Object Detection and Classification using FMCW Radar

Dayakar Sathya Sai Jagata, Velagala Prashanth,

Inumala Vighnesh

TurboCompute, 9156

National Institute Of Technology , Surathkal

Table of Contents

Include the name of each section/subsection and the page number.

[I. Introduction 5](#_Toc199796912)

[II. Background Research 7](#_Toc199796913)

[1. Introduction 11](#_Toc199796914)

[2. Definition of Hanning Window 12](#_Toc199796915)

[3. Frequency Domain Characteristics 12](#_Toc199796916)

[III. Goal and Objectives 15](#_Toc199796917)

[IV. Design Process 16](#_Toc199796918)

[i. Refined Solution 16](#_Toc199796919)

[ii. Functional Specification 18](#_Toc199796920)

[iii. SoC Design (SoC level block diagram with interfaces/sensors used) 23](#_Toc199796921)

[iv. Accelerator Design Implementation 23](#_Toc199796922)

[v. Test Plan/Test cases 29](#_Toc199796923)

[vi. Simulation result of accelerator (along with waveforms) 34](#_Toc199796924)

[V. Results and Discussion 46](#_Toc199796925)

[VI. Conclusion 49](#_Toc199796926)

[VII. References 50](#_Toc199796927)

List of Tables

**Table –1 FMCW VS PULSED Radar.....................................9**

Table –2 AI Model Before vs After quantization......................47

List of Figures

Fig-1 FMCW WAVE..................................................................................................6

Fig –2 Sawtooth modulation.......................................................................................8

Fig-3 RD map.............................................................................................................9

Fig-4 Pulse vs CW vs FMCW waveforms.................................................................10

Fig-5 Hanning window...............................................................................................11

Fig-6 Hann window....................................................................................................13

Fig-7 Accelerator Design..........................................................................................16

Fig-8 Radix-4 FFT......................................................................................................20

Fig-9 Ping-Pong Buffer design...................................................................................21

Fig-10 SOC DESIG....................................................................................................25

Fig-11 Hanning Window and Multiplier......................................................................27

Fig-12 Hanning window (Hardware version)............................................../...........................27

Fig-13 Implementation of FFT and Dual buffer...............................................................28

Fig-14 FFT adapter................................................................................................................29

Fig-15 Configuration of fft and block ram..............................................................................29

Fig-16 Final accelerator.........................................................................................................31

Fig-17output Logger..............................................................................................................31

Fig-18 Frame of a Moving truck................................................................................32

Fig-19 RD-Map of moving truck................................................................................33

Fig-20 Moving Car and Pedestrian...........................................................................34

Fig-21RD-Map of moving car and pedestrian………………………………………….35

Fig-22 Matlab generated RD map.............................................................................40

Fig-23 Vivado generated RD map.............................................................................41

Fig-24 Time for matlab..............................................................................................42

Fig-25 RD maps stored in respective folders............................................................43

Fig -26One of the generated RD map for biscycle....................................................43

Fig-27 Labelling of RD maps.....................................................................................44

Fig-28 Architecture of CNN.......................................................................................45

Fig –29 Model summary............................................................................................46

Fig-30 Input RD map Generated from Vivado for Testing.....................................48

Fig-31 Output response..........................................................................................48

# Introduction

FMCW radar gives us an effective, low-power way to measure range and velocity. Techniques like windowing ensure clean signal processing. With real-world data from the IEEE dataset, we can validate our hardware pipeline and train accurate AI models. And with CNNs, we can turn radar data into high-level scene understanding—giving vehicles the ability to not only detect but also understand what’s around them.

In today’s automotive world, radar sensors are a key part of advanced driver-assistance systems (ADAS) and autonomous vehicles. Among the different radar technologies, **Frequency-Modulated Continuous-Wave (FMCW) radar** is widely used because it can simultaneously measure both the distance to an object and how fast that object is moving — which is critical for real-time decision-making on the road.

FMCW radars transmit a signal known as a *chirp*, whose frequency increases linearly over time. When this chirp reflects off an object and returns to the radar, the time delay and frequency shift between the transmitted and received signals can be used to estimate both range and velocity. This is achieved through two main stages of processing: the **Range FFT**, which analyzes individual chirps to extract distance, and the **Doppler FFT**, which looks across multiple chirps to determine velocity.

However, radars operating at high frequencies like **77 GHz** generate a large volume of data. Processing this data in real time is a significant challenge, especially with conventional processors that may not meet the strict timing requirements of automotive applications.

**Significance of Our Work**

To address this, we designed and implemented a **hardware-accelerated radar processing pipeline** on an FPGA. Our system includes a **dual-buffer architecture** that enables continuous, real-time data handling between the two FFT stages. We tested our design using the **IEEE 77 GHz radar dataset**, which provided realistic automotive scenarios. In addition, we designed the system to support integration with an **AI model** for post-processing tasks like object classification.

What sets our work apart is its focus on **practicality and performance**. By offloading key processing stages to hardware, we achieved low-latency, high-throughput operation suitable for real-world automotive radar platforms. This makes our solution a strong candidate for future applications in autonomous driving and advanced safety systems

# Background Research

**A. Fundamentals of FMCW Radar**

Radar systems help vehicles detect and track objects around them by sending out electromagnetic waves and analyzing the reflections. One of the most widely used radar types in automotive applications is the **Frequency-Modulated Continuous-Wave (FMCW) radar**. It’s known for its efficiency, precision, and ability to measure both how far away an object is (range) and how fast it's moving (velocity)—all while consuming less power than traditional radar systems.

FMCW radar transmits a signal called a **chirp**, which increases in frequency over time:

( f(t)=fo+μt for 0 ≤ t ≤Tc )

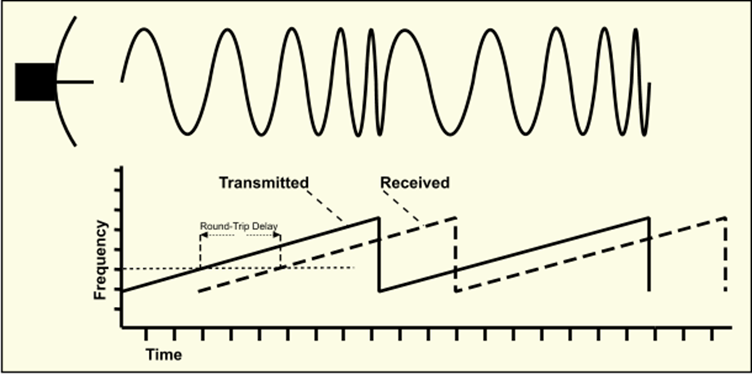


Fig-1 FMCW WAVE

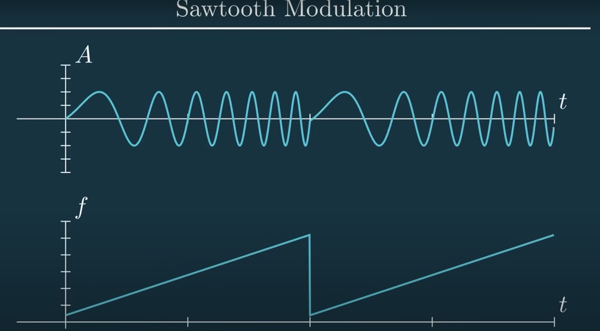


Fig –2 Sawtooth modulation

Where

* fo is the starting frequency,
* μ=B/Tc is the slope, with as the bandwidth and as the chirp duration.

When the chirp hits a target and reflects back, it arrives slightly delayed:

τ = 2R/c

By comparing the received signal to the transmitted one (a process called mixing), the radar extracts a **beat frequency**, which reveals how far away the object is. By analyzing multiple chirps, it also identifies **Doppler shifts** to calculate how fast the object is moving. These calculations are carried out using:

* A **Range FFT** to estimate distance,
* A **Doppler FFT** to determine relative speed.

The result is a **Range-Doppler Map (RD Map)**—a visual grid showing where objects are and how fast they’re moving.

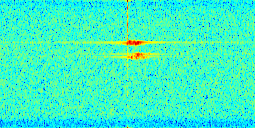


Fig-3 RD map

**B. FMCW Radar vs. Pulsed Radar**

|  |  |  |
| --- | --- | --- |
| **Feature** | **FMCW Radar** | **Pulsed Radar** |
| Waveform | Continuous chirp | Short high-power pulse |
| Power consumption | Low | High |
| Range resolution | High (with large bandwidth) | Depends on pulse width |
| Velocity estimation | Simultaneous with range (via FFT) | Requires pulse Doppler processing |
| Real-time use | Ideal for automotive systems | Common in aviation/military |

**Table –1 FMCW VS PULSED Radar**

FMCW radar is a better fit for automotive systems because it operates continuously, consumes less power, and extracts both range and velocity in real time using compact and affordable hardware.

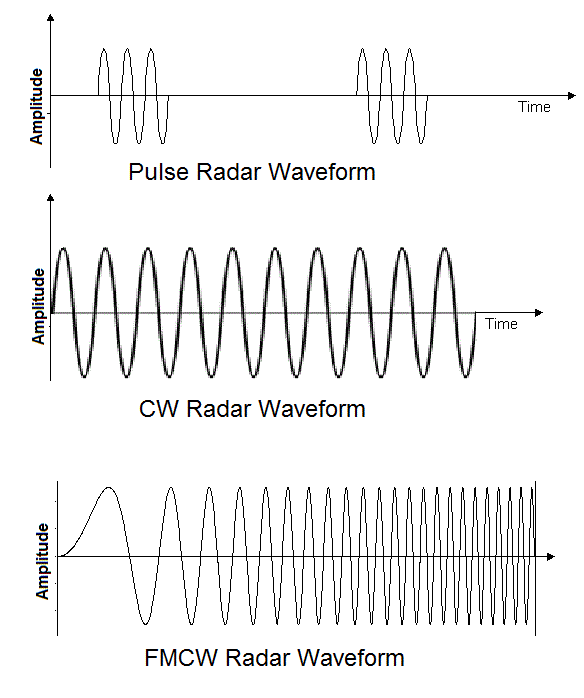


Fig-4 Pulse vs CW vs FMCW waveforms

**C. Spectral Leakage and Windowing**

When we use FFTs to analyze signals, we often assume the signal fits neatly into the sampled window. But in reality, this rarely happens, and the result is **spectral leakage**—where energy from one frequency spills into others, blurring the result.

