Department of Electrical and Electronic Engineering
Imperial College London

EE3-05 Digital System Design

Report 3: Task 6-8

Kaiyue Sun
ks4016@ic.ac.uk
01197813

Yuwei Wang
yw6316@ic.ac.uk
01260801

Contents

Introduction

Objective

This report mainly focuses on presenting and analysing the result of exploring and designing additional hardware supports (e.g. CORDIC for computing cosine function) to accelerate the function evaluation process on FPGA as well as discussing and justifying any design choices made.

General Design Briefing

The design basically starts with the best performed case in Task 5 which is equipped with 32 KB instruction cache, no on-chip memory, 8 MB SDRAM memory and the support from embedded multiplier.

Task6: Add Hardware Floating-Point Units

So far, there is not hardware support for floating point arithmetic in NIOS II and all the floating point operations are emulated through software. Therefore, in order to accelerate the computation of given function, this section aims to design the hardware blocks which perform floating-point addition, subtraction and multiplication, but at the expense of dedicating more FPGA resources.

Floating-point Adder/Subtractor

A block implementing both operations is chosen rather than creating one for each because this block utilises less hardware resources than the two separate blocks, which is confirmed in the *Resources Usage* box shown within the *MegaWizard Plug-In Manager* interface (see **Figure-21** and **22** in **Appendix**), and can realize addition and subtraction at the same time simply by changing the *add_sub* signal.

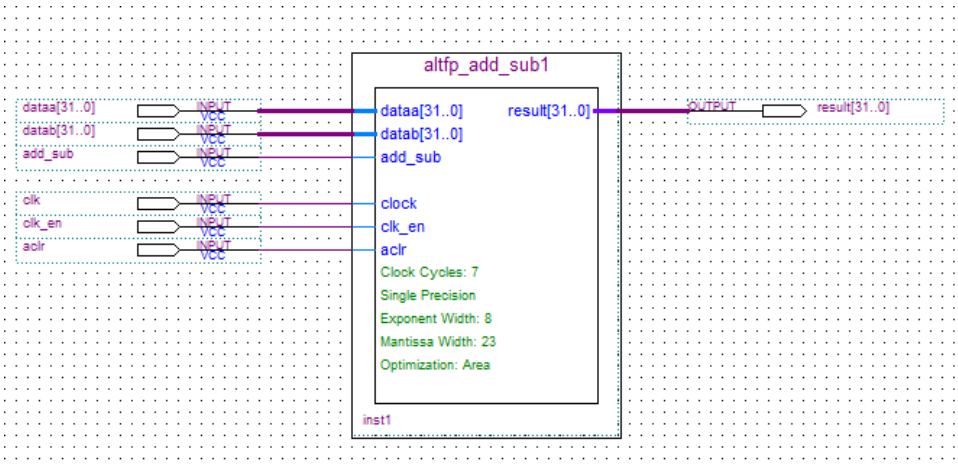


Figure-1 Block Diagram of *fp_add_sub*

Figure-1 shows the block diagram of the floating-point adder/subtractor, it is created in Quartus project from the *Megafunction* library by selecting *ALTFP_ADD_SUB*. The *fp_add_sub* block takes 2 single precision floating-point inputs *dataa* and *datab* and generates an output *result* of the same format. The input *add_sub* indicates addition or subtraction to be performed by setting it high or low, respectively.

Area optimization has been used instead of speed because the block occupies less resources in this way at the current frequency.

The output latency of 7 cycles has been chosen because it is the lowest possible option. Therefore, it is a multicycle logic block. The multicycle custom instruction can be completed within either a fixed or variable number of clock cycles. With a variable number of clock cycles, *start* and *done* ports need to be instantiated, indicating the start and completion of custom instruction. On the other hand, with a

fixed number of clock cycles which is selected in our design case, the specification of clock cycles is needed instead.

The *clk*, *clk_en* and *aclr* ports are always required and all automatically fed by NIOS II processor. *clk* and *clk_en* are synchronous clock and clock enable inputs. *aclr* is the asynchronous clear input. *clk_en* is used by the processor to stall the custom instruction during execution in case it needs.

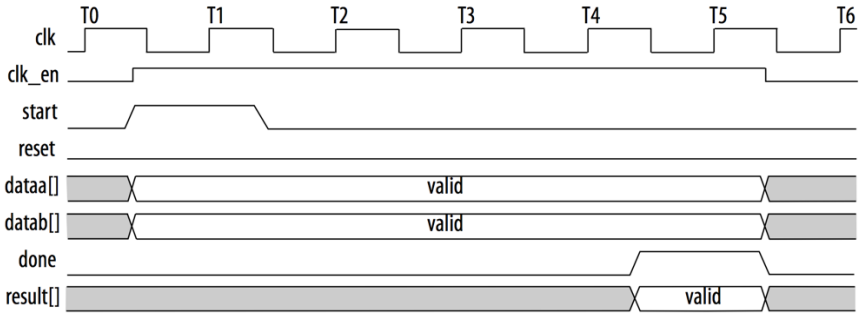


Figure-2 Timing Diagram of Multicycle Custom Instruction

Both addition (**Figure-3**) and subtraction (**Figure-4**) were tested with *dataa and datab* forced to be the floating-point representation of 3.25 (01000000101000000000000000000000) and 1.75 (00111111111000000000000000000000) as these two numbers can be represented in floating-point without error. As expected, the sum and difference were 5(0x40A00000) and 1.5(0x3FC00000), respectively, both were outputted after seven clock cycles at the eighth after the inputs.

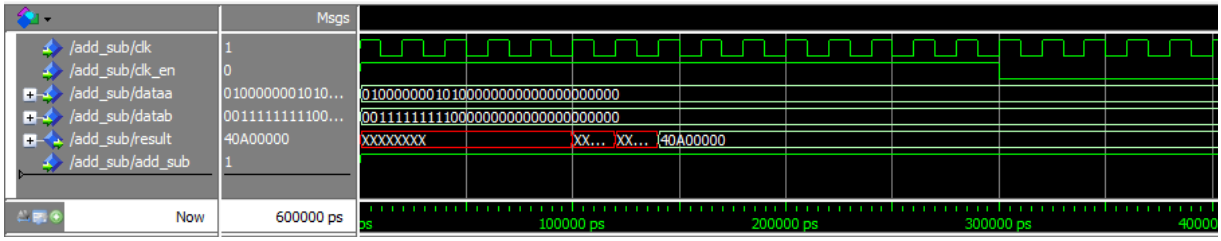


Figure-3 Simulation of Addition in *fp_add_sub*

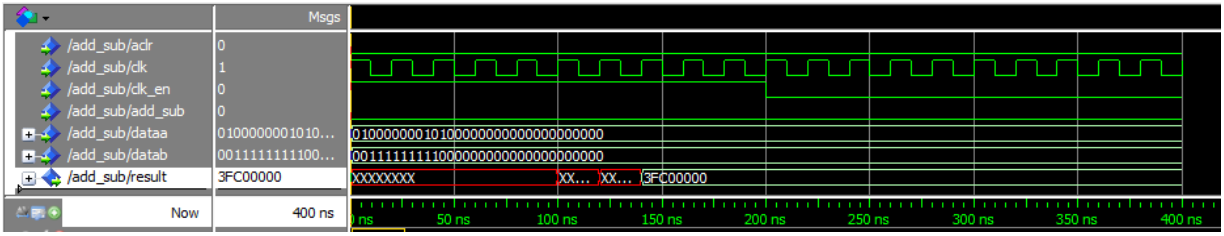


Figure-4 Simulation of Subtraction in *fp_add_sub*

As specified in user guild, for an n-cycle operation, the custom logic block must present a valid result on the n^{th} rising edge after the start of execution. Since the simulation shows that *add_sub* block outputs a final result on the eighth rising edge of the clock, clock cycle is set to eight in *add_sub* interface.

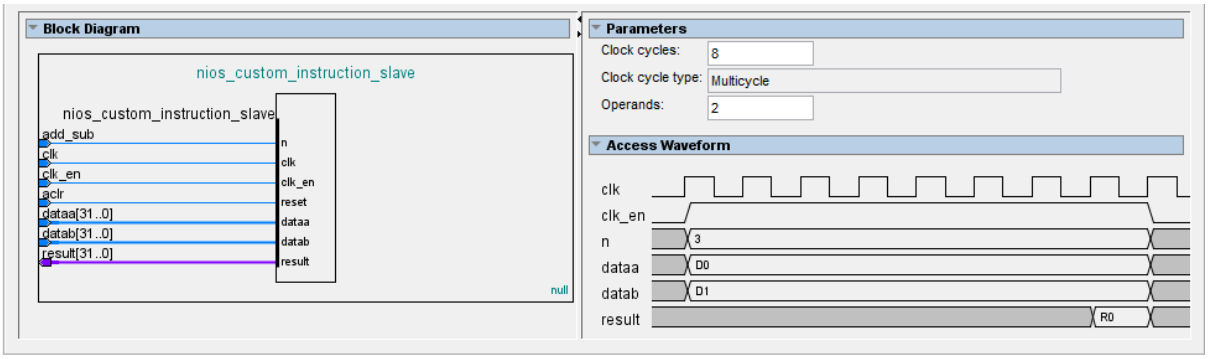


Figure-5 Interface of *fp_add_sub* in Qsys

Floating-point Multiplier

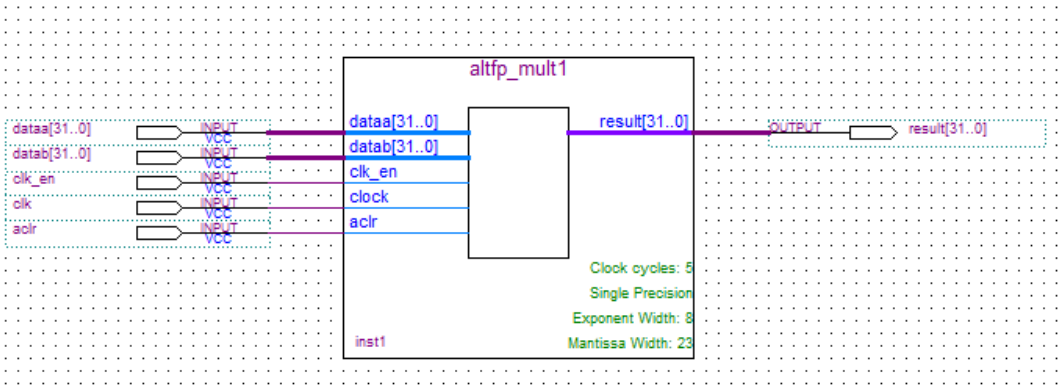


Figure-6 Block Diagram of *fp_add_sub*

The multiplier *ALTP_MULT* is found in *Megafunction* library and is multicycle as well, with a latency of five.

