Digital Systems Design Coursework - Report 3

Group 3

1st Kwok Tsz Wing
01708991
twk119@ic.ac.uk

2nd Josiah Mendes
01760165
jam419@ic.ac.uk

I. Introduction

The following report discusses Task 6 to Task 8 assigned in the coursework specification for Digital Systems Design. The following investigation was carried out using Intel Quartus Lite 20.1, on a Terasic DE-1 SoC board with a Cyclone V FPGA.

Previous work [1] [2] discussed a software implementation of Equation 1 for 3 different cases listed in Table I.

$$f(x) = \sum_{i=1}^{N} 0.5x_i + x_i^2 \times \cos(\frac{x-128}{128}) \qquad (1)$$

TABLE I
VECTOR SIZE IN EACH CASE

		Number of Elements	Size (Bytes)
Case 1	[0:5:255]	52	208
Case 2	[0:1/8:255]	2041	20164
Case 3	[0:1/1024:255]	261121	1044484

An optimised program written in C with a 4-term Taylor Series cos implementation was used to calculate the result in each case on a Nios II processor. The Taylor series approximation was used in order to take advantage of the integer multipliers within the processor. As the Nios II processor has no floating point hardware, all floating point operations are implemented using software routines.

II. Task 6 - Custom Floating Point Instructions

It was noted that a significant performance uplift could be obtained by moving the floating point operations to hardware. Therefore, the following section describes the addition of floating point hardware to the system through the use of custom instructions and the resultant performance uplift.

A. Floating Point Hardware Blocks

Quartus provides as part of its IP library parametric arithmetic floating point hardware blocks [3]. These blocks can be customised for latency or with a target frequency, and as they are verified blocks, it was decided that they should be used rather than a custom implementation. As according to the instructions, floating point blocks for multiplication, addition and subtraction were added to the system.

Each block's latency was chosen individually to minimise the latency of the block while meeting the current timing constraints of the system running at 50MHz. As these floating point blocks are going to be executed through the use of custom instructions, the Nios II's inability to issue multiple instructions means that the latency of the block is of higher importance than throughput.

For floating point multiplication, the block was configured to have a 2 cycle latency as this was the minimum possible latency. In the case of the adder and subtraction block, a block could have been added for each one, but this would be duplication of resources, as subtraction is just addition with a negated input. Therefore a combined block was used with 1 cycle latency as the combinatorial block was not able to meet timing constraints. The amount of resources consumed by each block is shown in Table II.

TABLE II
PER-BLOCK FPGA RESOURCE UTILISATION

Arithmetic Operation	Multiplier	Add & Sub
Logic utilization (in ALMs)	75	353
Total registers	32	189
Total pins	98	99
Total virtual pins	0	0
Total block memory bits	0	0
Total RAM Blocks	0	0
Total DSP Blocks	1	0
Total PLLs	0	0
Total DLLs	0	0

B. Custom Instruction Implementation

Although it would have been possible to combine all of these blocks and add them as a single component within the Platform Designer, it was deemed unnecessary as it would introduce additional unnecessary operation select logic that is already carried out by the Nios II processor. Therefore, each block was added as a separate component, as a fixed multi-cycle instruction. These were then implemented as macros within the C software side to call the hardware blocks when necessary as shown in Listing 1.

```
#define FP_ADD(A,B) __builtin_custom_fnff(0x3,A,B)
#define FP_SUB(A,B) __builtin_custom_fnff(0x2,A,B)
#define FP_MULT(A,B) __builtin_custom_fnff(0x1,A,B)
```

Listing 1: Custom Instruction Macros

Each use of addition or subtraction or multiplication was replaced with these custom floating point operators for both within the function for calculating each iteration of Equation 1 and also within the implementation of the 4-term Taylor Series. The execution time of the new system with all software optimisations are recorded in Table III. The performance uplift over the previous system and the error relative to a reference double implementation are also recorded.

TABLE III
PERFORMANCE WITH FLOATING POINT OPERATORS

	Average Time (us)	Uplift	Error
Case 1	1532.97	+51.96%	1.3760e-7
Case 2	60689.272	+50.92%	6.2143e-7
Case 3	7951508.436	+49.76%	9.3351e-6

As the case array is allocated in stack memory and is independent of the program size, the program size remains constant as 76 Kbtytes. The entire system's resource score is 6.5812% [1], showing a 0.81% increase over the previous system with no hardware floating point, with the full resource utilisation for the entire system shown in Table IV.

TABLE IV
TASK 6 SYSTEM FPGA RESOURCE UTILISATION

Logic utilization (in ALMs)	2039 / 32070 (6%)
Total registers	2955
Total pins	47/457 (10%)
Total virtual pins	0
Total block memory bits	357248 / 4065280 (5%)
Total RAM Blocks	50 / 397 (11%)
Total DSP Blocks	4 / 87 (0%)
Total PLLs	1 / 6 (0%)
Total DLLs	0 / 4 (0%)

It is observed that there is a 50% reduction in execution latency across all three cases, which is a very large improvement. The increase in resource utilisation mainly comes in more logic elements being used, but this is insignificant when compared to the total number of logic elements available on this particular FPGA. The error also remains the same when compared to the previous software implementation as the order of floating point operations has not changed, and it is expected that the software routines for floating point emulation would not introduce any errors.

Pipelining of operations is not possible at this stage due to the Nios II execution model which stalls the processor until completion, rather than issuing multiple instructions that use different blocks simultaneously to achieve instruction-level parallelism. This is something that can be exploited later on when the inner expression is completely calculated within hardware.

