Digital Systems Design
Report 2

Group 9

Johan Jino johan.jino21@imperial.ac.uk

Sohailul Islam Alvi sohailul.alvi@imperial.ac.uk

Department of Electrical and Electronic Engineering, Imperial College London

Abstract

The need for specialized computer architectures and accelerators is growing over time due to the demise of Moore's Law and Denard Scaling. To suffice this need, using FPGAs in prototyping custom RTL designs has become prominent over time. Hence, this coursework aims to give a detailed tour of the fundamentals involved in the design and development of custom digital systems on FPGAs, to accelerate specific computations.

Objective Overview

As discussed in Report 1, following the implementations of Task 1 and 2 from the Coursework Instructions Manual, we are in the process of designing a custom digital system for our DE1-SoC FPGA, to accelerate the mathematical function (1) written as software running on a Nios II soft-core processor.

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

As a first step in the design process, we started with an aim to accelerate a simpler mathematical function (2), and discussed Resource Utilization, Timing Analysis, and overall System Performance of the design, alongside analyzing the Program Size and Memory Footprint of the three given test cases.

$$f(x) = \sum_{i=1}^{N} x_i + x_i^2 \quad (2)$$

Now, in Tasks 3, 4, 5, and 6, which are the focus of this report, we systematically proceed to enhance the performance of the digital design, based on our conclusions from Report 1 as summarized below:

CPU:

- Nios II/f
- No FP ALU or Multiply/Divide Units
- 2 KB Instruction Cache
- 2 KB Data Cache
- 30 KB On-Chip RAM

FPGA Resources:

- Utilization is at 3.76%
- No Off-Chip Memory

Test Case 1:

- Execution Time: 0 ms
- Maximum Memory Required: 20 KB

Test Case 2:

- Execution Time: 28 ms
- Maximum Memory Required: 30 KB

Test Case 3:

- Test Could Not Be Run
- Minimum Memory Required: 1044 KB
- Memory Available On-Chip: < 496 KB

Program Size and Memory Footprint:

- Application Size: 14 KB
- Free for Stack & Heap: 13 KB

Notes:

- The memory space free for stack and heap increases as we increase the allocated on-chip RAM size, but the application size remains consistent throughout as the input vector is only allocated at run-time.
- All execution times are averaged from 10 runs.
- Varying cache sizes have no effect for now, as the memory being used is already on-chip and there is no off-chip memory available to the system.
- The FPGA resource utilization is computed in percentage using the formula (3) where LE is Logic Utilization in ALMs, MB is Total Block Memory Bits, and EM is the DSP Blocks.

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

Introducing the Off-Chip Memory

So for now, the primary bottleneck for our system is its available on-chip memory. Test Case 3 requires a minimum of 1044 KB (for N=261121, it needs 261121 * 32 bits) as available memory for stack and heap to be used during runtime. Whereas, the Cyclone V FPGA has less than 496 KB of RAM (M10K) space.

Hence, we decide to incorporate off-chip memory into our design, the 64 MB SDRAM on the FPGA. This also required a PLL block to be added to generate a shifted version of the clock signal from the Nios II to drive the off-chip SDRAM. Now that there is an off-chip memory available, the previously instantiated on-chip memory is completely removed from the design, which also brings the current FPGA resource utilization down to 2.09%.

At this stage, memory is not a bottleneck anymore. With enough memory available, all three test cases are successfully run, with the support for full C++ libraries enabled and reduced drivers turned off. The application size is now 73 KB with 8111 KB free for stack and heap, and the execution times are as below:

- Test Case 1: 0 ms
- Test Case 2: 36 ms
- Test Case 3: 6262 ms

Unlike in Report 1, the **presence of the off-chip memory indicates the need to study the impact of varying cache configurations** of the Nios II processor. For ease of comparison, we decided to focus solely on the system's performance in executing Test Case 3 on function (2), as we increase the Data and Instruction cache sizes independently from 2 KB, 4 KB to 8 KB.

