Digital Systems Design
Report 3

Group 9 - Latency Minimization

Sohailul Islam Alvi

sohailul.alvi@imperial.ac.uk

Department of Electrical and Electronic Engineering, Imperial College London

Abstract

The demand for specialized computer architectures and accelerators continues to rise as Moore's Law and Denard Scaling face limitations. In response, using field-programmable gate arrays (FPGAs) for both prototyping and deployment of custom RTL designs has become increasingly prominent. This coursework aims to provide a comprehensive tour of the fundamentals involved in the design, development, and implementation of custom digital systems on FPGAs, to accelerate specific computations and address the evolving needs of modern computing systems.

Objective Overview

As discussed in Reports 1 and 2, following the implementations of Tasks 1-6 from the Coursework Instructions Manual, we are in the process of designing a custom digital system for our DE1-SoC FPGA, to accelerate the computation of the mathematical function (1) below:

$$f(x) = \sum_{i=1}^{N} \left(0.5 \cdot x_i + x_i^2 \cdot \cos\left(\frac{x_i - 128}{128} \right) \right) \quad (1)$$

So far, we have had this function written as software running on a NIOS II soft-core processor, while we tried to accelerate its computation by adding dedicated hardware units to do the addition, subtraction, multiplication, and division seen in the function. However, the *Cos* evaluation was still done in software. Now, in this phase of the coursework, we aim to map the complete function into hardware. That is, all the arithmetic operations, including the *Cos* evaluation, will be done fully in hardware. **In this report, our focus revolves around the hardware implementation of the CORDIC algorithm, which evaluates the *Cos*, as we try to minimize its overall latency**.

So, to build the desired hardware block, we proceed with a systematic approach, as listed below:

- **Step 1:** Find the minimum number of iterations and word length required for the CORDIC to meet the mean error's target confidence interval.
- **Step 2:** Design the folded CORDIC architecture.
- **Step 3:** Design the rest of the system, and add the CORDIC block.
- **Step 4:** Upgrade the folded CORDIC block with multiple CORDIC stages per pipeline stage.
- **Step 5:** Add hardware so that the instruction also takes in the current sum value, and returns the updated sum value, on each call.
- **Step 6:** Improve the design to allow multiple inputs to be processed sequentially, with an in-hardware results accumulation mechanism, that returns the total sum, only at the end.

As a point of reference, we use our results from Report 2, using the single precision *cosf* function (from *math.h* library) along with the FP IP blocks:

CPU:
- NIOS II/f
- 3 x 16-Bit Multiplier
- 4 KB Instruction Cache
- 4 KB Data Cache
- No On-Chip RAM
- 50 MHz Clock Frequency

FPGA Resources:
- Utilization is at 4.18%
- 2 x ALTFP_FUNCTIONS
- 64 MB Off-Chip SDRAM

Program Size and Memory Footprint:
- Application Size: 81 KB
- Free for Stack & Heap: 8103 KB

Test Performance Summary:

	Execution Time (ms)
Test 1	4
Test 2	187
Test 3	24024

Table 1: *Report 2 Performance Summary*

$$\text{Resource Utilization} = \frac{1}{3} \left\{ \frac{\text{LEs Used}}{\text{Available LEs}} + \frac{\text{MBs Used}}{\text{Available MBs}} + \frac{\text{EMs Used}}{\text{Available EMs}} \right\} * 100\% \quad (2)$$

Optimal CORDIC Configuration

With the key focus being the CORDIC block, we proceed to find its ideal configurations to meet the mean error's target range, that is $[-0.5 \times 10^{-6}, +0.5 \times 10^{-6}]$ with a confidence level of 95%.

In the interest of time and resources to spend on simulation, we decide to first run simulations on a MATLAB implementation of the CORDIC algorithm, to determine the configuration to suffice our requirement. **For this purpose, we use the Monte Carlo method as a simulation technique to assess the performance and accuracy of the algorithm for various combinations of the numbers of iterations, and the in-CORDIC fixed-point word lengths**.

In each simulation run, 1 million uniformly distributed random samples are generated within [-1, +1], the possible range of inputs stated in the coursework specifications.

We run the Monte Carlo simulations over a wide range of numbers of iterations (14 to 18 iterations) and word lengths (32 to 18 bits). After running the simulations for all possible combinations, we first plot a graph of the mean errors vs interaction counts. At this stage, we plot the average of all the mean errors and their upper and lower bounds of the confidence intervals, for each number of iterations.

