INDIAN INSTITUTE OF TECHNOLOGY, KHARAGPUR

Department of Electrical Engineering
IIT Campus, Kharagpur, West Bengal,721302

INTERNSHIP REPORT

on

"Hardware-Accelerated Convolutional Neural Networks on RISC-V Processor Integrated with FPGA"

Achutha Aswin Naick
Second Year BTech ECE
College of Engineering Trivandrum

July 2025

ACKNOWLEDGEMENT

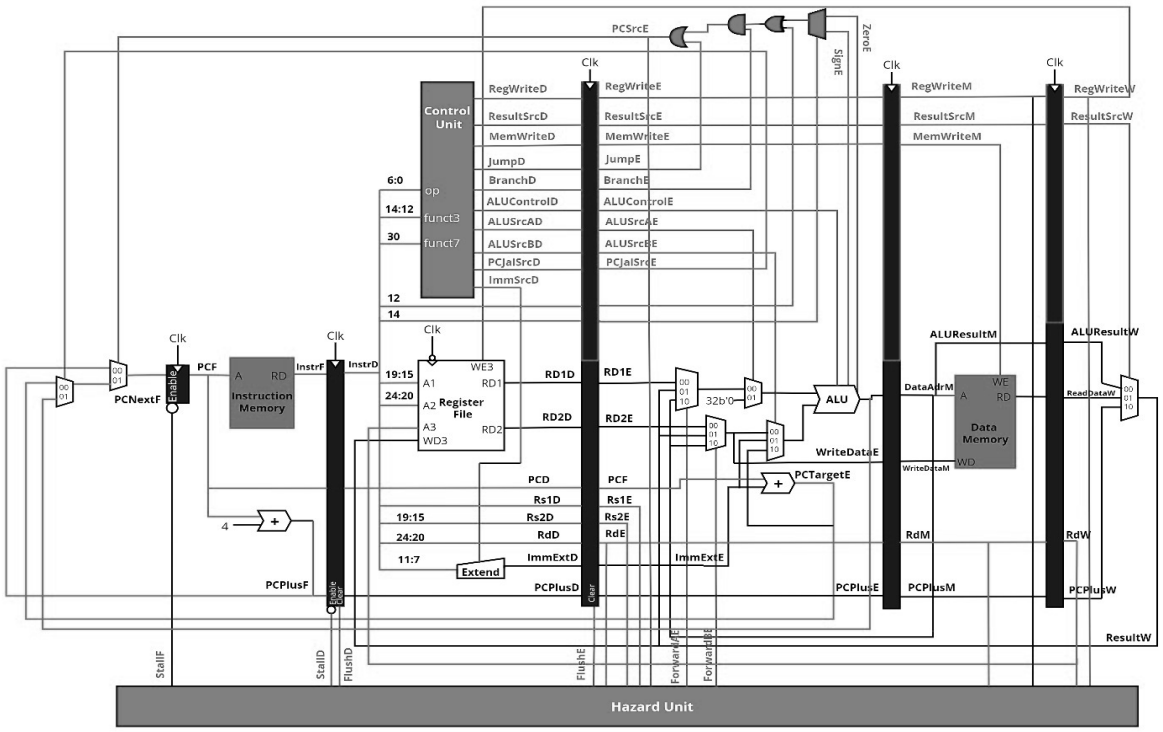
I would like to thank College of Engineering Trivandrum for allowing me to pursue an internship at Indian Institute of Technology, Kharagpur and I would also like to thank Indian Institute of Technology, Kharagpur for providing me the facilities for completing my internship successfully. I would like to thank
Prof. Aurobinda Routray for providing me this great opportunity to perform an internship under his reputed concern and I would also like to thank all the MS(Research) Scholars and PhD Scholars and the support staff under him who helped me in making this internship successful.

CONTENTS

Section I
Design and Implementation of 32-bit RISC-V Processor (RV32I) in Verilog HDL

Abstract

This section presents the design, development, and integration of a fully functional 32-bit RISC-V processor (RV32I subset) using Verilog HDL. The processor is implemented in a pipelined architecture with support for AXI4-Lite interfacing, making it suitable for embedded AI applications such as driving a neural network accelerator (Zynet). This document details the processor architecture, pipelined stages, forwarding and hazard detection mechanisms, integration with a neural network IP, and system-level implementation in AMD Vivado 2023.2 on the ZedBoard FPGA.



32-bit RISC-V Processor: Complete Diagram

LIST OF ABBREVIATIONS

adder……...	32-bit Adder
alu……...	Arithmetic Logic Unit
alu_decoder……	Arithmetic Logic Unit Decoder
controller……	Control Unit
data_memory…….	Data Memory Unit
datapath……...…...	Datapath
ex_mem_reg……...	Data Pipeline Register between the Execute (EX) and Memory Access (MEM) stages.
ex_mem_reg_ctrl……...	Control Pipeline Register between the Execute (EX) and Memory Access (MEM) stages.
forward_Amux……...	Forwarding Unit A
forward_Bmux……...	Forwarding Unit B
hazard_detection_unit……	Hazard Detection Unit
id_ex_reg……...	Data Pipeline Register between the Instruction Decode (ID) and Execute (EX) stages.
id_ex_reg_ctrl……	Control Pipeline Register between the Instruction Decode (ID) and Execute (EX) stages.

if_id_reg…….	Data Pipeline Register between the Instruction Fetch (IF) and Instruction Decode (ID) stages.
immediate_extend……..	Immediate Extend
instruction_memory……...	Instruction Memory
main_decoder…….	Main Decoder
mem_wb_reg……..	Data Pipeline Register between Memory Access (MEM) and Write Back (WB) stages.
mem_wb_reg_ctrl……...	Control Pipeline Register between Memory Access (MEM) and Write Back (WB) stages.
mux_2_1…….	2 x 1 Multiplexer
mux_3_1…….	3 x 1 Multiplexer
mux_4_1…….	4 x 1 Multiplexer
pc_mux……...	Program Counter Multplexer
reg_file……...	Register File
reset_ff……...	Reset Flip-Flop
riscv_integrated…….	RISC-V Integrated Module
riscv_top……	RISC-V top Module
wb_mux…….	Write Back Multiplexer

Introduction to RISC-V

RISC-V is an open-source Instruction Set Architecture (ISA) known for its simplicity and modularity. In this project, a 5-stage pipelined RV32I processor is implemented in Verilog HDL, enhanced to interface with an AI neural network IP via AXI4-Lite for smart embedded applications. The processor is designed from scratch with support for forwarding, hazard detection, and memory-mapped peripherals.

Features of RISC-V:

- **Simple Instruction Set**: By incorporating fewer and simpler instructions, the processor's control unit becomes less complex, enabling quicker decoding and execution.
- **Single-Cycle Execution Time**: Most instructions are designed to execute in a single clock cycle, ensuring predictable behaviour and facilitating straightforward timing analysis and pipeline integration.
- **Load/Store Design**: Memory is accessed only through dedicated instructions like load and store, while all other operations occur between registers. This reduces dependency on slower memory accesses and improves overall execution efficiency.
- **Efficient Pipelining**: RISC is inherently suited for pipelined execution, where instructions are processed in parallel through sequential stages (such as fetch, decode, execute, memory, and writeback). This overlap in processing boosts throughput significantly.
- **Enough Register Availability**: RISC processors commonly include a larger set of general-purpose registers, minimizing the need for frequent memory access and enabling faster computation through efficient data storage and retrieval.

Software Used: AMD Xilinx Vivado 2023.2

Vivado is an FPGA design and simulation tool by Xilinx, widely used for implementing hardware designs in Verilog.

For RISC-V processor implementation, Vivado provides features like RTL synthesis, timing analysis, IP integration, and simulation, enabling verification and deployment on FPGA boards such as the Basys3, Zynq 7000 ZedBoard etc.

In the project, Vivado will be the main software used to design and simulation the RISC-V processor. Various module such as controller, datapath, hazard unit, pipeline registers, instruction memory, adder, register, data memory, alu and alu decoder etc will be coded in Vivado using Verilog. The functionality for each element will also be tested and verified by simulation using Vivado. Finally, each element will be integrated in a main module to form the top design of a 32-bit 5-stage pipeline RISC-V processor.

A1. ALU (Arithmetic Logic Unit) *(alu.v)*

Arithmetic Logic Unit (alu) is a fundamental component of a CPU. It is responsible for performing arithmetic and logical operations on binary data. It reads the data from pipeline register id_ex_reg as an input and perform various arithmetic operation based on the signals from alu_decoder. The output is stored as ALUResult. In this RISC-V processor design, 10 instructions are implemented in the ALU

The 10 instructions are:
1. Addition
2. Subtraction
3. AND operation
4. OR operation
5. SLL/SLLI (shift left logical/shift left logical immediate)
6. SLT/SLTI (set less than/set less than immediate)
7. XOR
8. SRL
9. SLTU/SLTIU
10. SRA

A2. ALU Decoder *(alu_decoder.v)*

Arithmetic Logic Unit Decoder (alu_decoder) is a unit inside the control unit and is used to decode the instructions. It received the signal from the Main Decoder Unit (main_decoder) and determine the type of operation that had to be performed by the alu.

A3. Main Decoder *(main_decoder.v)*

Main Decoder (main_decoder) is a unit present inside the control unit and is used to generate the control signals from the 7 bits opcode (instruction[6:0]) to determine the types of instruction. The control signals are RegWrite, ImmSrc, ALUSrcA, ALUSrcB, MemWrite, ResultSrc, Branch, ALUOp, and Jump. Each of these control signals control the multiplexer for making decisions in the datapath to allow the data flow accordingly to the instructions.

