Digital Systems Design Coursework Report 3

Group 7
Aadi Desai ad3919@ic.ac.uk
Alden Joseph ajs19@ic.ac.uk

1 Introduction

In this report, the system previously implemented was modified to improve performance in several ways. The former design of the system had computed the function given below using emulated floating-point operations, which are implemented using the built-in fixed-point hardware of the NIOS/f soft-core.

$$y = \frac{x}{2} + x^2 \cos\left(\frac{x - 128}{128}\right)$$

The system was modified in the following ways:

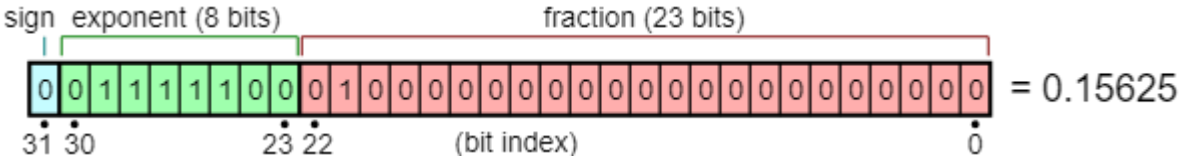
- Adding hardware blocks to compute floating point operations
- Adding a dedicated hardware block to compute the cosine function with sufficient accuracy
- Wrapping the entire inner part of the function in a hardware block to reduce instruction fetching overheads, as well as the data flow critical path, since data does not need to go back and forth from the custom instruction area of hardware to the NIOS softcore
- Further function optimisations to improve performance

This report will narrate through the details and design choices undertaken in each task, as well as the associated consequences and effects on performance and resource utilisation.

2 Task 6: Add Hardware Floating-Point Units

This section focused on improving performance by adding support for floating point arithmetic in NIOS using custom instructions to perform the associated operations. The previous implementation had emulated such operations using fixed-point additions, which added a high overhead to the performance of the design.

The IEEE-754 floating point format was the given standard for all computations performed by the hardware:



Upon inspection of the given function, the floating-point operations required were addition and multiplication. The division and subtraction blocks could be omitted by restructuring the function as follows:

$$y = \frac{x}{2} + x^2 \cos\left(\frac{x-128}{128}\right) \quad \text{à} \quad y = \frac{x}{2} + x^2 \cos\left(\frac{1}{128}(x + (-128))\right)$$

The associated functions for addition and multiplication are given as follows:

Floating point addition/subtraction:

$$(\pm M_1 \cdot 2^{E_1}) \pm (\pm M_2 \cdot 2^{E_2}) = \left(\pm M_1 + \frac{M_2}{2^{E_1 - E_2}}\right) \cdot 2^{E_1}$$

Floating point multiplication:

$$(\pm M_1 \cdot 2^{E_1}) \times (\pm M_2 \cdot 2^{E_2}) = \pm M_1 \cdot M_2 \cdot 2^{E_1 + E_2}$$

To implement the custom instructions to support floating point arithmetic, the ALTERA FP_FUNCTIONS IP was used. The FP_FUNCTIONS block was chosen due to its flexibility in choosing the number of cycles needed by the generated block and superior performance (lower minimum latency) when compared against other options, such as the ALTFP IP blocks. The general specifications of each block can be seen in the table below:

Hardware Block	Latency (Cycles)	Resources (ALMs)
FP_FUNCTIONS: Addition	2	291.6
FP_FUNCTIONS: Multiplication	2	51.8

2.1 Performance Evaluation

The performance of the updated design was then compared and verified against the old design of the system:

Test Case	Latency – New (ms)	Latency (ms)	Error – New (%)	Error (%)
1	12.8	14.0	0.0000141	0.0936
2	490	525	0.0000511	0.00411
3	60436	68321	0.000911	0.00195

The resources utilized by both systems were also compared:

	ALMs (%)	Memory Bits (%)	PLLs (%)	Total (%)
New System	0.0667	0.0116	0.1667	0.0816
Old System	0.0518	0.0159	0.1667	0.0781

One thing to note is that the FP_FUNCTIONS block has pipeline capabilities. The block can take in inputs every cycle and produce outputs every cycle (after the 2-cycle computation latency). However, due to the nature of custom instructions within the NIOS processor, the pipeline capabilities of the block cannot be utilized without bypassing the CPU through direct memory access. This is because the NIOS processor blocks when a custom instruction is entered, until the done signal is asserted and the result from the instruction is retrieved.

