Imperial College London

Digital Systems Design

Report 3

Dada Oluwatomisin, CID: 0084614

Pranav Malhotra, CID: 00823617

April 29, 2016

Contents

1 Introduction

The task set out in the Digital Systems Design coursework is to accelerate the computation of the complex expression listed in equation (1).

$$f(x) = \sum_{i=1}^{N} 0.5x_i + x_i^2 cos(floor(\frac{x_i}{4}) - 32) \tag{1}$$

In the two previous reports multiple aspects of this task have been explored.

Firstly, an analytic approach to solving the problem was explored. Although this worked for previous elementary tasks, it is not applicable for more complex expressions, like equation (1), that do not have an analytical form. **Also, such an implementation strays away from the objectives and learning lessons set out in the coursework.** Secondly, brief error analysis was conducted. The error analysis provided some insight into the shortcomings of floating point numbers. Interfacing with external memories was introduced; this is necessary since it not possible to store all instructions and data near the cpu on on-chip memories. The overhead associated with fetching data/instructions was studied. It was discovered that the use of caches can significantly reduce the overall latency of the system. Analysis was then conducted to determine the ideal cache size. **A balanced approach would suggest using $8Kb$ of cache memory. However, the task requires accelerating the evaluation of the complex equation given all the resources on the Altera DE0 Board and thus a cache size of $32Kb$ was decided upon.** Lastly, dedicated hardware to implement fixed-point multiplication was added to the processor. This additional hardware had a sizable effect in accelerating the evaluation of the complex expression.

The next step is to fully map the evaluation of the complex expression onto hardware. This report will present multiple design considerations, learning lessons, as well as the final design. Error analysis will be conducted to evaluate the accuracy of the system. The trade-off between resource utilisation and achieved latency will be considered.

2 Corrections from Report 2

In report 2, table 1 was presented.

N	Step	MATLAB Result	`generateVectorAddition` Percentage Error	`generateVectorMultiplication` Percentage Error
52	5	57880.00	0.0002%	0.0002%
511	0.5	34026.00	-0.0004%	-0.0004%
2551	0.1	-68594.00	-12.2159%	-0.0022%
25501	0.01	-1223100.00	-30.0492%	0.0024%
255001	0.001	-12768000.00	389.9631%	0.0034%

Table 1: Incorrect error analysis for task 4

The function `generateVectorMultiplication` presented in listing 1 was observed to produce a result with a smaller percentage error with respect to the reference value generated in MATLAB. This is however incorrect. **The reference value that the result was compared to was wrong; it was incorrectly generated on MATLAB.**

```
1  void generateVectorMultiplication(float x[N]){
2          int i;
3
4          // initialise first element
5          x[0]=0;
6
7          // generating vector through multiplication
8          for(i=1; i<N; i++)
9                  x[i]=i*step;
10 }
```

Listing 1: `generateVectorMultiplication`

The most accurate way to obtain a reference value is to import the input vector generated on the NIOS II processor onto MATLAB, perform all calculations using double-precision floating point numbers and obtain a

result. Calculated in such a way, the reference value, for test case 3, that results should be compared to is 37022686.6086. Table 2 presents the updated error analysis.

N	**Step**	**MATLAB Result**	`generateVectorAddition` **Percentage Error**	`generateVectorMultiplication` **Percentage Error**
52	5	57879.87	-3.17E-06%	-3.18E-06%
511	0.5	34026.04	9.73E-05%	9.74E-05%
2551	0.1	-76972.25	-1.04E-04%	-10.88%
25501	0.01	-1590647.07	-6.309E-05%	-23.11%
255001	0.001	37022686.60	3.085E-05%	-134.49%

Table 2: Correct error analysis for task 4

This does not mean that the assertion made in the previous report, that the way that the input vector is generated introduces errors into the final value, is completely invalid. The input vector generated on the NIOS II processor is not ideal. There are errors in the vector; the vector is stored as an array of single-precision floating point numbers and thus the precision of the numbers is limited especially since small and large numbers are summed when the vector is being generated. **This is a limitation of the single-precision floating point number system. That being said, no practical number system can fully represent all real numbers and thus a balanced trade-off between accuracy, memory requirements and latency has to be made.**

For the task at hand, the input vector is an invariant. Thus, it is best to focus on the inaccuracies introduced into the system through the custom hardware rather than the input vector. As such, all systems are tested with an input generated using the function `generateVectorAddition` presented in listing 2.

```
1  void generateVectorAddition(float x[N]){
2          int i;
3
4          // initialise first element
5          x[0] = 0;
6
7          // generating vector by accumulating result
8          for(i=1; i<N; i++)
9                  x[i]=x[i-1]+step;
10 }
```

Listing 2: `generateVectorAddition`

3 Task 6: Floating Point Units through Custom Instructions

In task 5, dedicated hardware capable of performing fixed-point multiplications was added. Two different approaches were considered; logic elements and embedded multipliers. Embedded multipliers proved more efficient at accelerating the evaluation of the function. **However, both logic elements and embedded multipliers only provide support for fixed-point multiplication.** Floating point multiplication is a much more involved process.

The steps for floating point multiplication are:

- Add the exponents to find the new exponents.
- Adding biased exponents causes the bias to be doubled. This has to be accounted for by subtracting the additional bias.
- Multiply the mantissas together.
- Normalise the result.
- Perform rounding.

Adding embedded multipliers only provides dedicated hardware for the multiplication step. The other steps still need to be emulated in software. It would be ideal to perform the entire floating point multiplication process in hardware. This requires the use of custom instructions on the Altera DE0 Board. Three types of custom instructions can be implemented on the Altera DE0 board. They are:

- Combinational: Single clock cycle custom logic blocks.
- Multicycle: Multi-clock cycle custom logic blocks of fixed or variable durations.
- Extended: Custom logic blocks that are capable of performing multiple operations

3.1 Design Considerations

Before a custom instruction type is chosen and dedicated hardware is introduced, a set of design considerations needs to be formulated.

1. **The design should allow as many operations as possible to be performed on custom hardware.** As discussed above, any floating-point operation not performed on dedicated hardware will be emulated on software. This is not ideal as software emulation is costly. As such, the design should be capable of performing both floating-point additions and floating-point multiplications; both additions and multiplications are necessary to evaluate equation (1).
2. **The design should be implemented such that the number of custom instruction calls is minimised.** The custom hardware should aim to reduce the number of iterations of the `for` loop, needed to run through the entire input vector. This would be possible if the custom hardware is capable of processing more than one element in the input vector during each custom instruction call.
3. **The hardware should return a result as soon as possible.** There should not be unnecessary delays within the custom hardware block.
4. **The design should minimise the amount of resources used.** However, should there be a scenario where resources can be traded to reduce the latency, the design which reduces the latency will be preferred.

3.2 Choosing Custom Instruction Type

It is clear that extended would provide the greatest flexibility in terms of offloading multiple operations from the cpu onto the custom hardware. In addition, extended requires a ***done*** signal to be asserted once the result is ready. The ***done*** signal can be used to optimise a system in which different operations have a different latencies. Finally, extended takes two data inputs. The two inputs can be utilised to optimise the system, either by performing multiple calculations in parallel or by pipelining the system and performing multiple calculations sequentially.

The timing diagram for the extended custom instruction is provided in figure 1.

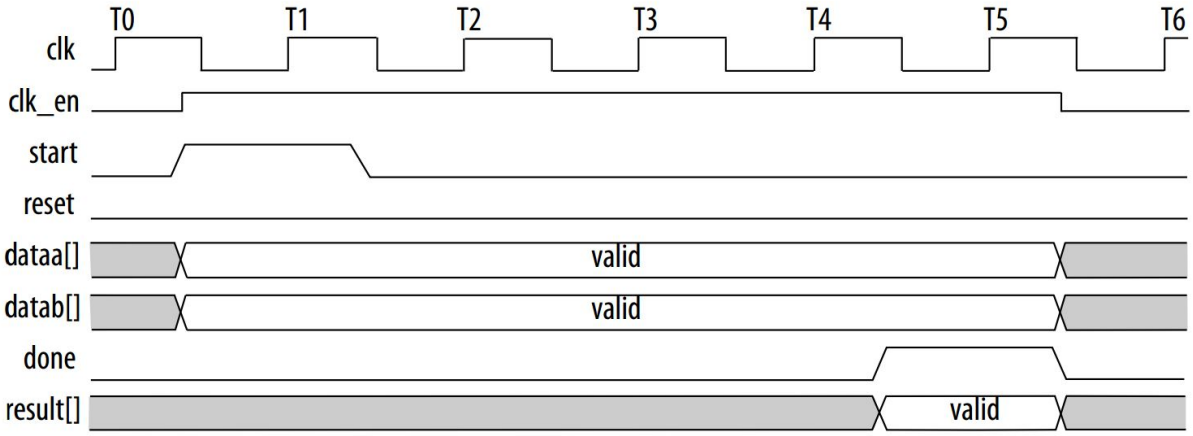

Figure 1: Timing diagram for extended custom instruction

3.3 Implemented Design

Based on design consideration 1, as many floating-point operations as possible should be mapped onto custom hardware. The custom instruction is designed to support both floating-point additions and floating point multiplications. Figure 2 shows the schematic of the design. Two inbuilt Altera IP blocks, namely `ALTFP_ADD_SUB` and `ALTFP_MULT` are used. A 1-bit select signal named ***sel*** is incorporated into the design.

