Imperial College London

ELEC96011 - Digital System Design

Imperial College London

Department of Electrical Engineering

Coursework Report 3

Authors:

Jaafar Rammal 01576194 (jr4918@ic.ac.uk)
Marian Lukac 01511163 (ml11018@ic.ac.uk)

Lecturer:

Dr Christos Bouganis

29th February 2021

Contents

1 Introduction

This final report covers the various hardware optimizations that can be utilized to maximize the performance of the function computation task while using the least possible resources and maintaining an acceptable precision. First are covered dedicated floating-point hardware, along which custom instructions are introduced. Then, the report explores the optimization of the cosine function using CORDIC as well as the various implementations and their tradeoffs. Finally, the last sections cover the implementation of the full inner expression within a custom IP, followed by the function sum, before concluding with the various optimizations brought to that complex IP.

2 Evaluation baseline

In terms of evaluation, all the following sections in this report will be based on an O3 compilation flag and the previously optimized cache values (16KB instruction cache and 8KB data cache). The tests presented in Table 1 are used, and all execution times are averaged over 10 runs (or more if necessary) and presented in ms unless mentioned otherwise. The theoretical minimum timing is the theoretical minimal time required by NIOS to compute the function assuming one clock cycle per input with a 50MHz clock (even though this is impossible, taking into account memory access for example). This will be useful for Task 8 when optimizing, for example, with DMA, since the timer interrupt might not fire and NIOS might report timings smaller than the values presented in this table.

	Input array size	Step size	Matlab result	Theoretical min timing
Testcase 1	52	5	920413.626649442	0.00104 ms
Testcase 2	2041	$\frac{1}{8}$	36123085.5519791	0.04082 ms
Testcase 3	261121	$\frac{1}{1024}$	4621489017.88862	5.22242 ms

Table 1: Parameters of different testcases. Step size denotes the difference between neighbouring elements and is constant across all elements.

As a reminder, the goal is to evaluate the following function:

$$y = \sum_{i=1}^{N} 0.5 * x_i + x_i^2 * cos((x - 128)/128) \tag{1}$$

where $0.5x_i + x_i^2 cos((x_i - 128)/128)$ is referred to as the inner expression.

3 Adding dedicated Floating point hardware

In previous designs, the floating-point operations were emulated using integer hardware, which caused a slowdown of the whole system. In this section, the effects of adding dedicated floating-point adder/subtractor and multiplier are discussed and their performance analysed.

3.1 Implementation and design

To create dedicated floating-point units, Intel's IP library was used, which contained logic blocks for addition/subtraction as well as for multiplication. For multiplier default design was used, while for adder/subtractor the *OpSel* signal was introduced to switch between addition and subtraction. This was done to create a working interface with NIOS as passing 2 results back to NIOS as the original design suggested wouldn't be possible. As for the internal design of these blocks, both of them are pipelined and take multiple clock cycles to produce results. More precisely, the multiplier takes 2 clock cycles and the adder/subtractor takes 3 clock cycles at a target frequency of 50 MHz. It is worth noting, that even though both of these blocks are pipelined, NIOS cannot make use of the pipeline and thus these blocks will be treated as simple multicycle instruction.

After the blocks were designed, their connection to NIOS was created. Due to its multicycle nature, the multiplier needed a clock to function correctly. Therefore, it was connected as a multicycle custom instruction. The decision to make the instruction fixed length was made, as it allows the multiplier to be connected to the NIOS directly without any need for interface adaptor since *done* signal is not necessary and other signals are already available on the multiplier block itself.

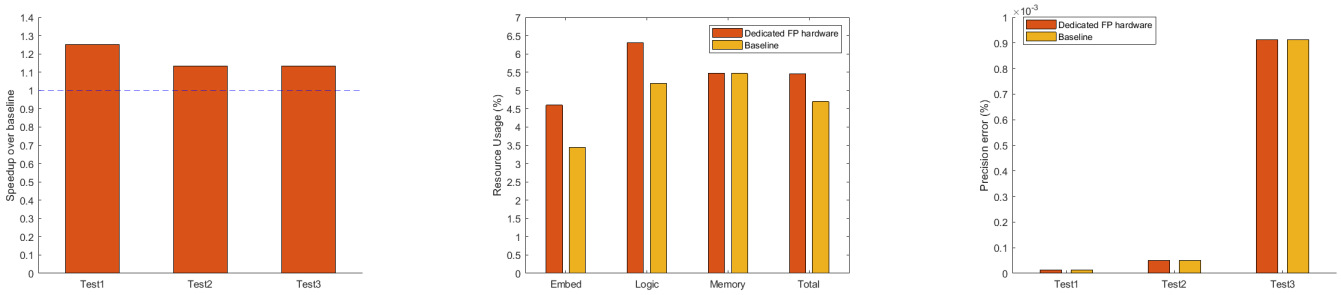
As for the adder/subtractor, it was implemented similarly to the multiplier. One exception was that an extended signal had to be added to control *OpSel* signal. This resulted in the final system having 2 separate custom instructions. The idea to implement all operations in one custom instruction was explored as well. However, it was quickly determined, that the complexity of the design and the extra resource usage caused by controlling which operation should happen, would outweigh the benefits.