A4. Data Memory *(data_memory.v)*

In computer architecture, data memory is a component of a computer system that is responsible for storing and retrieving data. Data memory is typically used to store data that is actively being processed by alu. Usually there are 3 inputs in this module, which are write_enable, data_address, and write_data. The module takes the memory address from the results of alu (data_address) and data from register file (write_data). Write enable (write_enable) is used to control the write permission of the data to the data memory.

A5. Instruction Memory *(instruction_memory.v)*

In computer architecture, instruction memory is a component of a computer system that stores the instruction of a program. It is responsible for holding the sequence of instruction that the CPU fetches, decodes, and executes during the program execution. In this module, 32-bit instruction set is generated and stored in the RAM array. The instruction to be fetch is based on the program counter fetch (PCF) input. As each instruction is 4 bytes, the value of PCF will be incremented by 4 to fetch the next instruction.

A6. Immediate Extend *(immediate_extend.v)*

The immediate_extend module plays a crucial role in a RISC-V processor by generating properly sign-extended immediate values based on instruction encoding formats. Different RISC-V instruction types—such as I-type, S-type, B-type, and J-type—encode their immediate fields differently across various bit positions so that it can be used in arithmetic, memory, and control operations. This module takes the relevant bits from the instruction (instr[31:7]) and a 3-bit imm_src control signal that specifies the instruction type.

PIPELINE REGISTERS

In computer architecture, pipeline registers are a temporary storage element used in a processor's pipeline to hold data between different stages of instruction execution process. It serves as a synchronization point between adjacent stages, allowing instruction to flow through the pipeline in a controlled manner. In the five-stage pipelined RISC-V processor, four primary pipeline registers—if_id_reg, id_ex_reg, ex_mem_reg, and mem_wb_reg—are implemented to handle the sequential flow of data through the datapath. To maintain clarity and modularity in signal management, control signals are handled separately using dedicated controller pipeline registers: id_ex_reg_ctrl, ex_mem_reg_ctrl, and mem_wb_reg_ctrl. This separation facilitates better organization, simplifies debugging, and enhances the scalability of the design. The registers are named for the two stages separated by that register. For example, the first pipeline register is if_id_reg because it separates the instruction fetch and instruction decode stages.

A7. Instruction Fetch / Instruction Decode Register (datapath) *(if_id_reg.v)*

if_id_reg register as it names called, it separates the instruction fetch and instruction decode stages. It used to store data such as instruction fetch from instruction memory and ready to be released to decode stage on the next clock cycle. Besides, the current PC and next incremented PC address (PCplus4F) is also saved in the if_id_reg register in case it needed later for an instruction, such as beq.

A8. Instruction Decode / Instruction Execute Register (datapath) *(id_ex_reg.v)*

id_ex_reg register as it names called, it separates the instruction decode and execute stages. It used to store information such as read data (RD1_D, RD2_D) from the register file and extended immediate value (immediate_extend_D). Besides, it carries forward the data of PC and PCplus4F from if_id_reg pipeline register. Instruction[11:7] (rd_D), Instruction[19:15] (rs1_D), and Instruction[24:20] (rs2_D) will also be stored to id_ex_reg register and send to Hazard Unit in execute stage for hazard handling.

A9. Instruction Execute / Memory Access Register (datapath) *(ex_mem_reg.v)*

ex_mem register as it names called, it separates the execute and memory stages. It is used to store the ALU results (ALUResult) and write data (write_data). At the same time, Instruction[11:7] (rd) and PCplus4F are also carried forward from previous pipeline registers and stored in ex_mem register.

A10. Memory Access / Write Back Register (datapath) *(mem_wb_reg.v)*

mem_wb_reg register as it names called, it separates the memory and writeback stages. It used to store ALU results (ALUResult) and read data (read_data) from data memory. Instruction[11:7] (rd) and PCplus4F are also carried forward from previous pipeline registers and store in mem_wb register.

A11. Instruction Decode / Instruction Execute Register (controller)*(id_ex_reg_ctrl.v)*

The id_ex_reg_ctrl module is a pipeline control register that separates the decode stage from the execute stage in a pipelined RISC-V processor.
id_ex_reg_ctrl specifically handles the control signals generated during the decode stage, and passes them forward into the execute stage.

A12. Instruction Execute /Memory Access Register (controller)*(ex_mem_reg_ctrl.v)*

The ex_mem_reg_ctrl module is a pipeline control register that separates the execute stage from the memory stage in a pipelined RISC-V processor.
ex_mem_reg_ctrl specifically handles the control signals generated during the execute stage, and passes them forward into the memory stage.

A13. Memory Access / Write Back Register (controller)*(mem_wb_reg_ctrl.v)*

The mem_wb_reg_ctrl module is a pipeline control register that separates the memory stage from the writeback stage in a pipelined RISC-V processor.

mem_wb_reg_ctrl specifically handles the control signals generated during the memory stage, and passes them forward into the writeback stage.

A14. Write Back MUX *(wb_mux.v)*

The ALU has the capability to carry out arithmetic operations like addition (A+B) or logical operation like equality comparison (A=B). Depending on the specific instructions being executed, the output of the ALU could be either a memory address or the result obtained from the ALU operation. To handle this situation, a MUX is needed to make decision between selecting the data address or the ALU output value for writing back to the register file. The MUX acts as a switch that selects one of the two inputs based on ResultSrc control signal. This allows flexibility and efficient data handling in the processor's pipeline.

A15. Program Counter MUX *(pc_mux.v)*

The Program Counter (PC) is a vital component used by the CPU to maintain the current instruction being executed. Under normal circumstances, the program counter increments by a fixed value, typically 4 (corresponding to a 32-bit instruction) for each clock cycle. This ensures that the program counter always points to the memory address of the next instruction to be executed. However, the program counter can be interrupted or modified by a jump signal (jump) from the control unit. When the predetermined conditions are met, the control unit instructs the program counter to deviate from its regular incrementation and instead updated its value to the jump address. Therefore, pc_mux is used to select the incremented instruction address (PCplus4F) or the jump address (JumpTargetE). The pc_mux is controlled by the PCSrcE signal. If PCSrcE signal is high, pc_mux will JumpTargetE on next clock cycle, else PCplus4F will be selected.

A16. Register File *(reg_file.v)*

The register file (reg_file) in a CPU is a crucial component responsible for temporarily storing and providing operands during program execution. It functions as a high-speed memory block containing a set of general-purpose registers, each capable of holding a fixed-width data word (typically 32 bits). The register file typically interfaces with four key inputs: RegWrite, rs1, rs2, and write_data. The RegWrite control signal enables or disables write operations. When asserted, the value in write_data is written to the register specified by rd_w. The rs1 and rs2 fields, derived from Instruction[19:15] and Instruction[24:20] in the IF/ID pipeline register, specify the source registers whose values are read and forwarded to the ID/EX pipeline register for ALU operations in the Execute stage. This structure ensures seamless data access and modification within the pipelined architecture.

A17. 3x1 Multiplexer *(mux_3_1.v)*

A 3x1 multiplexer is a fundamental combinational circuit that selects one of four input signals and forwards it to the output based on a single select line. Here, we considered the 4^{th} input signal to be 0. It acts like a digitally controlled switch.

A18. 2x1 Multiplexer *(mux_2_1.v)*

A 2x1 multiplexer is a fundamental combinational circuit that selects one of two input signals and forwards it to the output based on a single select line.

A19. Reset Flip-Flop *(reset_ff.v)*

The reset_ff module is a 32-bit register with an asynchronous reset and a stall control input. It is typically used in pipelined processor designs to store and propagate data between pipeline stages while supporting pipeline control mechanisms like stalling and flushing.

A20. Hazard Detection Unit *(hazard_detection_unit.v)*

The Hazard Unit is an essential part of a pipelined CPU that identifies and manages situations where instruction execution could go wrong due to data dependencies or conflicts. These scenarios, known as hazards, occur when instructions rely on data that hasn't yet been fully processed or written back. To maintain correct program execution and avoid unnecessary delays, the hazard unit detects these dependencies and applies solutions like stalling or forwarding. Instead of waiting for the Write Back (WB) stage to update the register file, the CPU retrieves the needed data directly from intermediate pipeline registers. This approach ensures that subsequent instructions receive their operands in time, preserving the efficiency and accuracy of the instruction pipeline.

There are 2 solutions to handle hazards namely forwarding and stalling.

A. *Forwarding Unit*

Forwarding is a technique used by hazard unit to handle data dependencies between instruction and mitigate hazards. It allows the CPU to forward data from the producer instruction to the consumer instruction, bypassing intermediate pipeline stages and avoid the need for stalls or bubbles.

Hazard conditions can be handled by 2 forwarding control signals namely forward_Amux and forward_Bmux. forward_Amux is used to control Forward MuxA and forward_Bmux is used to control Forward MuxB.

A20.A.1 Forward Multipliexer A*(forward_Amux.v)*

Forward Multiplexer A takes the input of RD1_E, ALUResult_M, and write_back_result. RD1_E is referred as register value. ALUResult_M is referred to the alu result in memory stage whereas write_back_result is referred to the results in writeback stage. Both ALUResult_M and write_back_result are forwarded values. ForwardA control signal is used to select either the register file value or the forwarded values.

A20.A.2 Forward Multipliexer B*(forward_Bmux.v)*

Forward Multiplexer A takes the input of RD2_E, ALUResult_M, and write_back_result. RD2_E is referred as register value. ALUResult_M is referred to the alu result in memory stage whereas write_back_result is referred to the results in writeback stage. Both ALUResult_M and write_back_result are forwarded values. ForwardB control signal is used to select either the register file value or the forwarded values.