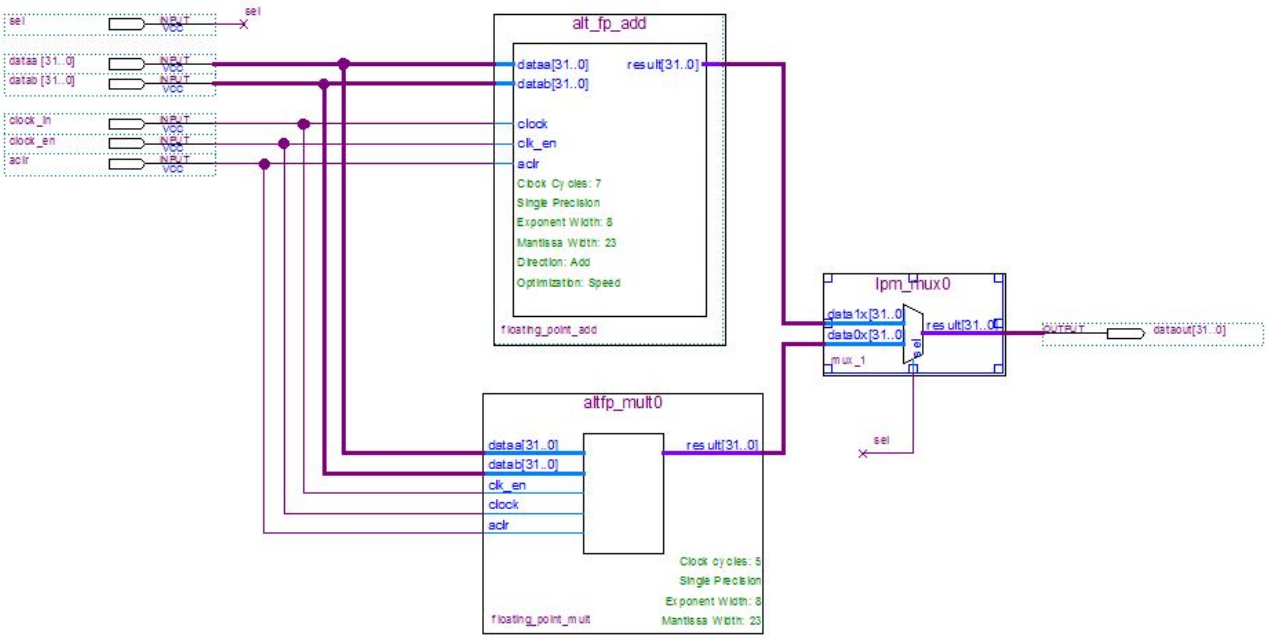

Figure 2: Task 6 design

Table 3 details the operation that is performed with respect to the select signal.

Extended Custom Instruction	
sel	**Operation Performed**
0	Multiplication
1	Addition

Table 3: Operations corresponding to ***select*** signal

It is important to note that both addition and multiplication are performed at the same time. The correct output is selected through the use of a multiplexer. Based on design consideration 3, it is

important that there is no unnecessary delay within the system. The two Altera IP blocks that are used have different latencies. Floating-point addition has a latency of 7 clock cycles whereas floating-point multiplication has a latency of 5 clock cycles. As such, it is important to assert the done signal at the appropriate time, based on the instruction that is being performed. Figure 3 shows the schematic of the hardware that generates the ***done*** signal.

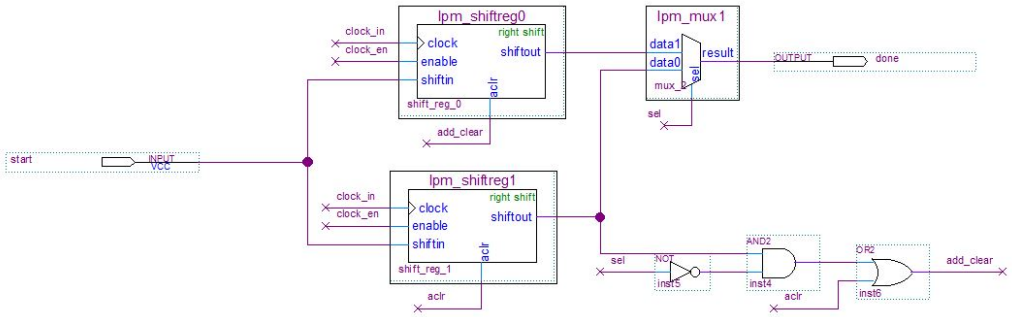

Figure 3: Task 6 ***done*** signal generator

3.4 Results for Task 6

3.4.1 Evaluation of Function on NIOS II Processor

Listing 3 presents the function `sumVector_FP_ARTHEMETIC_SUPPORT` that makes use of the custom instruction designed in section 3.3 to offload floating-point arithmetic to custom hardware.

```
1  // function performs operations in such a way so as to minimise custom instruction calls
2  float sumVector_FP_ARTHEMETIC_SUPPORT(float x[], int M){
3
4          int i;
5          // variable to hold final sum
6          float running_sum = 0;
7
8          // variable to hold value of complex expression for 1 input
9          float current_value = 0;
10
11         // number of iterations is equal to length of the vector
12         for(i=0; i<M; i++){
13
14                 // evaluate cosine using math.h library
15                 current_value = cos(floor(x[i]/4)-32);
16
17                 // performs multiplication
18                 // xcos(floor(x/4)-32)
19                 current_value = ALT_CI_FP_ARITHMETRIC_SUPPORT_0(0, x[i], current_value);
20
21                 // adds half to current sum
22                 // 0.5 + xcos(floor(x/4)-32)
23                 current_value = ALT_CI_FP_ARITHMETRIC_SUPPORT_0(1, 0.5, current_value);
24
25                 // multiplies by number
26                 // x(0.5 + xcos(floor(x/4)-32))
27                 current_value = ALT_CI_FP_ARITHMETRIC_SUPPORT_0(0, x[i], current_value);
28
29                 // updates running sum
30                 running_sum = ALT_CI_FP_ARITHMETRIC_SUPPORT_0(1, running_sum, current_value);
31         }
32
33         // returns running_sum
34         return running_sum;
35 }
```

Listing 3: `sumVector_FP_ARTHEMETIC_SUPPORT`

The evaluation of the equation has been broken down into multiple stages. This allows all floating-point arithmetic to be performed using the custom hardware while minimising the number of function calls. **Having a design that minimises custom instruction calls is design consideration 2; for task 6, this has been achieved by smart evaluation of the equation in software rather than by adding additional hardware support to deal with two inputs at the same time.**

3.4.2 Comparison of Latency

The latency of the system designed in section 3.3 is compared to the latency of the system designed for task 5. **In task 5, floating-point arithmetic was emulated in software and thus adding the custom instruction should greatly reduce the latency of the system.** Table 4 compares the latency achieved in task 5 to the latency achieved in task 6. Figure 4 presents the same data graphically.

