A Lab Report on Computer Organization and Architecture

Ishwor Basyal (077BCT031),[*] Kalpesh Manandhar (077BCT034),[†] and Mamata Maharjan (077BCT043)[‡]
Department of Electronics and Computer Engineering,
Pulchowk Campus, Institute of Engineering, Lalitpur.
(Dated: August 18, 2023)

This report highlights the major algorithms used in Computer Architecture for signed and unsigned integer arithmetic. It also covers the concepts of page replacement and the three most common algorithms used for page replacement in computer systems. It aims to give some insights into how these algorithms work, some examples and their software implementations using a high level language like Python.

I. INTRODUCTION

Computer arithmetic is an integral part of computing, basically how computers process inputs to give outputs. All the basic arithmetic and logic operations, including signed and unsigned addition, subtraction, multiplication, and division, are carried out by the arithmetic and logical unit (ALU), which is the core or essence of the computer. The data are present in registers within the CPU, which are passed into the ALU as inputs; the ALU processes these inputs and gives out some outputs, and sets the respective flags.

In the upcoming sections, we shall understand how a computer truly works under the hood.

The representation of binary numbers inside computers is of many forms: from signed integers to unsigned integers to floating point numbers and many more. The integer representations of these binary numbers are discussed below:

- **Unsigned integer representation:**
  The representation of numbers as unsigned numbers is relatively simple, being stored as regular n-bit binary numbers. It ranges from 0 to $2^n - 1$ in magnitude and can be represented by

  $$A = \sum_{i=0}^{n-1} 2^i a_i$$

- **Signed integer representation:**
  The representation of binary numbers as signed numbers is however a bit more complicated compared to unsigned numbers. Representing the sign of the number isn't as simple as attaching a minus sign at the front of the number, as only two logic levels exist inside the computer. Hence, the most significant bit of the number is used to represent the sign. However, simply using this convention has a major disadvantage: one being that arithmetic requires consideration of sign of both operands and another being the two representations of 0:
  $+0 = 00000000_2$
  $-0 = 10000000_2$
  Hence, a two's complement representation is used to represent signed numbers in computers. The most significant bit still represents the sign, and the representation can be defined as a weighted sum of bits represented by

  $$A = -2^{n-1}a_{n-1} + \sum_{i=0}^{n-2} 2^i a_i$$

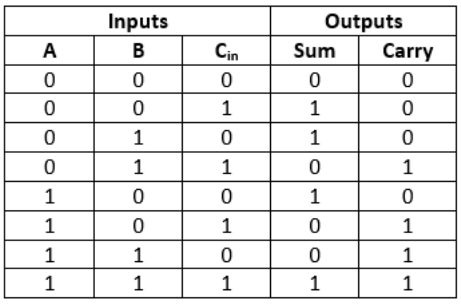
  The value usually ranges from -2^{n-2} to $+2^{n-2} - 1$ and it also eliminates the problem of multiple zeroes.

II. LABS

A. Lab 1 : ADDITION OF TWO UNSIGNED BINARY INTEGERS

- **Objective:** To Design an n-bit Adder for two Unsigned binary integers
- **Theory:** Full Adder is a logical component that adds three inputs and produces two outputs. The first two inputs are the two bits to be added; let's call them A and B and the third input is the carry bit from the previous addition result as C-IN. The outputs are the sum bit and a carry bit. These one-bit components are cascaded together to obtain n- bit adder component.

[*] Email: 077bct031.ishwor@pcampus.edu.np
[†] Email: 077bct034.kalpesh@pcampus.edu.np
[‡] Email: 077bct043.mamata@pcampus.edu.np



Inputs			Outputs	
A	B	C_{in}	Sum	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Figure 1. Fig 1.1: Full Adder Truth Table

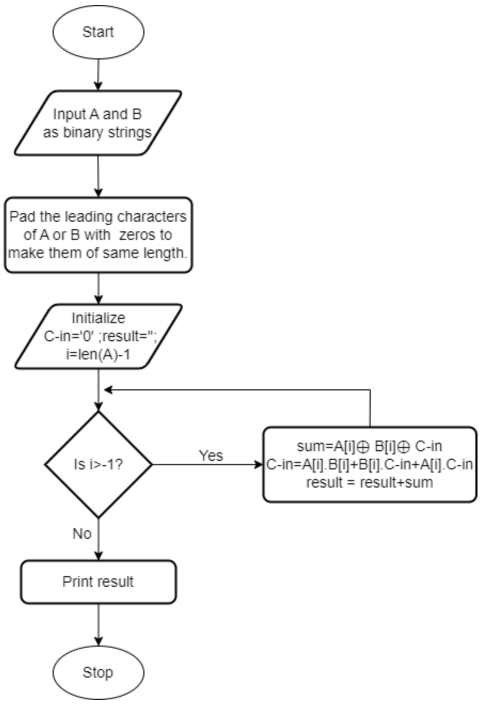
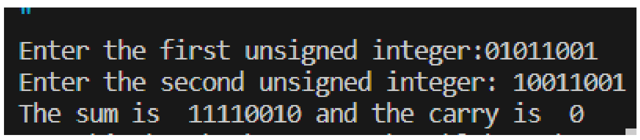

Start
Input A and B as binary strings
Pad the leading characters of A or B with zeros to make them of same length.
Initialize C-in='0' ;result=''; i=len(A)-1
Is i>-1?
Yes
sum=A[i]⊕ B[i]⊕ C-in
C-in=A[i].B[i]+B[i].C-in+A[i].C-in
result = result+sum
No
Print result
Stop

Figure 2. Fig 1.2: Flowchart of two unsigned integer addition

Source code

```
from helpers import XOR,OR,NOT,AND;

def add(str1, str2):

    n=max(len(str1),len(str2))
    str1=str1.zfill(n)
    str2=str2.zfill(n)
    res=’’
    carry=’0’

    for i in reversed(range(len(str1))):
        sum= XOR(XOR(str1[i],str2[i]),carry)
        carry=OR(OR(AND(str1[i],str2[i]),AND(
            str1[i], carry)),AND(str2[i],carry)
            )
        res += sum


    # to the reversed string, add the carry at
        the front
    return res[::-1], str(carry)


def adder(str1, str2):
    a,b=add(str1, str2)
    c=b+a
    return c
```