Additionally, as these floating point instructions are custom instructions, they are only used in the written program. Other code such as imported libraries are unable to take advantage of the new instructions.

[1]The Resource Percentage is obtained using the formula given in the specification: $\frac{1}{3}(\frac{\text{Multipliers Used}}{\text{Total Multipliers}} + \frac{\text{Memory Bits Used}}{\text{Total Memory Bits}} + \frac{\text{Logic Elements Used}}{\text{Total Logic Elements}})$

III. Task 7

Although the current cosine function is software optimised with custom floating-point instructions, it still takes up the majority of the loop body in the generated assembly code. It was thought that it would be the main bottleneck in performance improvements. Hence hardware optimisation on the cosine function is explored to further optimise the performance of the system by implementing a dedicated `CORDIC` block optimised for the system frequency of 50MHz.

As simulation and compilation in Quartus is slow, testing is done on a SystemVerilog testbench and simulated with Synopsis VCS 2020.03 before running and testing on the FPGA board. Timing analysis of each block is then evaluated by compiling the module on Quartus. This provides the latency of the critical path and the corresponding maximum frequency through the Timing Analyzer.

A. CORDIC Implementation

Listing 2 describes the implementation of the CORDIC block in quadrant 1, where input is in the range [0 - $\pi/2$]. All `int` variables represent a fixed-point number. And `K`, is the reciprocal of the gain, which is calculated by equation 2.

$$K = (\sum_{i=0}^{N} \sqrt{1 + 2^{-2i}})^{-1} \tag{2}$$

```c
int CORDIC (int angle, int count) {
 sin = 0;
 cos = K;
 for (int i=0; i<count; i++) {
  if (angle > 0) {
   sin += cos >> count;
   cos -= sin >> count;
   angle -= atan(pow(2,-count));
  } else {
   sin -= cos >> count;
   cos += sin >> count;
   angle += atan(pow(2,-count));
  }
 }
return cos;
}
```

Listing 2: CORDIC Implementation in C

It is implemented as a fully folded architecture. Each `for` loop in listing 2 is executed in one clock cycle. Figure 1 shows an overview of the overall architecture of the module. The `Counter Comparator` checks the condition of the loop and asserts a valid signal when output data is ready.

1) Fixed Point Representation: From Equation 1, it is known that the input is the range of [-1,1], and the cosine function is an even function. For input in the range [-1,0], the output should be the same as its negated value. Hence, the CORDIC block input can be limited in the range [0,1]. Thus, the output would also be limited in the range [0,1].

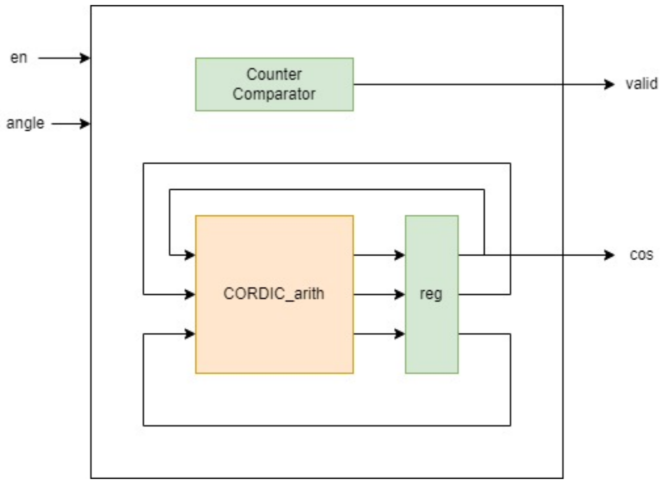

Fig. 1. Overall Architecture of folded CORDIC module

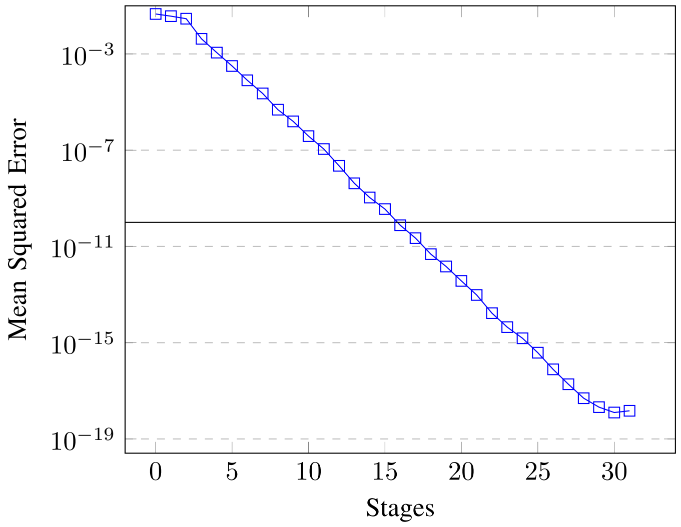

Fig. 2. Effect of Number of Stages on Accuracy

Given that both the input and output is in the range [0,1], the fixed-point number is chosen to have 31 bit of fraction which could represent numbers in range [0,1]. A higher resolution of the fraction allows a higher accuracy. Table V shows examples of the conversion between floating point to fixed point representation.

TABLE V
FLOATING POINT TO FIXED POINT

Decimal	Fixed Point expressed as 32-bit Hexadecimal
1.0	0x8000 0000
0.5	0x4000 0000
0.1	0x0ccc cccc

It is expected that the input and output of the *cos* function would be in single precision floating point IEEE 754 format. Hence conversions need to be applied around the CORDIC unit, which can be carried out by the Floating Point Functions Intel® FPGA IP Core [3].

