Processor architecture

practical 3

Division/Batch: A/A3  
Branch: Computer Engineering

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **SAP ID** | **Name of Student** | **Date of Experiment** | **Date of Submission** | **Remarks** |
| 60004190054 | Jazib Dawre | 30/10/21 | 30/10/21 |  |

# Aim

To study and implement non-restoring division algorithm.

# Theory

A division algorithm is an algorithm which, given two integers N and D, computes their quotient and/or remainder, the result of Euclidean division. Division algorithms fall into two main categories: slow division and fast division. Slow division algorithms produce one digit of the final quotient per iteration. Examples of slow division include restoring, non-performing restoring, non-restoring, and SRT division. Fast division methods start with a close approximation to the final quotient and produce twice as many digits of the final quotient on each iteration. Newton–Raphson and Goldschmidt algorithms fall into this category.

Non-Restoring Division Algorithm is used to divide two unsigned integers. This algorithm is used in Computer Organization and Architecture The algorithm is more complex but has the advantage when implemented in hardware that there is only one decision and addition/subtraction per quotient bit; there is no restoring step after the subtraction, which potentially cuts down the numbers of operations by up to half and lets it be executed faster.

First the registers are initialized with corresponding values (Q = Dividend, M = Divisor, A = 0, n = number of bits in dividend)Graphical user interface, application

Description automatically generated. Here, register Q contain quotient and register A contain remainder. Here, n-bit dividend is loaded in Q and divisor is loaded in M. Value of Register is initially kept 0. Non-restoring division uses the digit set {−1, 1} for the quotient digits instead of {0, 1}.

# Algorithm

The basic algorithm for binary (radix 2) non-restoring division of non-negative numbers is:

R := N

D := D << n *-- R and D need twice the word width of N and Q*

**for** i = n − 1 .. 0 **do** *-- for example 31..0 for 32 bits*

**if** R >= 0 **then**

q[i] := +1

R := 2 \* R − D

**else**

q[i] := −1

R := 2 \* R + D

**end** **if**

**end**

In simpler terms, let the dividend be Q and the divisor be M and the accumulator A = 0.

Therefore:

1. First the registers are initialized with corresponding values
2. Check the sign bit of register A
3. If it is 1 shift left content of AQ and perform A = A+M, otherwise shift left AQ and perform A = A-M
4. If sign bit of register A is 1 Q[0] become 0 otherwise Q[0] become 1
5. Decrements value of N by 1, If N is not equal to zero go to Step 2
6. If sign bit of A is 1 then perform A = A+M

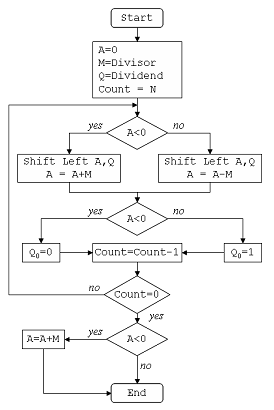
# Example

**Problem: 13/4 M=0100 Q=1101 -M=1100 N=4**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N** | **M** | **A** | **Q** | **Operation** |
| 4 | 0100 | 0000 | 1101 | Initialization |
|  | 0100 | 0001 | 101- | LS, N=N-1 |
|  | 0100 | 1101 | 101- | A=A-M |
|  | 0100 | 1101 | 1010 |  |
| 3 | 0100 | 1101 | 1010 |  |
|  | 0100 | 1011 | 010- | LS, N=N-1 |
|  | 0100 | 1111 | 010- | A=A+B |
|  | 0100 | 1111 | 0100 |  |
| 2 | 0100 | 1111 | 0100 |  |
|  | 0100 | 1110 | 100- | LS, N=N-1 |
|  | 0100 | 0010 | 100- | A=A+B |
|  | 0100 | 0010 | 1001 |  |
| 1 | 0100 | 0010 | 1001 |  |
|  | 0100 | 0101 | 001- | LS, N=N-1 |
|  | 0100 | 0001 | 001- | A=A-B |
|  | 0100 | 0001 | 0011 | Termination |

**Result: Quotient: 0011 Remainder: 0001**

# Flowchart



# Code

#include <iostream>

#include <string>

void decimal\_to\_binary(int n, std::string& bin\_num) {

    if (n / 2 != 0) {

        decimal\_to\_binary(n / 2, bin\_num);

