School of
Electrical and Electronics Engineering

ADIC Course Project

Problem Statement: Calculation of power[A ^{B}]$using MIPS32$

By:

1. Girish subhas Koni — USN: 01FE22BEC272
2. Prajwal Honnalli — USN: 01FE22BEC276
3. Vinayak Todakar — USN: 01FE22BEC280
4. Nagaraj Yallappa Hosamani — USN: 01FE22BEC286

Semester: **VI, 2024-2025**

Under the Guidance of

Dr.Saroja V Siddamal

K.L.E SOCIETY'S
KLE Technological University,
HUBBALLI-580031
2024-2025

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

CERTIFICATE

This is to certify that the project entitled **" Calculation of power using MIPS32 "** is a bona fide work carried out by the student team of **"Girish subhas koni: 01fe22bec272, Prajwal Honnalli: 01fe22bec276, Vinayak Todakar: 01fe22bec280, Nagaraj Yallappa Hosamani: 01fe22bbec286"**. The project report has been approved as it satisfies the requirements with respect to the ADIC course project work prescribed by the university curriculum for BE (VI Semester) in Department of Electronics and Communication Engineering of KLE Technological University for the academic year 2024-2025.

Dr. Saroja V Siddamal **Suneeta V Budihal** **B. S. Anami**

Guide Head of School Registrar

External Viva:

Name of Examiners	**Signature with date**
1.	
2.	

ACKNOWLEDGEMENT

We would like to thank our faculty and management for their professional guidance towards the completion of the project work. We take this opportunity to thank Dr. Ashok Shettar, Pro Chancellor KLE Technological University, Hubballi for their vision and support. We also take this opportunity to thank Dr. Suneeta V. Budihal, Head, School of Electronics and Communications, for providing us direction and facilitating enhancement of skills and academic growth.

We thank our guide Dr. Saroja V Siddamal, Head, School of Electrical and Electronics for the constant guidance during interaction and reviews. We extend our acknowledgement to the reviewers for critical suggestions and inputs. We also thank the project coordinator and faculty in-charges for their support during the course of completion. We express gratitude to our beloved parents for constant encouragement and support.

– Girish K,Vinayak T, Prajwal H, Nagaraj H

ABSTRACT

This project presents the implementation of exponentiation, specifically calculating the value of a^b, using the MIPS32 processor architecture. MIPS32 is a 32-bit RISC (Reduced Instruction Set Computing) architecture widely adopted in embedded systems and academic environments due to its simplicity and efficiency.

The core objective is to design an assembly-level algorithm that takes two positive integers—a as the base and b as the exponent—and computes the result of a^b. The computation is structured to follow the MIPS32 pipeline stages, involving instruction fetch, decode, execution, memory access, and write-back.

The developed MIPS32 assembly code is first tested through simulation to verify correctness and performance. Following successful simulation, the implementation is deployed on an FPGA board using a Verilog wrapper. Input values are provided via hardware switches or UART interface, and the result is observed through LED indicators.

This project highlights the efficiency of low-level programming in executing mathematical operations on hardware platforms. It also emphasizes how pipelined processor architectures like MIPS32 can be used to accelerate such operations. The integration of MIPS32 with FPGA demonstrates a practical approach to designing computational circuits for real-time embedded applications.

Contents

Chapter 1

Power Calculation using MIPS32

1.1 Motivation

The aim of this project is to design and implement exponentiation (A^B) using the MIPS32 architecture. MIPS32 is a 32-bit RISC (Reduced Instruction Set Computing) processor widely used in embedded systems and academics. This project enables students to gain practical experience in low-level programming and system architecture using assembly language.

Students implement the power calculation logic using MIPS32 assembly instructions and simulate the execution both in software and on FPGA hardware. The goal is to understand processor execution flow and instruction-level control by writing and deploying custom code on a MIPS processor.

- To implement a^b using low-level instructions

→ Helps develop logic for repeated multiplication and looping using MIPS32 registers and memory.

- To explore simulation and hardware testing

→ The MIPS32 code is simulated using a testbench and implemented on FPGA using Verilog wrappers.

- To integrate theory and practical knowledge

→ The project connects theoretical topics like instruction formats and ALU operations with hands-on experience in embedded system design.

1.2 Objectives

The aim is to develop an assembly program that calculates the power of two numbers (a^b) using the MIPS32 architecture. This project helps students understand low-level programming concepts and how mathematical operations like exponentiation can be implemented using system-level processor instructions.

