EE3-05 Digital System Design

Coursework Report 2: Task 3-5

Abstract— **This is the second report out of three for EE3-05 Digital System Design's (2015-2016) coursework of implementing, then accelerating a mathematical algorithm on an Altera Cyclone III FPGA. This second report will cover implementing an off-chip SDRAM to increase the amount of data that our test bench can process, as well as using different hardware configurations to accelerate the algorithm used. Multiple sections on benchmarking the performance of our system are included**

Jeremy Chan*
Department of Electrical and Electronic Engineering
Imperial College
London, United Kingdom
jc4913@ic.ac.uk | 00818433

Chak Yeung Dominic Kwok*
Department of Electrical and Electronic Engineering
Imperial College
London, United Kingdom
cyk113@ic.ac.uk | 00827832

*These authors contributed equally to this work

Table of Contents

I. Introduction

This report mainly focuses on finding the best hardware configuration for achieving a minimum latency when calculating a complex mathematical expression. Extensive benchmarks are provided to justify the concluding hardware configuration – along with graphs to show trends in latency vs resources used. Apart from latency benchmarks, a section on the accuracy of the final results are provided, when compared against a double precision floating point implementation using MATLAB. Furthermore, discussion on the implementation of this complex mathematical expression is also provided, as well as possible further research. The hardware configurations and test bench source are provided in the appendix.

II. System Design

A. *Generating the QSys system*

The QSys system was generated as specified (8MB SDRAM), with a minor change in the clock arrangement of the PLL – instead of connecting the delayed clock to the SDRAM Controller, it was connected to the SDRAM to ensure the NIOS II system clock led the SDRAM clock by 3ns.

From the discussion of results in section II.D, the on-chip memory was removed. The LED PI/O was also not used by our design and hence removed as well. The QSys system is shown in Appendix.B.

B. *Compiling the Quartus system*

The Quartus system was generated as specified from the QSys system in section II.A. The PLL was moved to within the QSys project to reduce clutter in the board design file. The SDRAM components were also connected to the SDRAM Controller. As the LED PI/O was unused, it was removed and hence left unconnected; Quartus will automatically remove all unconnected wires during compilation anyway.

C. *Creating the Eclipse project*

The Eclipse project was generated as specified, with the small C library switched off. From the previous report, `enable_lightweight_device_device_driver_api` was also turned back on as there was now enough memory. C++ support was not turned on as there C supported our testbench capably.

D. *Removing the On Chip Memory*

After the discussion of results in report 1, the on-chip memory was removed to test if this affected latency.

Latency was measured using the system clock. The output is in ticks, defined in `system.h`.

From `system.h`, ticks are defined as

```
#define TIMER_TICKS_PER_SEC 1000.0
```

Hence, 1 tick = 1 ms.

Table II shows the latency before and after removing the on-chip memory. The configuration used for the test case was as follows:

TABLE I.

Configuration	Value
Algorithm Used	$\sum x + x^2$
Hardware Multiplication	None
Instruction Cache Size	2KB
Test Case	3

TABLE II.

Ticks	Case 1	Case 2	Case 3
Before Removal	6	282	28221
After Removal	5	281	28148

Since the latency improves after removing the on-chip memory, all further testing was done using this configuration. This also saves on system resources, as seen in the Table III.

TABLE III.

Resource	With on-chip memory	Without on-chip memory
Logic Elements	3207	2999
Embedded Multiplier	0	0
Memory Bits	422272	29056

III. Hardware Configuration for Testing

A. *Task 3*

TABLE IV.

Configuration	Value
Algorithm Used	$\sum x + x^2$
Hardware Multiplication	None
Instruction Cache Size	Variable
On-Chip Memory	None

B. *Task 4*

TABLE V.

Configuration	1	2
Algorithm Used	$\sum \frac{x}{2} + x^2 \cos(\lfloor \frac{x}{4} \rfloor - 32)$	$\sum \frac{x}{2} + x^2 \cosf(\lfloor \frac{x}{4} \rfloor - 32)$
Hardware Multiplication	None	
Instruction Cache Size	Variable	
On-Chip Memory	None	

C. *Task 5*

TABLE VI.

Configuration	1	2
Algorithm Used	$\sum \frac{x}{2} + x^2 \cos(\lfloor \frac{x}{4} \rfloor - 32)$	$\sum \frac{x}{2} + x^2 \cosf(\lfloor \frac{x}{4} \rfloor - 32)$
Hardware Multiplication	Embedded Multiplier, Logic Elements	
Instruction Cache Size	Variable	
On-Chip Memory	None	

IV. Results

A. *Latency, with variable instruction cache size*

The cache size was also varied to investigate the result of changing the instruction cache size on latency. The lowest amount of cache tested was 2KB; the max 32KB. 64KB was not tested as there was not enough memory bits on the board to accommodate 64KB of instruction cache.

a) *Task 3*

TABLE VII.

Cache Size	2KB	4KB	8KB	16KB	32KB
Case 1	5	4	4	4	4
Case 2	254	215	213	213	213
Case 3	26808	21552	21329	21329	21330

b) Task 4

TABLE VIII.

Cache Size	2KB	4KB	8KB	16KB	32KB
Case 1	78	74	59	57	56
Case 2	3823	3636	2963	2825	2800
Case 3	382038	362601	296649	282920	288024

c) Task 5

Task 5 required the testing of three different configurations of hardware multiplication.

Table IX uses no hardware multiplication. Therefore, it must be identical to Task 4 as they use the same hardware.

Table X uses Embedded Multipliers.

Table XI uses Logic Elements.

TABLE IX.

Cache Size	2KB	4KB	8KB	16KB	32KB
Case 1	78	74	59	57	56
Case 2	3823	3636	2963	2825	2800
Case 3	382038	362601	296649	282920	288024

TABLE X.

Cache Size	2KB	4KB	8KB	16KB	32KB
Case 1	55	41	30	27	27
Case 2	2272	2067	1482	1366	1356
Case 3	227074	206562	148078	136506	136496

TABLE XI.

Cache Size	2KB	4KB	8KB	16KB	32KB
Case 1	46	42	30	28	28
Case 2	2299	2094	1509	1393	1384
Case 3	229874	209334	150830	139247	138379

B. *FPGA Resource Usage*

Table XII denotes our resource usage on the Cyclone III FPGA, after the addition of the SDRAM, and after compilation and fitting using Quartus.

Table XIII changes the Hardware Multiplication from None to Embedded Multipliers, otherwise identical to Table IX.

Table XIV changes the Hardware Multiplication from None to Logic Elements, otherwise identical to Table IX.

Resource usage was extracted from the Quartus compilation report – expressed in terms of Logic Elements (LE), Embedded Multipliers (EM), and Memory Bits (MB). The DE0 board consists of a total of 15408 LE's, 112 EM's, and 516096 MB's.

The equation to quantitate these measurements is expressed in terms of resource usage (RU) (lower means less resources used):

$$RU = \frac{1}{3}\left(\frac{LE}{Total\ LE} + \frac{EM}{Total\ EM} + \frac{MB}{Total\ MB}\right)$$

TABLE XII.

