CPU on FPGA

Design and Implementation of a
16-Bit Computer System

Report Author:
Rishti R Kulkarni
MARVEL Project Student
ECE 27, UVCE

Under the Guidance of:
Anish Krishnakumar
Silicon Design Engineer II
AMD

February 2026

MARVEL, UVCE

Acknowledgement and Gratitude

I would like to express my sincere gratitude to the **UVCEGA** and **MARVEL**, our student-run laboratory, for making this project possible and for providing a platform that fosters innovation and technical growth.

I am deeply grateful to our Alumni Coordinator, **Adrian P Isaac**, for this invaluable opportunity and for the continued support.

I also express my gratitude to **Shimon Schocken** and **Noam Nisan**, the creators of the *NAND2Tetris* course. Their exceptional framework and course materials provided the fundamental roadmap for this project, making complex architectural concepts accessible and helping me build a solid understanding of the system I aimed to create.

A special note of thanks goes to my project mentor, **Anish Krishnakumar**. He has been an amazing guide throughout this journey. His willingness to share his immense knowledge, coupled with his patient mentorship, helped me navigate the steep learning curve of this project and significantly deepened my understanding of the subject.

I also wish to acknowledge and thank my peers, **A V Sohan Aiyappa** and **Virajit G P**. They have been incredibly supportive throughout this process, and witnessing their exceptional work and dedication has been a constant source of motivation for me.

Finally, I owe a special thanks to my friend, **Priyamvada K C**. It was her encouragement that led me to pursue this opportunity in the first place, and I am deeply grateful for the push that started this journey.

CONTENTS

1. Introduction

This project focuses on the design and implementation of a complete CPU on an FPGA by following the ***NAND2Tetris*** course framework, which emphasizes building a computer system from first principles. The work began at the most fundamental level by constructing basic logic gates using the NAND gate as the universal building block. These gates were then composed to form combinational components such as multiplexers, demultiplexers, and adders, which were further extended to design the Arithmetic Logic Unit (ALU).

Subsequently, sequential elements including single-bit registers, multi-bit registers, RAM modules of various sizes, and a program counter were implemented to enable stateful computation. These components were systematically integrated to construct higher-level modules, ultimately resulting in a functional memory system, CPU, and a simple computer architecture.

The entire design was first developed using the hardware description language specified in the NAND2Tetris course and later reimplemented in **Verilog**, followed by synthesis and implementation on an FPGA using **Vivado**. This transition provided valuable insights into HDL syntax, hardware modeling, and how high-level descriptions are synthesized into physical hardware structures. Throughout the process, extensive debugging and verification were required to resolve functional mismatches and integration issues, contributing significantly to practical problem-solving skills. Overall, this project offered a deep understanding of digital circuits, computer architecture, and FPGA-based implementation by constructing a computing system incrementally from the gate level upward.

2. From Instructions to Execution: CPU Overview

The implemented system is a **16-bit computer architecture** designed using the NAND2Tetris methodology, where computation is built incrementally from fundamental hardware components. The CPU processes **16-bit instructions**, operates on **16-bit data operands**, and generates **16-bit results**, ensuring uniform data width across all stages of execution.

The overall system consists of an **instruction memory**, a **data memory**, and a **central processing unit (CPU)**. The instruction memory stores the binary representation of the program and supplies instructions sequentially to the CPU. The data memory supports read and write operations, enabling the CPU to access and store intermediate as well as final computation results. Instructions specify memory addresses and control signals that determine the flow of data between the CPU and memory.

The CPU is responsible for fetching instructions, decoding them, generating the required control signals, and executing the specified operations. Internally, the CPU contains an **Arithmetic Logic Unit (ALU)** for performing arithmetic and logical computations, along with dedicated registers for operand storage and intermediate results. A **Program Counter (PC)** is used to track the address of the current instruction and to control program flow through sequential execution and branching operations. Load and control signals regulate updates to the registers and ensure correct synchronization of data movement within the CPU.

Instruction execution follows a structured sequence: an instruction is fetched, decoded to determine the required operation, operands are selected and processed by the ALU, and the resulting data is written back to a register or memory location as dictated by the instruction. Through this process, the CPU translates simple binary instructions into meaningful computational behavior.

To support flexibility during testing and demonstration, the system allows execution in two modes: a **manual mode**, where instructions are supplied directly to the CPU one at a time, and an **automatic mode**, where instructions are fetched sequentially from instruction memory. This dual-mode operation enables both step-by-step observation of instruction execution and continuous program execution, without altering the underlying CPU architecture. The manual execution mode was introduced as an FPGA-specific extension, as the original NAND2Tetris architecture assumes instruction execution solely from instruction memory.

The complete computer system is formed by integrating the CPU with instruction and data memories into a single top-level module. This modular organization reflects the bottom-up design philosophy of the project, in which complex system behavior is achieved through the systematic composition of simpler, well-defined hardware blocks.

2.1 Fundamental Logic Components

The CPU and computer were developed incrementally following the NAND2Tetris design philosophy, starting from elementary logic gates and progressing toward a complete 16-bit computing system. The design evolves through the implementation of arithmetic units, sequential memory components, and system-level integration, illustrating how complex computational behavior emerges from the systematic composition of simpler hardware blocks. This bottom-up approach provides a cohesive understanding of digital logic, computer architecture, and hardware abstraction.

2.1.1 Logic Gates (Project 1)

The first phase of the project focused on implementing fundamental logic gates using the NAND gate as the only primitive component, in accordance with the NAND2Tetris design philosophy. This approach demonstrates that all logical operations can be constructed from a single universal gate, reinforcing first-principles understanding of digital logic.

The design of logic gates followed a **progressive, hierarchical approach**, beginning with the NAND gate as the sole primitive. The NOT gate was first constructed using NAND gates, after which higher-level gates such as *AND, OR, and XOR* were implemented by reusing previously built components. Each subsequent block was developed using only the modules designed in earlier stages, reinforcing the concept of building complex logic through systematic composition of simpler units. This incremental construction provided practical insight into Boolean logic, modular hardware design, and the reusability of verified components.

