EE3-05 Digital System Design

Coursework Report 3: Task 6-8

Abstract— **This is the final 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 final report will cover methods used to accelerate the computation by using hardware blocks to do the bulk of the calculations. Multiple sections on benchmarking the performance of our system are included, as well as justifications on methods used, based on constraints such as, but not limited to, hardware resource usage, accuracy when compared with MATLAB, and throughput in terms of bytes per second.**

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 builds on the last. With the hardware configuration already decided, different approaches were taken to improve the latency of our arithmetic expression. Approaches include mapping instructions to hardware and using custom memory read/write blocks. Apart from latency benchmarks, accuracy results are provided, and whether these accuracy hits are worth trading off for extra performance. Furthermore, there are explanations for each of the custom hardware blocks, and whether they were included in the final system.

$$\sum \frac{x}{2} + x^2 \cos\left(\left\lfloor \frac{x}{4} \right\rfloor - 32\right)$$

Figure 1: Function to evaluate

Figure 1 shows the arithmetic expression to evaluate over the array of variable size, holding floating point values from 0-255. Table I shows test cases used for this report. Figure 2 shows the formula used to calculate resource usage, based on a maximum amount of 15408 logic elements, 516096 bits of memory, and 112 embedded multipliers on the DE0 board.

TABLE I.

Test Case	Array Size	Step Size
1	5	52
2	0.1	2551
3	0.001	255001

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

Figure 2: Resource Usage formula [1]

II. Previous System Layout

The previous legacy system (Table II) used for tasks 3-5 was also used for tasks 6-8 as a base design as this report is concerned with accelerating the computation of the arithmetic expression, and not with improving the hardware configuration.

TABLE II.

Setting	Value
On Chip Memory	Removed
NIOS II	Hardware Multiplier = Embedded Multipliers, Instruction cache = 32KB
PLL	50Mhz, `clk0` to SDRAM controller, `clk1` to SDRAM
JTAG	As specified
Timer	As specified
System ID	As specified
LED PI/O	Removed
NIOS II BSP	Reduced device drivers = 1, else 0

III. Task 6: Add Hardware Floating-Point Units

A. *Aim*

To replace the software emulation of arithmetic with hardware blocks to increase throughput at the expense of FPGA resources. Choosing between fixed and floating point implementations are justified, as well as the choice of multi-cycle latencies (where applicable).

B. *Methodology and Implementation*

The main difference between the fixed and floating point hardware blocks is the time it takes to compute a result. Fixed point hardware blocks are purely combinational logic, and can therefore output a result in same clock cycle, valid on the next clock cycle. Floating point blocks take longer (multi-cycle), and have an option to choose the desired latency. This also means that floating point blocks are therefore pipelined, and the critical path in the system will be affected by the chosen latency. Fixed point blocks also have an option to be pipelined, but this is by default turned off.

Since our array of numbers are not all integers, using fixed point hardware blocks will provide us with the wrong result as they do not have the dynamic range needed for our arithmetic expression. Therefore, it was an obvious choice to use floating point hardware blocks. Altera Quartus provides a tool named MegaWizard to help instantiate these hardware blocks.

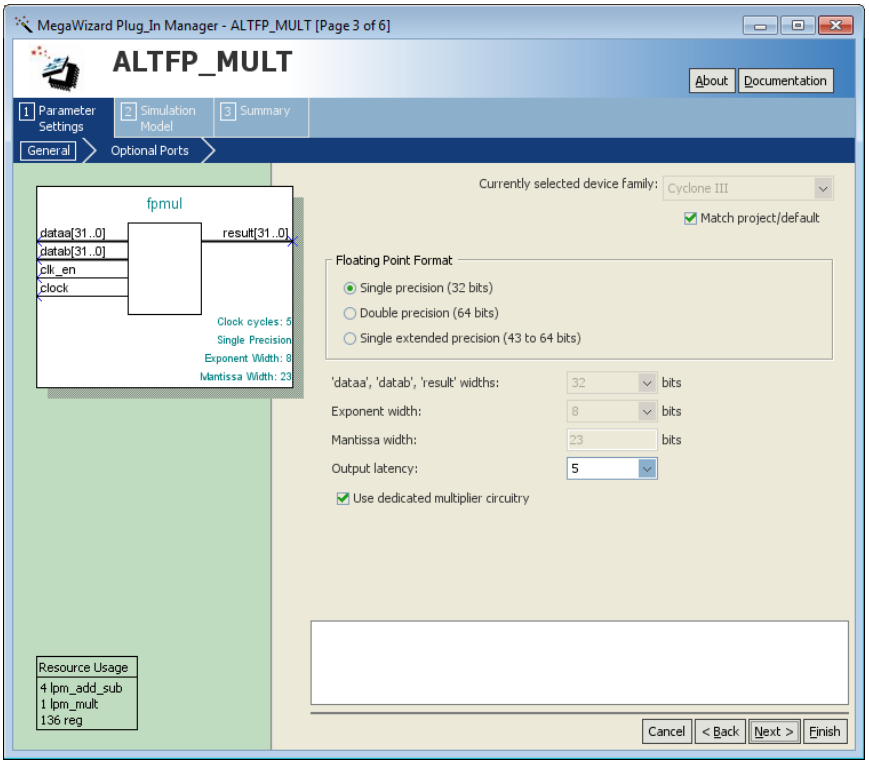

Figure 3: Floating Point Hardware Instantiation

These hardware blocks are instantiated to a `.qip` file, which then have to be converted to a QSys component using the inbuilt "New Component" function in QSys. This then creates a `.tcl` file (the component) which contains all the parameters set in the component, as well as locations of source files. After compilation using Quartus, the custom instruction can be tested by using a macro to call the custom instruction. The default return type of all custom instructions is `int`; this can be changed by setting the macro custom return type to `__builtin_custom_<output type>n<input type>`. In the case of a floating point multiply, this macro will read `__builtin_custom_fnff`, as both two inputs and output types are of floating point.

The components were also tested in Modelsim to verify both the result and the desired latency. Using the following command (or similar), Modelsim was used to test all the blocks to ensure a correct floating point output. The simulation of the floating point multiplication of 2x3=6 (0x40C00000) is shown below.

```
restart -force -nowave; add wave -r /*; force clock 1 0ns, 0 10ns -repeat 20ns; force
dataa 32'h0x40000000; force datab 32'h0x40400000; force clk_en 1; run 10000ns
```

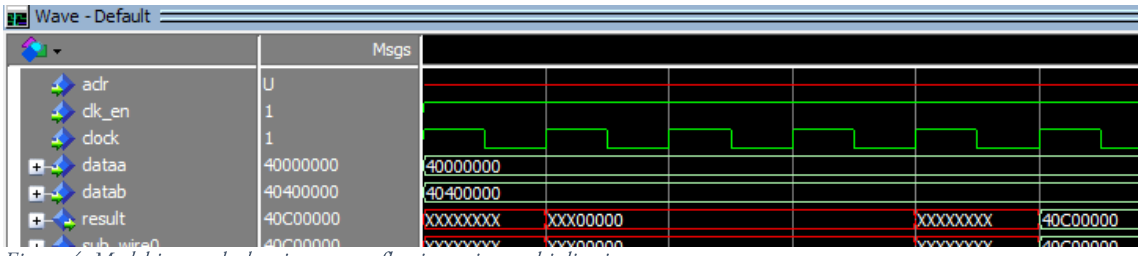

Figure 4: Modelsim result showing correct floating point multiplication

C. *Hardware Improvement of Floating Point Blocks*

```
void custom(float x[], unsigned int M)
{
        float tmp;
        int i=0;
        for (i=0; i<M; i++){
                tmp = x[i]/4;
        }
}

void custom_2(float x[], unsigned int M)
{
        float tmp;
        int i=0;
        for (i=0; i<M; i++){
                tmp = ALT_CI_FPMUL_0(x[i], (float)0.25);
        }
}
```

Using the above custom test bench snippet as an example, the difference in latency between using the software emulated floating point arithmetic and the hardware block can be found. Test case 3 was used as it was the longest test case. All temporary variables are single precision floating point.

TABLE III.

Test Case 3 (size: 255001)/Latency	tmp = a + b	tmp += a	tmp = a × b
Software Emulation	8795	3174	2631
Hardware Block	755	920	750
Improvement (%)	91.4%	71%	71.4%

This improvement of using hardware blocks to calculate arithmetic expressions is expected, as the NIOS II has to emulate floating point arithmetic in terms of assembly instructions.

Perhaps surprising is the difference in adding two new floating point numbers with accumulation of the previous floating point number. This may be attributed to the fact that in software, the NIOS II has to fetch the value of two new registers before summing them together in the case of a+b, but when accumulating, does not need to re-fetch the previous result as it would have already been held in a register.

However, this is opposite in hardware – the accumulation takes longer than addition. This is due to there being no feedback path from the result of the adder – the result is instead passed back into the NIOS II, registered in the NIOS II, then passed back, as a new result. The hardware block does not know that the passed back result is the previous result, and just treats it as two new numbers to add. The ~200 cycle difference can be attributed to the passing of the result back and forth between the NIOS II and the hardware block.

D. *NIOS II's Custom Instruction Handshaking Scheme*

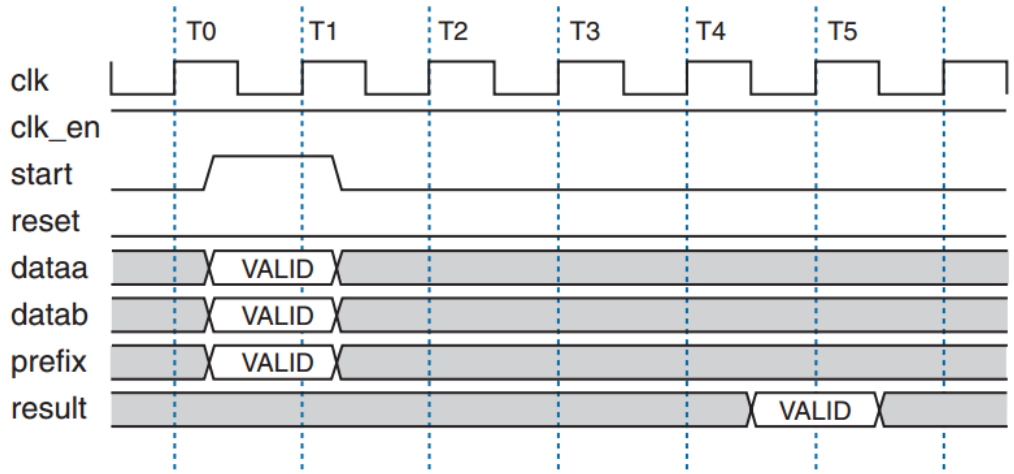

Figure 5: Shows handshaking scheme of NIOS II custom instruction blocking pipelined calculation

The NIOS II does not fully utilize the pipeline available with our custom instructions as it utilizes a handshaking system [2] to push data into and out of the custom instruction. When the NIOS II first pushes a number through, a start signal is pulsed high, and the corresponding done signal will be pulsed high when the custom instruction has finished. Then this cycle restarts, with the NIOS II pushing the second number into the custom instruction once the done signal for the first number is received. Therefore, the custom instruction only works on one floating point number at a time (clock cycles T1-T4 are wasted in Figure 5). Hence, this is one of the reasons our custom instruction did not see more improvement in latency.

E. *Overall Performance*

```
#define ALT_CI_COS_0(A) __builtin_custom_fnf(ALT_CI_COS_0_N,(A))
#define ALT_CI_FPADD_0(A,B) __builtin_custom_fnff(ALT_CI_FPADD_0_N,(A),(B))
#define ALT_CI_FPMUL_0(A,B) __builtin_custom_fnff(ALT_CI_FPMUL_0_N,(A),(B))

float sumVector_cos_custom(float x[], unsigned int M)
{
        float result = 0.0;
        float x_squared = 0.0;
        float cos_out = 0.0;
        float mult_out2 = 0.0;
        float mult_out1 = 0.0;
        int i = 0;
        for (i = 0; i < M; i++) {
                x_squared = ALT_CI_FPMUL_0(x[i], x[i]);
                cos_out = cos(floor(ALT_CI_FPMUL_0(x[i], (float)0.25))-32);
                mult_out1 = ALT_CI_FPMUL_0(x[i], (float)0.5);
                mult_out2 = ALT_CI_FPMUL_0(x_squared, cos_out);
                result = ALT_CI_FPADD_0(ALT_CI_FPADD_0(mult_out1, mult_out2), result);
        }
        return result;
}
```

The test bench snippet above was used to test the latency of the arithmetic expression after converting all the previously emulated floating point operations (addition, multiplication, division) by their respective custom instruction. Division was implemented by multiplying by the reciprocal of the divisor.