INTEGRATING OUR RISC-V PROCESSOR

The RISC-V processor was developed by integrating all the individually verified modules into a cohesive design. The integration focused on three main units: the controller unit, the hazard detection unit, and the datapath unit.

- The **controller unit** generates the necessary control signals based on the instruction currently being executed.
- The **hazard unit** identifies potential data or control hazards during execution and takes appropriate steps to handle them, ensuring smooth pipeline operation.
- The **datapath unit** facilitates the flow of data across all five pipeline stages, enabling efficient instruction processing.

B.1 Controller Unit *(controller.v)*

The controller unit is responsible for generating control signals that drive the operation of the RISC-V processor's datapath. Based on the instruction's opcode, funct3, and funct7 fields, it decodes the instruction type and determines how data flows through the pipeline.

It internally connects a main decoder, ALU decoder, and pipeline control registers (id_ex_reg_ctrl, ex_mem_reg_ctrl, mem_wb_reg_ctrl) to produce stage-specific control signals. Additionally, it handles branch decision logic, including special conditions for bne, blt, and bge using the ALU's Zero and Sign outputs.
Through effective instruction decoding, hazard flushing, and PC control signal generation, the controller ensures correct execution across all five pipeline stages.

B.2 Datapath *(datapath.v)*

The datapath module forms the core of the pipelined RISC-V processor, facilitating the sequential flow and execution of instructions across all five stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB). It orchestrates the movement of instructions and data through pipeline registers (if_id_reg, id_ex_reg, ex_mem_reg, mem_wb_reg) while handling ALU operations, register file interactions, forwarding, and hazard control. This module connects major components like the ALU, multiplexers, register file, immediate extender, and memory interfaces, ensuring proper data computation and control signal propagation. It is designed to be stall- and flush-aware to accommodate hazard resolution and maintain instruction correctness in the pipelined architecture.

B.3 Hazard Detection Unit main *(hazard_detection_unit.v)*

The hazard_detection_unit module plays a crucial role in maintaining correct execution semantics in a pipelined RISC-V processor by identifying and resolving data hazards and control hazards. This unit ensures that data dependencies between successive instructions do not lead to incorrect computations or unintended behaviour by stalling, flushing, or forwarding data appropriately.
Specifically, the HDU monitors register dependencies between instructions at different stages of the pipeline—Decode (ID), Execute (EX), Memory (MEM), and Write Back (WB)—and applies logic to determine when to stall or flush parts of the pipeline, or to enable data forwarding from later stages to earlier ones.

B.3 RISC-V Top *(riscv_top.v)*

The riscv_top module is the central integration unit of the pipelined 32-bit RISC-V (RV32I) processor, responsible for orchestrating the interaction between the processor core, instruction and data memory subsystems, hazard management, and an AXI4-Lite master interface for memory-mapped peripheral communication. It acts as a platform wrapper, connecting and coordinating all major functional units, and is structured to support both standard RAM-based memory access and external communication via the AXI-Lite protocol.

This top-level design encapsulates the complete datapath, controller, hazard detection logic, instruction memory, data memory, and a custom-designed AXI4-Lite bridge, enabling flexible, efficient, and extendable RISC-V processor operation with support for external accelerators or memory-mapped I/O devices.

Key Functional Units:

- Datapath (datapath)
  Implements the core of the 5-stage pipeline (IF, ID, EX, MEM, WB), managing data flow, register operations, ALU execution, pipeline control signals, and integrates forwarding and memory access logic.
- Controller (controller)
  Decodes instructions and generates control signals that drive the datapath behaviour, including branching, ALU control, memory access type, and register file operations. It also handles AXI-specific control signals for memory transactions.
- Hazard Detection Unit (hazard_detection_unit)
  Monitors pipeline stages to detect data and control hazards, generates stall and flush signals, and controls forwarding paths (ForwardAE, ForwardBE) to maintain correct instruction execution in presence of dependencies.
- Instruction Memory (instruction_memory)
  Provides instruction fetch capability based on current PC value. It delivers the fetched instruction (Instr_F) to the Decode stage of the pipeline.

- Data Memory (data_memory)
  Represents the internal memory (RAM) used for data loads and stores. Interfaced via the AXI-Lite bridge to support future external memory/peripheral access.
- AXI4-Lite Bridge (axi_lite_bridge)
  Translates simple load/store memory transactions from the datapath into AXI4-Lite protocol operations for interfacing with external IPs or memory-mapped devices. It handles handshaking, response codes, and read/write channels (awvalid, wvalid, arvalid, etc.), enabling seamless off-chip communication.

AXI4-Lite Master Interface Support:

The module exposes full AXI4-Lite master signals, allowing the processor to:

- Write to external devices or memory-mapped registers (via awvalid, wvalid)
- Read from external sources (via arvalid, rready)
- Handle valid/ready signaling for transaction synchronization
- Interpret response codes (bresp, rresp) for error handling and interface compliance

The stall_axi signal is used by the datapath to synchronize memory accesses with AXI handshaking, preventing pipeline hazards due to incomplete memory transactions.

Hazard and Stall Awareness:

The processor is designed to be hazard-aware, integrating stall and flush signals into the pipeline stages to maintain instruction correctness. This allows it to handle:

- Load-use hazards with pipeline stalls
- Branch/jump instructions via pipeline flushing
- Forwarding for data hazards from MEM/WB stages to EX stage

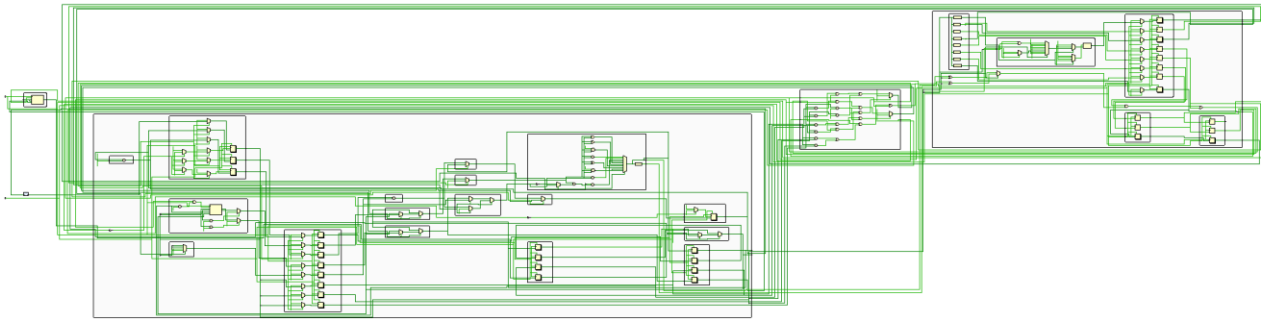
Modular and Extensible Design:

The riscv_top module is structured to support future enhancements:

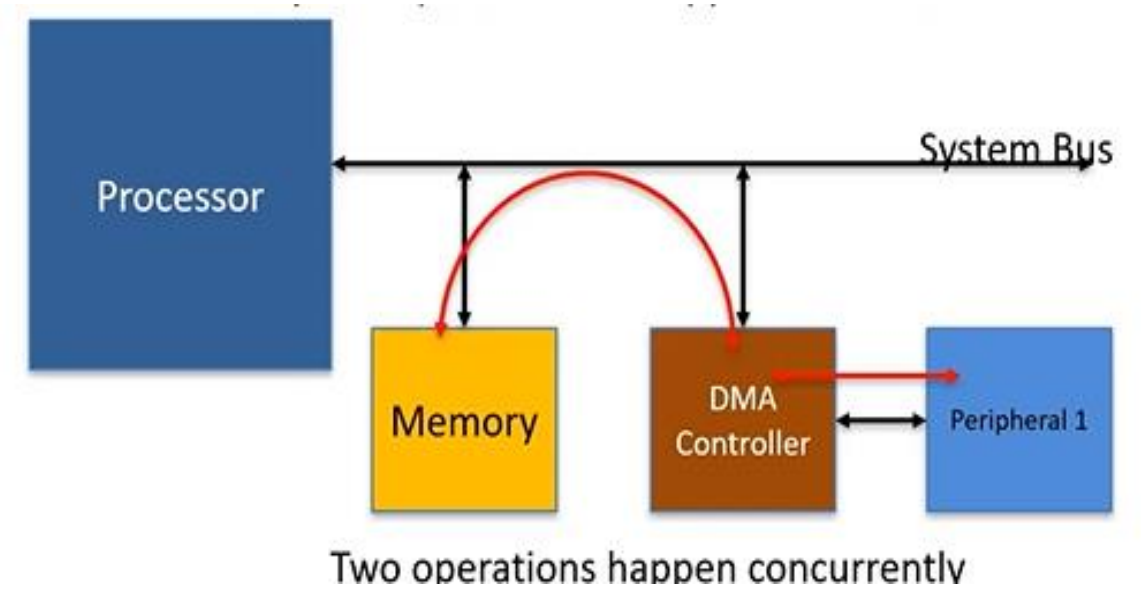
- Additional memory-mapped accelerators or peripherals (e.g., a CNN accelerator)
- External RAM or ROM interfacing
- CSR (Control and Status Registers) or interrupt handling logic

This module serves as the entry point for simulation, synthesis, and hardware deployment, making it the most critical unit for system-level behaviour and platform integration of the pipelined RISC-V processor.

RISC-V RTL Design



Section II
Direct Memory Access



What is DMA?

