**1. INTRODUCTION**

Adders are the most fundamental part a microprocessor and many digital signal processing chips. Adders are essential not only for addition, but also for subtraction, multiplication, and division. Addition is one of the basic arithmetic operation. A fast and accurate operation of a digital system is greatly influenced by the performance of the resident adders.

1. 1. **Brief History**

Addition, subtraction, multiplication and division. All these operations are most fundamental part of the today's computational world. Man kind has evolved from counting to computers through series of changes. Not just in the present, if we go way back thousands of years, we find some sort of method used to accomplish the required calculations. Initially hand was the tool used to calculate, which then evolved into tallies made by carving notches in wood, bone, and stone. Well obviously the way humans evolved, so as their need for processing huge data which gave birth to many algorithms to perform required task with many calculation tools. Let us see some of the tools that were used which lead to the development of present day computing.

**The abacus**, probably invented in Sumer between 2700 and 2300 BC, is an early computing device invented by humans. Whilst it has no automatically moving parts or gears the movement of its beads on its structured rods provides a powerful mechanism for carrying out all four arithmetic operations. As late as the middle of the last century the speed of these on the abacus could rival or exceed the speed of the same computations performed on the latest electric calculating machines. Over centuries, support for arithmetic was provided either by the abacus, or other counting technologies (such as the counting board and calculi and "the pen and jeton"). In 1617 John Napier built on a known method of "lattice multiplication" to publish his "rods" or **Napier's Bones** which, appearing in a range of forms over subsequent centuries, were found to considerably facilitate the performance of multiplication and division, especially by those not well equipped with memorised multiplication tables.

After the era of manually operated devices came the era of mechanical devices .

### 1.1. Pascal's calculator

Blaise Pascal along with Wilhelm Schickard was one of two inventors of the mechanical calculator in the early 17th Century. Pascal made his invention in 1642. He was spurred to it when participating in the burden of arithmetical labor involved in his father's official work as supervisor of taxes at Rouen. First called the Arithmetic Machine, Pascal's Calculator and later Pascaline, his invention was primarily intended as an adding machine which could add and subtract two numbers directly, but its description could, with a bit of a stretch, be extended to a "mechanical calculator, in that at least in principle it was possible, admittedly rather laboriously, to multiply and divide by repetition.

Pascal designed the first mechanical calculator to still survive from the 17th century. His device was particularly successful in the smooth working of the so-called "carry mechanism" - the mechanism which allows an addition of 1 to 9 on one dial to replace the 9 with a 0, and carry 1 to the next dial. His innovation made each digit independent of the state of the others allowing for multiple carries to rapidly cascade from one digit to another regardless of the capacity of the machine.

### 1.2. The Engines

Charles Babbage (1791-1871), computer pioneer, designed two classes of engine, Difference Engines, and Analytical Engines. Difference engines are so called because of the mathematical principle on which they are based, namely, the method of finite differences. The beauty of the method is that it uses only arithmetical addition and removes the need for multiplication and division which are more difficult to implement mechanically.

Difference engines are strictly calculators. They crunch numbers the only way they know how by repeated addition according to the method of finite differences. They cannot be used for general arithmetical calculation. The Analytical Engine is much more than a calculator and marks the progression from the mechanized arithmetic of calculation to fully-fledged general-purpose computation. There were at least three designs at different stages of the evolution of his ideas. So it is strictly correct to refer to the Analytical Engines in the plural.

Well all of what we have today ie., the computers and many more computational devices evolved from what we have seen above to tackle the problem of computation, which is basically additions in the lowest level.



**ABACUS**

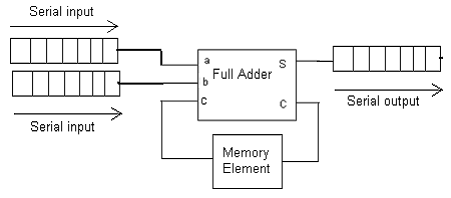
**2. PROBLEM STATEMENT**

“**Addition of two large numbers**”- We know how to add two integers using a perfectly simple and useful algorithm learned from school or even earlier. This is perhaps one of the very first techniques we learn in mathematics.

### What about big integers?

Since we know how to add “small” integers, its not so hard to apply the same algorithm on big integers. The only problem is that the addition will be slower and sometimes (done by humans) can be error prone.

So practically the algorithm is the same, but we can’t just put a 1 million integer into a standard computer type INT, right? That means, the question here is how we represent/store/execute such “big integers” in our application. A common solution is to store the “big integer” into an array, but in this problem we store the “big integer” in a “CIRCULAR LINKED LIST” which makes the traversing much more easier and helps us to implement the serial ripple carry adder.

****

**The Block Diagram Of A Serial Adder**

**3.INPUT**

Two large positive numbers.