To reduce this, we apply a **windowing function**—a mathematical shape that tapers the signal’s edges to zero. One common example is the **Hanning window**:

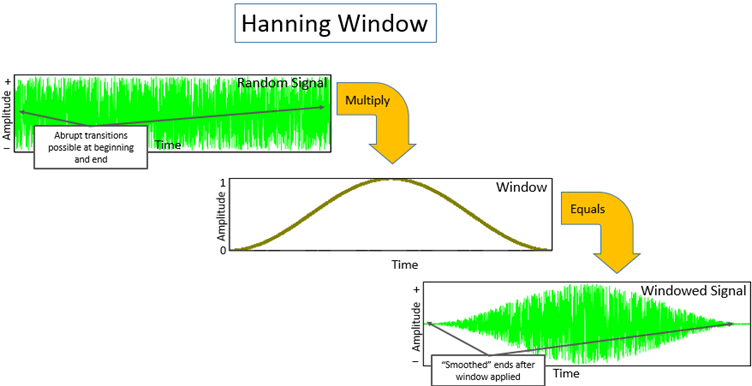


Fig –5 Hanning window

# 1. Introduction

In digital signal processing, especially in applications involving spectral analysis using the Fast Fourier Transform (FFT), windowing functions are applied to minimize spectral leakage caused by the discontinuities at the signal boundaries. One such commonly used window is the Hanning window (also called the Hann window).

# 2. Definition of Hanning Window

The Hanning window is a tapered window that smoothly reduces the amplitude of the signal at the beginning and end of the sampling period. It is defined by the following equation for a window of length N:  
   
 w[n] = 0.5 \* (1 - cos(2πn / (N - 1))), 0 ≤ n ≤ N-1  
   
This window function has a bell-shaped curve, peaking in the middle and tapering to zero at the edges.

# 3. Frequency Domain Characteristics

The application of the Hanning window modifies the frequency spectrum of the input signal:  
 - Main lobe: Wider than the rectangular window, leading to slightly reduced frequency resolution.  
 - Side lobes: Much lower (approximately -31 dB), which significantly reduces spectral leakage.  
   
This makes the Hanning window an excellent tradeoff between main lobe width and side lobe suppression.

Windowing helps us get clearer, more accurate FFT results—especially when trying to separate closely spaced objects in the radar scene.

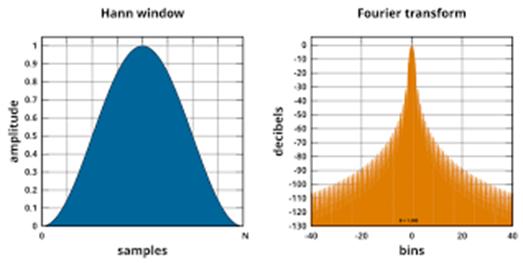


Fig-6 Hann window

**D. IEEE 77 GHz Automotive Radar Dataset**

For our project, we used the **IEEE 77 GHz radar dataset** [1], which contains real radar signals captured from a high-resolution automotive FMCW radar system. This dataset includes scenes with:

* Moving and parked vehicles,
* Pedestrians, cyclists, and other road users,
* Real-world conditions like occlusions and traffic clutter.

The dataset gives us:

* Raw I/Q ADC samples from multiple chirps,
* Rich environments for testing signal processing pipelines,
* Ground truth for training AI models to classify RD maps.

It provides a strong foundation for validating both the radar pipeline and the AI inference layer.

**E. CNNs for RD Map Interpretation**

The **Range-Doppler Map** is essentially a 2D image showing object intensity by distance and speed. Naturally, we looked to **Convolutional Neural Networks (CNNs)**—the same technology used in image recognition—for interpreting these maps.

CNNs are especially well-suited for this task because they:

* Automatically detect patterns and features in 2D data,
* Work regardless of where the object appears in the map (translation invariance),
* Learn and generalize better than manual feature extractors.

Compared to traditional classifiers like SVMs or decision trees, CNNs are more scalable, better at handling complexity, and easier to optimize for embedded platforms using frameworks like TensorFlow Lite.

**F. Summary**

FMCW radar gives us an effective, low-power way to measure range and velocity. Techniques like windowing ensure clean signal processing. With real-world data from the IEEE dataset, we can validate our hardware pipeline and train accurate AI models. And with CNNs, we can turn radar data into high-level scene understanding—giving vehicles the ability to not only detect but also understand what’s around them.

# Goal and Objectives

**Goal**

The goal of this project is to build a real-time radar signal processing system designed for automotive applications, using an FMCW radar. The system uses an FPGA to handle heavy-lifting tasks like Range and Doppler FFTs, producing 128×256 Range-Doppler maps in real time through a dual-buffer ping-pong architecture. These RD maps are stored in DDR memory and passed to a VEGA processor, which runs a machine learning model trained on real-world 77 GHz automotive radar data. The aim is to combine fast hardware-based processing with smart AI-based object classification, enabling low-latency decision-making that’s suitable for edge devices in autonomous vehicles.

**Objectives**

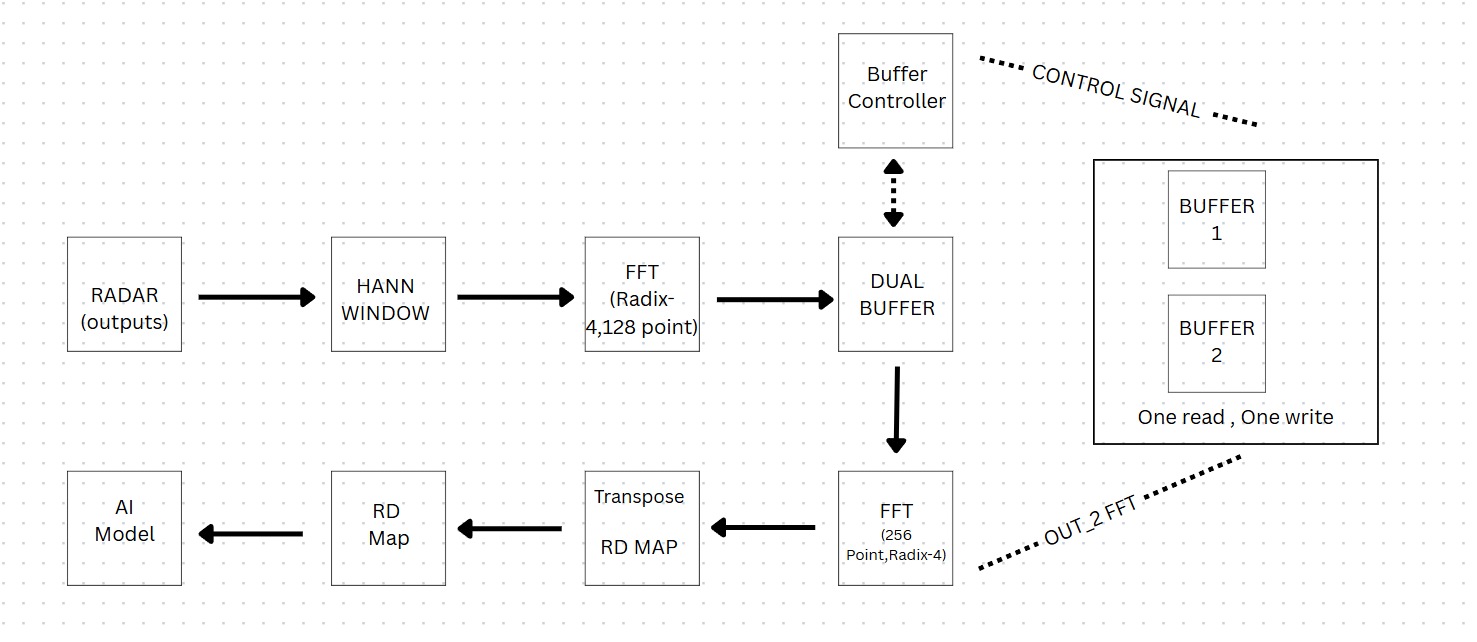
* To find and use a high-quality FMCW radar dataset that includes different types of objects like bicycles, pedestrians, cars, trucks, and combinations of them.
* To convert the raw radar data into clear and reliable Range-Doppler maps using signal processing techniques.
* To enhance and clean up the RD maps so they’re well-suited for training a machine learning model.
* To develop a custom hardware accelerator on FPGA that speeds up the radar signal processing tasks.
* To design and train a machine learning model that can accurately classify objects based on the RD maps.
* To test the full system’s performance by measuring how fast it works (latency) and how much data it can handle (throughput).

# Design Process

## Refined Solution

The proposed solution for object detection and classification is to develop a custom accelerator on the PL (Programmable Logic) side of the FPGA to handle the radar signal processing tasks efficiently. This accelerator leverages Xilinx's built-in FFT IP cores, on-chip BRAM, and our custom-designed ping-pong buffer with control logic to enable seamless execution of both Range FFT and Doppler FFT operations in a pipelined fashion. The resulting Range-Doppler (RD) maps are written to DDR memory, which is then accessed by the VEGA processor on the PS (Processing System) side. A machine learning model running on the VEGA processor processes these RD maps and classifies the detected objects based on the learned features. This hardware-software co-design approach ensures real-time performance and low-latency inference, making it suitable for edge deployment in automotive radar systems.

Fig-7 Accelerator Design



**The figure above illustrates the block diagram of our custom accelerator IP.** This architecture is fully pipelined and designed to handle continuous streaming of radar data in real time. It efficiently processes input through a sequence of signal processing stages including windowing, FFT, and dual-buffered data management. The design supports seamless handoff between Range FFT and Doppler FFT operations using a ping-pong buffer system, ensuring uninterrupted data flow. Further details regarding the hardware modules, control logic, and implementation specifics will be covered in the Accelerator Implementation section. Simulation results and functional validation will be presented in the subsequent Working and Results section.

In our solution, **Fast Fourier Transforms (FFT)** play a crucial role in converting the radar’s time-domain signals into the frequency domain, which is necessary to extract both range and Doppler (velocity) information. The first stage applies a **128-point FFT** on each chirp to determine object distances (Range FFT), and the second stage applies a **256-point FFT** across multiple chirps for each range bin to extract relative velocity (Doppler FFT). Together, these two stages generate the **Range-Doppler (RD) map**, which provides a 2D spatial-velocity representation of the scene — a critical input for object detection and classification.

Unlike a general-purpose **CPU**, which performs FFT computations sequentially and is limited by instruction cycles and memory bottlenecks, our solution **offloads FFT computation to the FPGA**. The FPGA leverages **dedicated DSP slices and pipelined FFT IP cores**, enabling **massively parallel processing** of incoming data streams with extremely **low latency and high throughput**. Additionally, the use of **on-chip BRAM for buffering** avoids the delays associated with accessing external DDR memory. This hardware acceleration ensures that the system can meet **real-time performance requirements**, making it highly suitable for edge applications like autonomous driving — where every millisecond matters.

## Functional Specification

This section we would like to explain each block from the earlier figure as clear as possible.

Simultaneously we will also explain the part how and where we are actually accelerating. This would be a rough theoretical comparison between a CPU and our accelerator IP implemented on FPGA.

#### **1) Radar Signal Acquisition – I/Q Data Input**

The system receives **complex I/Q data** from an FMCW radar front-end. This data represents the digitized version of beat signals captured from reflected radar chirps. Each chirp produces a frame of I (in-phase) and Q (quadrature) samples, typically captured as **16-bit signed integers**.

