1 Problem Statement

The travelling salesman problem (TSP) is a classic problem in computer science that is important in both theoretical and practical applications for its use in planning and assembly. You can find a description of the classic problem at: http://en.wikipedia.org/wiki/Travelling_salesman_problem.

For this problem, it’s LDOC morning, and you’re ready for a nice day of clean fun. Before you can start relaxing, however, you have several important papers and errands to run all around campus, from E Quad all the way to Lilly Library. You have to hit each location exactly once, but you want to get back home as soon as possible to begin your university approved celebrating, so you want to find the optimal path that will minimize travel time for running all the errands.

You will be given a list of locations in rectangular (x,y) coordinates, and should print out the length of the shortest path that touches each location. The input will be of the following form:

Number of locations
X coordinate location 1
Y coordinate location 1
X coordinate location 2
Etc.

Example: 5, 0, 1, 2, 3, 1, 2, 5, 8, 10, 1

At the end you should print out the distance travelled on the shortest path.
Input can be taken by pressing enter after each number entered, if desired.

The true distance between any two points is unlikely to be an integer, so we’re actually going to compute the **square of the distance** to keep everything in integers.

Good luck, and have fun with this!

### 2 Requirements

In this part of the project you will demonstrate that you can use your pipelined processor to run software that you write that solves TSP. As part of the project, you will download your (possibly enhanced) processor design into the FPGA prototyping board, demonstrate that it works, and show how fast it can solve the TSP instance we give you.

Section 6 provides you with C++ code that implements TSP. Use the C++ code to understand how TSP works. Based on this understanding, you will write an assembly program (in the Duke 152/32 ISA) that solves TSP.

### 3 The Need For Speed

I will award bonus points to the three groups whose demos are the fastest (and correct!). 50 points for 1st place, 40 for 2nd, and 30 for 3rd. I also reserve the right to award up to 20 bonus points to any group that implements a particularly interesting feature, even though their demo is not one of the fastest three demos.

You are free to make the processor go faster using either or both of the following two approaches. You do not have to do any of this, though. You will get full credit if your processor correctly solves TSP.

- Use the currently unused opcodes in the ISA: I intentionally left several opcodes unused. You can use these opcodes for any purposes you want. You will want to slightly modify the assembler to recognize these opcodes and generate appropriate code.

- Enhance the microarchitecture: If you want to add branch prediction, more pipe stages, a 2-wide pipeline, or any other microarchitectural feature, you may do so.

Whatever you do, though, must still comply with the ISA specification. Remember: speed is great, but correctness is the most important feature.
4 Hardware Demos

As with the previous part of the project, we will have graded demos of the FPGA implementations that will be held in the ECE 154 lab (Hudson 202A) during class time on Weds, April 25. You will want to test your designs on the hardware long before then, to make sure they work correctly (even if they worked without problem in Quartus).

5 Submission

Submit two files for this assignment: enchilada.qar (the processor design) and software.s (the software).

You may re-submit as often as you like, but a re-submission will overwrite whatever you’ve previously submitted for this assignment. I will grade whatever has been submitted before 10:00AM on Wednesday, April 25.

6 C++ Code for TSP

```
#include <iostream>
#include <list>
#include <queue>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <cstdlib>
#include <stack>
#include <time.h>
using namespace std;

int* xPoints;
int* yPoints;
int minPath;
queue< int > availablePoints;
stack< int > bestPath;
int count;

int ptDistance(int pt1, int pt2)
```
{  
count++;  
int x1 = *(xPoints+pt1);  
int y1 = *(yPoints+pt1);  
int x2 = *(xPoints+pt2);  
int y2 = *(yPoints+pt2);  
int dist = pow(x2-x1,2)+pow(y2-y1,2); // note: no sqrt!!  
return dist;  
}
void find_path(int pointsLeft, int prevPoint, int dist,  
stack< int >  *currPath)  
{
  if (pointsLeft==0)
  {
    if (dist<minPath)
    {
      minPath=dist;  
      bestPath = *(currPath);  
    }
    return;
  }
  for (int i=0; i<pointsLeft; i++)
  {
    int ptOut= availablePoints.front();  
    availablePoints.pop();  
    int toNextPoint = ptDistance(ptOut, prevPoint);  
    currPath->push(ptOut);  
    find_path(pointsLeft-1, ptOut, dist+toNextPoint, currPath);  
    availablePoints.push(ptOut);  
    currPath->pop();
  }
  
}
int main(int argc, const char* argv[])  
{
  int numPoints = atoi(argv[1]);  
  count=0;
xPoints = (int *) malloc(numPoints*sizeof(int));
yPoints = (int *) malloc(numPoints*sizeof(int));
time_t start, end;
time(&start);
for (int i=1; i<numPoints+1; i++)
{
    *(xPoints+i-1) = atoi(argv[i*2]);
    *(yPoints+i-1) = atoi(argv[i*2+1]);
    availablePoints.push(i-1);
}
minPath = 0x0FFFFFFF;
int firstPoint = availablePoints.front();
availablePoints.pop();
stack< int > *currPath = new stack< int >();
currPath->push(firstPoint);
find_path(numPoints-1, firstPoint, 0, currPath);
printf("Distance: %i \n", minPath);
stack< int > printStack;
stack< int > bestPath = *currPath;
while (!bestPath.empty())
{
    printStack.push(bestPath.top());
    bestPath.pop();
}
while (!printStack.empty())
{
    printf("%i ", (int) printStack.top());
    printStack.pop();
}
time(&end);
printf("Comparisons: %i \n", count);
double dif = difftime(end, start);
printf("Program runtime: %f \n", dif);