IP Creation Using HLS for 1 Dimensional RSP on Zynq MPSoC

B.Tech in Electronics and Communication Engineering

Arnav Shukla (2022103)

Under the guidance of
Dr. Sumit J. Darak

Indraprastha Institute of Information Technology, Delhi

May 2024 - July 2024

Certificate

This is to certify that the project report entitled **"IP Creation Using HLS for 1 Dimensional RSP on Zynq MPSoC"** submitted by **Arnav Shukla (2022103)** in partial fulfillment of the requirements for the degree of B.Tech in Electronics and Communication Engineering of Indraprastha Institute of Information Technology, Delhi, is a record of the candidate's own work carried out under my supervision. The matter embodied in this project is original and has not been submitted for the award of any other degree.

Dr. Sumit J. Darak
Associate Professor,
Department of Electronics and Communication Engineering,
IIIT Delhi

Acknowledgement

I would like to express my deep gratitude to Dr. Sumit J. Darak, my research supervisor, for his patient guidance, enthusiastic encouragement, and useful critiques of this research work. I would also like to thank my parents for their constant support and encouragement throughout my study. My grateful thanks are also extended to my friends and colleagues for their help in offering me the resources in running the program.

Abstract

This document discusses the process of creating Intellectual Property (IP) using High-Level Synthesis (HLS). The focus is on the development steps, control logic extraction, hardware and software co-design, clock uncertainty management, and optimization techniques applied to an FPGA environment.

1 Introduction

High-Level Synthesis (HLS) is a process that converts C/C++ code into hardware description languages like Verilog or VHDL. After converting C/C++ code to verilog, HLS packages it and hence can be used to make Vivado IPs . This methodology facilitates the creation of efficient hardware designs with shorter development cycles. The integration of hardware and software co-design is crucial in modern embedded systems to optimize performance, flexibility, and resource utilization.

2 Field-Programmable Gate Array (FPGA)

Field-programmable gate arrays (FPGAs) are semiconductor devices designed to be configured by designers after manufacturing. This flexibility is achieved through the FPGA's architecture, which consists of a sea of configurable logic blocks (CLBs) interconnected via programmable interconnects. Due to their programmable nature, FPGAs have penetrated various markets, offering unique advantages in terms of customization, performance, and rapid prototyping.

2.1 FPGA Architecture

The architecture of an FPGA includes:

- **Configurable Logic Blocks (CLBs)**: These are the fundamental building blocks of an FPGA, consisting of look-up tables (LUTs), flip-flops, and multiplexers. CLBs perform logic operations and store intermediate results.
- **Programmable Interconnects**: These interconnects allow CLBs to be connected in various configurations, enabling designers to create complex circuits.
- **Block RAM (BRAM)**: FPGAs include embedded memory blocks to store data and intermediate results, providing fast and efficient data access.
- **Digital Signal Processing (DSP) Slices**: These specialized blocks are optimized for arithmetic operations, crucial for applications like signal processing and matrix multiplication.
- **I/O Blocks**: These blocks manage the interface between the FPGA and external devices, supporting various communication protocols.

2.2 Hardware-Software Co-Design

To effectively map our Radar Signal Processing (RSP) algorithms onto the FPGA, we employ a technique called hardware-software co-design. This approach involves partitioning the design into hardware and software components to optimize performance and flexibility.

2.2.1 Design Partitioning

In hardware-software co-design, computationally intensive tasks are offloaded to hardware, while control and configuration tasks are managed by software. This partitioning maximizes the strengths of both hardware and software, providing an efficient and scalable solution.

- **Hardware Accelerators**: Key processing tasks of the RSP algorithm, such as filtering, FFT, and matrix operations, are implemented as hardware accelerators on the FPGA. These accelerators exploit parallelism and pipelining to achieve high performance.
- **Software Control**: The software component, typically running on an embedded processor within the FPGA (e.g., ARM Cortex-A cores in Zynq SoC), manages the configuration and coordination of hardware accelerators. This includes setting parameters, triggering operations, and handling data transfers. Operations like Peak Searching after digital beam-forming were performed on PS(Processing subsystem).

2.2.2 Communication Interfaces

Effective communication between hardware and software components is critical in co-design. We use industry-standard AMBA(Advanced Microcontroller Bus Architecture) interfaces to ensure seamless data transfer and control signaling.