* **Interface:** The I/Q data is streamed into the FPGA via high-speed input (e.g., LVDS, SPI, or from a memory-mapped interface depending on the radar board).
* **Data Format:** Each sample pair consists of a signed 16-bit I value and a signed 16-bit Q value, combined into a 32-bit word.
* **Data Rate:** The system is designed to handle **128 samples per chirp**, and **256 chirps per frame**, resulting in **128×256 = 32,768 complex samples per RD map**.

Acceleration insight :

Unlike a CPU that would have to store this data in RAM and iterate through it for processing, the FPGA streams this data directly into processing blocks in a **clock-by-clock pipelined fashion**, eliminating overhead.

**2) Windowing – Hann Window**

Each incoming I/Q sample is multiplied by a **Hann window coefficient** to reduce spectral leakage before FFT. The coefficients are stored in BRAM, and the multiplication is done using a **pipelined multiplier block with a latency of 4 cycles**.

* **Input:** 32-bit complex sample (16-bit I, 16-bit Q)
* **Operation:** I and Q are each multiplied by a 16-bit fixed-point Hann coefficient
* **Latency:** 4 clock cycles (due to pipelined multiplier)
* **Throughput:** 1 sample per cycle after initial delay
* **Output:** 32-bit windowed complex sample

Acceleration insight :

CPUs process windowing sequentially. Our pipelined hardware performs windowing continuously, one sample per cycle, enabling real-time processing with minimal delay. Not much but a good fit !

3) RANGE and DOPPLER -FFT

* The system uses two stages of FFT to convert raw radar signals into the frequency domain:
* **Range FFT (128-point):** Applied across time samples of each chirp to determine object distances.
* **Doppler FFT (256-point):** Applied across chirps for each range bin (after transpose) to extract object velocities.
* Both FFTs are implemented using **Radix-4 streaming FFT IP cores**, optimized for high-speed operation. The architecture uses a **4-multiplier structure** to parallelize key butterfly computations, enabling faster FFT execution and reducing the number of pipeline stages compared to Radix-2 designs.
* **Input/Output Format:** 32-bit complex values (16-bit signed real and imaginary)
* **Architecture:** Radix-4, fully pipelined
* **Multipliers:** 4 parallel pipelined multipliers (used inside butterfly units)
* **Mode:** Streaming, accepting one sample per clock cycle
* **Latency:** Determined by FFT size and pipeline depth; output starts after initial pipeline fill
* **Transpose between FFTs:** Performed using dual-buffer BRAM to rearrange data from row-wise (range FFT) to column-wise (doppler FFT)

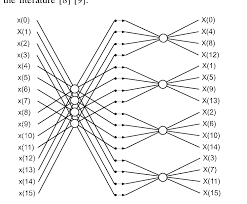


Fig-8 Radix-4 FFT

Acceleration Insight:

On a CPU, FFTs are processed sequentially using software libraries and temporary memory buffers, which introduces delays and limits throughput. In contrast, our FPGA-based design uses **dedicated FFT IP cores** with **4 parallel multipliers** and deep pipelining, enabling continuous, high-speed processing of complex samples with **low and predictable latency** — ideal for real-time radar applications.

4 ) Ping-Pong Dual buffer , BRAM

To maintain continuous streaming between the Range FFT and Doppler FFT, we use a **ping-pong dual-buffer architecture** implemented using **on-chip BRAM**. This allows one buffer to be written while the other is read, enabling seamless, non-blocking data flow.

* **Structure:** Two BRAM-based buffers (Buffer A and Buffer B)
* **Control:** A **Buffer Controller FSM** manages switching between read and write modes based on sample count
* **Operation:**
  + While Buffer A is filled with Range FFT output, Buffer B is read by the Doppler FFT (and vice versa)
  + Each buffer holds 128 × 256 complex samples (sized for full RD map data)
* **Access:** Single-port BRAM access, time-multiplexed for read/write
* **Data Width:** 32-bit (16-bit real + 16-bit imaginary)

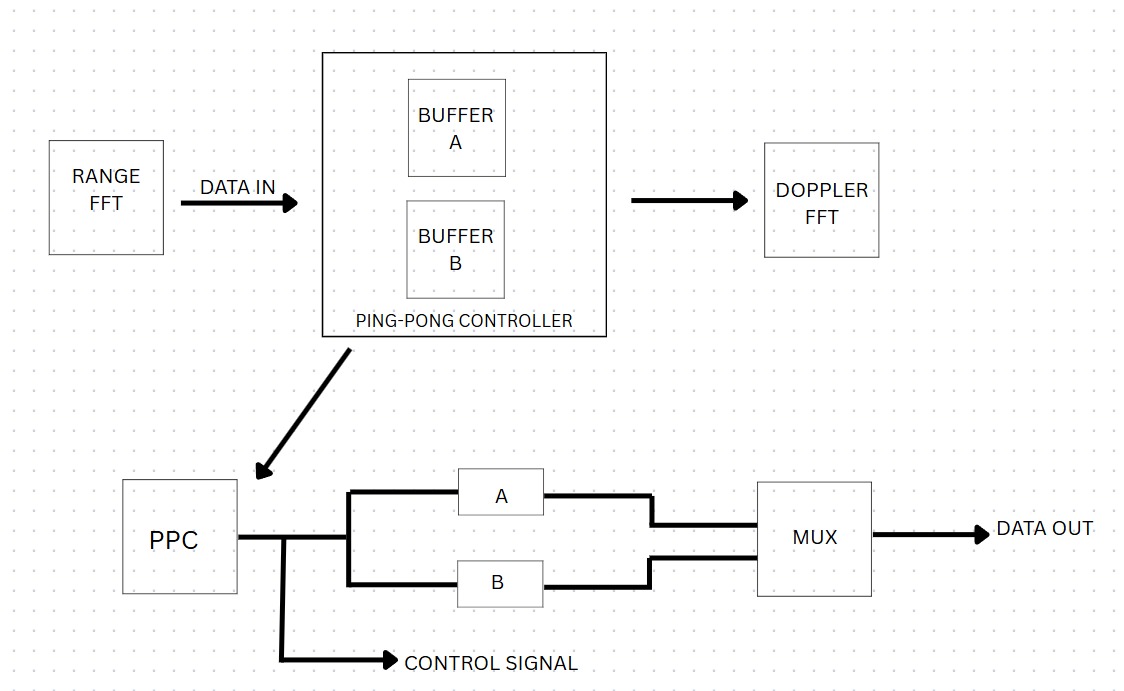


Fig-9 Ping-Pong Buffer design

Acceleration Insight:

On CPUs, buffering often depends on external RAM, which adds delay and needs complex synchronization. FPGA BRAM offers **low-latency, high-bandwidth access**, and the FSM-based ping-pong logic enables **tight control and real-time switching** with zero-copy efficiency.

5 ) ML MODEL

**As part of the project, a machine learning model was developed to classify objects based on the Range-Doppler (RD) maps generated by the simulated radar pipeline. While the signal processing portion was implemented and tested in Vivado through simulation, the classification task was carried out offline using Python.**

**The RD maps, each of size 128×256, were exported in hexadecimal format after simulation. These were converted into 2D matrix form and normalized to make them suitable for neural network input.**

**To train the model, a publicly available dataset containing labeled RD maps from 77 GHz automotive radar sensors was used. The dataset included a variety of object classes such as cars, pedestrians, and cyclists. Preprocessing steps such as scaling, reshaping, and class balancing were applied.**

**A lightweight convolutional neural network (CNN) was designed to classify the RD maps. The model was trained using the Adam optimizer and categorical cross-entropy loss, with training, validation, and test splits.**

**Once trained, the model was used to classify RD maps generated from the Vivado simulation. This allowed us to verify that the simulated pipeline produced maps of sufficient quality to support downstream AI-based object detection, demonstrating the system's practical viability for automotive applications.**

## SoC Design (SoC level block diagram with interfaces/sensors used)

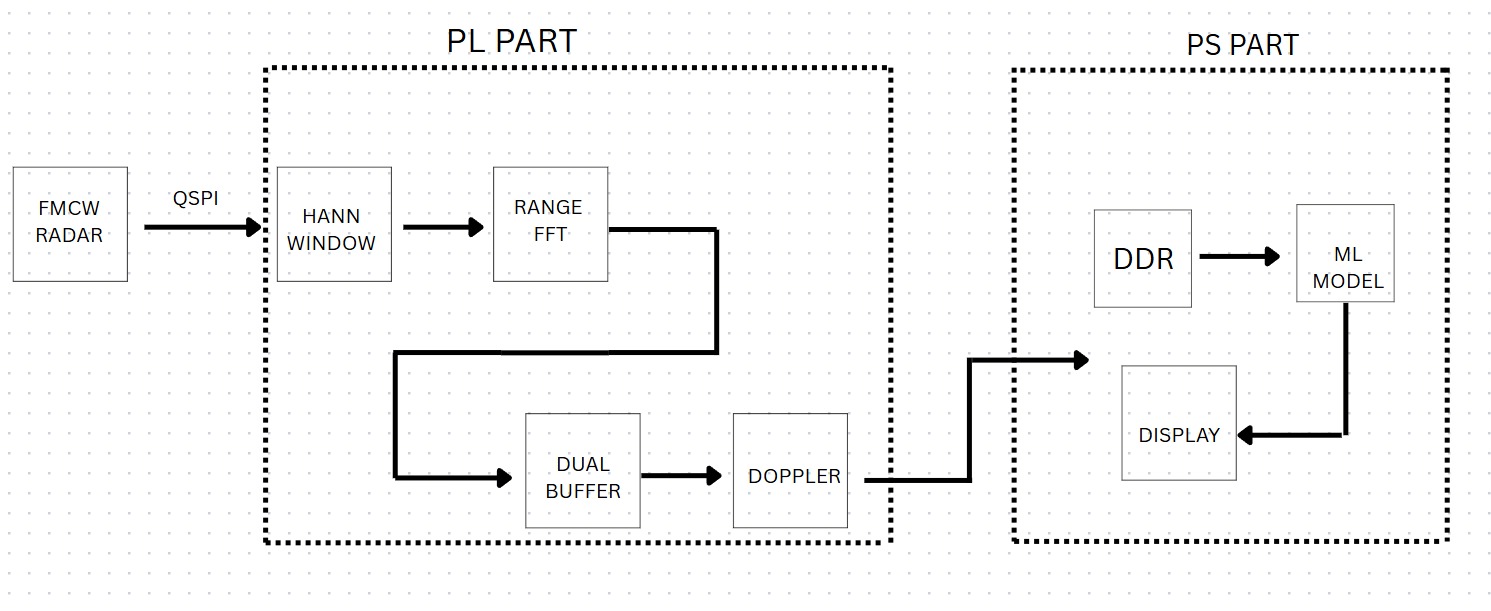


Fig-10 SOC DESIGN

## Accelerator Design Implementation

Every block has been implemented and simulated on vivado.