The project requires writing a MIPS32 assembly program that multiplies a base number a by itself b times using loop and register operations. It strengthens understanding of pipelined processor architecture, register manipulation, and instruction flow in real embedded systems.

- To implement power calculation (a^b) using MIPS32 assembly language

→ The main goal is to write a MIPS32 program that accepts base and exponent inputs and returns a^b as output.

- To understand and apply pipelined processor architecture for efficient execution

→ Learn how pipelining enhances execution speed by overlapping instruction stages.

- To simulate the instruction flow and verify correct logic operation

→ Use simulation tools to validate correct logic implementation and conversion accuracy.

1.3 Problem statement

POWER CALCULATION (A^B) USING MIPS32 PROCESSOR. The problem focuses on designing and implementing a MIPS32-based assembly program that computes the result of raising a base number `a` to an exponent `b` (i.e., `a^b`). The system takes two positive integers as input and uses iterative multiplication within a pipelined MIPS32 processor to compute the power. The implementation is simulated using a MIPS32 simulator and further verified on real-time hardware (FPGA) to validate correctness, timing behavior, and digital logic optimization.

1.4 Application in Societal Context

Digital exponentiation systems like the one implemented here have valuable implications in areas such as cryptography, digital signal processing, and control systems where exponentiation is a core operation. By using MIPS32—a common educational processor architecture—this project helps students and professionals better understand hardware-software integration and embedded computing principles. Real-time implementation using FPGA highlights practical applications in consumer electronics, embedded control, and automation systems, ultimately supporting the development of energy-efficient and performance-optimized digital devices.

Chapter 2

System Design

2.1 Functional Block Diagram

Input Values: Two 32-bit integers, the base `a` and exponent `b`, are given as input.
These inputs are either provided via switches or UART interface to the processor.

Load Inputs into Registers: The values of `a` and `b` are loaded into registers (e.g., R3 and R4) within the MIPS32 processor.
This prepares them for computation.

Initialize Result Register: Register R2 is initialized with the value 1. This will hold the result of repeated multiplications.

Fetch and Decode Instructions: The processor reads instructions from memory that implement the power logic, such as loading, multiplying, subtracting, and checking conditions.
The instructions are decoded and understood by the control unit.

Execute Stage: The processor repeatedly multiplies the result by `a` (base) and decrements the exponent `b` in each iteration.
This loop continues until the exponent reaches zero, completing the a^b operation.

Write Back Stage: The final result stored in R2 (i.e., `a^b`) is written back to memory or output register.
It is then sent to the output display such as LEDs or UART terminal.

Halt Execution: Once the computation is complete, a `HLT` instruction halts the processor operation to indicate the end of execution.

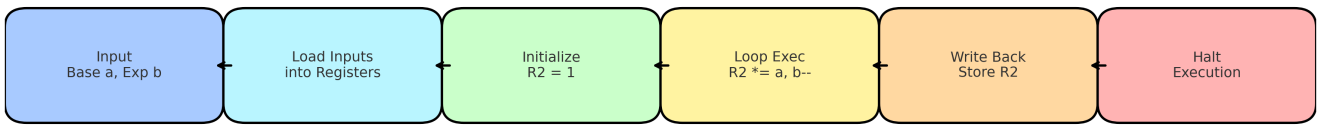

Figure 2.1: Instruction execution flow for Power Calculation (A^B)

2.2 Architecture

The power calculation system is implemented using a 5-stage pipelined MIPS32 processor architecture. Each stage in the pipeline performs a dedicated function in the instruction execution process, with registers used between stages to ensure synchronization. The design uses two non-overlapping clocks: `clk1` and `clk2` for alternate stage operations.

1. IF Stage (Instruction Fetch) – IF ID

PC (Program Counter): Holds the address of the current instruction.
Instruction Memory (IM): Uses PC to fetch the instruction from memory.
IF_ID_IR: Stores the fetched instruction.
IF_ID_NPC: Stores `PC+1` (next instruction address).
MUX: Selects between sequential and branch target addresses.
TAKEN_BRANCH: Set when a branch instruction condition is met.