- **Results and Analysis**



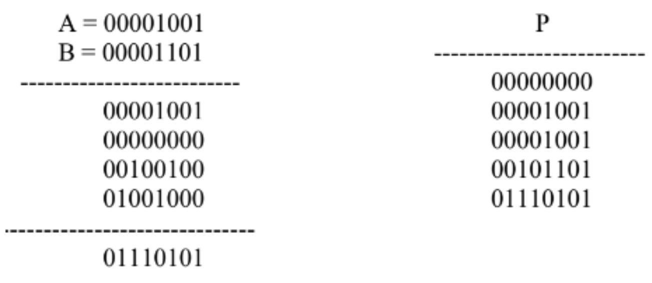
```
Enter the first unsigned integer:01011001
Enter the second unsigned integer: 10011001
The sum is  11110010 and the carry is  0
```

For the binary addition program, the two inputs were A= 01011001 i.e. 89, and B= 10011001 i.e. 153. The output was as expected i.e. 11110010. Since this is exactly one byte, there was no carry generated.

B. Lab 2 : MULTIPLICATION OF TWO UNSIGNED INTEGER BINARY NUMBERS BY PARTIAL-PRODUCT METHOD.

- **Objective:** To simulate binary multiplication by partial product method.
- **Theory:** The program for multiplying two numbers is based on the procedure we use to multiply number with paper and pencil. Multiplication process consists of checking the bits of the multiplier B and adding the multiplicand A, as many times as there are 1’s in B, provided that the value of A is shifted left from one line to the next. As the computer can add only two numbers at a time, we reserve a memory location, P (say) to store intermediates sums. The intermediate sum is called partial products as they hold a partial product until all numbers are added. This is the reason why the method named partial product method. Partial product is initially started with the zero. The multiplicand A is added to the content of P for each bit of the multiplier B that is 1. The Value of A is shifted left after checking each bit of the multiplier. The final value in P gives the products of the two unsigned integer binary number. For 4-bit numbers, when multiplied, the product contains eight significant bits.

An example with four significant digits is shown below:



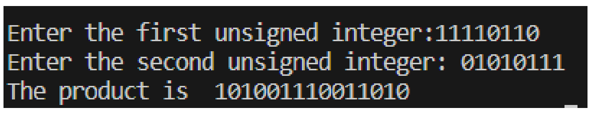
```
 A = 00001001                          P
 B = 00001101                 -------------------------
--------------------------             00000000
     00001001                          00001001
     00000000                          00001001
     00100100                          00101101
     01001000                          01110101
------------------------------
     01110101
```

Figure 3. Fig 2.1: Multiplication by partial product method

Source code

```python
# unsigned partial product multiplication

from lab1 import add
from helpers import reverse
from helpers import nBitPadUnsigned
from helpers import shiftRight

def partialProduct(a,b,n = 8):
    a = nBitPadUnsigned(a,n)
    y = reverse(nBitPadUnsigned(b,n)) #y = B
    res = nBitPadUnsigned('0',2*n)   #res = AQ

    for i in range(0,n):
        if y[i] == '1':
            sum,cy = add(a,res[0:n],n) # sum =
                A + a, cy = carry
            res = sum + res[n:2*n]    # res = AQ
        res = shiftRight(res,cy)      # right
             shift Cy AQ
    return(res)
```

- **Results and Analysis**



```
Enter the first unsigned integer:11110110
Enter the second unsigned integer: 01010111
The product is  101001110011010
```

The partial product algorithm has been demonstrated with two 8 bit numbers 11110110_2 (246_{10}) and 01010111_2 (87_{10}). The output received is a 16 bit number 101001110011010_2 (21402_{10}), which is the required product of the given inputs.

C. Lab 3 : SUBTRACTION OF TWO UNSIGNED INTEGER BINARY NUMBER

- **Objective:** To design n-bit (4-bit) sub tractor for unsigned integer binary numbers.
- **Theory:** Subtraction is the basic arithmetic operation. In digital computers complement is used for simplifying the subtraction. There are two types of compliments for base 2 system: 1's complement and 2's complement. The 1's complement of binary number is obtained by subtracting each digit from 1.However, the subtraction of a binary digit from 1 causes the bit to change from 0 to 1 or from 1to 0.Therefore, the 1's complement of a binary number is formed by changing 1's into 0's and 0's into 1's. For example 1's complement of 0111 is 1000. 2's complement can be formed by leaving all least significant 0's and first 1 unchanged, and then replacing 1's by 0's and 0's by 1's in all other higher significant bits.

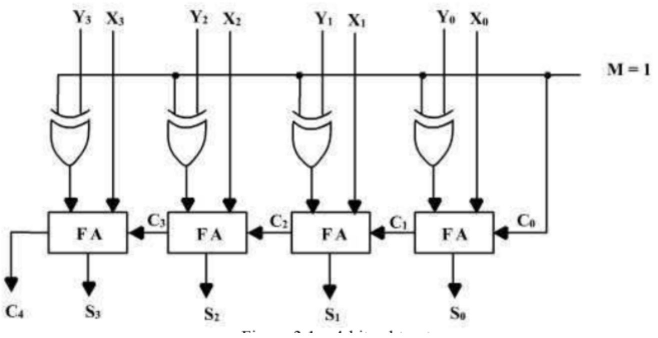
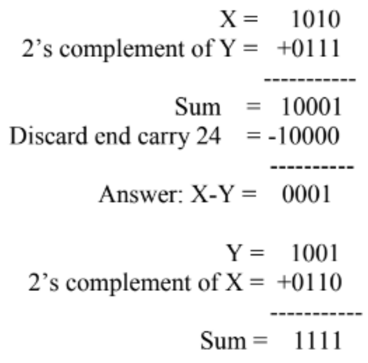


Figure 4. Fig 3.1: 4-bit Subtractor

Each exclusive –OR gate receives input M and one of the input bit of Y. We have Y + 1 = Y' And C0 = 1. The Y inputs are all complemented and a 1 is added through the input carry. The circuit performs the operation X plus the 2's complement of Y. For unsigned numbers, this gives X-Y if X >=Y or the 2's complement of (Y-X) if X<Y.