Cache Size	2KB	4KB	8KB	16KB	32KB
LE	2994	2991	2993	2991	2989
EM	0	0	0	0	0
MB	29056	46720	81920	152064	291840
RU	0.084	0.095	0.118	0.163	0.253

TABLE XIII.

Cache Size	2KB	4KB	8KB	16KB	32KB
LE	3214	3218	3218	3216	3218
EM	4	4	4	4	4
MB	29056	46720	81960	152064	291840
RU	0.100	0.112	0.134	0.180	0.270

TABLE XIV.

Cache Size	2KB	4KB	8KB	16KB	32KB
LE	3345	3348	3343	3343	3353
EM	0	0	0	0	0
MB	29056	46720	81960	152064	291840
RU	0.091	0.103	0.125	0.171	0.261

Given using LE's as hardware multiplication adds an average of 355 LE's, using embedded multipliers save approximately 89 LE's per embedded multiplier.

C. *NIOS II .elf size*

TABLE XV shows the size of the initialized array (consisting of 32 bit floats) for different test cases.

The size of the compiled .elf was obtained using the command '`nios2-stackreport`' using the NIOS II Command Shell. '`nios2-stackreport`' is advantageous to '`nios2-elf-size`' as it provides a breakdown of the total size.

Since tasks 4 and 5 used the same algorithm, their .elf size was the same. Hence, only tasks 3 and 4 will be shown – the differences get negligible at large array sizes.

TABLE XV.

	Case 1	Case 2	Case 3
Array Size	52	2551	255001
Compiled Size	208B	9.96KB	996KB

a) *Task 3*

TABLE XVI.

	Case 1	Case 2	Case 3
Program Size	38KB	48KB	1035KB
Free Size for stack+heap	8151KB	8141KB	7155KB

b) *Task 4/5*

TABLE XVII.

	Case 1	Case 2	Case 3
Program Size	66KB	76KB	1063KB
Free Size for stack+heap	8122KB	8113KB	7126KB

D. *Throughput*

The throughput was measured in terms of results per second, then converted to bytes per second. The array consists of 32-bit floats. As previous results showed that using an instruction cache size of 32KB, the following throughput results use an instruction cache size of 32KB.

a) *Task 3*

TABLE XVIII.

	Case 1	Case 2	Case 3
Array Size	52	2551	255001
Latency (ms)	4	213	21330
Throughput (Bps)	3328.00	3065.99	3060.49

b) Task 4

TABLE XIX.

	Case 1	Case 2	Case 3
Array Size	52	2551	255001
Latency (ms)	56	2800	288024
Throughput (Bps)	237.71	233.23	226.65

c) Task 5

TABLE XX.

	Case 1	Case 2	Case 3
Array Size	52	2551	255001
Latency (ms)	27	1356	136496
Throughput (Bps)	493.04	481.60	478.26

TABLE XXI.

	Case 1	Case 2	Case 3
Array Size	52	2551	255001
Latency (ms)	28	1384	138379
Throughput (Bps)	475.43	471.86	471.75

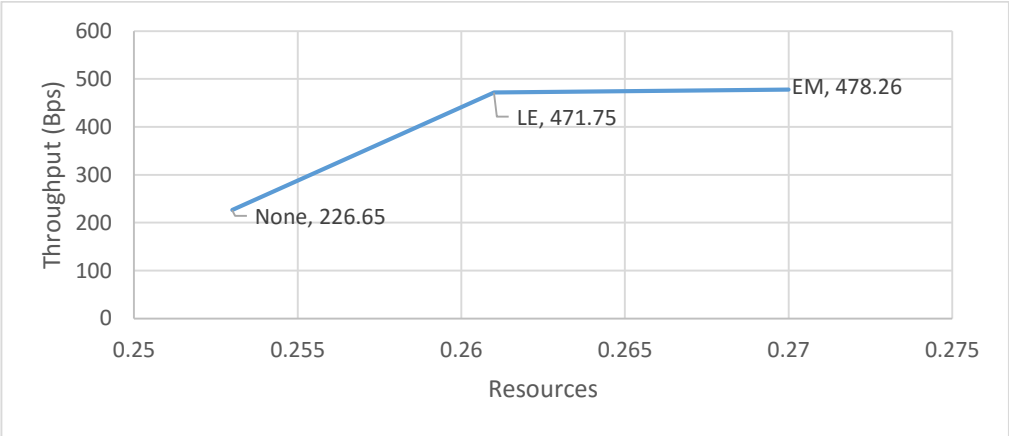


Figure 1: Throughput vs Resources Used

Figure 1 shows a graph of throughput in Bps against resources used. As performance is the priority, Embedded Multipliers are chosen as the hardware multiplication method.

E. Program Output/Error with MATLAB implementation

Since Task 4 and 5 only differ by hardware implementation, their results are the same. Errors are non-negligible; more testing was hence done.

a) Task 3

TABLE XXII.

	MATLAB (double/double)	NIOS II	Error with MATLAB	%Error with MATLAB
Case 1	1144780	1144780	0.0000	0.0000
Case 2	55629019.25	55630340.00	1,320.7500	0.0024

Case 3	5559670140.07	5580554240.00	20,884,099.9300	0.3756

b) Task 4/5

TABLE XXIII.

	MATLAB (double/double)	NIOS II	Error with MATLAB	%Error with MATLAB
Case 1	57879.8730	57879.8672	-0.0058	0.0000
Case 2	-126818.14	-76973.39	49,844.7500	39.3041
Case 3	-12774366.3	37022532	49,796,898.3000	389.8189

V. Discussion of Results

A. *Error with MATLAB*

Since errors with MATLAB are non-negligible, a variety of number types were tested – (double/float) means the formula uses doubles, but the intermittent result is casted to floats. The following function was used to print out all the intermittent values in the summation.

```
void sumVector_cos(float x[], double xd[], unsigned int M)
{
        float ff=0, df=0, naive=0;
        double fd=0, dd=0;
        int i=0;
        for (i=0; i<M; i++){
                ff += (x[i]/2) + (x[i]*x[i]*cosf(floor(x[i]/4)-32));
                df += (xd[i]/2) + (xd[i]*xd[i]*cos(floor(xd[i]/4)-32));
                fd += (x[i]/2) + (x[i]*x[i]*cosf(floor(x[i]/4)-32));
                dd += (xd[i]/2) + (xd[i]*xd[i]*cos(floor(xd[i]/4)-32));
                naive += (x[i]/2) + (x[i]*x[i]*cos(floor(x[i]/4)-32));
                printf("index: %d, ff: %f, df: %f, fd: %f, dd: %f, naive: %f\n", i,
                result1, result2, result3, result4, result5);
        }
}
```