3.2 Results

As a baseline for this task, a system with a dedicated integer multiplier was used. As figure 1 shows adding dedicated floating-point units increases performance by as much as 20%. However, this comes with increased resource usage, as figure 1b shows. The total resources used by this implementation are higher by 0.7% compared to baseline. As for precision, it stays surprisingly the same even though it was expected that floating-point hardware would be able to deliver better accuracy than emulation. This suggests, that instruction emulation prioritizes precision over performance and might explain why the performance difference is so significant. It is also worth noting, that code size also increased when using custom instructions. The increase was from 91 KB for baseline to 94 KB for custom instructions implementation. This increase can be attributed to the complexity of executing custom instructions and the number of instructions needed to get the data and signals ready.

3.3 Design limitations

As was discussed in section 3.1 the blocks used in this design are pipelined and thus are able to produce a new result on every clock cycle. Unfortunately, this is not utilized by NIOS as it does not have out-of-order execution and can process only one call to custom instruction at once. This means, that the NIOS will be waiting for the custom instruction to finish before executing the next one. If out-of-order execution would be present, NIOS could make use of the pipeline by, for example, calculating values for multiple elements of the input array at once. However, this is not the case and thus this design, even though it performs better than emulation, is not optimal. This is due to the fact, that floating-point blocks could be implemented in their rolled non-pipelined version, which would significantly decrease the amount of resource used by the blocks.



(a) Speeedup (b) Resource usage (c) Precision error

Figure 1: Graphs comparing implementation with dedicated floating point hardware to the baseline implementation across different metrics.

4 Optimizing Cosine with CORDIC

The next hardware optimization (or customization) targets the cosine function in the inner expression. The cosine can be computed iteratively using the CORDIC method, which involves multiple iterations using precomputed trigonometric parameters until the target value is reached within a desired error range. The following constraints apply:

- The goal is to compute the $cos(x), \forall x = [-1, 1]$
- The target MSE is $e \leq 1.E - 10$

Given the nature of the CORDIC algorithm and the possibility to pipeline the implementation in the future, it was decided to implement the algorithm using fixed-point representation. For that reason, it is crucial first to determine both the number of iterations required as well as the width of the fixed-point representation in order to achieve the target MSE. Since the input to the cosine is between -1 and 1, only two bits are required for the integer part.

In the following subsections, the Monte Carlo simulation is first used to determine the number of iterations and the width required. Then, different CORDIC implementations are designed and compared in terms of performance and resources with the different trade-offs analyzed.

4.1 Monte Carlo simulation

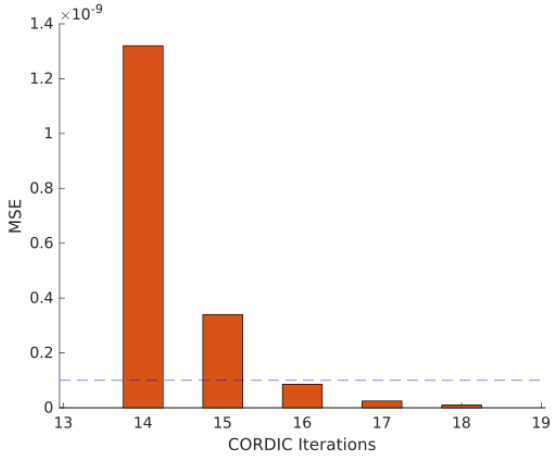
It was crucial to first determine the minimum number of iterations and the smallest fixed-point width that would satisfy the requirements, since minimizing those parameters would improve both performance and resources used.

A few key reminders about the Monte Carlo simulation:

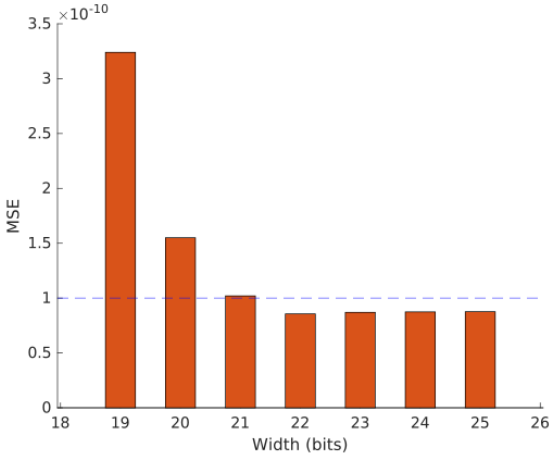
- The MSE is computed using $MSE = \frac{1}{N}.\sum e_i^2$
- For a confidence interval with 95% confidence, it is possible to use:
  - Lower end-point = $MSE + \frac{1.96std(e^2)}{\sqrt{N}}$
  - Upper end-point = $MSE - \frac{1.96std(e^2)}{\sqrt{N}}$
  - STD: standard deviation
- N is set to 10 000

Iterations Width	14	15	16	17	18
19	1.66E-09	5.30E-10	3.24E-10	3.00E-10	2.88E-10
20	1.45E-09	3.95E-10	1.55E-10	9.21E-11	7.65E-11
21	1.41E-09	3.72E-10	1.02E-10	4.03E-11	2.54E-11
22	1.32E-09	3.40E-10	8.56E-11	2.47E-11	1.01E-11
23	1.41E-09	3.03E-10	8.20E-11	2.23E-11	6.16E-12
24	1.27E-09	3.35E-10	8.14E-11	2.15E-11	5.56E-12
25	1.38E-09	3.34E-10	8.06E-11	2.06E-11	5.42E-12

Table 2: Monte Carlo simulation with the MSE values obtained for varying iterations and width. Target is MSE $\leq 1.e^{-10}$



(a) MSE with word width fixed at 22 and varying iterations



(b) MSE with iterations fixed at 16 and varying word width

Figure 2: MSE varying with different word width and iterations count

The results of the MSE values obtained for the 95% confidence interval can be seen in Table 2, with values meeting the requirements in green and others in red. Slices at iterations=16 and width=22 can be seen in Fig. 2a and Fig.

