Report 2 Group 15

Zhihao Jia CID: 02215217 email:zj722@ic.ac.uk
Tingxu Chen CID: 02221097 email: tc1822@ic.ac.uk

February 13, 2025

1 Task 3: Evaluate a more complex mathematical expression

The goal of this experiment is to implement and evaluate the performance of our processor on the following function:

$$f(x) = \sum(0.5 \cdot x + x^3 \cos((x - 128)/128)) \tag{1}$$

To assess the performance of the processor, we focusing on three main result, the execution time, hardware resources utilization and numerical accuracy. The result are present in table 1.

As shown in Table 1, comparing the calculation results from the Nios processor (32-bit single precision) with MATLAB (64-bit double precision) shows a noticeable accuracy error. This limitation arises because the IEEE 754 standard represents 32-bit floating-point numbers with a 23-bit mantissa plus an implicit leading bit, with a precision limited to approximately 7 significant decimal digits. In our case, the calculation contains recursive accumulation of previous results. When the result is small, the float type (32-bit single precision) representation maintains relatively high accuracy. However, as the accumulated value increases, precision limitations prevent the full result from being displayed. The result is rounded to the maximum precision allowed, which can lead to the elimination of floating-point digits if they exceed the precision limit. In extreme cases, this can even cause errors in the integer digits. Table 1 has present mean squared error for three test cases, increase sharply with increase in accumulated result.

The software can emulate double-precision calculations, producing results identical to MATLAB. However, when the test case is too large, the NiosII processor may run out of execution time or exceed memory limitations (especially the stack and heap limitation). In such cases, the processor fails to provide a result.

Execution time increases sharply with the number of iterations. Case 1, with the fewest iterations, completes within 1 second, whereas Case 3 takes over two minutes to compute the result. Interestingly, when comparing execution time between float and double precision within the same test case, double precision shows slightly shorter execution time. This is likely due to better-optimized software emulation. Additionally, resource usage (program size) is lower for double, also suggest that the software library for double is more efficient, maybe requiring fewer operations compared to float emulation.

Test Case	Execution time(s)	Program Size (KBytes)	Accuracy (.6f)	MSE with MATLAB (double precision)
Case 1 (float)	0.109	83	170105728.000000	3.19×10^1
Case 1 (double)	0.107	80	170105733.646853	0
Case 2 (float)	4.391	83	6627435520.000000	5.69×10^5
Case 2 (double)	4.314	80	6627436274.372950	0
Case 3 (float)	143.351	83	211932430336.000000	3.44×10^{21}
Case 3 (double)	unable	80	failed to compute (expect 270624654078.500366)	0

Table 1: Performance on function implementation with MSE values

2 Task 4: Add multiplier support

To improve performance, one approach is to utilize additional hardware to accelerate specific operations.

Without dedicated hardware, multiplication is emulated using a combination of operations supported by existing hardware, such as shift and add. To implement multiplication with dedicated hardware, we can integrate it into Platform Designer, where we have three main options for integer multiplication: logic element multiplier, 1×32-bit multiplier, and 3×16-bit multipliers.

The logic element multiplier is implemented using LUTs within the FPGA. For small operands, this combinational approach performs multiplication quickly and with minimal resource usage. However, as operand sizes grow, the required combinational logic increases dramatically, leading to high resource utilization. In

addition, the multiplier's bit-level dependencies (e.g., the carry propagation in shift-add operations) introduce long latency and limit the maximum clock frequency. As shown in platform designer, it takes 11 cycles to compute, result in bad performance.

The other two multipliers are implemented using DSP Blocks in the FPGA. DSP blocks usually using algorithms to optimize computation (E.g Booth recoding). So these multipliers usually provide high performance for large wordlength operands and are able to produce results in a single cycle.

Table 2 shows the resources usage and execution time when calculating our target function 1. As shown in table, all three hardware methods yield better performance than software emulation compared to table 1

The comparison highlights several key points. Pipelined designs significantly outperform non-pipelined ones in terms of latency, with only a minor increase in resource usage. Between extended and non-extended 3 $\times$ 16-bit multipliers, the performance difference is negligible in our use case, as high-precision calculations are not required (we used float testcase instead of double). However, the extended version consumes more resources, including an additional DSP block, making the non-extended design more suitable.