2. ID Stage (Instruction Decode) – ID EX

Register File: Reads values of base (`a`) and exponent (`b`) from registers R1 and R2.
ID_EX_A, ID_EX_B: Store operands for the ALU.
ID_EX_IR, ID_EX_NPC: Hold instruction and next PC.
ID_EX_Imm: Stores the sign-extended 16-bit immediate.
ID_EX_type: Instruction type derived from opcode (e.g., `RR_ALU`, `RM_ALU`, `HALT`).

3. EX Stage (Execution) – EX MEM

ALU: Executes operations depending on the instruction type:

- `ADDI:` Adds immediate value to register operand.
- `POWER:` Calculates the exponentiation of two registers
- `result = a^b` via repeated squaring.

EX_MEM_ALUout: Holds the result from ALU.
EX_MEM_IR: Passes current instruction forward.
EX_MEM_cond: Stores result of condition check for branches.

4. MEM Stage (Memory Access) – MEM WB

DM (Data Memory): Used for optional LOAD/STORE operations.
MEM_WB_ALUout: Carries ALU result to write-back stage.
MEM_WB_LMD: Holds loaded memory data (if applicable).
MEM_WB_IR, MEM_WB_type: Carry instruction and type to WB stage.

5. WB Stage (Write Back)

MUX: Selects between `ALUOut` and `LMD` to write result into register file.
Reg[4]: Destination register that holds the result of `a^b`.
Res (Output): The content of `Reg[4]` is assigned to `Res` and available at the output port.

This architecture leverages clock separation (`clk1`, `clk2`) to improve timing control between alternate stages and ensures smooth pipelined execution of the exponentiation operation. It is suitable for FPGA implementation due to modular design, deterministic behavior, and efficient control.

