Digital Coursework 3

Ashcharya Kela (CID: 01841948), ashcharya.kela20@imperial.ac.uk
Julio Castillejo Motta (CID: 01857221), julio.castillejo-motta20@imperial.ac.uk
Group 15

June 18, 2023

Contents

1 Task 6 - Adding Hardware Floating Point Units **3**
- 1.1 Implementing the ALTFP Hardware Blocks . . . 3
- 1.2 Design pipeline . . . 3
- 1.3 Performance . . . 3

2 Task 7 - Mapping the inner function **4**
- 2.1 CORDIC algorithm . . . 4
- 2.2 Mathematical model . . . 4
- 2.3 Monte Carlo Simulation . . . 4
  - 2.3.1 Word length optimisation . . 5
  - 2.3.2 Confidence Interval . . . 5
- 2.4 CORDIC implementation . . . 5
  - 2.4.1 Arctan table . . . 5
  - 2.4.2 Scaling factor . . . 6
  - 2.4.3 Latency optimised design . . 6
  - 2.4.4 Throughput optimised design 7
  - 2.4.5 Comparison of latency and throughput optimised designs 7
  - 2.4.6 Floating point cosine . . . 7
- 2.5 Design Pipeline . . . 8
  - 2.5.1 Original design pipeline . . . 8
  - 2.5.2 Improved design pipeline . . 9
  - 2.5.3 Pipeline optimisation . . . 9
  - 2.5.4 Performance vs Resources . . 9

3 Task 8 - Mapping the whole function **10**
- 3.1 Design pipeline and optimisations . . 10
- 3.2 Optimum cache size . . . 10
  - 3.2.1 Comparison with other designs 11
- 3.3 Comparison with previous tasks . . . 11

4 Further-work **12**
- 4.1 DMA . . . 12
- 4.2 Variable cycle Custom instruction . 12

5 Conclusion **12**

6 Appendix **13**

A Task 5 Latency with multipliers **13**

B Arctan Table Error **13**

C CORDIC Simulations in Python **13**
- C.1 CORDIC Class instantiation . . . 13
- C.2 Generating random angles for testing 13
- C.3 Confidence Interval Function . . . 14
- C.4 Monte Carlo Implementation . . . 14
- C.5 Word Length Optimisation . . . 14

D CORDIC **15**

E CORDIC Algorithm **15**
- E.1 Rotation Mode . . . 15
- E.2 Vector Mode . . . 16
- E.3 Latency Optimised design pipeline . 16

The aim of this report is to investigate and conclude the impact of implementing custom instructions to accelerate the evaluate of the target function.

$$f_2(x) = \Sigma(\frac{1}{2}x + x^2 cos(\frac{x-128}{128})) \quad (1)$$

1 Task 6 - Adding Hardware Floating Point Units

In this task the ALTFP_MULT and ALTFP_ADD_SUB IP Cores were added to the design to provide support for floating point multiplication, floating point addition and floating point subtraction. This support relieves the system of emulating these floating point operations through software and instead compute them through hardware, enabling latency for the computation of mathematical equations to decrease. The IEEE-754 floating point standard is the format used by the IP cores and the C programming language that programs the NIOS II processor.

1.1 Implementing the ALTFP Hardware Blocks

The hardware units added were an ADD, SUB and MULT block. The multi-cycle custom instruction was utilised as more than one clock cycle (and thus, registers) were needed to calculate the function f(x). Fixed number of clock cycles was used for each of the respective blocks. The multiplier utilised 12 clock cycles whilst the adder utilised 15.

1.2 Design pipeline

To evaluate the target function (1) efficiently using the ALTFP hardware, the following design pipeline can be implemented as seen in Figure 1.

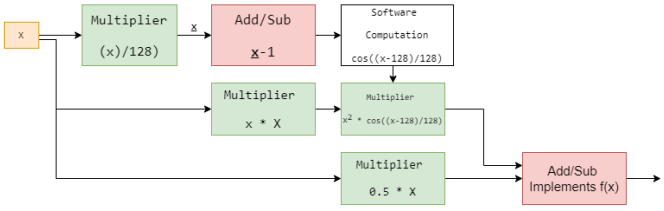


Figure 1: Pipeline using ALTFP Hardware

The multiplier hardware blocks were arranged in such a way such that the same multiplier block could carry out multiplication for different sections of the target function, thereby reducing the resources used. Hence, this design pipeline version only requires 2 multiplier blocks and one add/sub block to compute $f(x)$. Moreover, it requires a minimum of 4 clock cycles where the frequency of the clock cycle is dependent on the software speed of computing cosine.

1.3 Performance

The table below shows how the latency varied for different test-cases.

Block Type	Test case 1 (ms)	Test case 2 (ms)	Test case 3 (ms)
Multiplier + Add	3	110	13,840
Multiplier + Sub	3	115	14,440
Multiplier + Add/Sub	3	107	13,356

Table 1: Latency vs Floating Point Hardware blocks

The latency decreased by a factor of 10 for all test cases compared to Task 5 latency which can be seen in Appendix A. Moreover, in comparison to Task 4 latency where the computation of test case 3 took approximately 180 seconds, the latency has dropped to 13 seconds, a 93% decrease.

The block with all of the multiplier, adder and subtract implemented as hardware had the smallest latency with the sacrifice of increased resource utilisation which is summarized in the table below.

Moreover, the latency is one more cycle than needed. This is because the clock starts of as high and in this time, the input data has not been constant before the positive edge of the clock for at least the duration of the setup time. To avoid violating setup time, the clock enable during the multi-cycle custom instruction goes high at the negative edge of the first clock cycle. Hence, since data is read at positive edge of the clock, we have a delay of one

Block Type	Logic Usage	Total block memory bits	Total Registers
Multiplier + Adder	2282	170,995	3798
Multiplier + Sub	1839	170,648	3172
Multiplier + Adder/-Sub	2499	171,036	4170

Table 2: Hardware Resources Task 6

clock cycle and the output data is read one clock-cycle later.

2 Task 7 - Mapping the inner function

2.1 CORDIC algorithm

The CORDIC algorithm is a hardware efficient solution to computing hyperbolic and trigonometric function values. It achieves this through utilising a linear convergence algorithm and making approximations geared towards minimising resource usage. For our purposes, the CORDIC algorithm was used to implement the cosine function in order to evaluate the function completely in hardware:

CORDIC is based on two fundamental concepts:

1. Rotating a point (x , y) is equivalent to rotating the point first by $+45°$ and then by $-15°$.

2. Choosing angles (θ_i) to rotate by such that it satisfies $tan(\theta_i) = 2^{-i}$ as it allows multiplication by $tan(\theta_i)$ in the equation to simply be binary shifts. The reason $tan(\theta_i) = 2^i$ is not chosen as an angle selection is because increasing powers of two quickly converge to π and thus, does not provide enough small angles to accurately evaluate cosine.