Comparing the non-extended 3 $\times$ 16-bit multiplier and the 1 $\times$ 32-bit multiplier, both provide similar performance, but the 1 $\times$ 32-bit multiplier consuming fewer logic elements and one less DSP blocks. Therefore, the 1 $\times$ 32-bit multiplier is the most optimal choice, balancing performance and resource usage effectively.

Design Choice	Logic Utilization	Total Registers	Total DSP Blocks	Test Case	Latency (s)
Logic Elements Multipliers and Shift/Rotate non-pipelined	1554/32070 (5%)	2443	2/87 (2%)	Case 1 Case 2 Case 3	0.061 2.431 78.770
Logic Elements Multipliers and Shift/Rotate pipelined	1554/32070 (5%)	2429	2/87 (2%)	Case 1 Case 2 Case 3	0.045 1.824 58.780
3 $\times$ 16-bit Multipliers, no extended and Shift/Rotate non-pipelined	1526/32070 (5%)	2269	3/87 (3%)	Case 1 Case 2 Case 3	0.053 2.133 69.109
3 $\times$ 16-bit Multipliers, no extended and Shift/Rotate pipelined	1569/32070 (5%)	2273	3/87 (3%)	Case 1 Case 2 Case 3	0.038 1.527 49.133
3 $\times$ 16-bit Multipliers, extended and Shift/Rotate non-pipelined	1593/32070 (5%)	2379	4/87 (5%)	Case 1 Case 2 Case 3	0.053 2.127 69.080
3 $\times$ 16-bit Multipliers, extended and Shift/Rotate pipelined	1633/32070 (5%)	2384	4/87 (5%)	Case 1 Case 2 Case 3	0.038 1.500 47.575
1 $\times$ 32-bit Multiplier	1504/32070 (5%)	2189	3/87 (3%)	Case 1 Case 2 Case 3	0.038 1.521 49.105

Table 2: Resources usage and performance of different multiplier implementations

3 Task 5: Add custom instruction to NIOS

Apart from using provided hardware, we can design custom hardware to execute specific instructions. In this task, we utilized an existing Intel-provided IP to implement an integer hardware multiplier. Custom instruction hardware can be implemented in various ways; in this task, we conduct a combinational design example, which is effective for optimize simple calculations.

We first instantiating the IP, then verify its functionality using a testbench and simulated in ModelSim. Once the simulation passed, the hardware was integrated into the existing processor as a custom instruction slave, with all ports correctly configured. Detailed guidance can be found in Intel's documentation.

However, the benefits of such hardware always based on resource usage. It is important to analyze resource usage and performance when deciding to add custom hardware, which will be discussed in Task 6.

4 Task 6: Floating point arithmetic hardware

To accelerate floating-point arithmetic without software emulation, we integrate custom hardware instructions into our NIOSII system to implement floating-point addition, subtraction (add-sub) and multiplication (mul).

Custom hardware for add-sub and mul can be implemented either as a combinational circuit or as a multicycle design. However, we decided not to use combinational logic in our design since floating-point operations involve multiple computational steps. For instance, add-sub require bit alignment before computation, and both add-sub and mul, need rounding afterward. Implementing all these steps in a single combinational circuit would create a long, complex logic path with high latency and limits the clock frequency.

Since our goal is high-performance with both high frequency and high throughput, we choose a multicycle design. This approach breaks the operations into several stages separated by registers, reducing the critical path delay and allowing for pipelining to further boost throughput.

When configuring the custom block, the target frequency and the number of cycles it operates must be considered. We set the frequency to 50 MHz to match the processor's clock, ensuring stable operation and preventing risks (e.g metastability). The number of cycle(Throughput) is determined by the hardware structure and desired operating frequency and is automatically calculated by MegaWizard, where mul block requiring 2 cycles and the add-sub block taking 3 cycles. The reason for extra cycles of add-sub block is usually due to the need of an extra stage (e.g align the operands) before computation as mentioned earlier. Figure 1 shows the simulation results of the two IP blocks. After a few cycles, the design successfully produces correct results at every cycle, indicating 100% pipeline utilization theoretically.

