EE3-05 Digital System Design
Report – Coursework Tasks 6 to 8

Group ID: 15
Fung Yee Lorraine Choi (CID: 00733301 | E-mail: fyc12@imperial.ac.uk)
Pok Yee Kwan (CID: 00734359 | E-mail: pyk12@imperial.ac.uk)

Table of Contents

1 Introduction

In previous tasks, in-built functions are used to evaluation expression shown in *Equation 1*, and the elements of input vector x is within the range of 0 to 255. This report focuses on accelerating the system by designing custom instructions in Quartus II. The report below will discuss the design choices made in different stages, as well as the advantages and limitations of each choice. *Table 1* shows the test cases used in this report.

Total resources available for this system includes 15408 logic elements, 347 pins, 516096 memory bits, 112 embedded multipliers and 4 phase-lock loops (PLL). When comparing execution time against total resources used, the percentage of total resources consumed is calculated by the ratio of available resources, illustrated in *Equation 2*.

$$f(x) = \sum_{i=1}^{N} x + x^2 \tan^{-1}\left(floor\left(\frac{x}{4}\right) - 32\right)$$

Equation 1: Function to be evaluated in System

$$\%\,Resources\ Used = \frac{\frac{logic\ elements\ used}{15408} + \frac{pins\ used}{347} + \frac{memory\ bits\ used}{516096} + \frac{embedded\ multipliers\ used}{112} + \frac{PLL\ used}{4}}{5}$$

Equation 2: Expression to Evaluate Proportion of Resources Consumed

Test Case	Step	N
1	5	52
2	0.1	2551
3	0.001	255001

Table 1: Test Cases Used in Computation

2 Task 6: Dedicated Hardware for Basic Arithmetic Operations

2.1 Aim

In previous tasks, in-built functions are emulated through the system to evaluate the expression. In *Task 6*, the four basic arithmetic functionalities (addition, subtraction, multiplication and division) are designed in hardware in order to speed up the computation. Fixed-point and floating-point hardware performance are investigated in order to choose the best fit for the system.

2.2 Approach

The two types of basic arithmetic hardware operation options are fixed-point and floating-point. In general, fixed-point instructions are combinational and floating-point instructions are multi-cycle, so fixed-point arithmetic blocks consumes less resources and computes quicker in general. Yet floating-point modules gives a much wider dynamic range.

Therefore, design choices have to be made according to the requirements of each term within *Equation 1* to optimise performance. To obtain an answer with better precision and accuracy, floating-point arithmetic operations should be used. Therefore, floating-point operations should be chosen for the four basic arithmetic operations.

2.3 Addition and Subtraction

Floating-point addition and subtraction are required to compute *Equation 1*. To investigate the performance of floating-point and fixed-point adder, both types of adders are created using MegaWizard and their performance are compared. Custom instructions are tested using ModelSim and from software independently to verify its functionality.

2.3.1 Performance

Each instruction is called by one million times in the software to give more accuracy in the execution time. Fixed-point and floating-point modules are compared against the in-built add instructions using type **`int`** and **`float`** respectively.

From Quartus II, it is seen that floating-point adder require 7 clock cycles for computation while fixed-point adder only needs 1 clock cycle. Therefore, it is expected that the floating-point module should evaluate 7 times slower than fixed-point module.

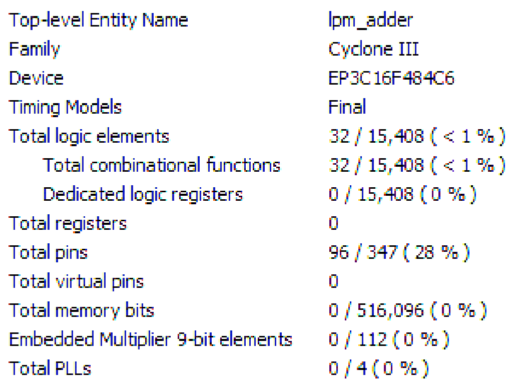
However, from *Table 2*, the execution time of floating-point adder is only 1.17 times quicker than that of fixed-point adder, with a difference of 0.17 µs. One possible reason behind this is there is latency in calling the custom function from the software to hardware, which is a communication latency. Assuming that the latency of calling custom instruction is constant, no matter using fixed-point or floating point adder, then execution time of the fixed-point adder and floating-point adder will be 0.0283 µs and 0.1983 µs respectively.[1] This gives a communication latency of 0.9617 µs. [2]

In-Built Instruction		Custom Instruction	
***Type* `int`**	***Type* `float`**	***Fixed-point***	***Floating-point***
0.989 µs	12.211 µs	0.99 µs	1.16 µs

Table 2: Comparison of Execution Time per Instruction between In-Built and Custom Adder from Software

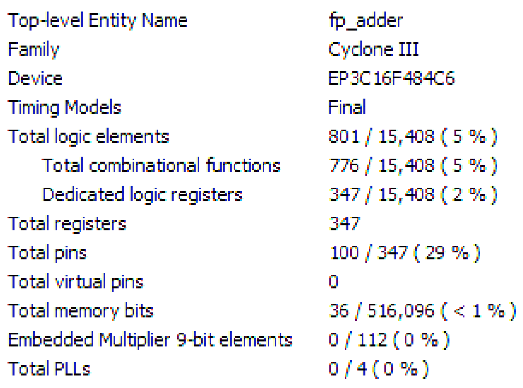
2.3.2 Resources

Figure 1 and Figure 2 shows the resources used by the custom fixed-point and floating-point adder respectively. Fixed-point adder and floating-point adder consumes 5.6% and 6.8% of the available resources respectively.



Top-level Entity Name	lpm_adder
Family	Cyclone III
Device	EP3C16F484C6
Timing Models	Final
Total logic elements	32 / 15,408 (< 1 %)
Total combinational functions	32 / 15,408 (< 1 %)
Dedicated logic registers	0 / 15,408 (0 %)
Total registers	0
Total pins	96 / 347 (28 %)
Total virtual pins	0
Total memory bits	0 / 516,096 (0 %)
Embedded Multiplier 9-bit elements	0 / 112 (0 %)
Total PLLs	0 / 4 (0 %)

Figure 1: Flow Summary of Fixed-Point Adder



Top-level Entity Name	fp_adder
Family	Cyclone III
Device	EP3C16F484C6
Timing Models	Final
Total logic elements	801 / 15,408 (5 %)
Total combinational functions	776 / 15,408 (5 %)
Dedicated logic registers	347 / 15,408 (2 %)
Total registers	347
Total pins	100 / 347 (29 %)
Total virtual pins	0
Total memory bits	36 / 516,096 (< 1 %)
Embedded Multiplier 9-bit elements	0 / 112 (0 %)
Total PLLs	0 / 4 (0 %)

Figure 2: Flow Summary of Floating-Point Adder

[1] As floating-point adder should be 7 times slower than fixed-point adder, and from *Table 2* floating-point adder is slower than fixed-point adder by 0.17 µs. Therefore, execution time of fixed-point adder is 0.17/6 = 0.0283 µs and execution time of floating-point adder is 0.0283 * 7 = 0.1983 µs.

[2] Latency in function calling is calculated by the difference between the execution time shown in *Table 2* and the calculated instruction execution time as stated in *Footnote 1*, i.e. 0.99 – 0.0283 = 0.9617 µs.

2.4 Multiplication and Division

Floating-point multiplication and division are required to evaluate *Equation 1*. Both fixed-point and floating-point multipliers are created to investigate their performance. Although floating-point division is needed, a divider is not necessary in this system. Dividing a floating-point number by 4 is equivalent to subtracting 2 from the exponent. Since a fixed-point subtractor evaluates quicker than a floating-point divider, a fixed-point subtractor is implemented to compute the division by 4 in the expression. Looking at the format of the IEEE standard floating point in *Figure 3*, subtracting exponent by 2 is subtracting 1 at the 25th bit in binary. Using a fixed-point subtractor, that is to subtract 2^{24} or 16777216 from the input number. *Figure 4* illustrates the implementation of the division by 4 using a fixed-point subtractor.

