Adding a Composable Extension for Custom Instructions to the MicroBlaze-V core

Master's thesis in Embedded Electronic System Design

ARAVIND PRASANNANPILLAI SREEVILASAM

SHAILESH SURESH VELLOLI

Adding a Composable Extension for Custom Instructions to the MicroBlaze-V core

ARAVIND PRASANNANPILLAI SREEVILASAM
SHAILESH SURESH VELLOLI

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Adding a Composable Extension for
Custom Instructions to the MicroBlaze-V core
ARAVIND PRASANNANPILLAI SREEVILASAM
SHAILESH SURESH VELLOLI

Cover: Block diagram of the design from Vivado

Typeset in LaTeX
Gothenburg, Sweden 2025

Adding a Composable Extension for Custom Instructions to the MicroBlaze-V core
ARAVIND PRASANNANPILLAI SREEVILASAM
SHAILESH SURESH VELLOLI
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This report presents the design and implementation of a Composable Extension (CX) for custom instructions in the MicroBlaze-V core, which is a customisable RISC-V core offered by AMD. The implementation is done on a Field Programmable Gate Array (FPGA) and the performance is evaluated with accelerators against the current MicroBlaze-V design.

Integration of a new CX interface allows the designer to add any number of custom instructions and accelerators according to the requirement. The accelerators can be either freshly designed or by using the existing Xilinx Intellectual Property (IP) cores with additional parameters. In this project, existing IP cores have been used as accelerators as this demonstrates how easy it is to integrate the IP cores with the interface design.

The accelerator functions were also programmed in software using C to compare and analyze the performance of the CX extension in MicroBlaze-V. Different metrics like speedup, resource utilisation and power consumption were considered to evaluate the efficiency of the entire system. A significant performance improvement has been observed with the accelerators at the expense of higher resource utilisation.

Acknowledgements

Contents

Glossary

AXI   An on-chip communication protocol used between two hardware modules. 6, 8, 9, 12, 23, 24, 26, 29, 31, 35

CF   single or a group of custom instructions. 6

CLB   A basic logic block that executes complex logic functions and implement memory functions. 3

CPI   number of cycles divided by the number of instructions. 15, 35

CPU   a hardware component that is the core computation unit in a server. 1–3, 6, 9, 21

CSR   Special register that holds control and status information in a processor. ix, 2, 5, 6, 8, 21–25, 35

CX   interface contract of a composable extension consists of custom function instructions, csr, and their behavior.. ix, 3, 6

CXU   A hardware core that implements composable extension.. 6, 7, 21, 22, 25, 26, 29, 30, 36

FPGA   A reconfigurable integrated circuit. 3, 4, 9, 17, 18, 21, 34–36

HDL   definition for all language model used to describe the behavior of hardwares. 4

IP   a reusable unit of logic or integrated circuit layout design. ix, 10–13, 15

ISA   An abstract model that defines how software controls the CPU in a computer. 1, 5, 8

register   fast memory units in the hardware that stores binary data. 3–7, 23, 24

VHDL   A language model used to describe the hardware. 4, 10, 18

1

Introduction

In the continuously evolving world of CPU architectures, the pursuit of enhanced performance and efficiency remains a constant driving force. As the demand for specialised applications rises, it turns out to be very important to have the required hardware support. This thesis project focuses on advancing processor architecture by integrating a proposed composable extension (CX) for custom instructions to the MicroBlaze-V soft-core processor.

The RISC-V processor architecture is popular for its openness and flexibility [1]. The custom extension definition in the RISC-V core is aimed at optimising the execution of specialised tasks and accelerate the tasks using dedicated hardware. However, the custom opcode space in RISC-V is unmanaged and this makes it difficult to create an ecosystem where different parties can publish and exchange their own composable custom instruction. An open composition like this requires routine and robust integration of elements authored by different parties into a stable system that can work together as a unit.

This thesis work involves not only the research, design, implementation, and evaluation of the proposed CX extension, and its integration on the RISC-V core, but also incorporating dedicated accelerators to offload and accelerate critical tasks in the system, such that it follows AMD's design philosophy which has embraced RISC-V as a foundation of its processor designs. Through a thorough research study on the RISC-V Instruction Set Architecture (ISA), our work intends to identify key areas of improvement and devise suitable solutions for the same, that will consider the energy efficiency of the overall system performance. This thesis hopes to contribute valuable insights to the customisable processor architectures, thereby ensuring that AMD's innovation technology remains a front-runner in the computing sector.

1.1   Related Work

The reserved opcodes for the custom instructions in the RISC-V architecture enable integration of the CX extension. The design of the CX extension in this thesis work is based on a proposed RISC-V Composable Custom Extension specification in [2]. The specification document defines the procedure and the process flow for handling the custom instructions. The document serves as a foundation for the thesis work as it explains the relevance of CX extension, the parameters required, and how it functions as a whole system. It describes how the instructions can be encoded and how they can be interfaced to communicate with the processor, which is followed

in this thesis work. This specification document also speaks about the required Control and Status Registers (CSRs) and the logic interface to connect and control the processor core and the accelerators.

Several researchers have worked on running hardware accelerators in the RISC-V core. In [3], a convolution neural network has been implemented based on RISC-V architecture using the reserved opcode space for the custom instructions. The CORDIC based hardware accelerators were implemented on the VexRiscv CPU core which is a 32-bit RISC-V chip [4] and the performance and resource utilisation have been assessed, similar to the metric analysis followed in this project. These works provide a significant contribution to the design and implementation phase of our thesis work.

1.2 Purpose and Goal

In order to define a standard extension for the new instructions, it must be of general interest, broad utility and non-proprietary. Usually, defining a new standard extension is a long process managed by RISC-V International. The RISC-V architecture allows independent vendors to define their own CX extension. Sharing the custom opcode space for the custom instructions in a 32b processor is critical and it obviates the need to transition to a higher-bit processor in some settings.

In this thesis work, a CX extension interface will be designed and integrated into the MicroBlaze-V soft-core processor for managing the custom instructions. Potential hardware accelerators will be plugged in through an interconnect module so that we can demonstrate the custom instructions and analyze the performance of the core. The performance and resource utilisation of the implemented design will be compared with the CPU without the accelerators, and the improvement will be noted.

1.3 Thesis Outline

In Chapter 2, all the relevant background knowledge required for the thesis will be described. In Chapter 3, the methods followed to carry out the thesis work and the tools used will be depicted. A detailed description of the design and the constraints in the design will be presented in Chapter 4. Chapter 5 will focus on the results obtained from the implementation of the design and comparison with the software model with respect to different evaluation metrics. Finally, chapter 6 covers the challenges faced throughout the course of the work and its future scope.

2

Technical Background

This section describes the specification standards used in this thesis work and the necessary background information required for the reader to understand the work done. The section begins with an introduction to Field Programmable Gate Arrays (FPGA), followed by information on pipelining and the Reduced Instruction Set Computing (RISC) architecture. In the subsequent sections, we describe more on RISC-V, the MicroBlaze soft-core, and the proposed CX extension. While detailing the CX extension, we describe the overall working of the extension, how the instructions are encoded and processed in the MicroBlaze-V and how the results and status are written back to the corresponding registers. Further, we move to the description of accelerators and the potential accelerators that we could use for our current work. We end the chapter with the metrics that we could consider for evaluation.

2.1   Field Programmable Gate Array

Field Programmable Gate Arrays (FPGA) are semiconductor devices that can be configured to meet the desired functionality or application. They contain configurable logic blocks (CLB) connected by a set of programmable interconnects which allows the designer to perform both simple and complex tasks. The FPGAs include different memory elements ranging from single-bit flip flops to very dense memory arrays, for digital storage. The FPGAs provide better performance compared to a general CPU and can be reprogrammed according to the requirement. This versatility and the re-programmable feature of FPGAs allow them to be used in various applications, including image processing, wireless communications and medical diagnosis.

The FPGA was first introduced by Xilinx in 1985 and they continue to be one of the leading manufacturers today. Apart from Xilinx, modern FPGAs which are manufactured by firms like Intel, Altera, etc. offer a large range of features like impressive logic densities, flash memory, embedded processors and digital signal processing (DSP) blocks [5]. The FPGAs can be configured by altering electrical inputs and outputs and figuring out how each resources are used and connected to form the hardware design. However, in software perspective, FPGA designs can be streamlined by using pre-designed libraries of digital circuits and functions, also known as intellectual property (IP) cores. Often, third-party suppliers and FPGA vendors offer these libraries, which are available for purchase or lease, which is the case with AMD IP cores.

FPGAs can be programmed by loading a bitstream which describes the configuration blocks like the lookup tables, registers and other blocks. The bitstream is generated by a tool called Vivado [6], where the program is written in Hardware Description Language (HDL), for example, Very High-Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL).

2.2   Pipelining

Pipelining is a technique used in digital systems, particularly in processor design, to enhance the performance by allowing multiple instructions to be processed simultaneously rather than waiting for one instruction to complete execution before starting the next [7]. The overall process can be divided into stages and each stage in the pipeline does a specific task. Every instruction enters a stage once the previous instruction has been completed. This helps in more efficient usage of resources and increases the overall throughput.
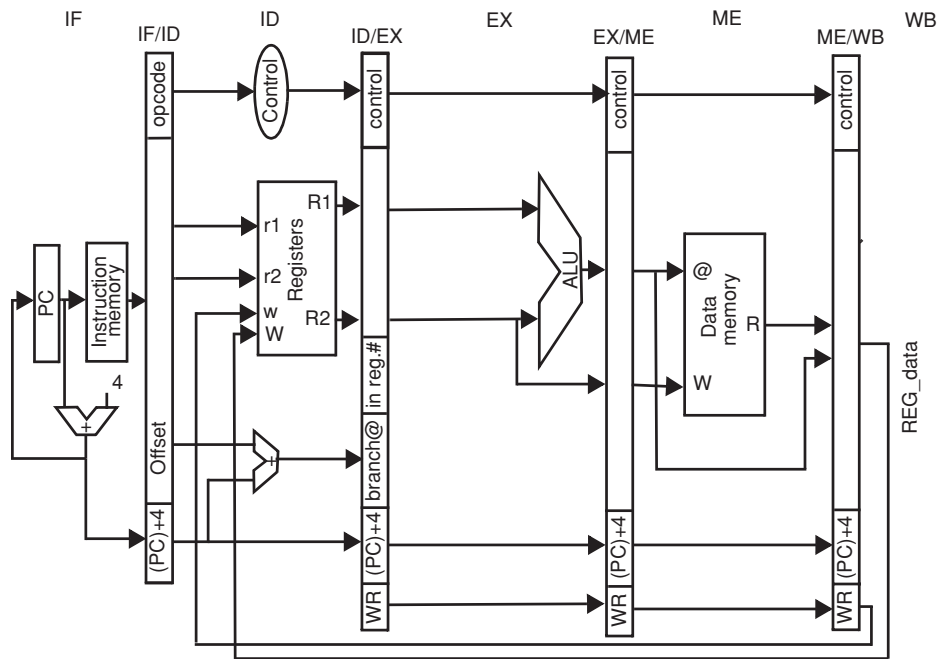
Figure 2.1: Five-stage processor pipelining [7]

The typical stages as shown in a 5-stage pipeline as seen in figure 2.1 include :

1. Fetch: In this stage, the instruction is fetched from the memory, and the program counter where the address of the next instruction is stored, is incremented.

2. Decode: Here, the opcode of the instruction is decoded and the corresponding function to be done is established.

3. Execute: In this stage, the actual operation is carried out.

4. Memory Access: In this stage, the memory is accessed to load or store the data, if it is a memory operation.

5. Write Back: The result of the operation is written to the register in this stage.

However, when multiple instructions are executed at the same time, data hazards could occur due to the data dependency between the instructions. In addition to this, instructions like jump and branch could also introduce control hazards since they disturb the order in which the instructions are expected to be executed. Advanced algorithms like branch prediction and out-of-order executions are implemented in modern processors to mitigate these challenges [7].

2.3   RISC-V

The RISC-V is an open-source ISA based on the RISC model. It was developed by Krste Asanovic, Andrew Waterman and Yunsup Lee of the University of California at Berkeley in 2010 [3]. The RISC-V architecture has evolved from RISC which operates at a very high speed, integrating pipelining in its operations. It has become very popular due to its openness and flexibility and is used in small embedded processors to high-end processor configurations. It is designed as a modular ISA composed of a small set of standard instructions and a set of extensions that can be added according to the needs of the developer. RISC-V provides both 32-bit and 64-bit instruction sets, and also various extensions like floating point, multiply and accumulate, vectors, etc [8].

MicroBlaze-V

AMD makes use of the RISC-V architecture in their MicroBlaze-V soft-core processor. The MicroBlaze-V processor offers a wide variety of customisable and easy to integrate microprocessor configurations based on the RISC Harvard model. It is used in many areas including the medical industry, automotive industry, and communication markets due to its flexibility. The MicroBlaze-V core is fully embedded into AMD's Vivado tool which makes it easier to use its functionalities in our designs.

The MicroBlaze-V soft-core processor is a 32-bit processor, which implies it consists of thirty-two 32-bit general purpose registers, a 32-bit Program counter, and a 32-bit address bus [9]. It includes a standard set of instructions and CSRs but its flexibility allows us to customise and add more instructions according to the requirements of the developer. The architecture of the MicroBlaze-V core is shown in figure 2.2.
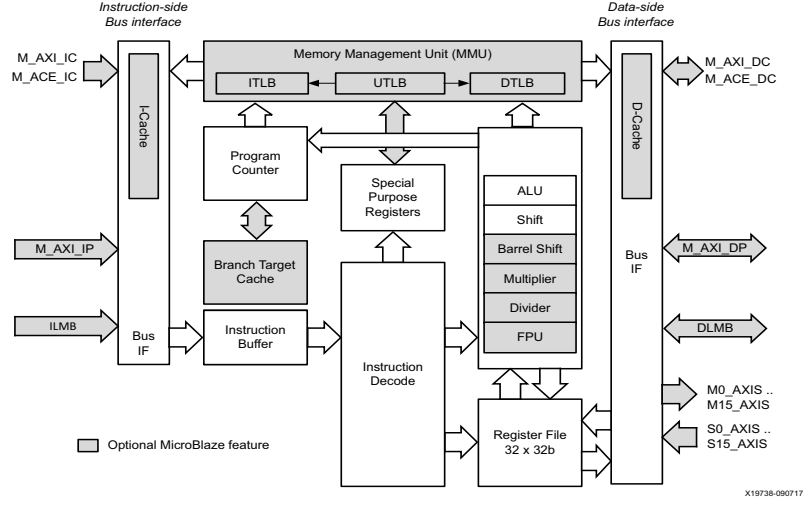
Figure 2.2: MicroBlaze Architecture [10]

AMD has a few Fast Simplex Logic (FSL) custom instructions that enable data in and out of the processor through the AXI4- Stream interface but the need for a CX extension is in demand. The focus of this work is to integrate a Composable Extension (CX) to the current MicroBlaze-V soft-core.