- **AXI4 (Advanced eXtensible Interface)**: This interface standard is widely used for connecting different IP blocks within the FPGA. AXI4 ensures high-throughput and low-latency communication between the processor and hardware accelerators.
- **APB (Advanced Peripheral Bus)**: Used for low-bandwidth control signaling, APB interfaces facilitate the configuration of hardware accelerators by the processor.

2.3 Implementation in Vivado

The implementation of our RSP algorithm on FPGA involves several steps using Xilinx Vivado, a comprehensive suite for FPGA design.

1. **High-Level Synthesis (HLS)**: Convert the algorithmic description (written in C/C++) into RTL code (Verilog/VHDL). HLS allows for rapid prototyping and design space exploration.
2. **RTL Design and Verification**: Refine the generated RTL code, optimizing it for performance and resource utilization. Verify the design through simulation and HLS testbench to ensure correctness.
3. **Block Design in Vivado**: Integrate the RTL modules with other IP blocks, such as memory controllers and I/O interfaces, to form a complete system. Use the Vivado block design environment to connect and configure these blocks.
4. **Synthesis and Implementation**: Perform logic synthesis to map the RTL design onto the FPGA's resources. Place and route the design to ensure optimal performance and meet timing requirements.

5. **Hardware-Software Integration**: Develop software drivers and application code to control and utilize the hardware accelerators. Test the integrated system to ensure smooth operation. This step involves using Vivado - SDK (software development kit).

2.4 Conclusion

FPGAs provide a flexible and powerful platform for implementing complex algorithms like RSP. Through hardware-software co-design, we can leverage the strengths of both hardware accelerators and software control to achieve high performance and adaptability. This approach not only enhances the efficiency of our RSP implementation but also allows for rapid prototyping and iterative development, making FPGAs an ideal choice for advanced signal processing applications.

3 DMA in Zynq SoC

Direct Memory Access (DMA) in Zynq System on Chip (SoC) involves generating a wrapper and bitstream via Vivado. The source and destination addresses are defined, and the hardware is exported for integration into the software development kit (SDK). This process exemplifies hardware and software co-design, where hardware accelerators are tightly coupled with software drivers to enhance data transfer efficiency.

3.1 DMA Polling in Vivado SDK

The following code snippet performs DMA (Direct Memory Access) polling using the AXI DMA controller in Vivado SDK. The objective is to wait for the DMA transfer to complete by checking specific status bits continuously.

```
status = XAxiDma_ReadReg(XPAR_AXI_DMA_0_BASEADDR, 0x04) & 0x00000002;
while(status != 0x00000002)
{
    status = XAxiDma_ReadReg(XPAR_AXI_DMA_0_BASEADDR, 0x04) & 0x00000002;
}

status = XAxiDma_ReadReg(XPAR_AXI_DMA_0_BASEADDR, 0x34) & 0x00000002;
while(status != 0x00000002)
{
    status = XAxiDma_ReadReg(XPAR_AXI_DMA_0_BASEADDR, 0x34) & 0x00000002;
}
```

Explanation:

- **Initialization and Polling:** The first status check reads the status register at offset `0x04` and continuously polls until bit `0x00000002` is set, indicating DMA transfer completion. The second status check performs a similar operation at offset `0x34`.

This polling mechanism ensures the program waits until the DMA operations are completed before proceeding further.
This process can be further optimized by using *fabric interrupts* which when process is completed tells the processor to stop the polling.

4 High-Level Synthesis (HLS)

4.1 Concepts and Definitions

- **Latency**: Number of clock cycles for execution.
- **Trip Count**: Number of iterations in a loop.

HLS converts C/C++ code to hardware description languages like Verilog through a series of steps, including logic synthesis, implementation, and bitstream generation. This transformation process highlights the interplay between software algorithm design and hardware implementation. Development time is increased and so the performance for HW codes.
Hence, HLS transforms C/C++ into RTL.

5 High-Level Synthesis (HLS) Phases

High-Level Synthesis (HLS) is a process that converts a high-level algorithmic description, typically written in C, C++, or SystemC, into a hardware description language (HDL) like Verilog or VHDL. This transformation is essential for designing digital circuits, particularly when targeting Field-Programmable Gate Arrays (FPGAs). The HLS process consists of two primary phases: Iterative Scheduling and Binding.

