Digital Systems Design Report 2

(task 3-5)

Yeong Kyun Han

Benjamin Tan Jun Yi

Group number: 26

Implementing off-chip memory

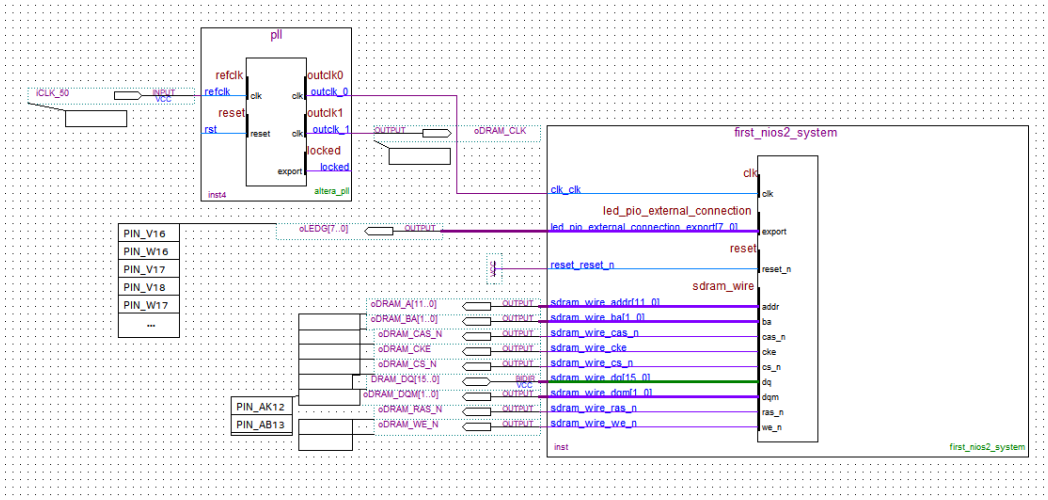

Figure 1

To provide enough time for the data to be available in NIOS, DRAM clock should lead the NIOS II clock. Thus, in the above connection, outclk_1 has a phase shift of -46 degrees compared to outclk_0. The results for the off-chip memory configuration can be seen in Table 1.

Note: All the following results for test3 have been scaled by 1/1024 so it can be printed

Application	Latency (ms)	Size (bytes)
Test1	1	14116
Test2	33	14116
Test3	7212	14200

Table 1

Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	1,636 / 32,070 (5 %)
Total registers	2697
Total pins	47 / 457 (10 %)
Total virtual pins	0
Total block memory bits	47,360 / 4,065,280 (1 %)
Total DSP Blocks	0 / 87 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	1 / 6 (17 %)
Total DLLs	0 / 4 (0 %)

Figure 2

A metric for the total FPGA resources used was found using the following formula:

$$\text{FPGA resources} = \frac{1}{3}\left(\frac{LE}{Total\ LE\ in\ the\ device} + \frac{EM}{Total\ EM\ in\ the\ device} + \frac{MB}{Total\ MB\ in\ the\ device}\right)$$

LE = 5%, EM = 0%, MB = 1%
Therefore, FPGA resources = 2%

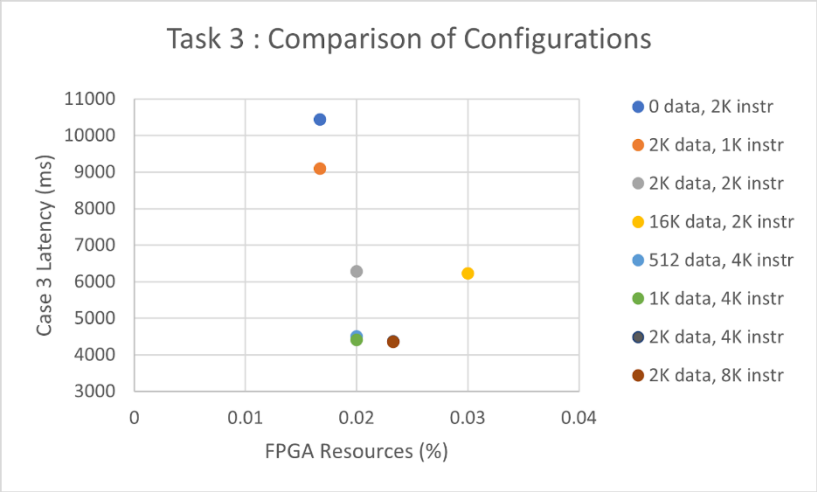

Figure 3

Under fixed instruction cache settings, increasing data cache shows small gain diminishing. However, when instruction cache is increased while data cache is fixed, latency decreases significantly until instruction cache reaches 4K bytes. After this point, gain diminishes again. Regarding resources, the more cache we allocate, the more FPGA resources are used. This kind of behaviour is as expected.

