Digital Systems Design Report 3

(task 6-8)

Yeong Kyun Han

Benjamin Tan Jun Yi

Group number: 26

Introduction

The goal of this report is to evaluate the function below with the lowest latency

$$y = 0.5*x + x^2*\cos((x-128)/128)$$

The tests are all run using a fixed array of size 261121 unless stated otherwise, in which case a random array of size 2323 is used.

The errors of the designs are calculated relative to a double implementation using python code found in the appendix

The following designs will be compared using the following formula for resources:

$$\text{FPGA resources} = \frac{1}{3}\left(\frac{LE}{Total\ LE\ in\ the\ device} + \frac{EM}{Total\ EM\ in\ the\ device} + \frac{MB}{Total\ MB\ in\ the\ device}\right)$$

Adding floating point hardware

The nios processor does not have any floating point hardware and is hence only emulating it in software, this should be resulting in increased latency so we have proceeded to implement floating point hardware for addition, subtraction and multiplication as custom instructions to be used in the function evaluation. The hardware is implemented as multi-cycle custom instructions rather than combinational because the combinational version would result in a decrease of our clock frequency, resulting in a slower system. The latency and throughput of the floating point hardware is shown in *Table 1.*

Floating point hardware	Latency (cycles)	Throughput (results/cycle)
ADD	14	1
SUB	14	1
MULT	11	1

Table 1

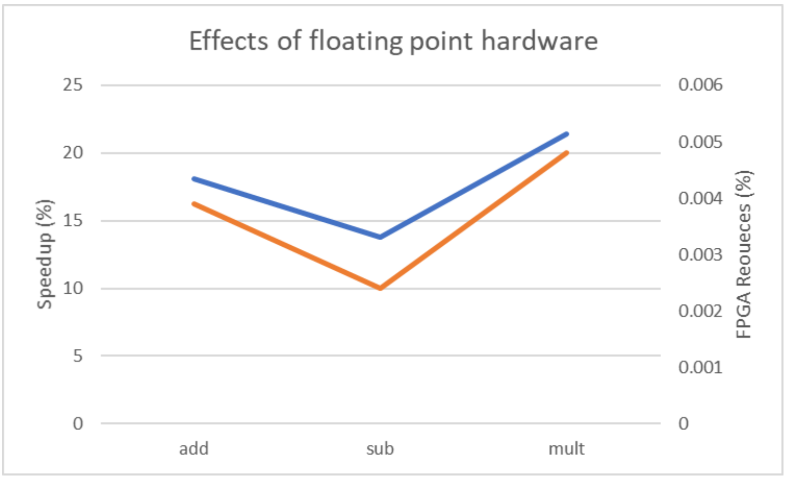


Figure 1

Figure 1 shows the results of the floating point hardware after integrating it with the nios processor. It shows that all the floating point hardware results in a decrease in latency as expected, giving speedups between 13.7% to 21.4%. The multiplication hardware also provides the largest speedup which is expected because it is the most performed operation in the function being evaluated. The resources used by each block are also small relative to the full design which uses 11% FPGA resources.

We then combine all the fp instructions and compare them to 2 designs without floating point hardware, one using the cos function provided in math.h and the other using a software lookup table. The results are shown below:

Implementation	Resource (%)	Latency (ms)	Code Size (byte)	Error (%)
Math.h cos function	9.67	25155	86992	0
Software lookup table	9.67	10478	94184	0.087521
Floating point hardware	11.33	18566	34032	0

Table 2

Table 2 shows that the floating point implementation is much faster compared to the version with the math.h cos function but slower compared to the software-lookup table. This shows that the cos function is a clear bottleneck of the current design. However the software-lookup table version requires a large code size to store the table values and also has a large error in the result.

