Exploring Heterogeneous Computing Platforms for Molecular Dynamics-Based Calculations

A THESIS

Submitted in partial fulfillment of the
requirements for the award of the degree
of
Master of Technology

by
AMIT SINGH

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY INDORE
May 2025

INDIAN INSTITUTE OF TECHNOLOGY INDORE

CANDIDATE'S DECLARATION

I hereby certify that the work which is being presented in the thesis entitled **Exploring Heterogeneous Computing Platforms for Molecular Dynamics-Based Calculations** in the partial fulfillment of the requirements for the award of the degree of **MASTER OF TECHNOLOGY** and submitted in the **DEPARTMENT OF ELECTRICAL ENGINEERING, Indian Institute of Technology Indore**, is an authentic record of my own work carried out during the time period from July 2024 to May 2025. This thesis has been submitted under the supervision of Prof. Srivathsan Vasudevan and Prof. Satya S. Bulusu, IIT Indore.

The matter presented in this thesis has not been submitted by me for the award of any other degree of this or any other institute.

Amit 22/05/2025

Signature of the student with date
(AMIT SINGH)

This is to certify that the above statement made by the candidate is correct to the best of my/our knowledge.

V. Srivathsan 22/05/2025

Signature of the Supervisor of
M.Tech. thesis (with date)
(Prof. Srivathsan Vasudevan)

22/05/2025

Signature of the Supervisor of
M.Tech. thesis (with date)
(Prof. Satya S. Bulusu)

AMIT SINGH has successfully given his M.Tech. Oral Examination held on **May 5, 2025.**

V. Srivathsan

Signature of M.Tech Supervisors
Date 22/05/2025
Prof. Srivathsan Vasudevan

Signature of M.Tech Supervisors
Date 22/05/2025
Prof. Satya S. Bulusu

Saptarshi Ghosh

Convener DPGC
Date
23-05-2025

ACKNOWLEDGEMENTS

First and foremost, I would like to express my sincere appreciation to my thesis supervisors, **Prof. Srivathsan Vasudevan**, Professor, Department of Electrical Engineering, IIT Indore, and **Prof. Satya S. Bulusu**, Professor, Department of Chemistry, IIT Indore. Their generous guidance, support, mentoring, constant encouragement, and invaluable suggestions have been instrumental in making this work possible.

I am grateful to the M.Tech Coordinators, **Assistant Prof. Dr Rinkee Chopra** and **Prof. Abhinav Kranti** for their valuable advice, insightful comments and co-operation during my Research work presentation.

I express my sincere gratitude to IIT Indore for making available all facilities for this work. The cooperation from faculty and staff members at the Department of Electrical Engineering, IIT Indore, is gratefully acknowledged. Thanks are due to IIT Indore and the Ministry of Education, Government of India, for providing the stipend during the M.Tech. (VDN) programme.

I would also like to express my heartfelt gratitude to my parents and my younger brother for their unwavering support throughout my journey.

I would like to extend my wholehearted thanks to my lab members **Mr. Abhishek Ojha**, **Ms. Aparna Gangwar, Mr. Suvirn Upadhyay and Mr. Ashutosh Patel** and **Mr. Habit Tatin** for their help, support and for creating a positive environment in the lab throughout the year.

I would also like to express my gratitude to my seniors **Mr. Ankit Patel** and **Mr. Dharmendra Kartikey** for their support and guidance in my academic journey.

Last but not the least I am thankful to my friends without whom this journey would not have been so memorable. Special thanks to **Abu Said Parvez Alam**, **Akash Pandey**, **Suvirn Upadhyay**, **Atharv Limaye** and **Ashreta Sahay** for their help and support throughout this journey.

Amit Singh

THIS THESIS IS DEDICATED TO MY FAMILY AND FRIENDS

Abstract

Molecular dynamics (MD) simulations demand high computational power to model atomic interactions over time, often limiting their scalability on conventional processors. This thesis explores the role of FPGA-based heterogeneous computing in accelerating MD calculations by offloading key computational tasks such as force computation and integration. Through custom hardware design using high-level synthesis, the study demonstrates how FPGAs can offer significant performance gains with improved energy efficiency compared to CPUs and even GPUs in specific scenarios. By placing FPGAs at the heart of the MD simulation workflow, this work highlights their potential to drive scalable and efficient molecular simulations, paving the way for broader adoption in computational science.

Table of Contents

Contents

List of Figures

List of Tables

ACRONYMS

HPC--High Performance Computing

CPU--Central Processing Unit

GPU-- Graphics Processing Unit

ASIC-- Application Specific Integrated Circuit

FPGA- Field Programmable Gate Array

MD- -Molecular Dynamics

IP—Intellectual Property

CLB—Configuration Logic Block

LUT—Look Up Table

MUX—Multiplexer

HDL—Hardware Description Language

AXI—Advance Extensible Interface

HLS—High Level Synthesis

RISC—Reduce Instruction Set Computer

IAP—Interatomic Potential

DSP—Digital Signal Processing

PCIe—Peripheral Component Interconnect Express

UART-- Universal Asynchronous Receiver-Transmitter

DSO—Digital Storage Oscilloscope

SRAM—Static Random-Access Memory

IO—Input Output

GPIO—General Purpose Input Output

SDK—Software Development Kit

BRAM—Block Random Access Memory

DDR-- Double Data Rate

SPI-- Serial Peripheral Interface

IC- Instruction Cache

DC-- Data Cache

FIFO-- First In First Out

MIG-- Memory Interface Generator

RTL--Register Transfer Level

B/D—Block Diagram

Chapter 1

Introduction

1.1 Background and Motivation

In today's world, the need to understand matter at the atomic and molecular level is of paramount importance. From designing next-generation materials and drugs to building effective energy storage systems [1] and knowledge of biological processes [2], the behavior of systems at the molecular level explains innumerable technological and scientific advances. Many of the critical issues in modern science, including developing cancer treatments, building lightweight and durable materials, or maximizing catalysts [3] for green energy, require a deep understanding of how atoms and molecules interact over time.

However, direct observation of atomic-scale phenomena is difficult due to limitations in the existing experimental techniques. While tools like X-ray crystallography and electron microscopy provide structural insights, they are often static or averaged over time, and may not capture dynamic processes such as molecular folding, diffusion, or reaction pathways. Additionally, experimental investigations at this scale are often expensive, time-consuming, and limited in scope.

To complement and sometimes even replace physical experimentation, computational methods have emerged as powerful tools for probing molecular systems. Among these, Molecular Dynamics (MD) simulation [4] stands out as one of the most widely used techniques. By numerically modeling the physical laws that govern atomic interactions, MD simulations allow researchers to investigate how molecular systems evolve under different conditions. This enables the scrutiny of structural,

thermodynamic, and kinetic aspects of systems whose access would otherwise be confined experimentally.

1.2 A Brief Introduction to MD Simulation

An important tool in computational science, molecular dynamics (MD) simulation is used widely to examine the physical movements of atoms and molecules. The numerical solution of Newton's equations of motion[5] for a system of interacting particles it enables researchers to look into intricate physical systems at the atomic level. In domains where understanding molecular interactions and predicting system behavior under diverse circumstances are essential, such as materials science, chemistry, biology, and nanotechnology, MD simulations are essential.

MD simulations are well-known for being computationally demanding, particularly when working with large systems or lengthy simulation times, despite their scientific value. Since some computation tasks are inherently sequential and require massive data throughput, traditional CPU-based platforms frequently struggle to deliver real-time or high-throughput performance. The need for faster and more effective computing solutions is increasing along with the demand for longer and more detailed simulations.

Heterogeneous computing platforms[6] have become a viable remedy for these constraints. These platforms combine hardware accelerators like Graphics Processing Units (GPUs) and Field-Programmable Gate Arrays (FPGAs) with general-purpose CPUs (CPUs). Because of their low latency, energy efficiency, and adaptable hardware architecture, FPGAs provide a distinct advantage over the others. FPGAs can be specifically designed to speed up the computational kernels of MD algorithms, potentially resulting in significant performance gains, in contrast to GPUs, which have a fixed architecture optimized for high-throughput parallelism.

Here we investigate how to speed up MD-based simulations by designing and implementing a heterogeneous computing platform that uses FPGAs. We hope to show how accuracy can be maintained while performance and energy efficiency are increased by offloading crucial computational functions, like force calculations and Neighbour list updates, onto an FPGA. In addition to contributing to the continuous endeavor to speed up MD simulations, this work sheds light on the practical challenges and architectural compromises associated with FPGA-based hardware acceleration for scientific computing.

1.3 Computational challenges in MD simulation

The high computational cost of Molecular Dynamics (MD) simulations frequently limits their practical application, although they offer rich insights into molecular behavior. On conventional CPU-based platforms, it can take days or weeks of computation to simulate even a few nanoseconds of real-time dynamics for a system with thousands or millions of atoms. This is because MD simulations require the computation of forces resulting from all pairwise and bonded interactions as well as the repetitive numerical integration of particle motion, both of which must be completed at each femtosecond-scale time step.

The number of particles and interactions that need to be calculated increases quickly with the complexity of the systems being studied, such as biomolecular assemblies, polymer blends, or nanostructured materials. The simulation burden may also be increased by coarse-grained models, long-range electrostatics, or more complex force fields. Because of this, researchers must frequently choose one of the three trade-offs—accuracy, system size, and simulation length—to maintain realistic runtimes.

The computing community has increasingly resorted to heterogeneous computing platforms, which combine general-purpose CPUs with specialised accelerators like Graphics Processing Units (GPUs) and Field-Programmable Gate Arrays (FPGAs), in order to get around these bottlenecks. Although GPUs have already shown significant speedups for MD tasks by performing force calculations in parallel, FPGAs provide a special and frequently overlooked benefit: hardware-level customizability.

The design and implementation of a heterogeneous computing platform that uses FPGAs to speed up MD-based simulations is examined in this thesis. Our goal is to show how accuracy can be maintained while performance and energy efficiency are increased by offloading crucial computational functions, like force calculations and neighbour list updates, onto an FPGA. In addition to adding to the continuous endeavour to speed up MD simulations, this work sheds light on the real-world difficulties and architectural compromises associated with FPGA-based hardware acceleration for scientific computing.

Why FPGAs?

With Field-Programmable Gate Arrays (FPGAs), designers can configure up the hardware exactly according to the computational problem's structure. FPGAs can be programmed to implement custom pipelines and parallel data paths for particular algorithms, in contrast to CPUs and GPUs, which function with fixed instruction sets and general-purpose pipelines. This makes it possible to execute MD kernels with significantly reduced latency and power consumption, including Neighbour generation, non-bonded force evaluation, and particle updates.

FPGAs are especially appealing for uses like embedded systems, portable medical devices, and massive data centers where real-time performance and energy efficiency are crucial. This implies that simulations

can be extended to longer periods or larger systems in the context of MD while still requiring a manageable amount of computing power.

Moreover, it is now more possible to incorporate FPGAs into heterogeneous computing architectures alongside CPUs and GPUs thanks to recent developments in high-level synthesis (HLS) tools and FPGA development platforms. This creates new possibilities for offloading the most time-consuming portions of MD simulations to FPGAs, increasing speed and energy efficiency without compromising accuracy or flexibility.

1.4 Heterogeneous Computing for MD Simulations

Utilizing the strengths of each component, a heterogeneous computing platform combines various processing unit types, usually CPUs, GPUs, and FPGAs, to carry out tasks collectively. This architectural model offers an effective approach to handle the algorithm's various computational requirements in the context of MD simulations more effectively than any particular processor type can.

MD simulations consist of multiple tasks, each with unique computational properties:

- Force calculations are parallelizable and computationally difficult, especially for non-bonded interactions such as Lennard-Jones or Coulomb forces.
- Creating neighbour lists necessitates memory-intensive processes like sorting and spatial data structures.
- The process of integrating motion equations is comparatively simple and has a consistent pattern.
- Restriction enforcement and boundary conditions are less appropriate for parallel execution and may call for conditional logic.

- High-level control logic, preprocessing, and data flow coordination are the tasks that CPUs are best suited for.
- Massively parallel tasks, like calculating pairwise forces across numerous particles at once, are a strength of GPUs.

FPGAs are perfect for offloading bottleneck tasks that benefit from special data flow and parallelism because they can be set up to run particular computational kernels (such as cell list updates or short-range force evaluation) with deep pipelining, low latency, and high throughput.

Heterogeneous systems can handle larger or more complex models, improve energy efficiency, and drastically cut down simulation runtime by dividing MD tasks among these units. For example, an FPGA can continuously compute non-bonded interactions in real time, offloading one of the most burdensome operations from the CPU/GPU and enabling better overall throughput, whereas a CPU might handle simulation setup and output.

As demonstrated by hybrid MD packages that employ GPUs for force calculation or specialized FPGA-accelerated modules integrated into HPC clusters, recent research and industrial tools have started adopting such architectures. These developments pave the way for wider use and additional simulation acceleration innovation.

1.5 Perspective on Heterogeneous FPGA-based Computing Platforms

The overarching goal of this thesis is to explore and demonstrate the use of FPGA-based acceleration within a heterogeneous computing platform to improve the performance and energy efficiency of MD-based simulations.

1.5.1 A Study of Heterogeneous Computing Platforms Based on FPGA

The term "heterogeneous computing platform" presumes a more expansive meaning in this work; it not only describes a combination of processing components, such as CPUs and FPGAs, cooperating, but it also describes the actual implementation of FPGAs on various hardware platforms with different architectural characteristics. The implementation examines at platforms like the Zynq-7000 series, which combines ARM processors (Processing System, or PS) and programmable logic (PL) on a single chip, allowing for deeper hardware-software co-design, and the Basys 3 board, which offers a simple, lightweight FPGA fabric (perfect for testing core logic).

1.5.2 Advantages of FPGA

Because of its ease of use and accessibility for quick prototyping, the Basys 3 Board (Artix-7 FPGA) is appropriate for implementing and validating basic hardware accelerators (such as force calculators and integrators).

A dual-core ARM processor and programmable logic are combined in the Zynq Series (such as Zed Board or ZCU boards), enabling close integration between hardware (custom logic in PL) and software (running on ARM). For full-stack MD implementations, where control logic, memory management, and data streaming can all be optimised simultaneously, this is especially advantageous.

PC-FPGA configurations: Simulating a real-world heterogeneous setup between high-level MD engines and low-level accelerators, simulation tasks can be offloaded to an FPGA and results returned asynchronously when the FPGA is connected to a host PC via interfaces such as UART, USB, or PCIe.

This perspective is important since MD accelerator deployment in the real world is rarely confined to a single, consistent platform. Smaller FPGAs are often used for kernel development in research and practical development, followed by embedded SoCs like Zynq for tighter integration and, ultimately, scaling to datacenter-grade FPGAs or hybrid cloud-based systems. This work investigates several boards, assessing not only acceleration potential separately but also portability, design reusability, and device-to-device performance scaling—all critical attributes for future MD hardware engines.

Therefore, this illustrates a platform-aware heterogeneous computing vision by showing how FPGAs deployed in various configurations — from academic prototypes to embedded systems — can gradually accelerate MD simulations and how these setups individually and collectively contribute to faster, more efficient molecular simulations.

1.6 What are kernels?

In simple terms, a kernel in this context refers to a self-contained, reusable block of computation — a small part of a larger algorithm that performs a specific task. In Molecular Dynamics (MD) simulations, kernels are the core units of computation that are repeated many times.

So, when we say: "Identifying which kernels are best suited for FPGA acceleration". It means figuring out which parts of the MD simulation (like force calculation or neighbor list building) can be broken out and then efficiently run on the FPGA to speed up the simulation.

Examples of MD kernels:

- Force calculation kernel – calculates the interaction forces between pairs of atoms.
- Neighbor list kernel – builds a list of nearby atoms for each particle.

- Integration kernel – updates particle positions and velocities using algorithms like Verlet integration.

These are considered "kernels" because they run many times (often millions of iterations). They are compute-intensive. They can often be separated from the rest of the code and offloaded to a specialized processor (like an FPGA or GPU).

Kernels Well-Suited for FPGA Acceleration in MD Simulations. example:

a) Non-Bonded Force Calculation (e.g., Lennard-Jones Potential)

Why is it suitable?

- It's repetitive, parallelizable, and has regular arithmetic.
- FPGA advantage: Can be pipelined to compute forces between many atom pairs in parallel.
- Implementation: Use fixed-point or reduced-precision floating-point on Basys 3 or Zynq PL.

b) Neighbor List Construction

Why is it suitable?

- It involves spatial searches, but can be optimized via cutoff distance checks.
- FPGA advantage: Efficient at comparing many distances in parallel using loop unrolling.
- Implementation: Suitable for Zynq boards with more BRAM for storing coordinates.

c) Verlet Integration/Time stepping

Why is it suitable?

- A small, deterministic loop for updating position and velocity.
- FPGA advantage: Can be a pipelined arithmetic block, ideal for Basys3.
- Implementation: Use block RAMs to store positions, update in-place with minimal latency.

d) Constraint Solvers

Why is it more complex?

- These often involve iterative solvers and are harder to pipeline.
- FPGA suitability: Only partially suitable unless highly optimized, more fitting for Zynq.

Strategy Based on Platform:

1) Basys 3: Ideal for developing and testing smaller, arithmetic-heavy kernels like Verlet integration or a simple Lennard-Jones force loop.

2) Zynq: Ideal for building end-to-end MD pipelines, combining an ARM processor (for control) and PL (for force calculation + neighbor list).

1.7 Partitioning

Partitioning: A key concept in realizing the actual purpose of heterogeneous computing.

What is Data Partitioning?

Data partitioning refers to dividing the simulation data into smaller, manageable chunks that can be distributed across different compute resources, like FPGA fabric and CPUs, to enable parallel processing.

Why is It Important in Heterogeneous Platforms?

In a heterogeneous system (e.g., ARM CPU + FPGA), each component has strengths

- FPGA: Excellent at parallel and pipelined arithmetic tasks. Ideal for computing interactions, applying forces, or processing multiple atom pairs in parallel.
- CPU (ARM or Host): Better at handling complex logic, control flow, file I/O, or memory allocation tasks.

By partitioning the data effectively, we can

- Offload only the high-compute, parallel parts (e.g., force calculations) to the FPGA.
- Keep control and orchestration (e.g., managing simulation steps or writing results) on the CPU.
- Avoid idle hardware — maximize hardware utilization.
- Reduce memory bottlenecks — process data in-place on the right unit.

How It Applies to MD Simulations

- Let’s take a practical example:
- Suppose you're simulating a system with N particles. You typically need to:
- Loop over all particle pairs to compute forces.
- Update positions/velocities.
- Rebuild neighbor lists every few steps.

Without Data Partitioning:

- All data resides in CPU memory.
- The CPU runs all calculations serially or via software loops.
- FPGA is underutilized (if used at all).
- Bottleneck: Single memory interface, no hardware acceleration.

With Data Partitioning:

- Divide the particle list into smaller blocks (e.g., blocks of 64 or 128 atoms).

- Each block is sent to the FPGA via AXI4 or UART for force computation.
- The CPU keeps track of time steps, boundary conditions, etc.
- Overlap computation on FPGA with neighbor list rebuilding on CPU.

Types of Partitioning Approaches:

a) Spatial Partitioning: Divide the simulation space (e.g., a 3D box) into regions. Assign different regions to different units.
   - FPGA processes atoms in region A.
   - The CPU handles region B or overall orchestration.
   - Helpful in simulations with millions of particles.

b) Functional Partitioning: Divide the simulation tasks (not space).
   - FPGA: Force calculation and position updates.
   - CPU: Neighbor list updates, output logging, temperature scaling.

c) Temporal Partitioning: Assign different time steps or intervals to different units.
   - While FPGA processes the current step, CPU prepares the next input.

Benefits of Data Partitioning: Especially on platforms like Basys 3 or Zynq.

- **Basys 3**: Limited memory, but perfect for functional partitioning of one kernel at a time (e.g., force calc).
- **Zynq (ARM + PL):** Enables true hardware/software co-design — perfect for both spatial and functional partitioning.

Data partitioning:

- Reduces latency by enabling in-place computation on an FPGA.
- Minimizes data movement — process data where it resides.
- Scales better — more logic = more blocks processed in parallel.
- Makes the MD accelerator modular and portable.

Summary

- Data partitioning is not just about splitting data — it's about smart delegation:
- Sending the right kind of data to the right hardware at the right time.
- Enabling both CPU and FPGA to work in parallel and avoid idle time.
- Essential for scaling MD simulations across heterogeneous platforms, like a Basys 3 prototype and a Zynq deployment.

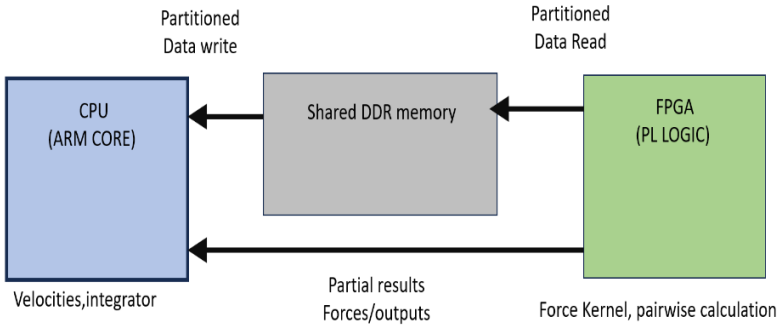


Fig.1.1 Simplified Data Partitioning & Flow in Heterogeneous MD Simulation (Zynq-based platform)

Here's a block diagram that illustrates how data partitioning enables efficient MD simulation on a heterogeneous platform like the Zynq board:

Key Concepts in the Diagram:

The CPU (ARM Core) handles velocity updates, integration (Verlet Algorithm), and control and orchestration at the same time, FPGA (Programmable logic) accelerates the process of pairwise calculations and distance check. Shared DDR memory holds partitioned data sets (positions, forces), and it acts as a communication buffer between the CPU and the FPGA.

This architecture allows concurrent processing of MD tasks, maximizing both performance and energy efficiency.

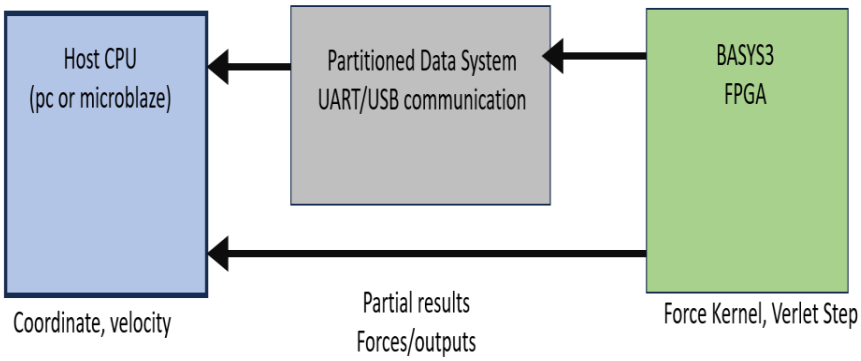


Fig.1.2 Simplified Data Partitioning & Flow in MD Simulation (BASYS 3 with UART Communication

Here is the block diagram showing data partitioning and flow for MD simulation on the Basys 3 board, where communication is handled via UART or USB instead of shared memory

Host CPU / MicroBlaze Prepares simulation data (positions, velocities) and sends it to the FPGA via UART.it also receives result back (computed forces). FPGA mainly focused on force computation and numerical integration and it operates in real time as the data arrives

Why This Matters

It allows MD simulation on low-cost hardware (Basys 3).it demonstrates the feasibility of FPGA- based acceleration even without a full OS or shared memory. Efficient data partitioning enables a clear division of responsibilities between host and the FPGA.