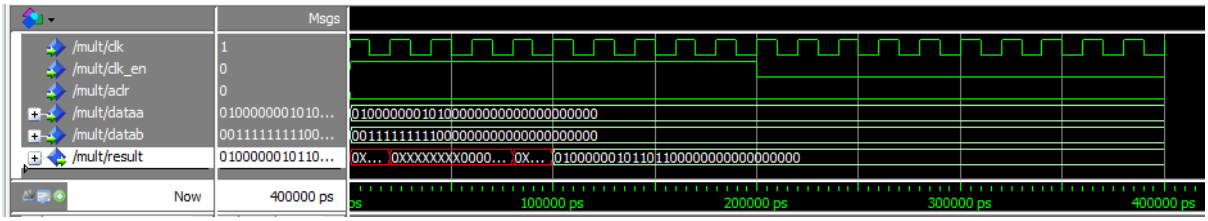


Figure-7 Simulation of *mult*

This multiplier takes also two 32-bit floating-point value as inputs (i.e. dataa and datab), generating a 32-bit result also in floating point. In the simulation, dataa is forced to $01000000010100000000000000000000_2$ (i.e. 3.25 in decimal) and datab to $00111111111000000000000000000000_2$ (i.e. 1.75), yielding the result of $01000000101101100000000000000000_2$ (i.e. 5.6875) after fix clock cycles, which is correct in value as their product. The two specific numbers are chosen for inputs because they can be represented as 32-bit floating-point without error.

For the same reason mentioned above, clock cycle is set to six in *mult* interface.

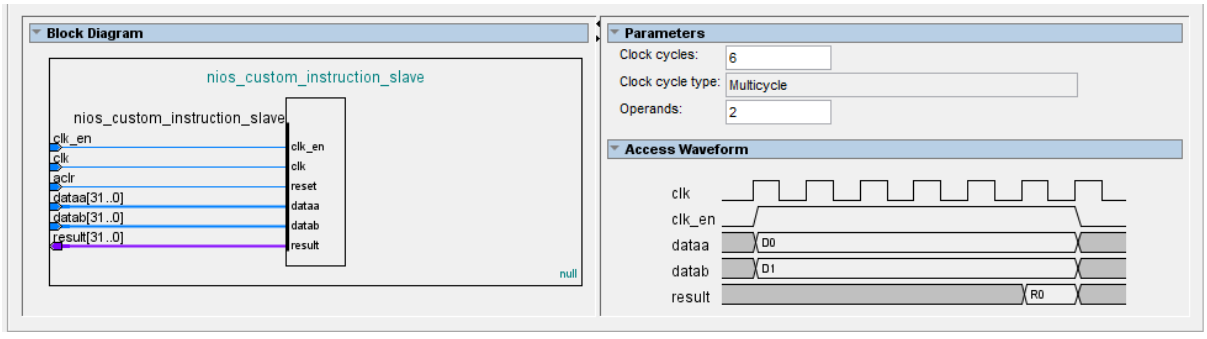


Figure-8 Interface of *mult* in Qsys

Although the evaluation of the function includes division, the divider block is not created. Instantiating this additional block means dedicating more FPGA resourcesT but without much benefit because the implementation of division can also be achieved through multiplying the corresponding reciprocal. Also, the *ALTFP_DIV* block requires one more cycle than *ALTFP_MULT*.

The connections with NIOS in Qsys is shown in Appendix (**Figure-23**).

Implementation and Performance

Resources	Task 5		Task 6	
	Usage	%	Usage	%
Logic Elements	3,259/15,408	21%	4,364/15,408	28%
Combinational functions	2,933/15,408	19%	3,972/15,408	26%
Dedicated logic registers	2,172/15,408	14%	2,807/15,408	18%
Registers	2240		2875	
Pins	47/347	14%	47/347	14%
Virtual pins	0		0	
Memory bits	291,840/516,096	57%	291,885/516,096	57%
Embedded Multiplier 9-bit elements	4/112	4%	11/112	10%
PLLs	1/4	25%	1/4	25%

Table-1 Resources Usage Comparison between Task 5 and Task 6

	Task 5	Task 6
FPGA Resources	27.09%	31.57%

Table-2 FPGA Resources Comparison between Task 5 and Task 6

Compared with **Task 5**, **Task 6** added the dedicated hardware blocks, floating-point adder/ subtraction and multiplier, which are synthesised by logic elements and embedded multipliers in FPGA. As indicated in Table x, **Task 6** utilised 4.477% more resources than **Task 5**.

When implemented in C, the name of the adder, subtraction and multiplier are modified for clearance and the *fp_add_sub* block splits into two separate instructions for easier implementation. The input and output types are redefied.

```
#define ADD_FP(A,B) __builtin_custom_fnff(0x2+(((1<<1)-1)),(A),(B))
#define SUB_FP(A,B) __builtin_custom_fnff(0x2+(0&((1<<1)-1)),(A),(B))
#define MULT_FP(A,B) __builtin_custom_fnff(0x1,(A),(B))
```

The code after *__builtin_custom* implies return type (letter before *n*) and parameter types (letters after *n*) which are encoded with *i* being *int*, *f* being *float*, *p* being *pointer* and *(empty)* being *void*. Since those two blocks are both designed to perform floating-point operation, *f* is chosen for both input and output types.

Test Case	Task 5		Task 6		Improvement
	Latency (ms)	Throughput (bytes/s)	Latency (ms)	Throughput (bytes/s)	
1	10	20800	7	29714.28571	42.8571429%
2	397	20564.23174	279	29261.64875	42.2939068%
3	50741	20584.61599	35639	29307.33186	42.3749263%

Table-3 Comparison between Best Performance in Task 5 and Task 6

With only 4.477% more resources contributing to hardware support for floating-point arithmetic, around 42% improvement is achieved in throughput compared with using software emulation, which implies that this trade-off is worthwhile.

The error with respect to MATLAB and code size are not changed compared to the ones in Task 5.

Task7: Add Dedicated Hardware Block to Compute the Inner Part of the Arithmetic Expression

CORDIC Algorithm

The CORDIC algorithm provides a method of performing vector rotations to evaluate the trigonometric functions like cosine. The vector rotations can be implemented using only shifts and adds in hardware, thus it is simple and does not require a complex hardware structure as in the case of multiplier. The idea is to iteratively perform the rotations by a series of specific angles to make the result converge to the actual value.

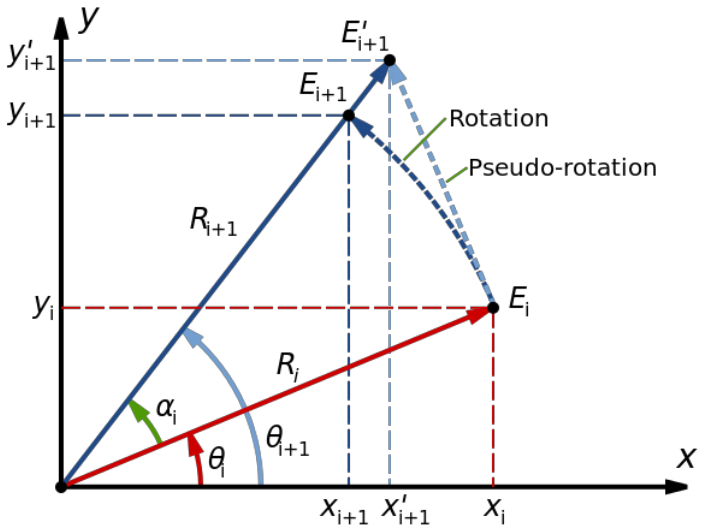


Figure-9 Visualisation of Rotation in CORDIC

Figure-9 shows a rotation and pseudo-rotation by angle α_i of a vector of length R_i and phase θ about the origin. The vector with initial Cartesian co-ordinates (x_i, y_i).

A rotation produces the following co-ordinates:

$$x_{i+1} = x_i cos\alpha_i - y_i\, sin\alpha_i$$
$$y_{i+1} = y_i cos\alpha_i + x_i\, sin\alpha_i$$
$$\theta_{i+1} = \theta_i + \alpha_i$$

Recall the identity: $cos\theta = 1/\sqrt{1 + tan^2\alpha_i}$

Therefore,

$$x_{i+1} = (x_i - y_i\, tan\alpha_i)/\sqrt{1 + tan^2\alpha_i}$$
$$y_{i+1} = (y_i + x_i\, tan\alpha_i)/\sqrt{1 + tan^2\alpha_i}$$

The strategy is to eliminate the factor of $1/\sqrt{1 + tan^2\alpha_i}$ and somehow remove the costly multiplication.

A pseudo-rotation produces a vector with the same angle as the rotated vector, but with a different length. In fact, the pseudo-rotation changes the length to:

$$R_{i+1} = \frac{R_i}{cos\alpha_i} = R_i\sqrt{1 + tan^2\alpha_i}$$

Thus, the co-ordinates following a pseudo-rotation become:

$$x'_{i+1} = x_i - y_i\, tan\alpha_i$$
$$y'_{i+1} = y_i + x_i\, tan\alpha_i$$
$$\theta_{i+1} = \theta_i + \alpha_i$$

The vector will grow by a factor of *K* after a sequence of *n* pseudo-rotations:

$$K = \prod_{i=0}^{n-1} \sqrt{1 + tan^2\alpha_i}$$

$$x_n = K(x_i\, cos \sum_{i=0}^{n-1} \alpha_i - y_i\, sin \sum_{i=0}^{n-1} \alpha_i)$$

$$y_n = K(y_i \cos \sum_{i=0}^{n-1} \alpha_i + x_i \sin \sum_{i=0}^{n-1} \alpha_i)$$

$$\theta_n = \theta_i + \sum_{i=0}^{n-1} \alpha_i$$

If the angles are always the same set, then *K* is fixed and can be accounted for later. The angles are chosen according to the following two criteria.