A different vector was also generated for use when double input was required.

```
void generateVector(float* x, unsigned int n, float s)
{
        int i;
        x[0] = 0;
        for (i=1; i<n; i++){
                x[i] = x[i-1] + s;
        }
}
void generateVector_d(double* x, unsigned int n, double s)
{
        int i;
        x[0] = 0;
        for (i=1; i<n; i++){
                x[i] = x[i-1] + s;
        }
}
```

a) Graph - Case 1

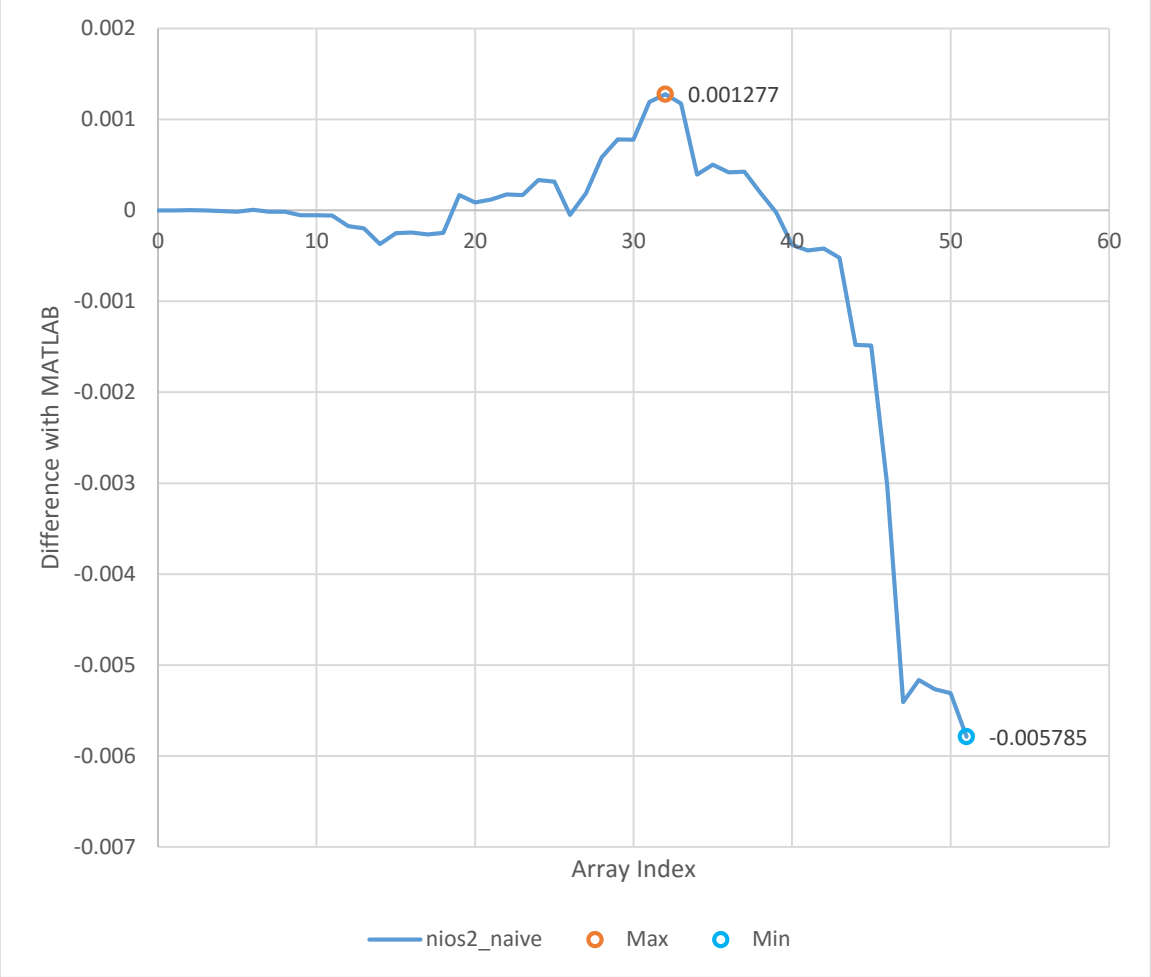


Figure 2: Case 1 Difference with MATLAB

From errors while using the previous algorithm, we suspected that Task 4 and 5 will have the same monotonic trend in the error with MATLAB. However, the error has the makings of a sinusoidal waveform; further testing was performed using different combinations of precisions.

