) Half Subtractor ,Full Subtractor:-

```

#include <stdio.h>

// Function to perform half subtractor operation
void half_subtractor(int a, int b, int *diff, int *borrow) {
    *diff = a ^ b; // XOR operation gives difference
    *borrow = !a & b; // AND operation gives borrow
}

```

```

// Function to perform full subtractor operation

void full_subtractor(int a, int b, int borrow_in, int *diff, int *borrow_out) {
    int diff1, diff2, borrow1, borrow2;

    // First half subtractor operation
    half_subtractor(a, b, &diff1, &borrow1);
    // Second half subtractor operation with previous borrow
    half_subtractor(diff1, borrow_in, &diff2, &borrow2);

    // Final difference is XOR of the two half subtractors
    *diff = diff2;
    // Final borrow out is OR of the two borrows
    *borrow_out = borrow1 | borrow2;
}

int main() {
    // Inputs
    int a = 1;
    int b = 0;
    int borrow_in = 1;

    // Outputs
    int diff, borrow_out;

    // Perform half subtractor operation
    half_subtractor(a, b, &diff, &borrow_out);
    printf("Half Subtractor: Difference = %d, Borrow = %d\n", diff, borrow_out);

    // Perform full subtractor operation

```

```

full_subtractor(a, b, borrow_in, &diff, &borrow_out);

printf("Full Subtractor: Difference = %d, Borrow = %d\n", diff, borrow_out);

return 0;
}

```

OUTPUT:

Half Subtractor: Difference = 1, Borrow = 0

Full Subtractor: Difference = 0, Borrow = 0

5) SINGLE BUS ORGANISATION:-

```

#include <stdio.h>

#define CPU 0
#define MEMORY 1
#define IO 2

int main() {
    int bus;

    // Let's assume CPU, Memory, and IO are connected to a single bus
    // CPU is requesting access to the bus
    bus = CPU;
    printf("CPU requesting access to the bus...\n");
    // Memory is waiting
    if (bus == MEMORY) {
        printf("Memory is waiting while CPU uses the bus.\n");
    }
}

```

```
// IO is waiting
else if (bus == IO) {
    printf("IO is waiting while CPU uses the bus.\n");
}
// CPU gets access to the bus
else {
    printf("CPU is using the bus.\n");
    // Perform CPU operations
}

// Memory is requesting access to the bus
bus = MEMORY;
printf("Memory requesting access to the bus...\n");
// CPU is using the bus
if (bus == CPU) {
    printf("CPU is waiting while Memory uses the bus.\n");
}
// IO is using the bus
else if (bus == IO) {
    printf("IO is using the bus while Memory waits.\n");
}
// Memory gets access to the bus
else {
    printf("Memory is using the bus.\n");
    // Perform Memory operations
}

// IO is requesting access to the bus
bus = IO;
```

```
printf("IO requesting access to the bus...\n");  
// CPU is using the bus  
if (bus == CPU) {  
    printf("CPU is using the bus while IO waits.\n");  
}  
// Memory is using the bus  
else if (bus == MEMORY) {  
    printf("Memory is using the bus while IO waits.\n");  
}  
// IO gets access to the bus  
else {  
    printf("IO is using the bus.\n");  
    // Perform IO operations  
}  
  
return 0;  
}
```

OUTPUT:-

CPU requesting access to the bus...

CPU is using the bus.

Memory requesting access to the bus...

Memory is using the bus.

IO requesting access to the bus...

IO is using the bus.