5.1 Scheduling

Scheduling in HLS involves translating the high-level algorithm into a sequence of operations that can be executed in hardware. This phase is iterative and influenced by several factors:

- **Clock Frequency**: The speed at which the FPGA operates. A higher clock frequency means more operations can be completed within a single clock cycle.
- **Target Device and Directives**: The specific FPGA device being targeted and the user directives provided, which guide the synthesis tool in optimizing the design.
- **Resource Availability**: The availability of computational resources on the FPGA, such as logic units and memory blocks.

During scheduling, the algorithm's operations are assigned to specific clock cycles. The goal is to optimize the performance while adhering to the constraints of the target FPGA. This process ensures that operations are efficiently mapped onto the hardware resources available.

5.2 Binding

Binding is the process of assigning the scheduled operations to physical resources on the FPGA. This phase takes into account the functional and timing characteristics of these resources. Binding involves:

- **Resource Assignment**: Allocating combinational and sequential circuits to execute the scheduled operations.

- **Physical Constraints**: Ensuring that the assigned resources meet the timing and spatial constraints of the FPGA design.

The binding phase is closely linked to scheduling, as the information about resource assignment is fed back into the scheduling process to refine the operation scheduling. This iterative feedback loop ensures that the final RTL design is both efficient and meets the desired performance criteria.

5.3 Iterative Scheduling and Binding

The iterative nature of scheduling and binding in HLS is crucial for achieving an optimal hardware design. The process can be summarized as follows:

1. **Initial Scheduling**: Initial mapping of operations to clock cycles based on the high-level algorithm.
2. **Initial Binding**: Assigning initial resources to the scheduled operations.
3. **Feedback Loop**: Evaluating the initial scheduling and binding results, identifying bottlenecks and inefficiencies.
4. **Refinement**: Iteratively refining the scheduling and binding based on feedback, adjusting the assignment of operations and resources.
5. **Final RTL Generation**: Converting the optimized schedule and resource binding into an RTL design, represented in Verilog, VHDL, or SystemC.

5.4 Practical Considerations

In practical HLS design for FPGA, several additional considerations come into play:

- **User Directives**: Specific instructions provided by the designer to guide the synthesis tool, such as unrolling loops or pipelining operations.
- **Technology Library**: Information about the target FPGA technology, including available resources and their characteristics.
- **Clock Domain Crossing**: Managing data transfers between different clock domains, which may have different clock frequencies.

5.5 Conclusion

HLS simplifies the process of designing complex digital circuits by abstracting the design away from low-level HDL coding to high-level algorithmic descriptions. The phases of scheduling and binding are critical in this process, ensuring that the design is both efficient and meets the performance requirements of the target FPGA. By iteratively refining the schedule and resource assignment, HLS tools can generate highly optimized RTL designs suitable for implementation on modern FPGAs.

