Digital Systems Design Report 3

Tsz Hang Wang
thw20@ic.ac.uk
01864937

Bryan Tan
bet20@ic.ac.uk
01861592

March 20, 2023

1 Introduction

In this report, the performance of different hardware accelerator designs were evaluated for the computation of the mathematical function in Eq.1. Evaluation was conducted on commercial Intel floating point IP blocks, the usage of CORDIC blocks and customized arithmetic blocks. The aim is to obtain a hardware accelerator that will compute the target function with minimum execution time and minimum resource utilization.

$$f(\vec{x}) = \sum_{n=1}^{N} (0.5 \times x_i + x_i^2 \times \cos\left((x_i - 128)/128\right)) \quad (1)$$

2 Hardware Floating Point Units (Task 6)

2.1 Custom Instruction Interface

Given that input X in Eq.1 is an array of floating point numbers, the mathematical arithmetic should be performed in a floating point format. One approach to mapping these operations with hardware would be to utilize floating arithmetic blocks (FP-block) provided by Quartus.

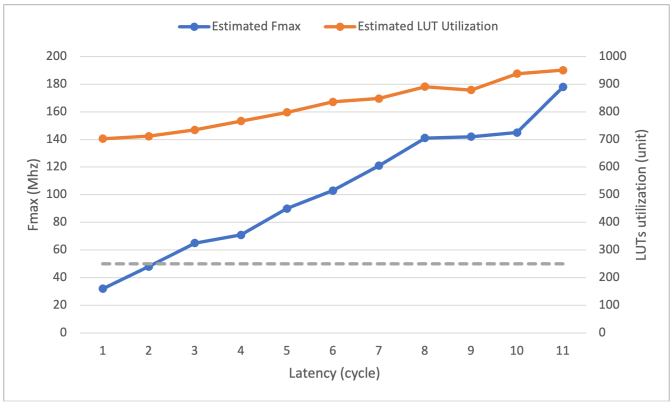
To enable the communication between NIOS2 CPU and floating arithmetic blocks, we can use the custom instruction interface provided in the platform designer, by configuring the FP-block as custom instruction slaves and connecting them to the master post of the NIOS2 CPU.

Depending on the timing of the dedicated hardware such as being combinatorial or multiple cycles circuits, the custom instruction interface provides several types of custom instructions that come with different input and output ports to allow more control from the CPU.

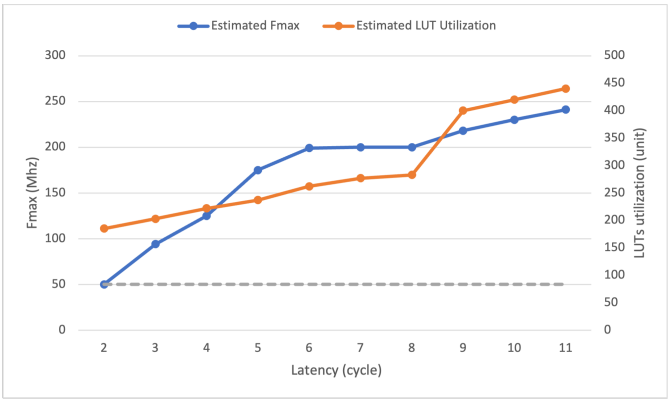
Once the system is produced, specific functions corresponding to the custom instruction slave are generated in system.h file. The CPU can then invoke the custom instruction blocks by calling these functions in software and passing the input by argument. Noticing that these instructions are blocking, the CPU will have to wait until the function returns the result.

2.2 Floating point arithmetic block

Quartus provides some built-in FP-blocks that enable floating point summation and multiplication with adjustable input/output ports, data width, and latency. Fig.1 illustrates the estimated LUT utilization of FP_ADD/SUB and FP_MULT blocks with different latency. The table shows there is a correlation between low latency and the upward trend of LUT utilization. This is because the built-in blocks are pipelined blocks and extra logic is needed to manipulate and store the data at each stage, and more logic is required to control the data synchronization between stages. In addition, as the number of cycle required to achieve the result increased, each cycle can perform less computation, reducing the critical path of the block. As a result, blocks with lower latency can be driven by a higher clock frequency.



(a) FP_ADD/SUB block



(b) FP_MULT block

Figure 1: Estimated LUT utilization and estimated Max clock frequency(Fmax) of built-in floating point arithmetic blocks

Without adding any software logic, NIOS2 CPU can not execute any pipeline design by interacting with a custom instruction interface and all operations are done sequentially. Therefore, to achieve the lowest overall latency, all FP-blocks were set to their lowest latency configuration that can operate under the assumed operation clock of the CPU(50MHz). As a result, FP_ADD/SUB block with a latency of 3 cycles and FP_MULT block with a latency of 2 cycles were selected as our baseline model.

To verify the correctness of the model, test benches targeting various block types were developed. These test benches compared the produced output from the targeted block with the expected output obtained from the software. To validate the timing of the model, ModelSim is utilized to execute the test bench and generate the clockwise waveform of the block interface signals.

Table.1 illustrates the FPGA resource utilization for designs utilizing FP-blocks and fixed-point multipliers. The Table shows an increase in LUT and register utilization for design with FP-blocks. This is because more complex logic is required to compute numbers with scientific notation as it involves more steps, and more memory is needed to store extra bits for the exponent field and sign compared to the 16-bit multiplier used in task 5. It is worth noting that depending on the latency of the instantiated FP-MULT blocks, different numbers of hardware multipliers are needed. The FP-blocks designs utilized FP-MULT with a latency of 2 cycles which requires 1 multiplier. As a result, an additional 1 multiplier

is seen compared to task 5.