Adding CORDIC Hardware

A rolled and unrolled CORDIC architecture are implemented in the following section. A monte carlo simulation with 10000000 runs is first run in matlab to find the number of CORDIC iterations required to achieve a mean square error of <1x10^-10. The code for the simulation can be found in the appendix. *Table 3* shows the results of the monte carlo simulation:

Iterations	16
Mean-square error	0.8458×10^{-10}
95% Confidence interval upper end-point	0.8464×10^{-10}
95% Confidence interval lower end-point	0.8450×10^{-10}

Table 3

The CORDIC computation will be done in fixed point for both designs to avoid the need to use floating point hardware, since 16 iterations of floating point multiplications and additions/subtractions will be very expensive. The CORDIC rotations will also be implemented as shifts to reduce the latency of each iteration. The fixed point format used has 2 integer bits to represent the valid inputs of [-1,1] and 16 fractional bits because 16 iterations gives the same precision.

The designs for both the CORDIC implementations are shown and evaluated below:

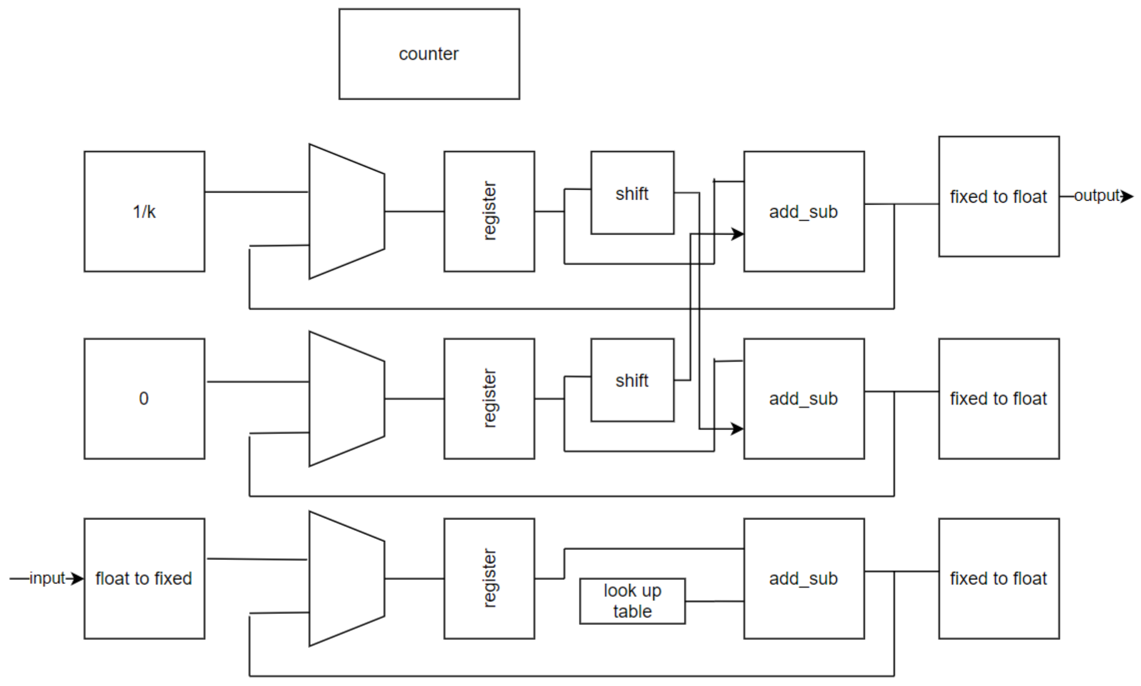


Figure 2

Figure 2 shows a rolled CORDIC architecture, the counter above is used to keep track of the current iteration and the 1/k is the pre-calculated scaling factor required for 16 iterations of CORDIC. Since this design feedbacks its data through the same hardware block, it uses minimal resources, however it has no potential to be pipelined. The architecture has a latency of 30 cycles and a throughput of 0.033 results/cycle.