1.8 Conclusions

While Molecular Dynamics (MD) simulations have become essential across scientific disciplines, they remain constrained by performance bottlenecks, especially when scaling to large systems or simulating long time durations. Although GPUs have accelerated MD algorithms to a degree, they remain limited by fixed architectures and

inefficient memory access patterns for certain irregular tasks. FPGAs, despite their flexibility and performance potential, are underutilized in mainstream MD applications. Most existing implementations rely heavily on CPUs and GPUs, overlooking how FPGAs could be strategically integrated into heterogeneous systems to accelerate specific, high-impact parts of the MD pipeline, such as non-bonded force computations, neighbor list generation, and bonded interactions, with lower power and potentially higher throughput for certain workloads.

However, integrating FPGAs into MD simulation workflows presents several challenges: Identifying which kernels are best suited for FPGA acceleration, efficiently partitioning workloads between CPU, GPU, and FPGA, managing data flow and synchronization across heterogeneous components, and evaluating trade-offs between performance, energy efficiency, and resource utilization

This thesis aims to address these challenges by exploring a heterogeneous computing platform that executes a simple mathematical operation on a smaller prototype IP—serving as a foundation that can be scaled to more complex IPs—while integrating FPGA acceleration into the MD simulation process to enable optimized task scheduling and efficient data distribution across multiple processing units.

1.9 Objective

1) To explore potential scalability to multi-FPGA or CPU-GPU-FPGA systems, demonstrating the viability of FPGA acceleration in broader high-performance computing (HPC) contexts for MD.

2) To integrate these FPGA modules within a CPU-FPGA heterogeneous computing framework, ensuring seamless communication, task delegation, and memory consistency to ensure hardware control as well as easy debugging of the given system.

3)Implement the simple ALU operations in the FPGA-FPGA-based system using UART communication protocol, which will later translate into a complex MD simulation by using a larger IP framework.

1.10 Organization of the Thesis

The thesis will be divided into five chapters, and in each chapter, the generic flow of content representation will be as follows: the **first chapter** will provide a brief introduction of the background and motivation behind the given research work and all the elements associated with it. As we move ahead into the **second chapter,** this module explores the theoretical fundamentals and literature survey required to develop a system that can imitate the FPGA Accelerator for IAP-based MD Simulation. Subsequently, **the third module** will give the details of the hardware and software development and implementation of different operations involved, and interboard communication to realize the final work. The **fourth chapter** will discuss the various results obtained after the real implementation of the system. Finally, as we reach into the final phase of the Thesis, that is, **chapter five,** this module delves into the possibilities of future work and the large-scale implementation of the existing project work.

Chapter 2

Literature Review

These three metrics—throughput, latency, and timing—are essential for assessing and improving system performance in the larger High-Performance Computing (HPC) framework. The goals of HPC platforms, which frequently combine CPUs, GPUs, and FPGAs in heterogeneous architectures, are to optimize throughput for processing large amounts of data, reduce latency for responsiveness in real time, and make sure that designs adhere to stringent timing requirements for dependable operation at high clock frequencies. Specifically, when FPGAs are used in an HPC environment, they are used to speed up data-intensive kernels by taking advantage of their built-in parallelism and pipelining features, which have a direct effect on latency and throughput. Furthermore, satisfying timing constraints in FPGA designs guarantees that these accelerators can operate at ideal clock frequencies without going against setup or hold specifications, maintaining the high-speed performance required for HPC workloads like deep learning inference, scientific computing, and molecular dynamics simulations. High-throughput architectures are designed to maximize the number of bits processed per second, while low-latency architectures aim to minimize the time delay between the input and output of a module. At the same time, timing optimizations focus on reducing the combinatorial delay along the critical path to ensure the design meets its target clock frequency

As discussed above, the three main components that decide how fast any computing machine will perform are **THROUGHPUT**, **LATENCY**, and **TIMING OPTIMIZATIONS**. Now, it will be discussed how these metrics play a key role in high-performance computing.

These key terms of high-performance computing are defined as

2.1 High throughput

A design focused on high throughput prioritizes maintaining a continuous and efficient flow of data over time. It aims to maximize the overall data processing rate, placing less emphasis on how long it takes for an individual data element to travel from input to output (latency).

2.2 Low Latency

A low-latency design focuses on transferring data from input to output in the shortest possible time by reducing delays in intermediate stages. To achieve this, techniques such as parallel execution, pipelining, and simplified logic are often employed, even if they might compromise the design’s throughput or limit its peak operating frequency.

2.3 Timing

Timing in a digital design refers to its operating clock frequency. This frequency is fundamentally constrained by the longest delay between any two sequential components, which sets an upper bound on how fast the clock can run. Unlike broader speed/area trade-offs covered in other parts of this chapter, clock speed operates at a more fundamental level of abstraction and isn't inherently tied to specific architectural choices. However, the internal decisions made within a particular architecture, such as how resources are organized, do influence timing performance. For instance, determining whether a pipelined architecture outperforms an iterative one in terms of speed requires detailed insight into the actual implementation. The design’s peak performance or maximum operating frequency can be described using the standard minimum-frequency formula, excluding factors like clock-to-clock jitter.

$$F_{max} = \frac{1}{T_{clk-q} + T_{logic} + T_{routing} + T_{setup} - T_{skew}} \qquad (2.1)$$

In Eqn 2.1, **Fmax** is the **maximum allowable frequency** for the clock; T clk-q is the time from clock arrival until data arrives at Q; T-logic is propagation delay through logic between flip-flops; Trouting is routing delay between flip-flops; Tsetup is minimum time data must arrive at D before the next rising edge of clock (setup time); and Tskew is propagation delay of clock between the launch flip-flop and the capture flip-flop.

2.4 Evolution in High-Performance Computing (HPC)

HPC covers all aspects of technology, methodologies, and applications aimed at achieving the maximum computing capability possible using clusters of powerful processors. These processors work simultaneously to handle large-scale multi-dimensional datasets, often referred to as big data, and to solve complex problems at very high speeds. The primary goal of HPC is to provide solutions to questions that cannot be adequately addressed solely through empirical methods, theory, or commercially available computers. The most visible components of HPC are high-performance computers, commonly known a supercomputers, which occupy large areas and consume significant power.

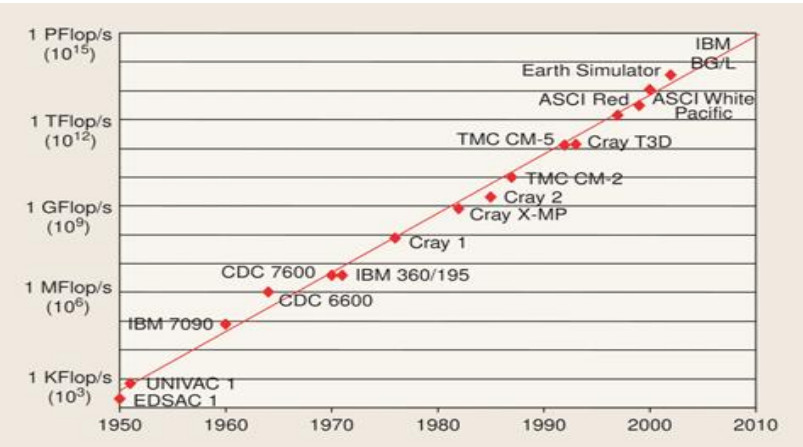


Fig.2.1 Moore's Law and Performance of Various Computers Over Time.

Moore’s law states that processor performance approximately doubles every 18 months. HPC performance is typically measured by two fundamental metrics: "Time" and "Number of operations". The most commonly used metric in HPC is "floating point operations per second" or "flops". These measurements include thousands (KFLOPS), millions (MFLOPS), billions (GFLOPS), trillions (TFLOPS), and quadrillions (PFLOPS) of floating-point operations executed per second, representing increasing levels of computational performance.Fig.2.1 clearly illustrates the consistent trend of Moore’s Law, which has held throughout much of the modern computing era, with performance increasing by approximately two orders of magnitude every decade. This ongoing advancement continues to push the boundaries of computing technology. However, as Moore’s law slows down and manufacturing reaches physical limits, managing heat and power becomes increasingly challenging due to higher processor speeds and core counts. In response to evolving user needs and power constraints, heterogeneous computing platforms have emerged as a viable solution. FPGA emerged as the best solution among all the heterogeneous computing platforms due to its parallel processing capabilities.

2.5 CPU

The Central Processing Unit (CPU) acts as the core of a computer system, executing program instructions by handling basic tasks like arithmetic calculations, logical decisions, control functions, and input/output operations. It processes information and oversees the functioning of all other hardware components. A basic CPU block diagram illustrates key units including the Arithmetic Logic Unit (ALU), the Control Unit (CU), internal registers, and data buses. These components collaborate to execute operations, manage control signals, and handle the movement of data within the processor.

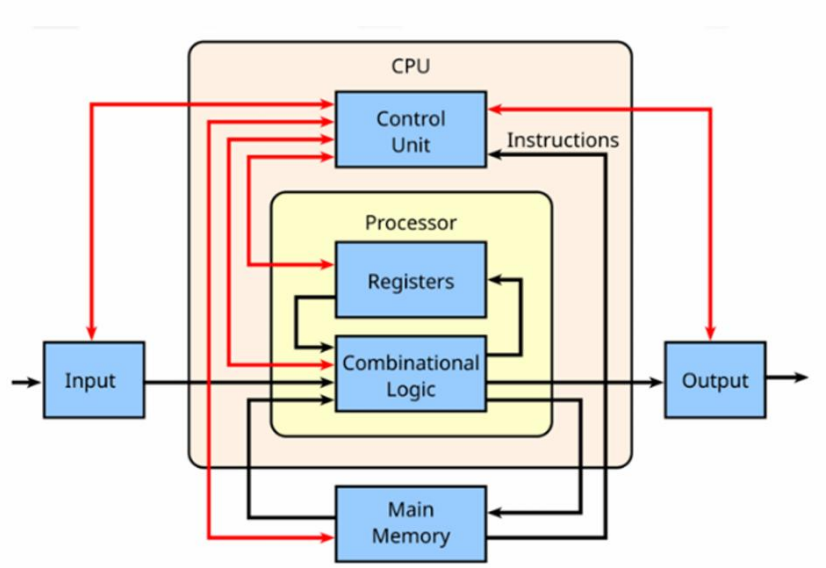


Fig.2.2 Block diagram for CPU

2.6 GPU

A graphical processing unit, or GPU, is a type of specialized processor that is mainly used for image and graphics processing. A GPU is designed to execute numerous basic computations simultaneously, which makes it perfect for tasks requiring massive parallel processing, as opposed to a CPU, which is optimized for general-purpose tasks.

The GPU performs parallel processing by using hundreds or thousands of small cores to handle many operations at once. It is responsible for graphics rendering, originally built to manage images, videos, and animations for visual display. Today, it is also widely used for data acceleration in fields like scientific computing, AI, and machine learning, enabling faster execution of large-scale mathematical tasks. Additionally, the GPU supports the CPU by offloading repetitive and computation-heavy operations, improving overall system performance.

The general block diagram, which depicts the basic functionality of the GPU, is given below:

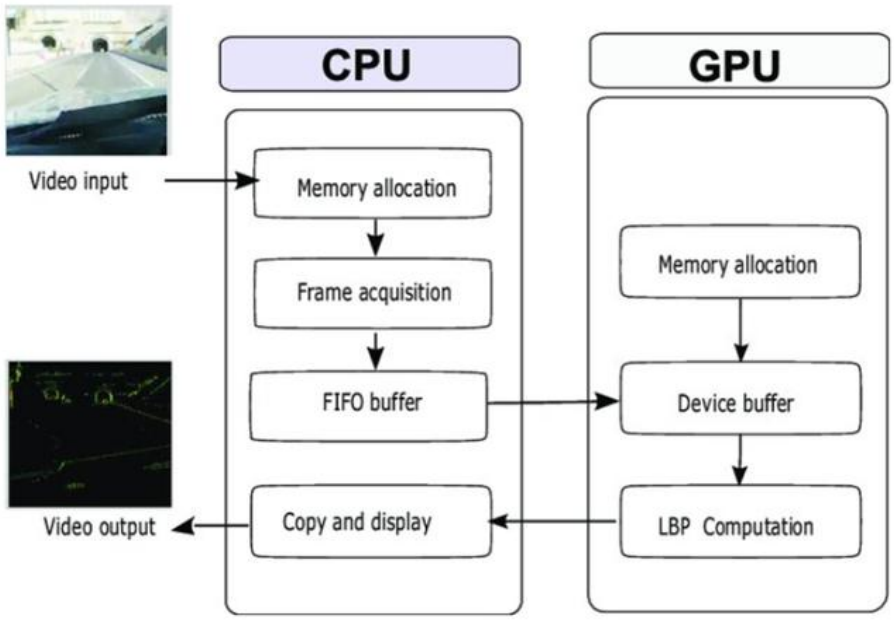


Fig.2.3 Block diagram of GPU

2.5 FPGA

FPGAs (Field-Programmable Gate Arrays) are customizable digital logic devices that can be programmed to perform any specified hardware function. They are built using a matrix of configurable elements (such as logic blocks, I/O modules, etc.) that are connected using built-in wiring channels and adjustable switches. The operations of these blocks and interconnections are governed by millions of SRAM (Static RAM) cells, which are loaded with configuration data during initialization to define the intended behavior. Users define this behavior using a hardware design language like VHDL or Verilog, or alternatively, leverage high-level synthesis (HLS) tools to convert code written in languages like C or OpenCL into HDL form. The HDL is then compiled through a sophisticated electronic design automation (EDA) toolchain into a binary bitstream that programs the configuration memory of the FPGA.

In contrast to designing a custom application-specific integrated circuit (ASIC), FPGAs provide much lower upfront development costs and a faster path to deployment. By utilizing readily available off-the-shelf FPGA hardware, an entire system can be built in a matter of weeks, eliminating the need for extensive physical design, layout, manufacturing, and validation typically required for ASICs. Additionally, FPGAs support easy post-deployment hardware modifications by loading a new bitstream in the field—hence the term "field-programmable." This capability makes FPGAs especially appealing for small to medium-sized projects, particularly in markets with rapid product evolution. These advantages have driven FPGA adoption across domains like wireless systems, embedded processing, data networking, ASIC emulation, and high-speed financial trading.

2.5.1 Internal architecture of FPGA

The internal architecture of an FPGA is composed of many blocks as follows: the **Configurable Logic Blocks (CLBs)** serve as the central units for computation, containing **lookup tables (LUTs)**, flip-flops, and multiplexers to perform both combinational and sequential logic operations. Surrounding these are **Input/Output Blocks (IOBs),** which handle data transfer between the FPGA and external components by supporting various I/O protocols. These blocks are interconnected through a **programmable routing network**, a flexible mesh of wires and switches that define the signal paths across the chip. To ensure proper timing and synchronization, clock management resources such as **phase-locked loops (PLLs)** and clock distribution trees are embedded in the architecture. The FPGA also includes **Block RAM (BRAM)** for fast, on-chip data storage and buffering, and **DSP slices**, which are specialized for high-speed arithmetic operations like multiplication and accumulation, essential for applications in digital signal processing and machine learning. At the foundation of the FPGA’s reconfigurability lies the configuration memory, which stores the

programming bitstream and enables the dynamic arrangement of logic functions and routing paths, making the FPGA a highly flexible and adaptable platform for a variety of digital applications.

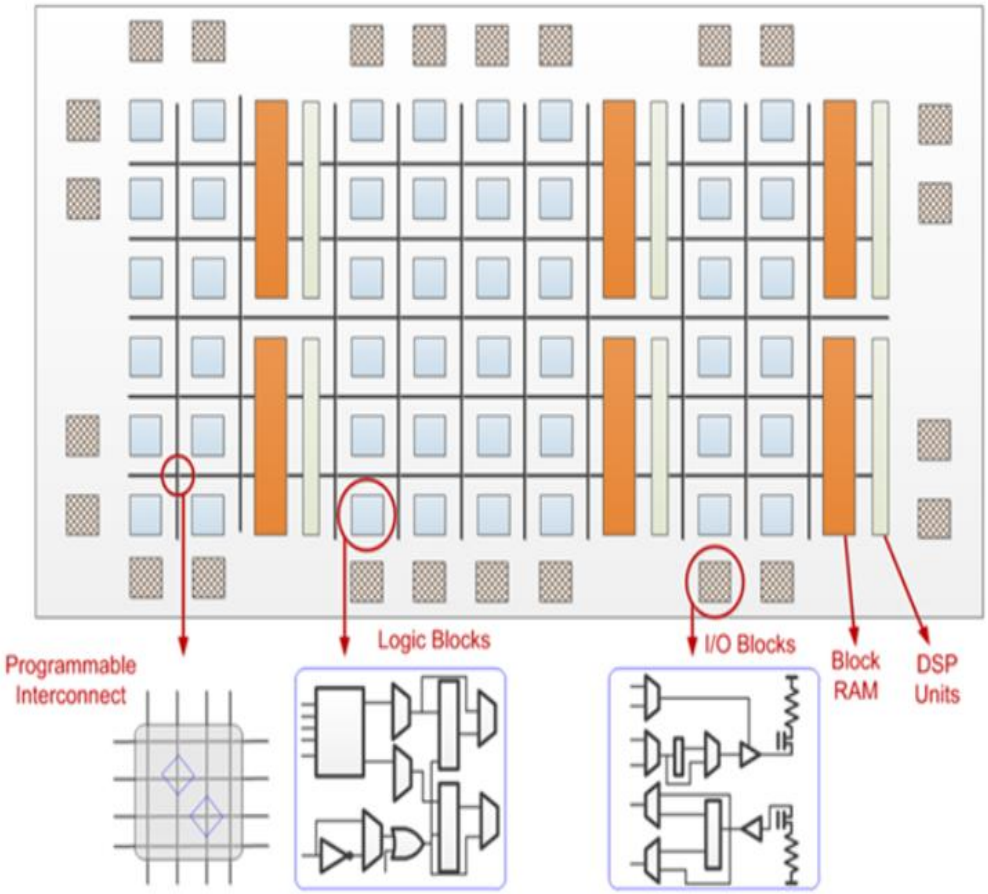


Fig.2.4 Internal architecture of FPGA

Configurable Logic Blocks (CLB)

A CLB is the core component of an FPGA, enabling it to adopt various hardware setups. Essentially, an FPGA is comprised of numerous CLBs, forming its structure. With thousands of these on modern FPGAs, they can be programmed to execute nearly any logical operation. Each CLB contains its own set of discrete logic elements, including look-up tables (LUTs), multiplexers, and flip-flops. The configurable logic block general diagram is shown below

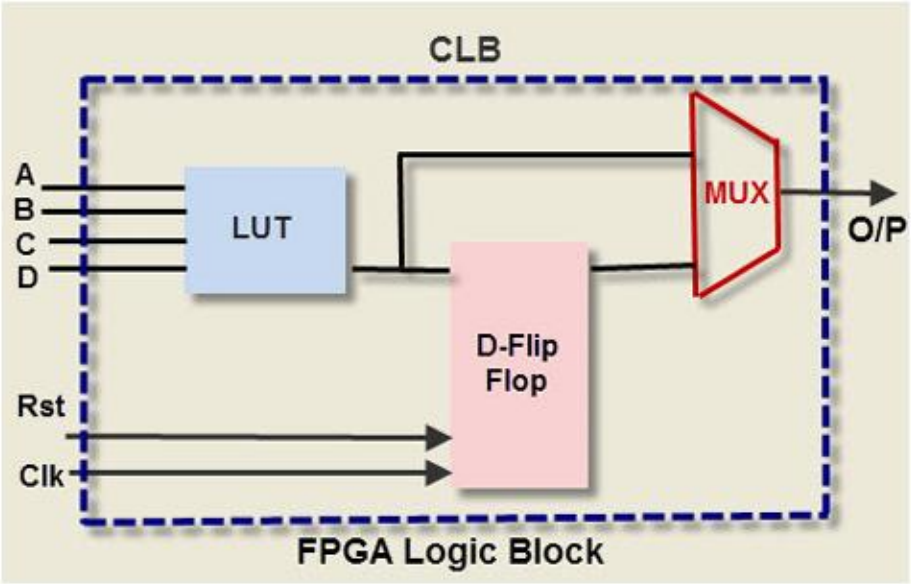


Fig.2.5 Block diagram of CLB

1) Look up table

The Lookup Table (LUT), a pivotal element within the CLB of an FPGA, serves as a cornerstone for its functionality. It operates as a customizable truth table, associating input configurations with corresponding output values. LUTs possess programmable capabilities, enabling them to execute various combinational logic functions. Their flexibility is paramount in digital circuit design, as they empower engineers to implement diverse logic operations tailored to specific requirements. From basic logic gates to intricate functions such as adders and multiplexers, LUTs provide a versatile platform for crafting custom digital circuits, accommodating a spectrum of computational tasks with precision and efficiency.

2) Flip-flop or Latches

A **flip-flop** is a basic digital memory element used to store one bit of binary data. It is edge-triggered, meaning it changes its output only on the rising or falling edge of a clock signal, making it ideal for synchronous circuits. Flip-flops are widely used in sequential logic for registers, counters, and state machines.

A **latch**, on the other hand, is a level-sensitive storage device that changes its output as long as the enable signal (or control signal) is active. It is used in asynchronous circuits where data needs to be stored or held without relying on a clock edge.

Flip-flops or latches, integral components within a CLB, play a vital role in FPGA functionality by providing the capability to both store and synchronize data. Acting as memory elements, they facilitate the retention of crucial state information within the FPGA, ensuring the stability and accuracy of ongoing processes. These flip-flops or latches serve multiple purposes within the digital circuitry: they can store intermediate results during complex computations, implement sequential logic to control the order of operations, and create memory elements for temporary or permanent data storage. Their versatility enables the FPGA to manage and manipulate data streams effectively, enhancing its overall performance and functionality in various applications.

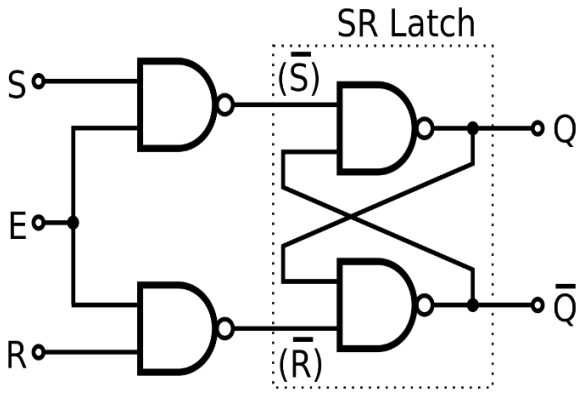


Fig.2.6 S-R flip flop along with S-R latch

3)Multiplexer (MUX)

The multiplexer (MUX) housed within a CLB is critical in managing data flow within an FPGA. It acts as a switchboard, enabling the selection of different inputs based on control signals. This capability allows signals from various sources to be routed and connected to different components within the CLB as needed. By providing this flexibility, the multiplexer

facilitates the adaptation of the FPGA to diverse operational requirements. For instance, it permits the selection of appropriate input signals for specific computational tasks or conditions, thereby enhancing the versatility and efficiency of the FPGA's operation. Additionally, the MUX is crucial in optimizing resource utilization within the CLB, ensuring that signals are efficiently directed to the appropriate destinations for processing or further routing.