Hanning Window

The hanning window has been implemented by pre-loading weights into mem file and adding it to simulation sources.

Note- this is only for simulation and is not hardware compatible. We only used this in early stages of our simulation to keep our simulation results more clean and understandable.

We have also implemented the hardware compatible version by using block memory generator(ROM).

Implementation of both will be shown here and the waveforms to be attached in the later section.

We have got the hanning window coefficients using a matlab code , where we generated the coefficients and we have scaled to 32768. The usual coefficients are in the range of [-1 , 1] ,

But to make sure it works with our q15 format or signed 2’s complement we have scaled the coefficients.

After multiplication we scale back by

* Taking [30:15] of result if q15 format
* By taking [31:16] if its signed 2’s complement

For the multiplication we have used the multiplier IP and set it at optimum 4 pipeline stage. It is also configured for a faster output.

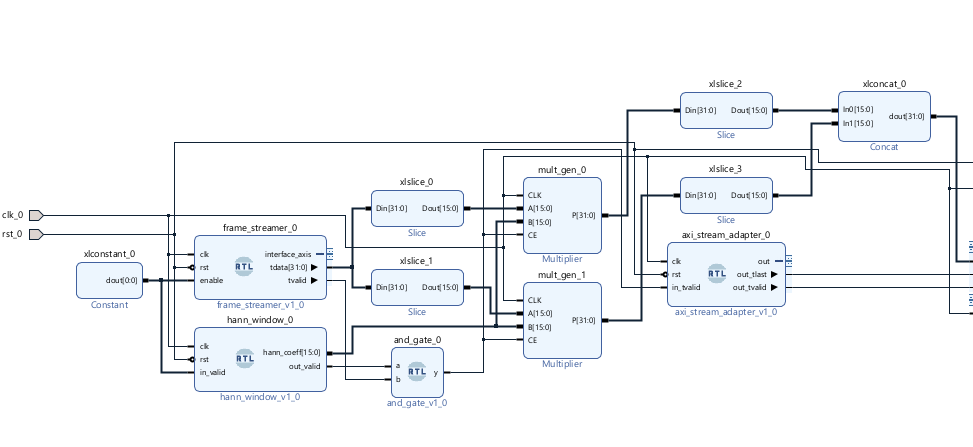


Fig-11 Hanning Window and Multiplier

In the above figure we have used the frame streamer to stream input from a text file. For the actual implementation we will directly stream from the fmcw radar.

Here is the hardware compatible version.

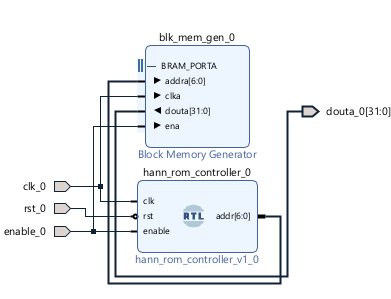


Fig-12 Hanning window (Hardware version)

This will replace the hann\_window module from the first figure in the subtopic.  
then it will be hardware compatible as we will use bram to preload the coefficients.

Both the waveforms of both version to be attached later.  
Note : clock will be connected to a clocking wizard during hardware implementation.

All the codes , mem files and .coe files to be added to our github.   
(github to be linked at the end.)

Range-Doppler FFT , ping-pong buffer

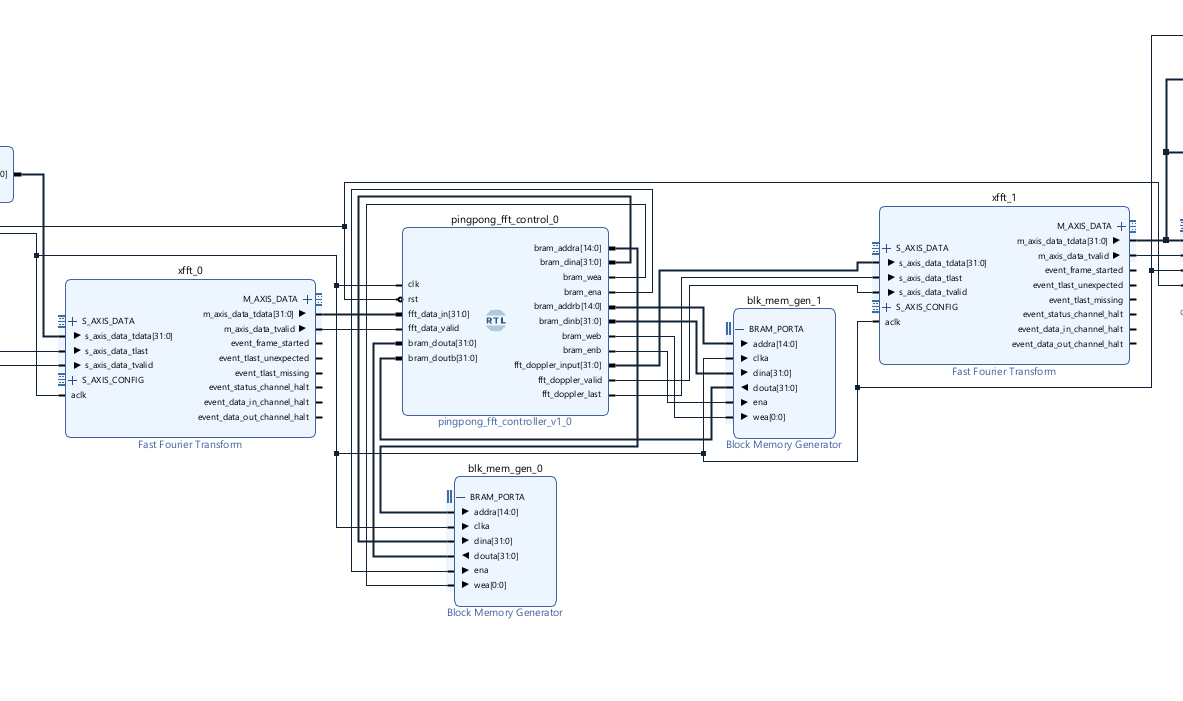


Fig-13 Implementation of FFT and Dual buffer

The above figure show the usage of FFT- IP matrix along with our custom built ping-pong buffer to support our design.

The data from FFT is driven into buffer , and once done its given as input to the doppler FFT (256 point).

The working of the ping-pong buffer can be clearly understood in the simulation section where we carefully show its both read and write mode to a buffer.

For the range fft we have implemented a adapter to make it compatible with the earlier hann window.

All the modules are hardware compatible have been successfully synthesized

Even the axi\_stream adapter is hardware compatible!

The adapter is given below.

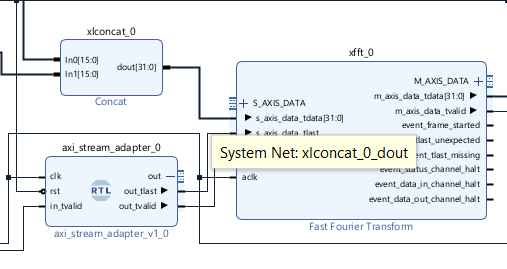
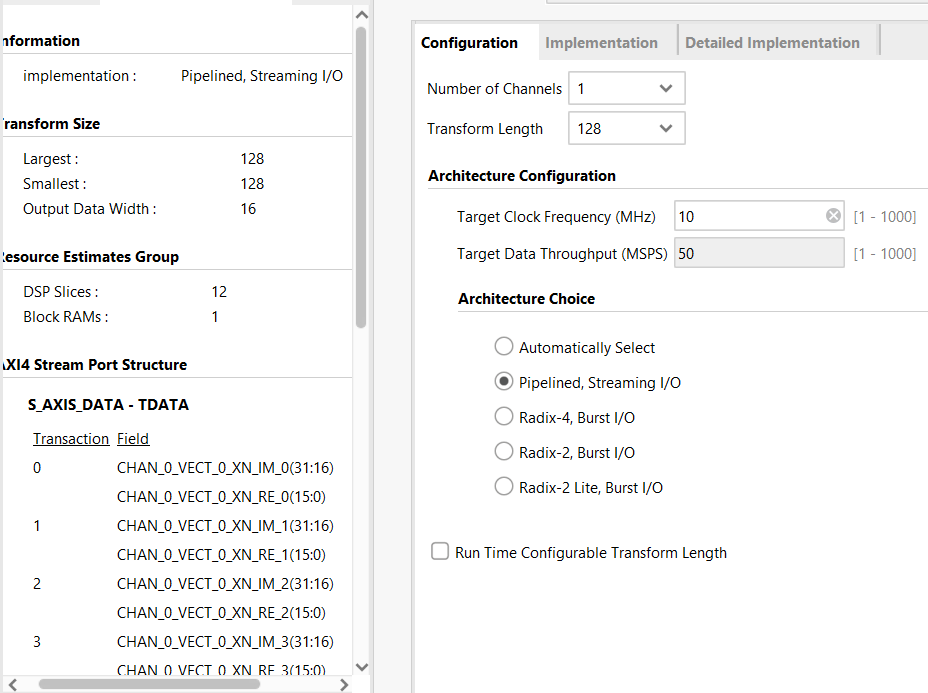
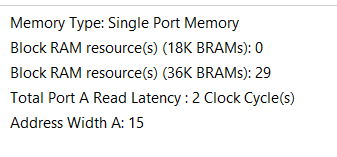


Fig-14 FFT adapter

Down below we display the ip customization for both FFT ip and the buffer.

Fig-15 Configuration of fft and block ram





All the codes to ping-pong buffer and adapter will be uploaded to git.

Finally we get the RD-maps as a stream of data from doppler FFT as 256 doppler bins \* 128 range doppler.

This is actually the transpose of the rd map and we need to build transpose before feeding into the processor,. This is optional to do in vivado and can be easily done on processor , so left to be done on processor.