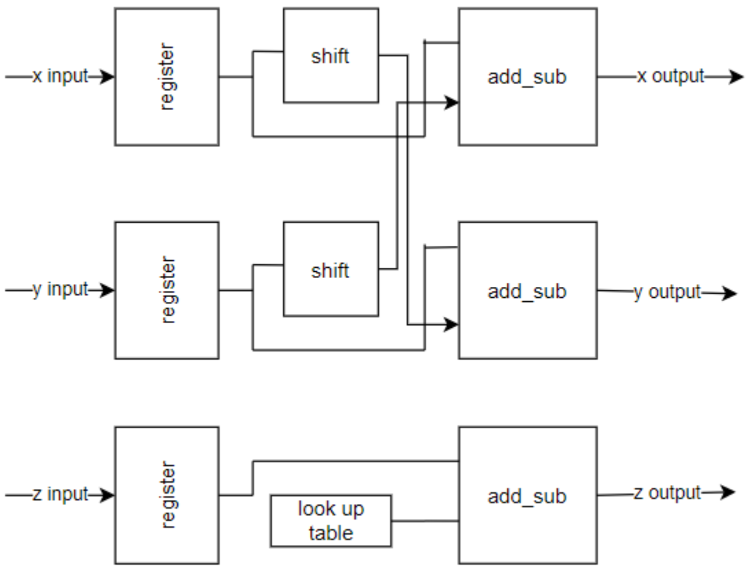


Figure 3

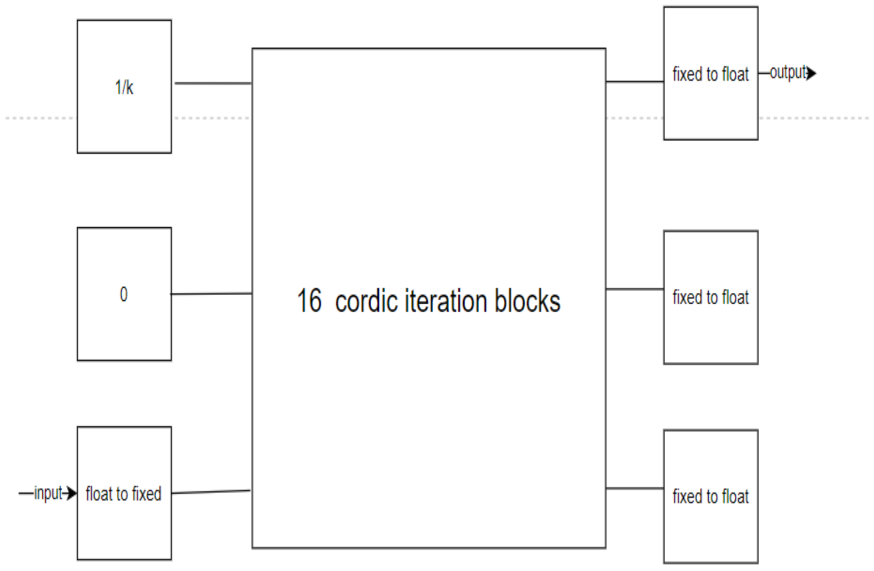


Figure 4

Figure 3 shows a single iteration of CORDIC which is used in the design shown in *Figure 4* to give an unrolled CORDIC architecture. The architecture has a latency of 30 cycles but a throughput of 1 results/cycle since it is pipelined.

The inputs of both architectures are passed into the block through the nios processor's custom instruction master and after a specified latency time, the result will be returned. However, the rolled version also requires a start signal from the nios processor to begin the counter.

The error in table 4 is found using the random test case.

Architecture	Resource (%)	Latency (ms)	Code Size (Byte)	Error (%)
Rolled CORDIC	11.67	1380	75044	0.000012
Unrolled CORDIC	12	1385	75044	0.000011

Table 4

Table 4 compares the 2 architectures, showing that the rolled CORDIC uses less resources as expected. Although the throughput of the unrolled architecture is 1, the latencies of the 2 architectures when integrating with the nios processor are similar, this is because the nios processor blocks when calling custom instructions until the result is available and does not take advantage of the pipeline. The CORDIC designs also introduce errors in the result as expected.

Although the 2 implementations are similar and the rolled implementation uses less resources, the following designs will continue using the unrolled version since the goal is to minimize latency, and it has more potential to increase throughput via pipelining.

Mapping the whole function to hardware

Using the unrolled CORDIC design and floating point hardware blocks, the whole function is then mapped into hardware as a custom instruction.