2) Constants: `K` and `atan(pow(2,-count))` in listing 2 can be precalculated and stored as a lookup table. A Python script is written to have the values precalculated and hard-coded in the design file.

3) Stages: In general, less stages would be more favourable in terms of latency and resources of the block, which will be discussed in more details in follow subsections. However, a decrease in number of stage comes in the cost of accuracy.

Hence the relationship between number of stages and mean square error is explored. Accuracy of the module is evaluated by a Python script using the Monte Carlo simulation technique. It is ran with 1000 iterations with random uniformed inputs in the range [0,1] and the mean square error is recorded.

Figure 2 shows the relationship between stages and mean-squared error, with the horizontal grey line representing the threshold: $e = 10^{-10}$. It can be seen that a 16 stage CORDIC block has a mean square error of 7.67×10^{-11} within confidence level of 95%, meeting the threshold error requirement.

B. Throughput Optimisation

The latency of the above design is proportional to the number of stages in CORDIC. To meet the error threshold, it takes a minimum of 16 clock cycle to evaluate one value with this naive design. In order to increase throughput of the module, stages are unfolded and pipelined. Figure 3 shows the architecture of a pipelined design. The figure only shows 3 stages of pipeline for easier illustration.

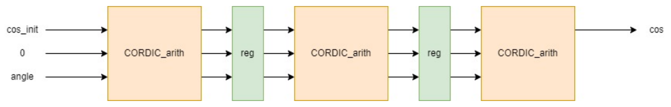

Fig. 3. Overall Architecture of unfolded CORDIC module

As the design is fully pipelined and unfolded, it can produce one valid output every clock cycle when the pipeline is full. Data are propagated down the pipeline, processing more than one value in the module every clock cycle. For a 16 stage CORDIC, although it still has a latency of 16 clock cycle, the module can reach a throughput of 1 value per cycle.

Data are propagating down the pipeline, having valid data output every clock cycle, the `Counter Comparator` block is not needed for condition checking. Thus it is removed from the design.

C. Latency Optimisation

It was observed that with the implementation in subsection III-A, the module critical path takes 6.373ns with each stage taking up one clock cycle. Assuming that the path is also the critical path of the entire system, the system can reach a frequency of 156.91MHz.

The current system is driven by a 50MHz clock, to optimise latency for this frequency, the module should be designed to have a critical path close to 20ns. This can be achieved by unrolling bits, having more than one stage performed per cycle. With 20 / 6.373 = 3.138, it is theoretically possible to fit 3 stages in one cycle. This is confirmed with a critical path of 15.42ns when a 3-bit-unrolled CORDIC module is implemented.

Figure 4 shows the architecture of a folded and unrolled design.

With a 3-bit-unrolled design, number of stages should be a multiple of 3. 18 CORDIC stages is chosen, because it is the minimum number to meet the accuracy threshold. Hence, the design has a latency of 6 clock cycles, as compared to 16 clock cycles in the original design with an improve in accuracy.

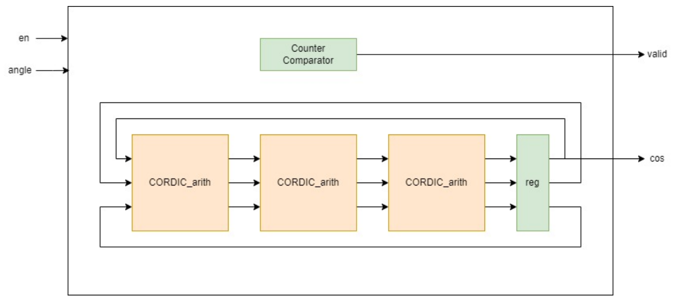

Fig. 4. Overall Architecture of unrolled CORDIC module

Table VI shows a comparison between the two designs in terms of latency, hardware units and throughput.

TABLE VI
COMPARISON OF THE TWO CORDIC IMPLEMENTATION

	Throughput Optimised	Latency Optimised
Latency	16 cycles	6 cycles
`CORDIC_arith` Units[a]	16	3
32-bit Registers	48	12
Throughput (per cycle)	1	0.1667

[a] `CORDIC_arith` refers to a combinatorial block that implements one stage of CORDIC.

D. Inner Expression

With the two designs of the CORDIC module, they are used to evaluate the inner expression of equation 1 by providing a custom instruction to the CPU that returns the result for a given x value.

```
#define ACCEL(A) __builtin_custom_fnf(0x1,A)
float math_expr(float x[], int M){
  float rtn = 0;
  for (int i=0; i<M; i++){
    rtn += ACCEL(x[i]);
  }
  return rtn;
}
```

Listing 3: Software Implementation

Figure 5 shows an data flow graph to evaluate the inner expression. Only combinatorial logic is shown, registers are added in actual design to pipeline and ensure correct timing. Table VII describes the high level functionality of each block in figure 5.