2b respectively. It is clear from the table and the figures that increasing the number of iterations reduces the error exponentially, while the word width improves the error only up to a certain value and then it converges to a value. The optimal selection would be 16 iterations and 22 bits for the fixed-point representation word width (which corresponds to 2 bits for the integer part and 20 bits for the fractional part).

4.2 FPGA implementations

Now that the necessary parameters have been selected using the Monte Carlo simulation, it is possible to implement the CORDIC IP in Verilog. Both the input and the output of the CORDIC IP will be a 22-bits fixed-point representation, since the input is between -1 and 1, and the output is between 0 and 1. Four different designs are explored as required by the specifications of the coursework:

- (A) Fully folded: single stage with a 16-iterations loop
- (B) Fully unrolled: 16 stages one after the other
- (C) Folded with multiple iterations per pipeline stage (4 pipeline stages in total)
- (D) Unrolled with multiple iterations per pipeline stage (4 pipeline stages in total)

4.2.1 Iterator block

One CORDIC iteration will follow the following equations, keeping in mind the goal is to grab $x^{(16)}$ which will be the cosine of the input after 16 iterations:

- $x^{(i+1)} = x^{(i)} - d_i.y^{(i)}.2^{-i}$
- $y^{(i+1)} = x^{(i)} + d_i.x^{(i)}.2^{-i}$
- $z^{(i+1)} = z^{(i)} + d_i.tan^{-1}(2^{-i}) = z^{(i)} - d_i.e^{(i)}$
- $d_i = -sign(x_i.y_i) = -sign(y_i)$

The generated RTL can be seen in Fig. 3. It takes as inputs $x^{(i)}$, $y^{(i)}$, $z^{(i)}$, n (the current iteration stage, in the case of the equations $n = i$), and $e^{(i)}$. The iterator was then tested independently by trying our different iteration indices (setting i and $e^{(i)}$ in ModelSim) and crosschecking the values with Matlab to confirm the correctness of the iterator.

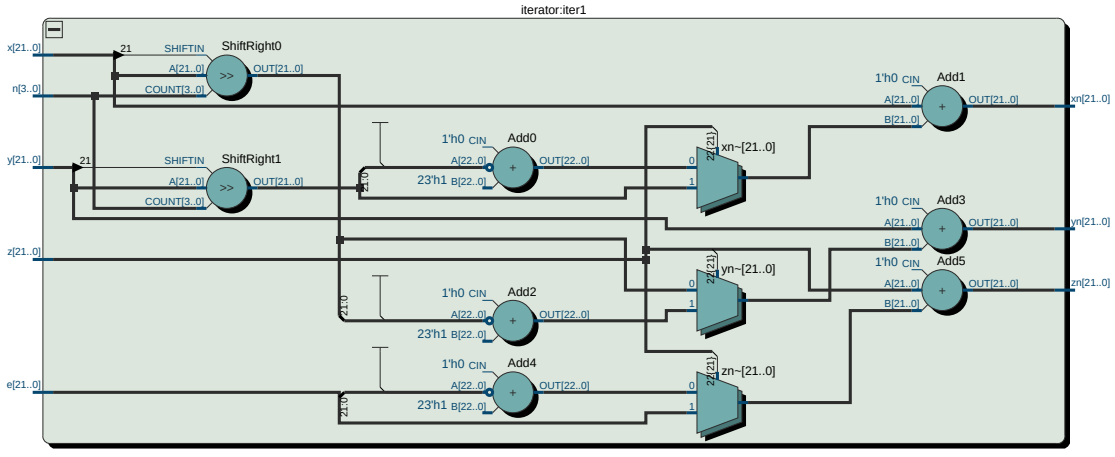


Figure 3: Iterator block for one CORDIC iteration

4.2.2 Fixed-point wrapper

The next step is to wrap the CORDIC implementations with fixed-point conversions. The ALTERA_FP_FUNCTIONS IP was used to convert between 32-bit floating-point and 22-bit fixed-point (with 20-bit wide fraction part).

As can be seen from Fig. 4, the input to the CORDIC system is a 32-bit floating-point angle, and the output is a 32-bit floating-point cosine of that angle. A start signal is also used to indicate to the CORDIC block when to start computing the cosine. Both conversion blocks are synchronous and execute within one clock cycle, hence the use of a register at the start signal to ensure it goes high only when the input finished converting and is ready to enter the CORDIC block.