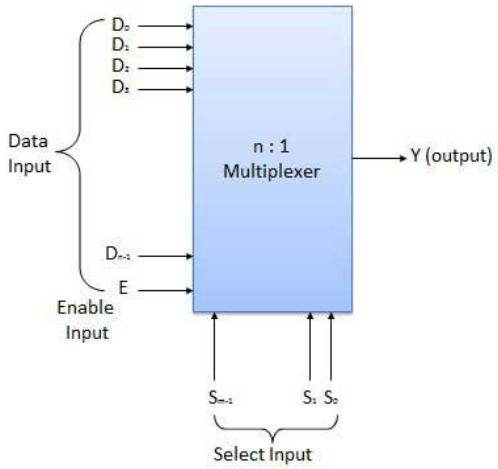


Fig.2.7 Multiplexer block diagram

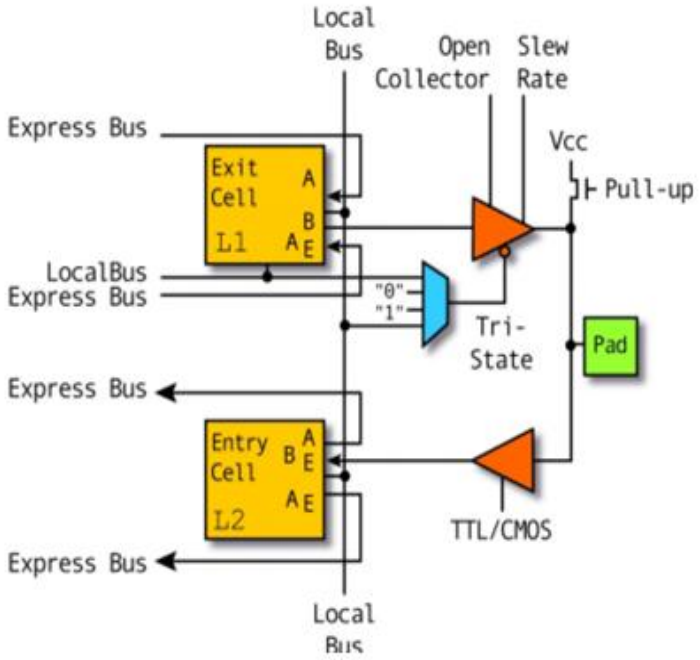
Programmable Interconnect

In FPGAs, routing consists of wire segments of varying lengths interconnected by electrically programmable switches. The density of logic blocks within an FPGA is influenced by the length and number of these wire segments used for routing. Balancing the number of connecting segments is crucial as it affects both the density of logic blocks and the space occupied by routing. Programmable routing links logic and input/output blocks to complete a user-defined design unit. This routing circuit comprises multiplexers, pass transistors, and tristate buffers. Pass transistors and multiplexers within a logic cluster establish connections between the logic units.

Programmable I/O Blocks

FPGAs feature flexible IO architectures that enable them to connect with a wide variety of devices, making them essential communication centers in many systems. Nevertheless, supporting multiple IO interfaces and standards through a single group of physical IOs poses considerable difficulties, demanding compatibility with different voltage ranges, signal properties, timing requirements, and communication protocols.

Fig.2.8 General block diagram for programmable I/O blocks



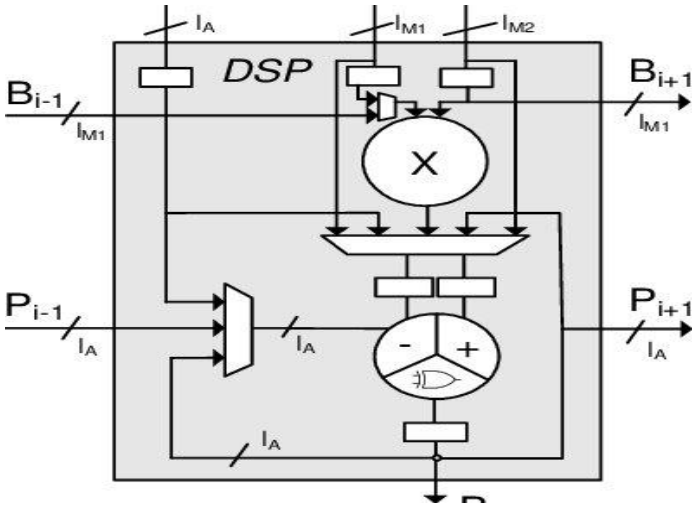
A Configurable Input/Output (I/O) Block, shown in Fig. 2.8, manages signal reception into the FPGA and signal transmission out of it. It contains input and output buffers that support control settings for three-state and open-drain modes. Usually, pull-up resistors are integrated at the outputs, while optional pull-down resistors can be added to internally terminate signals or buses, thus eliminating the requirement for external termination components. Moreover, the output polarity can be set to either active high or active low, and the slew rate can be adjusted to support either fast or slow transitions in signal rise and fall times Using flip-flops on the output side allows clocked signals to be sent directly to the output pins,

which helps minimize delays and ensures that setup time requirements for connected external devices are met. Likewise, placing flip-flops on the input side reduces the signal delay before it reaches internal logic, effectively lowering the hold time demands within the FPGA.

Digital Signal Processing (DSP) Slice

The DSP slice, also known as a DSP block or cell, is a specialized component within an FPGA designed specifically for handling digital signal processing tasks. Unlike the more versatile Configurable Logic Blocks (CLBs), which perform a wide range of general functions, DSP slices are purpose-built to efficiently carry out operations like multiplication, filtering, and accumulation. These units come with dedicated hardware elements, including multipliers, accumulators, and specialized adders, allowing them to execute DSP functions with greater speed and lower resource usage compared to implementations using CLBs. The inclusion of DSP slices greatly strengthens the FPGA's capacity for real-time signal processing, making them essential in fields such as telecommunications, audio processing, and image and video analysis.

Fig.2.9 Block diagram of DSP



Block Random Access Memory (BRAM)

Block RAM (BRAM) is the on-chip dedicated memory available within an FPGA, designed for high-speed data storage and access during processing. This embedded memory plays a vital role in buffering and temporarily holding data for use by logic elements. On most FPGA boards, memory configurations may vary, but BRAM stands out for being directly integrated into the chip. For instance, in Xilinx 7 series devices, each BRAM block typically has a fixed capacity of 36K bits. These blocks are flexible—they can be partitioned into smaller units or combined to form larger memory spaces as needed. BRAM also supports a variety of operating modes and can be configured with additional features, such as error correction capabilities, enhancing reliability in critical applications.

Internal clock circuitry

The internal clock circuitry of an FPGA is designed to manage and distribute timing signals efficiently across the chip. It includes specialized I/O blocks with high-drive clock drivers capable of handling strong clock signals. These drivers receive input from dedicated clock pads and route the signals onto global clock lines, which are engineered to ensure low skew and fast signal propagation throughout the FPGA. This infrastructure supports synchronous design, which is critical in FPGA development, as only these global clock lines can maintain uniform timing and reliable signal alignment across all logic elements.

2.6 Overview of FPGA Hardware Design Methodology

The standard hardware design flow for FPGA programming is shown in Fig.2.10. FPGAs need a specific hardware design process, in contrast to microcontrollers and microprocessors, which usually rely on software design. Below is a description of the key steps in FPGA programming.

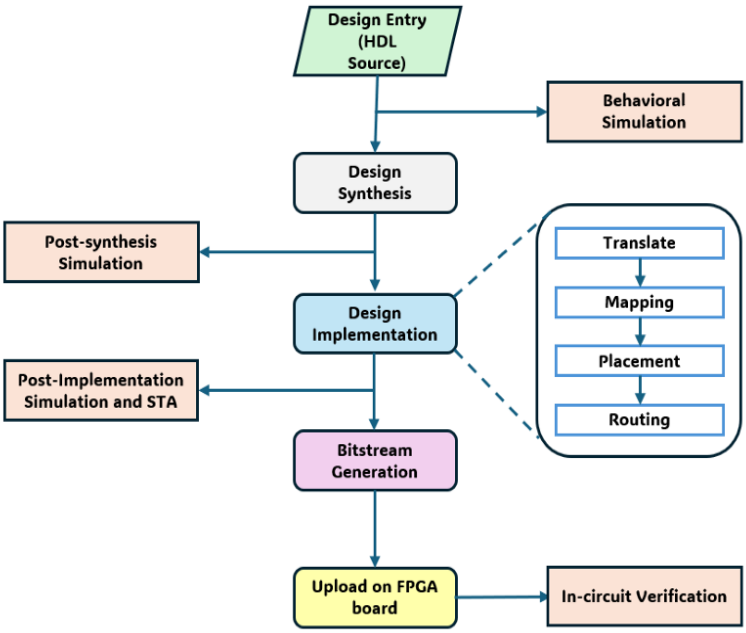


Fig.2.10 General flow of hardware design

The various blocks, which are interconnected above, when operating in synchronization with each other, give us the desired implementation of any logical and arithmetic circuit.

Design Entry

A hardware description language (HDL), a schematic editor, or an editor for finite-state machines (FSMs) can all be used to describe the logic of a design. This procedure entails selecting elements from a library and directly projecting the functionality of the design onto particular computational blocks. HDLs provide a means of defining the design either structurally or behaviorally for intricate designs where managing visual representation becomes challenging. Although Verilog and VHDL are the most popular HDLs, Handel-C, Impulse-C, and System are some C-like substitutes.

Behavioral Simulation

By contrasting the HDL's output with the behavioral model's, this step is crucial for confirming the HDL's accuracy. Functional testing is made possible by behavioral simulation using Electronic Design Automation (EDA) tools and the RTL (Register Transfer Level) description.

Design Synthesis

This process converts the HDL code into a device netlist, which represents the entire circuit using logical components. During synthesis, the tool verifies code syntax, analyzes the design's architectural hierarchy, and performs compilation along with various optimizations. The generated netlist is typically saved in a .ngc file format.

Design Implementation

The design implementation comprises the following steps

1. **Translate**

   During translation, netlists are combined into a single NGD file, followed by timing and logical design rule checks. Constraints from the user constraint file (UCF) are then integrated into the merged netlist.

2. **Map**

   During this stage, the tool applies timing and location constraints and distributes resources to basic logic components. The physical design database and a post-mapping Static Timing Analysis (STA) report detailing block and routing delays are then produced, along with optimization for the target device**.**

Place and Route This phase involves the placement and routing of the design, resulting in the post-place-and-route STA report detailing all nets and delays in the design.

Device Programming

The finalized routed design needs to be translated into a format compatible with the FPGA. To achieve this, the routed .ncd file is input into the BitGen utility, which produces a bitstream file that holds the complete configuration data required to program the FPGA.

Timing Analysis

In this phase, a timing analysis tool verifies that the implemented design satisfies the user-defined timing constraints, such as clock frequency, setup time violations, and hold time violations.

2.7 Software Design Flow for FPGA Using Xilinx Tools

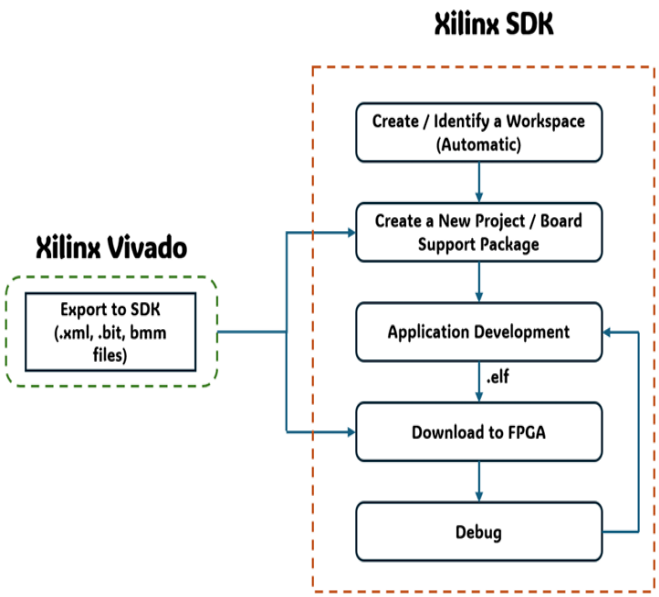


Fig.2.11 Block diagram for software design flow of FPGA

The standard workflow for developing a software application for a Vivado®-based embedded system using Xilinx® SDK includes the following steps:

1)Launch Xilinx SDK and choose to either open an existing workspace or create a new one using a hardware platform file generated by the Vivado® IP Integrator.

2)Start developing the software application. The SDK offers comprehensive documentation for software libraries and drivers provided within the Board Support Package (BSP).

3)The SDK automatically creates a default linker script for the application. we can modify the memory layout as needed using the linker script generation tools.

4)To test the application on the hardware target, set up a run/debug configuration. If required, load the hardware bitstream onto the FPGA device before executing or debugging the application.

2.8 Merits and demerits of FPGA

FPGAs offer several benefits. Their ability to be programmed at the logic level allows them to support high-speed and parallel signal processing, which are tasks that are frequently challenging for traditional processors. FPGAs are completely reprogrammable, even remotely, enabling numerous reuses in contrast to ASICs, which are fixed after fabrication. Faster time-to-market is made possible by their broad availability and rapid programming using HDL. Furthermore, FPGA development is less expensive than ASIC development, with no major Non-Recurring Engineering (NRE) costs and lower tool and development costs. Automated tools that manage timing, placement, and routing further streamline the design process by lowering the amount of manual labour and simplifying the design.

There are some restrictions on FPGAs. In contrast to the comparatively simpler C programming used in processor-based systems, programming them necessitates a thorough understanding of digital system design and VHDL or Verilog. In comparison to ASICs, they also typically use more power and provide less optimization flexibility. The initial FPGA selection is a crucial choice because, once chosen, the design is constrained by the resources of the device, which may limit the project's overall size and functionality. In contrast to ASICs, which are more affordable in large quantities, FPGAs are best suited for prototyping and small-scale production, but their cost decreases as production volume rises.

2.9 Boards utilized in data transmission and reception

As in the subsequent chapters, the primary focus of the thesis work has been centered around data communication between different peripherals. So, in this regard, different types of boards have been utilized. These are the **BASYS 3** Board and the **ZYNQ** Board.

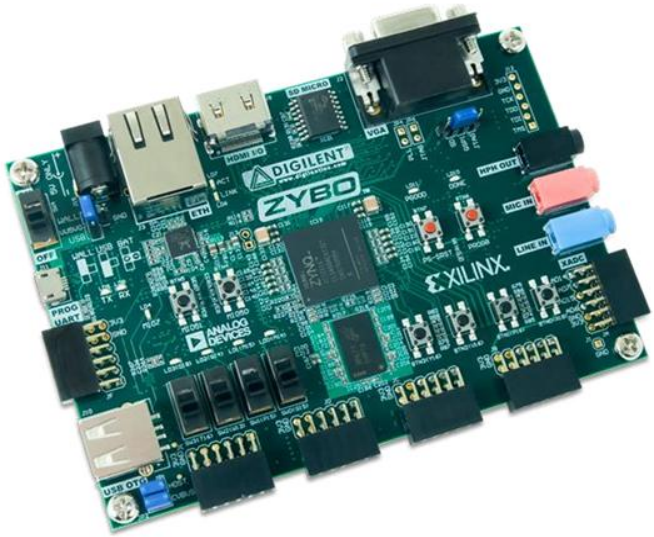


Fig.2.12 The ZYBO Zynq-7000 development board

A flexible, entry-level development platform for embedded software and digital circuit design is the **ZYBO** (Zynq Board). It offers an integrated solution that combines the flexibility of 7-series FPGA fabric with the power of **a dual-core ARM Cortex-A9 processor**, thanks to its construction around the **Xilinx Zynq-7000 Z-7010 SoC**, the smallest member of the Zynq family. Because of its All Programmable System-on-Chip (AP SoC) architecture, which facilitates smooth hardware/software integration, ZYBO is perfect for low-to-mid volume embedded applications, educational use, and rapid prototyping. A variety of multimedia and connectivity peripherals are included on the board, enabling full system development without the need for extra parts. There is plenty of space for customization and expansion thanks to its six MOD ports.

Key Features of the Zynq-7010 AP SoC

A DDR3 memory controller with eight DMA channels for effective data handling supports the Zynq-7010 AP SoC's potent 650 MHz dual-core ARM Cortex-A9 processor. It provides a wide range of low-bandwidth interfaces like SPI, UART, CAN, and I²C for general-purpose communication in addition to high-bandwidth peripherals like Gigabit Ethernet, USB 2.0, and SDIO.

The 4,400 logic slices that make up the integrated programmable logic, which is comparable to an Xilinx Artix-7 FPGA, each have eight flip-flops and four 6-input LUTs. It has two clock management tiles with PLL and MMCM for accurate timing control, 80 DSP slices for sophisticated arithmetic operations, and 240 KB of block RAM. High-performance, mixed-signal processing is also made possible by its on-chip XADC (analog-to-digital converter) and support for internal clock speeds exceeding 450 MHz.

2.10 BASYS 3 Board

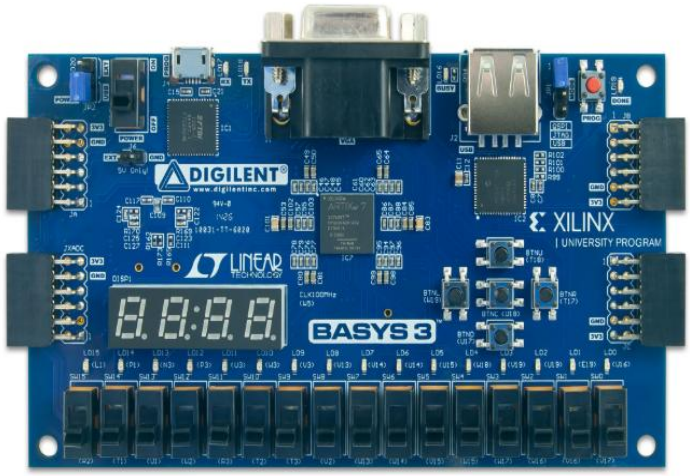


Fig.2.13 Digilent Basys 3 Artix-7 FPGA Board

The **Basys 3** board is a fully integrated, user-friendly digital circuit development platform built around the Xilinx Artix-7™ FPGA (**part number XC7A35T-1CPG236C**). Designed for both beginners and advanced users, it offers a cost-effective solution for implementing a wide range of digital designs, from simple combinational circuits to more advanced sequential systems like embedded processors and controllers. Featuring built-in USB, VGA, and other I/O ports, along with an ample set of switches, LEDs, and various I/O devices, the board enables users to develop and test numerous projects without needing external hardware. Additionally, the board provides sufficient unallocated FPGA I/O pins for further expansion via Digilent Pmods or custom peripherals, making it a versatile platform for both academic and practical applications.

Key Features of the Basys 3 Board

With more capacity, faster speed, and better resources than previous generations, the Artix-7 FPGA is made for high-performance logic applications. The Artix-7 35T version incorporates roughly 33,280 logic cells arranged in 5,200 slices, each of which has eight flip-flops and four 6-input look-up tables (LUTs). In addition, it has 90 DSP slices for effective

arithmetic operations, 1,800 Kbits of fast block RAM, and internal clock speeds higher than 450 MHz. It has five clock management tiles with phase-locked loops (PLLs) to control timing and synchronization. Furthermore, an on-chip analog-to-digital converter (XADC) is housed in the FPGA, allowing for internal analogue signal monitoring.

Peripherals available on the Basys 3 Board

The **Basys 3** board also has an enhanced array of ports and peripherals to support a wide range of digital designs. It features 16 user switches, 16 user LEDs, 5 pushbuttons, and a 4-digit 7-segment display for basic input/output operations. For expansion, it includes three Pmod ports and an additional Pmod dedicated to XADC signals. A 12-bit VGA output supports display capabilities, while communication and programming are facilitated through a USB-UART bridge, Serial Flash memory, and a Digilent USB-JTAG port. Moreover, it includes a USB HID host interface, allowing direct connection of mice, keyboards, and memory sticks.

2.11 UART as a Communication Protocol

UART is a fundamental communication protocol in various systems, including embedded systems, microcontrollers, and computers, primarily due to its simplicity and efficiency. Unlike other communication protocols, such as SPI or I2C, UART requires only two wires for transmitting and receiving data, making it a preferred choice in many applications. UART operates on asynchronous serial communication at its core, offering configurable speeds to accommodate diverse system requirements. In this asynchronous mode, data transmission occurs without a synchronized clock signal between the transmitting and receiving devices. Instead, UART devices rely on predefined baud rates to ensure proper data transfer.

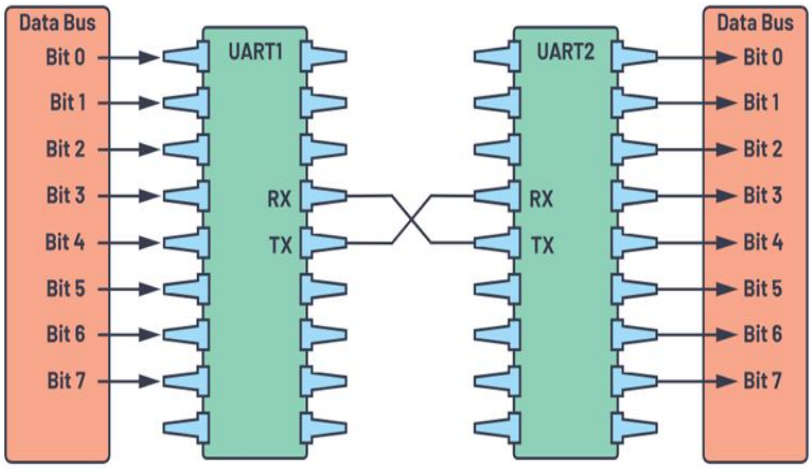


Fig.2.14 UART with Data Bus

Each UART device consists of two fundamental signals: Transmitter (Tx) and Receiver (Rx), which enable the exchange of serial data between devices, ensuring efficient communication. As illustrated in Fig. 2.14, the transmitting UART receives data in parallel format through a control data bus. This data is then converted into serial form and transmitted bit by bit to the receiving UART, where it is deserialized back into parallel form for use by the receiving device.

Although UART communication is relatively straightforward, proper synchronization is critical for reliable data exchange. Both the transmitting and receiving devices must operate at the same baud rate, allowing for a deviation of up to 10%. Exceeding this tolerance can result in timing mismatches, potentially causing errors in data transmission and reception. Therefore, aligning baud rates accurately is vital for effective UART communication. The typical UART frame structure is depicted in Fig. 2.15.

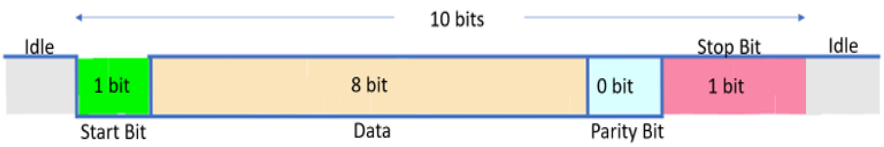


Fig.2.15 UART frame of data transmission and reception

2.12 Merits of UART as communication protocol

There are following advantages of using UART as communication protocol for data transmission and reception.

