ELE301T - VLSI SYSTEM DESIGN

PROJECT REPORT

MEAN SHIFT CLUSTERING

- By :

Firoz Mohammad CED17I017

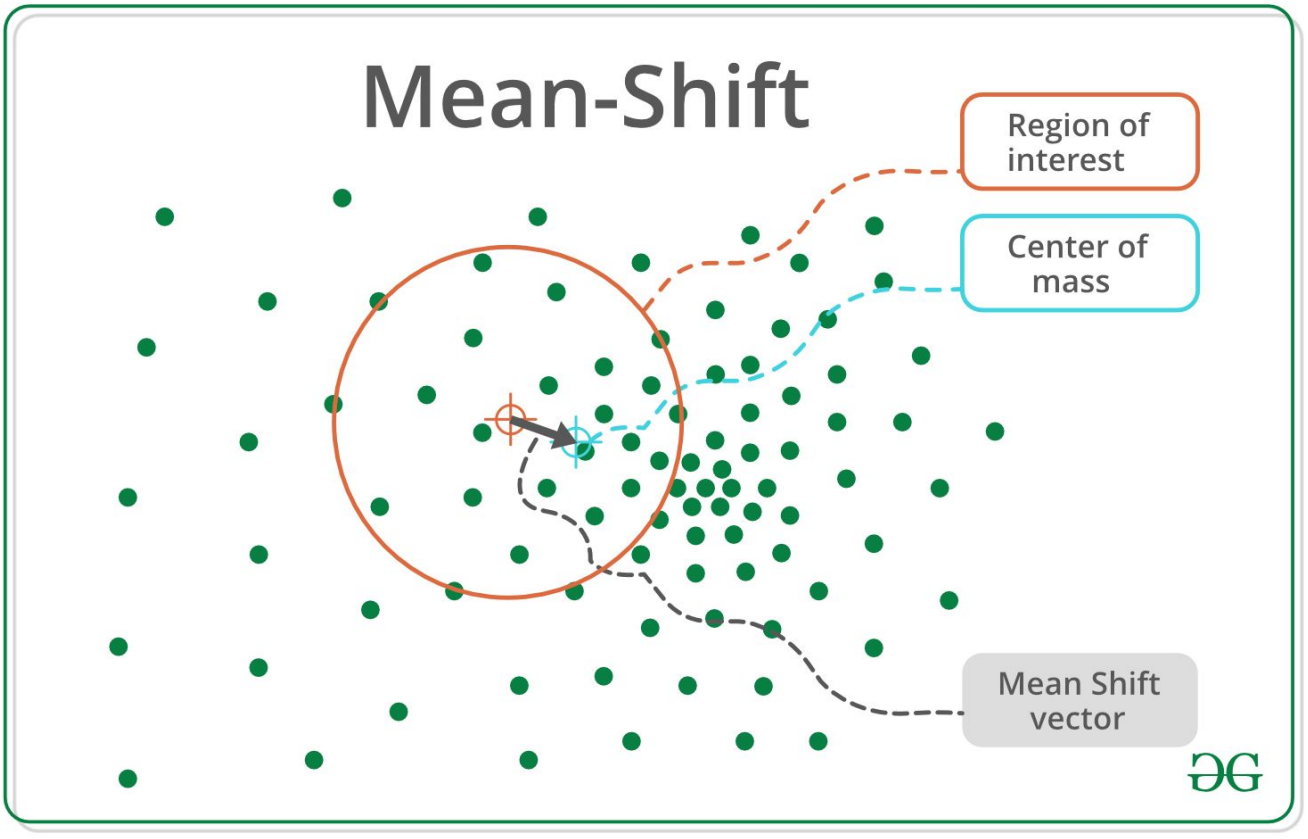
Vaibhav Singhal CED17I040

TABLE OF CONTENTS:

INTRODUCTION

Mean shift is a clustering algorithm that assigns the data points to the clusters iteratively by shifting points towards the mode. It is also known as the **Mode-seeking algorithm**. Mean-shift algorithm has applications in the field of image processing and computer vision.

Given a set of data points, the algorithm iteratively assigns each data point towards the closest cluster centroid and direction to the closest cluster centroid is determined by where most of the points nearby are at. So each iteration each data point will move closer to where the most points are at, which is or will lead to the cluster center. When the algorithm stops, each point is assigned to a cluster.



MEAN SHIFT ALGORITHM

INPUT: A set of data X={ X_1,X_2,X_3, ... ,X_n } where X_i belongs to R (having dimension >= 2) and bandwidth h or σ .

OUTPUT: A number of clusters .