Our final accelerator is show below

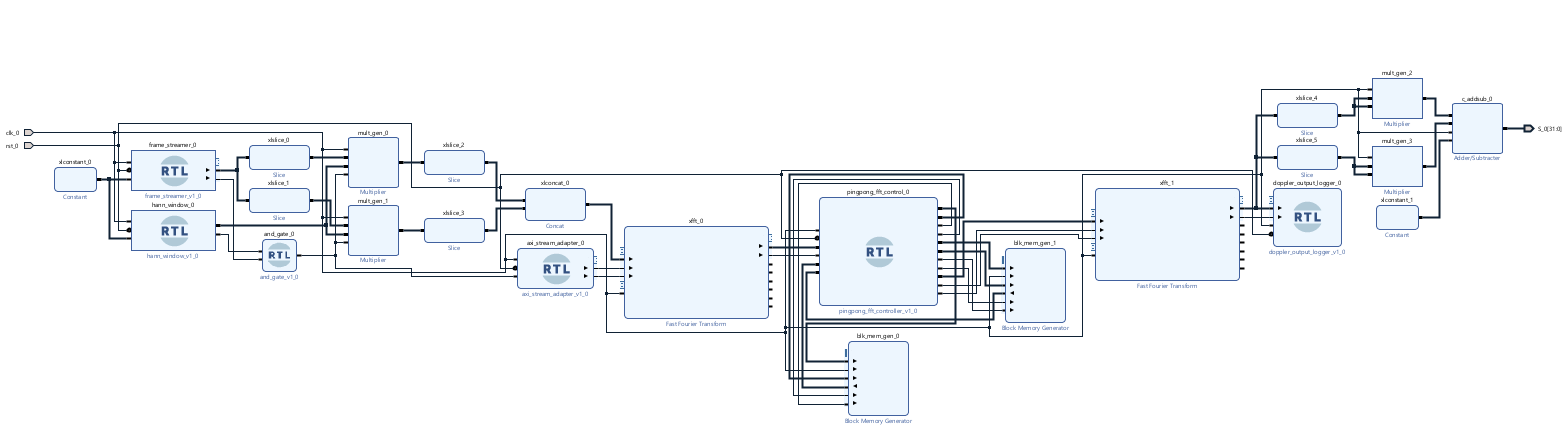
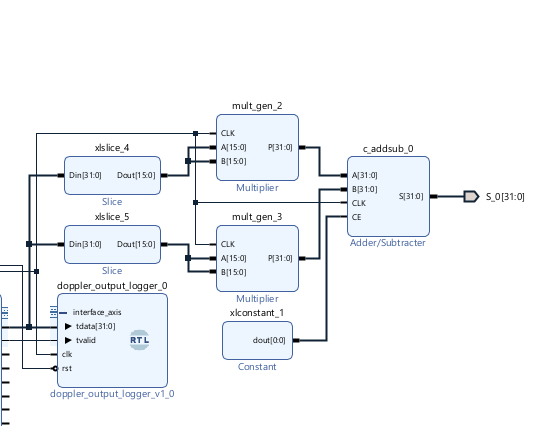


Fig-16 Final accelerator Design

The last part is to save the output and compare with our rd map generated from matlab and to test it against our model.

It also has the magnitude using square and add to get the absolute value.

Fig-17output Logger



## Test Plan/Test cases

From the fcmw radar , the data logged is 22 different sets of cases(car , pedestrian , truck etc).

Each contains 900 frames , and each frame has 256 chirps. Each chirp has 128 samples.  
so we have a mat file associated with each frame that has 128\*256 samples (32768 samples)

So we extracted this data and processed and saved into text file so that we can feed into it our accelerator IP.

This radar works on 10mhz clock and that is why our whole accelerator Ip was designed to work on the 10mhz clock

Further we used matlab to plot the rd map so that we can compare against our hardware generated rd map.

All matlab codes to be uploaded to git as well.

Now lets take a frame of a truck moving towards us.

Fig-18 Frame of a Moving truck



In the above picture we can see that the truck is relatively moving towards us.

Now lets see if the rd map tells us the same.

The below is the Rd map generated from matlab.

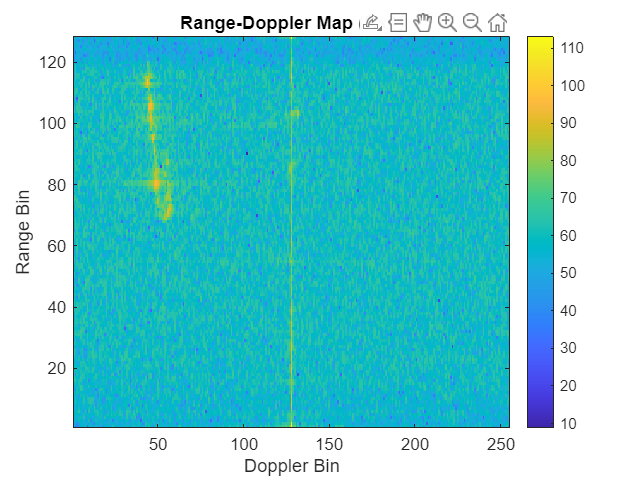


Fig-19 RD-Map of moving truck

We can observe two things , along the middle we can see a yellow line and we can see a bright yellow on the left part.

Yellow part gives us beat frequncy or to put simply tells the presence of an object.

Yellow part in the middle tells that the objects are stationary and basically are clutter. If it lies in the middle of doppler bin it has zero velocity , and if it lies to left the object is coming towards us and if to the right away form us.

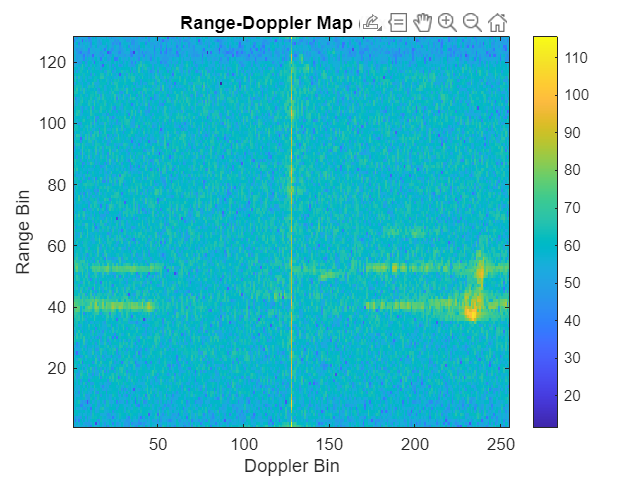
Comparing with another scenario,



Fig-20 Moving Car and Pedestrian

A car going away from us at high speed ( if high speed it remains to the extremes)

And a man but he is going sideways , he will be considered as 0 velocity because hardly any relative velocity towards to radar.

 Fig-21

Thus we can also see that for different moving object we can see different amount of reflection back too. For truck it was a bit more yellow spots and for car is relatively smaller and even smaller for pedestrian.

We will use this property to train our models and make the fmcw radar as smart radar ai powered radar!

Now coming back to our testing , as we mentioned earlier we converted all the data from mat file to a txt file and which will be streamer to accelerator ip via the streamer module.

The output to be logged into a txt file.

After the simulation we can take data from the txt file and plot it as rd map without any signal processing.

Thus we can compare how accurate our results were! And how well our module is working to make appropriate changes and updates to our IP.

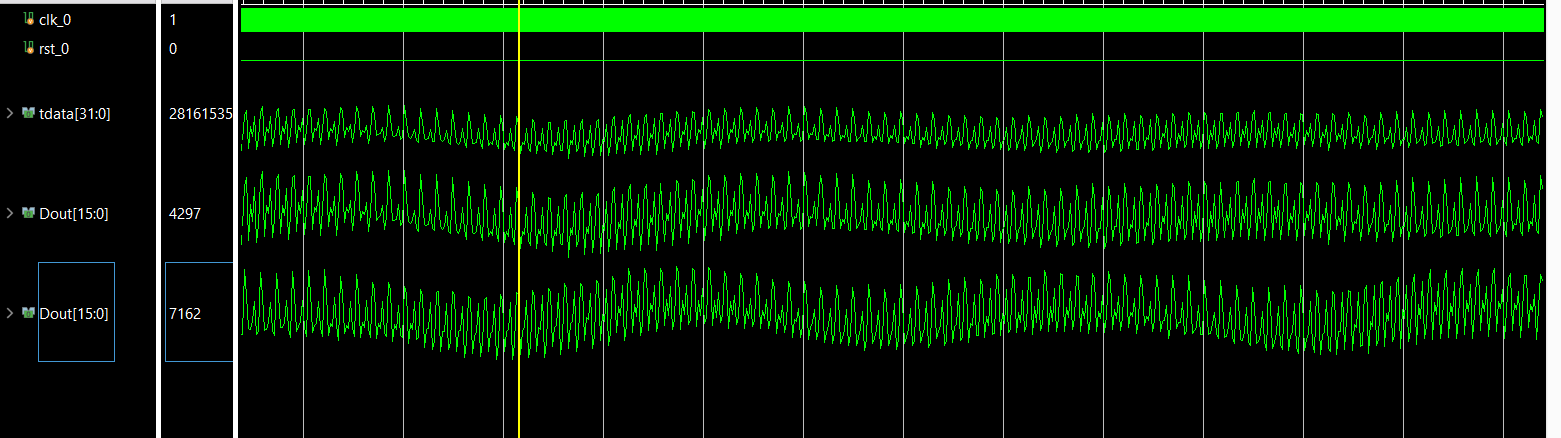
The rd map generated is also tested against our very own trained ai model to officially complete the whole simulation.

The text file and code to exctract data from matlab is also uploaded to our git!

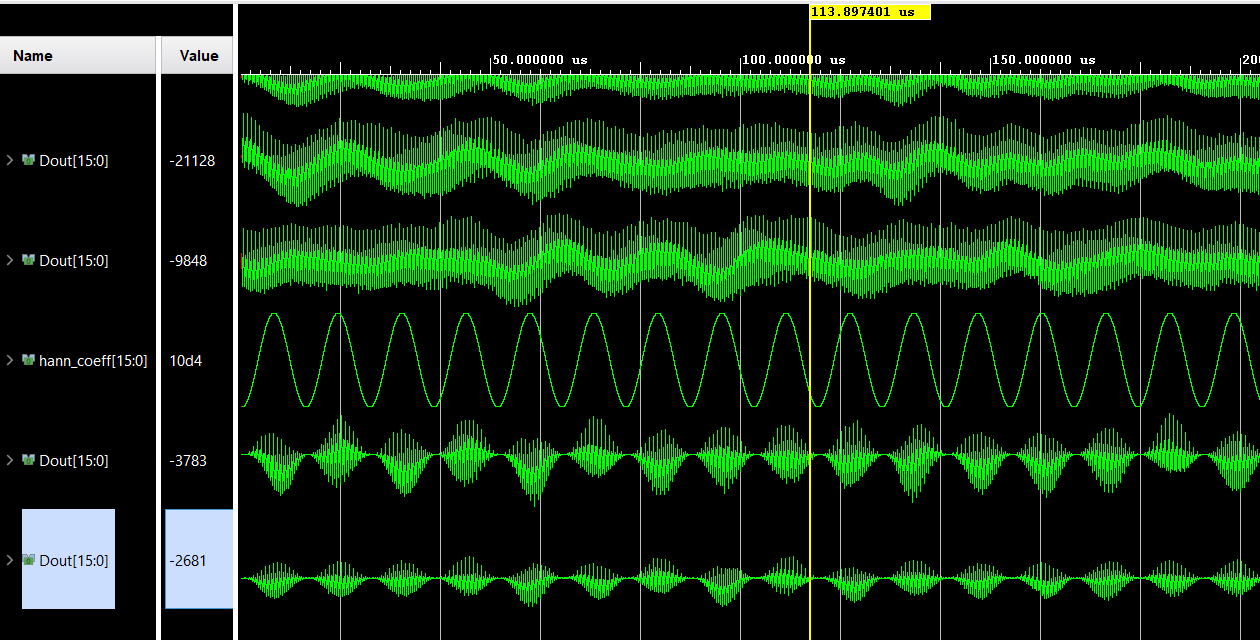
## Simulation result of accelerator (along with waveforms)

For the simulation results too , we are going in order of the block design to keep things clear.