Example.1:

* + 1. 1234567890123
    2. 1234567890237

Example.2:

1. 1872639812537659712537612386547826394512753
2. 1087263815237965213976421397148972153523795

Example.3.

1. 18726398125376597125376123865478263945127536750869857579
2. 1087263815237965213976421397148972153523795060876876795

4**. OUTPUT**

Sum of two large positive numbers.

Sum for Example.1. = 2469137580360

Sum for Example.2. = 2959903627775624926514033783696798548036548

Sum for Example.3. = 19813661940614562339352545262627236098651331811746734374

**5. ALGORITHM**

Even though we based our project on making a serial adder using linked list, we tried to tackle the problem using a fast adder too so that, we would get a better picture in comparing the working of these two models.

**5.1.ALGORITHM-Serial Adder**

Algorithm serialAdder(a[1,2,..,n], b[1,2,...m])

//Algorithm to compute the sum of two large positive numbers

//INPUT: two large positive numbers of size n and m, stored in the form of a linked list.

//OUTPUT: sum of numbers stored in the linked list.

carry = 0

for i = 1 to min(n , m) do //till we reach end of a number

s[i] = (a[i]+b[i]+carry) % 10

carry = (a[i]+b[i]+carry) /10

if(n<m) then // a has lesser number of digits

for i = n to m do

s[i] = (b[i]+carry) % 10

carry = (b[i]+carry) / 10

otherwise //b has lesser number of digits

for i = m to n do

s[i] = (a[i]+carry) % 10

carry = (a[i]+carry) / 10

Here a[i], b[i], s[i] are the ith elements of the linked list.

**5.2.ALGORITHM-Fast Adder**

Algorithm fastAdder(a[1,2,..,n], b[1,2,...m])

//Algorithm to compute the sum of two large positive numbers

//INPUT: two large positive numbers of size n and m, stored in the form of an array.

//OUTPUT: sum of numbers stored in the array.

Calculate all Gi's and Pi's parallely where,

G[i] = (a[i] + b[i])/10

P[i] = (a[i] + b[i])%10

Calculate all Ci's parallely using the carry look ahead algorithms

C[i] = clal(i,G,P)

Calculate all Si's parallely where,

S[i] = (P[i] + C[i])%10

Algorithm clal ( i,G,P)

//Algorithm computes the carry produced for ith place

if(i==0)

return 0;

if(G[i-1] == 1)

return 1;

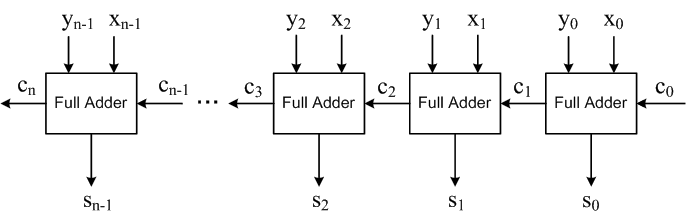
else if(P[i-1]==9)

return clal(i-1,g,p);

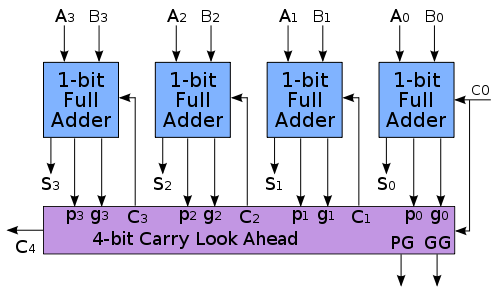
else

return 0;

**Serial Adder Algorithm (pictorial)**



**Carry Look Ahead Adder Algorithm(pictorial)**



**6. IMPLEMENTATION**

**Programming languages used are C and Python.**

**6.1. Configuration for hardware and software**

It doesn't need any additional hardware or software to operate the program, but the following requirements should be strongly maintained:

**6.1.1**. **Requirements for hardware:**

1. Pentium II or higher.

2. 512MB of RAM or higher.

3. 800MHz processor or above.

4. 20 MB of hard disk space.

**6.1.2. Requirements for software:**

1. Operating System LINUX or WINDOWS.

2. GCC compiler and Python 2.\* or higher interpreter with pygtk.

**6.2. About Program Logic:**

Although adders can be constructed for many numerical representations, such as binary-coded decimal or excess-3, the most common adders operate on binary numbers. But the adder we are implementing is for the decimal numbers. We have implemented it using two models.

1. The Ripple Carry Adder (using circular linked list)
2. The Carry Look Ahead Adder (using arrays)

**6.2.1. Ripple Carry Adder**

The two input numbers are stored in a circular linked list. The number is traversed from right to left adding them bit by bit. A variable carry is set if the sum ((sum)A = A + B + Carry) is greater than 10 or else it is 0. Sum modulo 10 is taken to get the sum bit . In this way, the whole list is traversed with carry rippling to the next bit. The final answer ie the sum is stored in one of the input list.