- **Definition & Purpose**
  DMA enables peripherals to transfer data to/from memory independently of the CPU, freeing CPU cycles and boosting efficiency—especially for large data movements.
- **Common Use Cases**
  Ideal for audio/video streaming, block transfers, high-speed I/O, and inter-device communication.

---

2. AXI DMA Architecture (Xilinx IP)

- **Primary Interfaces**
  - **MM2S (Memory-to-Stream)**: Transfers data from memory to AXI4-Stream (e.g., feeding a video encoder).
  - **S2MM (Stream-to-Memory)**: Moves stream data into memory (e.g., capturing video frames).
- **AXI Control Interface**
  Uses an AXI4-Lite slave port for configuration: setting source/destination addresses, lengths, control flags, and interrupt masks.

- **Descriptor vs. Direct Register Mode**
  - **Direct Register Mode**: Fixed, single-block transfers via control/status registers.
  - **Scatter-Gather Mode**: Supports chaining many transfers using descriptor lists for flexibility.

---

3. Configuration and Control

- **Flow Setup**
  1. **Reset** the DMA engine.
  2. Optionally clear or enable **circular buffering**.
  3. Set **framebuffer (FB)** addresses and sizes (H-size = bytes per line; V-size = lines/frame).
  4. Enable interrupts (like frame completion).
  5. Start MM2S and/or S2MM channel(s).
- **Circular Buffering**
  Enables continuous data flow across multiple frame buffers with automatic address cycling.
- **Genlock**
  Synchronization across channels—crucial for frame-based systems like video capture/playback.

---

4. Status and Interrupts

- DMA populates **status registers** to report:
  - Idle/running state
  - Error conditions
  - Frame counts and interrupt triggers
- **Interrupts** can be configured for various events: frame completion, delays, and error states.

---

5. Use in Video Systems

- **Dual-Channel Video Flow**: S2MM writes camera input into memory; MM2S reads from memory to display out—possibly with color conversion/scaling logic in between.
- **Demonstration Example**
  - Video capture loop using triple buffering.
  - Real-time status monitoring alongside simple driver code on ARM or bare-metal systems.
- **I/O Synchronization**
  Utilizes tuser/tlast signals for frame/line delimiting in AXI4-Stream audio/video pipelines.

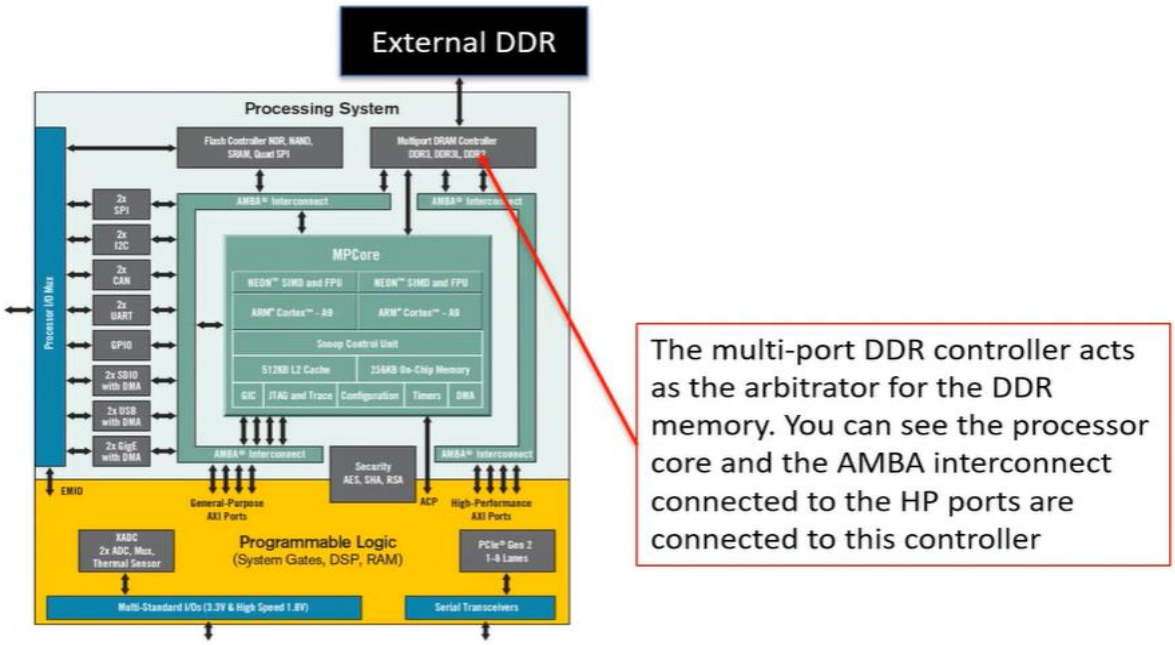
---

6. Software Integration Samples

- **C Driver Snippet**
  - open("/dev/mem") for direct register access.
  - mmap memory regions for register and framebuffer manipulation.
  - Register offsets defined as constants (e.g., MM2S_VSIZE, S2MM_HSIZE, etc.).
  - Functions for resetting, starting in circular mode, and status polling.
- **Triple Buffering Example**
  - Demonstrates setup of three FB addresses and operation start sequence.
  - Shows status reads to check for completion and interrupts.
- **Python HTTP Frame Server**
  - HTTP server reads framebuffer via /dev/mem and serves over network.

---

7. DMA on Zynq



---

8. Best Practices & Real-world Notes

- **Memory Safety**
  Avoid uncontrolled memory access—prefer kernel drivers over raw /dev/mem for production systems.
- **Buffer Management**
  Circular or scatter-gather buffering supports continuous, high-bandwidth transfers.
- **Error Handling**
  Always poll or clear error bits; implement recovery (reset, re-init) if needed.

Section III
AXI Interface

This section covers:

1. AXI Interface Fundamentals (AXI4-Lite, AXI4-Stream, AXI4-Memory Mapped)
2. Custom AXI4-Stream IP Core Design Flow
3. DMA Integration in Vivado Projects
4. Inverter Project Using AXI Interconnect

AXI Interface Overview

AXI4 Memory-Mapped (MM)

- **Purpose**: High-performance, burst-mode access to memory or peripherals.
- **Channels**: Read (AR/R), write (AW/W/B).
- **Burst transfers** support contiguous data access, addressing interleaving.

AXI4-Lite

- **Purpose**: Lightweight, simple control/status register access.
- **Non-burst single transactions**, minimal resources for register-mapped IP.

AXI4-Stream

- **Purpose**: High-throughput streaming data (e.g., video, audio, sensor).
- **Handshake signals**: TVALID, TREADY, plus optional signals (TKEEP, TLAST, TUSER).
- **Unidirectional point-to-point streaming**, no addressing overhead.

---

2. Custom AXI4-Stream IP Core in Vivado

Creating an AXI4-Stream IP (e.g., image filter) using Vivado's IP packager:

- **Block design**: Define sink/source interfaces (S_AXIS, M_AXIS) with handshake signals.
- **AXI4-Stream Interface**: Includes TVALID, TREADY, TDATA, optionally TLAST/TUSER.
- **Logic implementation**:
  - Accept data when S_AXIS_TVALID && S_AXIS_TREADY == 1.
  - Process data (e.g., filtering).
  - Output processed data via M_AXIS_TDATA; wait for destination TREADY.
  - Manage TLAST to signal end-of-frame or block.
- **Packaging**:
  - Connect to Vivado's IP integrator bus interfaces.
  - Define parameters like data width and buffer depths.
- **Integration**:
  - Insert IP into block design.
  - Connect source (e.g. DMA) to sink, processed data exits to another module or memory.
- **Debugging**:
  - Simulate data flow; ensure handshake and back-pressure function correctly.

---

3. DMA System-level Design with Custom IP

- **Setup**:
  - Insert AXI DMA IP supporting MM2S and S2MM channels.
  - Define data path: memory ↔ DMA ↔ custom IP ↔ DMA ↔ memory.
- **AXI4-Stream Interconnect**:
  - DMA MM2S output connects to custom IP S_AXIS.
  - IP M_AXIS connects to DMA S2MM input.
- **DMAC Configuration**:
  - Use AXI4-Lite for programming DMA (base address, lengths).
  - Supports scatter-gather descriptors or fixed transfers.

- **Design Flow**:
  - Generate block design with DMA and custom IP.
  - Connect clocks, resets, AXI-Lite to control path, and AXI-Stream to data path.
  - Export hardware to SDK/Vitis; write bare-metal code to:
    - Initialize DMA,
    - Point its MM2S to source memory,
    - Start transfer,
    - Poll for completion,
    - Capture processed output.

---

4. Inverter Project Using AXI Interconnect

- **Goal**: Build a custom IP that bitwise-inverts input data stream and interfaces via AXI4-Stream or AXI4-Lite.

IP Design

- Input via AXI4-Stream (S_AXIS_TDATA).
- Logic: out_data = ~in_data.
- Output via M_AXIS_TDATA, preserving handshake and TLAST.