From the graph, configuration with 2K bytes data cache and 8K bytes instruction cache shows the lowest latency. Nevertheless, if we aim to further reduce the FPGA resource usage, the optimal setting would be 512 bytes data cache and 4K bytes instruction cache since latency trade-off is only around 3µs by doing so.

Compared to task 2 where on-chip memory was implemented, less resources are now being used throughout all configurations. In the case of our optimal configuration, approximately 40% of FPGA resource reduction is achieved by adding an external SDRAM block. It also increases the maximum memory our program can use, allowing us to run test3.

Implementing cos function

$$f(x) = \sum(0.5x + x2*\cos((x-128)/128))$$

The function above is implemented using the cos function in the math.h library to give the following results:

Application	Latency (ms)	Result	File size (bytes)
Test1	29	898.841309	37328
Test2	1110	35276.468750	37328
Test3	154834	4513214.00000	37364

Table 2

Table 2 shows that the latency and file sizes across all tests have increased which is expected given that the new equation also computes cosines. The accuracy of the final function is the checked by running the python code as shown in the appendix to get the following results:

Application	Result
Test1	898.8414322748456
Test2	35276.45073435458
Test3	4513172.8690318195

Table 3

We can see in table 3 that the python code gives different results from the nios processor, this is expected because the nios processor does support floating point arithmetic and is only emulating it in software.

Adding hardware support for multiplication

Family	Cyclone V
Device	5CSEMA5F31C6
Timing Models	Final
Logic utilization (in ALMs)	1,670 / 32,070 (5 %)
Total registers	2777
Total pins	47 / 457 (10 %)
Total virtual pins	0
Total block memory bits	65,024 / 4,065,280 (2 %)
Total DSP Blocks	3 / 87 (3 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	1 / 6 (17 %)
Total DLLs	0 / 4 (0 %)

Figure 4

The resources utilized by the new design are shown in *Figure 4*. It is done using a 4K bytes instruction cache and a 2K bytes data cache with 3 16-bit multipliers, resulting in 0.03333% FPGA resources used. The tests were executed with the new configuration to give the following results:

Application	Latency (ms)	Result	File size(bytes)
Test1	11	898	37328
Test2	445	35276	37328
Test3	58994	4513214	37364

Table 4

Comparing this to the previous task, we can see that the result and file sizes are still the same but the latency has reduced by an average of 61.3% across the 3 tests.

Cos implementations

To further improve the latency of the tests, different implementations of the cos function are tested using a configuration with a 16K bytes instruction and data cache since some of the implementations are expected to utilize more memory and a small cache might result in too many cache line replacements to be representative of the minimum achievable latency of each implementation. To avoid nonsensical implementations, only implementations with below 0.1% error in the result will be considered. The code for the following implementations can all be found in the appendix.