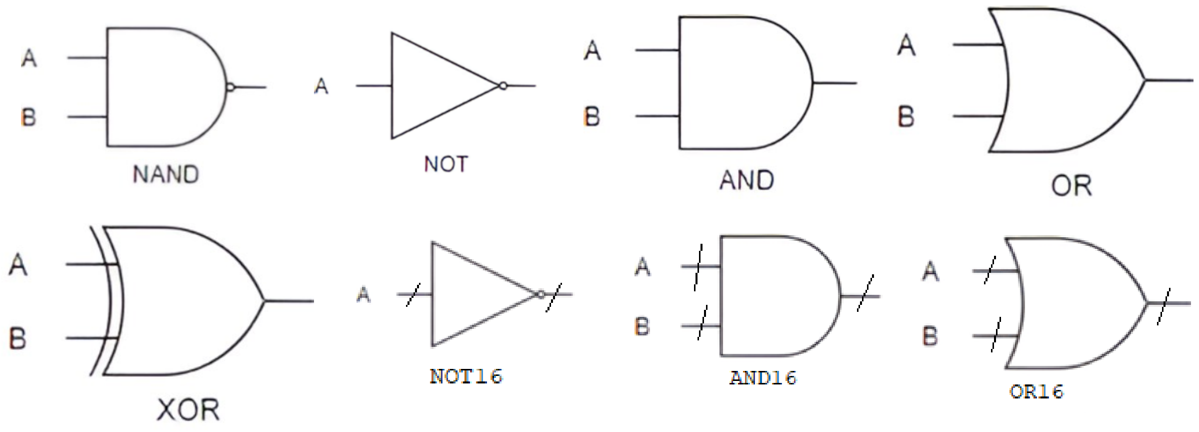


Image 1: Basic Logic Gates

Once single-bit logic gates were verified, they were extended hierarchically to support multi-bit operations. Bitwise gates such as *Not16, And16,* and *Or16* were constructed by replicating the single-bit designs across 16-bit data paths. This modular approach ensured consistency while enabling scalable data-width expansion required for CPU-level computation.

In addition to basic logic gates, *multiplexers and demultiplexers* were implemented to enable controlled data routing. These components allow selection and distribution of data based on control signals and form a critical foundation for register loading, ALU input selection, and memory interfacing in later stages of the CPU design.

Due to the hierarchical and repetitive nature of these components, representative gates and routing elements are illustrated, while the remaining modules follow analogous construction principles.

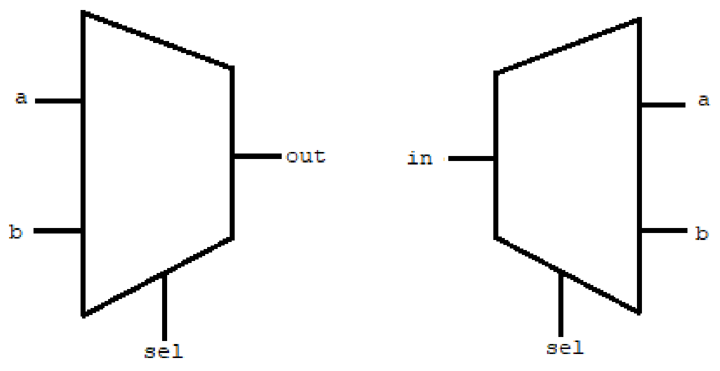


Image 2: Multiplexer and Demultiplexer

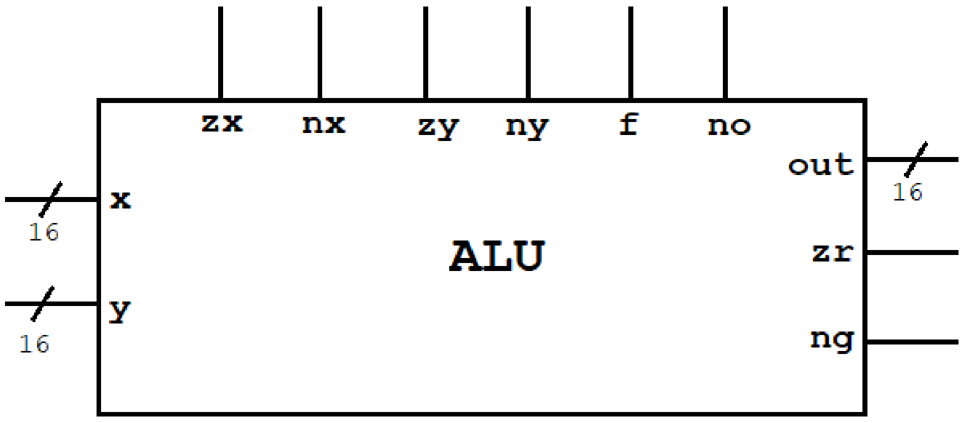
2.1.2 Adders and Arithmetic Logic Unit (Project 2)

Following the implementation of fundamental logic gates, the next stage of the project focused on building arithmetic components and integrating them into a functional **Arithmetic Logic Unit (ALU)**. This phase bridges basic combinational logic with meaningful computation and forms the computational core of the CPU.

The arithmetic foundation was established by implementing a **half adder** and **full adder**, which were then hierarchically composed to create a **16-bit ripple-carry adder**. This adder enables multi-bit arithmetic operations required for instruction execution. In addition, a **16-bit incrementor** was implemented to support operations such as program counter updates and address manipulation. These components demonstrate how simple arithmetic units can be scaled to operate on wider data paths through systematic composition.

At the core of this stage lies the **16-bit Arithmetic Logic Unit (ALU)**, which performs arithmetic and logical operations based on control signals derived from the instruction being executed. From a computer architecture perspective, the ALU is a critical component, as it is responsible for executing computations specified by instructions, including arithmetic operations, logical comparisons, and data transformations. The flexibility of the ALU directly determines the expressive power of the instruction set supported by the CPU.