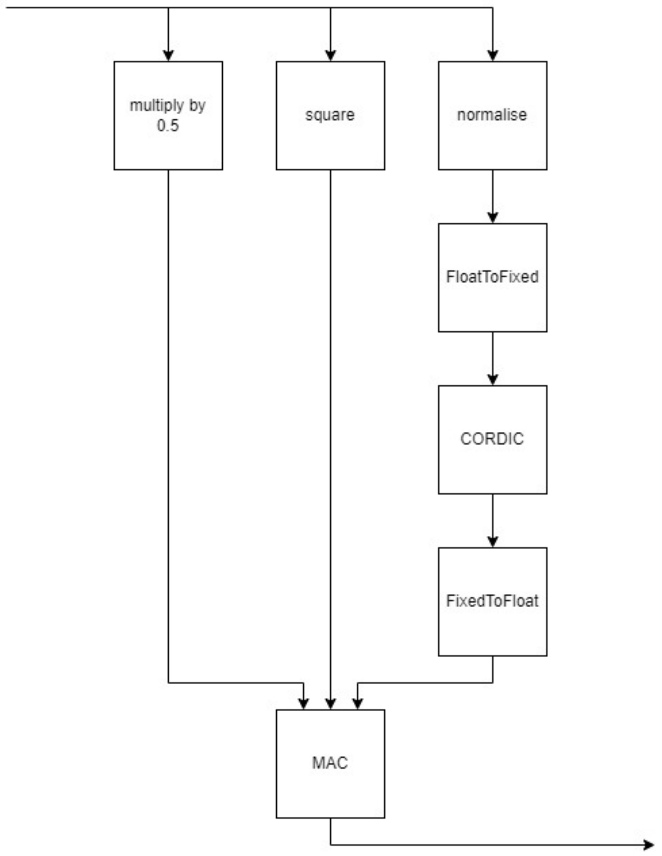

Fig. 5. Data Flow of Inner Expression

TABLE VII
BLOCK DESCRIPTIONS

Module	Details
CORDIC	CORDIC module
normalise	Normalise input of range [0-256] to [-1,1]
square	The output is the square of the input
FloatToFixed	Convert floating point numbers to fixed point representation with 31 bits of fraction
FixedToFloat	Convert fixed point numbers with 31 bits of fraction to floating point representation
MAC	Multiply and Accumulate

With Intel's floating point IP core, it takes 4 clock cycles to evaluate the values before passing into the CORDIC block and 4 clock cycles afterwards.

Table VIII shows the comparison between the throughput optimised design and latency optimised design when used to evaluate the inner expression. The throughput optimised design is configured to have 18 stages, ensuring designs are compared with the same accuracy.

It is apparent that the latency optimised CORDIC design is more suitable in this context. It provides both smaller latency

TABLE VIII
PERFORMANCE AND RESOURCE USAGE OF TWO INNER EXPRESSION IMPLEMENTATION

	Throughput Optimised	Latency Optimised
Latency	26 cycles	14 cycles
Case 3 Average Time (us)	204	156
Case 3 Error	8.892e-6	8.892e-6
Resource Usage	8.9568%	8.6034%

and resource usage. As only one element would be evaluated in the module at any instance, due to the Nios II execution model, there will not be any benefits by having the design unfolded. On the contrary, unfolding the design would lead to an increase in resource usage, which is confirmed both theoretically and in reality as shown in Table VI and Table VIII respectively.

Hence, the latency optimised design is chosen for this task, with performance recorded in table IX.

TABLE IX
PERFORMANCE AND ACCURACY WITH LATENCY OPTIMISED CORDIC DESIGN

	Average Time (ns)	Error
Case 1	34.30	1.3760e-7
Case 2	1072.42	6.2143e-6
Case 3	156386	8.892e-6
Case 4	1349.39	nil

E. Verilog Implementation of Floating Point Units

Intel IP cores were utilised for the floating-point modules. However, it was discovered that the latencies of the cores are too high. For instance, the latency of a floating point multiplier is 2 clock cycles. Thus it was thought that performance can be further improved by implementing the modules manually with `Verilog`. This gives more control of the logic and more specialised functionality of the blocks.

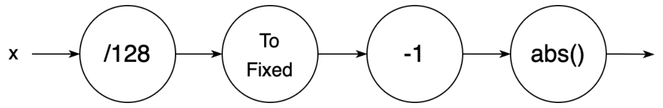

Fig. 7. Data Flow of the Normalise Module

1) Normalise: `normalise` was implemented with the data flow as shown in figure 7. It takes less logic to do division in floating point representation while more logic to do subtraction. Hence it is calculated as (x/128) -1 with the `FloatToFixed` module integrated with the `normalise` module, where the conversion happens after the division but before subtraction.

Division by 128 can be done easily be subtracting the exponent of the floating-point number by 7.

2) Square: A floating-point multiplier is written with algorithm as shown in listing 4.

```
assign c_significand =
 {1'b1,a[22:0]} * {1'b1,b[22:0]};
assign exp =  c_significand[47] ?
 a[30:23] + b[30:23] - 8'd126 :
 a[30:23] + b[30:23] - 8'd127;
assign out_significand = c_significand[47] ?
 c_significand[46:24] :
 c_significand[45:23];
assign c = {a[31]^b[31], exp, out_significand};
```

Listing 4: Floating Point Multiplier

3) FloatHalf: Instead of using a multiplier to multiply the input by 0.5. A specialised hardware to divide the input by 2 is designed. This is easily achieved by subtracting the exponent of the input by 1.

4) MAC: The `MAC` module takes in 3 inputs, multiply the first two and accumulate with the last input. The floating point multiplier in the `square` module is reused, along with an additional floating point adder.