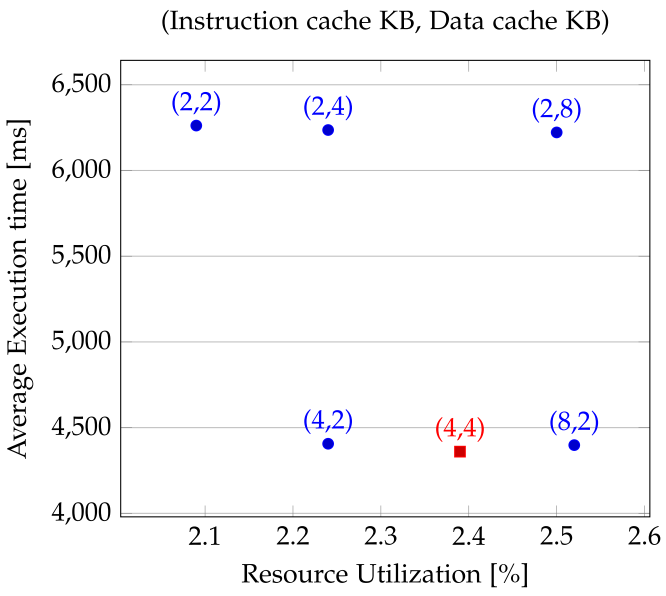


Figure 1: *Cache Performance vs Resource Utilization*

Above plot shows how the execution time for Test Case 3 varies as the cache configuration is changed, alongside showing the change in FPGA resource usage. We can see, that the execution time benefits the most by increasing the instruction cache size from 2 KB to 4 KB, increasing it further does not show any significant improvements. It implies, that the main loop of the test code (32-bit * number of instructions) now fits within the 4 KB instruction cache. Increasing the data cache size shows slight improvements, which come from spatial locality, as the cache can now load more consecutive data points at once.

As a sweet spot for our design at this stage, we decide to proceed with a 4 KB Instruction Cache and a 4 KB Data Cache configuration. This gives a current FPGA resource utilization of 2.39%.

Evaluating the Cosine Function

Now that our system has a large 64 MB memory, and executes all test cases, we shift focus to the primary objective of the coursework, which is to accelerate the cosine function (1). Rewriting the Nios II software with the function (1), which uses the math.h library, the application size becomes 83 KB with 8101 KB free for stack and heap usage.

The execution times for the test cases on the cosine function (1) in our current system (with 2.39% resource usage) are:

- Test Case 1: 29 ms
- Test Case 2: 1231 ms
- Test Case 3: 172828 ms

In the C code running on the Nios II, the floating-point numbers are single precision (32 bits). We now proceed to compare the results generated by this C code on the Nios II with an equivalent Python implementation on our computer, where floating-point numbers are implemented using double precision (64 bits). Care has been taken to ensure the Python implementation generates the input vectors in the same way as the C code.

Table 1 shows the results obtained for each test case, by the C code on Nios II, the Python implementation, and the accuracy of the C code results.

Test	C Result	Python Result	Accuracy (%)
1	920413.5	920413.6	99.999989%
2	36123104	36123085.6	99.999949%
3	4621531136	4621489018	99.999089%

Table 1: *Comparison of C and Python Results*

Despite relatively high accuracy, the single-precision (32 bits) C code does not generate the same results as the double-precision (64 bits) Python implementation. The **accuracy drops as N increases across the test cases due to single precision's lower resolution, which accumulates rounding errors more**

rapidly than double precision. As N grows, the increased number of operations signifies these errors, leading to a more noticeable discrepancy.

Adding Multiplier Support

So far, the Nios II soft-core processor of our digital system does not have any multiplication support. This means that all floating-point multiplications have been emulated in software at the cost of fixed-point addition operations in our Nios II. Hence, now we proceed to add hardware support for multiplications in our system.

At this stage of the design process, we add 3 16-bit multipliers to our Nios II CPU. These **multipliers are used to do fixed-point multiplications, which means that floating-point arithmetic still needs to be emulated in software**. At least, now the fixed-point multiplication operations can be used instead of the addition operations.