Below is the plots of our frame streamer that streams input data from a text file. It streams data continously one chirp after another , one frame after another.



Tdata is the scaled data from the frame\_streamer that streams from the input data.



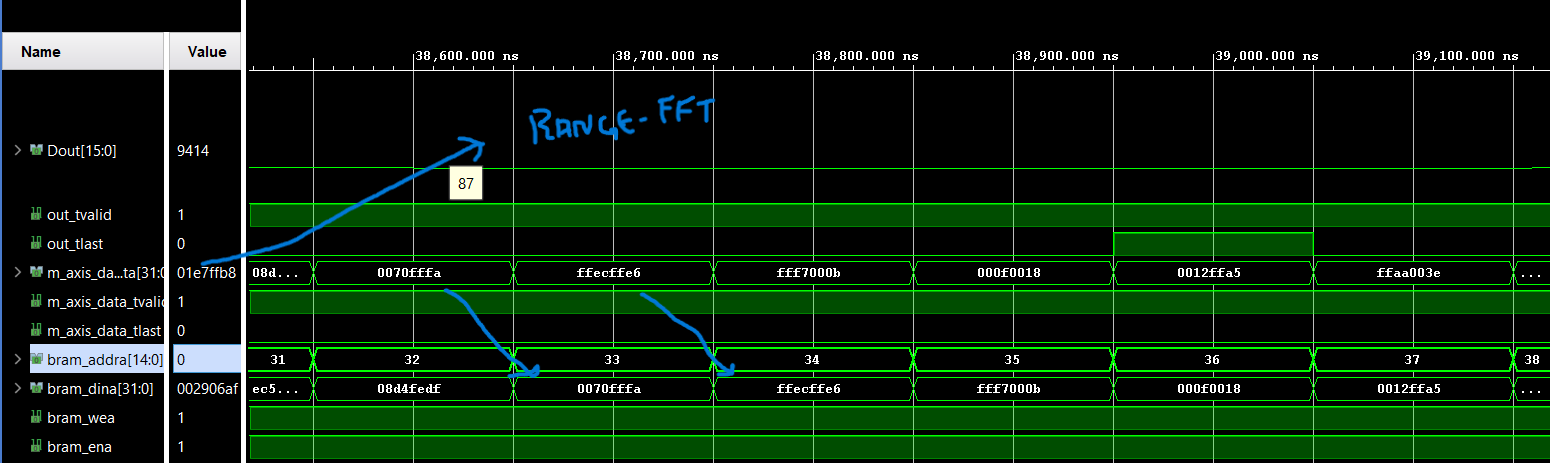
We can clearly see the hann\_window being applied. For more about hann window please refer to background research. The Dout below hann window are the result after we multiply the raw iq signal with hanning coefficients.

The result matched with the expected result from background research.



This is the waveform for the fft\_axi\_adapter. This ensures that valid signal is raised out high when there is valid\_data going in. This also ensures it raises a out\_last signal high every 128th valid sample.

This is working properly and hence there is no error on the simulation side. Else we would see a alignment error/

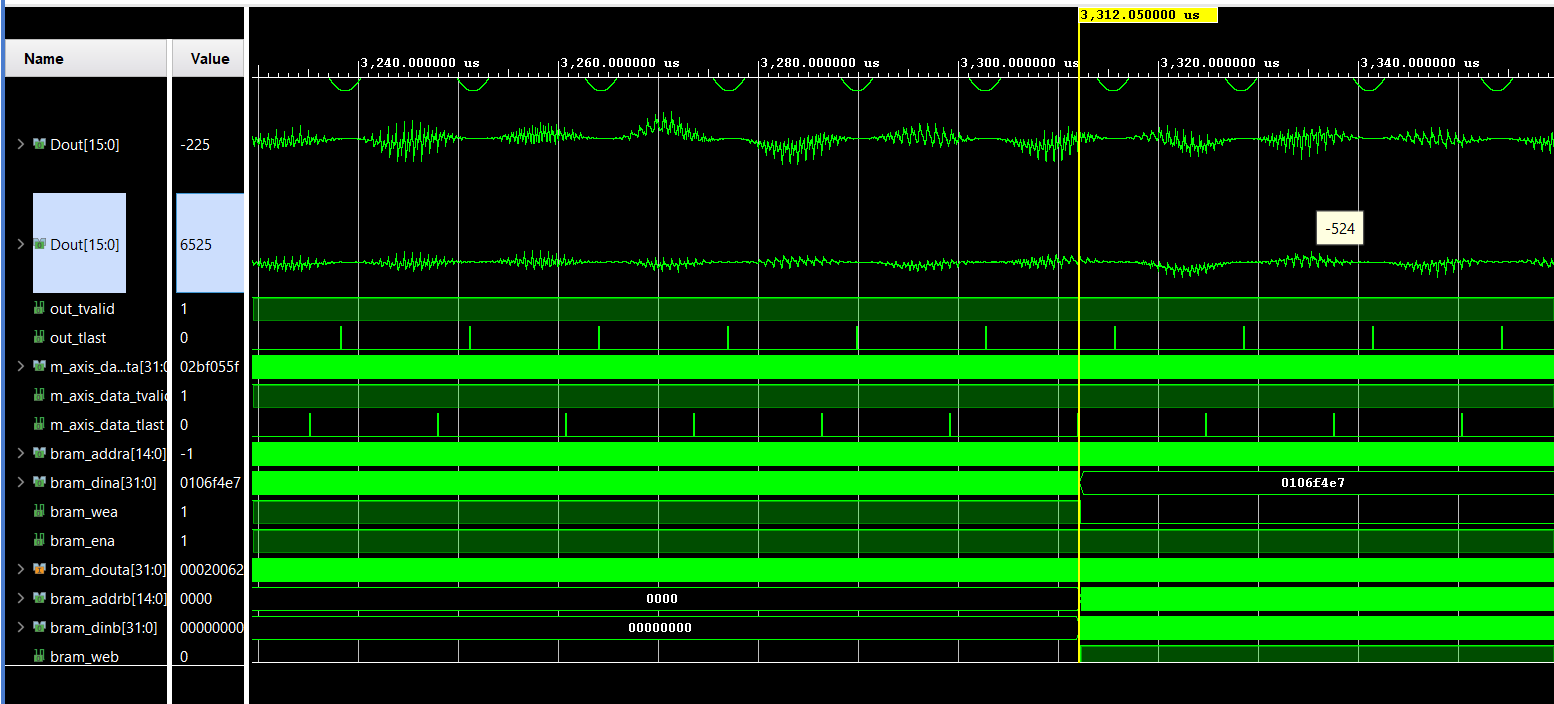


Once range fft produces result , it gives out valid signal and range fft data out , which is used to store inside buffer a using the pinpong buffer control.

Initally both buffers are empty and now Buffer A is being written linearly.

As seen in the image bram\_addra produces a linear address. Which adds 1 to the previous address , basically acts a linear counter.

Wea and ena signals are high and hence writing to buffer A is succesfull!



This is when we stop writing to a and start writing to a. we also start reading from a. From now this process keeps exchanging between each other.

We also can see that at 3,312(3.312ms) us we complete writing from buffer A, and similarly it will take another 3.312ms to read from A.

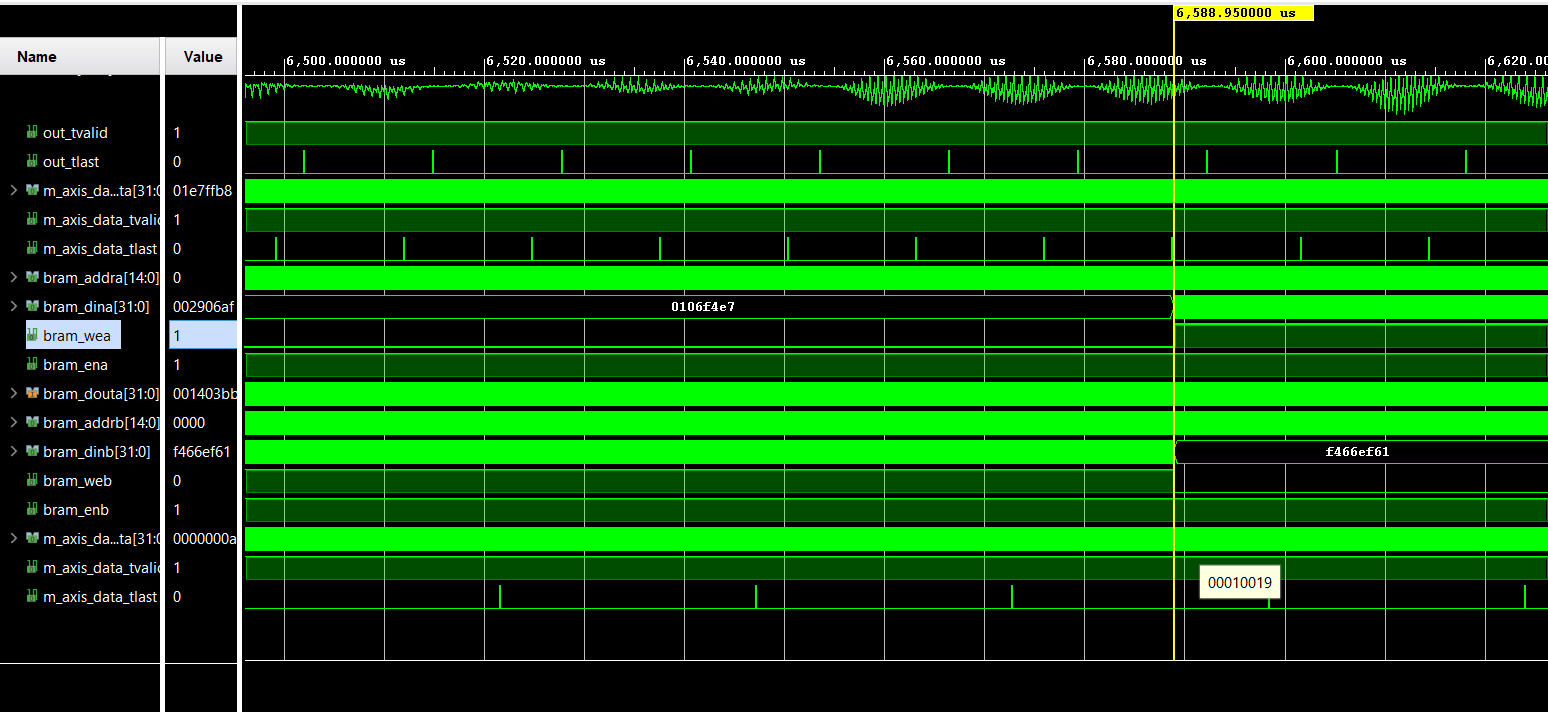
So our first RD map is expected to arive around

Expected time in hardware = 3.3ms \*2 + doppler fft ‘ s initial delay

Dopplers fft is negligible relatively

Hence we can expect our first rd map at 6.6ms

We will discuss about number of frames in later part.

