Imperial College London

Imperial College London

Department of Electrical and Electronic Engineering

Report 3

ELEC60011 - Digital Systems Design 2022-2023

Professor Christos-Savvas Bouganis

Authors (Group 5):
Michal Palič (CID: 01942994)
mp3120@ic.ac.uk
Vaclav Pavlíček (CID: 01868933)
vp920@ic.ac.uk

March 20, 2023

1 Task 6 - Hardware FPU

1.1 Introduction

The purpose of this task was to further improve on the hardware delivered in Task 5, through the inclusion of floating point operations in custom instruction hardware to be accessed by the processor. The custom instruction interface of the NIOS processor was used to interact with multi-cycle floating point hardware.

The approaches guiding our exploration are re-iterated below. At a clock frequency of 50 MHz, most of the performance of the NIOS/f system (Nominal F_{max} of 170MHz) and Memory (Rated F_{max} of 143MHz) is left untapped. Based on the frequency limit of the memory, a frequency of 143MHz was targetted. Further increases by decoupling the frequencies of the memory and processor systems are possible but were not investigated due to the added complexity. An increase in the clock frequency is expected to result in an approximately proportional increase in throughput. The only exception would be the SDRAM memory system, whose parameters such as RAS to CAS access time (t_{RCD}) and access time (t_{ac}) are specified in terms of absolute time and will therefore not scale with frequency. This is expected to be largely mitigated through the use of an appropriate cache hierarchy, whose performance will scale with frequency and service a majority of the memory traffic.

The reference approach to timekeeping was deviated from. Driven in part by the inclination of high compiler optimisation levels to remove redundant code complicating the averaging of repeated runs and the coarse 1ms increments of the system timekeeping, the alt_timestamp library was used instead to time the execution duration with cycle accuracy. This single run is more representative of real life scenarios as the cache begins unprimed unlike during repeated runs.

1.2 Block architecture

The target for this hardware design was to present all of the relevant floating point hardware instructions in a concise package. An FPU-like structure was devised, with two inputs and a two-bit opcode from the NIOS custom instruction slave interface.

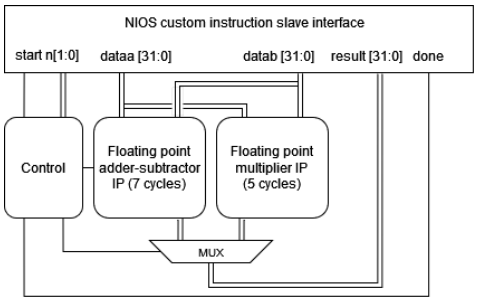

Fig. 1: FPU block diagram.

Fig. 1 shows a block diagram of the FPU. It consists of a floating point adder-subtractor and a multiplier. The opcode field was originally introduced to select the mode of operation for the adder subtractor allowing for hardware re-use and later expanded to include multiplication.

Given the $143MHz$ target frequency, for the adder-subtractor IP, the minimum achievable latency was 7 cycles. For the multiplier IP, the minimum achievable latency was 5 cycles. In order to extract the maximum performance from these instructions in one package, a variable duration custom instruction type was used. This way, the instruction is capable of terminating early for a multiplication, while utilising the full 7 cycles for an addition. A shift register and multiplexers were used to generate a done signal after the relevant number of cycles.

1.3 Cache re-tuning

Fig. 2 shows the effect of the instruction cache size on the execution time of test case 3. The data cache was set to its maximum value of $64KB$ to isolate the impact of instruction cache size during testing. The plot shows that the execution time was the largest when the instruction cache is disabled. A significant improvement was observed by increasing its size to $512B$ with an execution time plateau for larger instruction cache sizes. It was noted that this "L" shape has a significantly sharper elbow compared to the Task 4 cache tuning attempts. By using custom instruction calls instead of floating point emulation the number of instructions within the `sumVector` function was reduced to 136 ($544B$). The compiler inlined the function call to custom mantissa indexed lookup table cosine implementation. It is believed that this is the reason for the increased sharpness of the plateau. Calls to functions scattered across the address space would normally result in instruction cache conflicts as there will be a degree of overlap when directly mapping to a small number of cache lines. This problem would be gradually alleviated as the instruction cache grows, with the collisions decreasing in likelihood as the cached code fragments spread across the greater number of directly mapped indices.

In this case, a single contiguous section of instructions either fits in the cache or does not. No caching conflicts will occur if the cache size is greater than this contiguous section of instructions which can be seen in the plot.

The removal of large cosine calculation and floating point emulation routines has resulted in the point of diminishing returns shifting significantly from the $32KB$ determined for Task 4 all the way down to $512B$. It was noted that $544B$ of the 136 instructions of the `sumVector` is greater than the 512-byte cache size, but the actual hot section of the code is slightly smaller as the function begins and ends with sections executed only once performing operations such as pushing and popping registers from the stack when taking/returning control.

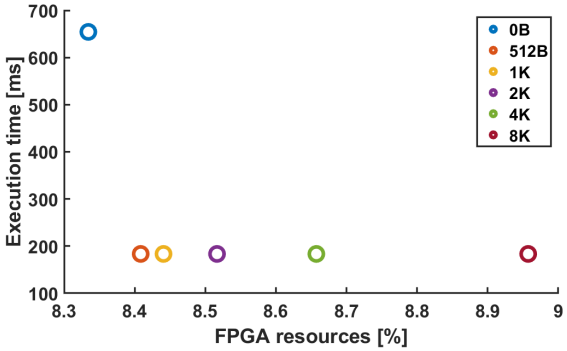

Fig. 2: I-cache size, latency and resource relationship.

Fig. 3 shows the impact of the data cache size on the execution of Test case 3 with the instruction cache being set to $64KB$. It also has a sharper elbow compared to the data cache experiment from Task 4. It is believed that this is also a consequence of a lack of function calls in the main loop, as the stack operations during function calls no longer take place and thus do not pollute the data cache. A $2KB$ data cache was selected as the performance plateaus beyond this point. This is again significantly lower than the corresponding $8KB$ determined in Task 4.

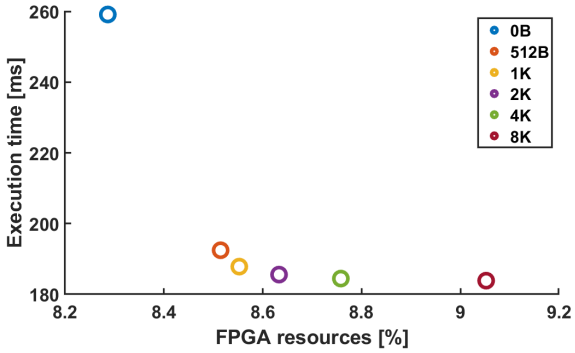

Fig. 3: Impact of changes in data cache size on the execution time and resource utilisation.