2.4 CX Extension

The CX extension is a group of named custom functions that bridge between software and hardware, enabling the software libraries and hardware cores that implement the same extension. The CX multiplexing enables the composition of a system of individually authored and versioned components [2]. The Custom Functions (CF), identified based on CF_IDs are executed using Composable Execution Units (CXUs), which are hardware units identified using unique CXU_IDs. Each CXU operates on the operands from register/immediate values and writes the result back to the destination register along with updating the status in the CSRs. Additionally, CSRs also contain the state_id which represents the state context of the CXU. The CXU logic interface defined between the CXUs and CPU manages the control flow of each instruction and its corresponding result.

2.4.1 CX standard encoding

When a particular CXU is selected, the software issues custom functions to the configured CXU using different types of instruction encodings: R-type, I-type, and flex type. For each encoding type, the instruction specifies the CF_ID, source operands (register or immediate) and possibly a destination register in the encoded format.

Custom-0 R-type encoding

This type of instruction encoding has two source register operands, a destination register, and CF_ID of 10 bits as given in figure 2.3.

The assembly instruction can be written as : `cx_reg cf_id, rd, rs1, rs2`
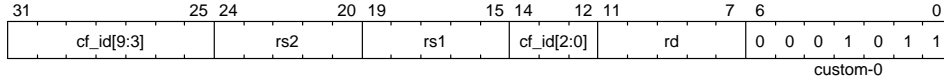
Figure 2.3: CX R-type instruction encoding [2]

Custom-1 I-type encoding

This type of instruction issues a CXU request for a CF_ID with 4 bits (zero padded) having one source register operand and one immediate operand, and the result is written back to the destination register.

The assembly instruction can be written as : `cx_imm cf_id, rd, rs1, imm`

The instruction encoding is shown in figure 2.4.
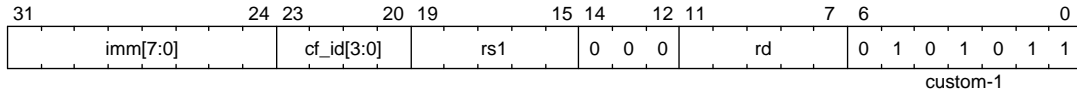
Figure 2.4: CX I-type instruction encoding [2]

Custom-2 flex type encoding

This instruction issues a CXU request for a 10-bit zero padded CF_ID with two source register operands. No destination register is involved in this operation and the response data is discarded.

The assembly instruction can be written as : `cx_flex cf_id, rs1, rs2`

The instruction encoding is shown in figure 2.5
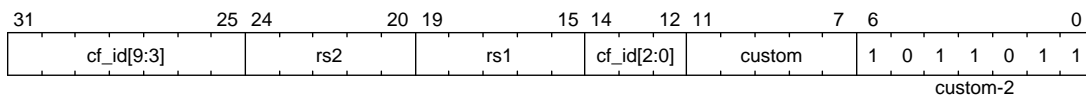

Figure 2.5: CX flex-type instruction encoding [2]

2.5   Control and Status Registers (CSRs)

The Control and Status Registers (CSRs) are special registers in a processor that monitor and manage the operation of a system. The CSRs store the control and status information of different units of the processor. They help the software to set the parameters, initiate and control the operations in the processor. They also provide feedback about the current state of the processor and its hardware components, with the help of flags or status bits. A few examples are interrupt mask registers, interrupt status registers, error status registers, etc.

The CSRs are crucial for debugging and understanding the state of the system as they provide information about what the processor is doing at the given moment. The CSRs control access to critical resources by enforcing certain privilege levels, thus ensuring the safety and integrity of the system. Fine-tuning these CSRs can optimise the performance of the system by enabling or disabling certain hardware features.

The CSRs are handled in specific privilege levels, mainly in the machine mode, where you have full access to all the controls in the processor. The CSRs are accessed through specific instructions provided by the ISA of the processor. The MicroBlaze-V core has some standard CSRs implemented in it. A few custom CSRs like `mstatus` (describing the Machine Status), `misa` (describing the supported ISA) are also defined already in MicroBlaze-V [9].

2.6   AXI4-Stream Interface

The Advanced eXtensible Interface (AXI) Stream protocol can be used as a standard interface for exchanging data between components connected to each other [11]. It facilitates high speed and efficient communication of data and this feature makes it useful for this thesis work.

The AXI-Stream Interface conforms to the ARM AMBA AXI4-Stream Protocol Specification [11]. It works by the principle of handshaking between the transmitter and receiver. The transmitter is treated as the Master and the receiver is treated as the Slave. The AXI4-Stream interface makes use of mainly three signals :

1. TVALID: sent out by the Master indicating that the datastream is valid and it wants to send the data.

2. TREADY: sent by the Slave device indicating it is ready to receive the data.

3. TDATA: The datastream sent out through AXI Stream from the source to the receiver.

The data transfer happens only when both TVALID and TREADY signals are asserted irrespective of the order in which they are set as shown in figure 2.6. It is possible to omit TREADY in some cases where the receiver can always accept a transfer. In such cases, TREADY is always assumed to be HIGH.
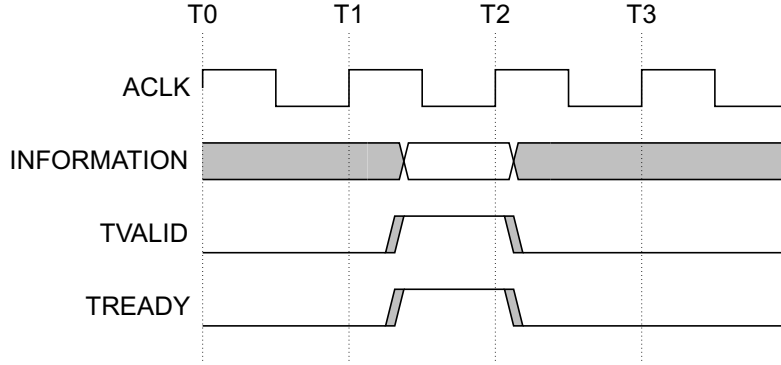
Figure 2.6: Timing Diagram of AXI4-Stream Data Transfer

In addition to these, a signal called TLAST can be configured in some cases to indicate the packet boundaries. Asserting the TLAST bit indicates the final transfer of an operation and no more bits should be followed after it.

2.7 Accelerators

The accelerators are used to accelerate a given task, which otherwise would require a considerable amount of time to execute. The accelerators function as the hardware units in our project, executing the custom function and writing the result and error status back to the processor.

A system composed of a processor and a hardware accelerator enhances the software programmability for much of the software run on the processor while improving the power and performance for the functions run on the hardware accelerator. Since the hardware accelerators are designed specifically to handle a particular function, their resources are rightly matched for the precision of the operation, rather than the typical processors which offer resources in bits of the multiple of 8. The hardware accelerators also reduce the performance time by covering the execution time required for branch prediction, data caching and other complex operation involved in modern processors. Moreover, as the hardware accelerators provide only the adequate amount of hardware resources required to perform the given operation, it also reduces the amount of power consumption involved in executing a stream of instructions.

FPGA based accelerators are quite convenient due to their flexibility and adaptability. They can be reprogrammed to run any function easily. Moreover, making use of an FPGA minimizes the latency and power consumption compared to using a CPU [12]. In this project, the accelerators are utilised to demonstrate the newly integrated extension.

2.8 Intellectual Property (IP) Core

An Intellectual Property (IP) core is a standalone reusable functional logic unit that performs a complex task and can be used in several digital designs. These are developed using hardware description languages like Verilog, VHDL etc. There are several IP cores in the Xilinx library available to be used as a plug-in and play module and a few of them can be used in our design.

2.8.1 CORDIC

CORDIC stands for Coordinate Rotational Digital Computer. This algorithm was initially developed by Volder to solve trigonometric equations in an iterative fashion, and was generalized later by Walther to include more functions like the hyperbolic and square root equations [13].

The CORDIC core can be configured in two ways in terms of its architecture.

- Fully parallel configuration with single cycle data throughput: In this case, the CORDIC core implements the operations in parallel using an array of shift-addsub stages.

- Word serial implementation with multiple cycle throughput: In this stage, the shift addsub operations are performed serially using a single shift addsub stage in a feedback loop.

The CORDIC IP core can be used to implement many different mathematical functions including trigonometric functions, square root, hyperbolic and also rectangular-polar conversions. The respective function can be configured in the IP core and the operands can be selected accordingly whether it is a phase or cartesian operand. The pin diagram of the CORDIC IP core is shown in figure 2.7.
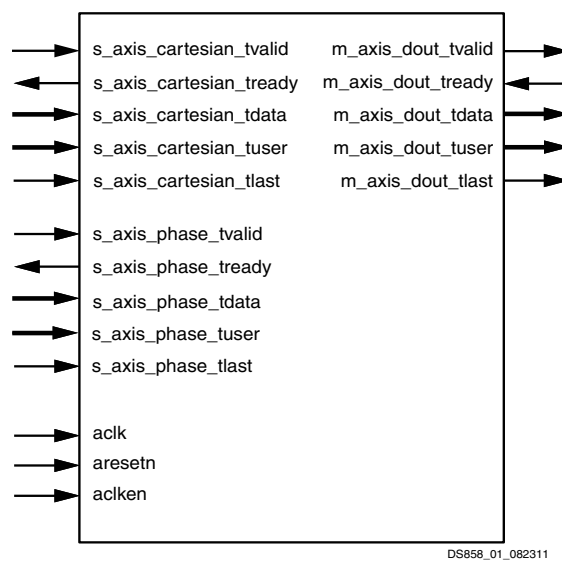

Figure 2.7: Pin Diagram of CORDIC IP core [13]

.

Trigonometric Functions

The trigonometric functions like sine, cosine, tangent, etc. have a wide range of applications in our day-to-day lives. The applications range from astronomy where it is used to find the distance of Earth from other planets and stars, to navigation, construction sites, marine engineering etc.

The CORDIC IP core calculates the sine and cosine of the given phase value (in radians). Since we need the phase value as input, only the phase input parameters are enabled in this case. The coarse rotation module in the IP core limits the value of input angle between $-\pi$ and $+\pi$. The input angle is expressed as a fixed-point two's complement number with an integer width of 3 bits while the output vector is expressed as a pair of fixed-point two's complement numbers with integer width of 2 bits [13]. The IP core gives both the sine and cosine values of the phase value in a single output vector of 32 bits with the lower 16 bits representing sine and the higher 16 bits representing cosine. The output format is given in figure 2.8.

31	16 15	0
COSINE		**SINE**
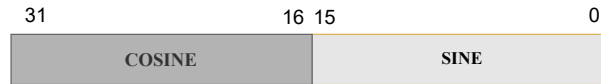
Figure 2.8: Output vector format of sin and cos function in CORDIC
.

Vector Translation

The vector translation function to convert rectangular to polar coordinates is used in different areas which include navigation, robotics and signal processing.

The CORDIC IP core performs the vector translation operation where it rotates the input vector around the circle in an angle θ until the Y component equals zero. The scaled magnitude and phase of the rotated input vector are obtained as outputs. In this case, both the Cartesian and phase input parameters are enabled. The vector translation shows linear behaviour with respect to magnitude. The number of significant magnitude bits of the input vector limits the accuracy of the phase output from CORDIC. The input and output representation is similar to the trigonometric function where the output magnitude is expressed as fixed two's complement numbers with integer width of 2 bits and the phase angle is expressed as fixed two's complement number with an integer width of 3 bits. The output vector format is similar to the trigonometric function with 32 bits but with the magnitude representing the lower 16 bits and phase value representing the higher 16 bits. The output vector representation is shown in figure 2.9.

31	16 15	0
PHASE		**MAGNITUDE**
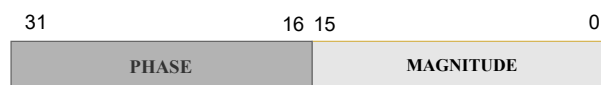
Figure 2.9: Output vector format of Translate function in Cordic
.

The CORDIC IP core makes use of the AXI4-Stream Protocol for sending in inputs and sending out output signals. It works by basic handshaking between `tvalid` and `tready` signals. The AXI4-Stream interface can be configured in CORDIC in Non-blocking as well as Blocking Mode. The Non-blocking mode does not have a `tready` signal and is always assumed to be asserted by default. On the other hand, the Blocking mode has a `tready` signal present and this helps to control the data flow through the core, making sure the output buffer is not overloaded with data.

2.8.2 Fast Fourier Transform (FFT)

FFT is a computationally efficient algorithm for computing the Discrete Fourier Transform (DFT) of a signal [14]. The FFT IP core provided in the Xilinx library uses the Cooley-Tukey FFT algorithm [15] for calculating the DFT. The Cooley-Tukey FFT algorithm is one of the most widely used methods for calculating the DFT.

The DFT $X(k), k = 0, ...N - 1$ of the sequence $x(n), n = 0, ...N - 1$ is defined as (2.1), where N is the transform length. The Inverse Discrete Fourier Transform (IDFT) is given by (2.2).

$$X(k) = \sum_{n=1}^{N-1} x(n)e^{-jnk2\pi/N} \tag{2.1}$$

$$x(n) = \frac{1}{N} \sum_{k=1}^{N-1} X(k)e^{jnk2\pi/N} \tag{2.2}$$

Cooley-Tukey FFT algorithm

The Cooley-Tukey algorithm decomposes the DFT for larger sizes into sub-components and performs the DFT. This divide-and-conquer strategy significantly reduces the computational complexity of the DFT from $O[N^2]$ to $O[NlogN]$. The radix-2 algorithm, the most common variant of the Cooley-Tukey algorithm, recursively decomposes the DFTs into smaller DFTs of half the size until the computations can be performed directly.

Xilinx FFT IP core

The FFT IP core has two different types of input signals for input and config instructions and one output signal to provide the output. The pin diagram of the FFT IP core is given in figure 2.10.
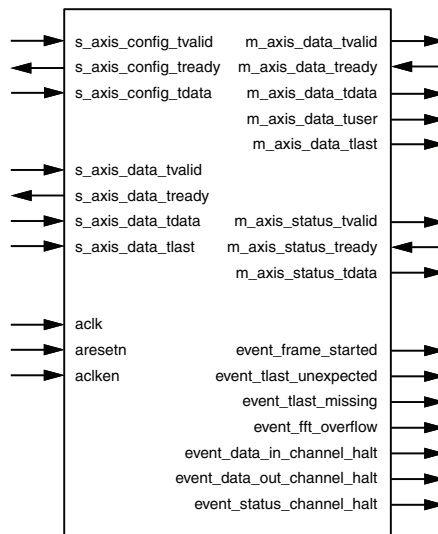
Figure 2.10: Pin Diagram of FFT IP core. [16]

The Xilinx FFT IP core provides four different types of architecture to implement the FFT computations. These architectures process the data as continuous streamlining, pipelined streaming I/O architecture or independent data frames and burst I/O architecture. In the pipelined streaming solution, the Decimation in Frequency (DIF) method has been used for FFT computations and several radix-2 butterfly processing engines implement continuous data processing. Each processing engine has independent memory units to store the input and the intermediate result. The burst I/O architecture uses the Decimation in Time (DIT) method for the FFT computation and is implemented as radix-4, radix-2, and radix-2 lite architectures. In the burst I/O architecture, the FFT core loads the data separately followed by computing the transform and unloading the result. The radix-4 and the radix-2 architectures use the radix-4 and radix-2 butterfly structures for the computations respectively. The radix-2 lite architecture uses the same butterfly processing engines as the radix-2 but uses the same adder/subtractor blocks for the computations [16]. The Radix-2 Burst I/O architecture is illustrated in figure 2.11.