**6.2.2. Carry Look Ahead Adder**

To reduce the computation time, engineers devised faster ways to add two binary numbers by using carry-look ahead adders. They work by creating two signals (P and G) for each bit position, based on whether a carry is propagated through from a less significant bit position (at least one input is a '1'), generated in that bit position (both inputs are '1'), or killed in that bit position (both inputs are '0'). In most cases, P is simply the sum output of a half-adder and G is the carry output of the same adder. After P and G are generated the carries for every bit position are created.

A carry-lookahead adder improves speed by reducing the amount of time required to determine carry bits.

The same logic has been implemented using C on decimal numbers.

Here, **G = (A + B)/10** and **P = (A+B)%10**..

Carry is generated simultaneously using the recurrence relation

**C[i] = G[i] | ((P[i] == 9) ? C[i-1] : 0 where, i>=1.**

**//Implements serial ripple carry adder using circular linked list**

**//Authors:Mussadiq Faraz**

**Sameer Shaik**

**Shashi Kiran Reddy**

**Sujay**

**Program 1: Serial adder**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

**//Defines the structure number which holds the info of a single bit**

typedef struct node

{

unsigned int data;

struct node\* link;

}number;

**//allocates memory for a single bit of a number**

number\* getnode()

{

return (number\*)malloc(sizeof(number));

}

**//Creates the circular linked list using the given input**

void create(number\*\* num,char\* str)

{

int temp,i;

number \*temp1;

temp1=getnode();

temp1->data = strlen(str);

\*num=temp1;

temp1->link=NULL;

for(i=strlen(str)-1;i>=0;i--,temp1=temp1->link)

{

temp1->link=getnode();

temp=str[i]-'0';

temp1->link->data=temp;

temp1->link->link=NULL;

}

for(temp1=\*num;temp1->link;temp1=temp1->link);

temp1->link=\*num;

}

**//Terminates the program if the given inputs are wrong**

inline void terminate()

{

printf("incorrect arguments\nPROGRAM TERMINATED\n");

exit(0);

}

**//Displays the large integer num**

void display(number\* num)

{

number\* temp;

char \*str = (char\*) malloc (sizeof(char)\*(num->data));

int i;

for(temp=num->link,i=0;temp!=num;temp=temp->link,i++)

str[i]=(char)(temp->data+'0');

str[i]='\0';

for(i=strlen(str)-1;i>=0;i--)

printf("%c",str[i]);

printf("\n");

}

**// The funtion adds the two numbers in linked list pointed by sum and by num. stores the result in the linked list pointed by sum.**

void addnext(number\* sum, number\* num)

{

number \*temp1,\*temp2;

int temp,carry=0;

for(temp1=sum,temp2=num;temp1->link!=sum&&temp2->link! =num;temp1=temp1->link,temp2=temp2->link)

{

temp=temp1->link->data+temp2->link->data+carry;

if(temp>=10)

carry=1;

else

carry=0;

temp1->link->data=temp%10;

}

if(temp1->link==sum)

{

while(temp2->link!=num)

{

temp1->link=getnode();

temp1=temp1->link;

temp2=temp2->link;

temp=temp2->data+carry;

if(temp>=10)

carry=1;

else

carry=0;

temp1->data=temp%10;

}

if(carry)

{

temp1->link=getnode();

temp1=temp1->link;

temp1->data=1;

}

temp1->link=sum;

}

else

{

while(temp1->link!=sum && carry)

{

temp1=temp1->link;

temp=temp1->data+carry;

if(temp>=10)

carry=1;

else

carry=0;

temp1->data=temp%10;

}

if(carry)

{

temp1->link=getnode();

temp1=temp1->link;

temp1->data=1;

temp1->link=sum;

}

}

}

**//The main function manages all the working of the program. Gets the input numbers from the command line.**

void main(int argc,char\* argv[])

{

int i,max;

struct timeval t;

double start,end;

number \*summands,\*sum;

if(!(argc-1))

terminate();

create(&sum,argv[1]);

create(&summands,argv[2]);

gettimeofday(&t,NULL);

start=t.tv\_sec+(t.tv\_usec/1000000.0);

addnext(sum,summands);

gettimeofday(&t,NULL);

end=t.tv\_sec+(t.tv\_usec/1000000.0);

display(sum);

printf("%lf time\n",end-start);

}

***//*Implements carry look ahead adder**

**//Authors:Mussadiq Faraz**

**Sameer Shaik**

**Shashi Kiran Reddy**

**Sujay**

**Program 2: Fast Adder**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<omp.h>

#include<sys/time.h>

**//Terminates the program if the given inputs are wrong**

inline void terminate()

{

printf("incorrect arguments\nPROGRAM TERMINATED\n");

exit(0);

}

**//Creates the array to be computed based on the given input**

void create(short int \*a, char c[], int len)

{

int i;

for(i=len;i>0;i--)