Table 1 shows the execution duration of Task 6 compared to Task 5. The table shows, there is a significant improvement in the latency of the system - Task 6 (32KB I and 8KB D) runs approximately 6.7 times faster than Task 5 and Task 6 (512B I and 8KB D). We also confirm that there is only a marginal increase in latency when comparing this new cache configuration with the old one while decreasing resource usage by 40%.

System	Test 1 [μs]	Test 2 [μs]	Test 3 [μs]	Resources [%]
Task 5 (32K I 8K D)	263	9611	1233061	5.46
Task 6 (32K I 8K D)	39	1511	183856	6.75
Task 6 (512B I 2K D)	52	1571	185564	4.04

Table 1: Test cases execution time comparison with Task 5 hardware.

1.4 Conclusion

The importance of cache re-tuning was shown in this task, with significant improvements in terms of both latency and resource utilisation being drawn from the decreased main loop instruction count.

2 Task 7 - CORDIC

2.1 Introduction

The aim of this task was to develop a hardware block that evaluates the expression $y = 0.5*x+x^2*cos((x-128)/128)$ in dedicated custom instruction hardware. This included evaluation of the cosine function using the CORDIC algorithm. The whole process was split into multiple steps. First a Monte Carlo simulation with a MATLAB CORDIC implementation was run to determine the parameters of the fixed point type and the number of CORDIC stages such that an MSE of $< 1 * 10^{-10}$ was achieved. Then the CORDIC block was implemented in Verilog and verified using test benches. Before integrating into the final design, a clock frequency analysis of different points in the CORDIC design space was conducted to hit the $143MHz$ clock frequency target. Finally, the full NIOS Custom instruction was implemented and evaluated.

2.2 MATLAB simulation

To determine the optimum number of CORDIC iterations and fixed point data type parameters while fulfilling the MSE requirements, CORDIC was implemented from scratch in MATLAB. The MATLAB fixed point number toolbox and its type `fi` was used for the fixed-point number representation and operations. In our vectorized implementation, a vector of signs was first calculated using expressing the pseudorotation direction for each stage. The pseudorotations were then applied to the initial factor K and outputs x and y were calculated. This MATLAB simulation was implemented with flexibility in mind with an adjustable number of CORDIC iterations and a variable fractional length.

2.3 Design space exploration

The design space was parametrised in two dimensions, with the fixed point fraction length and the number of CORDIC stages being explored.
A Monte Carlo simulation was used, with 1000 samples per design space point, to determine the upper bound of the 95% MSE confidence interval. 1000 samples are expected to be more than sufficient for the exploitation of the central limit theorem in the following analysis. A single-tailed statistical test was conducted to estimate the upper boundary of the likely MSE with a probability of 0.95. The expression used to calculate this upper boundary can be seen in Eq. 1.
The coefficient 1.68 comes from the Z value of the standard normal such that $P(Z < 1.68) = 0.95$. This relaxes the requirements of the two-tailed test shown in the lectures, since we do not care about the lower tail of the MSE distribution. The MSE being significantly lower than the sample mean is acceptable.

$$T_{upper} = MSE + 1.68\frac{std(e^2)}{\sqrt{N}} \quad (1)$$

For our optimisation, the cost function is the resource utilization subject to the accuracy constraints. A simple resource model was devised for the CORDIC block. As an approximation, it was assumed that the resources of a single CORDIC stage are entirely determined by its adders-subtractors. Fixed point shifts can be approximated as zero cost as they require no logic. In the Cyclone V ALM, there is a flip flop corresponding to each hardware adder, so the registering of pipeline stages is also assumed to have zero resource cost, as otherwise these flip flops would merely be bypassed. For any fully unrolled CORDIC implementation, the tangents of the angle steps for each stage are constants, presumably hard-coded within the LUTs of the existing ALMs at little cost. The extraction of the sign of a number is also zero cost as merely the MSB is taken. Hence it believed that this approximation is warranted.
The problem becomes more complicated for time-multiplexed CORDIC hardware. Additional control logic is introduced in the form of multiplexers to direct the flow of data repeatedly back into the hardware block. It is expected that this mechanism introduces a term into

the resource modelling equation which grows linearly with the increased data type width, as more logic is necessary to switch wider inputs, but which is constant with the number of times that the hardware is re-used as it is only instantiated once. The applied angle constants have to be changed between cycles to reconfigure the effective stage of the block. This requires an array of memory to be synthesized as an input to each CORDIC stage to hold and switch these values. The size of this memory is expected to be proportional to the word length, as an increased bit count requires a linear increase in the memory size. This term is also expected to be proportional to the number of stages as this determines the number of coefficients stored. Overall, the contribution of these structures is expected to be relatively small and their analysis was not attempted further due to the difficulty of extracting the exact synthesis details from Quartus which are variable.

Having concluded that the adder-subtractors were going to be the main contributor to resource usage, the relationship between the width of an adder-subtractor and its resource usage was characterised. Fig. 4 shows the results of an experimient confirming the linear relationship between the width of the adder subtractor and its resource usage. The slope is 0.51, close to the theoretical value of 0.5, as each ALM contains two 1bit-adders.

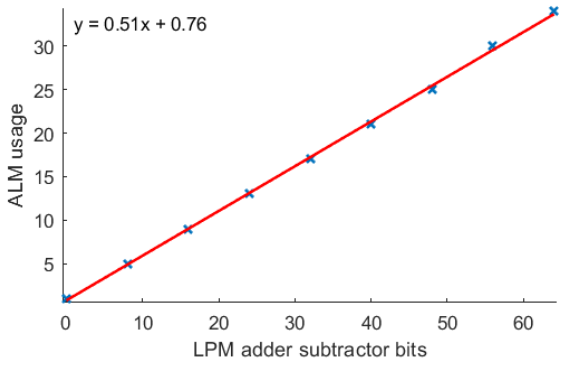

Fig. 4: Adder model data collection.

Based on the bit widths of the data type and the number of stages, the total number of single-bit adders required can be estimated. This was then used to calculate the total ALMs required and based on the provided resource metric formula the total resource value was estimated as shown in eq. (2).

$$R = \frac{1}{3}\left(\frac{3 * 0.51 * Stages * Width}{MAX\_FPGA\_ALUT}\right) \tag{2}$$

With this metric in mind, a grid search of the design space was conducted, with the number of stages varied from 10 to 20 and the fractional width of the fixed point data type varied from 16 to 32. Fig. 5 shows the relationship between the upper bound of the MSE as a function of the fractional part and the number of CORDIC stages. As expected, the error decreases with an increase in the word length and the number of stages. It can be seen that the dependence of error on fraction length plateaus for any given number of CORDIC stages and vice versa as the error of either parameter begins to dominate.