The transform length will determine the number of stages for the FFT algorithm. The transform length can be configured during runtime and in the design stage of the FFT IP core. The configuration port in the core enables the selection of FFT and Inverse Fast Fourier Transform (IFFT) as well as the scaling performed in each stage.

The streaming I/O architecture would give the highest throughput by utilising the most resources while the radix-2 lite burst I/O architecture would have the least throughput. The performance of each architecture is given in figure 2.12.

13

Figure 2.11: Radix-2 Burst I/O architecture [16].

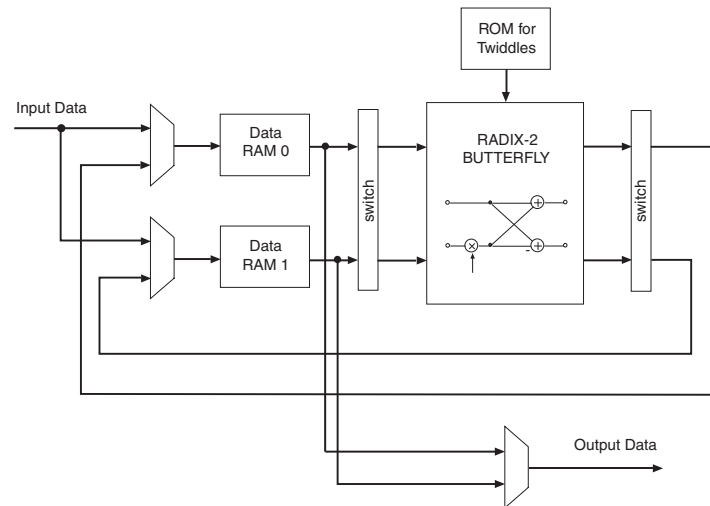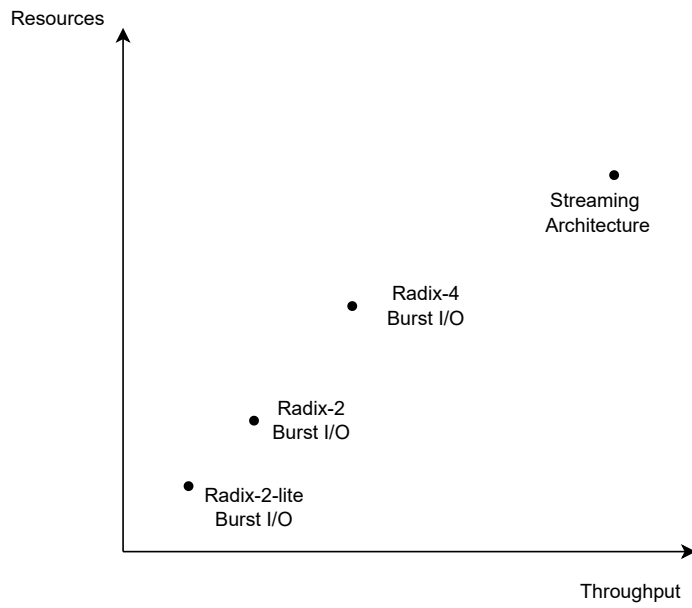**Figure 2.12:** Resource versus throughput for different architecture types.

The data format of the input, output, and configuration instructions are shown in figures 2.13.

(a) Input and Output instructions

(b) Config instruction
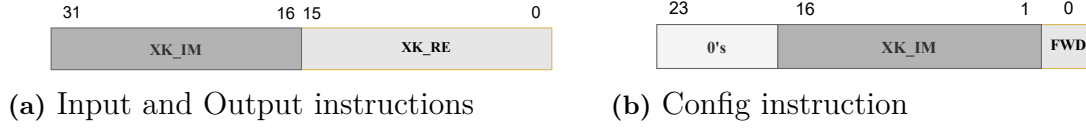
Figure 2.13: Data format of instructions in FFT IP core

Here, XK_IM and XK_RE represent the real and imaginary parts of the data while the FWD bit represents whether a FFT or IFFT should be performed.

2.9 Evaluation Metrics

Performance always comes into importance when we integrate anything into an existing processor design. It is of interest to assess the benefits of running a certain function in the hardware with the help of a processor, rather than running the same function in software. The performance of the processor and the hardware usage can be evaluated using various metrics. Some of the commonly used metrics are given below:

1. **Cycles Per Instruction (CPI)** - This denotes the average number of clock cycles that each instruction takes to execute on the machine [7].

$$CPI = \frac{T_{exe}}{IC * T_C} \qquad (2.3)$$

   where T_{exe} is the execution time, IC is the instruction count and T_C is the machine cycle time.

2. **Speedup** - The speedup of a machine over a reference machine is defined as the fraction of the execution time that a program takes to run in the reference machine to the execution time that the same program takes to run in the current machine.

$$CPI = \frac{T_{exe,Ref}}{T_{exe}} \qquad (2.4)$$

   where $T_{exe,Ref}$ is the execution time of the program on the reference machine and T_{exe} is the execution time of the program on the machine that is being evaluated [7].

3. **Resource utilisation** - This denotes the amount of hardware usage for executing a program in the specified machine.

4. **Power Consumption** - This denotes the amount of power consumed (both static and dynamic) for executing a program in the specified machine.

3

Methods

This chapter describes the overall process flow of the thesis work. This is followed by an overview of the tools used for the design and implementation of the prototype and to test the entire module on an FPGA.

3.1 Workflow

The first step in our thesis work was to understand the MicroBlaze-V core and the relevance of integrating a CX extension to it. The initial few weeks of the work were invested in literature survey on custom instructions and the proposed specification for the CX extension. Once we had a good understanding about the work to be done, we started by designing a block diagram of the entire system. We designed an interface for the CX extension on paper that would be implemented on RTL and later integrated into the MicroBlaze core. We also did some surveys in parallel to select the accelerators that could be used for demonstrating the CX extension.

After the design has been done on paper, we proceeded to work on the RTL implementation of the selected accelerators and testing them to function as a unit using a testbench. Further on, we moved to developing the RTL for the design and verifying its behaviour and timing using Vivado. The design was implemented using Vivado after verification along with checking the power and the timing constraints to make sure that the requirements were met. Later, the newly developed interface was integrated to the existing MicroBlaze-V core and the same process of RTL design and verification was repeated. After the bugs and timing issues were fixed, we implemented the entire core design in the KCU105 FPGA to test the behaviour on hardware.

In parallel to the RTL design, we also worked on the software implementation of the accelerators using C. The C program for finding sine and cosine, magnitude and phase of the vector and FFT were written and built in Vitis and later simulated using Questasim. In order to record the output results, we logged the results from the waveform to an output file via UART by using a TCL script. The execution time of all the three programs were noted so that it can be compared with the execution time of the acceleration with CX-enabled MicroBlaze-V. The resource utilisation of the core with and without the interface and accelerators were also reported for further comparisons.

The entire plan of the project was prepared on an excel sheet with all the tasks and sub-tasks and the person assigned to work on each specific task. In addition to

this, we had weekly sync meeting with our advisors at AMD every Tuesday where we reported our work done in the previous week and discussed our plans on moving forward.

3.2 TestBench

A top-level test bench was designed for the functional verification of the CX interface along with the interconnect and accelerator modules. The test bench was designed to imitate the software libraries which supply the instructions. The test vectors are generated using a Python script by concatenating the input of the IP core along with the CXU_ID and CF_ID. The input to the IP core was extracted from the inbuilt test bench of the IP core library. The test also features a self-checking system where the computed result from the accelerators is compared against the expected result obtained from the IP core library.

The test bench was written to check for errors by itself using the assert and report statements in VHDL. The test bench design can be explained clearly using the flowchart given in figure 3.1.

3.3 Tools Used

The following tools given in table 3.1 were used during the thesis work.

Table 3.1: Software and Hardware tools used in the thesis work

| Tools	Purpose
Emacs	Text editor for the VHDL code.
QuestaSim	To perform functional verification of the design.
Vivado	To implement design on FPGA.
Vitis	Environment to test the functionality in software.
Python	Used to generate the test vectors for the testbench.
Xilinx Kintex Ultrascale KCU105	FPGA board used to run the program.
Latex	For documentation
Inkscape	For sketching
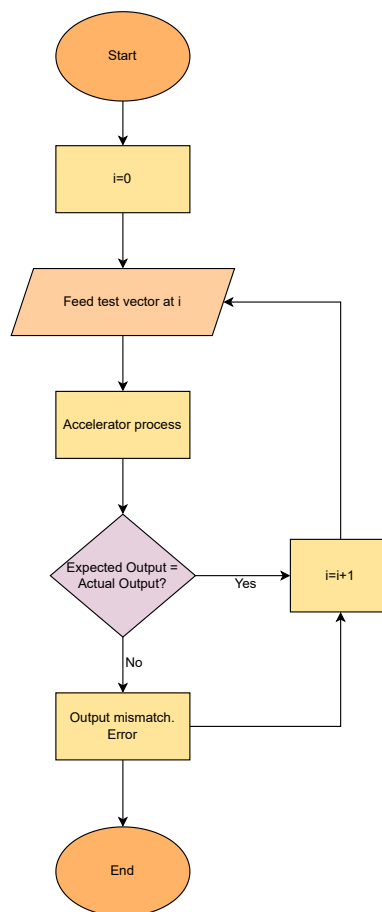
Figure 3.1: Top-level TestBench Design Flow

4

Design

In this section, the design of each component facilitating the implementation of the CX extension will be described in detail including the process flow. First, an overview of the design will be introduced in Section 4.1 followed by the modifications in MicroBlaze-V including the pipeline flow, design of CX interface, and additional CSRs required for the implementation of CX in Section 4.2. In Section 4.3, the Interconnect that acts as a bridge between the CPU and accelerators will be described in detail. Section 4.4 introduces the design of the implemented accelerators and last, the reference model used for the performance evaluation will be described in Section 4.5.

4.1   Overview

Implementing the CX extension in the RISC-V architecture involves modifications to the MicroBlaze-V core and integrating software libraries and hardware modules. These modifications to the core include the definitions of an additional hardware module (CX interface), definitions of new CSRs, and required changes in the decode and write-back pipeline stages to handle the execution of custom instructions. Custom instructions required to execute a custom function are fetched with the help of software libraries. Software libraries generate and feed the instruction code to the IF stage in the RISC-V architecture. When custom instructions are decoded, the CX interface handles their execution with the help of external accelerators.

AMD's support for the MicroBlaze RISC-V processor allows compiling the hardware specification defined in Vivado with the software code built on the Vitis IDE. The Electronic Design Automation (EDA) support in Vitis allows exporting the current hardware design as a platform, and building applications. This application helps in generating the required instruction codes using inline C assembler in elf format. By importing the generated elf file to the Vivado environment, MicroBlaze-V can be implemented on FPGA along with the custom instructions. These generated instruction codes will be stored in the external memory (LMB RAM) of the MicroBlaze-V. While running the design on hardware, each instruction will be fetched in each clock cycle and executed by the MicroBlaze-V processor.

To demonstrate the CX extension, the FFT and CORDIC function accelerators are considered as CXUs. Unique CXU_IDs will be defined for the accelerators along with function IDs (CF_ID) for each custom instruction. These defined IDs are fetched during execution through the custom instructions and CSRs. As shown in

figure 4.1, the CXU_ID will be fetched from the CSR called mcx_selector while the CF_ID will be fetched from the instruction itself.
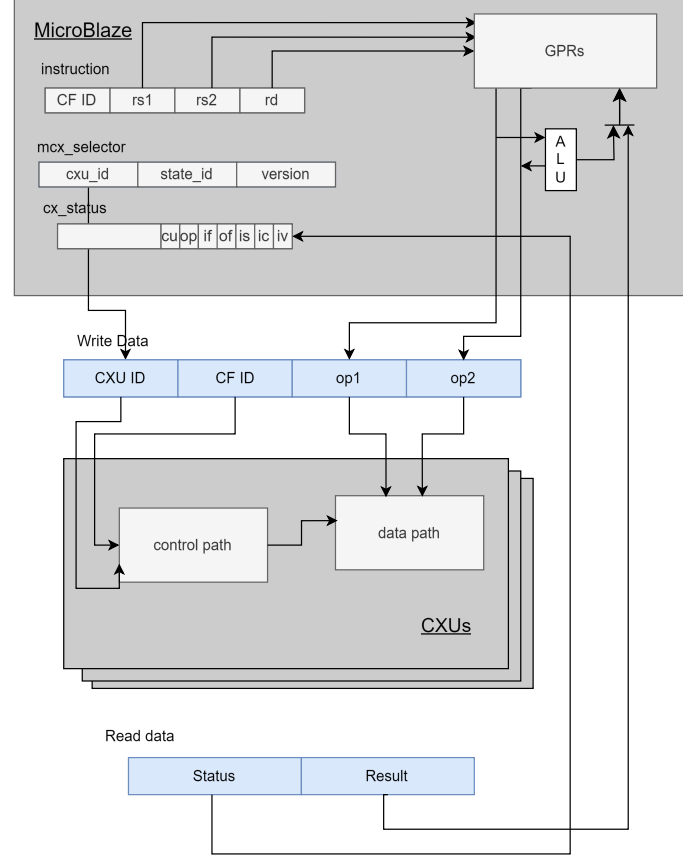
Figure 4.1: Block diagram of the overall design.

As the accelerators considered do not have any state context, state ID is ignored. This information along with the operand values will be sent to the CXU after verifying the version compatibility. When the CXU completes its execution, the result along with the status will be written back to the MicroBlaze core.

4.2   MicroBlaze-V

In this thesis work, the MicroBlaze-V core has been modified for the implementation of the CX extension. In order to handle the execution of the custom instruction, a hardware module called CX interface has been designed and integrated into the processor core. In order to support this new module, some of the other components also need to be modified. This includes integrating the new CSRs and updating the decoder module to support the custom instructions. Figure 4.2 shows the architecture of the modified MicroBlaze-V core to handle the CX extension.
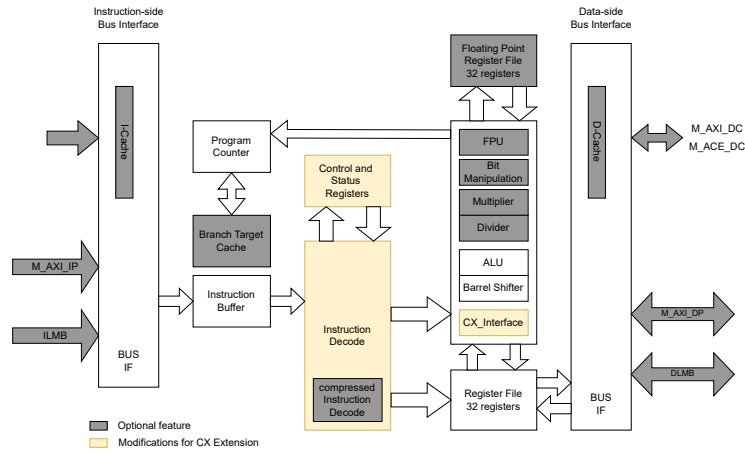
Figure 4.2: Architecture of the modified MicroBlaze-V core. The highlighted modules in yellow show the blocks that are modified to integrate the CX interface.