Sign	Exponent	Significand
1 bit	8 bit	23 bit

Figure 3: Standard IEEE-754 Floating Point Format

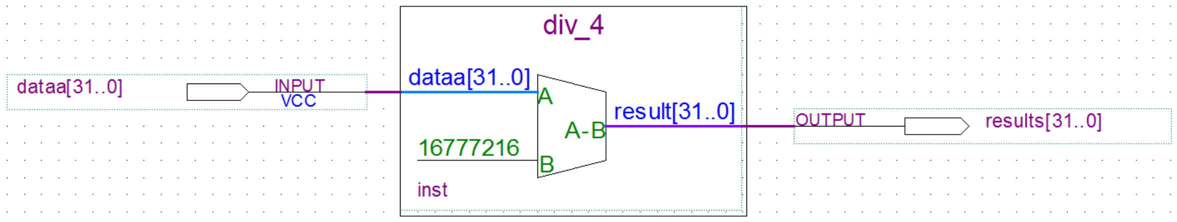


Figure 4: Schematic Design of Divider (Fixed-Point Subtractor)

2.4.1 Performance

To ensure accuracy of the time required to execute one instruction, the function is called by one million times in the software. Fixed-point and floating-point multipliers are compared against the in-built multiplying instruction using type **`int`** and **`float`** respectively.

From Quartus II, a floating-point multiplier would take 5 clock cycles to run while fixed-point multiplier only takes 1 clock cycle. Therefore, theoretically the execution time of the floating-point multiplier instruction should be 5 times of that of the fixed-point multiplier. Yet from *Table 3*, it is observed that the fixed-point multiplier only operates 1.12 times quicker than the floating-point multiplier, with a difference of 0.122 μs.

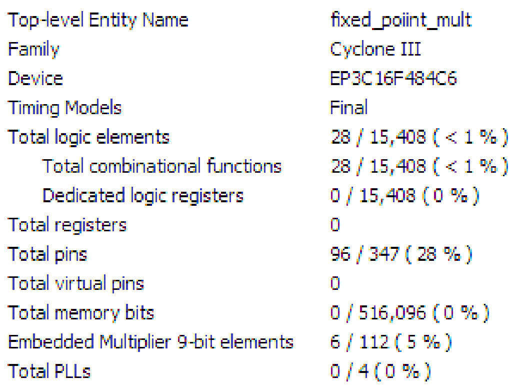
As discussed in *Section 2.3.1*, this could be due to a latency in function calling from software to hardware. Assuming that the communication latency is the same for fixed-point and floating-point multipliers, the real execution time of a fixed-point and floating-point multiplier is 0.0305 μs and 0.1525 μs respectively[3], and the latency for function calling is 0.9585 μs.[4] This communication latency matches with the latency investigated in *Section 2.1.1* (0.9617 μs), giving an average latency of 0.96 μs for calling the custom instruction.

In-Built Instruction		Custom Instruction	
***Type* `int`**	***Type* `float`**	***Fixed-point***	***Floating-point***
1.068 μs	15.4 μs	0.989 μs	1.111 μs

Table 3: Comparison of Execution Time between In-Built and Custom Multiplier from Software

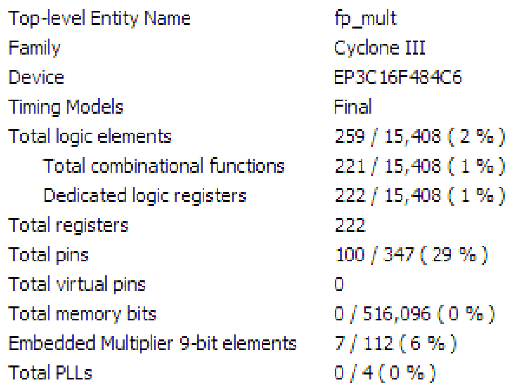
2.4.2 Resources

The total resources consumed in fixed-point and floating-point multipliers are shown in *Figure 5* and *Figure* 6. Out of the total resources available, fixed-point and floating-point multipliers uses 6.65% and 7.35% resources respectively.



Top-level Entity Name	fixed_poiint_mult
Family	Cyclone III
Device	EP3C16F484C6
Timing Models	Final
Total logic elements	28 / 15,408 (< 1 %)
Total combinational functions	28 / 15,408 (< 1 %)
Dedicated logic registers	0 / 15,408 (0 %)
Total registers	0
Total pins	96 / 347 (28 %)
Total virtual pins	0
Total memory bits	0 / 516,096 (0 %)
Embedded Multiplier 9-bit elements	6 / 112 (5 %)
Total PLLs	0 / 4 (0 %)

Figure 5: Flow Summary of Fixed-Point Multiplier



Top-level Entity Name	fp_mult
Family	Cyclone III
Device	EP3C16F484C6
Timing Models	Final
Total logic elements	259 / 15,408 (2 %)
Total combinational functions	221 / 15,408 (1 %)
Dedicated logic registers	222 / 15,408 (1 %)
Total registers	222
Total pins	100 / 347 (29 %)
Total virtual pins	0
Total memory bits	0 / 516,096 (0 %)
Embedded Multiplier 9-bit elements	7 / 112 (6 %)
Total PLLs	0 / 4 (0 %)

Figure 6: Flow Summary of Floating-Point Multiplier

[3] As floating-point multiplier is 5 times slower than fixed-point multiplier and the difference in execution time is 0.122 μs, the real execution time of fixed-point and floating-point instructions without function call latency is (0.122/4 =) 0.0305 μs and (0.0305*5 =) 0.1525 μs respectively.

[4] Communication Latency = 0.989 – 0.0305 = 0.9585 μs

2.5 Integrated System with Custom Instructions

From the above discussions, the hardware blocks required for the basic arithmetic operations are floating-point adder, fixed-point subtractor and floating-point multiplier. After verifying the functionality of each individual block, the custom instructions are implemented in the software system in order to test the improvement. *Code 1* shows the software implementation of the *sumVector()* function using floating-point multiplier and adder, and fixed-point subtractor.

```
#define ALT_CI_FP_MULT(A,B) __builtin_custom_fnff(ALT_CI_FP_MULT_0_N,(A),(B))
#define ALT_CI_FP_ADD(A,B) __builtin_custom_fnff(ALT_CI_FP_ADD_0_N,(A),(B))
#define ALT_CI_LPM_ADD(A, B) __builtin_custom_fnfi(ALT_CI_LPM_ADD_0_N,(A),(B))

float sumVector(float x[], int M)
{
	int i =0 ;
	float sum=0.0;
	float temp;

	for (i=0;	i<M;	i++)
	{
		temp = floor(ALT_CI_LPM_ADD(x[i] , -16777216));
		temp = atan(ALT_CI_FP_ADD(temp , -32));
		temp = ALT_CI_FP_MULT(x[i] , temp);
		temp = ALT_CI_FP_MULT(x[i] , temp);
		temp = ALT_CI_FP_ADD(x[i] , temp);
		sum = ALT_CI_FP_ADD(sum , temp);
	}

	return sum;
}
```

Code 1: Software Implementation of sumVector() with Custom Instructions

2.5.1 Performance

2.5.1.1 Execution Time

In order to further verify the improvements on the design, the newly designed custom system is compared against the legacy system from Task 5 and a basic system using custom floating-point adder and multiplier only. *Table 4* displays the execution time of the three systems for easy comparison. It is seen that there is significant improvement in execution time by using custom instructions, especially in *Test Case 2* and *3*, which speeds up the system by 69%.

The main difference between the basic system and custom system is an extra fixed-point subtractor in the custom system, which executes the floating-point division. This shows that using a fixed-point subtractor instead of a floating-point multiplier to compute floating-point division reduces the execution time by an average of 0.86%.

Test Case	**Execution time (s)**			**Reduction in Time (Task 5 vs Custom)**
	Task 5 System[5]	***Basic System***[6]	***Custom System***[7]	
1	0.0564	0.04756	0.04635	17.819%
2	7.5791	2.3133	2.31245	69.489%
3	759.6486	232.0558	232.0404	69.454%

Table 4: Comparison of Execution Time with Different Systems

2.5.1.2 Accuracy and Precision

Table 5 gives the result of evaluating the expression through the system and MATLAB respectively. *Test case 1* gives almost 100% accuracy, while *Test Case 2* and *3* have a tiny offset of 0.07% and 0.55% respectively. Therefore, *Test Case 2* and *3* are investigated in further detail, and the percentage error of each cumulative sum is plotted in *Figure 7* and *Figure* 8.

It is observed that the system gives minimal error at almost 0%, until the cumulative sum changes from negative to positive (zero-crossing). It could be caused by a small rounding error cumulated throughout the summation and led to a shift in the zero-crossing position. Yet the error then falls back to around 0% after the zero-crossing part and gives a final result with high accuracy.