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

A majority of the design's execution time stems from the computation of the cosine function. The current design uses the cosf function defined in the math.h header to perform the operation. This method provides a very accurate result but takes a significantly longer time to compute compared to the other parts of the function. To circumvent this critical path, this section focuses on designing and implementing a CORDIC-based method to compute cosine.

3.1 The CORDIC Algorithm

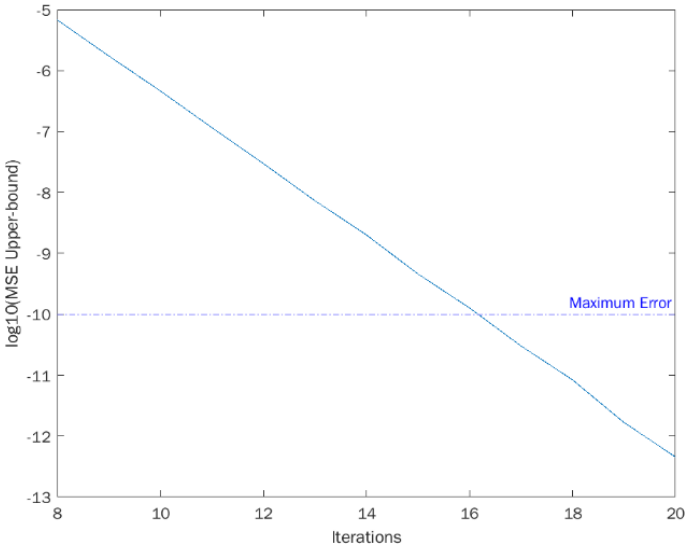
The CORDIC algorithm is a method of computing the cosine function using matrix rotations iteratively. The algorithm takes an input of an angle and attempts to rotate the vector [1,0] by fixed amounts for a certain number of iterations, with its argument decreasing in deviation from the input angle with each iteration. The resultant vector is then multiplied by a gain constant and the x component of the vector corresponds to the cosine of the input angle. The pseudocode for the entire algorithm is shown below:

```
X = {GAIN_CONSTANT}
Y = 0
Z = {INPUT_ANGLE}
FOR i FROM 1 TO {ITERATIONS}
    if z > 0
        X = X - (Y >>> i)
        Y = Y + (X >>> i)
        Z = Z - ATAN(2^(-i))
    else
        X = X - (Y >>> i)
        Y = Y + (X >>> i)
        Z = Z + ATAN(2^(-i))
```

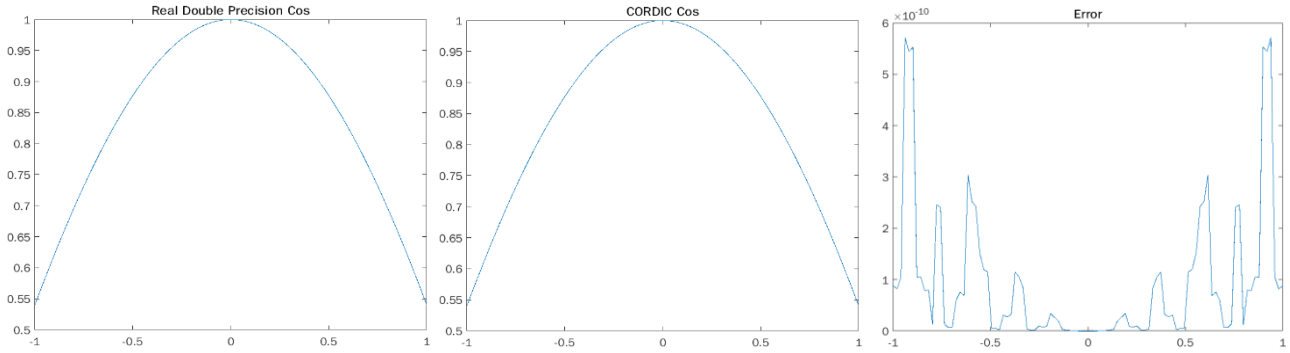
3.2 Estimating Parameters: Monte Carlo Simulation