The table below shows the reduction in latency when using the floating point adders and multipliers, both set to their default minimum latencies (add-5, multiply-7). Task 3-5 results are extracted from our previous report, report 2. The throughput was measured using test case 3 as it has the longest test case.

TABLE IV.

Test Case Latency	Task 3-5	Task 6	Improvement (%)

1	28	23	17.9
2	1384	1144	17.3
3	136496	114332	17.4
Throughput (Bps)	467	557.6	21.2

However, the floating point blocks have a variable latency. The floating point blocks' latency had a range of 5-14 and 7-11 for the floating point add and multiply blocks respectively. Only the combinations of the extremes were tested to see if there was a correlation.

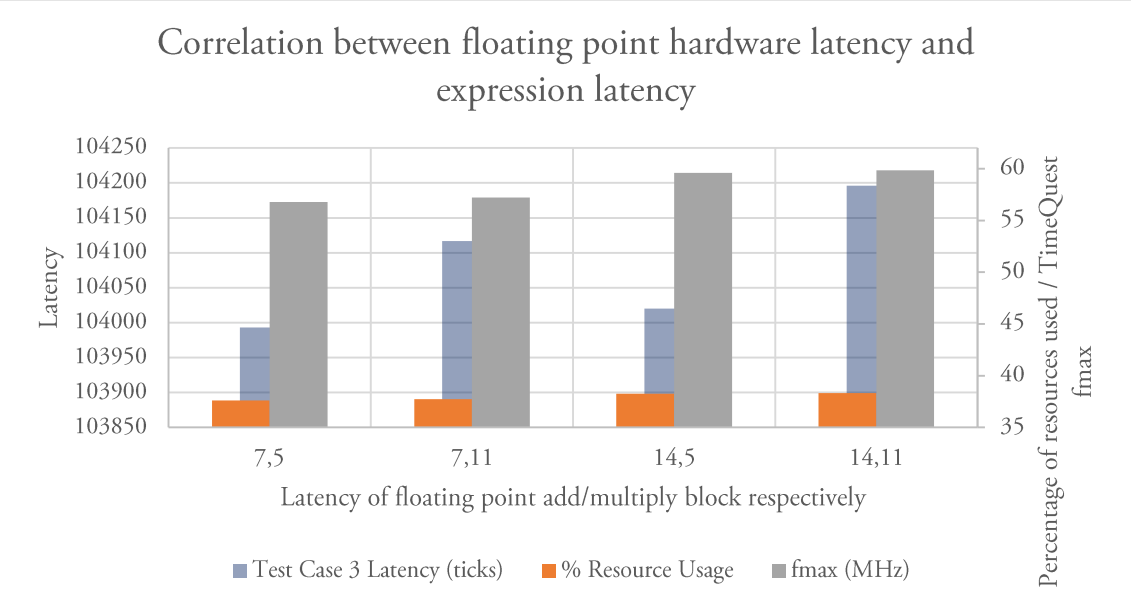

Figure 6: Plot showing relationship between pipeline stages and resultant latency

Figure 6 shows some very interesting results. The maximum frequency as reported by TimeQuest analyser is plotted. Theoretically, using a longer pipeline decreases the critical path, leading a higher fmax.

Additionally, a longer pipeline should not alter the latency with obvious changes. The expected difference in latency between a floating point hardware block with latency 7 and 14 can be calculated as follows:

$$Latency\ difference = Clock\ cycle\ difference \times test\ case\ size \div clock\ frequency$$
$$= 7 \times 255{,}001 \div 50{,}000{,}000 = 35.7$$

ticks. This difference change should be only the difference of the pipeline stages since result is valid every clock cycle as soon as the pipeline is filled. However, our graph shows that this is not the case, and hence implies that custom instructions are never pipelined by the NIOS II. The average clock cycle per custom instruction is hence greater than the expected latency of each custom instruction.

The expected latency difference when implementing multiple instructions, some nested in another custom instruction is complicated to calculate, but the small difference in latency in Figure 6 can be attributed to this. Hence, based on the result that the smallest latency is achieved by using pipeline stages of 7 and 5, this is what our future improvements will be based upon.

The percentage of resources used also stayed relatively constant, with around a 200LE difference between the smallest (add 7, mult 5) and largest (add 14, mult 11) implementations. This is to be expected as a longer pipeline occupies more memory and logic elements.

In conclusion, there were improvements on the latency from using custom instructions, but not as much as we expected from doing floating point operations in hardware as opposed to emulation using the NIOS II processor. The improvements from only testing one instruction were drastically reduced (91% to 21%). This can be attributed to the computation of the cosine bottlenecking the computation of the loop.

Additionally, calling a custom instruction inside a custom instruction can also have an effect on latency. Testing on test case 3, there was a latency difference of +83 ticks (+9.42%) when trying to calculate $tmp = x[i] + 3.0$ as itself and $tmp = (x[i] + 1.0) + 2.0$ respectively. 83 ticks for now is negligible for test case 3, but it may become significant after further acceleration.

F. *Resource Usage*

The following table shows the resource usage for different hardware blocks as reported by Quartus after compilation, as well as their output latency as defined in MegaWizard. The maximum amount of logic elements consumed is by the floating point add – it takes approximately 5% of the total number of logic elements. Increasing the latency increases the amount of logic elements used, as well as the amount of memory used – extra registers and logic are needed for the increased size of the pipeline, hence the result is to be expected.

TABLE V.

Resource	Fixed Point Add	Fixed Point Multiply	Floating Point Add	Floating Point Multiply	Floating Point Multiply
Logic Elements	32	79	795	253	346
Memory Bits	0	0	36	0	173
Embedded Multipliers	0	8	0	7	7
Block Latency	0	5 (minimum)	7 (minimum)	5 (minimum)	11 (maximum)

Adding the needed custom instructions (floating point add, multiply) then compiling gives the following resource usage (Figure 7). Compared with the legacy system in Task 5 (3218 LE, 291840 MB, 4 EM), there is a 55% increase in LE's and 175% increase in EM's. Compared with the total number of resources, this is an 11.5% increase in LE's used, and 6.25% increase in EM's used. Using the aforementioned resource usage formula, the resource usage rises from 0.271 to 0.330, an increase of 6% of the total system resources.

However, since this increase in resources gives decreased latency, it justifies the aim of the project, which is accelerate the performance. Hence, it is part of our design.

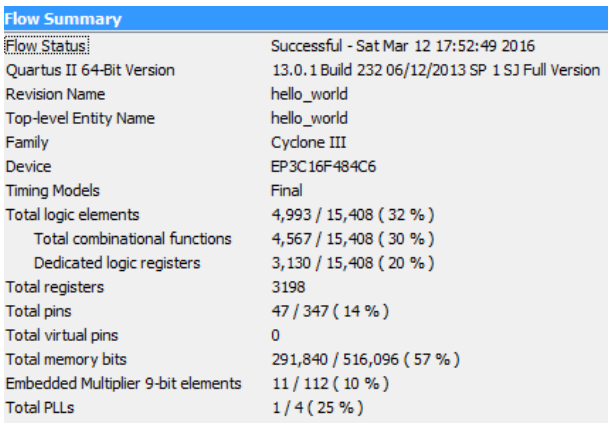

Figure 7: Hardware resource usage of custom instruction

G. *Accuracy of the Results*

TABLE VI.

Test Case	MATLAB Result	t6 Result	Error	Error %
1	57879.873	57879.867	0.006	1.00415E-05
2	-126818.140	-76973.641	-49844.499	39.304
3	-12774366.300	37022500.000	-49796866.300	389.819

The accuracy of the results are shown in Table VI. They are the same values as those found in Report 2 (task 5). This is to be expected as the NIOS II floating point arithmetic merely emulates the hardware – they should have the same output. Figure 8 is from Report 2, again illustrating the sinusoidal form of the error,

plotted for every array index of test case 3. This comes from the fact that floating point arithmetic on a numbers that are large orders of magnitude apart have a loss of precision, stemming from losing bits in the mantissa due to shifting when normalizing the smaller number to the size of the larger number.

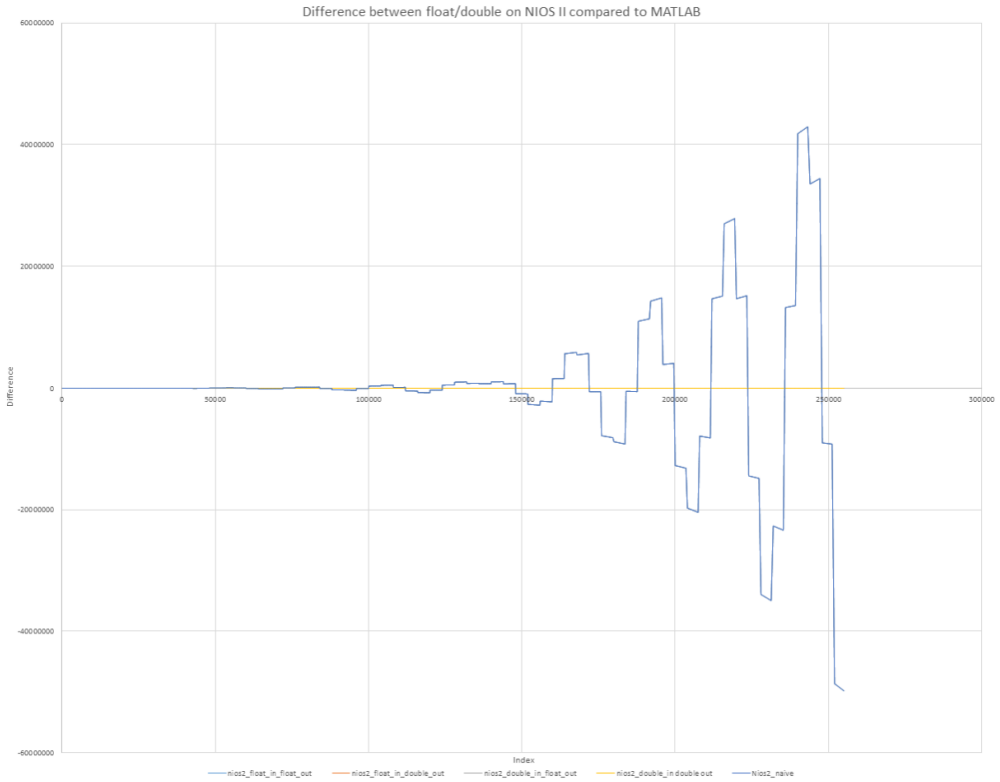

Figure 8: Report 2 cosine error of test case 3

IV. Task 7: Add Dedicated Hardware Block to compute the inner part of the arithmetic expression

A. *Aim*

From Task 6's performance results, the main bottleneck was determined to be the computation of the cosine. Currently, `cosf()` from math.h is used. Digging into the source code [3] of the cos function's implementation inside the `math.h` library reveals a bunch of logic which determines which cosine algorithm to use. However, this is still all done in software. A hardware approach was needed to accelerate the cosine computation significantly. The COordinate Rotation DIgital Computer (CORDIC) algorithm was chosen.

B. *Methodology and Implementation*

$$\cos(\left\lfloor \frac{x}{4} \right\rfloor - 32)$$

The inner part of the cosine calculation requires that the input value be divided by 4, floored, then have 32 subtracted away. These will all be converted into hardware.

1) Another software floor function

`floor()` in math.h is slow [4] when compared to other implementations. The floor function was replaced by `my_floor`, a custom software function.

```
static inline int my_floor(float x)
{
    return (int) x - (x < (int) x);
}
```

By executing this instruction over the whole of the largest array (test case 3), an improvement over math.h's floor function was seen Table VII. This is to be expected as the custom floor function only computes the flooring, and does not have corner cases for error, infinity, NaN, etc, saving on a lot of clock cycles. However, the floor function can be implemented in hardware instead.

TABLE VII.

Test Case Size/Latency	math.h floor()	my_floor()	Improvement
255001	10422	5105	49%

2) Implementing floor(x/4)-32 in hardware

Let $\cos\left(\left\lfloor\frac{x}{4}\right\rfloor - 32\right)$ *be* $\cos(x)$. A lookup table method was used to calculate x. Since the input values are from 0 to 255, the dynamic range of x is from -32 to 31. Using a lookup table method bypasses the need for a floor, floating point multiplication, and floating point addition/subtraction block. MATLAB (Appendix B) was used to calculate the possible values, then an entity was created using VHDL (Figure 9) to convert the input floating point value to its fixed point floor(x/4)-32 value. `const thres()` is an array of multiples of 4, so an input value of <4.0 is mapped to the first value in `rom()`, -32, 4.0≤ x<8.0 to the second value, -31, etc.