1. Initialize : Set X to the value of the point to classify
2. Repeat :
    - Move X by the corresponding mean shift vector :

$$X \leftarrow X + M(X) = \frac{\sum_i X_i g\left(\frac{\|X - X_i\|^2}{h^2}\right)}{\sum_i g\left(\frac{\|X - X_i\|^2}{h^2}\right)}$$

    - Where g is gaussian kernel and (h or σ) is bandwidth
    - The Gaussian kernel is defined in 1-D, 2D and N-D respectively as :

$$G_{1D}(x;\sigma) = \frac{1}{\sqrt{2\pi}\,\sigma} e^{-\frac{x^2}{2\sigma^2}},\quad G_{2D}(x,y;\sigma) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}},\quad G_{ND}(\vec{x};\sigma) = \frac{1}{(\sqrt{2\pi}\,\sigma)^N} e^{-\frac{|\vec{x}|^2}{2\sigma^2}}$$

3. Until X converges (Note : convergence is guaranteed unless or until data is not uniform)

CODE EXPLANATION

```
def main():
    items = ReadData('data1.txt')
    items = CutToTwoFeatures(items,0,1)
    print(items)
    for item in items:        #loop for plotting the data points
        plt.scatter(item[0],item[1],c="gray",alpha=0.5,s=10)
        plt.pause(0.00001)
    meanShift(items) #fn call to meanShift() which calculates mean shift
    plt.show();
```

This is the main function which is called when we run the python file. The first line calls the function ReadData() which reads the data from the given input file. The next line calls the function CutToTwoFeatures() which gives only the first two dimensions of the data. This is just done for plotting and visualising purposes. This step can be removed as the algorithm works for any dimension data. Given below are the respective functions:

```
def ReadData(fileName):
    #Read the file, splitting by lines
    f = open(fileName,'r');
    liness = f.read().splitlines();
    f.close();
    items = [];

    for i in range(0,len(liness)):
        line = liness[i].split(',');
        itemFeatures = [];
        for j in range(len(line)):
            v = float(line[j]);     # Convert feature value to float
            itemFeatures.append(v);    # Add feature value to dict
        items.append(itemFeatures);

    shuffle(items);   # For shuffling the the data items
    return items;
```

```
def CutToTwoFeatures(items,indexA,indexB):
    n = len(items);
    X = [];
    for i in range(n):
        item = items[i];
        newItem = [item[indexA],item[indexB]];
        X.append(newItem);
    return X;
```

```
def meanShift(original_X):
    X = np.copy(original_X)
    past_X = [[0,0]]*len(X)

    for it in range(no_iter):
        for i in range(0,len(X)):       #for i, x in enumerate(X):
            print("i = "+str(i))
            runner(X,X[i],i)
            l_d=look_distance
            circle=plt.Circle((X[i][0],X[i][1]),radius=l_d,fill=false,color='b')
            plt.gcf().gca().add_artist(circle)
            circles.append(circle)
            plt.pause(0.0001)

        for c in circles:
            c.remove()
            plt.pause(0.0001)
        circles.clear()

        past_X=np.copy(X)
    unique=[]
    color=["red","blue","green","purple","yellow","orange","pink","cyan"]
    for i in range(len(X)):
        flag=0
        for j in range(len(unique)):
            if(euclid_distance(X[i],unique[j]) < 0.1 ):
                flag=1
                break
        if(flag==0):
            unique.append(X[i])
    print(unique)
    for i in range(len(X)):
        for j in range(len(unique)):
            if(euclid_distance(X[i],unique[j]) < 0.1):
                ind=j
    plt.scatter(original_X[i][0],original_X[i][1],c=color[ind],alpha=0.5,s=10)
    plt.pause(0.0001)
```

The meanShift() function implements the mean shift algorithm. It runs for a specified number of iterations and passes each point to the runner() function which assigns the new location of the point. After this we define a unique list which stores the unique means whose euclidean distance is greater than 0.1 . And using that list the points have been assigned colors and plotted in the last loop.

```
def runner(X,x,i):
    # Step 1. For each datapoint x E X, find the neighbouring points N(x) of x.
      neighbours = neighbourhood_points(X, x, look_distance)
    # Step 2. For each datapoint x E X, calculate the mean shift m(x).
      numerator = [0]*len(x)
      denominator = 0
      for neighbour in neighbours:
            distance = euclid_distance(neighbour, x)
            weight = gaussian_kernel(distance, kernel_bandwidth)
            for j in range(len(x)):
                  # numerator[j] += (weight * neighbour[j])
                  numerator[j]=add(numerator[j],multiplier(weight,neighbour[j]))
            denominator = add(denominator,weight)
            # denominator+=weight

      new_x=[0]*len(x)
      for j in range(len(x)):
            new_x[j] = numerator[j] / denominator
        # Step 3. For each datapoint x E X, update x <- m(x).
      X[i] = new_x
      print(new_x)
```

The runner() function calculates the new mean of the given point. It first calculates the neighbourhood points by calling the neighbourhood_points() function and stores it in a list called neighbours. Now each point in the neighbor is assigned with a weight by calling the gaussian_kernel() function. Now the weighted mean is calculated and assigned to every dimension of the given point. The new location of the point is stored in the respective array index and the value is printed.

```
def gaussian_kernel(distance, bandwidth):
      div=(distance / bandwidth)
    # div_sq=div*div
      div_sq=multiplier(div,div)
      # val = (1/(bandwidth*math.sqrt(2*math.pi))) * np.exp(-0.5*div_sq)
      m1=multiplier(2,math.pi)
      m2=multiplier(-0.5,div_sq)
      m3=multiplier(bandwidth,math.sqrt(m1))
      val = multiplier((1/m3),np.exp(m2))
      return val
```