In general, floating point addition is done as shown below with an example of $8.70 \times 10^{-1} + 9.95 \times 10^{1}$ calculated in base 10 for easier illustration. [4]

1) Rewrite the smaller number such that its exponent matches with the exponent of the larger number.
   $8.70 \times 10^{-1} = 0.087 \times 10^{1}$
2) Add the significands
   $9.95 + 0.087 = 10.037$
   and write the sum
   10.037×10^{1}
3) Put the result in normalised form by shifting significant and adjusting exponent
   $10.037 \times 10^{1} = 1.0037 \times 10^{0}$

However, it is known that the addend and augend will always be positive numbers in this context, so the exponent of the sum will always be equal or +1 of the exponent of the larger number. Instead of shifting the significand in step 3, the exponent can be determined by the MSB of the sum of significands as shown in listing 5:

```
assign q = c_significand[24] ?
 {1'b0,exp+8'b1,c_significand[23:1]} :
 {1'b0,exp,c_significand[22:0]};
```

Listing 5: Floating Point Adder

The proposed custom implementation of the modules show significant improvements in latency as shown in table X. The improvement in performance comes in the cost of slight increase in resource usage and less generality.

These modules are implemented with specific functionality and can be only used for this use case. For instance, the floating-point adder in `MAC` block can only calculate positive numbers. It is also assumed that inputs will always be a "normal" number, they would have undefined behaviour for inputs such as `NaN`, `Inf`. Although such practice is not ideal for commercial design, it was decided to be sufficient for this project.

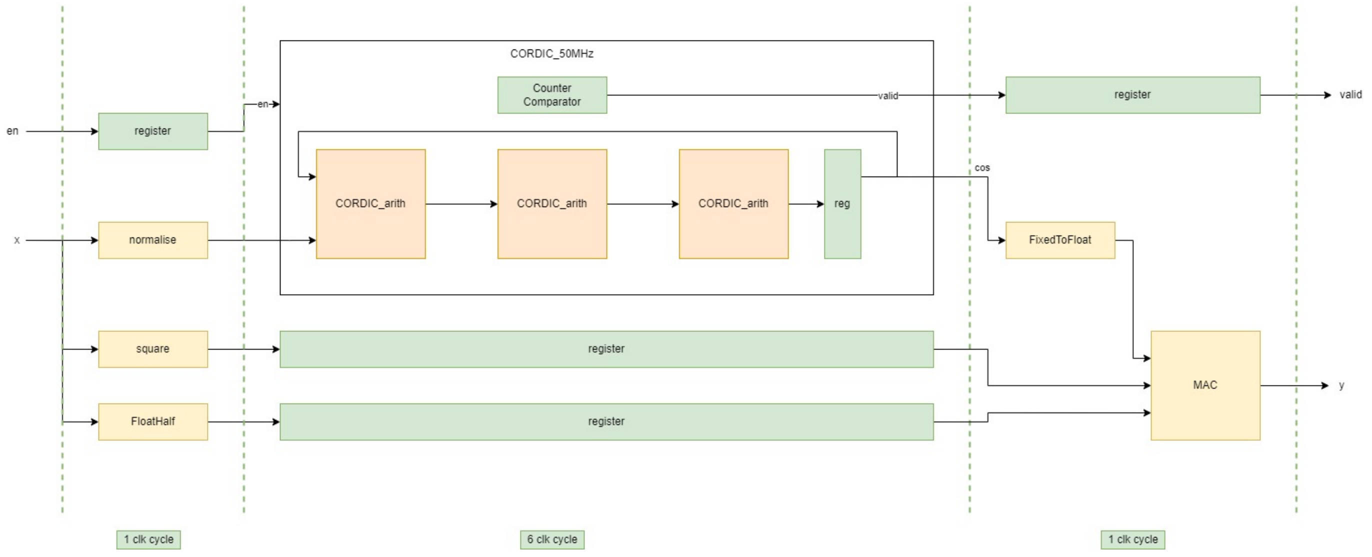

Fig. 6. Architecture of Implementation of Inner Expression

TABLE X
MODULE'S CRITICAL PATH

Modules	Critical Path (ns)
Normalise	7.67
Square	5.92
FloatHalf	1
MAC	12.34

TABLE XI
PERFORMANCE AND ACCURACY

	Average Time (ns)	Error
Case 1	23.04	1.3760e-7
Case 2	826.071	6.2143e-6
Case 3	126779	8.892e-6
Case 4	936.688	nil

With these modules, it is possible for the system to only take 1 clock cycle before and after the CORDIC unit to evaluate the inner expression. In such case, the design has a latency of only 8 clock cycles with 18 CORDIC stages, as compared to 14 clock cycles in previous section. Figure 6 shows the architecture and timing of the module.

Table XI and table XII shows a summary of the performance and resource utilisation respectively after adding dedicated hardware block to compute the inner part of the expression

TABLE XII
RESOURCE UTILISATION

Logic utilization (in ALMs)	3616 / 32070 (11%)
Total registers	3069
Total pins	47/457 (10%)
Total virtual pins	0
Total block memory bits	357428 / 4065280 (9%)
Total RAM Blocks	52 / 397 (11%)
Total DSP Blocks	5 / 87 (6%)
Total PLLs	1 / 6 (17%)
Total DLLs	0 / 4 (0%)

The hardware implementation of the inner expression of Equation 11 leads to a significant uptick in performance in comparison to previous software implementations with general purpose hardware (decreasing Case 3 latency by more than 98% in comparison to Task 6), showing the improvement that can be brought about by customisation of hardware.

However, this can be improved further by moving more of the expression into hardware for large n, as the use of the CPU to control execution incurs a significant overhead cost, which will be explored in the following section.