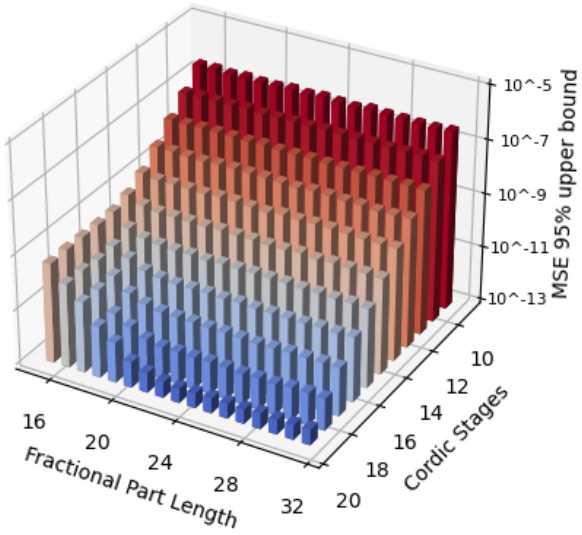

Fig. 5: CORDIC MSE upper bound as a function of stages and fractional length

Fig. 6 shows the values of the objective function (resource metric) for each explored CORDIC design point as shown in eq 2. Configurations that satisfy the 95% mean squared error constraint, are highlighted in green, with the remainder in red. The minimum of this constrained function within the window is at 16 stages, and 19 fractional bits (21-bit word). Two integer bits were selected such that the type can hold values in the range $[-2, 2)$ this is required to prevent overflow as for inputs close to 0, intermediate CORDIC values were prone to overshooting 1 as they converge on the final value. Therefore this 16 stage 21-bit word configuration was chosen for implementation and further experimentation. The window of combinations is believed to cover all the relevant combinations as additional design points with lower stage count or fractional part length do not meet the error constraint, while larger values use more resources than the found optimum.

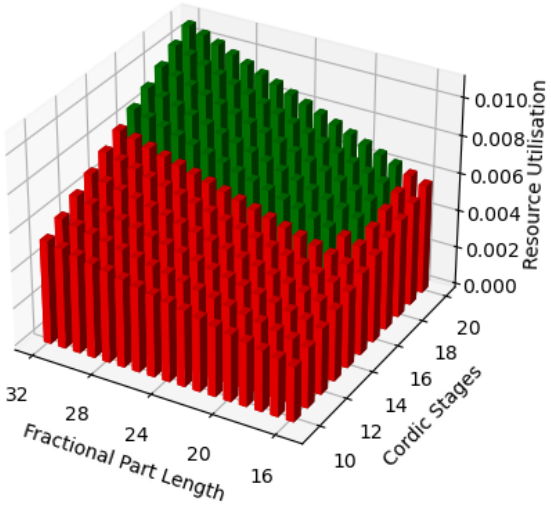

Fig. 6: CORDIC constrained minimisation

In practice, the model resource utilisation model had its limitations. For the full CORDIC block, the ratio of average ALUTs per ALM was often lower than

the the 2 encountered for the small adder-subtractor characterisation designs. It is believed that this points to the difficulty of effective resource utilisation during placement for large designs, especially under tight timing requirements.

CORDIC designs re-using hardware (explored more later) were used to validate the model. These designs were used to approximate a variable number of sequential CORDIC stages. Fig. 7 shows the comparison between the synthesised ALMs and the predictions of the model. The plot shows a good correlation between the model and the real resource usage. A systematic error can be seen, which is likely the result of the instantiated control hardware not modeled by the model. The initial non-linearity is believed to be the result of the aforementioned fitting difficulties. While not exact, the model was suitable to orient decision-making.

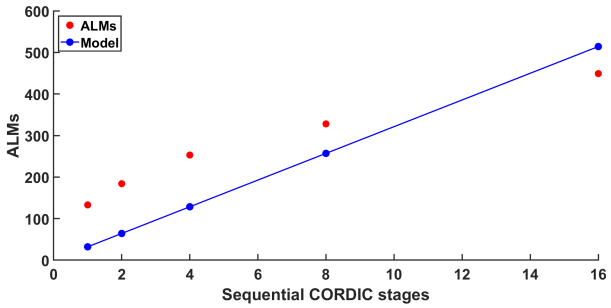

Fig. 7: Comparison of the hardware resource usage and the model predictions.

2.4 CORDIC term block

For the sake of ease of testing and re-usability, the full CORDIC hardware implementation was broken up into CORDIC term blocks which formed the base unit of the design.

Fig. 8 shows the architectural overview of the `CordicTerm` block. Inputs include `x_in`, `y_in`, `input_angle`, `term_angle` and `term_order` which was set either as a Verilog parameter or as an input, depending on if hardware was re-used. First, the `CordicTerm` block determines the value of `d` based on the sign of the input angle error, which then controls whether the `AddSubFi` blocks should perform addition or subtraction. `RightShiftFI` block performs the shift of `x_in` and `y_in`. For the CORDIC architecture that does not reuse hardware, the `shift_amount` was set through a Verilog parameter to reduce the hardware footprint whereas the implementation of the CORDIC reusing the `CordicTerm` block required the `shift_amount` to be variable, presumably resulting in a barrel shifter after synthesis. The outputs from the shift units are then passed to the `AddSubFI` block, which calculates the outputted final values. This whole block performs all of the operations using only the combinational logic. Subblocks within the design such as `AddSubFI` are re-used.

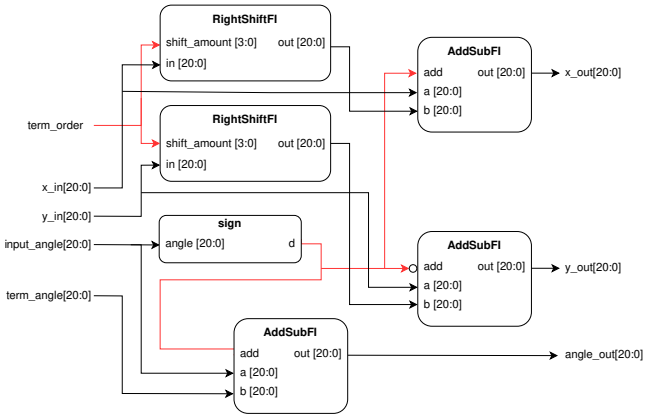

Fig. 8: CORDIC Term block achitecture.