6) MULTIPLE BUS ORGANISATION:-

```
#include <stdio.h>
```

```
// Structure representing a bus
```

```
typedef struct {
```

```
    int data;
```

```
    int address;
```

```
} Bus;
```

```
// Structure representing a CPU
```

```
typedef struct {
```

```
    Bus *bus;
```

```
} CPU;
```

```
// Structure representing a Memory
```

```
typedef struct {
```

```
    Bus *bus;
```

```
    int data[100]; // For simplicity, assuming memory size of 100 locations
```

```
} Memory;
```

```
// Structure representing an I/O Device
```

```
typedef struct {
```

```
    Bus *bus;
```

```
} IO_Device;
```

```
// Function to read data from memory
```

```
int memory_read(Memory *mem, int address) {
```

```
    return mem->data[address];
```

```
}
```

```
// Function to write data to memory
```

```
void memory_write(Memory *mem, int address, int data) {
```

```
    mem->data[address] = data;
```

```
}
```

```
// Function to perform CPU operation (read from memory)
```

```
int cpu_operation(CPU *cpu, Memory *mem, int address) {
```

```
    return memory_read(mem, address);
```

```
}
```

```
// Function to perform I/O device operation (write to memory)
```

```
void io_device_operation(IO_Device *device, Memory *mem, int address, int data) {
```

```
    memory_write(mem, address, data);
```

```
}
```

```
int main() {
```

```
    // Initialize buses, CPU, memory, and I/O device
```

```
    Bus data_bus;
```

```
    Bus io_bus;
```

```
    CPU cpu;
```

```
    Memory memory;
```

```
    IO_Device io_device;
```

```
    // Set bus pointers
```

```
    cpu.bus = &data_bus;
```

```
    memory.bus = &data_bus;
```

```
    io_device.bus = &io_bus;
```

```

// Write data to memory
memory_write(&memory, 0, 10); // Writing value 10 at address 0

// CPU reads data from memory
int data_read = cpu_operation(&cpu, &memory, 0);
printf("Data read by CPU: %d\n", data_read);

// I/O device writes data to memory
io_device_operation(&io_device, &memory, 1, 20); // Writing value 20 at address 1

// CPU reads updated data from memory
data_read = cpu_operation(&cpu, &memory, 1);
printf("Data read by CPU after I/O operation: %d\n", data_read);

return 0;
}

```

OUTPUT:

Data read by CPU: 10

Data read by CPU after I/O operation: 20

7) TWO STAGE PIPELINING:-

```
#include <stdio.h>
```

```
// Structure representing an instruction
```

```
typedef struct {
```



```
int opcode;
int operand1;
int operand2;
} Instruction;
```

```
// Function to simulate instruction fetch stage
```

```
void fetch_stage(int *instruction_count, Instruction *instruction_buffer) {
    // Simulating fetching instructions from memory
    // Increment instruction count
    (*instruction_count)++;
    // Simulating instruction decoding and filling instruction buffer
    instruction_buffer->opcode = (*instruction_count) % 3; // Example: alternating
    opcodes
    instruction_buffer->operand1 = (*instruction_count) * 2;
    instruction_buffer->operand2 = (*instruction_count) * 2 + 1;
}
```

```
// Function to simulate instruction execution stage
```

```
void execute_stage(Instruction *instruction_buffer, int *result) {
    // Simulating instruction execution
    switch (instruction_buffer->opcode) {
        case 0:
            *result = instruction_buffer->operand1 + instruction_buffer->operand2;
            break;
        case 1:
            *result = instruction_buffer->operand1 - instruction_buffer->operand2;
            break;
        case 2:
            *result = instruction_buffer->operand1 * instruction_buffer->operand2;
            break;
    }
```

```

        default:
            printf("Invalid opcode\n");
            break;
    }
}

int main() {
    int instruction_count = 0;
    Instruction current_instruction;
    int execution_result;

    // Perform multiple cycles of instruction fetch and execution
    for (int i = 0; i < 5; i++) { // Example: 5 cycles
        // Instruction fetch stage
        fetch_stage(&instruction_count, &current_instruction);

        // Instruction execution stage
        execute_stage(&current_instruction, &execution_result);

        // Output the result of the executed instruction
        printf("Cycle %d: Result = %d\n", i + 1, execution_result);
    }

    return 0;
}

```

OUTPUT:

Cycle 1: Result = -1

Cycle 2: Result = 20

Cycle 3: Result = 13

Cycle 4: Result = -1

Cycle 5: Result = 110

8) FOUR STAGE PIPELINING:-

```
#include <stdio.h>
```

```
// Structure representing an instruction
```

```
typedef struct {
```

```
    int opcode;
```

```
    int operand1;
```

```
    int operand2;
```

```
} Instruction;
```

```
// Structure representing the pipeline registers
```

```
typedef struct {
```

```
    Instruction instruction;
```

```
    int result;
```

```
} PipelineRegister;
```

```
// Function to simulate instruction fetch stage
```

```
void fetch_stage(int *instruction_count, Instruction *current_instruction) {
```

```
    // Simulating fetching instructions from memory
```

```
    // Increment instruction count
```

```
    (*instruction_count)++;
```

```
    // Simulating instruction decoding
```

```
    current_instruction->opcode = (*instruction_count) % 3; // Example: alternating  
    opcodes
```

```
    current_instruction->operand1 = (*instruction_count) * 2;
```

```

    current_instruction->operand2 = (*instruction_count) * 2 + 1;
}

// Function to simulate instruction decode stage
void decode_stage(Instruction *current_instruction, PipelineRegister *decode_reg) {
    // Transfer the instruction to the decode register
    decode_reg->instruction = *current_instruction;
}

// Function to simulate execute stage
void execute_stage(PipelineRegister *decode_reg, PipelineRegister *execute_reg) {
    // Simulating instruction execution
    switch (decode_reg->instruction.opcode) {
        case 0:
            execute_reg->result = decode_reg->instruction.operand1 + decode_reg->instruction.operand2;
            break;
        case 1:
            execute_reg->result = decode_reg->instruction.operand1 - decode_reg->instruction.operand2;
            break;
        case 2:
            execute_reg->result = decode_reg->instruction.operand1 * decode_reg->instruction.operand2;
            break;
        default:
            printf("Invalid opcode\n");
            break;
    }
}