First of all, any angle can be constructed from the sum of all them, with appropriate signs. Secondly, all $tan\ \alpha_i$ are designed to be a power of 2, so that the multiplication can be performed by a simple logical shift of a binary number.

This means that the angles $\alpha_i = tan^{-1}2^{-i}$ will satisfy both criteria as shown in **Table-4**.

i	$\boldsymbol{\alpha_i = tan^{-1}(2^{-1})}$	
	Degrees	Radians
0	45.00	0.7854
1	26.57	0.4636
2	14.04	0.2450
3	7.13	0.1244
4	3.58	0.0624
5	1.79	0.0312
6	0.90	0.0160
7	0.45	0.0080
8	0.22	0.0040
9	0.11	0.0020

Table-4 Predefined Angle Set

As each of the angle in the list is more than half of the previous one, therefore any angle θ in the domain of $-99.7° \leq \theta \leq 99.7°$ can be reconstructed.

Therefore,

$$x_{i+1} = x_i - d_i y_i 2^{-i}$$

$$y_{i+1} = y_i + d_i x_i 2^{-i}$$

$$\theta_{i+1} = \theta_i + d_i \theta_i$$

$$X_m = K(xcos(z) - ysin(z))$$

$$Y_m = K(ycos(z) + xsin(z))$$

$$\theta_m = \theta_0 + d_i \alpha_i$$

By setting the initial values as the following: $x = \frac{1}{K}, y = 0, z = 0$,

$$X_m = cos(z)$$

$$Y_m = sin(z)$$

It can be noticed that $\theta_0 = \theta_m - d_i \alpha_i$, by initialising θ to be θ_m, which is the angle to be rotated, so the final θ will be θ_0, which is 0. Thus, d_i, the direction (or sign) of the rotation will be chosen at each step to converge the angle to 0, implying $d_i = 1$ if $\theta_i < 0$; $d_i = 0$ if $\theta_i > 0$.

Selection of CORDIC Iteration

In order to meet the requirement that given the input following a uniform distribution in the range of [-1, 1], the Mean Square Error (MSE) of the output of CORDIC should be less or equal to $1x10^{-10}$ with the confidence level of 95%, so far a 31-iteration CORDIC algorithm is implemented. However, it yields a far more accurate result than required, which happens to provide an opportunity to reduce the number of iterations and improve the latency performance.

The Monte Carlo simulation done in MATLAB (see **Code Snippet-6** and **7** in **Appendix**) is used to determine the MSE. The code basically loops through every possible number of iterations (e.g. *i* from 1 to 31), computes the MSEs and determines whether *i*-iteration algorithm gives accurate enough result. If it meets the requirement, the loop stops early and *i*-iteration is chosen for the final implementation of CORDIC. Otherwise, it will continue increasing the number of iterations by 1 at a time until 31.

As to judging whether *i*-iteration algorithm generates result with enough precision or not, firstly a random sample of 10 numbers from a uniform distribution in the range of [-1, 1] is acquired. Then, they are passed to CORDIC algorithm to calculate their approximated cos values and the MSE with respect to the double precision cos values computed in MATLAB. After 100 iterations producing 100 MSEs, the number of MSE larger than $1x10^{-10}$ is counted. If the count is less than or equal to 5 (i.e. less than 5% of all situations exceed the threshold for MSE), the *i*-iteration, which is the lowest one meeting the requirement, will be chosen to implement the CORDIC algorithm.

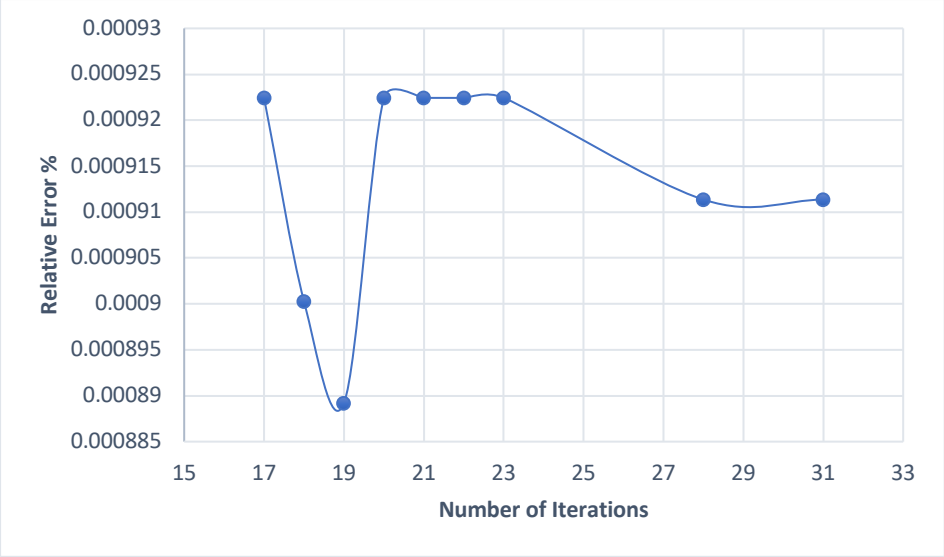
For this design, the program stopped after 17 iterations of CORDIC which yielded a count of zero, meaning no MSE violation.

When CORDIC were actually implemented in hardware with 17 iterations, it indeed required less execution time but undesirably produced less precise result compared with the 31-iteration one, which was expected since it went through fewer iterations. For the purpose of this project, 17 iterations will be adopted since the new precision required for the system is the one that dictates CORDIC precision and 17 iterations are enough to meet the requirement.

The number of iterations can be gradually increased to search for better accuracy. All the exploration done are based on Test Case 3.

Number of Iterations	Latency	Result	Relative error
17	1971	4621531648	0.000922432
18	1980	4621530624	0.000900275
19	1980	4621530112	0.000889196
20	1987	4621531648	0.000922432
21	1993	4621531648	0.000922432
22	1993	4621531648	0.000922432
23	1994	4621531648	0.000922432
28	2026	4621531136	0.000911354
31	2043	4621531136	0.000911354

Table-5 Performance for Different Number of Iterations



s

Figure-10 Relative Error (%) versus Number of Iterations

The increment of number of iterations causes more latency. However, larger number of iterations does not necessarily mean more precise results, since fewer iterations sometimes can directly rotate the unit-length vector to the angle *z*, thus computing the exact result for *z*. Any more iterations, implying rotating the vector back and forth, unavoidably result in some errors. For example, in the case which *z* is 30 degrees, at 9th iteration, the result is the most accurate since it just rotates precisely 30 degrees. After 9th iteration, such as at 10th iteration, it gives less accurate answers than the one produced at 9th iteration.

i	$z^{(i)}$	–	$d_i\, e^{(i)}$	=	$z^{(i+1)}$
					+30.0
0	+30.0	–	45.0	=	–15.0
1	–15.0	+	26.6	=	+11.6
2	+11.6	–	14.0	=	–2.4
3	–2.4	+	7.1	=	+4.7
4	+4.7	–	3.6	=	+1.1
5	+1.1	–	1.8	=	–0.7
6	–0.7	+	0.9	=	+0.2
7	+0.2	–	0.4	=	–0.2
8	–0.2	+	0.2	=	+0.0 ⇐
9	+0.0	–	0.1	=	–0.1

Table-6 Iterations Performed for 30 Degrees Case

From the observation done so far as recorded in **Table-5**, 19-iteration algorithm seems to be optimal since it has not only better latency performance but also result with higher precision than the 31-iteration one. As shown, when the number of iteration reach to 28, the result generated is the same as the one generated by 31 iterations. Therefore, keeping increasing the iteration above 28 will only result in more execution time but without any benefit for the accuracy.

Implementation of CORDIC in Verilog

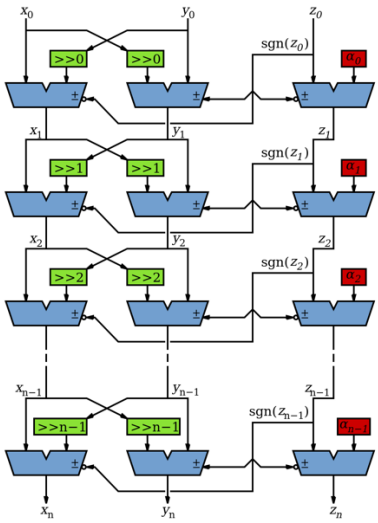


Figure-11 Bit-parallel Unrolled CORDIC

In all the three test cases, the domain of the input angle of the cos function is [-1,1] which is within the domain of convergence. In order to implement CORDIC in Verilog, which does fixed-point operations, the simplest way is to use fixed-point representation of the angle in radian to iterate z_i and determine d_i. Two bits are used to represent the integer part and the thirty bits left for the fractional part. Therefore, the input of CORDIC needs to be connected to a float-to-fix converter, the output should be connected to a fix-to-float converter.

Figure-11 visualises the bit-parallel unrolled CORDIC architecture, which is implemented in Verilog as shown in **Code Snippet-1**. The unrolled loop is constructed by using *generate* to instantiate the iteration code multiple times, controlled by the variable STG. The calculation is performed at each positive edge of the clock cycle. The input data could be passed to the architecture once per cycle.

```
always @(posedge clock)
   begin
      X[0] <= 32'b00100111011011101001110110110101O;
       Y[0] <= 1'd0;
       Z[0] <= angle;
end
genvar i;
generate
   for (i=0; i < (STG-1); i=i+1)
   begin: XYZ
      wire                    Z_sign;
      wire signed  [XY_SZ:0] X_shr, Y_shr;

      assign X_shr = X[i] >>> i; // signed shift right
      assign Y_shr = Y[i] >>> i;

      assign Z_sign = Z[i][31]; // Z_sign = 1 if Z[i] < 0

      always @(posedge clock)
      begin
         // add/subtract shifted data
         X[i+1] <= Z_sign ? X[i] + Y_shr   : X[i] - Y_shr;
         Y[i+1] <= Z_sign ? Y[i] - X_shr   : Y[i] + X_shr;
         Z[i+1] <= Z_sign ? Z[i] + arctan[i] : Z[i] -
arctan[i];
      end
```

```
27      end
28      endgenerate
28
        assign Xout = X[STG-1];
```

Code Snippet-1 CORDIC Algorithm

After a delay of the number of STG cycles, the results would be outputted at the same rate. This was verified in the simulation shown in **Figure-12**. If setting STG into 19, three fixed-point inputs of 0.5 (0x2000000), 0.25 (0x1000000), 0.75(0x3000000) were added at the beginning of three consecutive clock cycles. Three consecutive results of cos(inputs) were produced after 19 cycles (e.g. cos(0.75)= 0.87758256189 (0x2ED40223)), confirming the CORDIC is working.