	Baseline model	Software emulation fixed-point unit (Task 5)
Logic utilization (in ALMs)	2048 / 32,070 (6%)	1634/ 32,070 (5%)
Total Register	3121	2678
Total block memory bits	28%	28%
Total DSP Blocks	4 / 87	3 / 87
Total PLLs	1 / 6	1 / 6

Table 1: Resource Utilisation Report: 64KB I-cache, 64KB D-cache, SDRAM, Built-in FP-blocks and Software emulation with fixed-point unit

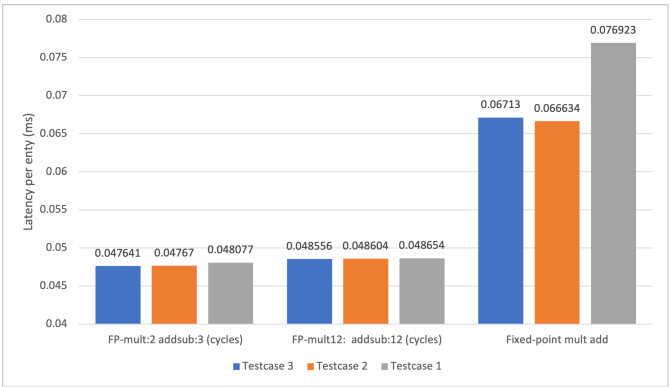


Figure 2: Latency per entry across test cases under different designs

Fig.2 shows the latency per entry for three test cases of the baseline model, the model utilizing FP-blocks with latency, and the model from the previous task. A significant improvement in latency was achieved by both models utilizing floating-point hardware compared to the fixed-point arithmetic unit utilized in the previous task. As mentioned in section 2.1, since all the custom instructions are blocking. The function calls shown in List.1 are performed in a sequential manner and given the blocks are driven under the same clock frequency(50Mhz), minimizing the latency of each operation leads to the minimum overall latency based on the greedy algorithm. Consequently, the baseline model shows a slight improvement over the model with larger latency. In addition, the results of all test cases achieved by using FP-blocks are the same as those using software emulation with fixed-point operators. The application size achieved by all designs is the same.

FP	FP_ADD/SUB: 3	FP_ADD/SUB: 12
Logic utilization (in ALMs)	2048 / 32,070 (6%)	2152/ 32,070 (7%)
Total Register	3121	3947

Table 2: Resource Utilisation Report: 64KB I-cache, 64KB D-cache, SDRAM, Built-in FP-blocks with different latency

```
const float recip128 = 1.0/128.0;
```

```
float cosf_fp(x)
{   return cosf(
        FP_MULT(FP_ADD_SUB(0,x[i],128), recip128
    ));}

float functionX(float x[], int M)
{
  int i;
  float y = 0;
  for(i=1; i<M; i++){
    y = FP_ADD_SUB(1,y,
       FP_MULT (x[i],
                (FP_ADD_SUB (
                1,0.5,
                FP_MULT(x[i],
                cosf_fp(x[i])))))
        )
    );
  }
  return y;
}
```

Listing 1: Sequentially calling custom instruction

3 CORDIC Implementation (Task 7.1)

The Coordinate Rotation Digital Computer (CORDIC) algorithm was used to implement the cosine term of the inner function in hardware. The aim is to design to two versions of the block - one optimised for latency and one for throughput. The blocks must meet the target frequency of 50 MHz.

3.1 CORDIC Algorithm

The key idea behind the algorithm is to rotate a unit length vector by an angle z - the resulting x and y components of the vector will be $cos(z)$ and $sin(z)$.

By instead performing a fixed-length sequence of pseudo-rotations (which rotate the vector but also scale its length) on a length-corrected vector, only addition and bit-shift operations are required in the computation, making it suitable for an efficient hardware implementation.

In CORDIC rotation mode, the arithmetic operations required for one pseudo rotation are described by Eq.2. x, y are the components of the vector, and z is the remaining angle left to rotate. Three addition/subtractions are required, as well as one lookup table operation to obtain $tan^{-1}(2^{-i})$.

$$x_{i+1} = x_i - d_i \cdot 2^{-i} \cdot y_i \tag{2a}$$
$$y_{i+1} = y_i + d_i \cdot 2^{-i} \cdot x_i \tag{2b}$$
$$z_{i+1} = z_i - d_i \cdot tan^{-1}(2^{-i}) \tag{2c}$$
$$d_i = sign(z_i) \tag{2d}$$

The initial vector lies at an angle of zero and has length $1/K$, where K is the length of the unit vector after applying all n pseudo-rotations. Increasing the number of pseudo-rotations helps z converge to 0 and improves the accuracy of the final result.

$$x_0 = 1/K \tag{3a}$$
$$y_0 = 0 \tag{3b}$$
$$z_0 = z \tag{3c}$$
$$K = \prod \sqrt{1 + 2^{-2i}} \tag{3d}$$

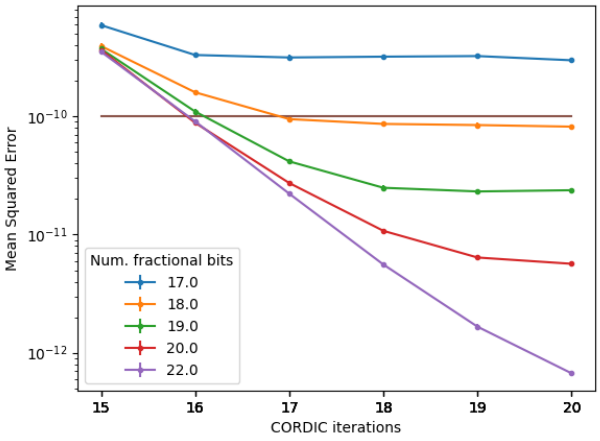


Figure 3: The MSE achieved by different parameterisations of coscordic.

3.2 Iteration Count and Word Length Optimisation

It is efficient to perform the CORDIC algorithm on fixed point numbers. Therefore, the overall accuracy of a CORDIC block is a function of the word length of the fixed point representation and the number of pseudo-rotations, forming a 2D parameter space.

The specification states that our hardware cosine block must achieve a mean squared error of less than 10^{-10} with a confidence level of 95%, assuming the input follows a uniform distribution in the range [-1, 1].