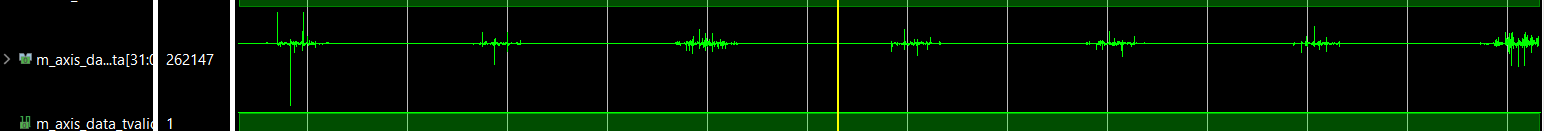


As expected we are getting waveform at almost 6.6ms. a bit less than 6.6 ms because for the initial 3.3ms there was a data processing delay too (multplier , range fft delay etc)

Here we also can see that we stop writing to b and start writing to A again.

We start reading data from b now

This clearly proves our pinpong dual buffer is working flawlessly!.



Doppler output as waveform is show here and its as expeted.

We can further verify by plotting the logged values from the text file!

We have plotted via matlab to visualize and compare.

For the same truck earlier we have computed the rd maps via matlab and as well as from our simulator.

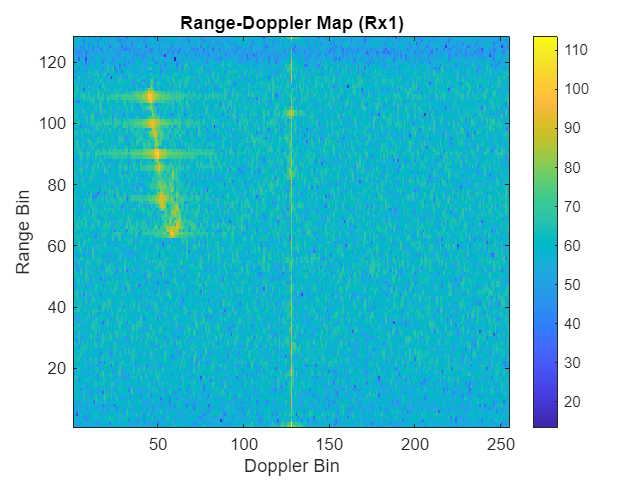


Fig-22 Matlab generated RD map

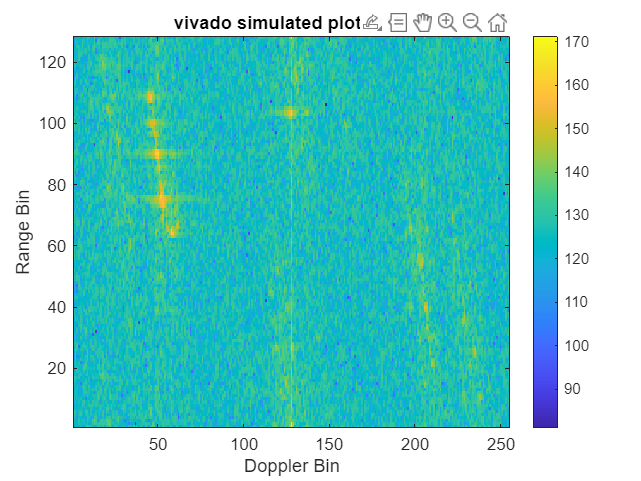


Fig-23 Vivado generated RD map

From both the plots we can clearly see the yellow spot .

Hence we can conclude that our simulation was correct.

From our previous inferences ,

We have seen that our first rd map takes 6ms

But this for our first rd map , since our dual buffer is pipelined

We get a rd map every 3ms

And when we ran the same calculations in matlab

We got a rd map we 0.015 seconds

This shows that our design is 5x much faster !

Also to note that this has been run on ryzen 7 processor , the same processor cannot be expected in common situations ,

Thus making us more rely on a custom Accelerator IP and a low end processor for this task  
Thus obtaining and managing both cost and speed!

ALL THE RTL CODES , SCRIPT FILES AND other TXT files have been both uploaded to our git as well as included in zip file.

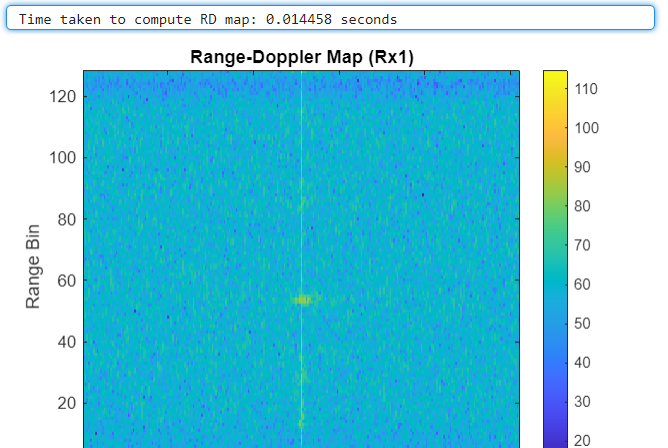


Fig-24 Time for Matlab

AI MODEL AND QUANTIZATION

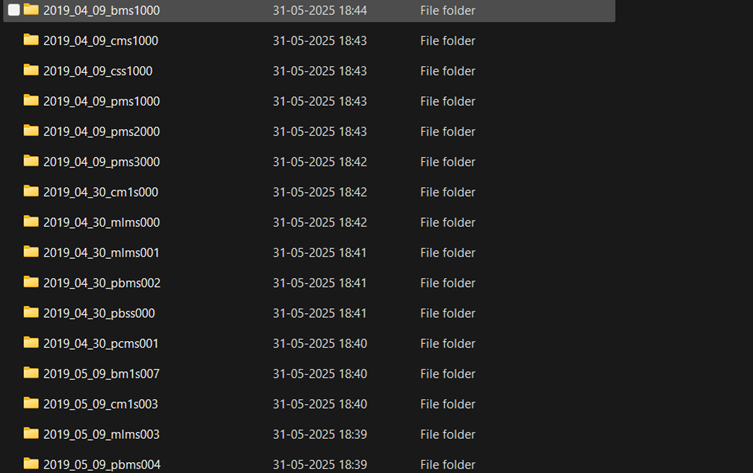
**1. Dataset Acquisition and Preprocessing**

We began by sourcing a 77 GHz automotive radar dataset from the IEEE DataPort repository. This dataset provided raw ADC .mat files representing multiple real-world driving scenarios with objects like cars, bicycles, and pedestrians. Each .mat file contained a 4D tensor representing chirp-wise I/Q samples across antennas.

To make this data usable for machine learning, we developed a Python script that:

* Applied a **Hann window** to reduce spectral leakage
* Performed a **2D FFT** (range then Doppler)
* Converted the result to **dB scale**
* Resized each frame as a **128×128 RD map image and saved In their respective folders**

Fig-25 RD maps stored in respective folders



The final output? A set of **128×128 colorized RD map images** that visually captured the motion signatures of cars, bicycles, pedestrians, Trucks and more.

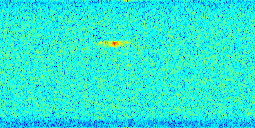


Fig -26 One of the generated RD map for bicycle

**2. Labeling and Class Definition**

Once we had a large dataset of RD map images, the next step was to **label them correctly**. Manually labeling thousands of images would have been slow and error-prone — so we wrote a smart script that automatically **assigned labels** to each image based on the original filename and folder structure.

We organized all the RD maps into five classes:

* car
* bicycle
* pedestrian
* pedestrian\_mixed (scenes with both people and cycles)
* multi (scenes involving multiple types of objects eg:Trucks along with pedestrains and bicycles)

This organization made it possible to feed labeled data directly into our machine learning model for training.

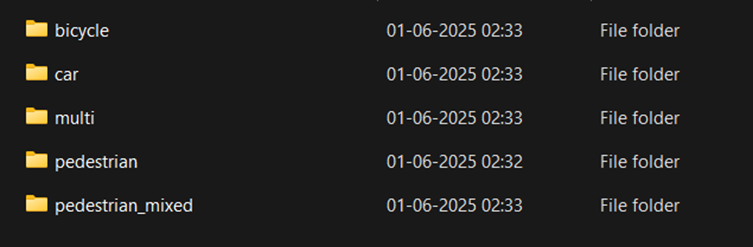


Fig-27 Labelling of RD maps

**3. Designing and Training the CNN**

To classify these RD maps, we built a lightweight yet powerful **Convolutional Neural Network (CNN)** — a type of model well-suited for image data.

The architecture of the CNN was as follows:

* Two convolutional layers with **ReLU activations** to extract visual patterns
* Max pooling layers to reduce dimensionality
* A flattening step followed by a dense (fully connected) layer
* A final output layer with a **softmax activation**, predicting one of the five target classes

We trained this model using the **Adam optimizer** and **categorical cross-entropy loss**. Over several epochs, the model learned to distinguish subtle differences in RD patterns — like how a bicycle’s motion looks very different from a fast-moving car in Doppler space.

After training, the model achieved over **97% accuracy** on our labeled validation dataset.

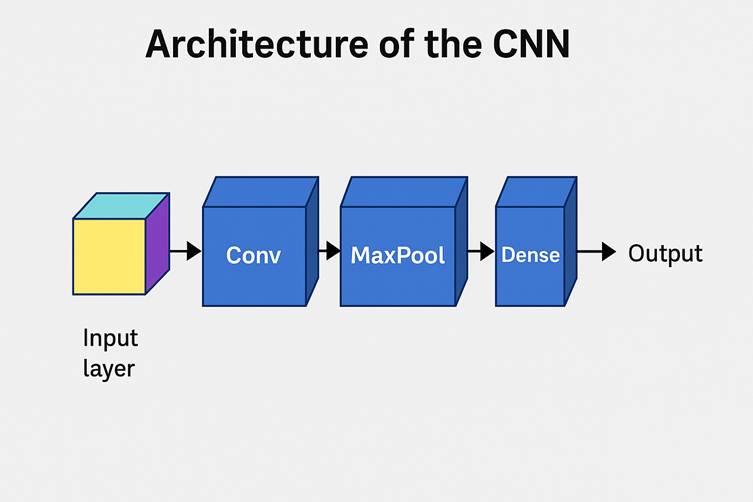


Fig-28 Architecture of CNN

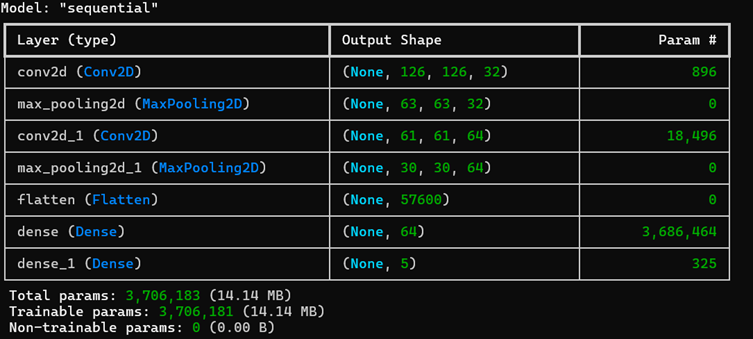


Fig –29 Model summary

**4. Making It Smaller and Faster: Quantization**

To run our trained model on a hardware-constrained environment like an FPGA-based processor, we needed to **compress it without losing accuracy**. That’s where **quantization** came in.