1) It operates efficiently by utilizing only two wires, simplifying the hardware setup required for communication between device.
2) Unlike synchronous communication protocols, UART does not necessitate a dedicated clocksignal,reducing system complexity.
3) The data format which is sent from one device to another device includes a parity bit to check that if any error is present in the data packet.
4) UART's flexibility allows the data packet structure to be modified, provided that both the transmitter and receiver are configured to match the changes.

2.13 Demerits of UART as communication medium

Despite its various advantages as communication protocal, it has certain limitations too.

1)A significant drawback is the limited size of the data frame, which can be a maximum of 10 bits. This constraint can create difficulties when handling larger data sets.

2) UART does not support multi-master or multi-slave configurations, which restricts its use in scenarios that require such system architectures.

3) Ensuring consistent baud rates across UART devices is essential, as deviations greater than 10% can cause synchronization problems and data

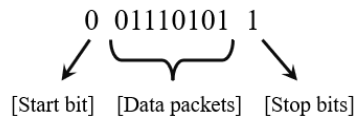
transmission errors, making reliable communication between devices challenging.

2.14 Identification of communicated data

In UART communication, data is transmitted by first sending a start bit, represented by a “0.” This is followed by the 8-bit data. Once the entire data packet is sent, a stop bit, represented by a “1,” is transmitted to indicate the completion of the data transmission process.

Another important point to note is that after the start bit is transmitted, the data packet is sent beginning with the least significant bit (LSB), proceeding sequentially up to the most significant bit (MSB).

For example, suppose if it is required to send the data from FPGA to a PC and if the data is specified as (10101110) then first bit will be start bit that is 0 and after that LSB of the given data like 0 in this case and subsequentially other bits till MSB is reached and after that stop bit 1 as final bit to show the completion of transmission/reception process.



The above text accurately represents the format in which data is transmitted or received by any device.

Chapter 3

System Design and Implementation

In this module the whole work carried out throughout my project will be discussed in length and breadth. Here all the hardware and software components involved in the realization of my project will also be discussed in vivid details.

As we know, heterogeneous computing plays an important role in the realization of complex systems where there is large-scale involvement of data in executing the assigned task. In view of the above problem statement several tasks have been performed, which eventually led to the development of a prototype system, which, if implemented successfully with some rectification in it, will surely lead to the real implementation of MD simulation using heterogeneous computing.

To realize the final work following steps have been taken and these are as follows:

1) To establish communication between PC and an FPGA by sending and receiving the data correctly.
2) To store the data in the BRAM of FPGA.
3) To check the integrity of sent data by using a DSO.
4) To check the integrity of received by using inbuilt led on the BASYS 3 Board.
5) Before establishing the interboard communication between two Basys 3 Board the signal strength and its integrity is verified by using STM 32 Microcontroller and DSO.
6) Finally, the interboard communication has been implemented and various arithmetic and logical operations were performed on

one board and its result were displayed and stored on the other board.

To realize all above functions some certain IPs are used which will be discussed further in details so that functions of each IP can be understood properly.

3.1 Xilinx FPGA IPs

3.1.1 Microblaze: Soft core processor [8]

MicroBlaze is a 32-bit general-purpose soft-core processor based on RISC architecture. It has a 32-bit general-purpose register file, follows RISC Harvard Architecture, and includes a 3-stage pipeline with an interrupt module. Using the Local Memory Bus (LMB), MicroBlaze accesses on-chip memory and supports IEEE 754 single-precision floating point format. It also features an instruction cache (IC), data cache (DC), exception handling, a debug module, and a barrel shifter. MicroBlaze supports most Xilinx FPGA families like Artix-7, Kintex-7, and Spartan, but not the Zynq family. Xilinx provides the Software Development Kit (SDK) for programming MicroBlaze[10], supporting C and C++, and controlling all connected peripheral IPs [9]. Figure 3.1 shows the MicroBlaze IP configuration.

Fig.3.1 Microblaze IP block diagram

3.1.2 AXI UartLite:

The AXI UARTLite[11] is a control interface designed for asynchronous serial data transfer, supporting full-duplex communication. It provides access to control and data registers through an AXI4-Lite interface and facilitates reliable data transmission. Both the transmitter and receiver are equipped with FIFOs, each with a capacity of 16 bytes. The module supports configurable baud rates, such as 9600, 115200, 421800, and 921600, allowing flexibility based on application requirements. It performs parallel-to-serial conversion for data received via the AXI4-Lite interface and serial-to-parallel conversion for data received from a serial input. These capabilities make it suitable for communication between a processor and external serial devices. Figure 3.2 illustrates the AXI UARTLite IP.

Fig.3.2 AXI UartLite IP

3.1.3 Constant IPs:

These are standard IPs used to provide constant input to realise various arithmetic and logical operation. Suppose if an addition is need to be performed using an adder Ip and we want constant input then these constant IPs are utilised. There are two factors that decide the input value in a constant IP and those are input width and constant value according to the range of input width. If input width is 4 then constant input value will be ranged from (0 to 15).

this constant Ip has input width 4 and we can take any fixed input value from 0 to 15.

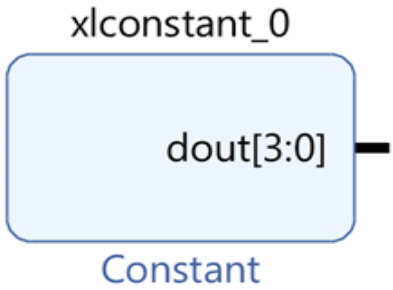


Fig.3.3 Constant

3.1.4 AXI BRAM Controller:

The AXI BRAM Controller IP is a memory interface block provided by Xilinx that enables communication between an AXI-based master, such as the MicroBlaze processor, and Block RAM (BRAM) in an FPGA. It acts as a bridge between the AXI4 interface and the internal dual-port BRAM, supporting read and write operations through standard AXI4 or AXI4-Lite protocols.

This IP is typically used in embedded systems where fast and deterministic access to on-chip memory is required. It supports single and burst transactions and provides configuration options to match system requirements, including data width and memory depth. The AXI BRAM Controller ensures efficient memory utilization and simplifies system integration within the Vivado IP Integrator environment.

Fig.3.4 AXI BRAM CONTROLLER IP

3.1.5 Block Memory Generator:

The Block Memory Generator (BMG) IP is a versatile memory generation tool provided by Xilinx that allows designers to configure and instantiate block RAM (BRAM) resources in FPGAs. It supports both single-port and dual-port configurations, enabling simultaneous read and write operations. The BMG IP can be customized for various memory depths and data widths, and it supports features such as initialization from memory files (.coe), write enable control, and byte-level masking.

This IP is commonly used to create memory blocks for data storage, lookup tables, buffering, and caching within custom hardware designs. When used in combination with the AXI BRAM Controller, it forms a complete solution for memory-mapped data access in AXI-based embedded systems.

Fig.3.5 Block memory generator IP

This block memory generator can be used in dual mode RAM port also.it is used in two modes 1) standalone mode 2) BRAM controller mode.

While performing data storage operation we use it in BRAM controller mode. The AXI BRAM Controller depends on the Block Memory Generator (BMG) to provide the actual on-chip memory (BRAM) for storing data. In turn, the BMG relies on the AXI BRAM Controller to manage and interface with AXI-based masters (like MicroBlaze), handling read/write transactions. Together, they enable seamless memory-mapped access in AXI systems.

3.1.6 AXI Interconnect:

The AXI Interconnect [12] core is specifically designed for use within a Vivado® IP Integrator block design in the Vivado Design Suite. It is a hierarchical module that integrates several LogiCORE™ IP infrastructure components, which are configured and interconnected during system design. These individual infrastructure cores can also be independently added to a block design or selected directly from the Vivado IP Catalog for use in HDL-based designs.

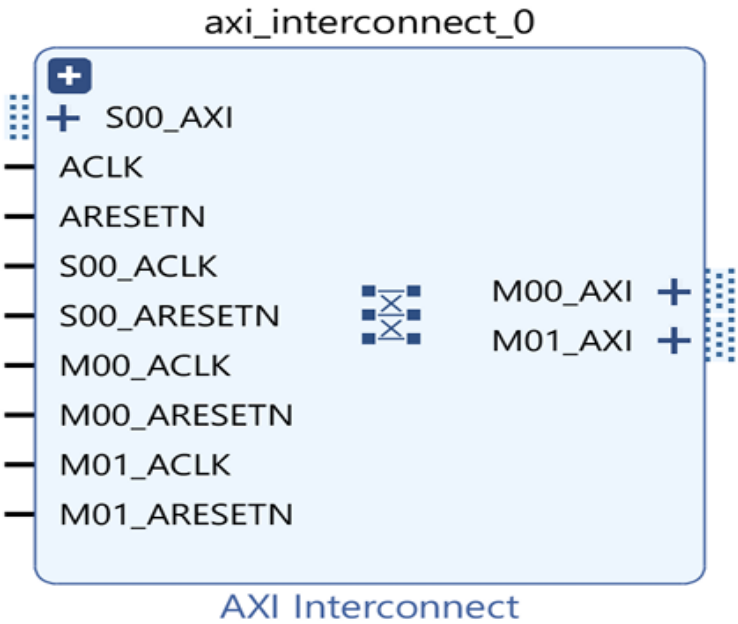


Fig.3.6 AXI Interconnect IP

The AXI Interconnect core enables the integration of multiple AXI master and slave devices, even when they differ in data width, clock domains, or AXI sub-protocols (AXI4, AXI3, or AXI4-Lite). When there are mismatches between the interface properties of connected devices and the internal crossbar switch, the interconnect automatically infers and connects the required infrastructure cores to perform the necessary protocol and signal conversions.

3.1.7AXI GPIO IP:

The AXI GPIO (General Purpose Input/Output) IP core is a peripheral provided by Xilinx in Vivado, designed to allow communication between a processor (like MicroBlaze) and external I/O devices (e.g., LEDs, switches, buttons, or custom hardware signals).

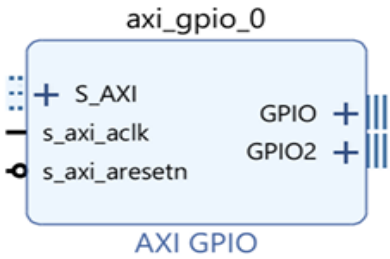


Fig.3.7 AXI GPIO IP

The AXI GPIO IP is a lightweight, AXI4-Lite compliant peripheral used to interface a processor like MicroBlaze with external input/output devices. It offers up to two independent 32-bit channels, each configurable as either input or output, but not both simultaneously. Commonly used in Vivado designs, this IP enables memory-mapped access to devices like LEDs, switches, or sensors. It is controlled via C/C++ code using the XGpio driver and connects to the processor through the AXI Interconnect. Easy to configure in the Vivado IP Integrator, it simplifies hardware/software communication without requiring complex protocols, making it ideal for debugging, testing, or real-time control applications. The I/O signals are assigned to FPGA board pins using an XDC constraints file. This IP can be used in input as well as in output mode based on the requirement of application. When it is required that processor should read the output of any specific ip like adder Ip output then it is used in all output mode and when we need to give variable input to adder IP by mapping the input to the switches in the given peripheral (BASYS 3 Board) and each time processor reads the input externally provided by the user via AXI GPIO.

3.1.8 Clocking Wizard:

The Clocking Wizard is an IP core provided by Xilinx in Vivado that simplifies the process of generating and managing clock signals in FPGA designs.

Function of Clocking Wizard: The Clocking Wizard is used to generate clock signals of desired frequencies by taking an input clock and producing one or more output clocks with specific frequencies, phases, and duty cycles. It configures and instantiates clock management tiles like MMCM (Mixed-Mode Clock Manager) or PLL (Phase-Locked Loop) internally to achieve these transformations. The wizard allows you to set output clock parameters, enable dynamic reconfiguration, and include reset or feedback options, providing precise and stable clocking needed for timing-sensitive logic and synchronized subsystems.

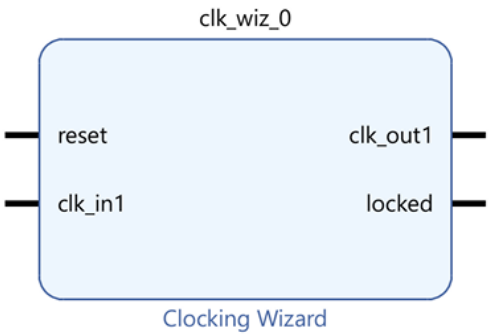


Fig.3.8 Clocking Wizard IP.

3.1.9 Adder/Subtractor IP:

This ip is used to perform addition and subtraction. Here we can modify the input width according to the requirement of our application. Here carry in and carryout are also specified according to the need of the application project. This is standard ip mainly used to verify add/sub operation on FPGA and the results of ip is mapped to the led available on the BASYS 3.

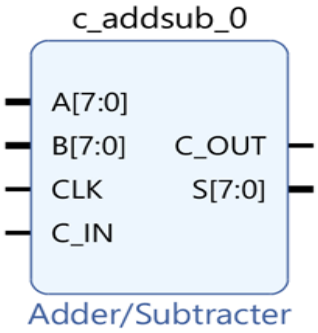


Fig.3.9 Adder/Subtractor IP

This ip has taken two 8-bit input and one bit Cin and the result is stored in sum S [7:0] and carry out is stored in Cout.By this IP the application project is implemented in such a way that we have utilised two BASYS 3 boards .one board is used as transmitter where this IP is being employed for add operation and the result of this IP is being fed to receiver board to display it on the LED of the receiver board. The result of the Adder IP is being read by the MicroBlaze via GPIO and it is sent to receiver board via UART communication protocol.

3.2 STM 32 Microcontroller:

The Nucleo-L476RG development board features the STM32L476RG, a powerful yet ultra-low-power microcontroller from STMicroelectronics' STM32L4 series. Built around an ARM Cortex-M4 core with a Floating-Point Unit (FPU), it operates at up to 80 MHz, delivering around 100 DMIPS performance. It includes 1 MB of Flash, 128 KB of SRAM, and supports EEPROM emulation and external memory interfaces like SPI/QSPI. Designed for energy-efficient applications, it offers multiple low-power modes (Sleep, Stop, Standby, Shutdown), along with features like a real-time clock, low-power timers, and wake-up capabilities. The microcontroller is equipped with a rich set of peripherals, including high-speed 12-bit ADCs, DACs, operational amplifiers,

comparators, as well as I2C, SPI, UART/USART, CAN, USB 2.0 FS, and SDIO interfaces. It also includes hardware security features like CRC, a random number generator, and read/write protection.

The Nucleo-L476RG board itself offers a user-friendly platform with Arduino Uno V3 and ST Zio connectors for expansion, along with an on-board ST-LINK/V2-1 debugger/programmer that also acts as a virtual COM port and mass storage. It can be powered via USB or external sources and includes user LEDs and buttons for interaction. Development is supported through STM32CubeIDE, STM32CubeMX, Mbed OS, and Arduino, as well as commercial toolchains like Keil and IAR. This board is ideal for prototyping low-power embedded systems in applications such as IoT, wearables, sensor-based systems, and portable medical devices.

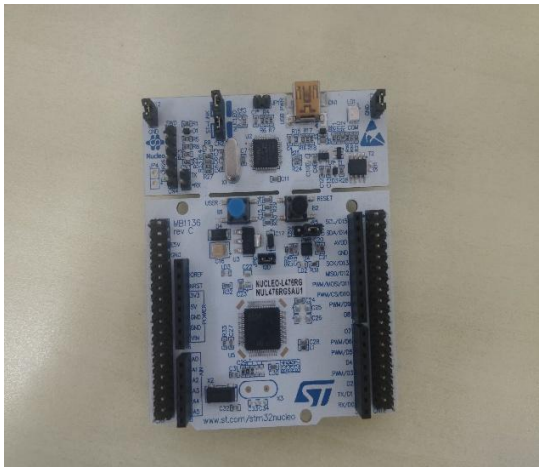


Fig.3.10 STM32 microcontroller(nucleo-L476RG)

3.3 Jumper Wire:

These wires are used especially while connecting the two boards either one STM32 to BASYS 3 or BASYS 3 to BASYS 3 for interboard communication through PMOD terminals present in the hardware Board.

So, all the components that will be used for the implementation of interboard communication imitating the MD simulation are discussed above

in vivid details. From the next section their implementation will be discussed thoroughly.

3.4 DSO:[17][18]

A Digital Storage Oscilloscope (DSO) is a test instrument used to capture, display, and analyze electrical signals in digital form. In your setup, the DSO is used to observe digital signals transmitted from the Basys 3 FPGA board and the STM32 microcontroller. When these boards send data, such as via UART[13], the DSO captures the voltage transitions over time and displays them as waveforms on its screen. This allows you to verify if the signals are being correctly sent, check the timing, bit patterns, and voltage levels, and troubleshoot communication issues if needed. The DSO's ability to pause, zoom, and measure signal properties in real-time makes it an essential tool for debugging and validating digital communication between embedded systems.

3.5 Implementations of data transmission and reception between BASYS 3 [14] board and PC

In this section the block diagram utilized for sender and receiver will be given below. **The required C code for this implementation will be given separately in a specific section called [Code Section]**.

The results of the data transmission and reception have been shown in the Chapter 4.

Here, data transmission is verified by sending the data from FPGA to PC, and the data is received on the terminal GTK term. The send data is shown on the GTK terminal in (**chapter 4)Fig.4.1**

This block diagram is used for data transmission and reception between FPGA and PC.

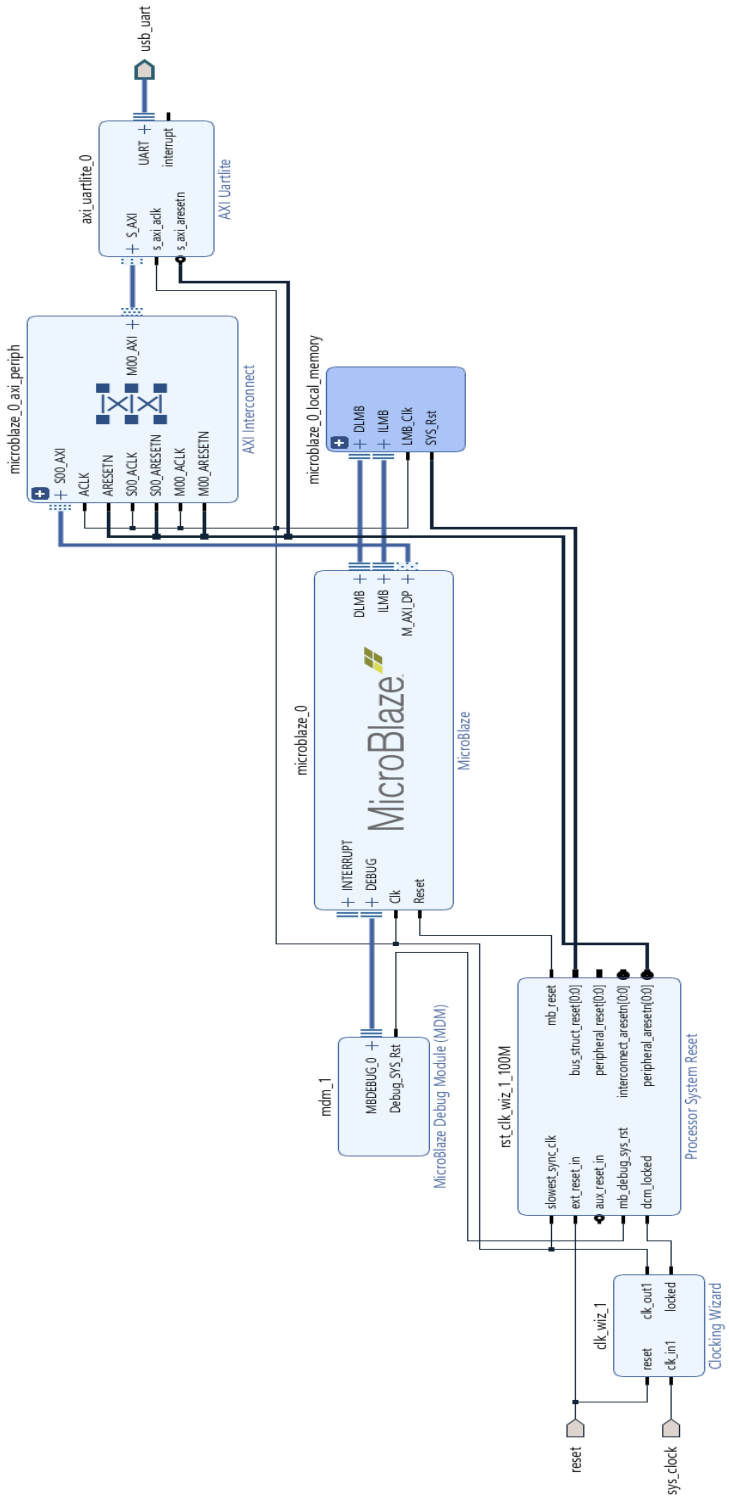


Fig.3.11 Block diagram for implementation of sender and receiver

3.6 Block diagram for data Storage operations in BRAM of FPGA[15]

Before doing any operation, we need to ensure that how to effectively store the data into the memory of FPGA for HPC application.

To store the data, we need different IPs to execute these operations and these IPs are Microblaze, AXI UartLite, Block memory generator BRAM controller and AXI interconnect any many more associated with Microblaze. The block diagram is shown below in **Fig.3.12**.

The C code to execute the data storage operations will be given separately in the **[code section]**.

3.7 Block diagram for data sending operation from FPGA to DSO

The **Fig.3.11** will be used for the data transmission from FPGA to DSO and the results of this data sending operation is shown in the chapter 4 Fig.4.6 where it is clearly shown that data (AA) is correctly received on the DSO via PMOD and the signal strength of the received signal is 3.3 Volt. The C code associated with sending the number from FPGA to DSO is given in the [**code section**].

3.8 Block diagram for receiving the data on LED of the receiver board

Here to verify that the send data is correctly received we utilised the LEDs available on the BASYS 3 board and upon receiving any number the LEDs

shows correct equivalent Binary number. The block diagram related to it is given below and C code related to it is given in the [code section].

