Final Project and Report

Course Code: ENEL 864
Course Title: FPGA Design Applications

Submitted to:
Dr. Lei Zhang
Associate Professor
Electronic Systems Engineering

Submitted by:
MARUF AHMAD
ID: **200460690**
Email: mah370@uregina.ca
MASc in Electronic Systems Engineering
The University of Regina.

Date of Submission: **15 December 2022**

Complex Exponential Neuron Implementation on FPGA

In the spiking neural networks, the firing patterns are the summation of oscillating futures of the interconnected neurons. The oscillating features such as frequency, amplitude, and phase shift can easily be encoded by the complex exponential form. The simple phase addition of complex exponential function simplifies the intensive amount of calculation that is required in conventional artificial neural networks.

Mathematical model:

In the phase plane, an oscillating neuron E(t) can be written by the complex exponential form as (1),

$$E(t) = e^{i\omega t+\theta} = e^{\theta}e^{i\omega t} = e^{\theta}(\cos\omega t + i\sin\omega t)$$

(1)[1]

Where ω is the angular frequency, the real component θ is used to make the oscillation amplitude.

The synaptic weight can also be represented by the complex exponential form as $W = e^{i\phi}$. Where ∅ represents the phase delay of the neural connection.

Therefore a weighted input can be represented as E(t).W = $e^{i\omega t+\theta}e^{i\phi} \ = \ e^{\theta}e^{i(\dot{\omega}t+\bar{\phi})}$.

In the neural network, pre-synaptic neurons are multiplied by their corresponding weight and then summed up to the post-synaptic neuron. This summation (S) in the post-synaptic neuron can also be represented by a complex exponential form as in (2).

$$\begin{aligned} S &= E_1W_1 + E_2W_2 \\ &= e^{i\omega_1 t+\theta_1}e^{i\phi_1} + e^{i\omega_2 t+\theta_2}e^{i\phi_2} \\ &= e^{\theta_1}e^{i(\omega_1 t+\phi_1)} + e^{\theta_2}e^{i(\omega_2 t+\phi_2)} \end{aligned}$$

(2)

Three parameters ω, ϴ and ∅ are used to calculate the weighted sum.
In this report the equation of the complex exponential function is represented by

$$e^{i\phi} = cos\phi \ + \ isin\phi \qquad (3)$$

MATLAB Simulation Report:

In my MATLAB model weighted sum of two neurons is considered as an example. Figure 1 shows the output of the weighted sum of two neurons E1W1 + E2W2 with different parameter values.

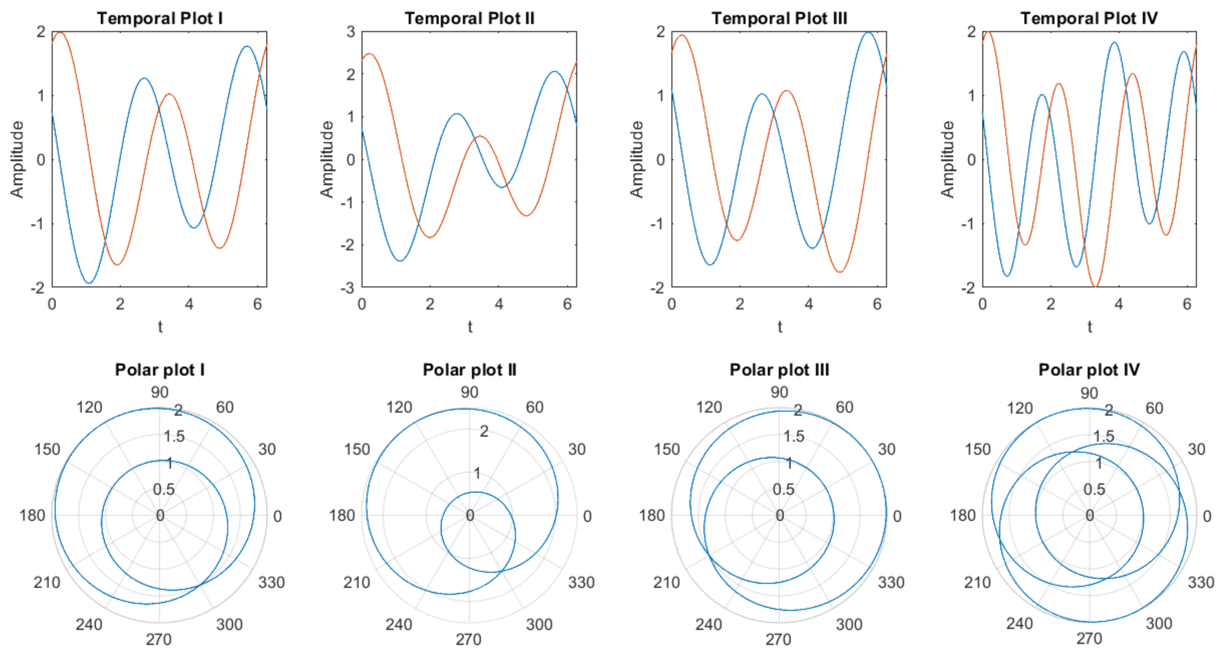


Figure 1: Examples of the Weighted sum of two Complex Exponential neurons with different parameter values

- Example 1 (temporal plot I and Polar plot I) shows the output of the weighted sum of two neurons where ω1 = 1, ω2 = 2, Ɵ1 = log(0.5), Ɵ2 = log(1.5) , ∅1 = π/2 and ∅2 = π/3 are used.
- In example 2 (temporal plot II and Polar plot II) all parameters of the weighted input of two neurons remain the same except Ɵ1 = log(1) is taken. This causes a change in amplitude.
- In example 3, Ɵ1 is changed to π/4. This changes the phase and orientation of the weighted sum.
- For example, 4 ω2 is set to 3; this dramatically changes the pattern of the weighted sum, which is shown in the polar plot iv.

Table 1: Parameters for two complex exponential neurons

Inputs:	$E1 \times W1 = e^{i\omega 1t+\theta 1}\ e^{i\varphi 1}$			$E2 \times W2 = e^{i\omega 2t+\theta 1}\ e^{i\varphi 2}$		
	ω1	*θ1*	*φ1*	*ω2*	*θ2*	*φ2*
I	1	ln(0.5)	*π/2*	2	ln(1.5)	*π/3*
II	1	ln(1)	*π/2*	2	ln(1.5)	*π/3*
III	1	ln(0.5)	*π/4*	2	ln(1.5)	*π/3*
IV	1	ln(0.5)	*π/2*	3	ln(1.5)	*π/3*

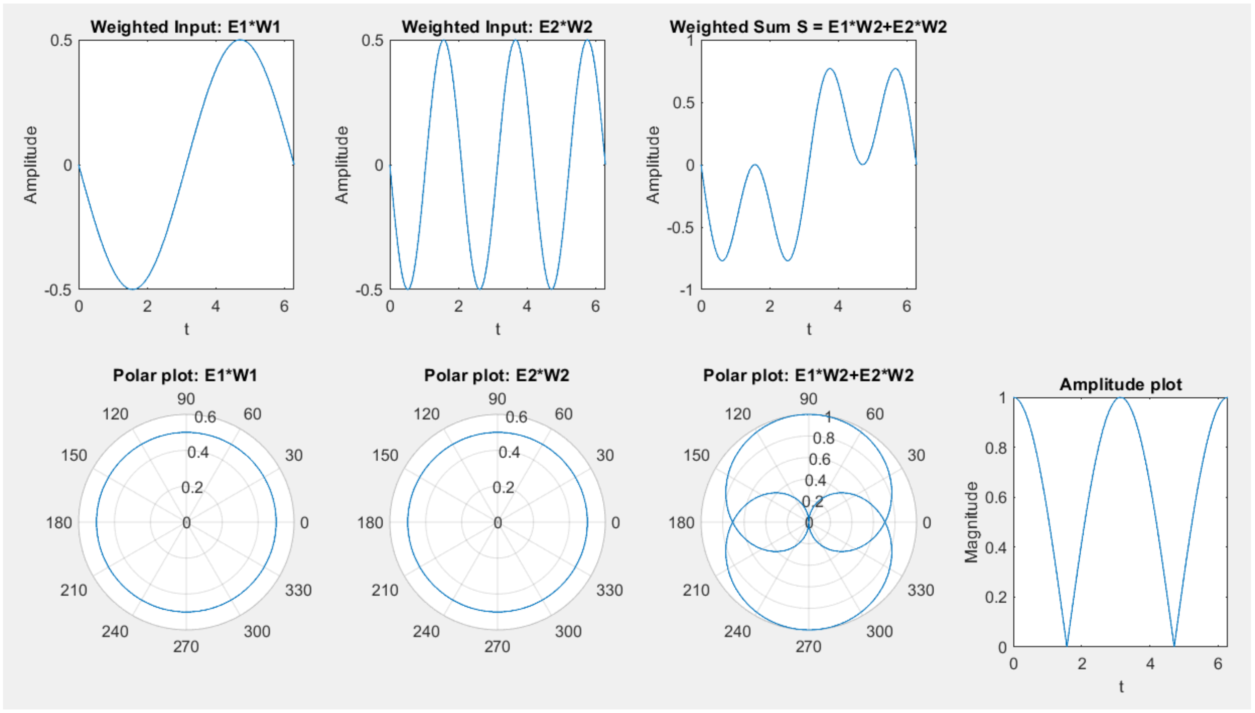


Figure 2: Weighted sum of two complex exponential neurons

VHDL and MATLAB Simulink implementation:

In this report, I have presented three ways of implementing the weighted sum of two neurons.

In the first approach, the complex exponential function of the weighted sum of two neurons is implemented using the lookup table method where Block Random-Access-Memory(BRAM) HDL Blocks from the Simulink Xilinx toolbox are used. Three separate designs are implemented using 8, 16, and 32-bit fixed-point data format configurations, respectively and then generate IP core using Xilinx System Generator.

In the second approach, a CORDIC 6.0 HDL block from the Xilinx toolbox in model composer in Simulink is used to get the value of the exponential function (sinɸ and cosɸ). In this case, also, three separate designs are implemented using 8, 16, and 32-bit fixed-point data format configurations respectively.

In the third approach, the weighted sum of two neurons is directly implemented in VHDL coding using CORDIC 6.0 IP core from the Xilinx Vivado design suite. In this approach also, the IP core is configured with 8, 16, and 32-bit fixed-point data format and three separate projects are created, respectively.

Finally, the above outputs are compared with the model design implemented in MATLAB Simulink.

In all approaches below configuration are used.

Inputs	$E_1 ✖ W_1 = e^{iw_1 t+Θ_1} e^{iØ_1}$			$E_1 ✖ W_1 = e^{iw_2 t+Θ_2} e^{iØ_2}$		
	w_1	$Θ_1$	$Ø_1$	w_2	$Θ_2$	$Ø_2$
	1	log(2)	pi/2	2	log(2)	pi/2

And the Period $t = -4\ to\ +4$ is used for all the setups.

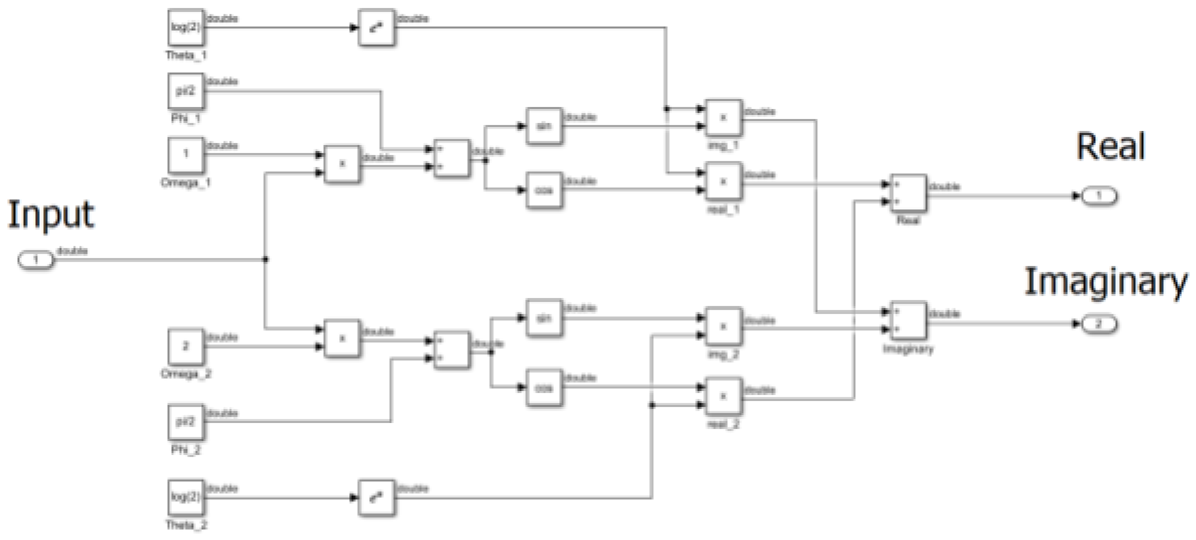
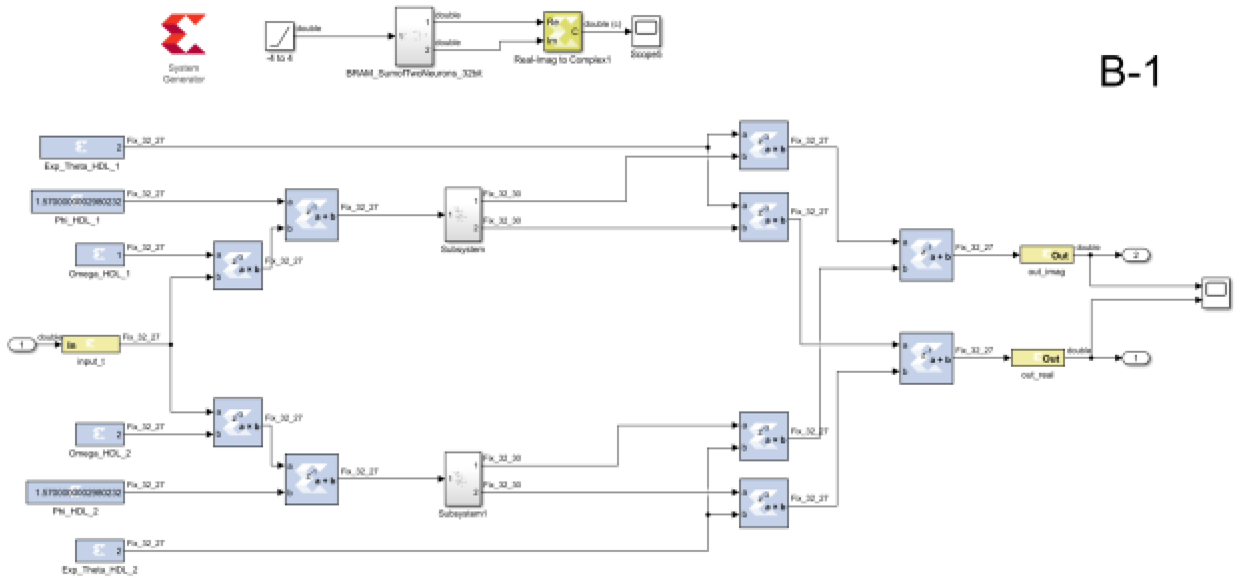


