AIM: Conversion of decimal number into Base-n number system (1≤ n ≤9).

Code:

```
#include<iostream>
#include<vector>
using namespace std;
// Name: Vaishnavi Vilas Thamke
// DIV: A
// Roll No.: 155
void decTObinary(double num,int base) {
    long long int c;
    string int_binary = "";
    long long int temp = (long long int) num;
    if (temp ==0) {
        int_binary = '0';
    } else {
        while(temp>0) {
            int rem = temp%base;
            int_binary = to_string(rem)+int_binary;
            temp /= base;}
    }
    double frac = num - int(num);
    string frac_binary ="";
    int count =0;
    while (frac!= 0 && count<10){
        frac *= base;
        int bit = int(frac);
        frac_binary += to_string(bit);
        frac -= bit;
        count ++;
    }
    cout<<"Decimal "<<int(num)<<" converted into Base-"<<base<<" system: "<<int_binary<<endl;
    cout<<endl;
    cout<<"Fractional decimal "<<num-int(num)<<" converted into Base-"<<base<<" system: 
0."<<frac_binary<<endl;
```

```
    cout<<endl;
    cout<<"Hence, Base-"<<base<<" equivalent of input decimal: "<<int_binary<<'.'<<frac_binary<<endl;
    cout<<endl;
    }
int main () {
    double num;
    cout<<"Enter the Input decimal number:";
    cin>>num;
    cout<<endl;
    int base;
    cout<<"Enter the base of the destination number system:";
    cin>>base;
    cout<<endl;
    decTObinary(num,base);
    return 0;
}
```

Output:

```
Enter the Input decimal number:45.23

Enter the base of the destination number system:3

Decimal 45 converted into Base-3 system: 1200

Fractional decimal 0.23 converted into Base-3 system: 0.0200122000

Hence, Base-3 equivalent of input decimal: 1200.0200122000
```

Aim: Study and implementation of the Booth's Multiplication Algorithm.

Code:

```
//NAME: Vaishnavi Thamke
//Roll No.: 155
//Div: A
#include<iostream>
#include<bitset
#include<cmath>
using namespace std;
string ToBinary (int num, int bits) {
    return bitset<32>(num).to_string().substr(32-bits, bits );
}
int main () {
    int M, Q;
    cout<<"Enter the multiplier(M):";
    cin>>M;
    cout<<"Enter the multiplicand(Q):";
    cin>>Q;

    int bits = max((int)ceil(log2(abs(M)+1)),(int)ceil(log2(abs(Q+1))) + 1)+1;
    cout <<"Binary representation of Multiplicand(Q):"<<ToBinary(Q, bits) <<endl;
    cout <<"Binary representation of Multiplier(M):"<<ToBinary(M,bits)<<endl;

    int A = 0;
    int Qn1 =0;
    int count = bits;
    while(count>0){
        int Q0= Q&1;
```

```
        if(Q0==0 && Qn1 ==1){
          A+= M;
        }else if (Q0==1 && Qn1==0){
          A-=M;
        }
        int combined = (A<<bits) | (Q& ((1<<bits)-1));
        Qn1 = Q & 1;
        combined >>= 1;
        A = combined>>bits;
        Q = combined & ((1<<bits)-1);
        count--;
    }
        int product = (A<<bits)|Q;
        cout<< "Result of multiplication in binary:" << ToBinary(product,bits*2)<<endl;
    return 0;
}
```

Output:

```
Enter the multiplier(M):42
Enter the multiplicand(Q):2
Binary representation of Multiplicand(Q):0000010
Binary representation of Multiplier(M):0101010
Result of multiplication in binary:00000001010100
```

Aim: Study and implementation of the Restoring Division Algorithm.