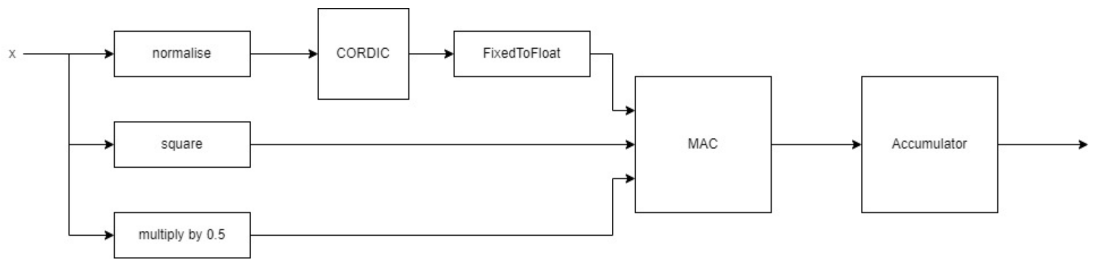

Fig. 8. Data Flow of Complete Acceleration Block

IV. Task 8

With the inner expression of Equation 1 now moved to hardware, the next obvious improvement is to move more of the expression into hardware, including the array access and also the summation of each inner expression calculation.

A. Data Transfer

Within the last iteration of the system, each value of the array is moved from memory to the CPU and then passed into the custom accelerator block for the inner expression that does the multiplication, addition, and cosine calculation. After the inner expression is evaluated, the result is then passed back to the CPU and summed by passing that value and the current accumulator value to a floating point addition block. Therefore, there can be multiple reads and writes to and from the off-chip memory where the stack is located. As the off-chip memory can require multiple cycles for each access, this is a potential bottleneck in the system. The management of the loop and accumulation being in software also introduces additional overheard for each loop iteration.

Therefore, an accelerator which makes use of Direct Memory Access (DMA) to where the array is held is proposed. Using DMA takes the CPU out of the data transfer path and the data goes directly from memory to the accelerator block.

As discussed in [2], the off-chip memory is mapped to the system using an SDRAM controller block which provides an Avalon Memory Mapped (Avalon-MM) agent interface to the off-chip memory. Therefore in order to perform direct access to memory, the accelerator block needs to have an Avalon-MM host interface to initiate data transactions [5]. The Modular Scatter-Gather Direct Memory Access (MSGDMA) block included within the Intel Embedded IP library provides such a controller that can either write to another Avalon-MM agent, or as an Avalon Streaming (Avalon-ST) source [6]. The core also provides a FIFO to buffer transactions so that memory transactions from off-chip memory can take place without waiting for the write transactions to complete. As it is a scatter-gather core, if the arrays were kept in non-contiguous memory, it would also be able to handle them.

The MSGDMA was added to the system in the Avalon-MM host to Avalon-ST source configuration. This was deemed to be more suitable for the current problem as the accelerator can be configured to take in a stream of data.

The MSGDMA is configured through two Avalon-MM agent ports, one which takes in descriptors providing the start address and number of bytes to read from memory. Multiple descriptors can be sent as an array to support reads from non-contiguous memory. The other Avalon-MM agent port holds the control and status registers that are used to check the transaction status and also initiate the data transfer. Although this could theoretically be done in hardware, it was decided to do the configuration of the MSGDMA through the Nios II core as it is more flexible and requires less time to design, verify to meet the device protocol requirements.

B. Acceleration Block

1) Custom Instruction: In order to read the value of the accumulator, and also inform the accelerator of how many values are being passed into the accelerator block, a custom instruction interface is maintained.

```
#define ACCEL(A) __builtin_custom_fnf(0x2,A)
float math_expr(float x[], int M){
  alt_msgdma_construct_standard_mm_to_st_descriptor
  (
        dev_ptr, a_desc_ptr,
        &x[0], N*sizeof(float), 0
  );
  alt_msgdma_standard_descriptor_async_transfer(
   dev_ptr, a_desc_ptr
  );
  return ACCEL(M);
}
```

Listing 6: Software Configuration of DMA and Accelerator

Within the software, the asynchronous data transfer is started, and the accelerator block immediately begins operating on the values that are streamed in. Within the accelerator, a counter register is initialised that keeps track of how many values have been added to the accumulator. During the streaming process, at some point in time, the custom instruction is also called and the number of elements is passed to the accelerator. The accelerator then uses this value to compare the given value with its internal counter to determine when the evaluation is complete. When the evaluation is complete, the valid signal of the custom instruction interface is asserted, and the value of the accumulator is passed back to the processor. The software implementation of this process is shown in Listing 6.

The asynchronous nature of the transaction, where the data transfer and the computation is carried out at the same time means that the runtime of the process is the maximum of the memory access time and the computation time, given by the following equation:

$$T_{\text{runtime}} = \max(T_{\text{memory access}}, T_{\text{computation}})$$

2) CORDIC Implementation: As one value can be passed into the block every cycle, the design can benefit from unfolding and fully pipelining the design. The throughput optimised design as stated in Section III with 18 stages is utilised, reaching a throughput of 1 value per clock cycle. Figure 8 shows the architecture of the design.

A register and a floating point adder is added to the final stage of the pipeline. They act as an accumulator to calculate the sum of the loop in equation 1. Such implementation does not have a valid output signal as data are being propagated down the pipeline. Hence it solely relies on the counter register stated in previous subsection to keep track of whether the output data is valid.

TABLE XIII
PERFORMANCE AND ACCURACY OF INITIAL SYSTEM

	Average Time (ns)	Error
Case 1	57.29	1.3760e-7
Case 2	169.74	6.2143e-6
Case 3	14926	8.892e-6
Case 4	185.98	nil

Table XIII records the performance of the system with DMA and acceleration block. The system has a resource usage of 4.7470%.