Figure 3: MATLAB simulink design of the weighted sum of two neurons

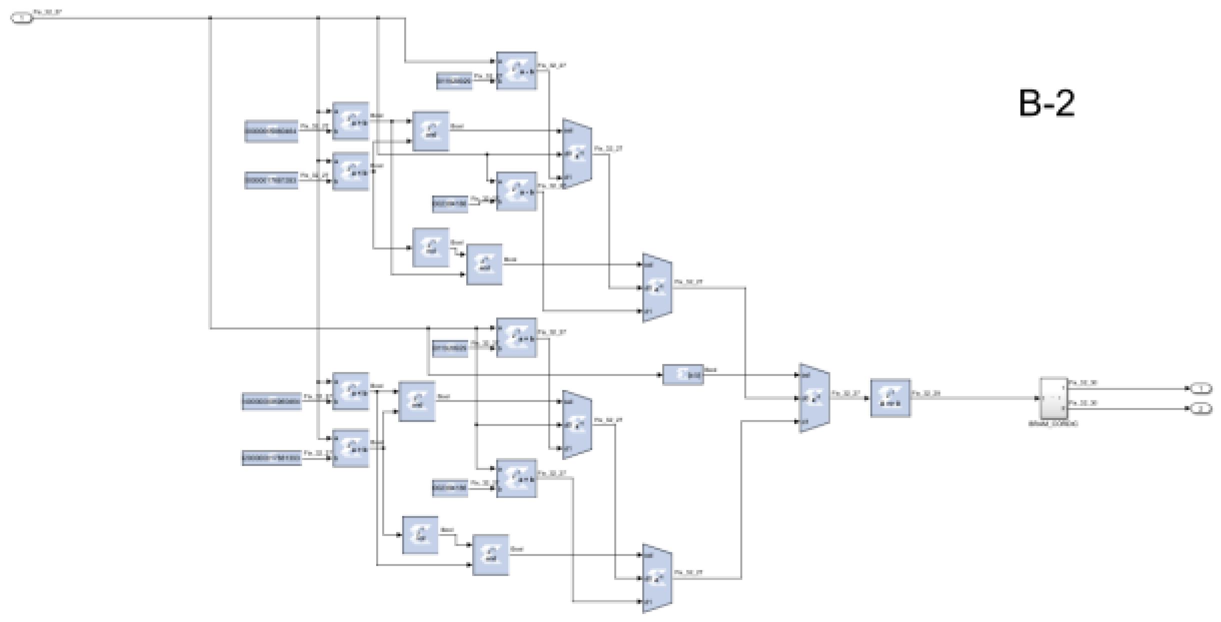
1. BRAM based Design in Model composer:

One of the important tasks for complex exponential neurons model is to calculate the real ($cosØ$) and imaginary ($sinØ$) value based inputs. For this purpose, we used a Random-access-memory block called BRAM, where the $cosØ$ and $sinØ$ values agonist inputs are pre-stored in a memory cell. In this project, the input for the real and imaginary value generator (BRAM block) is -3.14 to +3.14 with 0.01 precision. Therefore we only needed 629 depth of word cells in Block-RAM memory. In figure 2, subsystem B-3 is the basic input setup for the BRAM. The setup matches the input to the corresponding output memory words. Subsystem B-3 wraps the overflow inputs in the range of -3.14 to 3.14. Section B-1 represents the main block diagram for the weighted sum of complex exponential neurons implemented on MATLAB Simulink by Xilinx model composer HDL blocks.

B-1



B-2



B-3

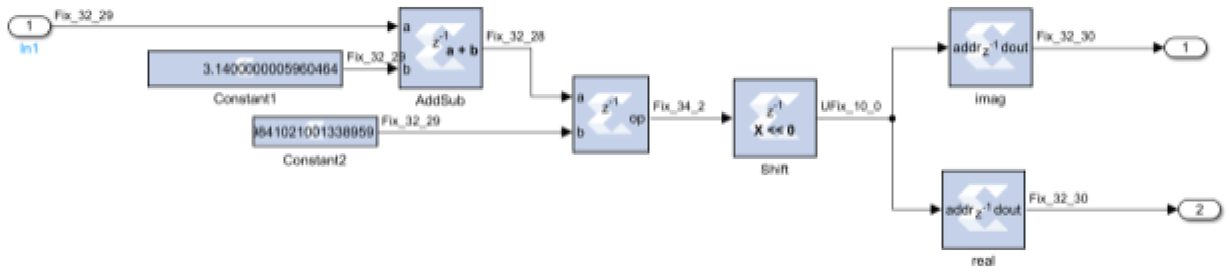


Figure 2: Lookup table method-based design using Block-RAM of the Xilinx System generator Toolbox. (B-1: weighted sum of two neurons, B-2: normalization block, B-3: Lookup table block)

2. CORDIC IP-based design in model composer:

In this approach to generate real and imaginary values, a pair of CORDIC IP cores are used. However, to wrap the input signal between -3.14 to 3.14, a wrapping subsystem is implemented before the CORDIC IP Core shown in figure 4 section A-2. Section A-1 represents the main block diagram of the IP-Core-based design for the weighted sum of complex exponential neurons implemented on MATLAB Simulink by Xilinx model composer HDL blocks.

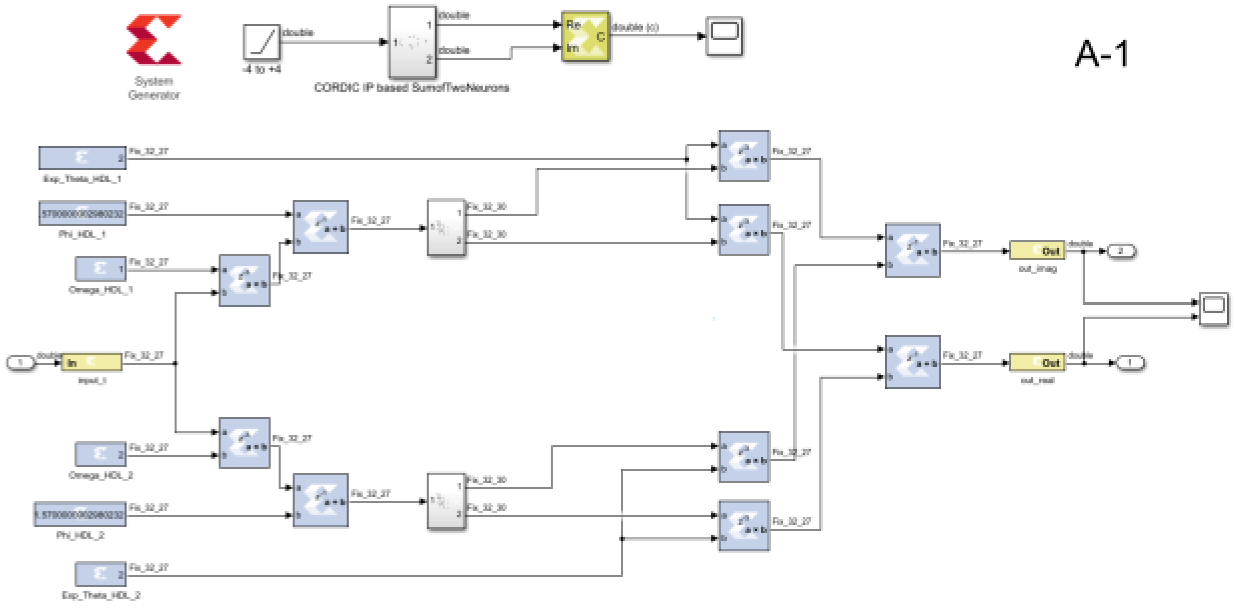


Figure 4.1: CORDIC-IP-core-based design in model composer

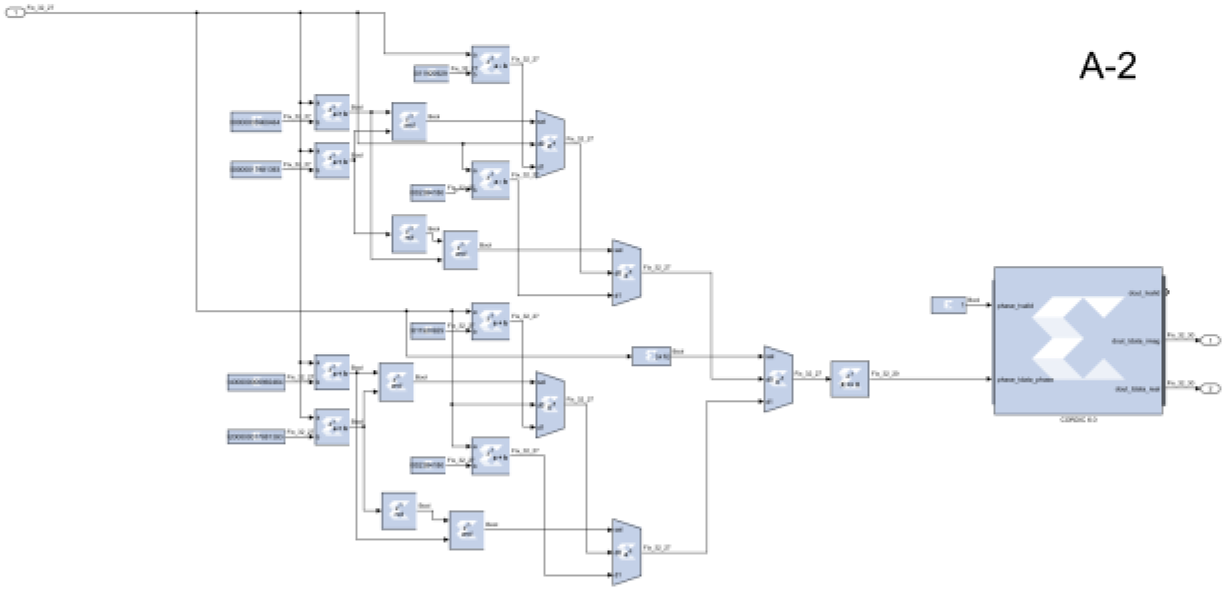


Figure 4.2: Wrapping subsystem for the CORDIC-IP-Core-based design in model composer

3. Direct VHDL implementation of the weighted sum of two neurons:

In this approach to generate real and imaginary values, a pair of CORDIC IP cores are used and a reset of the model is designed by VHDL coding in Vivado design suite. In this project 8, 16, and 32-bit fixed-point implementations are implemented and compared with the other approaches.

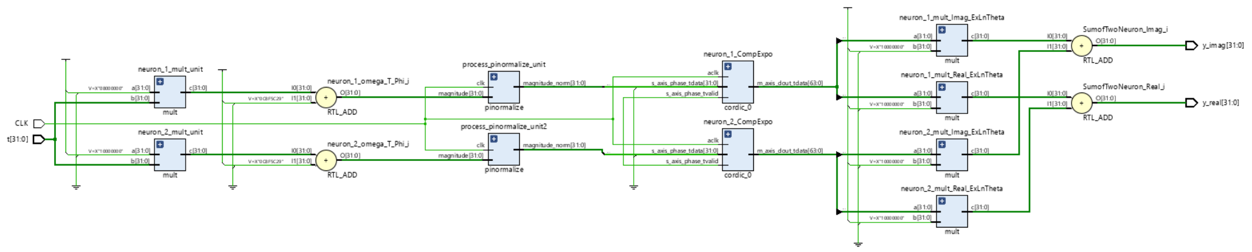


Figure 5: Direct VHDL coding-based implementation of the weighted sum of two neurons using CORDIC IP core in the Video.

The fixed-point implementation:

In this project, the input lies between -4 and 4, and the output lies between -1 and 1. Therefore, to represent decimal 3 in binary, only three bits are enough, one bit for the sign, and the rest of the bit could be set for fractional bits. However, internally for addition and multiplication input value goes up to 16. Therefore, five bits are reserved for decimal numbers. And since the maximum and minimum value of the output is 1 and -1, one bit is for the sign, one bit is for decimal 1, and the rest are for the fractions.