b) Graph - Case 2

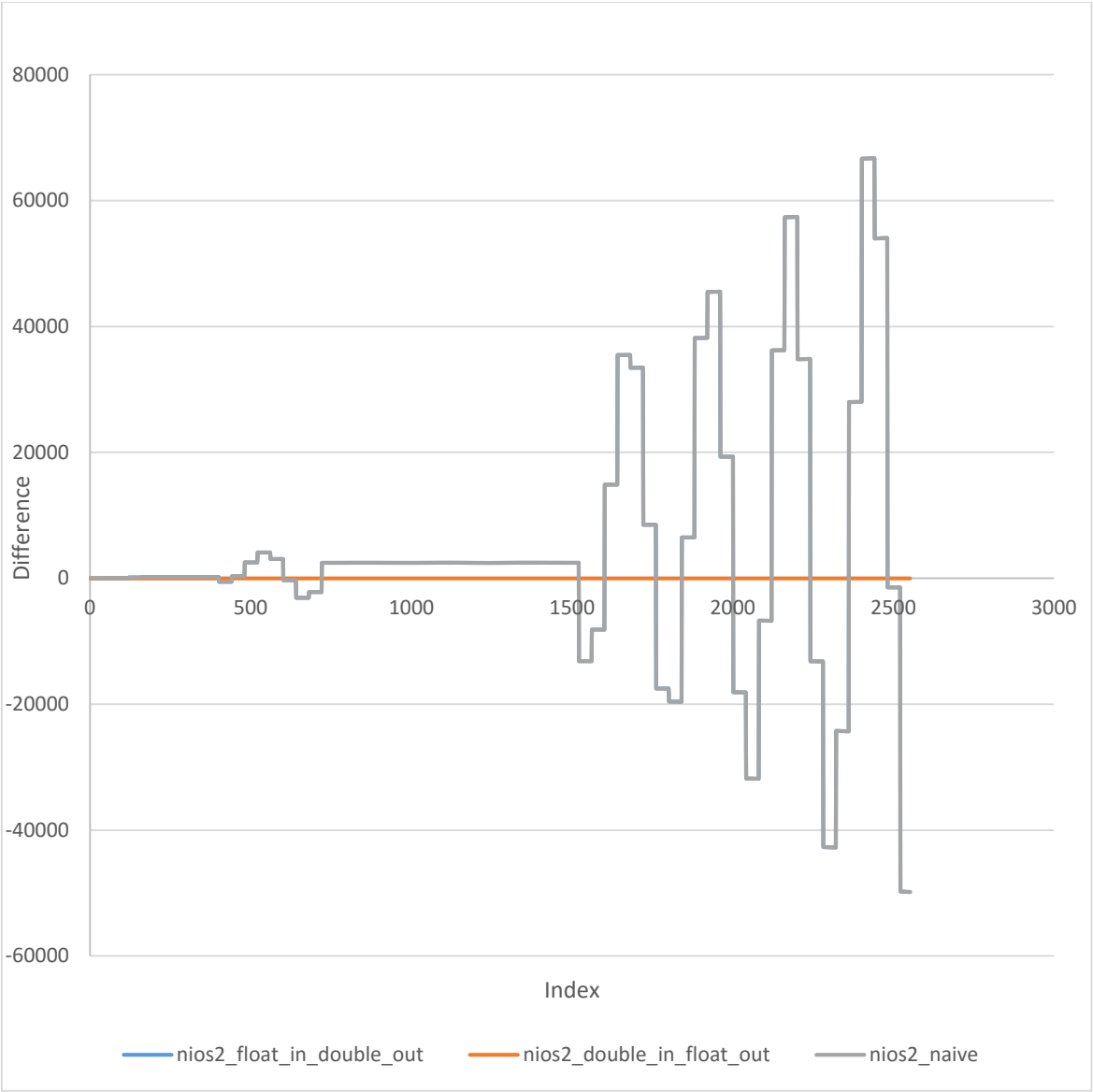


Figure 3: Case 2 Difference with MATLAB

c) *Graph - Case 3*

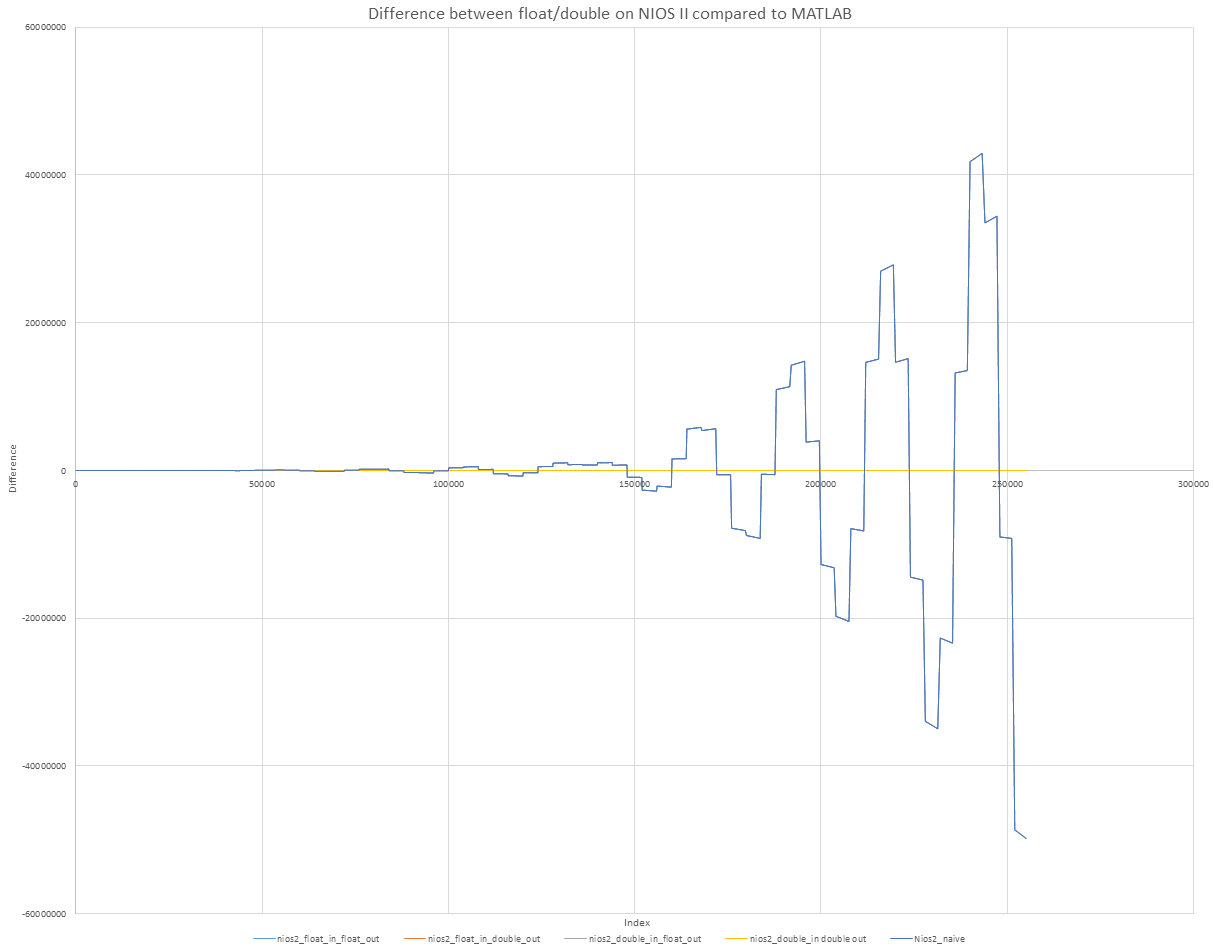


Figure 4: Case 3 Difference with MATLAB

d) Tables of Results

1) Task 3

TABLE XXIV.

Implementation (type in/out)	MATLAB (double/double)	NIOS II (double/double)	NIOS II (double/float)	NIOS II (float/double)	NIOS II (float/float)	NIOS II (original)
Case 1	1144780	1144780	1144780	1144780	1144780	1144780
Case 2	55629019.25	55629019.25	55628960.00	55630320.01	55630340.00	55630340.00
Case 3	5559670140.07	5559670140.07	5559633920.00	5580575477.79	5580554240.00	5580554240.00

TABLE XXV.

Error with MATLAB	MATLAB (double/double)	NIOS II (double/double)	NIOS II (double/float)	NIOS II (float/double)	NIOS II (float/float)	NIOS II (original)
Case 1	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Case 2	0.0000	0.0000	-59.2500	1,300.7600	1,320.7500	1,320.7500
Case 3	0.0000	0.0000	-36,220.0700	20,905,337.7200	20,884,099.9300	20,884,099.9300

TABLE XXVI.

Abs %Error with MATLAB	MATLAB (double/double)	NIOS II (double/double)	NIOS II (double/float)	NIOS II (float/double)	NIOS II (float/float)	NIOS II (original)
Case 1	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Case 2	0.0000	0.0000	0.0001	0.0023	0.0024	0.0024
Case 3	0.0000	0.0000	0.0007	0.3760	0.3756	0.3756

2) Task 4/5

TABLE XXVII.

Implementation (type in/out)	MATLAB (double/double)	NIOS II (double/double)	NIOS II (double/float)	NIOS II (float/double)	NIOS II (float/float)	NIOS II (original)
Case 1	57879.8730	57879.8730	57879.8672	57879.8687	57879.8672	57879.8672
Case 2	-126818.14	-126818.14	-126819.41	-76972.44	-76973.14	-76973.39
Case 3	-12774366.3	-12774366.29	-12774366.38	37022500	37022500	37022532

TABLE XXVIII.