There are two main ways to implement the CORDIC algorithm to compute cosine values: vector mode and rotation mode. Refer to Appendix B for further explanation on these modes.

2.2 Mathematical model

By referring to Appendix E and expanding the equation 13, the following equations can be derived:

$$x[i+1] = x[i] - \sigma_i 2^{-i} y[i] \tag{2}$$

$$y[i+1] = y[i] - \sigma_i 2^{-i} x[i] \tag{3}$$

The σ_i is a variable that defines whether the rotation is clockwise or anti-clockwise. To determine this, we need to know whether the desired rotation is larger or smaller than the current achieved rotation. Introducing a new variable z and initially setting $z = \theta_{desired}$, the difference between our desired and achieved rotation is given by:

$$z[i+1] = z[i] - \sigma_i \theta_i \tag{4}$$

If $z[i+1] > 0$, then the desired rotation is greater than the previous achieved rotation and hence, we need to rotate clockwise, making $\sigma = +1$. This is a form of negative feedback accumulating all previous angle of rotations and comparing it with the desired angle of rotation.

2.3 Monte Carlo Simulation

Monte Carlo method is a mathematical technique that assigns random values to a variable in order to determine a result that may be deterministic. In our case, $cos(\theta t)$ is a deterministic signal whose value is always known for all given times t.

The implementation of the Monte Carlo Simulation can be seen in Appendix (C.4). The simulation generated an array of 1,000,000 different angles through pythons **random library**.

The cosine of all the different angles in the angle array was computed for different iterations. For each angle, the error between the CORDIC algorithm with respect to the result from math.h library (utilising double precision) was computed and squared. The mean-squared error was calculated using the formula below:

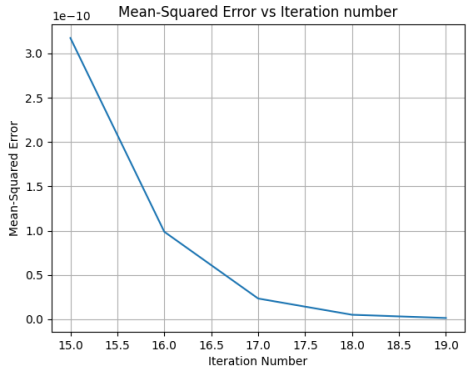


Figure 2: Mean Squared Error vs Iterations

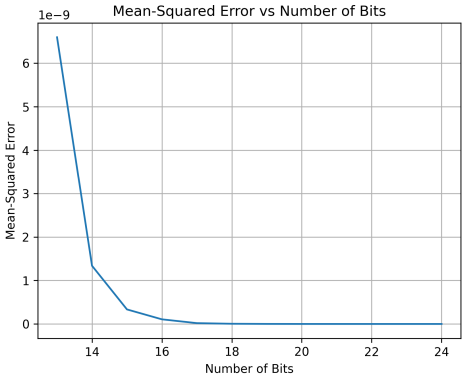


Figure 3: Word Length (Bits) vs MSE

$$MSE = \frac{1}{N}\sum_{i=1}^{i=N}(Y_i - \hat{Y}_i)^2 \tag{5}$$

At the end of the different iterations of computing $cos(\theta)$, an array of mean-squared errors for different iterations was returned as shown by Figure 2 below.

Figure 2 shows that having a minimum iteration number of 16 produced a mean-square error of less than 10^{-9}. The number of iterations are directly linked with the bit-accuracy achieved.

Increasing the number of iterations will improve the accuracy of cosine value however, if the cosine value can be achieved in fewer iterations, by continuing to do further iterations introduces a very small error as it will no longer be **exactly** the cosine value we are looking for. Moreover, doing more iterations than necessary also increases latency and thus, 16 iterations was chosen as it met the mean-squared error specification.

2.3.1 Word length optimisation

The word length optimisation is to determine the optimum bits required in order to maintain high accuracy whilst simultaneously decreasing latency. In order to do this, two functions were created: one to convert floating-point to fixed-point and another to convert fixed-point to floating-point. These can be found in Appendix C.5. Different word length in bits were iterated over with different number of iterations. The mean-squared error was analysed and it was found that a minimum of 20 bits (with 18 fractional bits) were required to achieve a mean-squared error of 10^{-10} with 16 iterations. To help increase the accuracy a bit further whilst still using 20 bits, iterations was increased to 18.

2.3.2 Confidence Interval

The design was implemented to achieve 95% of confidence interval, suggesting that 95% of the time, the mean value of mean-squared error is within the confidence interval range. The following equation computed the confidence interval, where $\bar{x}$ = mean of MSE, z = confidence level, s = standard deviationm and n = sample size:

$$CI = \bar{x} \pm z\frac{s}{\sqrt{n}} \tag{6}$$

For a sample size of 100 different MSE, the confidence interval was in the range of $5.282682767264608E^{-12}$ to $5.3214671992724506E^{-12}$, which is below the specified requirement of the 10^{-10} MSE. For the code, please refer to Appendix C.3.

2.4 CORDIC implementation

2.4.1 Arctan table

In this implementation of the CORDIC algorithm 18 iterations were used for one angle computation, as aforementioned in the Monte Carlo simulation. A consequence of this choice is that the Arctan table must be composed of 18 values in order to update the value of the Z accumulator on each iteration.

Furthermore, the second consequence of the Monte Carlo simulation is to restrict the word length to 20 bits, this must be applied to the X and Y accumulators but does not have to be applied to the Z accumulator. The reason being that the Z accumulator is independent of the X and Y accumulators. As a result of this exemption the Arctan table values do not have to be restricted to 20 bits. Going in this direction will lead to a higher accuracy result for the Z accumulator.

On the other hand when evaluating the error of restricting the bit length on the Arctan table to 20 bits, it was found that the error incurred by this format was in the range $-8.29E^{-7} < Error < 4.34E^{-5}$ and had a mean squared error of $6.23E^{-10}$, this can be seen in Appendix 18. When adjusting the Monte Carlo simulation with this new format for the Z accumulator the results were the same, as a result of this investigation the Arctan table values were also restricted to 20 bits. The benefit of doing this, other than uniform word length, is that the same hardware that is used for the X and Y accumulators can be applied to the Z accumulator, therefore resources are saved.

