

EXP NO: 25 DECIMAL TO BINARY CONVERSION

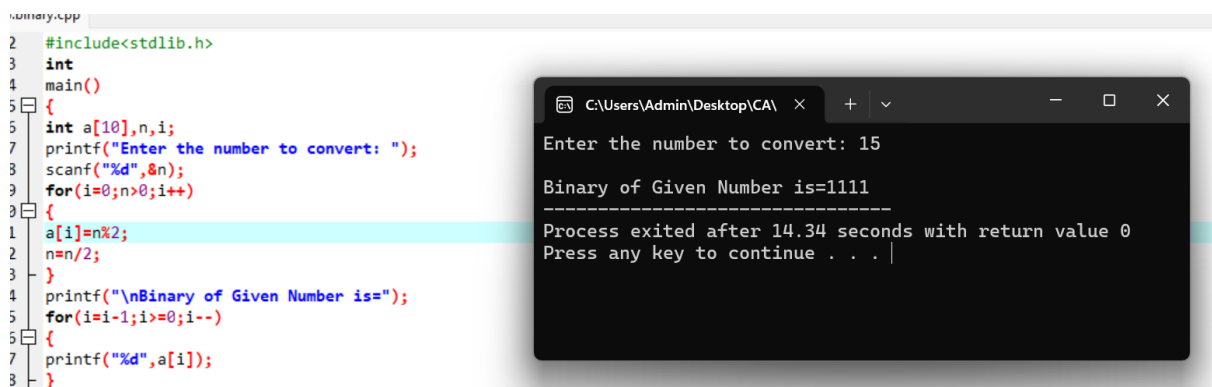
AIM: To write a C program to implement decimal to binary conversion.

ALGORITHM:

- 1) Check if your number is odd or even.
- 2) If it's even, write 0 (proceeding backwards, adding binary digits to the left of the result).
- 3) Otherwise, if it's odd, write 1 (in the same way).
- 4) Divide your number by 2 (dropping any fraction) and go back to step 1. Repeat until your original number is 0.

PROGRAM:

```
#include<stdio.h>
#include<stdlib.h>
int
main()
{
int a[10],n,i;
printf("Enter the number to convert: ");
scanf("%d",&n);
for(i=0;n>0;i++)
{
a[i]=n%2;
n=n/2;
}
printf("\nBinary of Given Number is=");
for(i=i-1;i>=0;i--)
{
printf("%d",a[i]);
}
return 0;
}
```



```
1 #include<stdio.h>
2 #include<stdlib.h>
3 int
4 main()
5 {
6     int a[10],n,i;
7     printf("Enter the number to convert: ");
8     scanf("%d",&n);
9     for(i=0;n>0;i++)
10     {
11         a[i]=n%2;
12         n=n/2;
13     }
14     printf("\nBinary of Given Number is=");
15     for(i=i-1;i>=0;i--)
16     {
17         printf("%d",a[i]);
18     }
19 }
```

```
C:\Users\Admin\Desktop\CA\ x + v - □ x
Enter the number to convert: 15
Binary of Given Number is=1111
-----
Process exited after 14.34 seconds with return value 0
Press any key to continue . . . |
```

BINARY TO DECIMAL CONVERSION

EXP NO: 26

AIM: To write a C program to implement binary to decimal conversion.

ALGORITHM:

- 1) Start
- 2) Read the binary number from the user, say 'n'
- 3) Initialize the decimal number, d=0
- 4) Initialize i=0
- 5) Repeat while n != 0:
 - i. Extract the last digit by: remainder = n % 10

- ii. $n = n/10$
- iii. $d = d + (\text{remainder} * 2^{\text{sup}i\text{</sup>})$
- iv. Increment i by 1
- 6) Display the decimal number, d
- 7) Stop

PROGRAM:

```
#include <stdio.h>

#include <math.h>

int main() {

    int binary, decimal = 0, base = 1, remainder;

    // Input binary number from user

    printf("Enter a binary number: ");

    scanf("%d", &binary);

    int temp = binary; // Store the binary number for later use

    while (binary > 0) {

        remainder = binary % 10; // Get the last digit of the binary number

        decimal = decimal + remainder * base;

        binary = binary / 10;

        base = base * 2; // Increase the base (2^n)

    }

    // Output the result

    printf("The decimal equivalent of binary %d is: %d\n", temp, decimal);

    return 0;

}
```

Output:

```
Enter a binary number: 1111
The decimal equivalent of binary 1111 is: 15

-----
Process exited after 6.643 seconds with return value 0
Press any key to continue . . . |
```

HEXADECIMAL TO DECIMAL CONVERSION

EXP NO: 27

AIM:To write a C program to implement hexadecimal to decimal conversion.