N	Step	Task 5 (ticks)	Task 6 (ticks)
52	5	19	16
511	0.5	185	164
2551	0.1	927	820
25501	0.01	9259	8182
255001	0.001	92801	82010

Table 4: Comparison of latency improvement

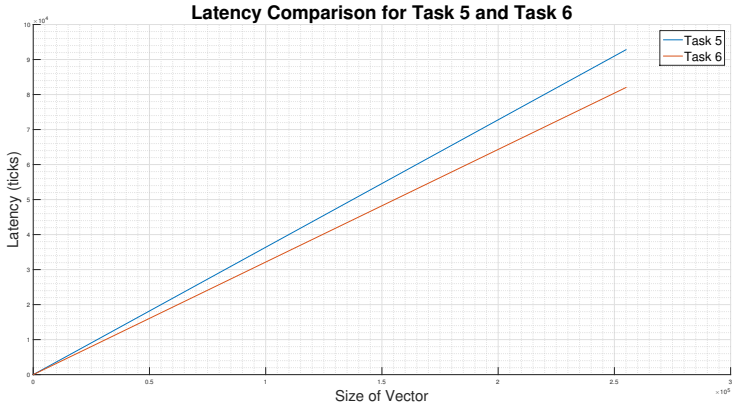

Figure 4: Comparison of latency improvement

There is certainly an improvement in the latency, however the improvement is not as significant as conjectured. There are multiple reasons for this:

- Floating-point arithmetic might not be the computationally expensive step in the evaluation of the complex expression.
- Floating-point arithmetic is emulated well in software, especially if dedicated hardware for fixed-point multiplication has been provided.
- The decrease in latency is offset by the overhead cost of making multiple instruction calls.
- The NIOS II/s processor utilises a 5 stage pipeline. Pipeline stall occurs when multicycle/extended instructions are called. The custom instruction utilised has a variable length; it will increase the number of pipeline stalls.

3.4.3 Comparison of Accuracy

The accuracy of the results obtained for task 6 is compared to that of the results obtained for task 5 and also to the reference value generated in MATLAB. The results are presented in table 5. **Note that the error from the reference value is in absolute terms and not in terms of percentage error. The errors are so small that using a percentage error would not reveal much.**

N	Step	Error From Reference	
		Task 5 (ticks)	Task 6 (ticks)
52	5	-0.003	-0.007
511	0.5	0.081	0.093
2551	0.1	-1.141	-0.875
25501	0.01	14.945	11.945
255001	0.001	-202.600	-214.600

Table 5: Comparison of accuracy

The results obtained in task 6 are not that different from the results obtained in task 5. Both results are very close to the MATLAB reference value. There is however a small discrepancy. **The discrepancies come from the fact that floating-point arithmetic is not commutative or dissociative in the way that standard arithmetic is.** For example, $(A+B)+C$ should yield the same result as $A+(B+C)$. This is however, not the case when certain combinations of floating point numbers are considered. Floating point numbers are truncated/rounded/zero-padded after each operation and thus the errors incurred vary if the order of operations is changed. **As mentioned in section 3.4.1, the order of operations has been manipulated so as to minimise the number of function calls. The order of operations should have been manipulated to minimise the error as well.**

$$Step\ 1 : current\_value = cosine(floor(x[i]/4) - 32)$$
$$Step\ 2 : current\_value = x[i] * current\_value$$
$$Step\ 3 : current\_value = 0.5 + current\_value$$
$$Step\ 4 : current\_value = x[i] * current\_value$$
$$Step\ 5 : running\_sum = running\_sum + current\_value$$

Step 3 adds 0.5 to the *current_value* variable. This step requires adding a small number to a larger number. The order of operations can be changed to minimise the error in this calculation; however this will increase the number of instruction calls. **The key learning lesson here is that often floating-point arithmetic is not commutative/dissociative. As such the order of operations should be analysed and the trade-off between latency and accuracy considered.**

3.5 Final Comments

The design presented in section 3.3 has a signal called ***add_clear***. This was added after the initial design produced inconsistent results. The hardware that generates the ***done*** signal utilises two shift registers. Both shift registers are fed with an input simultaneously using the ***start*** signal. The output of only 1 shift register is chosen to produce the ***done*** signal; this is achieved through a multiplexer. If a multiply operation is called, the ***done*** signal is generated after 5 clock cycles. If the system is not cleared, the shift register that generates the ***done*** signal for the addition operation still contains a 1. The 1 is stuck in the 5^{th} stage of the shift register. If an addition operation is called right after a multiply operation, the 1 that is stuck in the shift register will cause the ***done*** signal to be asserted before a valid result is ready. **As such, the digital system should always be cleared between function calls. The clearing process is usually taken for granted however a robust design certainly needs to be cleared after every instruction call.**

The design presented in section 3.3 is not capable of processing two inputs at the same time. Design consideration 2 states that the custom hardware should be designed so as to minimise the number of iterations of the `for` loop. This requires processing 2 data points at the same time. Although the custom instruction is capable of taking in 2 inputs, it can only return 1 result. **As such, for 2 elements in the input vector to be processed at once but only return 1 result, the cosine function has to be evaluated in hardware. This will allow their sum to be returned as the result.** Once the cosine has been mapped to hardware, the system will be able to deal with two inputs simultaneously and the number of iterations of the `for` loop can be halved. This will be explored in section 5.

4 Task 7: Adding Hardware to Compute Cosine

As discussed in the section 3.5, reducing the number of iterations of the `for` loop hinges on computing the cosine in hardware rather than through software emulation. In this section, the implementation of the cosine in hardware using the cordic algorithm is discussed.

4.1 Cordic Algorithm

The cordic alogrithm is an iterative, bit-by-bit algorithm that takes an angle as an input, and through vector rotations computes the sine and cosine of that angle. The algorithm aims to zero the angle by successively adding or subtracting a set of elementary angles α_i. **A vector rotation is performed during each iteration and so to fully understand the computational complexity of the algorithm, it is essential to first understand how vector rotations works.** Rotating (x, y) by α produces (x', y'); x' and y' are mathematically expressed in equations (2) and (3).

$$x' = xcos(\alpha) - ysin(\alpha) \tag{2}$$
$$y' = ycos(\alpha) - xsin(\alpha) \tag{3}$$

Equations (2) and (3) can be simplified using the trigonometric identity $cos(\alpha) = 1/\sqrt{1+tan^2(\alpha)}$. As such, vector rotation can be expressed mathematically as in equations (4) and (5).

$$x' = \frac{x - ytan(\alpha)}{\sqrt{1 + tan^2(\alpha)}} \tag{4}$$
$$y' = \frac{y + xtan(\alpha)}{\sqrt{1 + tan^2(\alpha)}} \tag{5}$$

Evidently, vector rotation is a complicated process. Both multiplications and divisions are necessary.

The process can be greatly simplified if the set of elementary angles α_i are carefully chosen. **Multiplications and divisions by factors of 2 can be performed for "free" in hardware; they are implemented through simple re-wiring of signals. Thus it is intuitive that the set of elementary angles should be such that the multiplicative constant at each iterative step, $tan(\alpha_i)$, is a power of 2.** In addition, the algorithm requires division by a constant during each iteration. To simplify this process, the division can be performed just once; for the cordic algorithm, the initial value of x is scaled to account for this constant. The last important aspect of the cordic algorithm that should be noted is that it guarantees convergence in the range stated in equation (6).

$$\theta_{max} = \sum_{i=0}^{\infty} tan^{-1}\left(2^{-i}\right) \approx 1.7429\ rad\ (99.88^\circ) \tag{6}$$

Further details about the cordic algorithm are excluded from the report for the sake of brevity. **The key point is that a complex process such as vector rotations can be mapped efficiently to hardware if the digital architecture is well-understood.** In the case of the cordic algorithm, the efficient mapping comes at almost no loss of precision. This is in the sense that choosing elementary angles such that $tan(\alpha_i)$ is a power of 2 does not reduce precision. In multiple other applications, approximations have to be made so that the computations can be performed efficiently in hardware. In these scenarios, the accuracy of the calculations is traded for reduced computational complexity.