It is inefficient to generate and evaluate a hardware implementation for each design in the parameter space, therefore we estimate the error of a given design using a software model. We used the MATLAB `coscordic` function, which can be parameterised on the number of CORDIC iterations to perform, and the number of signed, integer and fractional bits in the fixed point representation. The range of cosine is [-1, 1], so one signed bit and one integer bit is used for all designs. A grid search was performed over the remaining two parameters. Each design was evaluated using a Monte Carlo simulation with 2000 samples in the range [-1, 1]; the reference output used is the double precision result produced by the MATLAB `cos` function.

The results of searching can be seen in Fig. 3. As expected, MSE decreases with more CORDIC iterations and fractional bits used. For any given word length, increasing the number of iterations only decreases the error up to a certain point - after which the accuracy is limited by the precision. Note the upper-bound value of the 95% confidence interval has been plotted as an error bar in the graph, but is too small to be visible.

The two designs which satisfy the specification with the least resources use 20 fractional bits and 16 iterations (C.I. Upper Bound: 8.84e-11), and 18 fractional bits and 17 iterations (C.I.U.B.: 9.50e-11). The former design is favourable, as it achieves a lower error and 16 rotations is evenly divisible by many numbers (making it easier to design the hardware such that the delay paths are balanced between different stages).

Thus, the experiment suggests to use a fixed point representation with 22 bits in total: 1 signed bit, 1 integer bit and 20 fractional bits. We further reasoned that the integer bit may be unnecessary. The range of a 1 signed bit and 20 fractional bits fixed point representation is [-1, $1 - 2^{20} \approx 0.99999904632$]. The representation may be used if all the intermediate values within the CORDIC block fall in this range.

First we consider the $tan^{-1}(2^{-i})$ terms. They are a decreasing sequence of positive angles starting from ≈ 0.7854 radians, so they are always in the range. The vector being rotated initially has a length of $1/K \approx 0.60725$ and converges to 1 or -1 over the course of the pseudo-rotations. Being the y component, y_i will always be bounded by the range [-1, 1]. It will only be 1 if the input angle is $\pi/2$, but the input range has been constrained to [-1, 1] so this will never happen. z_i is the running count of the angle left to rotate - it converges from the input angle to 0 over the rotations. Therefore, it has the same range as the input angle - [-1, 1]. An input angle of 1 rad is therefore a problematic input, but we can detect this and return the correct pre-calculated output. Finally x_i is the x component of the rotating vector, and also the desired cosine output. In the input range, the cosine output can be in the range [cos(1) ≈ 0.54302, 1]. Therefore x_i will only be 1 if the input angle is 0 - another problematic input which we can correct.

To fully verify our hypothesis, we implemented the two versions of the CORDIC block in Verilog, one with 22 bits and one with 21 bits. Over a 5000 random input samples uniformly distributed in [-1, 1], we verified that the outputs from both blocks were identical. The MSE of the block is 8.84e-11, with a 95% confidence interval upper bound of 9.13e-11. This is higher than the MATLAB model estimate, but still meets the specification.

3.3 CORDIC Hardware Implementation

The top level view of an unpipelined CORDIC block is shown in Fig. 4. The input floating point number is first converted into fixed point, then k rotations are applied

iteratively every cycle until the correct result can be converted back to the floating point output.

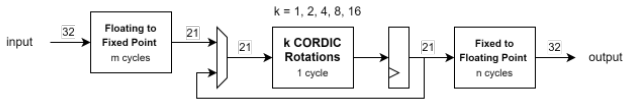


Figure 4: Top Level CORDIC block

The k rotation block is a combinatorial block comprised of k separate sub-blocks that perform 1 CORDIC rotation each (Fig. 5). To minimise the total output latency of the block, we wish to maximise k - the number of rotations performed in 1 clock cycle - while still meeting the target frequency.

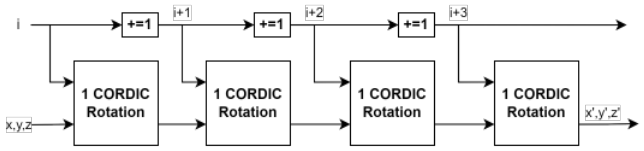


Figure 5: $k = 4$ Rotation CORDIC block

The 1 CORIDC rotation block (Fig. 6) implements the operations described by Eq. 2. The inputs to the block are x_i, y_i, z_i: the components of the vector after the ith rotation and the total angle left to rotate. The outputs are the components of the rotated vector. The LUT is indexed by the top bits of i, and stores the angles that the block will rotate the vector by.

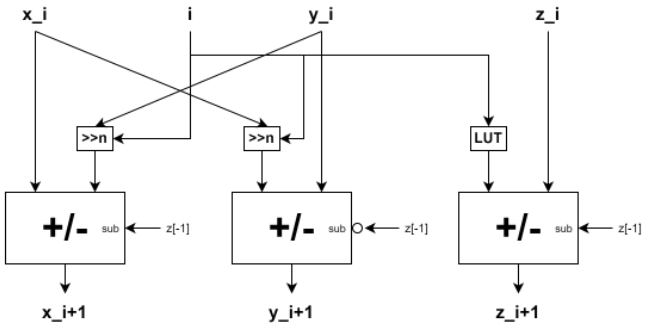


Figure 6: Single Rotation CORDIC block

Two versions of the CORDIC rotation block were investigated. In the first design (List. 2), an if statement was used to select whether to perform an add or a subtract. The LUT for each block also stored all 16 possible rotation angles, which we later noticed was more than needed.

```
if(z[20]==0) begin
    rot_x = x - (y >>> i);
    rot_y = y + (x >>> i);
    rot_z = z - lut_angles[i];
end else begin
    rot_x = x + (y >>> i);
    rot_y = y - (x >>> i);
    rot_z = z + lut_angles[i];
end
```

Listing 2: CORDIC Rotation Naive Instantiation

In List. 3 , the two's complement negation operation was explicitly written out as a bitwise negation using XOR gates and an addition by 1 bit. The LUT was also changed to only store the needed number of angles (e.g. if the block is 2nd in a total of 4, it performs the 2nd, 6th and 10th rotations and thus only stores those angles), and is indexed by the top bits of the rotation count i. The number of angles stored by each LUT therefore decreases with increasing values of k.

```
z_replicated = {21{z[20]}};
y_shift = (y >>> i);
x_shift = (x >>> i);
rot_x = x + (y_shift ^ ~z_replicated) + !z[20];
rot_y = y + (x_shift ^ z_replicated) + z[20];
rot_z = z + (lut_angles[i[3:2]] ^ ~z_replicated)
          + !z[20];
```

Listing 3: CORDIC Rotation Explicit Instantiation

3.4 Fixed/Floating Point Conversion

The floating-fixed point converter IP blocks available in Quartus have an output latency of 6 cycles at the 50 MHz target frequency. We noticed that the input domain of these blocks is very constrained in our CORDIC block, which could simplify the needed hardware. Therefore, we implemented our own versions of the blocks to achieve a better output latency.