In this implementation, there is flexibility in the number of iterations which can be utilized by the algorithm, with 32 iterations providing the maximum amount of accuracy. However, the trade-off for this level of accuracy would be a much slower execution time. For the design, a MSE error of 1x10^-10 was required with a confidence interval of 95%. The algorithm was performed in MATLAB using a Monte Carlo simulation to estimate the error with different numbers of iterations. The algorithm was passed a uniform distribution of 100 inputs over the input range of [-1,1] in accordance with the specification of the inputs for the actual function.

Iters	MSE Error	Error upper-bound
8	5.40867215331666e-06	6.83381922147368e-06
9	1.37045106496890e-06	1.74900784972358e-06
10	3.64864438660503e-07	4.66222785010528e-07
11	9.08725716888478e-08	1.16461353963496e-07
12	2.36354277590584e-08	2.99707276500208e-08
13	5.63753143039553e-09	7.34968909200592e-09
14	1.57293241626055e-09	2.01148803986168e-09
15	3.57714359001948e-10	4.60987442281457e-10
16	9.86743299916104e-11	1.27909506281859e-10
17	2.31752455767675e-11	3.03055599365498e-11
18	6.70275120129293e-12	8.42163760950143e-12
19	1.29167076340364e-12	1.67469528869479e-12
20	3.55972196164345e-13	4.54795052312607e-13



After observation of the error across different iterations, in order to achieve a MSE less than $1 \times 10^{-10}$with a confidence interval of 95%, 17 iterations were needed at the very minimum. This was then verified with the actual CORDIC block using a Verilog test-bench to get a mean-squared error of 9.51e-11.



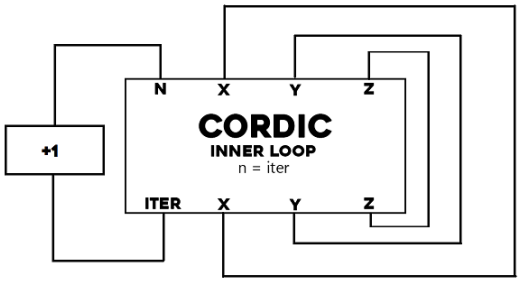
Graphs showing results obtained using MATLAB cos(), our CORDIC block, and the Squared Error between the two implementations

3.3 Latency-Oriented Design

To implement the CORDIC block, two designs were devised. One which was optimized for latency and the other, which was optimized for throughput, the input was a single element of the vector. This subsection focuses on the latency-oriented design. By folding the block, such that the only hardware instance present in the design would be a single iteration of the inner CORDIC loop, the number of resources could be minimized while providing similar latency. The latency-oriented design computes one iteration of the loop per cycle, taking 17 cycles to complete the full CORDIC operation. The throughput of this design is 1/17 as the block produces 1 output every 17 cycles.

The performance of the block was then measured using each of the four test-cases:

Test Case	Total Latency (ms)
1	0.2
2	6.6
3	885.8
4	7.3



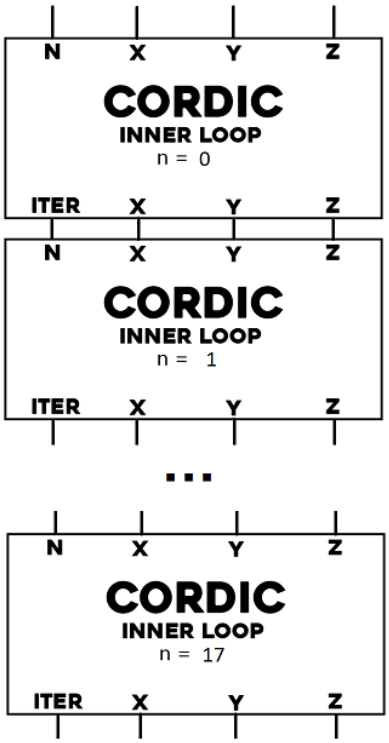
The resources used by this block was found to be:

ALMs (%)	Memory Bits (%)	PLLs (%)	Total (%)
0.0915	0.0116	0.167	0.0899

3.4 Throughput Oriented Design

The throughput-oriented design involved unrolling the CORDIC block, such that each iteration of the inner loop was converted into a separate hardware instance.