ALGORITHM:

- 1) Start from the right-most digit. Its weight (or coefficient) is 1.
- 2) Multiply the weight of the position by its digit. Add the product to the result.
(0=0, 1=1, 2=2, ... 9=9, A=10, B=11, C=12, D=13, E=14, F=15)
- 3) Move one digit to the left. Its weight is 16 times the previous weight.
- 4) Repeat 2 and 3 until you go through all hexadecimal digits.

PROGRAM:

```
#include <stdio.h>

#include <string.h>

#include <math.h>

int hexToDecimal(char hex[]) {
    int length = strlen(hex);

    int base = 1; // Base for hexadecimal (16^0)

    int decimal = 0;

    // Convert hex to decimal
    for (int i = length - 1; i >= 0; i--) {
        if (hex[i] >= '0' && hex[i] <= '9') {
            decimal += (hex[i] - 48) * base;
            base = base * 16;
        }
        else if (hex[i] >= 'A' && hex[i] <= 'F') {
            decimal += (hex[i] - 55) * base;
            base = base * 16;
        }
        else if (hex[i] >= 'a' && hex[i] <= 'f') {
            decimal += (hex[i] - 87) * base;
            base = base * 16;
        }
    }

    return decimal;
}

int main() {
    char hex[20];
```

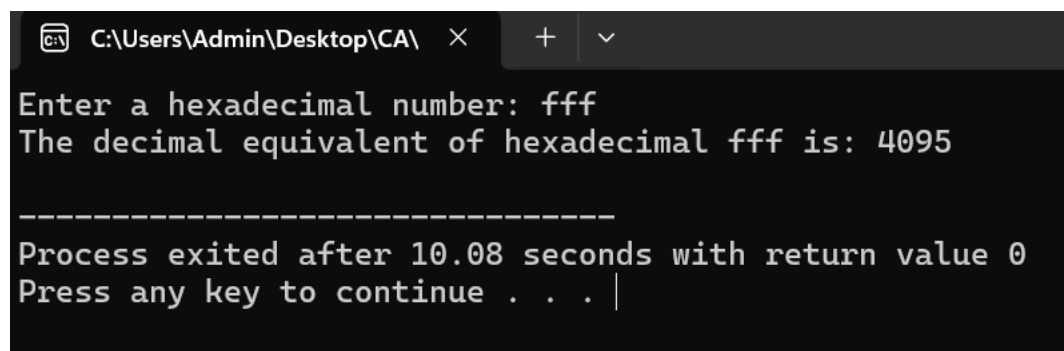
```

// Input hexadecimal number from user
printf("Enter a hexadecimal number: ");
scanf("%s", hex);

// Convert and display the decimal equivalent
int decimal = hexToDecimal(hex);
printf("The decimal equivalent of hexadecimal %s is: %d\n", hex, decimal);
return 0;
}

```

Output:



```

C:\Users\Admin\Desktop\CA\  ×  +  ▾
Enter a hexadecimal number: fff
The decimal equivalent of hexadecimal fff is: 4095

-----
Process exited after 10.08 seconds with return value 0
Press any key to continue . . . |