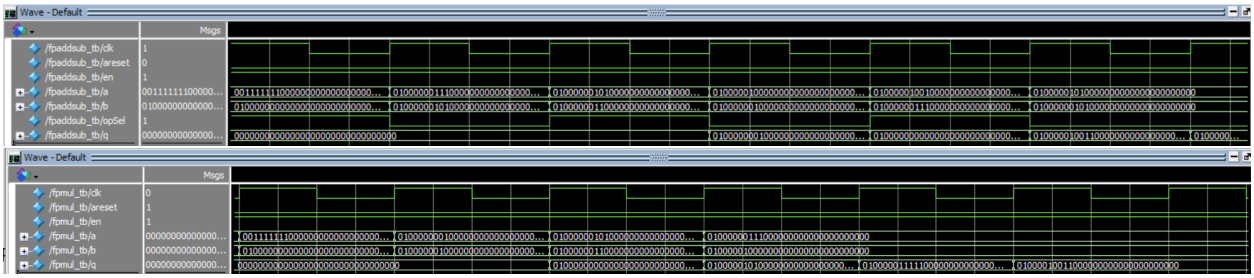

Figure 1: pipeline assessment simulation result

However, when uploading the hardware to the FPGA and running the software, we observed that hardware sometimes failed to produce a result within the expected cycle for some test cases. This is a common problem when mapping a design to real hardware, as simulations often do not account for practical issues such as additional wiring latency and data hazards. To ensure correctness, we kept the same block design but added an extra cycle to the processor for both blocks. As a result, pipeline utilization is no longer 100%, as the design stalls in most cases when the result is ready, target to make sure the few special cases that require additional cycle can be computed correctly.

The final hardware utilization details are shown in Table 3.

The mul block utilizes 1 DSP because DSP blocks in FPGA are specifically optimized for multiplication. So in case of mul block, the DSP block handles the core multiplication task for higher efficiency than doing multiplication by logic elements. As a result, logic elements are only required for tasks like rounding or normalization, so few ALM consumption. In contrast, add-sub relies entirely on logic element to perform all stages, including exponent alignment, computation, rounding, and so on. Since these operations are not inherently for DSP blocks. Finally result in higher ALM usage while consuming zero DSP resources.

IP	Logic Utilization (in ALMs)	Total registers	Total DSP Blocks
add-sub	317	153	0
multiplication	75	32	1

Table 3: Hardware utilization for IPs

Table 4 presents the performance of the new processor when calculating Equation 1. The execution speed improved a lot by using custom instruction to calculate all floating point operations (except $cosf()$) using specific hardware instead of software emulations.

The hardware IP are configured as single precision which accepts 32-bit floating-point inputs, consistent to the IEEE 754 standard. Still, this results has slight error compared to double-precision calculations in MATLAB. To achieve alignment with double-precision results, the IP can be reconfigured to support double precision, but this would increase execution time and resource usage. The overall hardware utilization for single precision is summarized in Table 5.

Test Case	Execution Time (s)	Code Size	Result(float)(.6f)
Case 1	0.0050	83	170105728.000000
Case 2	0.1620	83	6627435008.000000
Case 3	5.1290	83	211932430336.000000

Table 4: Performance comparison of function implementation

Logic (in ALMs)	Total registers	Total pins	Total block memory bits	Total DSP Blocks
2,038 (6 %)	2712	47 (10 %)	3,159,680 (78 %)	5 (6 %)

Table 5: Processor's hardware utilization with two extra floating-point IPs

In <math.h>, two function are provided for calculating cosine which are $cos()$ and $cosf()$. $cos()$ are designed for double precision calculation where $cosf()$ are single precision version. In standard C math library, both function is evaluated using an approximation-based method, which first implement an range reduction to map input x to a smaller interval(e.g $-\frac{\pi}{4}$ to $\frac{\pi}{4}$), then followed by a polynomial or rational approximation. $cos()$ often takes more terms when doing approximation for better precision, but this introduce more operation and result in larger latency in design and finally longer execution time when doing software. However, Both of these implementations does not utilize our hardware's custom instructions, as the standard library does not integrate hardware-specific optimizations for calculation.

$$\cos(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n} \tag{2}$$

This can be further verified by calculating the average execution time for a single add-sub or multiply operation. The average operation time is obtained by summing the execution time of these specific operations in Test Case 2 and dividing by the total number of operations. The results, presented in Table 6, show that $cosf()$ takes around 60 times longer than a single add-sub or multiply operation, confirming that it is not accelerate by hardware.

IP add-sub	IP multiply (s)	$cosf()$
0.000001287	0.000001674	0.000066804

Table 6: Execution time for single operation.