Report 3 Group 15

Zhihao Jia CID: 02215217 email:zj722@ic.ac.uk

March 17, 2025

1 Task 7

In this task, I designed hardware block specifically for equation 1, where $cos()$ is implemented by using **Fixed-point Cordic** algorithm.

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

Compare to IEEE-754 single precision representation, fixed point representation has following advantages:

1. Fixed-point arithmetic is generally faster in hardware implementations. This is because fixed-point operations do not require the additional processing needed to normalize numbers, align decimal points, or perform rounding operations that are inherent in floating-point computations.
2. Fixed-point representation allows for customized number representation based on accuracy and range requirements. So, only the necessary number of bits are used, avoiding using extra resource to supporting a wider range than needed.

2 Designing fixed-point representation and number of CORDIC stages.

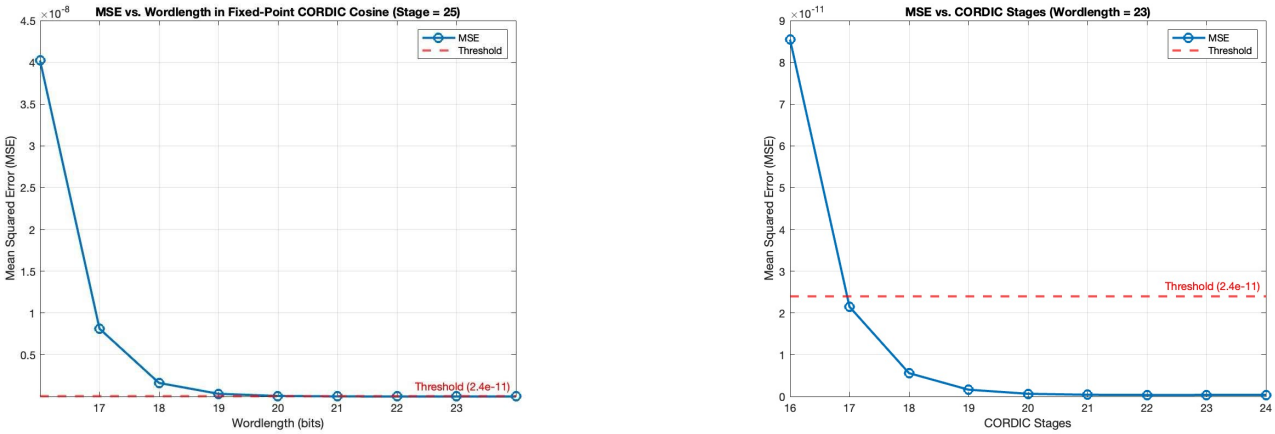
Fixed-point representation design requires consideration of both range and precision. The range is determined by the number of integer bit. According to coursework specification, the input of cos() function are guaranteed to be within $[-1, 1]$ this means it needs 2 bits to represent the sign and magnitude. However, since cos() is an even function ($\cos(-x) = \cos(x)$), the sign of the input does not affect the output. Moreover, the output of cos() for inputs in the range $[-1, 1]$ is always positive, so, no need for a sign bit for both the input and the output. As a result, I can safely omit the sign bit and optimize the design by using only 1 integer bit.

For precision, The design must meet the specification requirement: the mean squared error between the CORDIC result and MATLAB's single precision output must be less than 2.4×10^{-11} with 95% confidence. There are two factors affect this precision in this specific CORDIC case:

1. The fractional word length, which determines the maximum accuracy this fixed-point representation can hold.
2. The number of CORDIC stages, which influences the computed result's accuracy.

My strategy is to first set the stages to a sufficiently high number to ensure good accuracy, then determine the minimum fractional word length required to hold required accuracy. After that, I fix to this word length and reduce the number of stages to find the minimum necessary stages to achieving the target accuracy.

By implementing Monte Carlo simulation in MATLAB, with 1e5 random samples, I got the MSE result with different wordlength and stages shown in figure 1a and 1b and decide to use 22 bits decimal points (Q1.22) and 19 CORDIC stages.



(a) Fractional wordlength determination

(b) CORDIC stages determination

Figure 1: Determine fractional wordlength and CORDIC stages based on specification using MATLAB

3 CORDIC Architecture

There are two methods I tried to implement The CORDIC, combinatianl and sequential.

For combinational methods, it need to unrolled stages(one copy of hardware for each stages). The general advantage for unrolled architecture is that it computes indepandent part of different bits simultaneously, potentially speeding up the design. However, with the CORDIC algorithm each stage's result depends entirely on the previous stage. As a result, even if the design is unrolled the following stage remains not working until the previous stage passes its result.So this sturcutre consuming more resources but not increasing performance of our case.

Another theoretical benefit of combinational logic is it may complete all stage computations within one clock cycle so that an input can be processed in every cycle, yielding an output per cycle, which is a high throughput. I implemented this idea and report the simulation results in the following table. Unfortunately, the latency in combinational path surpass the clock period. Finally, this combinational approach is ineffective for our CORDIC design.