Based on the above conditions, for the CORDIC and BRAM configuration, in an 8-bit fixed-point implementation, the input is configured as Fix8_5 data format, with one sign bit, two integer bits and five fractional bits. The output comes with Fix8_6 data format, with one sign bit, one integer bit and six fractional bits.

In 16-bit fixed-point implementation, the input is configured as Fix16_13 data format, with one sign bit, two integer bits and 13 fractional bits. The output comes with Fix16_14 data format, with one sign bit, one integer bit and 14 fractional bits.

In 32-bit fixed-point implementation, the input is configured as Fix32_29 data format, with one sign bit, two integer bits and 29 fractional bits. The output comes with Fix32_30 data format, with one sign bit, one integer bit and 30 fractional bits.

Simulation Results:

8-bit Fixed-point implementation result: Fix_8_3:

Below shows the simulation output of the Block-RAM-based design in Simulink, Cordic-IP-Core based design in Simulink and direct VHDL coding-based design in Vivado for 8-bit fixed-point implementation.

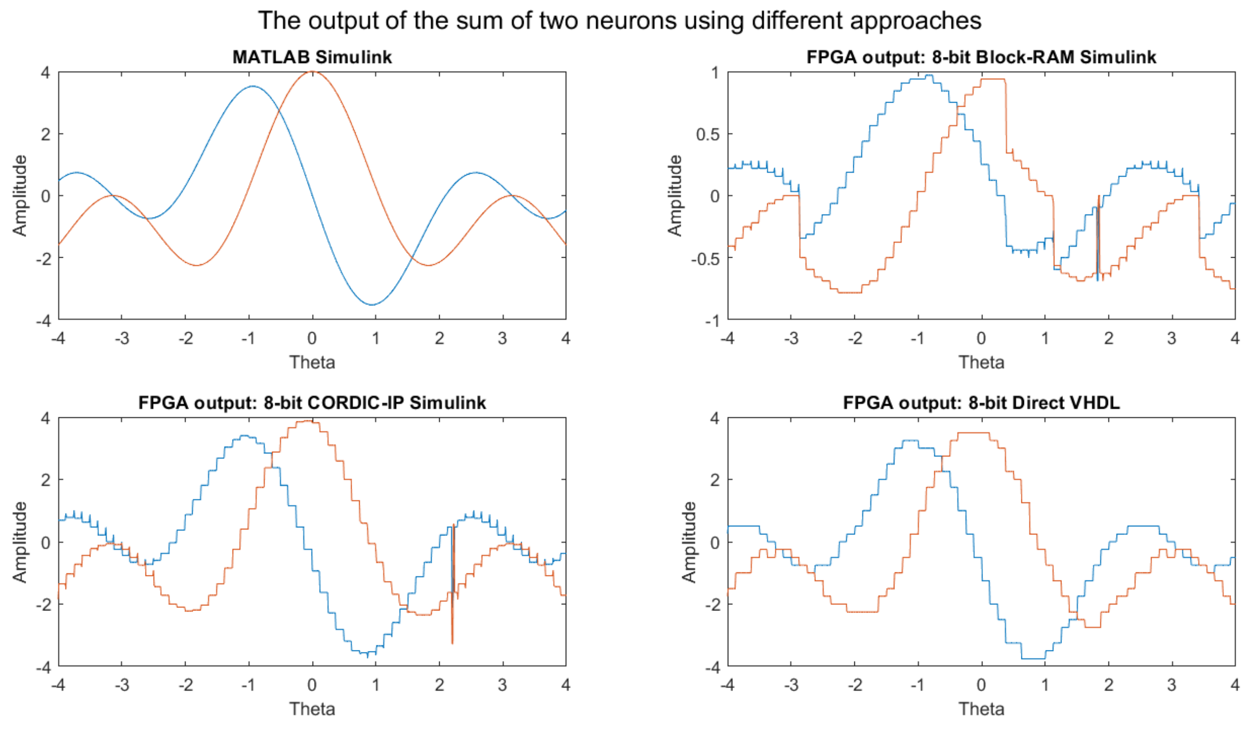


Figure 6: FPGA simulation output graphs of the weighted sum of complex exponential neurons in three different approaches for 8-bit fixed point implementation.

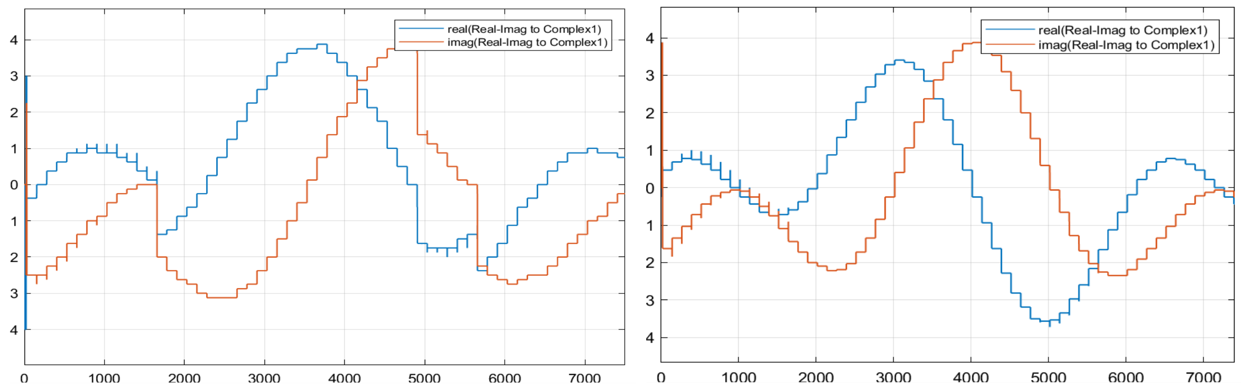


Figure 7: BRAM-based (left) and IP-Core-based (right) design simulation output in MATLAB Simulink scope for 8-bit fixed point implementation.

The graphs in figure 6 show the FPGA simulation output of the weighted sum of complex exponential neurons implemented by three different approaches in 8-bit fixed point implementation and compare these to the MATLAB simulation output. We can observe that there are some sharp spikes (near theta =2) on the BRAM-based and CORDIC-IP-Core-based design FPGA simulation outputs, which are not present in the Simulink Model Composer scope outputs shown in figure 7. The reason and solution for this will be discussed later in this report.

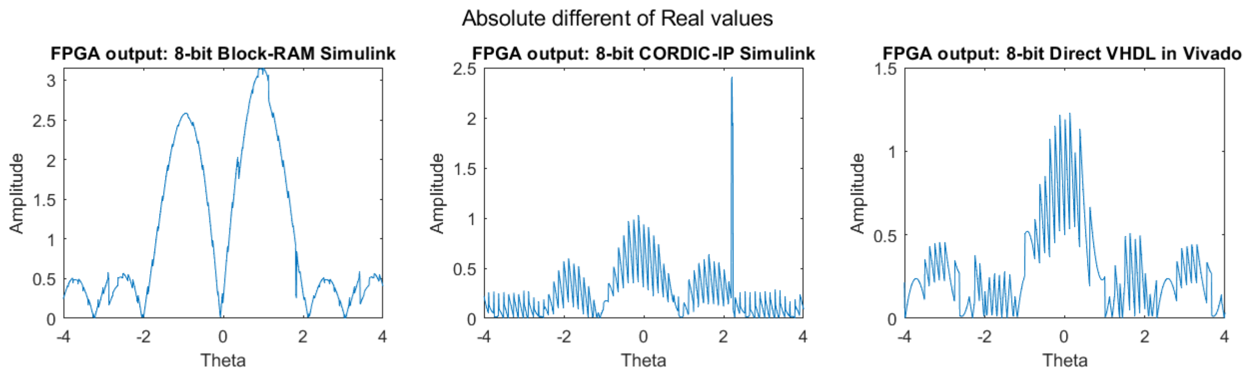


Figure 8: Absolute difference of MATLAB simulation real values with three different approaches.

The above graphs in figure 8 show the absolute difference of real values of the weighted sum of complex exponential neurons with three different approaches: Block-RAM-based design, IP-Core-based design and direct VHDL coding-based design for 8-bit fixed point implementation.

It is observed that 8-bit fixed-point implementation with three different approaches can produce output signals similar to the actual output but the precision of the output is very poor in comparison with the expected output shown in figure 8.

16-bit Fixed-point implementation result: Fix_16_11:

Below shows the simulation output of the Block-RAM based design in Simulink, Cordic-IP-Core based design in Simulink and direct VHDL coding-based design in Vivado for 16-bit fixed-point implementation.

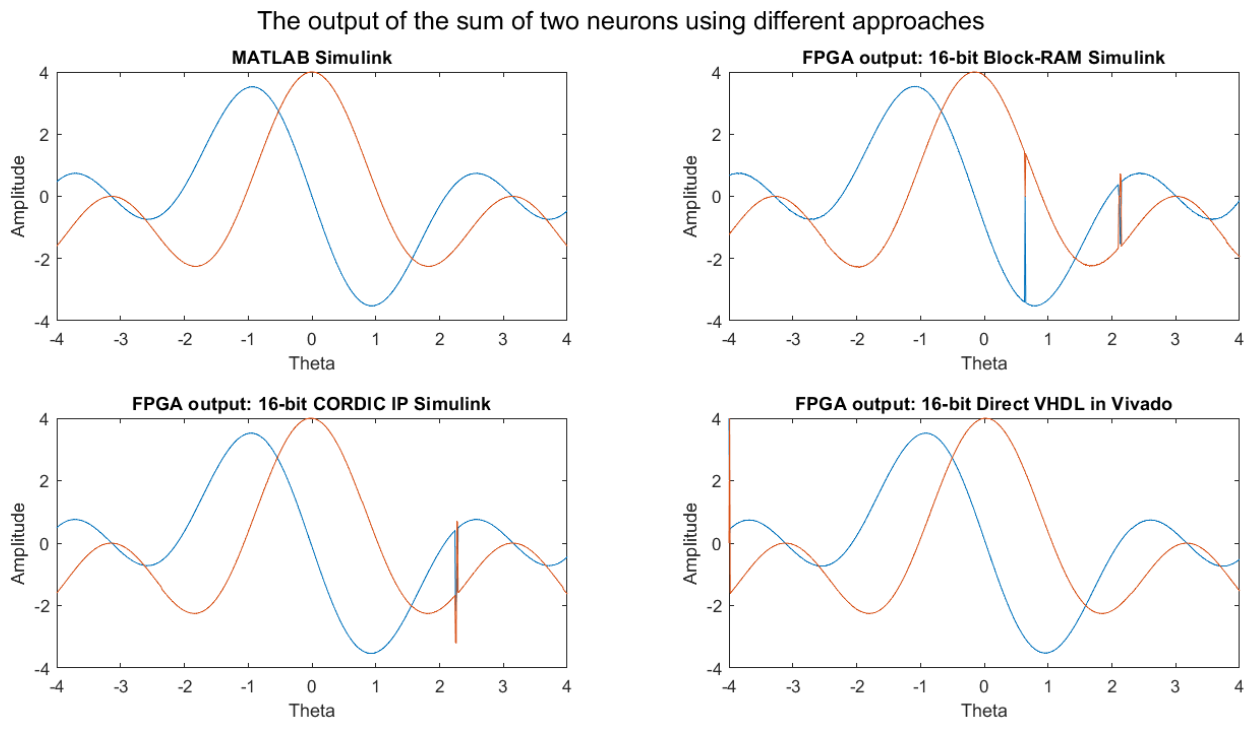


Figure 9: FPGA simulation output graphs of the weighted sum of complex exponential neurons in three different approaches for 16-bit fixed-point implementation.

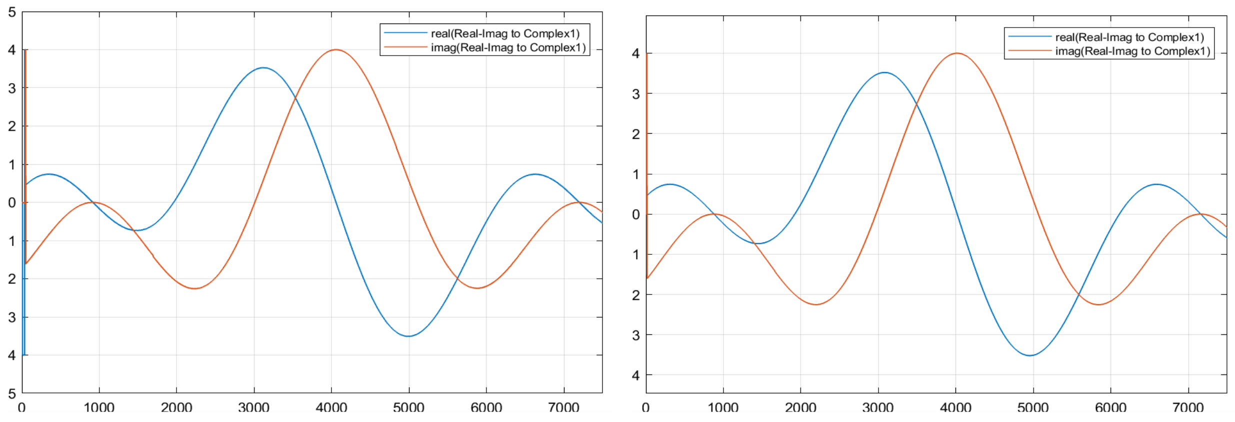


Figure 10: BRAM-based (left) and IP-Core-based (right) design simulation output in MATLAB Simulink scope for 16-bit fixed-point implementation.