Test Case	**Result**		**Error**
	Task 6 System	***MATLAB***	
1	1285232	1285232.047667944	-3.7x10^6 %
2	61628392	61673882.55173704	-0.074%
3	6193364480	6159802445.884553	0.55%

Table 5: Testing Result of Custom System

[5] Original system using instructions emulated through software with the use of embedded multiplier
[6] System with custom floating-point adder and multiplier implemented only
[7] System using design choices discussed and made in *Section 2*

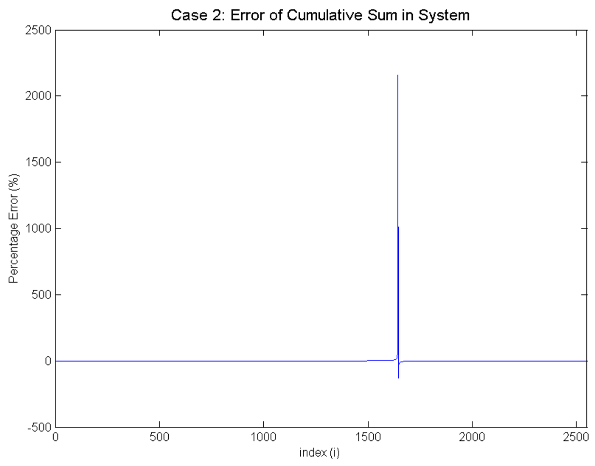


Figure 7: Cumulative Sum Error for Case 2

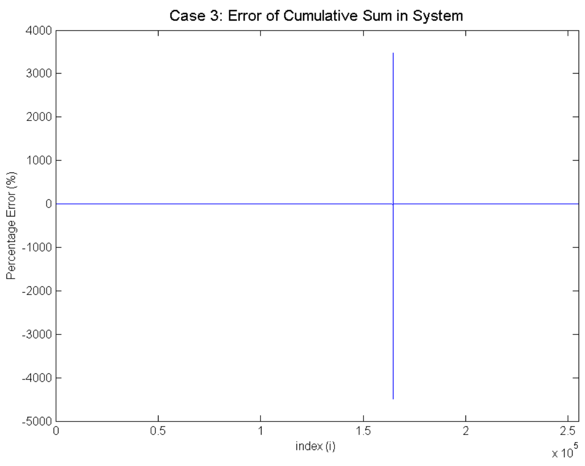
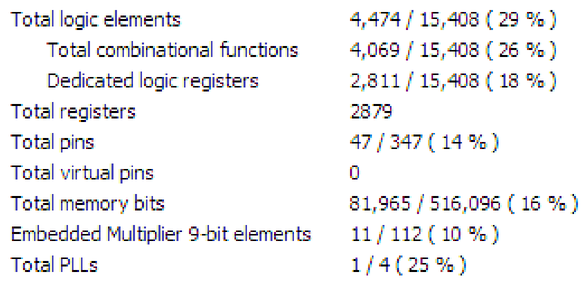


Figure 8: Cumulative Sum Error for Case 3

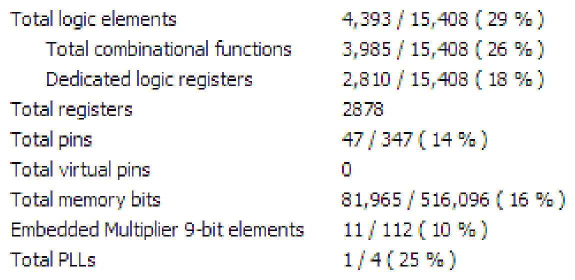
2.5.2 Resources

The flow summaries of the custom-designed system and basic system is displayed in *Figure 9* and *Figure* 10. The proportion of resources consumed in the custom system and basic system are 18.7% and 18.6% respectively. Comparing the two systems, an increase of 0.1% in resources consumption resulted in an improvement of 0.86% in execution time, which is an acceptable trade-off between resources and performance.



Total logic elements	4,474 / 15,408 (29 %)
Total combinational functions	4,069 / 15,408 (26 %)
Dedicated logic registers	2,811 / 15,408 (18 %)
Total registers	2879
Total pins	47 / 347 (14 %)
Total virtual pins	0
Total memory bits	81,965 / 516,096 (16 %)
Embedded Multiplier 9-bit elements	11 / 112 (10 %)
Total PLLs	1 / 4 (25 %)

Figure 9: Flow Summary of Custom System



Total logic elements	4,393 / 15,408 (29 %)
Total combinational functions	3,985 / 15,408 (26 %)
Dedicated logic registers	2,810 / 15,408 (18 %)
Total registers	2878
Total pins	47 / 347 (14 %)
Total virtual pins	0
Total memory bits	81,965 / 516,096 (16 %)
Embedded Multiplier 9-bit elements	11 / 112 (10 %)
Total PLLs	1 / 4 (25 %)

Figure 10: Flow Summary of Basic System

3 Task 7: Dedicated Hardware Block for Inner Part Computation

3.1 Aim

In *Section 2*, the basic arithmetic operations are implemented using hardware blocks. In the following section, the *arctan* and *floor* function used in the expression will be designed using hardware.

3.2 Floor Function Implementation

One way of implementing *floor* function is to make use of the default *ALTFP_CONVERT* module in Quartus II that rounds off a floating point to an integer. Yet the required function needs to round down the floating point, and an additional subtractor is needed to subtract 0.5 from the input number to give the *floor* functionality.

Another way to implement the *floor* function is shown in *Code 2*. This method uses the concept of truncation and the IEEE floating point format. Rounding down is equivalent to taking the integer part of a number, or truncating the fraction part. As floating point number is given by $\pm significand \times 2^{exponent}$, the exponent determines how many bits does the binary point shift. For example when exponent is 0, the rounded number must be 1 bit long. When exponent is 1 the rounded number must be 2 bit long, or shifting the binary point of the significand by 1 bit to the right. The code reads the exponent (bit 23 to 30) and determines how many bits of the mantissa to be obtained.

As the input value to the *floor* function is *x/4* and *x* is within the range of 0 to 255, so the input range to the *floor* function is 0 to 64. Therefore, only exponents from 0 to 6 are within the range, and the code takes 7 cases only. Any exponents out of the range will be set to 0 as default. As it is stated in the specification that x lies within [0 , 255], this dedicated hardware unit will be functional within this range only.

3.2.1 Performance

ModelSim is used to simulate the two possible implementations of the *floor* function. The *ALTFP_CONVERT* method would take 5 clock cycles to execute, while the truncation method would only take 1 clock cycle. As the aim of this system is to give the best performance, the truncation method is chosen.

To verify the functionality and improvement of the dedicated *floor* function, the instruction is ran a million times and compared against that of the in-built *floor()* function emulated through software. *Table 6* shows the execution time per instruction, using the default *floor* function and the custom designed *floor* function respectively. The dedicated hardware reduces 98.86% of the execution time, or 87 times quicker as compared to the in-built *floor* function.

```
module floor (clock, dataa, result);

input        clock;
input        [31:0] dataa;
output       [31:0] result;
reg          [31:0] result;
wire         [31:0] dataa;

always @ (posedge clock)
begin
       case(dataa[30:23])
              8'b10000101: result = {25'b0,1'b1,dataa[22:17]};
              8'b10000100: result = {26'b0,1'b1,dataa[22:18]};
              8'b10000011: result = {27'b0,1'b1,dataa[22:19]};
              8'b10000010: result = {28'b0,1'b1,dataa[22:20]};
              8'b10000001: result = {29'b0,1'b1,dataa[22:21]};
              8'b10000000: result = {30'b0,1'b1,dataa[22]};
              8'b01111111: result=32'b1;
              default: result[31:0]=32'b0;
       endcase
end
endmodule
```

Code 2: Floor Function Hardware Implementation Using Verilog HDL

Execution Time (µs)		**Reduction in Time**
floor() from math.h library	***custom floor() using truncation***	
66.199	0.756	98.86%

Table 6: Comparison of Execution Time using Default and Custom floor Function

3.3 arctan Function Implementation

3.3.1 Design Choices

CORDIC algorithm is used to implement the *arctan* function, and a 10-stage bit-parallel unrolled CORDIC is chosen for this purpose. The design of the single-stage CORDIC is displayed in *Figure 12* and the 10-stage design is put together using Verilog HDL. The three parameters used in CORDIC is denoted as *x*, *y* and *z*.

CORDIC computes trigonometric functions by performing pseudorotations. Error in the answer will be lowered by performing more pseudorotations. Each pseudorotation is conducted by a single-stage CORDIC, and more resources will be consumed by using more stages. Considering the trade-off between resources and accuracy, a 10-stage CORDIC is chosen as it gives a small error of 0.1 degree or 0.00195 radians, while consuming an acceptable amount of resources.