4.2 Implementation of Cordic Algorithm

4.2.1 Evaluating Inner Function

Before the cordic algorithm can be implemented, the argument of the cosine has to be evaluated. This requires evaluating equation (7).

$$f_1(x) = floor\left(\frac{x}{4}\right) - 32 \tag{7}$$

It is key to note that the range of the input has been limited to $[0, 255]$. This is not an unusual constraint that has only been stipulated in the Digital Systems Design coursework. Most digital systems have

predefined upper and lower bounds. Ensuring that a valid input is used for the system designed is usually performed in software. **Implementing such a fail safe in hardware is possible however it does not provide much utility for the amount of resources required. Also, for such a specific application, it is best to implement fail safes in software.**

With the input limited to $[0, 255]$, the range of $f_1(x)$ is limited to the range $[-32, 31]$. In addition, the *floor* function present in equation (7) ensures that the argument of the cosine will be an integer and not a decimal. **It is possible to represent the range of $f_1(x)$ as a 6-bit signed number.**

The argument of the cosine function is determined by carefully analysing the structure of single-precision floating point numbers. **By checking the value of the exponent, the number of shifts that have to be performed on the mantissa to convert the number from its scientific notation to its decimal form can be evaluated. To divide the number by 4, the number of shifts is reduced by 2.** By considering the appropriate number of mantissa bits, the floating point number is converted into a 6-bit unsigned number. A number in the range $[0, 63]$ is obtained. **Lastly, the 6-bit unsigned number is interpreted as a 6-bit signed number. Since the number is now signed, the most significant bit (MSB) represents -2^5. Inverting the MSB is equivalent to subtracting by 32.**

The module `div_floor_sub` designed in verilog and presented in listing 4 evaluates equation (7) in 1 clock cycle. The function takes as an input a 32-bit single-precision floating point number and converts it to a 6-bit signed number.

```
//-------------------------------------------------------------------------------------------------
// MODULE NAME:
//      div_floor_sub
//
// MODULE FUNCTION:
//      divide input by 4, floor and subtract 32
//      input is a 32-bit single precision floating point number
//      output is a 6 bit SIGNED number
//      range of inputs restricted to 0 - 255
//      output range restricted to -32 - 32

//-------------------------------------------------------------------------------------------------
module  div_floor_sub   (clk,
                         clk_en,
                         datain,
                         dataout);

//-------------------------------------------------------------------------------------------------
// inputs to div_floor_sub module
input           clk;
input           clk_en;
input   [31:0]  datain;

//-------------------------------------------------------------------------------------------------
// output from div_floor_sub module
output reg  [5:0]   dataout;

//-------------------------------------------------------------------------------------------------
// initialise registers
initial dataout = 6'b0;

//-------------------------------------------------------------------------------------------------
//module returns a result every clock cycle
always @(posedge clk) begin
    if(clk_en == 1'b1) begin
        case(datain[30:23])
            8'b10000001:    dataout <= 6'b100001;               // exponent: 129, input: 4-8
            8'b10000010:    dataout <= {5'b10001,datain[22]};   // exponent: 130, input: 8-16
            8'b10000011:    dataout <= {4'b1001,datain[22:21]}; // exponent: 131, input: 16-32
            8'b10000100:    dataout <= {3'b101,datain[22:20]};  // exponent: 132, input: 32-64
            8'b10000101:    dataout <= {2'b11,datain[22:19]};   // exponent: 133, input: 64-128
            8'b10000110:    dataout <= {1'b0,datain[22:18]};    // exponent: 134, input: 128-256
            default    :    dataout <= 6'b100000;               // default setting, input is 0-4
        endcase
    end
end

endmodule
```

Listing 4: `div_floor_sub`

It should be noted that cosine is an even function and thus, the range $[-32, 31]$ can be truncated to $[0, 32]$. **The benefit of this truncation, in terms of resources used, is marginal since 6-bits will still be required.**

4.2.2 Normalising Angle to the Fundamental Range

As discussed in section 4.1, the cordic algorithm only guarantees convergence in the range $(-1.7429, 1.7429)$. The range of $f_1(x)$ obtained in section 4.2.1 is $[-32, 31]$. To evaluate the cosine using the cordic algorithm, all angles need to be normalised to $[0, \pi/2]$. **The normalisation will need to take note of the fact that the cosine of angles in the 2^{nd} and 3^{rd} quadrant is negative.**

After normalisation, the values obtained need to be converted into binary digits. The cordic algorithm will be designed to work for fixed-point numbers. Multiple sophisticated implementations are able to evaluate the cordic algorithm on floating-point numbers however these will not be considered. Since the range of angles that are used as inputs to the cordic block is limited to $[0, \pi/2]$, the integer part of the fixed point number only requires 1 bit. However, 2 bits will be reserved for the integer part of the fixed-point number because during the cordic algorithm, the angle may take on a negative value. **As such, the integer part of the fixed-point number will be 2-bits long and signed.**

Since the cordic algorithm is a bit-by-bit algorithm, the algorithm will converge to the true value by correcting 1 bit at a time. For such algorithms, precision can be increased by increasing the number of iterations. As a baseline, the fixed-point number is designed to have 30 fractional bits. This was chosen such that the fixed-point number is 32-bits long. In-depth analysis of the optimum number of iterations and number of fractional bits is provided in section 5.3

The MATLAB script presented in listing 5 is used to normalise the angles obtained in section 4.2.1 to the fundamental range $[0, \pi/2]$. The MATLAB function `fi` is used to generate the fixed-point numbers. Note that normalised value that is generated is actually represented as a 31-bit fixed-point number; 1 integer bit and 30 fractional bits. The 32^{nd} bit is a sign bit. The sign bit represents whether the answer obtained from the cordic algorithm has to be negated, i.e, if the angle was in the 2^{nd} or 3^{rd} quadrant. The sign bit will be used in the cosine block. Also, in the cosine block, the 31-bit fixed-point integer that is generated will be padded with a 0 to make it a 32-bit fixed-point number. More information about how the sign bit is utilised is provided in section 4.2.3.

The script writes the binary output onto a text file such that the values can be easily copied into a case statement. It should be noted that, when compiled, the case statement, which has 64 cases, gets mapped to Read-Only Memory (ROM). It is possible to initialise a ROM directly as well. Initialising a ROM requires a Memory-Initialisation File (.mif).

```
clear all

% define number of fractional bits
num_frac = 30;

% open file to store normalised values
filename = sprintf('normalisevalues_%d.txt', num_frac);
fileID = fopen(filename,'w');

% form input vector
x = linspace(-32, 31, 64);

% store input vector in first column of matrix
b(:,1) = x;

% map all values to the 0 to 2pi
b(:,2) = mod(x, 2*pi);

% map all values to the first quadrant
% if negation is required in final cosine values, store 1 in column 4
% if no negation is required, store 0 in column 4
for i = 1:length(x)
    % first quad
    % no negation
    if (b(i,2)<=(pi/2))
        b(i,3) = b(i,2);
    % second quad
    % negation required
    elseif (b(i,2)<=pi)
        b(i,3) = pi - b(i,2);
        b(i,4) = 1;
    % third quad
    % negation required
    elseif (b(i,2)<=(3*pi/2))
        b(i,3) = b(i,2)-pi;
        b(i,4) = 1;
    % fourth quad
    % no negation
    elseif (b(i,2)<=(2*pi))
        b(i,3) = 2*pi-b(i,2);
```

```
    end
end

for  i = 1:length(x)
    % form fixed point number
    temp = fi(abs(b(i,3)), 0, num_frac+1, num_frac);

    % print onto file
    % append value contained in column 4, the sign bit
    % value represents if negation is required
    fprintf(fileID, '%d%s\n', b(i,4), temp.bin);
end

% close file
fclose(fileID);
```

Listing 5: MATLAB code to generate binary numbers needed for case statement

4.2.3 Calculating Cosine using Cordic Algorithm

Once the `div_floor_sub` module evaluates equation (7) and the ROM, initialised with binary data calculated using the MATLAB script presented in listing 5, provides a fixed-point representation of the argument of the cosine, the cordic algorithm can start. As mentioned in section 4.2.2, the output from the ROM will be a 31-bit fixed point number padded with a sign bit. The sign bit represents if the cosine needs negation. **As such, if the sign bit is set, the value of `X` is initialised to the 2's complement of the parameter `K`.** Note that the parameter `K` is in place to account for the constant multiplication that happens at each iteration of the cordic algorithm.

Listing 6 defines the module `cosine` that evaluates the cosine of `datain`.

```
//-----------------------------------------------------------------------------------------------
// MODULE NAME:
//      cosine
//
// MODULE FUNCTION:
//      takes a fixed-point number as an input and returns the cosine
//      MSB of the input is a sign bit
//      if the MSB is set, the result of the cosine requires negation
//      negation is performed by initialising X to -K: done by taking 2's complement
//      num_frac is a parameter that defines the number of fractional bits
//      num_iter is a parameter that defines the number of iterations of the cordic algorithm

//-----------------------------------------------------------------------------------------------
module  cosine  (clk,
                 clk_en,
                 start,
                 datain,
                 dataout);

//-----------------------------------------------------------------------------------------------
// parameters for cosine module
parameter num_frac = 30;
parameter num_iter = 31;
// length of K depends on the number of fractional bits
parameter K = 'b00100011011011101001110110110101O;

//-----------------------------------------------------------------------------------------------
// inputs to cosine module
input                   clk;
input                   clk_en;
input                   start;
input   [num_frac+1:0]  datain;

//-----------------------------------------------------------------------------------------------
// output from cosine module
output wire [num_frac+1:0]  dataout;


//-----------------------------------------------------------------------------------------------
// registers for cosine module
reg signed  [num_frac+1:0]  X;
reg signed  [num_frac+1:0]  Y;
reg signed  [num_frac+1:0]  Z;
reg signed  [num_frac+1:0]  TEMP_X;
reg signed  [num_frac+1:0]  TEMP_Y;

//-----------------------------------------------------------------------------------------------
// wires for cosine module
wire    [num_frac+1:0]  ARCTAN[num_iter-1:0];

//-----------------------------------------------------------------------------------------------
// integers for cosine module
integer i;