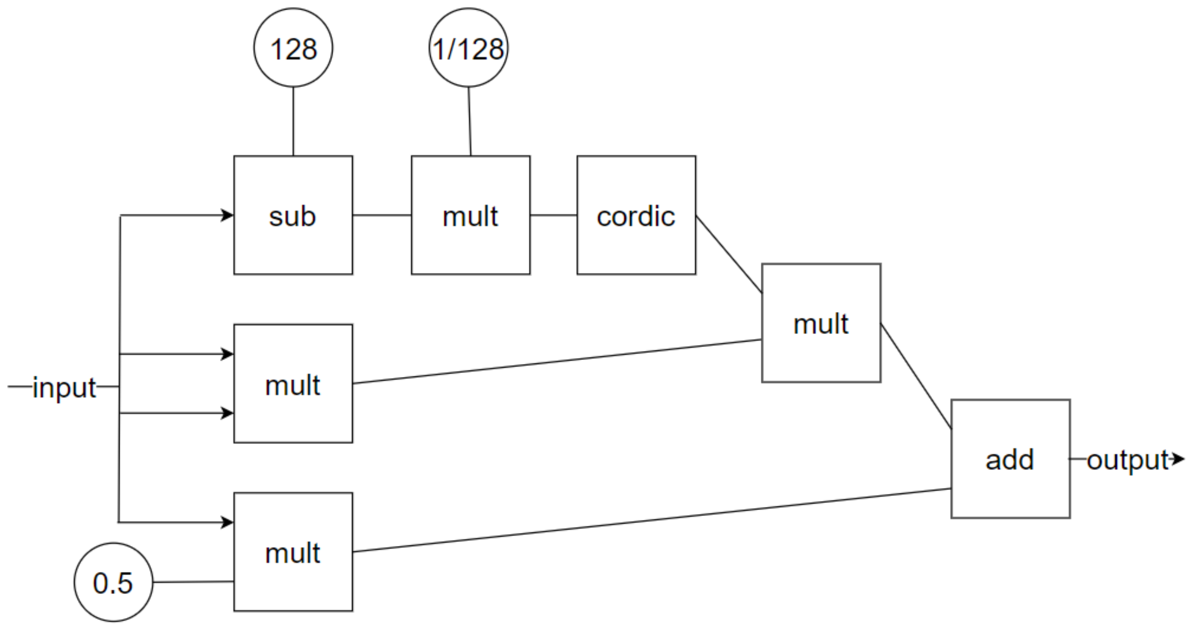


Figure 5

Note: more registers would need to be added to the design to sync the arrival of data at the last mult and add, however since we will not be implementing DMA to pipeline the block due to timing constraints of the coursework, the design does not have the extra registers.

We can see in *Figure 5* that the design also introduces some parallel computation, with the *0.5x* and *x^2* calculated alongside the *x-128* instead of sequentially. This results in a latency decrease equivalent to 2 floating point multiplications or 22 cycles. The entire hardware block has a latency of 80 cycles and a throughput of 0.0125 results/cycle.

Resources (%)	13.67
Latency (ms)	764
Code Size (bytes)	74876
Error (%)	0.000011

Table 5

Table 5 shows the results of the functions integration with the nios processor, the latency decreases as expected, and the file size also decreases because less custom instruction calls are needed. It is also worth noting that the resources increased relative to the original implementation without floating point hardware by 4%.