An unrolled architecture is picked instead of the iterative structure as the former allows pipelining while the latter does not. Bit-parallel input is selected to reduce the time to read input. Using the relationship $\arctan\left(\frac{1}{y}\right) = \frac{\pi}{2} - \arctan(y)$ could limit the range of fixed-point numbers encountered. However, this design would require extra resources to build a multiplexer that determines whether a reciprocal is needed and a subtractor to compute $\frac{\pi}{2} -$

$\arctan(y)$. As the input range of the *arctan* function in this system is -32 to 31, the input size should be manageable and it is not necessary to use the identity.

As seen in the schematic diagram that illustrates the single-stage CORDIC design, fixed-point adders are used in the design. As the output of *arctan* is within the range of -1.5 to 1.5, it will only need 2 bits for the integer part of the number, leaving 30 bits for the fraction part. This gives a sufficient precision for the system.

The parameters *x* and *z* are set to be 1 and 0 respectively. Input *y* has to be fixed-point number as fixed-point adders are used in the single-stage CORDIC design. With an input range of -32 to 31, 5 bits for integer part of the number should be sufficient. However, the number could overflow due to summations in the following stages as it passes forward, 8 extra buffer bits are assigned for the integer part. This results in a fixed-point number format of 13 bit integer part and 19 bit fraction part for input *y*.

Figure 11 shows the implementation of the CORDIC algorithm. The *clshift_LL* module sets the fixed-point number format as 13 integer bits and 19 fraction bits by shifting the binary point. The fixed-point number is then fed into the *CORDIC_unrolled* block to perform the 10 pseudorotations, returning an answer in fixed-point format. Therefore, a fixed-point to floating-point converter is required to output the result in floating-point format.

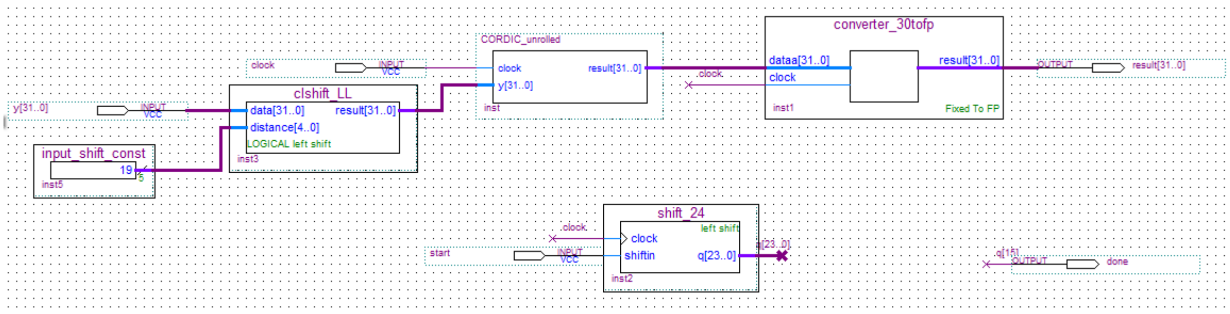


Figure 11: Schematic Diagram of CORDIC Implementation

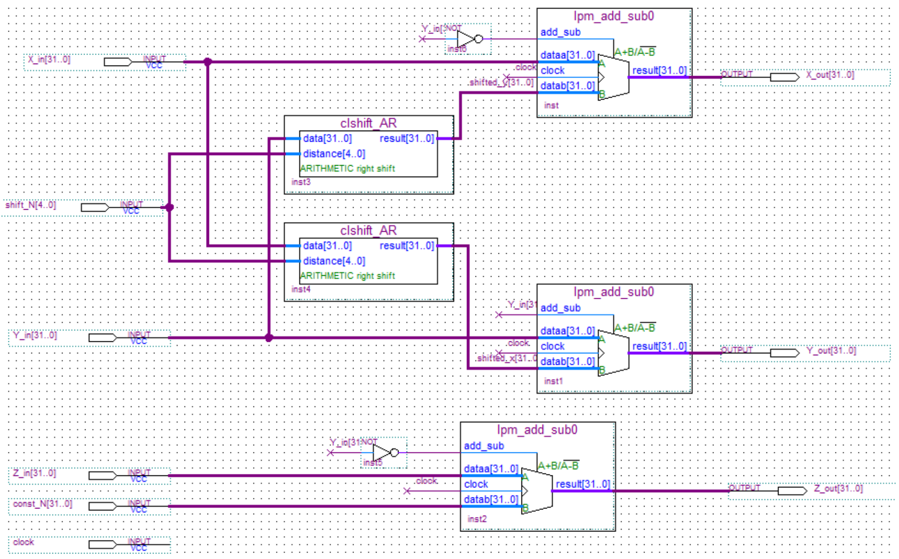


Figure 12: Schematic Design of Single Stage CORDIC

3.3.2 Performance

3.3.2.1 Execution Time

Each CORDIC stage would take 1 clock cycle to execute, giving a total of 10 clock cycles for the *CORDIC_unrolled* block. The converter also takes 6 clock cycles to operate, thus the total latency for each instruction using CORDIC hardware implementation is 16 clock cycles.

The in-built *atan()* function from the math.h library and the custom designed *arctan* function is ran for a million times respectively. *Table 7* gives the execution time per instruction. The dedicated hardware that implemented the CORDIC algorithm reduces the computation time by 99.97%, or runs 2296 times quicker than the in-built *atan* function.

Execution Time (µs)		**Reduction in Time**
atan() from math.h library	***custom atan() using CORDIC***	
2715.6	1.183	99.97%

Table 7: Comparison of Execution Time using Default and Custom arctan Function

3.3.2.2 Accuracy and Precision

The output of the *arctan* function implemented by CORDIC algorithm is compared against the default MATLAB *atan* command. *Figure 13* and *Figure 14* shows the output values and percentage error of the CORDIC implementation as compared to the MATLAB results. The percentage error of all elements are within the range of ±0.25%, giving an accuracy of 99.75%.

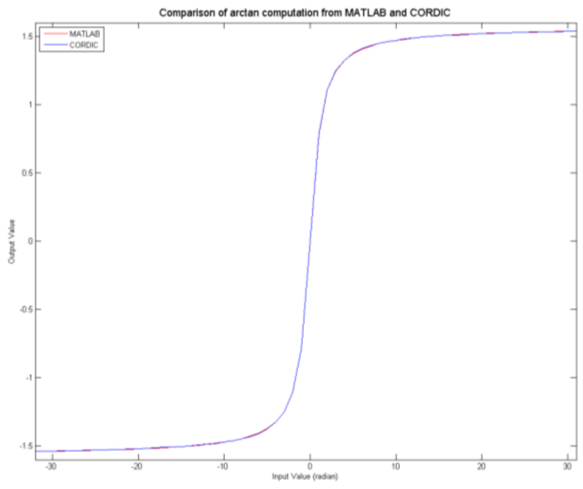


Figure 13: Output of atan() from MATLAB and CORDIC

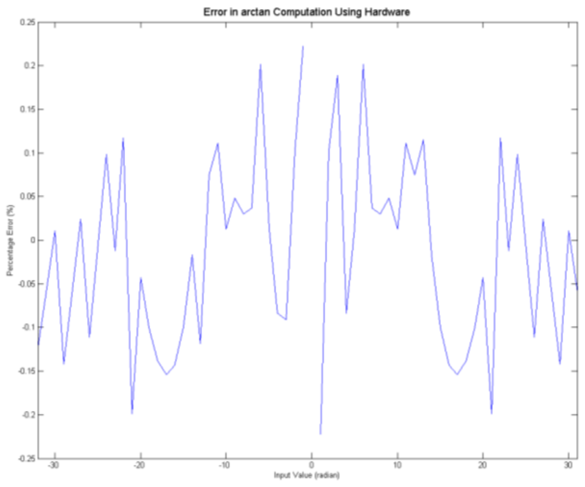
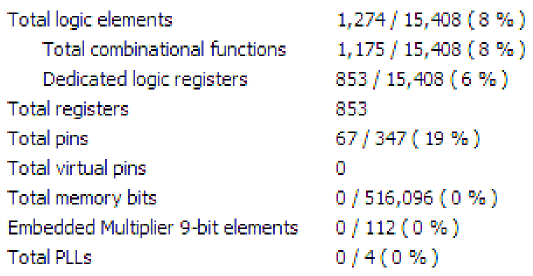


Figure 14: Output Error in CORDIC Implementation

3.3.3 Resources

Flow summary of the custom-designed *arctan* function is displayed in *Figure 15*. 5.5% of total resources is used to implement the CORDIC algorithm, and leads to a reduction of 99.75% in time. Although there is a small trade-off of losing accuracy by 0.25%, the benefit of accelerating the system outweighs this unfavourable effect.



```
Total logic elements                    1,274 / 15,408 ( 8 % )
    Total combinational functions       1,175 / 15,408 ( 8 % )
    Dedicated logic registers           853 / 15,408 ( 6 % )
Total registers                         853
Total pins                              67 / 347 ( 19 % )
Total virtual pins                      0
Total memory bits                       0 / 516,096 ( 0 % )
Embedded Multiplier 9-bit elements      0 / 112 ( 0 % )
Total PLLs                              0 / 4 ( 0 % )
```

Figure 15: Flow Summary of CORDIC Implementation

3.4 Integrated System with Custom-Designed Floor and CORDIC Functions

The hardware implementation of *floor* and *arctan* function are included in the system to replace the instructions emulated through the software, as displayed in *Code 3*. Since the output of $floor\left(\frac{x}{4}\right)$ could be set as integer by using the custom *floor* function, the original floating-point adder that evaluates the term $floor\left(\frac{x}{4}\right) - 32$ is replaced by a fixed-point subtractor, as fixed-point subtractor executes quicker than a floating-point adder.