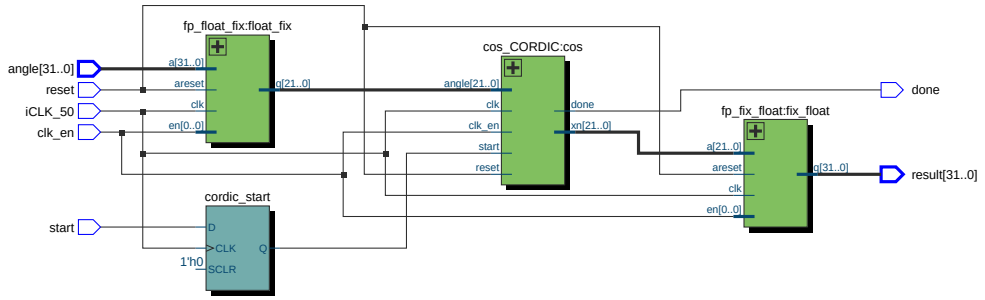
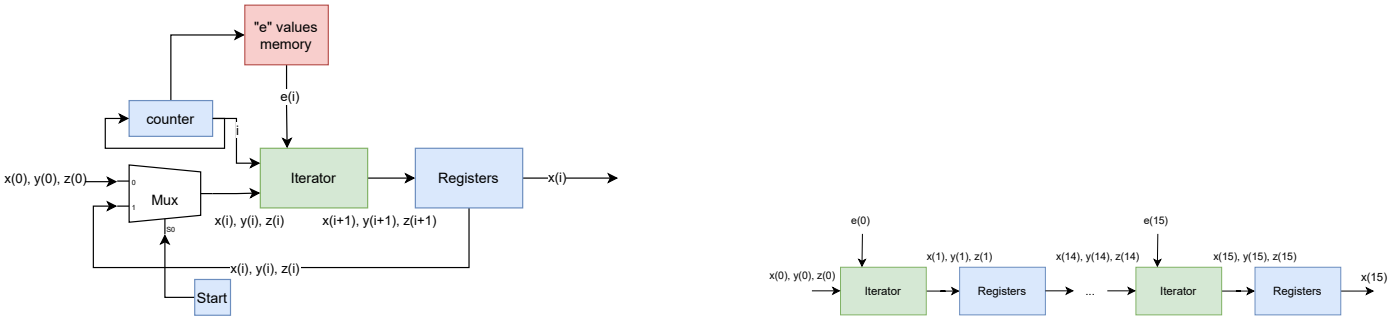


Figure 4: Fixed-point conversion wrapper

4.2.3 Different CORDIC designs

Following the correct input conversions, it is now easy to implement designs A and B. Their simplified hardware diagrams can be seen in Fig. 5. The designs are then tested for correctness in ModelSim and they both provide an answer in 16 cycles (excluding the conversion cycles) with similar precision to the Matlab implementation that respects the precision requirements.



(a) Fully folded: single stage with a 16-iterations loop

(b) Fully unrolled: 16 stages one after the other

Figure 5: CORDIC implementations with 1 iteration per stage

It is now possible to explore a design where multiple iterations are executed within one clock cycle. For that purpose, the process is to simply try and wrap multiple iterator blocks in series before a single register-file and then verify with timing-analyzer if the timing constraints are met. In other words, the goal is to make things as combinatorial as possible while maintaining an acceptable critical path. After testing different combinations, putting 5 iterators would be possible, yet it wouldn't yield to any advantage as it would still need 4 stages (5-5-5-1), therefore using 4 iterators per stage (4-4-4-4) is chosen as best. A CORDIC stage RTL can be seen in Fig. 6.

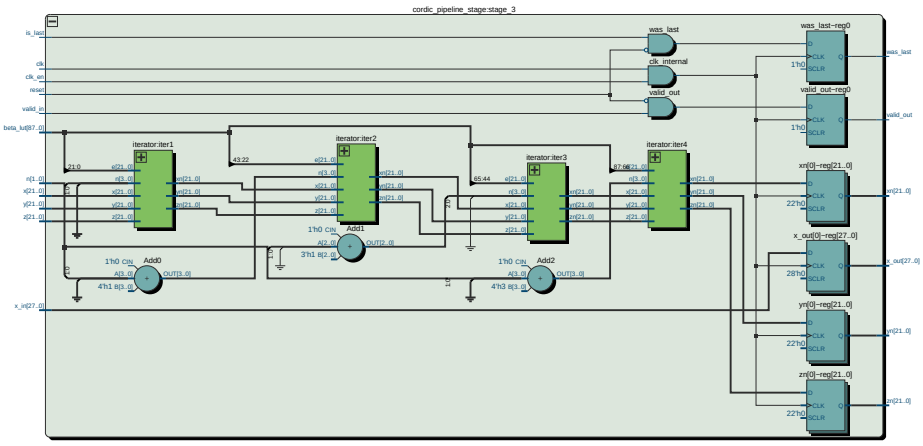


Figure 6: One CORDIC stage with 4 iterators

As can be seen in the RTL, a stage take in inputs, in addition to the previous CORDIC state (x, y, and z), 4 "e" values entries (denoted as BETA_LUT) and the current stage position (n, between 1 and 4). Note that two other signals, "valid_in" and "is_last" are included, and these will be discussed in a later section. The same applies for the registers "was_last", "valid_out", and "x_out".