This design traded off resources and an increased critical path for a significantly higher throughput. Since the entire operation would occur simultaneously, the design could take inputs every cycle, compute them combinatorically and produce outputs in the next cycle. In this case, both the latency and throughput would be 1.



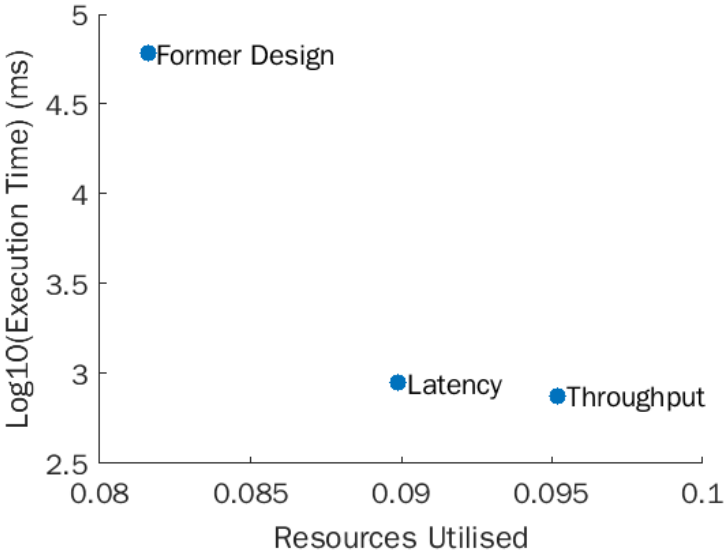
The performance of this block was then measured using all four test-cases:

Test Case	Total Latency (ms)
1	0.2
2	5.8
3	751.0
4	6.5

The resources used by this block was found to be:

ALMs (%)	Memory Bits (%)	PLLs (%)	Total (%)
0.107	0.0116	0.167	0.0952

3.5 Evaluation and comparison of both designs



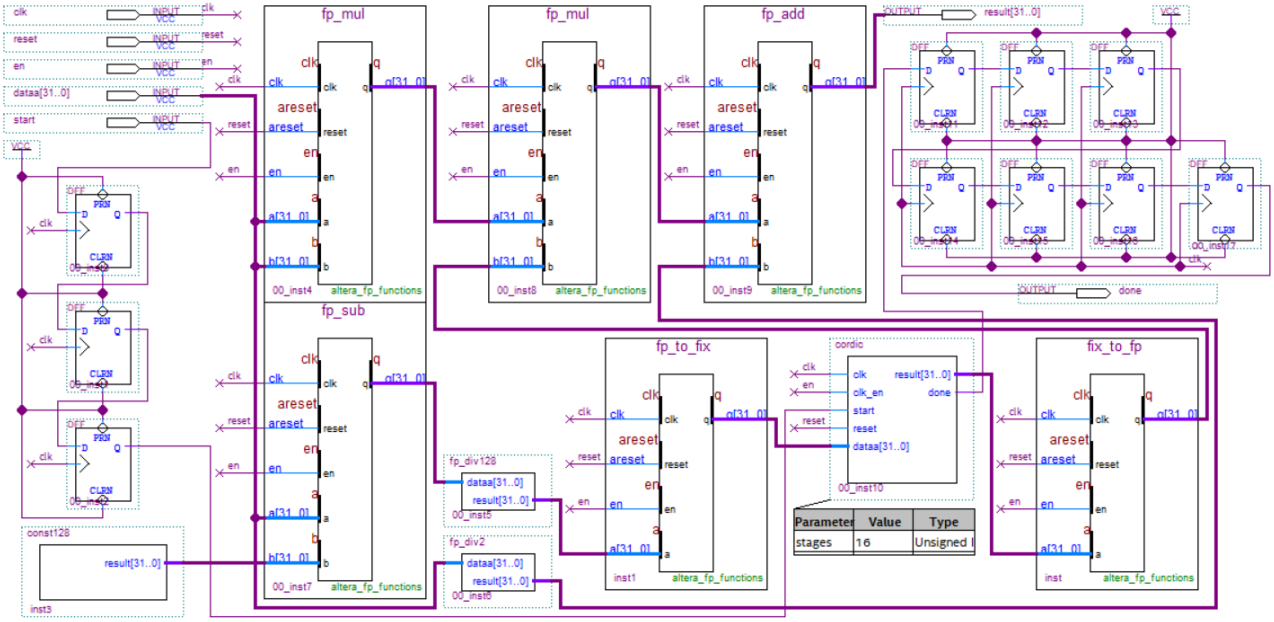
Comparing the performance and resources utilized by both designs, it can be observed that the aims of each design lie on different segments of the performance-resource graph. The latency-oriented design uses a much more efficient design with less resources at the cost of lower throughput and a worse overall performance. Whereas the throughput-oriented design had a higher level of performance achieved and higher performance ceiling, if pipelining were used, but had used more resources and had increased the overall critical path of the design.

