Report 3

EE3.05 Digital System Design

Timo Thans[1] [01668411], **Omar Tahir**[2] [01342470] (Group 6)

1. Department of Electrical and Electronic Engineering, Imperial College London
2. Department of Computing, Imperial College London

April 1, 2020

1 Introduction

The goal of the course is to configure an FPGA in the most efficient and performant way to evaluate a given mathematical function. In the first part of this report we upgrade our system to included dedicated hardware blocks for floating-point addition and multiplication. In the second part we design and implement custom hardware blocks for evaluating the cosine function, which uses the floating-point blocks from the preceding section. Finally, we combine these into a dedicated block specifically for computing the function on an entire vector at once.

In reports one and two we have designed a digital system that is capable of computing the function. We did some optimizations as shown in the Fig 1. In this report we will elaborate on these results.

	Technique	Elaboration
1	Benchmark	Default design that resulted from following the steps in task 1, 2 & 3
2	Lookup Table	Store the values of the cosine
3	Data Cache	Adding a data cache to store the cosine values
4	Cache configuration	Increased the size of the instruction cache and the block size of both data and instruction cache
5	Embedded Multiplier	Using Embedded multipliers
6	Logic Elements	Using Logic Elements

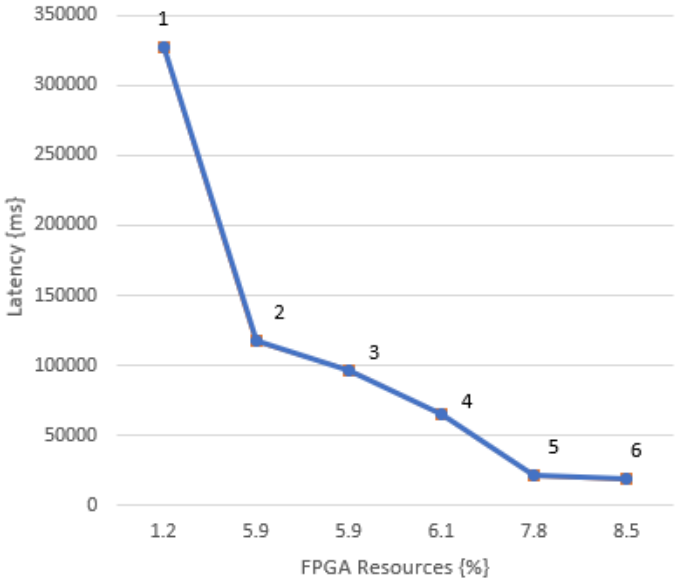

Figure 1 – Design Proces reports 1 & 2

Our best design so far had a latency of 18,654 miliseconds and used 7.8 % of the FPGA resources.

2 Analysing the problem

The function we are required to evaluate is

$$y = \sum_{i=0}^{N-1} \frac{1}{2} x_i + x_i^2 \cos\left(\frac{x_i - 128}{128}\right). \tag{1}$$

Since x is a vector of floats, every arithmetic operation done while calculating y is a floating-point operation. Therefore, having dedicated hardware blocks for floating-point operations will greatly increase the performance of the system, at the expense of some resources.

Firstly, we can ignore implementing a floating-point subtraction block, because $a - b = a + (-b)$ and for a 32-bit floating-point number b, $-b = b \wedge (1 << 31)$ so a dedicated hardware block is not necessary.

We also don't need any conversions between integers and floating-point numbers, because the input x is already specified to be a vector of 32-bit floating-point numbers and the constants 128 in eq. (1) are simply constant floating-point literals. The result y should also be a float, so there is no rounding or conversion necessary the other way either.

The last function needed is cos. There are many different ways to implement this function in hardware, but the most common is CORDIC, which uses floating-point adders only.

In conclusion, we only require hardware for floating-point addition and multiplication.

3 Hardware floating-point units

In this section we will discuss our implementation of hardware floating-point units for addition and subtraction, and their performance.