Error with MATLAB	MATLAB (double/double)	NIOS II (double/double)	NIOS II (double/float)	NIOS II (float/double)	NIOS II (float/float)	NIOS II (original)
Case 1	0.0000	0.0000	-0.0058	-0.0043	-0.0058	-0.0058
Case 2	0.0000	0.0000	-1.2700	49,845.7000	49,845.0000	49,844.7500
Case 3	0.0000	0.0100	-0.0800	49,796,866.3000	49,796,866.3000	49,796,898.3000

TABLE XXIX.

Abs %Error with MATLAB	MATLAB (double/double)	NIOS II (double/double)	NIOS II (double/float)	NIOS II (float/double)	NIOS II (float/float)	NIOS II (original)
Case 1	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
Case 2	0.0000	0.0000	0.0010	39.3049	39.3043	39.3041
Case 3	0.0000	0.0000	0.0000	389.8187	389.8187	389.8189

e) Discussion on Case 1-3 for Task 4

As shown in the graphs, the error does indeed take the form of a sinusoidal wave. This is due to error caused by using single precision floating point numbers on the computation of cosine.

Calculating intermittent results using double and then casting to a float generated an extremely accurate answer, whereas calculating cosines using floats generated an answer that was 389% off.

Case 3 had the largest error when using the original float in float out method. This is due to increasing errors when summing together a large floating point number with a much smaller floating point number, as the normalization of the mantissa will cause a lot of LSB's to be lost. In an extreme case, $a + b = a, when\ a \gg b$. In Figure 4, the error increases with the number of summations.

B. *Ability to compile for different test cases*

As there is now an 8MB SDRAM instead of a 48KB on-chip memory, all 3 test cases compile when the array is defined as a global variable – the check for memory is done at compile time.

C. *Instruction Cache*

Increasing the size of the instruction cache size was shown to affect the latency. All graph results are normalized – the longest latency is defined to have a normalized latency of 1. Hence, a smaller normalized latency is better. All data labels have been normalized to 3d.p.

a) Task 3

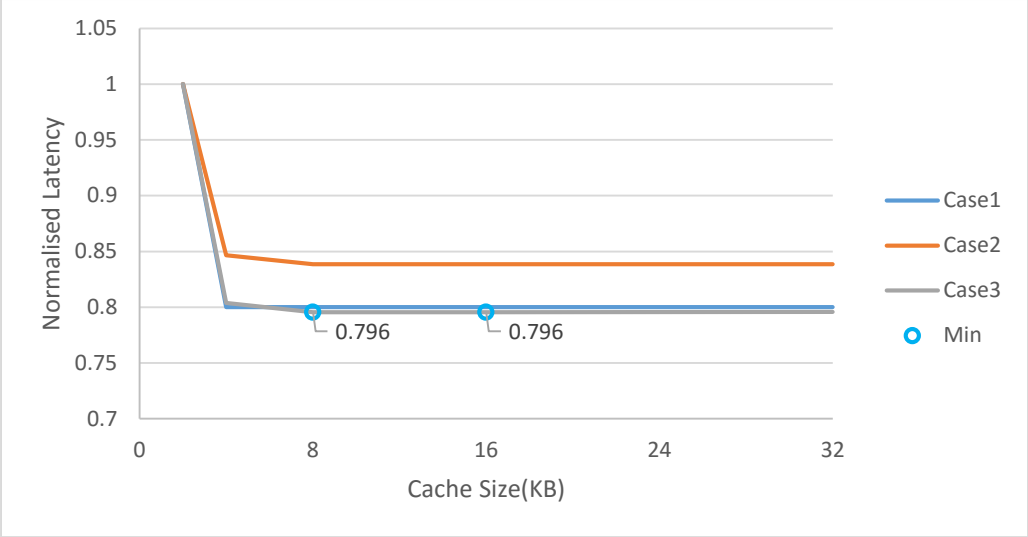


Figure 5: Effect of instruction cache size on latency

In Figure 5, the minimum latency achieved comes from using an instruction cache size of 8KB or more. There is no improvement from using a cache size larger than 8KB. Hence, the compiled instructions of task 3 should be between 4KB and 8KB in size.

b) Task 4

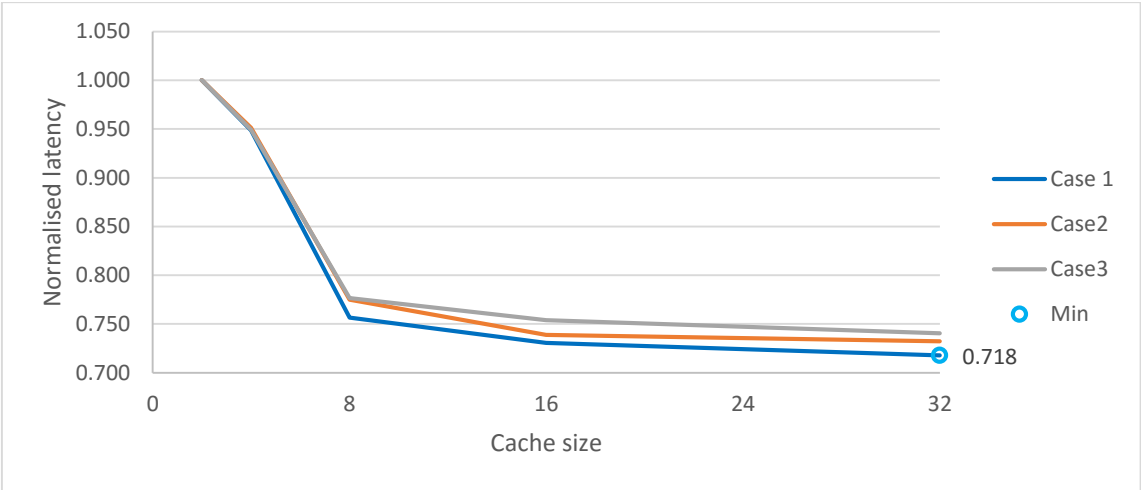


Figure 6: Effect of instruction cache size on latency

In Figure 6, the minimum latency comes from using an instruction cache size of 32KB. Hence, the compiled instruction size is larger than 24KB. Further testing is needed to determine an upper bound. Using 64KB of instruction cache is not possible due to memory bit restrictions of the DE0 board. The DE0 has 63KB of memory.

