Computer Architecture
Spring 2019

Course Project

Phase I

Team 10

2015312769 심은지

2014310407 이준혁

2015310312 임소현

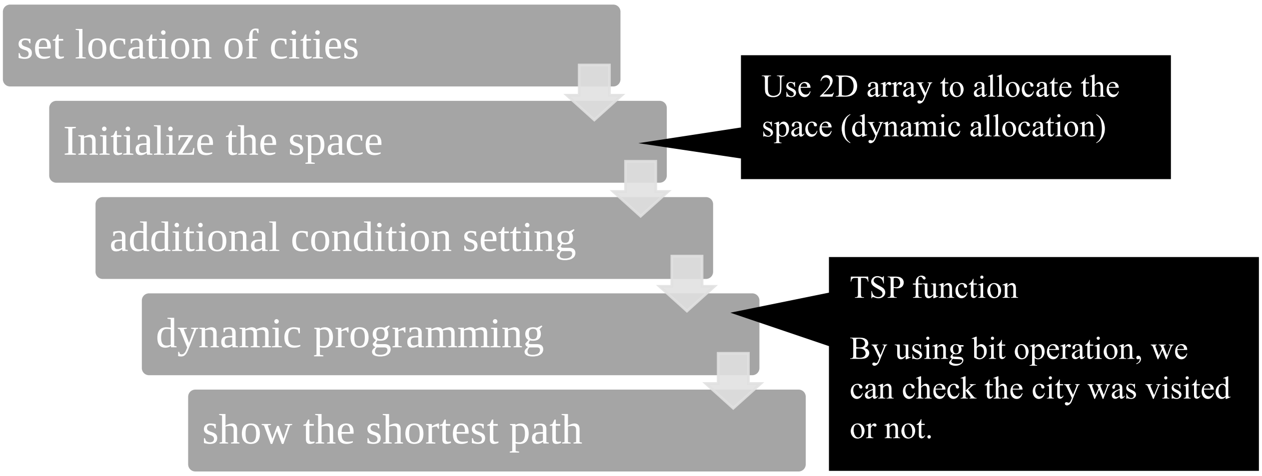
1. Preface

The purpose of this project is to solve Traveling Salesperson Problem(TSP) by using high level language, then convert into assembly code, and finally understand the relation between high level language, ISA, and assembly.

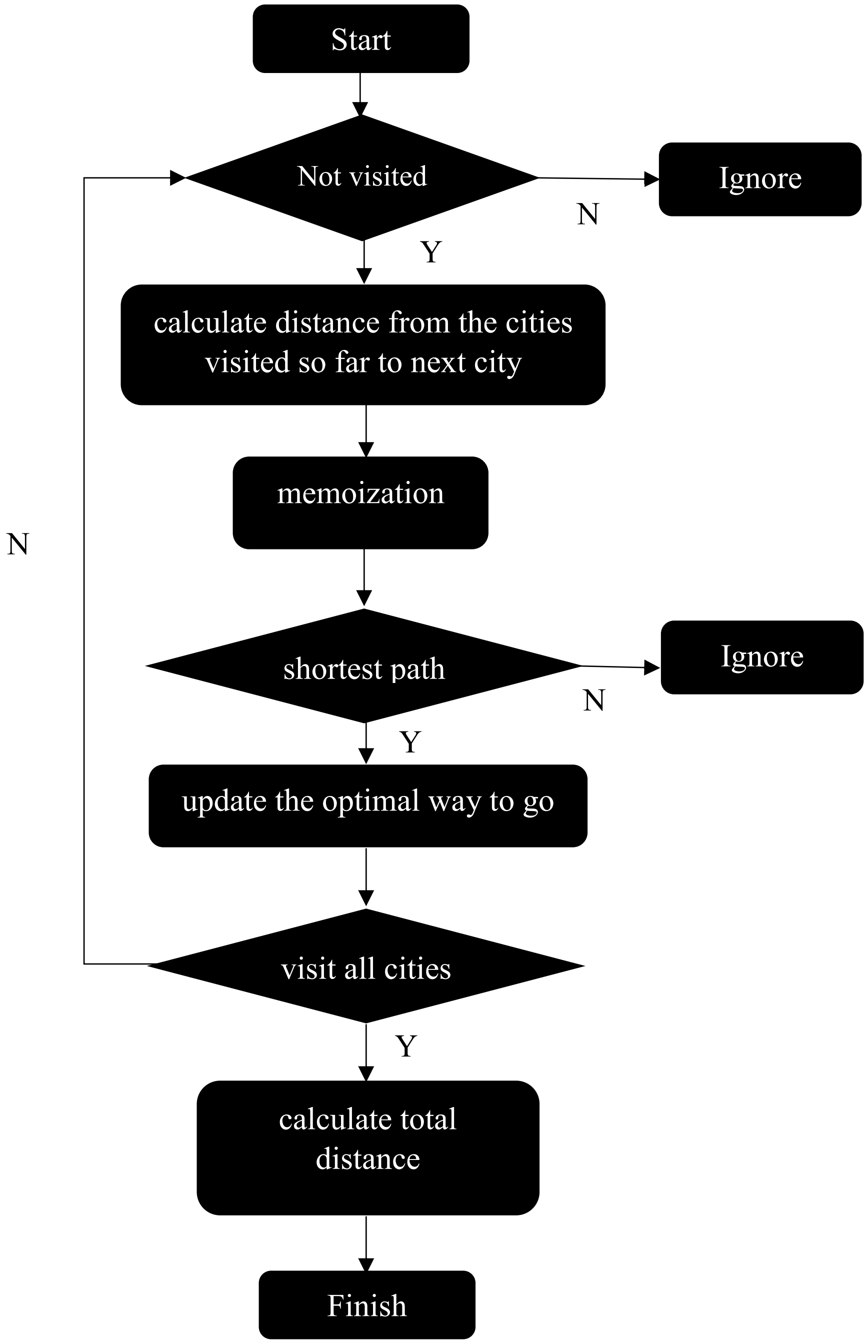
In the class, we learned the performance is reversely proportional to clock cycles, which is same with multiplication of number of instruction and cycles per instruction. Still, we are not experts in MIPS instruction, to become an expert, we tried to write the code using dynamic programming. We will calculate algorithm's performance, and try to minimize MIPS instructions.

We wrote the program in C language, The reason why we chose the dynamic programming which is the proper algorithm to solve this problem is, we can divide this TSP problem into several subproblems. We have the value which checks the status of cities whether we visited or not by using bit operation. For the memory efficiency, we allocated memory dynamically with using malloc. The algorithm uses memoization to calculate the shortest path and recursion to calculate the total shortest path. This algorithm has a time complexity $O(n^2 \times 2^n)$ for traveling with the same space complexity.

2. Flow Chart



In dynamic programming, we saved the information of visit (or not) by using bit operation in variable whose name is visited in our code. And the total distance from the shortest path visited is saved in memoize array, the next optimal location from visited so far is saved in optimal matrix. In the case of additional condition, if the City-7 comes out earlier than City-3, the program will not indicate the next location. The flow chart which is drawn below is the structure of the TSP function.