3.4.1 Performance

3.4.1.1 Execution Time

As discussed in *Section 3.2.1* and *3.3.2.1*, the dedicated hardware that implements the *floor* and *arctan* function could accelerate the computation significantly. On average, the system performs 34 times quicker than the system from *Task 6*.

Test Case	**Execution Time (ms)**		**Reduction in Time**
	Task 6 System	***Task 7 System***	
1	46.35	1.35	97.087%
2	2312.45	67.9	97.064%
3	232040.4	6819.95	97.061%

Table 8: Execution Time of System in Task 6 and Task 7

```
#define ALT_CI_FP_MULT(A,B) __builtin_custom_fnff(ALT_CI_FP_MULT_0_N,(A),(B))
#define ALT_CI_FP_ADD(A,B) __builtin_custom_fnff(ALT_CI_FP_ADD_0_N,(A),(B))
#define ALT_CI_LPM_ADD0(A, B) __builtin_custom_fnfi(ALT_CI_LPM_ADD_0_N,(A),(B))
#define ALT_CI_LPM_ADD1(A, B) __builtin_custom_inii(ALT_CI_LPM_ADD_0_N,(A),(B))
#define ALT_CI_CORDIC(A) __builtin_custom_fni(ALT_CI_CORDIC_0_N,(A))
#define ALT_CI_FLOOR(A) __builtin_custom_inf(ALT_CI_FLOOR_0_N,(A))


float sumVector(float x[], int M)
{
        int i =0 ;
        float sum=0.0;
        float temp;
        int intTemp;

        for (i=0;     i<M;   i++)
        {
                temp = ALT_CI_LPM_ADD0(x[i] , -16777216)
                intTemp = ALT_CI_FLOOR(temp);
                intTemp = ALT_CI_LPM_ADD1(intTemp , -32);
                temp = ALT_CI_CORDIC(intTemp);
                temp = ALT_CI_FP_MULT(x[i] , temp);
                temp = ALT_CI_FP_MULT(x[i] , temp);
                temp = ALT_CI_FP_ADD(x[i] , temp);
                sum = ALT_CI_FP_ADD(sum , temp);
        }

        return sum;
```

Code 3: Software Implementation of sumVector() with Custom floor and arctan Function

3.4.1.2 Accuracy and Precision

From *Table 9*, the latest design has sacrificed a small degree of accuracy to boost the time performance. While comparing the accuracy performance by plotting the cumulative sum of each iteration in *Test Case 2* and *3* with MATLAB results, shown in *Figure 16* and *Figure 17*, it is seen that the results are reasonably similar with an insignificant error building up towards a higher index number. This could be an accumulated rounding error throughout the calculations, as well as the loss of precision by using the CORDIC implementation to compute *arctan*. Given that the main objective of the system is to compute as quickly as possible, this minor drop in accuracy is accepted as a trade-off to speed.

Test Case	**Result**		**Error**	
	Task 7 System	***MATLAB***	***Task 6 System***	***Task 7 System***
1	1284675	1285232.0476679	-3.7x10^6 %	-0.0433%
2	61611196	61673882.551737	-0.074%	-0.1016%
3	6191600128	6159802445.8846	0.55%	0.5162%

Table 9: Comparison of Accuracy against MATLAB and System from Task 6

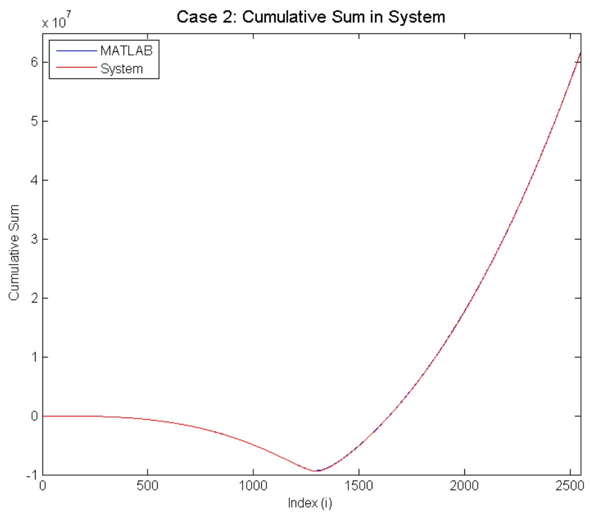


Figure 16: Comparison of Cumulative Sum in Case 2

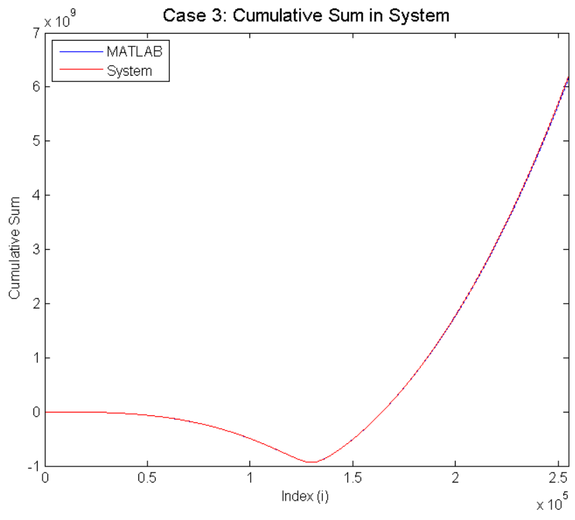
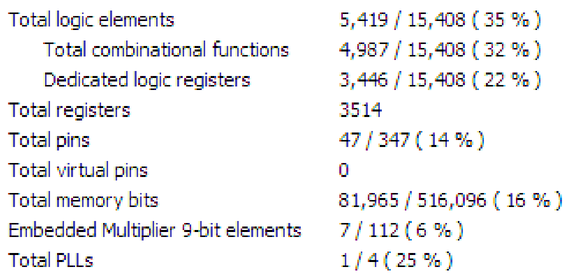


Figure 17: Comparison of Cumulative Sum in Case 3

3.4.2 Resources

The legacy system in *Task* 6 consumes 18.7% of the total resources, and this new system with custom *floor* and *arctan* functions uses 19.2%. Therefore, the extra 0.5% of new resources consumed lead to a reduction of 97% in execution time.



Total logic elements	5,419 / 15,408 (35 %)
Total combinational functions	4,987 / 15,408 (32 %)
Dedicated logic registers	3,446 / 15,408 (22 %)
Total registers	3514
Total pins	47 / 347 (14 %)
Total virtual pins	0
Total memory bits	81,965 / 516,096 (16 %)
Embedded Multiplier 9-bit elements	7 / 112 (6 %)
Total PLLs	1 / 4 (25 %)

Figure 18: Flow Summary of System in Task 7

4 Task 8: Accelerate Computation of Arithmetic Expression

4.1 Aim

In previous tasks, mathematical operation instructions are implemented using hardware. The aim of this task is to boost the performance of the system, which is to shorten the execution time, as much as possible.

4.2 Integrated Functional Block

In *Section 2.3.1*, there is a communication latency of 0.96 μs between each function call from the software to hardware. In order to minimise the effect of this latency, the dedicated hardware to compute the mathematical operations are integrated together in one hardware module, so less instructions are called from the software, thus reduced the execution time by avoided multiple communication latencies. The new software implementation of the *sumVector()* function is shown in *Code 4*.