The gaussian_kernel() function returns the weight of the points on the basis of their distance/proximity to the point from whom the mean is being calculated. The formula for calculating the weight based on gaussian (for n dimensions) is as follows:

$$G_{\mathrm{ND}}(\vec{x};\sigma) = \frac{1}{\left(\sqrt{2\pi}\,\sigma\right)^{N}} e^{-\frac{|x|^2}{2\sigma^2}}$$

```
def neighbourhood_points(X, x_centroid, distance):
    eligible_X = []
    for x in X:
        distance_between = euclid_distance(x, x_centroid)
        if distance_between <= distance:
            eligible_X.append(x)
    return eligible_X
```

The neighbourhood_points() function calculates and returns the list of points whose euclidean distance with the given point(x_centroid in this case) lies within the given distance. Such points are known as the neighbourhood points of the given point.

```
def euclid_distance(x,y):
      S = 0; #The sum of the squared differences of the elements
      for i in range(len(x)):
        # diff=x[i]-y[i]
            diff=add(x[i],-y[i])
        # S+=diff*diff
            S=add(S,multiplier(diff,diff))
      ret=math.sqrt(S)
      return ret; #The square root of the sum
```

The euclid_distance() function calculates the euclidean distance between two points which is given by the formula:

$$d(\mathbf{p},\mathbf{q}) = d(\mathbf{q},\mathbf{p}) = \sqrt{(q_1-p_1)^2+(q_2-p_2)^2+\cdots+(q_n-p_n)^2}$$

$$= \sqrt{\sum_{i=1}^{n}(q_i-p_i)^2}.$$

```
def add(a_int,b_int):
      a_int=int(a_int*1024)
      b_int=int(b_int*1024)
      a=np.binary_repr(a_int, width=64)
      b=np.binary_repr(b_int, width=64)
      cmd="./adder +a="+str(a)+" +b="+str(b)+" > out.txt"
      os.system(cmd)
      sum=ReadOutput()
      sum=Bits(bin=sum)
      return(round((sum.int/1024),3))
```

The add() function does the work of passing its formal parameters to the 64 bit full adder's testbench verilog module for addition/subtraction of those two arguments and fetching the output from the file and returning the calculated value back to the calling function.

The first and the second lines of the add() function multiply the given arguments by 1024= 2^{10} which is equivalent to left shifting the numbers by 10 bits. As 2^{10} = 1024 ≈ 1000 = 10^3 , this gives us a precision of three decimal places. By doing int(a_int*1024) we drop rest of the decimal places. We now convert these two numbers into their respective binary representations, by giving the width as 64 the function np.binary_repr() automatically converts the numbers into 2s complement if they are negative. Now the binary representations of the two numbers are passed as arguments(or rather plus arguments) to the iverilog compiled binary file of the 64 bit full adder's testbench and the output to the addition is stored in the binary format in a file called output.txt .The output is read from the file and converted back from binary to base 10 number. The sum is now right shifted by 10 bits (in binary) and rounded up to 3 decimal places and the result is returned back to the calling function.

```
def multiplier(a_int,b_int):
      a_int=int(a_int*1024)
      b_int=int(b_int*1024)
      if(a_int<0 and b_int<0):
            a=-a_int
            b=-b_int
            cmd="./multiplier +a="+str(a)+" +b="+str(b)+" > out.txt"
            os.system(cmd)
            prod=ReadOutput()
            prod=int(prod)
      elif(a_int<0 and b_int>0):
            a=-a_int
            b=b_int
            cmd="./multiplier +a="+str(a)+" +b="+str(b)+" > out.txt"
            os.system(cmd)
```

```
            prod=ReadOutput()
            prod=-1*int(prod)
        elif(a_int>0 and b_int<0):
            a=a_int
            b=-b_int
            cmd="./multiplier +a="+str(a)+" +b="+str(b)+" > out.txt"
            os.system(cmd)
            prod=ReadOutput()
            prod=-1*int(prod)
        else:
            a=a_int
            b=b_int
            cmd="./multiplier +a="+str(a)+" +b="+str(b)+" > out.txt"
            os.system(cmd)
            prod=ReadOutput()
            prod=int(prod)
        return(round((prod/(1024*1024)),3))
```

The multiplier() function passes the formal arguments to the 16 bit wallace multiplier's testbench. After shifting the numbers, it checks if the number is negative, if so it converts it into a positive number and passes the decimal numbers as arguments to the 16 bit wallace multiplier's test bench and fetches the output from the output.txt file. After reading the output it converts into a negative number if necessary. Finally it returns the result after shifting the product by (10+10) 20 bits (in binary) and rounding it to 3 decimal digits.