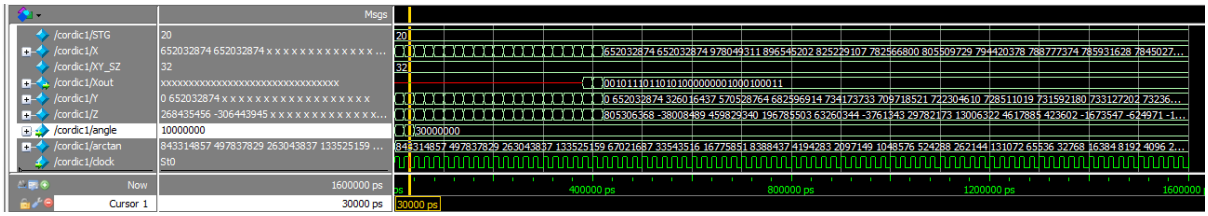


Figure-12 Simulation of *cordic*

Converters between Fix-point and Floating-point

The *ALTFP_CONVERTER* in *Megafunction* is used to do the float-to-fix and fix-to-float conversion before and after the CORDIC block. **Figure-13** demonstrates the simulation of float-to-fix converter, with the floating-point input of -0.1 ($10111101110011001100110011001101_2$), the fixed-point output of -0.1 ($11111001100110011001100110011000_2$) is produced at the 6th cycle. **Figure-14** simulates the fix-to-float converter, taking the input of 0.25 gives the output of its floating-point representation.

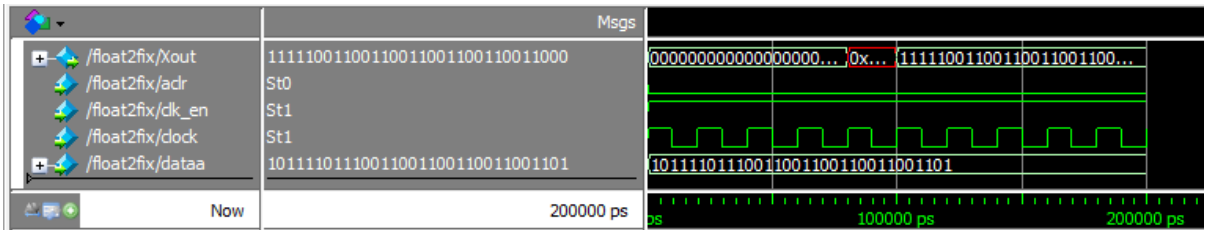


Figure-13 Simulation of *float2fix*

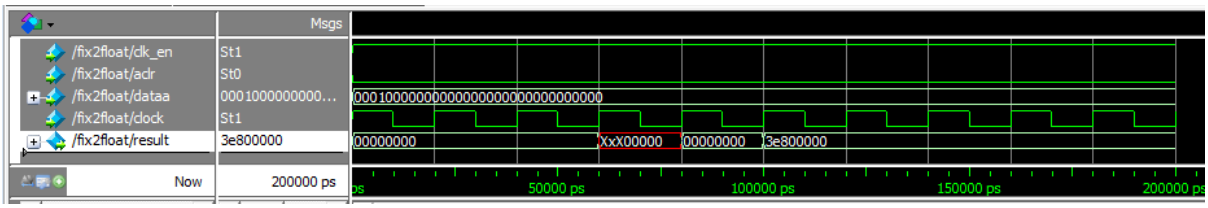


Figure-14 Simulation of *fix2float*

Targeted Improvement on the Inner Part of the Cosine Function

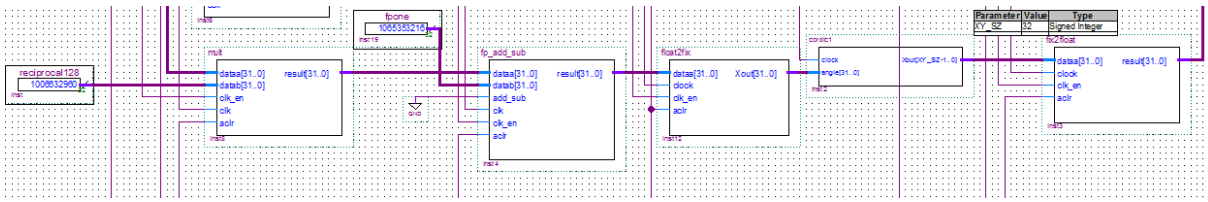


Figure-15 Block Diagram of Inner Part of the Cosine Function

Before executing CORDIC, $\frac{x}{128} - 1$ should be performed. A specific Verilog code was designed to implement this step as shown in code snippet x. To convert the floating-point number *dataa* into the specific format (00.00…00) of the fixed-point number, we need to find out the right position at which each digit in mantissa (plus the not included integer component 1) of the float should be put, in other words, the correct number of times that shift should be done to these digits to fit them in the new format. Then, at the same time, $\frac{x}{128}$ could be performed simply by right shifting 7 times more. Thus, the 2 shifts for different purposes can be combined and the number of times is equal to (the exponent of the float-the bias(127)-7). Because x is between 0 and 255, 2 bits are sufficient to represent the integer part of $\frac{x}{128}$ in the fixed-point representation, and the shift must be to the right or no shift is needed at all. As *x* is positive, the first digit of the fixed number is 0, the rest of the digits is determined by the significand from the float. At last, fixed-point operation of -1 is done conveniently. All the tasks mentioned above can be compressed within 1 clock cycle.

Code Snippet-2 Inner Part of Cosine Function

```
wire  [7:0] shift;
wire  [30:0] frac;
reg [31:0] result;

assign shift=8'b10000110-dataa[30:23];//exp-127-7
assign  frac= {1'b1,dataa[22:0],7'b0} >>> shift;//shift to right

always @(posedge clock)
begin
  if(dataa==32'b0)
    result<=32'b0;
  else
    result<={1'b0,frac}-32'b01000000000000000000000000000000;
end
```

After CORDIC outputs the fixed-point cos result, we need to convert it to a float. Again, a dedicated snippet of code was made of use. The range of cos[-1,1) is [0.54030…,1]. If it's 1, it'll output the float 1 directly. Otherwise, the exponent of its floating-point representation must be 126 (2^{-1}), therefore, the mantissa will be 2 times the fractional part of the fix-point number, which can be understood as shifting once to the left. As the integer 1 is ignored in a float, the mantissa takes 23 bits from the 28th bit only.

```
always @(posedge clock)
begin
  if(Xout==32'b01000000000000000000000000000000)
    resultout<=32'b00111111100000000000000000000000;
  else
    resultout<={9'b001111110,Xout[28:6]};
end
```

Code Snippet-3 Fix-point to Floating-point Converter

CORDIC without Pipeline Structure

Due to the fact the pipeline structure leads to higher throughput, our CORDIC block was chosen to be pipelined. However, there is no pipeline choice in custom instruction slave interface which means that there is no way to notify the processor that the CORDIC is actually pipelined. For this reason, the processor may still consider the CORDIC to be *Multicycle* and feed the input every fixed number of clock cycles even though the CORDIC is able to output data every clock cycle after a delay at the beginning.

Pipeline structure was abandoned and efforts were made to unroll the whole structure so that the CORDIC only need less than one clock period instead of 19 to complete the computation and output the result, thus reducing the latency by considerable amount.

```
assign X[0] = 32'b00100110110111010011101101101010;
assign Y[0] = 1'd0;
assign Z[0] = result;

genvar i;
generate
for (i=0; i < (STG-1); i=i+1)
begin: XYZ
   wire signed  [XY_SZ:0] X_shr, Y_shr;

   assign X_shr = X[i] >>> i; // signed shift right
   assign Y_shr = Y[i] >>> i;

   assign X[i+1] = Z[i][31] ? X[i] + Y_shr          : X[i] - Y_shr;
   assign Y[i+1] = Z[i][31] ? Y[i] - X_shr          : Y[i] + X_shr;
   assign Z[i+1] = Z[i][31] ? Z[i] + arctan[i] : Z[i] - arctan[i];

end
endgenerate
```

Code Snippet-4 CORDIC without Pipeline Structure

Merging Converters and CORDIC

Originally, computing cosine function need three clock cycles, one for calculating the value inside cosine, one for CORDIC and one for fix-point to floating-point conversion. However, if the algorithm of the other two custom instructions can merge into CORDIC one, creating only one block, the clock cycles needed will be reduced to one.

Combining Instruction Blocks

So far, the implementation of hardware support is done separately for each functionality and follows the expression below to compute function *f(x)*.

$$f(x) = \sum x(0.5 + x\,\mathrm{cosf}\left(\frac{x}{128} - 1\right))$$

The whole computation takes 29 clock cycles. Six and eight clock cycles are for multiplication and addition/subtraction respectively. The actual execution of function *cosf* requires one clock as stated above.

$\mathrm{cosf}\left(\frac{x}{128} - 1\right)$	$x\,\mathrm{cosf}\left(\frac{x}{128} - 1\right)$	$0.5 + x\,\mathrm{cosf}\left(\frac{x}{128} - 1\right)$	$x\left(0.5 + x\,\mathrm{cosf}\left(\frac{x}{128} - 1\right)\right)$	$x\left(0.5 + x\,\mathrm{cosf}\left(\frac{x}{128} - 1\right)\right) + f(x)$
1	6	8	6	8

Figure-16 Workflow of the Whole System

However, if the expression is decomposed into smaller partitions, several computations can be performed in parallel which will certainly reduce the clock cycles required. For example, in our design case, x^2, $0.5x$ and $\mathrm{cosf}\left(\frac{x}{128} - 1\right)$ can proceed at the same time.

$$f(x) = \sum 0.5x + x^2 \text{cosf}\left(\frac{x}{128} - 1\right)$$

Also, when all custom instructions integrate to one, there is no need to give an extra clock cycle in between for next block to fetch the output. So, together with the parallel structure, the total time required is recued to 24 clock cycles.