c) Task 5 w/ Embedded Multipliers

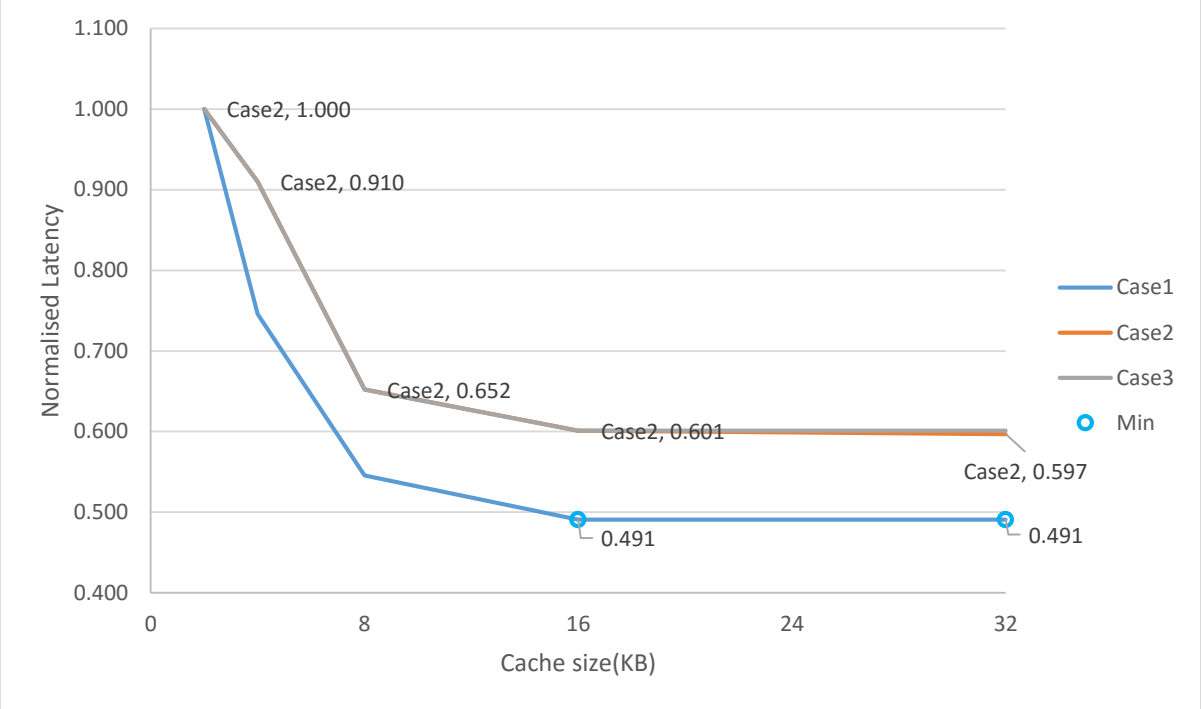


Figure 7: Effect of instruction cache size on latency

In Figure 7, there are several points to note. Since the software implementation has not changed, using a cache size of 32KB is expected to have the minimum normalized latency. However, the latency of case 1 when using 16KB and 32KB of instruction cache size was 27ms and 27ms respectively. Case 1's test case is too small – our latency measurement's resolution is 1ms. Case 2 and 3 also have very tiny changes between 16KB and 32KB. Full results can be found in the appendix; the difference between 16KB and 32KB are too small to be readily visible.

d) *Task 5 w/ Logic Elements*

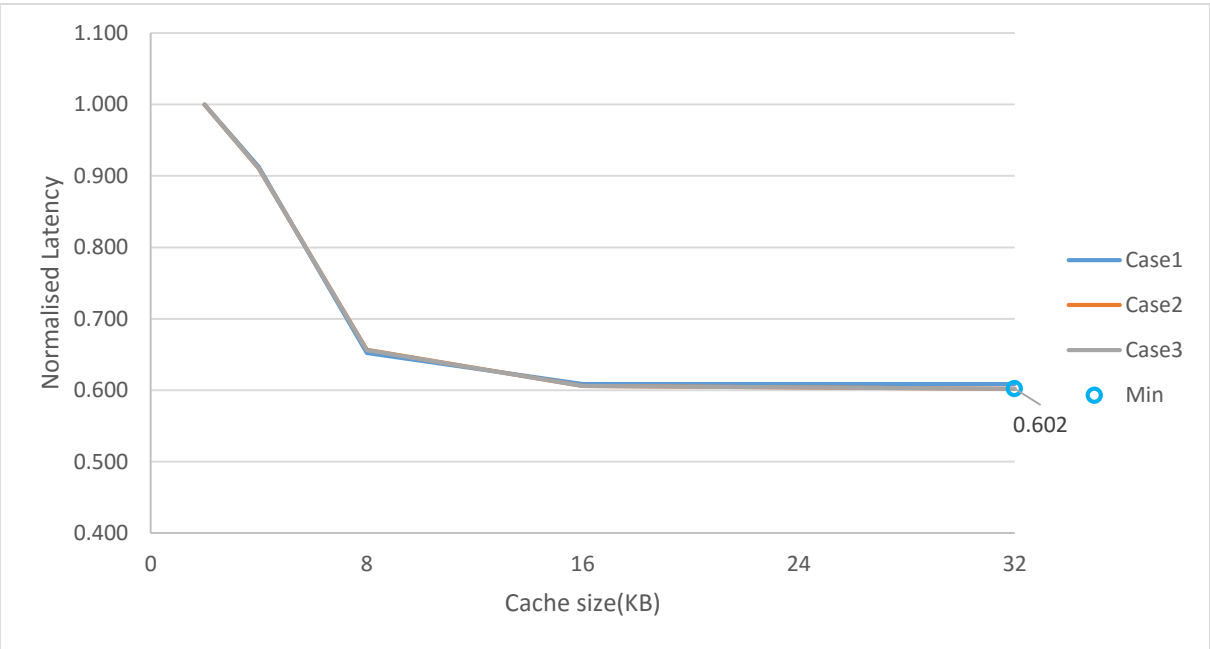


Figure 8: Effect of instruction cache size on latency

In Figure 8, the change is even less apparent. However, albeit small, there is a decrease in latency between 16KB and 32KB for all 3 test cases.