BUDGETING

Note: *The code snippets below are just functions calls to calculate the budgeting and not the entire logic/calculation done in actual code (actual code snippets are above this section) .*

n - number of data items

d - dimension of each data item

I - number of iterations

```
def euclid_distance(x,y):
      for i in range(len(x)): ------------------------------------>d times
            diff=add(x[i],-y[i])
            S=add(S,multiplier(diff,diff))
```

Function	# of add calls	# of multiplier calls	# of MAC calls
euclid_distance()	2*d	1*d	0

```
def neighbourhood_points(X, x_centroid, distance):
    for x in X:  ------------------------------------------------->n times
        distance_between = euclid_distance(x, x_centroid)
```

In neighbourhood_points() the function euclid_distance() is called n times. So total number of verilog calls are as follows:

Function	# of add calls	# of multiplier calls	# of MAC calls
neighbourhood_points()	(2*d)*n	d*n	0

```
def gaussian_kernel(distance, bandwidth):
      div=(distance / bandwidth)
      div_sq=multiplier(div,div)
      m1=multiplier(2,math.pi)
      m2=multiplier(-0.5,div_sq)
      m3=multiplier(bandwidth,math.sqrt(m1))
      val = multiplier((1/m3),np.exp(m2))
```

Function	# of add calls	# of multiplier calls	# of MAC calls
gaussian_kernel()	0	5	0

```
def runner(X,x,i):
      neighbours = neighbourhood_points(X, x, look_distance)
      for neighbour in neighbours:  ----------------------------------->n times
            distance = euclid_distance(neighbour, x)
            weight = gaussian_kernel(distance, kernel_bandwidth)
            for j in range(len(x)):  ----------------------------------->d times
                numerator[j]=add(numerator[j],multiplier(weight,neighbour[j]))
            denominator = add(denominator,weight)
```

The runner functions first calls neighbourhood_points() which has **2*d*n** add calls and **d*n** multiplier calls.

Inside the neighbour loop which runs **n** times:

Function	# of add calls	# of multiplier calls	# of MAC calls
euclid_distance()	2*d	1*d	0
gaussian_kernel()	0	5	0
e_d()+g_k()	2*d	1*d+5	0

So the total number of add and multiplier calls from e_d()+g_k() is **(2*d)*n** and **(d+5)*n** respectively.

Numerator is calculated **n*d** times so total number of add calls and multiplier calls due to this would be **n*d** , **n*d** respectively.

Denominator is calculated **n** times so total number of add calls due to this would be **n**.

Function	# of add calls	# of multiplier calls	# of MAC calls
runner()	(2*d*n) + ((2*d)*n) + (n*d) + n = 5*n*d+n	(d*n) + ((d+5)*n) + (n*d) = 3*n*d+5*n	0

```
def meanShift(original_X):
    for it in range(no_iter): ----------------------------------------->l times
        for i in range(0,len(X)):  ---------------------------------->n times
            runner(X,X[i],i)
```

The meanShift() function calls runner function for **l*n** times.

Function	# of add calls	# of multiplier calls	# of MAC calls
meanShift()	(5*n*d+n)*l*n	(3*n*d+5*n)*l*n	0

APPLICATIONS

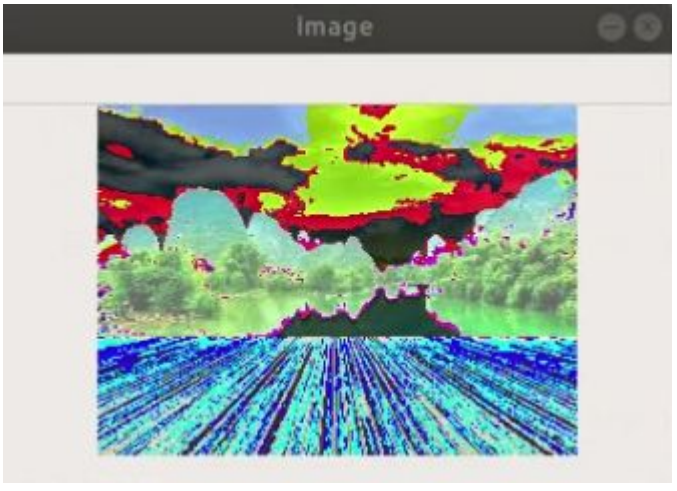
1. Image Segmentation

Image segmentation is one of the mostly used methods to classify the pixels of an image correctly in a decision oriented application. It divides an image into a number of discrete regions such that the pixels have high similarity in each region and high contrast between regions. Image segmentation is used in many fields, some are as follows :

- Health care
- Image processing
- Traffic image
- Pattern recognition etc...