4.2.1 CX interface

The CX interface designed in MicroBlaze-V handles the execution of custom instructions through a series of operations shown in figure 4.3. When the ID stage decodes the custom instruction, the operand values and the CF_ID from the instruction code along with the CXU_ID and version from the CSRs will be forwarded to the CX interface. The CX interface fetches this information and compares the version in CSR with the MicroBlaze version. If the versions match, then the operand values along with the identifiers (CXU_ID, CF_ID) will be encoded and transmitted via the AXI stream interface. In case of an incorrect version, the `IV` flag in the CSR will be updated and the execution of the custom instruction will be dropped. After the transmission of data, the MicroBlaze-V will wait for the result and the pipeline will be stalled.
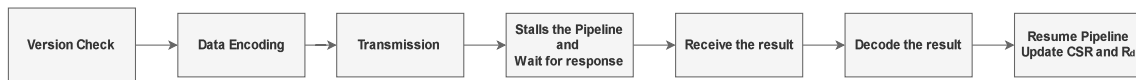
Figure 4.3: Workflow of the CX interface

The functional block diagram of the CX interface is shown in figure 4.4. The execution of a custom instruction is controlled by the EX_CX_Instr register, where the signal will be asserted while decoding a custom instruction. CX_EX_Firstcycle denotes the first cycle in the EX stage and helps with handshake signals in the AXI-4 stream protocol. EX_CX_Write determines whether the custom instruction expects a result or not. If the value of this signal is low during the execution of a custom instruction, the CX interface will consider the instruction as a Write instruction and will not wait for the response. EX_Piperun indicates whether the current stage of the pipeline is in the EX stage or not. EX_CX_Stall flag is used to stall the pipeline and EX_CX_Result contains the result of the execution. The EX_MCX_Sel and EX_New_MCX_Status are the CSRs designed for the CX extension and the EX_Write_MCX_Status flag is used as an enable signal to write

to the CSRs. The CX interface communicates with the accelerators via the AXI-4 stream protocol. The master interface helps to write the data and the slave interface helps to read the data. The control and data signals of the master and slave AXI interface can also be seen from the block diagram.
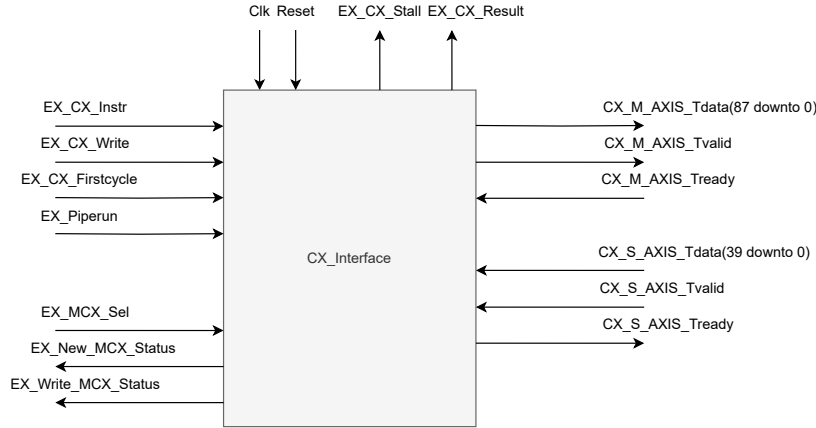
Figure 4.4: Design of CX interface with signals

As per the custom instruction encoding format, the destination register address will not be present for the custom-2 flex type encoding type. Therefore the custom-2 flex type instructions are considered as write instructions where the MicroBlaze core would not expect a result. In the case of write instructions, the pipeline will be stalled until the transfer of payload in the master interface of the AXI-4 stream has finished. The execution of each instruction for a CX extension might require more than one clock cycle. Therefore, stalling the pipeline would be critical as it can lead to various pipeline hazards. In this design, the pipeline is stalled in the decode stage during the execution of a custom instruction. The pipeline will be resumed when the custom instruction finishes its execution.

The AXI master and slave interfaces have been designed for the transfer of the information from the MicroBlaze-V to the accelerators. The required parameters needed for the execution of the custom instruction will be encoded as shown in figure 4.5 and transferred to the accelerators via the master interface. Correspondingly, the executed result of the custom instruction along with the status of execution will be read by the MicroBlaze through the slave interface. The received information, shown in figure 4.6 will be decoded and the destination register and the status CSR will be updated.
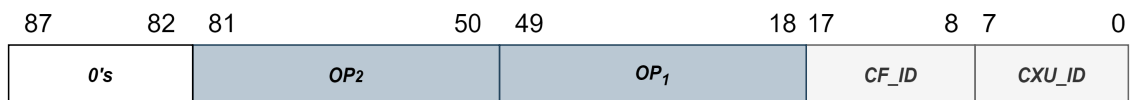
Figure 4.5: Encoded datastream write by the RISC-V core
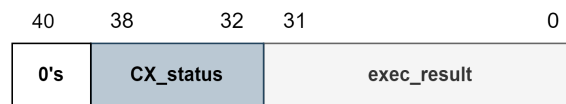
Figure 4.6: Encoded datastream read by the RISC-V core

4.2.2 CSR

The implementation of CX requires to define additional CSRs for extension multi-plexing and custom instruction execution. For the execution of a collision-free in-struction, the compatibility of MicroBlaze-V IP core as well as accelerators need to be verified. As the execution of a custom instruction follows the specified procedure, verifying the status of the execution of a custom instruction is also important.

1. mcx_selector - The mcx_selector CSR enables CX multiplexing and allows the developer to select the corresponding CXU required to run the particular instruction. It can be read or written only in the machine level. The format of this CSR is given in figure 4.7.
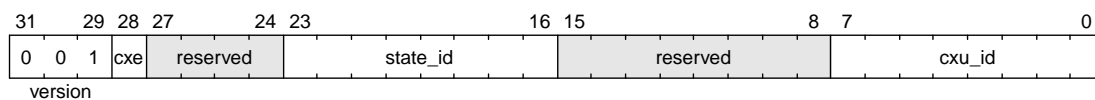
Figure 4.7: Definition of mcx selector CSR

Here, cxu_id field identifies the corresponding CXU and the state_id field identifies the corresponding state.

2. cx_status - This CSR accumulates the CXU error flags and it may be read and written in all privilege levels. All the fields of this CSR are set to 0 by the application software by default, before the execution of an instruction and then the values are updated by the CXU in case of any errors. The fields of cx_status are given in figure 4.8.
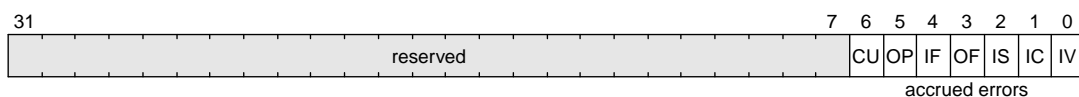
Figure 4.8: Definition of cx status CSR

The flags seen in figure 4.8 are described below.

- IV - Invalid CX version error: set when the mcx_selector version is in-valid.

- IC - Invalid CXU_ID error: set when the cxu_id provided to the mcx_selector is invalid.

- IS - Invalid STATE_ID error: set when cxu_id is valid but the state_id is invalid.

- OF - State context is off error: set when cxu_id and state_id are valid but the state context is in off state.

- IF - Invalid CF_ID error: set when cxu_id and state_id are valid but the CF_ID of the instruction in invalid.

- OP - CXU operation error: set when cxu_id, state_id and CF_ID are valid but there is an error in the requested operation or operands.

- CU - Custom CXU operation error: set when cxu_id, state_id and CF_ID are valid but there is an error in the requested operation or operands with its custom error state available.

4.3 Interconnect

The interconnect has been designed to act as a bridge between the MicroBlaze-V core and the accelerators. The interconnect has independent AXI-4 stream interfaces for communicating with the MicroBlaze-V processor and the accelerators as shown in figure 4.9. The overall execution of a custom instruction has been handled in the interconnect as two phases, the write and read phase. The write phase handles the communication from the MicroBlaze to the accelerators and the read phase handles the opposite. A side lobe channel, CXU hit has also been designed to denote the accelerators executing the custom instruction.

When broadcasting the information to all the accelerators, the interconnect need not know the header ID (CXU_ID) of the accelerators, thereby avoiding the need to have lookup tables in the interconnect module. The accelerator with the correct CXU_ID will respond to the broadcast request and the interconnect will wait for the response. If none of the accelerators respond to the request, the custom instruction will fail with a CXU error.
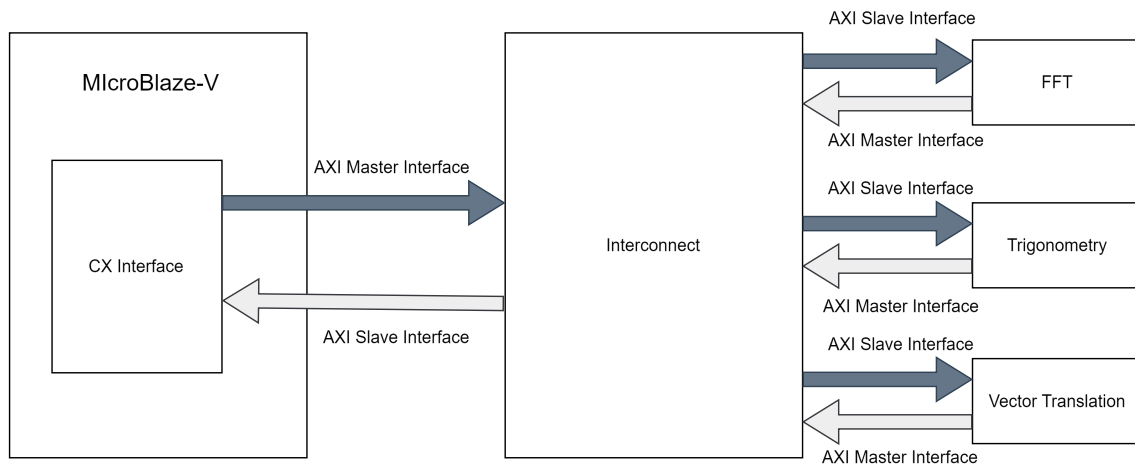
Figure 4.9: AXI-4 stream interfaces in the design

4.3.1 Write Phase

The write phase of the interconnect handled by the broadcaster, evaluator, and error handling units in figure 4.10 establishes the communication from MicroBlaze to the accelerators. During the write phase, the interconnect broadcasts the payload from the master interface of the CX interface to all the accelerators. Broadcasting the information eliminates the need to store the header IDs including the CXU ID and CF ID of the accelerators in the MicroBlaze or in the interconnect. Accelerators with the matching CXU_ID will respond to the request and indicate the interconnect with the help of the side lobe channel (CXU Hit). The error handling in the interconnect uses the CXU Hit flag from the accelerator to check for CXU Error. In case of a mismatch, the CXU error flag will be set and accumulated. Accumulating the CXU error helps to restore any previous invalid CXU errors.
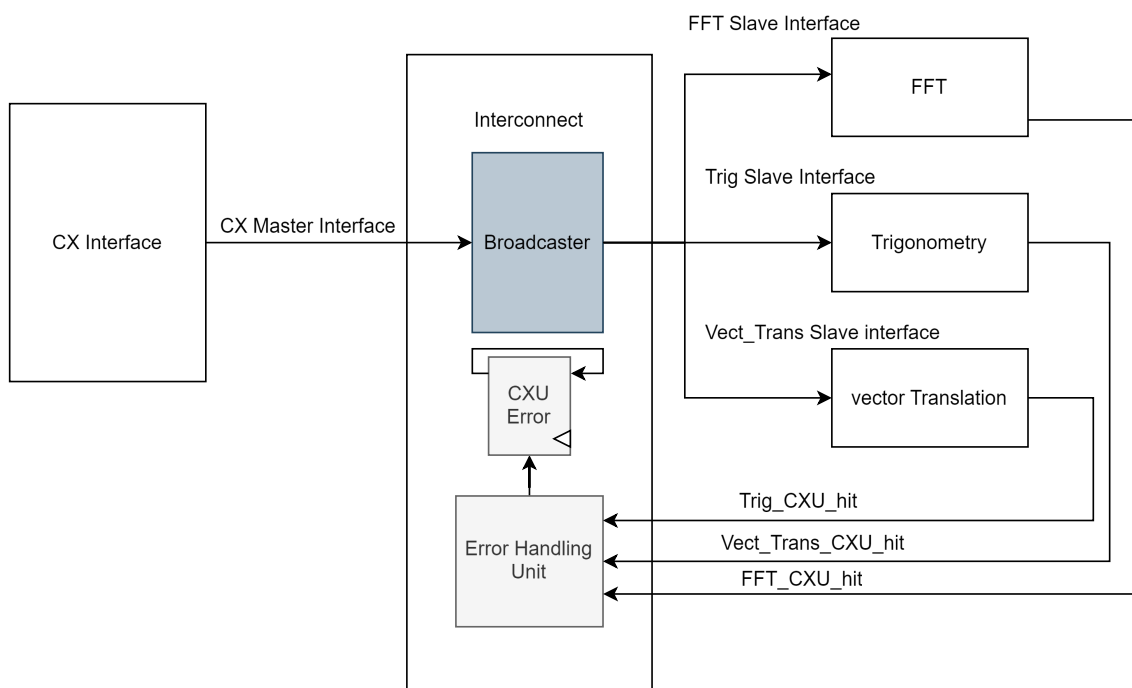
Figure 4.10: Interconnect functionality in the write phase