The graphs in Figure 9 show the FPGA simulation output of the weighted sum of complex exponential neurons implemented by three different approaches in 16-bit fixed point implementation and compare these to the MATLAB simulation output. We can observe that there are some sharp spikes (near theta =2) on the BRAM-based and CORDIC-IP-Core-based design FPGA simulation outputs, which are not present in the Simulink Model Composer scope outputs shown in figure 10. However, there are no unwanted spikes in direct VHDL coding-based design. The reason and solution for this will be discussed later in this report.

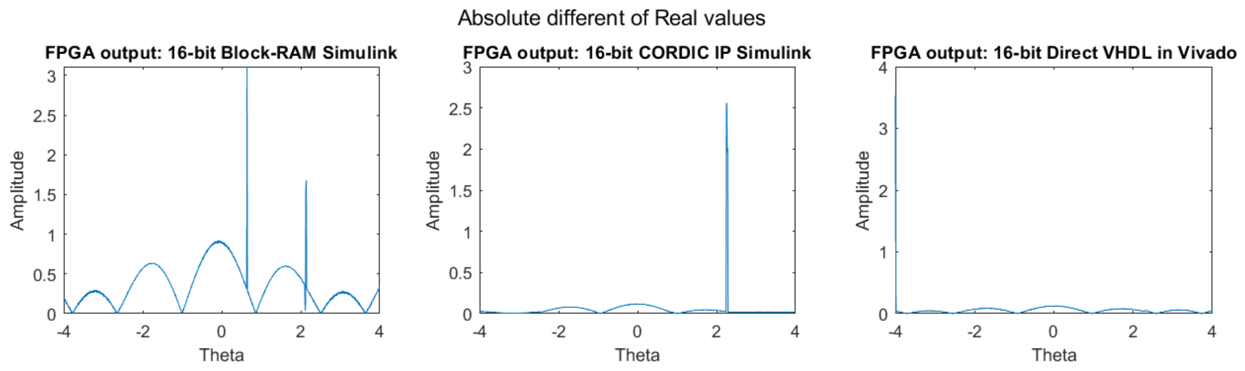


Figure 11: Absolute difference of MATLAB simulation real values with three different approaches for 16-bit implementation.

The above graphs in figure 8 show the absolute difference of real values of the weighted sum of complex exponential neurons with three different approaches: Block-RAM-based design, IP-Core-based design and direct VHDL coding-based design for 16-bit fixed-point implementation.

We can observe from the graphs that precision is much improved in 16-bit fixed point implementation in all three approaches. Though some spikes are shown on graphs in figure 11, those are because of unwanted spikes in the FPGA simulation output. We will resolve this problem in the later section.

32-bit Fixed-point implementation result: Fix_32_27:

Below shows the simulation output of the Block-RAM based design in Simulink, Cordic-IP-Core based design in Simulink and direct VHDL coding-based design in Vivado for 16-bit fixed-point implementation.

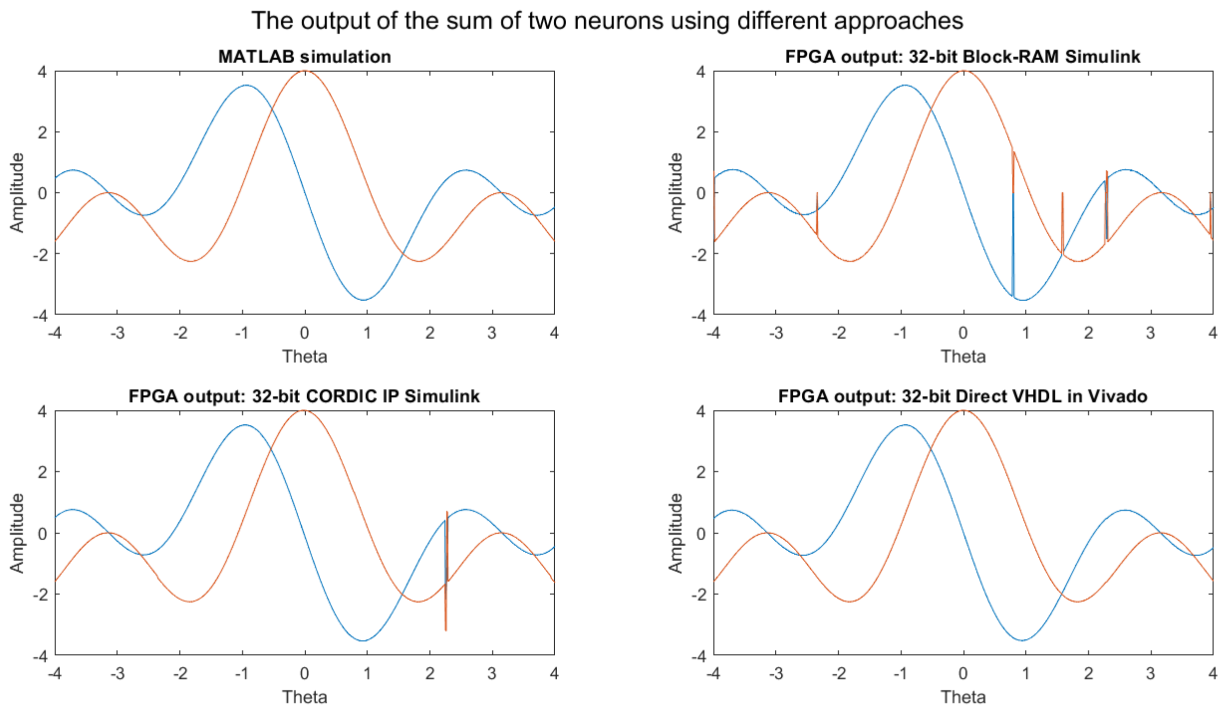


Figure 12: FPGA simulation output graphs of the weighted sum of complex exponential neurons in three different approaches for 32-bit fixed-point implementation.

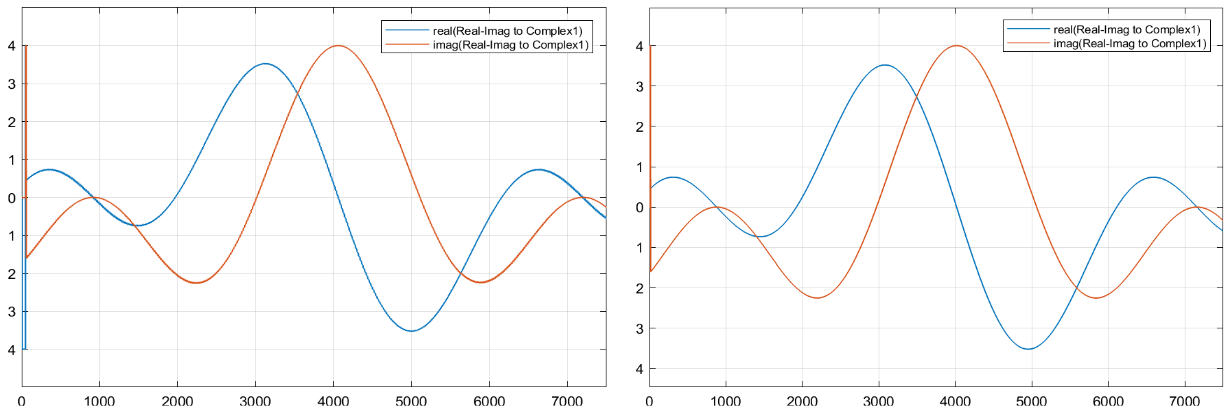


Figure 13: BRAM-based (left) and IP-Core-based (right) design simulation output in MATLAB Simulink scope for 32-bit fixed-point implementation.

The graphs in Figure 12 show the FPGA simulation output of the weighted sum of complex exponential neurons implemented by three different approaches in 32-bit fixed point implementation and compare these to the MATLAB simulation output. We can observe that there are some sharp spikes (near theta =2) on the BRAM-based and CORDIC-IP-Core-based design FPGA simulation outputs, which are not present in the Simulink Model Composer scope outputs shown in figure 13. However, there are no unwanted spikes in direct VHDL coding-based design. The reason and solution for this will be discussed later in this report.

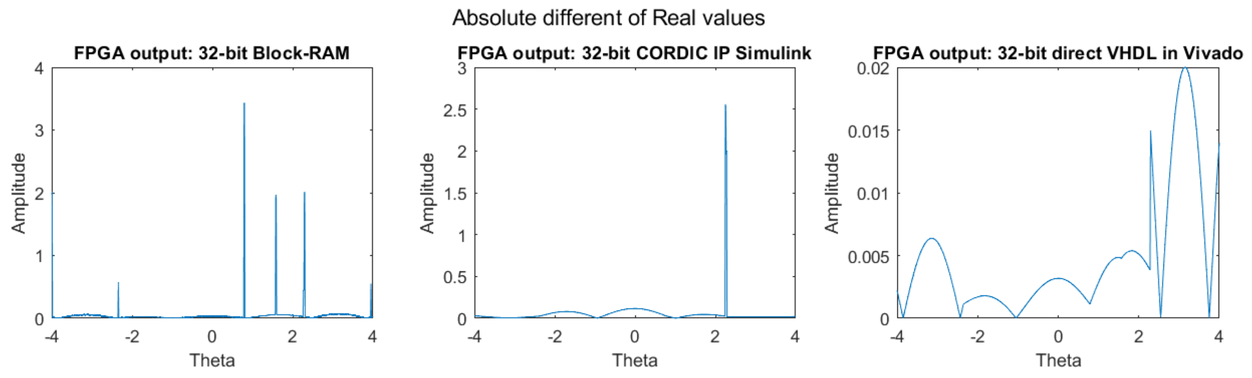


Figure 14: Absolute difference of MATLAB simulation real values with three different approaches for 32-bit implementation.

The above graphs in figure 8 show the absolute difference of real values of the weighted sum of complex exponential neurons with three different approaches: Block-RAM-based design, IP-Core-based design and direct VHDL coding-based design for 16-bit fixed-point implementation.

We can observe from the graphs that precision is much improved in 16-bit fixed point implementation in all three approaches. Though some spikes are shown on graphs in figure 11, those are because of unwanted spikes in the FPGA simulation output. We will resolve this problem in the next section.

Unwanted Spikes Removing:

It can be inferred from the nature of the spikes on the FPGA simulation output that the spikes come from a timing mismatch issue. It can be solved by adding registers(flip-flops) before and after the input and output respectively.

The below diagram shows how it is implemented on CORDIC IP core-based design in Simulink model composer.

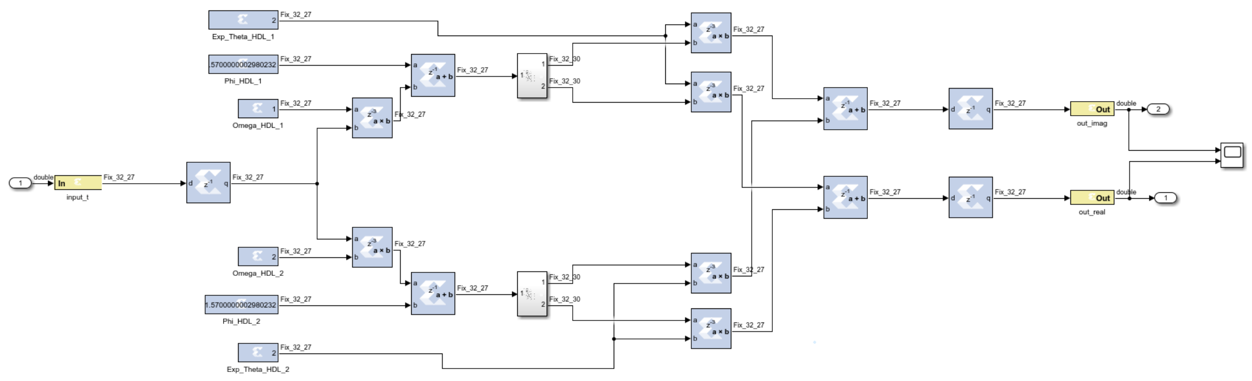


Figure 15: the CORDIC IP core-based design with spike removal registers in Simulink model composer

The spikes removal registers are only added to the BRAM-based design and the CORDIC IP core-based design in Simulink model composer. The direct VHDL coding-based implementation does not face these kinds of problems, so no registers are added to this design approach.

Output graphs with spike removal register:
Below show the FPGA simulation output graphs with spike removal register in BRAM-based and CORDIC IP core-based design in Simulink model composer.