The functional block *function_pipeline*, shown in *Figure 19*, evaluates the term $x + x^2 \tan^{-1}\left(floor\left(\frac{x}{4}\right) - 32\right)$ by putting the fixed-point subtractor, floating-point adder, floating-point multiplier, truncation *floor* function and CORDIC *arctan* function together. The multiplier below the fixed-point subtractor for division is in parallel with the chain of two subtractors and CORDIC block, and therefore, x^2 and $\tan^{-1}\left(floor\left(\frac{x}{4}\right) - 32\right)$ are computed in parallel and reduces total latency. The *function_pipeline* module is then connected to a floating-point adder to perform summation, as shown in *Figure 20*.

```
#define ALT_CI_FX_ACC(A,B) __builtin_custom_fnff(ALT_CI_FX_ACC_0_N,(A),(B))

float sumVector(float x[], int M)
{
	int i =0 ;
	float sum=0.0;

	for (i=0;	i<M;	i++)
	{
		sum = ALT_CI_FX_ACC(x[i] , sum);
	}

	return sum;
}
```

Code 4: Software Implementation of sumVector() using Integrated Module

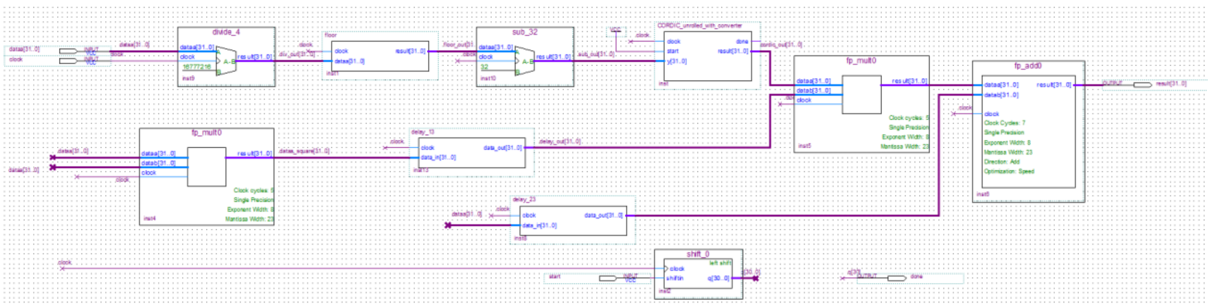


Figure 19: Schematic Diagram of Function_Pipeline Module (Evaluates Expression without Summation)

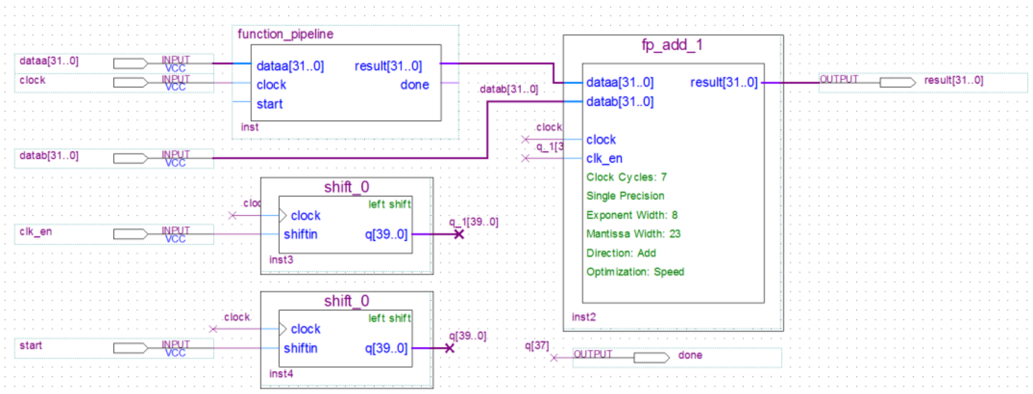


Figure 20: Schematic Diagram of Entire System (Including Summation)

4.2.1 Performance

Without integrating the hardware blocks in the same module, 8 custom instructions have to be called from the software per iteration to compute the expression. After including all blocks in the *function_pipeline* module, only 1 custom instruction is used per iteration, which cuts down 7 unnecessary communication latencies per iteration in the *sumVector()* function.

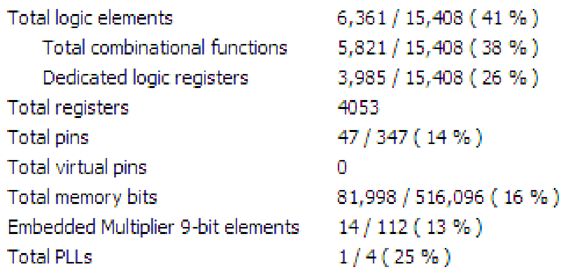
Table 10 records the execution time of the new system using the integrated module approach, and the integrated block system executes 6.7 times quicker than that of the legacy system from *Task 7*.

Test Case	Execution Time (ms)		Reduction in Time
	Task 7 System	***Integrated Block System***	
1	1.35	0.199741	85.204%
2	67.9	10.2952	84.837%
3	6819.95	1024.11	84.984%

Table 10: Execution Time with Integrated Block System Compared to Legacy System

4.2.2 Resources

The legacy system in *Task 7* and the system with integrated module consumes 19.2% and 21.6% of the total resources available respectively. Given that this system reduces the execution time by 85%, the extra 2.4% resources utilised is an acceptable trade-off.



Total logic elements	6,361 / 15,408 (41 %)
Total combinational functions	5,821 / 15,408 (38 %)
Dedicated logic registers	3,985 / 15,408 (26 %)
Total registers	4053
Total pins	47 / 347 (14 %)
Total virtual pins	0
Total memory bits	81,998 / 516,096 (16 %)
Embedded Multiplier 9-bit elements	14 / 112 (13 %)
Total PLLs	1 / 4 (25 %)

Figure 21: Flow Summary of System Using Integrated Module

4.3 Overlap Latencies

With the current design, the total latency of the system is made up of the communication and computational latencies. To reduce the total latency, the communication and computational delays could be overlapped. This is to make use of the delay between each function call. Instead of leaving the system idle, the system would run the module during the communication latency. This approach is illustrated in the signal graph shown in *Figure 22*. To achieve this, the *done* signal is set to be high at all time, which ensures the block is being fed with input constantly. The schematic diagram of this top-level design is displayed in *Figure 23*.

By implementing this method, the first output data must be invalid. Although the first data has to be discarded, it reduces the total latency. The previous system in *Section 4.2*, the number of latencies is the number of times a function is called from software. While the current design would give 1 latency only, no matter how many times a custom instruction is used.

In previous sections, the average latency per instruction call is calculated to be 0.96 µs. This gives a communication latency of 48 clock cycles[8]. The computational latency of the module *function_pipeline* is 31 clock cycles, and that of the floating-point adder is 7 clock cycles, summing to a total computational latency of 38 cycles. This leaves a buffer of 10 clock cycles, as the communication latency is calculated as an average, but not an absolute time for every single instruction call.

[8] $Communication\ Latency\ = \frac{0.96\ \mu s}{\frac{1}{50MHz}}\ =\ 48\ clock\ cycles$