2.5 Pipelining CORDIC terms

Driven by the 16-stage, 19 fractional bits requirement determined by design space exploration, the first explored CORDIC architecture involved connecting 16 `CordicTerm` blocks in series without any pipelining registers. This architecture was used mainly to verify the correctness of the block's output, but it was not used with the NIOS system due to an unfeasibly long critical path.
Fig. 9 shows the overview of the unrolled pipelined architecture, which utilises pipelining registers periodically inserted between `CordicTerm` blocks. The number of `CordicTerm` blocks between the two consecutive pipelining registers was varied to investigate its effect on the performance characteristics of the block.

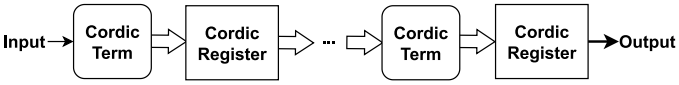

Fig. 9: CORDIC Term block

Table 2 shows timing analysis results and the resource usage of different unrolled designs with a varying number of combinatorial `CordicTerm` blocks between each pair of registers. The table shows the trend that a decrease in the number of `CordicTerm` blocks between the two registers leads to a higher maximum frequency. The synthesized ALUT count remains approximately constant as the combinatorial logic structure between registers remains unchanged. An increase in the number of pipeline stages results in an increased register count. This however has little impact on the ALM count as each ALM already contains a register per ALUT, which just commonly isn't used. It can be seen that from this family of designs, only configurations with one and two CORDIC stages between each register pair reaches the F_{max} of target of 143MHz.

Serial CORDIC blocks	F_{max} [MHz]	ALUT count	ALMs	Register count
16	27.00	919	462	0
8	50.82	920	465	63
4	91.89	920	473	230
2	170.59	957	506	402
1	286.12	929	525	856

Table 2: Performance and resources of unrolled CORDIC.

2.6 Sharing hardware resources

The sharing of hardware resources between CORDIC terms was investigated. Fig. 10 shows the high-level diagram of an architecture that shares the `CordicTerm` blocks for the calculation of different effective stages. A different number of combinatorial `CordicTerm` blocks ranging from one to eight was connected to the `CordicRegister` that store calculation values between hardware reuse iterations. During the first run, initial values are inputted to the first `CordicTerm` block. During the following iterations of the algorithm, the multiplexer passes the intermediate results of the algorithm back to the combinatorial `CORDICTerm` blocks. The internal logic of the top level CORDIC block determines which term angle should be connected to each `CordicTerm` block during different cycle of the algorithm.

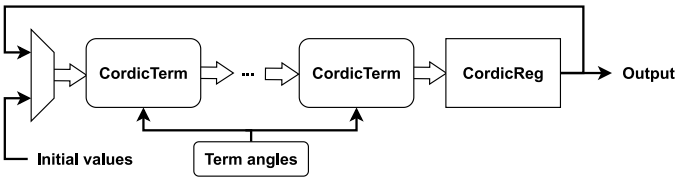

Fig. 10: CORDIC Terms sharing resources.

Table 3 shows the performance and the resource utilisation of hardware reusing CORDIC. The F_{max} of the architectures with 16, 8 and 4 `CordicTerm` blocks in series shows a similar trend compared to the pipelined architecture. Compared to the unrolled CORDIC family of designs, the achieved F_{max} for any number of serial stages was worse. It is believed that this is due to the additional control logic in series with the CORDIC term blocks increasing the critical path. The main advantage of this architecture compared to the pipelined architecture is the significantly lower number of resources used, with the folding factor being approximately inversely proportional to the resource requirements when neglecting the fixed size control logic. The folded design with one`CordicTerm` block uses 7.82 times fewer ALUTs (86% decrease) and 6.25 times less registers (84%) compared to an unrolled implementation.

CORDIC blocks	F_{max} [MHz]	ALUT count	ALMs	Register count
16	16.74	895	449	20
8	34.65	551	328	87
4	76.35	429	253	95
2	118.82	301	184	110
1	179.31	183	133	148

Table 3: Performance of the iterative CORDIC.

2.7 Latency analysis

The effect of different above documented design approaches on the latency of the CORDIC block was investigated. The following analysis is expressed in terms of CORDIC stages, where this corresponds to the number of pipeline stages for unrolled designs and the folding factor of time multiplexed designs. Based on the achieved F_{max} and properties for each of our CORDIC design families, the latency in absolute time can be expressed according to Eq. (3) for folded CORDIC and according to Eq. (4) for unrolled CORDIC.

$$Latency_{HWreuse} = \frac{16}{F_{max} \cdot Series\ CordicTerm\ blocks} \tag{3}$$

$$Latency_{Pipelined} = \frac{Number\ of\ pipeline\ stages}{F_{max}} \tag{4}$$

Fig. 11 shows a plot of the absolute time latency for each CORDIC family as a function of the number of stages. The plot confirms that in terms of pure latency, a single stage design has the lowest latency. We also note that for both unrolled and folded families, reducing the number of stages to 1 arrives at a near identical latency. Indeed these are defacto identical fully combinatorial designs. The plot shows an approximately linear increase in the latency of the unrolled design as the number of CORDIC stages increases. This per stage increase is likely due to the setup and hold times and control logic for each added pipelining register. In the case of the hardware that reuses `CordicTerm` blocks, the slope of the latency is more significant. This is likely due to the fact that this design contains additional control logic such as the multiplexer selecting the input or intermediate result to the first `CordicTerm` block or logic selecting the appropriate angles for each `CordicTerm` block during different iterations. This more complicated control hardware introduces a larger per cycle latency as the data propagates through compared to unrolled realisations of the design.

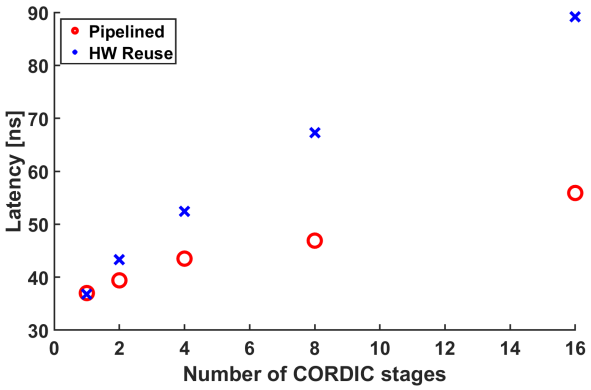

Fig. 11: Latency vs CORDIC stages.