The fixed point representation was chosen to be a 4 bit integer, followed by 28 fractional bits. This was chosen because 28 fractional bits is more than enough for the precision needed for our floating point calculations. Moreover, having a 4 bit integer would allow for easy debugging in Modelsim as the most significant hex digit is the hexadecimal value of the integer.

```
146 BEGIN
147     C1 : PROCESS(x)
148     BEGIN
149         IF signed(x) < signed(thres(0)) THEN
150             fx_out <= std_logic_vector(rom(0));
151         ELSIF signed(x) < signed(thres(1)) THEN
152             fx_out <= std_logic_vector(rom(1));
153         ELSIF signed(x) < signed(thres(2)) THEN
154             fx_out <= std_logic_vector(rom(2));
155         ELSIF signed(x) < signed(thres(3)) THEN
156             fx_out <= std_logic_vector(rom(3));
157         ELSIF signed(x) < signed(thres(4)) THEN
```

Figure 9: VHDL lookup table for calculating floor(x/4)-32

Testing this combinational hardware block using a custom instruction on the NIOS II processor gives results as follows in Table VIII. Let ALTFP be the hardware multiplication block defined in task 6.

TABLE VIII.

Test Case Size/Latency	math.h floor(x/4)-32 (reference)	math.h floor(ALTFP(x/4))-32)	my_floor(x/4)-32	my_floor(ALTFP(x/4))-32	Custom hardware (x/4)-32 block
255001	16862	14948	7059	5163	714
Improvement	N/A	11.4%	58.1%	69.4%	95.8%

Again, the improvement here is to be expected, as our custom lookup table block computes a result extremely quickly. As the math.h and custom floor implementations have variable clock cycle lengths, the NIOS II pipeline must be stalled until the result finishes. A lookup table method bypasses all of this, giving a floor(x/4)-32 result in the next clock cycle, leading to the 95.8% increase.

3) Implementing the CORDIC Algorithm on Hardware

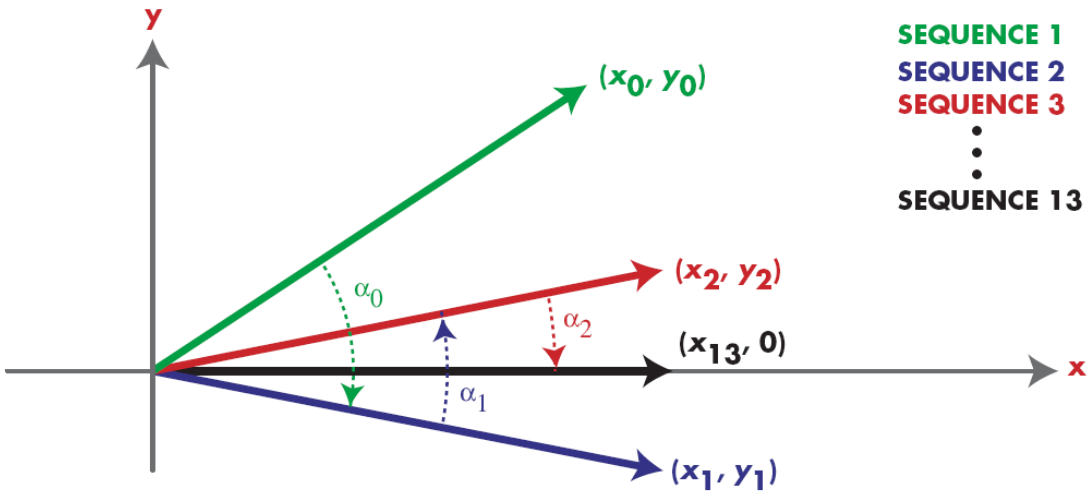

Figure 10: CORDIC Iterations [5]

$$x_{i+1} = x_i - d_i\ y_i 2^{-i}$$
$$y_{i+1} = y_i + d_i\ x_i 2^{-i}$$
$$z_{i+1} = z_i - d_i\ \alpha_i$$

Figure 11: CORDIC algorithm [6]

The CORDIC algorithm was chosen to calculate the cosine expression as it only requires addition and shift operations only, avoiding the use of multiplication. It gets a result by performing sub rotations of pre-defined angles (Figure 10), and the error of the final result is inversely proportional the number of sub rotations done, i.e. the number of iterations. The following code snippet shows the main loop, written in VHDL, based off the equations in Figure 11 and the logic flow in Figure 12 (exact implementation differs).

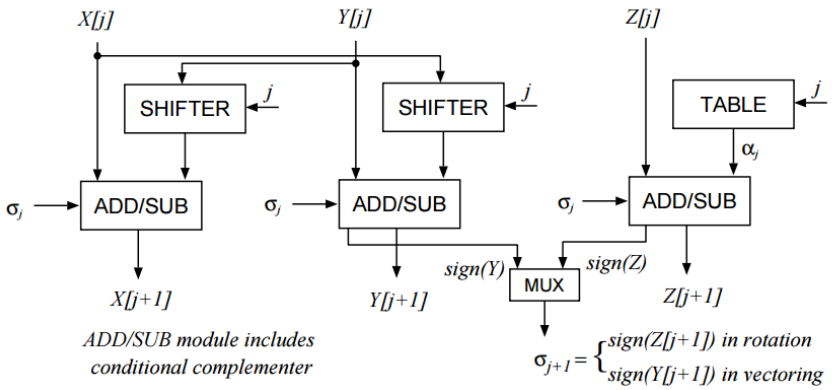

Figure 12: CORDIC Hardware logic flow reference [7]

```
for index in 1 to 27 loop
	if z(index-1)(31)='1' then
		z(index) <= slv(signed(z(index - 1)) + signed(table(index-1)));
		x(index) <= slv(signed(x(index - 1)) + signed(asr(y(index-1),index-1)));
		y(index) <= slv(signed(y(index - 1)) - signed(asr(x(index-1),index-1)));
	else
		z(index) <= slv(signed(z(index - 1)) - signed(table(index-1)));
		x(index) <= slv(signed(x(index - 1)) - signed(asr(y(index-1),index-1)));
		y(index) <= slv(signed(y(index - 1)) + signed(asr(x(index-1),index-1)));
	end if;
end loop;
result <= x(27);
```

The floating point input is first converted to a fixed point value between -32 and 31 using the aforementioned custom lookup table hardware block. All constants inside our CORDIC block use our defined custom fixed point format (4:28 bits). The resultant fixed point value after the CORDIC block therefore needs to then be converted back to floating point for use in the summation - an Altera fixed to floating point converter block was used – we did not want to write a custom VHDL block and thoroughly test all the corner cases for fixed to floating point conversion.

For every new iteration, the accuracy of the CORDIC output is improved by one digit in binary representation ($error = 2^{-stages}$) as the sub rotation angle is around half the previous angle (Figure 13). Therefore, the maximum amount of stages our CORDIC block can have is 28 as we have defined 28 fractional bits – this corresponds to a loop length of 27 – the first stage outside the loop normalizes the input value to a maximum of 2π. However, since the output value is floating point, the precision of single precision floating point numbers mean that using a very high number of stages is wasted.

```
constant table  : table_t := (
    "00001100100100001111110110101010",
    "00000111011010110001100111000001",
    "00000011111010110110111010111111",
    "00000001111111010101101110101010",
    "00000000111111111010101011011110",
    "00000000011111111111010101010111",
    "00000000001111111111111010101011",
    "00000000000111111111111111010101",
    "00000000000011111111111111111011",
    "00000000000001111111111111111111",
    "00000000000000100000000000000000",
    "00000000000000010000000000000000",
```

Figure 13: VHDL table, showing increasing accuracy of final CORDIC answer (decreasing angle rotation size)

However, since VHDL unrolls all for loops, accuracy comes at a tradeoff for resources. To find the optimal number of stages, the accuracy of 100 random values of test case 3 was tested using `float((double)rand()/(double)(RAND_MAX/255.0));`. The FPGA resource usage was also taken into account. The graphed LE's used (other resources did not change) subtract away the base design with only the NIOS II and no CORDIC block (3210 LE's). Figure 14 shows quickly diminishing returns in accuracy from using a high number of stages. The resource usage is also relatively linear.

Since single precision floating point numbers have ~7 significant digits [8], we decided to use 16 stages as it provides good accuracy - the floating point cosine output ($0 \leq output \leq 1$) will be multiplied by $0 \leq x[i]^2 \leq 65025$. The average absolute error when compared with math.h's implementation of 4.873E-6 (0.01% error) ensures that the error is on the limit of the amount of significant digits floats can represent.

16 stages use a reasonable amount of resources (~20% of total LE's). Moreover, since 16 is a multiple of 2, the 16 stages can be unrolled into 8 stages with twice the resources consumed, 4 stages with four times the resources consumed, etc. These reasons fully justify using 16 stages; unrolling will be implemented later.

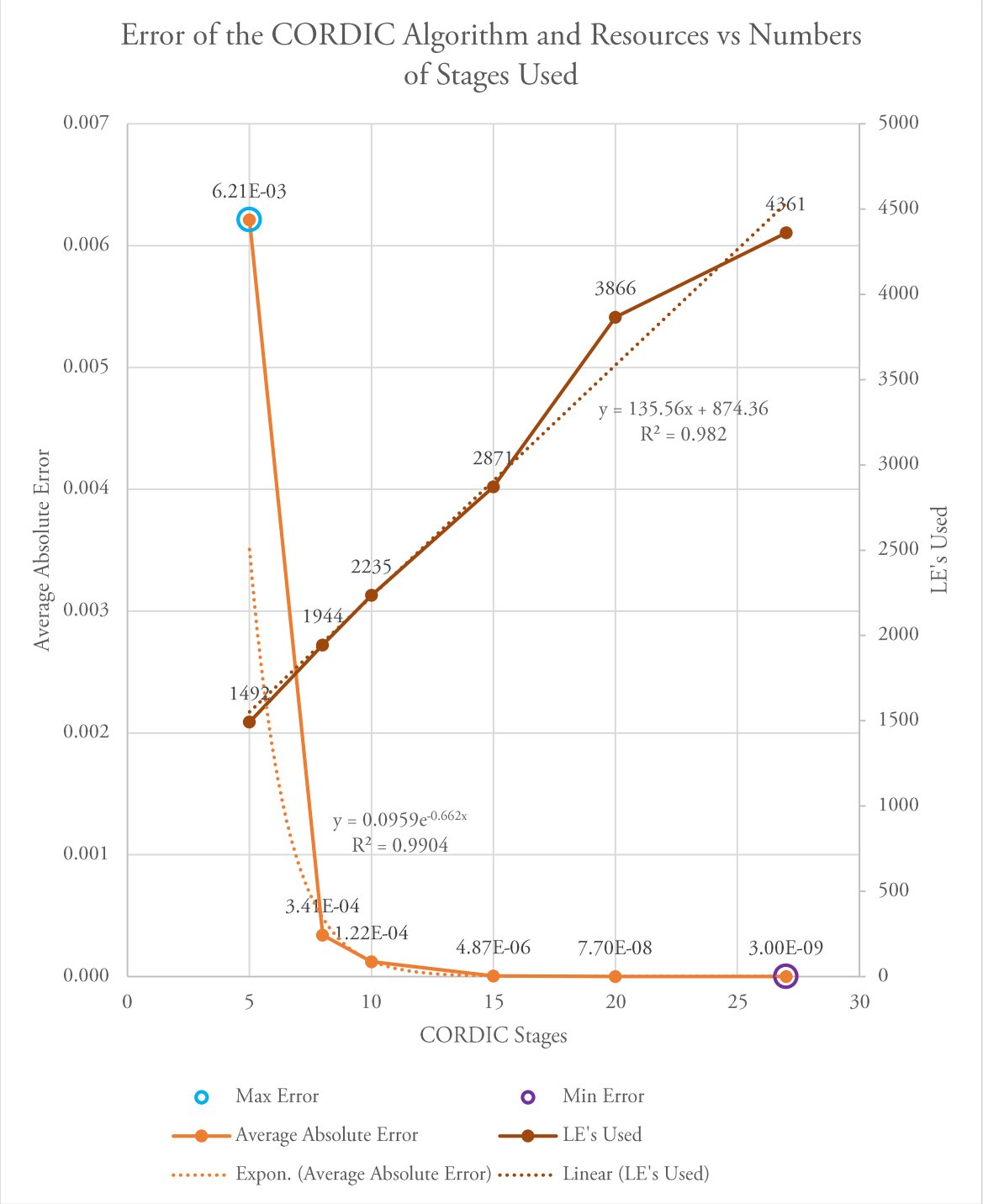

Figure 14: CORDIC accuracy and resources graph

C. *Performance*

a) *CORDIC only*

Performance was measured using test case 3 as it was the longest test case. The results are shown below, in Table IX. Note that the benchmark loop only calculates cosine values, and does not compute the outer part of the arithmetic expression, i.e., the $\frac{x}{2}$, x^2 values. This ensures that the improvement in latency is due to the only the hardware CORDIC block and the floor lookup table block, and not the floating point arithmetic blocks.