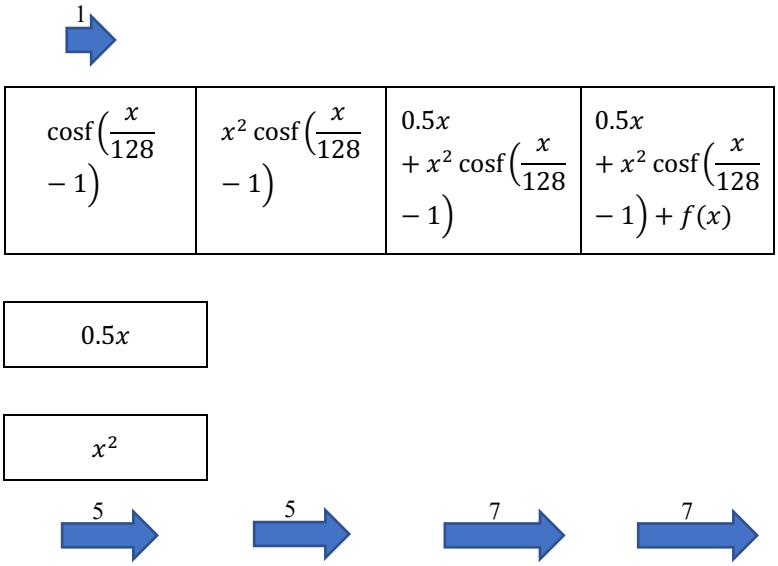


Figure-17 Workflow of the Whole System after Parallel

Performance

Test Case	Task 6	Relative Error (%)	Task 7 (17)	Relative Error (%)	MATLAB Result
1	920413.5	-1.37601E-05	920414	4.05633E-05	920413.626649442
2	36123104	5.10699E-05	36123136	1.39656E-04	36123085.5519791
3	421531136	9.11354E-04	4621531648	9.2243E-04	4621489017.88862

Table-7 Relative Error Comparison between **Task 6** and **Task 7**

When 31 iterations for CORDIC were chosen, which is the maximum number of iterations that can be implemented at current stage in **Task 7**, the results were the same as in **Task 6**. However, when the number of iterations was set to 17, the relative errors increased for all test cases, which is expected since less iteration implies lower resolution. The larger the size of vector *x*, the greater the errors (both relative and absolute) for both **Task 6** and **Task 7**. It seems like that the errors due to the insufficient iterations are accumulated.

Test Case	Task 7	Relative Error (%)	Task 7 (Self-designed Converter)	Relative Error (%)	MATLAB Result
1	920414	4.05633E-05	920414	4.05633E-05	920413.626649442
2	36123136	1.39656E-04	36123136	1.39656E-04	36123085.5519791
3	4621531648	9.2243E-04	4621490176	2.50593E-05	4621489017.88862

Table-8 Relative Error Comparison between Modifications done in **Task 7**

By replacing *ALTFP_CONVERTER* with the self-designed fix-to-float converter at the end of the CORDIC system, the result remains unchanged in Test Case 1 and 2 but becomes more accurate in 3 than all the previous tasks. This finding implies that the software emulation in Task 5 and the converter used in Task 6 induced errors, while this self-designed converter, which is particular for the input value in the range of [-1,1) for the cos function, helps to reduce the errors.

This improvement is not a coincidence for only this test case because the results remains the same for the first two test cases, which means the converter in Task 6 worked as well as the new converter in Task 7 for the two cases and the errors in the two cases are due to fewer iterations.

Test Case	Task 6		Task 7		Improvement
	Latency (ms)	Throughput (bytes/s)	Latency (ms)	Throughput (bytes/s)	
1	7	29714.28571	0 (0.201)	N/A (1036285.714)	N/A (3387.50%)
2	279	29261.64875	8	1020500	3387.50%
3	35639	29307.33186	990	1055034.343	3499.90%

Table-9 Performance Comparison between **Task 6** and **Task 7**

Compared to Task 6, Task 7 achieved an improvement in throughput of almost 3500%.

The latency of **Test Case 1** is zero but it is more likely to be the case that the system only shows latency with resolution of 1ms and this test case has latency lower than 1 ms which cannot be represented here. Assuming **Test Case 1** has the same percentage improvement of throughput as **Test Case 2**, the corresponding throughput calculated is 1036285.714 and therefore the latency is supposed to be 0.201 ms.

Resources	Task 6		Task 7	
	Usage	%	Usage	%
Logic Elements	4,364/15,408	28%	8,276/15,408	28%
Combinational functions	3,972/15,408	26%	7,583/15,408	26%
Dedicated logic registers	2,807/15,408	18%	3,521/15,408	18%
Registers	2875		3589	
Pins	47/347	14%	47/347	14%
Virtual pins	0		0	
Memory bits	291,885/516,096	57%	291,957/516,096	57%
Embedded Multiplier 9-bit elements	11/112	10%	25/112	10%
PLLs	1/4	25%	1/4	25%

Table-10 Resources Usage Comparison between Task 6 and Task 7

	Task 6	Task 7
FPGA Resources	31.57%	44.20%

Table-11 FPGA Resources Comparison between Task 6 and Task 7

The code size decrease from 85KB to 75KB due to the fact that individual custom instructions are combined in to one. Instead of calling each blocks separately and therefore writing long code to compute the function value, only one instruction is given, which reduces the code size to some extent.

Task 8: Add Dedicated Hardware Block to Compute the Arithmetic Expression

Optimiser

Another approach used to reduce the latency performance is to utilise the compiler's optimisation level setting.

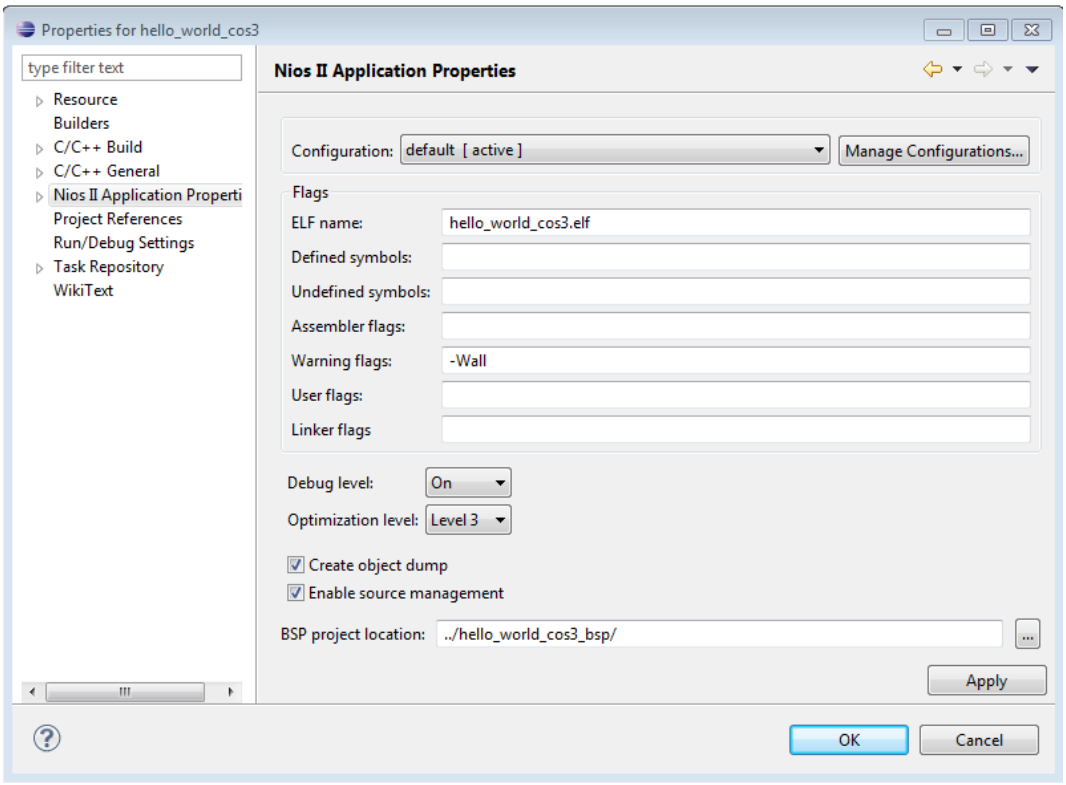


Figure-18 Optimisation Level Setting

There are various choices listed below for optimisation level. With optimisation turned on and increasing level up to level 3, the compiler performs software optimisation such as loop optimisation, loop unrolling, software pipelining and so on.

The results and FPGA resources used were unchanged since there was only optimisation applied to software part and no change made on hardware side.

Test Case	Optimisation Level				
	Level 3	Level 2	Level 1	On	Size
1	0	0	0	1	1
2	2	2	2	2	2
3	243	243	246	250	256

Table-12 Latency Performance for Different Optimisation Levels

As a result, the execution time was largely reduced. With optimisation level 3, the system gave it best performance.

Reducing Cache Size

Our objective is to design a system giving the minimal latency but at the same time dedicating as few resources as possible. Therefore, the instruction cache size was shrunk in order to reduce the resources used.

Cache Size		32 KB	16 KB	8 KB	4 KB	2 KB	1 KB	512 Bytes
Total Logic Element		8,276	8,256	8,260	8,286	8,302	8,231	8,250
Total Memory Bits		291,957	152,181	82,037	46,837	29,173	20,309	15,861
Embedded Multiplier 9-bit Elements		25	25	25	25	25	25	25
FPGA Resources		44.20%	35.13%	30.61%	28.39%	27.29%	26.56%	26.31%
Latency (ms)	Test Case 1	0	0	0	0	0	0	0
	Test Case 2	2	2	2	2	2	2	2
	Test Case 3	243	243	243	243	243	244	247

Table-13 Resources and Latency Summary for Different Cache Sizes

As can be seen, the latency does not change as cache size decreases until it reaches 1KB since combining all custom instructions into one largely reduce the code size. Thus, the size of the instruction cache needed decreases. The system is able to maintain its latency performance with smaller resources usage (2 KB).