2.8 Throughput analysis

Equally, based on the F_{max} of our block and its stage properties, the throughput in terms of the number of cosine evaluations per unit second can be analysed. Under the assumption of steady state, that is neglecting pipeline fills and flushes, the throughput of the folded CORDIC family can be expressed as Eq. (5). The same can be done for the unrolled CORDIC family with Eq. (6).
Fig. 12 shows the relationship between throughput for different numbers of CORDIC stages for each family of designs. The plot shows that the throughput of the pipelined designs increased nearly linearly as the number of registers was increased. The relationship however is not exactly linear with a small downward deviation, due to the nonlinearity of the relationship between the number of

hardware stages and F_{max}. Here the increase is less than proportional due to the added constant delay introduced by the setup and hold times of the pipelining registers.
On the other hand, the throughput of the folded hardware decreases with the increasing number of `CordicTerm` blocks increases. An inversely proportional relationship is predicted and consistent with the shape of the curve seen. As the number of CORDIC stages increases, the computation is divided across more and more cycles, with the theoretical throughput decreasing by $\frac{1}{Nstages}$. As the number of stages increases, the contribution of the control logic and register setup and hold times begins to contribute significantly.

$$Throughput_{HWreuse} = F_{max} \cdot \frac{Series\ CORDICTerm\ blocks}{16} \tag{5}$$

$$Throughput_{Pipelined} = F_{max} \tag{6}$$

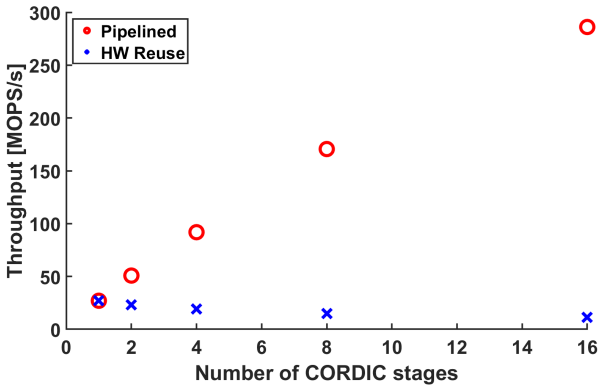

Fig. 12: Throughput vs CORDIC stages.

2.9 CORDIC testing

Initially, separate test benches were prepared for each component of the `CordicTerm` block to verify the correctness of each block on a curated test verifying key data points such as the domain extrema. These sub-blocks were then integrated into the `CordicTerm` and the correctness of calculations of this block was further verified by comparing the outputs of the block with the MATLAB implementation of the algorithm. Before the final `CORDIC` block was integrated into the final system, extensive testing was conducted. The testing process involved first randomly generating N angles in the range from -1 to 1 with a uniform probability distribution in MATLAB and saving them as a fixed-point representation into a text file. The inputs stored in this file were then read by the Verilog testbench using `$fopen` and `$fscanf` function and fed into the tested `CORDIC` block. The calculated cosine from the `CORDIC` block was written to the text file using `$fwrite`. The file with outputs was then read by another MATLAB script which compared the calculated cosine with the double-precision reference value. According to this test, the hardware implementation achieved MSE of $1.00E{-}10$ for 10000 samples. This is close to the value $9.675e-11$ predicted by the smaller Monte Carlo simulation, based on which the number of stages and fractional bits was specified. This deviation could feasibly be explained by the random nature of the sampling, but in retrospect the likely cause are the default nearest rounding for MATLAB fixed point operations, where as the Verilog implementation relied on truncation.

2.10 Task 7 hardware

For task 7 hardware, a custom NIOS instruction was developed to calculate the inner expression shown in Eq. (7). It relies on the Altera FP library of IP blocks for addition, subtraction, multiplication and fixed to floating point conversions.

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

The weakness of the NIOS custom instructions is that they are not pipelined, the processor waits until a value fully propagates through the instruction hardware before supplying another piece of data. Fig. 13 shows the instruction waveform expressing this fact with the illustration taken from the NIOS custom instruction reference manual. Once again, the chosen type of instruction was variable multicycle. While not strictly necessary for this fixed duration computation, this type of instruction lends itself to easier debugging, as it is easy to verify in simulation that the generation of the output data occurs in the same cycle as the done pulse.

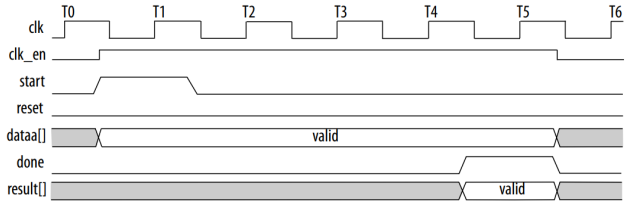

Fig. 13: NIOS multicycle instruction.

This single value constraint can be exploited to decrease the resource usage by time multiplexing the hardware resources. Naively, the full expression could be constructed as a dataflow graph involving 4 multipliers and 2 addition/subtraction blocks alongside the CORDIC and fixed point conversion hardware. Instead, a hardware schedule is presented in Figure 14 does not increase the instruction latency in cycles, while only requiring one block per operation type. When constructing this schedule, the longest series of data dependencies in the expression was prioritised when allocating time slots for computation. The remaining shorter paths were computed during periods when the given block was unused. Ultimately, this critical path remained the longest series of dependent computations. This means that it is no slower than a fully pipelined dataflow implementation of the same expression given the constraints. Note that due to the pipelined nature of the individual blocks, it was possible to have multiple calculations in flight within a given block, just that they can not begin at the same time.
Similarly, a fully unrolled CORDIC block was not necessary as only one value is in flight. Therefore a block with hardware reuse was chosen, such that the latency is the lowest while meeting the required F_{max}. This resulted in the selection of a folded design with only CORDIC term in series.

The schedule can be further analysed in terms of the average block utilisation. For a dataflow schedule, this would be $\frac{1}{54} \approx 0.016$ as each block would only be used once in the 54 cycles of the instruction being executed. For the proposed time multiplexed schedule, this metric is $\frac{9}{54*5} \approx 0.033$. which approximately doubles the utilisation of the hardware resources, but remains suboptimal, with up to a theoretical 30-fold increase (assuming equal block load) possible with the same hardware.

Fig. 15 describes the structure of the hardware implementation of the proposed schedule. A number of constants and registers were maintained. These were switched onto the buses within the block during the appropriate cycles, capturing and releasing intermediate results. Note that for computations on the critical path, outputs were not explicitly registered, but forwarded directly to the following block for the sake of latency. This works because based on the experimentation with the design in the timing analyser, it was concluded that the FP IP block outputs were registered, as the insertion of registers between these blocks did not improve the critical path. Instead the timing requirements were met by adjusting the number of pipeline stages for each of the IP blocks until a 143MHz clock was achieved.