Start
Not visited
Ignore
N
Y
calculate distance from the cities visited so far to next city
memoization
N
shortest path
Ignore
N
Y
update the optimal way to go
visit all cities
Y
calculate total distance
Finish

3. Code and execution result

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

#define NODE 7   // The number of Nodes
#define VISIT_MAX (((1) << (7)) - (1))    // Tag which we visit every cities in

typedef struct city {
        int x;
        int y;
} City;

City cities[8] = { {0, 0},{0, 0}, { 8,6 },{ 2,4 },{ 6,7 },{ 1,3 },{ 9,4 },{ 2,3 } };
double** Distance;// Matrix for saving the distances of the cities
double** Memoize; // Matrix for memoizing the optimal distance from visited cities to current city
int** Optimal;   // Matrix for saving index of the optimal way for each visited cities
int* path;       // Array that stores the next place

double tsp(int cur, int visited);
double l2distance(City c1, City c2);

int main() {

        // Initialize the matrices
        Distance = (double **)malloc(sizeof(double*) * (NODE + 1));
        for (int i = 0; i <= NODE; i++)
        {
                Distance[i] = (double *)malloc(sizeof(double) * (NODE + 1));
                for (int j = 0; j <= NODE; j++)
                {
                        if (i > 0 && j > 0)
                                Distance[i][j] = l2distance(cities[i], cities[j]);

                        else
                                Distance[i][j] = 0;
                }
        }

        Memoize = (double **)malloc(sizeof(double*) * (NODE + 1));
        Optimal = (int **)malloc(sizeof(int *) * (NODE + 1));

        for (int i = 0; i <= NODE; i++)
        {
                Memoize[i] = (double *)malloc(sizeof(double) * (VISIT_MAX + 1));
                Optimal[i] = (int *)malloc(sizeof(int) * (VISIT_MAX + 1));
                for (int j = 0; j < VISIT_MAX; j++)
                {
                        Memoize[i][j] = 0;
                        Optimal[i][j] = 0;
                }
        }
```

```c
	path = (int *)malloc(sizeof(int) * (NODE + 1));

	// Run TSP algorithm, start point = 1, visit = '0000001'
	double distance = tsp(1, 1);

	printf("Distance : %6f ₩n", distance);

	// Save the result of optimal to path
	int cur = 1, visited = 1;
	for (int index = 1; index < NODE; index++)
	{
		cur = Optimal[cur][visited];
		path[index] = cur;

		visited = visited | (1 << (cur - 1));
	}

	// First and last node is 1
	path[0] = 1;
	path[NODE] = 1;

	printf("Path is : ");
	for (int i = 0; i <= NODE; i++)
		printf("%d ", path[i]);
	printf("₩n");

	getchar();
	for (int i = 0; i < NODE; i++)
		free(Memoize[i]), free(Optimal[i]), free(Distance[i]);

	free(Memoize), free(Optimal), free(Distance), free(path);

	return 0;
}


// Calculate L2 distance between two cities
double l2distance(City c1, City c2) {
	return sqrt(pow((c1.x - c2.x), 2) + pow((c1.y - c2.y), 2));
}

double tsp(int cur, int visited) {
	// If we visit all the cities, return the distance of the current place and start point(1)
	if (visited == VISIT_MAX)
		if(cur != 1)
			return Distance[cur][1];

	// If the distance is memoized, then simply returns the marked value
	if (Memoize[cur][visited] != 0)
		return Memoize[cur][visited];

	double temp_distance = INFINITY;  /* temporal variable that stores the minimum distance */
```

```
        // For each nodes
        for (int i = 1; i <= NODE; i++) {
                // Ignore the already visited nodes
                if (visited & 1 << (i - 1))
                        continue;

                // Ignore the current node
                if (cur == i)
                        continue;

                // Calculate the total distance for each nodes, using recursion
                double val = tsp(i, visited | (1 << (i - 1))) + Distance[cur][i];

                // If the found one is better than previous one
                if (temp_distance > val)
                {
                        if (i == 7)
                                                        // If the next shortest location is 7
                                if (!(visited & (1 << 3)))
                                // If I did not visited 3, then does not go to 7
                                        continue;

                        temp_distance = val;
                        Optimal[cur][visited] = i;
                        Memoize[cur][visited] = temp_distance;
                }

        }
        return Memoize[cur][visited];

}
```

```
C:₩Users₩EunJi Sim₩source₩repos₩ComputerArchitecture
Distance : 24.483271
Path is : 1 6 2 4 3 7 5 1
```

Computer Architecture
Spring 2019

Course Project

Phase II

Team 10

2015312769 심은지

2014310407 이준혁

2015310312 임소현

1. Before starting phase 2

Before we start phase 2, we edited the C code to reduce the MIPS instruction. In the original code, we used pow function to calculate the second power, but we thought it may increase the total MIPS instruction by defining pow in MIPS. So we decided not to use math function, but to just multiply two double type by deleting l2distance function. The edited code attached below.

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

#define MAX_VALUE 987654321.0

#define NODE 7                              // The number of Nodes
#define VISIT_MAX ((1 << (NODE)) - 1)       // Tag which we visit every cities in

#define CITY_3 (3)
#define VISIT_3 (1 << 3)


typedef struct city {
        int x;
        int y;
} City;

City cities[8] = { { 0, 0 },{ 0, 0 },{ 8, 6 },{ 2, 4 },{ 6, 7 },{ 1, 3 },{ 9, 4 },{ 2, 3 } };
double** Distance;              // Matrix for saving the distances of the cities
double** Memoize;              // Matrix for memoizing the optimal distance from visited cities to
current city
int** Optimal;                 // Matrix for saving index of the optimal way for each visited cities
int* path;                    // Array that stores the next place

double tsp(int cur, int visited);

int main() {

        // Initialize the matrices
        Distance = (double **)malloc(sizeof(double*) * (NODE + 1));
        for (int i = 0; i <= NODE; i++)
        {
                Distance[i] = (double *)malloc(sizeof(double) * (NODE + 1));
                int tempx = 0;
                int tempy = 0;
                for (int j = 0; j <= NODE; j++)
                {
                        if (i > 0 && j > 0 && i != j) {
                                tempx = cities[i].x - cities[j].x;
                                tempy = cities[i].y - cities[j].y;
                                Distance[i][j] = sqrt((tempx * tempx) + (tempy * tempy));  //
Calculate L2 distance between two cities
```

```
			}
			else
				Distance[i][j] = 0;
		}
	}

	Memoize = (double **)malloc(sizeof(double*) * (NODE + 1));
	Optimal = (int **)malloc(sizeof(int *)* (NODE + 1));

	for (int i = 0; i <= NODE; i++)
	{
		Memoize[i] = (double *)malloc(sizeof(double) * (VISIT_MAX + 1));
		Optimal[i] = (int *)malloc(sizeof(int)* (VISIT_MAX + 1));
		for (int j = 0; j <= VISIT_MAX; j++)
		{
			Memoize[i][j] = 0;
			Optimal[i][j] = 0;
		}
	}

	path = (int *)malloc(sizeof(int)* (NODE + 1));

	// Run TSP algorithm, start point = 1, visit = '0000001'
	double distance = tsp(1, 1);

	printf("Distance : %6f ₩n", distance);

	// Save the result of optimal to path
	int cur = 1, visited = 1;
	for (int index = 1; index < NODE; index++)
	{
		cur = Optimal[cur][visited];
		path[index] = cur;

		visited = visited | (1 << (cur - 1));
	}

	// First and last node is 1
	path[0] = 1;
	path[NODE] = 1;

	printf("Path is : ");
	for (int i = 0; i <= NODE; i++)
		printf("%d ", path[i]);
	printf("₩n");

	getchar();

	for (int i = 0; i < NODE; i++)
		free(Memoize[i]), free(Optimal[i]), free(Distance[i]);

	free(Memoize), free(Optimal), free(Distance), free(path);
```

```
        return 0;
}

double tsp(int cur, int visited) {
        // If we visit all the cities, return the distance of the current place and start point(1)
        if (visited == VISIT_MAX)
                return Distance[cur][1];

        // If the distance is memoized, then simply returns the marked value
        if (Memoize[cur][visited] != 0)
                return Memoize[cur][visited];

        double temp_distance = MAX_VALUE;    /* temporal variable that stores the minimum distance */

        double val;
        // For each nodes
        for (int k = 1; k <= NODE; k++) {
                // Ignore the already visited nodes
                if (visited & (1 << (k - 1)))
                        continue;

                // Ignore the current node
                if (cur == k)
                        continue;

                if (k == 7 && !(visited & VISIT_3))
                        continue;

                // Calculate the total distance for each nodes, using recursion
                val = tsp(k, visited | (1 << (k - 1))) + Distance[cur][k];

                // If the found one is better than previous one
                if (temp_distance > val)
                {
                        temp_distance = val;
                        Optimal[cur][visited] = k;
                        Memoize[cur][visited] = temp_distance;
                }

        }
        return Memoize[cur][visited];

}
```