```

Exp No:28. Decimal to Hexadecimal

Program:

```

#include <stdio.h>

void toHexadecimal(int decimal) {
    char hex[32]; // Array to store hexadecimal number
    int i = 0;

    // Convert decimal to hexadecimal
    while (decimal != 0) {
        int remainder = decimal % 16;
        if (remainder < 10) {
            hex[i] = remainder + 48; // Convert remainder to character (0-9)
        } else {
            hex[i] = remainder + 55; // Convert remainder to character (A-F)
        }
        decimal = decimal / 16;
        i++;
    }
}

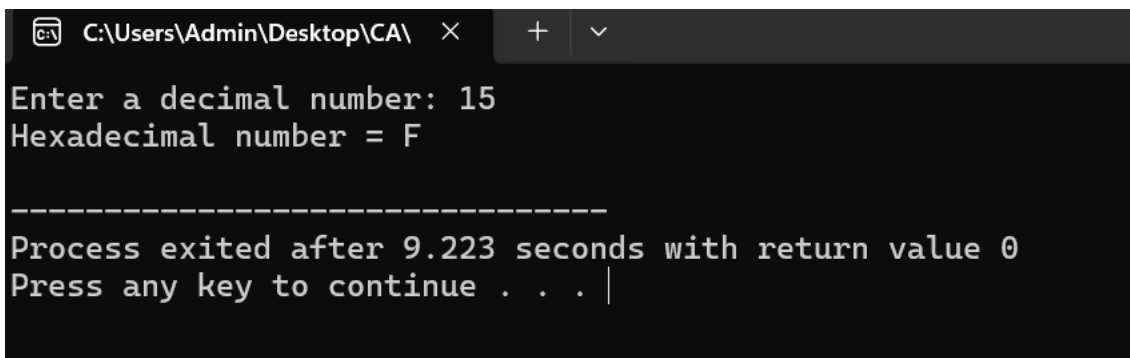
```

```

    }
    //Print hexadecimal number in reverse order
    printf("Hexadecimal number = ");
    for (int j = i - 1; j >= 0; j--)
        printf("%c", hex[j]);
    printf("\n");
}

int main() {
    int decimal;
    // Input decimal number from user
    printf("Enter a decimal number: ");
    scanf("%d", &decimal);
    // Convert and display the hexadecimal equivalent
    toHexadecimal(decimal);
    return 0;
}

```



```

C:\Users\Admin\Desktop\CA\ >
Enter a decimal number: 15
Hexadecimal number = F

-----
Process exited after 9.223 seconds with return value 0
Press any key to continue . . . |

```

EXP NO: 29

AIM:To write a C program to implement decimal to octal conversion.

ALGORITHM:

- 1) Store the remainder when the number is divided by 8 in an array.
- 2) Divide the number by 8 now
- 3) Repeat the above two steps until the number is not equal to 0.
- 4) Print the array in reverse order now.

PROGRAM:

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

```
void toOctal(int decimal) {
```

```
    int octal[32]; // Array to store octal number
```

```

int i = 0;

// Convert decimal to octal
while (decimal != 0) {
    octal[i] = decimal % 8;
    decimal = decimal / 8;
    i++;
}

// Print octal number in reverse order
printf("Octal number = ");
for (int j = i - 1; j >= 0; j--)
    printf("%d", octal[j]);
printf("\n");
}

int main() {
    int decimal;

    // Input decimal number from user
    printf("Enter a decimal number: ");
    scanf("%d", &decimal);

    // Convert and display the octal equivalent
    toOctal(decimal);

    return 0;
}

```

Output:

```
Enter a decimal number: 56
Octal number = 70
```

```
-----
Process exited after 6.135 seconds with return value 0
Press any key to continue . . . |
```

EXP NO 30 Convert Octal to Decimal

Program:

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

```
#include <math.h>
```

```
int octalToDecimal(int octal) {
    int decimal = 0, base = 1, remainder;

    // Convert octal to decimal
    while (octal > 0) {
        remainder = octal % 10;
        decimal += remainder * base;
        base *= 8;
        octal /= 10;
    }

    return decimal;
}

int main() {
    int octal;

    // Input octal number from user
    printf("Enter an octal number: ");
    scanf("%d", &octal);

    // Convert and display the decimal equivalent
```

```

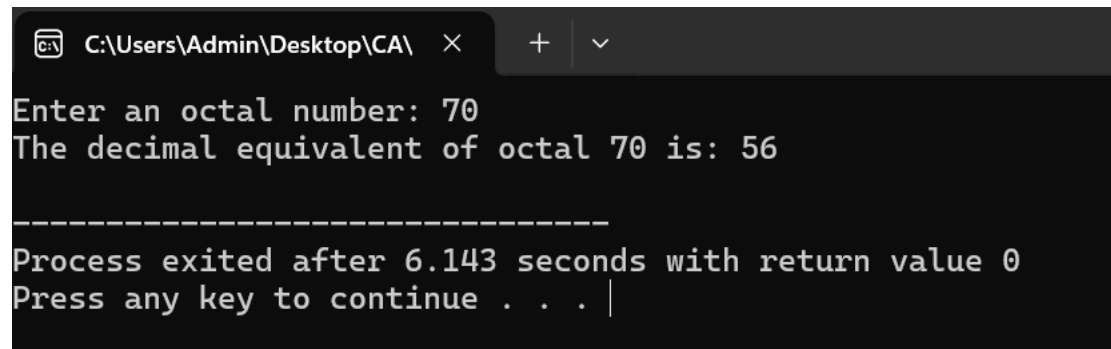
    int decimal = octalToDecimal(octal);

    printf("The decimal equivalent of octal %d is: %d\n", octal, decimal);

    return 0;
}

```

Output:



```

C:\Users\Admin\Desktop\CA\  ×  +  ∨
Enter an octal number: 70
The decimal equivalent of octal 70 is: 56
-----
Process exited after 6.143 seconds with return value 0
Press any key to continue . . . |

```

Exp. No: 31. Octal to Binary

Program:

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

```
// Function to convert octal to decimal
```

```

int octalToDecimal(int octal) {
    int decimal = 0, base = 1, remainder;

    while (octal > 0) {
        remainder = octal % 10;
        decimal += remainder * base;
        base *= 8;
        octal /= 10;
    }

    return decimal;
}

```

```
// Function to convert decimal to binary
```

```
void decimalToBinary(int decimal) {
```



```

int binary[32], i = 0;

while (decimal > 0) {
    binary[i] = decimal % 2;
    decimal /= 2;
    i++;
}

printf("Binary number = ");
for (int j = i - 1; j >= 0; j--)
    printf("%d", binary[j]);
printf("\n");
}

int main() {
    int octal;

    // Input octal number from user
    printf("Enter an octal number: ");
    scanf("%d", &octal);

    // Convert octal to decimal
    int decimal = octalToDecimal(octal);

    // Convert decimal to binary and print the result
    decimalToBinary(decimal);

    return 0;
}

```

Output:

```
C:\Users\Admin\Desktop\CA\ x + v
Enter an octal number: 70
Binary number = 111000

-----
Process exited after 5.86 seconds with return value 0
Press any key to continue . . . |
```

CPU PERFORMANCE

EXP NO: 32

AIM: To write a C program to implement CPU performance measures.

ALGORITHM:

Step 1: start

Step 2: Declare the necessary variables: cr

(clock rate), p (number of processors), p1 (a copy of the number of processors), i (loop variable), and cpu (array to store CPU times).

Step 3: Initialize the cpu array elements to 0.

Step 4: Prompt the user to enter the number of processors (p).

Step 5: Store the value of p in p1.

Step 6: Start a loop from 0 to p-1:

- Prompt the user to enter the cycles per instruction (cpi) for the current processor.
- Prompt the user to enter the clock rate (cr) in GHz for the current processor.
- Calculate the CPU time (ct) using the formula: $ct = 1000 * cpi / cr$.
- Display the CPU time for the current processor.
- Store the CPU time in the cpu array at index i.

Step 7: Set max as the first element of the cpu array.

Step 8: Start a loop from 0 to p1-1:

- If the CPU time at index i is less than or equal to max, update max to the current CPU time.

Step 9: Display the processor with the lowest execution time (max).

Step 10: Exit the program.