TABLE IX.

Test Case Size/Latency	math.h cos(floor(x/4)-32) (reference)	math.h cos(my_floor(x/4)-32)	CORDIC(x)
255001	118143	109195	895
Improvement	N/A	7.6%	99.2%

The performance improvement when using the CORDIC block is significant. Partly due to the aforementioned improvement of using a lookup table, computing cosines using the CORDIC algorithm is much faster as it only requires 16 clock cycles to get an output, whereas the math.h implementation is variable.

b) Overall

TABLE X.

Latency	Task 6 (reference)	Math.h cos(), custom software floor()	CORDIC(x)
Case 1	23	21	1
Case 2	1143	1040	27
Case 3	114331	103993	2718
Improvement of Case 3 (%)	N/A	9.0%	97.6%

The overall improvement from task 6 to 7 can be seen in Table X, where with the CORDIC block, we achieved a 97.6% decrease in latency. This result justifies the increased resource usage, as the primary aim is to achieve a higher throughput. Moving all the arithmetic to hardware gives a significant boost to performance – the NIOS II now does not do any software emulation of floating point arithmetic, and only serves to push and pull results to and from the custom instructions.

D. Accuracy against math.h

a) CORDIC only

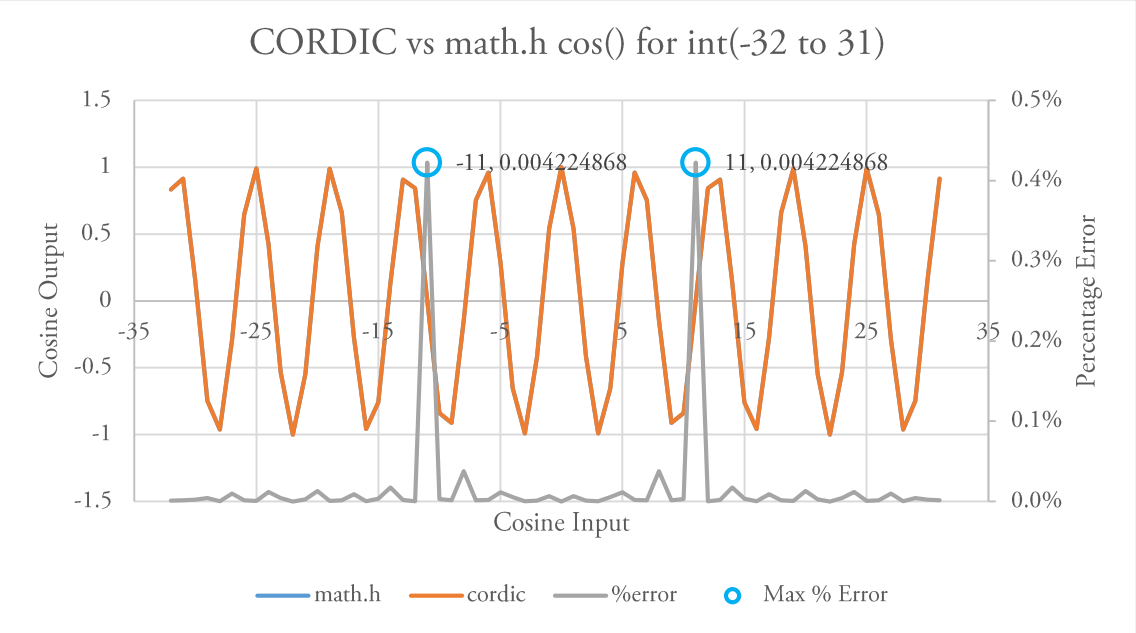

Figure 15: The possible 64 CORDIC outputs plotted

The output of our hardware CORDIC block was plotted for each of the possible input values (-32 to 31). The output is a sinusoidal waveform, albeit exhibiting quantisation properties, confirming that our CORDIC output's shape is correct. The stepping effect is due to the input being a finite set of integers from -32 to 31, giving only 64 possible CORDIC outputs, a low resolution for graphing.

The grey error series on Figure 15 shows regular peaks. The following derivation of the error shows that this is to be expected, as the maximum error should occur at multiples of π.

$$\Delta y = \cos(x + \Delta x) - \cos(x)$$
$$\Delta y = \cos x \cos \Delta x - \sin x \sin \Delta x - \cos x$$
$$\Delta x = \sim 2^{-17}$$
$$\sin \Delta x \sim 0$$
$$\Delta y = \cos x \cos \Delta x - \cos x$$
$$\frac{d\Delta y}{d\Delta x} = -\sin x \cos \Delta x + \sin x$$
$$= (1 - \cos x) \sin x$$
$$\frac{d\Delta y}{d\Delta x} = 0$$
$$x = n\pi, \qquad n = 0,1,2,\cdots$$
$$\Delta y_{max} = \cos x \, (\cos \Delta x - 1) \; for \; x = n\pi$$

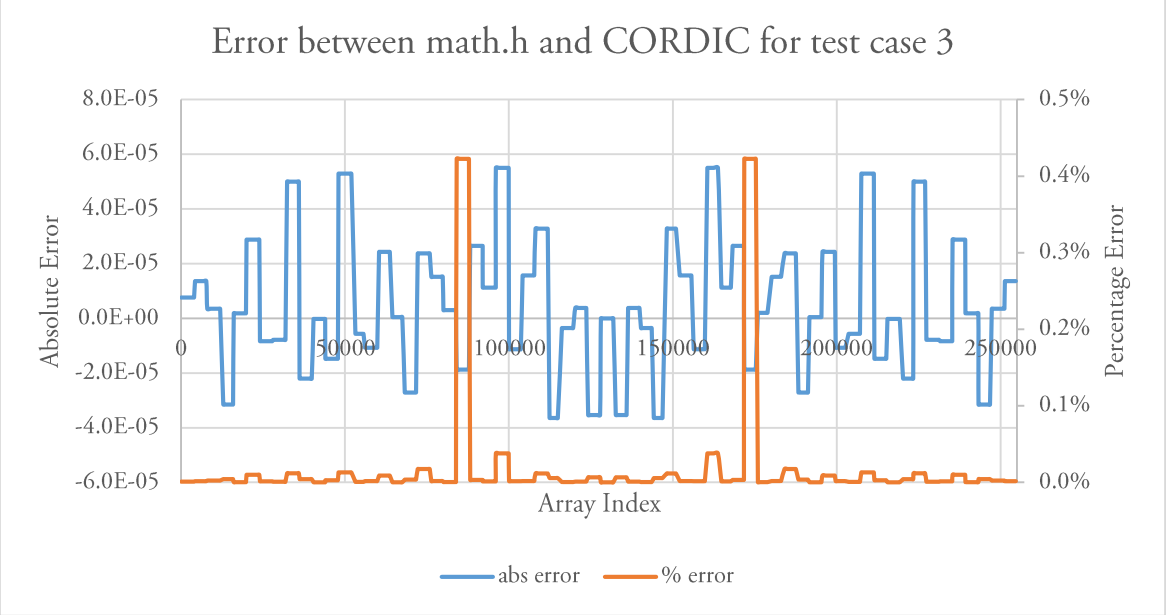

Figure 16: Showing error between math.h and our CORDIC algorithm

Running test case 3 again, the accuracy of our 16 stage CORDIC block was tested against the math.h cosine implementation. The absolute error and the percentage error at every 4000th value, i.e. every integer, were plotted in Figure 16. The percentage error is greatest when the cosine output value is around zero, as then a tiny error in the CORDIC algorithm corresponds to a large percentage difference between the two implementations ($\cos(11) = 0.00442569798$). However, the maximum percentage error is still only 0.42%, a very respectable result.

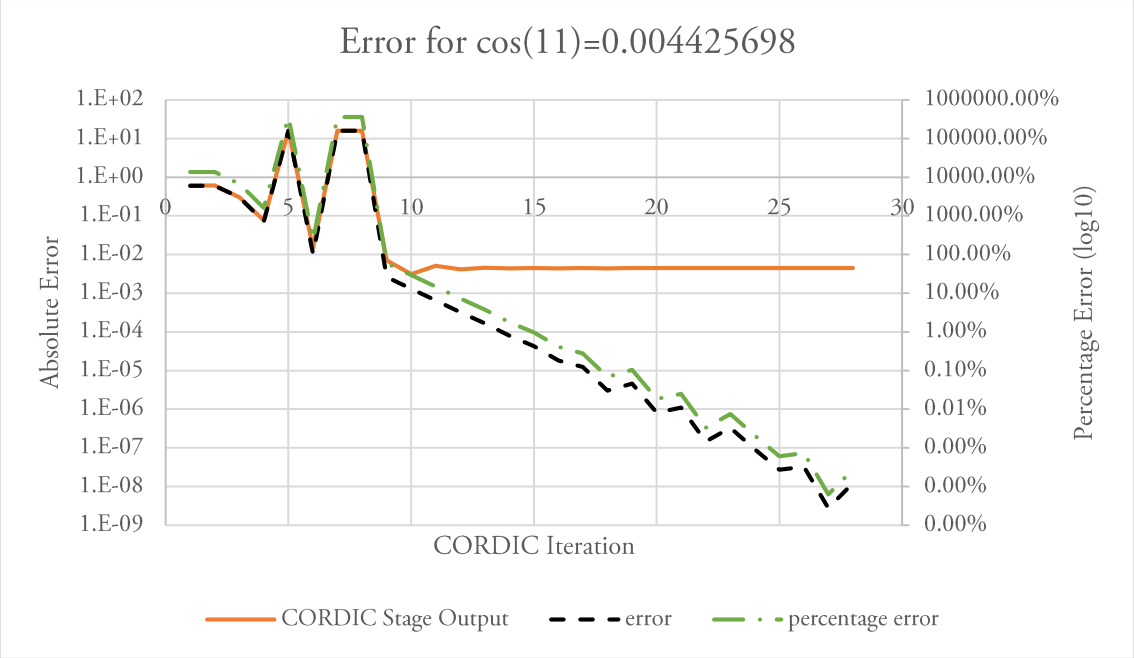

Figure 17: cos(11) Error Comparison

To find out more about cos(11), the Modelsim CORDIC iterations were plotted on Excel. Interesting, there are two spikes of 360000% difference between the actual result and the CORDIC stage result. In Figure 18, theses two points are clearly seen as the 4th, 6th, and the 7th stage. This operation however is normal, as the CORDIC algorithm can rotate. Hence, this simulation further justifies using a minimum amount of CORDIC stages – in our case, 16.

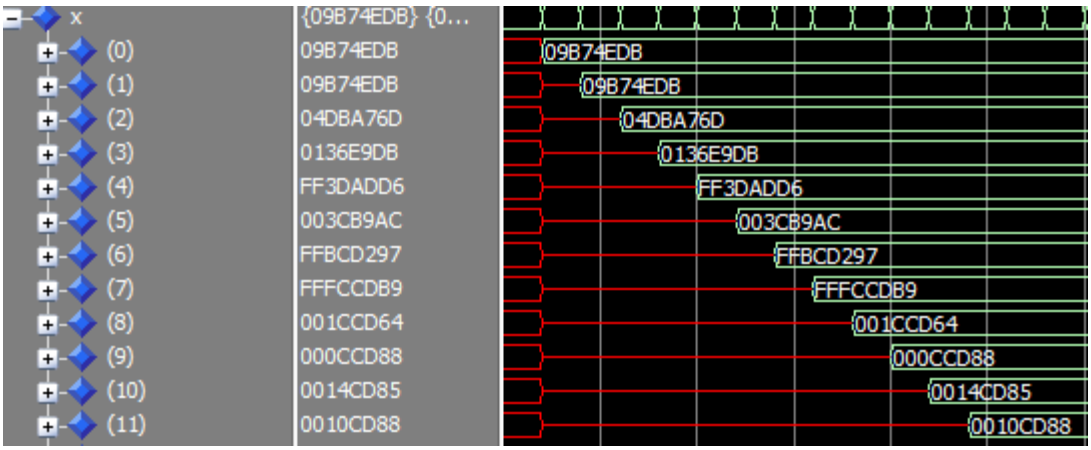

Figure 18: cos(11) CORDIC stages

b) Overall

TABLE XI.