Fig. 7 is the converter from floating point to our chosen fixed point format (1 signed bit, 0 integer bits, 20 fractional bits). The input range is [-1, 1], so the exponent will always be less than or equal to 127, and we will only ever need to right shift the mantissa. Cosine is an even function, so the sign of the floating point number is ignored. A special case input is 1.0, which will be converted to -1 in our fixed point representation (top bit set, others all zero). This is desired behaviour as explained in Section 3.2. Synthesizing the block gives an Fmax of 276.5 MHz, fast enough to keep as a combinatorial block.

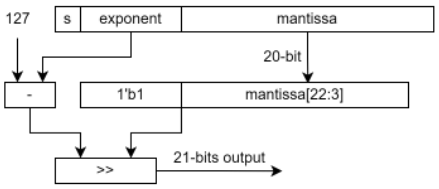


Figure 7: Custom Floating to Fixed Point Converter block

Fig. 8 is the converter from the fixed point format back to the floating point format. Since the result obtained from the CORDIC rotation will always be a positive number, no hardware is required to take its absolute value. The result is passed to a priority encoder (we adapted a design from [1]), which determines the location of highest 1 bit in the binary format of the result. Based on the priority encoder output, corresponding left shift is performed to remove all the 0 in front and form the upper 20 bits of the significant slot. The exponent slot is calculated by subtracting the 127 bias from the priority encoder output. Synthesizing the block gives an Fmax of 143.4 MHz, so it was decided to buffer the output with one register stage,

giving the block 1 cycle output latency. However, by carefully balancing the critical paths in the design, it may have been possible to merge the converter with another stage.

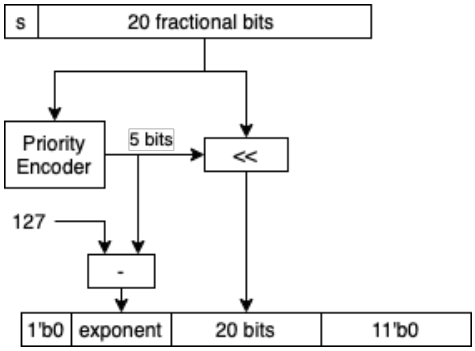


Figure 8: Custom Fixed to Floating-point Converter block

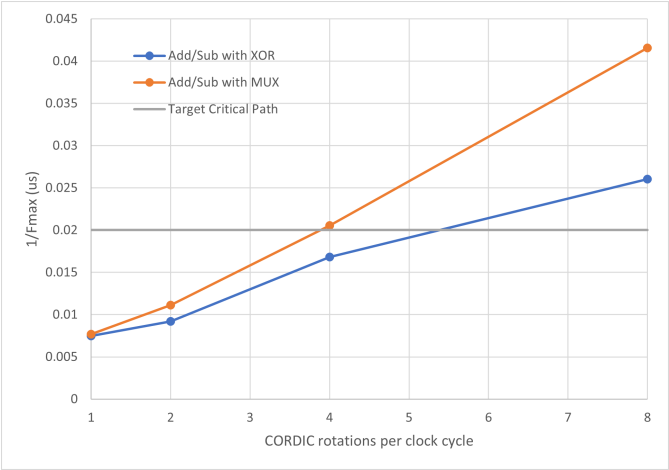


Figure 9: The Critical Path of Different CORDIC blocks. The value of Fmax reported is for "Slow 1100mV 0C" (the worst case condition)

	Fmax (MHz)	ALMs
CORDIC 1 Rotation	253.87	76
Floating to Fixed	276.49	43
Fixed to Floating	143.39	51

Table 3: Error and Latency achieved by Task 8

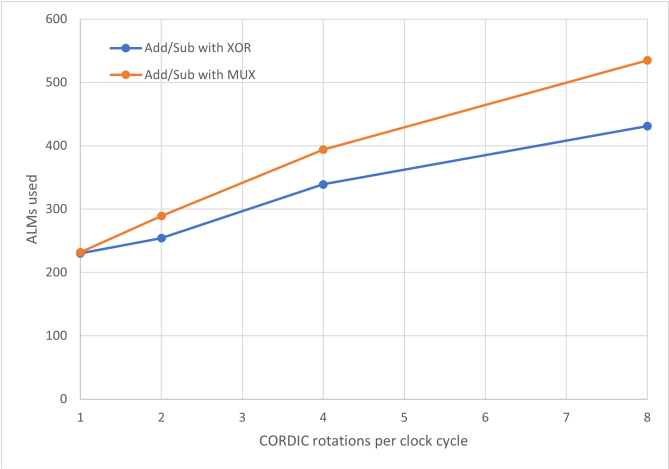


Figure 10: ALM Usage of Different CORDIC blocks.

3.5 Synthesis Results

The top level CORDIC block with different values of k was synthesized in Quartus. The target frequency of the Fitter was set to 1 GHz to obtain the maximum potential frequency for each design. For Timing Analyzer to produce an accurate estimate of the critical path, the block was placed in a wrapper block to ensure all paths started and ended at a register component.

Fig. 9 and 10 show that the critical path and the resource usage of the block increases with more rotations performed per cycle. The List. 3 instantiation outperforms List. 2 at all points. This suggests that the compiler is not able to automatically optimize List. 2 to an adder subtractor block, and instead synthesized it using a multiplexer and two separate adders.