Moreover, as shown in the CORDIC Mathematical model the X, Y and Z accumulators contain powers of two, this is a outcome of the choice of angles used to implement the CORDIC algorithm, the reason for this choice is because powers of two can be easily implemented in hardware through shifts thus saving resources, however any table of angles that tends towards 0 can be used.

2.4.2 Scaling factor

The scaling factor is a constant and is only dependent on the number of iterations. As mentioned in the 2.2 Mathematical model , the scaling factor needs to be accounted for at the end of the calculation through a multiplication. In order to avoid doing a multiplication we can assign the initial value of the X accumulator to be the scaling factor. The scaling factor was set as a constant using the 20 bit fixed point format shown in Word length section, its value was found to be 0.6072529 and any higher accuracy would not be detected by our 20 bit fixed point format.

2.4.3 Latency optimised design

Due to the design of the CORDIC algorithm the hardware implementation can be achieved through control logic, shifts and adders. In this design when executing the CORDIC algorithm, fixed point numbers are used. A consequence of this choice is the smaller FPGA resource footprint. This is because if floating point numbers were used then floating point adders would be needed which are more expensive than using the Verilog full adders, on top of this the floating point adders would add 7 clock cycles of latency each time an addition is needed while the Verilog full adders are combinatorial. On the other hand, the floating point shifts can be implemented combinatorially through subtraction of the exponent part of the floating point IEEE754 standard.

A downside of using the fixed point format is the need of the ALTFP_CONVERT IP at the input and output of the CORDIC block, this is further explored in Floating point cosine. However, even when accounting for these extra conversion units, the fixed point design would still save more resources and clock cycle latency than using floating point adders thus justifying the choice of a fixed point format.

The CORDIC latency optimised design was constructed to have 3 states START, PROCESS1 and PROCESS2. These states are controlled by three signals: counter (i) the enable signal (clk_en) and the reset signal (reset).

In the START state the X accumulator is loaded with the scaler factor (as stated in the Scaling factor section), the Y accumulator is loaded with 0 and the Z accumulator is loaded with the input angle. This marks the start of the algorithm, and the state is advanced to PROCESS1. In PROCESS1 the CORDIC equations are implemented, the block diagram for the equation $x = x_{next} - \sigma_i 2^{-i} y_{next}$ is in Figure 5. The other equations are implemented similarly.

Figure 4: CORDIC state machine

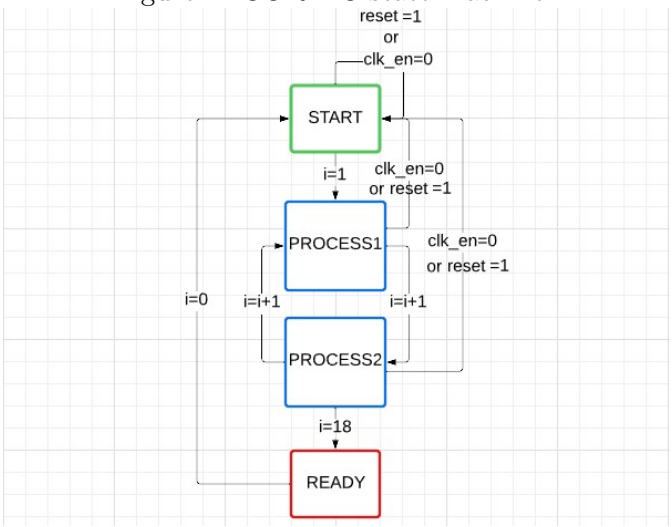
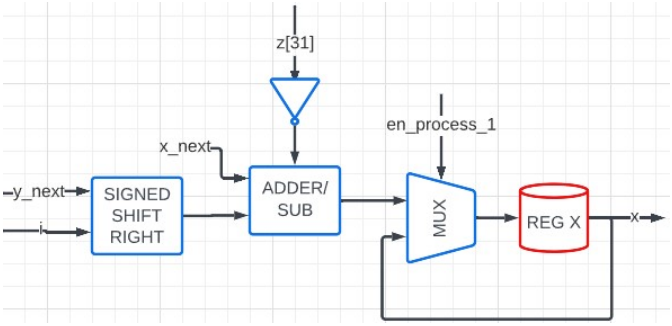


Figure 5: PROCESS1 design



Once the next clock edge arrives the counter is increased by 1 and the state is advanced to PROCESS2. The implementation of PROCESS2 is the same as in PROCESS1 but updates the x_next, y_{next} and z_{next} registers. When counter reaches 18 all the calculations have been performed and the IP waits for one clock cycle to keep the output stable ready to be read. The overview of the system is shown in Figure 19.

It can be observed how the PROCESS1 and PROCESS2 blocks are interfaced. Moreover the design was implemented using 20 bits for the word length since this was the output of the Monte Carlo simulation which resulted in an MSE $10E^{-10}$.

2.4.4 Throughput optimised design

To optimise the throughput, the pipeline design as shown by Figure 6 below was implemented.

With this optimisation implemented, for 18 iterations, the time till the first cosine output appeared was 19 clock cycles (equivalent to 19×20 ns $= 380$ ns) as each clock cycle had 20 ns period. However, after this initial delay, the block can produce cosine outputs for different angles every 20ns (i.e. every clock cycle). Moreover, having a latency optimised CORDIC block for each iteration and pipe-lining them in series further reduces the latency whilst increasing the throughput.

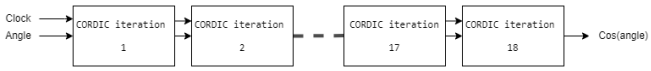


Figure 6: Throughput optimisation pipeline

2.4.5 Comparison of latency and throughput optimised designs

The latency optimised design achieved a latency of 0.971s, similarly the throughput optimised design achieved a latency of 0.901s, on the other hand the latency optimised design 9.34% while the throughput optimised design used 15.3%.

The throughput optimised design achieves a 7% improvement on latency at the cost of using 63% more resources. This brings forth the trade-off between latency and resources used.

2.4.6 Floating point cosine

In order to make the CORDIC block able to accept floating point numbers and output floating point numbers the ALTFP_CONVERT blocks were used. These blocks add 6 clock cycles of latency each.