It is now possible to implement designs (C) and (D) described previously in the section introduction by simply replacing the iterator-registers pairs with a CORDIC stage, and looping / sequencing 4 times instead of 16 times. The comparison between the different designs in terms of latency, throughput, and resources, can be seen in Table 3. In all scenarios, the CORDIC IP outputs the same cosine value, which matches the precision requirements with $MSE \leq 1.E-10$

Design	**Latency (cycles)**	**Iterators**	**Resources (ALMs)**	**Throughput**
A	16	1	130	1 input / 16 cycles
B	16	16	620	1 input / 1 cycle
C	4	4	300	1 input / 4 cycles
D	4	16	700	1 input / 1 cycle

Table 3: Comparing metrics between the different CORDIC designs

There is definitely an advantage when using multiple iterators per stage, as this improves the overall latency of the IP. The choice between (C) and (D) depends on the parent design using the IP: if the design will input one input and wait for the result, (C) is better as it uses less resources. Yet, with (D), it is possible to pipeline the overall design in the future and achieve a much higher throughput, making the overall function computation faster since a result is generated at every clock cycle instead of four clock cycles.

5 Custom IP for the inner expression

Following the previous section, it is now possible to extend the IP to integrate the full inner expression. That is, the IP will now compute $y = 0.5x + x^2 cos((x-128)/128)$. First, the design will be integrated in Verilog and tested using ModelSim, before integrating it with NIOS II and testing the IP in software.

5.1 Design

First, it is important to note that the input format needs to be modified. The input and output of the IP will both be a 32-bit floating point number. Yet, at the inside, while we still need at least 20 bits for a fixed-point representation for the fractional part, 16 bits are required for the integer part, since x is in the range $[0, 255]$, where 255 requires 8-bits, the multiplication x^2 will cause the output y to require at most 16 bits. The fixed-point wrapper is therefore modified.

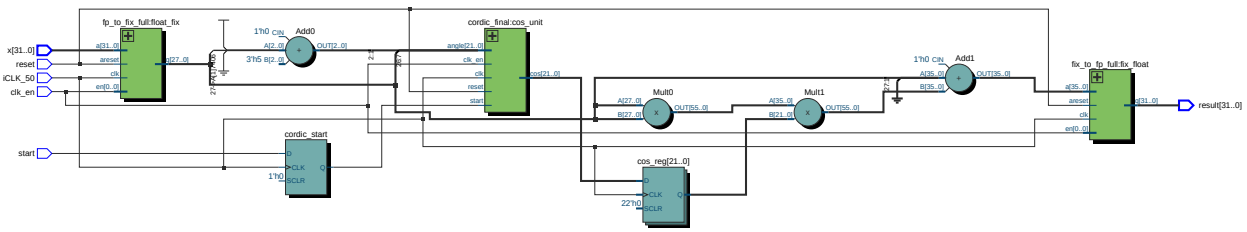


Figure 7: Inner expression implementation

The next step involves adding the expression elements. There is a need for two multipliers, two adders, and a few shifts. The generated RTL can be seen in Fig .7. The design assumes that NIOS holds the input for the whole process. The square term is computed in parallel with the cosine, then the product is computed afterwards. It was established that placing those multiplications after the cosine block in a combinational fashion was possible, as this would allow piping the first cordic stage directly with the float-to-fixed IP, reducing the latency by one clock cycle for the overall IP.

5.2 NIOS results

The IP was integrated into the NIOS II function and makes use of the FP Adders created in task 6 to compute the sum. The IP is used to compute the inner expression. NIOS calls the IP and then halts for 19 cycles before reading the result for each array element. The results can be seen in Table 4.

Design	Test 1			Test 2			Test 3			Resources
	Result	Error	Ticks	Result	Error	Ticks	Result	Error	Ticks	
Task 6	920413.5	1.3E-5	4	36123104	5.1E-5	157	4621531136	9.1E-4	20186	5.46%
(C)	920414.125	5.4E-5	0.05	36123136	1.39E-4	1.98	4621529600	8.78E-4	271	6.76%
(D)	920414.125	5.4E-5	0.05	36123136	1.39E-4	1.98	4621529600	8.78E-4	271	7.08%

Table 4: Metrics comparison between computing the inner expression in the IP and Task 6 design

As the table shows, there is a drastic improvement in the latency, especially when it comes to test 3, by about 100 times. Nevertheless, because fixed-point representation is used all the way through when computing the expression, precision is lost for smaller tests. This design choice has been made to simplify the design and the clocking of the circuit as well as take advantage of the combinatorial implementations of integer-based arithmetic circuits, at the expense of very little precision loss. Finally, as expected, the CORDIC-based IPs to compute the inner expression use more resources. In the case of this task, design C (folded with 4 iterations per stage) is the best as it would use less resources than design D (which provides no extra advantage when using one input per result computation).

6 Custom IP for the full expression

Following the custom IP to compute the inner expression, it is now time to extend the IP such that it can compute the full function (including the outer sum). In the first subsection, different pipelined designs are covered, before jumping onto a DMA-based implementation in the second section.

6.1 Pipelined designs

In the previous section, a folded-based design was selected since NIOS is only providing one input at a time and waiting for the result before adding in using the FP Adder and then providing the next input. Yet, it is possible to use the unfolded design and utilize the pipeline feature and high throughput characteristic to achieve a much lower latency. For that purpose, it is necessary first to have the sum accumulate internally in some way so that NIOS II can provide data at a higher throughput without awaiting for the result. The inner expression design is therefore modified and a register accumulating the result is added. Moreover, the output is now 54 bits fixed-point instead of 36 bits since the sum will be the output of the IP instead of the inner expression evaluation. The modified block can be seen in Fig. 8.