For 8-bit fixed-point implementation:

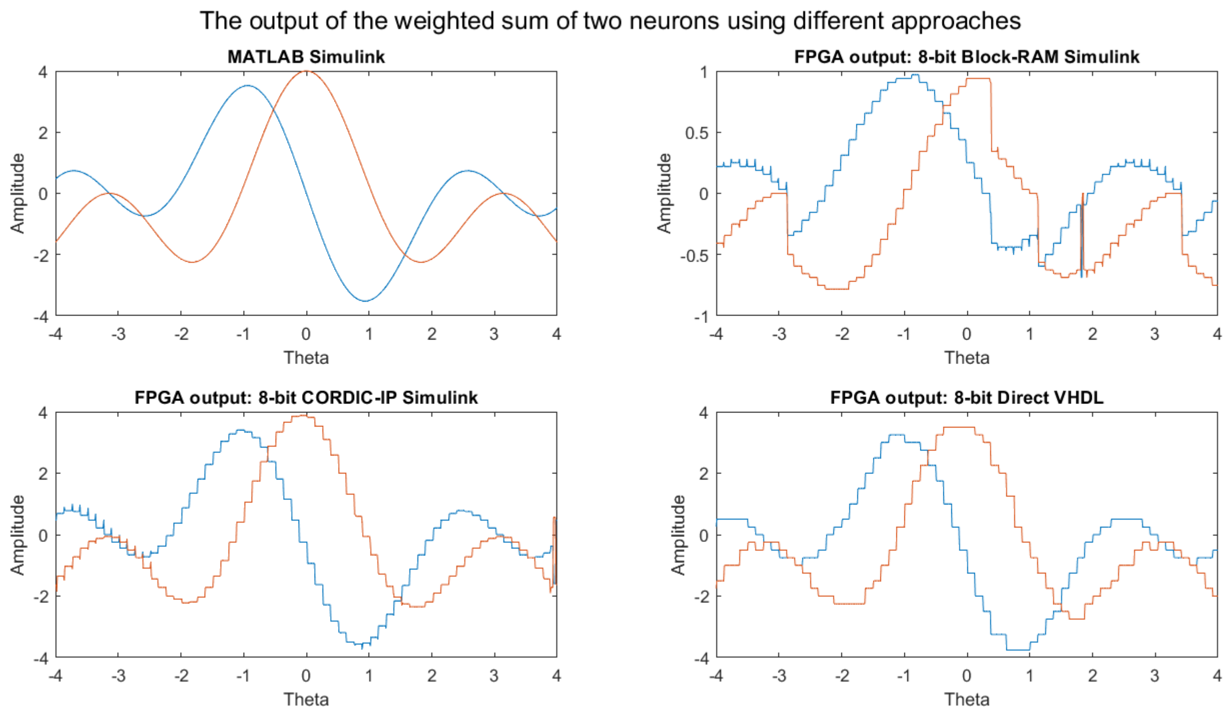


Figure 16: FPGA simulation output graphs of the weighted sum of complex exponential neurons in three different approaches with spiking removal registers for 8-bit fixed-point implementation.

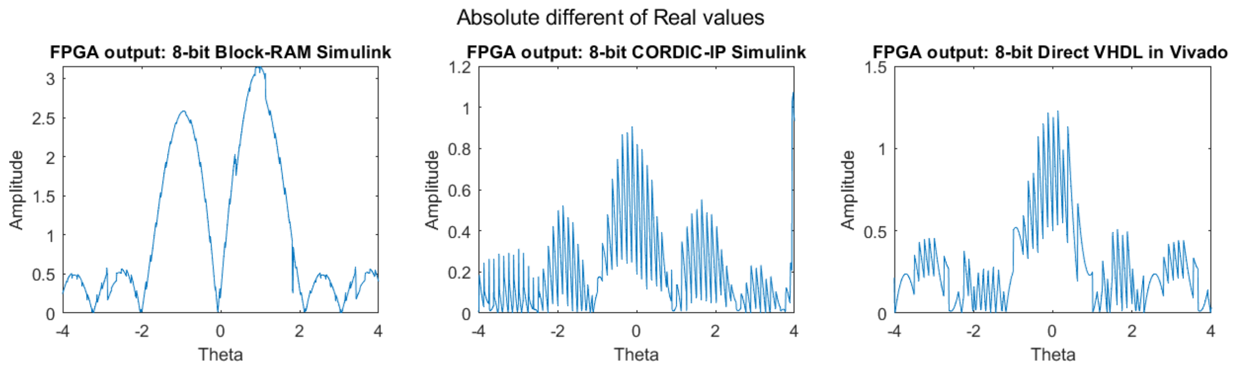


Figure 17: Absolute difference of MATLAB simulation real values with three different approaches with spike removal registers for 8-bit implementation.

For 16-bit fixed-point implementation:

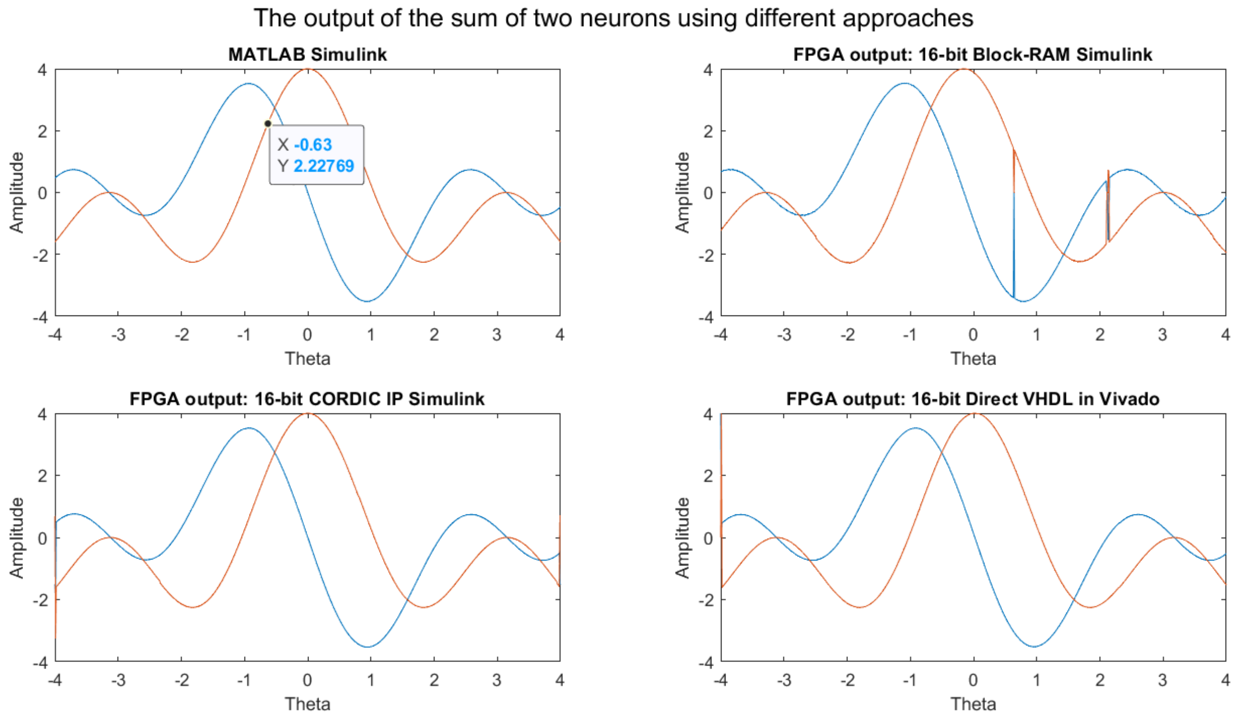


Figure 18: FPGA simulation output graphs of the weighted sum of complex exponential neurons in three different approaches with spiking removal registers for 16-bit fixed-point implementation.

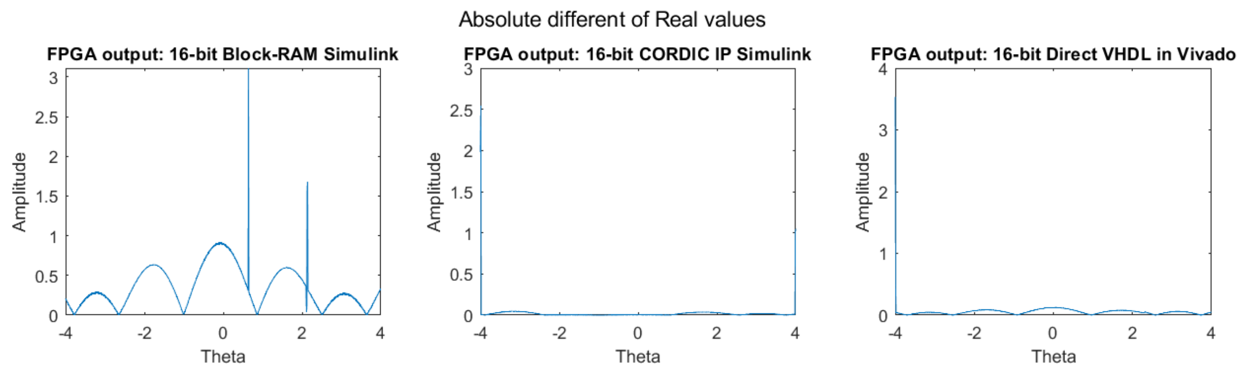


Figure 19: Absolute difference of MATLAB simulation real values with three different approaches with spike removal registers for 16-bit implementation.

For 32-bit fixed-point implementation:

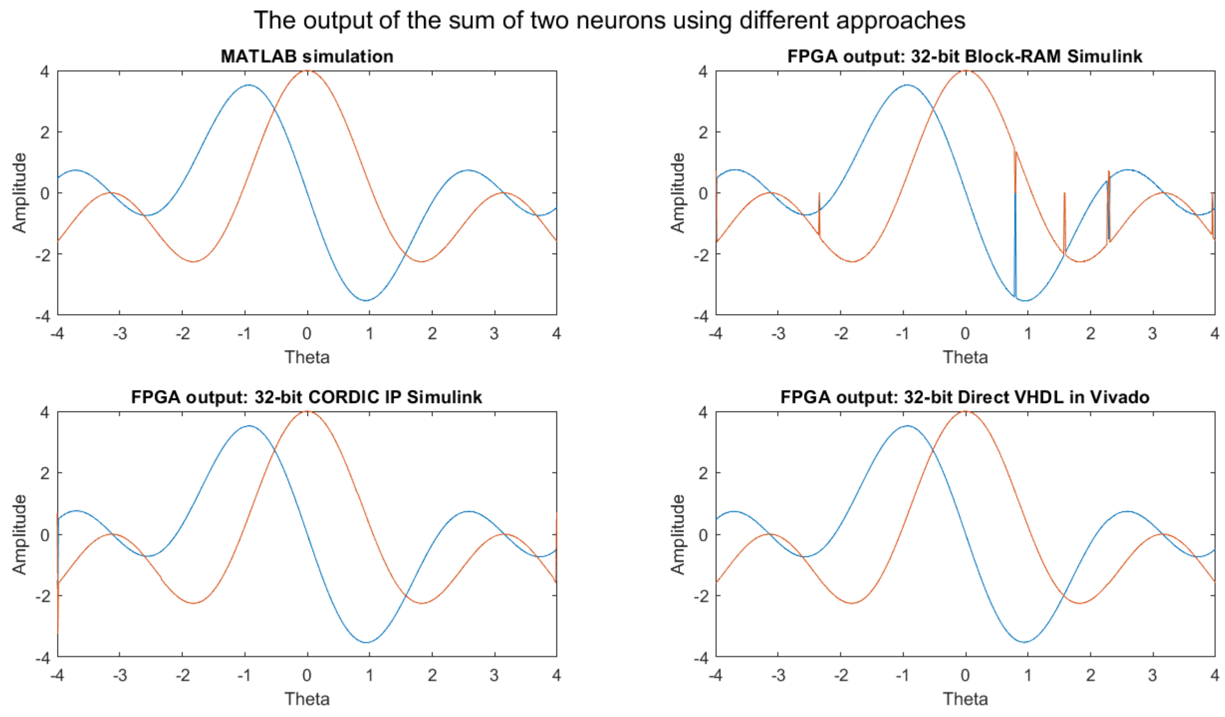


Figure 20: FPGA simulation output graphs of the weighted sum of complex exponential neurons in three different approaches with spiking removal registers for 32-bit fixed-point implementation.

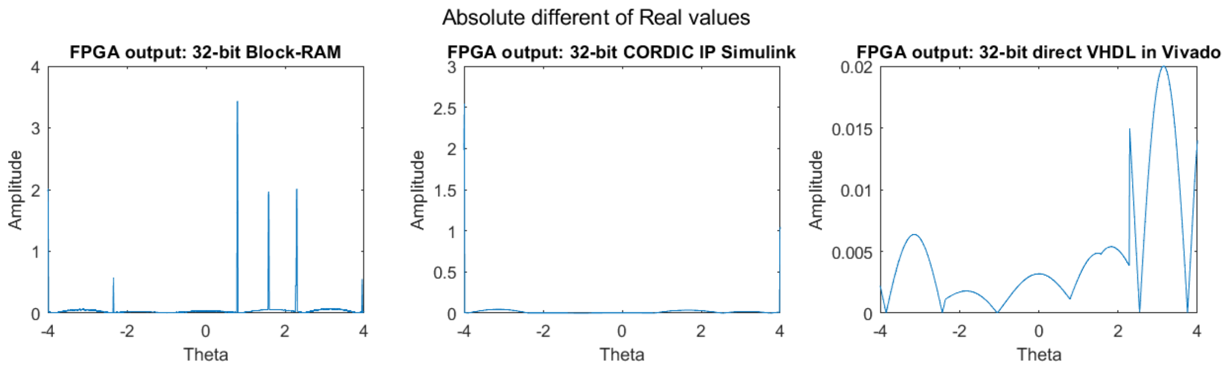


Figure 21: Absolute difference of MATLAB simulation real values with three different approaches with spike removal registers for 32-bit implementation.

From the graphs in figures 16,17,18,19,20, and 21, it can be observed that with added register for removing spiking in BRAM and CORDIC IP core-based design, only the CORDIC IP core-based design is improved. The spike removal register, which is caused for the timing issue, can only resolve spikes from CORDIC IP core-based design. The BRAM-based design remains the same problem. In our next research, I will find the reason for this.