{

a[i]=(short int)c[len-i] - '0';

if(a[i]>9 || a[i]<0)

terminate(); **//If the given inputs are not numbers** **, terminate**

}

a[0]=len;

}

**//Displays the large integer a**

void display(short int\* a)

{

int i=0;

for(i=a[0];i>0;i--)

printf("%d",a[i]);

printf("\n");

}

**//The below function computes carry using G and P for the i th bit**

short int clal(int i, short int g[], short int p[])

{

if(i==0)

return 0;

if(g[i-1])

return 1;

else if(p[i-1]==9)

return clal(i-1,g,p);

else

return 0;

}

**// The funtion adds the two numbers array a and array b and stores the result in array a**

void addnext(short int\* a,const short int\* b)

{

int max,min,i;

max = (a[0]>b[0])?a[0]:b[0];

min = (a[0]<b[0])?a[0]:b[0];

short int \*g = (short int\*) malloc (sizeof(short int)\*max);

short int \*p = (short int\*) malloc (sizeof(short int)\*max);

short int \*c = (short int\*) malloc (sizeof(short int)\*(max+1));

**//computes the generator G and Propagator P**

for(i=1;i<=min;i++)

{

g[i-1] = (a[i]+b[i])/10;

p[i-1] = (a[i]+b[i])%10;

}

if(a[0]==min)

for(i=min+1;i<=max;i++)

{

g[i-1] = 0;

p[i-1] = b[i];

}

else

for(i=min+1;i<=max;i++)

{

g[i-1] = 0;

p[i-1] = a[i];

}

**//Computes carry for all the bits simultaneously**

#pragma omp parallel for

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

c[i]=clal(i,g,p);

**//Computes sum for all the bits simultaneously**

#pragma omp parallel for

for(i=1;i<=max;i++)

a[i]=(p[i-1]+c[i-1])%10;

#pragma omp barrier

**//The last bit is 1 if carry is produced, otherwise it is 0**

if(c[i-1])

{

a[i]=1;

a[0]=i;

}

else

a[0]=i-1;

}

**//The main function, accepts the numbers through the command line**

int main(int argc, char\* argv[])

{

short int \*summand,\*sum;

int i,max;

struct timeval t;

double start,end;

if(!(argc-1))

terminate();

max=strlen(argv[1]);

for(i=2;i<argc;i++)

if(strlen(argv[i])>max)

max=strlen(argv[i]);

sum = (short int\*) malloc(sizeof(short int)\*(max+5));

create(sum,argv[1],strlen(argv[1]));

summand=(short int\*) malloc(sizeof(short int)\*strlen(argv[2]+1));

create(summand,argv[2],strlen(argv[2]));

gettimeofday(&t,NULL);

start=t.tv\_sec+(t.tv\_usec/1000000.0);

addnext(sum,summand);

gettimeofday(&t,NULL);

end=t.tv\_sec+(t.tv\_usec/1000000.0);

free(summand);

display(sum);

free(sum);

printf("%lf time\n",end-start);

return 0;

}

**#Program 3: Python script for GUI**

#!/usr/bin/python

from gi.repository import Gtk

import os

import serial\_graphics

**#Program to give GUI for the Project ADDER Authors: Musaddiq Faraz, Sameer Shaik, Shashi Kiran Reddy, Sujay**

class GUI(Gtk.Window):

**‘’’Provides the main window for the GUI’’’**

def toggle\_graphics(self,check):

if (self.show1 == 1):

self.show1 = 0

self.entry\_3.hide()

self.entry\_4.hide()

self.button2.hide()

self.label\_3.hide()

else:

self.show1=1

self.entry\_3.show()

self.entry\_4.show()

self.label\_3.show()

self.button2.show()

def \_\_init\_\_(self):

**#the constructor of the GUI Class**

Gtk.Window.\_\_init\_\_(self, title="ADDER")

self.set\_position(Gtk.WindowPosition.CENTER)

self.set\_border\_width(15)

self.entry\_text\_1 = "null"

self.entry\_text\_2 = "null"

table = Gtk.Table(10, 10, False)

self.add(table)

self.label = Gtk.Label("Answer : " + answer)

button = Gtk.Button("ADD")

button.connect("clicked", self.button\_pressed)

label\_app = Gtk.Label("Application Adder ")

label\_1 = Gtk.Label("Enter number1: ")

entry\_1 = Gtk.Entry()

entry\_1.connect("changed", self.enter\_callback\_1, entry\_1)

label\_2 = Gtk.Label("Enter number2: ")

entry\_2 = Gtk.Entry()

entry\_2.connect("changed", self.enter\_callback\_2, entry\_2)

self.label\_3 = Gtk.Label("Enter 4-bit Number ")

self.entry\_3 = Gtk.Entry()

self.entry\_4 = Gtk.Entry()

self.entry\_3.set\_max\_length(4)