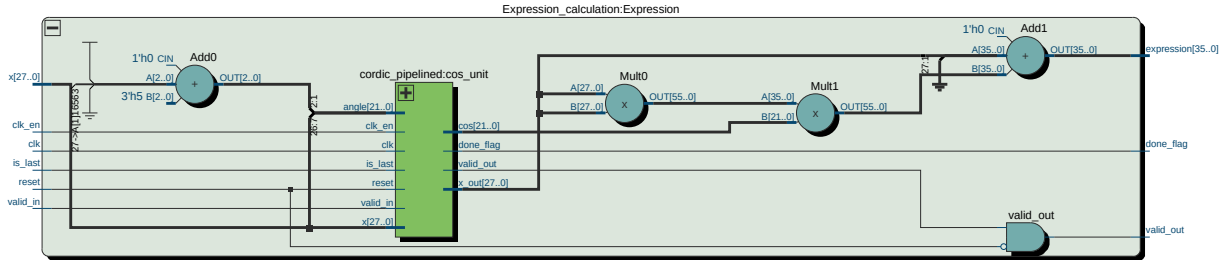


Figure 8: Modified inner expression computation, using unrolled CORDIC with 4 stages

As can be seen in the figure, the main change is that now the original input is pipelined and preserved within the CORDIC block so that it can be used afterwards (and therefore NIOS does not have to hold the input until its inner expression is computed anymore). In addition, it was established that placing the two multiplication blocks and the adders after the CORDIC block would save a clock cycle, as it was now possible (Fig. 9 to link in a combinatorial fashion the first CORDIC stage with the float-to-fixed converter, as well as the two multiplications and the adder with the first stage of the sum accumulator. Finally, a done flag and a valid signal are used to control the pipeline and communicate accordingly with the NIOS interface.

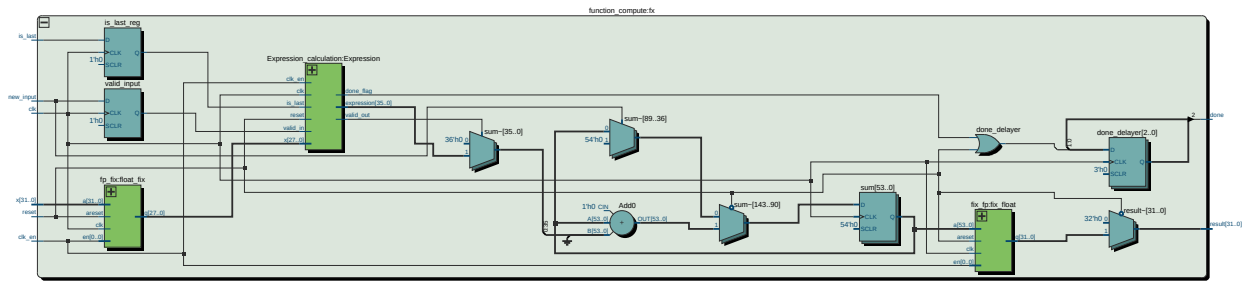
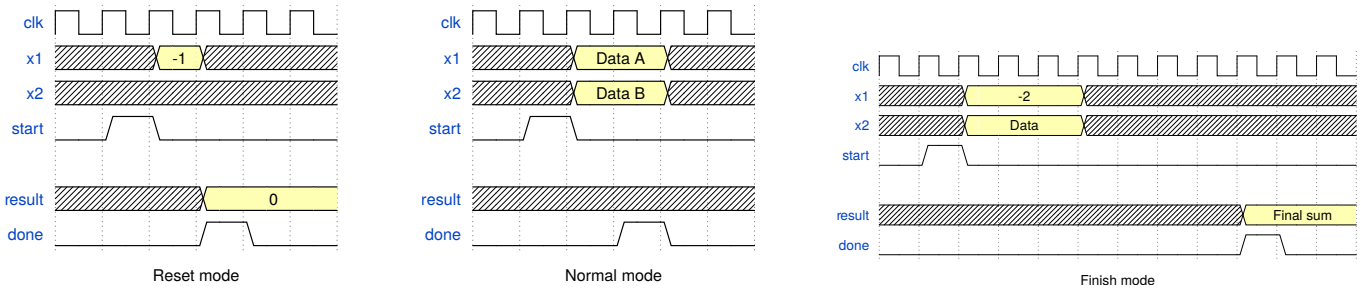


Figure 9: Full expression computation with the sum accumulation

Fig. 9 shows the integration with the accumulator for the sum. A reset signal can be used to reset the accumulator value to 0, and the done register, which signals to NIOS that it can provide the next input, is implemented using a shift register to accommodate for the 2 cycles delay from the fixed-to-float converter. The design can then communicate with NIOS using different modes, where it takes in two inputs x1 and x2, outputs a full expression sum, and sends a done signal to unhalt the NIOS:

- Reset mode: activated for x1 = -1. Sets the accumulated sum to 0. See Fig. 10a.
- Normal mode: NIOS provides two inputs x1 and x2. The IP puts them in the pipeline and directly sends a done signal without waiting for the sum to update. See Fig. 10b.
- Finish mode: NIOS provides x1 = -2 and a valid x2 input (set to 0 otherwise). The IP puts them in the pipeline and awaits for the sum to update correctly using all the pipeline valid values before sending the done signal. See Fig. 10c.