4.3.2 Read Phase

During the read phase, the computed result from the accelerators along with the accumulated error flag will be read by the MicroBlaze-V processor. The functions performed by the interconnect during the read phase are shown in figure 4.11. The mixer waits to get the response from the accelerators along with the status of operation and append the accumulated CXU error flag and write the information to the CX interface. During this function, bitwise OR operation will be performed for the response from the accelerators. Therefore, the non-triggered accelerators' response should be set as void or 0. If none of the accelerators are triggered, then the error handling identifies the error and writes back to the CX interface. An OR functionality has been used to distinguish between the response from the mixer function

and the error handling unit. The evaluator unit facilitates generating the response based on the instruction types (Read/Write) by analysing the ready flag in the slave interface of the CX interface. When the ready flag is high, the instruction will be identified as a read instruction where the CX interface expects a result from the accelerator. When the ready flag is low, the instruction will be identified as a write instruction where the CX interface will not expect a result.
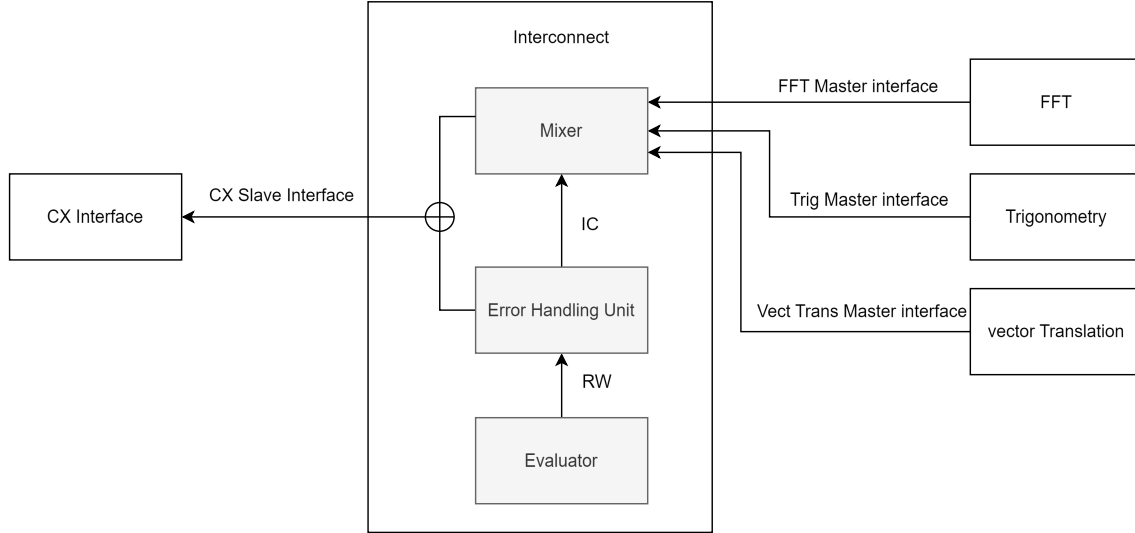
Figure 4.11: Interconnect functionality in the read phase

4.4   Accelerators

The accelerator executes the custom instruction by performing the computations based on the parameters provided and writes the result back to the MicroBlaze-V processor through the Interconnect module. The master and slave interfaces defined in the accelerator handle the reading and writing of the data respectively. Additionally, the CXU Hit channel identifies whether the custom instruction has the correct CXU_ID. Each accelerator has a unique CXU_ID which acts as the identifier for the accelerators. This thesis work considers the FFT and the CORDIC IP cores from the Xilinx library for hardware accelerations. Table 4.1 includes the identifiers for each accelerator and their corresponding function.

Table 4.1: Accelerator identifier

Accelerator	Instruction Type	CXU ID	CF ID
Trigonometry	Custom-0 R type	0x20	0b10
Vector Translation	Custom-0 R type	0x25	0b01
	Custom-2 flex type	0x30	0b01
FFT	Custom-2 flex type	0x30	0b10
	Custom-0 R type	0x30	0b11

A wrapper module, shown in figure 4.12 defined over the IP cores manages the

execution of the custom instruction. Upon receiving the `tvalid` and `tdata` signals, the wrapper module samples and checks the integrity of the incoming bit stream. If the CXU_ID is valid, the data stream will be received by asserting the ready signal with the help of the AXI slave interface and will be noted as CXU hit. In case of a CXU mismatch, the data stream will not be received and the CXU Hit signal will not be asserted. The required parameters for the IP core will be forwarded after the CF_ID check and operand check have been performed. If both checks fail, the error handling will manage and provide an error response back to MicroBlaze. The wrapper module waits until the IP core computes the result and the computed result will be encoded with the the status to generate the response back to the MicroBlaze. This combinatorial logic algorithm helps in the faster response of the accelerators. The AXI interfaces of the wrapper and the IP cores are connected with the help of the AXI master and slave interface defined in the wrapper. These interfaces facilitate the connection of the AXI handshake signals (valid and ready) and the data channel.

As mentioned before, the custom instructions are defined based on write-back. The accelerator module follows the normal process flow when the custom instruction expects a result back. In case the custom instruction does not expect a result back, then the accelerators will not write the result back to the MicroBlaze. The AXI master interface defined in the wrapper handles this functionality. When these custom instructions produce a CF error or an op error, then error handling stores and accumulates the corresponding status. The accumulated status will be written back later to the MicroBlaze-V when the accelerators run a read instruction.
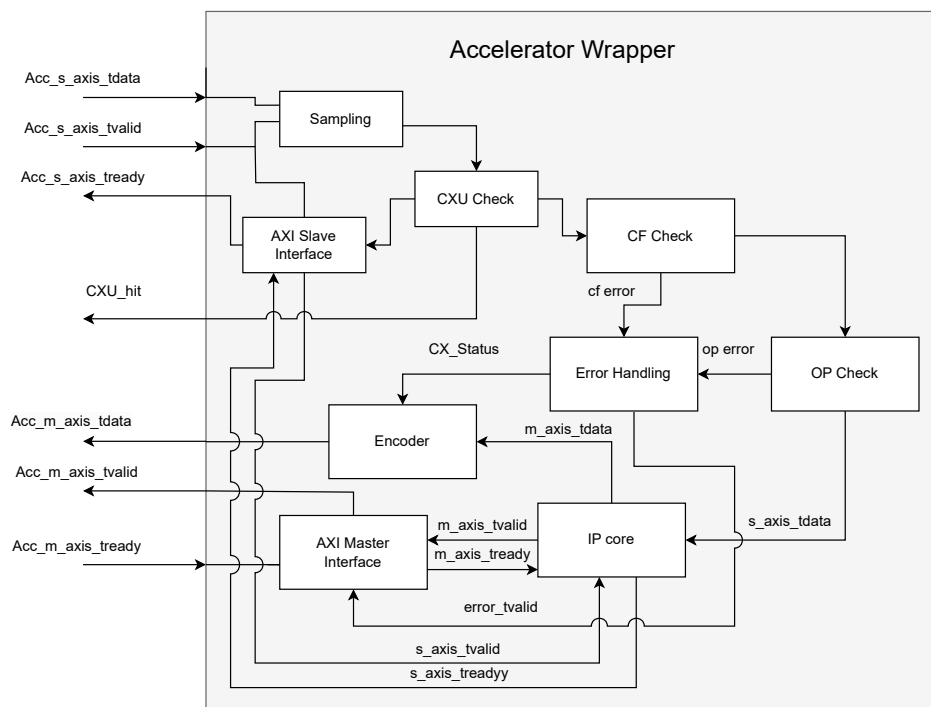

Figure 4.12: Wrapper module for the accelerators

4.4.1 CORDIC

This thesis work implements trigonometric and vector translation functionalities of the CORDIC IP core from Vivado. These functionalities are implemented as two independent CXUs with unique CXU_IDs as mentioned in table4.1. As the CORDIC accelerators require only one instruction to perform the computations, two LSB bits from the instruction code would be sufficient to represent the ID. As mentioned before, the IP core will be executed only after the error checking has been performed. In case of CORDIC functionalities, the state ID is ignored as the computation does not require any state context. Since both the CORDIC functions produce a result, custom -0 type instruction encoding is used. Therefore, the accelerator wrapper follows the normal process flow and returns the result to the MicroBlaze-V.

The trigonometric IP core has been configured for operating on a 16-bit fixed point input value which produces a 32-bit fixed point value, thereby enabling the write back of the computed result seamlessly. The 32-bit output value contains 16-bit sine and cosine values in the LSB and MSB parts respectively. As the operand values in the instruction code allow 32 bits, the input for the IP core has been defined on the LSB bits and the rest of the bits are ignored. The vector translation function of the CORDIC IP core is configured for a 32-bit Cartesian input producing a 32-bit result containing the scaled magnitude and rotated phase angle. Both the functions of CORDIC are configured to yield maximum performance by using the maximum pipelining mode and parallel architectural configuration. The truncation of the data helps to limit the result to 32-bits in both CORDIC functions.

Similar wrapper modules have been designed for both CORDIC functions following the normal process flow but they differ in input handling. This difference in the sampling unit ensures to drive the correct input for the IP cores. The overall timing required for the wrapper to perform these computations is shown in figure 4.13.

As mentioned, when receiving the bit stream having CXU valid, the CXU Hit flag will be asserted and will be de-asserted after the execution is completed. This flag also indicates the interconnect of the execution of a custom instruction. The wrapper module waits for the valid signal which indicates that the result is ready and returns the result along with the status in the accelerator master interface. By enabling the blocking mode of the IP core, the `tready` signal can be controlled manually.

4.4.2 FFT

The FFT IP core has been configured based on the radix-2 burst I/O architecture with a transform length of 256 enabling the number of elements, `N` or `k` in equations (2.1 and 2.2), in a data frame. Therefore, the number of butterfly stages in the FFT algorithm will be $\log_2 256 = 8$. The input to the FFT core has been designed to have 16 bits on both real and imaginary components. Therefore, the output will also have 16-bit real and 16-bit imaginary components. The IP core has been configured for fixed-point computations.

The Burst I/O architecture allows writing and reading each data frame in separate phases. Therefore, three custom instructions are required to configure, write, and
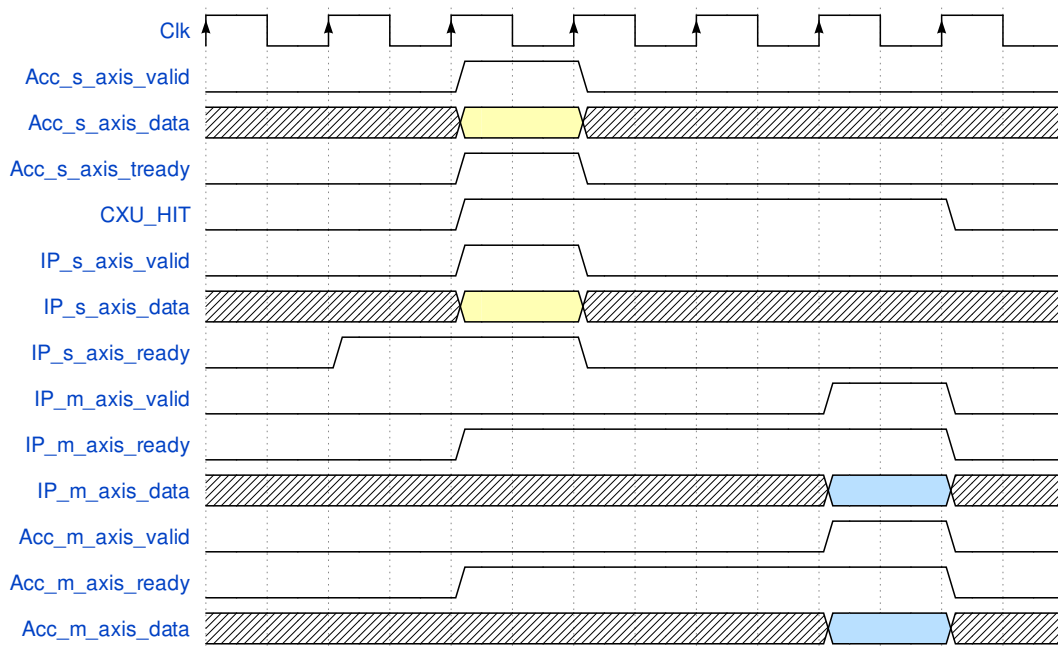
Figure 4.13: Timing diagram of accelerators using CORDIC IP core

read data from the FFT core. The design constraints set in the software manage the order of execution of these phases. In the write phase, the configurations and inputs required for the computations will be fed into the FFT core. During the read phase, the MicroBlaze will read the computed result using each instruction.

The wrapper module designed around the FFT core manages these independent phases along with the normal process flow including sampling and error checking. When receiving a bit stream with CXU valid, each phase of operation of the FFT acceleration will be differentiated using the CF_ID with the help of a demultiplexer unit as shown in figure 4.14. Since the property of the IP core enables to have independent AXI interface to configure, read, and write from the IP core, restricting the `tvalid` signal enables to control each phase of operation. This enables to connect the `tdata` to the IP core directly as `tvalid` restricts the IP core from reading the data. Similarly, `tready` in the slave interface of the wrapper has been configured to be dependent on the control signals in the AXI interface of the IP core as shown in the block diagram. In case of output instruction, the `acc_s_axis_tready` flag will depend on the `tvalid` in the output interface of the IP core. When considering the delay in computing the output, the wrapper module needs to receive the instruction and wait for the response. Therefore, the valid signal in the output interface has been logically OR'ed and forwarded to the demux. The FFT wrapper will only write back to the MicroBlaze for the custom instruction to read the output. In case of a write instruction, the error handling unit will store and accumulate the error. This means that the cf error and the operation error will be accumulated in the error handling unit as shown in figure 4.15 and are flushed out when a read instruction is processed. The ready flag in the master interface of the wrapper has been connected to the ready flag in the output interface. When the output instruction returns the result to the MicroBlaze, the ready flag will be de-asserted so that the following

data is not read.
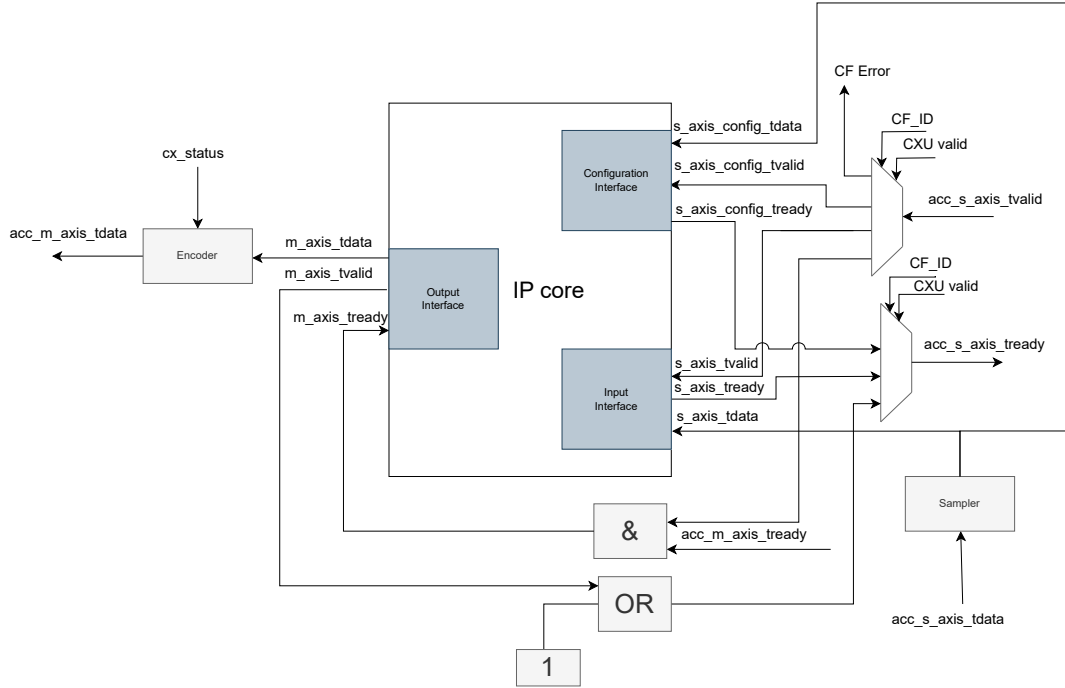
Figure 4.14: Block diagram of accelerator design using the FFT IP core

(a) cf error

(b) op error
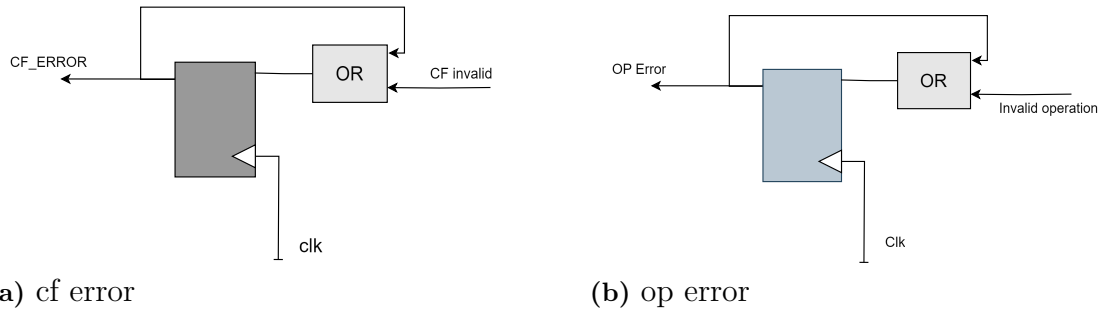
Figure 4.15: Accumulation of the errors by the error handling unit

When executing an input or configuration instruction, the FFT wrapper follows the same methodology and timing. The timing of the FFT write process when following a valid instruction is shown in figure 4.16.