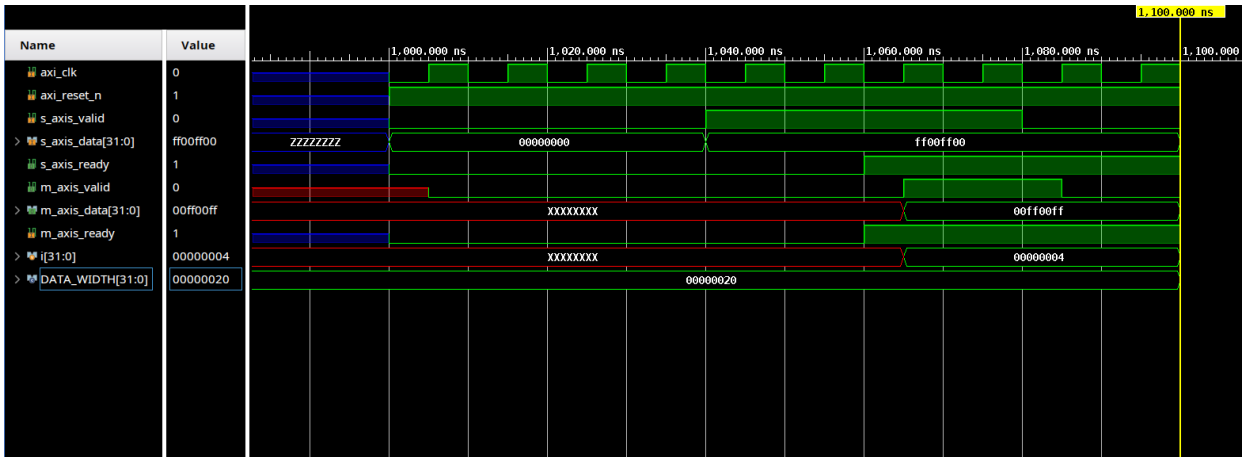
Block Integration

- Connect IP between DMA MM2S and S2MM.
- Present as an AXI4-Lite peripheral with control registers (e.g., enable invert).

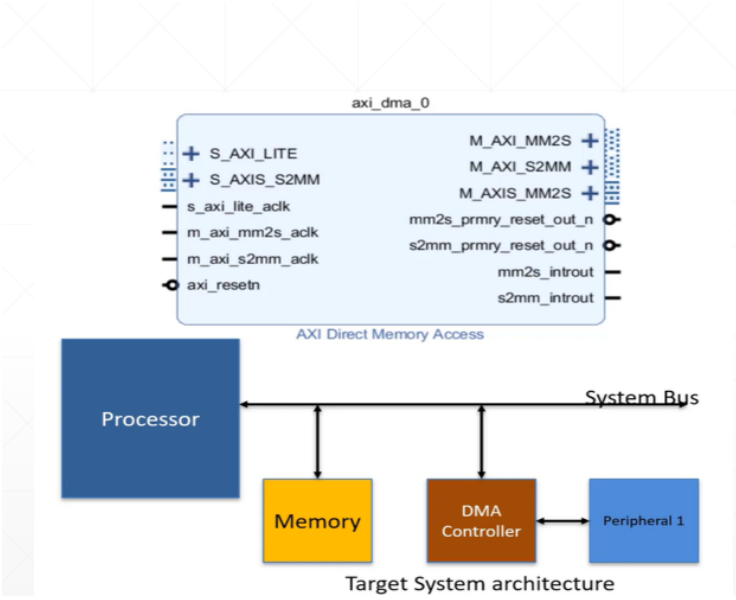
Testing

- Use Vivado simulation to feed known patterns; verify inverted output matches expectation.
- On hardware, stream test data through DMA pipeline and validate output in memory.

Simulation of Inverter in waveform



Xilinx AXI DMA

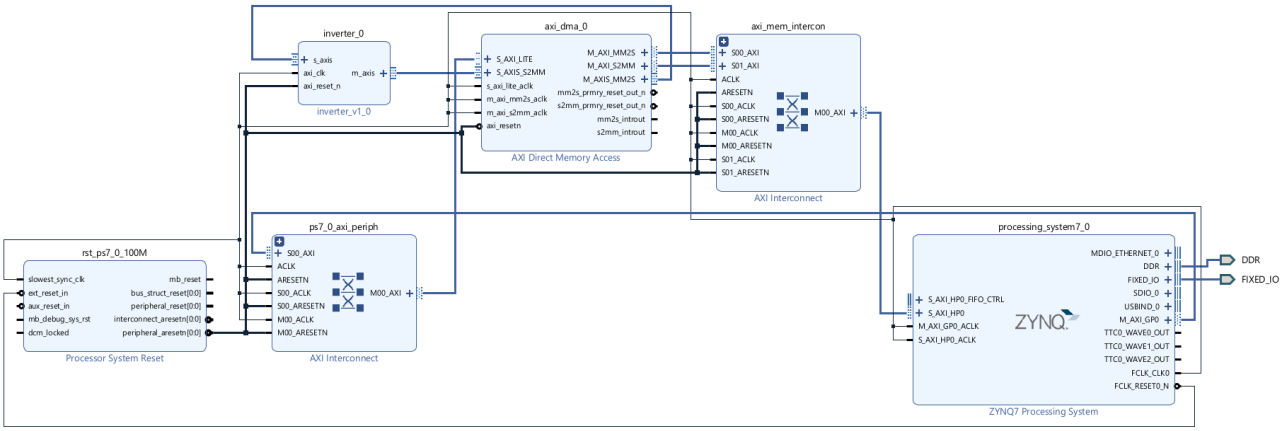


- S_AXI_LITE: interface for configuring DMA from processor (AXI4-Lite protocol)
- S_AXIS_S2MM: read interface from peripheral to DMA (AXI Stream protocol)
- M_AXI_MM2S: read interface from Memory to DMA (AXI4 Full protocol)
- M_AXI_S2MM: write interface from DMA to Memory (AXI4 Full protocol)
- M_AXIS_MM2S: write interface from DMA to peripheral (AXI Stream protocol)

****MM2S: Memory Mapped to Stream*
****S2MM: Stream to Memory Mapped*

(performance higher than AXI Lite)

AXI-4 Stream IP Tested for Inverter testing project



Summary of AXI Interface

Feature	AXI (Full)	AXI Lite	AXI Stream (AXIS)
Memory Mapped	✅ Yes	✅ Yes	❌ No
Point-to-Point	❌ No (via interconnect)	❌ No	✅ Yes (direct Master ↔ Slave)
Bursting Support	✅ Yes	❌ No (only single beat transfers)	✅ Yes (continuous stream, no address)
Use Case	High-throughput data (DMA, memory)	Configuration, register access	Real-time streaming (video, sensors)
Complexity	High	Low	Medium

Section IV
Design and Implementation of MNIST Dataset recognition using a fully connected Convolutional Neural Network in Verilog HDL

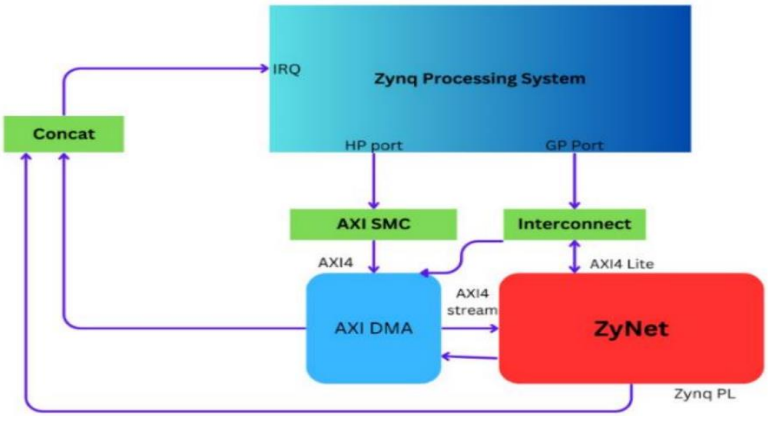
Abstract
This section outlines the design and FPGA implementation of a handwritten digit recognition system using the MNIST dataset and a fully connected CNN architecture described in Verilog HDL. The design leverages the FPGA's pipelining, memory-mapped architecture, and parallelism to enable high-speed inference. The report provides detailed insights into data preprocessing, architectural decisions, Verilog design methodology, and interfacing with AXI peripherals.

---

1. Introduction
The MNIST database is a large dataset of handwritten digits commonly used for training and testing image processing systems. Each image in MNIST is a 28x28 grayscale image, representing handwritten digits from 0 to 9. A neural network implemented on an FPGA offers real-time digit recognition with low power and high throughput.

---

2. System Architecture (only Neural Networks and PS)



3. Dataset Overview and Preprocessing

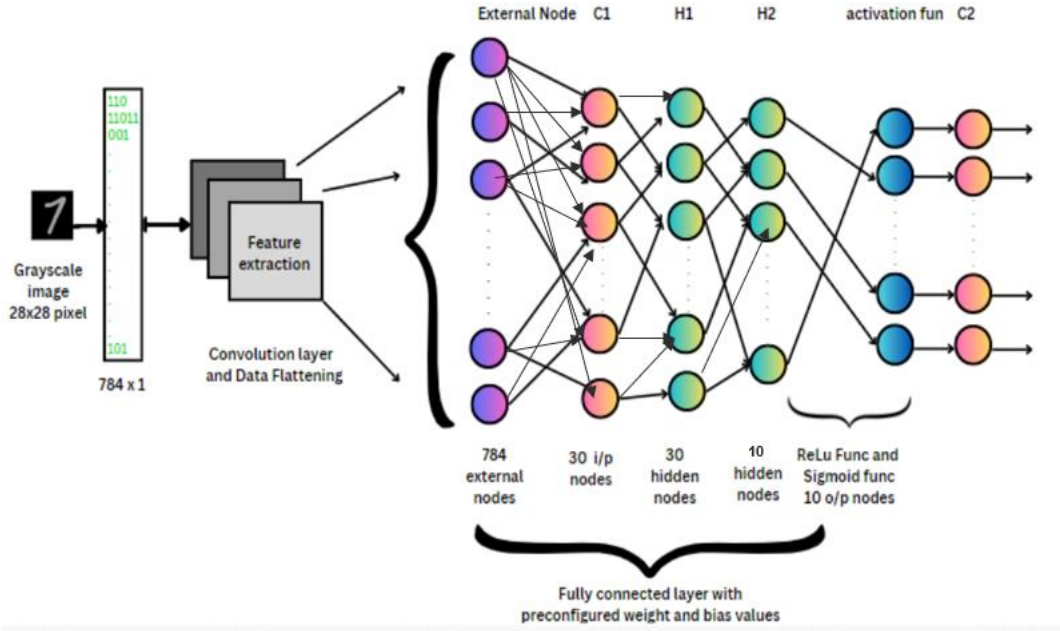
- **Input Format**: Each MNIST image is 28x28 grayscale pixels.
- **Normalization**: Pixel values (0-255) are scaled to a fixed range such as 0 to 1 or -1 to 1.
- **Binary Representation**: Normalized values are converted into fixed-point binary format (e.g., 16-bit) using a scaling function.
- **Flattening**: Images are flattened into a 784x1 vector for input to the first layer.