In the List. 2 design, the critical path forms an almost perfect linear relationship with the number of CORDIC iterations performed per clock cycle. This makes sense as the 1 CORDIC rotation block used by each value of k is identical, so stacking more blocks in series increases the critical path linearly. In contrast, for the List. 3 design, the relationship appears to be weaker than linear. This suggests that for List. 3, the critical path goes through the LUT, which gets smaller with increasing values of k.

The CORDIC block which performs 4 rotations per cycle is the block with the lowest output latency in clock cycles that still meets timing constraints. Its output latency is 5 cycles (4 rotation cycles, 1 for fixed to floating point conversion).

In this problem setting, because the clock rate is fixed, there is no tradeoff between throughput and latency. To create a block optimized for throughput, we simply pipelined the latency-optimized block by unrolling the rotation loop and adding pipeline registers in between the blocks (Fig. 11). This achieves a throughput of 1 result per clock cycle, and the latency is still 5 cycles.

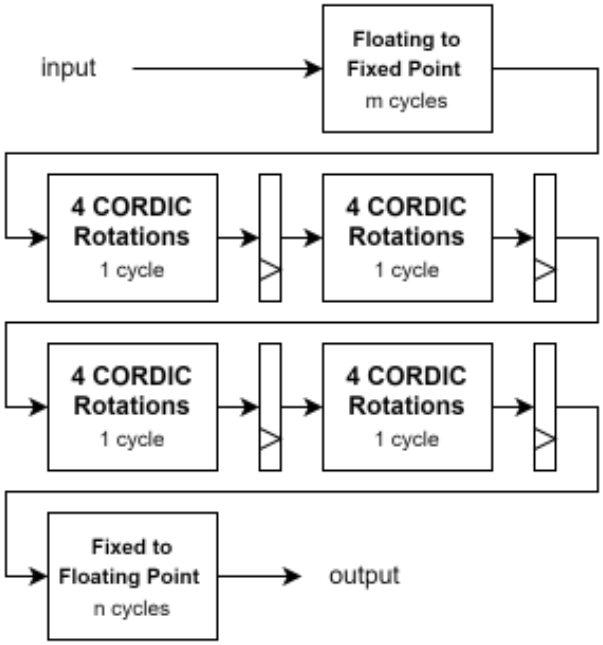


Figure 11: Fully Pipelined CORDIC block.

3.6 Test Results

Each of the synthesizable CORDIC blocks were instantiated as a custom instruction in the NIOS2 system from Task 6 (with hardware floating point add and multiply) and evaluated on each of the three testcases (Fig. 12). As expected, there is a tradeoff between latency and resources used. The unpipelined 5 cycle CORDIC block achieves the best execution time on all testcases and the pipelined version performs only marginally worse on testcase 1. As discussed in Task 6, the NIOS2 custom instruction interface blocks after being called, so the pipeline is not being utilized at all.

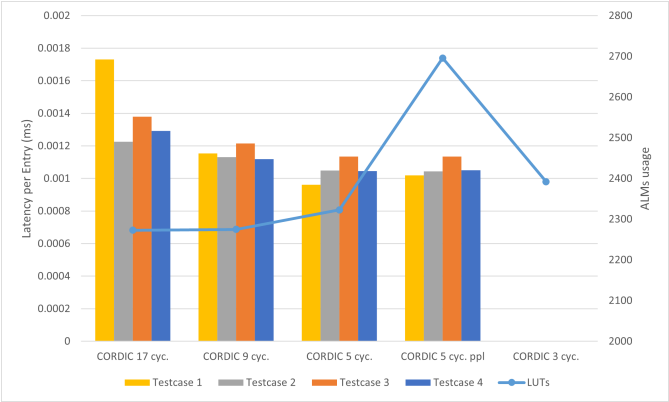


Figure 12: Custom Instruction CORDIC cosine Performance. 3 cycle CORDIC block did not meet timing.

	CORDIC 5 cycles	Task 6
Logic utilization (in ALMs)	2323 / 32,070 (6%)	2048 / 32,070 (5%)
Total block memory bits	28%	28%
Total DSP Blocks	4 / 87	4 / 87
Total PLLs	1 / 6	1 / 6

Table 4: Resource Utilisation Report: 64KB I-cache, 64KB D-cache, SDRAM

The 5 cycle CORDIC block achieves the best overall execution time on Testcase 3, 296.3 ticks. This is a 42x speed up over the latency achieved by the Task 6 design (12440 ticks). The application size is 72kB, lower than the 83kB used by Task 6, as the math.h library is no longer needed to compute cosine. 13.4% more ALMs are used when instantiating CORDIC as a custom instruction, the number of DSP blocks used remains the same.

Testcase	1	2	3	4
Problem Size	52	2041	261121	2323
Double Error (%)	-6.13e-5	-2.89e-5	-7.78e-4	-3.33e-2
Single Error (%)	-1.38e-5	5.11e-5	9.11e-4	-9.73e-5
Latency Speedup	50.0	45.7	42.0	46.7

Table 5: Error and Latency achieved by Custom Cosine

The error of the result calculated using the custom cosine instruction when compared to the double implementation is on the same order of magnitude as the error of the result calculated using floating point operations (Task 5, 6). Testcase 4 is an outlier, the error is significantly worse. This may be due to the fact that testcases 1-3 are generated systematically as steps of powers of 2, so consecutive terms are more similar so the floating point accumulation is more accurate (discussed in detail in 5.1).

4 Inner Function Implementation (Task 7.2)

In this section we implement the inner part (Eq. 4) of the target function in hardware.

$$f(x) = (0.5 \times x + x^2 \times \cos((x - 128)/128) \qquad (4)$$

A simple mapping of Eq. 4 into hardware using Intel FP arithmetic blocks and the lowest latency CORDIC block gives the datapath in Fig. 13. The critical path is highlighted in red - the overall block has a 15 cycle output latency. In a pipelined design, buffers (implemented as shift registers) are needed to ensure data from different paths arrive at the required blocks at the correct time.