Now, the execution times of the test cases are:

- Test Case 1: 12 ms
- Test Case 2: 509 ms
- Test Case 3: 67374 ms

This impressive reduction in the execution time of all test cases after adding the multiplier support comes at a definite cost of FPGA resource utilization, which now rises to 3.56%. Table 2 below displays the direct comparison of our system's performance (execution times) of each test case, before and after the addition of the multiplier support:

Test	Before (ms)	After (ms)	Improvement (%)
1	29	12	58.621%
2	1231	509	58.652%
3	172828	67374	61.017%

Table 2: *Performance Improvement*

Another key point to note is that the application size is now 81 KB, with 8103 KB free for stack and heap usage. Before adding the multiplier support, the application size was 83 KB. This **2 KB reduction in the application size could come from the elimination of software emulation routines** for multiplication operations. Without the need for those often large and complex routines, the compiled application becomes more compact. Additionally, the availability of hardware multiplication units might allow the compiler to generate more efficient and concise machine code, which further reduces the overall application size.

Despite the FPGA resource utilization increasing from 2.39% to 3.56%, the current design with the multiplier support should be selected over the earlier design to achieve maximum performance. As seen in Table 2, the 58-61% improvement in the execution times significantly outweighs the small increase in resource utilization, and it gets more significant over increasing number of N.

Custom Floating Point Units

So far our floating point operations where emulated through fix point arithmetic. In the last session, we observe the performance gain of adding integer multiplication support in NIOS II. We shall now implement and observe how enabling floating point operations in hardware will effect both our resource utilisation and performance.

To enable multiplication ALTFP_MULT IP is added to our design. We set the latency initially to 11, which met our multi-corner timing constraints. A multi-cycle design is required due to the complex hardware, **the critical path is longer than what is feasible with 50MHz clock frequency**. Faster execution time is observed as shown in table (3). We further add floating point adder hardware using the ALT_ADD_SUB IP with latency set to 14. We create the IP with a control input signal to enable both addition and subtraction. This enables usage of same hardware for both operations, reducing resource utilization. The design is optimised for speed initially. The results from each iteration of design can be seen in Table 3.

Test	MULT	MULT_ADD_SUB	Improvement
1	11	10	9.091%
2	465	407	12.473%
3	58717	53303	9.242%

Table 3: *Performance Improvement*

The accuracy from each of these still remain the same as before as we still use single precision. The latency of the custom hardware, directly correlates to the performance of the system. **Higher the latency, more is the time taken for each NIOS II custom instruction to receive the result**. Our aim of this task must to reduce the latency while at the same time meet our timing requirements for 50MHz. **Pipelining will not produce any increase in throughput since the NIOS II processor is in-order and will not issue a new instruction until a previous one is completed**. Hence if pipelined, all of the pipeline slots except one will be unused every cycle. This is why reducing latency is our main aim.

For moving towards our aim mentioned above we use a special IP provided by Altera. **Altera provides with an IP called ALTFP_FUNCTIONS. This IP allows us to instantiate designs for any operation and configure them with enough pipeline registers**

based on our set clock frequency. This allows us to reduce latency to 2 cycles and use fewer registers hence fewer resources. The improved performace can be compared in the table (4) below.

Test	FP_UNITS	FP_FUNCTIONS	Improvement
1	10	9	10.0%
2	407	382	6.142%
3	53303	50478	5.299%

Table 4: *Performance Improvement*

Since adding custom floating point units are expensive, the resource utilization has grown significantly with adding these components. Below we see a comparison of the different iterative designs so far and compare their resource utilization.
Resource Utilization:

- With ALT_FP_MULT: 6.44%
- With ALT_FP_MULT and ALP_FP_ADD_SUB: 6.87%
- With ALT_FP_FUNCTIONS: 6.76%

Hence we can say that using ALT_FP_FUNCTIONS is the ideal design when keeping the clock frequency constant at 50MHz. We have accelerated the floating point computations in the target function. Our current bottleneck is the evaluation of the trigonometric function.

Cos Evaluation Strategies

Below we shall look at different strategies that can be used to evaluate *cos* function. It is important to note that all of the below implementations only deal with software changes. Hardware evaluation of *cos* such as using CORDIC algorithm will be tested later in the upcoming tasks.

Math.h library

Until now we have used the *cos* function from the math.h library to evaluate our expression. We can also use the *cosf* function, which is the single precision version of cos.