(a) Reset mode (b) Normal mode (c) Finish mode

Figure 10: Different modes interfacing with the NIOS

It is worth noting that another design using almost the same blocks has been used prior to this one, where the only difference is that the inputs are x and the current sum. For the purposes of the report describing useful designs, it will only be reported in the results as a milestone to achieve the two-inputs design.

Design	**Test 1**			**Test 2**			**Test 3**			**Resources**
	Result	**Error**	**Ticks**	**Result**	**Error**	**Ticks**	**Result**	**Error**	**Ticks**	
Folded	920414.125	5.4E-5	0.05	36123136	1.39E-4	1.98	4621529600	8.78E-4	271	6.76%
1 input	920414.1875	6.0E-5	0.02	36123124	1.06E-4	1.22	4621520896	6.79E-4	160	5.98%
2 inputs	920414.1875	6.0E-5	0.02	36123124	1.06E-4	1.1	4621485056	8.57E-5	140	5.98%

Table 5: Metrics comparison between computing the full expression in the pipelined designs and folded design (C)

Table 5 presents the metrics of the two new designs compared to the previous folded design that lacked an accumulator. Using the pipeline presents a significant improvement in terms of speed, as well as resources since the FP Adder does not need to be used anymore. In terms of precision, the main reason precision is improved as we keep the accumulator independent from NIOS is because there isn't a back-and-forth in the conversions: for the 1-input design, the current sum is converted to floating-point, passed to NIOS, which passes it back as an input in the next cycle to be added. The 2-inputs designs eliminates this redundant conversion and therefore less precision is lost.

One might wonder why isn't there a major improvement between the 1-input and 2-inputs designs. Surely we should expect something almost twice as fast? This means that even though the pipeline has been optimized as much as possible, the throughput is limited by how much data NIOS can provide. Since NIOS is using caches that need to communicate with the SDRAM, some penalty applies, which results in the low utilization of the pipeline.

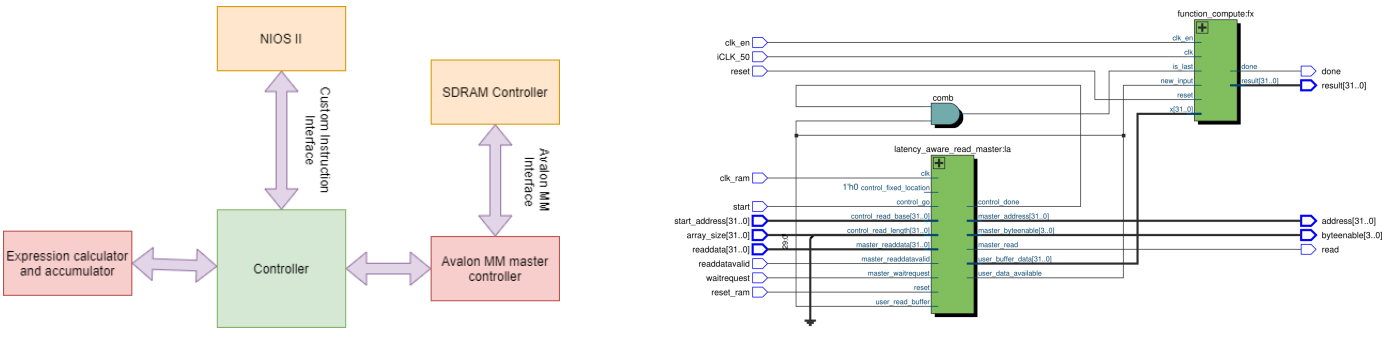
6.2 Eliminating memory bottleneck with DMA

Previous designs described in this section were trying to address the main problem with the system, which is the low utilization of pipelines. Even though some of them succeeded in increasing utilization, the main bottleneck, which is the necessity of asking NIOS for data, was not solved and the pipeline could not be used to its full potential. In order to address this issue, direct memory access (DMA) was used to access data directly from memory. This will not only significantly decreases communication overhead but also speeds up data access throughput as there is no cache policy to slow down the loading of data. Additionally, it makes use of both data and instruction cache obsolete in NIOS, as there is only one instruction to calculate target expression, which communicates with memory directly without caching.

6.2.1 Design

In order to implement DMA, communication with SDRAM had to be facilitated inside the custom block. Since the system already provides a controller for SDRAM, only the interface to the controller had to be created. In order to communicate with the controller Avalon Memory Mapped (MM) master interface was used to match the Avalon MM slave interface available on the controller. The interface block used in the design was obtained from Intel website[1]. The modules available there, support many different modes of operation, but the one chosen in the design was the interface supporting both normal and pipelined reads. It is worth noting, that implementation supporting burst reads was not chosen, even though the data access required could benefit from burst reads, as SDRAM controller does not support them. Chosen interface module uses FIFO block to store data returned by SDRAM controller and exposes an interface for the other module to retrieve them from the queue. This queue was shrunk to its minimal size, since the rest of the system is fully pipelined and thus it can process data as soon as they are available, making the queue only 1 value at a given time anyway. After the module was modified and adjusted for the target use-case, a control block for connecting it to the rest of the system was created.