```

```

// Function to simulate writeback stage
void writeback_stage(PipelineRegister *execute_reg) {
    // Printing the result obtained from the execution stage
    printf("Result: %d\n", execute_reg->result);
}

int main() {
    int instruction_count = 0;
    Instruction current_instruction;
    PipelineRegister decode_reg, execute_reg;

    // Perform multiple cycles of the pipeline stages
    for (int i = 0; i < 5; i++) { // Example: 5 cycles
        // Instruction fetch stage
        fetch_stage(&instruction_count, &current_instruction);

        // Instruction decode stage
        decode_stage(&current_instruction, &decode_reg);

        // Instruction execute stage
        execute_stage(&decode_reg, &execute_reg);

        // Instruction writeback stage
        writeback_stage(&execute_reg);

        // Output the current instruction being processed
        printf("Cycle %d: Instruction Opcode = %d, Operand1 = %d, Operand2 = %d\n",
            i + 1, current_instruction.opcode, current_instruction.operand1,
            current_instruction.operand2);
    }
}

```

```
}  
  
    return 0;  
}
```

OUTPUT:

```
Result: -1  
Cycle 1: Instruction Opcode = 1, Operand1 = 2, Operand2 = 3  
Result: 20  
Cycle 2: Instruction Opcode = 2, Operand1 = 4, Operand2 = 5  
Result: 13  
Cycle 3: Instruction Opcode = 0, Operand1 = 6, Operand2 = 7  
Result: -1  
Cycle 4: Instruction Opcode = 1, Operand1 = 8, Operand2 = 9  
Result: 110  
Cycle 5: Instruction Opcode = 2, Operand1 = 10, Operand2 = 11
```

9) STATIC PREDICTION:-

```
#include <stdio.h>  
  
#define TAKEN 1  
#define NOT_TAKEN 0  
  
// Function to simulate static prediction  
int static_prediction(int instruction_address) {  
    // Implement a simple strategy based on the instruction address  
    if (instruction_address % 2 == 0) {  
        // Predict taken for even instruction addresses  
        return TAKEN;  
    }  
}
```

```

    } else {
        // Predict not taken for odd instruction addresses
        return NOT_TAKEN;
    }
}

int main() {
    // Example instruction addresses
    int instruction_addresses[] = {100, 101, 102, 103, 104};
    int num_instructions = sizeof(instruction_addresses) /
sizeof(instruction_addresses[0]);

    printf("Static prediction results:\n");
    for (int i = 0; i < num_instructions; i++) {
        int prediction = static_prediction(instruction_addresses[i]);
        printf("Instruction at address %d: Prediction = %s\n", instruction_addresses[i],
            prediction == TAKEN ? "Taken" : "Not Taken");
    }

    return 0;
}

```

OUTPUT:

Static prediction results:

Instruction at address 100: Prediction = Taken

Instruction at address 101: Prediction = Not Taken

Instruction at address 102: Prediction = Taken

Instruction at address 103: Prediction = Not Taken

Instruction at address 104: Prediction = Taken

10) DYNAMIC PREDICTION:-

```
#include <stdio.h>

#define TAKEN 1
#define NOT_TAKEN 0
#define STRONGLY_TAKEN 3
#define STRONGLY_NOT_TAKEN 0

// Structure representing a branch predictor
typedef struct {
    int state; // State of the predictor (2-bit saturating counter)
} BranchPredictor;

// Initialize the branch predictor
void init_predictor(BranchPredictor *predictor) {
    predictor->state = STRONGLY_NOT_TAKEN;
}

// Predict the outcome of a branch instruction
int predict(BranchPredictor *predictor) {
    if (predictor->state >= 2) {
        return TAKEN;
    } else {
        return NOT_TAKEN;
    }
}
```



```

// Update the branch predictor based on the actual outcome
void update_predictor(BranchPredictor *predictor, int actual_outcome) {
    if (actual_outcome == TAKEN) {
        if (predictor->state < STRONGLY_TAKEN) {
            predictor->state++;
        }
    } else {
        if (predictor->state > STRONGLY_NOT_TAKEN) {
            predictor->state--;
        }
    }
}

int main() {
    BranchPredictor predictor;
    init_predictor(&predictor);

    // Simulate branch prediction for a sequence of branch instructions
    int branch_outcomes[] = {TAKEN, TAKEN, NOT_TAKEN, TAKEN, NOT_TAKEN};
    int num_branches = sizeof(branch_outcomes) / sizeof(branch_outcomes[0]);

    printf("Branch prediction results:\n");
    for (int i = 0; i < num_branches; i++) {
        int prediction = predict(&predictor);
        printf("Branch %d: Prediction = %s\n", i + 1, prediction == TAKEN ? "Taken" : "Not
Taken");
        update_predictor(&predictor, branch_outcomes[i]);
    }

    return 0;
}