In case of an output instruction, the wrapper modules need to wait until the IP core generates the output. The timing behaviour of an output instruction can be seen in figure 4.17.
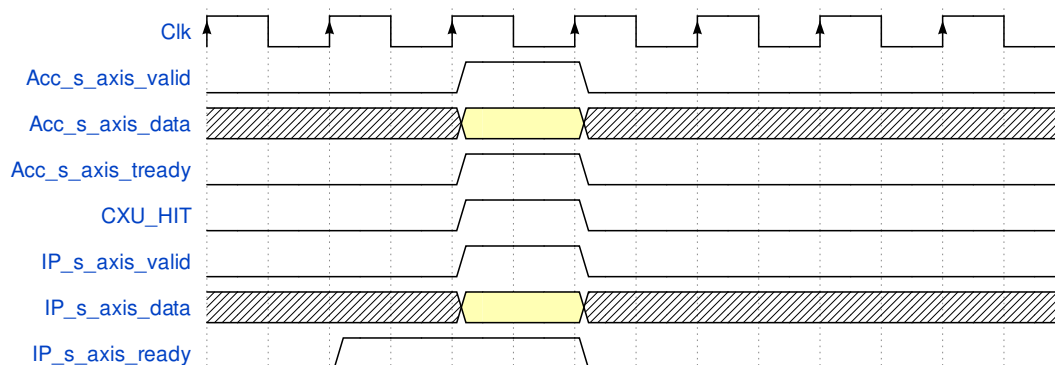
Figure 4.16: Timing diagram of input and config instructions in the FFT accelerator
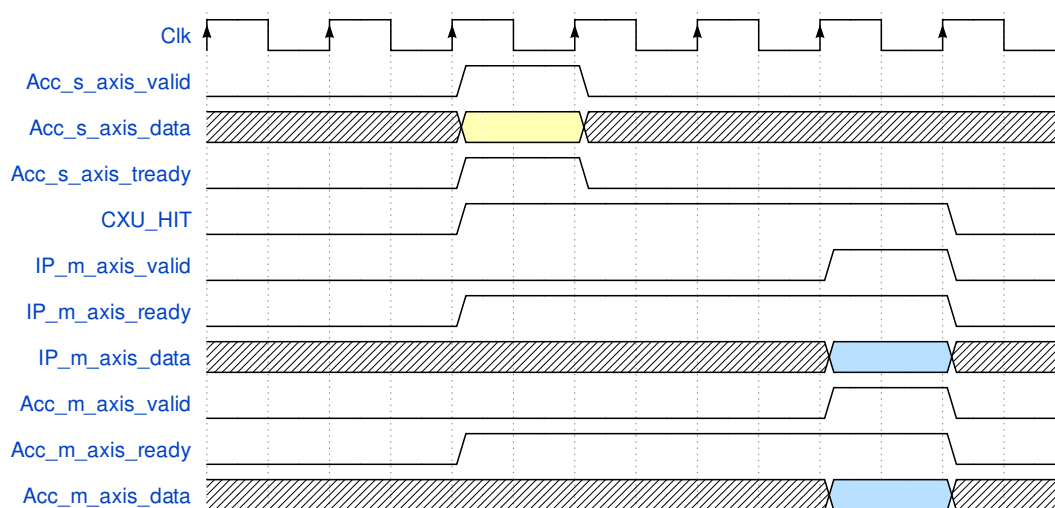
Figure 4.17: Timing diagram of output instruction in the FFT accelerator

4.5 Reference Model

It is necessary to compare the performance of the newly designed core and analyze its advantage over executing the same functions in software. So, for that reason, the trigonometric, vector translation, and FFT function codes were written in C programming language and were made to run in the existing MicroBlaze-V environment. The in-built math library in C has been used for calculating the trigonometric, square root and other mathematical operations involved in the calculations. Moreover, it was also made sure that the program size doesn't grow significantly in size so that it can be fit inside the core memory. A few design decisions were needed to be taken to satisfy this requirement. One of them was to make use of the Xilinx developed `xil_printf()` function instead of the normal `printf()` in order to compress the size. The size of the LMB RAM had to be increased to an extent to fit the nested loop operations involved in FFT calculation. Once the codes were written, each of them was compiled and built using Vitis, which generated an elf file. This elf file was imported to Vivado, which in turn allows the MicroBlaze design to communicate with the software code. Finally, this model was implemented and run on the KCU105 FPGA to analyze the performance metrics which are described in Section 2.9.

5

Results

In this section, we show the results obtained in this work. Through this work, AMD has got a new interface in their Microblaze-V core to plug in any accelerator for executing custom instructions. However, the accelerators should have the AXI-4 stream communication implemented and should have a similar wrapper designed to interface with the Microblaze core.

In this thesis work, we compared the results obtained by running the custom instructions using accelerators and by running the same function in the core without the accelerators. It was noted that the output value of the function written in C code does not match the output of the IP core. This is expected due to the difference in the way of implementing these functions in the IP core. For example, the truncation parameters are specific to the FFT IP core and might not match the conventional algorithm followed to program the same code in C. As a result, it is not ideal to compare the output of the performed operation, but would rather make sense to compare the performance in terms of speed up, resource usage and other metrics described in Section 2.9. These metrics were recorded by running the design in the KCU105 FPGA at a clock frequency of 100 MHz and the obtained results are illustrated below. As mentioned earlier, these metrics were recorded for both the existing MicroBlaze-V design and the modified core with the CX extension and accelerators. These results are then compared to show the benefits that have been achieved by having an interface to execute custom instructions.

5.1 Speedup

The speedup can be calculated from the execution time values of both the MicroBlaze-V core designs. The execution time was recorded by calculating the number of cycles using the CSR called `cycles` available in MicroBlaze-V. The `cycles` CSR samples the time from the beginning of the execution until it completes the execution. The difference between the time samples is recorded to calculate the total time taken to execute the specific instruction. This gives the value of CPI for that instruction. Execution time is obtained by multiplying the CPI with the time period of the system clock (10 ns). The reported execution time is given in table5.1.

Table 5.1: Execution time of the given functions in the modified design and current design.

	Execution Time	
Function	**CX-enabled MicroBlaze-V**	**MicroBlaze-V**
Sin and Cos	0.31 us	16.59 us
Vector Translation	0.31 us	43.68 us
FFT	57.69 us	17.85 ms

It makes more sense to calculate the speedup for each function using the execution time. This can be done using equation 2.4. The calculated speedup values are plotted for comparison in figure 5.1.
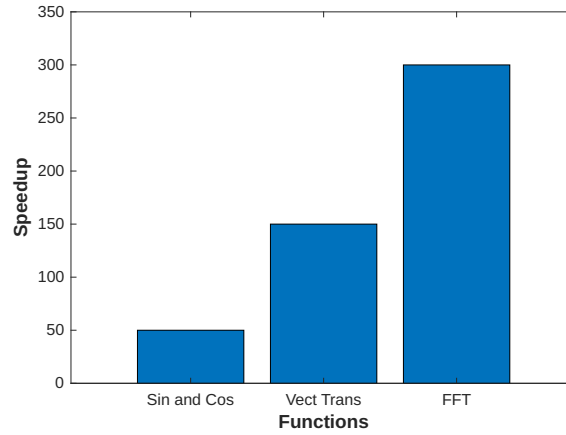
Figure 5.1: Speedup comparison of functions from the original version and by using accelerators

From the figure, it is clear that the execution time has improved significantly for all the functions. The use of accelerators has made the execution of the functions faster by executing them in hardware, which otherwise would take multiple loops to execute in software.

5.2   Hardware utilisation

The resource utilisation comparison of the hardware with the addition of CXUs and the interface is an essential parameter regarding the system cost. The resource utilisation of the system with and without accelerators in reference to KCU105 FPGA is shown in figure 5.2.
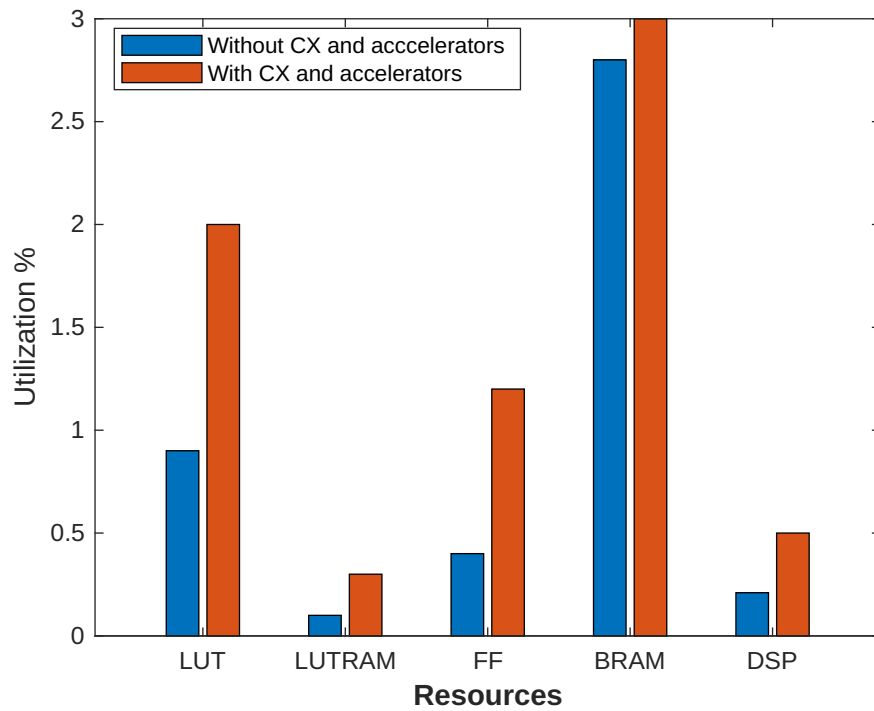
Figure 5.2: Resource utilisation percentage comparison

This resource utilisation comparison shows the additional resources required to implement the CX extension to perform the selected computations. The extra usage of resources is obvious as it is required to have more flip flops, LUTs, DSPs and other resources for the newly designed interface and accelerators. For instance, the accelerations including FFT have been configured to use DSPs for faster response time. Since we consider a single extension to connect all the accelerators, it is sufficient to compare the resource utilisation

5.3 Power Consumption

The extra hardware resources consume more power as well and thus the power should also be reported and ensured that it is well within the constraints and does not throw any warning in Vivado. A comparison of both dynamic and static power was done among both the designs and the results are illustrated in Figure 5.3.
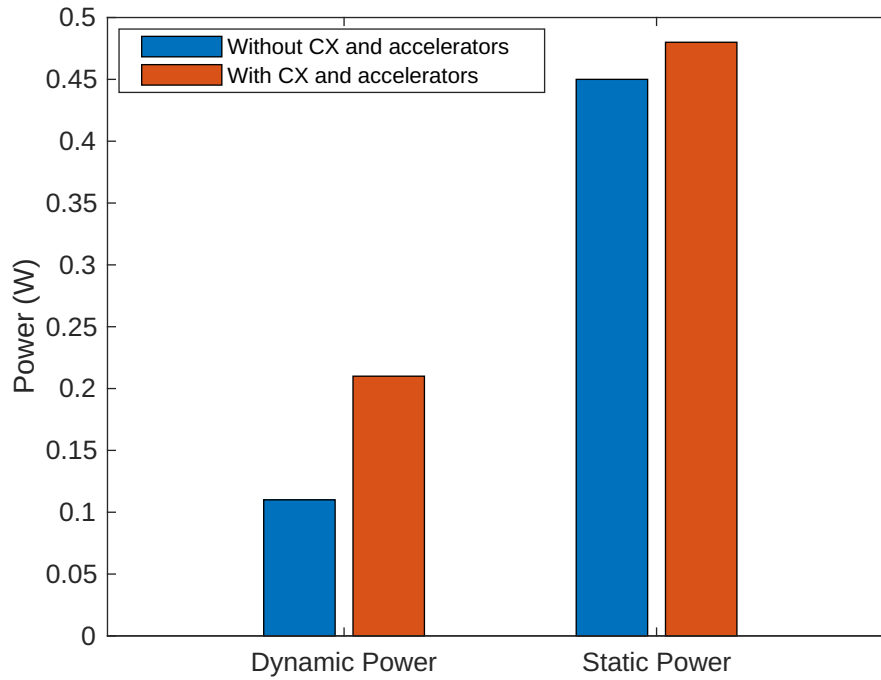
Figure 5.3: Comparison of power consumption

The implementation report from Vivado, shown in figure 5.3 explains the power dissipation in the FPGA board and does not illustrate the actual power consumption of the core with acceleration during runtime. In this case, we should have considered the energy consumption as the parameter which expects to show the improvement with acceleration. Our thesis work failed to explore this area due to timing constraints, which is one of our limitations.

The above results show that the introduction of the CX interface to the MicroBlaze-V core has enabled the execution of custom instructions with ease and at a higher performance. As the amount of hardware used is still in satisfactory limits, the design has been successful and can be a possible introduction in AMD's future customer releases.