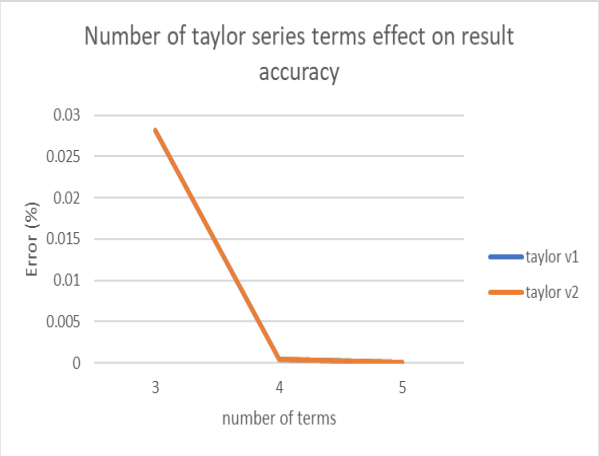

Figure 5

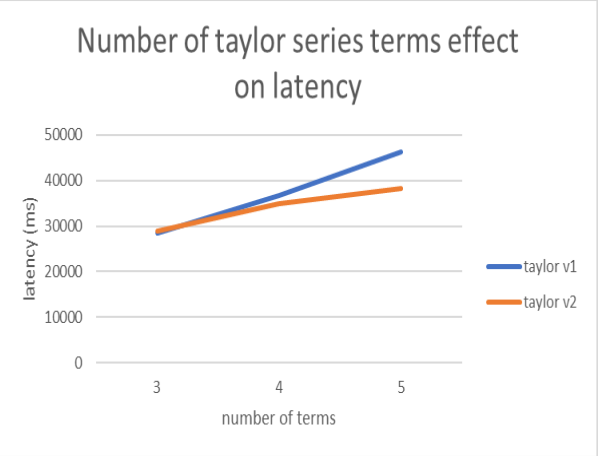

Figure 6

The first implementations tested are 2 different versions of the taylor series as shown in the *Figure 5* and *Figure 6*. We can see that there is no difference in result accuracy between the 2 implementations, but the latency of taylor v2 is less than taylor v1 when using 4 or more terms in the taylor series which is expected given that taylor v2 reuses the calculations for previous terms. However, since the 3 term implementation has the lowest acceptable error, we will only consider 3 term implementations for further tests.

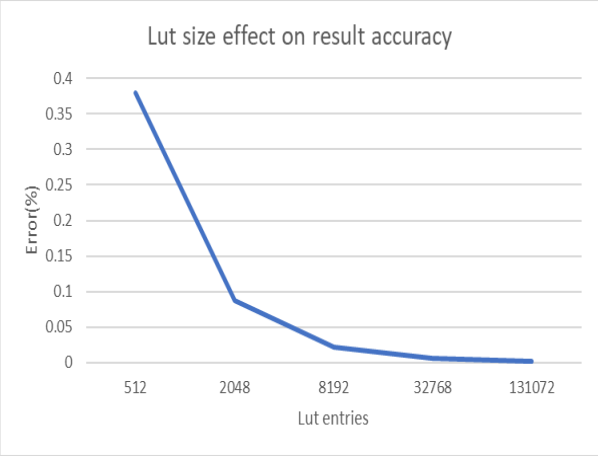

Figure 7

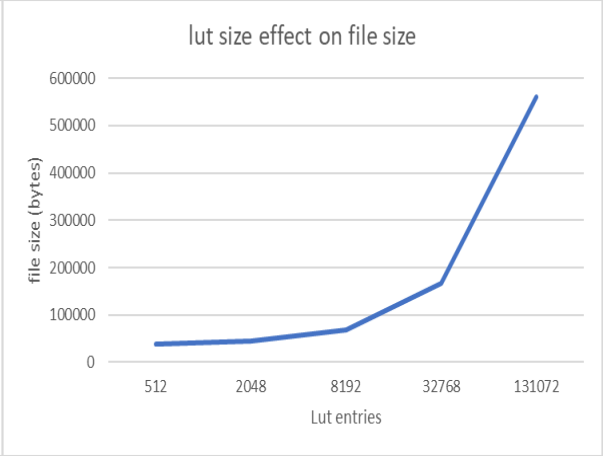

Figure 8

The next implementation tested is using a lookup table(lut) in software, we can see in the *Figure 7* that the result accuracy increases as the size of the lut increases but the memory utilized also increase since the lut requires (number of lut entries)*32 bits of run-time memory to store the cos values. It is also worth noting that the lut size has a minimal effect on the execution time with the largest recorded difference being 0.4% which is expected because the number of entries only affects the setup time which isn't timed.

Cos implementation	Latency (ms)	Result	Error (%)	File size (bytes)
Math.h cos	32589	4513214	0	37364
cosf	21996	4513215	0.00002	32832
Taylor v1 (3 terms)	19053	4514483	0.028117	25448
Taylor v2 (3 terms)	22176	4514483	0.028117	25428
Look-up table (2048 entires)	14070	4517164	0.087521	44136

Table 5

Table 5 compares all the different implementations of the cos computation. It can be seen that the lut implementation is the fastest but is also the least accurate and requires the largest file size, A graphical representation of the error and latency of the different implementations is shown in *Figure 8*.