Mean squared error in direct VHDL design with 8, 16, and 32 bits:

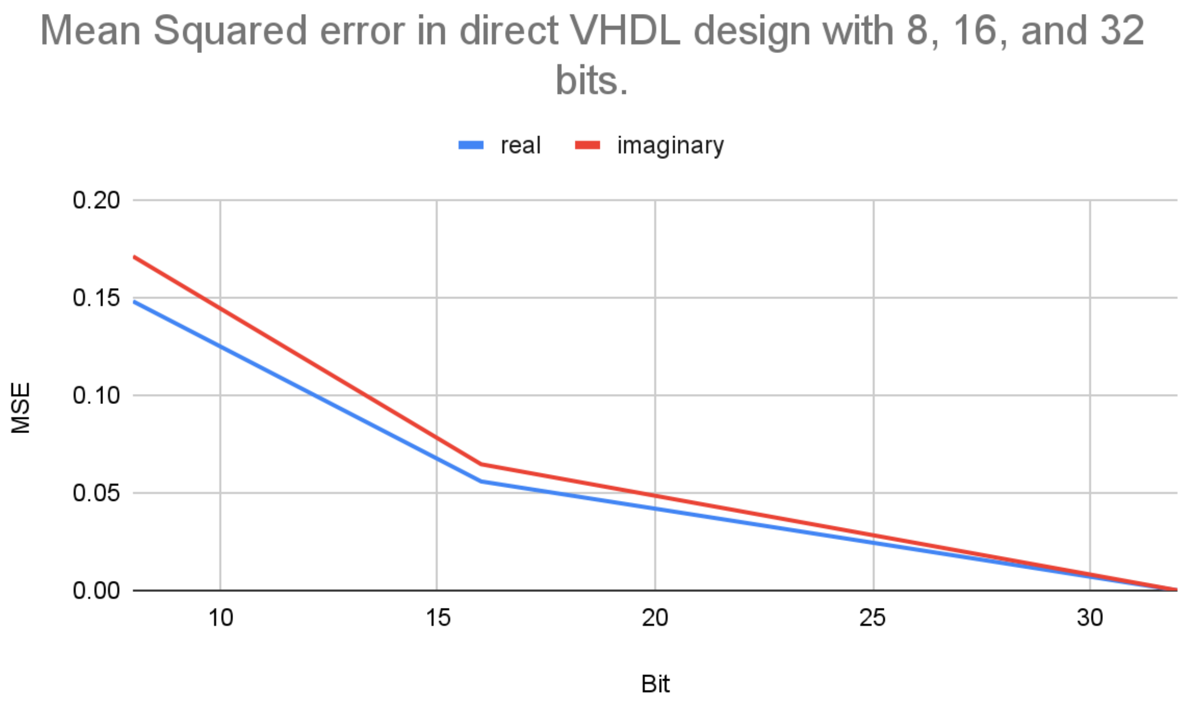


Figure 22: mean squared error in direct VHDL design with 8, 16, and 32 bits.

From the mean squared error graph it can be infrared that 32-bit fixed-point implementation outperforms over the the 8,16-bit implementations.

Maximum operating frequency calculation:

Maximum Operating Frequency is calculated by the formula below:

$$f_{max} = \frac{1}{T_s - WNS}$$ —---------------------------------- (4)

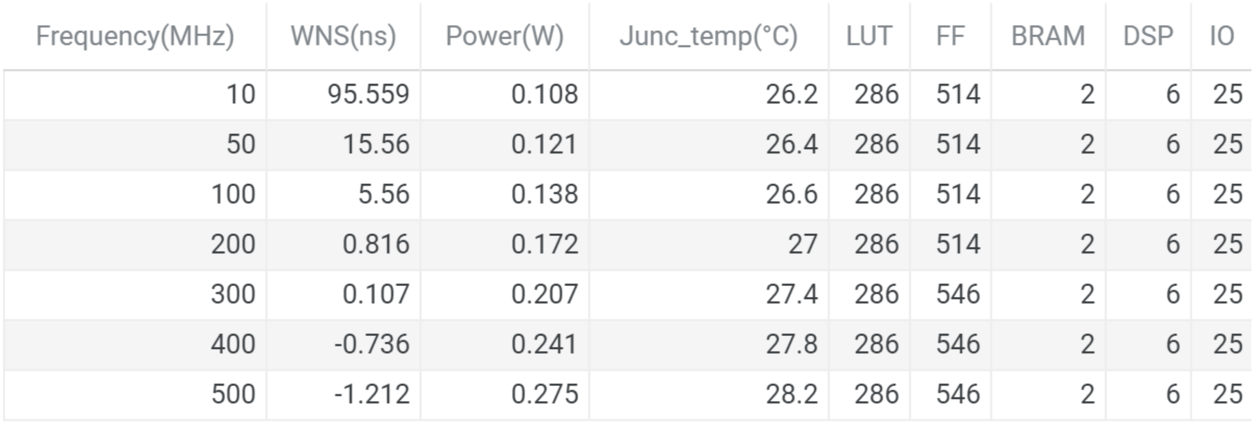
Where, f_{max} is maximum clock frequency, WNS is Worst Negative Slack, T_s is minimum time required to complete a sequence.

In maximum operating frequency, WNS will be nearly 0 and $f_{max} = 1/T_s$

Implementation reports:

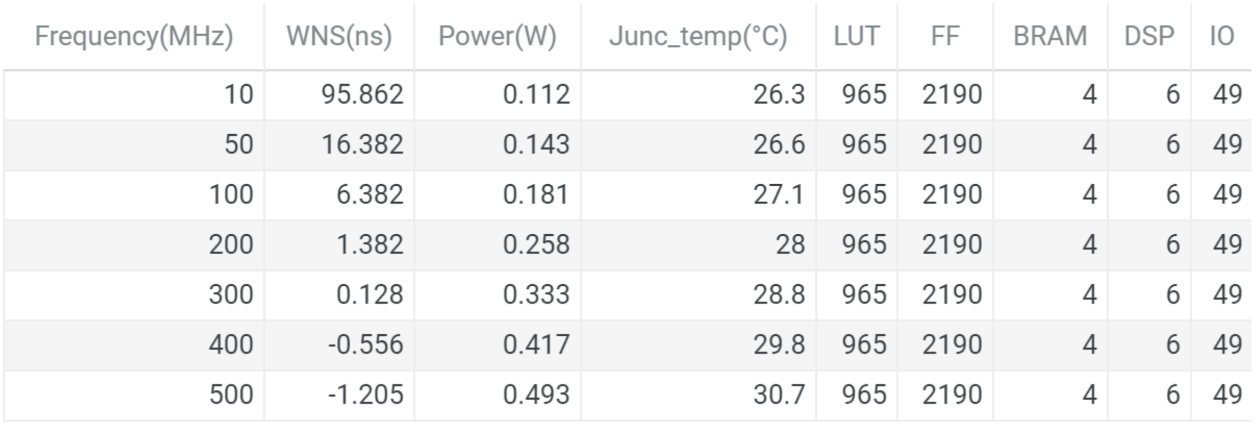
Below are the implementation reports for the Block-RAM based design in Simulink, Cordic-IP-Core based design in Simulink and direct VHDL coding-based design in Vivado for 8, 16, and 32-bit implementations.

Table 1: 8-bit BRAM-based implementation in Simulink Model Composer



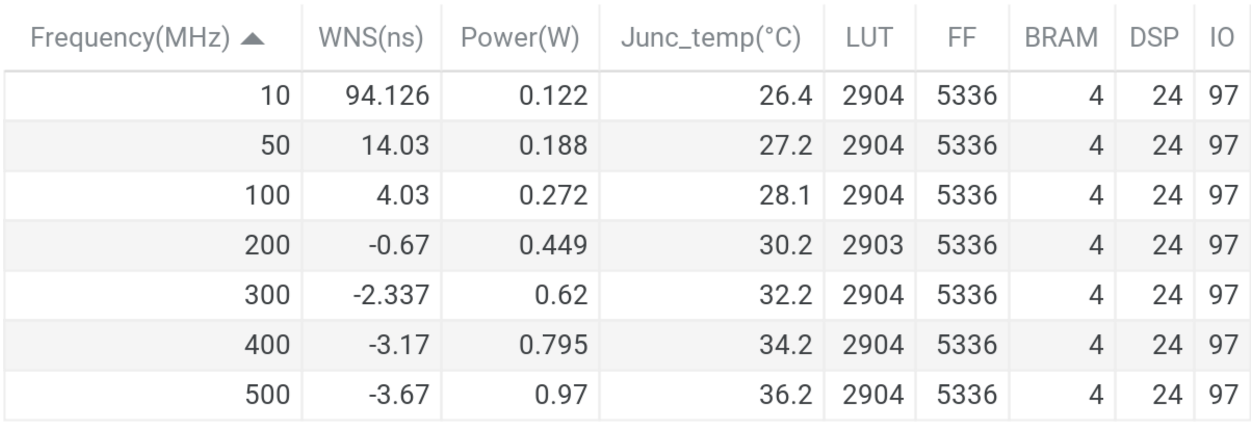
Frequency(MHz)	WNS(ns)	Power(W)	Junc_temp(°C)	LUT	FF	BRAM	DSP	IO
10	95.559	0.108	26.2	286	514	2	6	25
50	15.56	0.121	26.4	286	514	2	6	25
100	5.56	0.138	26.6	286	514	2	6	25
200	0.816	0.172	27	286	514	2	6	25
300	0.107	0.207	27.4	286	546	2	6	25
400	-0.736	0.241	27.8	286	546	2	6	25
500	-1.212	0.275	28.2	286	546	2	6	25

Table 2: 16-bit BRAM-based implementation in Simulink Model Composer



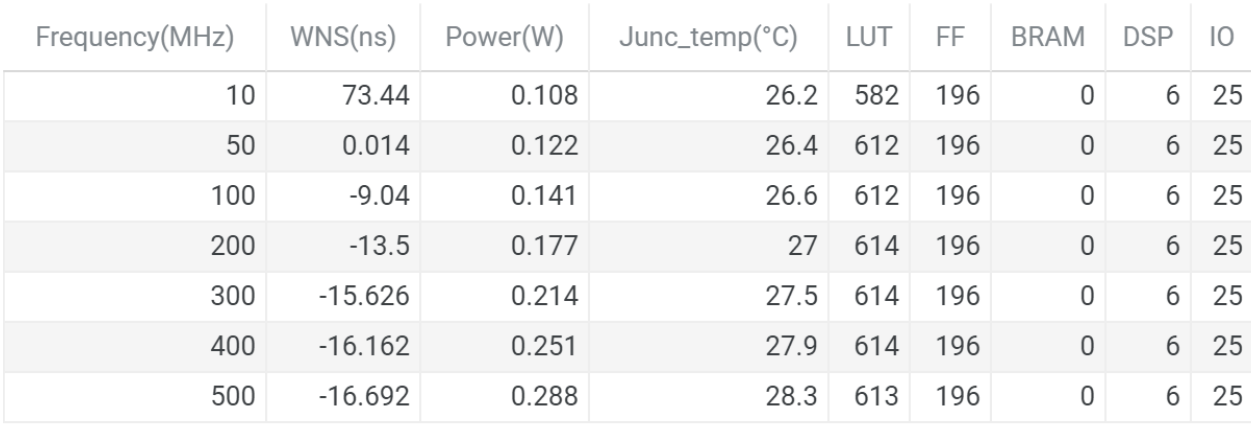
Frequency(MHz)	WNS(ns)	Power(W)	Junc_temp(°C)	LUT	FF	BRAM	DSP	IO
10	95.862	0.112	26.3	965	2190	4	6	49
50	16.382	0.143	26.6	965	2190	4	6	49
100	6.382	0.181	27.1	965	2190	4	6	49
200	1.382	0.258	28	965	2190	4	6	49
300	0.128	0.333	28.8	965	2190	4	6	49
400	-0.556	0.417	29.8	965	2190	4	6	49
500	-1.205	0.493	30.7	965	2190	4	6	49

Table 3: 32-bit BRAM-based implementation in Simulink Model Composer



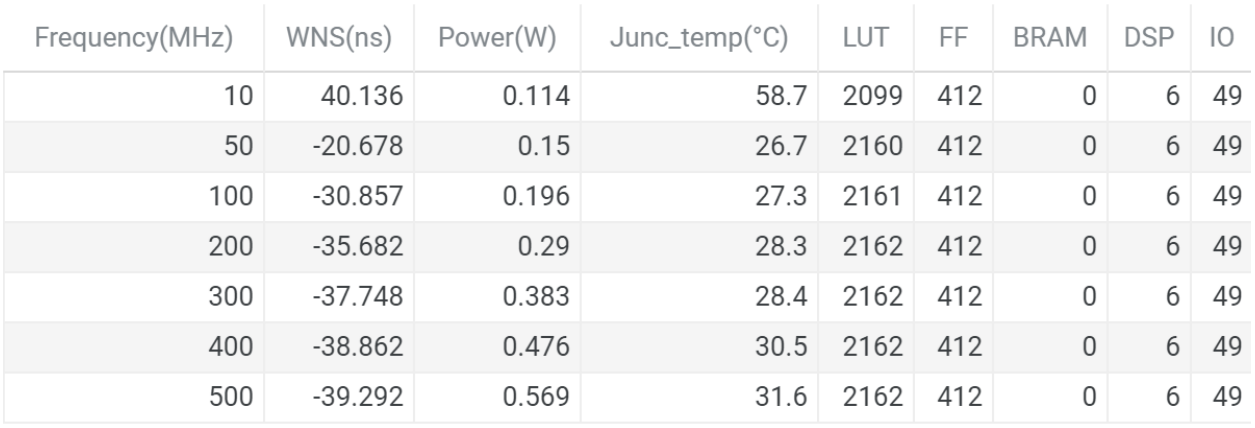
Frequency(MHz) ▲	WNS(ns)	Power(W)	Junc_temp(°C)	LUT	FF	BRAM	DSP	IO
10	94.126	0.122	26.4	2904	5336	4	24	97
50	14.03	0.188	27.2	2904	5336	4	24	97
100	4.03	0.272	28.1	2904	5336	4	24	97
200	-0.67	0.449	30.2	2903	5336	4	24	97
300	-2.337	0.62	32.2	2904	5336	4	24	97
400	-3.17	0.795	34.2	2904	5336	4	24	97
500	-3.67	0.97	36.2	2904	5336	4	24	97