Figure 7: Floating point cosine

Therefore, the floating point cosine implementation takes 32 clock cycles to execute. A different alternative to using the ALTFP_CONVERT blocks was explored by making usage of the fact that the input range of the cosine would be [-1, 1]. Making this observation allowed the construction of a combinatorial conversion from floating point to fixed point

and vice versa thus reducing the latency of the floating point cosine by 12 clock cycles.

To convert to fixed point from floating point number, the mantissa and exponent are extracted from the input floating point number through bit selection, note that the sign of the input is ignored due to the even symmetry of the cosine function therefore the input range is mapped to [0, 1].

Then the mantissa is concatenated with a 1'b1 at the MSB position, this acts as the summation with the integer 1 from the IEEE754 floating point standard. Then the shift amount is calculated from the exponent by $shamt = 127 - exponent$ which is then used to shift the mantissa. Finally for our system we select 20 bits from the shifted mantissa. All of these operations can be implemented combinatorially.

To convert to fixed point the MSB of the input is found through a MUX and the position of this MSB is the shift amount then the formula $exponent = 127 - shamt$ is applied to find the exponent. The mantissa is obtained by ignoring the MSB (subtraction of 1) and then shifting by the shamt. The sign bit is always 0 because the output range of the cosine function is always positive for the input range [-1, 1]. Finally the 1'b0, the exponent and the mantissa are concatenated to form the floating point number.

Using this combinatorial conversion method saves resources by 0.035% and saves 12 clock cycles.

2.5 Design Pipeline

In order to map the inner part of the target function

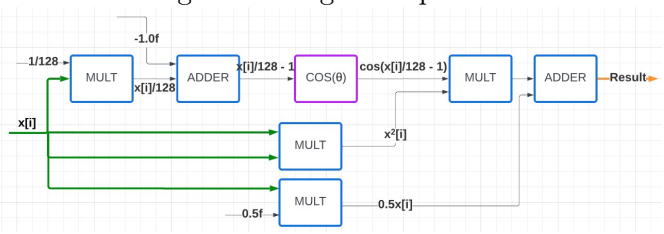
$$inner(x) = 0.5x[i] + x^2[i]cos(\frac{x[i] - 128}{128}) \quad (7)$$

the operations $x[i]/128, x^2[i], 0.5x[i]$ were identified to be candidates for parallelisation therefore there is potential for hiding latency. Furthermore the the division by 128 was converted to a multiplication by 0.0078125 since multiplication is less expensive than division.

2.5.1 Original design pipeline

In this design pipeline doing parallelisation increases the resource utilisation by using a floating point unit for each operation, as shown in the Figure below.

Figure 8: Original Pipeline



In this pipeline the parallelised operations were chosen to all happen at the start of the pipeline, but they do not have to necessarily happen at this time (this is further explored in the Improved design pipeline section).

As can be seen in the figure above there is a big sequence of operations in series, these operations cannot be parallelised because they depend on the output of the previous operation.

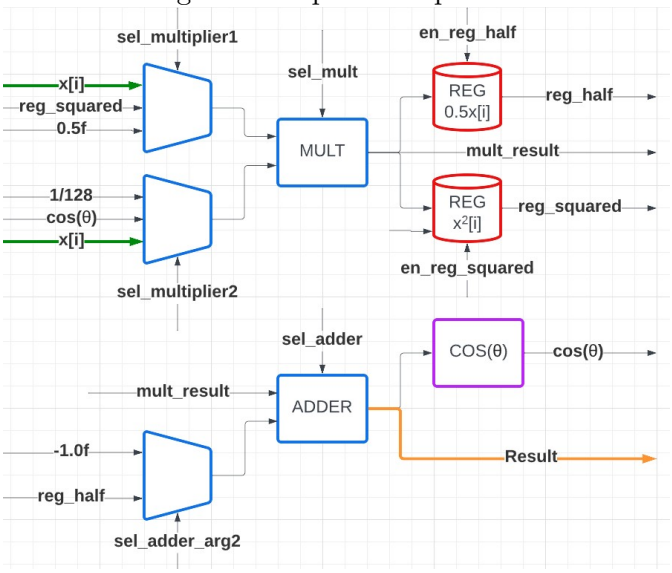
When analysing the pipeline latency it was found that the design would need 56 clock cyles to output a result. this is because the ALTFP_MULT IP takes 5 clock cyles, the ALTFP_ADD_SUB takes 7 clock cycles and the floating point cosine block takes 32 clock cycles. Overall this pipeline uses 6 floating point units and 1 cosine block with a latency of 56 clock cycles.

One way to improve this pipeline would be to fit registers in between each stage to pipeline the design, however this would require careful usage of the timing since the ALTFP_MULT, ALTFP_ADD_SUB and our CORDIC design need different numbers of clock cycles to complete one operation. For this reason the number of clock cycles for each call of the pipeline would need to be 32 cycles in order to give the COSINE block enough clock cycles to finish. Doing this would increase the throughput of the design by 43% and would make full use of the increased number of resources.

2.5.2 Improved design pipeline

The original pipeline can also be improved by reducing the resource utilisation. This is shown in the figure below.

Figure 9: Improved Pipeline



In this pipeline only 1 floating point multiplier and 1 floating point adder are needed.

This is achieved by initially using the multiplier and adder to compute $\frac{x}{128} - 1$ the result is passed onto the COSINE block which would take 32 clock cycles to execute. At this point in time the multiplier can be used to calculate $0.5x[i]$ which takes 5 clock cycles then store this value in a register and use a further 5 clock cycles to compute $x^2[i]$ and store the value into a register, due to the 32 clock cycle latency of the COSINE block this is possible. Once the cosine block is finished we can calculate the multiplication of $x^2[i]cos(\frac{x}{128} - 1)$ and then use the floating point adder output the result of the inner function.

Using this architecture reduces the resource usage from 6 floating point units down to 2 while having the same latency as the original pipeline. On the other hand this design uses control logic which increases the complexity while the original pipeline uses no control logic.