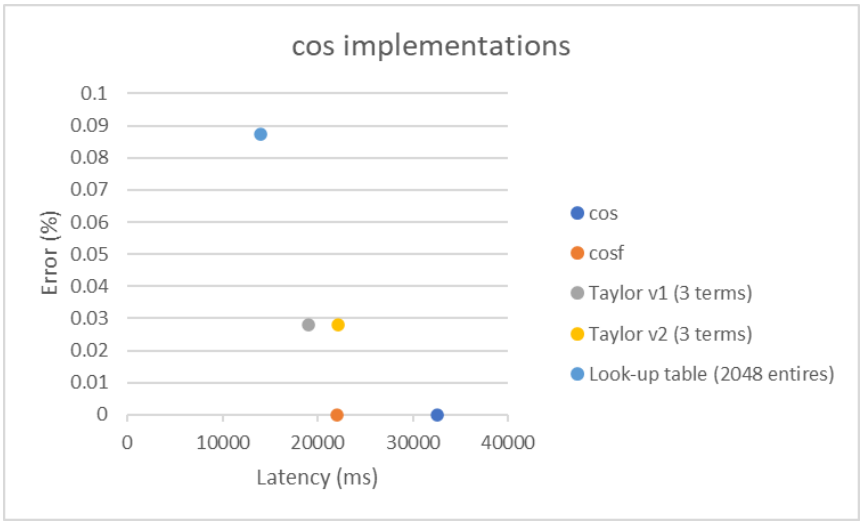

Figure 8

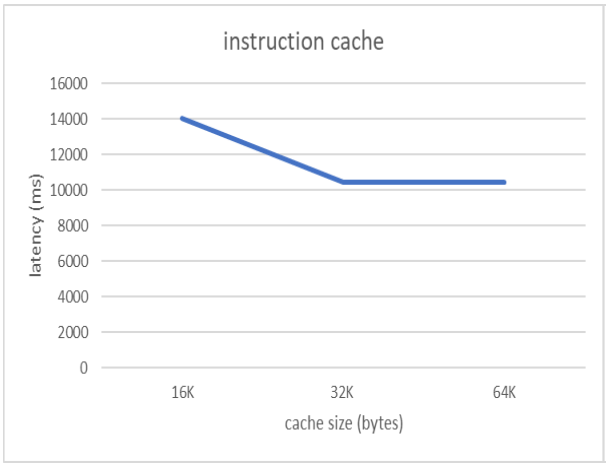

Figure 9

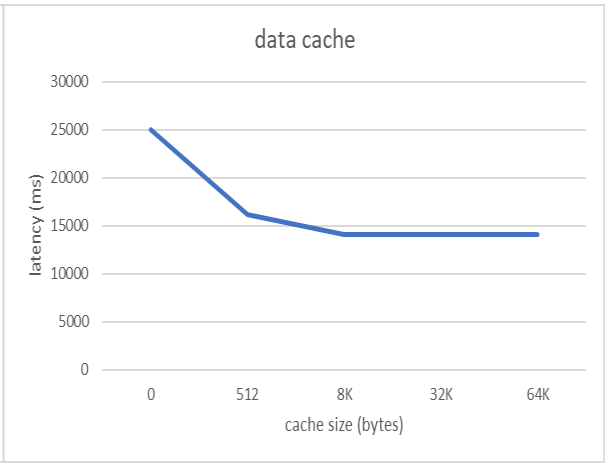

Figure 10

Figure 9 and *Figure 10* show the effect of the size of the data and instruction cache on the latency of the lut implementation. Increasing the cache sizes lowers the latency as expected since it reduces the amount of off-chip memory accesses. We can see from *Figure 11* that the configuration with a 32K bytes instruction cache and a 64K bytes data cache is the fastest.

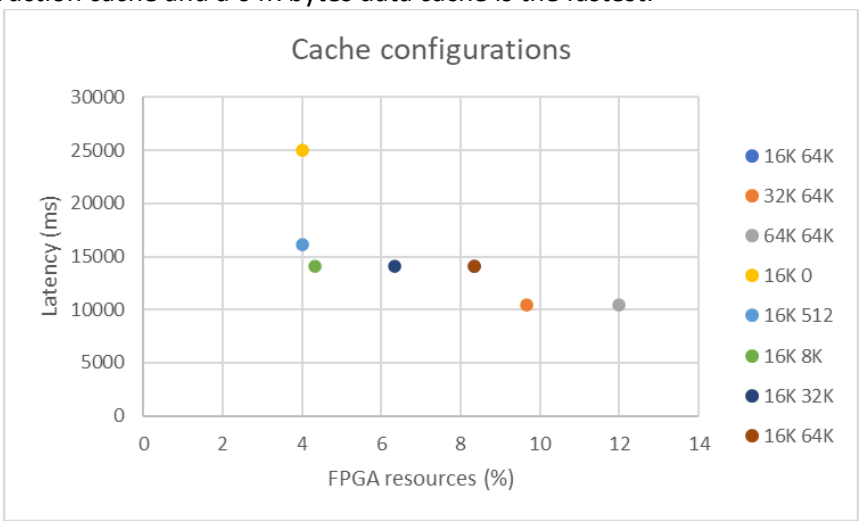

Figure 11

Software Optimizations

Different software optimizations are then implemented to further decrease the latency, the code for the optimizations can be found in the appendix.