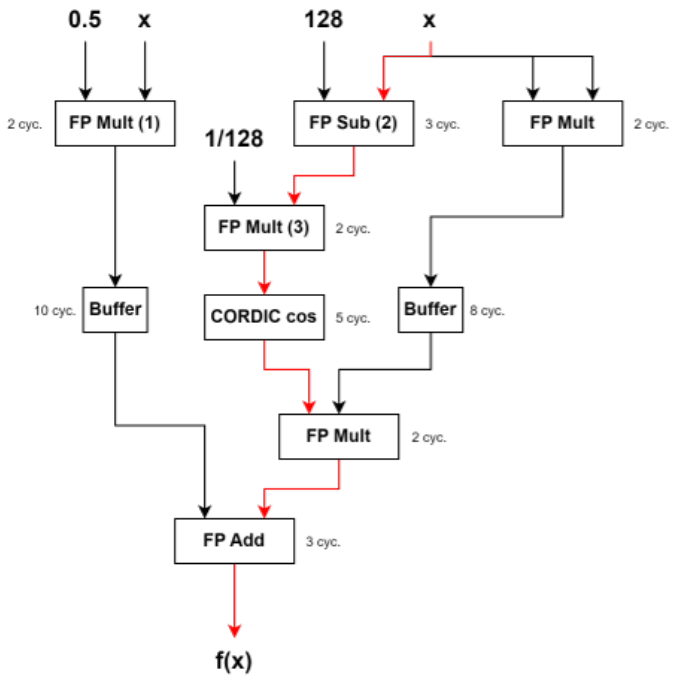


Figure 13: Datapath of inner function.

4.1 Custom IP Block Design

To further improve the output latency and resource usage of the inner function, we designed custom IP blocks to replace sections of the datapath.

Custom 0.5*x (Block 1)

The result of multiplying a floating-point number with 0.5 can be achieved by subtracting 1 from the exponent slot of the floating-point number, instead of using a full floating-point multiplier. The floating point multiplier can be replaced with a fixed-point subtractor in the first branch.

There is a special case that needs to be considered which is when the exponent slot is 0. In this case, subtracting 1 from the slot will set its value to 255, resulting the "not a number" case in floating point format. In fact, the exponent slot 0 corresponds to the denormalized case and is a number with the actual exponent number of -126. These numbers are too small to be represented by 21 fractional bits and can be treated as 0 directly. Therefore, we detect when the exponent is zero and set the output to be 0.

Custom x/128 (Block 3)

The same concept can be applied to dividing the floating-point number by 128. To achieve this, we can simply subtract the exponent slot by 7 while keeping the rest of the bits unchanged. However, if the exponent slot is smaller than 7, the subtraction will lead to a negative number and produce an incorrect result. To handle this case, the same principle as above was followed and the number was treated as 0. As exponent field is unsigned we can simply perform a cumulative OR on the upper 5 bits to check if the exponent field is greater than 7 (instead of using a subtractor). Since this combinatorial logic is on the critical path, any improvement will give a direct impact on the overall latency and ensure the block can still run at 50MHz.

Custom (x-128)/128 (Block 2 and 3)

The design can be further optimized by performing the calculation of $(x - 128)/128$ in fixed-point operations rather than floating point. This optimization is possible because the result will be converted to a 21 bits fixed-point before entering the CORDIC block, hence the level of precision in the previous step can be relaxed.

First we need to choose a suitable fixed point format to perform the calculation. The format needs to ensure that all the significant bits that appear in the final 21 bit fixed point representation are accounted for during the calculation. Start by converting the floating point input to a fixed point format with infinitely many integer and fractional bits, so all bits are accounted for. The input range is in the range [0, 255], so a maximum of 1 signed bit and 8 integer bits is required. In this format the two's complement representation of -128 is simply the top two bits set to 1 and all others zero, therefore subtracting a number by 128 only changes its top two bits. To divide by 128, we can simply right shift the number by 7 places with sign extension (pad the top 7 bits with the correct sign). The desired 21 bit fixed point format is trivially obtained by extracting the 1 bit before and 20 bits after the binary point.

(The following {s,i,f} notation defines a fixed point representation with s signed bits, i integer bits and f fractional bits). Notice that the {1, 20} bits of the output are the {8, 13} bits of the number before the bitshift, none of the other bits are involved. As we know subtracting by 128 only affects the top two bits of {1, 8, 13}, these are the only bits in our initial "infinite" fixed point format that are required. The floating point input can easily be converted to this fixed point format (Fig.14).

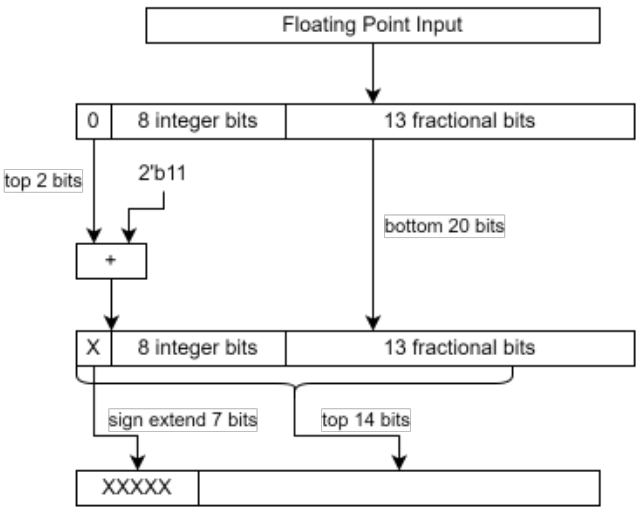


Figure 14: (x-128)/128 integrated block

4.2 Custom IP Block Evaluation Results

Fig.15 indicates the resource utilization and Fmax achieved by each of the three designs in 4.1 when integrated into the inner function datapath as combinatorial

blocks. All designs meet the target 50 MHz frequency.

There is a negative correlation between Fmax and the complexity of the IP block. The custom (x-128)/128 design achieved the smallest Fmax among all the designs since it replaces a 10-cycle floating-point operation with combinational logic.