---

4. Neural Network Architecture

- **Layer 1**: 784 neurons (one per pixel).
- **Layer 2**: 30 neurons.
- **Layer 3**: 30 neurons.
- **Layer 4**: 10 neurons.
- **Layer 5 (Output)**: 10 neurons (one per digit class).

Each neuron receives the same 784 inputs but has unique weights and biases.



---

5. Activation Functions

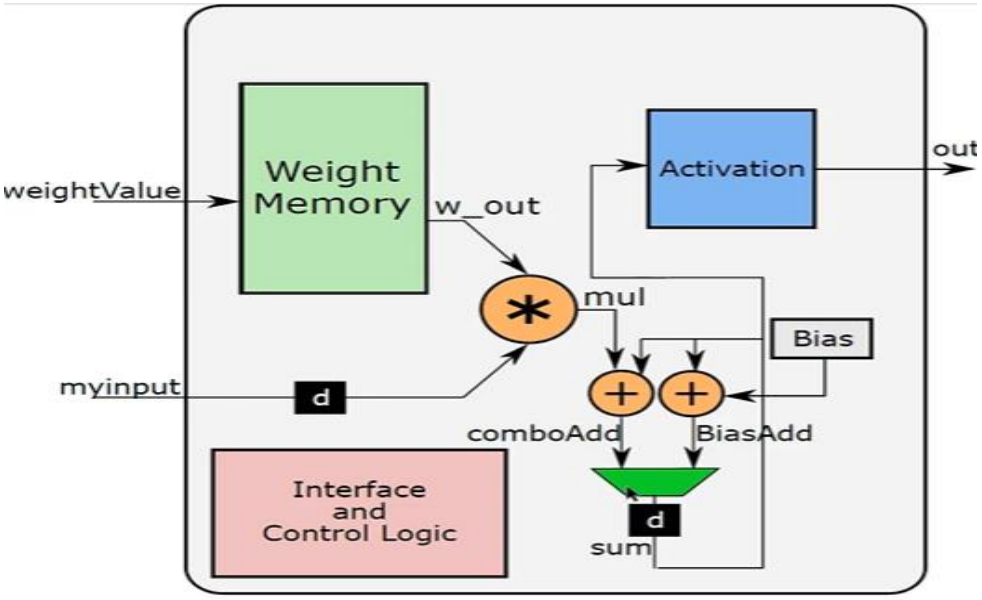
- **Sigmoid**: Smooth activation function approximated using a Look-Up Table (LUT). Output size is determined by sigmoidSize (e.g., 5 or 10).
- **ReLU (Rectified Linear Unit)**: Output is 0 for negative values and the value itself otherwise. Saturation is handled by checking overflow using sign bits.

---

6. Verilog Design Components

6.1 Neuron Module *(neuron.v)*

- Accepts one input at a time.
- Stores weights and bias from .mif files or dynamically via AXI.
- Multiplies delayed inputs with stored weights.
- Sums products and applies bias.
- Applies activation function.



6.2 Weight Memory *(WeightMemory.v)*

- Dual-purpose ROM/RAM based on the pretrained define.
- Readable and writable using AXI.
- Uses $readmemb() for preload.

6.3 Sigmoid_ROM *(Sig_ROM.v)*

- Implements the sigmoid activation using fixed-point LUT.
- Uses an address derived from the upper bits of the summed input.

6.4 ReLU Module *(relu.v)*

- Processes signed input.
- Implements overflow saturation logic.

6.5 Layer Modules *(Layer_1.v,Layer_2.v,Layer_3.v,Layer_4.v)*

- Multiple neurons instantiated in parallel.
- Output data collected in x_out using bit slicing: x_out[N*dataWidth+:dataWidth].

6.6 Softmax *(maxFinder.v)*

Purpose: Identifies the index of the maximum activation value from the final layer of a neural network (e.g., classifying digits 0–9 in MNIST).

Input/Output: Takes a 1D input vector of numInput values (each inputWidth bits), and outputs the index of the maximum value (o_data) along with a valid signal (o_data_valid).

Operation: Sequentially compares inputs over numInput clock cycles using pipelined logic to reduce hardware complexity.

Efficiency: Optimized for FPGA by avoiding large combinational comparators, making it both resource-efficient and timing-friendly.

6.7 AXI Lite Wrapper *(axi_lite_wrapper.v)*

The axi_lite_wrapper module acts as an AXI4-Lite Slave Interface used for configuration and control of the neural network accelerator implemented on FPGA.

This module is responsible for:

- Receiving weight and bias values for each neuron from the processor via the AXI-Lite interface.
- Receiving layer and neuron index to target the appropriate storage registers inside the neural network.
- Providing soft reset control to reinitialize network state from the processor.
- Exposing the classification output and other status signals (like nnOut_valid, axi_rd_data) back to the processor.
- Handling read/write transactions through a memory-mapped register set with minimal latency.

- Generating handshake logic and state machines for clean AXI protocol compliance.

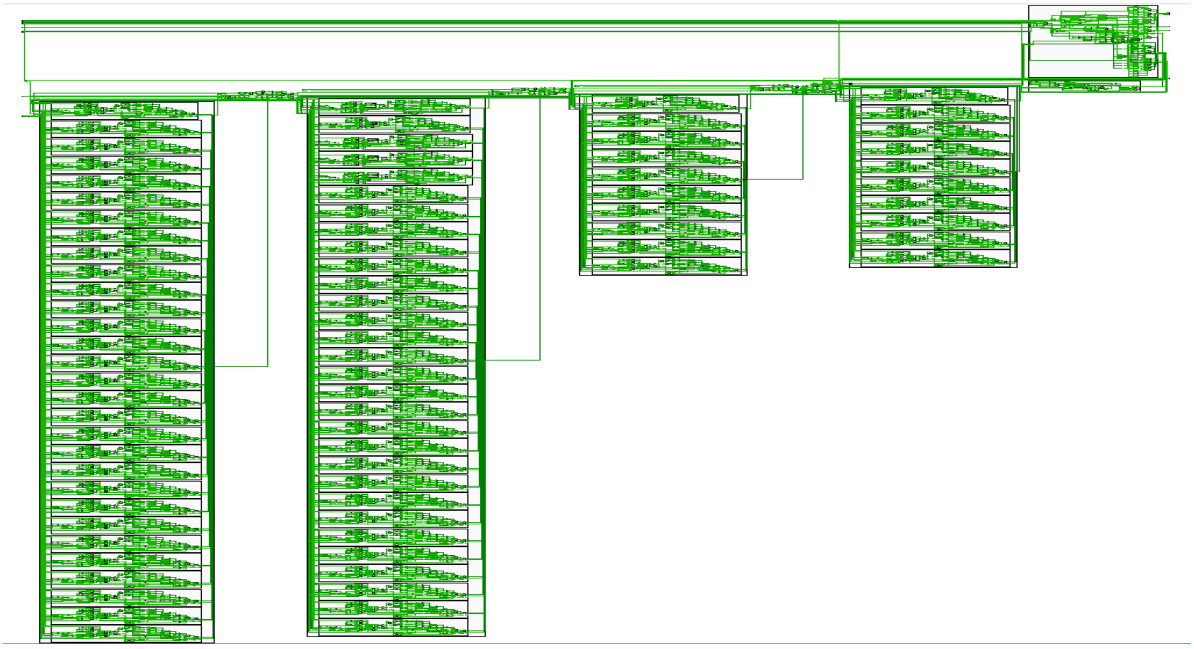
This wrapper isolates the complexity of the neural network internals and allows software control using standard AXI transactions from a Zynq Processing System

6.8 Zynet *(zynet.v)*

The zynet module is the top-level system wrapper for the CNN-based neural network accelerator on FPGA. It integrates:

- AXI4-Lite Interface (via axi_lite_wrapper) for register-based configuration/control.
- AXI Stream Interface for inputting the normalized MNIST image data (784 pixels).
- Multiple layer submodules (Layer_1, Layer_2, Layer_3, Layer_4) that simulate a deep feed-forward CNN structure.
- Shift register-based pipelining logic between layers to model the temporal behavior of data flowing through neurons.
- A maxFinder submodule that performs final digit classification by detecting the highest activated output neuron.
- An interrupt signal (intr) which flags the processor when inference is complete and the classification result is ready.

7. Neural Network RTL Design



7. Training modules (python)

7.1 MNIST Loader module *(mnistloader.py)*

This module handles loading and preprocessing the MNIST dataset.