There were 2 approaches explored with respect to the controller. The first one used the Avalon interface module in normal mode, where a read request is sent and only after data is received by the controller another request is sent. In order to implement this version, a full controller with a state machine was required. A high-level view of this approach can be seen in figure 11a. On the other hand, if the pipelined mode is used in the Avalon interface module, the interface module can be directly connected to *function_compute* block, which computes the whole expression and also contains an accumulator for the final result, without any additional control block in-between (see figure 11b). This makes the whole design simpler and less resource-intensive, suggesting that this approach might be better. However, both approaches were tested in order to see how big the performance benefit of pipelined reads will be.



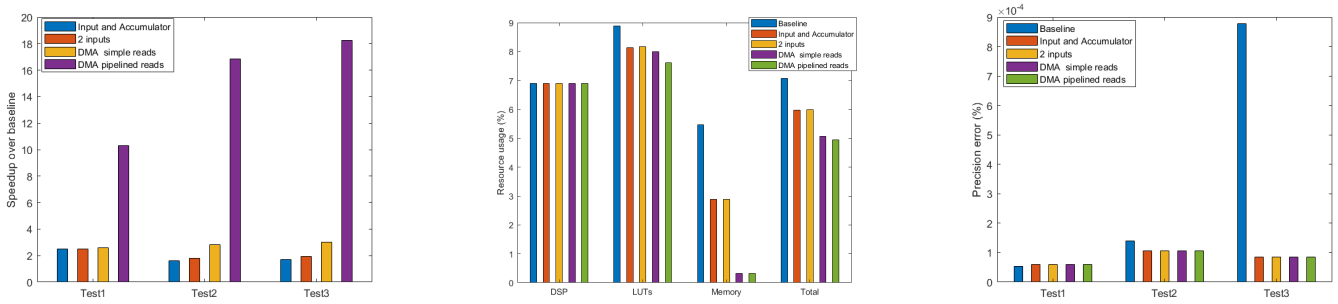
(a) Single reads

(b) Pipelined reads

Figure 11: System overview of 2 different designs. Each design uses different data reading mode.

6.2.2 Results

As the results in figure 12 show, DMA improves the design with respect to both performance and resource usage. Even though the amount of LUTs used is similar across all designs, the fact that cache is no longer necessary with DMA decreases the number of memory blocks needed drastically, decreasing the total resource usage by more than 1% compared to previous designs. As for accuracy, it stays the same. This is not surprising, since the design uses the same computing block for calculating the expression result as the previous designs. Performance-wise, figure 12a shows that single reads design is only slightly faster than previous designs, while pipelined reads one is more than 9 times faster than any previous design, confirming the hypothesis, that pipelined reads design is more efficient. The performance improvement of DMA approach was to be expected as DMA design minimizes the main bottleneck of the system, which was communication with NIOS II and allows the system to go as fast as SDRAM is able to supply data. However, the poor performance of single read design suggests, that the single reads are not the optimal way of fetching data from SDRAM and that in order to get higher throughput from SDRAM, pipelined reads should be used.



(a) Speeedup (b) Resource usage (c) Precision error

Figure 12: Graphs comparing different implementations to the baseline implementation across different metrics.

7 Conclusion

In this report, multiple hardware optimizations were covered. First, general-purpose floating-point arithmetic IPs were explored for multiplication, addition, and subtraction, to improve performance and move away from emulated operations. Following this, a focus was brought on the cosine function, which was optimized using the CORDIC algorithm. The algorithm parameters were decided using the Monte Carlo simulation, and different design approaches were tested for different purposes (throughput or latency). Finally, an IP was designed to compute the inner expression and then the full expression. This design process involved first designing a robust pipeline with a maximized throughput, before maximizing the pipeline utilization with Direct Memory Access (DMA) to bypass the NIOS caches penalty. The designs comparison can be seen on the scatter plot in Fig. 13. Overall, it was possible to fully optimize the system performance with great result precision, low latency, and minimal resources, demonstrating the strength of customized hardware accelerators on top of general-purpose processors like NIOS.

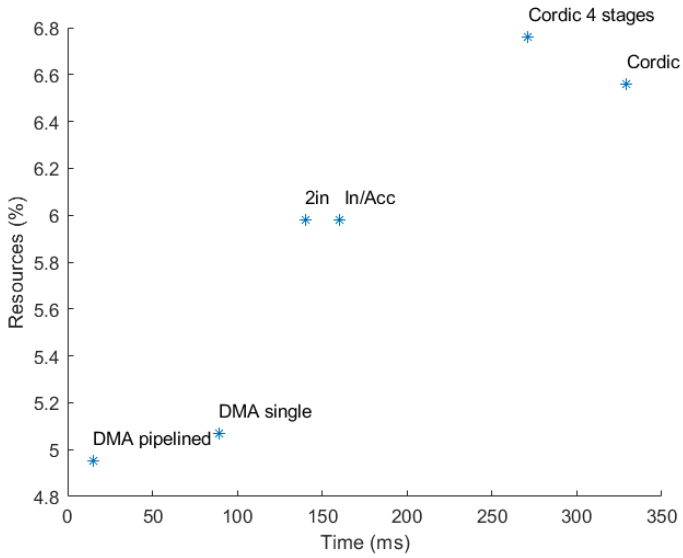


Figure 13: Graph plotting resource usage against execution time in Test 3 for selected designs.

References

[1] Avalon mm master templates. https://www.intel.com/content/www/us/en/programmable/support/support-resources/ Accessed 23 March 2021.