```

```
55  //----------------------------------------------------------------------------------------------------
56  // initialise registers
57  initial begin
58  X[i] =      K;
59  Y[i] =      'b0;
60  Z[i] =      'b0;
61  TEMP_X =    K;
62  TEMP_Y =    'b0;
63  end
64
65  //----------------------------------------------------------------------------------------------------
66  // assign wires
67  assign dataout   =     X;
68  assign ARCTAN[0] =     'b00110010010000111111011010101001;
69  assign ARCTAN[1] =     'b00011101101011000110011100000101;
70  assign ARCTAN[2] =     'b00001111101011011011101011111101;
71  assign ARCTAN[3] =     'b00000111111101010110111010100111;
72  assign ARCTAN[4] =     'b00000011111111101010101101110111;
73  assign ARCTAN[5] =     'b00000001111111111101010101011100;
74  assign ARCTAN[6] =     'b00000000111111111111101010101011;
75  assign ARCTAN[7] =     'b00000000011111111111111101010101;
76  assign ARCTAN[8] =     'b00000000001111111111111111101011;
77  assign ARCTAN[9] =     'b00000000000111111111111111111101;
78  assign ARCTAN[10] =    'b00000000000010000000000000000000;
79  assign ARCTAN[11] =    'b00000000000001000000000000000000;
80  assign ARCTAN[12] =    'b00000000000000100000000000000000;
81  assign ARCTAN[13] =    'b00000000000000010000000000000000;
82  assign ARCTAN[14] =    'b00000000000000001000000000000000;
83  assign ARCTAN[15] =    'b00000000000000000100000000000000;
84  assign ARCTAN[16] =    'b00000000000000000010000000000000;
85  assign ARCTAN[17] =    'b00000000000000000001000000000000;
86  assign ARCTAN[18] =    'b00000000000000000000100000000000;
87  assign ARCTAN[19] =    'b00000000000000000000010000000000;
88  assign ARCTAN[20] =    'b00000000000000000000001000000000;
89  assign ARCTAN[21] =    'b00000000000000000000000100000000;
90  assign ARCTAN[22] =    'b00000000000000000000000010000000;
91  assign ARCTAN[23] =    'b00000000000000000000000001000000;
92  assign ARCTAN[24] =    'b00000000000000000000000000100000;
93  assign ARCTAN[25] =    'b00000000000000000000000000010000;
94  assign ARCTAN[26] =    'b00000000000000000000000000001000;
95  assign ARCTAN[27] =    'b00000000000000000000000000000100;
96  assign ARCTAN[28] =    'b00000000000000000000000000000010;
97  assign ARCTAN[29] =    'b00000000000000000000000000000001;
98  assign ARCTAN[30] =    'b00000000000000000000000000000000;
99
100
101 always @(posedge clk) begin
102     if(clk_en == 1'b1) begin
103         // output of the ROM is ready to be read
104         if(start == 1'b1) begin
105             // take 2's complement of K if the output of cosine needs negation
106             // need for negation is signalled by MSB of input, which is a signed bit
107             if(datain[num_frac+1] == 1'b0) begin
108                 X <= K;
109             end else begin
110                 X <= ~K + 1'b1;
111             end
112
113             Y <= 'b0;
114             // pad 0 to z
115             Z <= {1'b0, datain[num_frac:0]};
116         // once a valid datain has been read into X
117         // cordic algorithm can start
118         end else begin
119             // if angle is positive, subtract to bring closer to 0
120             if(Z[num_frac+1]==1'b0) begin
121             // arthemetic shifts are required since registers hold signed values
122             TEMP_Y = Y>>>i;
123             TEMP_X = X>>>i;
124             Z <= Z - ARCTAN[i];
125             X <= X - TEMP_Y;
126             Y <= Y + TEMP_X;
127
128             // if angle is negative, add to bring closer to 0
129             end else begin
130             // arthemetic shifts are required since registers hold signed values
131             TEMP_Y = Y>>>i;
132             TEMP_X = X>>>i;
133             Z <= Z + ARCTAN[i];
134             X <= X + TEMP_Y;
135             Y <= Y - TEMP_X;
136             end
137         end
138     end
139 end
140 endmodule
```

Listing 6: `cosine`

It is important to note that the `cosine` module defined in listing 6 is only capable of dealing with 1 input at a time. As discussed in section 3.1, the digital system that is designed should minimise the number of iterations of the `for` loop. This will be discussed and implemented in section 5.

4.2.4 Fixed to Floating Point Conversion

The final stage requires converting the fixed-point number that the `cosine` module outputs into a floating-point number. The conversion can be performed in a similar fashion to the floating point to fixed-point conversion described in section 4.2.1; it would take 1 clock cycle. **However, for fixed-point to floating point conversion, a rule for truncating/rounding/zero-padding results to form a 23-bit mantissa has to be defined. This process is rather involved. An implementation that simply zero-extends or drops excess bits is possible however it was not used.** Instead the conversion is performed using the in-built Altera IP block called `ALTFP_CONVERT`. This block has a latency of 6 clock cycles.

4.2.5 Overall Design

The 4 modules defined above were generated and connected up and shown in figure 5. A shift register is included to delay the ***start*** signal by 2 clock cycles. This ensures that the data read into the cosine block is a valid output from the `normalise` block.

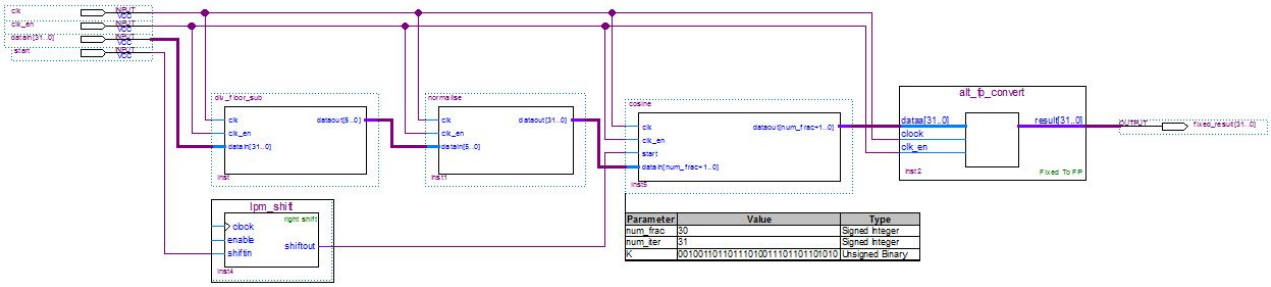

Figure 5: Implementation of cordic algorithm on quartus

The latency of the design is presented in table 6. The cordic block has a latency of 39 cycles. This is quite high and can be significantly improved. This will be considered in section 5.

Task 7 Design Latency	
Module	**Latency (clock cycles)**
`div_floor_sub`	1
`ROM`	1
`cosine`	31
`ALTFP_CONVERT`	6
TOTAL	39

Table 6: Latency of design for task 7

4.3 Overall Design for Task 7

The design proposed in section 4.2.5 computes the cosine in custom hardware. This design is incorporated with adders, multipliers and shift registers such that it is capable of evaluating the entire complex expression in hardware. The design is shown in figure 6.