```

```
}
```

OUTPUT:

Branch prediction results:

Branch 1: Prediction = Not Taken

Branch 2: Prediction = Not Taken

Branch 3: Prediction = Taken

Branch 4: Prediction = Not Taken

Branch 5: Prediction = Taken

11) Data hazards:-

```
#include <stdio.h>
```

```
int main() {
```

```
    int a = 5;
```

```
    int b = 10;
```

```
    int c;
```

```
    // Instruction 1: Add 'a' and 'b' and store the result in 'c'
```

```
    c = a + b;
```

```
    // Instruction 2: Multiply 'c' by 2 and store the result in 'c'
```

```
    c = c * 2;
```

```
    // Instruction 3: Print the value of 'c'
```

```
    printf("Result: %d\n", c);
```

```
    return 0;
```

```
}
```

OUTPUT:

Result: 30

12) Instruction Hazards:-

```
#include <stdio.h>

int main() {
    int a = 5;
    int b = 10;
    int c;

    // Instruction 1: Load the value of 'a' into a register
    int temp_a = a;

    // Instruction 2: Load the value of 'b' into a register
    int temp_b = b;

    // Instruction 3: Add the values stored in the two registers and store the result in 'c'
    c = temp_a + temp_b;

    // Instruction 4: Multiply 'c' by 2 and store the result in 'c'
    c = c * 2;

    // Instruction 5: Print the value of 'c'
    printf("Result: %d\n", c);

    return 0;
```

```
}
```

OUTPUT:

Result: 30

13) Structure Hazards:-

```
#include <stdio.h>
```

```
int main() {
```

```
    int a = 5;
```

```
    int b = 10;
```

```
    int c;
```

```
    // Instruction 1: Compare 'a' and 'b' and set a flag if 'a' is greater than 'b'
```

```
    int flag = (a > b);
```

```
    // Instruction 2: If the flag is set, add 'a' and 'b' and store the result in 'c', else store 'b'
    in 'c'
```

```
    if (flag) {
```

```
        c = a + b;
```

```
    } else {
```

```
        c = b;
```

```
    }
```

```
    // Instruction 3: Multiply 'c' by 2 and store the result in 'c'
```

```
    c = c * 2;
```

```
    // Instruction 4: Print the value of 'c'
```

```
printf("Result: %d\n", c);

return 0;
}
```

OUTPUT:

Result: 20

14) SUPER SCALAR processing:-

```
#include <stdio.h>

// Function to perform addition
int add(int a, int b) {
    return a + b;
}

// Function to perform subtraction
int subtract(int a, int b) {
    return a - b;
}

int main() {
    int a = 5;
    int b = 10;
    int c, d;

    // Superscalar processing:
    // Execute multiple instructions in parallel if possible
```

```
// Stage 1: Instruction Fetch
// Instructions are fetched simultaneously

// Stage 2: Instruction Decode
// Instructions are decoded in parallel

// Stage 3: Execution
// Instructions are executed in parallel
c = add(a, b);
d = subtract(a, b);

// Stage 4: Writeback
// Results are written back to registers

// Print the results
printf("Result of addition: %d\n", c);
printf("Result of subtraction: %d\n", d);

return 0;
}
```

OUTPUT:

Result of addition: 15

Result of subtraction: -5

DAY 4

1)RANDOM ACCESS MEMORY

```
#include <stdio.h>

#define SIZE 10 // Size of the array

int main() {

    int arr[SIZE]; // Array to store values
    int i, index, value, choice;

    // Initialize array with zeros
    for (i = 0; i < SIZE; i++) {
        arr[i] = 0;
    }

    // Menu for user interaction
    do {
        printf("\nRandom Access Memory (RAM) Operations\n");
        printf("1. Write to RAM\n");
        printf("2. Read from RAM\n");
        printf("3. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter index (0 to %d): ", SIZE - 1);
                scanf("%d", &index);
                if (index >= 0 && index < SIZE) {
```

```

        printf("Enter value to write: ");
        scanf("%d", &value);
        arr[index] = value;
        printf("Value written to RAM.\n");
    } else {
        printf("Invalid index. Please enter a valid index.\n");
    }
    break;
case 2:
    printf("Enter index (0 to %d): ", SIZE - 1);
    scanf("%d", &index);
    if (index >= 0 && index < SIZE) {
        printf("Value at index %d: %d\n", index, arr[index]);
    } else {
        printf("Invalid index. Please enter a valid index.\n");
    }
    break;
case 3:
    printf("Exiting...\n");
    break;
default:
    printf("Invalid choice. Please enter a valid choice.\n");
}

} while (choice != 3);

return 0;
}

```


OUTPUT:

Random Access Memory (RAM) Operations

1. Write to RAM
2. Read from RAM
3. Exit

Enter your choice: 1

Enter index (0 to 9): 5

Enter value to write: 2

Value written to RAM.

Random Access Memory (RAM) Operations

1. Write to RAM
2. Read from RAM
3. Exit

Enter your choice: 2

Enter index (0 to 9): 5

Value at index 5: 2

Random Access Memory (RAM) Operations

1. Write to RAM
2. Read from RAM
3. Exit

Enter your choice: 3

Exiting...

2) READ ONLY MEMORY

```
#include <stdio.h>
```

```

// Define the size of the ROM
#define ROM_SIZE 10

// Define the ROM data (example data)
int rom[ROM_SIZE] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};

int main() {
    int address;

    printf("Enter the address (0 to %d) to read from the ROM: ", ROM_SIZE - 1);
    scanf("%d", &address);

    // Check if the entered address is valid
    if (address >= 0 && address < ROM_SIZE) {
        printf("Data at address %d: %d\n", address, rom[address]);
    } else {
        printf("Invalid address! Please enter a valid address between 0 and %d\n",
ROM_SIZE - 1);
    }

    return 0;
}

```

OUTPUT

Enter the address (0 to 9) to read from the ROM: 6

Data at address 6: 13

3) VIRTUAL MEMORY

```

#include <stdio.h>

#include <stdlib.h>

```

```
#define MEMORY_SIZE 100
```

```
int virtual_memory[MEMORY_SIZE];
```

```
void write_to_memory(int address, int value) {  
    if (address >= 0 && address < MEMORY_SIZE) {  
        virtual_memory[address] = value;  
        printf("Value %d written to address %d\n", value, address);  
    } else {  
        printf("Error: Address out of bounds!\n");  
    }  
}
```

```
int read_from_memory(int address) {  
    if (address >= 0 && address < MEMORY_SIZE) {  
        printf("Value at address %d: %d\n", address, virtual_memory[address]);  
        return virtual_memory[address];  
    } else {  
        printf("Error: Address out of bounds!\n");  
        return -1; // Return an error value  
    }  
}
```

```
int main() {  
    int choice, address, value;  
  
    printf("Welcome to Virtual Memory System\n");
```

```
do {  
    printf("\n1. Read from memory\n");  
    printf("2. Write to memory\n");  
    printf("3. Exit\n");  
    printf("Enter your choice: ");  
    scanf("%d", &choice);  
  
    switch (choice) {  
        case 1:  
            printf("Enter address to read from: ");  
            scanf("%d", &address);  
            read_from_memory(address);  
            break;  
        case 2:  
            printf("Enter address to write to: ");  
            scanf("%d", &address);  
            printf("Enter value to write: ");  
            scanf("%d", &value);  
            write_to_memory(address, value);  
            break;  
        case 3:  
            printf("Exiting...\n");  
            break;  
        default:  
            printf("Invalid choice! Please try again.\n");  
    }  
} while (choice != 3);  
  
return 0;
```

}

OUTPUT

Welcome to Virtual Memory System

1. Read from memory

2. Write to memory

3. Exit

Enter your choice: 1

Enter address to read from: 50

Value at address 50: 0

1. Read from memory

2. Write to memory

3. Exit

Enter your choice: 1

Enter address to read from: 52

Value at address 52: 0

1. Read from memory

2. Write to memory

3. Exit

Enter your choice: 2

Enter address to write to: 52

Enter value to write: 5

Value 5 written to address 52

1. Read from memory

2. Write to memory

3. Exit

Enter your choice: 3

Exiting...