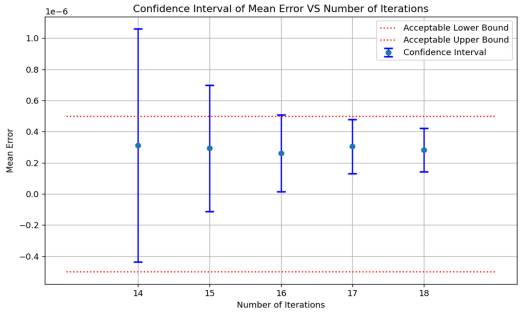


Figure 1: *Mean Error VS Iterations*

As in Figure 1, we see we need at least 16 iterations to stay within the acceptable error range. Hence, concluding that **16 iterations** are enough for our problem, we proceed to look for the minimum word length that also obeys the range.

From Figure 2, we see that all word lengths of 22 bits and more satisfy the acceptable error range.

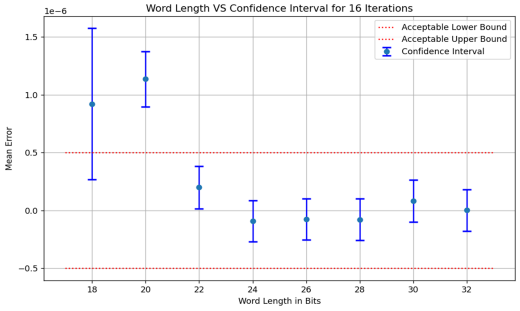


Figure 2: *Mean Error VS Word Lengths*

However, we choose **24 bits as the word length** for our design. **Here, we decide to choose 24 bits over 22 bits since the confidence interval for the 24-bit word length is narrower and includes the zero line**. This means the 24-bit results are more consistent and the average error is close to zero. The closeness to zero suggests less bias in the calculations at this word length.

Note, for the fixed-point words, we have used a configuration of the MSB being the sign bit, the next bit being the integer bit, and the remaining bits being the fractional bits. This is because, the input to the CORDIC will only be within [-1, +1], which means at maximum, we only need one integer bit. Thus, the chosen 24-bit word length is used as:

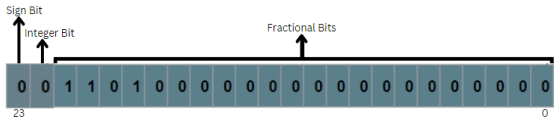


Figure 3: *Fixed Point Word Structure*

The Folded CORDIC Architecture

Now, having decided on the word length and number of iterations required, we proceed to implement the CORDIC block in hardware. **Post implementation, the block's functionality and accuracy are verified by testing it over a smaller Monte Carlo sample set in a Verilog test bench, with the mean error satisfying our range**. With the core CORDIC block designed, we implement it with a pipeline per stage to form the folded architecture as below:

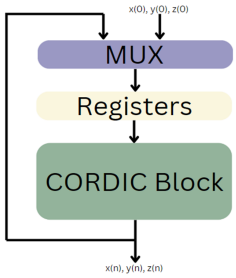


Figure 4: *Single-Stage Folded CORDIC Architecture*

Figure 4 shows the CORDIC block in its folded design, where the multiplexer at the top passes the initial values only at the start of the first iteration and then feeds back the generated values for the next 15 iterations. Hence, the latency with this architecture to produce a final cosine value is 16 cycles, from the moment the input has been passed in.

Designing the Complete Function

After a thorough simulation and cycle-accuracy check on ModelSim, we design the rest of the system required to evaluate the target function (1) in hardware and integrate the CORDIC architecture shown above.

The complete system to evaluate the function (1), with the CORDIC block placed, looks as follows:

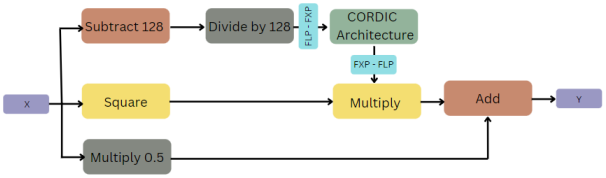


Figure 5: *Complete Function Implementation*

The latency of this complete function evaluation in our hardware designed on the FPGA is now 22 cycles. Table 2 below shows the cycle breakdowns of each operation required to evaluate the function. The cycles highlighted in red account for the longest delay path of our current design, which totals 22, giving the latency of our complete function. The multiplication with 0.5 was doable in 0 cycle delay since it is the same as dividing by 2, done just by a rift bit-shift. A similar logic applies for the division by 128, done simply via 7 right bit-shits. As for the conversions, those are also possible in 0 cycle delay, since both the conversions can be done using bit manipulations.