Latency	MATLAB	Math.h	CORDIC(x)
Case 1	57879.8730	57879.867188	57881.226562
Case 2	-126818.14	-76973.640625	-76943.906250
Case 3	-12774366.3	37022500	37025504

V. Task 8: Add Dedicated Hardware Bock to compute the arithmetic expression

A. *Aim*

Task 8 was an open-ended task – the aim is to accelerate the arithmetic expression using whatever methods available. Several methods, some successful, some not, are documented below.

B. *Method 1: Unrolling the CORDIC hardware block*

1) *Methodology and Implementation*

The CORDIC algorithm has a `for` loop to calculate the iterations of the algorithm. This loop takes n cycles to execute, where n is the number of stages of the CORDIC algorithm. In Figure 19, the Modelsim waveform is shown. Figure 21, 22 show the doubling of commands in the `for` loop, and the halving of the for loop iterations.

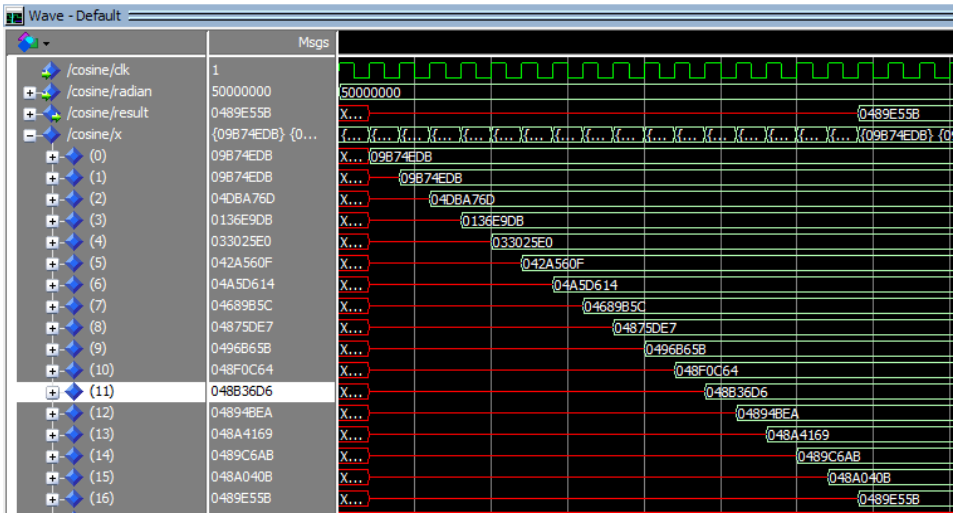

Figure 1919: Showing correct CORDIC result in 16 clock cycles

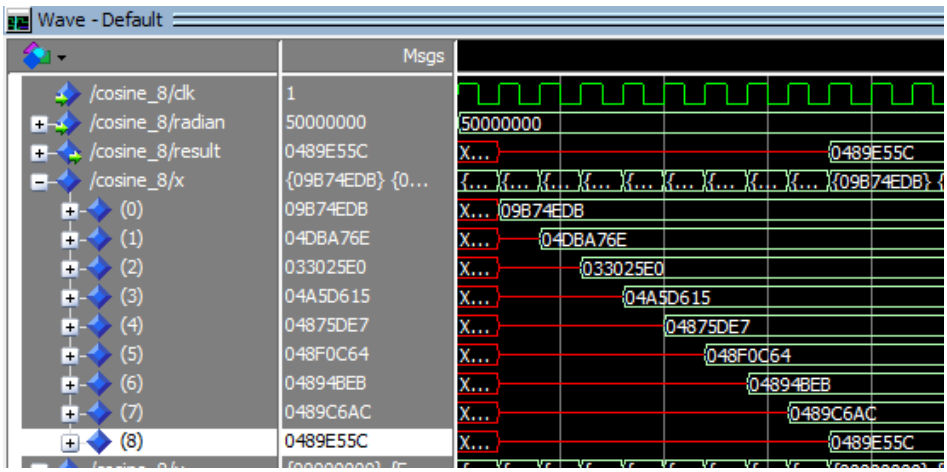

Figure 200: Showing correct CORDIC result in 8 clock cycles

Figures 19 and 20 show unrolling the loop cutting the cycles needed by half, and giving the same result. The very slight difference in the least significant byte can be attributed to the shifting when unrolling. Instead of shifting by i-1, the unrolled version shifts by (i-1)*2. This comes from recursively computing the CORDIC equation, substituting i-1 into i-2's iteration.

```
for index in 1 to 16 loop
    if z(index - 1)(31) = '1' then
        z(index) <= slv(signed(z(index - 1)) + signed(table(index - 1)));
        x(index) <= slv(signed(x(index - 1)) + signed(asr(y(index - 1), index - 1)));
        y(index) <= slv(signed(y(index - 1)) - signed(asr(x(index - 1), index - 1)));
    else
        z(index) <= slv(signed(z(index - 1)) - signed(table(index - 1)));
        x(index) <= slv(signed(x(index - 1)) - signed(asr(y(index - 1), index - 1)));
        y(index) <= slv(signed(y(index - 1)) + signed(asr(x(index - 1), index - 1)));
    end if;
end loop;
```

Figure 21: VHDL CORDIC implementation

```
for index in 1 to 8 loop
    if z(index - 1)(31) = '1' then
        if (signed(z(index - 1)) + signed(table((index - 1) * 2))) > to_signed(0, 32) then
            --di=-1 di+1 = 1
            z(index) <= slv(signed(z(index - 1)) + signed(table((index - 1) * 2)) - signed(table((index - 1) * 2 + 1)));
            x(index) <= slv(signed(x(index - 1)) + signed(asr(y(index - 1), (index - 1) * 2)) - signed(asr(y(index - 1), (index - 1) * 2 + 1)) + signed(asr(x(index - 1), (index - 1) * 4 + 1)));
            y(index) <= slv(signed(y(index - 1)) - signed(asr(x(index - 1), (index - 1) * 2)) + signed(asr(x(index - 1), (index - 1) * 2 + 1)) + signed(asr(y(index - 1), (index - 1) * 4 + 1)));
        else
            --di= -1 di+1= -1
            z(index) <= slv(signed(z(index - 1)) + signed(table((index - 1) * 2)) + signed(table((index - 1) * 2 + 1)));
            x(index) <= slv(signed(x(index - 1)) + signed(asr(y(index - 1), (index - 1) * 2)) + signed(asr(y(index - 1), (index - 1) * 2 + 1)) - signed(asr(x(index - 1), (index - 1) * 4 + 1)));
            y(index) <= slv(signed(y(index - 1)) - signed(asr(x(index - 1), (index - 1) * 2)) - signed(asr(x(index - 1), (index - 1) * 2 + 1)) - signed(asr(y(index - 1), (index - 1) * 4 + 1)));
        end if;
    else
        if (signed(z(index - 1)) - signed(table((index - 1) * 2))) > to_signed(0, 32) then
            --di= 1 di+1 = 1
            z(index) <= slv(signed(z(index - 1)) - signed(table((index - 1) * 2)) - signed(table((index - 1) * 2 + 1)));
            x(index) <= slv(signed(x(index - 1)) - signed(asr(y(index - 1), (index - 1) * 2)) - signed(asr(y(index - 1), (index - 1) * 2 + 1)) - signed(asr(x(index - 1), (index - 1) * 4 + 1)));
            y(index) <= slv(signed(y(index - 1)) + signed(asr(x(index - 1), (index - 1) * 2)) + signed(asr(x(index - 1), (index - 1) * 2 + 1)) - signed(asr(y(index - 1), (index - 1) * 4 + 1)));
        else
            --di= 1 di+1 = -1
            z(index) <= slv(signed(z(index - 1)) - signed(table((index - 1) * 2)) + signed(table((index - 1) * 2 + 1)));
            x(index) <= slv(signed(x(index - 1)) - signed(asr(y(index - 1), (index - 1) * 2)) + signed(asr(y(index - 1), (index - 1) * 2 + 1)) + signed(asr(x(index - 1), (index - 1) * 4 + 1)));
            y(index) <= slv(signed(y(index - 1)) + signed(asr(x(index - 1), (index - 1) * 2)) - signed(asr(x(index - 1), (index - 1) * 2 + 1)) + signed(asr(y(index - 1), (index - 1) * 4 + 1)));
        end if;
    end if;
```

Figure 212: VHDL unrolled CORDIC implementation

However, unrolling the CORDIC is at the expense of system resources. Unrolling the original 28 stage CORDIC into 14*2 stages already requires more LE's than the DE0 has (including the NIOS II). We therefore settled on using 8*2 stages, as the 74% LE requirement would mean there was enough space left over for the NIOS II processor. Table XII shows the LE's needed by each CORDIC implementation (base NIOS II system subtracted away).

TABLE XII.

Resource	CORDIC 16 Stages (8*2)	CORDIC 16 Stages	CORDIC 28 Stages (14*2)	CORDIC 28 Stages
Logic Elements	8200 (59%)	3007 (19%)	14431 (94%)	4413 (29%)
Embedded Multipliers	4	4	4	4
Memory Bits	291840	291840	291840	291840
Block Latency	8	16	7	28

2) Performance

Unrolling the loop by 2 will cut the number of cycles needed to calculate the result by half, 4 times will cut it by 4 times, etc.

Implementing this in a custom instruction gives a healthy boost of performance.

TABLE XIII.

Resource	Task 6	CORDIC 16 Stages	CORDIC 16 Stages (8*2)
Case 1	23	1	1
Case 2	1143	28	14
Case 3	114331	2714	1343
Improvement of Case 3	N/A	97.6%	98.8%

3) Accuracy

Accuracy did not change as the LSB differences after unrolling are beyond what floating point can represent. The accuracy is as expected.

TABLE XIV.

Resource	Task 6	CORDIC 16 Stages	CORDIC 16 Stages (8*2)
Case 1	57879.867188	57881.226562	57881.226562
Case 2	-76973.640625	-76943.906250	-76943.906250
Case 3	37022500	37025504	37025504

C. *Method 2: Implementing the whole arithmetic expression in hardware*

```
architecture main of combined_func2 is
    signal exponent : std_logic_vector(7 downto 0);
    signal half_x   : std_logic_vector(31 downto 0);
    signal cos_out  : std_logic_vector(31 downto 0);
    signal sq_out   : std_logic_vector(31 downto 0);
    signal mult_out : std_logic_vector(31 downto 0);
begin
    sub_exponent : process(x)
    begin
        exponent <= std_logic_vector(signed(x(30 downto 23)) - 1);
    end process sub_exponent;

    div2 : process(exponent, x)
    begin
        if to_integer(signed(x)) = 0 then
            half_x <= (others => '0');
        else
            half_x <= x(31) & exponent & x(22 downto 0);
        end if;
    end process div2;

    cosine_blk : entity cos_wrapper port map(
            clk,
            x,
            cos_out
        );

    sq_block : ENTITY combined_func_mult PORT map(
            clk,
            x,
            x,
            sq_out
        );

    mult_block : ENTITY combined_func_mult PORT map(
            clk,
            sq_out,
            cos_out,
            mult_out
        );

    adder_block : ENTITY combined_func_add PORT map(
            clk,
            mult_out,
            half_x,
            result
        );

end architecture main;
```

Figure 22: VHDL instantiations, instead of using block diagram wires

A custom VHDL entity was created to calculate the arithmetic expression in one custom instruction. This includes floating point division by 2, floating point squaring, and a floating point multiplication, multiplying the squared input with the cosine of the lookup table value of the input value, i.e. floor(x/4)-32. Since some custom instructions are nested inside others, and since the latencies of the custom instructions are different, doing all the computation in one custom instruction saves on wasted clock cycles in calling the many custom instructions. The custom instruction is implemented as float result = MY_ADD(MY_COMBINED_FUNCTION(x[i]), result). Again, as shown in Table XV, it further reduces the latency, without compromising on accuracy.

TABLE XV.

Resource	Task 6	CORDIC 16 Stages (8*2)	Combined Function
Case 1	23	1	1

Case 2	1143	14	11
Case 3	114331	1343	1024
Improvement of Case 3	N/A	97.6%	99.1%

D. Method 3: Implementing the whole arithmetic expression in hardware, utilizing other input ports

The same VHDL entity was used in a second version of the custom instruction hardware block, but this time, the datab input is utilized. Instead of having a floating point adder custom instruction being called in our C program, the previous floating point result is passed into datab, which is then directly passed into the final adder. Hence, the implementation now is float result = MY_COMBINED_FUNCTION_2(x[i], result). Interestingly, there is no obvious performance increase (+/-1 tick if present). The accuracy is unchanged.