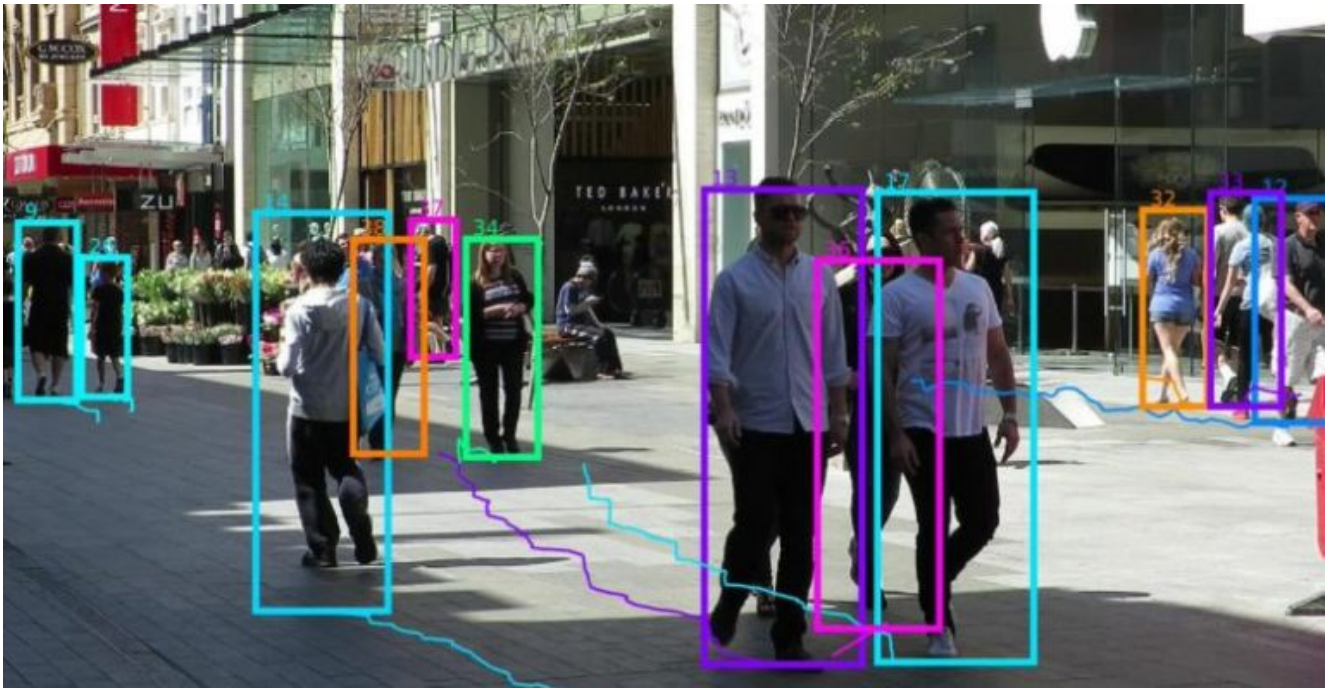
Steps in Segmentation :

- Find features (color, gradients, texture, etc)
- Initialize windows at individual pixel locations
- Perform mean shift for each window until convergence
- Merge windows that end up near the same “peak” or mode



2. Object Tracking

Mean shift clustering algorithm is also used in real time object tracking. Object can be tracked by creating a response map matrix with pixels weighted by “likelihood” that they belong to the object being tracked.



PROS AND CONS

Pros

- Finds variable number of modes
- Robust to outliers
- General, application-independent tool
- Model-free, doesn't assume any prior shape like spherical, elliptical, etc. on data clusters
- Just a single parameter (window size h) where h has a physical meaning (unlike k-means)
- Output doesn't depend on initialization.

Cons

- Output does depend on bandwidth: too small and convergence is slow, too large and some clusters may be missed.
- Computationally expensive for large feature spaces.
- Often slower than K-Means clustering (takes $O(n^2)$)

VERILOG MODULES

Adder

HalfAdder.v

```
module HalfAdder (a,b,sum, ca);
input a, b;
output sum, ca;
      assign sum = a ^ b;
      assign ca  = a&b;
endmodule
```

FullAdder.v

```
module full_adder(a,b,c,sum, carry);
input a, b, c;
output sum, carry;
wire half_sum_1, half_carry_1, half_carry_2;
HalfAdder HA1(a,b,half_sum_1, half_carry_1);
HalfAdder HA2( half_sum_1, c,sum, half_carry_2);
assign carry=half_carry_1 | half_carry_2;
endmodule
```

4BitAdder.v

```
module fourbit(a,b,cin,sum,carry);
input [3:0] a,b;
input cin;
output [3:0] sum;
output carry;
wire [3:0] w;
full_adder FA0(a[0],b[0],cin,sum[0],w[0]);
full_adder FA1(a[1],b[1],w[0],sum[1],w[1]);
full_adder FA2(a[2],b[2],w[1],sum[2],w[2]);
full_adder FA3(a[3],b[3],w[2],sum[3],w[3]);
assign carry=w[3];
endmodule
```