2. MIPS instruction

```
.data
cities:
        .word   0
        .word   0
        .word   0
        .word   0
        .word   8
        .word   6
        .word   2
        .word   4
        .word   6
        .word   7
        .word   1
        .word   3
        .word   9
        .word   4
        .word   2
        .word   3

$LC0:
        .ascii  "Distance : %6f \n"
$LC1 :
        .ascii  "Path is : \n"
$LC2 :
        .ascii  "%d \n"

.text
main :
        addiu   $sp, $sp, -104
        sw     $31, 100($sp)
        sw     $fp, 96($sp)
```

```
        sw      $16, 92($sp)
        move    $fp, $sp
        li      $4, 32                  # 0x20
        jal     malloc
        sw      $2, 56($fp)
        sw      $0, 24($fp)
        b       $L2

$L7 :
        lw      $2, 24($fp)
        sll     $2, $2, 2
        lw      $3, 56($fp)
        addu    $16, $3, $2
        li      $4, 64                  # 0x40
        jal     malloc
        sw      $2, 0($16)
        sw      $0, 60($fp)
        sw      $0, 64($fp)
        sw      $0, 28($fp)
        b       $L3

$L6 :
        lw      $2, 24($fp)
        blez    $2, $L4
        lw      $2, 28($fp)
        blez    $2, $L4
        lw      $3, 24($fp)
        lw      $2, 28($fp)

        beq     $3, $2, $L4

        lw      $2, cities
```

```
lw     $3, 24($fp)

sll    $3, $3, 3
la            $2, cities($2)
addu   $2, $3, $2
lw     $3, ($2)
lw     $2, cities
lw     $4, 28($fp)

sll    $4, $4, 3
la            $2, cities($2)
addu   $2, $4, $2
lw     $2, ($2)

subu   $2, $3, $2
sw     $2, 60($fp)
lw     $3, cities
lw     $2, 24($fp)
la            $3, cities($3)
sll    $2, $2, 3
addu   $2, $3, $2
lw     $3, 4($2)
lw     $4, cities
lw     $2, 28($fp)
la            $4, cities($4)
sll    $2, $2, 3
addu   $2, $4, $2
lw     $2, 4($2)

subu   $2, $3, $2
sw     $2, 64($fp)
lw     $2, 24($fp)
```

```
$L4 :

        sll     $2, $2, 2
        lw      $3, 56($fp)

        addu    $2, $3, $2
        lw      $3, ($2)
        lw      $2, 28($fp)

        sll     $2, $2, 3
        addu    $16, $3, $2
        lw      $3, 60($fp)
        lw      $2, 60($fp)

        mult    $3, $2
        mflo    $2
        lw      $4, 64($fp)
        lw      $3, 64($fp)

        mult    $4, $3
        mflo    $3
        addu    $2, $2, $3
        mtc1    $2, $f0

        cvt.d.w $f0, $f0
        mov.d   $f12, $f0
        jal     sqrt

        swc1    $f0, 4($16)
        swc1    $f1, 0($16)
        b       $L5

$L4 :
```

```
        lw      $2, 24($fp)

        sll     $2, $2, 2
        lw      $3, 56($fp)

        addu    $2, $3, $2
        lw      $3, ($2)
        lw      $2, 28($fp)

        sll     $2, $2, 3
        addu    $2, $3, $2
        sw      $0, 4($2)
        sw      $0, 0($2)

$L5:
        lw      $2, 28($fp)

        addiu   $2, $2, 1
        sw      $2, 28($fp)

$L3 :
        lw      $2, 28($fp)

        slt     $2, $2, 8
        bne     $2, $0, $L6

        lw      $2, 24($fp)

        addiu   $2, $2, 1
        sw      $2, 24($fp)

$L2:
```

```
        lw      $2, 24($fp)

        slt     $2, $2, 8
        bne     $2, $0, $L7

        li      $4, 32              # 0x20
        jal     malloc

        sw      $2, 68($fp)
        li      $4, 32              # 0x20
        jal     malloc

        sw      $2, 72($fp)
        sw      $0, 32($fp)
        b       $L8

$L11 :
        lw      $2, 32($fp)

        sll     $2, $2, 2
        lw      $3, 68($fp)

        addu    $16, $3, $2
        li      $4, 1024         # 0x400
        jal     malloc

        sw              $2, 0($16)
        lw              $2, 32($fp)

        sll     $2, $2, 2
        lw      $3, 72($fp)
```

```
        addu    $16, $3, $2
        li      $4, 512                 # 0x200
        jal     malloc

        sw      $2, 0($16)
        sw      $0, 36($fp)
        b       $L9

$L10 :
        lw      $2, 32($fp)

        sll     $2, $2, 2
        lw      $3, 68($fp)

        addu    $2, $3, $2
        lw      $3, ($2)
        lw      $2, 36($fp)

        sll     $2, $2, 3
        addu    $2, $3, $2
        sw      $0, 4($2)
        sw      $0, 0($2)
        lw      $2, 32($fp)

        sll     $2, $2, 2
        lw      $3, 72($fp)

        addu    $2, $3, $2
        lw      $3, ($2)
        lw      $2, 36($fp)

        sll        $2, $2, 2
```

```
        addu    $2, $3, $2
        sw      $0, 0($2)
        lw      $2, 36($fp)

        addiu   $2, $2, 1
        sw      $2, 36($fp)

$L9:
        lw      $2, 36($fp)

        slt     $2, $2, 128
        bne     $2, $0, $L10

        lw      $2, 32($fp)

        addiu   $2, $2, 1
        sw      $2, 32($fp)

$L8:
        lw      $2, 32($fp)

        slt     $2, $2, 8
        bne     $2, $0, $L11

        li      $4, 32              # 0x20
        jal     malloc

        sw      $2, 76($fp)
        li      $5, 1               # 0x1
        li      $4, 1               # 0x1
        jal     tsp
```

```
        swc1    $f0, 84($fp)
        swc1    $f1, 80($fp)
        lw      $7, 84($fp)
        lw      $6, 80($fp)
        la      $2, $LC0
        la              $4, $LC0($2)
        jal     printf

        li      $2, 1                   # 0x1
        sw      $2, 40($fp)
        li      $2, 1                   # 0x1
        sw      $2, 44($fp)
        li      $2, 1                   # 0x1
        sw      $2, 48($fp)
        b       $L12

$L13 :
        lw      $2, 40($fp)

        sll     $2, $2, 2
        lw      $3, 72($fp)

        addu    $2, $3, $2
        lw      $3, ($2)
        lw      $2, 44($fp)

        sll     $2, $2, 2
        addu    $2, $3, $2
        lw      $2, 0($2)

        sw      $2, 40($fp)
        lw      $2, 48($fp)
```

```
        sll     $2, $2, 2
        lw      $3, 76($fp)

        addu    $2, $3, $2
        lw      $3, 40($fp)

        sw      $3, ($2)
        lw      $2, 40($fp)

        addiu   $2, $2, -1
        li      $3, 1                       # 0x1
        sll     $2, $3, $2
        lw      $3, 44($fp)

        or $2, $3, $2
        sw      $2, 44($fp)
        lw      $2, 48($fp)

        addiu   $2, $2, 1
        sw      $2, 48($fp)

$L12:
        lw      $2, 48($fp)

        slt     $2, $2, 7
        bne     $2, $0, $L13

        lw      $2, 76($fp)
        li      $3, 1                       # 0x1
        sw      $3, ($2)
        lw      $2, 76($fp)
```

```
        addiu   $2, $2, 28
        li      $3, 1                        # 0x1
        sw      $3, ($2)
        la      $2, $LC1
        jal     printf

        sw      $0, 52($fp)
        b       $L14

$L15 :
        lw      $2, 52($fp)

        sll     $2, $2, 2
        lw      $3, 76($fp)

        addu    $2, $3, $2
        lw      $2, 0($2)

        move    $5, $2
        la      $2, $LC2