Table 4: 8-bit CORDIC IP_Core based implementation in Simulink Model Composer



Frequency(MHz)	WNS(ns)	Power(W)	Junc_temp(°C)	LUT	FF	BRAM	DSP	IO
10	73.44	0.108	26.2	582	196	0	6	25
50	0.014	0.122	26.4	612	196	0	6	25
100	-9.04	0.141	26.6	612	196	0	6	25
200	-13.5	0.177	27	614	196	0	6	25
300	-15.626	0.214	27.5	614	196	0	6	25
400	-16.162	0.251	27.9	614	196	0	6	25
500	-16.692	0.288	28.3	613	196	0	6	25

Table 5: 16-bit CORDIC IP_Core based implementation in Simulink Model Composer



Frequency(MHz)	WNS(ns)	Power(W)	Junc_temp(°C)	LUT	FF	BRAM	DSP	IO
10	40.136	0.114	58.7	2099	412	0	6	49
50	-20.678	0.15	26.7	2160	412	0	6	49
100	-30.857	0.196	27.3	2161	412	0	6	49
200	-35.682	0.29	28.3	2162	412	0	6	49
300	-37.748	0.383	28.4	2162	412	0	6	49
400	-38.862	0.476	30.5	2162	412	0	6	49
500	-39.292	0.569	31.6	2162	412	0	6	49

Table 6: 32-bit CORDIC IP_Core based implementation in Simulink Model Composer

Frequency(MHz)	WNS(ns)	Power(W)	Junc_temp(°C)	LUT	FF	BRAM	DSP	IO
10	-1.74	0.133	26.5	7658	874	0	24	97
50	-79.358	0.248	27.9	7658	874	0	24	97
100	-89.3401	0.389	29.5	7660	874	0	24	97
200	-94.366	0.677	32.8	7660	874	0	24	97
300	-95.939	0.966	36.1	7660	874	0	24	97
400	-96.335	1.257	39.5	7660	874	0	24	97
500	-96.754	1.542	42.8	7660	874	0	24	97

Table 7: 8-bit Direct VHDL implementation:

Frequency(MHz)	WNS(ns)	Power(W)	Junc_temp(°C)	LUT	FF	BRAM	DSP	IO
10	96.367	0.107	26.2	548	516	0	0	25
50	16.378	0.119	26.4	548	516	0	0	25
100	6.378	0.133	26.5	548	516	0	0	25
200	1.661	0.162	26.9	548	516	0	0	25
300	0.372	0.192	27.2	548	516	0	0	25
400	-0.172	0.22	27.5	558	518	0	0	25
500	-0.686	0.249	27.9	558	520	0	0	25

Table 8: 16-bit Direct VHDL implementation

Frequency(MHz)	WNS(ns)	Power(W)	Junc_temp(°C)	LUT	FF	BRAM	DSP	IO
10	94.524	0.114	26.3	2092	1944	0	0	49
50	15.618	0.15	26.7	2094	1944	0	0	49
100	6.072	0.195	27.3	2094	1944	0	0	49
200	1.108	0.287	28.3	2092	1944	0	0	49
300	-0.011	0.376	29.3	2097	1946	0	0	49
400	-0.482	0.46	30.3	2118	1950	0	0	49
500	-1.187	0.553	31.4	2118	1948	0	0	49

Table 9: 32-bit Direct VHDL implementation

Frequency(MHz)	WNS(ns)	Power(W)	Junc_temp(°C)	LUT	FF	BRAM	DSP	IO
10	92.887	0.134	26.6	7453	7216	0	0	97
50	13.43	0.255	27.9	7458	7216	0	0	97
100	4.783	0.404	29.7	7458	7216	0	0	97
200	0.572	0.701	33.1	7473	7217	0	0	97
300	-0.704	0.996	36.5	7513	7223	0	0	97
400	-1.274	1.291	39.9	7513	7220	0	0	97
500	-1.806	1.579	43.2	7513	7223	0	0	97

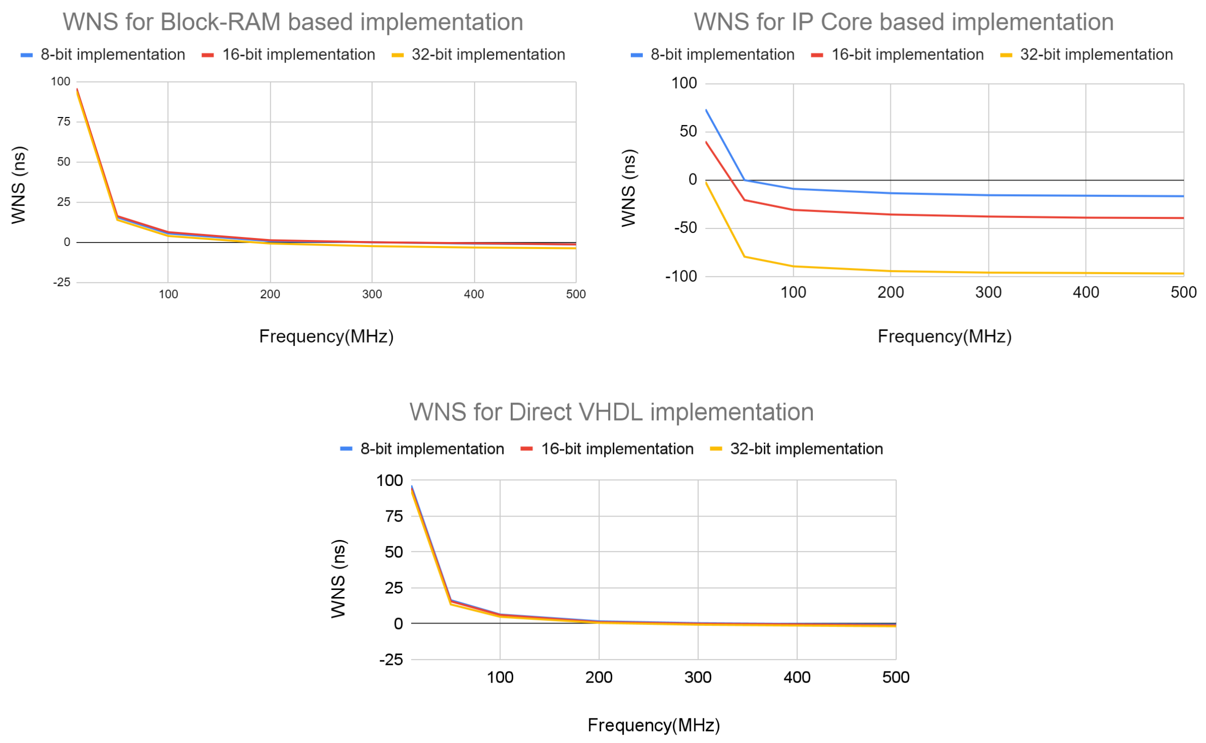


Figure 23: WNS vs. Operating Frequency graphs in three different approaches

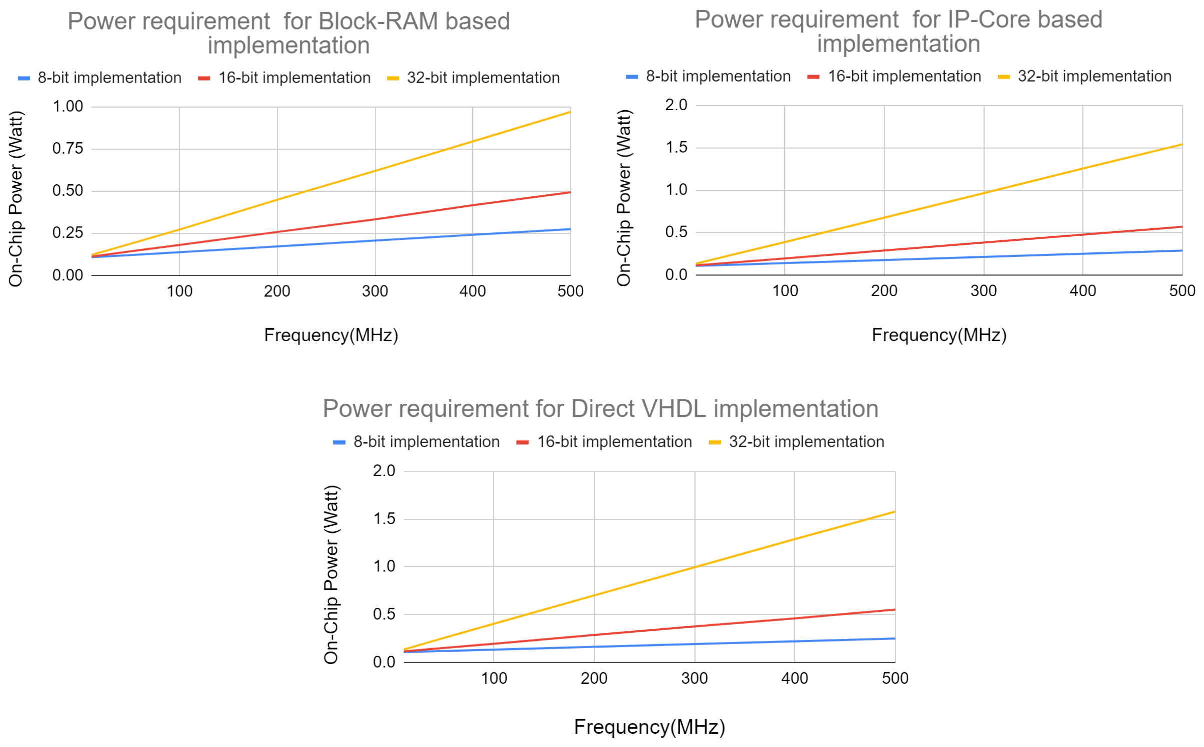


Figure 24: Power vs. Operating Frequency graphs in three different approaches

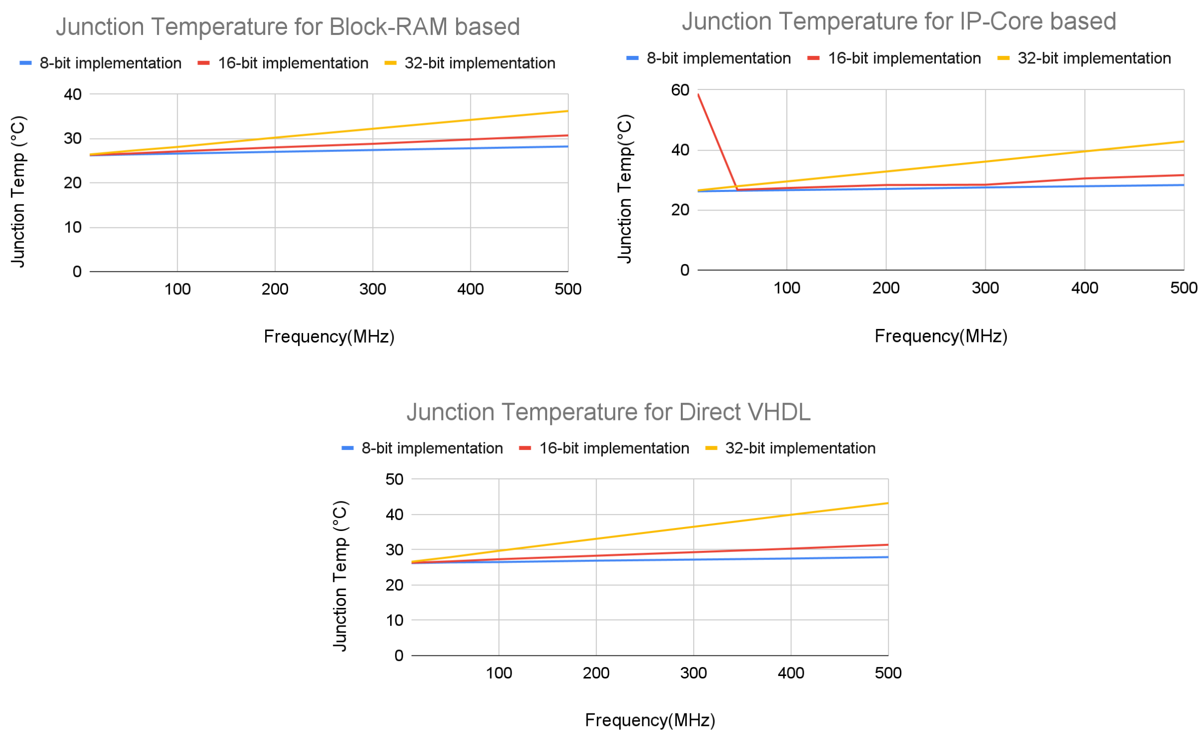
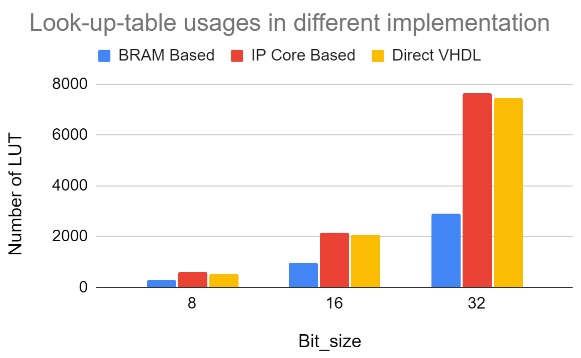