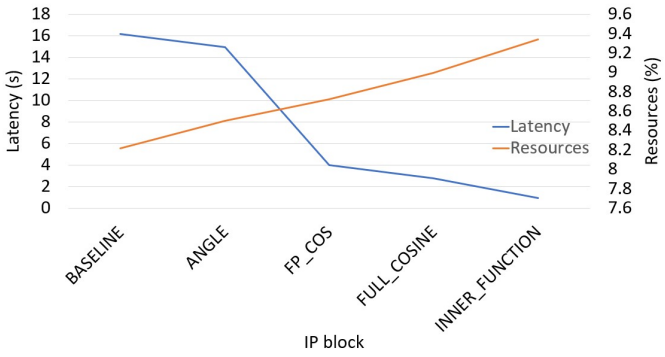
2.5.3 Pipeline optimisation

To further optimise the pipeline the multiplication by 2^{-7} or the division by 128 can be implemented combinatorially by subtracting 7 or $0b111$ from the exponent this can be done combinatorially because the exponent uses integer operations. Also the subtraction of -1 can be implemented through combinatorial logic by applying this operation after the floating point number has been converted to fixed point. Doing these optimisations saves the pipeline 12 clock cycles which brings the total down to 44 clock cycles without losing any accuracy and using less resources, if we applied the combinatorial floating point and fixed point conversions the latency of our inner function implementation is brought down to 32 clock cycles.

2.5.4 Performance vs Resources

In order to design the pipeline for Task 7 a hierarchical approach was taken which yielded the opportunity to test the performance of the blocks as they were being added to the pipeline.

Figure 10: Latency Optimised design performance



In the Figure 10 the Baseline is the design which contains no custom instructions thus it uses the least FPGA resources but has the biggest latency. The ANGLE block computes the inner part of the cosine $\frac{x-128}{128}$ when this was set as a custom instruction the design obtained a speed up of 1.12 but had to give up 0.2% FPGA resources. The FP_COS block uses the latency optimised CORDIC implementation and uses the ALTFP_CONVERT blocks to allow floating point input and outputs as discussed in Float-

ing point cosine section. This achieves the largest speedup of 3.63 (compared to previous block) while using 0.4% more FPGA resources than the baseline. The FULL_COSINE block interfaces the ANGLE and FP_COS into one unit and achieves a speedup of 1.90 while using 0.8% more FPGA resources than the baseline. The INNER_FUNCTION block calculates the inner function and decreases the latency down to 0.952 seconds achieving a speedup of 2.62 (compared to previous block) while consuming 1.2% more FPGA resources than baseline.

Overall, the graph in Figure 10 shows how the there is a positive trend in the usage of FPGA resources which is complemented by a negative trend in the realised latency of the system. On the other hand, the speedup that is achieved from continuing to use more resources is dramatically decreased after the FP_COS block, one reason for this is that computation of the cosine block is the function that takes the longest to compute. It can also be observed from the graph that the FPGA resources have a linear trend.

Finally, the error increases from 0.000767% to 0.000911% across all introduced IP blocks. This brings about the argument of the trade-off between latency, resources and error. In our system we are focused on decreasing the latency as much as possible hence the increment in error was not a problem.

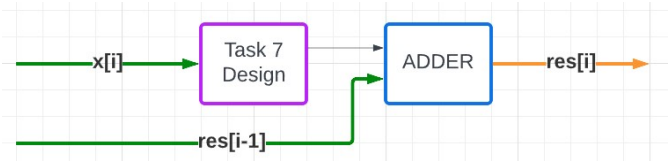
3 Task 8 - Mapping the whole function

3.1 Design pipeline and optimisations

To finalise the design of the full target function an adder is needed to perform the summation of the inner function over all the elements of the input vector.

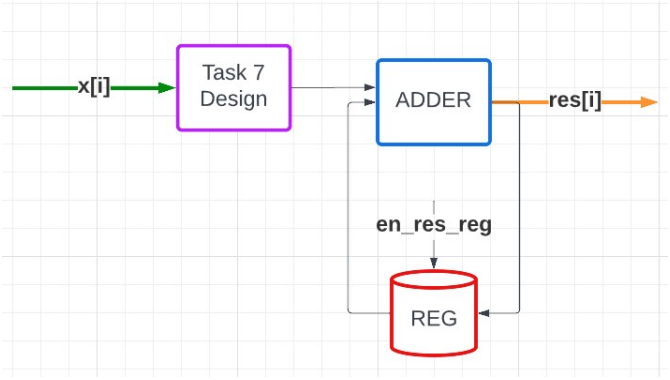
The first design that was attempted was to have two inputs to the Custom Instruction, the first input would be the current $x[i]$ and the second input would be the current sum of the target function. In Figure 11, the design uses the minimum possible amount of resources but it trades this for an average extra latency of 8ms compared to the next design,

Figure 11: Task 8 design pipeline 1



the reason for this extra latency is the loading and storing of the variable sum in software. The second design avoids this problem by using a register to store the sum. This register provides fast access and only one input is needed to the Custom Instruction.

Figure 12: Task 8 design pipeline 2



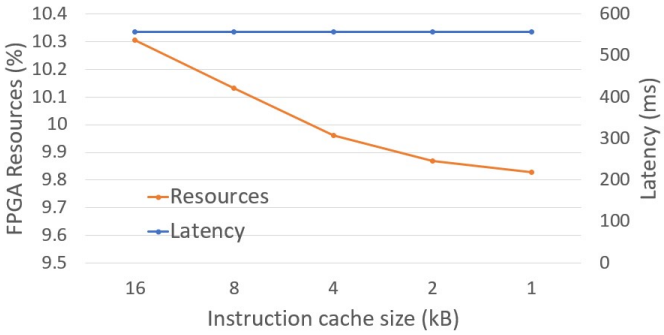
This design trades the 8ms of latency for the usage of 1 register. The block diagram can be seen in Figure 12.

3.2 Optimum cache size

By mapping the target function to hardware, the software functions used to execute the target function have been reduced to a for loop updating the value of a variable through a Custom Instruction, this points to the fact the Instruction cache size can be decreased.

As is shown in the Figure 13 the latency remains the same while the Instruction cache size is decreased. The optimum cache size in this investigation is 1kB of instruction cache. Decreasing the cache size to 1kB from 16 kB saves 0.5% FPGA resources.

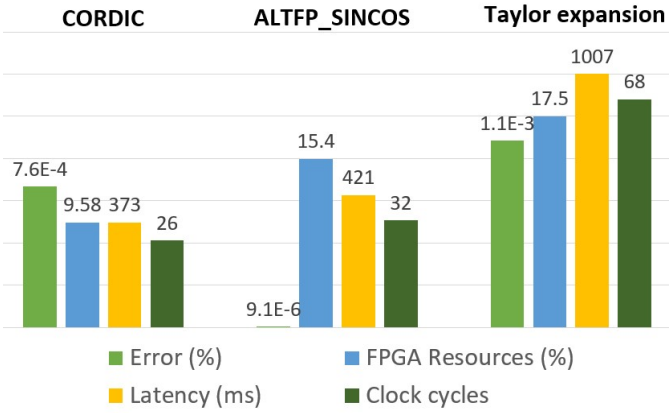
Figure 13: Optimum instruction cache size



3.2.1 Comparison with other designs

Our implementation of the CORDIC design was evaluated against the ALTFP_SINCOS IP core and a four term hardware implementation of the Taylor expansion.