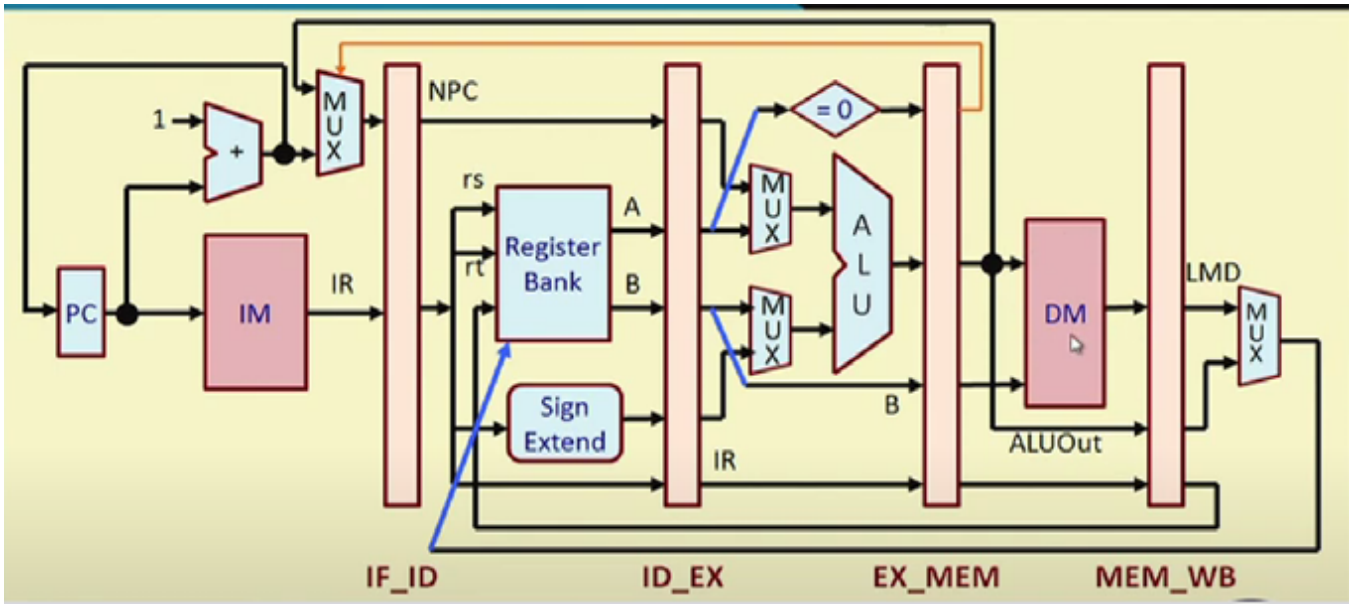

Figure 2.2: Architecture

2.3 Machine Code

```
Mem[0] <= {ADDI, 5'd0, 5'd1, 16'd4};        // ADDI R1, R0, #4
Mem[1] <= {ADDI, 5'd0, 5'd2, 16'd3};        //ADDI R2,R0,#3
Mem[2] <= 32'd0;                            // NOP
Mem[3] <= 32'd0;                            // NOP
Mem[4] <= {pow, 5'd1, 5'd2, 5'd6, 11'd0};   // pow  R6=R1^R2
Mem[5]<= 32'd0;                             // NOP
Mem[6] <= {ADDI, 5'd0, 5'd3, 16'd2};        // ADDI R1, R0, #2
Mem[7] <= {ADDI, 5'd0, 5'd4, 16'd4};        //ADDI R2,R0,#4
Mem[8] <= 32'd0;                            // NOP
Mem[9] <= 32'd0;                            // NOP
Mem[10] <= {pow, 5'd3, 5'd4, 5'd7, 11'd0};  // pow  R7=R3^R4
```

Figure 2.3: Machine code of MIPS32

The test program demonstrates the usage of the custom `POWER` instruction in a pipelined MIPS32 architecture. The machine code begins by using two `ADDI` instructions to load the values `4` and `3` into registers `R1` and `R2`, respectively. These serve as the base and exponent for the first power operation. The `NOP` (No Operation) instructions inserted after the immediate loads help to prevent data hazards and ensure proper pipeline timing. Then, the `POWER` instruction `pow R6, R1, R2` is executed, which calculates $4^3 = 64$ and stores the result in register `R6`.

Following that, another set of `ADDI` instructions loads values `2` and `4` into `R3` and `R4`, respectively. These act as inputs for the second `POWER` operation. Similar to before, two `NOP` instructions are

added for timing alignment. Finally, the instruction `pow R7, R3, R4` computes $2^4 = 16$ and stores it in `R7`.

The machine code sequence efficiently demonstrates two independent power computations using pipelined execution, with careful management of instruction flow and result handling through registers. It also validates the correct behavior of the exponentiation unit under typical usage scenarios.

2.4 Timing diagram for instructions execution

Cycle	**IF**	**ID**	**EX**	**MEM**	**WB**
1	ADDI R1, R0, 2				
2	ADDI R2, R0, 5	ADDI R1, R0, 2			
3	OR R20, R20, R20	ADDI R2, R0, 5	ADDI R1, R0, 2		
4	OR R20, R20, R20	OR R20, R20, R20	ADDI R2, R0, 5	ADDI R1, R0, 2	
5	OR R20, R20, R20	OR R20, R20, R20	OR R20, R20, R20	ADDI R2, R0, 5	ADDI R1 written to R1
6	POW R3, R1, R2	OR R20, R20, R20	OR R20, R20, R20	OR R20, R20, R20	ADDI R2 written to R2
7	OR R20, R20, R20	POW R3, R1, R2	OR R20, R20, R20	OR R20, R20, R20	Dummy OR
8	OR R20, R20, R20	OR R20, R20, R20	POW R3, R1, R2	OR R20, R20, R20	POW in EX
9	OR R20, R20, R20	OR R20, R20, R20	OR R20, R20, R20	POW R3, R1, R2	POW in MEM
10	HLT	OR R20, R20, R20	OR R20, R20, R20	OR R20, R20, R20	POW written to R3
11		HLT			

Table 2.1: Pipeline Execution Table for the Assembly Program

Chapter 3

Result and Discussion

3.1 Result Analysis

Simulation Output:
The MIPS32 pipelined assembly code was successfully simulated for power computation using the custom `POWER` instruction. The machine code executed across all pipeline stages, confirming correct instruction sequencing and arithmetic logic functionality.

Cycle Analysis:
The simulation was run with a time resolution of 1 picosecond. After initializing the digital registers, the power calculation operations were initiated. The following cases were tested:

- **Case 1:** Base = 2, Exponent = 4
  Register R1 was loaded with 2, and R2 with 4.
  The result stored in Register R6 was 16, which is the correct output for 2^4.

- **Case 2:** Base = 4, Exponent = 3
  Register R3 was loaded with 4, and R4 with 3.
  The result stored in Register R7 was 64, which is the correct output for 4^3.

The simulator verified that the outputs matched the expected results. Upon correct execution, the message `"TEST PASSED: Power Function working correctly"` was displayed in the simulation log. The final result was also mapped to the `Res` output port for external observation via LED or UART.