PROGRAM:

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

```
int
```

```
main()
```

```
{
```

```
    float cr;
```

```
    int p,p1,i;
```

```
    float cpu[5];
```

```
    float cpi,ct,max;
```

```
    int n=1000;
```

```
    for(i=0;i<=4;i++)
```

```

{
    cpu[5]=0;
}
printf("\n Enter the number of processors:");
scanf("%d",&p);
p1=p;
for(i=0;i<p;i++)
{
    printf("\n Enter the Cycles perInstrcution of processor:");

    scanf("%f",&cpi);
    printf("\n Enter the clockrate inGHz:");
    scanf("%f",&cr);

    ct=1000*cpi/cr;

    printf("The CPU time is: %f",ct);
    cpu[i]=ct;
}

max=cpu[0];

for(i=0;i<p1;i++)

{
    if(cpu[i]<=max)

        max=cpu[i];
}
printf("\nThe processor has lowest Execution time is: %f ", max);
return 0;
}
output:

```

```

Enter the number of processors:4

Enter the Cycles perInstrction of processor:2

Enter the clockrate inGHz:4
The CPU time is: 500.000000
Enter the Cycles perInstrction of processor:5

Enter the clockrate inGHz:4
The CPU time is: 1250.000000
Enter the Cycles perInstrction of processor:5

Enter the clockrate inGHz:7
The CPU time is: 714.285706
Enter the Cycles perInstrction of processor:2

Enter the clockrate inGHz:4
The CPU time is: 500.000000
The processor has lowest Execution time is: 500.000000
-----

```

Exp No:33. Integer Restroring Division

Program:

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

```
#include <math.h>
```

```
// Function to perform integer restoration division
```

```
void integerRestorationDivision(int dividend, int divisor) {
```

```
    int quotient = 0, remainder = 0;
```

```
    int bitSize = (int)log2(dividend) + 1;
```

```
    // Process each bit
```

```
    for (int i = bitSize - 1; i >= 0; i--) {
```

```
        // Bring down the next bit of the dividend
```

```
        remainder = (remainder << 1) | ((dividend >> i) & 1);
```

```
        // Perform subtraction if possible
```

```
        if (remainder >= divisor) {
```

```
            remainder -= divisor;
```

```

        quotient |= (1 << i); // Set the ith bit of the quotient to 1
    }
}

// Output the result
printf("Quotient = %d\n", quotient);
printf("Remainder = %d\n", remainder);
}

int main() {
    int dividend, divisor;

    // Input dividend and divisor from user
    printf("Enter the dividend: ");
    scanf("%d", &dividend);
    printf("Enter the divisor: ");
    scanf("%d", &divisor);

    // Check if divisor is not zero
    if (divisor == 0) {
        printf("Error: Division by zero is not allowed.\n");
        return 1;
    }

    // Perform integer restoration division
    integerRestorationDivision(dividend, divisor);

    return 0;
}

```

Output:

```
Enter the dividend: 8
Enter the divisor: 2
Quotient = 4
Remainder = 0

-----
Process exited after 7.51 seconds with return value 0
```

Exp No: 34. Booth Algorithm

Program:

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

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

```
#define BITS 8 // Number of bits in the binary representation
```

```
// Function to perform Booth's Algorithm
```

```
void boothAlgorithm(int M, int Q) {
```

```
    int A = 0;    // Accumulator
```

```
    int Q_1 = 0;  // Q-1
```

```
    int temp;
```

```
    // Create binary representations of M and Q
```

```
    int M_bin[BITS] = {0}, Q_bin[BITS] = {0};
```

```
    int result[2 * BITS] = {0};
```

```
    // Convert M and Q to binary
```

```
    for (int i = BITS - 1; i >= 0; i--) {
```

```
        M_bin[i] = M % 2;
```

```
        M /= 2;
```

```
        Q_bin[i] = Q % 2;
```

```
        Q /= 2;
```

```
    }
```

```
    printf("Initial values:\n");
```

```
    printf("A = ");
```

```

for (int i = 0; i < BITS; i++) printf("%d", (A >> (BITS - 1 - i)) & 1);
printf("\nQ = ");
for (int i = 0; i < BITS; i++) printf("%d", Q_bin[i]);
printf("\nQ-1 = %d\n", Q_1);

// Perform Booth's Algorithm
for (int i = 0; i < BITS; i++) {
    // Check Q[0] and Q-1
    if (Q_bin[BITS - 1] == 0 && Q_1 == 1) {
        // A = A + M
        temp = A + M;
        if (temp >= (1 << BITS)) temp -= (1 << BITS); // Handle overflow
        A = temp;
    } else if (Q_bin[BITS - 1] == 1 && Q_1 == 0) {
        // A = A - M
        temp = A - M;
        if (temp < 0) temp += (1 << BITS); // Handle underflow
        A = temp;
    }

    // Arithmetic right shift A, Q, and Q-1
    Q_1 = Q_bin[BITS - 1];
    for (int j = BITS - 1; j > 0; j--) {
        Q_bin[j] = Q_bin[j - 1];
    }
    Q_bin[0] = (A & 1); // Update Q[0] from A
    A >>= 1;           // Arithmetic shift of A
}

// Display final result
printf("Final result:\n");
printf("A = ");