self.entry\_4.set\_max\_length(4)

self.show1 = 1

self.button2 = Gtk.Button("Simulate")

self.button2.connect("clicked", self.button2\_pressed)

check = Gtk.CheckButton("4-bit Graphical Adder")

check.set\_active(True)

check.connect("toggled", self.toggle\_graphics)

check.show()

about = Gtk.Button("About")

about.connect("clicked", self.aboutdig)

table.attach(label\_app,1,10,1,2)

table.attach(label\_1,1,2,2,3)

table.attach(entry\_1,2,10,2,3)

table.attach(label\_2,1,2,3,4)

table.attach(entry\_2,2,10,3,4)

table.attach(self.label, 1, 10, 5, 6)

table.attach(button, 2, 10, 4, 5)

table.attach(about, 1, 2, 4, 5)

table.attach(check,1,10,6,7)

table.attach(self.label\_3,1,3,7,8)

table.attach(self.entry\_3,3,5,7,8)

table.attach(self.entry\_4,5,7,7,8)

table.attach(self.button2,7,10,7,8)

def aboutdig(self, widget):

**#defines the about dailog box which opens on clicking the about button**

message="Program to simulate adder...\nAuthors: Musaddiq Faraz, Sameer Shaik, Shashi Kiran Reddy, Sujay"

dialog = Gtk.MessageDialog(self, 0, Gtk.MessageType.INFO,Gtk.ButtonsType.OK, "About the program")

dialog.format\_secondary\_markup(message)

dialog.run()

dialog.destroy()

def button2\_pressed(self, button):

**#executes on clicking the simulate button**

x = self.entry\_3.get\_text()

y = self.entry\_4.get\_text()

c = ""

temp1 = int(x)

temp2 = int(y)

s = str(temp1 + temp2)

while temp1!=0 and temp2!=0 :

if (temp1%10+temp2%10 > 9):

c = c + '1'

else:

c = c + '0'

temp1 = temp1/10

temp2 = temp2/10

x = correct(x[::-1])

y = correct(y[::-1])

c = correct(c[::-1])

s = correct(s[::-1])

c = '0' + c

serial\_graphics.main(x,y,c,s)

def button\_pressed(self, button):

**#executes on clicking the a button**

os.system("gcc main.c -o add")

if (os.name == "posix"):

command="./add "+self.entry\_text\_1+" "+self.entry\_text\_2+" >ans"

else:

command="add "+self.entry\_text\_1+" "+self.entry\_text\_2+" >ans"

os.system(command)

fobj = open("ans","r")

answer = fobj.readline()

answer = answer[:len(answer)-1]

fobj.close()

os.system("rm add")

self.label.set\_text("Sum = "+answer)

def enter\_callback\_1(self, widget, entry\_1):

**#called when the entry box1 is edited**

self.entry\_text\_1 = entry\_1.get\_text()

return

def enter\_callback\_2(self, widget, entry\_2):

**#called when the entry box1 is edited**

self.entry\_text\_2 = entry\_2.get\_text()

return

def correct(str1):

**#Correcting the number of digits entered to 4, for simulation**

len1=len(str1)

while len1<4:

str1 = str1 + '0'

len1+=1

return str1

entry\_text\_1="null"

answer="null"

win = GUI()

win.connect("delete-event", Gtk.main\_quit)

win.show\_all()

Gtk.main()

**Program 4: Python script for simulating the working of serial ripple carry adder**

#!/usr/bin/python

**'''Program for Graphical simulation of 4 bit serial ripple carry adder**

**Authors: Musaddiq Faraz, Sameer Shaik, Shashi Kiran Reddy, Sujay'''**

from graphics import \*

import time

class Element:

**'''Provides the outline for single full adder block'''**

def \_\_init\_\_(self,point,size):

**#constructor for the Element class**

pt1 = point.clone()

pt2 = point.clone()

pt1.move(size/2,size/2)

pt2.move(-size/2,-size/2)

self.rect = Rectangle(pt2,pt1)

self.rect.setOutline('blue')

self.rect.setFill('red')

pt4 = point.clone()

self.label = Text(pt4,'Full Adder')

pt1 = point.clone()

pt1.move(size/4,-3\*size/4)

pt4 = pt1.clone()

pt4.move(0,-size/8)

self.label1 = Text(pt4,'x')

pt2 = pt1.clone()

pt2.move(0,size/4)

self.line1 = Line(pt1,pt2)

self.line1.setArrow('last')

pt1.move(-size/2,0)

pt2 = pt1.clone()

pt2.move(0,size/4)

pt4 = pt1.clone()

pt4.move(0,-size/8)

self.label2 = Text(pt4,'y')

self.line2 = Line(pt1,pt2)

self.line2.setArrow('last')

pt1 = point.clone()

pt1.move(-size/2,0)

pt2 = pt1.clone()

pt2.move(-size/4,0)

pt4 = pt1.clone()

pt4.move(-size/8,-size/8)

self.label3 = Text(pt4,'co')

self.line3 = Line(pt1,pt2)

self.line3.setArrow('last')

pt1.move(5\*size/4,0)

pt2 = pt1.clone()

pt2.move(-size/4,0)

pt4 = pt1.clone()

pt4.move(-size/8,-size/8)

self.label4 = Text(pt4,'ci')

self.line4 = Line(pt1,pt2)

self.line4.setArrow('last')

pt1 = point.clone()

pt4 = pt1.clone()

pt4.move(0,5\*size/6)

self.label5 = Text(pt4,'sum = ?')