e) Conclusion of instruction cache size testing

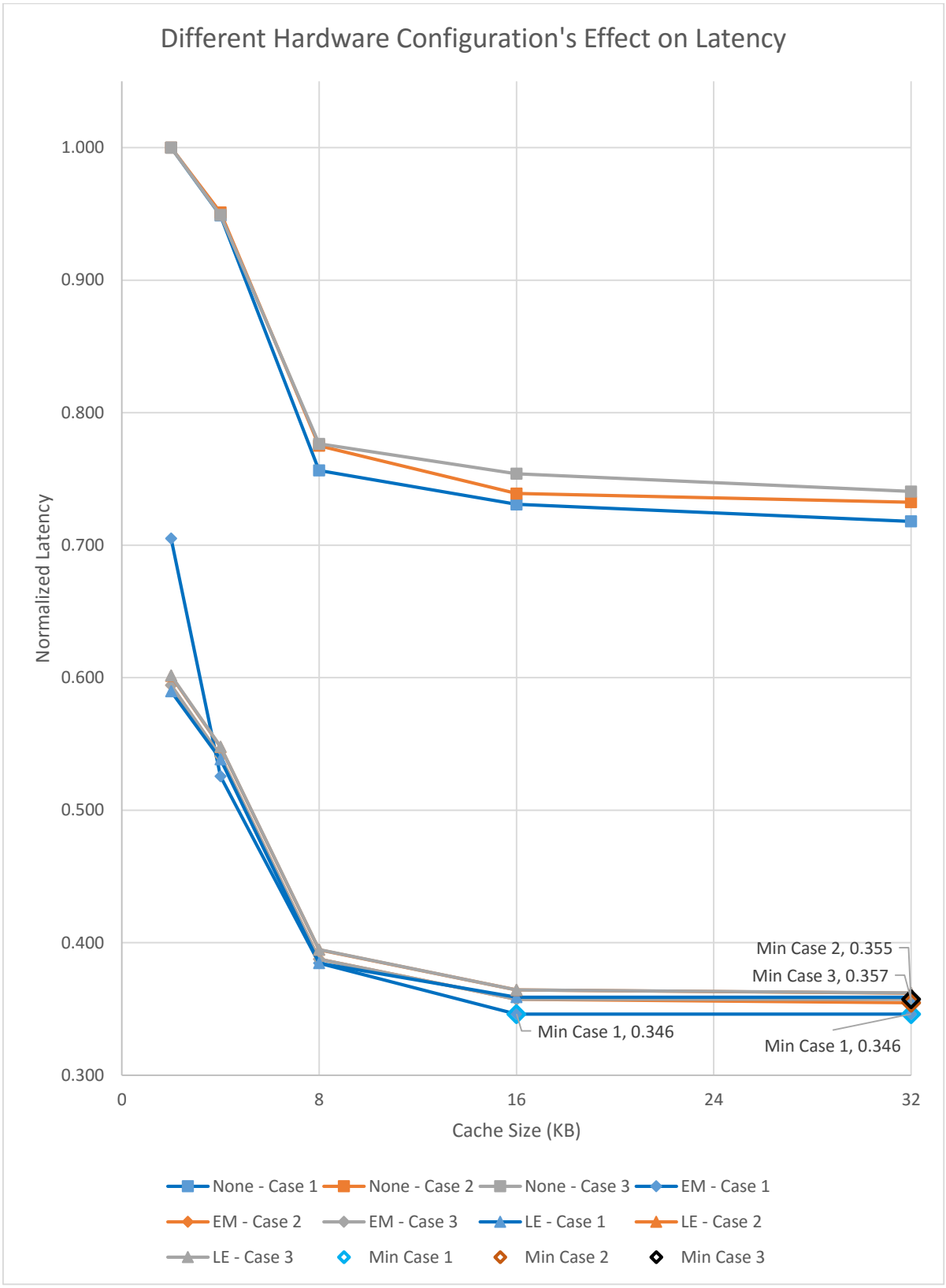


Figure 9: Different Hardware Configuration's Effect on Latency

Figure 9 shows all these results put together. The minimum normalized latencies all use the diamond shaped marker – Embedded Multipliers, for all test cases. Hence, if performance is prioritized, the hardware configuration to be used should be Embedded Multipliers.

32KB of instruction cache size was also shown to provide the lowest latency. However, since this is at the expense of memory resources, a good compromise would be 16KB if resources were an issue, as the difference in latency between using 16KB and 32KB is relatively small.

D. *Implementation of the cosine function*

Since the hardware synthesis only deals with the NIOS II processor and its helper hardware blocks, there is no such thing as a cosine hardware computation block. Calculating the cosine is therefore emulated with a series of multiplication and addition processes. Digging into the source code of the `math.h` [1] to find the implementation is possible, but doing so reveals very complicated logic that chooses different algorithms based on the input value and hardware configuration. However, a common implementation is using the Taylor Series.

$$\cos(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} + \cdots$$

E. *Further Research into latency differences*

ALU Option	Hardware Details	Cycles per instruction	Supported Instructions
No hardware multiply or divide	Multiply and divide instructions generate an exception	–	None
LE-based multiplier	ALU includes 32 x 4-bit multiplier	11	`mul, muli`
Embedded multiplier on Stratix III families	ALU includes 32 x 32-bit multiplier	3	`mul, muli, mulxss, mulxsu, mulxuu`
Embedded multiplier on Cyclone III families	ALU includes 32 x 16-bit multiplier	5	`mul, muli`
Hardware divide	ALU includes multicycle divide circuit	4 – 66	`div, divu`

Figure 10: Hardware Multiply and Divide Details for the Nios II/s Core [2]

From Figure 10, using Embedded Multipliers should theoretically be 11/5 times faster than using Logic Elements. However, this is not the case for our latency results. This may have to do with the bit widths of the multipliers, and is worthy of further research.

VI. Conclusion

From the results and the discussions about the results, our conclusion is that the optimal configuration for our NIOS II project is the following:

Setting	Value	Reason
On Chip Memory	Removed	Reduces latency, reduces resources
NIOS II	Hardware Multiplier = Embedded Multipliers, Instruction cache = 32KB	Reduces latency at the expense of resources
PLL	`clk0` to SDRAM controller, `clk1` to SDRAM	SDRAM controller should have the earlier clock
JTAG	As specified	N/A

Timer	As specified	N/A
System ID	As specified	N/A
LED PI/O	Removed	Unused, reduces resources
NIOS II BSP	Reduced device drivers = 1, else 0	C++ support is not needed, small C library turned off for `printf()`

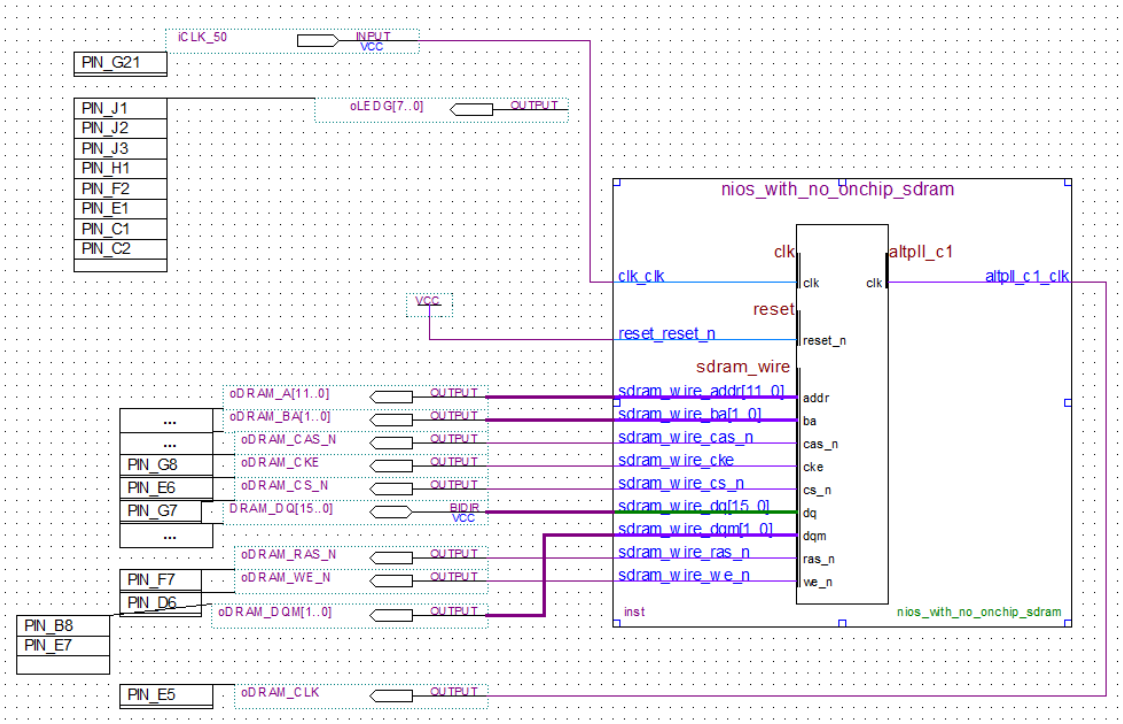
VII. References

[1] IBM Accurate Mathematical Library, "MODULE_NAME: usncs.c," 2001. [Online]. Available: https://sourceware.org/git/?p=glibc.git;a=blob_plain;f=sysdeps/ieee754/dbl-64/s_sin.c;hb=HEAD. [Accessed 13 2 2016].