6

Conclusion

The introduction of the CX interface to the MicroBlaze-V enables the plug-and-play of different accelerator modules. The current state of the CX extension in MicroBlaze-V supports any accelerator with an AXI4-stream interface. In the case of multiple accelerators, the same wrapper module in section 4.4 can be redefined for the identification of the accelerator and error handling. The instruction codes for enabling the accelerators will be fed into the MicroBlaze-V core using an inline assembler. On running the module, the instructions along with the operators will be taken in from the software, get processed in the core and accelerators, and output the result. With the help of the reference model, the accelerated functions have been compared with the same functions implemented on MicroBlaze-V using C functions.

To implement complex functions including FFT and trigonometry, a long set of standard instructions has to be fed to MicroBlaze-V. The same function can be executed with minimum CPU time by implementing a CX extension that uses external accelerators. An improvement of 300 times in the execution time can be observed when using FFT accelerators compared with the normal C functions. Similarly, an improvement of 50 times and 150 times in the CPU time can be observed for the trigonometric and vector translation functions. This improvement requires additional hardware resources in the FPGA including FFs, DSPs, etc. Therefore, the FPGA device dissipates additional total power compared to standard MicroBlaze-V implementation.

After implementing the CX extension, the MicroBlaze-V is expected to consume less power for the custom function.

6.1 Challenges

Initially, investigating and proposing potential accelerators that are industrially relevant was challenging. We were planning to design an accelerator function from scratch. Due to the constraint in the duration of this thesis work and as the main focus was on enabling MicroBlaze-V to support custom instructions, we decided to use the standalone IP cores from AMD's Vivado library. Therefore, from an industrial perspective, our thesis work demonstrates the easiness of connecting a standalone IP core to the MicroBlaze-V without extra effort.

Another challenge we faced was to get the software implementation of the accelerator functions to work. As these implementations, especially FFT, require multiple loops,

it require more memory space in BRAM, due to which the results were not obtained as expected. However, we did manage to unroll the loops and also increase the BRAM memory to an extent so that the expected output was obtained.

We also faced some difficulties when modifying the processor core to support the custom instructions, especially during the write-back to the registers and updating CSRs. Due to our inexperience and unfamiliarity with the MicroBlaze-V, we had to rely on our advisors at AMD for the necessary modifications in the core.

6.2 Future Work

The current state of the CX extension in MicroBlaze-V supports the implementation of FFT acceleration with a set of limitations defined in the software. That is, the execution order for the custom instructions for FFT should be in the chronological order of configuration, input, and output instructions. The chronological execution order can be managed by the hardware enabling the user to ignore the execution order. With the recent modification in the specification for enabling CX extension, the status of execution of a custom instruction should be updated in the CSRs.

The designed interconnect block managing the number of accelerators can be released as a standalone IP core with peripheral support to MicroBlaze-V thereby, allowing the user to seamlessly interface multiple accelerators. Furthermore, the wrapper module functions can be predefined in Vivado enabling the users to install independent accelerators to MicroBlaze-V.

References

[1] E. Cui, T. Li, and Q. Wei, "RISC-V Instruction Set Architecture Extensions: A Survey," *IEEE Access*, vol. 11, pp. 24 696–24 711, 2023.

[2] "Draft proposed RISC-V composable custom extensions specification," https://github.com/grayresearch/CX.

[3] Z. Li, W. Hu, and S. Chen, "Design and Implementation of CNN Custom Processor Based on RISC-V Architecture," in *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 2019, pp. 1945–1950.

[4] K.-D. Nguyen, D. T. Kiet, T.-T. Hoang, N. Q. N. Quynh, and C.-K. Pham, "A CORDIC-based Trigonometric Hardware Accelerator with Custom Instruction in 32-bit RISC-V System-on-Chip," in *2021 IEEE Hot Chips 33 Symposium (HCS)*, 2021, pp. 1–13.

[5] "What is a field programmable gate array (FPGA)?" https://www.ibm.com/think/topics/field-programmable-gate-arrays.

[6] *Vivado Design Suite User Guide*, Advanced Micro Devices, Inc, 10 2023.

[7] M. A. Michel Dubois and P. Stenström, *Parallel Computer Organization and Design*, 1st ed. Cambridge University Press, 2012.

[8] J. L. Hennessy and D. A. Patterson, *Computer Architecture - A Quantitative Approach*, 6th ed. Morgan Kaufmann, 2019.

[9] *MicroBlaze Processor Reference Guide (UG984)*, Advanced Micro Devices, Inc, 6 2021.

[10] I. Mhadhbi, S. Ben Othman, and S. Ben Saoud, "Impact of MicroBlaze FPGAs Design Methodologies of the Embedded Systems Performances," in *International Conference on Control, Engineering Information Technology*, 2014, pp. 146–151.

[11] *AMBA AXI-Stream Protocol Specification*, ARM Limited, 4 2021.

[12] G. Park, T. Taing, and H. Kim, "High-speed FPGA-to-FPGA Interface for a Multi-Chip CNN Accelerator," in *2023 20th International SoC Design Conference (ISOCC)*, 2023, pp. 333–334.

[13] *CORDIC v6.0 Product Guide (PG105)*, Advanced Micro Devices, Inc, 8 2021.

References

[14] P. Duhamel and M. Vetterli, "Fast Fourier transforms: a tutorial review and a state of the art," *Signal processing*, vol. 19, no. 4, pp. 259–299, 1990.

[15] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex Fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.

[16] *Fast Fourier Transform v9.1 Product Guide (PG109)*, Advanced Micro Devices, Inc, 5 2022.

A

Appendix 1

A.1 Reference Model C code

A.1.1 Trigonometric Function

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

int main()
{
    int angle = 0;

    // printing the sine value of angle
    xil_printf("sin(%d) = %d", angle,sin(angle));
    xil_printf("cos(%d) = %d", angle,cos(angle));

    return 0;
}
```

A.1.2 Vector Translation

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

#define PI 3.141592

int main()
{
    float x, y, r, theta;
    x=5;
    y=8;


    /* Calculating r */
    r = sqrt(x*x + y*y);

    /* Calculating theta in radian */
    theta = atan(y/x);

    /* Converting theta from degree to radian */
    theta = 180 * theta/ PI;

```

```
22    xil_printf("Magnitude = %d and Phase = %d in degree", r, theta);
23
24
25    return 0;
26 }
```

A.1.3   FFT

```
1  #include <stdint.h>
2  #include <string.h>
3  #include <stdio.h>
4  #include <math.h>
5  #include "xil_printf.h"
6  #include "platform.h"
7  #include "complex.h"
8  #include "input_array.h"
9
10
11 #define FIXED_POINT_FACTOR (1<<15)  // 2^15
12
13 // Fixed-point complex number structure
14
15
16 uint16_t reverse_bits(uint16_t x, int log2_n_points) {
17     uint16_t result = 0;
18     for (int i = 0; i < log2_n_points; i++) {
19         result = (result << 1) | (x & 1);
20         x >>= 1;
21     }
22     return result;
23 }
24
25 int main() {
26
27   init_platform();
28
29
30   //FILE *file;
31
32
33
34   //xil_printf("FFT started");
35   //  int read_elements;
36     // Read data from UART into the data array
37
38
39
40
41  /*   file = fopen("/opt/velloli/project_5/project_5.sim/sim_1/
       behav/questa/array_m.bin","rb");
42
43     if (file == NULL){
44     //  xil_printf("Unable to open file. \n");
45       return 1;
46     }
```

```
47
48
49      read_elements = fread(data,sizeof(int),255,file);
50
51      if (read_elements != 255){
52          //  xil_printf("Error reading the file. \n");
53              return 1;
54          }
55
56          */
57
58
59      // Reorder data array elements by bit-reversed indices order
60      complex_int16_t tmp;
61      uint16_t reversed;
62   //  xil_printf("Actual Data load");
63      for (uint16_t i = 0; i < n_points; i++) {
64          reversed = reverse_bits(i, log2_n_points);
65          if (reversed > i) {
66              tmp = data[i];
67              data[i] = data[reversed];
68              data[reversed] = tmp;
69          }
70      }
71
72   // xil_printf("Bits Reversed");
73 /*    for (uint16_t i = 0; i < n_points; i++) {
74          printf("%d,%d\n",data[i].real,data[i].imag);
75      }
76      */
77      // Perform the fixed-point FFT
78      complex_int16_t t, u;
79      for (uint16_t s = 1; s <= log2_n_points; ++s) {
80          uint16_t m = 1 << s;
81          int16_t cosine = FIXED_POINT_FACTOR * cos(-2.0 * M_PI / m);
82          int16_t sine = FIXED_POINT_FACTOR * sin(-2.0 * M_PI / m);
83          for (uint16_t k = 0; k < n_points; k += m) {
84              int16_t w_real = FIXED_POINT_FACTOR;
85              int16_t w_imag = 0;
86              for (uint16_t j = 0; j < m / 2; ++j) {
87                  uint16_t t_index = k + j;
88                  uint16_t u_index = k + j + m / 2;
89
90                  t.real = ((int32_t)w_real * data[u_index].real - (
    int32_t)w_imag * data[u_index].imag) / FIXED_POINT_FACTOR;
91                  t.imag = ((int32_t)w_real * data[u_index].imag + (
    int32_t)w_imag * data[u_index].real) / FIXED_POINT_FACTOR;
92                  u = data[t_index];
93
94                  data[t_index].real = u.real + t.real;
95                  data[t_index].imag = u.imag + t.imag;
96                  data[u_index].real = u.real - t.real;
97                  data[u_index].imag = u.imag - t.imag;
98
99                  int16_t tmp_real = w_real;
100                 w_real = ((int32_t)w_real * cosine - (int32_t)
```

```
    w_imag * sine) / FIXED_POINT_FACTOR;
101                 w_imag = ((int32_t)w_imag * cosine + (int32_t)
    tmp_real * sine) / FIXED_POINT_FACTOR;
102             }
103         }
104     //   printf("%d,%d\n",data[s].real,data[s].imag);
105     }
106
107     xil_printf("fft done!\n");
108
109     // Send output data via UART
110   // char output_buffer[128];
111     for (uint16_t i = 0; i < n_points; i++) {
112
113       //xil_printf("%d",data[i].real);
114         xil_printf( "Data[%d] = (%d, %d)\r\n", i, data[i].real,
    data[i].imag);
115         // Call the UART data sending function using output_buffer
116
117
118     }
119
120     //fclose(file);
121     cleanup_platform();
122     return 0;
123 }
```

Complex.h Header file

```
1 #ifndef COMPLEX_H
2 #define COMPLEX_H
3
4
5 typedef struct {
6     int real;
7     int imag;
8 } complex_int16_t;
9
10 #endif
```

Input Array header file

```
1
2 #ifndef INPUT_ARRAY_H
3 #define INPUT_ARRAY_H
4 #define n_points 256
5
6 #include "complex.h"
7
8 int log2_n_points = 8;
9
10 complex_int16_t data[256] = {
11     {.real = 0b0101000000000000, .imag = 0b0000000000000000},
12     {.real = 0b0100110101011000, .imag = 0b1111001101001011},
```