We have used the floating point multiplier from the IP catalog and created a Verilog file from this design, this is a combinatorial instruction. The FPGA resources increased by 2.95 % point when the HW block was implemented. Unfortunately due to some software and hardware bugs a lot of time was spent on this step. The two difficulties were that the compiler did not produce the correct custom instruction in the *system.h* and the second was that the UEFI file could not be uploaded to the board. The first problem was solved by adding the line from listing 1. As the lecturer Dr. Christos Bouganis is aware by now, for the second problem we had to follow a few arbitrary steps as was discovered by two fellow students. The steps can be found in appendix A.

```
#define ALT_CI_FP32MULT_0(A,B) __builtin_custom_fnff(ALT_CI_FP32MULT_0_N,(A),(B))
```

Listing 1 – Custom instruction of floating point multiplier

	Descritpion	Latency	Error	FPGA Resources
7	Usage of one FP multipliers	1 426 ms	0.0009 %	10.75 %
8	Usage of two FP multipliers	617 ms	0.0009 %	10.75 %
9	A look-up table is used for the cosine	380 ms	0.6428 %	10.75 %

Table 1 – Evaluation of floating point multiplier

4 Hardware implementation of cosine

In this section we will discuss the design of our hardware block for computing cosine and compare it to other algorithms.

The CORDIC algorithm is widely used for computing trigonometric functions and was suggested in the tutorial and lectures. The complete Verilog file of implementing CORDIC is shown in appendix B. I will describe the critical steps here and show Verilog code snippets where needed.
The first thing we need to realize is that the CORDIC algorithm for cosine can be stripped down to two basic formulas with i iterations. We do 32 iterations, which is the same as our input bit width as doing more would not increase the resolution.

$$X_{i+1} = K_i[X_i - d_i Y_i 2^{-i}] \tag{2}$$
$$Y_{i+1} = K_i[Y_i + d_i X_i 2^{-i}] \tag{3}$$
$$\theta_{i+1} = \theta_i - d_i atan(2^{-i}) \tag{4}$$

```
X[i+1] <= Z_sign ? X[i] + Y[i]_shr       : X[i] - Y[i]_shr;
Y[i+1] <= Z_sign ? Y[i] - X[i]_shr       : Y[i] + X[i]_shr;
Z[i+1] <= Z_sign ? Z[i] + atan_table[i] : Z[i] - atan_table[i];
```

Listing 2 – CORDIC: calculation of cosine

In Verilog we can automatically pipeline these iterations in hardware using a for loop. This will generate logic for every iteration. Equations 1 and 2 are shown in listing 2. The values for the `atan_table` are shown in appendix B. The name table however is a bit misleading here as when synthesised each pipeline stage is hardcoded with its own constant, and the table itself is never synthesised into hardware as it is never accessed outside this loop. Further more 2^{-i} is very easily implemented in Verilog as a number with bits shifted to the right as done in `X[i]_shr` and `Y[i]_shr`.

In our function the cosine is only used for calculating values in the first and fourth quadrants. In these quadrant no pre-rotation is needed. However for the algorithm to work in general we also need to rotate the data if it is in the second or third quadrant. An example of how we have done this in Verilog is shown in listing 3.

```
2'b10: //pi to 3/2 pi quadrant so pre rotation is needed
begin
        X[0] <= Yin;
        Y[0] <= -Xin;
        Z[0] <= {2'b11,angle[29:0]}; // add pi/2 from angle
```

Listing 3 – CORDIC: rotation

If we leave out K in the iteration it will result in a gain that converges to a value of 1.646. Remember that the CORDIC algorithm is a rotation so the output will be of the form $X_n = A(x\cos(\theta) - y\sin(\theta))$. If we initialize with $Y_0 = 0$ and $X_0 = 1/K = 0.6075$ the output will be the cosine of the angle. The results of our first CORDIC implementation are shown in table 2.

	Latency	Error	FPGA Resources
10	2156 ms	0.0033 %	11 %

Table 2 – Evaluation of CORDIC algorithm

5 Dedicated block for computing inner expression

Now that we have implementation of floating-point addition, subtraction and cosine, we will discuss our design of a block that computes the inner expression under the sigma in eq. (1), namely

$$\frac{1}{2}x_i + x_i^2 \cos\left(\frac{x_i - 128}{128}\right). \tag{5}$$

This block will be used to calculate the above expression for every element of x. We choose to optimise for throughput over latency. By pipelining this design with high throughput, we can increase the latency of task 8 as this expression will need to be calculated for thousands of values in succession.