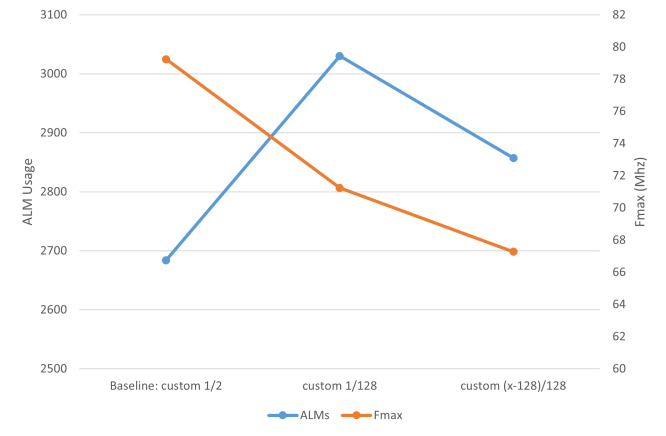


Figure 15: LUT usage and maximum frequency(Fmax) of designs integrated with three custom blocks respectively

In term of resources utilization, design with custom 1/128 has the highest utilization of ALMs. This is unexpected - our Verilog code may have been written in a non-compiler friendly way. The custom 1/2 design uses 6 embedded multipliers, while the other two designs use only 5, as the 1/128 operation is no longer performed using a FP multiplier.

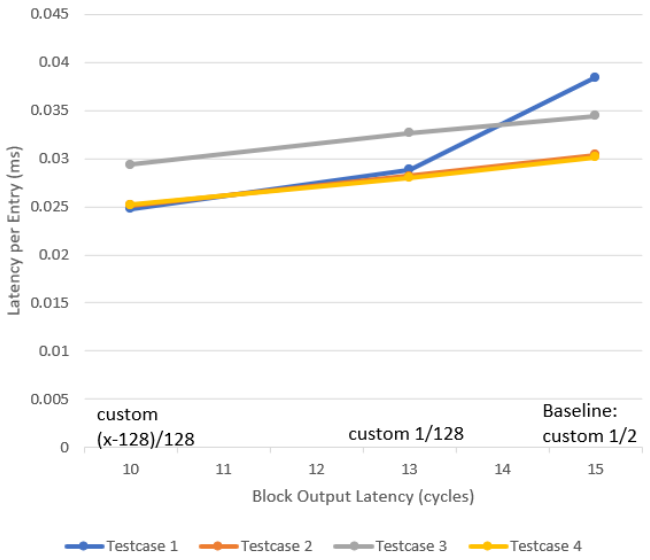


Figure 16: Latency achieved by different designs

Fig. 16 plots the execution time per entry for all testcases against the output latency achieved by each block. With the exception of the outlier testcase 1, all testcases demonstrate a linear correlation between execution time and block latency as expected. The testcases are slightly vertically offset, with testcase 3 having a larger constant offset than the rest. This may be due to factors such as cache miss rate: the testcase may be too large to be entirely stored in the cache, resulting in the cache miss penalties being amortised over all the entries. The result shows than design with 10 cycle latency (custom (x-128)/128) achieved the best performance in term of execution time. There is an average latency speed up of 2 averaged across all testcases when compared to Task7.1 - this is expected as the output latency is almost halved (19 cycles previously).

Pipelined versions of the inner function achieve identical latency as the unpipelined versions, again expected because of the blocking nature of the custom instruction interface; the ALM usage is also slightly higher (Table 6). Compared to the previous task (Table 4), implementing the inner function has added 1 more DSP block to the system and uses 30% more ALMs.

	Unpipelined	Pipelined
Logic utilization (in ALMs)	3030 / 32,070 (9.5%)	3344 / 32,070 (10%)
Total block memory bits	28%	28%
Total DSP Blocks	5 / 87	5 / 87
Total PLLs	1 / 6	1 / 6

Table 6: Resource Utilisation Report: Custom 1/128 system, 64KB I-cache, 64KB D-cache, SDRAM

All versions of the inner function produce the same result as in Table 7, there is no error introduced by our hardware optimisations.

5 Full Function Hardware Implementation (Task 8)

In this section we implement the full function (Eq. 1). The base design (Fig. 18) is simple, the inner function is applied to each element of the vector, and the overall summation of the terms is performed by a floating point accumulator block. The block is fully pipelined with 16 cycles latency - the accumulator adds 3 cycles.

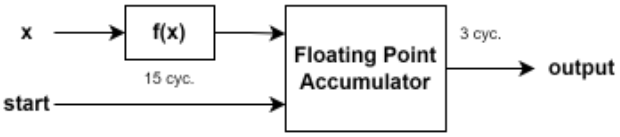


Figure 17: Full Function Block

5.1 FP Accumulator

The accumulator block takes in one input per cycle and adds it to a running count of all previously seen inputs. Compared to a naive floating point adder with a feedback register, accumulator block IPs are better optimised for accuracy, latency and resource usage.

The design space for accumulators is large, and the Quartus IP block is closed source; however based on its

available parameters the block seems to be based on an architecture described by [2]. The key idea is to store the accumulated result in a wide fixed point format. This improves accuracy greatly - in a regular floating point addition, if the magnitude of the operands are very different, there is minimal overlap of the mantissas and many bits are lost due to the limited precision. Having a large fixed size accumulated result also allows for efficient adder architectures to be used to speed up carry propagation through when adding inputs to the long result.