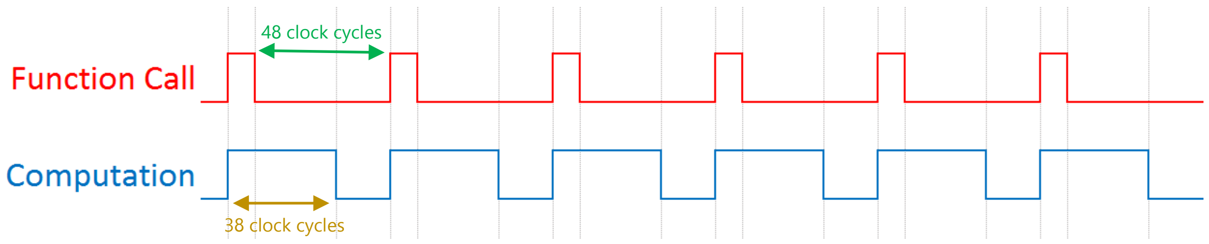


Figure 22: Signal Graph Illustration of Overlapping Latency Approach

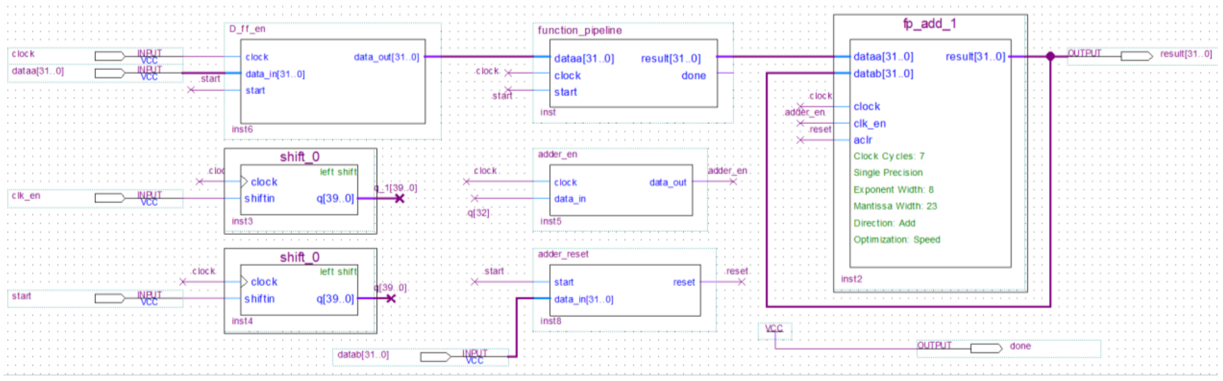


Figure 23: Schematic Diagram of System with Latencies Overlapped

4.3.1 Performance

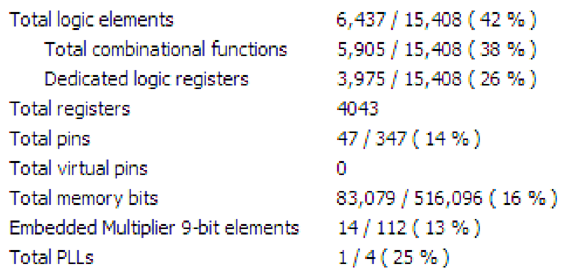
The operation time for one custom instruction is 48 clock cycles as discussed previously. The total execution time is dependent on the length of input vector (*N*). Throughput of the current system is 1, thus the total execution time of the system will be 48N clock cycles plus any overheads. *Table 11* records the time performance of the integrated block system and the current system. The new system operates around 1.3 times quicker than the integrated block system.

Test Case	**Execution Time (ms)**		**Reduction in Time**
	Integrated Block System	***Overlap Latency System***	
1	0.199741	0.15	24.903%
2	10.2952	7.36	28.510%
3	1024.11	731.19	28.602%

Table 11: Comparison of Execution Time in Different Systems

4.3.2 Resources

21.8% resources are consumed, which is 0.2% more than the integrated module approach. This extra 0.2% of resources gives a reduction of 27% in time on average.



Total logic elements	6,437 / 15,408 (42 %)
Total combinational functions	5,905 / 15,408 (38 %)
Dedicated logic registers	3,975 / 15,408 (26 %)
Total registers	4043
Total pins	47 / 347 (14 %)
Total virtual pins	0
Total memory bits	83,079 / 516,096 (16 %)
Embedded Multiplier 9-bit elements	14 / 112 (13 %)
Total PLLs	1 / 4 (25 %)

Figure 24: Flow Summary of System Using Overlap Latency Approach

4.4 Pipelining Using Direct Memory Access

As discussed previously, the communication latency between the software to the hardware is high (48 clock cycles) and is the main constraint in the system. To further improve the system, a direct memory access (DMA) block could be implemented together with Avalon slave to pipeline the whole system, which gets rid of the repeated communication between software and hardware. This is because the original system feeds in data every 48 clock cycles, controlled by the Nios II processor. But the proposed system here instructs the DMA to fetch data from the memory itself without going back to the software. In other words, the overlapping latency system has a total communication latency of $48\, clock\ cycles\ \times input\ vector\ length$ per *sumVector()* call, while the new system has only 48 clock cycles of communication latency pre *sumVector()* call. *Figure 25* illustrates the high level block diagram of this design.

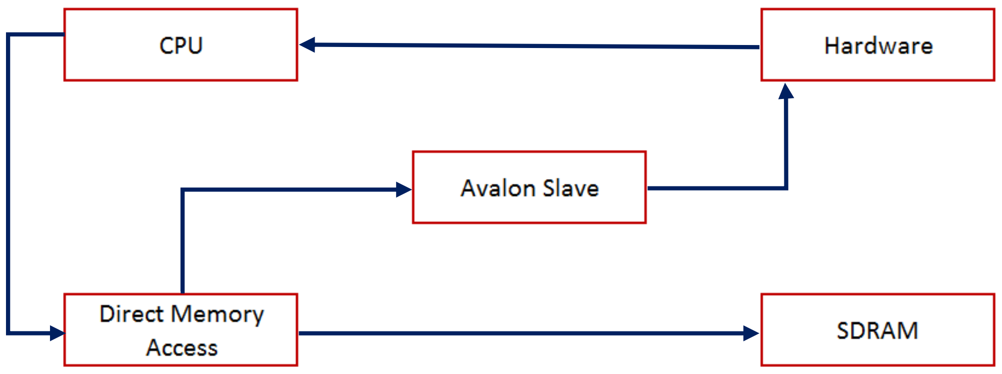


Figure 25: Diagram Illustration of Direct Memory Access Configuration

When the function is called in software, the Nios II processor will initiate the DMA module. The DMA block will fetch data from SDRAM and feed it to the hardware module every clock cycle. When the final result is computed, the hardware will send an interrupt signal to the processor and return the answer. As data is fed into the integrated module every clock cycle, the block also outputs data at the same frequency, since the integrated module is fully pipelined. Yet the floating-point adder, which is responsible for the summation, takes 7 clock cycles to evaluate. Therefore, the floating-point adder would fail to sum up consecutive data points, but will only add up every 7th data point.

In order to comply the hardware with the system, an "adder tree" is appended to the legacy system in *Section 4.3*. *Figure 26* illustrates the architecture of the adder tree and *Figure 27* displays the connections between the legacy system and the adder tree. The adder tree will compute 7 sub-summations in 7 consecutive clock cycles, and returns the sum of 7 consecutive outputs from the integrated hardware block.