4) MEMORY ALLOCATION

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
int main() {
```

```
    int *array; // Pointer to store dynamically allocated memory
```

```
    int size, i;
```

```
    // Asking user for the size of the array
```

```
    printf("Enter the size of the array: ");
```

```
    scanf("%d", &size);
```

```
    // Dynamically allocating memory for the array
```

```
    array = (int *)malloc(size * sizeof(int));
```

```
    // Checking if memory allocation was successful
```

```
    if (array == NULL) {
```

```
        printf("Memory allocation failed!\n");
```

```
        return 1; // Exiting the program with error code
```

```
    }
```

```
    // Getting input for the array elements from the user
```

```
    printf("Enter %d elements:\n", size);
```

```
    for (i = 0; i < size; i++) {
```

```
        scanf("%d", &array[i]);
```

```
}

// Displaying the array elements
printf("The elements in the array are: ");
for (i = 0; i < size; i++) {
    printf("%d ", array[i]);
}
printf("\n");

// Freeing dynamically allocated memory
free(array);

return 0; // Exiting the program successfully
}
```

OUTPUT

Enter the size of the array: 5

Enter 5 elements:

2 3 4 5 6

The elements in the array are: 2 3 4 5 6

5) CACHE MEMORY

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#define CACHE_SIZE 256 // Define cache size in bytes
```

```
#define MAIN_MEMORY_SIZE 1024 // Define main memory size in bytes
```

```

// Cache structure
typedef struct {

    int valid; // Valid bit to indicate whether the cache block contains valid data

    int tag; // Tag bits for identifying the memory block

    int data; // Data stored in the cache block
} CacheBlock;

// Function to simulate cache access
void accessCache(int address, CacheBlock *cache) {

    int cacheIndex = address % CACHE_SIZE; // Calculate cache index

    int tag = address / CACHE_SIZE; // Calculate tag bits

    // Check if cache block is valid and contains the required data
    if (cache[cacheIndex].valid && cache[cacheIndex].tag == tag) {
        printf("Cache hit! Data found at address %d\n", address);
    } else {
        // Cache miss scenario

        printf("Cache miss! Data not found at address %d\n", address);

        // Load data from main memory to cache

        cache[cacheIndex].valid = 1;

        cache[cacheIndex].tag = tag;

        cache[cacheIndex].data = address; // For simplicity, storing the address as data
    }
}

int main() {

    CacheBlock cache[CACHE_SIZE]; // Cache memory

    int memory[MAIN_MEMORY_SIZE]; // Main memory

```



```

// Initialize cache and main memory
for (int i = 0; i < CACHE_SIZE; i++) {
    cache[i].valid = 0;
    cache[i].tag = 0;
    cache[i].data = 0;
}
for (int i = 0; i < MAIN_MEMORY_SIZE; i++) {
    memory[i] = i; // Initialize main memory with sequential data
}

int address;

// Taking input for memory address
printf("Enter the memory address to access (0 - %d): ", MAIN_MEMORY_SIZE - 1);
scanf("%d", &address);

// Accessing cache with the given memory address
accessCache(address, cache);

return 0; // Exiting the program successfully
}

```

OUTPUT

Enter the memory address to access (0 - 1023): 1009

Cache miss! Data not found at address 1009

DAY 5

1)ACCESSING I/O DEVICES:

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
#include <signal.h>
```

```
// Interrupt handler function for Device 1
```

```
void device1_interrupt_handler(int signal) {
```

```
    printf("Device 1 Interrupt Handler\n");
```

```
    // Handle Device 1 interrupt
```

```
}
```

```
// Interrupt handler function for Device 2
```

```
void device2_interrupt_handler(int signal) {
```

```
    printf("Device 2 Interrupt Handler\n");
```

```
    // Handle Device 2 interrupt
```

```
}
```

```
// Enable interrupt for a device
```

```
void enable_interrupt(int device) {
```

```
    switch(device) {
```

```
        case 1:
```

```
            signal(SIGINT, device1_interrupt_handler);
```

```
            break;
```

```
        case 2:
```

```
            signal(SIGINT, device2_interrupt_handler);
```

```
        break;

        default:

            printf("Invalid device\n");

            break;

    }

}
```

```
// Disable interrupt for a device
```

```
void disable_interrupt(int device) {

    signal(SIGINT, SIG_DFL);

}
```

```
int main() {
```

```
    // Enable interrupts for both devices
```

```
    enable_interrupt(1);
```

```
    enable_interrupt(2);
```

```
    printf("Interrupts enabled for both devices. Press Ctrl+C to trigger interrupts.\n");
```