Conclusion

Improvement Step	Test Case 1	Test Case 2	Test Case 3
0: Task 5	10	397	50741
1: add/sub/mult	7	279	35639
2: CORDIC31	0	16	2043
3: CORDIC19	0	16	1980
4: parallel(19)	0	9	1202
5: parallel(17)	0	9	1187
6: (x/128)-1	0	8	1086
7: fix2float	0	9	1058
8: w/o pipeline	0	8	990
9: optimise	0	2	243

Table-14 Summary of Changes Made

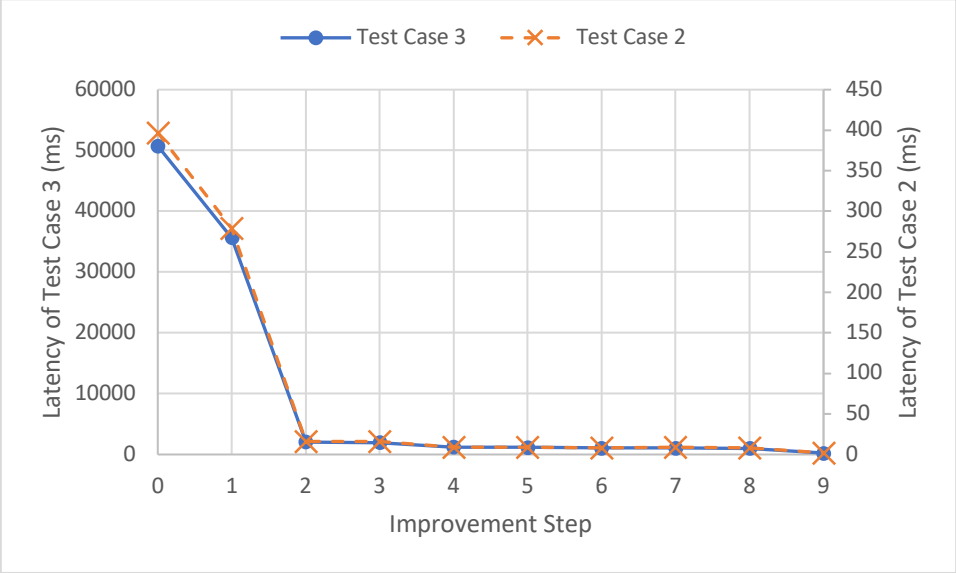


Figure-19 Improvements Made Across Test Case 2 and 3

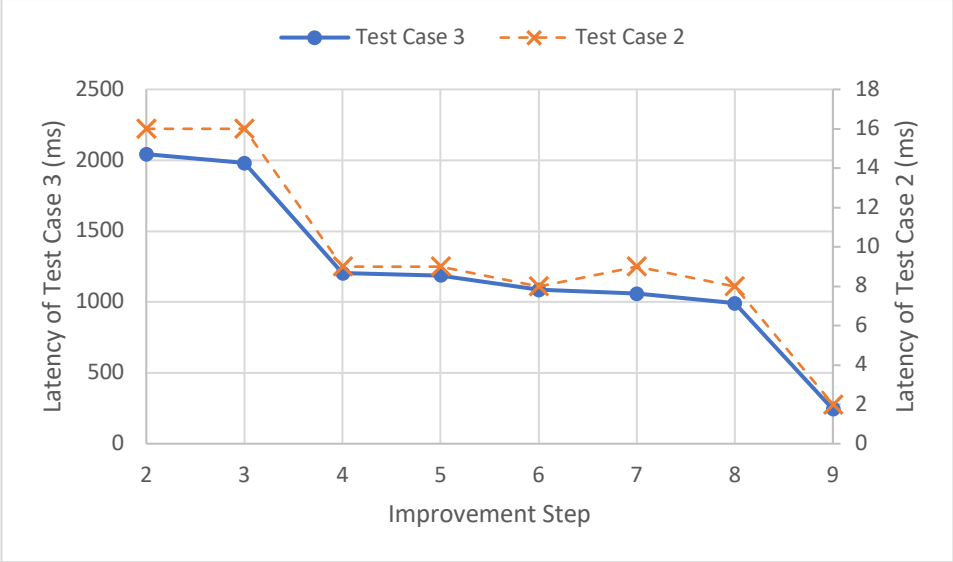


Figure-20 Improvements Made from Step 2 to 9 Across Test Case 2 and 3

In general, 10 steps of improvement have been made to accelerate the function evaluation. The latencies for each test case at each stage has been recorded in **Table-14**. The decreasing trend of the latencies for Test Case 2&3 are illustrated in **Figure-19**. **Figure-20** zooms in the last eight steps.

From **Figure-19**, it can be seen that adding dedicated hardware block to compute the cos function using CORDIC algorithm most significantly speed up the computation, resulting in an improvement of 1644% in the latency.

In all the remaining steps, from **Figure-20**, it is noticeable that step 4, which implements the parallel computation, also reduces the latencies by quite a large amount relatively. Numerically, the latencies declines 77.8% and 64.7% for Test Case 2&3 respectively. Then, step 5 to 8 shortens the computation time gradually.

Ultimately, complier optimiser raises 300% in latency performance by optimising the software such like loop operation.

In summary, although some benefits are not significant, large amount of effort has been put to design and verify the programs. Through the process, we gain a more comprehensive understanding about the floating-point arithmetic and digital system design.

Future Work

Self-designed Multiplier

As shown in **Figure-17**, the efficiency of parallel structure is limited by the multiplier. Although the whole cosine function can be computed within one clock cycle, it has to wait for x^2, which need five clock cycles, in order to perform next multiplication. Therefore, some modifications and improvements are possible for the multiplier.

We would have liked to implement a dedicated floating-point multiplier in Verilog for this task, but the test failed for unknown reasons. The results of each test cases This multiplier aims at finding out the product whose exponent is only in the range of [1,254]. Any exponent less than 1 gives the product of 0. The code was shown in Snippet x. The form below describes the logic behind the multiplier. Debugging this version and modifying it to adapt general purposes will be carried out in the future.

Carry=1, if MSB of {1,mantissa(dataa)}*{1,mantissa(datab)}==1; Otherwise, carry=0.

mantissa(prod)=remove bits before the most significant '1' (inclusive) of the 48-bit {1,mantissa(dataa)}*{1,mantissa(datab)}, then truncate 23 bits or extend zeros to fill 23 bits.
exp(prod)=exp(dataa)+exp(datab)+carry-127.

Table-14 Basic Logic of Self-designed Multiplier

DMA

DMA is a method of transferring data from the memory on the computer to the attached device and vice versa by some computer bus architectures. In our case, DMA is used to allow the generated vector x to be sent directly to the dedicated IP block for the evaluation of the targeted function without passing it through the CPU, and then return the computed results to the SDRAM. Therefore, DMA is able to speed up the overall function evaluation because the CPU is freed from involvement with the data transfer, only needs to setup the transmission. Moreover, most DMA implementation involves the use of a DMA controller which is shown in **Figure-20**, the DMA controller transfers data at the maximum pace allowed by the source or destination.

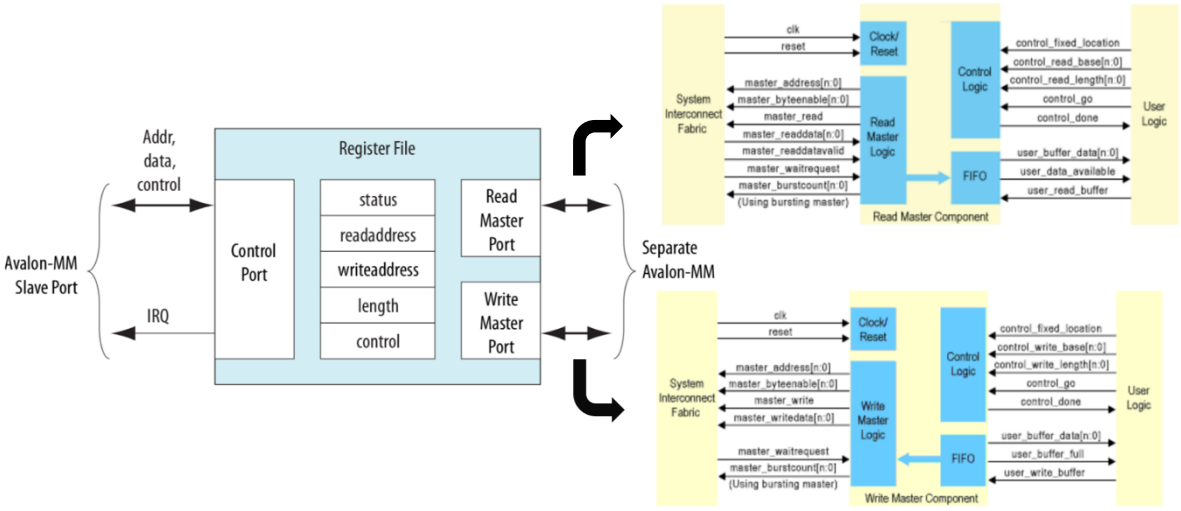


Figure-20 DMA controller block & read/write master component

A typical DMA process basically works as follow.

Firstly, the control port of the DMA controller is configured by the processor through the Avalon Memory-Mapped Interface.

In the second place, the DMA controller is enabled by the processor and then transmission begins. It reads data from source address through the read master port and writes to the destination address with the write master port. A shallow FIFO buffer with default depth of 2 is in-between the read and write ports, which makes the write action dependent on the data-available status of the FIFO instead of the status of the read master port.

Moreover, the DMA transaction supports a fixed-length transaction and a variable-length transaction, indicating that the transmission can terminate either after a specific number of bytes are transferred or by an end-of-packet signal. The DMA controller can provide an interrupt request (IRQ) if required.

The processor can also detect whether a transaction ends by visiting the DMA controller's status register.