- **For Example:** The 2's complement of 0111 is 1001 and is obtained by leaving first 1 unchanged, and then replacing 1's by 0's and 0's by 1's in the other three most significant bits. Let us see an example of subtraction using two binary numbers X = 10(1010) and Y = 9 (1001) and perform X-Y and Y-X.



```
                     X =    1010
  2's complement of Y =   +0111
                         -----------
              Sum   =   10001
 Discard end carry 24   = -10000
                         ----------
         Answer: X-Y =    0001

                     Y =    1001
  2's complement of X =   +0110
                         -----------
                  Sum =    1111
```

Figure 5. Fig 3.2: Example
There is no end carry Answer is negative 0001 = 2's compliment of 1111

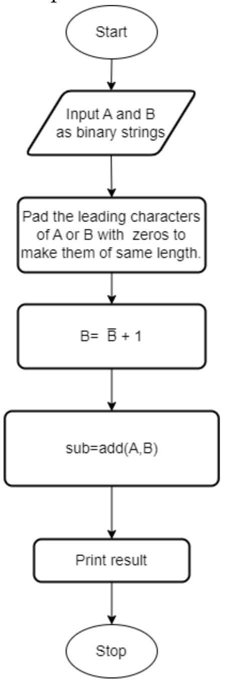
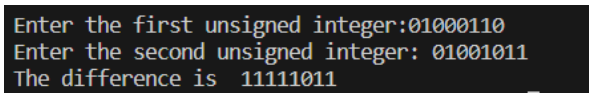

Start
Input A and B as binary strings
Pad the leading characters of A or B with zeros to make them of same length.
B= B̄ + 1
sub=add(A,B)
Print result
Stop

Figure 6. Fig: Flowchart for binary subtraction

Source code

```python
from helpers import nBitPadSigned;
from helpers import reverse;
from helpers import XOR,OR,NOT,AND;

def FullSubtractor(x,y,bIn):
    s = XOR(XOR(x,y),bIn)
    b = OR(OR(AND(NOT(x),y),AND(NOT(x),bIn)),
        AND(y,bIn))
    return s,b

def subtract(x,y,n=8):
    a = reverse(nBitPadSigned(x, n))
    b = reverse(nBitPadSigned(y, n))
    brw = '0'
    r = nBitPadSigned('0', n)
    for i in range(0,n):
        diff,brw = FullSubtractor(a[i], b[i],
            brw)
        r = r[:i] + diff + r[i+1:]
    r = reverse(nBitPadSigned(r, n))
    return(r,brw)
```

- **Results and Analysis**



```
Enter the first unsigned integer:01000110
Enter the second unsigned integer: 01001011
The difference is  11111011
```

In binary subtraction, the two inputs were 01000110 and 01001011 i.e. 70 and 75 respectively. The output was 11111011 which is the two's complement representation of 5, which is a standard way to represent negative numbers. We can verify this result by taking the two's complement of the result to obtain 00000101 which is the binary representation of the number 5.

D. Lab 4: DIVISION OF TWO UNSIGNED INTEGERS USING RESTORING DIVISION ALGORITHM

- **Objective:** To implement the restoring division for the division of two unsigned numbers.
- **Theory:** The basis for the restoring division algorithm is the paper-and-pencil approach to division. It involves repetitive shifting and addition or subtraction.

  The bits of the dividend are examined from left to right. Until the set of bits represents a number that the divisor can be subtracted from, a 0 is added to the quotient and the next bit is examined by left shifting. When the divisor can be subtracted from the set of bits, then this is referred to as the divisor being able to divide the dividend, and a 1 is added as the next bit of the quotient. This process is then repeated n times for a n-bit dividend, until all the bits are exhausted. At the end, the quotient is present in the quotient register and the remainder is present as the final remaining difference. This is the general process of the restoring division. However, how does the computer know whether the set of bits of the dividend are greater than the divisor? In this algorithm, the divisor is subtracted from the dividend bit even though the result may be negative. This is checked by the sign bit of the difference, and the divisor is re-added to the difference in order to restore the previous value. Hence, this algorithm is called the Restoring Division algorithm.

For example, the division of 7 (0111_2) by 3 (0011_2) can be shown as follows:

A	Q	Operations
0000	0111	Initial Value
0000	1110	Shift
1101		Subtract
0000	1110	Restore
0001	1100	Shift
1110		Subtract
0001	1100	Restore
0011	1000	Shift
0000		Subtract
0000	1001	Set Q_0 to 1
0001	0010	Shift
1110		Subtract
0001	0010	Restore

Table I. Restoring Division

Quotient $=$ **2** (0010_2)
Remainder $=$ **1** (0001_2)

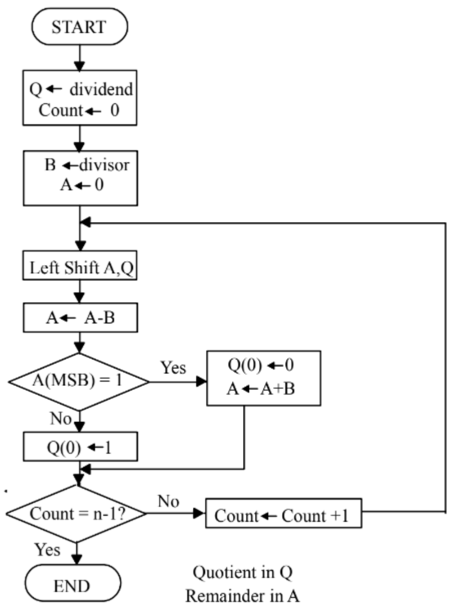
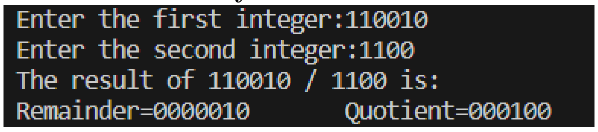