The ALU was implemented in **Verilog** following the NAND2Tetris control-bit specification. Two 16-bit inputs are first conditionally modified through input control signals that allow zeroing and negation. Based on a function-select control signal, the ALU then performs either an addition or a bitwise AND operation. The resulting output can be optionally negated before being produced as the final result. In addition to the computed output, the ALU generates **status flags**, including a zero flag and a negative flag, which are essential for conditional branching and decision-making at the CPU level.



Block Diagram 1: ALU

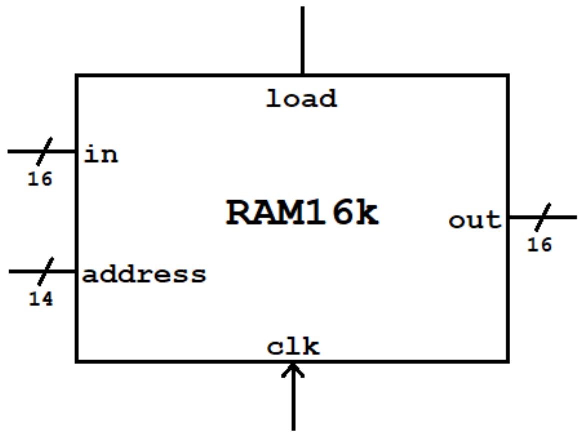
The Verilog implementation models the ALU as a purely combinational module, using intermediate signals to clearly represent each transformation stage. This structured approach improved readability, simplified debugging, and provided insight into how high-level behavioral descriptions are synthesized into combinational hardware. Through this process, the ALU evolved from a conceptual logic block into a central execution unit capable of supporting instruction-level computation within the CPU.

2.1.3 Sequential Components and Memory Elements (Project 3)

The introduction of **sequential components and memory elements** marked a critical transition in the project from purely combinational logic to stateful computation. These components enable the system to store data, retain intermediate results, and execute programs sequentially, which are essential characteristics of a functional computer.

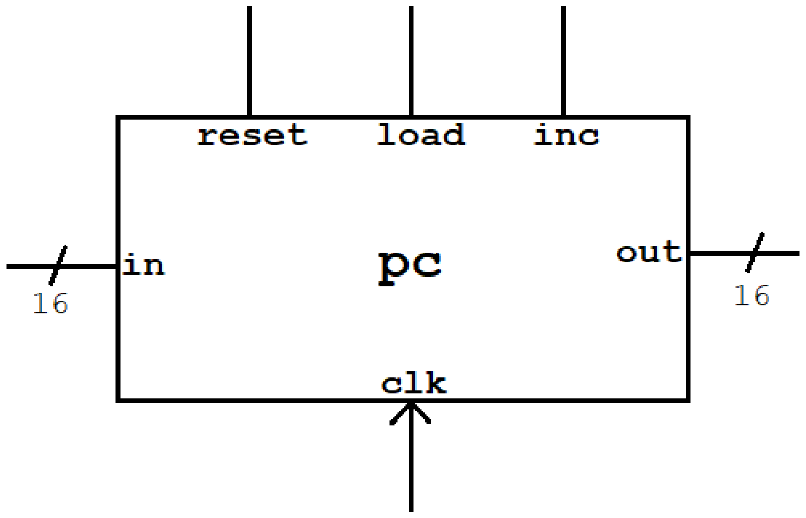
The foundational storage element implemented was a **single-bit register**, which captures and holds a value based on a load control signal. This design was extended hierarchically to form a **16-bit register**, enabling storage of word-sized data used throughout the CPU. These registers serve as the basic building blocks for larger memory structures and internal CPU storage.

Using the register as the core storage unit, a hierarchy of **Random Access Memory (RAM)** modules was constructed, including RAM8, RAM64, RAM512, RAM4K, and finally RAM16K. Each larger memory module was built by composing smaller RAM blocks along with multiplexing and demultiplexing logic for address decoding and data routing. The RAM16K module serves as the primary **data memory** of the system, supporting read and write operations addressed by the CPU during instruction execution.



Block Diagram 2: RAM16k

In addition to data memory, a **Program Counter (PC)** was implemented to manage the flow of instruction execution. The PC holds the address of the current instruction and supports increment, load, and reset operations, enabling sequential execution as well as branching and control flow changes. The PC plays a central role in fetching instructions from instruction memory and ensuring correct program progression.



Block Diagram 3: Program Counter

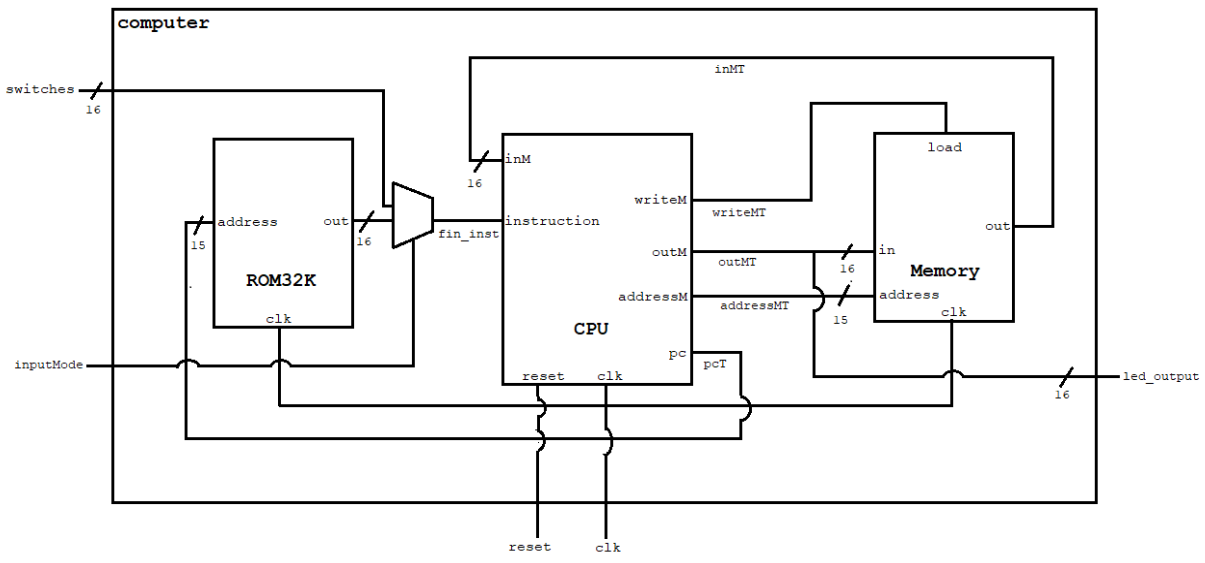
Together, the registers, RAM modules, and program counter provide the necessary state-holding capability for the computer. These sequential elements form the backbone of both the data memory system and instruction sequencing mechanism, enabling the CPU to execute programs reliably and deterministically.