```

```

    for (int i = 0; i < BITS; i++) printf("%d", (A >> (BITS - 1 - i)) & 1);
    printf("\nQ = ");
    for (int i = 0; i < BITS; i++) printf("%d", Q_bin[i]);
    printf("\nQ-1 = %d\n", Q_1);
}

```

```

int main() {
    int M, Q
    // Input multiplicand and multiplier from user
    printf("Enter the multiplicand (M): ");
    scanf("%d", &M);
    printf("Enter the multiplier (Q): ");
    scanf("%d", &Q);
    // Perform Booth's Algorithm
    boothAlgorithm(M, Q);
    return 0;
}

```

Output:

```

Enter the multiplicand (M): 16
Enter the multiplier (Q): -2
Initial values:
A = 00000000
Q = 000000-10
Q-1 = 0
Final result:
A = 00000000
Q = 00000000
Q-1 = 0

-----
Process exited after 14.72 seconds with return value 0
Press any key to continue . . . |

```

Exp.No.35. Single Precision Representation

Program:

#include <stdio.h>


```

// Function to display the binary representation of a float
void displayBinary(float num) {
    // Create a union to store the float and access its bits as an integer
    union {
        float f;
        unsigned int i;
    } u;

    u.f = num;

    // Print the sign bit
    printf("Sign: %d\n", (u.i >> 31) & 1);

    // Print the exponent bits
    printf("Exponent: ");
    for (int i = 30; i >= 23; i--) {
        printf("%d", (u.i >> i) & 1);
    }
    printf("\n");

    // Print the mantissa bits
    printf("Mantissa: ");
    for (int i = 22; i >= 0; i--) {
        printf("%d", (u.i >> i) & 1);
    }
    printf("\n");
}

int main() {
    float num;

    // Input the number from the user

```

```

printf("Enter a floating-point number: ");
scanf("%f", &num);

// Display the binary representation
displayBinary(num);

return 0;
}

```

Output:

```

Enter a floating-point number: 122.7
Sign: 0
Exponent: 10000101
Mantissa: 11101010110011001100110
-----
Process exited after 7.829 seconds with return value 0
Press any key to continue . . . |

```

Exp No. 36. Four stage Pipeline

Program:

```

#include <stdio.h>

// Function prototypes for each pipeline stage
void fetch(int instruction);
void decode(int instruction);
void execute(int instruction);
void write_back(int instruction);

// Simulate the instruction pipeline
void pipeline(int instructions[], int n) {
    for (int cycle = 0; cycle < n + 3; cycle++) {
        printf("Cycle %d:\n", cycle + 1);

        if (cycle < n) {
            fetch(instructions[cycle]);

```

```

    }
    if (cycle > 0 && cycle < n + 1) {
        decode(instructions[cycle - 1]);
    }
    if (cycle > 1 && cycle < n + 2) {
        execute(instructions[cycle - 2]);
    }
    if (cycle > 2) {
        write_back(instructions[cycle - 3]);
    }

    printf("\n");
}
}

void fetch(int instruction) {
    printf(" Fetching instruction %d\n", instruction);
}

void decode(int instruction) {
    printf(" Decoding instruction %d\n", instruction);
}

void execute(int instruction) {
    printf(" Executing instruction %d\n", instruction);
}

void write_back(int instruction) {
    printf(" Writing back result of instruction %d\n", instruction);
}

int main() {