Using TensorFlow Lite, we applied **post-training quantization**, converting all 32-bit floating-point weights and activations to 8-bit integers. The result was impressive:

* Model size dropped from **14 MB to just 3.6 MB**
* Accuracy remained very high — only a **0.7% drop**
* Inference became faster and more efficient

|  |  |  |
| --- | --- | --- |
| **Metric** | **Before Quantization** | **After Quantization** |
| Model format | .h5 (float32) | .tflite (int8) |
| File size | ~14.1 MB | ~3.6 MB |
| Parameters | 3,706,183 | 3,706,183 |
| Accuracy (test set) | ~97.2% | ~96.5% |
| Inference time (CPU) | 45–60 ms (float32) | 5–10 ms (int8) |
| Target platform | PC (sim) | VEGA (planned) |

Table –2 Before vs After quantization

Despite the quantization, there was **negligible drop in accuracy (~0.7%)**, with a significant improvement in inference speed and memory efficiency. The quantized model was saved as a .tflite file, ready for edge deployment

**5. Simulation-Based Testing and Validation**

Although we haven’t deployed the model on the FPGA yet, we took an important validation step: **testing the quantized model on RD maps generated by our Vivado-based radar simulation**.

We generated realistic RD plots using the same FFT logic and processed them as input to the CNN. The model successfully recognized the motion classes, such as multi or pedestrian, with high confidence — demonstrating that it **generalizes well even on unseen, simulation-generated data**.

This not only validates our pipeline but also builds a solid foundation for future real-time deployment on the **VEGA processor within the Genesys 2 FPGA board**, where RD maps will be streamed via DDR and processed frame by frame.

We generated realistic RD plots using the same FFT logic and processed them as input to the CNN. The model successfully recognized the motion classes, such as multi (As it is combination of truck along bicycle and a pedestrian), with high confidence — demonstrating that it **generalizes well even on unseen, simulation-generated data**.

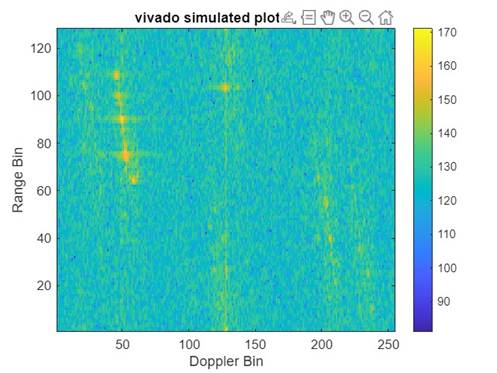


Fig-30 Input RD map Generated from Vivado for Testing

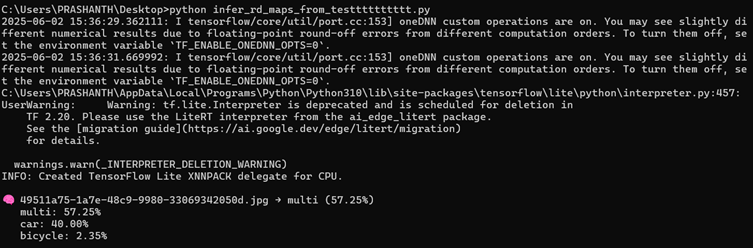


Fig-31 Output response

This not only validates our pipeline but also builds a solid foundation for future real-time deployment on the **VEGA processor within the Genesys 2 FPGA board**, where RDmaps will be streamed via DDR and processed frame by frame.

**6. Hardware Readiness and Future Deployment**

Although our quantized model has not yet been deployed to physical hardware, we simulated the entire end-to-end pipeline and validated its compatibility with our target platform: the **VEGA processor embedded in the Genesys 2 FPGA board**.

The VEGA core is designed to support AI inference on quantized models with efficient integer arithmetic and low memory overhead. Our .tflite model, which compresses the CNN to just **3.6 MB**, fits comfortably within VEGA’s instruction and data memory specifications. The **AXI-stream-based architecture** allows the FPGA logic (PL) to feed 128×256 RD map frames directly into the DDR, from which the processor can access them for inference.

We anticipate a high degree of performance efficiency, provided that:

* The AXI throughput is sustained at expected rates
* Preprocessing latency (FFT + windowing) is minimized in the hardware pipeline

The quantized model has already been tested on a host machine using simulated RD maps, and the predictions are consistent and confident. The next milestone will involve fully integrating the VEGA-side data fetch logic and running real-time inference, confirming that the system operates reliably when fed live data from the programmable logic.

This staged validation approach ensures that each component of the system — from signal capture to AI classification — is robust and optimized before full deployment

# Results and Discussion

During simulation, we observed that the first complete Range-Doppler (RD) map generated by our design took approximately **6 milliseconds**, which includes pipeline startup latency. However, due to the **dual-buffer ping-pong architecture**, once the pipeline is filled, we are able to produce a new RD map every **3 milliseconds** consistently.

For comparison, we implemented the same radar signal processing chain (including HANN windowing, Range FFT, transposition, and Doppler FFT) in MATLAB and ran it on a **Ryzen 7 processor**. The MATLAB implementation produced a single RD map in approximately **15 milliseconds**.

This proves that our design is 5 times faster and efficient.

That means we are able to update our result every 3ms ,

Hence our frame rate becomes nearly 300hz.

Which very much meets real time requirements.

**What Could Be Improved?**

**1. RD-Angle Map Generation**

While the current system successfully produces high-quality Range-Doppler (RD) maps, a potential improvement would be to extend the design to also compute **angle-of-arrival (AoA)** information, forming full **Range-Doppler-Angle (RDA) maps**. By incorporating multiple receiver antennas and applying algorithms such as **MUSIC** or **beamforming FFTs**, the system could estimate object direction in addition to range and velocity. This would elevate the radar’s capability from 2D motion detection to full **3D situational awareness**, enabling tasks like lane-level localization, multiple object separation, and more accurate tracking in complex scenes.

**2. Kalman Filter-Based Object Tracking**

Another area for enhancement lies in **integrating our previously developed FFT-accelerated Kalman Filter module**. This would allow continuous **temporal tracking of detected objects** across multiple RD map frames. While our current pipeline performs per-frame classification, incorporating a Kalman filter would enable smoother trajectory estimation, velocity prediction, and clutter rejection over time. This is especially useful for dynamic environments like autonomous driving, where maintaining object identity across frames is critical. By embedding the Kalman filter into the post-processing pipeline, we could turn static RD detections into **robust motion tracks**, significantly enhancing the intelligence and reliability of the system.

3. we could have used better rounding off technique or played around with data formats to see which of the following gives us a cleaner rd map.

Next Objective:

* 1. As soon as we find another better data format or rounding off technique to avoid data loss and get a better and cleaner rd map , we will incorporate that format/ technique into our present project
  2. We have also implemented a FFT based matched filer , using a Kalman filter we could combine both fmcw and pulsed radar and grab the best of both.
  3. Figure out of we will stream the data from fmcw radar to our accelerator IP

All the script file , RTL codes and other required text files to be uploaded here

<https://github.com/Loner2104/Object-Detection-and-Classification-using-fmcw-radar>

# Conclusion

Looking back, this project has been one of the most exciting and rewarding experiences we’ve had as a team. What started as a technical idea on paper quickly turned into a hands-on journey that taught us so much — not just about radar and AI, but about working together, thinking creatively, and solving real-world problems.

We began by digging into raw radar data, trying to make sense of it, and slowly built our way up — crafting custom hardware accelerators, optimizing data flow, and finally deploying a quantized AI model on the VEGA processor. Watching the Range-Doppler maps come to life in real-time felt surreal. That moment — when everything just worked — was honestly thrilling. It made all the debugging, the late nights, and the long discussions completely worth it.

What really made this project special was how many different worlds it brought together — signal processing, hardware design, AI, and embedded systems. It pushed us to think across layers and come up with solutions that were not just smart, but efficient and practical.

We didn’t just build a system — we built confidence, gained clarity on how things really work in embedded AI, and discovered how much we enjoy doing this kind of work. It showed us that we're capable of turning complex ideas into something that works, something we’re proud of.

# References

[1] M. N. Islam and X. Gao, “Raw ADC Data of 77GHz mmWave Radar for Automotive Object Detection,” *IEEE DataPort*, Dataset, 2022. [Online]. <https://ieee-dataport.org//documents/raw-adc-data-77ghz-mmwave-radar-automotive-object-detection>

2)

M. Khan et al., "Design and Implementation of FPGA based System for Object Detection and Range Estimation used in ADAS Applications utilizing FMCW Radar," 2024 IEEE International Symposium on Circuits and Systems (ISCAS), Singapore, Singapore, 2024, pp. 1-5, doi: 10.1109/ISCAS58744.2024.10558003. keywords: {Accuracy;Fast Fourier transforms;Estimation;Radar detection;Prototypes;Object detection;Hardware;ADAS;FFT;FMCW Radar;FPGA;range-estimation;Object-detection},<https://ieeexplore.ieee.org/document/10558003>

3)

M. Ibrahim and O. Khan, "Performance analysis of Fast Fourier Transform on Field Programmable Gate Arrays and graphic cards," 2016 International Conference on Computing, Electronic and Electrical Engineering (ICE Cube), Quetta, Pakistan, 2016, pp. 158-162, doi: 10.1109/ICECUBE.2016.7495215. keywords: {Graphics processing units;Field programmable gate arrays;Signal processing algorithms;Discrete Fourier transforms;Kernel;Hardware;Instruction sets;Fast Fourier Transform;Field Programmable Gate Arrays;Graphic Processing Units;Open Computing Language;Verilog HDL},<https://ieeexplore.ieee.org/document/7495215>

4)

Z. Qian and G. Gan, "Accelerating Real-Valued FFT on CPU-FPGA Platforms," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 43, no. 8, pp. 2532-2536, Aug. 2024, doi: 10.1109/TCAD.2024.3377160.

keywords: {Flow graphs;Computer architecture;Field programmable gate arrays;Hardware;Mathematical models;Signal processing algorithms;Design automation;FPGA;high-level synthesis;open computing language (OpenCL);real-valued fast Fourier transform (FFT)},<https://ieeexplore.ieee.org/document/10472631>