Figure 7. Fig 4.1: Restoring Division Flow Chart

Source code

```
# Restoring Division

from helpers import nBitPadUnsigned
from helpers import shiftLeft
from lab1 import add
from lab1 import subtract


# a is bit, b is 8 bit
def restoringDiv(a,b,n=8):
    if nBitPadUnsigned(b,n) == nBitPadUnsigned
        ('0',n):
        return "Divide by zero"

    divisor = nBitPadUnsigned(b,n)
                            # n bit divisor
    res = nBitPadUnsigned('0',n) +
        nBitPadUnsigned(a,n) # 2n bit res =
        AQ
    print(res)
    remainder = nBitPadUnsigned('0',n)
                        # n bit remainder

    for i in range(0,n):
        res = shiftLeft(res, '0')
                                # shift AQ
        remainder, dummy = subtract(res[0:n],
            divisor,n)
        res = remainder + res[n:2*n]

        if (remainder[0] == '0'):
                                # sub >
            divisor
            res = res[:-1] + '1'
                                    # set
                q0 to 1
        elif (remainder[0] == '1'):
                                # sub <
            divisor
            remainder, dummy = add(res[0:n],
                divisor, n) # restore value
                back and keep q0 = 0
            res = remainder + res[n:2*n]
        print(res)
    quotient = res[n:2*n]
    return quotient, remainder
```

- **Results and Analysis**



```
Enter the first integer:110010
Enter the second integer:1100
The result of 110010 / 1100 is:
Remainder=0000010       Quotient=000100
```

Restoring division is used with two unsigned numbers: 110010_2 divided by 1100_2, i.e., $50_{10}/12_{10}$. The acquired quotient is $000100_2 = 4_{10}$ and the remainder is $0000010_2 = 2_{10}$.

E. Lab 5 : DIVISION OF TWO UNSIGNED INTEGER INTEGERS USING NON-RESTORING DIVISION

- **Objective:** To implement non-restoring division algorithm in digital computer.

- **Theory:** In the non-restoring algorithm, B is not added if the difference is negative but instead, the negative difference is shifted left and then B is added. Here B is subtracted if the previous value of Q_{LSB} was 1, but B is added if the previous value of Q_{LSB} was 0 and no restoring of partial remainder is required. The first time the dividend is shifted, B must be subtracted. Also, if the last bit of the quotient is 0, the partial remainder must be restored to obtain the correct final remainder. Consider two binary numbers A and B. A is the dividend, B the divisor and ($Q = A/B$) the quotient. We assume that (A>B) and (B!=0).

  Example 1, The division of 12 (1011_2) by 4 (00100_2) can be shown as follows:

A	Q	Operations
00000	1011	Initial Value
00001	011|0	Shift
11100		Subtract
11101	100|0	Set Q_0 to 0
11011	00|00	Shift
00100		Add
11111	00|00	Set Q_0 to 0
11110	0|000	Shift
00100		Add
00010	0|001	Set Q_0 to 1
00100	0010	Shift
11100		Subtract
00000	0011	Set Q_0 to 1

  Quotient = 3 (0011_2)
  Remainder = 0 (0000_2)

  Let the number of bits stored in register Q is n Registers AQ is now shifted to the left with zero insertion into Q_{LSB}. Initialize the counter to zero value. And divisor is subtracted by adding 2's complement value. If A_{MSB} =1, set Q_{LSB} with value 0 and then increment the counter value by 1. The partial remainder is shifted to the left and then B is added to the partial remainder. . If A_{MSB} =0, set Q_{LSB} with value 1 and then increment the counter value by 1. Process is repeated until count = n-1 i.e. all quotient bits are formed. If the last bit of the quotient is 0, the partial remainder must be restored to obtain the correct final remainder. Finally result, Quotient is in Q and the final remainder is in A, is obtained.

Example 2, The division of 8 (1000_2) by 3 (00011_2) can be shown as follows:

A	Q	Operations
00000	1000	Initial Value
00001	000|0	Shift
11101		Subtract
11110	000|0	Set Q_0 to 0
11100	00|00	Shift
00011		Add
11111	00|00	Set Q_0 to 0
11110	0|000	Shift
00011		Add
00001	0|001	Set Q_0 to 1
00010	0010	Shift
11101		Subtract
11111	0010	Set Q_0 to 0
Add	11111	value at A
	00011	divisor
	00010	remainder

Quotient = 2 (0010_2)
Remainder = 2 (0010_2)

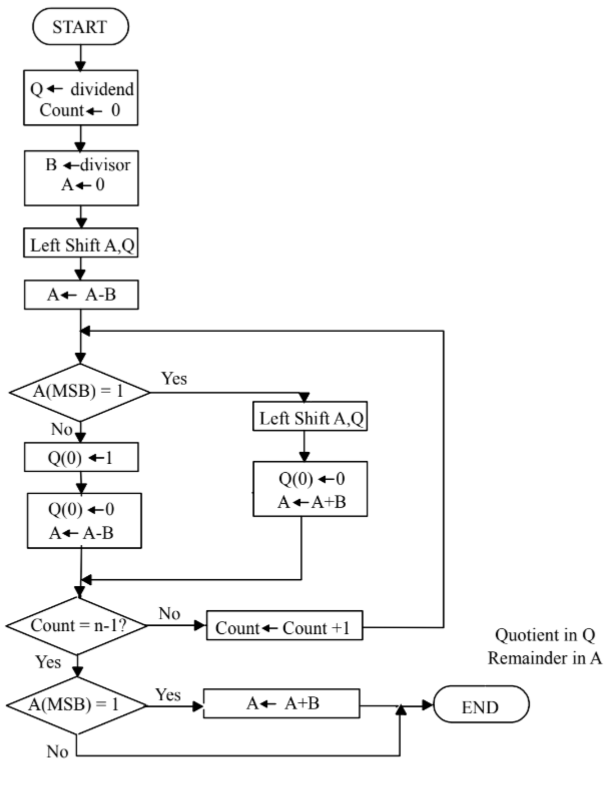
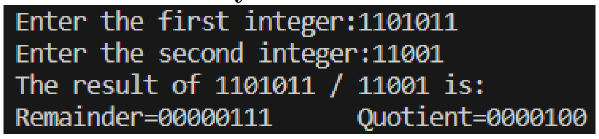