    }

    bin\_num.append(std::to\_string(n % 2));

}

void format\_decimal\_to\_binary(int n, std::string& bin\_num) {

    decimal\_to\_binary(n, bin\_num);

    while (bin\_num.length() != 4) {

        bin\_num.insert(0, "0");

    }

}

void add\_binary(std::string& bin\_num1, std::string& bin\_num2) {

    std::string resp;

    std::string carry = "0";

    for (int i = bin\_num1.length() - 1; i >= 0; i--) {

        int ones = 0;

        if (bin\_num1[i] == '1') {

            ones++;

        }

        if (bin\_num2[i] == '1') {

            ones++;

        }

        if (carry == "1") {

            ones++;

        }

        if (ones == 3) {

            resp.insert(0, "1");

            carry = "1";

        } else if (ones == 2) {

            resp.insert(0, "0");

            carry = "1";

        } else if (ones == 1) {

            resp.insert(0, "1");

            carry = "0";

        } else {

            resp.insert(0, "0");

            carry = "0";

        }

    }

    bin\_num1 = resp;

}

void ones\_complement(std::string& bin\_num) {

    std::string resp;

    for (char& c : bin\_num) {

        resp.append(c == '1' ? "0" : "1");

    }

    bin\_num = resp;

}

void twos\_complement(std::string& bin\_num) {

    std::string resp;

    std::string one = "0001";

    ones\_complement(bin\_num);

    add\_binary(bin\_num, one);

}

void subtract\_binary(std::string& Q, std::string& A, std::string M) {

    std::string \_M = M;

    twos\_complement(\_M);

    add\_binary(A, \_M);

}

int left\_shift(std::string& Q, std::string& A) {

    int tmp = A[0];

    A.erase(A.begin());

    A.push\_back(Q.front());

    Q.erase(Q.begin());

    Q.append("\_");

    return tmp;

}

void check\_result(std::string& Q, std::string& A) {

    Q.pop\_back();

    if (A[0] == '0') {

        Q.append("1");

    } else {

        Q.append("0");

    }

}

void print\_result(std::string& Q, std::string& A) {

    std::cout << "\nQuotient:  " << Q << "\nRemainder: " << A << std::endl;

}

int main() {

    std::string Q, M, A, A\_back;

    int q, m;

    std::cout << "\nEnter Dividend: ";

    std::cin >> q;

    std::cout << "Enter Divisor:  ";

    std::cin >> m;

    format\_decimal\_to\_binary(q, Q);

    format\_decimal\_to\_binary(m, M);

    A = "0000";

    std::cout << "\n n"

              << "\t"

              << " M"

              << "\t"

              << " A"

              << "\t"

              << " Q" << std::endl;

    for (int i = Q.length() - 1; i >= 0; i--) {

        std::cout << "\n " << i + 1 << "\t" << M << "\t" << A << "\t" << Q << std::endl;

        int key = left\_shift(Q, A);

        std::cout << "\t" << M << "\t" << A << "\t" << Q << std::endl;

        if (key == '0') {

            subtract\_binary(Q, A, M);

            std::cout << "\t" << M << "\t" << A << "\t" << Q << std::endl;

        } else {

            add\_binary(A, M);

            std::cout << "\t" << M << "\t" << A << "\t" << Q << std::endl;

        }

        check\_result(Q, A);

        std::cout << "\t" << M << "\t" << A << "\t" << Q << std::endl;

    }

    if (A[0] == '1') {

        add\_binary(A, M);

    }

    print\_result(Q, A);

    return 0;

}

# Output

1. **13 / 4** 2. **14 / 3**

Table, calendar

Description automatically generated Table, calendar

Description automatically generated

# Conclusion

The non-restorative division algorithm is an efficient way to perform binary division compared to traditional subtractive based algorithms by using the faster processed bit shift commands in the CPU registers. The algorithm is simple enough to be implemented in hardware in equipment like Arithmometers while also generalising to complex modern day systems. The algorithm serves as a good example in showing that considering lower-level system dependencies and physical limitations can be used to optimize algorithms. Non-restorative algorithm is more efficient than restorative algorithm as it uses simpler commands in terms of addition and subtraction however it is slower than other algorithms.