2.1.4 System-Level Integration: Memory, CPU, and Computer (Project 5)

At this stage of the project, the previously designed components were integrated to form a complete **16-bit computer system**, following the structural organization defined in the NAND2Tetris framework. This level focuses on system-level composition, where memory, the CPU, and control logic interact to enable instruction execution and data storage.

A unified **memory module** was implemented to provide a single interface for multiple memory-mapped components. Address decoding logic is used to distinguish between different address ranges and to route read and write operations to the appropriate memory block. The module internally instantiates a **RAM16K** unit to serve as data memory and a separate screen memory block, while also supporting a keyboard-mapped input address. Load signals are selectively generated based on the decoded address to ensure that write operations affect only the intended memory region. Although this memory module closely follows the NAND2Tetris computer structure, it was implemented primarily for architectural understanding and was not directly used in the final FPGA design, where instruction memory and data memory are maintained as separate modules.

The **CPU module** integrates the A register, D register, ALU, and Program Counter to decode instructions and perform computation. Instructions fetched from instruction memory are decoded to generate control signals that determine ALU operations, register updates, and memory access. The A register serves both as an operand register and as the source of memory addresses, while the Program Counter manages instruction sequencing. The CPU is capable of writing computed results to memory and reading data from memory as required by the instruction semantics, thereby acting as the central execution unit of the computer.



Block Diagram 4: Computer Top Module

At the highest level, a **top-level computer module** instantiates and interconnects the instruction memory, CPU, and data memory to form a complete 16-bit computing system. In the standard NAND2Tetris architecture, this module exposes only a reset signal to control execution flow. In the final FPGA implementation, additional inputs such as manual instruction input, initialization control, and clock handling were introduced to support enhanced testing and demonstration. The outputs of this module reflect the results of computation, enabling direct observation of system behavior. This top-level integration represents the culmination of the bottom-up design approach, where simple hardware blocks are composed to realize a functional computer.

2.2 Hack Instruction Format and Operation

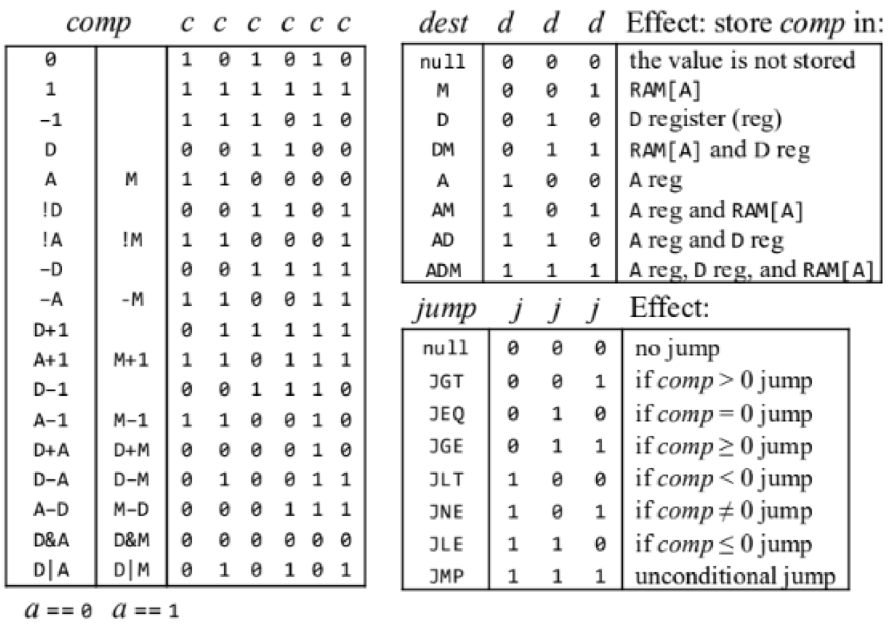
The Hack computer uses a **16-bit instruction format** designed to be simple yet expressive enough to support computation, memory access, and control flow. Each instruction belongs to one of two types: **A-instructions** and **C-instructions**. Together, these instructions define how data is loaded, processed, stored, and how program execution progresses.

A-Instruction: It is used to load a 15-bit value into the A register.

- **Format: 0vvv vvvv vvvv vvvv** (vv ... v = 15-bit value to be loaded into A register)
- The most significant bit (MSB) is 0, identifying it as an A-instruction. The remaining 15 bits represent a value or address.
- **Purpose and Usage:** Load a constant into the A register, specify a memory address for data access, provide an operand for subsequent computation
- **Once loaded, the A register can:** Act as an input to the ALU, point to a data memory location, be used by the Program Counter for jumps

C-Instruction: A C-instruction performs computation, data storage, and control flow in a single instruction.