It was observed that the execution time for case 1 increases compared to section III. It is suspected that this is caused by a high overhead when streaming data from the DMA. However, improvement in performances of case 2, 3 and 4 is significant. The improvement in these cases outweigh the decrease in performance of case 1. Hence it can still be concluded that the design is more optimised compared to section III.

3) Unrolling CORDIC Design: With such a high-speed performance, it is thought that performance can be further improved by reducing the pipeline depth, thus reducing the time needed to fill up the pipeline. Hence, the CORDIC block is implemented as 3-bit-unrolled and fully unfolded. Figure 9 shows the data flow of a 3-bit-unrolled 9-stage CORDIC design. In the actual hardware, it is implemented with 18 stages.

This does not have a huge execution time reduction as was seen in Task 7 as the system is now pipelined and limited by the memory access time. The greater impact is on the resource usage. As there are less pipeline stages, less registers are needed, for the CORDIC block and to maintain the timing of the inputs to the `MAC` for x^2 and $0.5x$. The resource usage of the system decreases from 4.7470% to 4.7136%.

C. Performance Analysis

Table XIV and table XV shows a summary of the performance and resource utilisation respectively with the optimised hardware block to compute the expression.

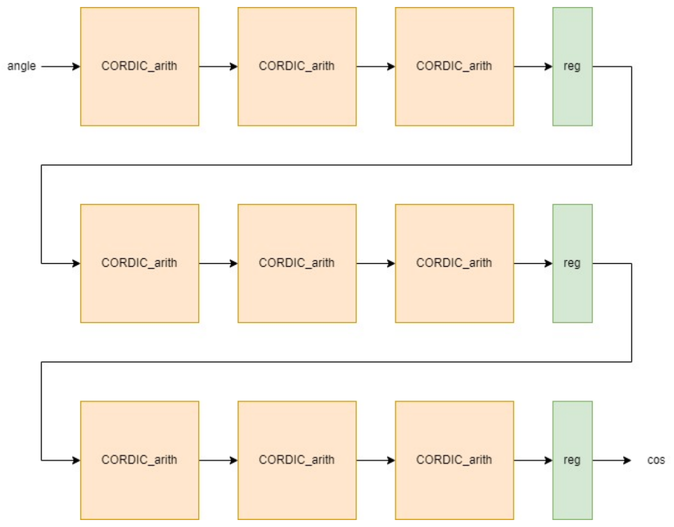

Fig. 9. Data Flow of the CORDIC Module

TABLE XIV
PERFORMANCE AND ACCURACY OF UNROLLED SYSTEM

	Average Time (ns)	Error
Case 1	57.29	1.3760e-7
Case 2	169.74	6.2143e-6
Case 3	14866	8.892e-6
Case 4	185.70	nil

TABLE XV
RESOURCE UTILISATION

Logic utilization (in ALMs)	3547 / 32070 (11%)
Total registers	2797
Total pins	47/457 (10%)
Total virtual pins	0
Total block memory bits	31785 / 4065280 (8%)
Total RAM Blocks	12 / 397 (3%)
Total DSP Blocks	2 / 87 (2%)
Total PLLs	1 / 6 (17%)
Total DLLs	0 / 4 (0%)

When compared with section III, it is observed that the resource usage significantly decreases by 45%. This is partially due to the requirement that the Nios II/e processor, that does not have a cache or multipliers, is used as opposed to the Nios II/f processor that was used previously. This significantly reduces total RAM blocks used in the FPGA board by 77% and the number of DSP blocks from 5 to 2.

Although the design of the accelerator block means it can process 1 new value per cycle, due to the SDRAM's read rate limitations, the runtime is now memory-bound rather than computation bound. Although the MSGDMA is also capable of producing a new value per cycle, as the SDRAM has a latency for reads and is not capable of bursts of more than 16, data is delievered to the accelerator in 4 consecutive blocks of data, followed by a period with no valid data. This behaviour was observed through the use of SignalTap to monitor the accelerator block when the program was running in order to profile the design.

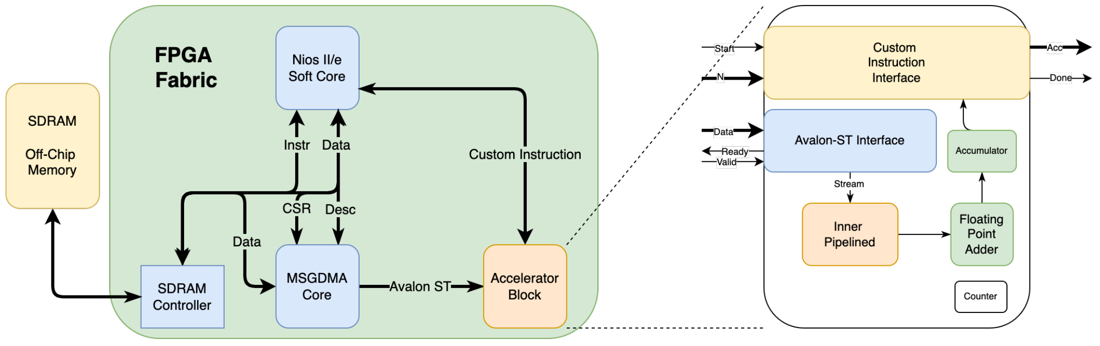

Fig. 10. Architecture of Overall Final System

V. Conclusion

Figure 10 presents the final system designed to accelerate the computation of Equation 1 over papers [1], [2] and this paper, moving gradually from a software computation on a general purpose computing system to a hardware computation making use of programmable logic. This paper presented the improvement that could be gleaned by adding floating point units for general arithmetic, the implementation of throughput optimised and latency optimised CORDIC blocks for cosine evaluation and the use of direct memory access to perform a streaming operation.