Figure 14: Comparison with other cosine implementations

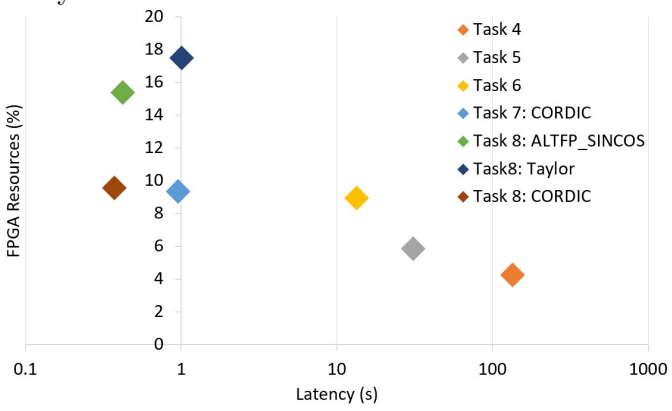


As can be shown in the Figure 14 the CORDIC design performs the best in every category apart from the Error where the ALTFP_SINCOS achieves an error percentage which is 2 orders of magnitude smaller, however this accuracy is traded for the need of 6% more FPGA resources and 12% higher latency. On the other hand, the Taylor expansion design performs the worst in every category. This is due to the fact that most of the operations in the Taylor expansion cannot be parallelised and therefore they must be put in series which increases the latency by 161% when compared to the CORDIC block. Also, in order to increase the accuracy more terms must be added to the equation which further increases the latency. The FPGA resource usage is also high since latency was prioritised.

3.3 Comparison with previous tasks

As we moved from Task 4 to Task 8 the FPGA resource usage has increased but the latency has decreased. In the logarithmic graph in figure 15 it can be seen that the FPGA resources increase linearly which equates to exponential growth outside of the logarithmic axis. The exponential usage of these resources allowed the latency to decrease exponentially thus justifying the increase of resources used.

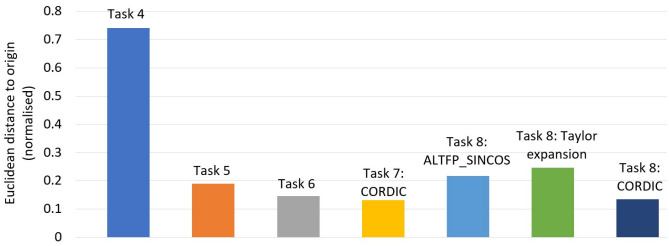
Figure 15: Task comparison FPGA resources vs Latency



The optimal region that the ideal design would tend towards the minimum use of resources while having the minimum possible latency and this region can be characterised as being as closed to the origin as possible. When comparing Task 8: CORDIC with Task 4 it can be observed that Task 8: CORDIC has gotten closer to the origin than Task 4 thus showing the overall improvement of the system. To quantify this improvement the Euclidean distance from the point to the origin can be calculated.

The figure 16 shows the normalised Euclidean distance which was calculated by normalising the FPGA resources and latency sets to a range of [0, 1] and then calculating the distance using $d = \sqrt{x^2 + y^2}$. One interesting observation from this evaluation is that the Task 7: CORDIC design is closer to the origin than Task 8: CORDIC design

Figure 16: Euclidean distance to origin



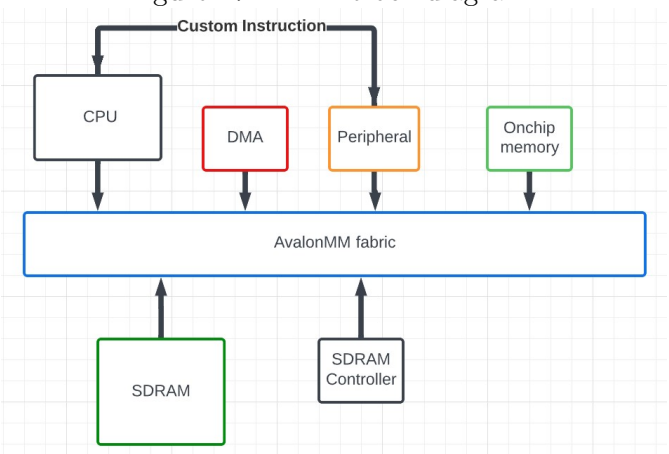
this means that if FPGA and latency were considered to be the same value then Task 7: CORDIC would be a better choice. In our system latency is prioritised therefore the fact that Task 7: CORDIC is closer does not matter since it has a lower latency.

4 Further-work

4.1 DMA

One optimisation that has been left on the table due to time constraints is the enabling of DMA to the target function peripheral.

Figure 17: DMA block diagram



This would have been implemented by attaching an AVALONMM_MASTER interface between target function peripheral and the DMA, this would enable the peripheral to enable or disable the DMA. The DMA would have an AVALONMM_MASTER interface to the SDRAM to be able to retrieve data and burst it onto the target memory unit. The DMA would need another AVALONMM_MASTER interface with the target memory unit. In this case the target memory unit can be On-Chip memory IP which would provide fast memory accesses once the data has been loaded by the DMA. Using DMA would enable the effective use of a pipelined design. This would also mean that the software would need to provide the starting address of the array and the Custom Instruction would output the solution therefore the latency from the for loop in software would disappear.

4.2 Variable cycle Custom instruction

One problem with using a fixed multi-cycle custom instruction is that the CORDIC algorithm loses accuracy and latency. This is because once the Z accumulator reaches 0 the algorithm should stop, this would improve the latency and improve the accuracy of the result. However it would increase the complexity of the design by adding more control logic. The Variable cycle Custom Instruction would allow this feature to be enabled. Once the Z accumulator is 0 the done signal would be asserted and the state of the CORDIC block would get reset.