Parallel function evaluation

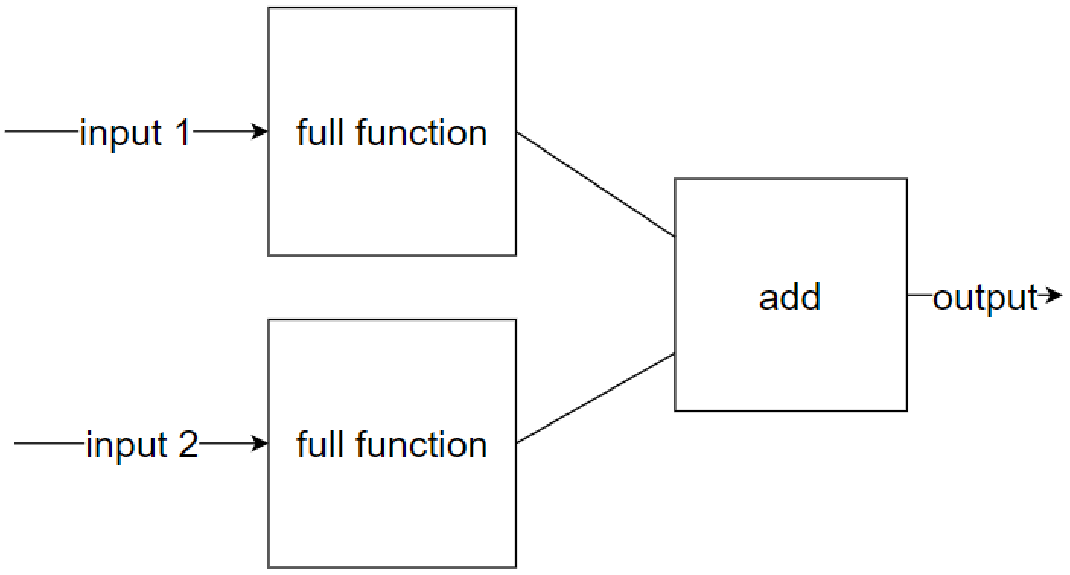


Figure 6

Noticing that the data for the function is completely independent, a design calculating 2 iterations of the function in parallel is implemented as in *Figure 6*. This new design had a latency of 94 cycles and a throughput of 0.0213 results/cycle.

Resources (%)	17.67
Latency (ms)	461
Code Size (bytes)	74736
Error (%)	0.000025

Table 6

Table 6 shows the results of the architecture's integration with the nios processor, the resource usage again increases by 4% as expected, since the architecture simply duplicates the function hardware and adds an adder which is negligible. The latency has decreased as expected since the throughput of the parallel block is higher. However the error has doubled, it is unclear why this is the case since simulating the hardware gives no error.

Conclusion

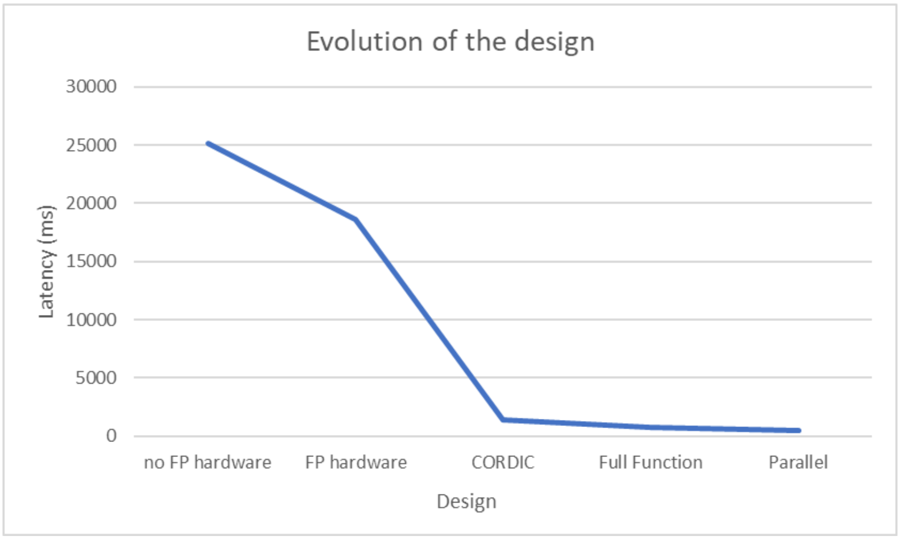
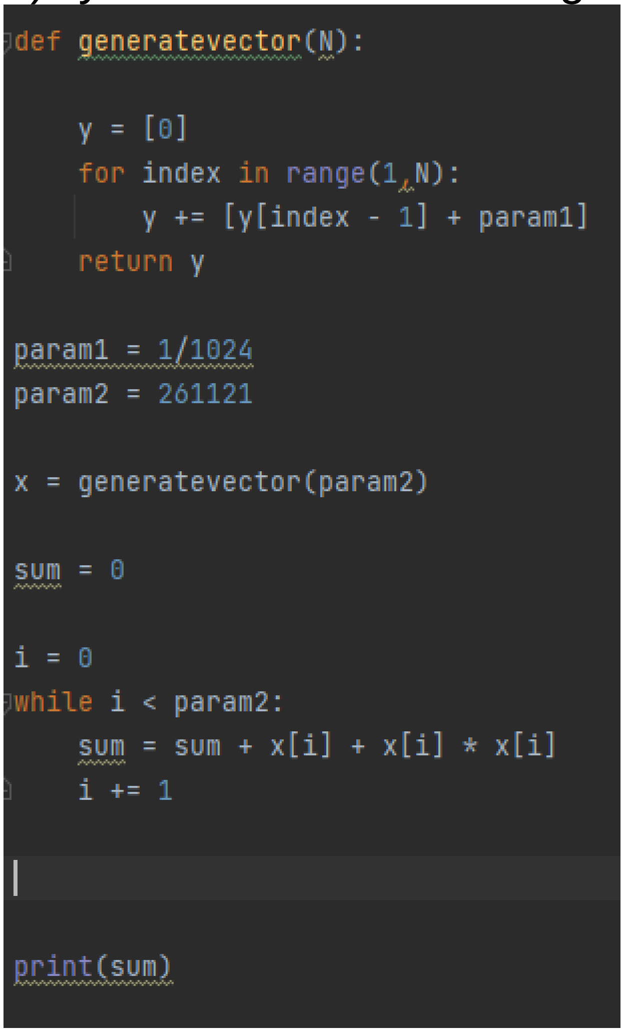