```

```

int instructions[] = {1, 2, 3, 4};
int n = sizeof(instructions) / sizeof(instructions[0]);

// Run the pipeline with 4 instructions
pipeline(instructions, n);

return 0;
}

```

Output:

```

Cycle 1:
  Fetching instruction 1

Cycle 2:
  Fetching instruction 2
  Decoding instruction 1

Cycle 3:
  Fetching instruction 3
  Decoding instruction 2
  Executing instruction 1

Cycle 4:
  Fetching instruction 4
  Decoding instruction 3
  Executing instruction 2
  Writing back result of instruction 1

Cycle 5:
  Decoding instruction 4
  Executing instruction 3
  Writing back result of instruction 2

Cycle 6:
  Executing instruction 4
  Writing back result of instruction 3

Cycle 7:
  Writing back result of instruction 4

```

Exp. No:37. Two Stage Pipeline

Program:

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

```
// Function prototypes for the two stages
```

```
void fetch(int instruction);
```

```
void execute(int instruction);
```

```
// Simulate the two-stage instruction pipeline
```

```
void pipeline(int instructions[], int n) {
```

```
    for (int cycle = 0; cycle < n + 1; cycle++) {
```

```
        printf("Cycle %d:\n", cycle + 1);
```

```
        if (cycle < n) {
```

```
            fetch(instructions[cycle]);
```

```
        }
```

```
        if (cycle > 0 && cycle < n + 1) {
```

```
            execute(instructions[cycle - 1]);
```

```
        }
```

```
        printf("\n");
```

```
    }
```

```
}
```

```
// Fetch stage: Simulate fetching an instruction from memory
```

```
void fetch(int instruction) {
```

```
    printf(" Fetching instruction %d\n", instruction);
```

```
}
```

```
// Execute stage: Simulate executing the fetched instruction
```

```
void execute(int instruction) {
```

```
    printf(" Executing instruction %d\n", instruction);
```

```
}
```

```

int main() {
    int instructions[] = {1, 2, 3, 4}; // Simulate 4 instructions
    int n = sizeof(instructions) / sizeof(instructions[0]);

    // Run the two-stage pipeline with 4 instructions
    pipeline(instructions, n);

    return 0;
}

```

Output:

```

Cycle 1:
    Fetching instruction 1

Cycle 2:
    Fetching instruction 2
    Executing instruction 1

Cycle 3:
    Fetching instruction 3
    Executing instruction 2

Cycle 4:
    Fetching instruction 4
    Executing instruction 3

Cycle 5:
    Executing instruction 4

```

Exo.No:38. Three stage AND operation

Program:

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

```
// Function prototypes for each stage
```

```
void fetch(int instruction);
```

```
void decode(int instruction);
```

```
void execute(int operand1, int operand2);
```

// Simulate the three-stage instruction pipeline

```
void pipeline(int instructions[][2], int n) {  
    for (int cycle = 0; cycle < n + 2; cycle++) {  
        printf("Cycle %d:\n", cycle + 1);  
  
        if (cycle < n) {  
            fetch(cycle);  
        }  
        if (cycle > 0 && cycle < n + 1) {  
            decode(cycle - 1);  
        }  
        if (cycle > 1 && cycle < n + 2) {  
            execute(instructions[cycle - 2][0], instructions[cycle - 2][1]);  
        }  
  
        printf("\n");  
    }  
}
```

// Fetch stage: Simulate fetching an instruction (in this case, the instruction is just the index)

```
void fetch(int instruction) {  
    printf(" Fetching instruction %d (AND operation)\n", instruction + 1);  
}
```

// Decode stage: Simulate decoding the instruction (identify it as AND)

```
void decode(int instruction) {  
    printf(" Decoding instruction %d (performing AND operation)\n", instruction + 1);  
}
```