Method	clk period	Slack	Approximate time per stage
Combinational	20ns (50Mhz)	-31	2.5ns

Table 1: Compilation report from combinational design

The second method I explored was to use sequential logic for the CORDIC design. The simplest way is to create hardware for a single stage and reuse it iteratively for all stages. While this conserves resources and can achieve low latency per stage, but the overall throughput remains low and pipelining is not supported, which limiting performance for larger computations. So I discarded this single-stage approach.

Instead, a more efficient design is to place multiple CORDIC stages within one clock cycle, forming a sequential structure that contains several combinational logic(for calculating multiple stages). This arrangement achieves a better balance between resource usage and throughput, also allowing pipelined operation.

I divide 19 stages into pattern of 5, 5, 5, 4 stages, which consume around 15ns for each combinational block. I didnt fully ocupies the clock period(20ns) and leaves a margin for extra timing constrained, e.g setup and hold time, wiring delays and so on.

Apart from CORDIC block, two more additional blocks are required for this $cos()$ operation, which are data conversion blocks. The input to the CORDIC block is in IEEE-754 single-precision format while the CORDIC computation operates on a Q1.22 fixed-point representation. So I designed two hardware modules to convert the input and output data between these formats. Both of these blocks operate as combinational logic, simply extracting the relevant components and performing straightforward conversions. However, to ensure data correctness when integrating them into the CORDIC pipeline, each block is allocated one clock cycle, thereby guaranteeing stable results.

Finally the whole $cos()$ computation finish in 6 cycles.

4 Overall $f(x)$ 1 Implementation

The final task is to integrate all individual components into a single hardware to compute 1. To enhance both throughput and latency, I divided the work into three parallel paths, allowing simultaneous computation. The high-level structure of the design is illustrated in Figure 2.

The entire structure is a sequential logic, with some branch outputs produced ahead of others. Therefore, it is essential to insert the appropriate number of pipeline stages on each path to ensure that valid data is extracted at the correct time. The cycle counts and delays for each path are illustrate in the Figure 3

Finally, the table 2 reports the summary of this hardware, including resources utilization, table 3 report the performance of this hardware, including execution time and error compared to MATLAB sinlge precision result. (The accumulation part for each test case performed at this point are using software add)

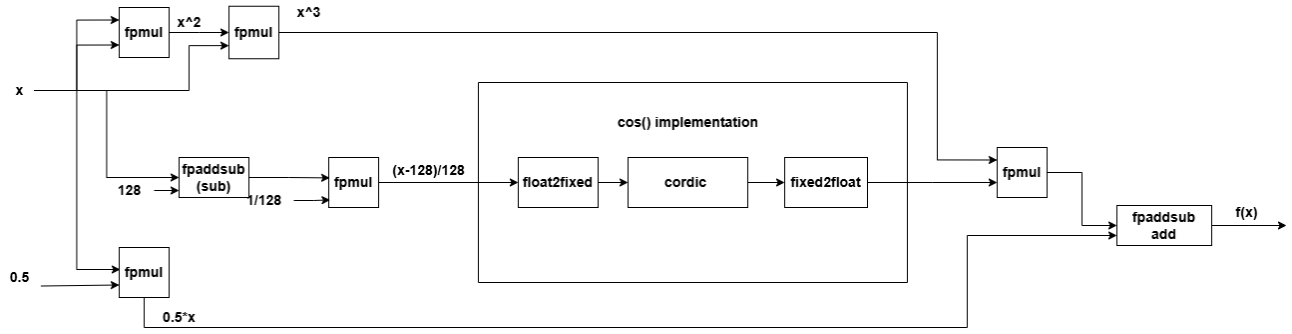


Figure 2: high level structure

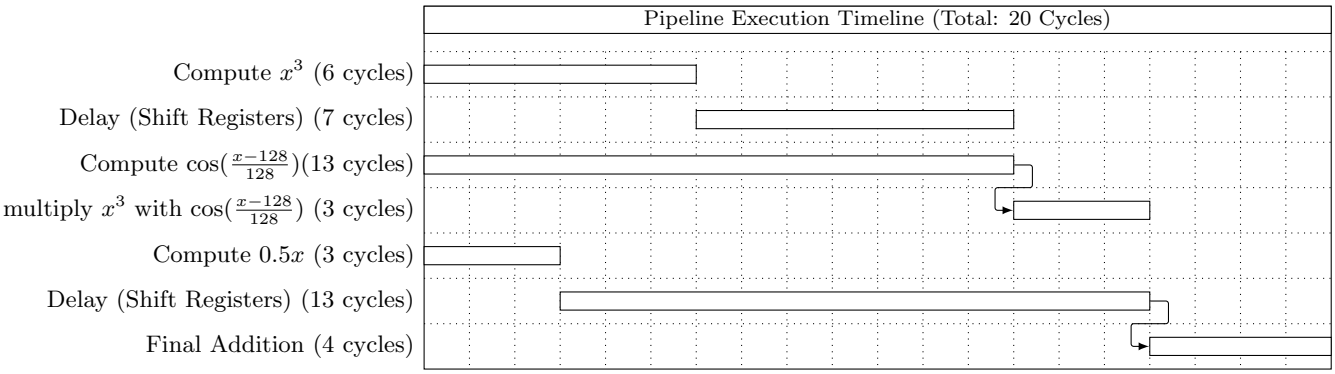


Figure 3: Pipeline execution breakdown with cycle distribution (Total: 20 Cycles)