In the end, the throughput version was used as the foundation for the final design as the trade-off between performance for resources was favourable for the final design.

One thing to note is, whilst the throughput-orientated design had a worse critical path than the latency-oriented design, the system clock was fixed at 50Hz, thus the latency-oriented design would not be able to exploit this advantage without modifying the board's clock frequency through use of a PLL. This would be counter productive as it would significantly impact the total resources utilized by the design.

3.6 Inner Block Implementation

All the hardware blocks which comprised the inner block of the arithmetic expression were then connected in a wrapper file and were implemented as a single custom instruction. This reduces the overhead in the design caused by the instruction pipeline stages when intermediate values were computed for the expression, as well as the overhead added due to the NIOS core calling *__builtin_custom* for each step.



4 Task 8: Add Dedicated Hardware Block to compute the arithmetic expression

4.1 Overview of final design

For the final design, two versions were made, in order to meet the requirement of using the NIOS/e soft-core in our design, as well as evaluate what the minimum latency for our design could be. In both cases, as many operations were done in hardware as possible, with the NIOS soft-core only being used to coordinate the use of custom hardware and transfer between memory and the custom hardware blocks.

4.2 Optimizations used in the final design

4.2.1 Bit manipulation (division)

Besides the CORDIC block, the most computationally expensive operation in the entire design would be the multiplication/division operation. An optimization opportunity was identified for the design which utilized bit manipulation instead of full multiplication/division. This was implemented in two instances in the expression, one being a division by 2 and the other being a division by 128.

$$y = \frac{x}{2} + x^2 \cos\left(\frac{x - 128}{128}\right)$$

Since the division was by a power of two in both circumstances, the floating-point format was exploited to achieve this optimization. The exponent of the dividend was extracted and subtracted by $\log_2(n)$ where n would be the divisor. This allowed for a combinatorial division which used a single fixed-point operation to achieve. Some of the edge cases which could occur would be if the exponent is less than 1 and 6 respectively, in this case a subtraction would cause the exponent to overflow resulting in an incorrect result.

To circumvent this, the hardware would check for these edge cases and set the number equal to 0. In some cases, the result could be represented using a sub-normal value within the IEEE-754 specification, however this was deemed acceptable as the error would be masked during the conversions between fixed and floating point, which limit the accuracy to 2^{-30}. Since the error in this term in the magnitude of 2^{-128}, it does not cause changes to performance of the algorithm.

4.2.2 Dual CORDIC implementation

To further increase throughput, the Dual CORDIC optimization had exploited the additional ports available for NIOS custom instructions. By making use of the *datab* port, the Dual CORDIC block could take in two inputs, compute the inner expression for each of them and sum the results. This increases throughput by over a factor of 2 as double the operations are occurring per second as well as a reduction in overheads as half the amount of floating-point addition instructions are being called. The updated design was implemented on both the latency-oriented version and the throughput-oriented version, to observe which one would have a better improvement in performance with respect to the additional resources needed.