```
    // Loop indefinitely to simulate device operations
```

```
    while (1) {
```

```
        // Perform other tasks or wait for interrupts
```

```
    }
```

```
    // Disable interrupts before exiting
```

```
    disable_interrupt(1);
```

```
disable_interrupt(2);

return 0;
}
```

OUTPUT:

Interrupts enabled for both devices. Press Ctrl+C to trigger interrupts.

Device 2 Interrupt Handler

2) BUS ARBITRARY:

```
#include <stdio.h>

#define NUM_DEVICES 3

// Function to simulate bus arbitration using round-robin
int busArbitrationRoundRobin(int currentDevice) {
    return (currentDevice + 1) % NUM_DEVICES;
}

int main() {
    int currentDevice = 0; // Start with the first device

    printf("Bus Arbitration Simulation\n");

    // Simulate 10 cycles of bus access
```

```

    for (int cycle = 1; cycle <= 10; cycle++) {
        printf("Cycle %d: Device %d accesses the bus\n", cycle, currentDevice);

        // Perform some operation or task for the device accessing the bus
        // ...

        // Update the current device using round-robin arbitration
        currentDevice = busArbitrationRoundRobin(currentDevice);

        // Print a newline for better readability
        printf("\n");
    }

    return 0;
}

```

OUTPUT:

/tmp/xEQvQsC0hU.o

Bus Arbitration Simulation

Cycle 1: Device 0 accesses the bus

Cycle 2: Device 1 accesses the bus

Cycle 3: Device 2 accesses the bus

Cycle 4: Device 0 accesses the bus

Cycle 5: Device 1 accesses the bus

Cycle 6: Device 2 accesses the bus

Cycle 7: Device 0 accesses the bus

Cycle 8: Device 1 accesses the bus

Cycle 9: Device 2 accesses the bus

Cycle 10: Device 0 accesses the bus

3) Direct Memory Addressing:

```
#include <stdio.h>
```

```
int main() {
```

```
    int variable = 42;
```

```
    int *pointer = &variable;
```

```
    printf("Original value at memory location: %d\n", *pointer);
```

```
    // Modify the value at the memory location directly
```

```
    *pointer = 99;
```

```
    printf("Modified value at memory location: %d\n", *pointer);
```

```
    return 0;
```

```
}
```

OUTPUT:

```
/tmp/xEQvQsC0hU.o
```

Original value at memory location: 42

Modified value at memory location: 99

4) PIPELINE INTERRUPTS:

```
#include <stdio.h>

#include <signal.h>

#include <unistd.h>


// Signal handler function

void handleInterrupt(int signum) {

    printf("Received interrupt signal (Ctrl+C) - exiting\n");

    // You can perform cleanup or additional actions here

    _exit(0); // Terminate the program immediately

}


int main() {

    // Set up the signal handler

    if (signal(SIGINT, handleInterrupt) == SIG_ERR) {

        perror("Error setting up signal handler");

        return 1;

    }


    // Run an infinite loop

    while (1) {

        printf("Program is running...\n");