Test	*cos*	*cosf*	Improvement
1	9	4	55.55%
2	382	187	51.047%
3	50478	24024	52.410%

Table 5: *Performance Improvement*

From the above table we can see that we gain performance of more than 50% which is very significant. Even though result from *cos* is more precise than *cosf*, we do not see any difference in final result.

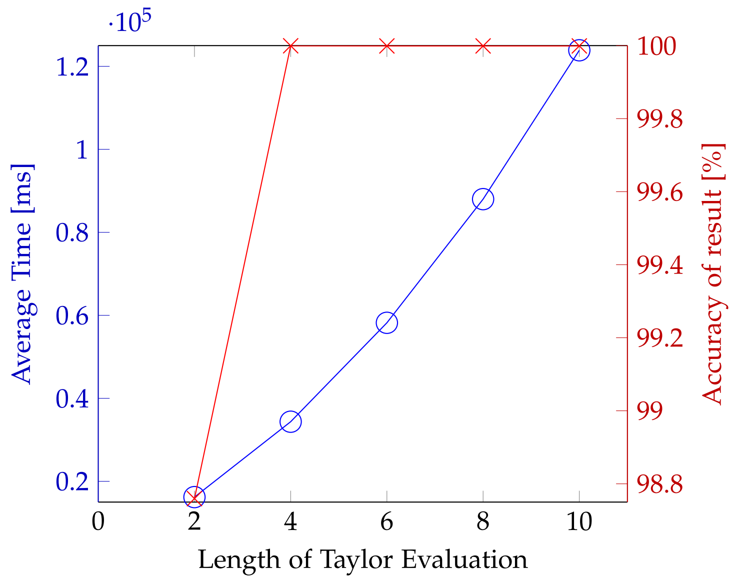


Figure 2: *Taylor Expansion Performance and Accuracy*

This is because of we use floats instead of double in all our other calculations (FP multiplication and addition), which makes our result still have only single precision accuracy.

Taylor Series Expansion

By default *cos* and *cosf* uses the integer multipliers in NIOS II since the math.h library uses default C operators and would not use our custom FP units.

Hence, our next try is to implement our own Taylor expansion in C using our custom FP blocks.

From the graphical representation in Figure 2, it can be understood that the performance of Taylor expansion is not near the performance vs accuracy of the *cosf* function from math.h .

Lookup Implementation

We would further attempt to implement *cos* functions based on look up tables, but storing values of certain intervals. **By increasing the number of samples we increase the accuracy of result at the same time we increase memory required for look up.**

We implement the lookup in software using a Array of predefined size. We perform linear interpolation to calculate values that occur between the two points stored in the array. It is also to be noted, there is overhead of converting values from outside $[0, 2\pi)$ to within, which adds up to our performance constraints. After sweeping through different array sizes and evaluating Test case 3, **we observe the performance, i.e. average execution time is always constant. This is approximately 10450 ms, which is a 56% performance gain compared to *cosf*.**

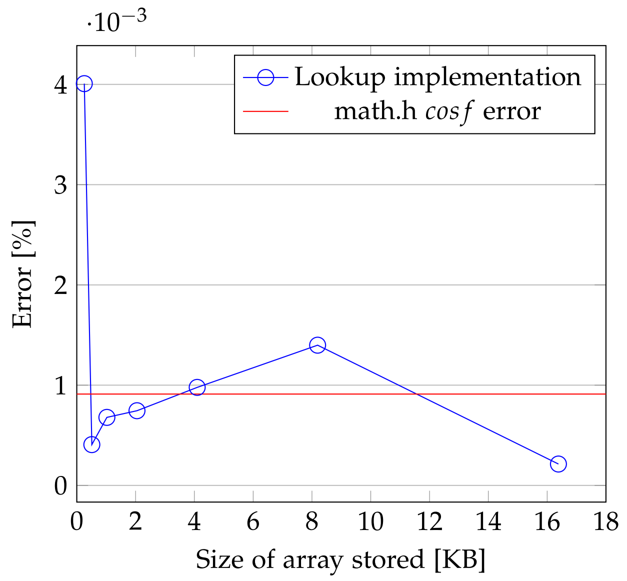


Figure 3: *Lookup based cosine Resource and Error*

There is deviation of about +-5ms from this average value, which correlates to poor cache performance since our array size gets larger. This only contributes to 0.05% of the performance since the most deciding factor in performance is the floating point operations that still needs to be performed.