Figure 25: Temperature vs. Operating Frequency graphs in three different approaches



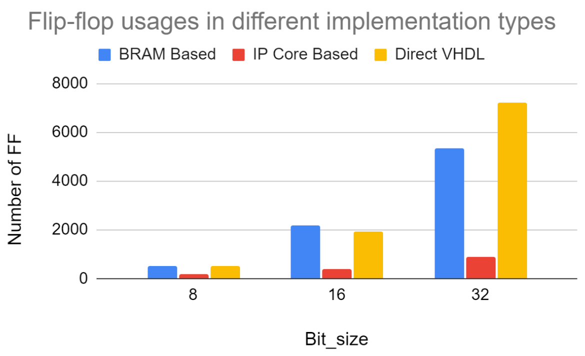
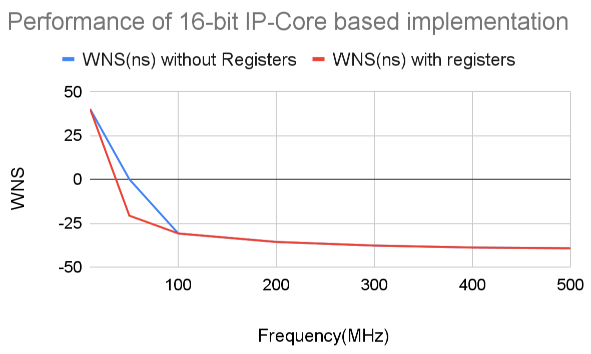


Figure 26: Resource Utilization graphs for three different approaches

The Performance difference between 16-bit IP-core-based implementation without spike removal registers and with spike removal registers:



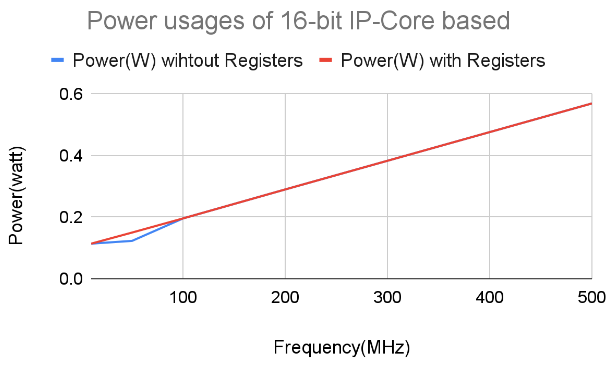


Figure 27: The Performance difference between 16-bit IP-core-based implementation without spike removal registers and with spike removal registers

Table 10: Maximum Operating frequency for three different implementation approaches.

Bitsize	Max_Freq_BRAM	Max_Freq_CORDIC_IP	Max_Freq_Direct_HVDL
8	300	50	300
16	300	10	200
32	100	10	200

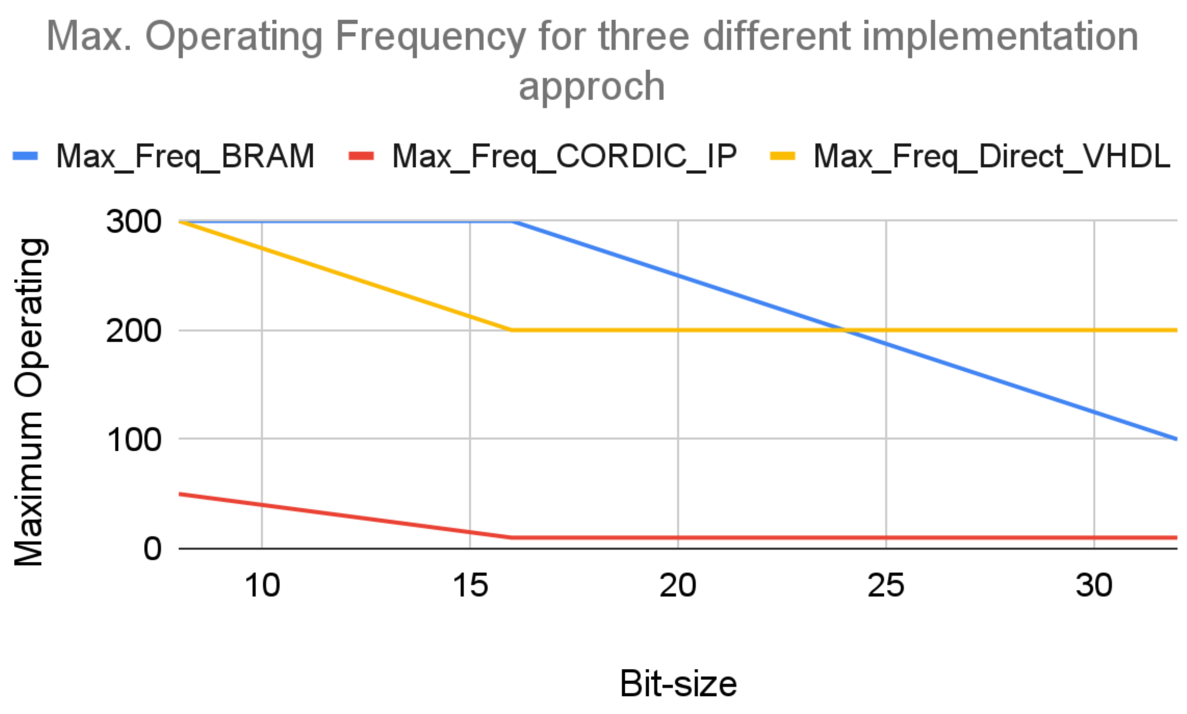


Figure 28: Maximum Operating frequency vs. bit-size graph for three different implementation approaches.

It is observed that direct VHDL implementation can run faster than other BRAM and CORDIC IP core-based implementations and It is also observed that 16-bit and 32-bit fixed point implementation runs on the same maximum operating frequency but 16-bit takes fewer resources.

Performance of 16-bit direct VHDL coding-based design with different Vivado Implementation Strategies:

Table 11: Implementation strategies report

	Implementation Strategy	Worst Negative Slack	On-Chip Power	Temperature	Look-up-table	Flip-flop
1	Vivado Implementation Defalut Strategy	1.108	0.287	28.3	2092	1944
2	Performance Explore	1.108	0.287	28.3	2092	1944
3	Performance_WLBlockPlacementFanoutOpt	1.108	0.287	28.3	2092	1944
4	Performance_netDelay_high	1.124	0.287	28.3	2092	1944
5	Performance_netDelay_low	1.078	0.286	28.3	2091	1944
6	Performance_retiming	1.108	0.287	28.3	2092	1944
7	Congestion_spreadLogic_high	1.076	0.288	28.3	2092	1944
8	Congestion_spreadLogic_low	1.065	0.286	28.3	2092	1944
9	Area_Explore	1.108	0.287	28.3	2092	1944
10	Power_ExploreArea	0.637	0.287	28.3	2093	1944
11	Flow_RuntimeOptimized	1.198	0.287	28.3	2114	1944
12	Flow_Quick	-0.346	0.287	28.3	2115	1944

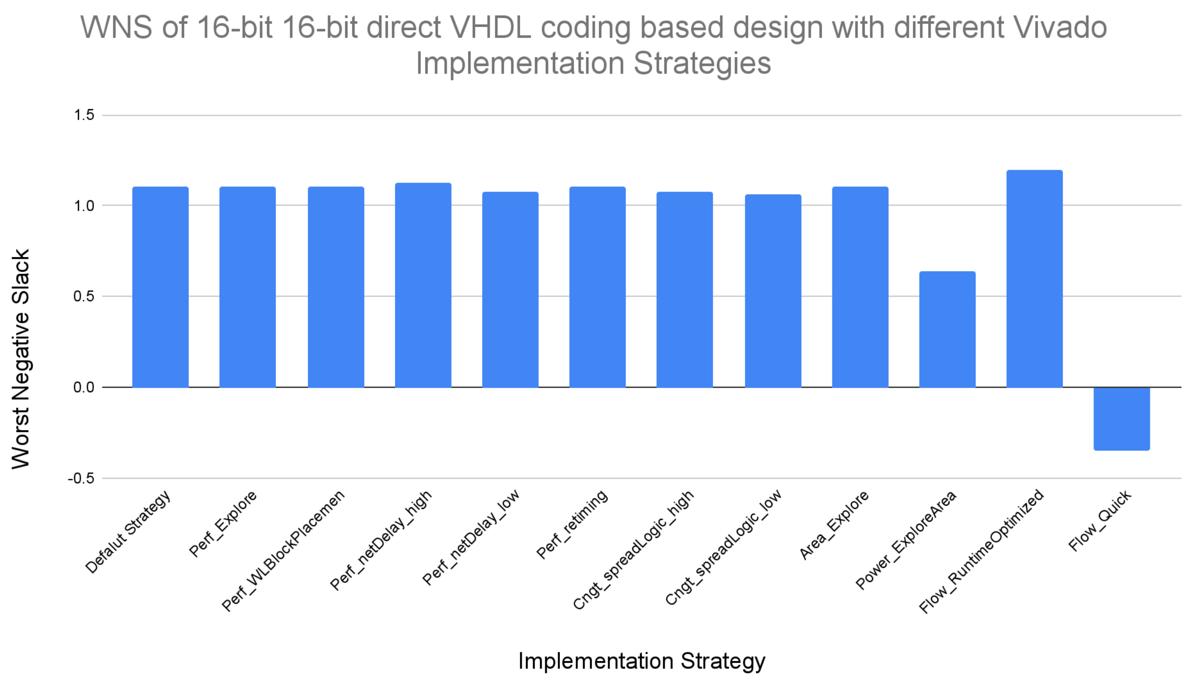


Figure 29: Worse Negative Slack (WNS) with different implementation strategies for 16-bit direct VHDL coding-based implementation.

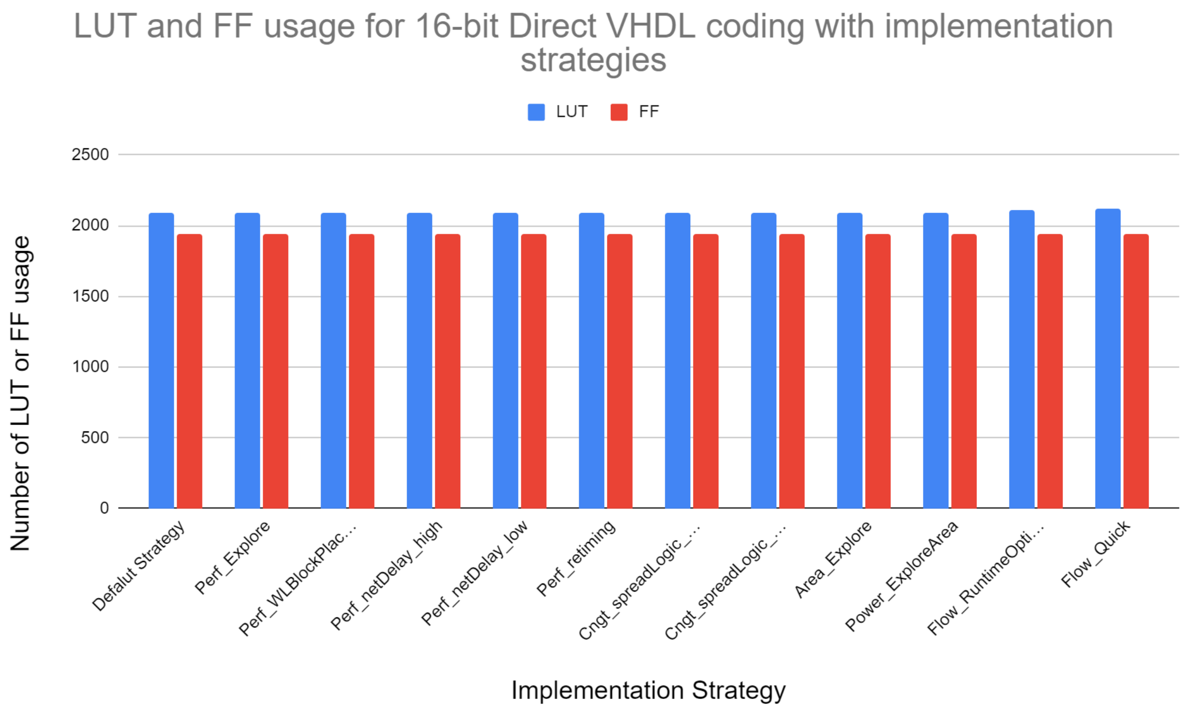


Figure 30: Look-up-table and flip-flop usage with different implementation strategies for 16-bit direct VHDL coding-based implementation.

Conclusion:

From the graphs and data it is observed to me that 16 and 32-bit direct VHDL coding-based implementation gives better accuracy in FPGA simulation output. On the other hand, direct VHDL coding-based 16-bit fixed-point implementation outperforms in all categories such as speed, resource and power. Therefore, for this project 16-bit fixed-point Direct VHDL implementation is a better choice. However, MATLAB Simulink-based design could be an alternative option for quick prototyping.

Reference:

1. Zhang, Lei. "Oscillation Patterns of A Complex Exponential Neural Network."