// Execute stage: Perform the AND operation on two operands

```
void execute(int operand1, int operand2) {
```

```
    int result = operand1 & operand2;

    printf(" Executing AND operation: %d & %d = %d\n", operand1, operand2,
result);
}
```

```
int main() {
    // Array of operand pairs to be used in AND operation
    int instructions[][2] = {
        {5, 3}, // 5 (101) & 3 (011) = 1 (001)
        {12, 9}, // 12 (1100) & 9 (1001) = 8 (1000)
        {15, 7} // 15 (1111) & 7 (0111) = 7 (0111)
    };

    int n = sizeof(instructions) / sizeof(instructions[0]);

    // Run the three-stage pipeline with AND operations
    pipeline(instructions, n);

    return 0;
}
```

Output:


```

Cycle 1:
    Fetching instruction 1 (AND operation)

Cycle 2:
    Fetching instruction 2 (AND operation)
    Decoding instruction 1 (performing AND operation)

Cycle 3:
    Fetching instruction 3 (AND operation)
    Decoding instruction 2 (performing AND operation)
    Executing AND operation: 5 & 3 = 1

Cycle 4:
    Decoding instruction 3 (performing AND operation)
    Executing AND operation: 12 & 9 = 8

Cycle 5:
    Executing AND operation: 15 & 7 = 7

```

Exp No:39. Four stage AND operation

Program:

```

#include <stdio.h>

// Function prototypes for each stage
void fetch(int instruction);
void decode(int instruction);
void execute(int operand1, int operand2);
void write_back(int result);

// Simulate the four-stage instruction pipeline
void pipeline(int instructions[][2], int n) {
    int results[n]; // Store the results of the AND operations
    for (int cycle = 0; cycle < n + 3; cycle++) {
        printf("Cycle %d:\n", cycle + 1);

        if (cycle < n) {
            fetch(cycle);

```

```

    }
    if (cycle > 0 && cycle < n + 1) {
        decode(cycle - 1);
    }
    if (cycle > 1 && cycle < n + 2) {
        execute(instructions[cycle - 2][0], instructions[cycle - 2][1]);
    }
    if (cycle > 2 && cycle < n + 3) {
        int result = instructions[cycle - 3][0] & instructions[cycle - 3][1];
        results[cycle - 3] = result;
        write_back(result);
    }

    printf("\n");
}

// Print final results
printf("Final results:\n");
for (int i = 0; i < n; i++) {
    printf("Result of AND operation %d: %d\n", i + 1, results[i]);
}
}

// Fetch stage: Simulate fetching an instruction
void fetch(int instruction) {
    printf(" Fetching instruction %d (AND operation)\n", instruction + 1);
}

// Decode stage: Simulate decoding the instruction
void decode(int instruction) {
    printf(" Decoding instruction %d (performing AND operation)\n", instruction + 1);
}

```

```
// Execute stage: Perform the AND operation on two operands  
void execute(int operand1, int operand2) {  
    printf(" Executing AND operation: %d & %d\n", operand1, operand2);  
}
```

```
// Write Back stage: Store the result of the AND operation  
void write_back(int result) {  
    printf(" Writing back result: %d\n", result);  
}
```

```
int main() {  
    // Array of operand pairs for AND operation  
    int instructions[][2] = {  
        {5, 3}, // 5 (101) & 3 (011) = 1 (001)  
        {12, 9}, // 12 (1100) & 9 (1001) = 8 (1000)  
        {15, 7} // 15 (1111) & 7 (0111) = 7 (0111)  
    };  
  
    int n = sizeof(instructions) / sizeof(instructions[0]);  
  
    // Run the four-stage pipeline with AND operations  
    pipeline(instructions, n);  
  
    return 0;  
}
```

Output:

Cycle 1:

Fetching instruction 1 (AND operation)

Cycle 2:

Fetching instruction 2 (AND operation)

Decoding instruction 1 (performing AND operation)

Cycle 3:

Fetching instruction 3 (AND operation)

Decoding instruction 2 (performing AND operation)

Executing AND operation: 5 & 3

Cycle 4:

Decoding instruction 3 (performing AND operation)

Executing AND operation: 12 & 9

Writing back result: 1

Cycle 5:

Executing AND operation: 15 & 7

Writing back result: 8

Cycle 6:

Writing back result: 7

Final results:

Result of AND operation 1: 1

Result of AND operation 2: 8

Result of AND operation 3: 7