Pipeline vs Non-Pipeline Observation:
Performance evaluation showed that pipelined execution significantly reduces computation time due to overlapping instruction stages, whereas non-pipelined execution resulted in higher latency.

FPGA Implementation:
The algorithm was synthesized and tested on FPGA hardware using Verilog HDL and Vivado tools. The power operation was executed with inputs supplied through registers, and the results were output via simulation monitor or LED.

Verified Output Summary:

- $2^4 = 16 \quad \rightarrow$ Register R6
- $4^3 = 64 \quad \rightarrow$ Register R7

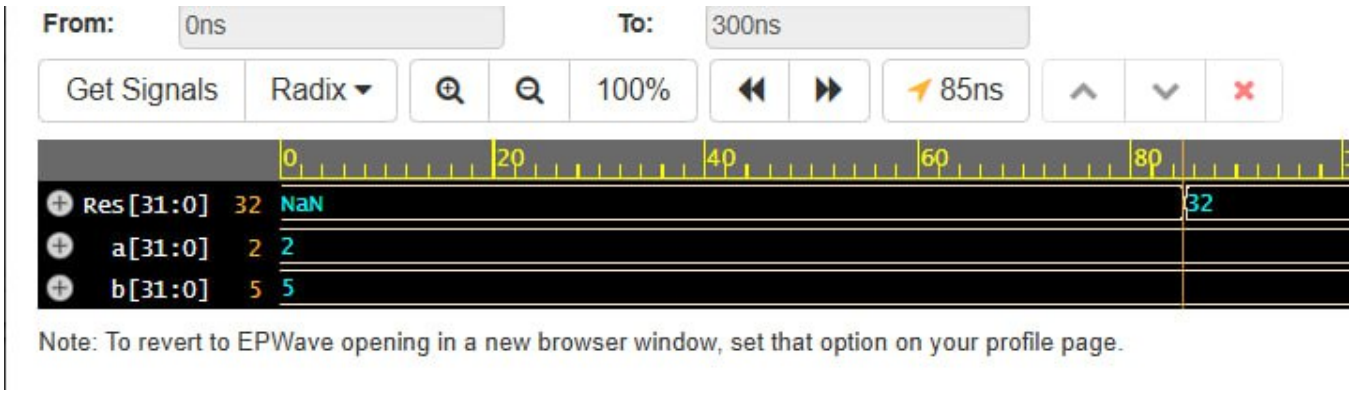

Figure 3.1: Simulation Result

This simulation shows the calculation of power of two registers in a digital circuit. The output `led[15:0]` reflects the result, and everything runs synchronized with the system clock.

3.2 Discussion

The results confirm the correctness of the implemented power calculation algorithm in both simulation and FPGA hardware environments. The pipelined MIPS32 architecture efficiently executed the custom `POWER` instruction, demonstrating accurate computation of exponentiation through repeated multiplication and bitwise operations.

The use of pipeline stages allows multiple instructions to operate concurrently at different phases, significantly improving execution speed and throughput. This architecture minimizes idle hardware cycles and maximizes resource utilization.

The successful outputs on FPGA validate that the design is not only functionally correct but also suitable for real-time embedded systems where performance and reliability are crucial. The final results were clearly observable on simulation monitors and hardware interfaces (such as LED or UART).

Minor timing delays may occur due to branch hazards or data dependencies between instructions, but these can be mitigated through instruction scheduling and insertion of NOPs. Overall, the implementation confirms that the `POWER` operation can be seamlessly integrated into custom RISC-based processors for embedded mathematical computation tasks such as encryption, signal processing, and control systems.