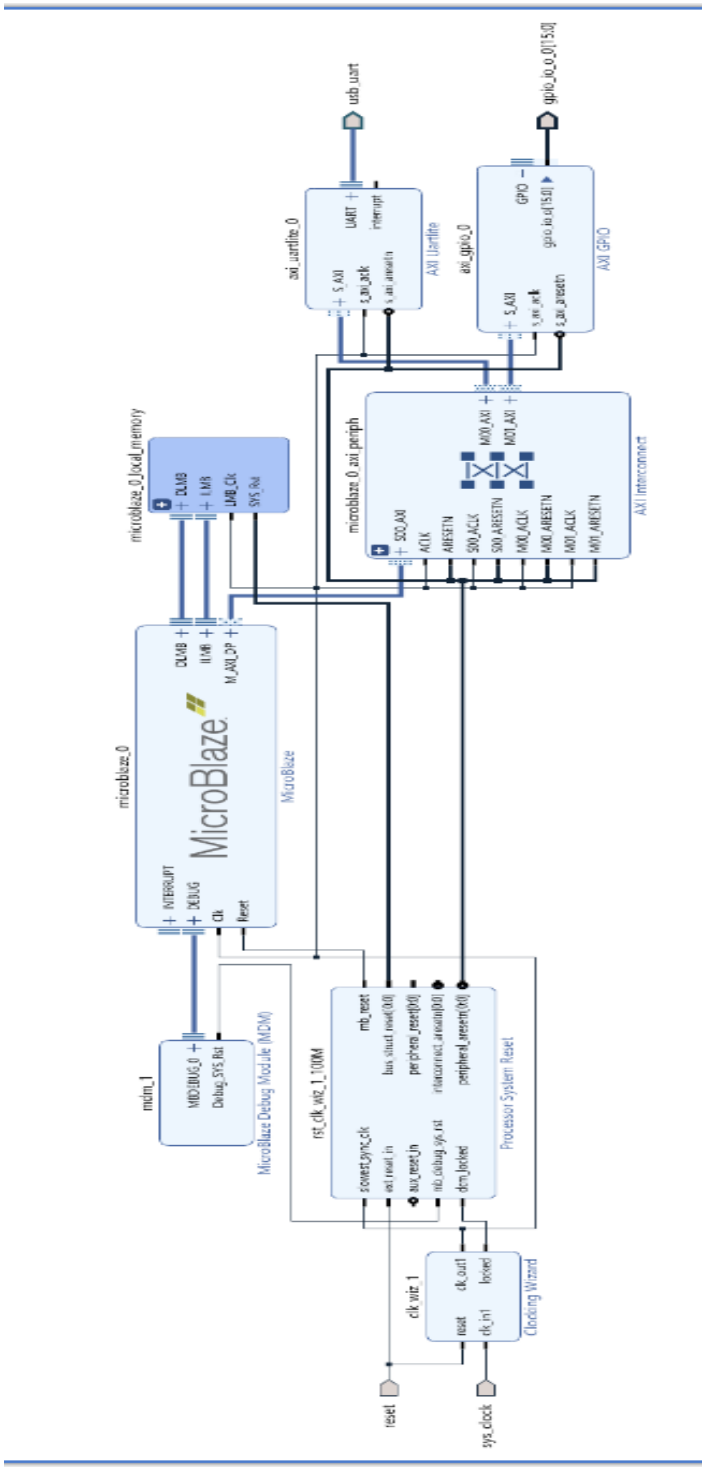


Fig.3.13 Block diagram for receiver to display received number on LED

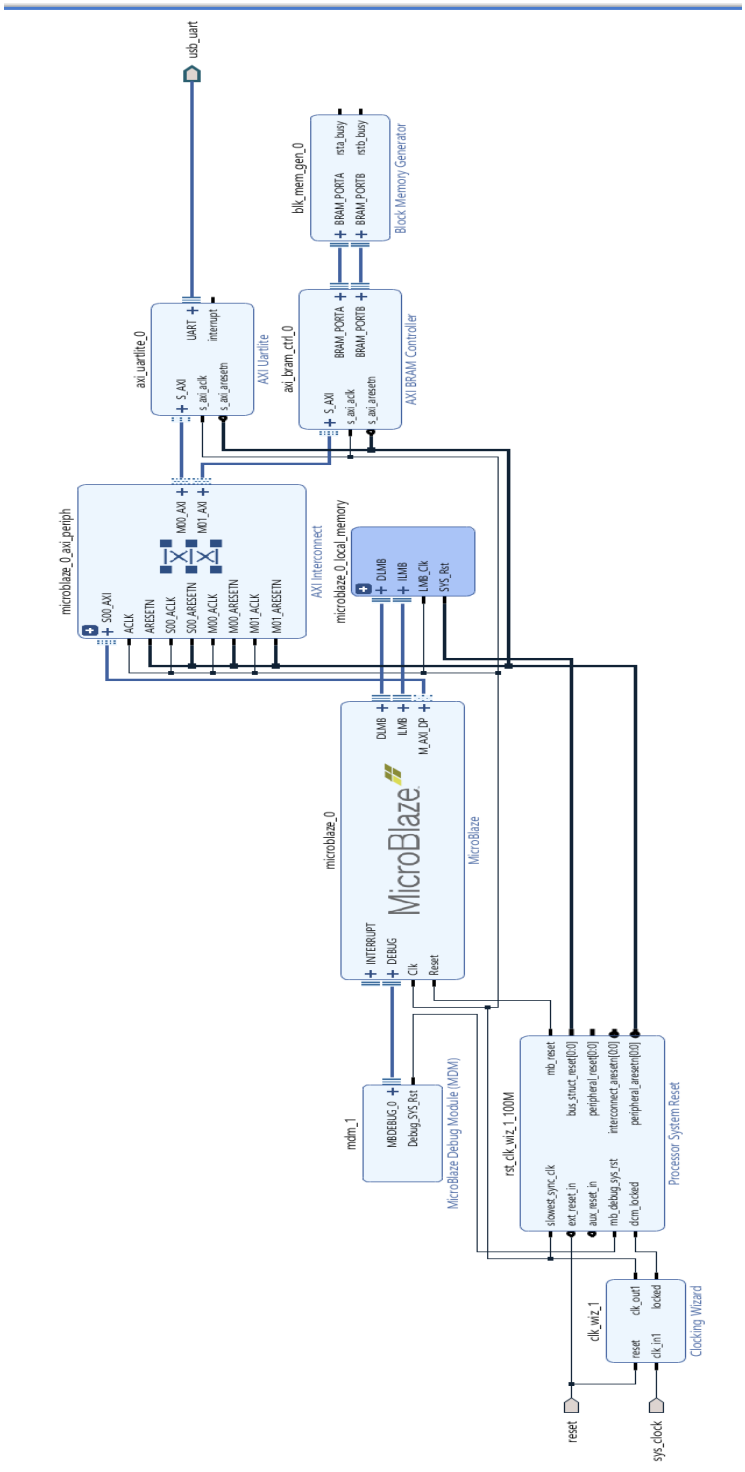


Fig.3.12 Block diagram for the implementation of data storage operation in BRAM

3.9 Block diagram for data transmission between STM 32 microcontroller and FPGA

In this section STM 32 is used as sender and BASYS 3 is used as receiver and the data is sent (BA) (10111010) from STM 32 to BASYS 3 and the results of it is shown in the **Fig.4.7 and Chapter 4.**

Fig.3.13 is used as block diagram for implementing this operation and the C code related to it is given in [**code block**].

3.10 Block diagram for data exchange between two BASYS 3 board.

The Fig.3.13 is used as receiver block and the Fig.3.11 is used as sender block and the interboard communication is established between two BASYS 3 boards. The results of data transmission and reception is shown in the **chapter 4 Fig.4.8 and Fig.4.9**.

3.11 Block diagram for implementation of ALU operation

To realize this operation Fig.3.13 is used as receiver block and the sender block is used which is given below in Fig.3.14. The C code that is used to realize this operation is given in [code block].

The various results obtained from ALU operations is given in the Chapter 4 and in Fig.4.10, Fig.4.11, Fig.4.12, Fig.13 where add, sub, multiplication and division are given respectively in all the figures mentioned in chapter 4.

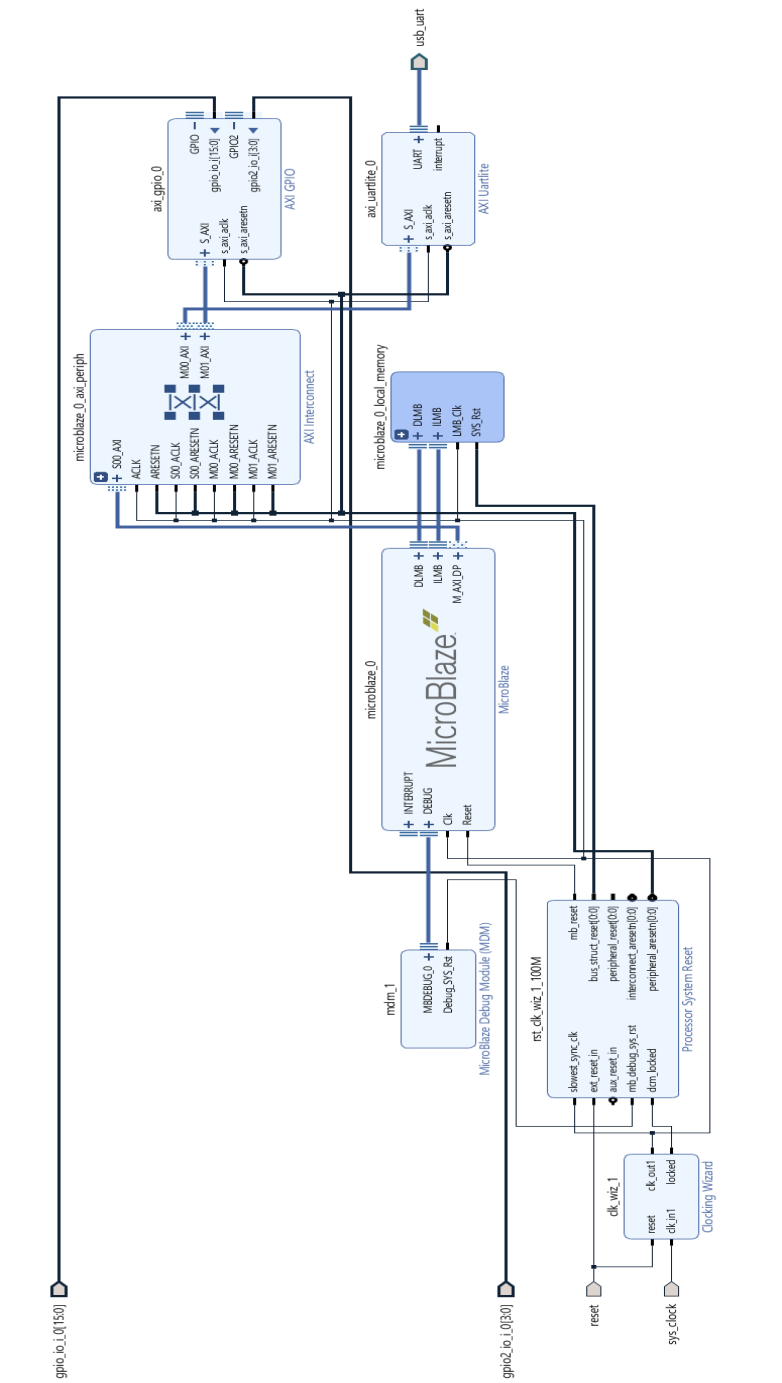


Fig.3.14 Block diagram for sender to implement ALU operations

3.12 Block diagram to implement Addition operation using Adder IP

To realize this operation constant IPs are used to provide input to the adder IP and results are stored in the receiver block. The output of adder IP is given to AXI GPIO through which Microblaze reads the output value and send it to the receiver block where the summation is successfully received. After receiving the result, the equivalent binary number is shown on the LEDs of the receiver.

The block diagram shown in the figure 3.16 is used as sender block to send the summation data and Fig.3.13 as receiver block. all the results pertaining to it are given in the Chapter 4 and Fig.4.14.

3.13 Block diagram to realize the data transmission and reception using ZYNQ board

The block diagram of ZYNQ board is given below in the Fig.3.15 to realise the data transmission and reception operation between ZYNQ board and PC.all the data sent from ZYNQ board to PC and the received in between the same peripherals are shown in the chapter 4 and in Fig.4.15,4.16 and 4.17 for receiving the data and showing it on the LEDs.

So, these are all the tasks that were performed to imitate the MD simulation process and to some extent it is applied in case of adder IP.

The ZYNQ [19] [20][21][22] Board for realization of sender and receiver between the PC and ZYNQ board.

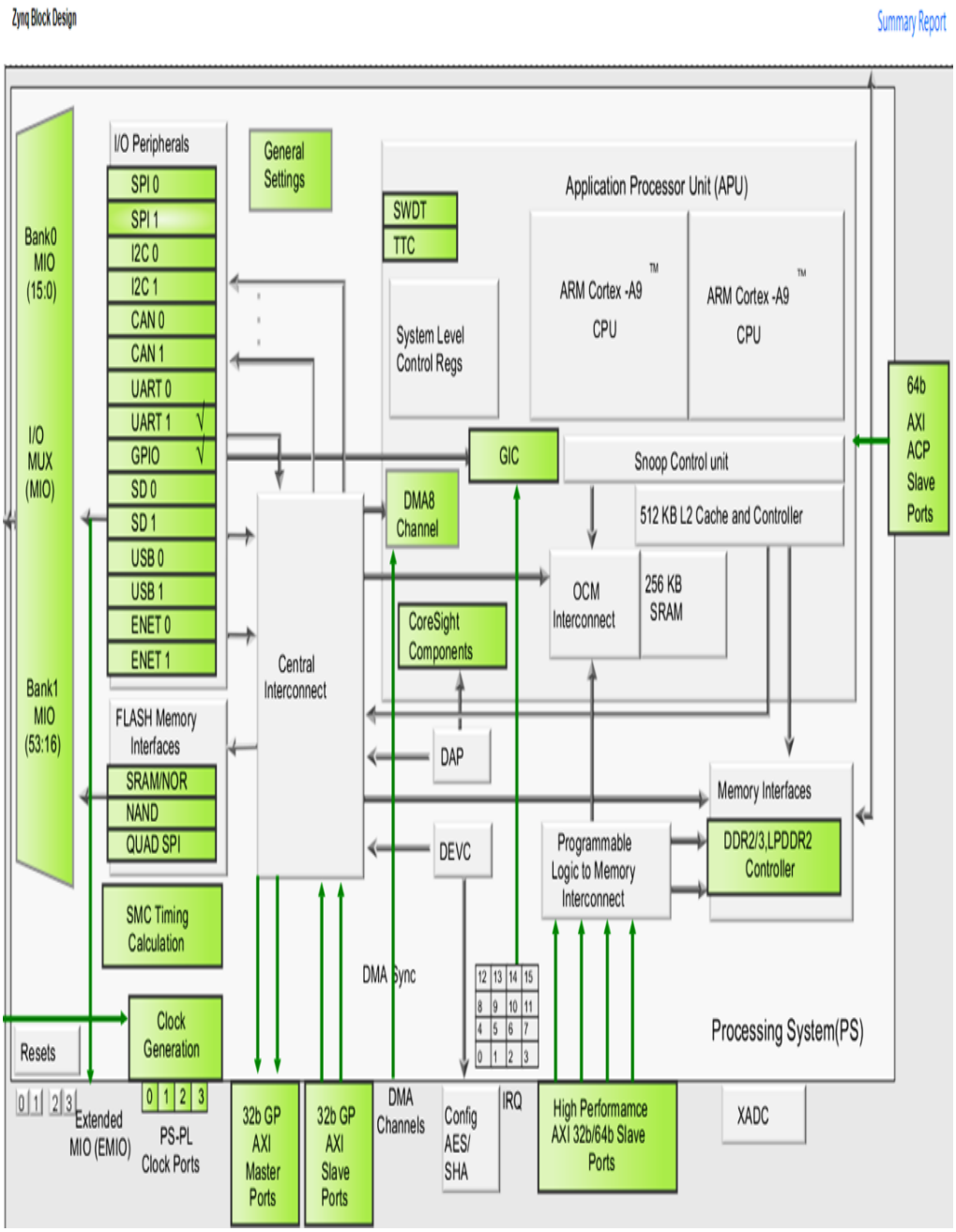


Fig.3.15 internal architecture of ZYNQ board

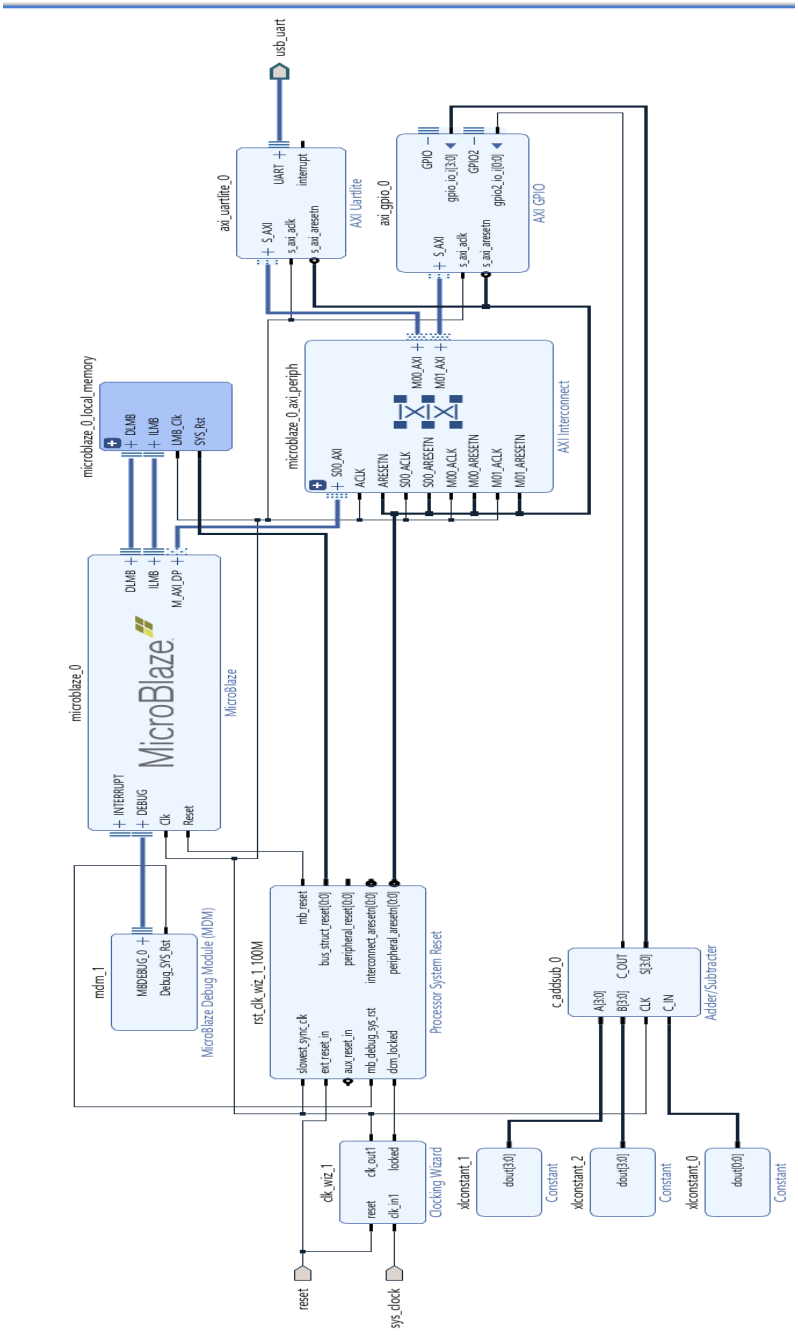


Fig.3.16 Block diagram to implement addition using adder IP

Chapter 3

Code Block

3.1 code for storage of integer in BRAM:

```
#include <stdio.h>
#include "platform.h"
#include "xparameters.h"
#include "xil_io.h"
#include "xil_printf.h"
#include "xuartlite_l.h"
#include "xuartlite.h"

#define BRAM_BASE_ADDRESS XPAR_AXI_BRAM_0_BASEADDRESS 0xc0000000 // Define BRAM base address (update based on your system)
#define BRAM_SIZE_BYTES 4096                                     // Assume BRAM size is 4096 bytes (update based on your configuration)
#define UART_BASE_ADDRESS XPAR_XUARTLITE_0_BASEADDR 0x40600000
#define BUFFER_SIZE 16  // Size of the UART receive buffer

// Function to store data in BRAM
void store_data_in_bram(u32 data, u32 address_offset) {
    Xil_Out32(0xc0000000  + address_offset, data); // Write data to BRAM
}

// Function to read data from BRAM and print via UART
void read_data_from_bram(u32 size) {
    for (u32 i = 0; i < size; i += 4) {  // Increment by 4 bytes for 32-bit words
        u32 value = Xil_In32(0xc0000000  + i); // Read data from BRAM
        xil_printf("Data at address %d: %d\n", i, value);   // Output to UART terminal
```

```
    }
}

// Main function
int main() {
    // Initialize platform
    init_platform();
    xil_printf("UART Initialized. Ready to receive data.\n");

    u8 recv_buffer[BUFFER_SIZE];  // UART receive buffer
    u32 current_address = 0;      // Current BRAM address offset (starts at 0)

    XUartLite_RecvByte(0x40600000);  // Initialize UART reception

    // Receive data over UART
    while (current_address < BRAM_SIZE_BYTES) {
        u8 byte = XUartLite_RecvByte(0x40600000);  // Read a byte from UART
        xil_printf("Received byte: %d\n", byte);

        // Store received byte in BRAM
        store_data_in_bram((u32)byte, current_address);   // Write byte to BRAM at current address
        xil_printf("Stored %d in BRAM at address %d\n", byte, current_address);

        current_address += 4;  // Move to the next address (BRAM is 32-bit aligned)
    }

    xil_printf("BRAM full. Stopping data reception.\n");

    // Read back and print the data from BRAM
    read_data_from_bram(current_address);

    // Clean up
    cleanup_platform();
    return 0;
}
```

3.2 c code to send num from FPGA to PC:

```c
#include "xparameters.h"
#include "xgpio.h"
#include "xuartlite.h"
// Define constants for AXI GPIO and UART
#define GPIO_DEVICE_ID        XPAR_GPIO_0_DEVICE_ID
#define UART_DEVICE_ID
   XPAR_UARTLITE_0_DEVICE_ID
#define GPIO_CHANNEL          1  // Channel 1 for
   LEDs
#define LED_MASK              0xFF  // Use 8 bits
   for LEDs
XGpio Gpio;             // GPIO instance
XUartLite UartLite;    // UART instance
int main(void) {
    int Status;
    u8 ReceivedData;  // Variable to store received
        data
    // Initialize AXI GPIO
    Status = XGpio_Initialize(&Gpio, GPIO_DEVICE_ID
       );
    if (Status != XST_SUCCESS) {
        xil_printf("GPIO Initialization Failed\r\n"
           );
        return XST_FAILURE;
    }
    // Set GPIO direction: output for LEDs
    XGpio_SetDataDirection(&Gpio, GPIO_CHANNEL, ~
       LED_MASK);
    // Initialize UARTLite
    Status = XUartLite_Initialize(&UartLite,
       UART_DEVICE_ID);
    if (Status != XST_SUCCESS) {
        xil_printf("UARTLite Initialization Failed\
           r\n");
        return XST_FAILURE;
    }
    xil_printf("Ready to receive data\r\n");
    while (1) {
        // Wait for a byte of data from the
           terminal
        if (XUartLite_Recv(&UartLite, &ReceivedData
           , 1) == 1) {
            xil_printf("Data received: %d\r\n",
               ReceivedData);

            // Display received data on LEDs
            XGpio_DiscreteWrite(&Gpio, GPIO_CHANNEL
               , ReceivedData & LED_MASK);
        }
    }
    return XST_SUCCESS;
}
```

3.3 C code to receive number from PC TO FPGA:

```
#include "xparameters.h"
#include "xuartlite.h"
#include "xgpio.h"

#define UARTLITE_DEVICE_ID
   XPAR_UARTLITE_0_DEVICE_ID
#define GPIO_DEVICE_ID XPAR_GPIO_0_DEVICE_ID
#define LED_CHANNEL 1

XUartLite UartLite;
XGpio Gpio;

int main() {
    XUartLite_Initialize(&UartLite,
       UARTLITE_DEVICE_ID);
    XGpio_Initialize(&Gpio, GPIO_DEVICE_ID);
    XGpio_SetDataDirection(&Gpio, LED_CHANNEL, 0
       x0000); // Set as output

    int number = 0;  // Store the full number being
        entered

    while (1) {
        uint8_t receivedByte;
        if (XUartLite_Recv(&UartLite, &receivedByte
           , 1)) {
            if (receivedByte >= '0' && receivedByte
                <= '9') {
                // Convert and shift previous
                   number left by 1 decimal place
                number = (number * 10) + (
                   receivedByte - '0');
            }
            else if (receivedByte == '\r') {
                // Display the final number on LEDs
                XGpio_DiscreteWrite(&Gpio,
                   LED_CHANNEL, number);

                // Reset number for new input
                number = 0;
            }
        }
    }
}
```