- **Format: 111 a c1 c2 c3 c4 c5 c6 d1 d2 d3 j1 j2 j3**
- The leading 111 identifies it as a C-instruction. It is divided into three logical fields, Computation Field (**c1 c2 c3 c4 c5 c6**), Destination Field (**d1 d2 d3**) and Jump Field (**j1 j2 j3**). Bit 'a' selects between the A register and memory (M).
- Performs arithmetic and logical computations using the ALU
- Selects operands from the **D register** and either the **A register or memory (M)**
- Stores computation results in one or more destinations (**A**, **D**, and/or **M**)
- Updates memory by enabling write operations to the addressed location
- Controls **program flow** through conditional and unconditional jump instructions
- Combines computation, storage, and control decisions into a **single instruction**
- Forms the primary mechanism through which the CPU executes meaningful operations



comp		c	c	c	c	c	c
0		1	0	1	0	1	0
1		1	1	1	1	1	1
-1		1	1	1	0	1	0
D		0	0	1	1	0	0
A	M	1	1	0	0	0	0
!D		0	0	1	1	0	1
!A	!M	1	1	0	0	0	1
-D		0	0	1	1	1	1
-A	-M	1	1	0	0	1	1
D+1		0	1	1	1	1	1
A+1	M+1	1	1	0	1	1	1
D-1		0	0	1	1	1	0
A-1	M-1	1	1	0	0	1	0
D+A	D+M	0	0	0	0	1	0
D-A	D-M	0	1	0	0	1	1
A-D	M-D	0	0	0	1	1	1
D&A	D&M	0	0	0	0	0	0
D|A	D|M	0	1	0	1	0	1
a == 0	*a* == 1						

dest	*d*	*d*	*d*	Effect: store *comp* in:
null	0	0	0	the value is not stored
M	0	0	1	RAM[A]
D	0	1	0	D register (reg)
DM	0	1	1	RAM[A] and D reg
A	1	0	0	A reg
AM	1	0	1	A reg and RAM[A]
AD	1	1	0	A reg and D reg
ADM	1	1	1	A reg, D reg, and RAM[A]

jump	*j*	*j*	*j*	Effect:
null	0	0	0	no jump
JGT	0	0	1	if *comp* > 0 jump
JEQ	0	1	0	if *comp* = 0 jump
JGE	0	1	1	if *comp* ≥ 0 jump
JLT	1	0	0	if *comp* < 0 jump
JNE	1	0	1	if *comp* ≠ 0 jump
JLE	1	1	0	if *comp* ≤ 0 jump
JMP	1	1	1	unconditional jump

Image 3: c-Instruction Fields

How Instructions Work Together

Hack instructions are typically used in pairs or sequences:

1. An **A-instruction** loads a value or address
2. A **C-instruction** uses that value for computation, storage, or branching

Example flow:

- Load an address into A
- Perform computation using D and memory
- Store the result
- Optionally jump based on the result

The Hack instruction set is minimal yet complete, consisting of only two instruction types, which makes it both simple to understand and powerful enough to support meaningful computation. This simplicity encourages a clear understanding of fundamental computer architecture concepts such as data paths, control signal generation, ALU behavior, and program flow. Each instruction maps cleanly to hardware, with control bits directly driving the ALU, registers, and Program Counter without the need for complex decoding logic. As a result, the Hack instruction set is particularly well suited for building a CPU from first principles, allowing the relationship between software instructions and hardware execution to be observed and understood directly.

3. FPGA Implementation

The complete 16-bit computer was implemented and verified on the **Basys 3 FPGA (Artix-7)** by mapping the architectural components to on-board inputs, outputs, and clock resources. The FPGA implementation was designed to support two distinct modes of operation, **manual mode** and **automatic mode**, to facilitate both step-by-step observation and continuous execution of instructions.

3.1 Implementation and Operating Modes

3.1.1 Operating Modes Overview

Two execution modes were implemented to balance observability and automation:

- Manual Mode (default)
  - Active when the mode button has not been toggled
  - Instructions are supplied directly through the FPGA slide switches
  - A push button acts as a **manual clock**, advancing execution one step at a time
- Automatic Mode
  - Enabled by pressing the mode button once
  - Instructions are fetched sequentially from instruction memory (ROM)
  - Execution proceeds continuously using the FPGA's system clock

The mode selection is latched, allowing the system to remain in automatic mode without requiring the button to be held.

3.1.2 Button Debouncing and Mode Latching

Mechanical push buttons on the FPGA generate spurious transitions due to contact bounce, which can lead to unintended multiple activations if not handled properly. In this implementation, **debouncing was applied selectively**, only to inputs where the **number of button presses directly affects system behavior**.

The debouncer samples the button input using the FPGA system clock and verifies that the input remains stable for a predefined duration before accepting it as valid. A counter is incremented while the sampled input remains unchanged, and the output is updated only after the debounce interval is completed. Any transient change resets the counter, ensuring reliable press detection.

In its default configuration, the debouncer outputs a cleaned version of the button signal and was used for the **step button**, where each press must generate exactly one clock pulse. The module also supports a **toggle (latch) mode**, enabled through a configurable parameter, in which a rising edge on the debounced signal toggles an internal state register. This mode was used for the **mode button** by overriding the parameter during instantiation, allowing a single press to switch between manual and automatic modes while retaining the selected state after the button is released.

- The **mode button** was debounced and configured in a **toggle (latch) mode**, allowing a single press to switch between manual and automatic execution modes. Once toggled, the selected mode is retained until the button is pressed again.
- The **step button**, which functions as a manual clock in manual mode, was debounced in **normal mode** to ensure that each physical press generates exactly one clean clock pulse.

Other control inputs that do not depend on precise press counts were not debounced, simplifying the design while maintaining reliable operation where it matters most. This selective debouncing approach ensured stable mode control and accurate manual stepping without introducing unnecessary hardware complexity.

3.1.3 Clock Generation, Selection, and Distribution

The system utilizes a hybrid clocking architecture to support both high-speed automatic execution and precise manual debugging.