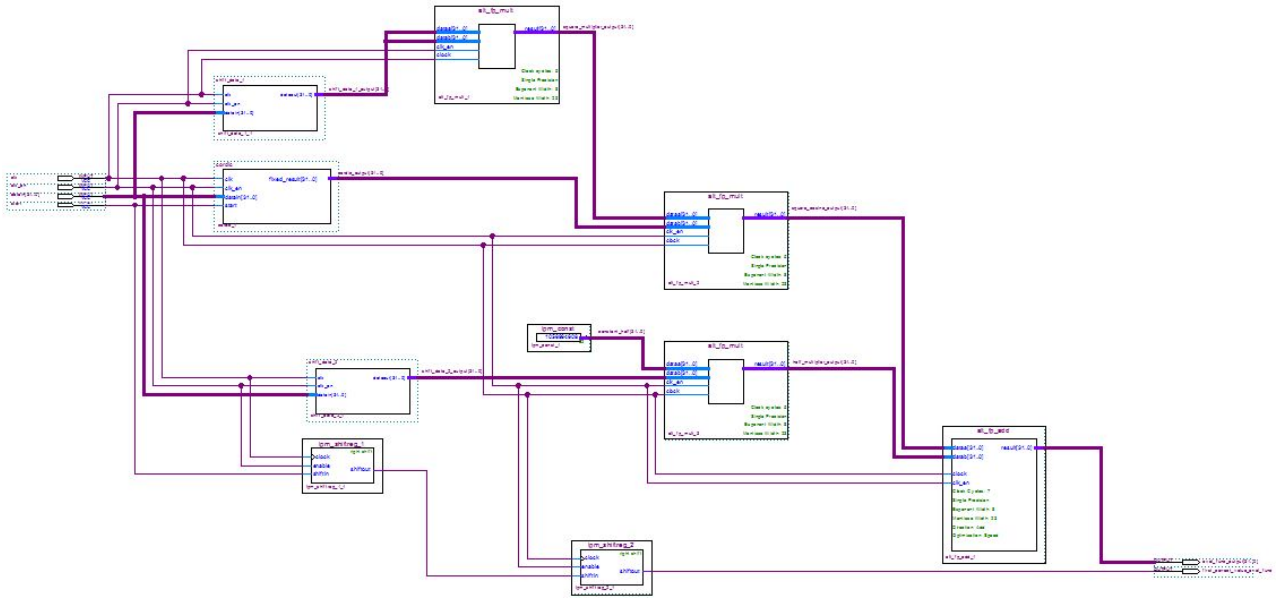

Figure 6: Mapping entire complex expression to hardware

Like the cordic block defined in section 4.2.5, the overall design is only capable of dealing with 1 input at a time. **The overall design has a latency of 48 clock cycles.** In addition, the design is only capable of performing 1 operation. As such, a select signal is not present. The amount of resources used are listed in table 7.

Hardware Description		**FPGA Resources**
Cache Size	32KB	36.79%
On-Chip Memory	0KB	
Logic Elements	6,365/15,408 (41%)	
Memory Bits	291,885 / 516,096 (57%)	
PLLs	1 / 4 (25 %)	
Embedded Multiplier 9-bit Elements	14 / 112 (12.5 %)	

Table 7: Resource usage for task 7

4.4 Results for Task 7

4.4.1 Evaluation of Function on NIOS II Processor

The design presented in section 4.3 is tested with the function `sumVector_COMPLEX_EXPRESSION` presented in listing 7.

```
// function that utilises custom hardware to evaluate custom instruction
float sumVector_COMPLEX_EXPRESSION(float x[], int M)
{
        int i;

        // variable to hold final sum
        float running_sum = 0;

        for(i=0; i<M; i++){
        // call custom instruction with just 1 input
        // value returned by function is added to the running
        running_sum += ALT_CI_COMPLEX_EXPRESSION_0(x[i]);
        }

        // return the value of running_sum
        return running_sum;
}
```

Listing 7: `sumVector_COMPLEX_EXPRESSION`

4.4.2 Comparison of Latency

The system is tested and results are presented in table 12.

N	Step	Task 5 (ticks)	Task 6 (ticks)	Task 7 (ticks)
52	5	19	16	1
511	0.5	185	164	5
2551	0.1	927	820	21
25501	0.01	9259	8182	209
255001	0.001	92801	82010	2090

Table 8: Comparison of latency improvement

The improvement in latency is significant. Note that in both task 5 and task 7, floating-point addition is performed through software emulation. As discussed in 3.4.2, the main reason that the improvement in latency from task 5 to task 6 is not very significant is due to the fact that floating point operations are not the most computationally expensive step in the evaluation of the complex expression. **The very large drop in latency once the cosine is evaluated in custom hardware suggests that computing trigonometric functions through software emulation using the `math.h` library is very costly. In addition, the number of custom instruction calls per iteration of the `for` loop has be dramatically reduced; for task 6, the custom instruction was called 4 times per iteration whereas for task 7 the custom instruction is called just once.**

4.4.3 Comparison of Accuracy

The accuracy of the baseline system is studied. The results are presented in table 9.

N	Step	Error From Reference		
		Task 5	Task 6	Task 7
52	5	-0.003	-0.007	-0.003
511	0.5	0.081	0.093	0.062
2551	0.1	-1.141	-0.875	-1.391
25501	0.01	14.945	11.945	11.945
255001	0.001	-202.600	-214.600	-154.600

Table 9: Comparison of accuracy

The results obtained from task 7 are closer to the reference value generated in MATLAB. This is exactly as expected. The baseline design performs 30 iterative steps in the computation of the cosine.

Accuracy up to 30^{th} bit is guaranteed since the cordic algorithm is a bit-by-bit algorithm. **It is not clear how many bits the `math.h` library utilises in intermediate computations however, single-precision floating point numbers only possess 23 mantissa bits and thus the computation is expected to be less accurate.**

4.5 Final Comments

The design presented in section 4.3 will be used as a baseline for comparison. As mentioned above, the design is only capable of dealing with 1 input and observing line 12 of listing 7, it is clear that the floating-point addition required to keep track of the running sum is performed through software emulation rather than through dedicated floating-point arithmetic units. Moreover, the fixed-point number system and the number of iterations for the cordic algorithm were arbitrarily chosen. There are significant improvements that can be made to the design and these will be discussed in section 5.

5 Task 8

5.1 Improved Design

In this section, modifications to the design presented in 4.3 will be considered. **The design that was tested in section 4.4 is a very naive implementation. It did not abide by the design considerations that were set out in section 3.1.** Firstly, it is desirable to map all floating-point arithmetic operations to hardware. The previous design did not provide support for floating point addition. Secondly, the extended custom instruction is capable of taking two inputs namely, `dataa` and `datab`. It is possible to use both these inputs to speed up the evaluation of the custom instruction. This was initially proposed in section 3.5. Using the cordic algorithm, the evaluation of the cosine has now been mapped to hardware and so it is possible to evaluate 2 instructions at once.

The design in section 4.3 requires two main changes if it is to be optimised. They are:

- Custom hardware should be included to perform floating-point addition.
- The custom instruction should take in 2 data points. It should evaluate the complex expression for both inputs and return their sum.

Support for floating point addition is provided using the Altera IP block `ALTFP_ADD_SUB`. A multiplexer similar to that used in section 3.3 is needed to generate the ***done*** signal. Since the custom hardware performs two operations, a select signal, ***sel*** is added. The operations corresponding to the select signal are defined in table 10.

Extended Custom Instruction	
sel	**Operation Performed**
0	Evaluation of Complex Expression
1	Addition

Table 10: Operations corresponding to select signal for task 8

In addition, a shift register has been added to the design to ensure that two elements of the input vector can be processed at once. The shift register stalls one input, namely `datab`, such that both inputs are sent to the cordic block sequentially. Previously, the cordic block was only able to handle 1 input however registers have been added to the cordic block so that multiple computations can be performed at once. The amount of resources used is shown in table 11.

Hardware Description		**FPGA Resources**
Cache Size	32KB	57.85%
On-Chip Memory	0KB	
Logic Elements	13,621/15,408 (88%)	
Memory Bits	291,972 / 516,096 (57%)	
PLLs	1 / 4 (25 %)	
Embedded Multiplier 9-bit Elements	32 / 112 (29%)	

Table 11: Resource usage for task 8

5.2 Results for Task 8

5.2.1 Evalution of Function on NIOS II Processor

Listing 8 presents the function `sumVector_COMPLEX_EXPRESSION_IDEAL` that is used to test the new design.

```
// function that utilises custom hardware to evaluate custom instruction
float sumVector_COMPLEX_EXPRESSION_IDEAL(float x[], int M)
{
    int i;
    // variable to hold final sum
    float running_sum = 0;
    // check if number of elements is even
    if(N%2 == 0){
        for(i=0; i<M; i = i+2){
            // perform both evaluation of function and floating point addition in hardware
            running_sum = ALT_CI_COMPLEX_EXPRESSION_IDEAL_0(1, sum, ALT_CI_COMPLEX_EXPRESSION_IDEAL_0(0, x[i], x[(i+1)]));
        }

    }
    // else if number is odd
    else{
        for(i=0; i<(M-2); i = i+2){
            // perform both evaluation of function and floating point addition in hardware
            running_sum = ALT_CI_COMPLEX_EXPRESSION_IDEAL_0(1,sum,ALT_CI_COMPLEX_EXPRESSION_IDEAL_0(0,x[i],x[(i+1)]));
        }
        // add last element
        running_sum = ALT_CI_COMPLEX_EXPRESSION_IDEAL_0(1,sum,ALT_CI_COMPLEX_EXPRESSION_IDEAL_0(0,x[M-1],0));
    }
    // return the value of running_sum
    return running_sum;
}
```

Listing 8: `sumVector_COMPLEX_EXPRESSION_IDEAL`

5.2.2 Comparison of Latency

The updated design is tested and the results are presented in table 8.