Figure 8. Fig 5.1: Non-Restoring Division Flow Chart

Source code

```
# non restoring division

from helpers import nBitPadUnsigned
from helpers import shiftLeft
from lab1 import add,subtract

def nonRestoringDiv(a,b,n=8):
    if nBitPadUnsigned(b,n) == nBitPadUnsigned(
        '0',n):
        return "Divide by zero"

    divisor = nBitPadUnsigned(b,n) # n bit
        divisor
    res = nBitPadUnsigned('0',n) +
        nBitPadUnsigned(a,n) # 2n bit res = AQ
    remainder = nBitPadUnsigned('0',n) # 2n bit
        remainder
    q0 = '1'
    for i in range(0,n):
        res = shiftLeft(res, '0') # shift AQ

        if (q0 == '0'): # prev q0 = 0: add
            divisor
            remainder, dummy = add(res[0:n],
                divisor,n)
        elif (q0 == '1'): # prev q0 = 1:
            subtract divisor
            remainder, dummy = subtract(res[0:n
                ], divisor,n)
        res = remainder + res[n:2*n]

        if (remainder[0] == '0'): # sub >
            divisor
            res = res[:-1] + '1' # set q0 to 1
        elif (remainder[0] == '1'): # sub <
            divisor
            res = res # leave q0 = 0
        q0 = res[-1]

    if (remainder[0] == '1'): # if last
        remainder is negative
        remainder,dummy = add(res[0:n], divisor
            ,n) # add divisor to get actual
            remainder

    quotient = res[n:2*n]
    return quotient, remainder
```

- **Results and Analysis**



```
Enter the first integer:1101011
Enter the second integer:11001
The result of 1101011 / 11001 is:
Remainder=00000111        Quotient=0000100
```

Non-restoring division, too, is used with two unsigned integers: 1101011_2 divided by 11001_2, i.e., $107_{10}/25_{10}$. The acquired quotient is $0000100_2 = 4_{10}$ and the remainder is $00000111_2 = 7_{10}$.

F. Lab 6: PAGE REPLACEMENT ALGORITHMS

- **Objective:** To implement various memory page replacement algorithms

- **Theory:** The concept of virtual memory combines both software and hardware techniques. Since the virtual address space is almost always significantly greater than the physical memory space available, the virtual memory pages need to be mapped to the limited physical memory to be used. Hence, in order to efficiently map the pages for the highest hit ratio, there are various algorithms used for page replacement. The ones that are discussed here are Least Recently Used (LRU), Least Frequently Used (LFU), and First In First Out (FIFO).

  In the LRU algorithm, the least recently used page is replaced when a new page needs to be loaded. As a page is used recently, the algorithm expects that the page will be used again in the near future. This is the concept of spatial locality.

  In the LFU algorithm, the least frequently used page, i.e., the page with the least amount of uses is replaced. The algorithm expects that a page with more uses will continue to have more uses.

  In the FIFO algorithm, the oldest page, i.e., the page loaded the earliest is replaced. The algorithm works best when there are less references to the earlier pages.

  For example, for a physical memory with 4 pages, the sequence of page accesses 2,3,2,1,5,2,4,5,3,2,3,2 produces the following page loads/replacements

Source code

```
# page restoration algorithms LRU/FIFO/LFU

class PageTable:
    def __init__(self, tableSize):
        self.maxsize = tableSize
        self.currentSize = 0
        self.table = []
        self.pageData = {}
        self.replacementAlg = "LRU"
        self.hits = 0
        self.miss = 0

    def mapNewPage(self, name, index):
        # new entry
        self.table[index] = name
```

Algorithm	Physical Pages				Operation
	-	-	-	-	Initial
	2	-	-	-	Load page 2
	2	3	-	-	Load page 3
	2	3	-	-	Page hit 2
	2	3	1	-	Load page 1
	2	3	1	5	Load page 5
LRU	2	3	1	5	Page hit 2
	2	4	1	5	Load page 4/Replace page 3
	2	4	1	5	Page hit 5
	2	4	3	5	Load page 3/Replace page 1
	2	4	3	5	Page hit 2
	2	4	3	5	Page hit 3
	2	4	3	5	Page hit 2
	-	-	-	-	Initial
	2	-	-	-	Load page 2
	2	3	-	-	Load page 3
	2	3	-	-	Page hit 2
LFU	2	3	1	-	Load page 1
	2	3	1	5	Load page 5
	2	3	1	5	Page hit 2
	2	4	1	5	Load page 4/Replace page 3
	2	4	1	5	Page hit 5
	2	4	3	5	Load page 3/Replace page 1
	2	4	3	5	Page hit 2
	2	4	3	5	Page hit 3
	2	4	3	5	Page hit 2
	-	-	-	-	Initial
	2	-	-	-	Load page 2
	2	3	-	-	Load page 3
	2	3	-	-	Page hit 2
FIFO	2	3	1	-	Load page 1
	2	3	1	5	Load page 5
	2	3	1	5	Page hit 2
	4	3	1	5	Load page 4/Replace page 2
	4	3	1	5	Page hit 5
	4	3	1	5	Page hit 3
	4	2	1	5	Load page 2/Replace page 3
	4	2	3	5	Load page 2/Replace page 3
	4	2	3	5	Page hit 2