Operation	Cycles
Multiply x with 0.5	0
Square x	2
Subtract 128 from x	2
Divide by 128	0
Floating-Point to Fixed-Point	0
Evaluate the cosine	16
Fixed-Point to Floating-Point	0
Multiply cosine value with x squared	2
Add the result with 0.5x	2
Present Overall Latency	22

Table 2: *Overall Function Latency*

The hack behind the 0 cycle delay fixed-point to floating-point conversion is the fact that **for our inputs lying within [-1, +1], the cosine result will only range within [0.5, 1) or be 1**. This means the equivalent floating-point representation can only have two possible constant exponents, 127 when the cosine result is 1, or 126 for all other results within [0.5, 1).

Once the complete hardware is written in Verilog, and verified via simulations in ModelSim, we add a custom instruction to the NIOS II, so that our designed IP can be called in software, as follows:

```
71  float sumFunction(float x[], int len){
72      float sum = 0;
73      for (int i = 0; i < len; i++){
74          sum += ALT_CI_EVAL_FUNC_(x[i]);
75      }
76      return sum;
77  }
```

Figure 6: *Custom Function-Evaluation Instruction Call*

Note, we instantiate our custom instruction on the NIOS as a Variable Multicycle Instruction. So, this instruction call works based on a START, and a DONE signal. Whenever the instruction is called by the NIOS, it sends a START signal to the IP block and waits until it receives back a DONE signal from the IP. So, for our implementation as in Figure 5, our IP block of the function sends a DONE signal at the same cycle when the final Y value is ready, that is 22 cycles after receiving the START signal.

Now, with the custom hardware in place, we remove the 2 x ALTFP_FUNCTION blocks and the 3 x 16-bit multipliers previously used in our design in Report 2. We also remove the *math.h* library since the software cosine evaluation is no longer used.

We calculate the present FPGA resource utilization, check the program size and memory footprints, and run the three test cases using our custom instruction.

	Before	After	Change
Resource	4.18%	4.32%	+3.35%
Application	81 KB	72 KB	-11.11%
Stack & Heap	8103 KB	8112 KB	+0.11%
Test 1 Timing	4 ms	0 ms	-100%
Test 2 Timing	187 ms	6 ms	-96.79%
Test 3 Timing	24024 ms	761 ms	-96.83%

Table 3: *Resource, Application Size & Performance Changes*

Table 3 shows the first comparison of the performance parameters, between our previous software-based implementation, and the current hardware-based implementation of the function (1). The execution time reductions achieved, in blue, signify a milestone towards the core aim of this coursework, which is to accelerate the function (1) computation speed. The increase in resource utilization is expected due to the large amount of hardware implemented on the FPGA by our design. As for the reduction in application size, this directly roots from not having to include the *math.h* library function anymore, and any necessary software emulation routines.

At this point, as seen in line 74 of Figure 5, the external summation is still done via software emulation. So, we proceed to add another FP Adder custom instruction to the NIOS II, and call it as below:

```
71  float sumFunction(float x[], int len){
72      float sum = 0;
73      for (int i = 0; i < len; i++){
74          sum = ALT_CI_FP_ADDER_( ALT_CI_EVAL_FUNC_(x[i]), sum);
75      }
76      return sum;
77  }
```

Figure 7: *Custom Instruction Calls for Adder & Function (1)*

Adding this FP Adder increases our resource utilization from 4.32% to 4.63%. **Note, we use an ALFP_FUNCTION IP from Quartus to instantiate the FP Adder. These ALTFP_FUNCTION IPs allow us to instantiate designs for any operation and configure them with enough pipeline registers based on a set CPU clock frequency, which in our case, is 50 MHz**.

For ease of comparison, we only mention the performance of Test Case 3, for the rest of our design process discussed in this report, since changes in Test Case 1 and 2's performance will soon saturate to 0 ms. **However, we do ensure that our performance remains test case agnostic, by testing it with other cases too**. Re-running the test case 3, we find significant improvement in the execution time now:

	Before	After	Improvement
Test 3	761 ms	368 ms	51.640%

Table 4: *Performance Improvement with FP Adder*

Adding Multiple CORDIC Stages

In our current architecture from Figure 4, we only have a single CORDIC stage per pipeline stage, which is one iteration for our folded design.