For the latency-oriented design, the design was evaluated on the given test cases:

Test Case	Total Latency (ms)
1	0.10
2	3.80
3	496.50
4	7.30

The resources used by this block was found to be:

ALMs (%)	Memory Bits (%)	PLLs (%)	Total (%)
0.141	0.011	0.167	0.106

The throughput-oriented design was also evaluated with the same test cases:

Test Case	Total Latency (ms)
1	0.1
2	1.7
3	209.5
4	1.9

The resources used by this block was found to be:

ALMs (%)	Memory Bits (%)	PLLs (%)	Total (%)
0.176	0.011	0.167	0.118

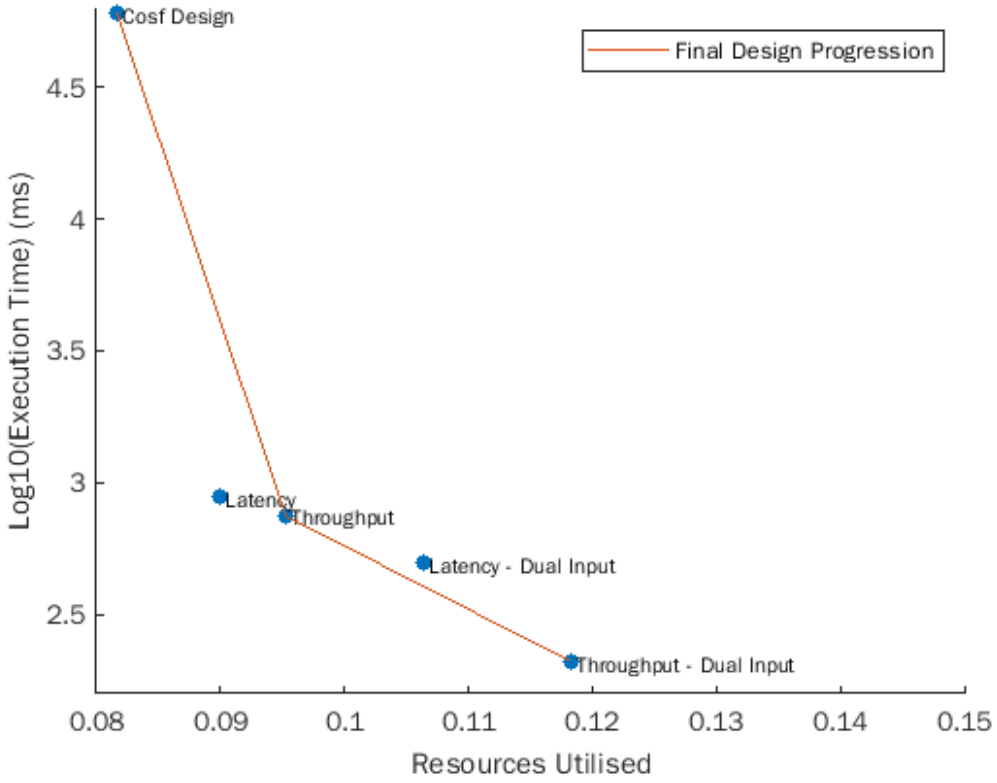
As seen from the results, the latency-oriented design had received around a ~170% boost in performance for a ~17% increase in resources for the full design. Whereas the throughput-oriented design has a significant boost in performance of ~350% for a ~24% increase in resources. The throughput-oriented design was once again chosen as it had a much better trade-off in performance for resources than the latency-oriented design.

4.3 Final Resource-Performance space visualization

The following graphs shows all the design's modifications in the resource-performance space throughout tasks 6-8. To summarize, the overall changes to the system involved:

- Adding floating-point hardware
- Adding a CORDIC block
- Wrapping the entire inner part of the sum function inside a single custom instruction
- Parallelizing the hardware block to allow for a dual input instruction

In conclusion, the final design had a substantial performance boost of ~300x the initial design without the use of floating-point hardware and the CORDIC block, with only a 50% increase in resources.



The performance of both designs was then evaluated with the same test cases as previously:

Performance	Total Execution Time (ms)	
Test Case	NIOS/e	NIOS/f
1	34.2	0.1
2	1250.4	1.7
3	159532.0	209.5
4	Not Tested (*)	1.9

And the resources used calculated for each design:

Resource Utilisation	ALMs (%)	Memory Bits (%)	PLLs (%)	Total (%)
NIOS/e	0.1571	0.0028	0.1667	0.1089
NIOS/f	0.1765	0.0116	0.1667	0.1182