pt1.move(0,size/2)

pt2 = pt1.clone()

pt2.move(0,size/4)

self.line5 = Line(pt1,pt2)

self.line5.setArrow('last')

def set\_text(self,ci,co,x,y,s):

**#changes the texts that has to be changed during animation**

self.label1.setText(x)

self.label2.setText(y)

self.label3.setText(co)

self.label4.setText(ci)

self.label5.setText(s)

def display\_ele(self,win):

**#Displays the single block of adder**

self.rect.draw(win)

self.line1.draw(win)

self.line2.draw(win)

self.line3.draw(win)

self.line4.draw(win)

self.line5.draw(win)

self.label.draw(win)

self.label1.draw(win)

self.label2.draw(win)

self.label3.draw(win)

self.label4.draw(win)

self.label5.draw(win)

def set\_color(self):

**#changes the color of the block, called during animation**

self.rect.setFill('green')

def main(x,y,c,s):

**#the interface for getting data and managing it**

win1 = GraphWin("4-bit DECIMAL SERIAL ADDER",1000,400)

pt = Point(500,50)

label = Text(pt,'Simple 4-bit Decimal Serial Adder')

label.setStyle('bold italic')

label.setSize(20)

label.setTextColor('blue')

label.draw(win1)

elements = []

create(elements)

display(win1,elements)

win1.getMouse()

animate(x,y,c,s,elements,win1)

answer = Text(Point(500,350),x[::-1]+" + "+y[::-1]+" = "+s[::-1])

answer.setSize(20)

answer.setTextColor('blue')

answer.draw(win1)

win1.getMouse()

win1.close()

def create(elements):

**#creates the 4 bit adder**

size = 120

point = Point(1000 - 3\*size/2,200)

i=0

for i in range(0,4):

e = Element(point,size)

elements.append(e)

point.move(- 7\*size/4,0)

def animate(x,y,c,s,elements,win):

**#animates**

line = elements[0].line4.clone()

i=0

line.move(30,0)

line.draw(win)

line.setOutline('green')

time.sleep(.01)

for i in range(0,4):

x1 = line.getP2().getX()

x2 = elements[i].line4.getP2().getX()

while x1!=x2:

x1-=1

line.move(-1,0)

time.sleep(.01)

line.setOutline('black')

elements[i].set\_text(c[i],c[i+1],x[i],y[i],s[i])

elements[i].set\_color()

time.sleep(.15)

line = elements[i].line3.clone()

line.setOutline('green')

line.draw(win)

def display(win,elements):

**#displays the adders**

i = 0

for i in range(0,4):

elements[i].display\_ele(win)

**7**. **Testing**

While testing for the time complexity of our program (The Carry Look Ahead Adder) we had swapped from dynamic initialization to static initialization so as to see the effect of large malloc on the time complexity. After knowing that the difference of the time between static and dynamic initialization (which was almost nil) we again swapped our program back to dynamic initializations.

During the above process we had produced a bug which didn't seem familiar to us. It was a run time error stating

**\*\*\*\* stack smashing detected \*\*\*: ./a.out terminated\***

Briefly, stack smashing occurs when a function allocates a static array on the stack and writes past the end of it, onto other local variables and eventually onto other function stack frames. When it comes time to return from the function, the return address has been corrupted and the program ends up some place it really shouldn’t. In the best case, the program just crashes; in the worst case, a malicious party crafts code to exploit this malfunction.

We found that the above error is due to one of the methods of Buffer overflow protection.

The compiler we used for compiling our C programs is GCC. GCC comes with many security measures and SSP(stack smashing protection) being one of it.

The program after debugging has been tested with lots of varieties of data sets and it has executed all of them without any garbage result.

**8. Discussion**

Our project was initially about making adder that adds any large integers using the circular linked list. Basically in the basic level implementation these adders are implemented using the shift registers which can exactly be implemented using the circular linked list.

In our discussion about ripple-carry adders, we said that adding two n-bit numbers requires O(n) time. The reason is the carry. As you perform the addition on the least significant bits, you may have a carry that "ripples" its way to the most significant bit. If we didn't have a carry, then we should be able to add in O(1), because adders can work in parallel.