        jal     printf

        lw      $2, 52($fp)

        addiu   $2, $2, 1
        sw      $2, 52($fp)

$L14:
        lw      $2, 52($fp)

        slt     $2, $2, 8
```

```
        bne     $2, $0, $L15

        li      $4, 10              # 0xa
        jal     putchar

        jal     getchar

        move    $2, $0
        move    $sp, $fp
        lw      $31, 100($sp)
        lw      $fp, 96($sp)
        lw      $16, 92($sp)
        addiu   $sp, $sp, 104
        j       $31

tsp :
        addiu   $sp, $sp, -56
        sw      $31, 52($sp)
        sw      $fp, 48($sp)
        move    $fp, $sp
        sw      $4, 56($fp)
        sw      $5, 60($fp)
        lw      $3, 60($fp)
        li      $2, 127             # 0x7f
        bne     $3, $2, $L18

        lw      $3, Distance
        lw      $2, 56($fp)

        sll     $2, $2, 2
        addu    $2, $3, $2
        lw      $2, 0($2)
```

```
        lwc1    $f0, 12($2)

        lwc1    $f1, 8($2)
        b       $L19

$L18 :
        lw      $3, Memoize
        lw      $2, 56($fp)

        sll     $2, $2, 2
        addu    $2, $3, $2
        lw      $3, ($2)
        lw      $2, 60($fp)

        sll     $2, $2, 3
        addu    $2, $3, $2
        lwc1    $f0, 4($2)

        lwc1    $f1, 0($2)
        mtc1    $0, $f2

        mtc1    $0, $f3

        c.eq.d  $f0, $f2

        bc1t    $L20

        lw      $3, Memoize
        lw      $2, 56($fp)

        sll     $2, $2, 2
```

```
        addu    $2, $3, $2
        lw      $3, ($2)
        lw      $2, 60($fp)

        sll     $2, $2, 3
        addu    $2, $3, $2
        lwc1    $f0, 4($2)

        lwc1    $f1, 0($2)
        b       $L19

$L20 :
        lw      $2, $LC3
        lwc1    $f0, 4($2)

        lwc1    $f1, $LC3

        swc1    $f0, 28($fp)
        swc1    $f1, 24($fp)
        li      $2, 1                   # 0x1
        sw      $2, 32($fp)
        b       $L21

$L27 :
        lw      $2, 32($fp)

        addiu   $2, $2, -1
        lw      $3, 60($fp)

        sra     $2, $3, $2
        andi    $2, $2, 0x1
        bne     $2, $0, $L29
```

```
        lw      $3, 56($fp)
        lw      $2, 32($fp)

        beq     $3, $2, $L30

        lw      $3, 32($fp)
        li      $2, 7                   # 0x7
        bne     $3, $2, $L25

        lw      $2, 60($fp)

        andi    $2, $2, 0x8
        beq     $2, $0, $L31

$L25 :
        lw      $2, 32($fp)

        addiu   $2, $2, -1
        li      $3, 1                   # 0x1
        sll     $3, $3, $2
        lw      $2, 60($fp)

        or $2, $3, $2
        move    $5, $2
        lw      $4, 32($fp)
        jal     tsp

        mov.d   $f2, $f0
        lw      $3, Distance
        lw      $2, 56($fp)
```

```
sll    $2, $2, 2
addu   $2, $3, $2
lw     $3, ($2)
lw     $2, 32($fp)

sll    $2, $2, 3
addu   $2, $3, $2
lwc1   $f0, 4($2)

lwc1   $f1, 0($2)

add.d  $f0, $f2, $f0
swc1   $f0, 44($fp)
swc1   $f1, 40($fp)
lwc1   $f2, 28($fp)

lwc1   $f3, 24($fp)
lwc1   $f0, 44($fp)

lwc1   $f1, 40($fp)

c.lt.d $f0, $f2

bc1f   $L23

lwc1   $f0, 44($fp)

lwc1   $f1, 40($fp)

swc1   $f0, 28($fp)
swc1   $f1, 24($fp)
lw     $3, Optimal
```

```
        lw      $2, 56($fp)

        sll     $2, $2, 2
        addu    $2, $3, $2
        lw      $3, ($2)
        lw      $2, 60($fp)

        sll     $2, $2, 2
        addu    $2, $3, $2
        lw      $3, 32($fp)

        sw      $3, ($2)
        lw      $3, Memoize
        lw      $2, 56($fp)

        sll     $2, $2, 2
        addu    $2, $3, $2
        lw      $3, ($2)
        lw      $2, 60($fp)

        sll     $2, $2, 3
        addu    $2, $3, $2
        lwc1    $f0, 28($fp)

        lwc1    $f1, 24($fp)

        swc1    $f0, 4($2)
        swc1    $f1, 0($2)
        b       $L23

$L29 :
```

```
        b       $L23

$L30 :
        b       $L23

$L31 :

$L23 :
        lw      $2, 32($fp)

        addiu   $2, $2, 1
        sw      $2, 32($fp)

$L21 :
        lw      $2, 32($fp)

        slt     $2, $2, 8
        bne     $2, $0, $L27


        lw      $3, Memoize
        lw      $2, 56($fp)

        sll     $2, $2, 2
        addu    $2, $3, $2
        lw      $3, ($2)
        lw      $2, 60($fp)

        sll     $2, $2, 3
        addu    $2, $3, $2
        lwc1    $f0, 4($2)
```

```
        lwc1    $f1, 0($2)

$L19:
        move    $sp, $fp
        lw      $31, 52($sp)
        lw      $fp, 48($sp)
        addiu   $sp, $sp, 56
        j       $31



$LC3 :
        .word   1103982388
        .word   1484783616

malloc:
        li      $v0,9
        li      $a0,4
        syscall
        move    $s0,$v0

        li      $t0,77
        sw      $t0,0($s0)

        lw      $a0,0($s0)
        li      $v0,1
        syscall

        li      $v0,10
        jr      $ra

putchar:
        li      $v0, 11
```

```
        syscall

printf:
        subu $sp, $sp, 36   # set up the stack frame,
        sw $ra, 32($sp)             # saving the local environment.
        sw $fp, 28($sp)
        sw $s0, 24($sp)
        sw $s1, 20($sp)
        sw $s2, 16($sp)
        sw $s3, 12($sp)
        sw $s4, 8($sp)
        sw $s5, 4($sp)
        sw $s6, 0($sp)
        addu $fp, $sp, 36
        move $s7,$fp

        # grab the arguments:
        move $s0, $a0               # fmt string
        move $s1, $a1               # arg1 (optional)
        move $s2, $a2               # arg2 (optional)
        move $s3, $a3               # arg3 (optional)

        li $s4, 0                   # set number of formats = 0

        subu $sp, $sp, 36
        move $s6, $sp

sqrt:
        mult $4, $4                   # {hi,lo} = x^2 in (64,28)
        mfhi $6                      # move hi part to register 6
        srl $6, $6, 18                # shift hi for (32,14) format
        mflo $7                      # move lo to register 7
```

```
        sll $7, $7, 14                # shift lo for (32,14) format
        or $8, $6, $7  # combine the hi and lo into a converted (32,14) value

        sub $9, $8, $28               # val = x^2 - S(input)

        bgez $9, gtz                  # if val >= 0, branch to gtz
        add $4, $4, $5                # else x = x + step
        srl $5, $5, 1                 # step = step/2
        bgez $5, sqrt                 # if step >= 0, go back into loop
        j BCD                        # else continue to BCD for output
gtz:                                 # greater than zero branch
        sub $4, $4, $5                # x = x - step
        srl $5, $5, 1                 # step = step/2
        bgez $5, sqrt                 # if step >= 0, go back into loop
        j BCD                        # else continue to BCD for output

BCD:    # function for BCD output to HEX

getchar:
        lbu    $t1,0($t0)           # get char
```