The Read/write master component shown in **Figure-20** exposes control and data interfaces to connect with custom logic. Use the control interface to specify information such as memory addresses, transfer lengths, and handshaking signals. The data interface provides data to or from the master

internal buffer using a simple send and acknowledge protocol. The control signals must be sent to the master component before any data is transmitted. The relevant control signals and the usage can be referred to resource[8].

Without DMA, the processor is fully occupied during the read or write operation, thus unable to perform other work. With DMA, the workload is mainly carried out by the DMA controller, therefore largely reducing the work load of the processor and making it available to execute other tasks.

Appendix

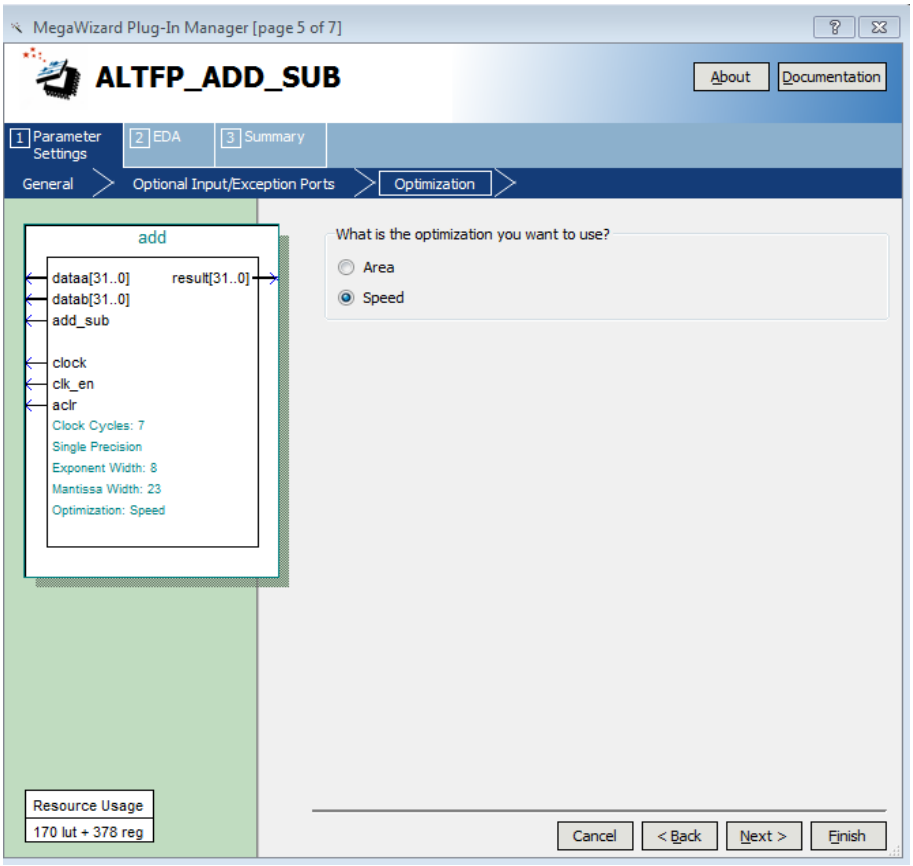


Figure-21

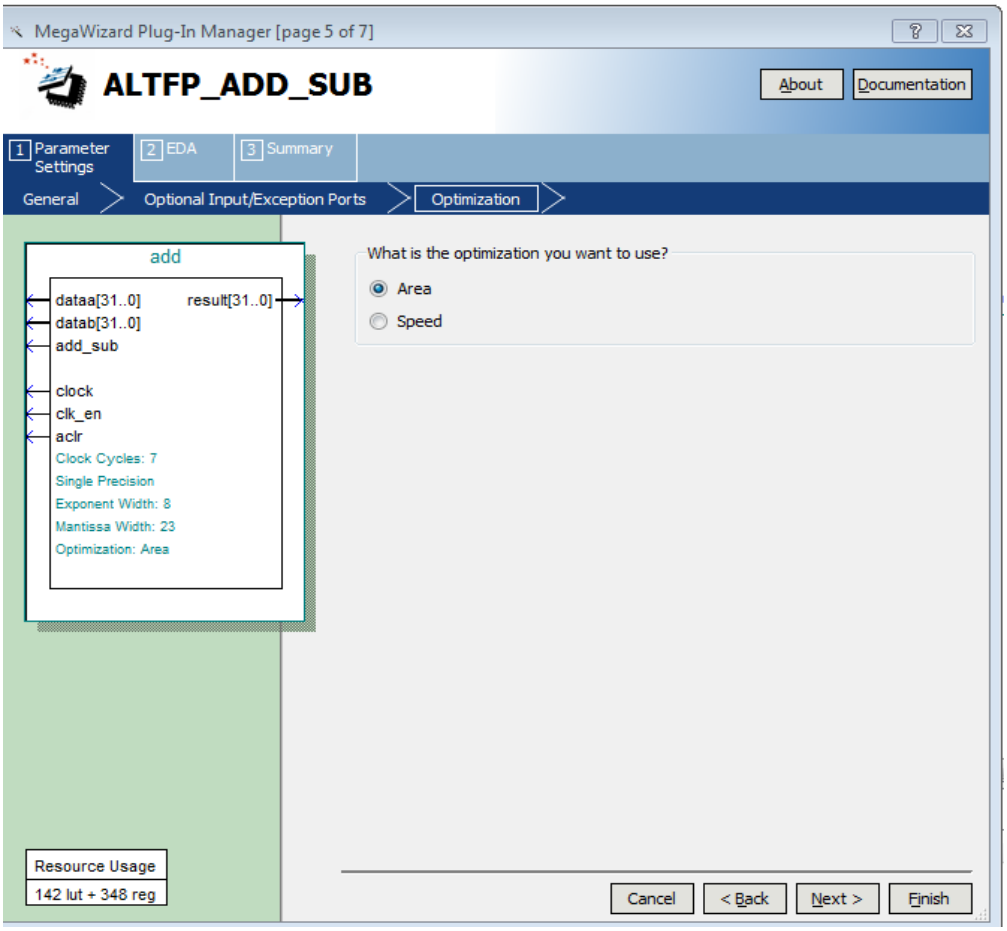


Figure-22

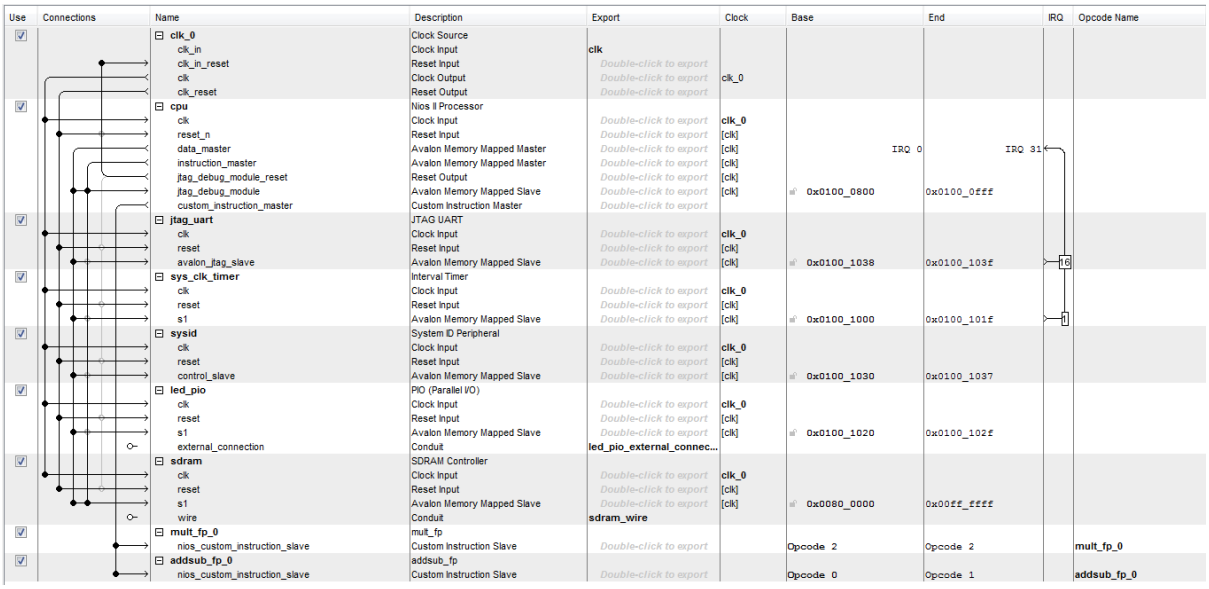


Figure-23

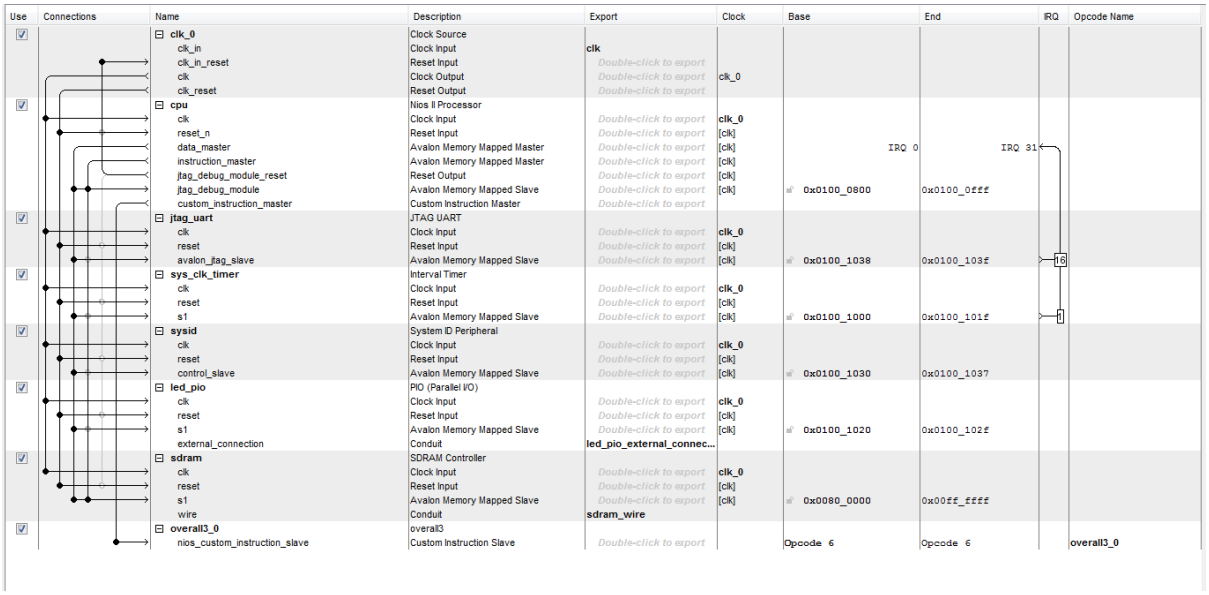


Figure-24

```
1  module mult (dataa,datab,clock,result);
2
3  input [31:0] dataa;
4  input [31:0] datab;
5  input            clock;
6  output[31:0] result;
7  reg [31:0] result;
8
9  wire [47:0] prod;
10 wire [7:0] exp;
11 wire [7:0] exp1;
12 wire [8:0] sum;
13
14    assign prod={1'b1,dataa[22:0]}*{1'b1,datab[22:0]};
15    assign sum={dataa[30:23]}+{datab[30:23]};
16    assign exp1=sum-9'd126;
17    assign exp=sum-9'd127;
18
19 always @(posedge clock)
20 begin
21
22   if((dataa==32'b0)||(datab==32'b0)||(sum<=9'd127))
23     result<=32'b0;
24
```

```
25    else if (prod>=48'h800000000000) result<={1'b0,exp1,prod[46 -:23]};
26    else if (prod>=48'h400000000000) result<={1'b0,exp,prod[45 -:23]};
27    else if (prod>=48'h200000000000) result<={1'b0,exp,prod[44 -:23]};
28    else if (prod>=48'h100000000000) result<={1'b0,exp,prod[43 -:23]};
29    else if (prod>=48'h080000000000) result<={1'b0,exp,prod[42 -:23]};
30    else if (prod>=48'h040000000000) result<={1'b0,exp,prod[41 -:23]};
31    else if (prod>=48'h020000000000) result<={1'b0,exp,prod[40 -:23]};
32    else if (prod>=48'h010000000000) result<={1'b0,exp,prod[39 -:23]};
33    else if (prod>=48'h008000000000) result<={1'b0,exp,prod[38 -:23]};
34    else if (prod>=48'h004000000000) result<={1'b0,exp,prod[37 -:23]};
35    else if (prod>=48'h002000000000) result<={1'b0,exp,prod[36 -:23]};
36    else if (prod>=48'h001000000000) result<={1'b0,exp,prod[35 -:23]};
37    else if (prod>=48'h000800000000) result<={1'b0,exp,prod[34 -:23]};
38    else if (prod>=48'h000400000000) result<={1'b0,exp,prod[33 -:23]};
39    else if (prod>=48'h000200000000) result<={1'b0,exp,prod[32 -:23]};
40    else if (prod>=48'h000100000000) result<={1'b0,exp,prod[31 -:23]};
41    else if (prod>=48'h000080000000) result<={1'b0,exp,prod[30 -:23]};
42    else if (prod>=48'h000040000000) result<={1'b0,exp,prod[29 -:23]};
43    else if (prod>=48'h000020000000) result<={1'b0,exp,prod[28 -:23]};
44    else if (prod>=48'h000010000000) result<={1'b0,exp,prod[27 -:23]};
45    else if (prod>=48'h000008000000) result<={1'b0,exp,prod[26 -:23]};
46    else if (prod>=48'h000004000000) result<={1'b0,exp,prod[25 -:23]};
47    else if (prod>=48'h000002000000) result<={1'b0,exp,prod[24 -:23]};
48    else if (prod>=48'h000001000000) result<={1'b0,exp,prod[23 -:23]};
49    else if (prod>=48'h000000800000) result<={1'b0,exp,prod[22 -:23]};
50
51    else if (prod>=48'h400000)  result<={1'b0,exp,prod[21:0],1'b0};
52    else if (prod>=48'h200000)  result<={1'b0,exp,prod[20:0],2'b0};
53    else if (prod>=48'h100000)  result<={1'b0,exp,prod[19:0],3'b0};
54    else if (prod>=48'h080000)  result<={1'b0,exp,prod[18:0],4'b0};
55    else if (prod>=48'h040000)  result<={1'b0,exp,prod[17:0],5'b0};
56    else if (prod>=48'h020000)  result<={1'b0,exp,prod[16:0],6'b0};
57    else if (prod>=48'h010000)  result<={1'b0,exp,prod[15:0],7'b0};
58    else if (prod>=48'h008000)  result<={1'b0,exp,prod[14:0],8'b0};
59    else if (prod>=48'h004000)  result<={1'b0,exp,prod[13:0],9'b0};
60    else if (prod>=48'h002000)  result<={1'b0,exp,prod[12:0],10'b0};