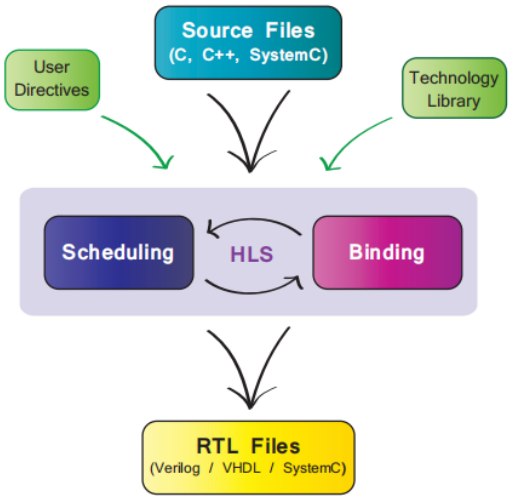

Figure 14.7: Vivado HLS scheduling and binding processes

Figure 1: *source: FPGAkey*

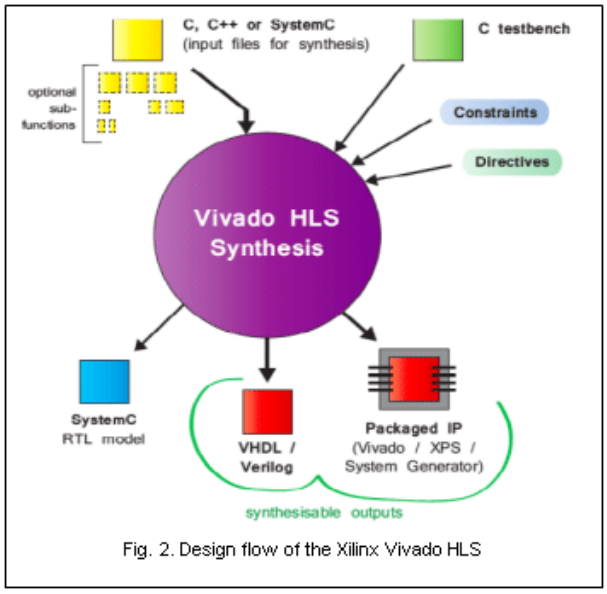

Figure 2: High-Level Synthesis Of Inverse Quantization And Transform Block For HEVC Decoder On FPGA - Scientific Figure on ResearchGate.

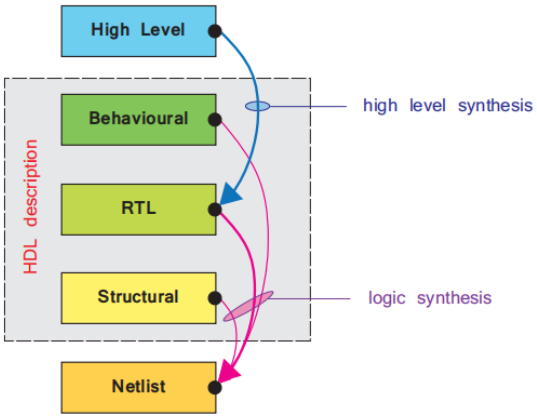

Figure 14.2: High level synthesis and logic synthesis

Figure 3: *Source : FPGAkey*

6 Control Logic Extraction

Control logic extraction is a critical phase where software-defined operations are translated into hardware control signals, demonstrating the principles of co-design.

- Identify datapath and control operations.
- Extract control logic to create a Finite State Machine (FSM) to implement sequences.
- Implement top-level functional arguments as ports in the final RTL design.

7 Register Transfer Level (RTL) Design

Register Transfer Level (RTL) design is a crucial phase in digital circuit design where the functionality of the circuit is defined in terms of the flow of signals between hardware registers and the logical operations performed on those signals. RTL is the abstraction level used to design synchronous digital circuits and is pivotal in the hardware and software co-design process.

7.1 Concepts and Components

RTL design involves several key concepts and components:

- **Registers**: Storage elements that hold data. In RTL, data is transferred between these registers on clock edges.
- **Combinational Logic**: Logic gates and circuits that perform operations on data without storing it. These operations include arithmetic calculations, comparisons, and logic functions.
- **Sequential Logic**: Circuits where the output depends on the current input and the history of past inputs, typically implemented using flip-flops and latches.

- **Finite State Machines (FSMs)**: Control units that manage the state transitions based on input signals and current states, essential for implementing control logic in digital systems.

7.2 RTL Design Flow

The RTL design flow includes the following steps:

1. **Specification**: Define the functionality, performance, and constraints of the design.
2. **Behavioral Description**: Write a high-level description of the design using hardware description languages (HDLs) like Verilog or VHDL.
3. **Synthesis**: Convert the behavioral description into a gate-level netlist using synthesis tools. This step includes logic optimization and mapping to the target technology.
4. **Simulation and Verification**: Validate the functionality of the RTL design through simulation and ensure it meets the specifications.
5. **Place and Route**: Physically place the synthesized netlist onto the silicon or FPGA fabric, routing the connections between logic elements.
6. **Timing Analysis**: Verify that the design meets timing requirements, ensuring that data transfers between registers occur within the clock period.

7.3 Hardware and Software Co-Design

In the context of hardware and software co-design, RTL design plays a critical role by providing a clear interface between hardware accelerators and software components:

- **Hardware-Software Partitioning**: Determine which parts of the design will be implemented in hardware and which in software. Computationally intensive tasks are typically offloaded to hardware, while control and configuration are handled by software.
- **Interface Definition**: Define the communication protocols and interfaces (e.g., AXI, APB) between the hardware accelerators and the software. This includes data buses, control signals, and status registers.
- **Driver Development**: Develop software drivers to manage the hardware accelerators. These drivers initialize the hardware, manage data transfers, and handle interrupts.
- **Performance Optimization**: Optimize the RTL design to ensure efficient data flow and minimal latency. This includes techniques like pipelining, parallelism, and resource sharing.