N	Step	Task 5 (ticks)	Task 6 (ticks)	Task 7 (ticks)	Task 8 (ticks)
52	5	19	16	1	0
511	0.5	185	164	5	2
2551	0.1	927	820	21	7
25501	0.01	9259	8182	209	74
255001	0.001	92801	82010	2090	745

Table 12: Comparison of latency improvement for task 8

The results obtained are exactly as expected. Evaluating two data elements at once and performing all floating point operations in custom hardware significantly decreased the latency.

5.2.3 Comparison of Accuracy

The accuracy of the updated system is studied. The results are presented in table 13.

N	Step	Error From Reference			
		Task 5	Task 6	Task 7	Task 8
52	5	-0.003	-0.007	-0.003	-0.005
511	0.5	0.081	0.093	0.062	-0.001
2551	0.1	-1.141	-0.875	-1.391	-0.172
25501	0.01	14.945	11.945	11.945	10.195
255001	0.001	-202.600	-214.600	-154.600	213.390

Table 13: Comparison of accuracy for task 8

The results obtained are as expected. There is a slight variation in the error obtained, however this is due to the fact that floating-point arithmetic is not commutative/dissociative; the order in which operations are performed will slightly change the final result obtained.

5.3 Fixed-Point Number System Analysis

In this section, analysis will be conducted to determine the ideal fixed-point number system. As mentioned in section 4.2.2, the fixed-point number system was chosen such that the word would be 32 bits long. By determining the range of the real numbers that the fixed-point number system was required to represent, it was noted that only 2 integer bits are required. The other 30 bits were used as fractional bits.

Following this, it was observed that the precision of the results obtained was actually better than the precision of using the `math.h` library on the NIOS II processor. This was expected. **The `math.h` library operates on single precision floating-point numbers that only contain** 23 **mantissa bits and thus the implementation discussed in section 4.2.2 that uses** 30 **fractional bits is expected to yield more accurate results.**

While considering the ideal fixed-point number system, it is key to identify the two variable parameters.

1. **The number of iterations of the cordic algorithm**: The number of iterations will determine the latency of the cordic block implemented in section 4.2.5. Although other modules within the cordic block contribute to the latency, the main delay comes from evaluating the cosine. The greater the number of iterations, the greater the latency.

2. **The number of fractional bits in the fixed-point number**: The number of fractional bits determines the amount of resources the system will utilise.

Next, it is important to note that the maximum number of iterations is determined by the number of fractional bits. At small angles, the approximation that $arctan(\alpha) \approx \alpha$ becomes valid. As such, if the fixed-point number has 30 fractional bits, the smallest elementary angle that can be represented is $tan(2^{-30}) \approx 2^{-30}$. Beyond the 31^{st} iteration of the cordic algorithm, the elementary angle that has to be subtracted/added to the reference angle cannot be represented by the fixed-point number system. **As such, the number of iterations is upper bounded by the number of fractional bits.**

The converse is not true. The fixed-point number system can have a greater number of fractional bits than the number of iterations of the cordic algorithm. The merits of having more fractional bits than the number of iterations are unclear and thus testing is performed with multiple fixed-point number systems.

To test the precision of the different fixed-point number systems, the test bench presented in appendix A was used. Using the test bench, multiple fixed-point number representations were tested. Three combinations produced results that should be analysed carefully. The combinations are:

1. 30 Fractional bits, 31 Iterations

2. 23 Fractional bits, 24 Iterations

3. 30 Fractional bits, 24 Iterations

The implementation used in section 4.2.5 that uses 30 fractional bits and 31 iterations is used as the baseline case. Figure 7 shows the error that the other two number representations incurred with respect to the baseline case. Note that the error that is graphed is the error incurred in the evaluation of the cosine for a set of input values. The error incurred in the cosine is very strongly correlated to the error incurred in the overall result.

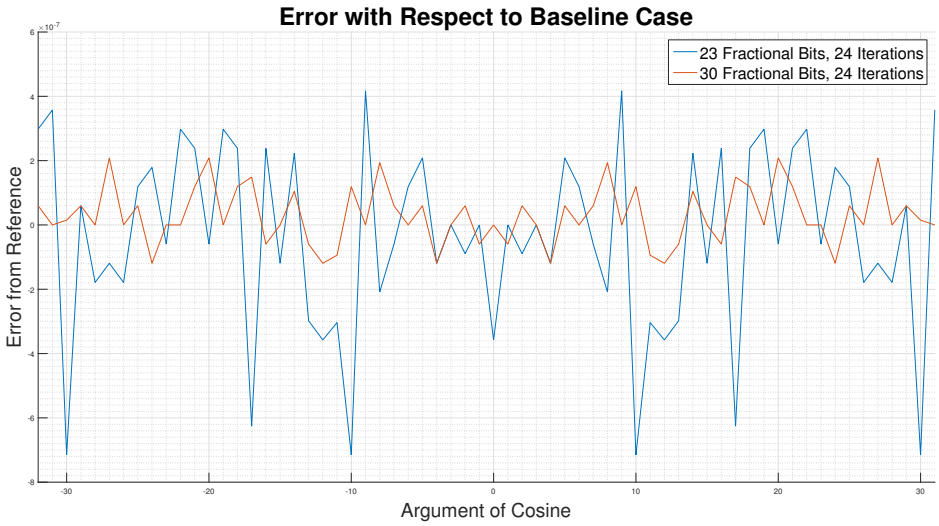

Figure 7: Error incurred for different fixed-point number representations

From the results obtained, two things are clear.

1. **The magnitude of the error incurred at each evaluation of the cosine using only 24 iterations of the cordic algorithm is very small.** The maximum error incurred in any 1 calculation of the cosine is $7.15e-07$. Even if this error was to be incurred for each cosine that is evaluated, the error in the final result will be approximately 4000. In terms of percentage error, this equates to incurring a 0.01% error.
2. **Secondly, it is clear that precision can be improved if the number of fractional bits is increased.** Initially, this result seems unexpected and thus closer inspection of the result is necessary.

Consider a fixed-point number system that has 30 fractional bits,

$$cos(0) = 1$$
$$Fixed\ Point\ Representation : 01.\ 000000000000000000000000\ 0000000$$

$$cos(30) = 0.1542514$$
$$Fixed\ Point\ Representation : 00.001\ 00111011111010000010111\ 1001$$

The highlighted bits indicate the bits that will be used to form the mantissa when the number is converted from a fixed-point representation to a floating point representation. Next consider a fixed-point number system that has only 23 fractional bits,

$$cos(0) = 1$$
$$Fixed\ Point\ Representation : 01.\ 00000000000000000000000$$

$$cos(30) = 0.1542514$$
$$Fixed\ Point\ Representation : 00.001\ 00111011111010000010\ 000 \tag{8}$$

The bits highlighted in red represents the bits that need to be padded to the result to convert the fixed-point number into a floating-point number. It is clear that the number of fractional bits affects the accuracy of result. **Another interesting observation is that there is a greater error incurred for inputs that produce small results. The smaller the result, the greater the number of bits that need to be padded and the greater the error that will be incurred.** Using only 24 iterations but 30 fractional bits is balanced trade off between speed and resources.

Lastly, the 3 fixed point number systems are tested on the NIOS II processor and the results obtained are presented in table 14. **The error presented is with respect to the MATLAB reference value.**

N	Step	30 Fractional Bits		23 Fractional Bits		30 Fractional Bits	
		31 Iterations		24 Iterations		24 Iterations	
		Time (ticks)	Error from Reference	Time (ticks)	Error from Reference	Time (ticks)	Error from Reference
52	5	0	0.005	0	0.044	0	0.013
511	0.5	2	-0.001	1	0.687	2	0.085
2551	0.1	7	-0.172	7	3.078	7	0.570
25501	0.01	74	10.195	72	43.070	72	13.195
255001	0.001	745	213.390	726	-389.390	727	161.390

Table 14: Fixed-Point Number System Analysis

The results corroborate the findings obtained through simulation.