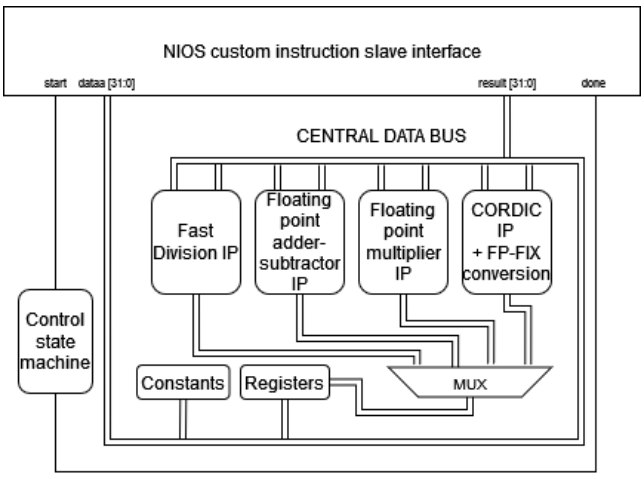

Fig. 15: Task 7 hardware diagram.

Reviewing the initial design, a few opportunities for improvements were seen, even within the constraints of the NIOS instruction framework. One is the reduction in latency for the division of a floating point number by a power of two. For the given use case and accuracy targets, it is not strictly necessary to handle the full IEEE 754 specification. Shortcuts can be taken on matters such as rounding or denormal numbers without significantly affecting the accuracy of the system as a whole, while reducing latency. Previously, scaling by a factor of $\frac{1}{128}$ was conducted over the course of 5 cycles by a multiplier. Instead, by exploiting the single precision format visible shown in Eq. (8), the mantissa can be decremented by $log2(divisor)$. The one condition that was guarded against, was mantissa underflow. Should the mantissa underflow during decrementing of the exponent, the result is clamped to 0, neglecting denormal numbers. This reduced the latency of the critical path by 4 cycles, with an improvement of 6.3% in terms of latency for test case 3.

$$F_{single} = -1^{sign} * 1.mantissa * 2^{exponent-127} \quad (8)$$

Given more time, the development of a custom floating point to fixed point and back converter set would have been undertaken, as a latency of 6 cycles for each of the provided ones appears excessive. For the use case, it would be sufficient for a handful of 8-bit arithmetic operations to condition the exponent in one cycle, for a barrel shifter to deliver the fixed point number in the second cycle, with negation occurring in a third cycle. It is possible that some of these stages could be combined while still meeting timing requirements. Hence even for a pessimistic three pipeline stages, an additional 9.2% improvement in latency would be expected.

2.11 Performance

An analysis in terms of error, performance and resource utilization in the context of task 6 hardware was conducted. Table 4 shows a near identical relative error for test case 3 compared to the task 6 implementation. An improvement in the relative error by two orders of magnitude for test cases 1 and 2 was noted. This is because, unlike the look-up table implementation in task 6, the calculation of cosine with CORDIC does not suffer from the same issues of sampling the cosine table at the point of maximum error when the number of linearly spaced samples is a factor of the lookup table entry count.

System	Test 1 [%]	Test 2 [%]	Test 3 [%]
Task 6	-1.564E-02	-1.520E-02	6.233E-04
Task 7	5.502E-04	-4.706E-04	6.344E-04
Task 7 fastdiv	-7.811E-04	-5.038E-04	6.122E-04

Table 4: Relative error comparison.

The latency of the custom instruction did not improve significantly - only a 26% improvement between the results for the best Task 7 implementation and the best Task 6 implementation. This is however unsurprising. A majority of the operations in the inner expression have data dependencies, so their execution in parallel does not present large improvements compared to their serial execution. It was also noted that the task 6 lookup table implementation, especially when also accelerated with the floating point instruction hardware, is expected to be comparable to the 28 cycles required to take the floating point number and calculate its floating point cosine representation with CORDIC and the type converters.

System	Test 1 [μs]	Test 2 [μs]	Test 3 [μs]	Resources [%]
Task 6	52	1571	185564	4.05
Task 7	29	1150	146490	4.84
Task 7 fastdiv	27	1080	137350	4.84

Table 5: Latency of different version of the system.

Table 6 shows the comparison of FPGA resource utilization for Task 6 and Task 7. The number of resources consumed increased for Task 7 because the custom hardware block was integrated into the design. The resource utilisation for the design with the aditional `fastdiv` hardware was marginally lower.

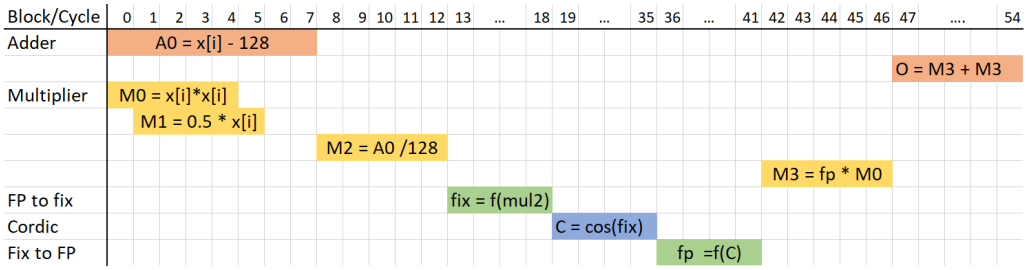

Fig. 14: Task 7 hardware schedule

Hardware rev	ALM	Bram	DSP	Resources [%]
Task 6	2,148	34,093	4	4.05
Task 7	2,915	34,283	4	4.84
Task 7 fastdiv	2,910	34,283	4	4.84

Table 6: Resource utilization as a function Task.

2.12 Conclusion

A hardware accelerator calculating the inner part of the given sum was implemented. This block utilises a custom-made CORDIC block that reuses its hardware during the calculation of the algorithm. There are still remaining areas that could be investigated. For example, per-stage variable widths were not investigated due to time constraints and optimisation difficulty. Another area for further investigation is an opportunity for reusing the hardware (`AddSubFI` and `RightShiftFI`) inside the `CordicTerm` block. This version of the design was not investigated, because it requires additional control logic and registers to store intermediate operands. This would require significant changes in the clocking of the whole CORDIC block. A possible improvement of the implemented design would be to add one more fractional bit to the fixed point input of the `CordicTerm` to increase the margin for MSE error.

3 Task 8 - Custom Hardware

The goal of Task 8 was to map the whole function `sumVector` into the hardware to further reduce the execution time. The biggest downside compared to the approach undertaken in task 7 was the fact that only one unit of data was in flight within the hardware at a time. Two approaches were devised to bypass this limitation. In the first approach, the constructed hardware allows multiple values to be in flight by creative use of NIOS custom instructions. In the second, an architecture reliant on DMA to feed the input array into the block was designed and tested.