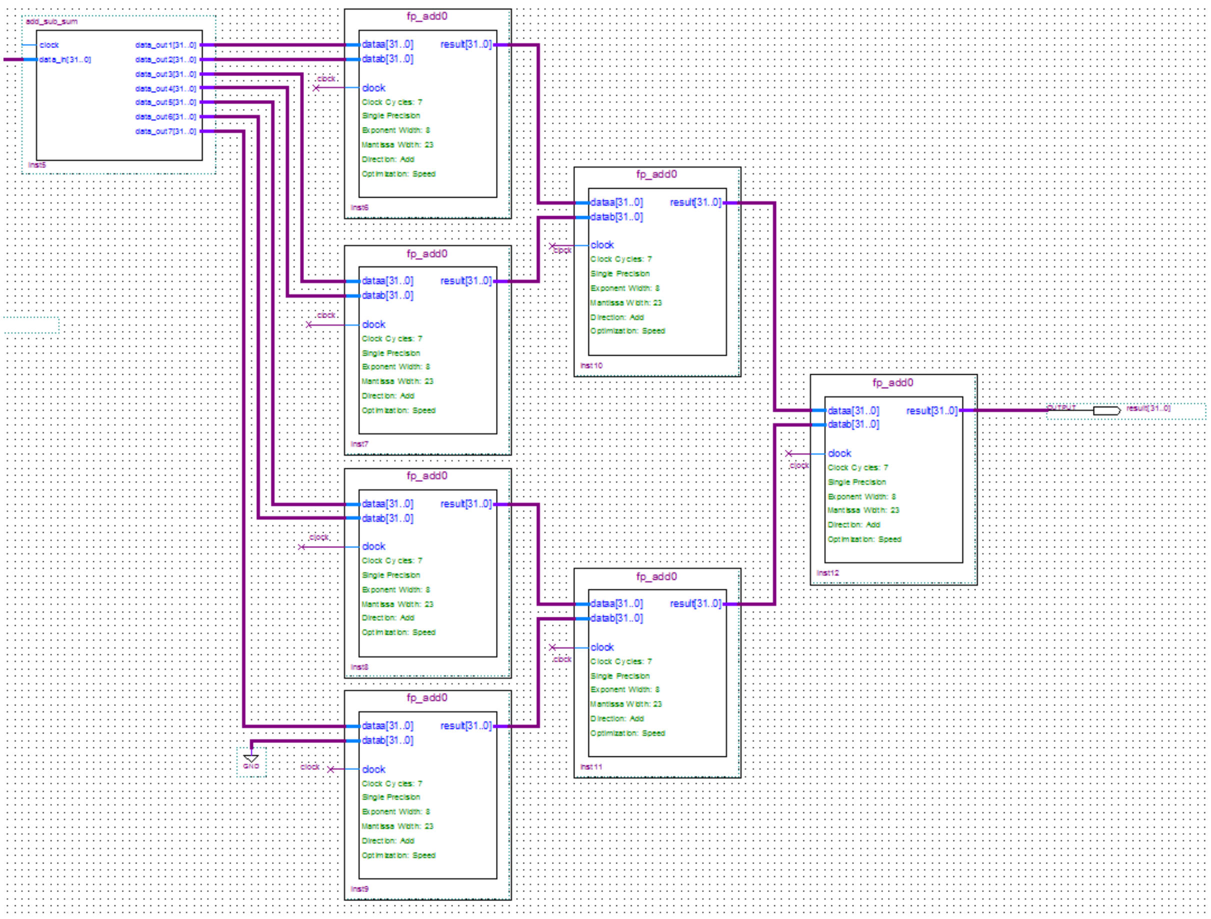


Figure 26: Architecture of "Adder Tree" for Summation

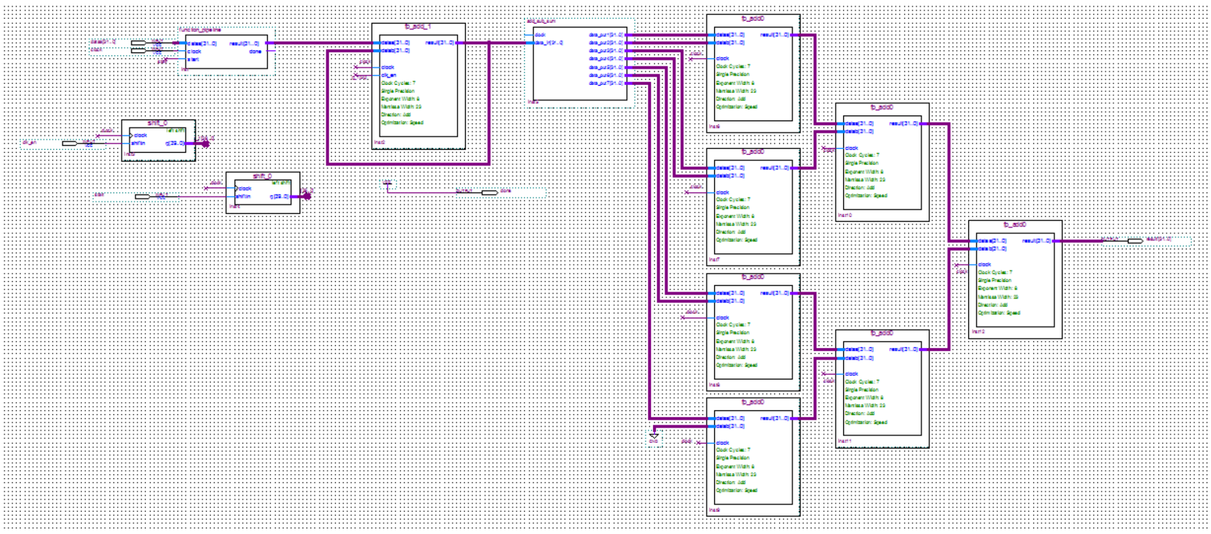


Figure 27: Schematic Diagram of Pipelined System

4.4.1 Advantages and Disadvantages

The main advantages of pipelining the system with DMA is to avoid the communication latency and speed up the computation. This proposed system should give a total execution time of *(107 + length of input vector + overheads from calling DMA)* clock cycles, and the breakdown of the calculation of execution time is explained in *Table 12*. Another advantage is that it frees up the CPU during the computation, so the processor can handle other tasks in parallel to the evaluation of the expression in the hardware. A minor unfavourable factor will be the extra consumption of resources. Meanwhile the ultimate aim of the system is to evaluate the expression as quick as possible, the usage of resources is of a lower priority and could be neglected.

However, this method requires advance technique and time to implement. Due to time constraints, pipelining the system by utilising direct memory access is not tested and verified, but is believed to further boost the time performance of the system.

	Latency (clock cycles)
Computation latency of integrated hardware block	38
Computation latency of adder tree	3 * 7 = 21
Communication latency *(from Section 2.4.1)*	48
Throughput x Length of Input Vector *N*	N
Overheads from calling DMA	Overheads
Total	107 + N + overheads

Table 12: Calculation of Execution Time Using DMA

5 Conclusion: Overall Improvement in System Performance

From *Task 4* onwards, systems are built with different approaches, from utilising in-built functions in software to designing best-fit custom functions. By creating different hardware modules to compute the expression, more resources are consumed, which is an unavoidable trade-off.

A small degree of accuracy (0.25%) is sacrificed by replacing the default *arctan* function by a CORDIC implementation. Since the priority of the design is to compute the expression as quickly as possible, and the degree of error is very insignificant, the improvement in time performance outweighs the small degree of inaccuracy.

Figure 28 shows the relationship between resources consumption and time performance, and data is detailed in the *Appendix*. It could be seen that the computation has been accelerated appreciably since *Task 4*. The final design has speeded up the computation by 1000 times, while consuming 7% more resources only.

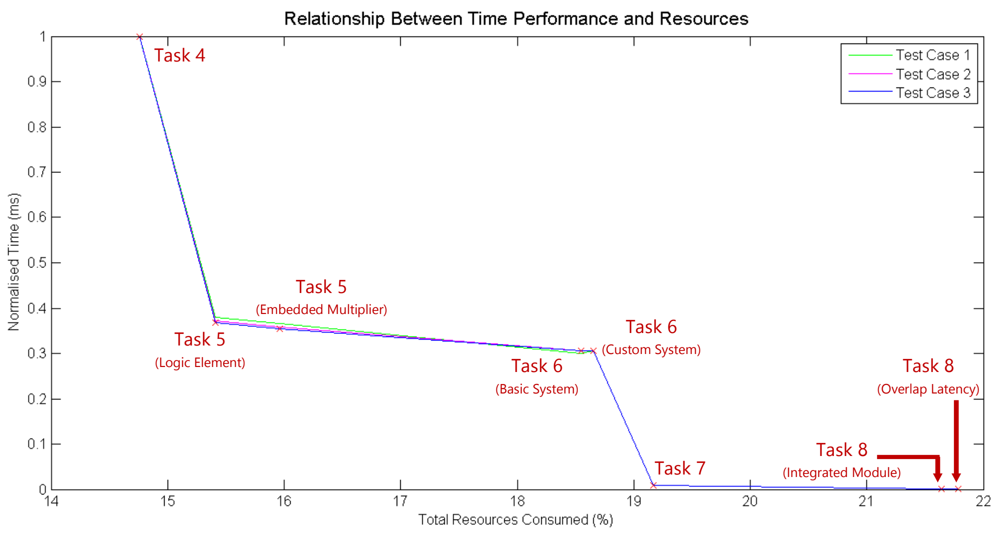


Figure 28: Relationship between Time Performance and Resources

6 Appendix

6.1 Execution Time of System in All Tasks

6.1.1 Execution Time

Test Case	Task 4	Task 5		Task 6		Task 7	Task 8	
		Logic Elements	*Embedded Multiplier*	*Basic System*	*Custom System*		*Integrated Block*	*Overlap Latency*
1	154.35	58.6	56.4	47.3	46.35	1.35	0.19974	0.15
2	7579.05	2821.6	2713.4	2313.3	2312.45	67.9	10.2952	7.36
3	759648.6	279678.8	268713.2	232055.8	232040.4	6819.95	1024.11	731.19

6.1.2 Normalised Execution Time

Normalised time is calculated by dividing the execution time by the baseline execution time, which is the execution time of *Test Case 3* in *Task 4*.

Test Case	Task 4	Task 5		Task 6		Task 7	Task 8	
		Logic Elements	*Embedded Multiplier*	*Basic System*	*Custom System*		*Integrated Block*	*Overlap Latency*
1	1.00000	0.37966	0.36540	0.30645	0.30029	0.00875	0.00129	0.00097
2	1.00000	0.37229	0.35801	0.30522	0.30511	0.00896	0.00136	0.00097
3	1.00000	0.36817	0.35373	0.30548	0.30546	0.00898	0.00135	0.00096

6.2 Resources Consumed in All Tasks

Resources	Task 4	Task 5		Task 6		Task 7	Task 8	
		Logic Elements	*Embedded Multiplier*	*Basic System*	*Custom System*		*Integrated Block*	*Overlap Latency*
Logic Elements	2987	3489	3365	4393	4474	5419	6361	6437
Pins	47	47	47	47	47	47	47	47
Memory Bits	81920	81920	81920	81965	81965	81965	81998	83079
Embedded Multipliers	0	0	4	11	11	7	14	14
PLL	1	1	1	1	1	1	1	1
% Used	**14.76%**	**15.41%**	**15.97%**	**18.55%**	**18.66%**	**19.17%**	**21.64%**	**21.78%**