8BitAdder.v

```
module eightbit(a,b,cin,sum,carry);
input [7:0] a,b;
input cin;
output [7:0] sum;
output carry;
```

```
wire [1:0] w;
fourbit FA0(a[3:0],b[3:0],cin,sum[3:0],w[0]);
fourbit FA1(a[7:4],b[7:4],w[0],sum[7:4],w[1]);
assign carry=w[1];
endmodule
```

16BitAdder.v

```
module sixteenbit(a,b,cin,sum,carry);
input [15:0] a,b;
input cin;
output [15:0] sum;
output carry;
wire [1:0] w;
eightbit FA0(a[7:0],b[7:0],cin,sum[7:0],w[0]);
eightbit FA1(a[15:8],b[15:8],w[0],sum[15:8],w[1]);
assign carry=w[1];
endmodule
```

32BitAdder.v

```
module thirtytwobit(a,b,cin,sum,carry);
input [31:0] a,b;
input cin;
output [31:0] sum;
output carry;
wire [1:0] w;
sixteenbit FA0(a[15:0],b[15:0],cin,sum[15:0],w[0]);
sixteenbit FA1(a[31:16],b[31:16],w[0],sum[31:16],w[1]);
assign carry=w[1];
endmodule
```

64BitAdder.v

```
module sixtyfourbit(a,b,cin,sum,carry);
input [63:0] a,b;
input cin;
output [63:0] sum;
output carry;
wire [1:0] w;
thirtytwobit FA0(a[31:0],b[31:0],cin,sum[31:0],w[0]);
thirtytwobit FA1(a[63:32],b[63:32],w[0],sum[63:32],w[1]);
assign carry=w[1];
endmodule
```

Testbench

```
module top;
wire [63:0]sum;
wire carry;
reg [63:0]a, b;
reg c;
sixtyfourbit bit(a, b, c, sum, carry);
initial
begin
#0  c=1'b0;
if($value$plusargs("a=%b",a) && $value$plusargs("b=%b",b)); //for taking cmd
line arg
end

initial
begin
      $monitor("%b",sum);
end

endmodule
```

Multiplier

16BitAdder.v

```
module sixteenbit(a,b,cin,sum,carry);
input [15:0] a,b;
input cin;
output [15:0] sum;
output carry;
wire [15:0] w;
full_adder FA0(a[0],b[0],cin,sum[0],w[0]);
full_adder FA1(a[1],b[1],w[0],sum[1],w[1]);
full_adder FA2(a[2],b[2],w[1],sum[2],w[2]);
full_adder FA3(a[3],b[3],w[2],sum[3],w[3]);
full_adder FA4(a[4],b[4],w[3],sum[4],w[4]);
full_adder FA5(a[5],b[5],w[4],sum[5],w[5]);
full_adder FA6(a[6],b[6],w[5],sum[6],w[6]);
full_adder FA7(a[7],b[7],w[6],sum[7],w[7]);
full_adder FA8(a[8],b[8],w[7],sum[8],w[8]);
full_adder FA9(a[9],b[9],w[8],sum[9],w[9]);
full_adder FA10(a[10],b[10],w[9],sum[10],w[10]);
full_adder FA11(a[11],b[11],w[10],sum[11],w[11]);
full_adder FA12(a[12],b[12],w[11],sum[12],w[12]);
full_adder FA13(a[13],b[13],w[12],sum[13],w[13]);
full_adder FA14(a[14],b[14],w[13],sum[14],w[14]);
```

```
full_adder FA15(a[15],b[15],w[14],sum[15],w[15]);
assign carry=w[15];
endmodule
```

24BitAdder.v

```
module twentyfourbit(a,b,cin,sum,carry);
input [23:0] a,b;
input cin;
output [23:0] sum;
output carry;
wire [23:0] w;
ffull_adder FA0(a[0],b[0],cin,sum[0],w[0]);
ffull_adder FA1(a[1],b[1],w[0],sum[1],w[1]);
ffull_adder FA2(a[2],b[2],w[1],sum[2],w[2]);
ffull_adder FA3(a[3],b[3],w[2],sum[3],w[3]);
ffull_adder FA4(a[4],b[4],w[3],sum[4],w[4]);
ffull_adder FA5(a[5],b[5],w[4],sum[5],w[5]);
ffull_adder FA6(a[6],b[6],w[5],sum[6],w[6]);
ffull_adder FA7(a[7],b[7],w[6],sum[7],w[7]);
ffull_adder FA8(a[8],b[8],w[7],sum[8],w[8]);
ffull_adder FA9(a[9],b[9],w[8],sum[9],w[9]);
ffull_adder FA10(a[10],b[10],w[9],sum[10],w[10]);
ffull_adder FA11(a[11],b[11],w[10],sum[11],w[11]);
ffull_adder FA12(a[12],b[12],w[11],sum[12],w[12]);
ffull_adder FA13(a[13],b[13],w[12],sum[13],w[13]);
ffull_adder FA14(a[14],b[14],w[13],sum[14],w[14]);
ffull_adder FA15(a[15],b[15],w[14],sum[15],w[15]);
ffull_adder FA16(a[16],b[16],w[15],sum[16],w[16]);
ffull_adder FA17(a[17],b[17],w[16],sum[17],w[17]);
ffull_adder FA18(a[18],b[18],w[17],sum[18],w[18]);
ffull_adder FA19(a[19],b[19],w[18],sum[19],w[19]);
ffull_adder FA20(a[20],b[20],w[19],sum[20],w[20]);
ffull_adder FA21(a[21],b[21],w[20],sum[21],w[21]);
ffull_adder FA22(a[22],b[22],w[21],sum[22],w[22]);
ffull_adder FA23(a[23],b[23],w[22],sum[23],w[23]);
assign carry=w[23];
endmodule
```