Logic utilization (in ALMs)	Total registers	Total pins	Total block memory bits	Total DSP Blocks
1,781/32,070(6%)	1294	99/457(22%)	448/4,065,280(1%)	5/87(6%)

Table 2: Hardware utilization $f(x)$ hardware block

Testcase	Execution time (s)	Result	MSE	Percentage Error
case 1	(failed to count)	170099808.000000	3.504e7	3.480e-5
case 2	0.005000	6627567104.000000	1.7314e10	-1.985e-5
case 3	0.178000	211943849984.000000	1.304e14	-5.388e-5

Table 3: Performance of $f(x)$ hardware block

5 Task 8

For this task, the goal is to perform the following equation:

$$f(x) = \sum 0.5 \cdot x + x^3 \cos((x - 128)/128) \tag{2}$$

This is the accumulated version of 1. To utilize the existing hardware, I can simply call two custom instructions in Eclipse to perform the accumulation; for example:

- `float result = ALT_CT_ADD(ALT_CT_FX(`$x[i]$`)), result);`

However, this approach does not fully utilizing the pipelining of the $f(x)$ hardware due to a data dependency: the `ALT_CT_ADD` instruction must wait for the result from `ALT_CT_FX` before it can begin computation. Fully utilizing the pipelining would require a new input to be inject to hardware at every cycle, but because of this dependency, the Eclipse compiler only inject a new input to `ALT_CT_FX` once `ALT_CT_ADD` has produce a valid output. So this method does not fully utilize the capability of my $f(x)$ hardware. However, even with this method, simply change software add to hardware block, It still can hugely improve the performance of execution time of equation2. Execution time droped from 0.178s to 0.089s result are report in table 4.

To further improve performance, I instantiated a second copy of the $f(x)$ hardware. This allows two accumulation paths to be computed in parallel, and the results can be merged in a final addition. While this approach relief the limit of data dependency, and it yields a higher throughput. The corresponding performance results are reported in Table 4.

Method	Test Case	Execution time(s)	Code size	Result	MSE	Percentage Error
Data dependent	3	0.089	73(KBytes)	211943849984.000000	1.304e14	-5.388e-5
Parallel	1	(failed to count)	73(KBytes)	170099792.000000	3.524e7	3.490e-5
Parallel	2	0.002000	73(KBytes)	6636615168.000000	8.427e13	-1.385e-3
Parallel	3	0.059000	73 (KBytes)	211950927872.000000	3.422e14	-8.728e-5

Table 4: Performance comparision of two version of design

Following table report the resource utilization of parallel design5. Compare to table 2, This design use approximately double of logic elements, register and DSPs, which align with the design where using two copy of $f(x)$ hardware to run in parallel.

Logic utilization (in ALMs)	Total registers	Total pins	Total block memory bits	Total DSP Blocks
3,869/32,070(12%)	2764	99/457(22 %)	896/4,065,280(1%)	10/87(11%)

Table 5: Hardware utilization $f(x)$ hardware block

However, the above design are still not fully utilize the pipeline sturecture of the $f(x)$block, because the data depandency are still there. To address this I redesigned the hardware to integrate the accumulation function directly with the $f(x)$ block, as shown in the figure 4. This hardware block computes $f(x)$ and then accumulates the result by adding it to the previous accumulation (ACC+) value stored in the register. The output of the `fpadd` block is fed into the register each clock cycle, forming the updated accumulation.

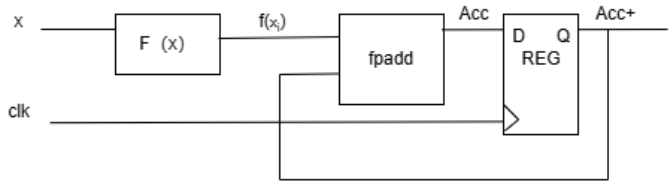


Figure 4: High level plot for hardware to compute 2

By integrating the addition stage (accumulation) directly after $f(x)$ and using pipeline registers, data dependencies between $f(x)$ and the accumulation are eliminated. This design allows inject new inputs in every clock cycle, thereby achieving full pipelining. However this design requires a super strict timing matching, unfortunately, i didnt finish the design, so only left here an idea.