5 Conclusion

In conclusion, this project introduced customised peripherals to the system to decrease the latency of the computation of the target function, however, we have also observed that doing this increases the FPGA resources usage by 107% when comparing the final design in Task 8 and Task 4. In order to decrease the amount of extra FPGA resources hardware optimised algorithms such as the CORDIC algorithm were implemented but they came with the trade-off of increasing the error of the computations of the target function, this trade-off was analysed in the Monte Carlo simulation and the final design followed the results of this simulation in order to abide to the $10E^{-10}$ MSE target of the cosine computations. One final optimisation that was applied to the system was setting enabling compiler flags for Optimisation level and removing the Debug flag this allowed the system to achieve a latency of 395 ms for Testcase 3 with an error of $3.68E^{-4}\%$.

6 Appendix

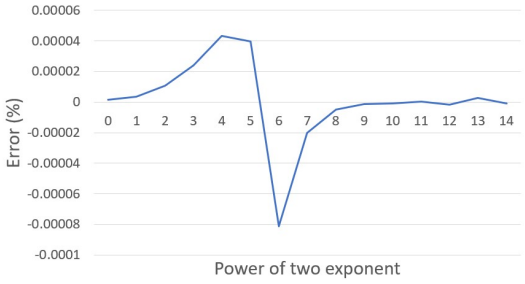
A Task 5 Latency with multipliers

Instruction Cache (kB)	Data Cache (kB)	Test case 1 (s)	Test case 2 (s)	Test case 3 (s)	FPGA resources
2	2	0.0229	0.8786	109.167	0.037066
8	2	0.0102	0.3603	48.215	0.041473
32	2	0.0072	0.2457	30.992	0.058686
64	2	0.0071	0.2458	30.992	0.081804
2	8	0.0210	0.8185	102.859	0.041416
2	32	0.0207	0.8049	101.322	0.058213
2	64	0.0207	0.8047	101.310	0.080734

Table 3: SDRAM latency versus Cache sizes for new function (2)

B Arctan Table Error

Figure 18: Arctan table Error



C CORDIC Simulations in Python

C.1 CORDIC Class instantiation

```
import math
import matplotlib.pyplot as plt
import random
import numpy as np

class CORDIC():
    def __init__(self, num_iterations):
        self.table_angles = [math.atan(2 ** -i) * 180 / math.pi for i in range(num_iterations)]
        self.num_iterations = num_iterations
        self.scaler_factor = 0.607252935

    def _set_sigma(self, z):
        if z < 0:
            return -1
        return +1

    def _compute(self, target_angle):
        x_current = self.scaler_factor
        x_next = 0
        y_current = 0
        y_next = 0
        z = target_angle
        sigma = 0

        for i in range(self.num_iterations):

            sigma = self._set_sigma(z)
            x_next = x_current - sigma * (2 ** -i) * y_current
            y_next = y_current + sigma * (2 ** -i) * x_current
            z = z - sigma * self.table_angles[i]
            x_current = x_next
            y_current = y_next

        return x_next, y_next

    def cosine(self, target_angle):
        return self._compute(target_angle)[0]

    def sine(self, target_angle):
        return self._compute(target_angle)[1]
```

C.2 Generating random angles for testing

```
    def angles_array_generation(number_angles):
        angle_radians = []
        for i in range(number_angles):
            angle_radians.append(180*random.uniform(-1, 1)/math.pi)
            print(f"angle degrees test: {angle_radians[i]}")
        return angle_radians
```

C.3 Confidence Interval Function

```
def confidence_interval():
    number_of_angles = 1000
    iterate_start = 16
    iterate_stop = 17
    iterate_step = 1

    MSE_array = []

    # Using 100 different angle_array sets, getting MSE for each angle_array and then use MSE_array to find mean
    for i in range(100):
        # Output of monte_carlo is MSE_array depending on iterations
        MSE_set1 = monte_carlo(number_of_angles, iterate_start, iterate_stop, iterate_step)
        MSE_array.append(MSE_set1)

    sample_size = len(MSE_array)
    sample_mean = np.mean(MSE_array)
    standard_deviation = np.std(MSE_array, dtype=np.float32)
    confidence_lvl_value = 0.95 # Should this be 0.95 or 95


    # Range of interval with 95% confidence.
    confidence_interval_val1 = sample_mean + (confidence_lvl_value * standard_deviation)/(math.sqrt(sample_size))
    confidence_interval_val2 = sample_mean - (confidence_lvl_value * standard_deviation)/(math.sqrt(sample_size))

    return confidence_interval_val1, confidence_interval_val2
```

C.4 Monte Carlo Implementation

```
def monte_carlo(number_of_angles, iterate_start, iterate_stop, iterate_step):
    MSE_array = []
    angles_array = angles_array_generation(number_of_angles)

    # Implement different iterations
    for iteration in range(iterate_start, iterate_stop, iterate_step):
        # Initialise number of iterations by calling CORDIC class
        test = CORDIC(iteration)
        summation = 0
        print(f"iterations value: {iterations}")

        for angle in range(0, number_of_angles-1, 1):
            #Output value of cos(angle) where angle is in degrees
            CORDIC_cosine_val = test.cosine(angles_array[angle])
            #Math.h value of cos(angle)
            math_cosine_val = math.cos(angles_array[angle] * math.pi / 180)

            #error squared
            error = CORDIC_cosine_val - math_cosine_val
            error_squared = error ** 2
            summation = summation + error_squared

        MSE = summation/number_of_angles
        MSE_array.append(MSE)
    return MSE_array
```

C.5 Word Length Optimisation

```
def floating_to_fixedpoint(floatingpoint, n_word, n_frac):

    fixedpoint = Fxp(floatingpoint, signed=True, n_word=n_word, n_frac=n_frac)
    fixedpoint_binary = fixedpoint.bin()
    print(f" Fixed point: {fixedpoint} ")
    print(f" Fixe_point binary: {fixedpoint_binary}")
    return fixedpoint_binary

def fixedpoint_to_float(fixedpoint,n_frac):
    c = abs(fixedpoint)
    sign = 1
    if fixedpoint < 0:
        # convert back from two's complement
        c = fixedpoint - 1
        c = ~c
        sign = -1
    f = (1.0 * c) / (2 ** n_frac)
    f = f * sign
    return f
```

D CORDIC

E CORDIC Algorithm

The CORDIC algorithm is a digital solution to computing trigonometric function values (cosx, sinx and tanx) through utilisng a linear convergence algorithm. The number of iterations are directly linked with the bit-accuracy achieved. There are two main ways to implement the CORDIC algorithm to compute sinusoid values: vector mode and rotation mode.