8BitWallaceMultiplier.v

```
module wallace(a,b,product);
input [7:0] a,b;
```

```
output [15:0] product;

wire [7:0]r0,r1,r2,r3,r4,r5,r6,r7;
// generating partial product

genvar i;
for(i=0; i<8; i=i+1)
begin
assign r0[i] = a[i] & b[0];
end
for(i=0; i<8; i=i+1)
begin
assign r1[i] = a[i] & b[1];
end
for(i=0; i<8; i=i+1)
begin
assign r2[i] = a[i] & b[2];
end
for(i=0; i<8; i=i+1)
begin
assign r3[i] = a[i] & b[3];
end
for(i=0; i<8; i=i+1)
begin
assign r4[i] = a[i] & b[4];
end
for(i=0; i<8; i=i+1)
begin
assign r5[i] = a[i] & b[5];
end
for(i=0; i<8; i=i+1)
begin
assign r6[i] = a[i] & b[6];
end
for(i=0; i<8; i=i+1)
begin
assign r7[i] = a[i] & b[7];
end


//iteration-1
wire [9:0]t0;
wire [7:0]c0;
assign t0[0]=r0[0];
HA h0(r0[1],r1[0],t0[1],c0[0]);

FA f0(r0[2],r1[1],r2[0],t0[2],c0[1]);
```

```
FA f1(r0[3],r1[2],r2[1],t0[3],c0[2]);
FA f2(r0[4],r1[3],r2[2],t0[4],c0[3]);
FA f3(r0[5],r1[4],r2[3],t0[5],c0[4]);
FA f4(r0[6],r1[5],r2[4],t0[6],c0[5]);
FA f5(r0[7],r1[6],r2[5],t0[7],c0[6]);

HA h1(r1[7],r2[6],t0[8],c0[7]);
assign t0[9]=r2[7];


wire [9:0]t1;
wire [7:0]c1;
assign t1[0]=r3[0];
HA h2(r3[1], r4[0],t1[1],c1[0]);

FA f6(r3[2], r4[1],r5[0],t1[2],c1[1]);
FA f7(r3[3], r4[2],r5[1],t1[3],c1[2]);
FA f8(r3[4], r4[3],r5[2],t1[4],c1[3]);
FA f9(r3[5], r4[4],r5[3],t1[5],c1[4]);
FA f10(r3[6],r4[5],r5[4],t1[6],c1[5]);
FA f11(r3[7],r4[6],r5[5],t1[7],c1[6]);

HA h3(r4[7],r5[6],t1[8],c1[7]);
assign t1[9]=r5[7];


//iteration-2
wire [12:0]t2;
wire [7:0]c2;
assign t2[0]=t0[0];
assign t2[1]=t0[1];
HA h4(t0[2],c0[0],t2[2],c2[0]);

FA f12(t0[3],c0[1],t1[0],t2[3],c2[1]);
FA f13(t0[4],c0[2],t1[1],t2[4],c2[2]);
FA f14(t0[5],c0[3],t1[2],t2[5],c2[3]);
FA f15(t0[6],c0[4],t1[3],t2[6],c2[4]);
FA f16(t0[7],c0[5],t1[4],t2[7],c2[5]);
FA f17(t0[8],c0[6],t1[5],t2[8],c2[6]);
FA f18(t0[9],c0[7],t1[6],t2[9],c2[7]);

assign t2[10]=t1[7];
assign t2[11]=t1[8];
assign t2[12]=t1[9];
//////////////////////

wire [9:0]t3;
```

```
wire [7:0]c3;
assign t3[0]=c1[0];
HA h5(c1[1],r6[0],t3[1],c3[0]);

FA f19(c1[2],r6[1],r7[0],t3[2],c3[1]);
FA f20(c1[3],r6[2],r7[1],t3[3],c3[2]);
FA f21(c1[4],r6[3],r7[2],t3[4],c3[3]);
FA f22(c1[5],r6[4],r7[3],t3[5],c3[4]);
FA f23(c1[6],r6[5],r7[4],t3[6],c3[5]);
FA f24(c1[7],r6[6],r7[5],t3[7],c3[6]);

HA h14(r6[7],r7[6],t3[8],c3[7]);
assign t3[9]=r7[7];


//iteration-3
wire [14:0]t4;
wire [9:0]c4;
assign t4[0]=t2[0];
assign t4[1]=t2[1];
assign t4[2]=t2[2];
HA h6(t2[3],c2[0],t4[3],c4[0]);
HA h7(t2[4],c2[1],t4[4],c4[1]);

FA f25(t2[5],c2[2],t3[0],t4[5],c4[2]);
FA f26(t2[6],c2[3],t3[1],t4[6],c4[3]);
FA f27(t2[7],c2[4],t3[2],t4[7],c4[4]);
FA f28(t2[8],c2[5],t3[3],t4[8],c4[5]);
FA f29(t2[9],c2[6],t3[4],t4[9],c4[6]);
FA f30(t2[10],c2[7],t3[5],t4[10],c4[7]);

HA h8(t2[11],t3[6],t4[11],c4[8]);
HA h9(t2[12],t3[7],t4[12],c4[9]);
assign t4[13]=t3[8];
assign t4[14]=t3[9];


//iteration-4
wire [14:0]t5;
wire [10:0]c5;
assign t5[0]=t4[0];
assign t5[1]=t4[1];
assign t5[2]=t4[2];
assign t5[3]=t4[3];
HA h10(t4[4],c4[0],t5[4],c5[0]);
HA h11(t4[5],c4[1],t5[5],c5[1]);
HA h12(t4[6],c4[2],t5[6],c5[2]);
```

```
FA f31(t4[7],c4[3],c3[0],t5[7],c5[3]);
FA f32(t4[8],c4[4],c3[1],t5[8],c5[4]);
FA f33(t4[9],c4[5],c3[2],t5[9],c5[5]);
FA f34(t4[10],c4[6],c3[3],t5[10],c5[6]);
FA f35(t4[11],c4[7],c3[4],t5[11],c5[7]);
FA f36(t4[12],c4[8],c3[5],t5[12],c5[8]);
FA f37(t4[13],c4[9],c3[6],t5[13],c5[9]);

HA h15(t4[14],c3[7],t5[14],c5[10]);


//last iteration
wire [10:0]carry;
assign product[0]=t5[0];
assign product[1]=t5[1];
assign product[2]=t5[2];
assign product[3]=t5[3];
assign product[4]=t5[4];

FA f38(t5[5],c5[0],1'b0,product[5],carry[0]);
FA f39(t5[6],c5[1],carry[0],product[6],carry[1]);
FA f40(t5[7],c5[2],carry[1],product[7],carry[2]);
FA f41(t5[8],c5[3],carry[2],product[8],carry[3]);
FA f42(t5[9],c5[4],carry[3],product[9],carry[4]);
FA f43(t5[10],c5[5],carry[4],product[10],carry[5]);
FA f44(t5[11],c5[6],carry[5],product[11],carry[6]);
FA f45(t5[12],c5[7],carry[6],product[12],carry[7]);
FA f46(t5[13],c5[8],carry[7],product[13],carry[8]);
FA f47(t5[14],c5[9],carry[8],product[14],carry[9]);

HA h13(c5[10],carry[9],product[15],carry[10]);
endmodule
```