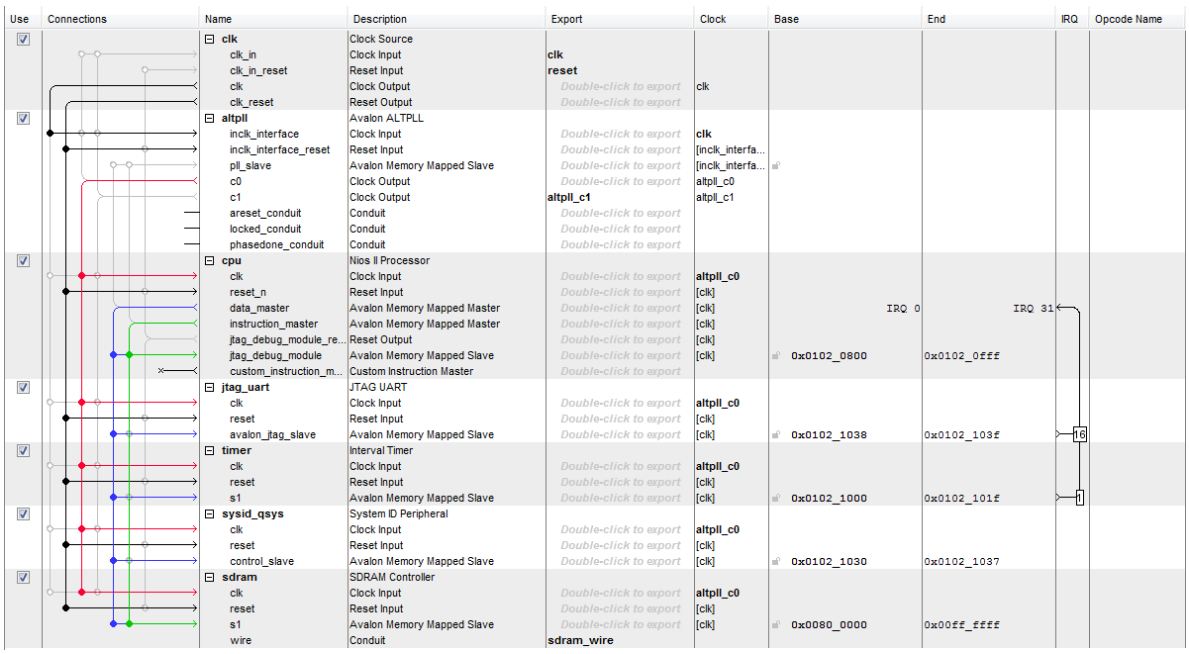
[2] Altera, "Nios II Core Implementation Details," 2 4 2015. [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/nios2/n2cpu_nii51015.pdf. [Accessed 13 2 2016].

VIII. Appendix

A. *Quartus Block Design File*



B. *QSys System Connections*



C. *Testbench Source Code*

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

//0 = x+x*x
//1 = x+cos
#define task 0
#define div_1024 0
//Test case 1
#define S1 5
#define N1 52
// Test case 2
#define S2 0.1
#define N2 2551
// Test case 3
#define S3 0.001
#define N3 255001
// Generates the vector x and stores it in the memory
float x1[N1]={0};
float x2[N2]={0};
float x3[N3]={0};

void generateVector(float* x, unsigned int n, float s)
{
        int i;
        x[0] = 0;
        for (i=1; i<n; i++){
                x[i] = x[i-1] + s;
        }
}
float sumVector(float x[], unsigned int M)
{
        float result=0;
```

```
        int i=0;
        for (i=0; i<M; i++){
                result += (x[i] + x[i]*x[i]);
                printf("%d; %f\n", i, result);
        }
        return result;
}
float sumVector_cos(float x[], unsigned int M)
{
        float result=0;
        int i=0;
        for (i=0; i<M; i++){
                result += (x[i]/2) + (x[i]*x[i]*cosf(floor(x[i]/4)-32));
        }
        return result;
}

int main()
{
        clock_t t1, t2, t3, t4, t5, t6;
        char a[50], b[50], c[50];
        ////////////////
        printf("Running algorithm for task: ");
        printf((task)?"4+\n":"1-3\n");
        printf("Dividing result by 1024 is: ");
        printf((div_1024)?"ON\n":"OFF\n");
        ////////////////
        //test case 1 //
        ////////////////
        printf("gen vector 1, ");
        generateVector(x1, N1, S1);
        printf("sum vector 1\n");
        float y1=0.0;
        if (task) {t1 = times(NULL); y1 = sumVector_cos(x1, N1); t2 = times(NULL);}
        else {t1 = times(NULL); y1 = sumVector(x1, N1); t2 = times(NULL);}
        gcvt(t2-t1, 10, a);
        alt_putstr("Time = "); alt_putstr(a); alt_putstr("; ");
        printf("Result = %f\n", ((div_1024)?y1/1024:y1));

        ////////////////
        //test case 2 //
        ////////////////
        printf("gen vector 2, ");
        generateVector(x2, N2, S2);
        printf("sum vector 2\n");
        float y2=0.0;
        if (task) {t3 = times(NULL); y2 = sumVector_cos(x2, N2); t4 = times(NULL);}
        else {t3 = times(NULL); y2 = sumVector(x2, N2); t4 = times(NULL);}
        gcvt(t4-t3, 10, b);
        alt_putstr("Time = "); alt_putstr(b); alt_putstr("; ");
        printf("Result = %f\n", ((div_1024)?y2/1024:y2));

        ////////////////
        //test case 3 //
        ////////////////
        printf("gen vector 3, ");
        generateVector(x3, N3, S3);
        printf("sum vector 3\n");
        float y3=0.0;
        if (task) {t5 = times(NULL); y3 = sumVector_cos(x3, N3); t6 = times(NULL);}
        else {t5 = times(NULL); y3 = sumVector(x3, N3); t6 = times(NULL);}
        gcvt(t6-t5, 10, c);
        alt_putstr("Time = "); alt_putstr(c); alt_putstr("; ");
        printf("Result = %f\n", ((div_1024)?y3/1024:y3));

        return 0;
}
```