        sleep(1); // Simulate some work

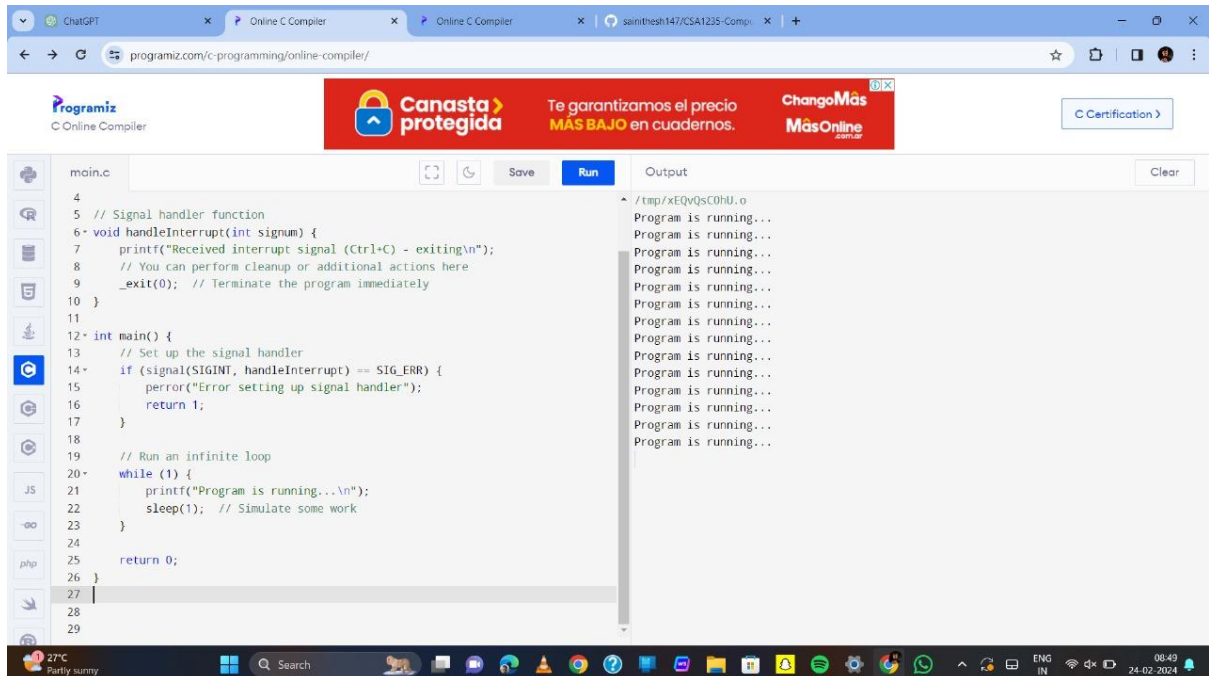
    }


    return 0;

}
```

```
}
```

OUTPUT:



The screenshot shows a web browser with multiple tabs, including 'ChatGPT', 'Online C Compiler', and 'sainthesh147/CSA1235-Comp'. The active tab is 'Online C Compiler' at the URL 'programiz.com/c-programming/online-compiler/'. The page features a header with 'Programiz C Online Compiler' and several advertisements for 'Canasta protegida', 'Te garantizamos el precio MAS BAJO en cuadernos.', and 'Change Más MasOnline'. The main area is divided into two panels. The left panel, titled 'main.c', contains the following C code:

```
4
5 // Signal handler function
6 void handleInterrupt(int signum) {
7     printf("Received interrupt signal (Ctrl+C) - exiting\n");
8     // You can perform cleanup or additional actions here
9     _exit(0); // Terminate the program immediately
10 }
11
12 int main() {
13     // Set up the signal handler
14     if (signal(SIGINT, handleInterrupt) == SIG_ERR) {
15         perror("Error setting up signal handler");
16         return 1;
17     }
18
19     // Run an infinite loop
20     while (1) {
21         printf("Program is running...\n");
22         sleep(1); // Simulate some work
23     }
24
25     return 0;
26 }
27
28
29
```

The right panel, titled 'Output', shows the program's execution output:

```
/tmp/xEQvQsCOH0.o
Program is running...
Program is running...
Program is running...
Program is running...
Program is running...
Program is running...
Program is running...
Program is running...
Program is running...
Program is running...
Program is running...
Program is running...
Program is running...
Program is running...
Program is running...
Program is running...
Program is running...
Program is running...
Program is running...
Program is running...
```

The bottom of the image shows a Windows taskbar with a search bar, several application icons, and system tray information including '27°C Partly sunny', 'ENG IN', and the date '24.02.2024'.

5) PCI INTERRUPTS:

```
#include <stdlib.h>
```

```
#include <unistd.h>
```

```
#include <signal.h>
```

```
// Define a signal handler function for the interrupt
```

```
void handle_interrupt(int signum) {
```

```
    printf("PCI Interrupt Received! Signal Number: %d\n", signum);
```

```
}
```

```
int main() {
```

```
    // Install the signal handler for SIGINT (you might need to use a different signal)
```



```

        if (signal(SIGINT, handle_interrupt) == SIG_ERR) {

perror("Error setting up signal handler");

exit(EXIT_FAILURE);

        }


        // Simulate some work in the main loop

        while (1) {

printf("Main Loop: Working...\n");

        sleep(1);

        }


return 0;

}

```

Output:

The screenshot displays the Programiz Online C Compiler interface. The code editor on the left contains the following C program:

```

1 #include <stdlib.h>
2 #include <unistd.h>
3 #include <signal.h>
4
5 // Define a signal handler function for the interrupt
6 void handle_interrupt(int signal) {
7     printf("PCI Interrupt Received! Signal Number: %d\n", signal);
8 }
9
10
11 int main() {
12     // Install the signal handler for SIGINT (you might need to use a different
13     // signal)
14     if (signal(SIGINT, handle_interrupt) == SIG_ERR) {
15         perror("Error setting up signal handler");
16         exit(EXIT_FAILURE);
17     }
18
19     // Simulate some work in the main loop
20     while (1) {
21         printf("Main Loop: Working...\n");
22         sleep(1);
23     }
24
25     return 0;
26 }

```

The output window on the right shows the program's execution results, displaying the message "Main Loop: Working..." repeatedly, indicating that the program is running in an infinite loop as intended.