16BitMultiplier.v

```
module mul(a,b,c);
input [15:0]a;
input [15:0]b;
output [35:0]c;

wire [15:0]q0;
wire [15:0]q1;
wire [15:0]q2;
wire [15:0]q3;
```

```
wire [35:0]c;
wire [15:0]temp1;
wire [23:0]temp2;
wire [23:0]temp3;
wire [23:0]temp4;
wire [15:0]q4;
wire [23:0]q5;
wire [23:0]q6;

wire cin1,cin2,cin3,c1,c2,c3;
assign cin1=0;
assign cin2=0;
assign cin3=0;
// using 4 8x8 multipliers
wallace z1(a[7:0],b[7:0],q0[15:0]);
wallace z2(a[15:8],b[7:0],q1[15:0]);
wallace z3(a[7:0],b[15:8],q2[15:0]);
wallace z4(a[15:8],b[15:8],q3[15:0]);

// stage 1 adders
assign temp1 ={8'b0,q0[15:8]};
sixteenbit z5(q1[15:0],temp1,cin1,q4,c1);
assign temp2 ={8'b0,q2[15:0]};
assign temp3 ={q3[15:0],8'b0};
twentyfourbit z6(temp2,temp3,cin2,q5,c2);
assign temp4={8'b0,q4[15:0]};

//stage 2 adder
twentyfourbit z7(temp4,q5,cin3,q6,c3);
// fnal output assignment
assign c[7:0]=q0[7:0];
assign c[31:8]=q6[23:0];
assign c[35:32]=16'b0;

endmodule
```

TestBench.v

```
module top;
reg [15:0]a;
reg [15:0]b;
wire [35:0] result;

mul W(a,b,result);
integer i;
```

```
initial
begin
if($value$plusargs("a=%d",a) && $value$plusargs("b=%d",b)); //for taking cmd
line arg
end

initial
begin
      $monitor("%d",result);
end

endmodule
```