Since one CORDIC stage is purely combinational, we can add more CORDIC stages to each pipeline stage of our design. **But, we do need to take care not to violate the timing constraints of the 50 MHz clock frequency of the NIOS II CPU**. We try placing 2, and 4 CORDIC stages within one pipeline stage, but the design does not meet the timing requirements when having 4 stages, as the critical path may have been longer than what is feasible at 50 MHz. So, we decide to proceed with 2 CORDIC stages, as below:

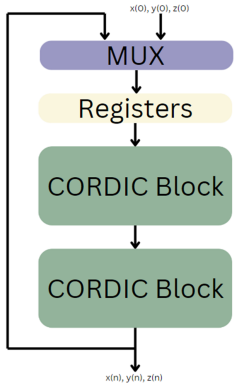


Figure 8: *2-Stage Folded CORDIC Architecture*

With this updated CORDIC architecture, two CORDIC operations happen on the data at each cycle. **So, to perform 16 CORDIC operations, our new architecture will take 8 cycles, bringing down the total latency of our complete function evaluation hardware, as seen in Figure 5, down to 14 cycles, from the previous 22 cycles**. With the addition of one more CORDIC block hardware, the resource utilization increases from 4.63% to 4.78%. And, as expected, the execution time improves as well:

	Before	After	Improvement
Test 3	368 ms	326 ms	11.41%

Table 5: *Performance Improvement with 2-Stage CORDIC*

Evaluating Function & Summing

Now, with our CORDIC taking 8 cycles, and the total function latency being 14 cycles, we look at the summation in the function (1). Previously shown in Table 4, we did get significant performance improvement by adding a separate custom instruction to do the FP addition in hardware, as used in line 74 of Figure 7.

However, this shows a clear possibility for further improvement. **NIOS II, being an in-order processor, will only issue a new instruction once the previous one has been completed**. This means, in Figure 7, the **FP Adder instruction** is only issued when the previously issued **Function Evaluation instruction** has returned its result. This mechanism results in the **NIOS II having to issue two separate custom instructions** to get the updated sum.

Thus, we come up with the new design shown in Figure 9, where the FP Adder is integrated with our complete function implementation, shown earlier in Figure 5.

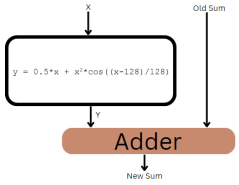


Figure 9: *Complete Design with Adder*

For this design, we **only need to issue a single custom instruction in the NIOS II**, where we pass both the input X we want to evaluate the function (1) for, and also the current sum value. This will then evaluate the function for X, and once the resultant Y is ready, add it to the old sum value, to produce the new sum result, which is the data returned from the custom instruction call shown below:

```
71  float sumFunction(float x[], int len){
72      float sum = 0;
73      for (int i = 0; i < len; i++){
74          sum = ALT_CI_EVAL_FUNC_AND_ADD_SUM(x[i], sum);
75      }
76      return sum;
77  }
```

Figure 10: *Custom Instruction Taking In X and Current Sum*

The resource utilization for this design remains at 4.78%, the same as the previous design since we are using the one FP Adder anyway. This time, it is just placed differently in the system, so that we do not require a separate custom instruction call for it.

	Before	After	Improvement
Test 3	326 ms	306 ms	6.14%

Table 6: *Performance With A Single Instruction Call*

As expected, we see a slight improvement in the execution time in Table 6. This improvement comes from the issue we wanted to fix, that is, now **the NIOS II only has to issue one instruction, with two operands, to get the updated sum** result. Unlike before, the **NIOS II does not have to first wait to receive the function's result, and then reissue a second instruction to get the updated sum**. These

savings in the NIOS II's waiting time directly give us this performance improvement.

Adding an Accumulator

Now that we have a design that allows us to evaluate the function (1) and get back the updated sum value for each input X, we look to improve the overall sum function execution time further by taking off some more burden from the NIOS II.