3. Expected results

When considering the highlighted instructions (branch, jump), we estimated the total instruction count and cycles about 300, 100 each.

Instruction Count	300
Total Cycles	100
CPI	0.33
Clock rate	1000
CPU Time	99ms

To calculate the CPU time, we need to know the clock rate, so we took temporary way to check the clock rate of computer which will be used to run SPIM in phase 3. The attached screen shot shows the results where CLOCK_PER_SEC means the actual clock rate, except the first one, the other values are close to 1000. We considered the clock rate as 1000.

```
C:₩Users₩Sim Eun-Ji₩Source₩Repos₩Project2₩Debug₩Project2.exe
Actual clocks per second = 242
CLOCKS_PER_SEC = 1000
Actual clocks per second = 999
CLOCKS_PER_SEC = 1000
Actual clocks per second = 998
CLOCKS_PER_SEC = 1000
Actual clocks per second = 998
CLOCKS_PER_SEC = 1000
Actual clocks per second = 997
CLOCKS_PER_SEC = 1000
Actual clocks per second = 998
CLOCKS_PER_SEC = 1000
Actual clocks per second = 998
CLOCKS_PER_SEC = 1000
Actual clocks per second = 998
CLOCKS_PER_SEC = 1000
Actual clocks per second = 997
CLOCKS_PER_SEC = 1000
Actual clocks per second = 998
CLOCKS_PER_SEC = 1000
```

Then, we can calculate the execution time.

$$\text{execution time} = \frac{Instruction\ Count\ \times CPI}{Clock\ Rate} = \frac{300 \times 0.33}{1000} = 0.099s = 99ms$$

Computer Architecture
Spring 2019

Course Project

Phase III

Team 10

2015312769 심은지

2014310407 이준혁

2015310312 임소현

1. Finding errors on previous code

Our MIPS code compiled well, but there are several problems when run on QtSpim: bad address, arithmetic overflow. The int register values at the time that each bad address and arithmetic overflow happens are like this:

Bad address	Arithmetic overflow
PC = 80000180 EPC = 400738 Cause = 30 BadVAddr = e Status = 3000ff13 HI = 3ffff3c6 LO = 957d24 R0 [r0] = 0 R1 [at] = 10010000 R2 [v0] = 64 R3 [v1] = 24 R4 [a0] = 80000c3a R5 [a1] = 3ffff9e0 R6 [a2] = fff R7 [a3] = 5f490000 R8 [t0] = 5f490fff R9 [t1] = 5f490fff R10 [t2] = 0 R11 [t3] = 0 R12 [t4] = 0 R13 [t5] = 0 R14 [t6] = 0 R15 [t7] = 0 R16 [s0] = 10 R17 [s1] = 0 R18 [s2] = 0 R19 [s3] = 0 R20 [s4] = 0 R21 [s5] = 0 R22 [s6] = 0 R23 [s7] = 0 R24 [t8] = 0 R25 [t9] = 0 R26 [k0] = 3000ff13 R27 [k1] = 10010000 R28 [gp] = 0 R29 [sp] = 7ffff354 R30 [s8] = 7ffff354 R31 [ra] = 400178	PC = 400738 EPC = 40073c Cause = 0 BadVAddr = e Status = 3000ff11 HI = 3ffffffd LO = 9 R0 [r0] = 0 R1 [at] = 10010000 R2 [v0] = 64 R3 [v1] = 24 R4 [a0] = 80000003 R5 [a1] = 0 R6 [a2] = fff R7 [a3] = 24000 R8 [t0] = 24fff R9 [t1] = 24fff R10 [t2] = 0 R11 [t3] = 0 R12 [t4] = 0 R13 [t5] = 0 R14 [t6] = 0 R15 [t7] = 0 R16 [s0] = 10 R17 [s1] = 0 R18 [s2] = 0 R19 [s3] = 0 R20 [s4] = 0 R21 [s5] = 0 R22 [s6] = 0 R23 [s7] = 0 R24 [t8] = 0 R25 [t9] = 0 R26 [k0] = 3000ff13 R27 [k1] = 10010000 R28 [gp] = 0 R29 [sp] = 7ffff354 R30 [s8] = 7ffff354 R31 [ra] = 400178

After arithmetic overflow happens, the MIPS instruction fell in infinite loop in calculating square root value. We thought there is a problem in branch instruction but did not know how to fix it. The part that problem occurred is attached below.

```
[00400708] 00840018  mult $4, $4            ; 637: mult $4, $4 # {hi,lo} = x^2 in (64,28)
[0040070c] 00003010  mfhi $6                ; 638: mfhi $6 # move hi part to register 6
[00400710] 00063482  srl $6, $6, 18         ; 639: srl $6, $6, 18 # shift hi for (32,14) format
[00400714] 00003812  mflo $7                ; 640: mflo $7 # move lo to register 7
[00400718] 00073b80  sll $7, $7, 14         ; 641: sll $7, $7, 14 # shift lo for (32,14) format
[0040071c] 00c74025  or $8, $6, $7          ; 642: or $8, $6, $7 # combine the hi and lo into a converted (32,14) value
[00400720] 011c4822  sub $9, $8, $28        ; 644: sub $9, $8, $28 # val = x^2 - S(input)
[00400724] 05210005  bgez $9 20 [gtz-0x00400724]; 646: bgez $9, gtz # if val >= 0, branch to gtz
[00400728] 00852020  add $4, $4, $5         ; 647: add $4, $4, $5 # else x = x + step
[0040072c] 00052842  srl $5, $5, 1          ; 648: srl $5, $5, 1 # step = step/2
[00400730] 04a1fff6  bgez $5 -40 [sqrt-0x00400730]; 649: bgez $5, sqrt # if step >= 0, go back into loop
[00400734] 081001d2  j 0x00400748 [BCD]     ; 650: j BCD # else continue to BCD for output
[00400738] 00852022  sub $4, $4, $5         ; 652: sub $4, $4, $5 # x = x - step
[0040073c] 00052842  srl $5, $5, 1          ; 653: srl $5, $5, 1 # step = step/2
[00400740] 04a1fff2  bgez $5 -56 [sqrt-0x00400740]; 654: bgez $5, sqrt # if step >= 0, go back into loop
```

With arithmetic overflow, fp register also faced problem.