8 IPs Created

I have created several Intellectual Property (IP) cores for various signal processing tasks. These IPs are designed to be used with both memory-mapped and stream interfaces. For each IP, word length optimization was performed, and other algorithmic optimizations were implemented using High-Level Synthesis (HLS) pragmas to enhance performance. The IP cores created are as follows:

- **Matrix Multiplication IP:** This IP performs matrix multiplication, essential for many linear algebra computations in signal processing and machine learning applications.
- **Digital Beamforming IP:** This IP, essentially a specialized form of matrix multiplication, is designed for beamforming applications. It is available for configurations with 8 and 32 antennas, supporting 1, 2, and 3-degree precision.
- **Complex Element Multiplication IP:** This IP handles the multiplication of complex numbers, a fundamental operation in many digital signal processing (DSP) tasks.
- **FFT (Fast Fourier Transform) IP:** This IP computes the FFT, a critical operation for frequency domain analysis in DSP.
  based on Xilinx FFT example
- **IFFT (Inverse Fast Fourier Transform) IP:** This IP performs the inverse operation of the FFT, converting frequency domain data back to the time domain.
  based on Xilinx IFFT example
- **FIR Filter IP:** This IP implements a Finite Impulse Response filter, which is widely used for signal filtering applications in DSP.
  based on HLS book(references)

Each IP core has been designed to ensure high performance and efficiency through rigorous word length optimization and the application of HLS pragmas for various algorithmic improvements. These optimizations make the IP cores suitable for a wide range of applications in embedded systems and signal processing.

9 Optimization Techniques

This section covers various optimization techniques used in High-Level Synthesis (HLS) to improve performance and efficiency.

9.1 Loop Pipelining

Loop pipelining allows concurrent operations and avoids loop carry dependencies to enable efficient execution.

```
#pragma HLS pipeline
```

This pragma also takes care of merging two loops. It increases the throughput, although it may require more resources. All loops below this pragma are automatically unrolled.

9.1.1 Loop-Carry Dependency

Loop pipelining can be affected by loop carry dependencies. A loop dependency causes the Initiation Interval (II) to be greater than 1. When moving from one loop iteration to another, a bubble is created due to write operations of the previous loop. This can be avoided by:

```
#pragma HLS pipeline II=<int> enable_flush rewind
```

This directive stops waiting for input when none is available and continues processing the available data.

9.2 Array Partitioning

Partitioning arrays improves access and performance, making the design more efficient. Since the number of ports on Block RAM (BRAM) is limited, only a limited number of samples can be accessed at a time. For faster access, particularly in matrix multiplication, arrays can be partitioned.

In matrix multiplication: - Matrix A can be completely partitioned along the columns (`dim=2`). - Matrix B can be completely partitioned along the rows (`dim=1`).

Thus, for Matrix A, three BRAMs will store three different columns, and for Matrix B, three BRAMs will store three different rows.

```
#pragma HLS array_partition variable=<array> complete dim=<dimension>
```

This pragma specifies the array to be partitioned and the dimension along which the partitioning should occur.

9.2.1 Unrolling and Pipelining

Loop unrolling and pipelining are techniques that overlap the execution of multiple loop iterations. This parallel execution of loop operations allows the same operators to work in parallel on different data stages.

For example:

- Applying loop unrolling to parallelize multiplication operations.
- Unrolling factors and the number of iterations can be specified to control the degree of parallelism.

```
#pragma HLS unroll factor=<int>
```

This directive specifies the unrolling factor and the number of iterations to be unrolled.

10 Stream and Memory Mapped

10.1 Stream Interface

The desired AXIS interface can be achieved using the following directive:

```
#pragma HLS INTERFACE axis register both port=my_port
```

To remove the control signal, the following directive is used:

```
#pragma HLS INTERFACE ap_ctrl_none port=return
```

10.1.1 Data Transfer Using Stream Interface

In High-Level Synthesis (HLS), data can be transferred between functions and modules using streams. This section explains how data transfer occurs for the stream interface with an example code of matrix multiplication.
Consider the following code snippet which demonstrates the use of HLS streams for data transfer:

```
axis_data local_read, local_write;
hls::stream<axis_data> in_vec, out_vec;

writeInVec: for(row=0;row<M;row++){
    for(col=0;col<N;col++)
    {
        local_write.data = input_A[row][col];

        if((row==M-1) && ((col==N-1)))
            local_write.last = 1;
        else
            local_write.last = 0;

        in_vec.write(local_write);
    }
}

digitalBeamforming_S_wlo(in_vec, out_vec);

int ind = 0;
for(row = 0; row< M; row++){
    for(col = 0; col < P; col++){
        local_read = out_vec.read();
        output_C_HW[row][col] = local_read.data;
    }
}
```