1. **Clock Sources and Generation**
   - **Automatic Clock (50 MHz):** The system accepts the native 100 MHz FPGA oscillator input. To resolve setup time violations on the critical path, this input is routed through an internal frequency divider to generate a stable 50 MHz clock signal. This reduced-frequency clock drives the automatic execution mode.
   - **Manual Step Clock:** A debounced push-button signal serves as a manual clock source, allowing the user to advance the program execution one cycle at a time.
2. **Clock Selection (Multiplexing):** A 2:1 Clock Multiplexer dynamically selects the active clock source based on the user-defined operation mode:
   - Manual Mode: Selects the debounced Step Clock.
   - Automatic Mode: Selects the generated 50 MHz System Clock.
3. **Global Distribution:** The output of the multiplexer (cpu_clk) is distributed globally to drive the CPU, Instruction Memory (ROM), and Data Memory (RAM). This unified clock distribution ensures that all synchronous components operate in lockstep, preserving data integrity regardless of the selected speed or mode.

3.1.4 Instruction Source Selection

Instruction input is also mode-dependent:

- **Manual Mode:** Instructions are provided directly via the 16-slide switches
- **Automatic Mode:** Instructions are fetched from a 32K instruction memory (ROM) using the Program Counter

A multiplexer selects between these two instruction sources based on the active mode, enabling seamless switching without modifying the CPU architecture.

3.1.5 LED Output Handling and Result Visualization

To ensure meaningful output visibility across both operational modes, the following display strategies were adopted:

- **Automatic Mode:**
    - The result written to a fixed memory location (RAM[2]) is captured using a synchronous latch.
    - This latched value is routed directly to the LEDs, providing a stable final output even after the program completes.
- **Manual Mode:**
    - The LEDs display the 16-bit data currently addressed by the A-register.
    - Since the address only changes when the step button is pressed, the internal memory states remain observable for debugging.

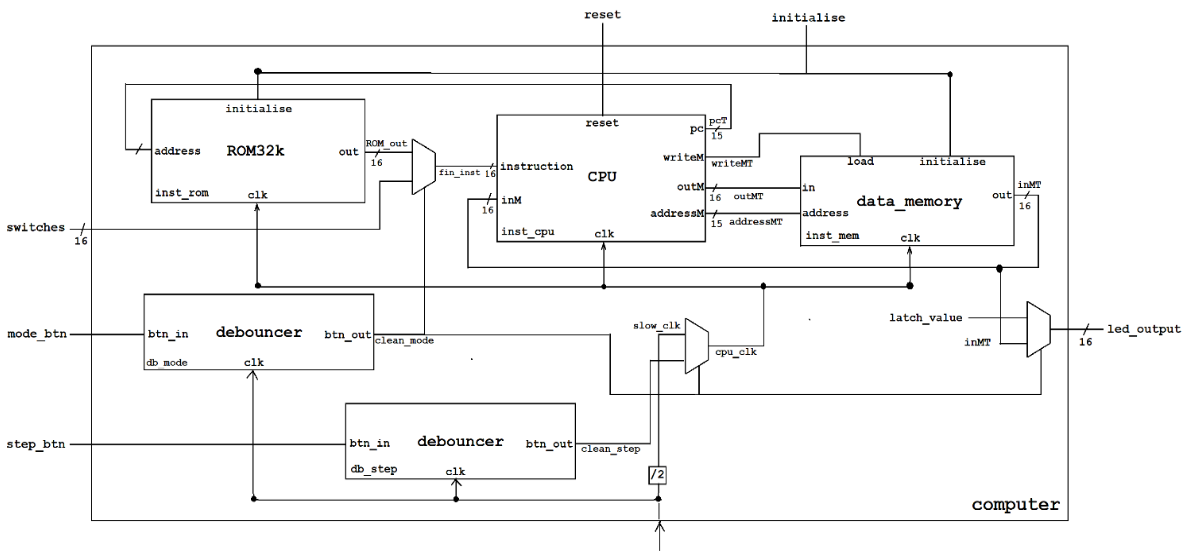
A multiplexer selects between these two data sources based on the active mode signal.

Design Rationale

This display logic was implemented to resolve a specific observation during the simulation of the multiplication program. To terminate execution, the program utilizes an **infinite loop**. While this effectively stops the algorithm, the Program Counter continues to oscillate between the jump instructions. Consequently, the address bus no longer points to the result location, causing the LEDs to display irrelevant data or no data in the tested case

To solve this, the design uses **RAM[2]** as a persistent display buffer. By continuously writing or "copying" the final result to this specific address within the program loop, the output remains visible on the LEDs while the original data remains safely stored in its intended memory location. This ensures the hardware provides a constant, readable output regardless of the background instruction cycling.

3.1.6 Top-Level FPGA Block Diagram and Mode-Dependent Data Flow



Block Diagram 5: Top level Computer FPGA Implementation

The complete FPGA implementation of the 16-bit computer is illustrated in the block diagram above, showing the integration of instruction memory, CPU, data memory, and FPGA-specific control logic. In addition to the core architectural blocks, the design incorporates mode selection, clock control, and output handling to support both manual and automatic execution.

The block diagram highlights how debounced button inputs control mode selection and manual stepping, how the selected clock drives the CPU and memory modules, and how computation results are routed to the LED outputs. Together, these elements demonstrate how architectural components are adapted for practical FPGA-based operation.

3.1.6.1 Mode Selection

Mode-dependent signal selection is implemented using **multiplexers realized through Verilog ternary operators**. These multiplexers are used to dynamically select the clock source, instruction source, and LED output based on the active mode. This approach simplifies the control logic while preserving clear separation between manual and automatic execution paths.