	Single Precision	Double Precision
FIR = 9800	FG0 = 0	FP0 = 4059000000000000
FCSR = 0	FG1 = 40590000	FP2 = 0
FCCR = 0	FG2 = 0	FP4 = 0
FEXR = 0	FG3 = 0	FP6 = 0
	FG4 = 0	FP8 = 0
	FG5 = 0	FP10 = 0
	FG6 = 0	FP12 = 4059000000000000
	FG7 = 0	FP14 = 0
	FG8 = 0	FP16 = 0
	FG9 = 0	FP18 = 0
	FG10 = 0	FP20 = 0
	FG11 = 0	FP22 = 0
	FG12 = 0	FP24 = 0
	FG13 = 40590000	FP26 = 0
	FG14 = 0	FP28 = 0
	FG15 = 0	FP30 = 0
	FG16 = 0	
	FG17 = 0	
	FG18 = 0	
	FG19 = 0	
	FG20 = 0	
	FG21 = 0	
	FG22 = 0	
	FG23 = 0	
	FG24 = 0	
	FG25 = 0	
	FG26 = 0	
	FG27 = 0	
	FG28 = 0	
	FG29 = 0	
	FG30 = 0	
	FG31 = 0	

2. New trial – completed version

We tried to change code to avoid arithmetic overflow, not to occur bad address exception. The edited version of code is more efficient in terms of execution, memory efficiency. The final code is attached the right after this analysis.

1) Use float data type instead of double: make simple instruction and reduce memory space

2) Use register evenly: reduce memory usage and use cache

In phase 2, we used \$2, \$3 register repeatedly, the code was longer and data hazard occurred. The code was more than 700 lines except for repetition, but it was reduced to less than 300 lines after modification.

3) Add distance calculation code

At the first time, we included math header file, and use pow and sqrt function in C. When the first code was converted into MIPS, there were several problems such as invalid register allocation, unspecified instructions. However, we removed pow and sqrt function from C code, instead multiplied itself – distance – twice and use sqrt.s in MIPS, we could solve invalid register allocation problem and arithmetic overflow. Also, we gave up using malloc because in MIPS, we defined malloc function with several instructions, but the branch problem occurred, so it also changed to two dimensional array. To reduce the code size, we reduced the number of iterations of the for loop by half (because dist[i][j] == dist[j][i].) We had added the destination, city 1, at the end of the route – so that the number of cities was eight, but to optimize for calculation and search, we deleted the duplicated city 1 – the number of cities become seven.

4) Replace naïve dynamic programming with dynamic programming using tail recursion: simplify the calculation of current distance and avoid using complex, new arrays

We used four arguments: current position, tsp end position, visited bitmask, current sum. And using bitmask, we could check we visited the city or not without using array. Instead of using memorize and optimal array, we used tail recursion which calculate the current distance and path, and finally optimize the calculation.

2.1. Final C code

```c
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

typedef struct city {
	int x;
	int y;
} City;

float min_dist = 987654321.0; // final distance result (start with max)
City cities[7] = { { 0, 0 },{ 8, 6 },{ 2, 4 },{ 6, 7 },{ 1, 3 },{ 9, 4 },{ 2, 3 } };
float  dist[7][7]; //Matrix for saving distance of the cities
int  optimal[7];  //array that stores next place
int  path[8]; //final path result

void tsp(int cur, int path_count, int visited, float sum);

int main() {
	int i, j;
	int tempx = 0;
	int tempy = 0;

	//calculate distance between two cities
	for (i = 0; i < 7; i++)
	{
		for (j = 0; j < i; j++)
		{
			tempx = cities[i].x - cities[j].x;
			tempy = cities[i].y - cities[j].y;
			dist[i][j] = sqrt((tempx * tempx) + (tempy * tempy));
			dist[j][i] = dist[i][j];
		}
	}

	path[0] = 1; //start po
	path[7] = 1; //
	optimal[0] = 1;
	tsp(0, 0, 1, 0);  //tsp

	//printf distance
	printf("Distance : %6f ₩n", min_dist);

	//printf path
	printf("Path is : ");

	for (int i = 0; i < 8; i++)
		printf("%d ", path[i]);
	printf("₩n");

	getchar();
```

```
        return 0;
}

void tsp(int cur, int path_count, int visited, float sum) {
        if (visited == ((1 << (7)) - 1)) {
                sum += dist[cur][0];
                if (sum < min_dist) {
                        min_dist = sum;
                        for (int i = 0; i < 7; i++) {
                                path[i] = optimal[i];
                        }
                }
                return;
        }

        for (int k = 1; k < 7; k++) {
                // ignore already visited nodes
                if (visited & (1 << (k)))
                        continue;

                //ignore current node
                if (cur == k)
                        continue;

                if (sum + dist[cur][k] < min_dist)
                {
                        // three person team additional condition - if I did not visit 3, then do not
go to 7

                        if (k + 1 == 7 && !(visited & (1 << (3 - 1))))
                                continue;

                        optimal[path_count + 1] = k + 1;

                        /* Calculate the total distance for each nodes, using recursion */
                        tsp(k, path_count + 1, visited | (1 << (k)), sum + dist[cur][k]);

                }
        }

        return;
}}
```

2.2. Final MIPS code and execution result

```
.data
cities:
.word   0
.word   0
.word   8
.word   6
.word   2
.word   4
.word   6
.word   7
.word   1
.word   3
.word   9
.word   4
.word   2
.word   3

min_dist:  .float    987654321.0  # Maximum value
dist:     .space 196  # double distance[7][7]
optimal:   .space 28   # int optimal[7]
path:     .space 32   # int path[8]

$LC0: .asciiz  "Distance : "
$LC1 :.asciiz  "Path is : "
space :.asciiz  " "
enter: .asciiz "\n"

.text
main:
   li   $s5, 0   #i
   j   cal_dist
   nop

cal_dist:   # to calculate distance between cities
   addi   $s5, $s5, 1
   nop
```

```
    beq   $s5, 7, call_tsp
    nop

    li    $s6, 0
    nop

cal_dist_loop:  # the loop for cal_dist
    bgt   $s6, $s5, cal_dist
    nop
    la    $t0, cities   #cities t0
    nop

    sll   $t1, $s5, 3
    addu  $t2, $t1, $t0
    nop

    lw    $t3, 0($t2)   #cities[i].x = t3
    lw    $t5, 4($t2)
    nop

    sll   $t1, $s6, 3
    addu  $t2, $t1, $t0
    lw    $t4, 0($t2)   #cities[j].x = t4
    lw    $t6, 4($t2)
    nop

    subu  $t1, $t3, $t4   #sub
    subu  $t2, $t5, $t6   #sub
    nop

    mult  $t1, $t1
    mflo  $2
    mult  $t2, $t2
    mflo  $3
    addu  $2, $2, $3
    mtc1  $2, $f0
    nop
```

```
    cvt.s.w  $f0, $f0
    mov.s    $f12, $f0
    sqrt.s   $f0, $f12    # calculate sqrt
    nop

    la    $t0, dist        # $t0 = &dist
    mul   $t1, $s5, 7      # col processing; $t1 = i * 7
    add   $t1, $t1, $s6    # row processing; $ti = i * 7 + j
    sll   $t1, $t1, 2      # address processing; size of float
    add   $t0, $t0, $t1    # $t0 = &dist[i][j]
    s.s   $f0, 0($t0)      # $f4 = dist[i][j]
    nop

    la    $t0, dist
    mul   $t1, $s6, 7      # col processing; $t1 = j * 7
    add   $t1, $t1, $s5    # row processing; $ti = j * 7 + i
    sll   $t1, $t1, 2      # address processing; size of float
    add   $t0, $t0, $t1    # $t0 = &dist[j][i]
    s.s   $f0,  0($t0)
    nop
    addi  $s6, $s6, 1
    j      cal_dist_loop
    nop

call_tsp:   # set the first and last point of path and call tsp
    li     $t0, 1
    la     $t1, path
    sw     $t0, 0($t1)        # path[0] = 1
    sw     $t0, 28($t1)       # path[7] = 1
    la     $t2, optimal
    sw     $t0, 0($t2)        # optimal[0] = 1

    li     $a0, 0             # cur
    li     $a1, 0             # count
    li     $a2, 1             # visited
    mfc1   $zero, $f1         # sum
    jal    tsp                # call tsp
```