Instead of receiving and storing the updated sum in each custom instruction call as in Figure 10, we can have a hardware accumulator in our design, to keep accumulating the function evaluation results and store the updated sum value in a register in the hardware. It only sends the final sum value back to the NIOS when the last input X from the sample array has been evaluated. The new design is as follows:

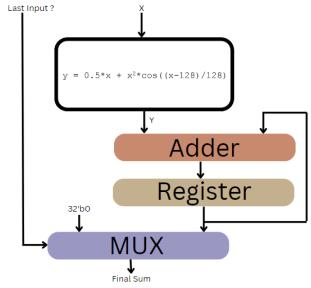


Figure 11: *Complete Design with Accumulator*

With this design, the DONE signal is still sent 14 cycles after receiving the START signal. However, it is just that the **NIOS does not have to read and store the result until the last input X** from the sample array, in which case the final DONE signal will be delayed by 2 more cycles so that the final sum value can be read after the last addition taking place. Note, we pass two parameters in our instruction call, one being the input X, and the other being a 1 or 0, indicating whether it is the last input or not, which also directly controls the multiplexer logic.

This design allows for multiple (or more accurately, all) inputs to be sent to the custom instruction before the NIOS even has to do its first read and store operation of the result. This is seen in the custom instruction calls in lines 74 and 77 below:

```
70  float sumFunction(float x[], int len){
71      float sum = 0;
72      for (int i = 0; i < len; i++){
73          if (i != len - 1){
74              ALT_CI_EVAL_FUNC_AND_ACCUM(0, x[i]);
75          }
76          else{
77              sum = ALT_CI_EVAL_FUNC_AND_ACCUM(1, x[i]);
78          }
79      }
80      return sum;
81  }
```

Figure 12: *Custom Instruction Taking In X and Current Sum*

The resource usage does increase from 4.78% to 4.82% due to added multiplexer and register. As for the execution times, improvement is evident again:

	Before	After	Improvement
Test 3	306 ms	285 ms	6.86%

Table 7: *Performance With Accumulator*

Software Optimization

Finally, we look into improving the performance with software changes, by setting the Compiler Optimization Flag to -03. **This level 3 optimization aims to optimize performance via advanced loop unrolling mechanisms and better cache handling strategies to reduce memory access delays**. A significant reduction in the execution time is evident in Table 8.

	Before	After	Improvement
Test 3	285 ms	146 ms	48.77%

Table 8: *Performance After Compiler Optimization*

At this stage, the application size also reduces from 72 KB to 68 KB, rooting from the compiler's removal of any redundant codes. This frees up 4 more KB of memory space for the stack and heap usage.

Conclusion

Throughout the development process, we saw a wide variety of design decisions and IP usage to accelerate the function (1) computation. Figure 13 shows the relative performance graph of all our iterative designs, with **the Pareto Front in red, going over designs where we cannot improve one of the metrics without making the other worse**.

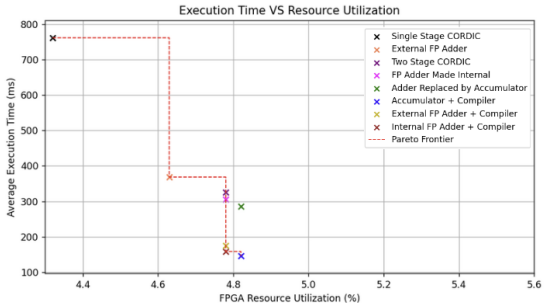
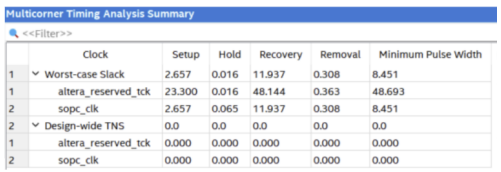


Figure 13: *Resource Usage VS Time Performance for Test 3*

We see that our designs gave the best execution time performances with the compiler optimization enabled. Of which, our final design with the accumulator gives the best performance with slightly more resource usage.

Finally, we conclude our development process by verifying the muti-corner timing analysis summary of our final design, to ensure it meets all constraints at 50 MHz clock frequency, as evident in Figure 14.


Multicorner Timing Analysis Summary

<<Filter>>

	Clock	Setup	Hold	Recovery	Removal	Minimum Pulse Width
1	Worst-case Slack	2.657	0.016	11.937	0.308	8.451
1	altera_reserved_tck	23.300	0.016	48.144	0.363	48.693
2	sopc_clk	2.657	0.065	11.937	0.308	8.451
2	Design-wide TNS	0.0	0.0	0.0	0.0	0.0
1	altera_reserved_tck	0.000	0.000	0.000	0.000	0.000
2	sopc_clk	0.000	0.000	0.000	0.000	0.000

Figure 14: *Multicorner Timing Analysis*