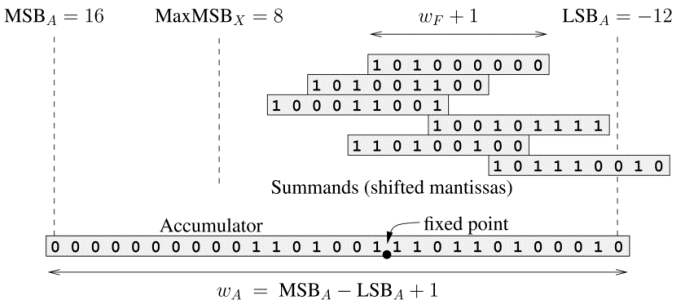


Figure 18: FP Accumulator [2]

The size of the fixed point accumulator is parameterized on $MaxMSB_x$ - an upper bound on the order of magnitude of the input, and MSB_A, LSB_A - upper and lower bounds on the accumulated result. In our application, $MaxMSB_x$ is set to 16 bits (based on the maximum value of $f(x)$ in the input range) and MSB_A was chosen to be 40 bits (so results 128x that of testcase 3 can be stored). LSB_A was set to the minimum value such that the overall block still meets timing (-90 bits) with 3 cycle latency.

5.2 NIOS2 C.I. "Fake" Pipeline Hack

While the NIOS2 custom instruction interface is blocking, there is a hack to get it to execute instructions in a pipelined way. The key observation is that custom instruction hardware is not reset between instructions, so any intermediate results left in the pipeline are preserved between instruction calls. Therefore, by setting the output latency of our block to 1 in C.I. interface, we can push one input into the pipeline every function call. After the last input enters the pipeline, a number of empty C.I. calls are required to push it to the end of the pipeline (4).

```
const int empty_cycles = PPL_DEPTH - 1;
float task8_function(float x[], int M){
  int i; float y = 0;
  full_fn(x[0], 1);
  for(i=1; i<M; i++)
    full_fn(x[i], 0);
  for(i=0; i<empty_cycles; i++)
    full_fn(0.0, 0);
  y = full_fn(0.0, 0);
  return y;
}
```

Listing 4: "Fake" Pipeline Hack

5.3 Results

The full function block uses 13% more ALMs (3783) than the pipelined inner function block, all other resources are unchanged. It achieves an average 3x latency improvement averaged across all testcases. This is lower than expected, pipelining a block should achieve a speed up approximately equal to the pipeline depth (13x) for large testcases. The execution time may be bottlenecked by memory accesses rather than computation. The error of the block has improved thanks to the FP accumulator, on testcase 4 the error is now the same order of magnitude as that of IEE754 single precision calculations.

Testcase	1	2	3	4
Task8 Error (%)	-6.13e-5	6.78e-6	-2.08e-4	-8.78e-5
Task7 Error (%)	-6.13e-5	-2.89e-5	-7.78e-4	-3.33e-2
Latency (ms)	9.58e-3	0.33	64.3	46.7
Latency Speedup	3.09	2.66	3.58	2.70

Table 7: Error and Latency achieved by Task 8

5.4 Relaxing the target clock frequency

To further improve the execution time, one approach would be to relax the constrained clock. As latency is defined by $L(D) = f(D)/g(D)$ where g is a function of the maximum clock frequency achieved by design D and f is the number of cycles of the design D. There is a trade-off between having a shorter critical path but larger latency or faster critical path with smaller latency. As the starting point, to find the minimum point of function L, a software model of the baseline custom(x-128) design is built, in which the maximum clock will only be capped by the critical path. The model takes the local Fmax and latency relationship of FP-blocks and CORDIC with various unroll factors into account. A grid search was performed by burn forcing a different combination of FP-blocks with different latency and CORDIC blocks with different unroll factors. The simulation result shows that CORDIC unrolls 2, FP-MULT with 3 cycles latency, and FP-ADD with 5 cycles combination operating under 90MHz would give rise to the best result. The synthesis result achieved 157ms for test case 3 reaching the same order of latency of design with custom(x-128)/128. By relaxing the target clock, we added another dimension to the search space, which requires a different design methodology to perform efficient searching. Due to limited time, this is left as futher work.

6 Conclusion

The use of custom hardware has significantly improved the execution time of the target application. Our final design achieves a resource utilization score of 14.7%, computes testcase 3 in 64.3 ms, a 805x speedup over the pure software solution from Report 2. The result has 2.08e-4% error, on the same order of magnitude as the error between the single and double precision result. Throughout the design process, we repeatedly encountered tradeoffs between resource usage, latency, and accuracy (Fig. 19).

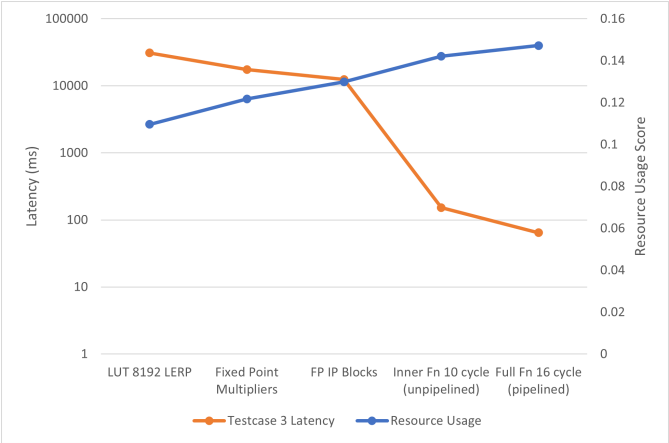


Figure 19: Latency and Resource Usage of Past Designs

Pipelining the function in task 8 did not achieve the expected performance improvements, suggesting that the application is now bottle-necked by the memory access time rather than the execution time. In the current design, data is first fetched from memory by the CPU and loaded into a register file, before being written to the custom instruction block. To increase the memory throughput, using a Direct Memory Access (DMA) controller could allow the custom instruction block to bypass the CPU and fetch data directly from the main memory.

Further improvements can be made to the datapath (13) to minimize resource usage and reduce output latency. The final addition and multiplication can be performed using a single Fused Multiply Add (FMA) block. In addition, by relaxing the target clock, a new dimension is added to the design space which might contain a better performance design.

References

[1] A. M. Abdelhadi and G. G. Lemieux, "Modular sram-based binary content-addressable memories," in *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2015, pp. 207–214.

[2] F. de Dinechin, B. Pasca, O. Cret, and R. Tudoran, "An fpga-specific approach to floating-point accumulation and sum-of-products," in *2008 International Conference on Field-Programmable Technology*, 2008, pp. 33–40.