Figure 7

Figure 7 shows that the initial design's latency of 25155ms has been improved to 461ms, decreasing by 98%. This improvement was achieved using only 8% extra resources, this shows that mapping specific functions to hardware is very efficient and FPGAs are a great tool for performing specific computations.

Appendix
1)Python code for checking error



```python
def generatevector(N):

    y = [0]
    for index in range(1,N):
        y += [y[index - 1] + param1]
    return y

param1 = 1/1024
param2 = 261121

x = generatevector(param2)

sum = 0

i = 0
while i < param2:
    sum = sum + x[i] + x[i] * x[i]
    i += 1



print(sum)
```

2)Matlab script for monte carlo simulation

```matlab
iterations = 16;
K = 1;
error = 0;

for index = 0:iterations-1
    K = K * ((1 + (1/(2^index))^2)^(1/2));
end

e = zeros(1,10000000);

for monte = 1:10000000
    x = 1/K;
    y = 0;
    z = unifrnd(-1,1);
    ideal = cos(z);
```

```
        for index = 0:iterations-1
            if (z > 0)
                x_next = x - y * 1/(2^index);
                y_next = y + x * 1/(2^index);
                z_next = z - atan(1/(2^index));
            elseif (z < 0)
                x_next = x + y * 1/(2^index);
                y_next = y - x * 1/(2^index);
                z_next = z + atan(1/(2^index));
            else
                x_next = x;
                y_next = y;
                z_next = z;
            end
            x = x_next;
            y = y_next;
            z = z_next;
        end
        e(monte)=(x-ideal)^2;
        error = error + (x - ideal)^2;
    end

    error = error/10000000;

    fprintf("Scaling Parameter: %.9f\n", K);
    fprintf("Ideal Result: %.9f\n", ideal);
    fprintf("CORDIC Result: %.9f\n", x);
    fprintf("MSE: %.15f\n", error);
    fprintf("CORDIC x: %.9f\n", x);
    fprintf("CORDIC y: %.9f\n", y);
    fprintf("CORDIC z: %.9f\n", z);

    fprintf("confidence interval: Upper %.20f, Lower
    %.20f\n",error + 1.96 * std(e)/sqrt(10000000),error -
    1.96 * std(e)/sqrt(10000000));
```