3.4 arthmetic receiver c code:

```c
#include "xparameters.h"
#include "xgpio.h"
#include "xuartlite.h"
#include "xil_types.h"

#define UART_DEVICE_ID   XPAR_UARTLITE_0_DEVICE_ID
#define GPIO_DEVICE_ID   XPAR_GPIO_0_DEVICE_ID
#define GPIO_CHANNEL_LED 1

XUartLite UartLite;
XGpio Gpio;

int main() {
    u8 received;

    XUartLite_Initialize(&UartLite, UART_DEVICE_ID);
    XGpio_Initialize(&Gpio, GPIO_DEVICE_ID);
    XGpio_SetDataDirection(&Gpio, GPIO_CHANNEL_LED, 0x0000); // All outputs for LEDs

    while (1) {
        if (XUartLite_Recv(&UartLite, &received, 1) == 1) {
            XGpio_DiscreteWrite(&Gpio, GPIO_CHANNEL_LED, received);
        }
    }

    return 0;
}
```

3.5 c code to store floating point number:

```c
#include <stdio.h>
#include "platform.h"
#include "xparameters.h"
#include "xil_io.h"
#include "xil_printf.h"
#include "xuartlite_l.h"
#include "xuartlite.h"
#include "xil_types.h"
#include <string.h>

#define BRAM_BASE_ADDRESS XPAR_AXI_BRAM_0_BASEADDRESS 0xc0000000 // Define BRAM base address (update based on your system)
#define BRAM_SIZE_BYTES 4096                    // Assume BRAM size is 4096 bytes (update based on your configuration)
#define UART_BASE_ADDRESS XPAR_XUARTLITE_0_BASEADDR 0x40600000
#define BUFFER_SIZE 16  // Size of the UART receive buffer
// Function to store data in BRAM
void store_data_in_bram(u32 data, u32 address_offset) {
    Xil_Out32(0xc0000000 + address_offset, data); // Write the 32-bit data to BRAM
}
// Function to read data from BRAM and print as floating-point values
void read_data_from_bram(u32 size) {
    for (u32 i = 0; i < size; i += 4) {  // Increment by 4 bytes for 32-bit words
        u32 value = Xil_In32(0xc0000000 + i); // Read data from BRAM
        float float_value;
        memcpy(&float_value, &value, sizeof(float)); // Convert the 32-bit integer to a floating-point value
        xil_printf("Stored floating-point number at address %d: %f\n", i, float_value);  // Output as floating-point
    }
}
// Main function
int main() {
    // Initialize platform
    init_platform();
    xil_printf("UART Initialized. Ready to receive floating-point data.\n");
    u8 recv_buffer[BUFFER_SIZE];  // UART receive buffer
    u32 current_address = 0;      // Current BRAM address offset (starts at 0)
```

```
35
36      XUartLite_RecvByte(0x40600000);  // Initialize
            UART reception
37      xil_printf("Waiting␣to␣receive␣a␣floating-point
            ␣number...\n");
38      // Receive data over UART and store it in BRAM
39      while (current_address < BRAM_SIZE_BYTES) {
40          // Receive 4 bytes (32-bit) for a floating-
                point number
41          u32 received_data = 0;
42          for (int i = 0; i < 4; i++) {
43              u8 byte = XUartLite_RecvByte(0x40600000
                    );  // Read a byte from UART
44              xil_printf("Received␣byte:␣%02x\n",
                    byte);  // Print the received byte
                    for debugging
45              received_data |= ((u32)byte << (i * 8))
                    ;  // Combine the 4 bytes into a 32-
                    bit integer
46          }
47          // Store the received 32-bit data in BRAM
48          store_data_in_bram(received_data,
                current_address);
49          xil_printf("Stored␣floating-point␣number␣in
                ␣BRAM␣at␣address␣%d\n", current_address)
                ;
50
51          current_address += 4;  // Move to the next
                address (BRAM is 32-bit aligned)
52
53          // Optionally, stop if BRAM is full
54          if (current_address >= BRAM_SIZE_BYTES) {
55              xil_printf("BRAM␣full.␣Stopping␣data␣
                    reception.\n");
56              break;
57          }
58      }
59
60      // Read back and display the stored floating-
            point data from BRAM
61      xil_printf("Reading␣back␣stored␣data␣from␣BRAM
            ...\n");
62      read_data_from_bram(current_address);
63
64      // Clean up
65      cleanup_platform();
66      return 0;
67  }
```

3.6 c code alu operation with push button

```c
#include "xparameters.h"
#include "xgpio.h"
#include "xuartlite.h"
#define UART_DEVICE_ID
   XPAR_UARTLITE_0_DEVICE_ID
#define GPIO_DEVICE_ID
   XPAR_AXI_GPIO_0_DEVICE_ID
#define ADD_BTN     0x01  // BTNU - T18
#define SUB_BTN     0x02  // BTND - B15
#define MUL_BTN     0x04  // BTNL - L15
#define DIV_BTN     0x08  // BTR - R17
XGpio gpio;
XUartLite uart;
int main() {
    u32 switches, inputA, inputB, result;
    u32 buttons, prev_buttons = 0;
    XGpio_Initialize(&gpio, GPIO_DEVICE_ID);
    XGpio_SetDataDirection(&gpio, 1, 0xFF);  //
       Channel 1 = Switches (input)
    XGpio_SetDataDirection(&gpio, 2, 0xFF);  //
       Channel 2 = Buttons (input)
    XUartLite_Initialize(&uart, UART_DEVICE_ID);
    while (1) {
        buttons = XGpio_DiscreteRead(&gpio, 2);  //
            Read push buttons
        if ((buttons & ~prev_buttons) != 0) { //
           Detect rising edge
            switches = XGpio_DiscreteRead(&gpio, 1)
               ;
            inputA = switches & 0x0F;    // Lower 4
               bits
            inputB = (switches >> 4) & 0x0F;  //
               Upper 4 bits
            if (buttons & ADD_BTN) {
                result = inputA + inputB;
            } else if (buttons & SUB_BTN) {
                result = inputA - inputB;
            } else if (buttons & MUL_BTN) {
                result = inputA * inputB;
            } else if (buttons & DIV_BTN) {
                result = (inputB != 0) ? (inputA /
                   inputB) : 0xFF;
            } else {
                result = 0;
            }
            u8 send_data = (u8)(result & 0xFF);
            XUartLite_Send(&uart, &send_data, 1);
        }
        prev_buttons = buttons;
    }
    return 0;
}
```

3.7 c code for adder IP implementation

```c
#include "xparameters.h"
#include "xgpio.h"
#include "xuartlite.h"
#include "xil_types.h"
#define UART_DEVICE_ID   XPAR_UARTLITE_0_DEVICE_ID
#define GPIO_DEVICE_ID   XPAR_GPIO_0_DEVICE_ID
#define GPIO_CHANNEL_SUM   1
#define GPIO_CHANNEL_COUT  2
XUartLite UartLite;
XGpio Gpio;
void SendHexBytePrefixed(u8 byte) {
    char hexChars[] = "0123456789ABCDEF";
    char str[4];
    str[0] = '0';
    str[1] = 'x';
    str[2] = hexChars[(byte >> 4) & 0x0F];  // High nibble
    str[3] = hexChars[byte & 0x0F];         // Low nibble
    // Send each character one by one
    for (int i = 0; i < 4; i++) {
        XUartLite_Send(&UartLite, (u8 *)&str[i], 1);
        while (XUartLite_IsSending(&UartLite));
    }
}
void SendText(const char *str) {
    while (*str) {
        XUartLite_Send(&UartLite, (u8 *)str, 1);
        while (XUartLite_IsSending(&UartLite));
        str++;
    }
}
int main() {
    XUartLite_Initialize(&UartLite, UART_DEVICE_ID);
    XGpio_Initialize(&Gpio, GPIO_DEVICE_ID);
    // Read values from GPIO
    u8 sum  = (u8)XGpio_DiscreteRead(&Gpio, GPIO_CHANNEL_SUM);
    u8 cout = (u8)XGpio_DiscreteRead(&Gpio, GPIO_CHANNEL_COUT);
    // Send results to terminal
    SendText("SUM: ");
    SendHexBytePrefixed(sum);  // Output like 0x03
    SendText("\r\n");
    SendText("COUT: ");
    SendHexBytePrefixed(cout);  // Output like 0x01
    SendText("\r\n");
    while (1);  // Infinite loop to hold
    return 0;
}
```

3.8 c code to receive the number from PC to zynq board on LED

```
1
2  #include "xgpio.h"
3  #include "xuartps.h"
4  #include "xparameters.h"
5  #define GPIO_DEVICE_ID XPAR_AXI_GPIO_0_DEVICE_ID
6  #define UART_DEVICE_ID XPAR_PS7_UART_1_DEVICE_ID
7  XGpio Gpio;  // AXI GPIO instance
8  int main() {
9      u8 received_data;
10     int status;
11     // Initialize AXI GPIO
12     status = XGpio_Initialize(&Gpio, GPIO_DEVICE_ID
           );
13     if (status != XST_SUCCESS) {
14         return XST_FAILURE;
15     }
16     // Set GPIO as Output (for LEDs)
17     XGpio_SetDataDirection(&Gpio, 1, 0x00);  // 0
           x00 = Output
18     // Infinite loop for UART data receiving
19     while (1) {
20         // Check if UART has received data
21         if (XUartPs_IsReceiveData(
               XPAR_PS7_UART_1_BASEADDR)) {
22             // Read the received data
23             received_data = XUartPs_ReadReg(
                   XPAR_PS7_UART_1_BASEADDR,
                   XUARTPS_FIFO_OFFSET);
24             // Convert received data to integer
                   from ASCII
25             if (received_data >= '0' &&
                   received_data <= '9') {
26                 received_data -= '0';  // Convert
                       ASCII to Integer
27             }
28             // Send to AXI GPIO (4-bit masking)
29             XGpio_DiscreteWrite(&Gpio, 1,
                   received_data & 0x0F);
30         }
31     }
32     return 0;
33 }
```

3.9 c code to receive data from stm 32 microcontroller:

```c
#include "xparameters.h"
#include "xuartlite.h"
#include "xgpio.h"

#define UART_DEVICE_ID XPAR_UARTLITE_0_DEVICE_ID // AXI UARTLite Instance
#define GPIO_DEVICE_ID XPAR_AXI_GPIO_0_DEVICE_ID // AXI GPIO Instance

XUartLite UartLite;  // UARTLite Instance
XGpio Gpio;          // GPIO Instance

int main() {
    int Status;
    u8 RecvByte; // Variable to store received data

    // Initialize UARTLite
    Status = XUartLite_Initialize(&UartLite, UART_DEVICE_ID);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    // Initialize GPIO for LED output
    Status = XGpio_Initialize(&Gpio, GPIO_DEVICE_ID);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    // Set GPIO direction as output
    XGpio_SetDataDirection(&Gpio, 1, 0x0000); // 16-bit output for LEDs

    while (1) {
        // Wait until a byte is received
        while (XUartLite_Recv(&UartLite, &RecvByte, 1) == 0);

        // Display received byte on LEDs
        XGpio_DiscreteWrite(&Gpio, 1, RecvByte);
    }

    return 0;
}
```

3.10 c code for inter board communication between BASYS 3 board

```c
c code for inter board communication between BASYS 3 board
#include "xuartlite.h"
#include "xparameters.h"
#include "xstatus.h"

// Define UART Device ID
#define UARTLITE_DEVICE_ID XPAR_UARTLITE_0_DEVICE_ID

XUartLite UartLite;  // UART instance

int main() {
    int Status;
    u8 SendBuffer[1] = {0xAB};  // Data buffer (single byte)

    // Initialize UARTLite
    Status = XUartLite_Initialize(&UartLite, UARTLITE_DEVICE_ID);
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;  // Return failure if initialization fails
    }

    while (1) {
        // Wait until UART is ready to send
        while (XUartLite_IsSending(&UartLite));

        // Send 1 byte (0xAA) via UART
        XUartLite_Send(&UartLite, SendBuffer, 1);

        // Add small delay to avoid continuous transmission
        for (volatile int i = 0; i < 1000000; i++);
    }
    return 0;
}
```

3.11 Pseudo c code to send data from STM32 to basys 3 board

```
#include "main.h"
#include <string.h>  // For string handling
/* Private define --*/
#define UART_TX_TIMEOUT 100  // Timeout for UART transmission
/* Private variables -*/
UART_HandleTypeDef huart1;
UART_HandleTypeDef huart2;
/* Private user code --*/
uint8_t binary_data = 0xBA; // Data to be transmitted
char message[] = "UART TX Test\r\n";  // Debug message
/* Function prototypes */
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART2_UART_Init(void);
static void MX_USART1_UART_Init(void);
int main(void)
{
  HAL_Init();  // Initialize HAL Library
  SystemClock_Config();  // Configure system clock
  MX_GPIO_Init();  // Initialize GPIO
  MX_USART2_UART_Init();  // Initialize USART2
  MX_USART1_UART_Init();  // Initialize USART1
  /* Transmit a test message to confirm UART is working */
  HAL_UART_Transmit(&huart1, (uint8_t*)message, strlen(message), UART_TX_TIMEOUT);
  /* Infinite loop */
  while (1)
  {
      /* Send 0xCC over USART1 */
      HAL_UART_Transmit(&huart1, &binary_data, 1, HAL_MAX_DELAY);
      HAL_Delay(500);  // Delay 500ms before sending again
  }
}
```

3.12 send the number from zynq to pc

```c
#include "xparameters.h"
#include "xuartps.h"

#define UART_DEVICE_ID XPAR_XUARTPS_0_DEVICE_ID

XUartPs Uart_PS;

int main() {
    XUartPs_Config *Config;
    int Status;
    u8 number = 25;  // Number to send
    char sendBuffer[10];

    // Initialize UART
    Config = XUartPs_LookupConfig(UART_DEVICE_ID);
    if (!Config) return XST_FAILURE;

    Status = XUartPs_CfgInitialize(&Uart_PS, Config
        , Config->BaseAddress);
    if (Status != XST_SUCCESS) return XST_FAILURE;

    // Set baud rate
    XUartPs_SetBaudRate(&Uart_PS, 115200);

    while (1) {
        // Convert number to string
        int len = sprintf(sendBuffer, "%d\n",
            number);  // Adds newline for
            readability

        // Send the string to PC via UART
        XUartPs_Send(&Uart_PS, (u8*)sendBuffer, len
            );

        // Simple delay loop (not accurate, for
            readability only)
        for (volatile int i = 0; i < 1000000; i++);
    }

    return 0;
}
```

3.13 c code for data send operations from zynq to pc

```c
#include <stdio.h>
#include "xparameters.h"
#include "xuartps.h"
#define UART_DEVICE_ID XPAR_XUARTPS_0_DEVICE_ID  // Update based on system
#define DELAY_COUNT 10000000  // Adjust for timing control
XUartPs Uart_PS;  // UART Instance
// Function to send a single character
void UartPs_SendChar(u8 ch) {
    XUartPs_Send(&Uart_PS, &ch, 1);
    while (XUartPs_IsSending(&Uart_PS));  // Ensure transmission is complete
}
// Simple delay function (replaces sleep)
void delay() {
    volatile int i;
    for (i = 0; i < DELAY_COUNT; i++);
}
int main() {
    XUartPs_Config *Config;
    u8 num = '0';  // Start from '0'
    // Initialize UART
    Config = XUartPs_LookupConfig(UART_DEVICE_ID);
    if (!Config) return XST_FAILURE;
    if (XUartPs_CfgInitialize(&Uart_PS, Config, Config->BaseAddress) != XST_SUCCESS)
        return XST_FAILURE;
    printf("UART Ready. Sending numbers 0-9 continuously...\n");
    // Send numbers 0-9 continuously
    while (1) {
        UartPs_SendChar(num);  // Send current number
        num++;  // Move to next number
        if (num > '9') num = '0';  // Reset to 0 after 9
        delay();  // Add delay
    }
    return 0;
}
```

Chapter 4 Results and Discussion

In this chapter the result of the project work, that has been carried out, will be discussed in detail.

4.1 Results (data sent)

This is the first result which has been obtained where some numbers are sent from fpga to pc and received on terminal.

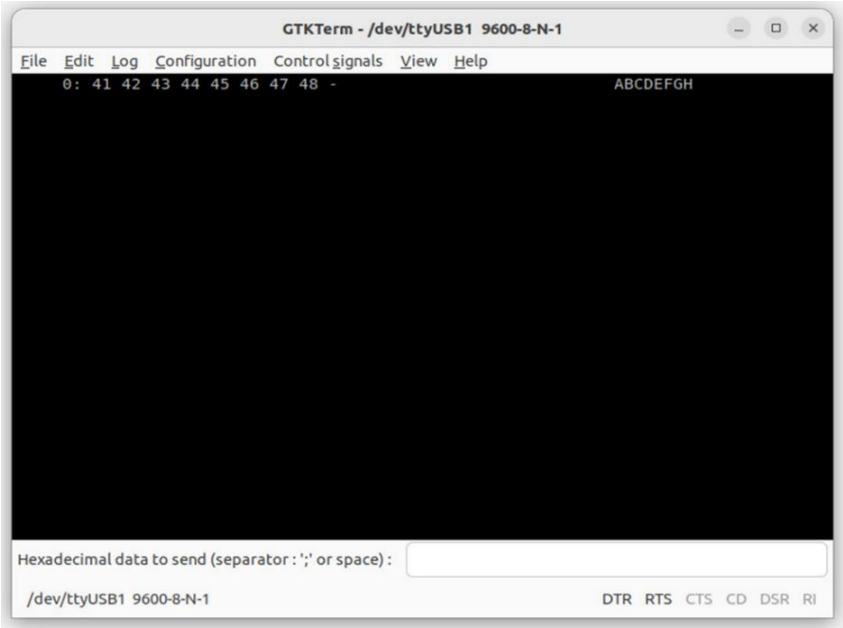


Fig.4.1 Results Displayed on GTK Term

Here number from FPGA to PC has been sent in such a way that the number and its equivalent ASCII code is printed on the GTKterm. The baud rate with which data is being sent is 9600 and Basys 3 board is used to perform this task. **The code for above result is written in the page no 63.**

4.2 Results (data received)

This time the data has been sent from external terminal to FPGA. this time, also if any number is being sent from terminal, it is received as its equivalent ASCII code. **The code for below result task is written in page no 66** .

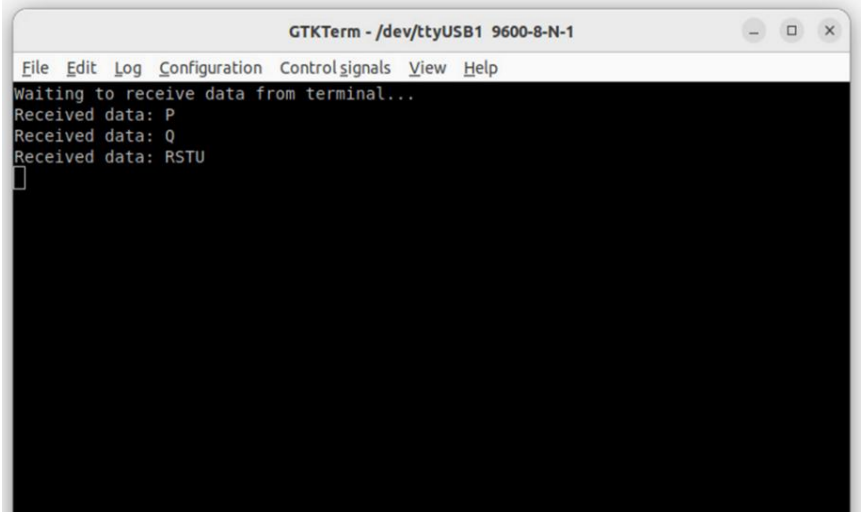


Fig.4.2 Results obtained on GTK term

Character	ASCII (decimal)	ASCII (hex)
P	80	0x50
Q	81	0x51
R	82	0x52
S	83	0x53
T	84	0x54
U	85	0x55

Table no:1(for received data)

character	ASCII (decimal)	ASCII (hex)
A, B	41,42	0x29,0x2A
C, D	43,44	0x2B,0x2C
E, F	45,46	0x2D,0x2E
G, H	47,48	0x2F,0x30

Table no:2 (for sent data) The table no. 2 and 1 show send and receive data between PC and FPGA.

```
#include<stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xparameters.h"
#include "xstatus.h"
#include "xuartlite.h"
#define UARTLITE_DEVICE_ID XPAR_AXI_UARTLITE_0_BASEADDR
int main()
{
XUartLite UartLite;
XUartLite_Initialize (&UartLite,UARTLITE_DEVICE_ID );
int num[8]={65,66,67,68,69,70,71,72};
char ascii_num[6];
for (int i=0;i<8;i++){
ascii_num[i] =(char) num [i];
}
XUartLite_Send(&UartLite,(u8 *)ascii_num,8);
return 0;
}
```

The code inside the box is responsible for the execution of data send operation from FPGA TO PC.

```
#include<stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xparameters.h"
#include "xstatus.h"
#include "xuartlite.h “
#define UARTLITE_DEVICE_ID  XPAR_AXI_UARTLITE_0_BASEADDR
#define BUFFER_SIZE 16
int while (1) {
Received Bytes = XUartLite_Recv (&UartLite, RecvBuffer, BUFFER_SIZE);
if (ReceivedBytes > 0) {
        xil_printf ("Received data: ");
        for (int i = 0; i < ReceivedBytes; i++) {
           xil_printf ("%c", RecvBuffer[i]); // Print received characters
        }
        xil_printf("\r\n");
.
```

```
        // Clear the buffer for the next set of data
        for (int i = 0; i < BUFFER_SIZE; i++) {
            RecvBuffer[i] = 0;
        }
    }
}

return 0;
}
```

The above code inside the box is responsible for receiving the data from PC to BASYS3.

4.3 Data storage in BRAM To store the data in BRAM of FPGA the arrangement is shown in **chapter 3,** where a block diagram interconnecting IPs is governed by the code written in the previous chapter **code section 3.1**. The following results have been obtained and are given below.