E. Method 4: Using FIFO's to synchronise timing

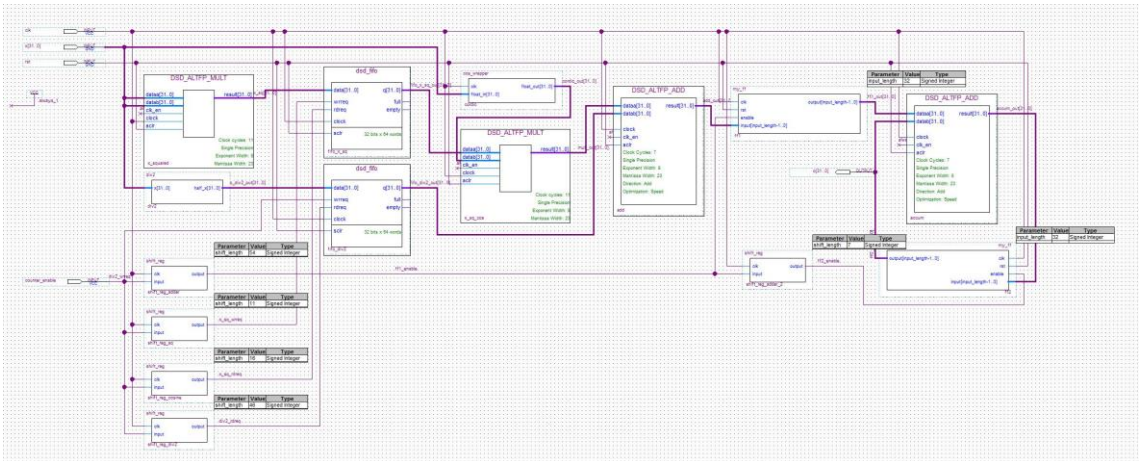

Figure 23: Using FIFO's in our pipe design

From section III.F, the NIOS II needs to be 'tricked' into pushing data into our custom instruction every clock cycle. This is done by tying the done signal to high. Hence, the NIOS II will keep pushing new data into the custom instruction, hopefully filling up the pipeline. However, since the different hardware blocks inside our custom instruction have different latencies, e.g. the division by 2 being combinational, addition being 7 cycles, multiplication being 5, CORDIC being 8, etc., if the timing is not taken into account, the final adder will be adding together results from different inputs. Hence, FIFO's are inserted in the middle of the data flow to essentially let the longer latency blocks catch up. Since FIFOs store the pushed in data, the read/write signals can be manually set to be high when all the floating point/CORDIC operations have finished in their respective blocks – this way, ,the final adder will be adding together the correct intermediate numbers.

To do this, custom VHDL entities were written, such as a variable length shift register to implement the delaying of any number of cycles, as well as registers to ensure the adder is receiving a constant input throughout the 7 cycles that it needs to add the two inputs. Figure 24 shows the final block diagram; Figure 25 shows the simulated block diagram in Modelsim, using the .do file below. The cyan waveform is the input, the magenta waveform is the output. The pipe was tested for the first few elements of test case 1; the outputs match what is expected (n.b. both input and output are floating point).

```
restart -force -nowave;

add wave -r /*

force clk 1 0ns, 0 10ns -repeat 20ns;
force rst 1 0ns, 0 40ns;
force x 32'h0 100ns, 32'h0x40a00000 900ns, 32'h0x41200000 1800ns, 32'h0x41700000 2700ns;
force counter_enable 1 100ns, 0 120ns, 1 900ns, 0 920ns, 1 1800ns, 0 1820ns, 1 2700ns, 0
2720ns;
run 10000ns
```

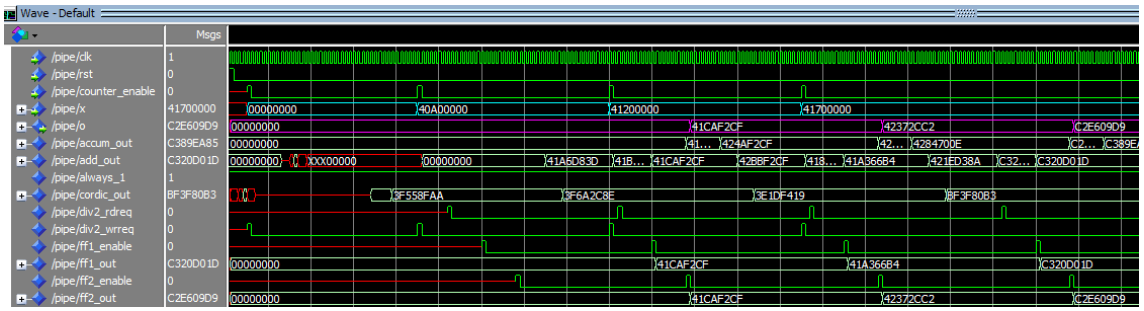

Figure 24: Correct Modelsim simulation

However, this method only worked in simulation. After synthesis onto the DE0, the output results were wrong. In reality, there is no way of figuring out the exact time that the NIOS II decides to push data into the custom instruction, even if the done signal is tied to 1 (our Modelsim test bench assumed one word every 40 clock cycles to try to simulate the function call latency). Furthermore, if there are any hardware interrupts, or processor stalls, the timing that our dataflow expects will be wrong. As the test cases have accumulators, any timing errors will lead to mistakes in the output; an accumulator will only make this error propagate throughout the entire test case vector. Hence, this method only works in theory, and not in reality.

F. *Method 5: Direct Memory Access (DMA)*

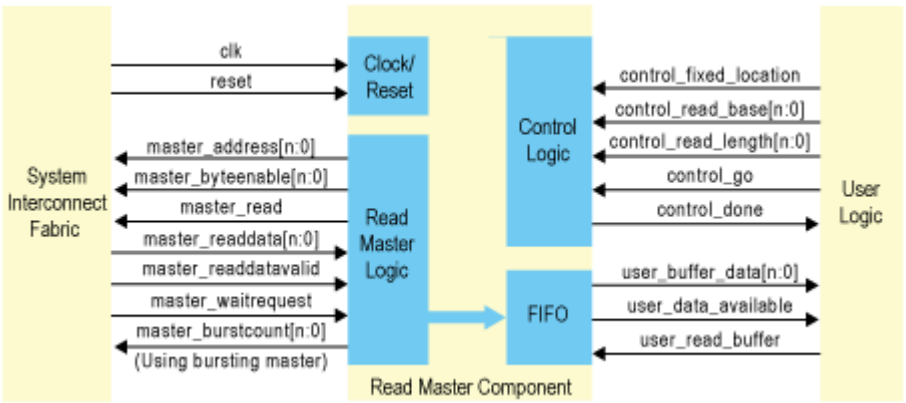

Figure 25: Altera DMA template connection flow [9]

Following method 4's idea of having the NIOS II push data into the custom instruction as quickly as possible, a direct memory access method was investigated. A DMA module would pull in a block of data directly from the SDRAM, bypassing handshaking schemes with the NIOS II. This module would then feed the data into the custom instruction at a known rate (default is one word per clock cycle), making timing when processing on the data block has finished straightforward. With a FIFO to buffer the incoming data to our combined function block, the theoretical performance increase should be substantial.

```
begin

    --count_byte: entity counter
    --port map(n_byte,clk_mm,start,rdreq,delay);
    control_finish : entity fsm_finish port map(clk_mm, start, empty, control_done, delay);
    tmp_delay <= '0';

    DMA : entity MemRead port map(clk, reset, start, transfer_done, clk_en, in_addr, n_byte,
                                  clk_mm, reset_mm, master_address, master_read, master_byte
                                  full, load_en, fsm_wrreq, control_done, data_avaliable
        );

    data_register : process
    begin
        wait until clk_mm'event and clk_mm = '1';
        if load_en = '1' then
            cordic_in <= ram_out;
        end if;
    end process data_register;

    fifo : entity fifo_cordic port map(
            start, clk_mm, cordic_out, rdreq, fifo_wrreq, empty, full, fifo_out
        );

    accum : entity accumulator port map(
            clk_mm, fifo_out, start, empty, result, rdreq
        );

    count_finish : entity lsr34 port map(
            start, clk_mm, clk_en, delay, done
        );

    delay_wrrqt : entity lsr26 port map(
            start, clk_mm, clk_en, fsm_wrreq, fifo_wrreq
        );

    cordic_blk : entity combined_func2 port map(clk_mm, cordic_in, cordic_out, clk_en);

end architecture main;
```

Figure 26: VHDL DMA blocks

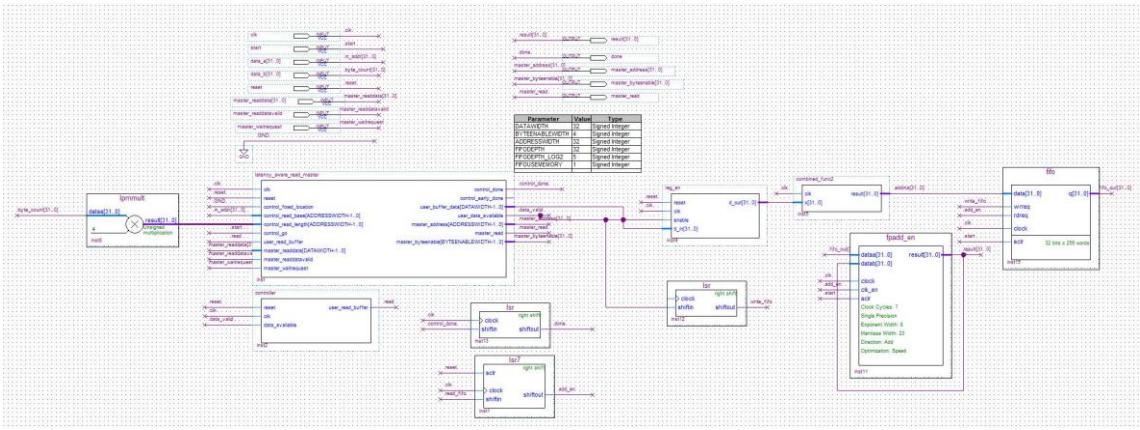

Figure 27: DMA block diagram

By using Altera's DMA template [9], a design was created in Quartus using a custom VHDL entity to instantiate custom FSM's and Altera's own DMA controller, as well as other hardware blocks (Figure 26-28). With varying latencies of the different mathematical blocks, start/done/enable/reset signals would need to be manually set. Three finite state machines (FSMs), were written VHDL, to control the DMA module, the combined function block, and the accumulator. They control the memory reading process by the DMA, the enabling of the FIFO to pop data into our combined function, and the clocking of the accumulator respectively to enable a correct output.

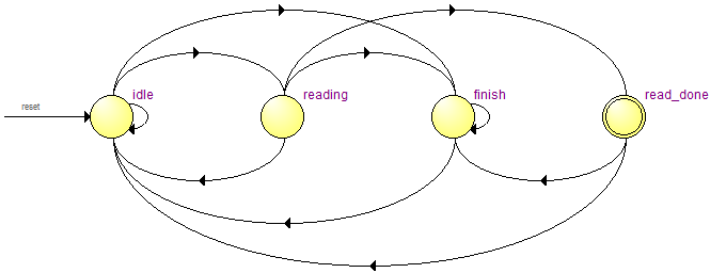

Figure 28: DMA FSM

Quartus has an in-built state machine viewer to visualize our custom FSM's. Figure 29 shows the custom FSM used to control the DMA memory access module. During the reading state, a counter is used to count the number of words the DMA memory access module reads. For test case 3, this would be 255001*32/8. During the block read, the FSM goes through idle->reading->read_done. Reading is two cycles long, one for the data to update to the new word, and another for register to load the value in. If the whole test vector has been read in, the state goes from read_done to finish, entering an infinite loop to stop the DMA module from doing anything else. The results from the word read are passed into a FIFO.

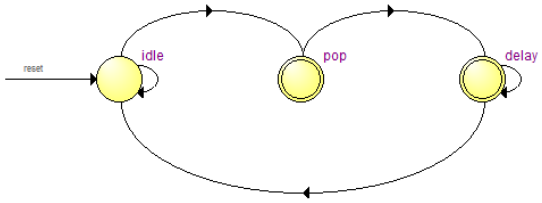

Figure 29: Accumulator FSM

Lastly, another FSM was needed to control the accumulator since it needs 7 clock cycles to perform the floating point add. This FSM merely pops one result from FIFO into the accumulator, and then waits 7 cycles by the use of a down counter in the delay state. After 7 cycles, another result is popped in – the previous result will have finished calculating and the fed-back result will be stable for the next accumulate.