3.1 Approach 1: NIOS Instruction driven pipeline

Even within the NIOS instruction framework, the pipelined nature of the hardware can be exploited. Fig. 16 shows a block diagram of the proposed block. At the core of the block is a dataflow graph expressing the "inner expression" consisting mainly of Intel floating point IP blocks with custom fully unrolled 8-register stage CORDIC block. The fully unrolled 8-register stage CORDIC block was chosen as this realization of the architecture has the lowest latency while meeting the 143MHz frequency target. The number of register stages along each path of the dataflow graph was balanced using shift registers, delaying the input value by the appropriate number of cycles to match the delay introduced by the longest dependent computation path.
This inner expression hardware is able to generate the correct output with a latency of 42 cycles. Afterwards, an Intel accumulator IP block was used to sum the computed inner expression terms.
The control logic is what allows for multiple values to be in flight simultaneously. The clock for the dataflow hardware is kept constantly running, with a shift register used to keep track of the data validity in the pipeline. The input to the pipeline is only valid during the first cycle of a `Push Input` instruction. Otherwise, the data must be ignored by the accumulator once it propagates through the system.

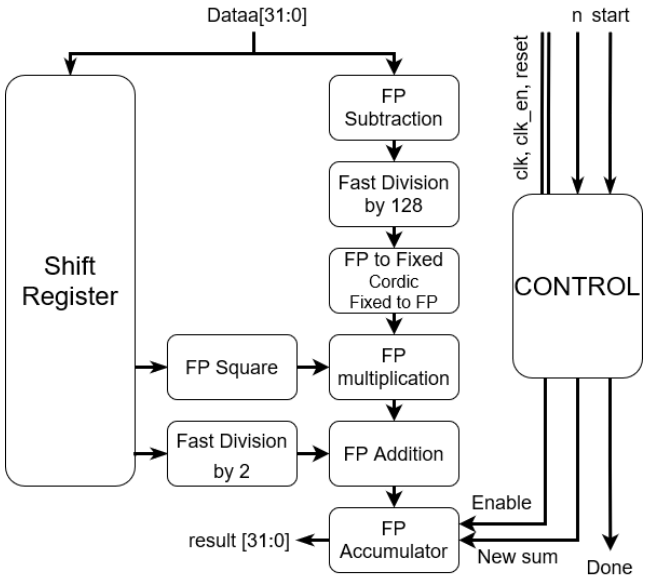

Fig. 16: Proposed Task 8 NIOS architecture.

The block has a NIOS custom instruction slave interface with a two-bit opcode. A variable duration multicycle instruction interface was used. Table 7 shows the instructions used to interface with the block.

Instruction	Opcode	Input Value	Return value	Latency
Clear Sum	0	N/A	Non-blocking read	1
Read status	1	N/A	Accumulator status flags	1
Push input	2	x[i]	Non-blocking read	1
Read Sum	3	N/A	Blocking read	N

Table 7: Instructions supported by Task 8 NIOS hardware

The `Clear sum` instruction causes the control logic to clear the FP accumulator by signalling to it the start of a new accumulation series and feeding the initial value of 0 in.

The accumulator block is internally significantly different to a floating point adder. When generating the IP, there was a need to specify the maxMSBX (Bits of magnitude per input), MSBA (Maximum bits of accumulator magnitude), and LSBA (Accumulator fractional bits). These parameters indicate that internally the accumulator operates in a fixed point mode as this makes it possible to complete an addition in a single cycle, which simplifies the design of the accumulator.

The maxMSBX parameter, which specifies maximum magnitude of a single input to the accumulator, was obtained by numerically determining the maximum value of the inner expression on the domain [0, 255]. This yields a per term maximum magnitude of ca. 37,146, whose $\lceil log_2(37,146) \rceil = 16$. Hence maxMSBX was set to 16 bits.

The MSBA parameter which specifies the maximum accumulator output magnitude was conservatively determined by taking the numerical maximum of the function in its domain and multiplying it by the maximum number of terms expected to be summed. Taking the number of terms in test case 3 this yields $\lceil log_2(37,146 * 261121) \rceil = 34$. Hence MSBA was set to 34.

There is no equally straightforward way to determine the LSBA accumulator fractional length as it is highly data-dependent. It is a valid operation to to supply an array of a single small input, resulting in a near arbitrariily small value in the accumulator. The target was no loss of precision compared to floating point accumulation for the test case 1. The reference result for this calculation is 920408.5625 with $\lceil log_2(920408.5625) \rceil = 20$. Therefore the most significant magnitude bit will be on the 20th integer position, so the least significant bit will be in the 4th fractional position. Test case 1 contains 56 inner terms. For the least significant accumulation bit to be unaffected, it needs to have a weight greater than 56 times the ULP of the accumulation (Under the worst-case assumption of truncation). By rounding this up to 64, it can be concluded that an additional 6 fractional bits are necessary in order for the error not to propagate into the result. This gives an LSBA requirement of 10 bits.

The accumulator IP provides a set of flags which are asserted in case any of the above assertions do not hold and any of the accumulator capabilities are exceeded. These flags are made available through the Read status opcode, which when read after computation is complete, provide information about whether and how the hardware capabilities were exceeded. This shows whether the results can be trusted.

Inputs can be pushed into the dataflow graph over the course of one cycle with the `Push input` instruction. This instruction takes one floating-point argument.

The real reason why a variable duration instruction was used is the `Read Sum` opcode. On top of the per-cycle validity bits, a saturating count of the cycles since the last data push is maintained by the hardware block. This allows us to delay the return of the accumulated sum by a variable number of cycles to make sure that the done flag is set and the result provided only once the final piece of data has fully propagated through the system and into the accumulator. This works around an interrupt system which might have otherwise been used for this purpose.

3.2 Performance

The performance of the design system was tested and compared with the previous tasks.
Table 8 shows the relative error in percent as a function of the hardware configuration and the test case. It can be seen that no appreciable increase in the total error for the test cases took place, validating the choice of accumulator parameters.

System	Test 1 [%]	Test 2 [%]	Test 3 [%]
Task 7 fastdiv	-7.811E-04	-5.038E-04	6.122E-04
Task 8 NIOS/f	5.503E-04	4.042E-04	3.295E-04

Table 8: Relative Error

Table 10 shows the improvement in performance compared to Task 7 hardware. The result is a 50% decrease in latency compared to only one value in flight at a time. This is a little underwhelming, as it suggests that only about two values are in flight in the hardware at a time. Given the single cycle duration of the push instruction, this points to a bottleneck with the memory system. The result can be partially be explained by the cache architecture. By consulting the NIOS processor reference handbook, it was determined that the cache line size is only 32 bytes. Therefore after every 8 floats pushed to the hardware, a number of pipelined requests to the SDRAM need to be made to service the resulting data cache miss. These were expected to take a significant amount of time. When switching the processor type to e, there appears to be a difference in the custom instruction port which results in to result being read. The timing should be unaffected and is included for the sake of completeness below.