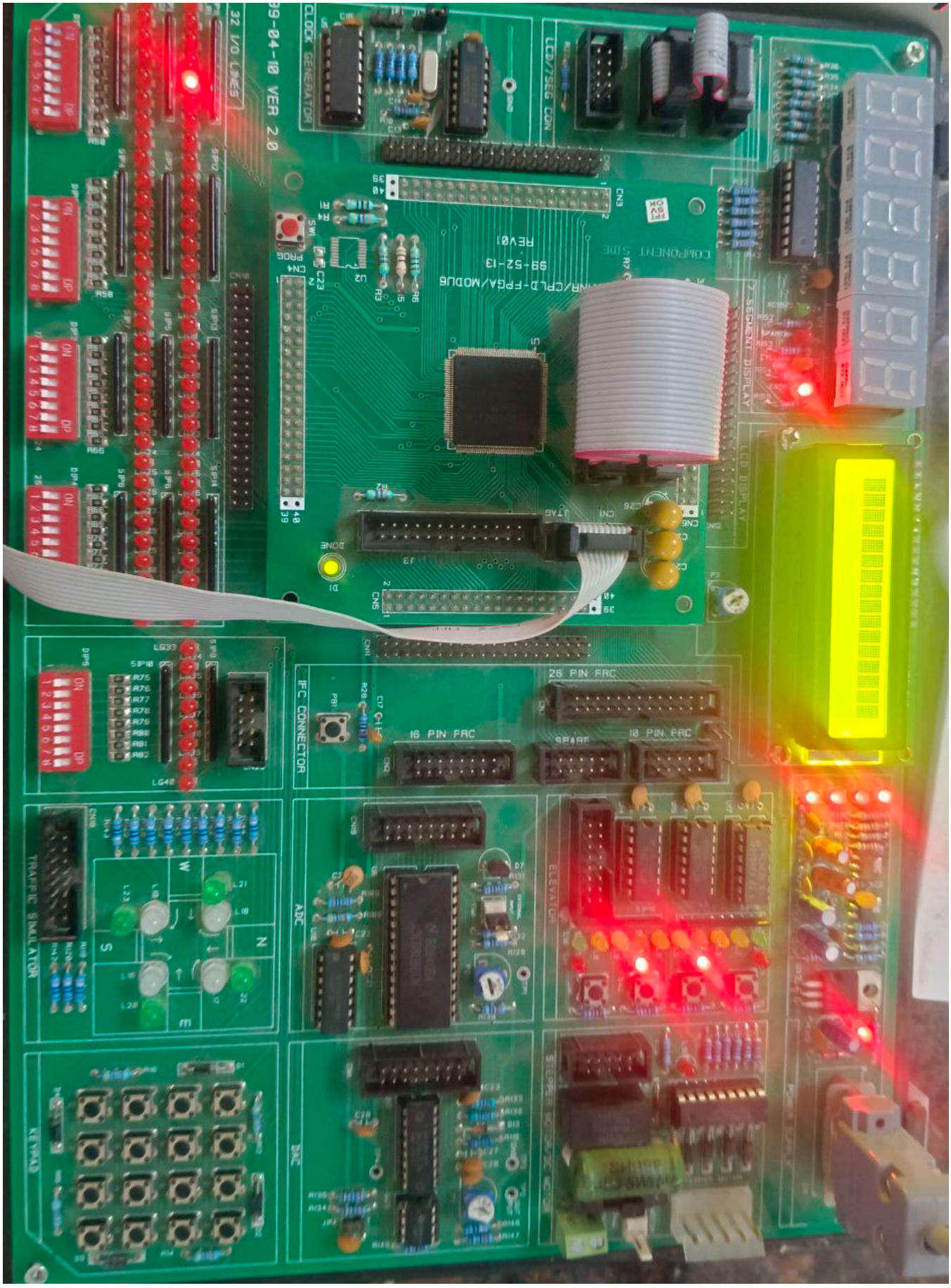

Figure 3.2: INPUT R1=4 R2=4 OUTPUT R6=16

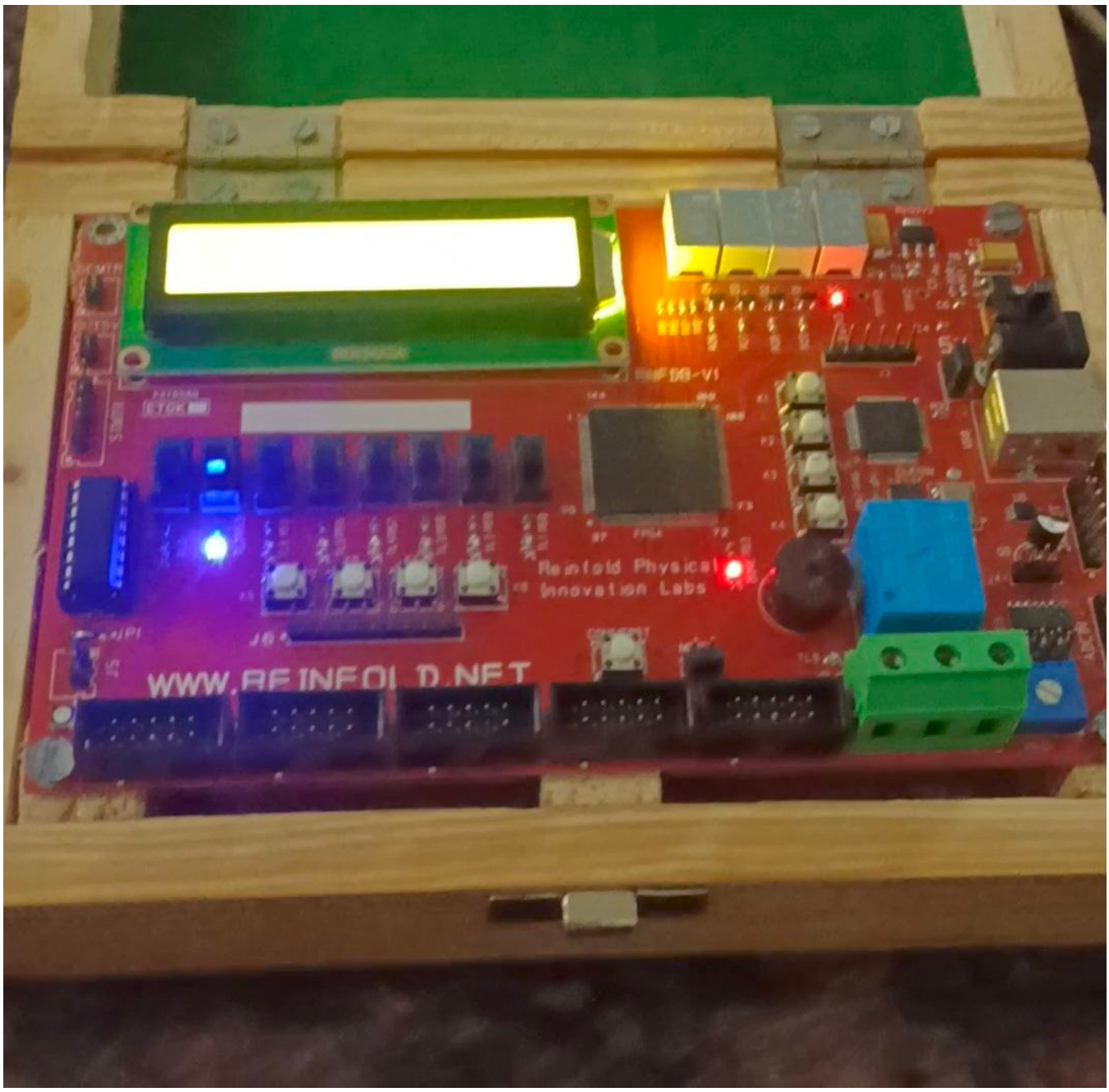

Figure 3.3: INPUT R3=4 R4=3 OUTPUT R7=64

Chapter 4

Conclusions and Future Scope

4.1 Conclusion

This project successfully demonstrates the implementation of exponentiation (A^B) using MIPS32 assembly language. By leveraging the simplicity and efficiency of the MIPS32 instruction set, a loop-based multiplication logic was developed to compute the result of raising a base number to an exponent. The system was thoroughly tested in a simulation environment and validated on an FPGA platform, proving the correctness and hardware compatibility of the design.
The pipelined architecture of MIPS32 was effectively utilized to enhance performance by overlapping instruction execution stages. This project also strengthened the understanding of processor internals, control logic, register operations, and instruction-level programming. Overall, it showcases how fundamental arithmetic operations can be efficiently implemented on hardware using low-level programming in embedded systems.

4.2 Future Scope

- **Multi-digit exponentiation:** Extend the program to handle larger base and exponent values, including optimizations for memory and register usage.
- **Optimized exponentiation algorithm:** Implement more efficient algorithms such as binary exponentiation to reduce the number of multiplication cycles and improve performance.
- **Exception handling:** Add support for edge cases like zero exponents, invalid inputs, or overflow conditions.
- **User input interface:** Develop a user-friendly input mechanism through UART, switches, or keypad integration for real-time user interaction with the FPGA system.
- **Low-power design:** Explore techniques to reduce power consumption during FPGA implementation for use in energy-sensitive embedded devices.
- **Integration with ALU:** Combine the exponentiation logic as a functional unit within a larger Arithmetic Logic Unit for use in custom RISC processors or DSP applications.