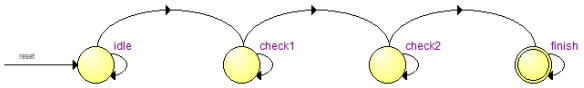

Figure 30: Overall FSM

A FIFO has two signals to signify the status, empty and full. If the FIFO is empty, our overall FSM will assert write request, and a result from the combined function will be read in. If the FIFO is full, the DMA module will be held in its idle state until some of the FIFO filled from our combined arithmetic function for calculation. There are also check states for when the FIFO is empty, as the nature of the DMA should mean that an empty FIFO means the DMA has entered the finish state.

However, after synthesis and Modelsim testing, a case was found in that the last word from our test vector was still propagating through the intermediate registers into the FIFO for the next clock cycle. At the rising edge of the clock at that instance, the FIFO is empty, so our FSM went to its finish state. However, as the last word has still not been processed, one element of the test vector is lost, leading to a wrong result.

Hence, there are two states to check for the finish flag. If the FIFO stays empty for two clock cycles, the DMA has indeed finished reading, and there are no new incoming values (intermediate results in registers will enter the FIFO on the next clock cycle, deserting the empty flag). Hence the system is in a finished state.

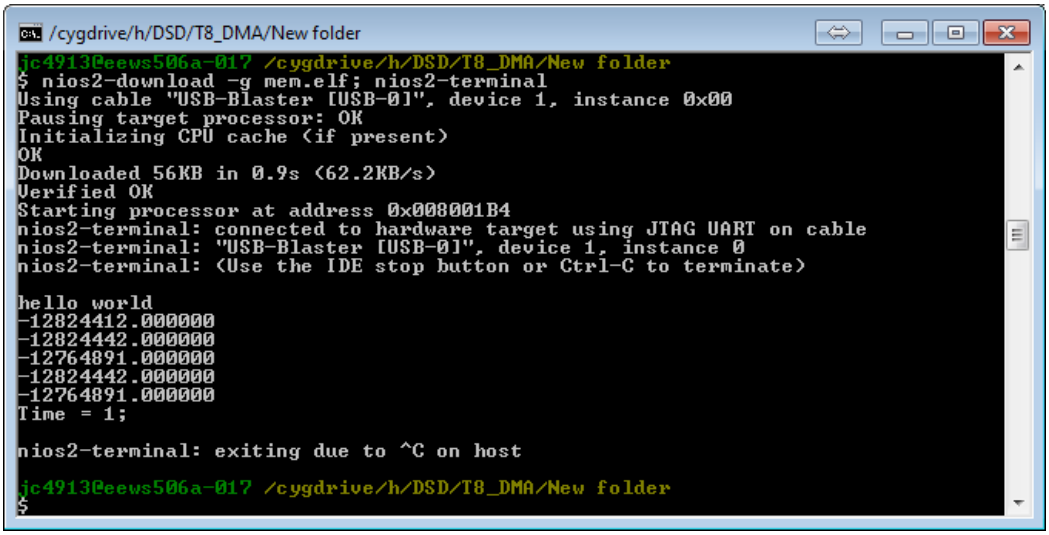

Figure 31: Wrong DMA Results

During synthesis and testing, our DMA module showed that it ran the whole of test case 3 in one tick. However, the result is both inconsistent and wrong.

As testing was time constrained, this may be due to a wide range of factors. First, the master template may need offset addressing [10]. If so, this will cause our data fetched to be wrong. As such, the wrong result will be accumulated and the end result will be obviously wrong. Secondly, since our DMA template used two clocks, the reset signal logic may be wrong, leading to inconsistent cutoffs in fetching the SDRAM data. These problems can be further investigated using SignalTap, allowing us to see values of different signals while the board is running.

G. *Method 6: Overclocking the Design*

Overclocking the PLL is the most obvious tweak to increase performance. The base clock should not overclocked; the PLL's ratio should. However, from previous TimeQuest reports, our highest fmax was around 60MHz. However, there is nothing stopping us from forcing the PLL to have a clock frequency higher than 60MHz [11]. However, if we set the system to have a PLL clock speed exceeding the reported fmax, there might be other errors. Since our test cases have a small range of numbers (0-255), there is no way of exhaustively testing the test bench without a test vector that includes an exhaustive list of floating point numbers, as such, we did not overclock our design, staying with the original 50MHz.

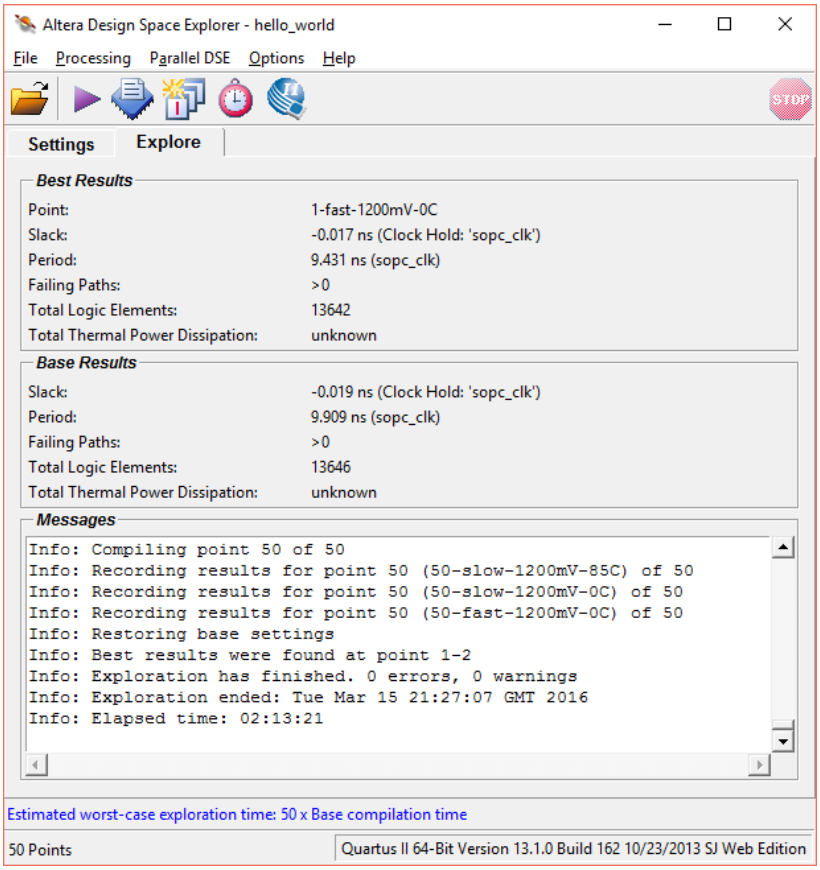

Figure 32: Seed Sweep Window

TimeQuest's fmax comes from finding the critical path in our design. However, this critical path is dependent on the fitter, and how it maps FPGA resources positions in the FPGA. If it places two dependent hardware blocks at opposite sides of the FPGA fabric, the critical path will be extremely long, leading to a small fmax [12]. Quartus includes a tool to modify the behavior of the fitter, changing the location where the fitter begins searching for free FPGA fabric, called Altera Design Space Explorer.

Running a seed sweep [13] (seed determines where the fitter starts), gave the following results:

TABLE XVI.

Seed	sopc_clk fmax	Sopc_clk worst-case slack	Logic Elements	Embedded Multipliers	Memory Bits	Resource Usage
1 (Base)	57.43 MHz	2.558 ns	13646	7	387072	57%
2	106.03 MHz	8.194 ns	13642	7	387072	57%
...	...	...	...	...	...	...
51	57.41 MHz	2.578 ns	13646	7	387072	57%

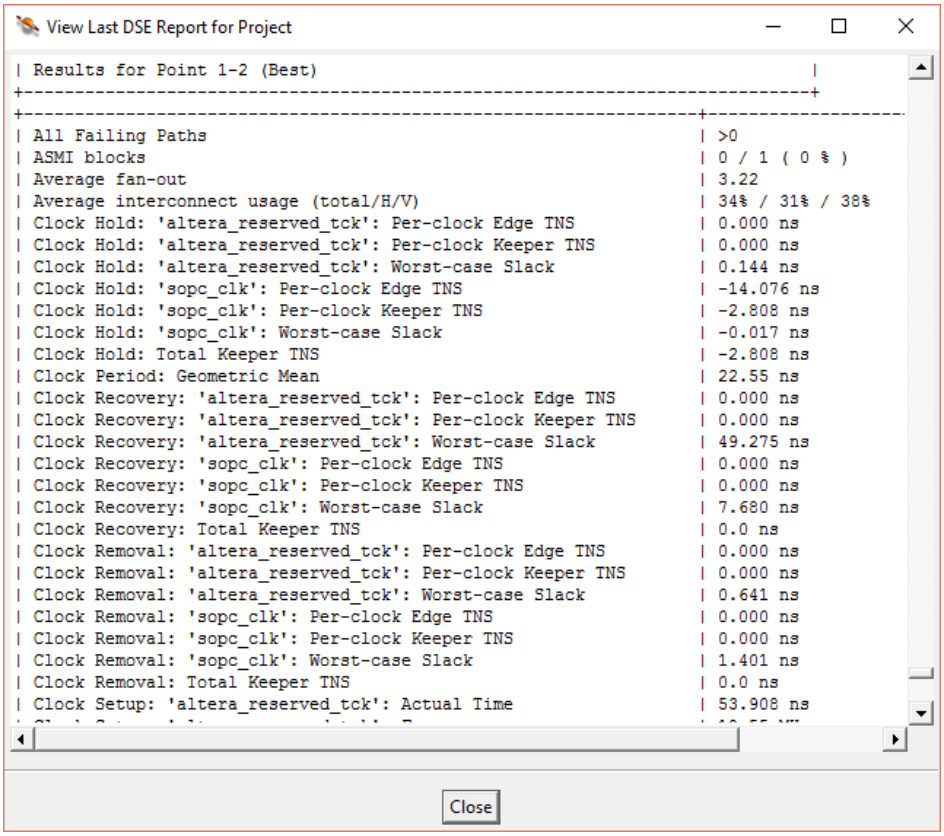

Figure 33: Seed Sweep Timing Results

Figure 34 shows the final report after the seed sweep. As such, with a seed of 2, and no negative worst case slack, we were free to overclock the PLL to 100MHz. Both clk0 and clk1 were overclocked to give the best performance. As the hardware now runs on twice the clock speed, the expected latency decrease was by 50%. This was seen in the testing, as shown in Table XVII. Accuracy was not compromised.

TABLE XVII.

Resource	Task 6	50MHz	100MHz
Case 1	23	1	0
Case 2	1143	11	5
Case 3	114331	1024	515
Improvement of Case 3	N/A	99.1%	99.5%

H. *Method 7: Turning on NIOS II Optimizations*

Turning on software optimizations to Level 3 increased our performance by a healthy amount. This was both unexpected and expected. From previous projects that used the gcc compiler, turning on the optimization level gave healthy boosts to performance. However, all those projects were purely done in software, and from the optimization guide [14], Level's 2 and 3 both utilize techniques such as loop unrolling, software pipelining, function optimization etc. Therefore, a software performance was expected. However, turning on Level 3 gave a big performance increase, even with our combined function custom instruction. This was unexpected as the software optimization should not affect our hardware instruction. Hence, we concluded that the NIOS II had optimized some way of running our test bench and interfacing with our custom instruction. Table XVIII shows the latency improvements. Our hardware utilizes the aforementioned overclock of 100MHz. Accuracy was not compromised.

TABLE XVIII.