61    else if (prod>=48'h001000)  result<={1'b0,exp,prod[11:0],11'b0};
62    else if (prod>=48'h000800)  result<={1'b0,exp,prod[10:0],12'b0};
63    else if (prod>=48'h000400)  result<={1'b0,exp,prod[9:0],13'b0};
64    else if (prod>=48'h000200)  result<={1'b0,exp,prod[8:0],14'b0};
```

```
  else if (prod>=48'h000100)  result<={1'b0,exp,prod[7:0],15'b0};
  else if (prod>=48'h000080)  result<={1'b0,exp,prod[6:0],16'b0};
  else if (prod>=48'h000040)  result<={1'b0,exp,prod[5:0],17'b0};
  else if (prod>=48'h000020)  result<={1'b0,exp,prod[4:0],18'b0};
  else if (prod>=48'h000010)  result<={1'b0,exp,prod[3:0],19'b0};
  else if (prod>=48'h000008)  result<={1'b0,exp,prod[2:0],20'b0};
  else if (prod>=48'h000004)  result<={1'b0,exp,prod[1:0],21'b0};
  else if (prod>=48'h000002)  result<={1'b0,exp,prod[0],22'b0};
  else if (prod>=48'h000001)  result<={1'b0,exp,23'b0};
end


endmodule
```

Code Snippet-5 Self-designed Multiplier

```
a = -1;
b = 1;
e = 10^-10;

c = 0;

for niters = 1:31

    mse = zeros(100,1);
    count = 0;

    for i = 1:100

        thNorm = (b-a).*rand(10,1) + a;

        sumWL  = 32;

        theta  = fi(thNorm, 1, sumWL);

        z_NT   = numerictype(theta);
        xyNT   = numerictype(1, sumWL, sumWL-2);
        x_out  = fi(zeros(size(theta)), xyNT);
        y_out  = fi(zeros(size(theta)), xyNT);
        z_out  = fi(zeros(size(theta)), z_NT);

        inpLUT = fi(atan(2 .^ (-((0:(niters-1))'))), z_NT);
        AnGain = prod(sqrt(1+2.^(-2*(0:(niters-1)))));
        inv_An = 1 / AnGain;

        for idx = 1:length(theta)
            [x_out(idx), y_out(idx), z_out(idx)] = ...
                fidemo.cordic_rotation_kernel(...
                    fi(inv_An, xyNT), fi(0, xyNT), theta(idx),
inpLUT, niters);
        end

```

```
        error = cos(thNorm) - double(x_out);
        se = sum(error.^2);
        mse(i) = se/length(theta);
        if mse(i) > e
            count = count + 1;
        end
    end

    if count<=5
        c = c+1;
    end

end
```

Code Snippet-6 MATLAB Code for Monte Carlo Simulation

```
function [x, y, z] = cordic_rotation_kernel(x, y, z, inpLUT, n)
% Perform CORDIC rotation kernel algorithm for N iterations.
xtmp = x;
ytmp = y;
for idx = 1:n
    if z < 0
        z(:) = accumpos(z, inpLUT(idx));
        x(:) = accumpos(x, ytmp);
        y(:) = accumneg(y, xtmp);
    else
        z(:) = accumneg(z, inpLUT(idx));
        x(:) = accumneg(x, ytmp);
        y(:) = accumpos(y, xtmp);
    end
    xtmp = bitsra(x, idx); % bit-shift-right for multiply by 2^(-idx)
    ytmp = bitsra(y, idx); % bit-shift-right for multiply by 2^(-idx)
end
```

Code Snippet-7 MATLAB Code for Monte Carlo Simulation-2

Reference

[1]*INTRODUCTION TO CORDIC,* Available at: *http://shodhganga.inflibnet.ac.in/bitstream/10603/141715/13/13_chapter%205.pdf*(Accessed: 12th March 2019).

[2](December 2018) *Digital Circuits/CORDIC,* Available at: *https://en.wikibooks.org/wiki/Digital_Circuits/CORDIC* (Accessed: 12th March 2019).

[3] *Verilog Turotials,* Available at: *http://www.hdlexpress.com/Verilog/VT.html* (Accessed: 10th March 2019).

[4](December 2017) *Nios II Custom Instruction User Guide,* Available at: *https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_nios2_custom_instruction.pdf*(Accessed: 10 March 2019).

[5] *Compute Sine and Cosine Using CORDIC Rotation Kernel,* Available at: *https://uk.mathworks.com/help/fixedpoint/examples/compute-sine-and-cosine-using-cordic-rotation-kernel.html* (Accessed: 8 March 2019).

[6] *DMA Definition,* Available at: *https://techterms.com/definition/dma* (Accessed: 14th March 2019).

[7]Margaret Rouse (September 2005) *Direct Memory Access (DMA),* Available at: *https://whatis.techtarget.com/definition/Direct-Memory-Access-DMA* (Accessed: 14th March 2019).

[8] *Avalon Memory-Mapped Master Templates,* Available at: *https://www.intel.com/content/www/us/en/programmable/support/support-resources/design-examples/intellectual-property/embedded/nios-ii/exm-avalon-mm.html* (Accessed: 14 March 2019).

[9](November 2018) *Embedded Design Handbook,* Available at: *https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_embedded_ip.pdf*(Accessed: 14 March 2019).