Code:

```
#include <iostream>

#include <string>

#include <cstdlib>

using namespace std;

string to_bin(unsigned long long x, int w){

    string s(w,'0');

    for(int i=w-1; i>=0; --i) s[w-1-i]= (x & (1ULL<<i)) ? '1':'0';

    return s;

}

int bitlen(unsigned long long x){ return x ? 64 - __builtin_clzll(x) : 1; }

unsigned long long mask_n(int n){ return (n>=64)? ~0ULL : ((1ULL<<n)-1ULL); }

int main(){

    long long dividend, divisor;

    cout<<"Enter the divisor (M): "; cin>>divisor;

    cout<<"Enter the dividend (Q): "; cin>>dividend;

    if(divisor==0) return cout<<"Error: Division by zero.\n",0;

    bool negQ = dividend<0, negM = divisor<0;

    unsigned long long Q = llabs(dividend), M = llabs(divisor);

    int n = max(bitlen(Q), bitlen(M));

    if(n<1) n=1; if(n>64) n=64;

    unsigned long long MASK = mask_n(n);

    cout<<"Binary representation of Dividend (Q): "<<to_bin(Q,n)<<"\n";

    cout<<"Binary representation of Divisor (M): "<<to_bin(M,n)<<"\n";

    long long A = 0;

    for (int i=0; i<n; ++i) {

        int q_msb = (Q>>(n-1)) & 1U;
```

```
        A = (A<<1) | q_msb;
        Q = (Q<<1) & MASK;
        A -= (long long)M;
        if(A<0){ A += (long long)M; Q &= ~1ULL; }
        else   { Q |=  1ULL; }
    }
    long long r = A;
    unsigned long long qmag = Q & MASK;

    if(negQ && r!=0){
        qmag += 1ULL;
        r = (long long)M - r;
    }
    long long quotient = (negQ ^ negM) ? -(long long)qmag : (long long)qmag;
    cout<<"Quotient in Binary: "<<to_bin((unsigned long long)llabs(quotient), n)<<"\n";
    cout<<"Remainder in Binary: "<<to_bin((unsigned long long)r, n)<<"\n";
    return 0;
}
```

Output:

```
Enter the divisor (M): 3
Enter the dividend (Q): 15
Binary representation of Dividend (Q): 1111
Binary representation of Divisor (M): 0011
Quotient in Binary: 0101
Remainder in Binary: 0000
```

Aim: Study and implementation of the Non-Restoring Algorithm.

Code:

```
#include <iostream>
using namespace std;
string to_bin(unsigned long long x, int width) {
  string s(width, '0');
  for (int i = width - 1; i >= 0; --i)
    s[width - 1 - i] = (x & (1ULL << i)) ? '1' : '0';
  return s;
}
int bitlen(unsigned long long x) {
  return x ? 64 - __builtin_clzll(x) : 1;
}
int main() {
  long long dividend, divisor;
  cout << "Enter the divisor  (M): "; cin >> divisor;
  cout << "Enter the dividend (Q): "; cin >> dividend;

  if (divisor == 0) return cout << "Error: Division by zero.\n", 0;

  bool negQ = dividend < 0, negM = divisor < 0;
  unsigned long long absQ = llabs(dividend), absM = llabs(divisor);
  int n = max(bitlen(absQ), bitlen(absM));

  cout << "Binary representation of Dividend (Q): " << to_bin(absQ, n) << "\n";
  cout << "Binary representation of Divisor  (M): " << to_bin(absM, n) << "\n";

  long long A = 0;
```

```
    for (int i = 0; i < n; ++i) {
        int q_msb = (absQ >> (n - 1)) & 1U;
        A = (A << 1) | q_msb;
        absQ = (absQ << 1) & ((1ULL << n) - 1);
        if (A >= 0) A -= absM; else A += absM;
        absQ = (absQ & ~1ULL) | (A >= 0);
    }

    if (A < 0) A += absM;
    long long quotient = (negQ ^ negM) ? -(long long)(absQ & ((1ULL << n) - 1)) : (absQ &
((1ULL << n) - 1));

    cout << "Quotient in Binary: " << to_bin(llabs(quotient), n) << "\n";
    cout << "Remainder in Binary: " << to_bin((unsigned long long)A, n) << "\n";
    return 0;
}
```

<u>Output:</u>

```
Enter the divisor  (M): 6
Enter the dividend (Q): 18
Binary representation of Dividend (Q): 10010
Binary representation of Divisor  (M): 00110
Quotient in Binary: 00011
Remainder in Binary: 00000
```

Aim : Study and implementation of the IEEE 754 Floating point Representation standard.