The effect of mode selection on key system signals is summarized in the table below:

Sources for	in Manual Mode (Select line = 0)	in Auto Mode (Select line = 1)
Clock	clean_step (Manual Clock)	clk (System Clock)
Instructions	switches (Slide Switches)	ROM_out (From instruction memory)
LED Output	inMT (The value stored at address A)	latch_value (The value stored at RAM[2])

3.1.6.2 IO Table

Direction	Name	Width	Description
Input	switches	16	Provides 16-bit manual instruction input in manual mode
Input	mode_btn	1	Toggles between manual and automatic execution modes
Input	step_btn	1	Acts as a manual clock pulse to advance execution by one-step
Input	reset	1	Resets the CPU and program counter to restart execution
Input	initialize	1	Initializes instruction and data memory to a known state
Input	clk	1	FPGA system clock used for automatic execution and internal synchronization
Output	led_output	16	Displays computation result or memory contents based on mode

3.2 Testing and Results

To ensure the functional integrity of the 16-bit computer, a comprehensive testing strategy was adopted, divided into two distinct phases: pre-silicon verification via testbench simulation and post-silicon validation on the FPGA hardware. This dual-phase approach allowed for the early detection of logic errors and ensured that the physical implementation behaved deterministically under real-world timing constraints.

3.2.1 Phase I: Testbench Simulation

Before generating the bitstream for the FPGA, the design was subjected to extensive behavioral simulation using Verilog testbenches. The primary objective was to verify the interaction between the CPU, Memory, and Input/Output subsystems without the interference of physical noise or switch bouncing.

The simulation strategy focused on mimicking the physical user interface within the software environment:

- **Mode Switching Simulation:** The testbench was designed to toggle the modeBtn signal effectively, verifying the system's ability to transition between the system clock (Auto Mode) and the manual step clock (Manual Mode).
- **Manual Input Emulation:** To validate Manual Mode, the testbench programmatically injected binary sequences into the switches input while simultaneously pulsing the stepBtn signal. This confirmed that the CPU could fetch instructions from the switches rather than the ROM when the mode was set to Manual.
- **Waveform Verification:** All internal signals, including the Program Counter (pc), Instruction Register, and ALU outputs, were monitored via waveform analysis. The simulation was considered successful only after verifying that the CPU correctly

latched the multiplication result in Auto Mode and executed individual instructions in Manual Mode with cycle-accurate precision.

3.2.2 Phase II: FPGA Hardware Verification

Following successful simulation, the design was synthesized and deployed onto the FPGA, translating the RTL logic into a device-specific bitstream using Vivado's implementation tools while meeting all physical constraints and pin mappings. The hardware verification process validated three core functionalities: Automatic Execution, Manual Instruction Processing, and Memory Access.

The **Basys 3 Artix-7** board was used, where live computational workloads were applied to evaluate the stability of the Control Unit and ALU under real-time conditions. By toggling between the 50 MHz system clock and a debounced manual pulse, the internal state transitions were verified against simulation results, confirming reliable instruction fetching and data write-back without signal degradation or logic errors.

Auto Mode Verification (Multiplication)

The initial test validated the system's ability to execute a pre-loaded assembly program stored in ROM.

- **Procedure:** The system was toggled to **Auto Mode**.
- **Observation:** The CPU executed the multiplication program (3 * 4) at a frequency of 50 MHz, derived from the on-board 100 MHz oscillator via a custom clock divider.
- **Result:** The LEDs correctly displayed the latched result **12** (0000 0000 0000 1100). The display remained stable even after the program entered its infinite loop, confirming the stability of the hybrid LED latch logic.

Manual Mode & Memory Access Verification

The most extensive testing occurred in **Manual Mode**, where the CPU was stepped through individual clock cycles. This phase utilized the LEDs in "Monitor Mode," where they display the data stored at the address currently held in the A-Register (RAM[A]).

1. **Read/Write Integrity Test:** A specific sequence of operations was performed to verify that data could be written to and read from memory correctly. Since this was done in Manual Mode, the process involved physically setting the address on the input switches and latching it into the A-Register first. Once the address was locked in, the data was set to the value on the switches and triggered the write signal.

   The test deliberately targeted both low addresses (Addresses 5 & 6) and a much higher address (Address 1000). This specific jump was important for checking the hardware connections. By forcing the system to access Address 1000, verified that the upper bits of the address bus were fully connected and functional.

Step	Operation	Math	Switch Code (Binary)	LED Output (Monitor)
1	Load 10	@10	000 0 000000 001 010	0000 0000 0000 0000 (RAM[10])
2	Move to D	D=A	111 0 110000 010 000	0000 0000 0000 0000
3	Load 3	@3	000 0 000000 000 011	0000 0000 0000 0000 (RAM[3])
4	Add	D=D+A	111 0 000010 010 000	(Unchanged – Result is in D)
5	Select Addr 5	@5	000 0 000000 000 101	0000 0000 0000 0000 (RAM[5])
6	Write Result	M=D	111 0 001100 001 000	0000 0000 0000 1101 (13)
7	Load 10	@10	000 0 000000 001 010	0000 0000 0000 0000 (RAM[10])
8	Move to D	D=A	111 0 110000 010 000	0000 0000 0000 0000
9	Load 3	@3	000 0 000000 000 011	0000 0000 0000 0000 (RAM[3])
10	Subtract	D=D-A	111 0 010011 010 000	(Unchanged – Result is in D)
11	Select Addr 6	@6	000 0 000000 000 110	0000 0000 0000 0000 (RAM[6])
12	Write Result	M=D	111 0 001100 001 000	0000 0000 0000 0111 (7)

Following the write operations, a final verification step was performed to ensure data persistence. The address pointer was manually reset to revisit previously written locations, specifically returning to Address 5, 6 after writing to Address 1000. The retrieved data matched the originally written values, confirming that writing to higher memory addresses did not accidentally overwrite or corrupt data stored in lower addresses. This validated that the memory correctly retains information across different locations without interference.

Step	Instruction	Math	Switch Code (Binary)	LED Output (Monitor)
1	Load Value (3855)	@3855	000 0 111100 001 111	0000 0000 0000 0000 Shows RAM[3855]
2	Move to D	D=A	111 0 110000 010 000	(Unchanged)
3	Load Addr 1000	@1000	000 0 001111 101 000	0000 0000 0000 0000 Shows RAM[1000]
4	Write Memory	M=D	111 0 001100 001 000	0000 1111 0000 1111 (RAM[1000])

2. **ALU Computational Verification:** To verify the Arithmetic Logic Unit (ALU), a "Rapid Fire" test was conducted. The CPU was set up with **Operand 1 (D-Register) = 10** and **Operand 2 (A-Register) = 3**. The instruction set was then exercised to validate all supported computations.