Table II. Page replacement using various algorithms

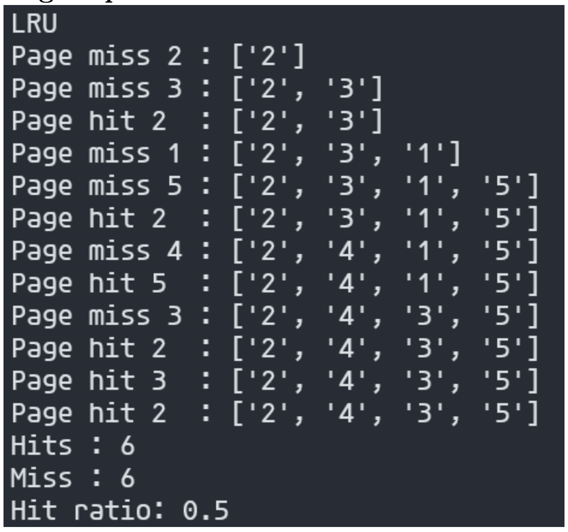
```
    self.pageData[name] = {
        "uses": 1,
        "recent" : 0,
        "order" : 0
    }

    print(f"Added new page {name} at index
         {index}")


def replaceFIFO(self, name):
    # find the first mapped page
    for key in self.pageData.keys():
        if (self.pageData[key]["order"] ==
             (self.maxsize-1)):
```

```
    # get index of replaceable entry
         and update
    index = self.table.index(key)
    prevPage = self.pageData.pop(key
        )

    # update remaining entries
        depending on recency of
        replaced page
    for k in self.pageData.keys():
        if (self.pageData[k]["recent
            "] < prevPage["recent"])
            :
            self.pageData[k]["recent"
                ] += 1
        self.pageData[k]["order"] +=
                    1

            print(f"[FIFO] Replaced page {
                key} at index {index}")
            self.mapNewPage(name, index)
            break


def replaceLRU(self, name):
    # find the least recently used page
    for key in self.pageData.keys():
        if (self.pageData[key]["recent"] ==
             (self.maxsize-1)):
            # get index of replaceable entry
                 and update
            index = self.table.index(key)
            prevPage = self.pageData.pop(key
                )

            # update remaining entries
                depending on mapped order of
                 replaced page
            for k in self.pageData.keys():
                if (self.pageData[k]["order"
                    ] < prevPage["order"]):
                    self.pageData[k]["order"]
                         += 1
                self.pageData[k]["recent"]
                    += 1

            print(f"[LRU] Replaced page {key
                } at index {index}")
            self.mapNewPage(name, index)
            break


def replaceLFU(self, name):
    # find the least frequently used page
    leastFreqkey = self.table[0]
    for key in self.pageData.keys():
        # if key has less uses than least
            freq key: replace least freq
            key
        if (self.pageData[key]["uses"] <
            self.pageData[leastFreqkey]["
            uses"]):
            leastFreqkey = key
        # if key has same uses as least
            freq key: replace least freq
            key if key was inserted first
        elif (self.pageData[key]["uses"] ==
             self.pageData[leastFreqkey]["
            uses"]):
            if (self.pageData[key]["order"]
                > self.pageData[leastFreqkey
                ]["order"]):
                leastFreqkey = key

    index = self.table.index(leastFreqkey)
    prevPage = self.pageData.pop(
        leastFreqkey)
```

```
    for key in self.pageData.keys():
        if (self.pageData[key]["order"] <
            prevPage["order"]):
            self.pageData[key]["order"] += 1
        if (self.pageData[key]["recent"] <
            prevPage["recent"]):
            self.pageData[key]["recent"] +=
                1

    print(f"[LFU] Replaced page {
        leastFreqkey} at index {index}")
    self.mapNewPage(name, index)


def usePage(self, name):
    try:
        # check if page already exists in
            table
        index = self.table.index(name)

        # if exists
        # update all page data
        for key in self.pageData.keys():
            if (self.pageData[key]["recent"]
                 < self.pageData[name]["
                recent"]):
                self.pageData[key]["recent"]
                     += 1

        # set as most recently used and
            increase uses
        self.pageData[name]["uses"] += 1
        self.pageData[name]["recent"] = 0
        self.hits += 1
        print (f"page hit {name}")

    # if page doesnt exist previously
    except ValueError:

        print (f"page miss {name}")
        self.miss += 1
        # if table is not full add entry
        if (self.currentSize < self.maxsize
            ):
            # update all entries
            for key in self.pageData.keys():
                self.pageData[key]["recent"]
                     += 1
                self.pageData[key]["order"]
                    += 1

            self.table.append("")
            self.mapNewPage(name, self.
                currentSize)
            self.currentSize += 1

        # if table is full
        else:
            if (self.replacementAlg == "FIFO
                "):
                    self.replaceFIFO(name)
                elif (self.replacementAlg == "
                    LFU"):
                    self.replaceLFU(name)
                elif (self.replacementAlg == "
                    LRU"):
                    self.replaceLRU(name)


ptable = PageTable(4)


input=['2', '3', '2', '1', '5', '2', '4', '5',
     '3', '2', '3','2']

ptable.replacementAlg = "FIFO"
for i in input:
    ptable.usePage(i)
    print(ptable.table)
    # print(ptable.pageData)
print(f"Hits : {ptable.hits}")
print(f"Miss : {ptable.miss}")
print(f"Hit ratio: {ptable.hits/(ptable.hits +
     ptable.miss)}")
```