Software optimizations	Latency (ms)	Speedup (%)
original	10445	0
Result v2	9822	4.65
Temp2	9095	11.7076
Inverse pi constant	8919	13.4162
No lookup function call	10227	0.7184
Loop unrolling	10166	1.3106
result v3 + temp2 + inverse pi constant + no lookup function call	6782	34.1617
Loop unrolling + result v3 + temp2 + inverse pi constant + no lookup function call	7166	30.4339

Table 1

Table 4 shows the data for the different optimizations attempted, all of them decrease the latency as expected following the reasoning below:

1) Result v2 – reduces the array accesses

2) Temp v2 – uses multiply instead of divide which is a faster instruction

3) Inverse pi constant – removes 1/pi calculations

4) Removed lookup function call – removes the branch needed for the function call

5) Loop unrolling – reduces the branches needed for the loop

Conclusion

The final configuration uses 3 16-bit multipliers, a 32K bytes instruction cache and a 64K bytes data cache to achieve a latency of 6782ms using the lut implementation of cos. It can also be seen that both hardware and software optimizations can be used to reduce latency and should both be applied to get an optimal implementation.

Appendix

Python result check

```python
def generatevector(N):

    y = [0]
    for index in range(1,N):
        y += [y[index - 1] + param1]
    return y

param1 = 1/1024
param2 = 261121

x = generatevector(param2)

sum = 0

i = 0
while i < param2:
    sum = sum + x[i] + x[i] * x[i]
    i += 1


print(sum)
```

Cos Implementations

1)Math.h cos

```
float sumVector(float x[], int M)
{
    float result=0;

    for(int i=0; i<M; i++){
        float temp = (x[i]-128)/128;
        result += 0.5*x[i] + x[i]*x[i]*cos(temp);
    }
    return result;
}
```

2)Cosf

```
float sumVector(float x[], int M)
{
```

```
    float result=0;

    for(int i=0; i<M; i++){

        float temp = (x[i]-128)/128;
        result += 0.5*x[i] + x[i]*x[i]*cosf(temp);

    }
    return result;
}
```

3)Taylor series v1

```
float sumVector(float x[], int M)
{
    float result=0;

    for(int i=0; i<M; i++){

        float temp = (x[i]-128)/128;
        float cosx = 1  - temp*temp/2 + temp*temp*temp*temp/24 -
temp*temp*temp*temp*temp*temp/720;
        result += 0.5*x[i] + x[i]*x[i]*cosx;
    }
    return result;
}
```

4)Taylor series v2

```
float sumVector(float x[], int M)
{
    float result=0;

    for(int i=0; i<M; i++){

        float temp = (x[i]-128)/128;
        float cosx1 = temp*temp/2;
        float cosx2 = cosx1*temp*temp/12;
        float cosx3 = cosx1*cosx2/15;
        float cosx4 = cosx2*cosx2/70;
        float cosx = 1 - cosx1 + cosx2;// - cosx3;// + cosx4;
        result += 0.5*x[i] + x[i]*x[i]*cosx;
    }
    return result;
}
```

5)Lookup table

```
#define MAX_CIRCLE_ANGLE      2048
#define HALF_CIRCLE_ANGLE (MAX_CIRCLE_ANGLE/2)
#define QUARTER_CIRCLE_ANGLE (MAX_CIRCLE_ANGLE/4)
#define MASK_MAX_CIRCLE_ANGLE (MAX_CIRCLE_ANGLE - 1)
#define PI 3.14159265358979323846f
#define IPI 0.31830988618379067153f

float cos_table[MAX_CIRCLE_ANGLE];

float lookup_cos(float n)
```

```
{
    float f = n * HALF_CIRCLE_ANGLE / PI;

    int i = (int)f;
    if (i < 0)
        return cos_table[-i];
    else
        return cos_table[i];
}

float sumVector(float x[], int M)
{
    float result=0;

    for(int i=0; i<M; i++){
        float temp = (x[i]-128)/128;

        float cosx = lookup_cos(temp);
        result += 0.5*x[i] + x[i]*x[i]*cosx;
    }

    return result;
}

int main()
{
    //Build cos table
    for (int j=0 ; j<MAX_CIRCLE_ANGLE ; j++)
    {
        cos_table[j] = (float)cosf((double)j * PI /
HALF_CIRCLE_ANGLE);
    }
```