Code:

```
// Name :Vaishnavi Thamke
//Roll No.: SEA55

#include <iostream>
#include <bitset>
#include <cmath>
using namespace std;

int main() {
    float num;
    cout << "Enter the Decimal Number = ";
    cin >> num;

    int sign = (num < 0);
    num = fabs(num);

    // Integer and fractional parts
    int intPart = (int)num;
    float fracPart = num - intPart;

    string intBin = "", fracBin = "";
    while (intPart) { intBin = char('0' + intPart % 2) + intBin; intPart /= 2; }
    for (int i = 0; i < 10 && fracPart; i++) {
        fracPart *= 2;
        fracBin += (fracPart >= 1 ? '1' : '0');
        if (fracPart >= 1) fracPart -= 1;
    }

    int exponent = intBin.size() - 1;
    string mantissa = intBin.substr(1) + fracBin;
    int biasedExp = exponent + 127;

    string mantissa23 = mantissa.substr(0, 23);
    mantissa23.append(23 - mantissa23.size(), '0');

    string ieee = to_string(sign) + bitset<8>(biasedExp).to_string() + mantissa23;

    cout << "Given number in Binary = " << intBin << "." << fracBin << endl;
    cout << "Given number in Scientific Notation = 1." << mantissa << " *2^" << exponent << endl;
    cout << "Real Exponent = " << exponent << endl;
    cout << "Select the destination floating point format = 32 bit" << endl;
    cout << "Biased Exponent = " << exponent << " + 127 = " << biasedExp
```

```
        << " = " << bitset<8>(biasedExp) << endl;
    cout << "Actual fractional part = " << mantissa.substr(0,5) << endl;
    cout << "Mantissa of 23 bits = " << mantissa23 << endl;
    cout << "Sign bit = " << sign << endl;
    cout << "32 bit representation of the given number = " << ieee << endl;
    cout << "Hex representation = " << hex << uppercase << bitset<32>(ieee).to_ulong() << endl;

    return 0;
}
```

Output:

```
Enter the Decimal Number = 15.5
Given number in Binary = 1111.1
Given number in Scientific Notation = 1.1111 *2^3
Real Exponent = 3
Select the destination floating point format = 32 bit
Biased Exponent = 3 + 127 = 130 = 10000010
Actual fractional part = 1111
Mantissa of 23 bits = 11110000000000000000000
Sign bit = 0
32 bit representation of the given number = 01000001011110000000000000000000
Hex representation = 41780000
```

AIM: Analysis of multilevel memory hierarchy.

Code:

```
#include<iostream>
#include<iomanip>

int main() {
    int n;
    cout << "Enter number of memory levels (2 or 3): ";
    cin >> n;

    double C[4], S[4], H[4], ta[4];

    for(int i = 1; i <= n; i++) {
        cout << "Enter Cost/bit for level " << i << ": ";
        cin >> C[i];
        cout << "Enter Size (bits) for level " << i << ": ";
        cin >> S[i];
        cout << "Enter Hit rate for level " << i << ": ";
        cin >> H[i];
        cout << "Enter Access time for level " << i << " (microseconds): ";
        cin >> ta[i];
    }

    double num = 0, den = 0;
    for(int i = 1; i <= n; i++) {
        num += C[i] * S[i];
        den += S[i];
    }
    double Cav = num / den;

    double tav = 0;
    if(n == 2) {
        tav = H[1]*ta[1] + (1-H[1])*ta[2];
    } else if(n == 3) {
        tav = H[1]*ta[1] + (1-H[1])*H[2]*ta[2] + (1-H[1])*(1-H[2])*ta[3];
    }

    cout << fixed << setprecision(4);
    cout << "\nAverage cost per bit (INR) = " << Cav;
    cout << "\nAverage access time = " << tav << " microseconds\n";

    return 0;
}
```

Output:

```
Enter number of memory levels (2 or 3): 2
Enter Cost/bit for level 1: 0.01
Enter Size (bits) for level 1: 1000
Enter Hit rate for level 1: 0.52
Enter Access time for level 1 (microseconds): 200
Enter Cost/bit for level 2: 0.0001
Enter Size (bits) for level 2: 100
Enter Hit rate for level 2: 0.78
Enter Access time for level 2 (microseconds): 1

Average cost per bit (INR) = 0.0091
Average access time = 104.4800 microseconds
```

AIM : Study of various cache mapping policies.