Resource	Task 6	No Optimization	Level 3 Optimization

Case 1	23	0	0
Case 2	1143	5	1
Case 3	114331	515	153
Improvement of Case 3	N/A	99.5%	99.9%

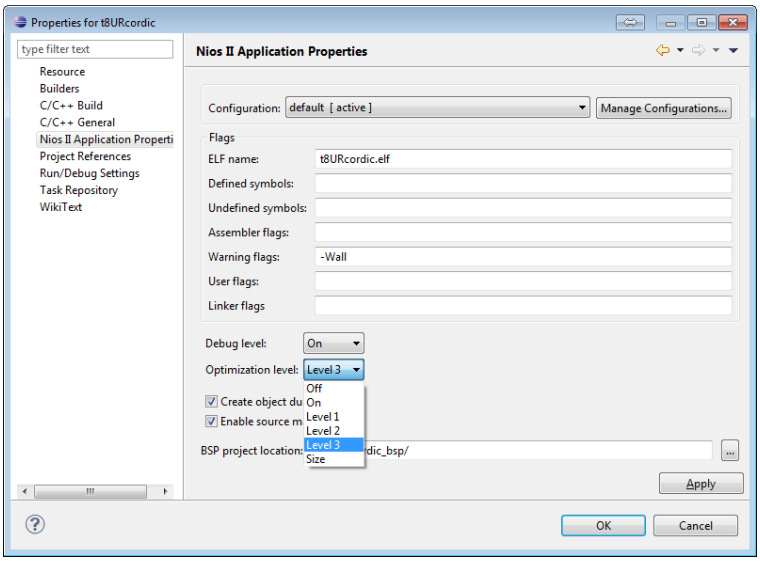

Figure 34: Optimization Level

I. *Resource Usage and Accuracy*

Flow Summary	
Flow Status	Successful - Wed Mar 16 19:09:33 2016
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Full Version
Revision Name	hello_world
Top-level Entity Name	hello_world
Family	Cyclone III
Device	EP3C16F484C6
Timing Models	Final
Total logic elements	13,667 / 15,408 (89 %)
Total combinational functions	13,203 / 15,408 (86 %)
Dedicated logic registers	4,248 / 15,408 (28 %)
Total registers	4316
Total pins	47 / 347 (14 %)
Total virtual pins	0
Total memory bits	291,975 / 516,096 (57 %)
Embedded Multiplier 9-bit elements	18 / 112 (16 %)
Total PLLs	1 / 4 (25 %)

Figure 35: Last resource usage

Figure 35 shows the final improvement's synehesised resources. The unrolled CORDIC block took up a lot of the available resources, and can be remedied by using an iterative fixed point adder. However, since the main point is performance, this was not done due to time constraints. Accuracy was also uncompromised throughout task 8.

VI. Further Improvements

There are a few further improvements that can be done with more time. First, instead of using a floating point arithmetic hardware blocks, a combinational fixed point hardware block can be used to calculate results in 1 clock cycle. This requires converting each incoming result to 64-bit fixed point. A simple system of 32 integer to 32 fractional bits can be used. However, this method can have wrong results for different test vectors. As our current test vector is between 0-255, the 32:32 format gives enough precision. However, different test cases might not. Another improvement is to use the Altera floating point accumulator IP block. This gives a result in 1 cycle with a 4 cycle latency, a much better performance than our current feedback adder block. Lastly, the DMA module can utilize burst read, which increase the memory fetching speed.

VII. Conclusion

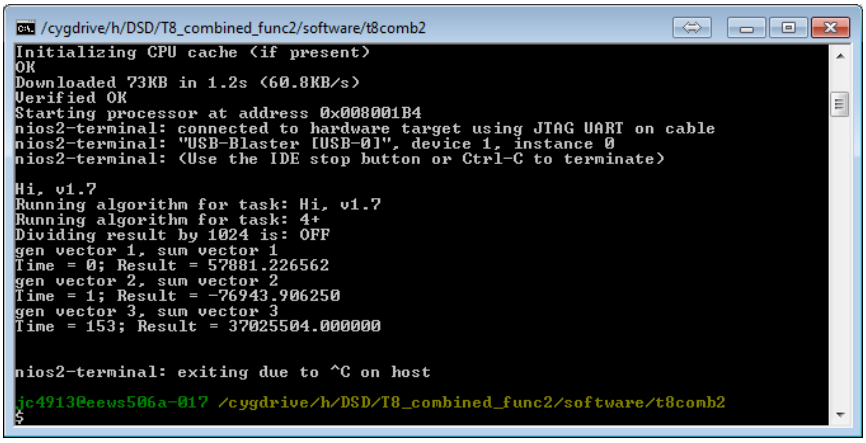

Figure 36: Final Console Result

From task 4-8, many different improvements were made to our system to improve the performance. Many calculations were mapped to hardware, which led to increased resources but increased performance. The accuracy stayed mostly constant throughout, and there were no instances where a huge improvement had to be traded off for much less accuracy. Although our CORDIC function has some errors, these errors are negligible for the performance gained. The DMA result still requires work on the timing of the done signal – the final result is incorrect.

Our final design utilizes the unrolled CORDIC block in a combined arithmetic function, called by a Level 3 Optimized NIOS II on a board overclocked to 100MHz. The following table illustrates how each modification on test case 3 changed the latency. With the improvements in task 8, the arithmetic expression was indeed sped up by a very significant amount – success!

TABLE XIX.

Task	Modification	Test Case 3 Latency (ticks)	Test Case 3 Throughput (Bps)	Latency Improvement from Reference Design (%)	Throughput Improvement from Reference Design (%) (%)
4	Reference	382038	166.8688717	N/A	N/A
4.1	2KB->32KB Cache	288024	221.33659	24.6085%	32.6410%
5	Embedded Multipliers	136496	467.0484849	64.2716%	179.8895%
6	Add Floating Point Add/Mult	114332	557.588864	70.0731%	234.1479%
7	Change cos() to hardware CORDIC	2718	23454.83812	99.2886%	13955.8499%
8	Unroll CORDIC to 8*2 stages	1373	46431.3547	99.6406%	27725.0546%
8.1	Convert all to block diagram hardware	1024	62256.10352	99.7320%	37208.3984%
8.2	Use datab to reduce nested add	1023	62316.95992	99.7322%	37244.8680%
8.3	Overclocking to 100MHz	515	123786.8932	99.8652%	74082.1359%
8.4	Level 3 Optimization	153	416668.3007	99.9600%	249598.0392%
(8.5)	DMA	1	63750250	99.9997%	38203700.0000%

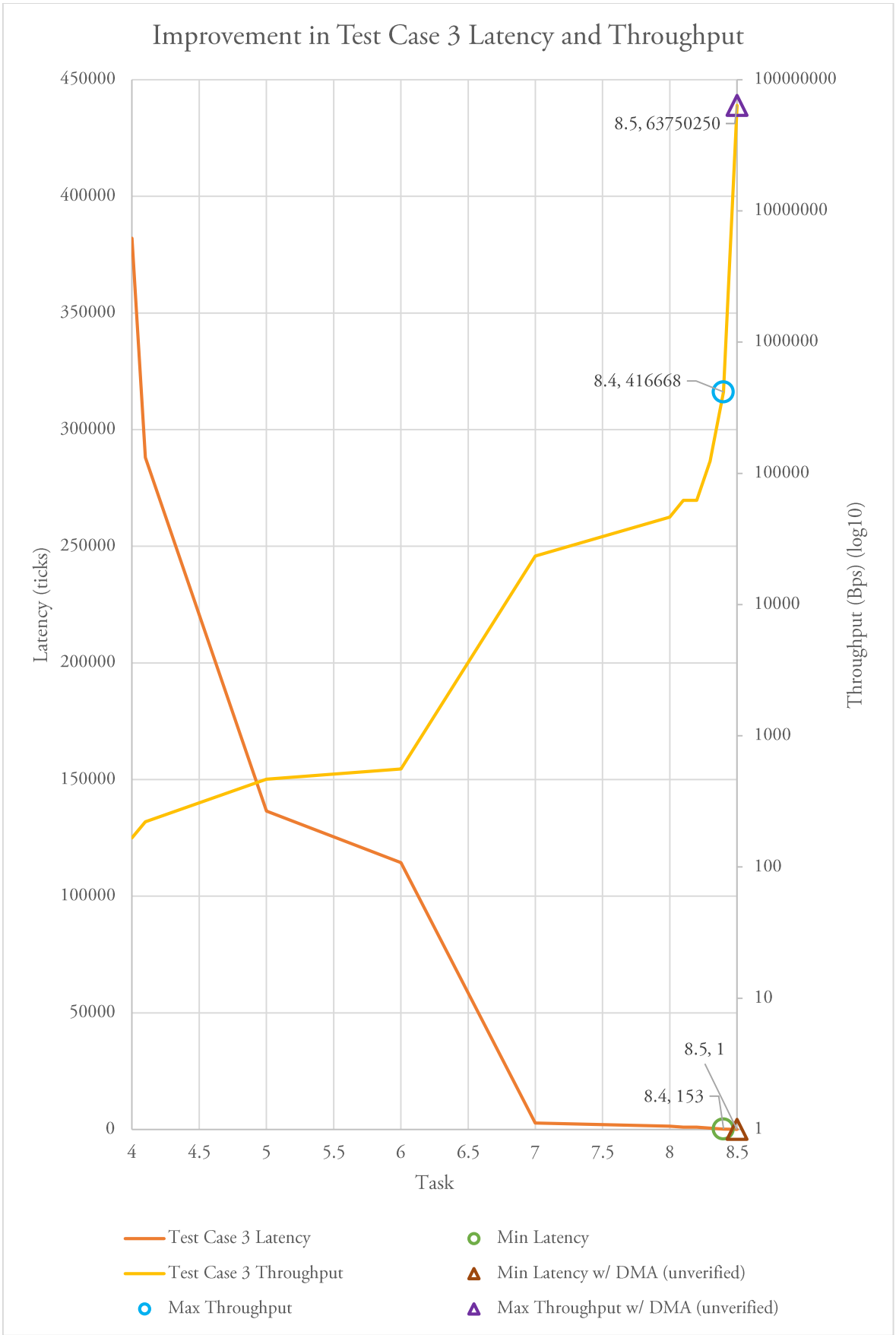

Figure 37: Final Latency/Throughput Graph

VIII. References

[1] C. Bouganis, "DSD Coursework," [Online]. Available: https://bb.imperial.ac.uk/bbcswebdav/pid-766162-dt-content-rid-2726885_1/courses/DSS-EE3_05-15_16/DSD_coursework.pdf. [Accessed 15 3 2016].

[2] Altera, "AN 188: Custom Instructions for the Nios Embedded Processor," [Online]. Available: http://www.eecg.toronto.edu/~pc/courses/432/doc/an188.pdf. [Accessed 15 3 2016].

[3] 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].

[4] N. Pipenbrinck, "Why is floor() so slow?," 5 5 2009. [Online]. Available: http://stackoverflow.com/questions/824118/why-is-floor-so-slow. [Accessed 13 3 2016].

[5] F. Lis and G. Pan, "Speeding up the CORDIC algorithm with a DSP," [Online]. Available: http://www.embedded.com/print/4007665. [Accessed 12 3 2016].

[6] C. Bouganis, "Digital System Design Topic 7: Function Evaluation," [Online]. Available: https://bb.imperial.ac.uk/bbcswebdav/pid-590093-dt-content-rid-2415282_1/courses/DSS-EE3_05-15_16/Topic%207%20-%20Function%20Evaluation.pdf. [Accessed 15 3 2016].

[7] Ercegovac and Lang, "CORDIC Algorithm and Implementations," 2003. [Online]. Available: http://web.cs.ucla.edu/digital_arithmetic/files/ch11.pdf. [Accessed 14 3 2016].

[8] W. Kahan, "IEEE Standard 754 for Binary Floating-Point Arithmetic," 1 10 1997. [Online]. Available: http://www.cs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF. [Accessed 14 3 2016].

[9] Altera, "Avalon Memory-Mapped Master Templates," [Online]. Available: https://www.altera.com/support/support-resources/design-examples/intellectual-property/embedded/nios-ii/exm-avalon-mm.html. [Accessed 15 3 2016].

[10] Altera, "DMA Controller Core," [Online]. Available: https://www.altera.com.cn/zh_CN/pdfs/literature/hb/nios2/n2cpu_nii51006.pdf. [Accessed 18 3 2016].

[11] D. Tweed, "Internal fmax of FPGA program," stackexchange.com, 27 1 2014. [Online]. Available: http://electronics.stackexchange.com/questions/97762/internal-fmax-of-fpga-program. [Accessed 15 3 2016].

[12] K. Ayob, Fundamentals of Timing in FPGAs, Amazon, 2015.

[13] R. Scoville, "Fitting Algorithms, Seeds, and Variation," 1 11 2011. [Online]. Available: http://www.alterawiki.com/uploads/e/e6/FittingAlgorithms_and_SeedSweeps.pdf. [Accessed 15 3 2016].

[14] Texas Instruments, "TMS320C6000 Optimizing Compiler v7.4," 7 2012. [Online]. Available: http://www.ti.com/lit/ug/spru187u/spru187u.pdf. [Accessed 23 2 2016].

IX. Appendix

A. *Table of Figures*

B. *MATLAB floor(x/4)-32 Script*

```
i=1;
for index = -32:1:31
        mod_vec(i)=rem(index,2*pi);
        i=i+1;
end ;
mod_vec=sfi(mod_vec,32,28); bin(mod_vec); thres_vec(1)=0;
```

C. *All latency/resource usage results*

TABLE XX.

Task	4	5	6	7	8
1	55	27	23	1	0
2	2272	1356	1144	27	1
3	227074	136496	114332	2718	153

TABLE XXI.

Task	4	5	6	7	8
LE	2989	3218	4993	7509	13667
EM	0	4	11	11	18
MB	291840	291840	291840	291912	291915
RU	0.253	0.27	0.329	0.384	0.538