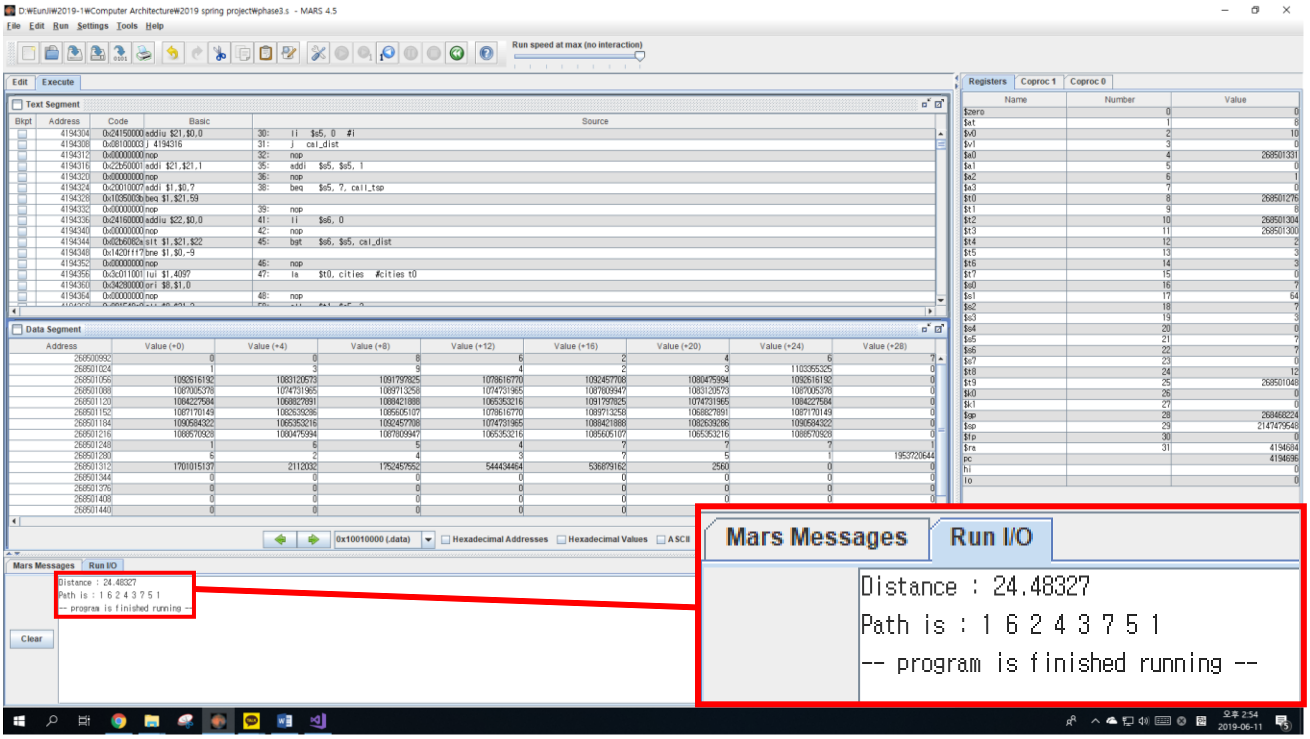
```
    nop

exit_program:   # exit the program
    la      $a0, $LC0
    li      $v0, 4
    syscall

    lwc1     $f12, min_dist
    li       $v0, 2
    syscall

    nop
    la      $a0, enter
    li      $v0, 4
    syscall
    nop

    jal     print_result
    nop

    li      $v0, 10
    syscall

print_result:  # to print path
    la      $a0, $LC1
    li      $v0, 4
    syscall

    la       $t0, path
    li       $t1, 0     # i = 0

print_result_loop:
    beq      $t1, 8, print_result_end   # if i >= 8 then print_path_tsp_end
    sll      $t2, $t1, 2
    add      $t2, $t0, $t2   # dist[i]

    lw       $a0, 0($t2)     # $a0 = dist[i]
    li       $v0, 1        # print integer
```

```
    syscall
    nop

    la     $a0, space
    li     $v0, 4
    syscall
    nop

    addiu  $t1, $t1, 1     # i++
    j      print_result_loop
    nop

print_result_end:
    jr     $ra          # jump to $ra
    nop

# $a0: current position, $a1: count, $a2: visited(bit masking), $f1: sum of
distance, $s0: index
tsp:  # Find optimal tsp path; use tail recursion
    beq   $a2, 127, tsp_end   # visit all cities then end tsp (127 == (1 << 7) -
1)
    nop

    li   $s0, 0

tsp_loop:  # for each cities
        addi   $s0, $s0, 1             # ++k
    beq     $s0, 7, tsp_loop_end    # condition for stopping loop
    nop

    # If we visit the city
    li     $t1, 1
    sllv   $s1, $t1, $s0     # $s1 (= 1 << k)
    and     $t1, $a2, $s1
    beq     $t1, $s1, tsp_loop

    # If the city is same with current position
    beq     $a0, $s0, tsp_loop
```

```
    nop

    la     $t0, dist        # $t0 = &dist
    mul     $t1, $a0, 7      # col processing; $t1 = cur * 7
    add     $t1, $t1, $s0    # row processing; $ti = cur * 7 + k
    sll     $t1, $t1, 2      # address processing; size of float
    add     $t0, $t0, $t1    # $t0 = &dist[cur][k]
    l.s     $f3, 0($t0)      # $f3 = dist[cur][k]
    add.s   $f0, $f3, $f1     # $f0 = sum + dist[cur][k]

    la     $v0, min_dist
    l.s     $f2, 0($v0)       # $f2 = min_dist

    c.lt.s   $f2, $f0       # if sum + dist[cur][k] > min_dist then tsp_loop
    bc1t   tsp_loop
    nop

    # 3 people team condition: check that 3rd should be visited before 7th city
    addi   $s2, $s0, 1   # $s2 (= k + 1)
    bne     $s2, 7, recursive_call
    andi   $v1, $a2, 4   # $v1 = visited & 1 << (3-1)
    bne     $v1, 0, recursive_call
    j tsp_loop
    nop

recursive_call:
    addi   $s3, $a1, 1       # $s3 (= count + 1)
    sll     $t8, $s3, 2
    la     $t1, optimal
    add     $t1, $t8, $t1
    sw     $s2, 0($t1)

    # save register before call tsp
    addi   $sp, $sp, -24
    sw     $ra, 20($sp)
    sw     $a0, 16($sp)
    sw     $a1, 12($sp)
    sw     $a2, 8($sp)
```