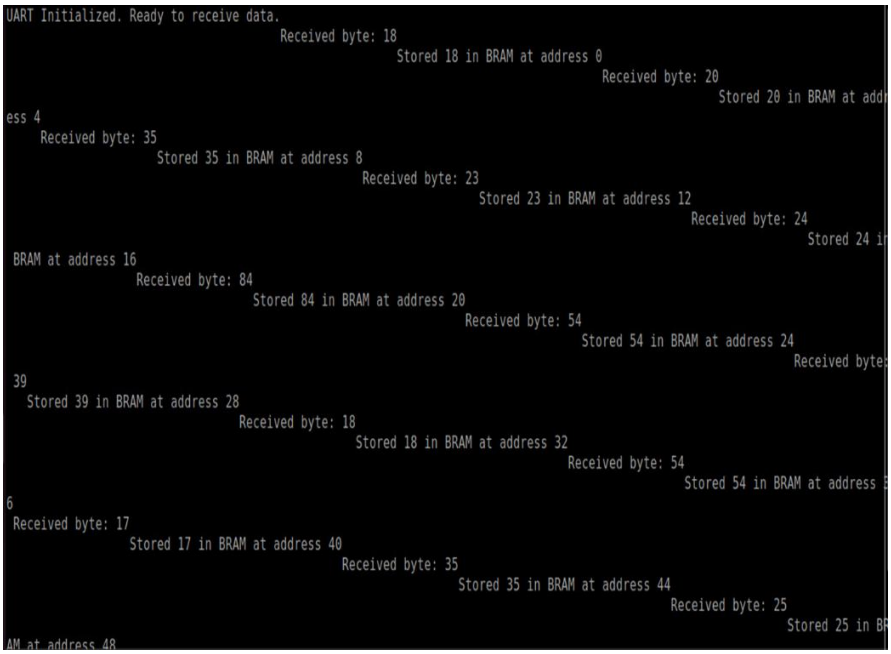


Fig.4.4 storage of integer type data

These results show the storage of floating-point data in BRAM.

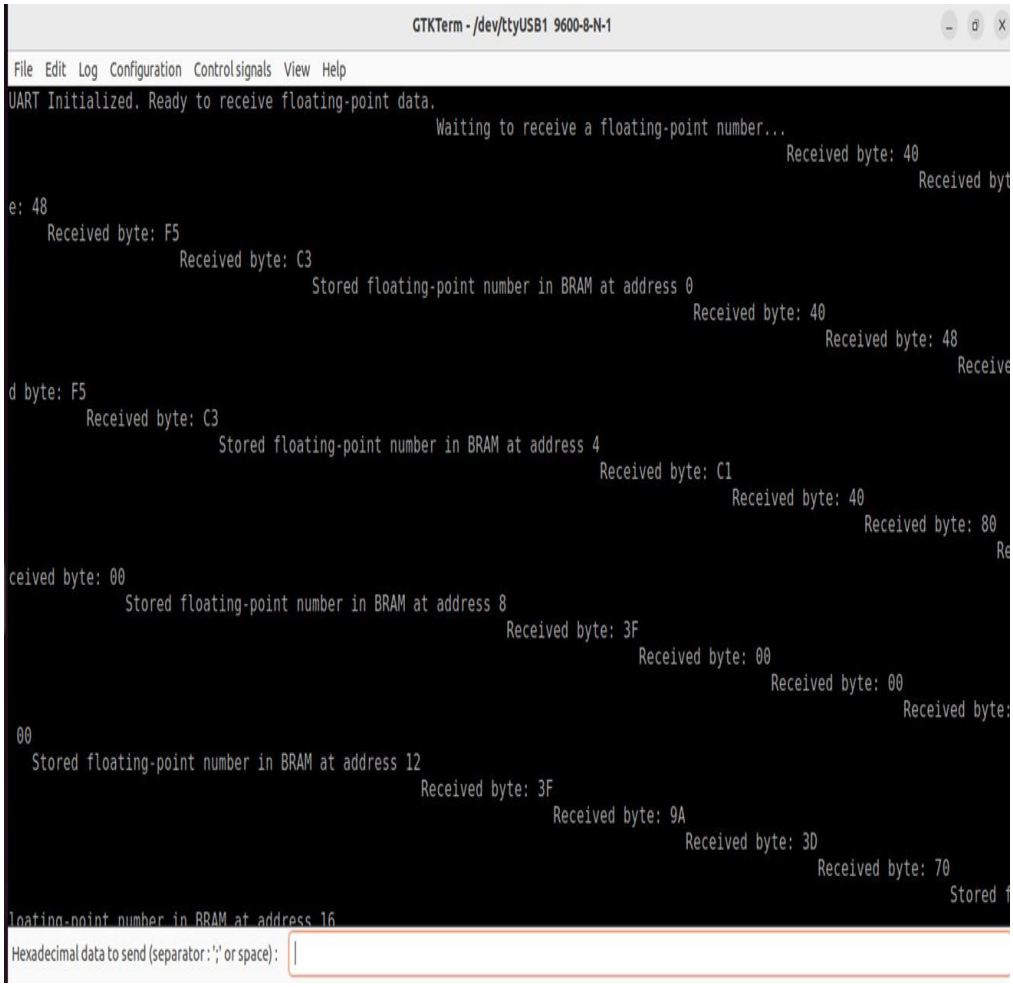


Fig.4.4 storage of floating-point data

The numbers shown in **table no:3** are number stored in the BRAM memory address and these are represented in IEEE 754 little endian format.

BRAM address	Hex value	Float value
0	0x4048F5C3	3.14
4	0x4048C1F5	3.140625
8	0x3F800000	1.0
12	0x3F9A3D70	1.21

Table no:3 for the stored values displayed on GTK term

These results show the storage of floating-point data in BRAM.

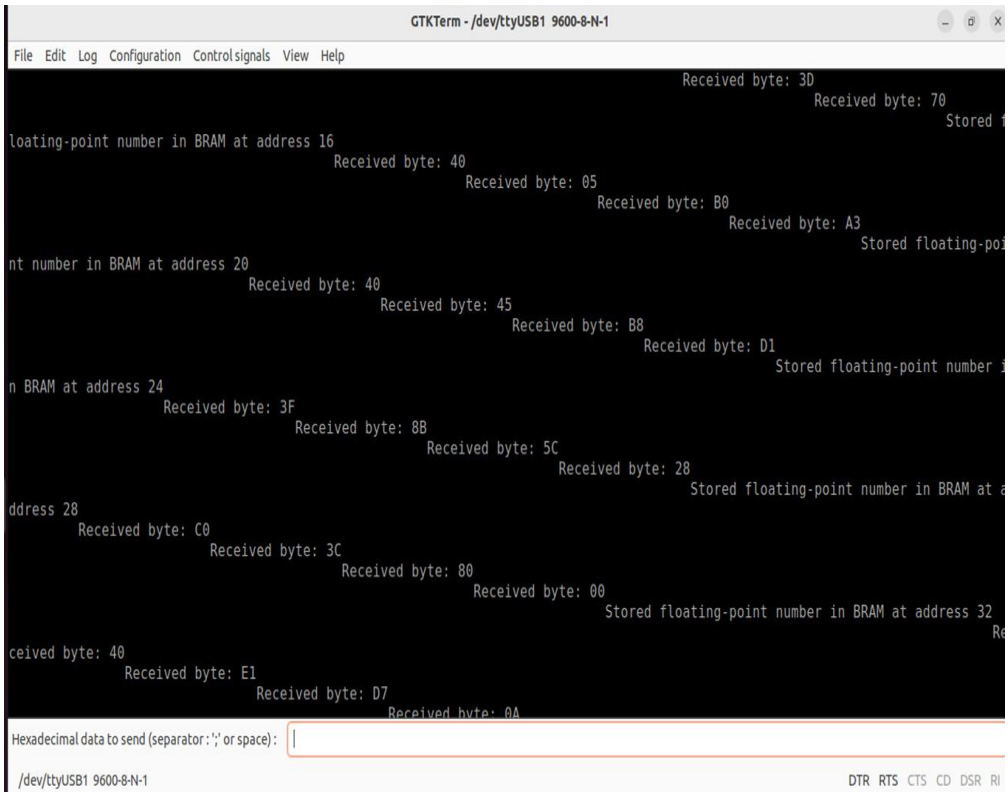


Fig.4.5 storage of floating-point data

BRAM address	Hex value	Float number
16	0x40 05 B0 A3	2.093305
20	0x40 45 B8 D1	3.0870578
24	0x3F 8B 5C 28	1.091
28	0xC0 3C 80 00	-2.94
32	0x40 E1 D7 0A	7.0689

Table no:4 for the stored values displayed on GTK term

The data that are shown in the **table no:4** are represented in IEEE 754 big endian format to correctly show the stored floating-point number in the BRAM of FPGA.

So, in this section of result discussion, we have observed clearly that how the integer as well as floating data are stored in the BRAM of FPGA.

4.4 Results for Interboard Communication

In this section the various results pertaining to all the actions that were taken for completion and verification of the project work will be discussed in details. These results will be discussed in different sections because interboard communication was performed in various ways to explore different aspects of interboard communication.

4.4.1 Data sent from FPGA (BASYS 3) Board to DSO

While doing so the intent was to know that if any data is being sent from BASYS 3(FPGA) board to any external peripheral then it is correctly received or not to that external device. Here it is also verified that received data is in sync with the data format of UART. For this DSO is utilized as a receiver to receive the data and BASYS 3 board is utilized as sender. The **necessary block diagram** for sender operation is given in the **chapter 3** through which this operation is realized.

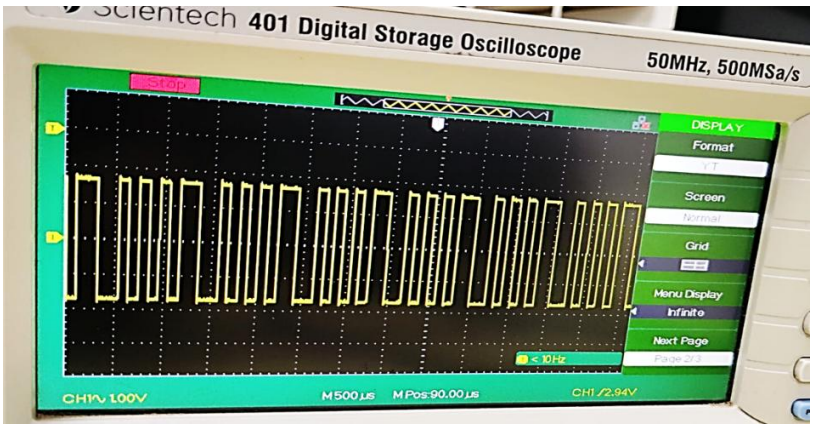


Fig.4.6 DSO as data receiver from FPGA

Data sent =(AA) (10101010), Baud rate =9600

Pmod terminal JA1 ⟶ anode terminal

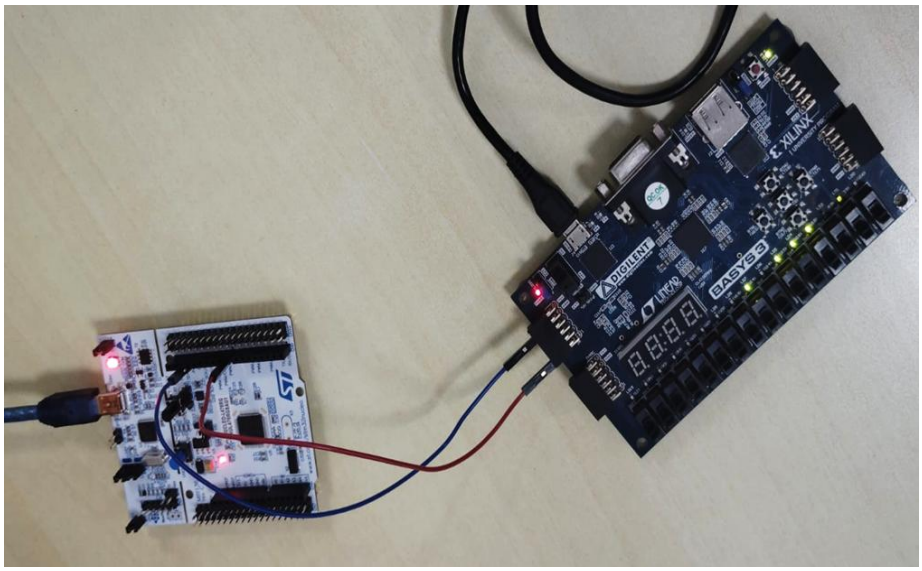
Pmod terminal JA5 ⟶ Cathode terminal (DSO)

4.4.2 Data transmission from STM 32 to BASYS 3 board

The only purpose behind doing this experiment was to ensure that the BASYS 3 board can receive the signal of strength 3.3 volts without damaging any Pmod terminal. **The block diagram for the implementation of this operation is given in Chapter 3.**

In this experiment STM 32 is used as sender and BASYS 3 board is used as receiver. The following terminal were connected for execution of this task.

- Pmod terminal of stm32 (D8) → Pmod terminal JA1
- Gnd terminal of Stm32 → Pmod terminal JA5(GND)



Data sent (BA)= (10111010)

Fig.4.7 Data transfer between STM32 and BASYS 3

The figure shows correct demonstration of data transmission from STM 32 to BASYS 3 board. Here Pmod terminal of BASYS 3 board is used as a physical connector to connect with STM 32 microcontroller via a jumper wire. The data sent from STM 32 was an 8-bit data 10111010, and it was correctly received on the LED of BASYS 3 board.

4.4.3 Data transmission from one BASYS 3 Board to other BASYS 3 board

After doing previous task successfully now we move ahead to implement interboard communication between two FPGA via UART as communication medium. Here one board behaves as sender and the other board as receiver and two 8-bit data are being sent from one board to other board for the verification of successful transmission and reception.

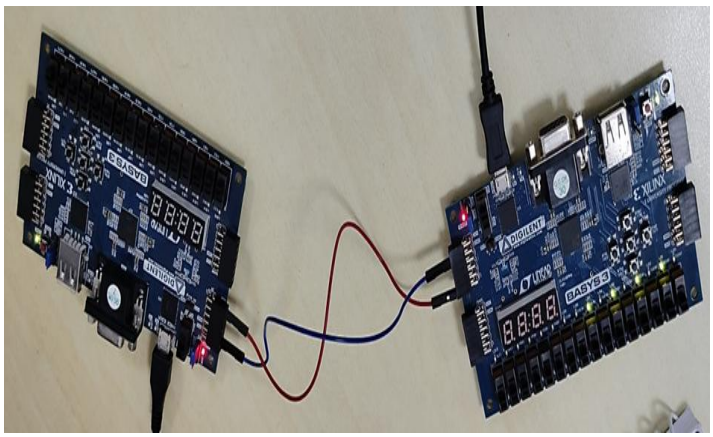


Fig.4.8 BASYS 3 as sender and receiver

Data sent = 10101010 (AA) Received successfully.

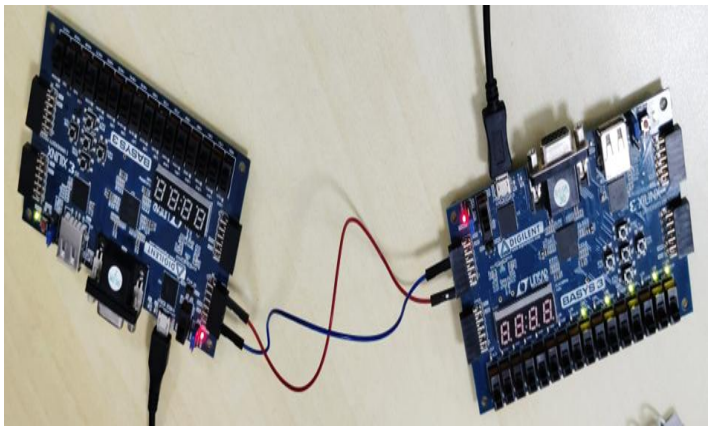


Fig.4.9 BASYS 3 as sender and receiver

Data sent =10101011(AB) received successfully

To implement the above task the block diagram is given in Chapter 3 and Fig.3.13

4.4.4 Results regarding implementation of ALU operations without IP core

Here two BASYS 3 boards are utilized and ALU operation is implemented. one board is used in such a way that it becomes platform for all the ALU operation (addition, subtraction, multiplication and division) and send all the results to other board which is working as receiver to store the result of these mathematical operations.

To implement these operations the required IP block diagram is shown in chapter 3 and Fig.3.14.

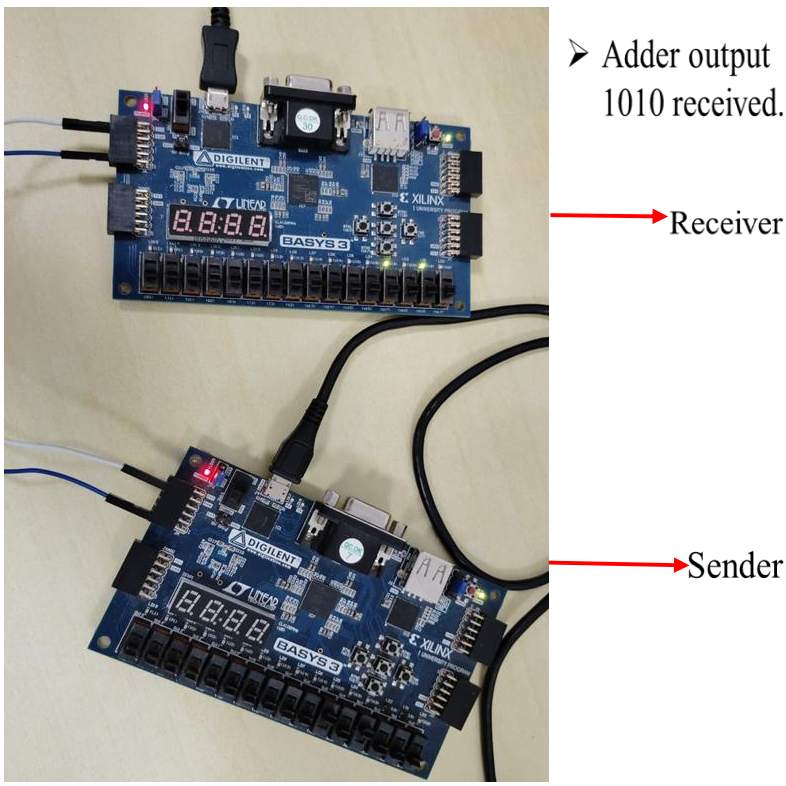


Fig.4.10 add operation on BASYS 3 board

Two binary inputs are used 1000 and 0010 and their addition result is shown above which is reflected on the receiver led. The C code to implement the above circuit is given in Chapter 3.

The given arrangement of the two basys 3 board is used to implement subtraction.

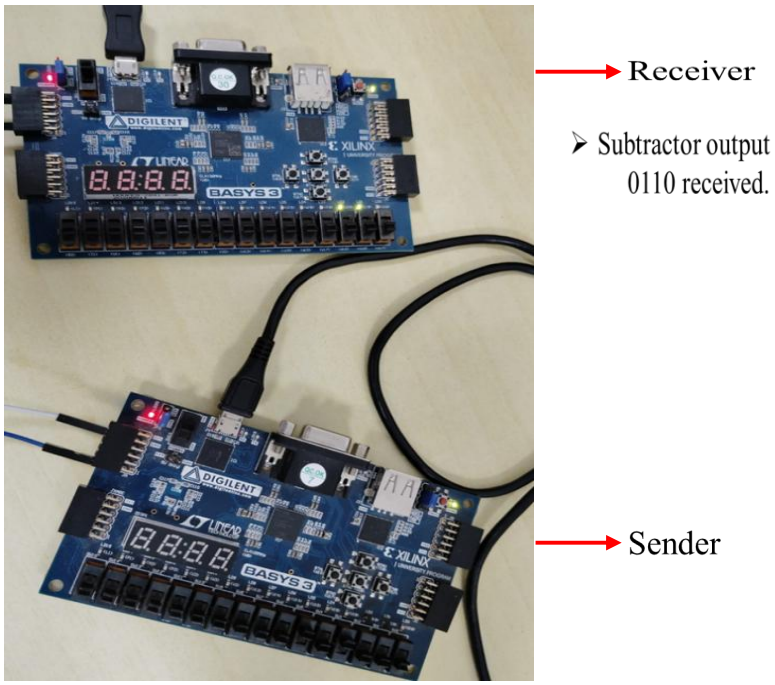


Fig.4.11 sub operation on BASYS 3 board

Two binary inputs are taken as 1010 and 0010 and these numbers when being applied at the sender board where FPGA is programmed in such a way that that it gives the subtraction operation after using the push button assigned for subtraction operation.

The block diagram arrangement for realization of this operation is given in **chapter 3** and Fig.3.14.

Now moving ahead remaining two operations will be realized. First multiplication operation is realized and in the same way division operation will be realized.

The block diagram representation is same for these operations also and mentioned in the **chapter 3** and Fig.3.14.

While performing these two operations the same data input has been taken and multiplication output is 10000 and division output is 0100.these data have been successfully obtained on the receiver board.

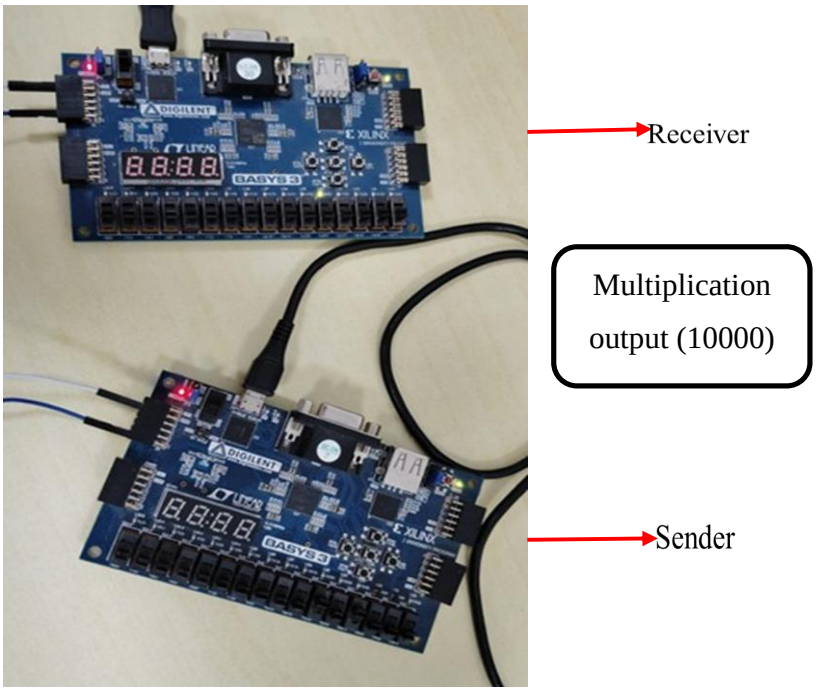


Fig.4.12 multiplication operation on BASYS 3

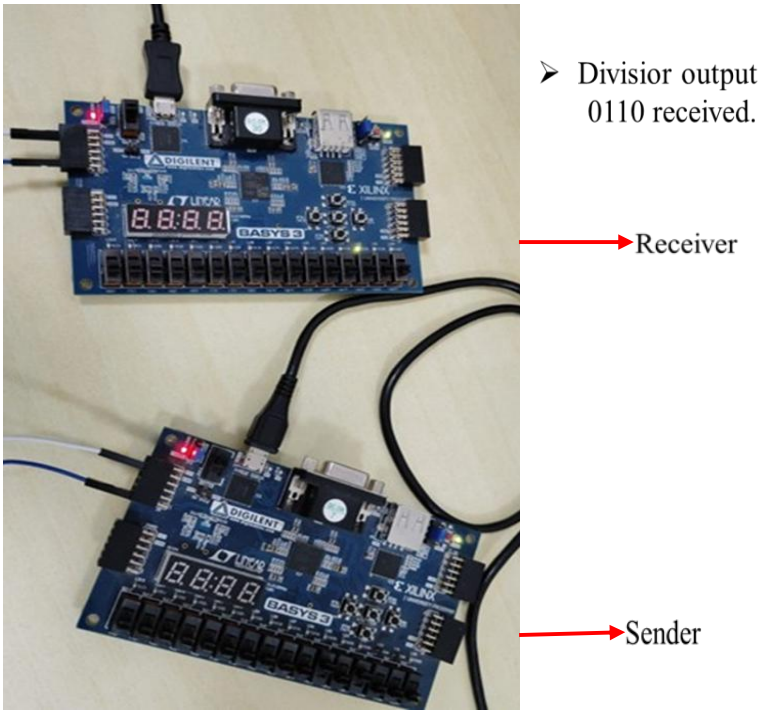


Fig.4.13 Division operation on BASYS 3 board

4.4.5 Results obtained using adder IP

In previous work it has been observed that ALU operations were realized without any core IP. In this section a specific IP (here adder IP) will be utilized to realize the addition operation. Here like previous case one board will be used as sender on which the addition operation will be performed and the result will be stored into receiver board. Two inputs **4-bit binary inputs are taken as 0111 and 0101** and a carry input is also taken. so final result will be 1101.the same result has been obtained on the receiver board correctly and displayed on the LED of receiver board.