- **Functionality:** Loads training, validation, and test datasets from a gzipped pickle file (mnist.pkl.gz).
- **Format Conversion:** The load_data_wrapper() function reshapes inputs into column vectors and converts labels into one-hot vectors (only for training data).
- **Usage:** Used primarily to prepare data for training and evaluating neural networks.
- **Data Format:** Training data is formatted as (x, y) pairs where x is a 784×1 input vector and y is a 10×1 one-hot encoded label.

7.2 Activation Function Defining module *(network2.py)*

This is the core neural network implementation using stochastic gradient descent (SGD).

- **Main Features**: Implements a feedforward neural network with support for cross-entropy and quadratic cost, L2 regularization, and backpropagation.
- **Initialization**: Uses improved weight initialization for better convergence (weights scaled by 1/sqrt(n)).
- **Training**: Supports mini-batch SGD with flexible monitoring options like training accuracy, cost, evaluation accuracy, etc.
- **Saving and Loading**: Can save and load trained weights, biases, and architecture via JSON files.

7.3 Training Script *(trainNN.py)*

This script uses the above two modules to train a neural network on MNIST.

- **Network Structure**: Defines a network with 5 layers: [784, 30, 30, 10, 10], where the last 10 units are likely for class index voting.
- **Training Settings**: Trains for 30 epochs using mini-batch SGD with a learning rate of 0.1 and L2 regularization (lambda = 5.0).
- **Monitoring**: Tracks evaluation accuracy during training for performance analysis.

- **Result**: Saves the trained model's weights and biases to a file (WeightsAndBiases.txt) for deployment or inference.

7.4. Training and Epoch

- **Epoch**: One complete pass through all training images.
- During training, weights and biases are updated.
- During testing, inference is done with fixed weights (pretrained).

---

8. Simulation Timing Results

Tested ReLU, Sigmoid size 10 and size 5

ReLU

- Worst Negative Slack : -0.799
- Fmax: 208MHz
- Accuracy: 90.00000%

****Ideally considered: 250MHz*

Sigmoid Size 10,5

- Worst Negative Slack : -0.717
- Fmax: 212MHz
- Accuracy: 98.00000%

- *Sigmoid Size 5,5
  - Worst Negative Slack : -0.816
  - Fmax 207MHz
  - Accuracy: 96.00000%

---

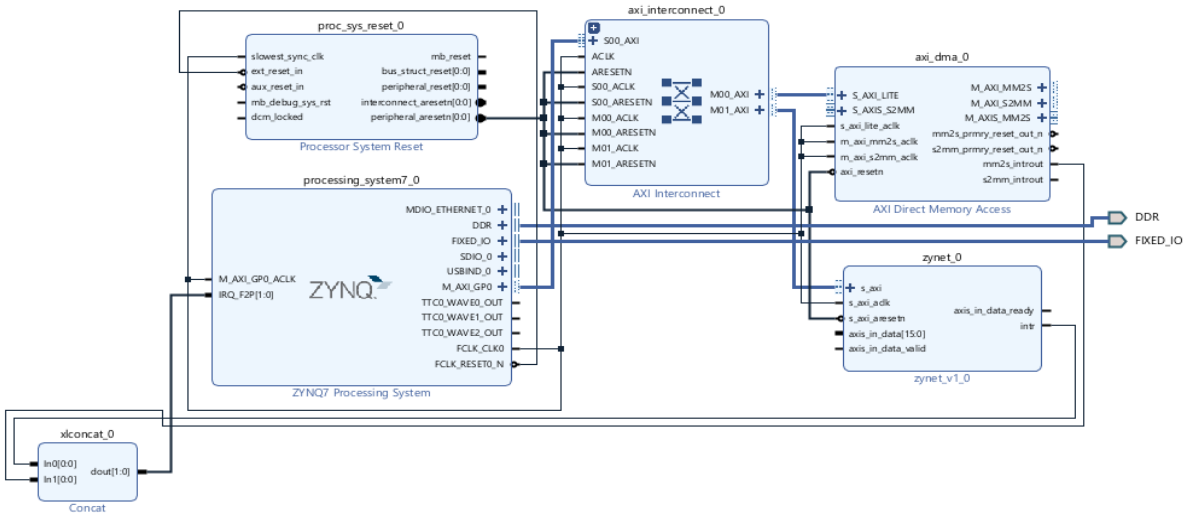
9. Shift Registers and Pipelining

- Between layers, shift registers are used to hold data temporarily.
- Implements a latch mechanism (Hold_Data1, Out_Data1) for synchronous data flow.
- Supports pipelined architecture allowing simultaneous input to layer 1 while layer 2 processes previous data.

---

10. AXI DMA and Zynq Interfacing IP Block Design

- **Data is transferred** from PS (Processing System) to PL (Programmable Logic) via AXI-DMA.
- Interrupts handled using XScuGic controller.
- Final classification output read via memory-mapped IO (Xil_In32(BASEADDR + offset)).



11. Resource Utilisation data for both functions

ReLU

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	75	53200	0.14
FF	66	106400	0.06
BRAM	0.50	140	0.36
DSP	2	220	0.91
IO	36	200	18.00
BUFG	1	32	3.13

Sigmoid Size 10,5

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	68	53200	0.13
FF	52	106400	0.05
BRAM	1	140	0.71
DSP	2	220	0.91
IO	36	200	18.00
BUFG	1	32	3.13

Sigmoid Size 5,5

Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	73	53200	0.14
FF	56	106400	0.05
BRAM	0.50	140	0.36
DSP	2	220	0.91
IO	36	200	18.00
BUFG	1	32	3.13

12. Final IP block design timing summary

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.951 ns	Worst Hold Slack (WHS):	0.016 ns	Worst Pulse Width Slack (WPWS):	3.750 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	20992	Total Number of Endpoints:	20992	Total Number of Endpoints:	6019

All user specified timing constraints are met.

Worst Negative Slack: 0.951ns
Fmax: 327MHz (higher than 250MHz (considered))

12. ZedBoard Considerations

- Project adapted for **ZedBoard** by modifying the .xpfm platform.
- Used Vitis 2023.2 to create application and hardware platform.
- Corrected errors due to missing headers and DMA base ID macros.

---

13. Challenges and Resolutions

- **File inclusion errors** in Vitis (xaxidma.h, ctype.h) fixed by verifying platform settings.
- Platform .xpfm for ZedBoard added manually.
- Sigmoid LUT generation script permissions fixed (PermissionError solved by writing to valid path).
- Verified neuron accuracy using AXI registers.

Section V
Integration of Zynq Processing System & RISC-V Processor, Neural Networks as Programmable Logic using AXI Interconnects

Modifying RISC-V Processor to Control Neural Network via AXI

Introduction

The goal of this section is to describe the architectural and design modifications necessary to enable a custom 32-bit RISC-V processor (RV32I) to interface with a hardware neural network accelerator (Zynet IP) using the AXI4-Lite protocol. This integration enables the processor to configure, initiate, and read back results from the neural network through memory-mapped I/O.

System Overview till Now

The integrated system comprises the following components:

- RISC-V Processor implemented in Verilog (RV32I ISA).
- Zynet Neural Network Accelerator implemented in Verilog and packaged as an AXI4-Lite + AXI-Stream IP.
- AXI4-Lite Bridge Interconnect for register-based communication between RISC-V and NN. (**new add-on**)
- Memory-Mapped Address Decoder / AXI-Lite Master introduction in RISC-V datapath module. (**new add-on**)
- Zynq Processing System (only for clock/reset generation, not interacting directly with RISC-V or Zynet).

Functional Objective

Enable the RISC-V processor to:

- Configure the NN accelerator (layer number, neuron number, control flags).
- Write weight and bias data into the NN accelerator.
- Send input image data.
- Trigger execution and poll for classification output.
- Receive results via AXI-Lite reads.

Architectural Modifications in RISC-V Processor

Addition of AXI4-Lite Bridge Interface

Module: axi_lite_bridge.v

Purpose:

This module bridges a custom RISC-V processor's memory stage to both:

- Regular data memory (RAM)
- AXI4-Lite-compliant peripheral, the Zynet neural network accelerator.

The module allows memory-mapped I/O (MMIO) to be cleanly handled in the RISC-V memory stage, with automatic decoding and FSM-based AXI control, stalling the processor pipeline only when necessary.

Features

Dual Interface Access

- **RAM Interface** (standard): Used for regular load/store instructions.
- **AXI4-Lite Master Interface**: Used for writing/reading configuration, weights, biases, and control signals to/from hardware accelerators mapped in the 0x40000000 - 0x4000FFFF region.

Automatic Address Decoding

- **AXI Region**:
  All addresses in the range 0x40000000 to 0x4000FFFF are routed to the AXI interface.
- **RAM Region**:
  All other addresses go to the data memory directly.

Working Principle

AXI Access Detection:

```
wire axi_write_req = mem_write_en && (mem_addr >= AXI_BASE_ADDR && mem_addr <= AXI_HIGH_ADDR);

wire axi_read_req  = ~mem_write_en && (mem_addr >= AXI_BASE_ADDR && mem_addr <= AXI_HIGH_ADDR);
```

These control signals identify if the memory access should be redirected to the AXI-Lite bus.

Finite State Machine (FSM)

States:

State	Purpose
IDLE	Wait for new memory access
AXI_WRITE_ADDR	Send address for AXI write
AXI_WRITE_DATA	Send data for AXI write
AXI_WRITE_RESP	Wait for write response (BVALID)

State	Purpose
AXI_READ_ADDR	Send address for AXI read
AXI_READ_DATA	Wait and capture AXI read data

This FSM ensures AXI protocol compliance for both read and write transactions.