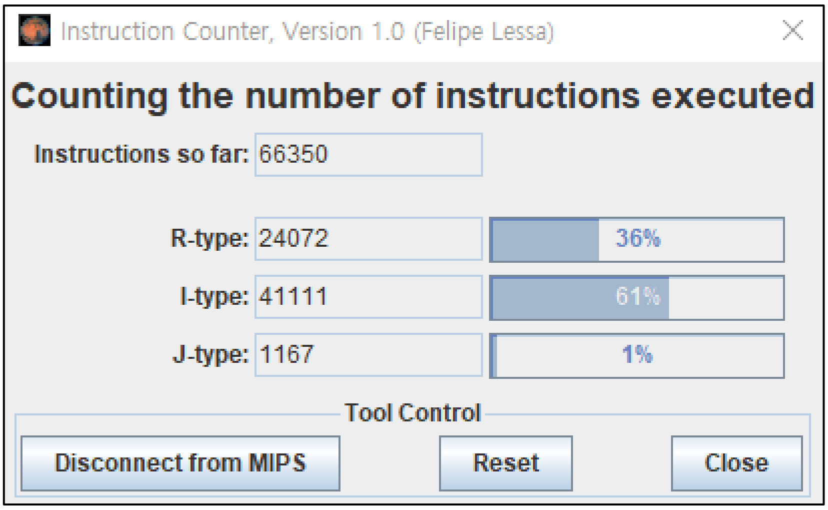
```
    s.s     $f1, 4($sp)
    sw      $s0, 0($sp)

    # update argument
    move    $a0, $s0
    move    $a1, $s3
    or      $a2, $a2, $s1

        mov.s   $f1, $f0
    jal     tsp         # recursive
    nop

    # reposit stack
    lw      $s0, 0($sp)
    l.s     $f1, 4($sp)
    lw      $a2, 8($sp)
    lw      $a1, 12($sp)
    lw      $a0, 16($sp)
    lw      $ra, 20($sp)
    addi    $sp, $sp, 24

    j       tsp_loop            # jump to tsp_for
    nop

tsp_end:
    la      $t0, dist
    mul     $t1, $a0, 7
    sll     $t1, $t1, 2
    add     $t0, $t0, $t1       # $t0 = &dist[cur][0]
    l.s     $f4, 0($t0)         # $f4 = dist[cur][0]
    add.s   $f8, $f1, $f4       # sum += dist[cur][0]
    la      $t9, min_dist       # $t9 = &min_dist
    l.s     $f6, 0($t9)         # $f6 = min_dist
    c.lt.s  $f8, $f6            # if sum < min_dist
    bc1t    update              # then goto update
    nop

    jr      $ra
```

```
    nop

update:
    s.s     $f8, 0($t9)         # min_dist = sum
    addi    $sp, $sp, -4
    sw      $ra, 0($sp)
    jal     update_path
    lw      $ra, 0($sp)
    addi    $sp, $sp, 4
    nop

tsp_loop_end:
   jr     $ra
   nop

update_path:
    li      $v0, 0   # i

update_path_loop:
    bge     $v0, 7, update_path_end
    sll     $v1, $v0, 2
    la      $t1, optimal
    add     $t1, $t1, $v1

    lw      $t2, 0($t1)
    la      $t3, path
    add     $t3, $t3, $v1
    sw      $t2, 0($t3)

    addiu   $v0, $v0, 1
    b        update_path_loop
    nop

update_path_end:
    jr    $ra
    nop
```

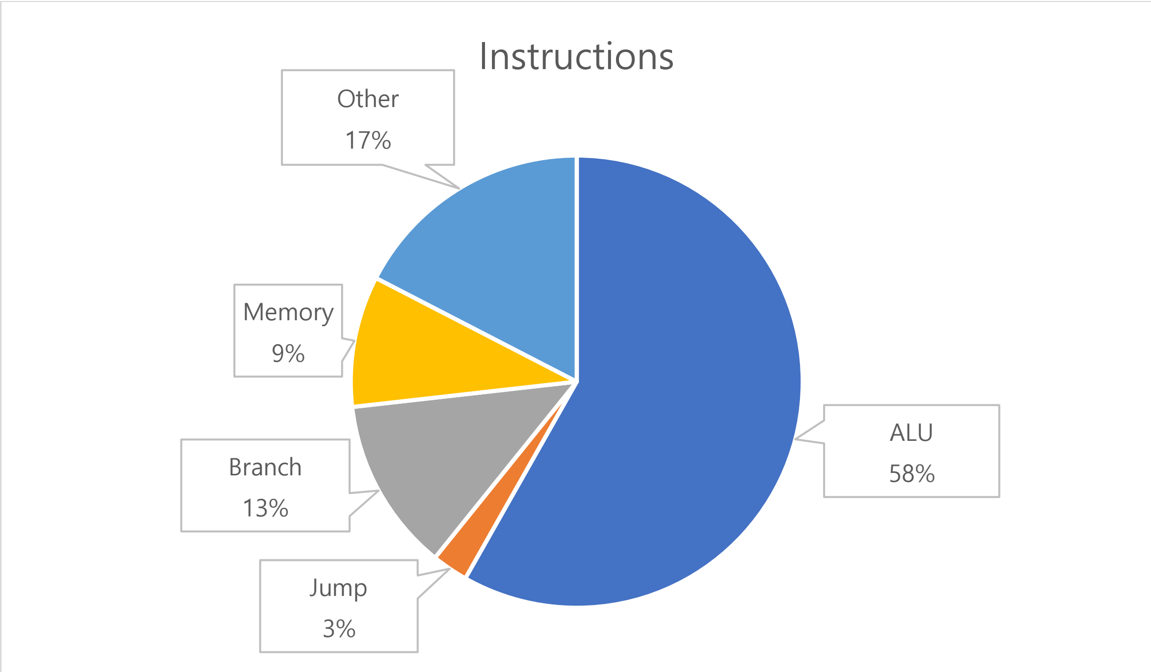


3. execution time and analysis

The number of total instruction is 66350:



The largest part of our MIPS instructions is ALU. Because this tsp problem's goal is visiting all cities within the shortest path, so it always tries to update the current distance or path. The percentages of each instructions are shown in next page.



To calculate the real execution time for MIPS, we need to know CPI, but there was no way to check the cycle by using SPIM simulator. The real execution time of our C code was 2ms, we know our computer has 1000 clock rate (Phase 2 show that result.) If our manually converted MIPS code would show the same performance with C code, the CPI is 0.03(IC is about 60000, clock rate is 1000), it seems unreliable – the actual CPI will be larger. Then why this unreliable result occurs?

First, the actual computer has multicore, so it reduces CPI, finally it affects the execution time. Second, we used visual studio to compile C code, this platform offers more effective instructions than the manually made one. In addition, the laptop we used embark the latest intel CPU, reasonably use better ISA than MIPS 32bit instruction. Third, our MIPS code occurs syscall, because of these syscall, throughput occurs and affects performance. By those reasons, we thought the actual performance and our MIPS instructions' performance has noticeable difference.