System	Test 1 [μs]	Test 2 [μs]	Test 3 [μs]	Resources
Task 7 fastdiv	27	1080	137350	4.84
Task 8 NIOS/e	26	1732	65977	3.88
Task 8 NIOS/f	3	178	22186	4.97

Table 9: Comparison of Task 8 implementation performance

3.3 Approach 2: DMA

The second executed approach was the use of a Direct Memory Access (DMA) block that reads the data from the memory and inputs them into the `Full-IP` block that performs the calculation of the whole sum. Fig. 17 shows the proposed architecture.

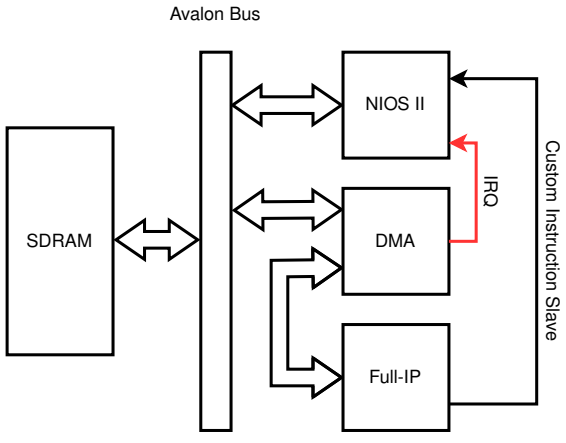

Fig. 17: Proposed DMA architecture integrating Full-IP block.

The DMA block triggers an interrupt when the calculation of the result is finished. The interrupt trigger is sent through the IRQ line and to the `NIOS II` processor. This signal then triggers the appropriate software interrupt service routine. In this routine, the CPU reads the result from the `Full-IP` block. The valid flags from Task 7 are kept, and the variable duration instruction returns only once the final piece of data propagates to the accumulator. The developed full IP block implements a minimal Avalon MM interface with facilities to handle writing data. The design evolved from the Task 8 NIOS implementation. A few alterations in hardware were required. The instantiated dataflow graph, was fed from the Avalon MM interface, with the write line being read to determine if the data at the input of the block is valid. This data validity per pipeline stage was once again maintained in a shift register, so that only valid data is accumulated.

3.4 Performance

In terms of the error, the instantiated datapath is identical to the datapath of the Task 8 NIOS realisation, so the error was identical.

Table 10 shows an improvement in performance by approximately a factor of 4 compared to the previous best, the custom instruction-driven pipeline. This is the best design that was achieved throughout the project. Again, a NIOSe-based system was tested, even if the result port gets synthesized away since the control logic remains in place and the latency of the system is expected to be representative of a system where this bug has been resolved. The test case latencies are nearly identical between the NIOS/f and NIOS/e implementations. This is expected, as the processor itself does a minimal number of computations in the timed window, with the limiting factor being the performance of the DMA block, which is entirely separate of the processor. Therefore, the NIOS/e design would be deployed as it uses 18% fewer resources for the same performance.

System	Test 1 [μs]	Test 2 [μs]	Test 3 [μs]	Resources
Task 8 NIOS/f	3	178	22186	4.97
Task 8 DMA/e	5	50	5218	4.09
Task 8 DMA/f	9	50	5227	4.98

Table 10: Comparison of Task 8 implementation performance

For our design, 5218 μs is 747,461 clock cycles. Test case 3 has 261121 elements. For test case 3 we can approximate the system to be in steady state, as the pipeline flush of 48 cycles, timing and interrupt overhead are expected to be in the order of unit microseconds based on the performance of test case 1. We can therefore neglect this. This gives us an average of 2.86 cycles per data element. This is within 30% of 2 cycles per data element, which is the theoretical maximum of the system. It is bounded by the 16bit width of the SDRAM, which takes 2 cycles to transfer a 32bit float. There is expected to be some overhead in this process. The memory needs to be periodically refreshed, during which time it is not available. It is also possible that the FIFO depth of 32 for our DMA is insufficient to hide the round trip time to memory, as the memory bus passes through multiple adapters, which add to the latency of any read. This final possible cause was however left un-investigated due to time constraints.

Overall, the proximity of our design to the theoretical maximum is considered satisfactory for the time being.

4 Further work

There are a number of avenues for improvement and further development.

Radix-4 CORDIC could be investigated. The main advantage is the reduced number of stages to achieve a given accuracy, which would be beneficial for latency-sensitive use cases, and hardware architectures such as Task 7

The NIOS/e processor result port removal bug should be investigated as it is likely caused by an adapter which was not inserted correctly by the Platform Designer, but this problem was not resolved due to time constraints. Similarly, the DMA FIFO depth relationship should be explored as it would likely bring the design closer to the Pareto front.

The main avenue for further performance gains would likely come from introducing a degree of parallelism to the system. Multiple DMA-driven blocks could be instantiated, as the FPGA has plentiful remaining resources. The issue would then become memory bandwidth. Our single DMA block was already memory bandwidth limited. For small data sizes, it may be plausible to instantiate one on-chip memory block per DMA-driven block, which would likely result in performance close to the theoretical 1 cycle per data element per DMA-driven block. The memory bus network would have to be architected in such a way as to avoid congestion by providing direct paths between a DMA block and its respective memory. For larger test cases, exploiting the DDR3 memory attached to the FPGA would be desirable. The two IS43TR16256A chips are capable of operating at up to 1066MHz (2133 MT/s) each at a width of 16 bits. This would raise the theoretical maximum computation bandwidth from 286MB/s to 8.5GB/s approximately a factor of 30.

5 Conclusion

Fig. 18 shows the comparison of all designed systems throughout the whole coursework. Due to a large difference in the latency of the different systems, log scaling was introduced. The plot suggests, almost all designs apart from the design with DMA implementation are very suboptimal because these designs do not form a Pareto front.

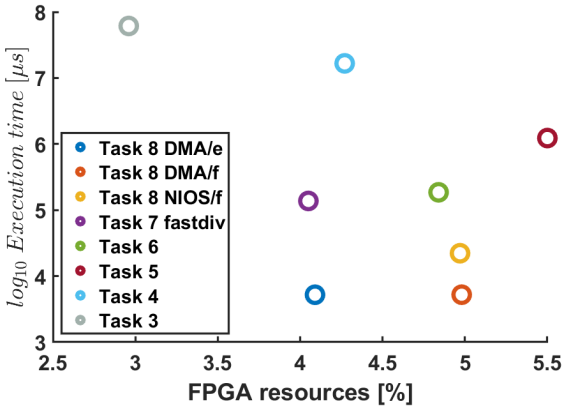

Fig. 18: Final systems comparison.