The aim of CORDIC is to avoid the use of multiplications and simply use logic shift to the left and right (equivalent to multiplying and dividing by 2) and additions, providing a hardware efficient computation of complex mathematics. CORDIC is based on two fundamental concepts:

1. Rotating a point (x , y) is equivalent to rotating the point first by $+45°$ and then by $-15°$.

2. Choosing angles (θ_i)to rotate by such that it satisfies $tan(\theta_i) = 2^{-i}$ as it allows multiplication by $tan(\theta_i)$ in equation (13) to simply be binary shifts.

E.1 Rotation Mode

If we have an initial point at (x , y) and rotate it about the origin by θ, our new rotated coordinate can be defined by the matrix below:

$$\begin{bmatrix} x_R \\ y_R \end{bmatrix} = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} \tag{8}$$

Equation 12 can equivalently be expressed as:

$$x_R = x_1 cos(\theta) - y_1 sin(\theta) \tag{9}$$

$$y_R = x_1 sin(\theta) + y_1 cos(\theta) \tag{10}$$

Having the initial positions, (x_1, y_1), of equations (9) and (10) as (1,0), the rotated points can now provide values for cosine and sine of a particular angle as follows:

$$\begin{bmatrix} x_R \\ y_R \end{bmatrix} = \begin{bmatrix} cos(\theta) - 0 \\ sin(\theta) + 0 \end{bmatrix} = \begin{bmatrix} cos(\theta) \\ sin(\theta) \end{bmatrix} \tag{11}$$

Factorising the matrix in equation (12) by taking the $cos(\theta)$ out allows the matrix to be expressed in terms of $tan(\theta)$.

$$\begin{bmatrix} x_R \\ y_R \end{bmatrix} = cos(\theta) \begin{bmatrix} 1 & -tan(\theta) \\ tan(\theta) & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} \tag{12}$$

Note: The $cos(\theta)$ term is the system's gain as both the x and the y components are scaled by the same amount. It can be seen that after N iterations, the system's gain is simply the product of $cos(\theta_i)$ values :

$$\begin{bmatrix} x_{Ri} \\ y_{Ri} \end{bmatrix} = \prod cos(\theta_i) \times \prod_{i=1}^{i=N} \begin{bmatrix} 1 & -tan(\theta_i) \\ tan(\theta_i) & 1 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} \tag{13}$$

Since θ_i is chosen such that $tan(\theta_i) = 2^{-i}$ and $\prod cos(\theta_i)$ is dependent on the number of iterations, we can pre-compute the system's gain and store it into a LUT to be applied only once at the end of the product of rotations in equation (13). For maximum accuracy, the system's gain does not need to dependent on the number of iterations our computation may use; The system's gain is simply 0.607252935 in floating-point and the number of significant figures utilised is determined by the number of bits that is being used in the fixed-point format representation of floating points.

Expanding the equation (13) whilst ignoring the system gain, the following system of equations are:

$$x[i+1] = x_i - \sigma_i 2^{-i} y_i \tag{14}$$

$$x[i+1] = y_i + \sigma_i 2^{-i} x_i \tag{15}$$

The σ_i is a variable that defines whether the rotation is clockwise or anti-clockwise. To determine this, we need to know whether the esired rotation is larger or smaller than the current achieved rotation. Introducing a new variable z and initially setting $z = \theta_{desired}$, the difference between our desired and achieved rotation is given by:

$$z[i+1] = z[i] - \sigma_i \theta_i \tag{16}$$

If $z[i+1] > 0$, then the desired rotation is greater than the previous achieved rotation and

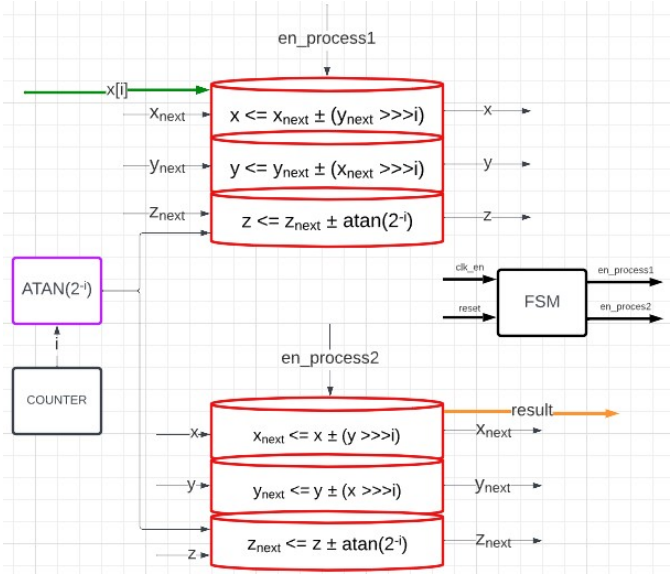
hence, we need to rotate clockwise, making $\sigma = +1$. This is a form of negative feedback accumulating all previous angle of rotations and comparing it with the desired angle of rotation.

E.2 Vector Mode

The vector mode is a modification to the rotation mode where the vector is initially pointing towards a coordinate with an arbitrary y-coordinate and a positive x-coordinate. The vector is then successively rotated such that it aligns with the x-axis (y-component approaches zero). Here, σ contains the total angle of rotation whilst y determines the direction of rotation. The vector mode is essentially the transformation of rectangular coordinates to polar coordinates.

E.3 Latency Optimised design pipeline

Figure 19: Latency optimised CORDIC design



References

[1] https://www.allaboutcircuits.com/technical-articles/an-introduction-to-the-cordic-algorithm/

[2] https://www.h-schmidt.net/FloatConverter/IEEE754.html

[3] https://digitalsystemdesign.in/wp-content/uploads/2019/01/cordic1.pdf

[4] https://www.intel.com/content/www/us/en/docs/prograr 1/altfp-add-sub-signals.html

[5] https://www.intel.com/content/www/us/en/docs/prograr 1/altfp-mult-ip-core.html

[6] intel.com/content/www/us/en/docs/programmable/6837 1/altfp-convert-features.html

[7] https://www.intel.com/content/www/us/en/docs/prograr 1/direct-memory-access.html

[8] https://www.youtube.com/watch?vM̄16l_ymlfcs&ab_chan

[9] https://www.youtube.com/watch?v8̄GAqT3nzHeQ&t=24

[10] https://www.youtube.com/watch?vw̄i-po2SesoE&ab_channelZ̄eroCode

[11] https://www.geeksforgeeks.org/ieee-standard-754-floating-point-numbers/