- **Reducing the number of iterations speeds up the evaluation of the complex expression. If the number of iterations is kept constant, but the number of fractional bits is increased, the latency of the system does not increase. The latency depends solely on the number of iterations of the cordic algorithm.**
- **Likewise increasing the number of fractional bits without changing the number of iterations reduces the error.**

A more detailed analysis for the ideal fixed point number representation can be performed, however is skipped due to the lack of time.

5.4 Final Design

The final implementation has 30 fractional bits in the fixed-point number and will perform 24 iterations of the cordic algorithm. The custom instruction takes 2 elements as inputs, evalutes the complex expression of each input sequentially; a pipeline is in place and thus the delay between the computations is just 1 clock cycle. It returns the sum of the two results. The custom instruction is also capable of performing floating point addition. The overall system has a latency 727 ticks and a percentage error of 4.359e-4%. It uses 82% of the logic elements, 57% of the memory bits and 29% of the embedded multipliers available; this adds up to using 56% of the FPGA resources.

5.5 Final Comments

The final design presented in section 5.3 can still be improved. Presented below is a discussion of what further improvements would entail.

Firstly, there are two approaches to evaluating 2 elements of the input vector within 1 call of the custom instruction. **The data points can either be evaluated sequentially or they can be evaluated in parallel.** Design consideration 4 states that the amount of resources should be minimised, however if resources can be traded for a reduction in latency, then the design that reduces latency will be preferred. In section 5, the design processed the two inputs `dataa` and `datab` sequentially. Processing data sequentially will increase the latency, albeit not significantly. As such, the two data points could have been evaluated in parallel. Without testing, it is hard to say how sizable the impact of processing the two elements in parallel would be. By decreasing the number of iterations of the cordic algorithm from 31 to 24, 7 clock cycles, for each custom instruction call, were saved. If the instruction is called 255001 times, the decrease in latency should be about 35 ticks. As seen from table 14, the decrease in latency was actually 19 ticks for test case 3. Saving 2 clock cycles by processing the two elements in parallel will ideally save 10 ticks. If not for the lack of time, experimentation would have been conducted. The entire hardware presented in section 4.3 could have been duplicated and the latency analysed.

Secondly, it was noted that the final design in section 5.3 only performed 1 iteration of the cordic algorithm in each clock cycle. This is an extremely inefficient implementation. Processing 1 iteration of the cordic algorithm using combinations logic does not take $20ns$**. To reduce the latency**

of the design, it is preferable to use as much of each clock cycle to perform computations, i.e to minimise the slack time. The design in section 4.3 spends a large proportion of each clock cycle idle. As such, more iterations of the cordic algorithm should be performed in each clock cycle.

If all 31 iterations were performed in 1 clock cycle, the critical path of the combinatorial logic will be too long; the design will need a clock that is slower than $50Mhz$. To reduce the length of the critical path and ensure that the design works well with a $50Mhz$ clock, registers can be added to hold intermediary values. This is called pipelining. Registers are typically introduced in digital systems to break computations into smaller parts. This will reduce the critical path between combinatorial logic and increase the maximum operating frequency. In this application, the clock frequency is fixed and thus it is advisable to only break the computations into the minimum number of stages required to meet an operating frequency of $50MHz$.

Lastly, a great deal of time was spent trying to configure the control signals needed for direct memory access. A 24 stage pipeline was designed and simulated on ModelSim. The proposed design worked exactly as expected; it was able to cope with data that was streamed in; the system could handle processing a new data every clock cycle. In hindsight, it was foolish to implement a 24 stage pipeline. As discussed above, by performing multiple iterations in 1 clock cycle, the number of pipeline stages could have been significantly reduced. The control signals for the Avalon Memory-Mapped Master were configured. Initial testing was done and the system was able to read the input vector directly from memory. However, once the entire system was designed and data was streamed in from memory, the results obtained were extremely inconsistent.

6 Conclusion

In conclusion, although the final design can still be further optimised, it is clear that incorporating custom hardware to perform complex equations can significantly improve the performance of a digital system.

Throughout the Digital Systems Design coursework, both students picked up invaluable skills necessary for designing digital systems. It was thoroughly enjoyable.

A Test Bench used to Evaluate Fixed-Point Number Systems

```
1  //----------------------------------------------------------------------------------------------------
2  // define timescale and operating precision
3  `timescale 1ns/100ps
4
5  //----------------------------------------------------------------------------------------------------
6  // MODULE NAME:
7  //      cosine_tb
8  //
9  // MODULE FUNCTION:
10 //      test bench to evaluate different fixed point number representations
11 //      works only with cosine block rather than entire system
12 //      cosine block is fed with inputs read from text file
13 //      64 inputs are tested
14 //      outputs are written to text file such that date can be analysed in MATLAB
15 //      num_frac is a parameter that defines the number of fractional bits
16 //      num_iter is a parameter that defines the number of iterations of the cordic algorithm
17 //
18 module cosine_tb();
19
20
21 //----------------------------------------------------------------------------------------------------
22 // parameters for cosine_tb module
23 parameter num_frac = 30;
24 parameter num_iter = 30;
25 // length of K depends on the number of fractional bits
26 parameter K = 'b00100011011011101001110110110101O;
27
28 //----------------------------------------------------------------------------------------------------
29 // parameters needed for file and for loop
30 // N_vectors defines the number of inputs that the cosine block is tested for
31 parameter num_inputs = 64;
32 reg [num_frac+1:0]  inputs_to_be_tested [num_inputs:0];
33
34 //----------------------------------------------------------------------------------------------------
35 // inputs to Device-Under-Test (DUT): need to be regs
36 reg                 clk;
37 reg                 clk_en;
38 reg [num_frac+1:0]  datain;
39
40 //----------------------------------------------------------------------------------------------------
41 // outputs from DUT: need to be wire
42 wire  [num_frac+1:0]  dataout;
43
44 //----------------------------------------------------------------------------------------------------
45 // integers needed for file and for loop
46 integer f_input;
47 integer f_output;
48 integer i, j;
49
50 //----------------------------------------------------------------------------------------------------
51 // instantiate DUT
52 // using named instantiation
53 cosine CFP1(.clk(clk),
54             .clk_en(clk_en),
55             .datain(datain),
56             .fixed_result(result));
57
58 //----------------------------------------------------------------------------------------------------
59 // create a 50MHz clock
60 always
61     // invert clock every 10ns seconds
62     #10 clk = ~clk;
63
64 //----------------------------------------------------------------------------------------------------
65 // read input from file and store in vector t
66 initial begin
67
68     // open input file in read mode
69     f_input = $fopen("input", "r");
70
71     // display that input file is being read on console
72     $display($time, " << Reading Input Vector >> ");
73
74     // all inputs to be tested are read into vectored named inputs_to_be_tested
75     for(i = 0; i<num_inputs; i = i+1) begin
76         // read input from file into vector
77         $fscanf(f_input, "%b", inputs_to_be_tested[i]);
78
79     end
80
81     // display that all inputs to be tested have been read
82     $display($time, " << Finished Reading Input Vector >> ");
83
84     // close the input file
85     fclose(f_input);
86
87 end
88
```

```
89  //----------------------------------------------------------------------------------------------------
90  // initial block sends input into cordic block
91  initial begin
92
93      // initialise all inputs to 0 at time 0
94      clk = 1'b0;
95      en  = 1'b0;
96      datain = 'b0;
97
98      // wait 10 nanoseconds and assert clk_en
99      #10
100     clk_en = 1'b1;
101
102     for( i = 0; i < N_vectors; i = i +1) begin
103
104         // feed input
105         datain = inputs_to_be_tested[i];
106
107          // wait 1 clock cycle
108         #20
109
110     end
111
112 end
113
114 //----------------------------------------------------------------------------------------------------
115 // initial block reads outputs and prints to file
116 initial begin
117
118     // open the output file where results are to be printed
119     // file opened in write mode
120     // name of file depends on num_iter and num_frac
121     f_output = $fopen("cordic_output_30_fractional_31_iterations.txt", "w");
122
123     // initial delay: waiting for pipeline to fill up
124     // delay is depended on number of iterations
125     #630
126
127     // after initial delay results will be ready every clock cycle
128     for( j = 0; j < N_vectors; j = j +1) begin
129
130         // display result on screen
131         // display index of for loop as well
132         $display($time, " index: %d, dataout = %b ", j-32, dataout);
133
134         // write result to output file
135         $fwrite(f_output, "%b\n", dataout);
136
137         // wait 1 clock cycle
138         #20
139
140     end
141
142     // close output file
143     $fclose(f_output);
144
145 end
146 endmodule
```

Listing 9: Test bench used to evaluate number systems