IV

```
13    {.real = 0b0100011000101110, .imag = 0b1110100101010011},
14    {.real = 0b0011110010100011, .imag = 0b1110001111111001},
15    {.real = 0b0011001110001001, .imag = 0b1110001110101101},
16    {.real = 0b0010110101110101, .imag = 0b1110011101001000},
17    {.real = 0b0010101111110111, .imag = 0b1110110001101111},
18    {.real = 0b0010111100011000, .imag = 0b1111000001010000},
19    {.real = 0b0011010101010110, .imag = 0b1111000010001000},
20    {.real = 0b0011110000100101, .imag = 0b1110101111100100},
21    {.real = 0b0100000010110101, .imag = 0b1110001011000100},
22    {.real = 0b0100000011011110, .imag = 0b1101011011111101},
23    {.real = 0b0011101111000111, .imag = 0b1100101101001000},
24    {.real = 0b0011001000110010, .imag = 0b1100001001100101},
25    {.real = 0b0010011000111100, .imag = 0b1011111000111110},
26    {.real = 0b0001101010110100, .imag = 0b1011111101001110},
27    {.real = 0b0001001000111001, .imag = 0b1100010001111101},
28    {.real = 0b0000111001101001, .imag = 0b1100101101110011},
29    {.real = 0b0000111101011101, .imag = 0b1101000101011110},
30    {.real = 0b0001001110100010, .imag = 0b1101001111010001},
31    {.real = 0b0001100010110011, .imag = 0b1101000110001010},
32    {.real = 0b0001101111000011, .imag = 0b1100101011011000},
33    {.real = 0b0001101010011111, .imag = 0b1100000110000100},
34    {.real = 0b0001010001100100, .imag = 0b1011100001000000},
35    {.real = 0b0000100111000111, .imag = 0b1011000111001011},
36    {.real = 0b1111110011011100, .imag = 0b1011000000011001},
37    {.real = 0b1111000001110010, .imag = 0b1011001110101111},
38    {.real = 0b1110011100101111, .imag = 0b1011101101111111},
39    {.real = 0b1110001010111010, .imag = 0b1100010100111001},
40    {.real = 0b1110001100111100, .imag = 0b1100111000000110},
41    {.real = 0b1110011101010001, .imag = 0b1101001101101111},
42    {.real = 0b1110110001111001, .imag = 0b1101010000100011},
43    {.real = 0b1110111111100100, .imag = 0b1101000001100001},
44    {.real = 0b1110111101010111, .imag = 0b1100100111100011},
45    {.real = 0b1110100111100010, .imag = 0b1100001101010010},
46    {.real = 0b1110000000101011, .imag = 0b1011111101101111},
47    {.real = 0b1101010000111100, .imag = 0b1100000000110101},
48    {.real = 0b1100100011011110, .imag = 0b1100011000110110},
49    {.real = 0b1100000010111100, .imag = 0b1101000001101100},
50    {.real = 0b1011110110001001, .imag = 0b1101110010001101},
51    {.real = 0b1011111101111001, .imag = 0b1110011111000010},
52    {.real = 0b1100010100110011, .imag = 0b1110111110001010},
53    {.real = 0b1100110000111011, .imag = 0b1111001010000100},
54    {.real = 0b1101000111000000, .imag = 0b1111000011100000},
55    {.real = 0b1101001101111100, .imag = 0b1110110001001001},
56    {.real = 0b1101000001101110, .imag = 0b1110011101100011},
57    {.real = 0b1100100100101100, .imag = 0b1110010011110000},
58    {.real = 0b1011111110110101, .imag = 0b1110011011110011},
59    {.real = 0b1011011011001011, .imag = 0b1110111000001001},
60    {.real = 0b1011000100011100, .imag = 0b1111100100111010},
61    {.real = 0b1011000001100010, .imag = 0b0000011001000011},
62    {.real = 0b1011010011100000, .imag = 0b0001001001001101},
63    {.real = 0b1011110101000011, .imag = 0b0001101011001111},
64    {.real = 0b1100011100010110, .imag = 0b0001111001011101},
65    {.real = 0b1100111110000010, .imag = 0b0001110100010110},
66    {.real = 0b1101010000110110, .imag = 0b0001100010011010},
67    {.real = 0b1101010000100011, .imag = 0b0001001110001000},
68    {.real = 0b1100111111001100, .imag = 0b0001000010100100},
```

```
69    {.real = 0b1100100100100010, .imag = 0b0001000111111101},
70    {.real = 0b1100001011100100, .imag = 0b0001100000111101},
71    {.real = 0b1011111110111111, .imag = 0b0010001001111010},
72    {.real = 0b1100000101110111, .imag = 0b0010111001111100},
73    {.real = 0b1100100001011001, .imag = 0b0011100101101110},
74    {.real = 0b1101001100011110, .imag = 0b0100000011000001},
75    {.real = 0b1101111101010011, .imag = 0b0100001011111110},
76    {.real = 0b1110101000100001, .imag = 0b0100000000111001},
77    {.real = 0b1111000100101011, .imag = 0b0011101000000110},
78    {.real = 0b1111001101010001, .imag = 0b0011001100000010},
79    {.real = 0b1111000100001001, .imag = 0b0010110111110101},
80    {.real = 0b1110110000110101, .imag = 0b0010110011111000},
81    {.real = 0b1110011110001111, .imag = 0b0011000011000111},
82    {.real = 0b1110010111001000, .imag = 0b0011100010001001},
83    {.real = 0b1110100010101101, .imag = 0b0100001000001111},
84    {.real = 0b1111000010010110, .imag = 0b0100101010001001},
85    {.real = 0b1111110001001001, .imag = 0b0100111101100101},
86    {.real = 0b0000100101011010, .imag = 0b0100111100100011},
87    {.real = 0b0001010011101111, .imag = 0b0100100111001010},
88    {.real = 0b0001110010100101, .imag = 0b0100000011100111},
89    {.real = 0b0001111101001100, .imag = 0b0011011100010001},
90    {.real = 0b0001110101001100, .imag = 0b0010111100011000},
91    {.real = 0b0001100001111110, .imag = 0b0010101100100001},
92    {.real = 0b0001001110010110, .imag = 0b0010101111111000},
93    {.real = 0b0001000101001010, .imag = 0b0011000011010101},
94    {.real = 0b0001001101110010, .imag = 0b0011011110010101},
95    {.real = 0b0001101001110110, .imag = 0b0011110101101011},
96    {.real = 0b0010010100101001, .imag = 0b0011111111000110},
97    {.real = 0b0011000100100111, .imag = 0b0011110100011001},
98    {.real = 0b0011101110010111, .imag = 0b0011010101100000},
99    {.real = 0b0100001000010000, .imag = 0b0010101000100000},
100   {.real = 0b0100001101011000, .imag = 0b0001110111101101},
101   {.real = 0b0011111111001001, .imag = 0b0001001110011001},
102   {.real = 0b0011100100110010, .imag = 0b0000011010101011},
103   {.real = 0b0011001001000111, .imag = 0b0000011000000001},
104   {.real = 0b0010110111000010, .imag = 0b0000111011100000},
105   {.real = 0b0010110110001001, .imag = 0b0001001111011011},
106   {.real = 0b0011001000010100, .imag = 0b0001100000101100},
107   {.real = 0b0011101001000101, .imag = 0b0001100100111011},
108   {.real = 0b0100001111000011, .imag = 0b0001010101110001},
109   {.real = 0b0100101110111001, .imag = 0b0000110010111111},
110   {.real = 0b0100111110110111, .imag = 0b0000000010011101},
111   {.real = 0b0100111001111001, .imag = 0b1111001110011011},
112   {.real = 0b0100100001010000, .imag = 0b1110100010001101},
113   {.real = 0b0011111100000001, .imag = 0b1110000110101111},
114   {.real = 0b0011010100111110, .imag = 0b1101111111101011},
115   {.real = 0b0010110111001010, .imag = 0b1110001010010110},
116   {.real = 0b0010101010010110, .imag = 0b1110011110100010},
117   {.real = 0b0010110000101100, .imag = 0b1110110001001011},
118   {.real = 0b0011000101111110, .imag = 0b1110110111110010},
119   {.real = 0b0011100000111110, .imag = 0b1110101011110111},
120   {.real = 0b0011110110011001, .imag = 0b1110001100111001},
121   {.real = 0b0011111100011101, .imag = 0b1101100000100100},
122   {.real = 0b0011101101111100, .imag = 0b1100110001000001},
123   {.real = 0b0011001011111000, .imag = 0b1100001001100100},
124   {.real = 0b0010011101010010, .imag = 0b1011110011001111},
```

```
125     {.real = 0b0001101100110111, .imag = 0b1011110001111011},
126     {.real = 0b0001000101101111, .imag = 0b1100000011000110},
127     {.real = 0b0000101111111011, .imag = 0b1100011110101100},
128     {.real = 0b0000101101110010, .imag = 0b1100111001101001},
129     {.real = 0b0000111011010110, .imag = 0b1101001001010111},
130     {.real = 0b0001001111100100, .imag = 0b1101000111001000},
131     {.real = 0b0001011111001101, .imag = 0b1100110010001011},
132     {.real = 0b0001100000010110, .imag = 0b1100001111111101},
133     {.real = 0b0001001101101000, .imag = 0b1011101010011101},
134     {.real = 0b0000100111111001, .imag = 0b1011001100111110},
135     {.real = 0b1111110101111101, .imag = 0b1011000000101001},
136     {.real = 0b1111000010100000, .imag = 0b1011001001011111},
137     {.real = 0b1110011000101100, .imag = 0b1011100101001011},
138     {.real = 0b1110000000101100, .imag = 0b1100001011110000},
139     {.real = 0b1101111101000111, .imag = 0b1100110010001001},
140     {.real = 0b1110001010001101, .imag = 0b1101001101101011},
141     {.real = 0b1110011111000010, .imag = 0b1101010111010110},
142     {.real = 0b1110110000011000, .imag = 0b1101001110001011},
143     {.real = 0b1110110100001110, .imag = 0b1100110111010101},
144     {.real = 0b1110100100111111, .imag = 0b1100011100101011},
145     {.real = 0b1110000011010010, .imag = 0b1100001001011101},
146     {.real = 0b1101010101110000, .imag = 0b1100000110111010},
147     {.real = 0b1100100110111101, .imag = 0b1100011001001101},
148     {.real = 0b1100000010000110, .imag = 0b1100111110001101},
149     {.real = 0b1011101111011110, .imag = 0b1101101110000101},
150     {.real = 0b1011110001111000, .imag = 0b1110011101110001},
151     {.real = 0b1100000101101111, .imag = 0b1111000010011101},
152     {.real = 0b1100100010010000, .imag = 0b1111010100111110},
153     {.real = 0b1100111100001001, .imag = 0b1111010100000010},
154     {.real = 0b1101001001010100, .imag = 0b1111000100101001},
155     {.real = 0b1101000011111100, .imag = 0b1110110000011101},
156     {.real = 0b1100101100011000, .imag = 0b1110100010110001},
157     {.real = 0b1100001001000001, .imag = 0b1110100100110110},
158     {.real = 0b1011100100010100, .imag = 0b1110111011000111},
159     {.real = 0b1011001001011111, .imag = 0b1111100011100110},
160     {.real = 0b1011000000111010, .imag = 0b0000010110100110},
161     {.real = 0b1011001101100100, .imag = 0b0001001001001000},
162     {.real = 0b1011101100000010, .imag = 0b0001110000010001},
163     {.real = 0b1100010011101000, .imag = 0b0010000100101101},
164     {.real = 0b1100111001000100, .imag = 0b0010000100111010},
165     {.real = 0b1101010010000100, .imag = 0b0001110101101010},
166     {.real = 0b1101011000100101, .imag = 0b0001100000100001},
167     {.real = 0b1101001100101101, .imag = 0b0001010000110010},
168     {.real = 0b1100110100100111, .imag = 0b0001001111111000},
169     {.real = 0b1100011010101000, .imag = 0b0001100010011001},
170     {.real = 0b1100001001111010, .imag = 0b0010000110100111},
171     {.real = 0b1100001010111111, .imag = 0b0010110101000001},
172     {.real = 0b1100100000111111, .imag = 0b0011100010101101},
173     {.real = 0b1101001000101101, .imag = 0b0100000100101101},
174     {.real = 0b1101111001100000, .imag = 0b0100010011100011},
175     {.real = 0b1110101000000111, .imag = 0b0100001101100010},
176     {.real = 0b1111001010001001, .imag = 0b0011110111010000},
177     {.real = 0b1111011001010100, .imag = 0b0011011010001011},
178     {.real = 0b1111010101011101, .imag = 0b0011000001100111},
179     {.real = 0b1111000100100001, .imag = 0b0010110111001011},
180     {.real = 0b1110110000101110, .imag = 0b0010111111101100},
```

```
181    {.real = 0b1110100101001110, .imag = 0b0011011001101011},
182    {.real = 0b1110101010101011, .imag = 0b0011111101110111},
183    {.real = 0b1111000100011010, .imag = 0b0100100001011001},
184    {.real = 0b1111101111011000, .imag = 0b0100111001010101},
185    {.real = 0b0000100011000111, .imag = 0b0100111110000100},
186    {.real = 0b0001010100011000, .imag = 0b0100101101101101},
187    {.real = 0b0001111000101000, .imag = 0b0100001100101101},
188    {.real = 0b0010001001011100, .imag = 0b0011100100011101},
189    {.real = 0b0010000110011010, .imag = 0b0011000000010011},
190    {.real = 0b0001110101010010, .imag = 0b0010101010000000},
191    {.real = 0b0001100000001100, .imag = 0b0010100110101001},
192    {.real = 0b0001010010010101, .imag = 0b0010110101000001},
193    {.real = 0b0001010100011111, .imag = 0b0011001110000100},
194    {.real = 0b0001101010001110, .imag = 0b0011100111000011},
195    {.real = 0b0010010000101110, .imag = 0b0011110101000001},
196    {.real = 0b0010111111101100, .imag = 0b0011110000001110},
197    {.real = 0b0011101011111010, .imag = 0b0011010110100110},
198    {.real = 0b0100001010110101, .imag = 0b0010101100011101},
199    {.real = 0b0100010101110110, .imag = 0b0001111011000101},
200    {.real = 0b0100001100010101, .imag = 0b0001001101110111},
201    {.real = 0b0011110011111010, .imag = 0b0000101110101100},
202    {.real = 0b0011010110100111, .imag = 0b0000100010111000},
203    {.real = 0b0010111111101100, .imag = 0b0000010100101111},
204    {.real = 0b0010111000001000, .imag = 0b0000111011101010},
205    {.real = 0b0011000011101110, .imag = 0b0001001110110000},
206    {.real = 0b0011011111111010, .imag = 0b0001010111110010},
207    {.real = 0b0100000100100101, .imag = 0b0001001110111000},
208    {.real = 0b0100100110101000, .imag = 0b0000110001110001},
209    {.real = 0b0100111011011100, .imag = 0b0000000100100101},
210    {.real = 0b0100111100010001, .imag = 0b1111010000011101},
211    {.real = 0b0100101000010101, .imag = 0b1110100000110011},
212    {.real = 0b0100000101000110, .imag = 0b1101111111101001},
213    {.real = 0b0011011100100010, .imag = 0b1101110010011110},
214    {.real = 0b0010111001111110, .imag = 0b1101111000100100},
215    {.real = 0b0010100110100010, .imag = 0b1110001011010001},
216    {.real = 0b0010100110010011, .imag = 0b1110011111111111},
217    {.real = 0b0010110110111110, .imag = 0b1110101011101101},
218    {.real = 0b0011010000101001, .imag = 0b1110100110011000},
219    {.real = 0b0011101000010010, .imag = 0b1110001101100001},
220    {.real = 0b0011110011010001, .imag = 0b1101100100111111},
221    {.real = 0b0011101010101100, .imag = 0b1100110101110101},
222    {.real = 0b0011001101100111, .imag = 0b1100001011011000},
223    {.real = 0b0010100001010100, .imag = 0b1011101111110000},
224    {.real = 0b0001101111101110, .imag = 0b1011101000100111},
225    {.real = 0b0001000100001100, .imag = 0b1011110101011100},
226    {.real = 0b0000101000000011, .imag = 0b1100001111101100},
227    {.real = 0b0000011111100101, .imag = 0b1100101100111000},
228    {.real = 0b0000101000110000, .imag = 0b1101000001111000},
229    {.real = 0b0000111011110101, .imag = 0b1101000110011101},
230    {.real = 0b0001001101111000, .imag = 0b1100110111110111},
231    {.real = 0b0001010100001110, .imag = 0b1100011001101110},
232    {.real = 0b0001000111110011, .imag = 0b1011110100111010},
233    {.real = 0b0000100111011101, .imag = 0b1011010100101011},
234    {.real = 0b1111111000010100, .imag = 0b1011000011001100},
235    {.real = 0b1111000100001011, .imag = 0b1011000110010010},
236    {.real = 0b1110010110011100, .imag = 0b1011011101101000},
```

VIII

```
237    {.real = 0b1101111000100011, .imag = 0b1100000010110010},
238    {.real = 0b1101101110111111, .imag = 0b1100101011010101},
239    {.real = 0b1101110111111110, .imag = 0b1101001100000010},
240    {.real = 0b1110001011111011, .imag = 0b1101011100011110},
241    {.real = 0b1110011111111101, .imag = 0b1101011001101001},
242    {.real = 0b1110101001010011, .imag = 0b1101000110111010},
243    {.real = 0b1110100000101110, .imag = 0b1100101100111101},
244    {.real = 0b1110000100110111, .imag = 0b1100010110111110},
245    {.real = 0b1101011010100101, .imag = 0b1100001111001011},
246    {.real = 0b1100101011100101, .imag = 0b1100011011100011},
247    {.real = 0b1100000011001110, .imag = 0b1100111011111011},
248    {.real = 0b1011101011000011, .imag = 0b1101101010000011},
249    {.real = 0b1011100111110001, .imag = 0b1110011011100001},
250    {.real = 0b1011110111101101, .imag = 0b1111000101000011},
251    {.real = 0b1100010011011111, .imag = 0b1111011110000010},
252    {.real = 0b1100110000001110, .imag = 0b1111100011001110},
253    {.real = 0b1101000011000100, .imag = 0b1111010111101111},
254    {.real = 0b1101000100100100, .imag = 0b1111000100000100},
255    {.real = 0b1100110011000101, .imag = 0b1110110011010111},
256    {.real = 0b1100010011010001, .imag = 0b1110101111111011},
257    {.real = 0b1011101110101001, .imag = 0b1110111111111001},
258    {.real = 0b1011010000100010, .imag = 0b1111100011010011},
259    {.real = 0b1011000010100100, .imag = 0b0000010100000101},
260    {.real = 0b1011001001100101, .imag = 0b0001000111111001},
261    {.real = 0b1011100100000111, .imag = 0b0001110011011011},
262    {.real = 0b1100001010111001, .imag = 0b0010001101111010},
263    {.real = 0b1100110011000011, .imag = 0b0010010011111000},
264    {.real = 0b1101010001101000, .imag = 0b0010001000010000},
265    {.real = 0b1101011110111111, .imag = 0b0001110011010111},
266    {.real = 0b1101011001001100, .imag = 0b0001100000010111},
267 };
268
269 #endif // ARRAY_H
```