Since we already had learnt about the carry look ahead adders we wanted to implement it in decimal so as to compare the working of both and its time complexities.

The carry look ahead logic requires a special hardware support so as to allow it to compute the carries simultaneously.

The implementation of the carry look ahead adder uses parallel programming which is achieved using the **openmp** library for C.

The carry generation is done using a recursicve relation..

There are two functions generated. G and P. They are computed as follows,

**Gi = (Xi + Yi) / 10**

**Pi = (Xi + Yi) % 10**

Using the above two values the carry and sum is produced.

**Ci = Gi | (Pi > 9) ? Ci-1 : 0 and C0 = 0**

**Si = (Pi + Ci) % 10**

By using these formulas, we can cut down the adders from O(n) to what appears to be O(1) time. After all, pi and gi only depend on xi and yi, which are the bits of x and y which are immediately available to us (Parallel computation) . They also only depend on c0, which is also immediately available. We don't have to wait for carries to perform this computation.

Still, it's not quite O(1). Why not?

After achieving the parallelism using OpenMP we understood many cons of software level parallel programming. The time complexity was hugely affected after the use of parallel programming but it was the opposite extreme to what we expected. When we compared the time complexity with various set of values for n(number of digits) for the serial and fast adder, we found that the serial adder was way faster initially( in the order of 50). We were confused initially but made the following conclusion after analising the working of parallel programming:

1. The amount of time taken to create and synchronize the thread had a lot of effect on the time complexity.
2. The actual number of threads that works simultaneously depends on the system architecture, (in our case no of threads = 4, quad core processor). Even though we are able to calculate all the carries in parallel the hardware limits the number of parallel computations.
3. Since the data is shared between the threads, shared memory model is used, which is again an overhead for the computation.

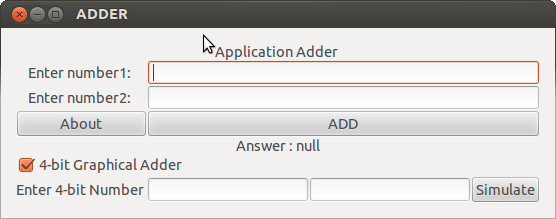
The theoretical value of the time complexity of the serial adder is O(n) and pratically too, it behaved the same way.

The theoretical value of the time complexity of the carry look ahead adder is O(log(n)), but practically it was k1 \* O(log(n)) + k2 , where k2 is very large(overhead of creating and managing thread). But it can be said that on really huge data the time complexity can approach its theoretical value.

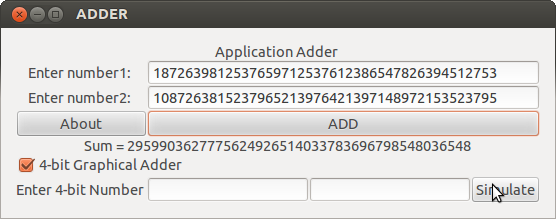
**9. USER MANUAL**

* To execute the program open the terminal and navigate to the folder containing the program.
* Run the python script **run.py**. It compiles and executes the program and gives a graphical user interface to accept the inputs from the user.
* There is also a graphical representation of the above program. A 4-bit serial adder done by using python script with the aid of py graphics. It can be executed by giving 4 bit input and by pressing the simulate button.

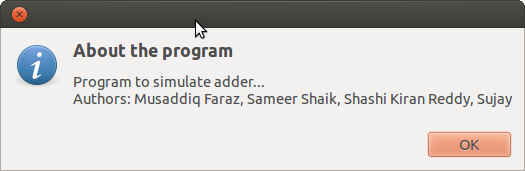
Executing the above script gives the below display:

****

Sample input and output:

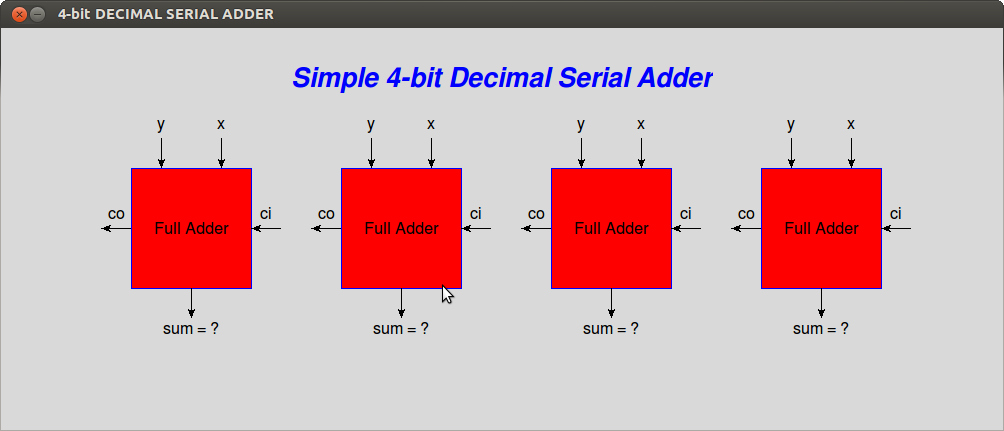


On clicking the about button

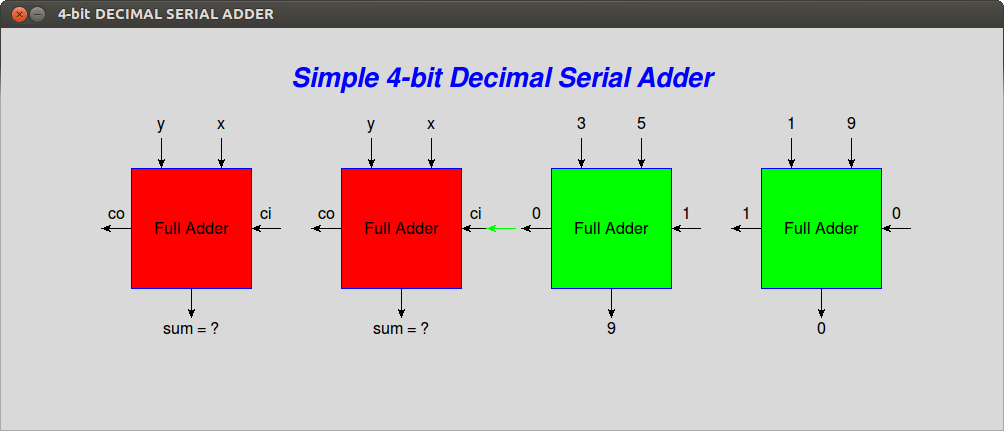
****

On clicking the simulate button with an appropriate input the following animation displays.

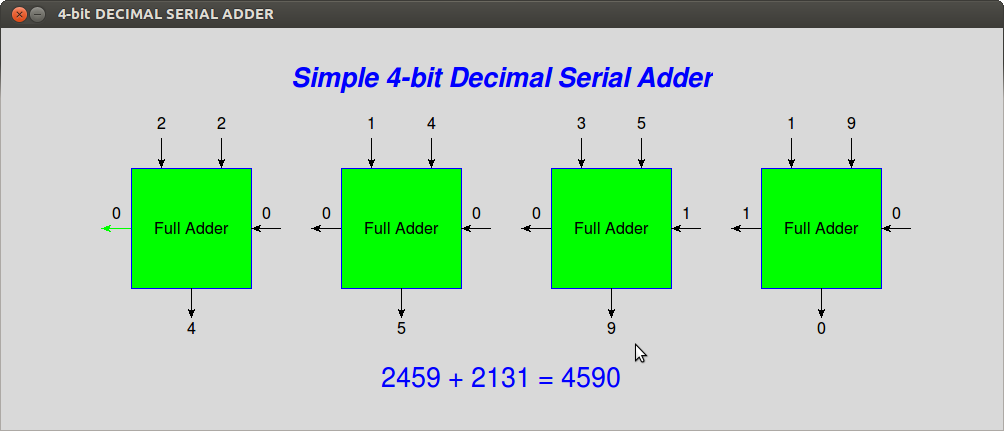
Animation 1 at t0:

****

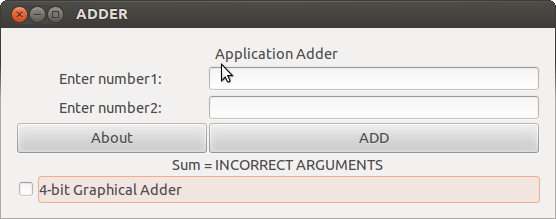
Animation 2 at t2:

****

Animation 3 at t4:

****

Toggle the check box 4-bit Graphical Adder to hide the 4 bit inputs

****

**10. CONCLUSION**

Pros:

1. The project successfully compares the working of a serial carry ripple adder and the carry look ahead adder.
2. The program serial adder successfully adds any large number with a time complexity of O(n) where n would be the number of digits in the number, which matches with the theoritical value.
3. The fastadder implementation ideally executes with the time complexity of O(log(n)), and here again n is the number of digits.

Cons:

1. The only con about the project would be failing to achieve the time complexity of O(log(n)) with the fast adder implementation.

**11. ACKNOWLEDGEMENT**

We would like to thank Dr.G.T.Raju H.O.D of Computer Science department of RNSIT for giving us the opportunity to build this project. We also extend our sincere thanks to Prof.Bhavanishankar for supervising our project and Prof.Sathish Kumar for giving us useful information regarding parallel programming. We would also like to thank Prof.M.J.Sudhamani for giving us an outstanding lecture on the topic adder which is the base of our project.

**12. Reference**

1. Computer Organization Lecture
2. Python Documentation
3. Wikipedia for history
4. StackOverflow for Technical Details