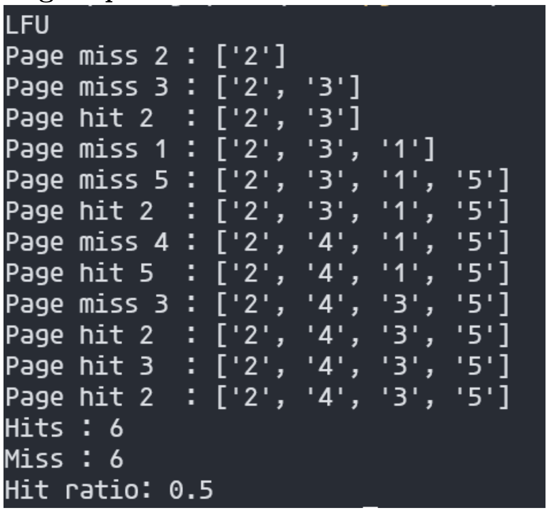
Results and Analysis

- **Page replacement: LRU**



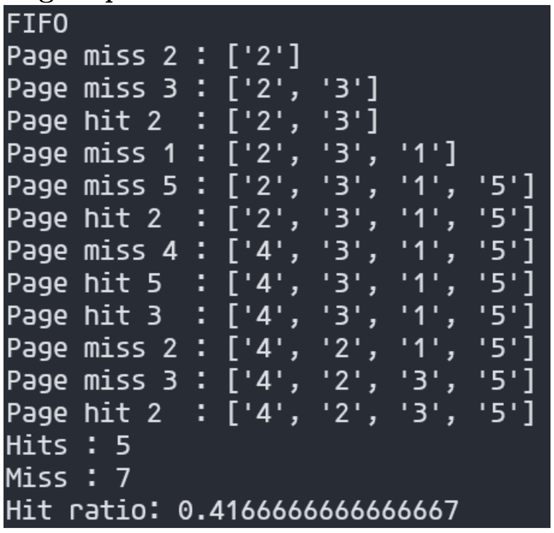
```
LRU
Page miss 2 : ['2']
Page miss 3 : ['2', '3']
Page hit 2  : ['2', '3']
Page miss 1 : ['2', '3', '1']
Page miss 5 : ['2', '3', '1', '5']
Page hit 2  : ['2', '3', '1', '5']
Page miss 4 : ['2', '4', '1', '5']
Page hit 5  : ['2', '4', '1', '5']
Page miss 3 : ['2', '4', '3', '5']
Page hit 2  : ['2', '4', '3', '5']
Page hit 3  : ['2', '4', '3', '5']
Page hit 2  : ['2', '4', '3', '5']
Hits : 6
Miss : 6
Hit ratio: 0.5
```

- **Page replacement: LFU**



```
LFU
Page miss 2 : ['2']
Page miss 3 : ['2', '3']
Page hit 2  : ['2', '3']
Page miss 1 : ['2', '3', '1']
Page miss 5 : ['2', '3', '1', '5']
Page hit 2  : ['2', '3', '1', '5']
Page miss 4 : ['2', '4', '1', '5']
Page hit 5  : ['2', '4', '1', '5']
Page miss 3 : ['2', '4', '3', '5']
Page hit 2  : ['2', '4', '3', '5']
Page hit 3  : ['2', '4', '3', '5']
Page hit 2  : ['2', '4', '3', '5']
Hits : 6
Miss : 6
Hit ratio: 0.5
```

- **Page replacement: FIFO**



```
FIFO
Page miss 2 : ['2']
Page miss 3 : ['2', '3']
Page hit 2  : ['2', '3']
Page miss 1 : ['2', '3', '1']
Page miss 5 : ['2', '3', '1', '5']
Page hit 2  : ['2', '3', '1', '5']
Page miss 4 : ['4', '3', '1', '5']
Page hit 5  : ['4', '3', '1', '5']
Page hit 3  : ['4', '3', '1', '5']
Page miss 2 : ['4', '2', '1', '5']
Page miss 3 : ['4', '2', '3', '5']
Page hit 2  : ['4', '2', '3', '5']
Hits : 5
Miss : 7
Hit ratio: 0.4166666666666667
```

For the page replacement algorithms, the input page reference sequence is given as 2,3,2,1,5,2,4,5,3,2,3,2.
Using LRU for replacement, a hit ratio of 0.5 is achieved with 6 page hits and 6 page faults/misses. Using LFU, the same hit ratio of 0.5 is achieved. With FIFO, a hit ratio of 0.4166 is achieved with 5 hits and 7 misses. It is due to the fact that there are maximum references to page 2 and 3, which are considered in LRU and LFU. However, since the references do not affect the replacement order in FIFO, the references to 2 and 3, which are the first to be loaded and first to be replaced, cause more misses when compared to LFU and LRU.

G. Lab 7: BOOTH'S ALGORITHM FOR SIGNED BINARY INTEGER MULTIPLICATION

- **Objective:** To find the multiplication of two signed binary integers using Booth's Algorithm
- **Theory:** The booth algorithm is a multiplication algorithm that allows us to multiply the two signed binary integers in 2's complement, respectively. It is also used to speed up the performance of the multiplication process. It is very efficient too. It works on the string bits 0's in the multiplier that requires no additional bit only shift the right-most string bits and a string of 1's in a multiplier bit weight 2k to weight 2m that can be considered as 2k+ 1 - 2m.

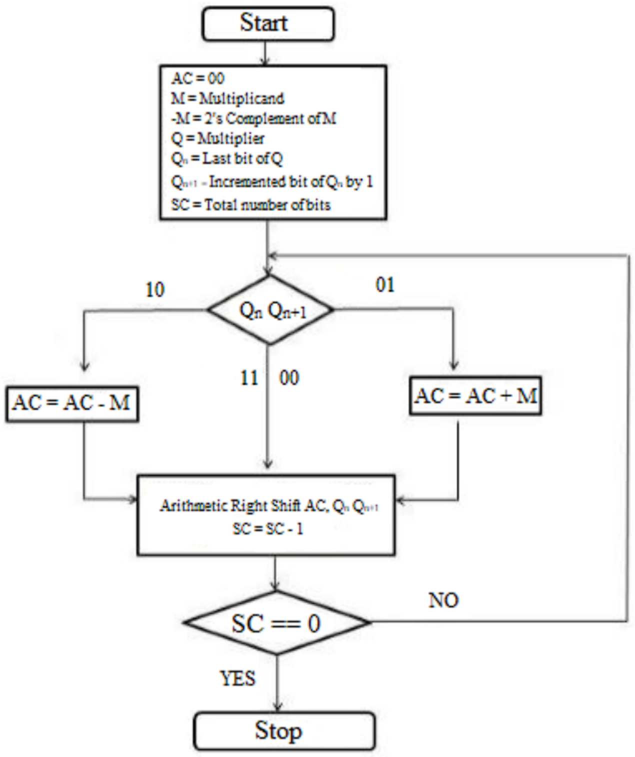


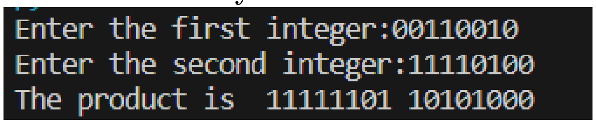
Figure 9. Fig 5.1: Booth's Algorithm Flow Chart

In the given flowchart, initially, AC and Q_{n+1} bits are set to 0, and the SC is a sequence counter that represents the total bits set n, which is equal to the number of bits in the multiplier. There are BR that represent the multiplicand bits, and QR represents the multiplier bits. After that, we encountered two bits of the multiplier as Q_n and Q_{n+1}, where Q_n represents the last bit of QR, and Q_{n+1} represents the incremented bit of Q_n by 1. Suppose two bits of the multiplier is equal to 10; it means that we have to subtract the multiplier from the partial product in the accumulator AC and then perform the arithmetic shift operation (ASHR). If the two of the multipliers equal to 01, it means we need to perform the addition of the multiplicand to the partial product in accumulator AC and then perform the arithmetic shift operation (ASHR), including Q_{n+1}. The arithmetic shift operation is used in Booth's algorithm to shift AC and QR bits to the right by one and remains the sign bit in AC unchanged. And the sequence counter is continuously decremented till the computational loop is repeated, equal to the number of bits (n).