Stall Logic

assign stall_axi = (current_state != IDLE);

The stall_axi signal goes high whenever an AXI transaction is in progress, stalling the RISC-V memory stage to ensure data consistency.

RAM Control

- RAM accesses are performed only in IDLE state, when not accessing AXI:

```
if (!axi_write_req && !axi_read_req && mem_write_en)
   ram_write_en <= 1;
```

AXI Signal Handling

- **Write Address Phase** (awvalid, awaddr, awprot)
- **Write Data Phase** (wvalid, wdata, wstrb)
- **Write Response Phase** (bready)
- **Read Address Phase** (arvalid, araddr, arprot)
- **Read Data Phase** (rready, rvalid, rdata, rlast)

Each AXI transaction is handled through proper sequencing of handshakes, and the data read from AXI is latched into mem_read_data.

Future Tasks for the integration of RISC-V and Neural Networks

Addition of AXI4-Lite Master Interface
Module: axi_lite_master.v
Function:
Acts as an AXI4-Lite Master to perform:

- Write transactions (e.g., NN_WEIGHT, NN_BIAS, NN_CTRL)
- Read transactions (e.g., NN_RESULT)

Integration:
This module is integrated into the memory stage of the pipeline. A subset of addresses is decoded and routed through AXI instead of the regular data memory.
Address Mapping:

```
#define NN_BASE         0x40000000
#define NN_CTRL         (NN_BASE + 0x00)
#define NN_WEIGHT       (NN_BASE + 0x10)
#define NN_BIAS         (NN_BASE + 0x20)
#define NN_INPUT        (NN_BASE + 0x30)
#define NN_RESULT       (NN_BASE + 0x40)
```

4.2 Address Decoder in Memory Stage
Purpose:
To detect whether the memory access from the RISC-V processor is targeting:

- Regular data memory
  OR
- Neural network IP via AXI-Lite

4.3 Stall and Handshaking Logic
Problem:
AXI4-Lite introduces wait cycles. The RISC-V core must stall if:

- AXI write is ongoing and not acknowledged.
- AXI read is not yet valid.

4.4 Result Capture and Forwarding

Problem:

AXI read result may arrive one or more cycles after initiating the read.

---

5. Pipeline Integration Summary

Pipeline Stage	Modification Summary
IF/ID	No change
ID/EX	No change
EX/MEM	Detect address target (data mem or AXI)
MEM	Route to data_mem or axi_lite_master
WB	AXI result forwarded here, same as memory data

---

6. Software API (C-Level)

The RISC-V program (written in C and compiled to RV32I) interacts with the NN via memory-mapped registers.

```
#define NN_CTRL   0x40000000
#define NN_WEIGHT 0x40000010
#define NN_BIAS   0x40000020
#define NN_INPUT  0x40000030
#define NN_RESULT 0x40000040
void write_reg(uint32_t addr, uint32_t data) {
   *(volatile uint32_t *)addr = data;
}
uint32_t read_reg(uint32_t addr) {
   return *(volatile uint32_t *)addr;
}
```

7. Test and Debug Considerations

Simulation method (probable):

- Testbench initializes weights, biases, inputs.
- Monitors AXI transactions using Vivado.

Verification Strategy (probable):

- Compare final output of NN (NN_RESULT) with expected classification.
- Monitor AXI handshakes (awvalid, wready, bvalid, etc.)

---

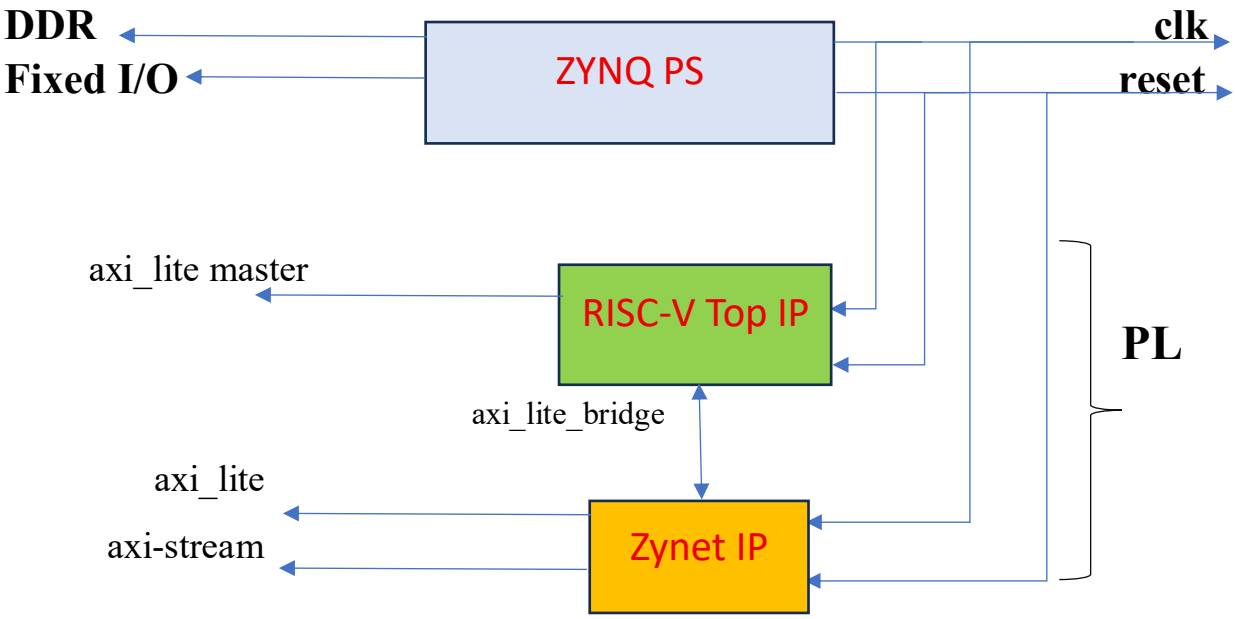
8. The C code discussed below:

```
#include <stdint.h>

#define NN_BASE         0x40000000
#define NN_CTRL         (NN_BASE + 0x00)
#define NN_WEIGHT       (NN_BASE + 0x10)
#define NN_BIAS         (NN_BASE + 0x20)
#define NN_IMG_IN       (NN_BASE + 0x100)
#define NN_OUTPUT       (NN_BASE + 0xC00)
#define NN_OUT_VALID    (NN_BASE + 0xC04)

int main() {
   // Write one weight value
   *(volatile uint32_t *)(NN_WEIGHT) = 0x3F800000;

   // Write one image pixel
   *(volatile uint32_t *)(NN_IMG_IN) = 0x000000AA;

   // Trigger the neural network
   *(volatile uint32_t *)(NN_CTRL) = 0x1;

   // Wait for output to be ready
   while (*(volatile uint32_t *)(NN_OUT_VALID) == 0);

   // Read the output result
   uint32_t result = *(volatile uint32_t *)(NN_OUTPUT);

   // Do something with result
   while (1); // halt
}
```

is then:

- Compiled using a RISC-V GCC toolchain (riscv64-unknown-elf-gcc)
- Linked so that the .text section goes to address 0x00000000 (your instruction memory base)
- Converted to a ".hex" or ".mem" file
- Loaded into instruction memory via simulation or FPGA programming

Final Integration basic figures should be somewhat similar to the below diagram flow.



Major Problems Encountered

- Failure of AMD Vivado and Vitis softwares multiple times and eventually hanging up of these softwares in later stages.
- Failure of Linux Operating System in later stages.
- Availability of RISC-V GNU GCC Complier Toolchain only for Linux OS and not for Windows OS.

Source Code links to all the work done in this internship:

GitHub
Hardware-Accelarated CNN on RISC-V integrated with FPGA
https://github.com/aswinsilicon/CNN_FPGA

GitHub
Image Processing on FPGA (Extras)
https://github.com/aswinsilicon/Edge_Detection_on_Zynq

GitHub
Custom AXI4- Stream IP
https://github.com/aswinsilicon/Custom_AXI4-Stream_IP

GitHub
Custom User IP Core
https://github.com/aswinsilicon/Custom_IP

Link to my LinkedIn
https://www.linkedin.com/in/aswinsilicon/

Referrences (Bibliography)

[1] A Hardware Accelerator for The Inference of a Convolutional Neural Network - Edwin González, Walter D. Villamizar Luna, Carlos Augusto Fajardo Ariza .

[2] FPGA Acceleration on a Multi-Layer Perceptron Neural Network for Digit Recognition Isaac Westby · Xiaokun Yang* · Tao Liu · Hailu Xu

[3] A Digits-Recognition Convolutional Neural Network on FPGA by Zenyu wang -2019.

[4] Handwritten Digit Recognition System on an FPGA, Jiong Si and Sarah L. Harris

[5] ZyNet: Automating Deep Neural Network Implementation on Low-cost Reconfigurable Edge Computing Platforms by Kizheppatt Vipin

[6] The Role of Activation Function in CNN Wang Hao; Wang Yizhou; Lou Yaqin; Song Zhili

[8] Giardino. D, Matta. M, Silvestri. F, Spanò. S, Trobiani. V, "FPGA implementation of hand-written number recognition based on CNN." International Journal on Advanced Science, Engineering and Information Technology, 2019, 9(1):

[9] Mittal. Sparsh, "A survey of FPGA-based accelerators for convolutional neural networks." Neural computing and applications.

[10] Research and Implementation of CNN Based on TensorFlow Liang Yu1, Binbin Li1 and Bin Jiao

[11] ZyNet git repository. Available: https://github.com/dsdnu/zynet

Thank You