The results from the exploration has been plotted in figure 3 and can be observed. The size of array stored was calculated by:

$$size(KB) = \frac{len * 4bytes}{1000} \tag{4}$$

It was quite surprising to note that increasing the samples after a point increased our error compared to double precision python implementation. This is because from a lookup array of size 512 bytes (0.5KB), the accuracy (100-error) is greater than what can be achieved by single precision. After this the error is random, due to rounding errors in the FP arithmetic. This is likely to change based on values within the array and the order in which they are computed. This also leads to a question if reordering the data to reduce rounding errors could help improve accuracy. This can be a software exploration but is beyond the scope of this report!

Software Optimizations

Finally we add few software changes to maximum our performance:

- -03 Compiler optimisation
- replacing divisions with constant reciprocal multiplication
- reordering additions and multiplications

In case of the lookup based implementation we got a performance improvement from 10450 ms to 7770 ms, which is 25.64% improvement. In case of the $cosf$ implementation we observe an improvement from 24024 ms to 22895 ms which is only 4.7%. This is because the lookup implementation was implemented by ourselves which hence has more scope for optimisation than $cosf$ math.h library functions is likely already optimised.

Conclusion

Our design exploration has seen a wide variety of techniques and hardware IP used to accelerate the calculation of our function. We shall now plot a relative performace graph to see all our iterative designs.

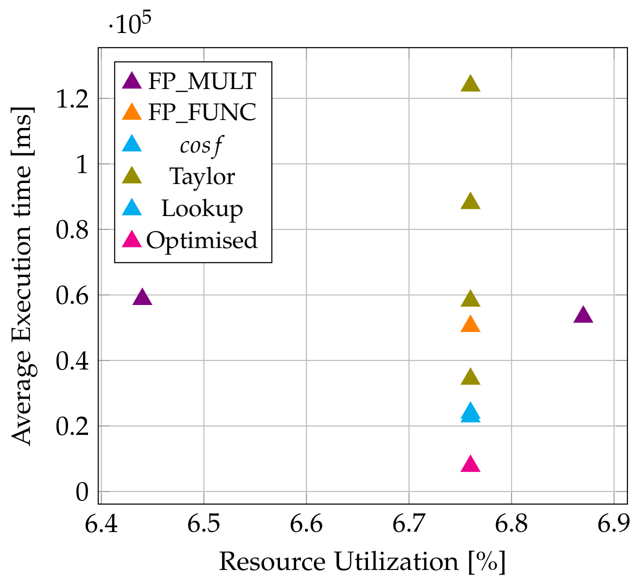
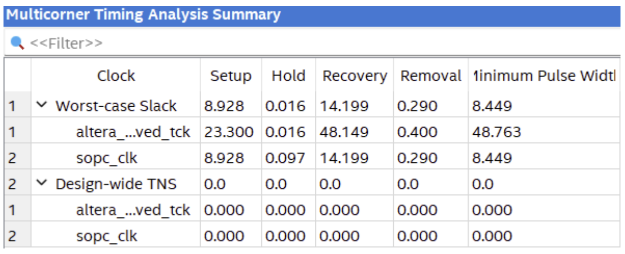


Figure 4: *Test case 3 Exploration*

In terms of pure performance, Lookup table of size 512KB along with FP_FUNCTIONS IP blocks and compiler optimisations, gave the most best computation time. This is indicated by the megenta marker on Figure 3.

We would like to conclude with the timing analysis of our final design which meets all constraints given the 50MHz clock frequency, as seen in Figure 5.


Multicorner Timing Analysis Summary

<<Filter>>

	Clock	Setup	Hold	Recovery	Removal	linimum Pulse Widtl
1	Worst-case Slack	8.928	0.016	14.199	0.290	8.449
1	altera_...ved_tck	23.300	0.016	48.149	0.400	48.763
2	sopc_clk	8.928	0.097	14.199	0.290	8.449
2	Design-wide TNS	0.0	0.0	0.0	0.0	0.0
1	altera_...ved_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 5: *Multicorner timing analysis*