The IP block diagram for the implementation of above function is mentioned in **chapter 3** and Fig.-----.

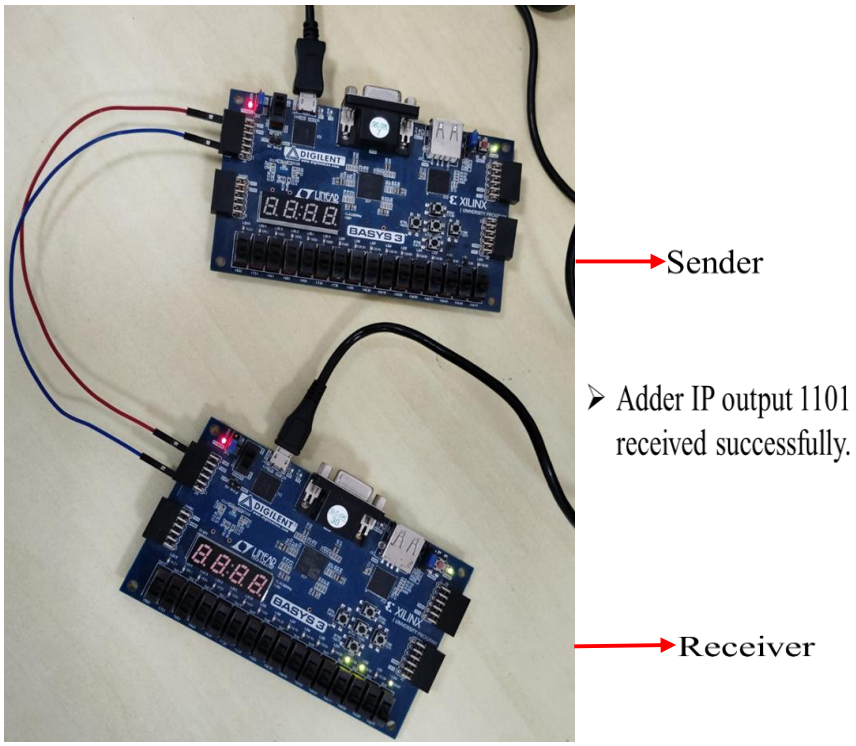


Fig.4.14 adder ip implementation on BASYS 3

This experiment shows that if we take any complex IP and implement it on one FPGA and execute all the complex mathematical operations on it then

for storing the various results other FPGA can be used so that partitioning can be applied properly for faster execution of complex task.

4.4.6 Data Received on ZYNQ Board from PC

In this experiment one ZYNQ board has been used to receive the number and display it on the LED available on ZYNQ board thus verifying the fact that upon sending the number it is correctly receiving.

The only limitation in this board is that here only four LEDs are available so we can check only upto 15.

The experimental setup and required C code is written in the Fig.----- and in the **chapter 3** respectively for the realization of the above task.

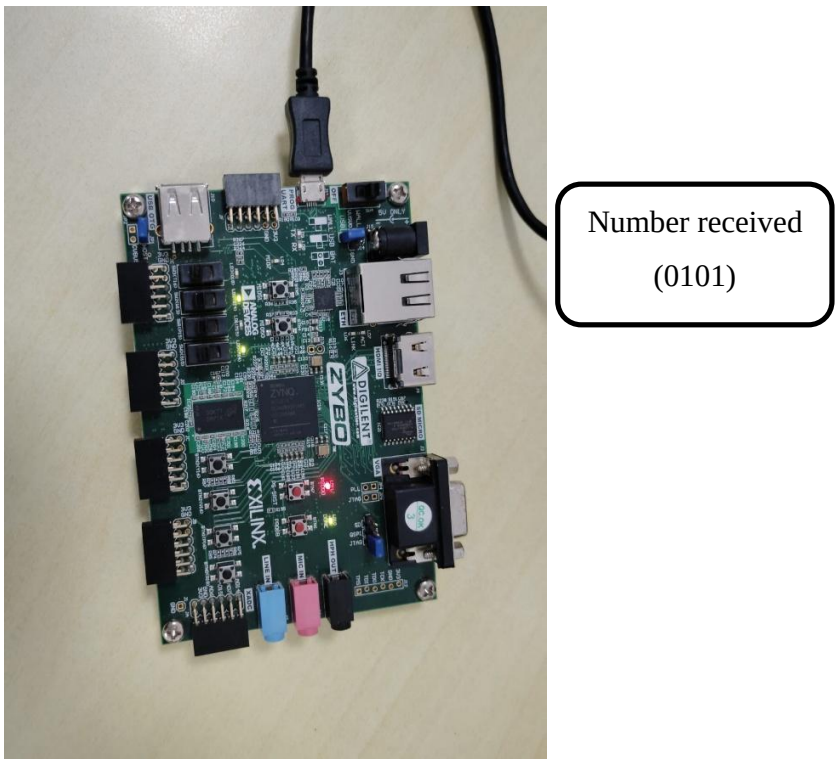


Fig.4.15 ZYNQ board as receiver

Here Baud rate is 115200, different from the BASYS 3 board.

In continuation of receiving the number from external terminal some more example is given below.

Fig.4.16 ZYNQ board as receiver

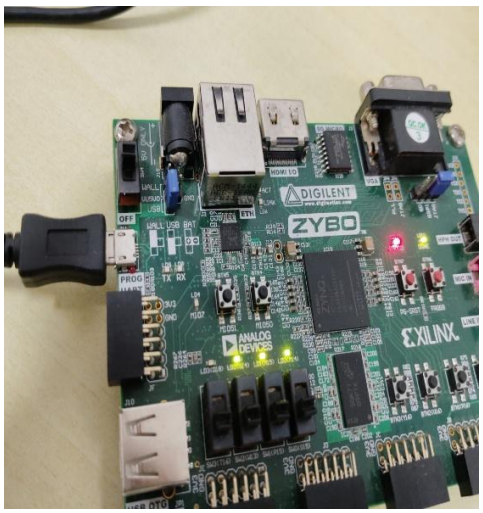


Fig.4.17 ZYNQ board as receiver

The ZYNQ PS system is used in the IP block diagram to realise this board as receiver and UART 1 is used to receive the data at 115200 baud rate.

4.4.7 ZYNQ board as sender using UART communication Protocol

Here ZYNQ board is used as sender and the FPGA inbuilt in it send the number to PC and those data are appeared on the external terminal.

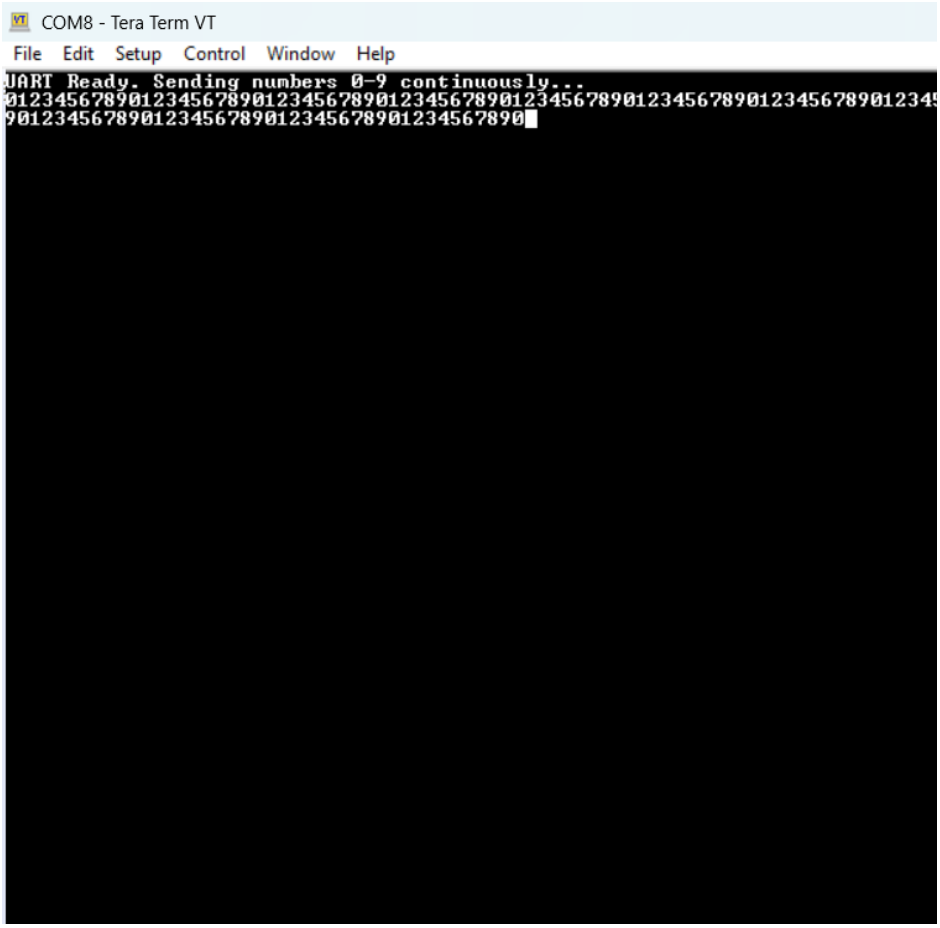


Fig.4.18 Zynq board as Sender

Here ZYNQ board is used as sender and send the sequence of data from 0 to 9 to the PC and it is received on the terminal.

Here data transmission rate is 115200.so it is faster in comparison to BASYS 3 board in terms of data transmission and reception

4.5 Summary of Results and discussion

In this chapter all the results have been presented in clear and concise form and to support those claims every proof is given so that credibility of the results be undeniable.

Starting from establishing the communication between PC and FPGA by sending and receiving the numbers to send the number from FPGA to an external peripheral like DSO basic communication protocol has been established.

Moving ahead data communication has been established between STM 32 microcontroller and BASYS 3 board (FPGA)by sending some hexadecimal numbers. in the same test it is also verified that signal strength of sent and received number is 3.3 volt and any signal more than this can damage the peripheral module of the given boards like BASYS 3 board.

As we established the data communication between two different board and successfully sent and receive the data by using one board as sender (STM 32) and the other board as receiver (BASYS 3) it has laid the ground for interboard communication.

Now we move ahead for interboard communication between two basys 3 board by using one board as transmitter and other board as receiver and the both the boards are connected via jumper wire through PMOD.

After completion of this interboard communication now the sender board has been utilised for implementation of ALU operation and the results of ALU operation were sent to the other board for displaying those results on the LED for the integrity of the received result. While implementing the mathematical operations no IP core is used here till now.

Finally moving towards the utmost important task that has imitated the MD simulation task in its simplest form. Here to implement a simple mathematical operation like addition an adder IP core is taken.it has

established the addition operation successfully and sent the results successfully to the receiver board.it was the final task which was performed in the interboard communication. If an IP that is having very complex function, then by applying the same IP complex functions can be solved which will lead to solution of larger and complex problem like MD simulation where computation is extensive and takes more time as the number of variables increase (that is complexity of system increases).

The other board which has been explored in this project work is ZYNQ board. This board is utilized to the extent that has established communication between PC and FPGA for transmission and reception of data.by externally sending the data from an external terminal like (TERA TERM) it has also been verified that the board has correctly received the data by mapping it out on the inbuilt LED available on the ZYNQ board.

One important aspect while executing the interboard communication was to store the integer and floating-point number in the BRAM of FPGA and it was also successfully completed.

Chapter 5

5.1 Future Scope of the Project Work

The work presented in this project lays a robust foundation for more complex and computationally intensive tasks in the domain of hardware-accelerated simulations. Starting from **basic UART communication** and progressing through **interboard data transmission**, **BRAM-based data storage**, and **IP core-based computation**, the system has successfully emulated essential components of a distributed processing environment. This modular and scalable architecture opens several possibilities for extending the current work into real-world applications such as Molecular Dynamics (MD) simulations. By incrementally increasing the computational logic, incorporating parallelism, and introducing complex arithmetic IP cores (e.g., for force fields or integration steps), the existing setup can evolve into a simulation engine capable of solving larger and more realistic physical systems. The proven ability to transmit data between different boards and validate outputs via on-board indicators suggests a strong potential for developing a full hardware-software co-design framework for scientific computing.

5.2 Analogy between my project and MD Simulation on FPGA

There are following analogy between my project work and MD Simulation on FPGA which will be discussed in details in the next section. In short it can be said that adder IP implementation on one board and its result storage on the board imitate to some extent the actual MD Simulation.

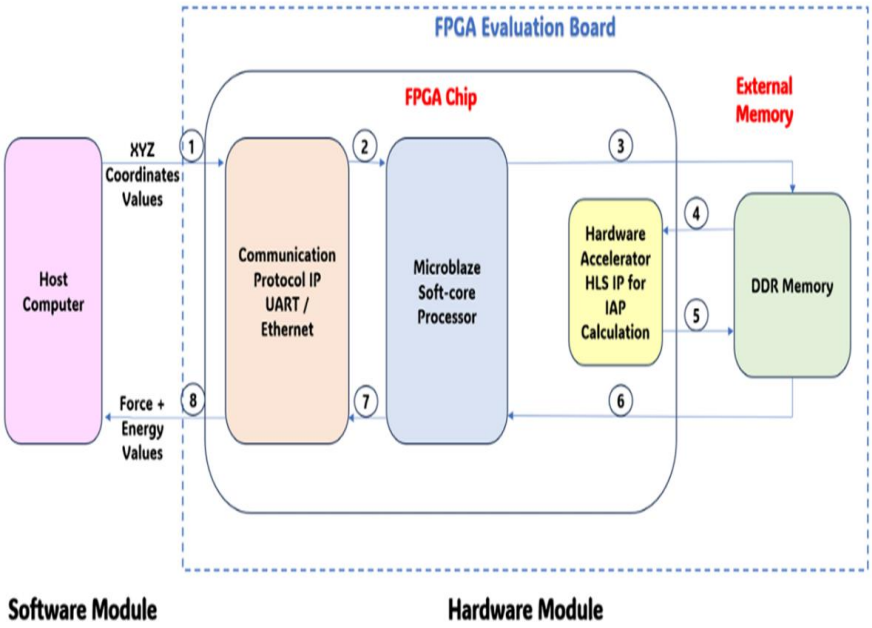


Fig.5.1 Block diagram for MD Simulation Operation

The diagram given below will give us a clear picture that how the MD Simulation will be an extension of my project work.

5.2.1 Hardware Communication and Data Exchange as Simulation Setup

In our project, the foundational step involved establishing UART communication between various systems like the STM32 microcontroller, Basys 3 FPGA board, and ZYNQ board. This process is analogous to setting up a simulation environment in MD where a host system initializes simulation parameters and distributes data to computational nodes. Sending data from STM32 to Basys 3 mirrors the act of dispatching particle positions or forces from a central controller to the simulation engine. The UART-based communication with a PC and the external terminal represents the interactive configuration of simulation parameters in real-time. The

verification of 3.3V logic levels ensures electrical integrity in your hardware system, just like verifying signal protocols in MD simulation ensures accuracy and safety in high-performance computing.

5.2.2 Memory Utilization and Operation Execution as Core Computational Engine

Our use of BRAM to store both integer and floating-point numbers resembles the way MD simulations store particle coordinates, velocities, and forces in dedicated memory structures. The implementation of basic ALU operations without any IP core simulates the behavior of computation units solving pairwise forces or time integration steps. Later, by incorporating an adder IP for mathematical operations and transmitting those results to another board, your system mimicked a specialized hardware block executing and propagating results within a simulation. This illustrates the concept of hardware acceleration in MD, where specific functions like force computation or energy calculation are offloaded to optimized hardware.

5.2.3 Interboard Communication and Visualization as Simulation Loop and Output

The interboard communication between two Basys 3 boards using PMOD jumpers signifies the exchange of simulation data between parallel processors or compute nodes. Here, one board acted as the transmitter executing mathematical functions (akin to particle interaction computations), and the other board as the receiver displayed the results on LEDs—mirroring the MD simulation loop where one module updates particle positions and another module visualizes or stores these values. Similarly, the GPIO-controlled LEDs in your system served as real-time indicators of data reception, analogous to trajectory or energy logging in MD. The FSM logic controlling the sender/receiver roles also parallels the

simulation loop that advances time steps and manages synchronization between tasks.

5.3 Possible Future Extensions

5.3.1 Integration of Complex IP Cores

Use of arithmetic-intensive IP cores (like floating-point multipliers or vector processors) to perform molecular force evaluations and potential energy calculations.

5.3.2 Parallel Computation Framework

Deploy multiple FPGAs working in parallel—each handling a subset of atoms or particles—to mimic domain decomposition techniques used in MD simulations.

5.3.3 Time-Step Based Simulation Loop

Implement a finite state machine (FSM) that handles time-step progression, similar to Verlet or Leapfrog integrators used in MD.

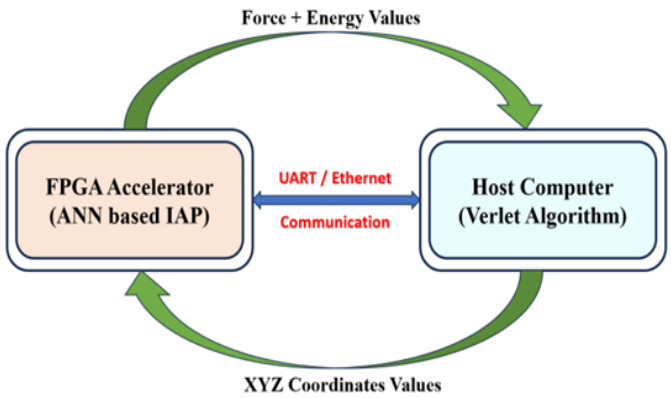


Fig.5.2 Proposed Hardware-Software Co-design Prototype for MD Simulation

5.3.4 Floating-Point Precision Optimization

Explore trade-offs between fixed-point and floating-point arithmetic for performance optimization without losing simulation accuracy.

5.3.5 On-chip Real-Time Visualization

Expand LED or seven-segment-based output to real-time position/velocity displays, or interface with a PC for graphical rendering.

5.3.6 Interfacing with External Sensors or Actuators

Integrate sensors (like temperature or pressure sensors) with the FPGA to mimic external conditions in a physical simulation setup.

5.3.7 BRAM Utilization for Simulation Buffering

Leverage BRAM as buffer storage for atomic positions, velocities, and forces across multiple time steps.

5.3.8 ZYNQ-based Full System Integration

Use the ARM processing system on the ZYNQ board to manage high-level simulation control, while offloading computationally expensive tasks to PL (programmable logic) fabric.

5.3.9 Custom Protocol Design for Data Transfer

Design and implement custom high-speed protocols for efficient inter-FPGA communication to scale the MD engine.

5.4 Conclusion

So, by analyzing these points, it can be said that the project, if employed well, can be extended to various tasks that help to increase the computing speed of the system involved in complex calculations.

References

[1] S. Thomas, N. Kalarikkal, O. S. Oluwafemi, J. Wu, and E. H. M. Sakho, "Nanomaterials for solar cell applications", Elsevier, 2019.

[2] K. Kerman, M. Saito, E. Tamiya, S. Yamamura, and Y. Takamura, “Nanomaterial-based electrochemical biosensors for medical applications”, TrAC Trends in Analytical Chemistry, vol. 27, no. 7, pp. 585–592, 2008.

[3] Q. Zhang, X. Yang, and J. Guan, “Applications of magnetic nanomaterials in heterogeneous catalysis”, ACS Applied Nano Materials, vol. 2, no. 8, pp. 4681–4697, 2019.

[4] S.A. Hollingsworth and R. O. Dror, “Molecular dynamics simulation for all”, Neuron, vol. 99, no. 6, pp. 1129–1143, 2018.

[5] A. Hospital, J. R. Goñi, M. Orozco, and J. L. Gelpí, “Molecular dynamics simulations: Advances and applications”, Advances and Applications in Bioinformatics and Chemistry, pp. 37–47, 2015

[6] M. Showerman, J.Enos, A. Pant, V. Kindratenko, C. Steffen, R. Pennington, and W. M. Hwu, "QP: A Heterogeneous Multi-Accelerator Cluster", pp.1, 2009.

[7] https://www.researchgate.net/publication/262954308Real_time_algorithm_invariant_to_natural_lighting_with_LBP_techniques_through_an_adaptive_thresholding_implement_d_in_GPU_processors/figures?lo=1

[8] Microblaze Processor Reference Guide UG984.

[9] J. G. Tong, I. D. Anderson, and M. A. Khalid, “Soft-core processors for embedded systems,” in 2006 International Conference on Microelectron ics, IEEE, 2006, pp. 170–173.

[10] Microblaze processor reference guide (UG984), Xilinx, 2021. [Online]. Available: https : //www.xilinx.com/content/dam/xilinx/ support/documents/SW manuals/xilinx2021_2/ug984-vivado microblaze-ref.pdf.

[11] Axi uart lite v2.0 product guide (PG142), Xilinx, 2017. [Online]. Avail able: https://docs.amd.com/v/u/en-US/pg142-axi-uartlite.

[12] Axi interconnect logicore ip product guide (PG059), Xilinx, 2022. [On line].Available:https://docs.amd.com/r/en-US/pg059-axi interconnect/AXI-Interconnect-v2.1-LogiCORE-IP-Product Guide.

[13] Vivado design suite user guide: Embedded processor hardware design (UG898), Xilinx, 2021. [Online]. Available: https://docs.amd.com/ v/u/2021.1-English/ug898-vivado-embedded-design.

[14] Ultrafast embedded design methodology guide (UG1046), Xilinx, 2018. [Online]. Available: https://docs.amd.com/v/u/en-US/ug1046 ultrafast-design-methodology-guide.

[15] Ultrafast embedded design methodology guide (UG1046), Xilinx, 2018.[Online].Available:https://docs.amd.com/v/u/en-US/ug1046ultrafast-design-methodology-guide.

[17] https://youtu.be/uX-BqXf34r8?si=GKnz_CdoJoJiW_0o

[18] https://youtu.be/uxKyLTMv9js?si=REuHu2Kemw4Sm4VT

[19] Zynq UltraScale+ MPSoC Data Sheet: Overview (DS891)

[20] Zynq UltraScale+ MPSoC Data Sheet: DC and AC Switching Characteristics (DS925).

[21] Zynq UltraScale+ MPSoC: Embedded Design Tutorial (UG1209).

[22] . Zynq UltraScale+ MPSoC Software Developer Guide (UG1137)