In this example, we use HLS streams `in_vec` and `out_vec` for transferring data. Data from a 2D array `input_A` is written to the `in_vec` stream using the following loop:

```
writeInVec: for(row=0; row<M; row++){
    for(col=0; col<N; col++) {
        local_write.data = input_A[row][col];

        if((row==M-1) && ((col==N-1)))
            local_write.last = 1;
        else
            local_write.last = 0;

        in_vec.write(local_write);
    }
}
```

- `local_write.data` stores the current element of `input_A`. - `local_write.last` is a flag indicating the last element of the stream. - `in_vec.write(local_write)` writes the data and the flag to the stream.

The function `digitalBeamforming_S_wlo` processes the data:

```
digitalBeamforming_S_wlo(in_vec, out_vec);
```

This function reads from the `in_vec` stream, processes the data, and writes the results to the `out_vec` stream.

The processed data is read from the `out_vec` stream and stored in the 2D array `output_C_HW`:

```
int ind = 0;
for(row = 0; row < M; row++){
    for(col = 0; col < P; col++){
        local_read = out_vec.read();
        output_C_HW[row][col] = local_read.data;
    }
}
```

- `local_read` stores the data read from the `out_vec` stream. - The data is then assigned to the corresponding element in `output_C_HW`.

10.2 Memory-Mapped Interface

The memory-mapped interface allows for direct reading and writing to memory without the need for DMA (Direct Memory Access). This reduces the number of transactions from 2 to 1, improving efficiency. The AXI Lite port is used to specify where to read or write data from memory.

In High-Level Synthesis (HLS), data can be transferred using the AXI Memory-Mapped (AXI MM) interface. The following example demonstrates how to implement matrix multiplication using the AXI MM interface.

```
#include "matmul.h"

// Interface for AXI MM
void matmul_MM_SP(Mat_Dtype *Matrix_In, Mat_Dtype *Matrix_C_HW)
{
    #pragma HLS INTERFACE s_axilite port=return
    #pragma HLS INTERFACE m_axi depth=64 port=Matrix_In offset=slave
    #pragma HLS INTERFACE m_axi depth=128 port=Matrix_C_HW offset=slave

    // Define the depth same as the number of elements to be read or write
    // slave offset means we can update the offset from AXI Lite interface
}
```

10.2.1 Data Transfer Using `memcpy`

In C, the `memcpy` function can be used to transfer data from one memory location to another. This is particularly useful for transferring data between the host and the hardware accelerator.

```
#include <string.h>

// Example usage of memcpy for data transfer
void transfer_data(Mat_Dtype *source, Mat_Dtype *destination, size_t size)
{
    // Copy data from source to destination
    memcpy(destination, source, size * sizeof(Mat_Dtype));
}
```

In this example, `memcpy` is used to copy data from the source array to the destination array. The `size` parameter specifies the number of elements to copy, and

`sizeof(Mat_Dtype)`

is used to determine the size of each element.

11 Word-Length Optimizations (WLO)

Word-Length Optimization (WLO) is a crucial step in designing efficient hardware for digital signal processing. By finding the optimal fixed-point representation (`ap_fixed<TOTAL_BITS, INTEGER_BITS>`), we can significantly reduce the hardware resource usage and power consumption while maintaining acceptable levels of accuracy.

11.1 Importance of WLO

WLO is helpful because it allows us to:

- Minimize the bit-width of data representations, leading to reduced area and power consumption on FPGAs and ASICs.
- Optimize the balance between resource usage and the precision of computations.
- Improve overall performance by using the least number of bits necessary to achieve the desired accuracy.

11.2 Methodology

The WLO process involves several steps:

1. **Determine the Range of Data:** Find the range of the data values that the algorithm will process.
2. **Find the Largest Integer Value:** Identify the largest integer value within this range.
3. **Calculate the Number of Integer Bits:** The number of integer bits is the number of bits required to represent the largest integer value plus 1 for the sign bit.
4. **Find the Smallest Decimal Value:** Determine the smallest decimal value that needs to be represented accurately.

5. **Calculate the Total Number of Bits:** The total number of bits is the sum of the bits required to represent the smallest decimal value and the number of integer bits.