The first step is to reduce the number of operations necessary, so we can rewrite eq. (5) as

$$x_i \left(0.5 + x_i \cos\left(2^{-7} x_i - 1\right)\right). \tag{6}$$

We now introduce ADD, MUL and COS as 32-bit floating-point addition, multiplication and cosine respectively, whose implementations were discussed in prior sections. We also introduce a new operation DIV which divides a 32-bit floating-point number by a power of two. This is trivial to implement in hardware as it is simply a right shift on the exponent, with a simple check for exponent overflow which should instead result in $\pm\infty$. The final form of our expression is now

$$\mathrm{MUL}\left(x_i,\ \mathrm{ADD}\left(0.5,\ \mathrm{MUL}\left(x_i,\ \mathrm{COS}\left(\mathrm{ADD}\left(\mathrm{DIV}\left(x_i,\ 128\right),\ -1\right)\right)\right)\right)\right). \tag{7}$$

Since ADD, MUL and COS are pipelined with a throughput of one instruction per cycle, and DIV is combinational, the entire block can also be pipelined with a throughput of one instruction per cycle. The results of the inner expression are evaluated in 4

	Latency	Error	FPGA Resources
10	1087 ms	0.0033 %	11 %

Table 3 – Evaluation of inner expression

6 Dedicated block for computing entire expression

For the final task we need to design a hardware block that computes the entire expression in eq. (1). This requires the block to have access to the entire vector $\mathbf{x}$. Our solution uses a custom instruction that can access SDRAM directly.

The instruction takes two inputs: a pointer to the base address of the vector, and the size of the vector (in number of elements). It requires that the vector is in a contiguous array. The elements of the vector are read from SDRAM and entered into the pipeline of the block designed in the previous section, and the result is added to a running total. Once the vector is exhausted, the instruction waits for the pipeline to clear and returns the sum.

This instruction utilises four interfaces: a clock interface, a reset interface, a NIOS custom instruction slave interface, and an Avalon memory-mapped master interface. The clock and reset interfaces are necessary for the Avalon interface as it requires special external (for lack of a better word) interfaces for clock and reset; the GUI-based editor in Quartus refuses to connect it to any existing clock or reset. The custom instruction interface takes a variable number of cycles, so unlike previous instructions this one requires start and done signals. It also requires its own clock signal or the software will not recognise it as a variable-cycle instruction, so it contains a superfluous clock that is unused.

The SDRAM in our top-level design only has a 16-bit data bus, so each element of the vector must be fetched in two passes, first the low half-word followed by the high half-word. This requires a request via the Avalon interface, and we then have to block until we receive a response. After the entire word has been fetched, we check if the end of the vector has been reached. If so, the instruction waits until all outstanding computations have been resolved, and asserts the done signal. If not, the instruction simply fetches the next element of the vector.

The block designed earlier that computes the inner expression has a throughput of one instruction per cycle. The multiplier has the same throughput. However, the expression actually being computed is

$$Q \leftarrow Q + f\left(x_i\right) \tag{8}$$

which is self-referential with Q. Therefore before the next addition can proceed, the first addition must complete in its entirety. The floating-point adder has a latency of two cycles. If one element is added to the pipeline per cycle then we could have the following example scenario:

t	**Q**	**f(x)**	**Pipeline stage 1**	**Pipeline stage 2**
0	0	0	$Q \leftarrow 0 + 0$	$Q \leftarrow 0 + 0$
1	0	1	$Q \leftarrow 0 + 1$	$Q \leftarrow 0 + 0$
2	0	2	$Q \leftarrow 0 + 2$	$Q \leftarrow 0 + 1$
3	1	3	$Q \leftarrow 1 + 3$	$Q \leftarrow 0 + 2$
4	2	4	$Q \leftarrow 2 + 4$	$Q \leftarrow 1 + 3$
5	4	0	$Q \leftarrow 4 + 0$	$Q \leftarrow 2 + 4$
6	6	0	$Q \leftarrow 6 + 0$	$Q \leftarrow 4 + 0$
7	4	0	$Q \leftarrow 4 + 0$	$Q \leftarrow 6 + 0$
8	6	0	$Q \leftarrow 6 + 0$	$Q \leftarrow 4 + 0$

The sum should be $1 + 2 + 3 + 4 = 10$ but because we are adding one instruction per cycle to an operation that operates on its own output and has two cycles of latency, we end up with the incorrect result of 6.