Code:

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

int main() {
    int cacheSizeKB, mainMemSizeMB, lineSize;
    cout << "Size of Cache memory (in KB) = ";
    cin >> cacheSizeKB;
    cout << "Size of Main memory (in MB) = ";
    cin >> mainMemSizeMB;
    cout << "Size of each cache line (in Bytes) = ";
    cin >> lineSize;

    int cacheBytes = cacheSizeKB * 1024;
    int mainMemBytes = mainMemSizeMB * 1024 * 1024;

    int addrBits = log2(mainMemBytes);
    int cacheLines = cacheBytes / lineSize;
    int blocks = mainMemBytes / lineSize;

    cout << "\nMain memory address = " << addrBits << " bits";

    cout << "\n\nSelect cache mapping policy:";
    cout << "\n1. Direct Mapping";
    cout << "\n2. 2-Way Set Associative Mapping";
    cout << "\n3. Fully Associative Mapping\n";
    int choice;
    cin >> choice;

    if(choice == 1) {
        // ---- Direct Mapping ----
        cout << "\nNumber of cache banks = " << 1;  // each line is a bank
        cout << "\nHence, Size of cache = " << cacheSizeKB << " KB";
        cout << "\nCache lines per cache bank= " << cacheLines;
        cout << "\nNumber of main memory blocks = " << blocks<<endl;

        int byteBits = log2(lineSize);
        int lineBits = log2(cacheLines);
        int tagBits = addrBits - (byteBits + lineBits);
            cout<< "\nMain memory address of "<< addrBits << " bits is interpreted in 3 fields as
calculated below"<<endl;
        cout << "\n\nLSB " << byteBits << " bits for Byte selection";
        cout << "\nMiddle " << lineBits << " bits for Cache line selection";
        cout << "\nMSB " << tagBits << " bits for the Tags";

        int blockNum;
        cout << "\n\nInput any Main memory block number for cache mapping = ";
        cin >> blockNum;

        int cacheLine = blockNum % cacheLines;
```

```
    cout << "Block " << blockNum << " is mapped into cache line number = "
        << cacheLine<< endl;
}
else if(choice == 2) {
    // ---- 2-Way Set Associative ----
    int banks = cacheLines / 2;
      int k=2;
    cout << "\nNumber of banks = " << k;
    cout << "\nSize of cache = " << cacheSizeKB/2 << " KB";
    cout << "\nCache lines per bank= " << cacheSizeKB/2/cacheLines;
    cout << "\nNumber of main memory blocks = " << blocks;

    int byteBits = log2(lineSize);
    int bankBits = log2(banks);
    int tagBits = addrBits - (byteBits + bankBits);

    cout << "\n\nLSB " << byteBits << " bits for Byte selection";
    cout << "\nNext " << bankBits << " bits for Bank selection";
    cout << "\nMSB " << tagBits << " bits for the Tags";

    int blockNum;
    cout << "\n\nInput any Main memory block number for cache mapping = ";
    cin >> blockNum;

    int bankNum = blockNum % banks;
    cout << "Block " << blockNum
        << " can be placed in Bank number = "
        << bankNum << " (2 lines per bank)" << endl;
}
else if(choice == 3) {
    // ---- Fully Associative ----
    cout << "\nNumber of banks = 1";
    cout << "\nSize of cache = " << cacheSizeKB << " KB";
    cout << "\nCache lines = " << cacheLines;
    cout << "\nNumber of main memory blocks = " << blocks;

    int byteBits = log2(lineSize);
    int tagBits = addrBits - byteBits;

    cout << "\n\nLSB " << byteBits << " bits for Byte selection";
    cout << "\nRemaining " << tagBits << " bits for the Tags";
    cout << "\nIn fully associative mapping, any block can be placed in any cache line.\n";
}
else {
    cout << "Invalid choice!" << endl;
}

return 0;
```

Output:

```
Size of Cache memory (in KB) = 16
Size of Main memory (in MB) = 4
Size of each cache line (in Bytes) = 8

Main memory address = 22 bits

Select cache mapping policy:
1. Direct Mapping
2. 2-Way Set Associative Mapping
3. Fully Associative Mapping
1

Number of cache banks = 1
Hence, Size of cache = 16 KB
Cache lines per cache bank= 2048
Number of main memory blocks = 524288

Main memory address of 22 bits is interpreted in 3 fields as calculated below


LSB 3 bits for Byte selection
Middle 11 bits for Cache line selection
MSB 8 bits for the Tags

Input any Main memory block number for cache mapping = 2053
Block 2053 is mapped into cache line number = 5
(base) computer@computer-ThinkCentre:~/Desktop/SEA55$
```