6. **Conduct Experiments:** Perform experiments by lowering the total number of bits while keeping the integer bits fixed.

7. **Evaluate RMSE:** Evaluate the Root Mean Square Error (RMSE) with respect to a golden reference (e.g., from MATLAB) and plot this against the float data type for various Signal-to-Noise Ratio (SNR) levels.

This methodology is applied in the context of Digital Beamforming, but the WLO approach remains the same across different algorithms.

11.3 RMSE Calculation in C

To calculate RMSE in C, consider the following example:

```
#include <math.h>
#include <stdio.h>

double calculate_rmse(double *reference, double *test, int length) {
    double sum = 0.0;
    for (int i = 0; i < length; i++) {
        double diff = reference[i] - test[i];
        sum += diff * diff;
    }
    return sqrt(sum / length);
}

int main() {
    int length = 100; // Example length of the arrays
    double reference[length]; // Array for golden reference data
    double test[length]; // Array for test data

    // Initialize reference and test arrays with sample data
    for (int i = 0; i < length; i++) {
        reference[i] = /* Reference data from MATLAB */;
        test[i] = /* Test data from fixed-point implementation */;
    }

    double rmse = calculate_rmse(reference, test, length);
    printf("RMSE: %f\n", rmse);

    return 0;
}
```

In this example, `calculate_rmse` computes the RMSE between the reference data (from MATLAB) and the test data (from the fixed-point implementation). This RMSE value helps in assessing the accuracy of the optimized word-length representation compared to the floating-point representation.

12 Results

This document provides an overview of IP creation using HLS, highlighting key concepts, design steps, and optimization techniques. The approach ensures efficient and effective development of hardware designs using high-level synthesis.
Some of the results (8x61):

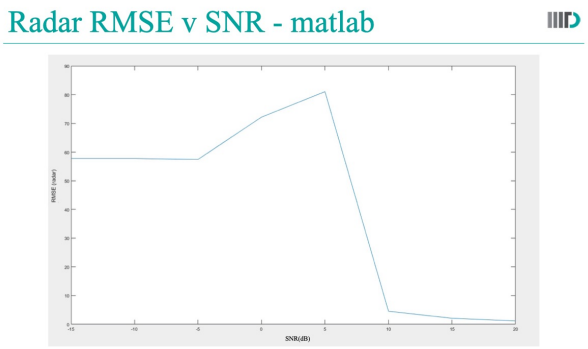

Figure 4: Matlab Radar RMSE for DB(8x61)

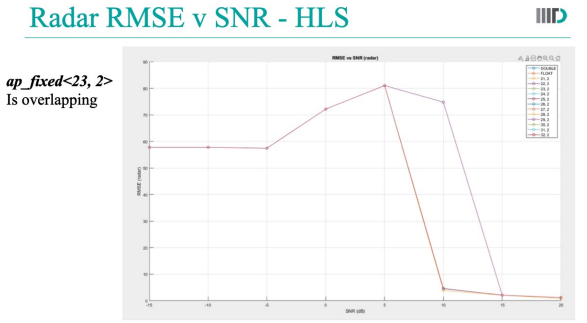

Figure 5: HLS Radar RMSE for DB(8x61)

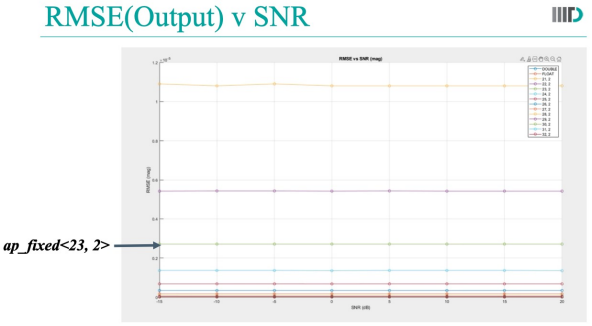

Figure 6: HLS IP Output RMSE for DB(8x)

All the results and Progress have been consolidated here

References

[1] Lectures of Advanced Embedded Logic Design (ECE573) by Dr. Sumit J. Darak, IIIT Delhi

[2] Xilinx Vivado Design Suite, `https://www.xilinx.com/products/design-tools/vivado.html`

[3] Parallel Programming for FPGAs by Ryan Kastner, Janarbek Matai, and Stephen Neuendorffer