There are two solutions to this problem. One is to reduce the throughput of the entire system to one instruction per two cycles. Another is to observe that half the elements are added on the odd cycles, and half on the even cycles, so we need only sum the results from the final two cycles to obtain our result - this is clear when looking at how the last few rows of the table alternate. In the example, adding the values of Q on the last two cycles, $t = 7$ and $t = 8$, yield the correct final value of $4 + 6 = 10$.

In practice however, the SDRAM limits the throughput as it never responds within a single cycle, so this never actually occurs.

	Latency	Error	FPGA Resources
12	219 ms	0.0033 %	11.9 %

Table 4 – Evaluation of inner expression

7 conclusion

The main goal of this project was to design a digital system that had the best performance with the least amount of resources. The complete design process that we have followed is shown in 2. Our best design had a latency of 219 ms and used 11.9 % FPGA Resources.

	Technique	Elaboration
7	Hardware floating-point units	Usage of one FP multiplier
8	Hardware floating-point units	Usage of one FP multiplier
9	Hardware floating-point units	A look-up table is used for the cosine
10	Hardware cosine unit	Implementation of the Cordic algorithm
11	Dedicated block for inner expression	We introduced ADD, MUL and COS and used them for the expression
12	Dedicated block for entire expression	Using a custom instruction to access SDRAM directly

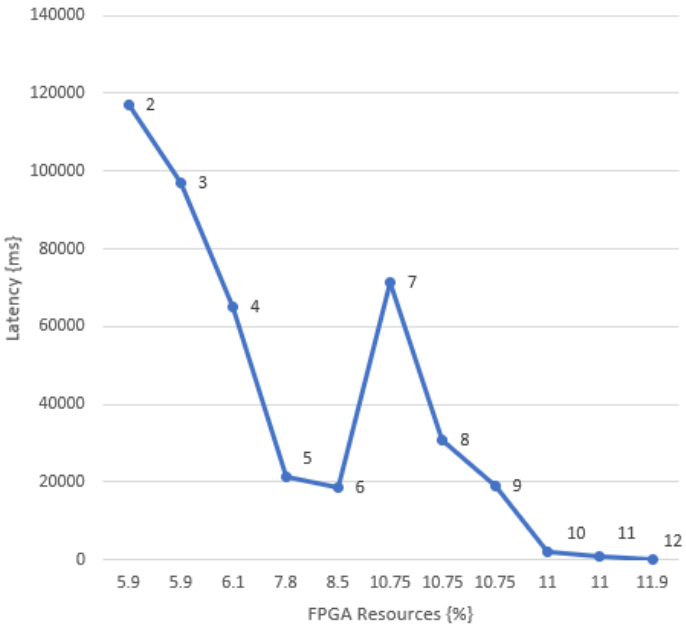

Figure 2 – Complete design proces

The official github repository for this project with all tasks is: https://github.com/TimoThans33/Digital_System_Design.

A Solution for: ERROR UEFI file not uploaded

```
Step 1: Eclipse needs to be on
Step 2: Turn off board and switch USB port
Step 3: Open programmer, the board needs to still be off
Step 4: Turn on board, and upload program as usual.
Step 5: CLOSE PROGRAMMER(It sometimes works without closing it as well, but rather safe than
    sorry)
Step 6: Build BSP, and run configurations!!
```

B arctangent table used for CORDIC

i	2^{-i}	atan(2^{-i})
0	1	45
1	½	26.565
2	¼	14.036
3	1/8	7.125
4	1/16	3.576
5	1/32	1.79
6	1/64	0.895
7	1/128	0.448
...	...	...
30	...	...

Figure 3 – atangent table