Software Optimizations

1)result v2

```
result += x[i]*(0.5+x[i]*cosx);
```

2)temp v2

```
float temp = (x[i]-128)*0.0078125;
```

3)1/pi as a constant

```
#define IPI 0.31830988618379067153f
float f = n * HALF_CIRCLE_ANGLE*IPI;
```

4)removed lookup function call

```
float sumVector(float x[], int M)
{
    float result=0;

    for(int i=0; i<M; i++){
        float temp = (x[i]-128)/128;

        float f = temp * HALF_CIRCLE_ANGLE / PI;

        float cosx;
        int temp2 = (int)f;
```

```
        if (temp2 < 0)
            cosx = cos_table[-temp2];
        else
            cosx = cos_table[temp2];

        result += 0.5*x[i] + x[i]*x[i]*cosx;
    }
    return result;
}
```

5)Loop unrolling

```
float sumVector(float x[], int M)
{
    float result=0;

    for(int i=0; i<M; i+=2){
        float temp = (x[i]-128)/128;

        float f = temp * HALF_CIRCLE_ANGLE / PI;

        float cosx1;
        int temp2 = (int)f;
        if (temp2 < 0)
            cosx1 = cos_table[-temp2];
        else
            cosx1 = cos_table[temp2];

        temp = (x[i+1]-128)/128;

        f = temp * HALF_CIRCLE_ANGLE / PI;

        float cosx2;
        temp2 = (int)f;
        if (temp2 < 0)
            cosx2 = cos_table[-temp2];
        else
            cosx2 = cos_table[temp2];

        result += 0.5*x[i+1] + x[i+1]*x[i+1]*cosx2 + 0.5*x[i] +
x[i]*x[i]*cosx1;
    }
    return result;
}
```

6)Final implementation

```
#include <stdlib.h>

#include <sys/alt_stdio.h>
#include <sys/alt_alarm.h>
#include <sys/times.h>
#include <alt_types.h>
#include <system.h>
#include <stdio.h>
#include <unistd.h>
#include <math.h>
```

```
//Test case 3
#define test 3
#define step 1/1024.0
#define N 261121

//lookup table
#define MAX_CIRCLE_ANGLE      2048
#define HALF_CIRCLE_ANGLE (MAX_CIRCLE_ANGLE/2)
#define QUARTER_CIRCLE_ANGLE (MAX_CIRCLE_ANGLE/4)
#define MASK_MAX_CIRCLE_ANGLE (MAX_CIRCLE_ANGLE - 1)
#define PI 3.14159265358979323846f
#define IPI 0.31830988618379067153f

float cos_table[MAX_CIRCLE_ANGLE];

// Generates the vector x and stores it in the memory
void generateVector(float x[N])
{
    int i;
    x[0] = 0;
    for (i=1; i<N; i++)
        x[i] = x[i-1] + step;
}

float sumVector(float x[], int M)
{
    float result=0;

    for(int i=0; i<M; i++){
        float temp = (x[i]-128)*0.0078125;

        float f = n * HALF_CIRCLE_ANGLE*IPI;// / PI;
        float cosx;
        int cont = (int)f;
        if (cont < 0)
            cosx = cos_table[-i];
        else
            cosx = cos_table[i];

        result += x[i]*(0.5+x[i]*cosx);
    }

    return result;
}

int main()
{
```

```
    printf("Test %d!\n",test);

    //Build cos table
    for (int j=0 ; j<MAX_CIRCLE_ANGLE ; j++)
    {
        cos_table[j] = (float)cosf((double)j * PI / HALF_CIRCLE_ANGLE);  //
eg: 1/512 *2pi
    }

    // Define input vector
    float x[N];

    // Returned result
    float y;

    generateVector(x);

    // The following is used for timing
    char buf[50];
    clock_t exec_t1, exec_t2;

    exec_t1 = times(NULL); // get system time before starting the process

    y = sumVector(x, N);

    exec_t2 = times(NULL); // get system time after finishing the process

    alt_printf(" proc time = 0x%x ticks \n", (int) ((exec_t2-exec_t1)/1));

    printf("done \n");
    int i;
    for (i=0; i<10; i++)
        y = y/2.0;

    printf("result: %d \n", (int)y);
    return 0;
}
```