Source code

```python
# Booth's multiplication algorithm

from helpers import nBitPadSigned;
from helpers import shiftRight;
from lab1 import add;
from lab1 import subtract;

def BoothsMul(a,b,n = 8):
    multiplicand = nBitPadSigned(a,n) #signed 8
         bit pad
    res = nBitPadSigned('0',n) + nBitPadSigned(
        b,n) # res = AQ
    qend = '0'
    for i in range(0,n):
        if (qend == '0' and res[-1] == '1'): #
            A = A - M
            sum,dummycy = subtract(res[0:n],
                multiplicand)
            res = sum + res[n:2*n]
        elif (qend == '1' and res[-1] == '0'):
            # A = A + M
            sum,dummycy = add(res[0:n],
                multiplicand)
            res = sum + res[n:2*n]
        qend = res[-1]
        res = shiftRight(res, res[0]) #
            arithmetic right shift AQ Qend
    return res
```

- **Results and Analysis**



```
Enter the first integer:00110010
Enter the second integer:11110100
The product is  11111101 10101000
```

Booth's signed multiplication has been demonstrated with a positive and a negative 8 bit number: 00110010_2 and 11110100_2 corresponding to 50_{10} and -12_{10} respectively. The product received is 1111110110101000_2 which is equal to -600_{10}.

III. RESULTS AND DISCUSSIONS

Since designing a physical ALU wasn't really feasible for implementing these algorithms, they were simulated using Python programs. Here, the registers acted as string variables, and the binary numbers were represented as strings.

The respective outputs and results of all the individual labs have already been discussed in their own sections.

Overall, the results were as expected. All the source files used some helper functions for padding, shifting and so on, which were stored in a module helper.py which can be found in the appendix. Furthermore, the add and subtract functions were reused from the addition and subtraction labs, also found in the appendix. The page replacement algorithms were simulated by making a page table class and replacing the pages as per the algorithm selected.

Some key points that can be noted throughout these labs are:

- Two's complement representation is used for the signed representation of integers.
- Partial product method of multiplication only works for unsigned integers while Booth's algorithm works for both unsigned and signed integers.
- Non-restoring division works for only unsigned integers while restoring division can be extended to work for signed integers as well.
- The most commonly used algorithms for page replacement and cache line replacement are LRU, LFU and FIFO. LRU works well when there are more references to pages that are recently used. LFU works well when majority of references are to a small group of pages. FIFO works well for a sequence of references where there are more references to pages recently added, and less references to pages added earlier.

IV. CONCLUSIONS

Hence, in conclusion, various algorithms for computer arithmetic and memory mapping were implemented by writing programs in Python. These arithmetic algorithms, in essence, define how computers compute. The page replacement algorithms are an important part in virtual memory spaces and are also used in associative cache mapping. Hence, the discussed algorithms were successfully implemented and observed in action throughout the course of these labs.

V. REFERENCES

1. Stallings, W. *Computer Organization and Architecture: Designing for Performance*, 4th Ed. New Jersey: Prentice-Hall Inc. 1996.
2. Mano, Morris M. *Computer System Architecture*, 3rd Ed. New York: Pearson Education Inc. 1993.

APPENDIX

helpers.py

```python
def nBitPadSigned(str,n):
    if (str[0] == '0'):
        a = '0'*(n-len(str))+str
    elif (str[0] == '1'):
        a = '1'*(n-len(str))+str
    return(a)

def nBitPadUnsigned(str,n):
    a = '0'*(n-len(str))+str
    return(a)

def reverse(str):
    return(str[::-1])

def shiftRight(str, msb = '0'):
    a = msb + str[:-1]
    return a

def shiftLeft(str, lsb = '0'):
    a = str[1:] + lsb
    return a

def NOT(x):
    if x == '1':
        return'0'
    return '1'

def AND(x,y):
    if (x == '1' and y=='1'):
        return'1'
    return'0'

def OR(x,y):
    if (x == '1' or y =='1'):
        return'1'
    return'0'

def XOR(x,y):
    if x != y:
        return'1'
    else:
        return'0'
```

lab1.py

```python
# addition and subtraction

from helpers import nBitPadSigned;
from helpers import reverse;
from helpers import XOR,OR,NOT,AND;

def FullAdder(x,y,cIn):
    s = XOR(XOR(x,y),cIn)
    c = OR(OR(AND(x,y),AND(x,cIn)),AND(y,cIn))
    return s,c

def FullSubtractor(x,y,bIn):
    s = XOR(XOR(x,y),bIn)
    b = OR(OR(AND(NOT(x),y),AND(NOT(x),bIn)),
        AND(y,bIn))
    return s,b


def add(x,y,n = 8):
    a = reverse(nBitPadSigned(x, n))
    b = reverse(nBitPadSigned(y, n))
    c = '0'
    r = nBitPadSigned('0', n)
    for i in range(0,n):
        s,c = FullAdder(a[i], b[i], c)
        r = r[:i] + s + r[i+1:]
    r = reverse(nBitPadSigned(r, n))

    return(r,c)

def subtract(x,y,n=8):
    a = reverse(nBitPadSigned(x, n))
    b = reverse(nBitPadSigned(y, n))
    brw = '0'
    r = nBitPadSigned('0', n)
    for i in range(0,n):
        diff,brw = FullSubtractor(a[i], b[i],
            brw)
        r = r[:i] + diff + r[i+1:]
    r = reverse(nBitPadSigned(r, n))
    return(r,brw)
```