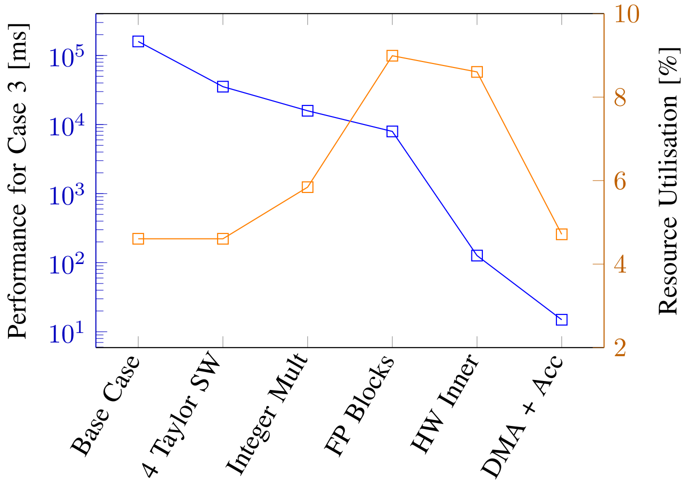

Fig. 11. Performance Optimisations Effect on Case 3

As the system moved from software to hardware, it was observed how there was a change in the composition of the runtime, moving from a system where it was dominated by CPU execution time of a software routine emulating floating point operations, to a system where the limiting factor was the memory access time. Therefore, future improvements for performance are focused on improving the rate at which data can be read from memory.

It is also noted while there is an initial increase in resource utilisation to increase performance of the general purpose system by adding more hardware, as the design becomes more specialised, the device footprint becomes smaller, at the expense of reduced flexibility.

VI. Future Improvements

A. Increasing Clock Frequency

The throughput of the system is limited to 1 value per clock cycle due to hardware constrain. Hence, if the clock frequency can be increased, the execution time can be decreased significantly while the throughput remains to be 1 value per cycle.

B. Increasing Throughput

Although the DMA specification has limit only one data transfer per clock cycle, it is possible to set the transfer up to quadword (128-bit) provided that there is a supported memory host. This would be equivalent as transferring 4 data (32-bit `float`) per clock cycle. By duplicating the accelerator block by 4 folds, the maximum throughput can increase to 4 values per clock cycle. The execution time can reduce by 4 time theoretically while keeping the clock frequency to 50MHz.

C. Decreasing Word Size

Resource usage can be further optimised by decreasing the word size of the `CORDIC` block. Current implementation has a word size of 32 bits with 18 stages. However, it is observed that same accuracy can be achieved by a smaller word size as depicted in Figure 12. For instance, implementing the CORDIC design with a word size of 20 bits with 16 stages can achieve the same accuracy as the current implementation. With a smaller word size, the arithmetic blocks (e.g. adder, shifter) in the `CORDIC` block can be implemented with reduced resource usage.

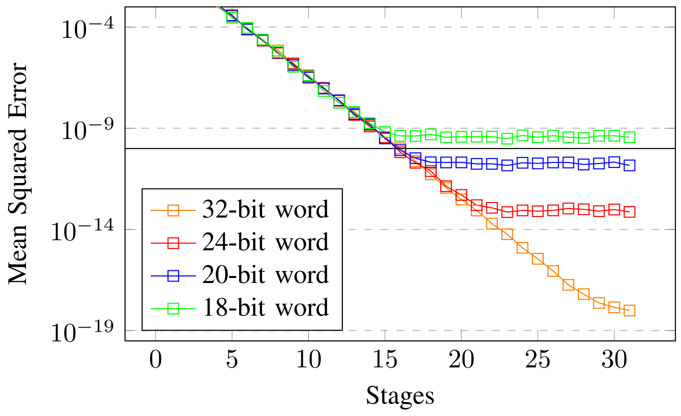

Fig. 12. Effect of Word Size on Accuracy of CORDIC

References

[1] Kwok, T. & Mendes, J. *Digital Systems Design Coursework - Report 1*. (2022,2)

[2] Kwok, T. & Mendes, J. *Digital Systems Design Coursework - Report 2*. (2022,2)

[3] Intel Cooperation. Floating-Point IP Cores User Guide, Section 16 FP_FUNCTIONS Intel® FPGA IP or Floating Point Functions Intel® FPGA IP Core. (2021, 09) available at https://www.intel.com/content/www/us/en/docs/programmable/683750/20-1/fp-functions-or-floating-point-functions-72394.html.

[4] Philip Eddie Edwards *Floating Point Numbers*. (2004,12) available at https://www.doc.ic.ac.uk/~eedwards/compsys/float/

[5] Intel Cooperation. *Avalon® Interface Specifications* (2022, 01).

[6] Intel Cooperation. *Embedded Peripherals IP User Guide*, Section 31 Modular Scatter-Gather DMA Core. (2020, 09) available at https://www.intel.com/content/www/us/en/docs/programmable/683130/21-4/modular-scatter-gather-dma-core.html.

[7] Intel Cyclone V Device Handbook Volume 1: Device Interfaces and Integration. *Intel*. (2018,11)

[8] Nios® II Processor Reference Guide. *Intel*. (2020,10)