   Setup: D=10, A=3, Target Address=2 (Results displayed immediately via Memory Monitor).

Step	Operation	Math	Switch Code (Binary)	LED Output (Monitor)
1	Load 10		000 0 000000 001 010	0000 0000 0000 0000
2	Input D=A		111 0 110000 010 000	0000 0000 0000 0000
3	Load 2		000 0 000000 000 010	0000 0000 0000 0000
4	ADD	10 + 2	111 0 000010 001 000	0000 0000 0000 1100 (12)
5	SUB	10 - 2	111 0 010011 001 000	0000 0000 0000 1000 (8)
6	AND	10 & 2	111 0 000000 001 000	0000 0000 0000 0010 (2)
7	OR	10 | 2	111 0 010101 001 000	0000 0000 0000 1010 (10
8	NOT D	!10	111 0 001101 001 000	1111 1111 1111 0101 (-11)
9	NOT A	!2	111 0 110001 001 000	1111 1111 1111 1101 (-3)
10	NEG D	-10	111 0 001111 001 000	1111 1111 1111 0110 (-10)
11	INC D	10 + 1	111 0 011111 001 000	0000 0000 0000 1011 (11)
12	DEC D	10 - 1	111 0 001110 001 000	0000 0000 0000 1001 (9)
13	CONST 1	1	111 0 111111 001 000	0000 0000 0000 0001 (1)
14	CONST -1	-1	111 0 111010 001 000	1111 1111 1111 1111 (-1)

Mode Switching Reliability

The final phase tested the seamless switching capability between modes. Using the implemented toggle logic, the system was repeatedly switched between Auto and Manual modes. The testing confirmed that the system could complete a high-speed calculation in Auto Mode, latch the result, and then immediately transition to Manual Mode to allow the user to inspect memory contents without requiring a system reset. This validated the stability of the clock multiplexing and debouncing logic.

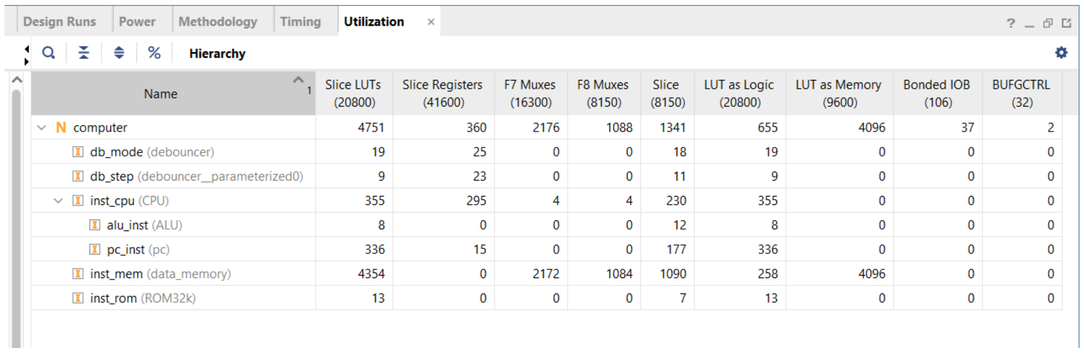
3.3 Quality of Results

This section presents the post-implementation analysis of the design, evaluating resource efficiency, timing compliance, and power consumption.

3.3.1 Resource Utilization Analysis

The design was synthesized and implemented on the target FPGA to assess hardware overhead. As shown in *Image 4*, the implementation relies heavily on Look-Up Tables (LUTs) rather than dedicated Block RAM (BRAM) or DSP slices, reflecting the distributed nature of the Hack architecture's memory and logic.

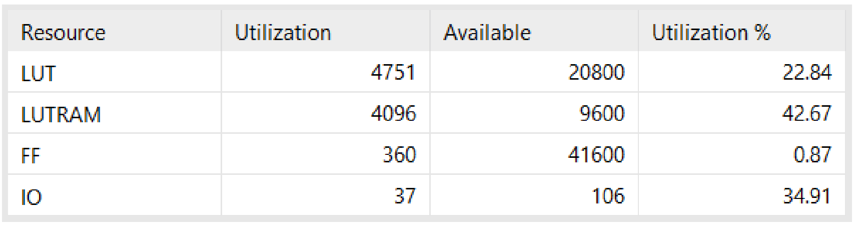
- **LUT Utilization:** The design utilizes 4,751 LUTs (22.84% of available capacity). A significant portion of this (43% of LUTRAM) is dedicated to Distributed RAM, confirming that the Data Memory was successfully mapped to slice logic rather than expensive BRAM blocks.
- **Register Usage:** Flip-flop usage is minimal at 0.87% (360 registers). This low utilization is expected for a single-cycle architecture, where data storage is primarily handled by the RAM and the CPU state is maintained by only three main registers (A, D, and PC).
- **IO Usage:** The design uses 37 IO ports (34.91%), accounting for the 16 switches, 16 LEDs, and control buttons required for the user interface.



Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Slice (8150)	LUT as Logic (20800)	LUT as Memory (9600)	Bonded IOB (106)	BUFGCTRL (32)
computer	4751	360	2176	1088	1341	655	4096	37	2
db_mode (debouncer)	19	25	0	0	18	19	0	0	0
db_step (debouncer__parameterized0)	9	23	0	0	11	9	0	0	0
inst_cpu (CPU)	355	295	4	4	230	355	0	0	0
alu_inst (ALU)	8	0	0	0	12	8	0	0	0
pc_inst (pc)	336	15	0	0	177	336	0	0	0
inst_mem (data_memory)	4354	0	2172	1084	1090	258	4096	0	0
inst_rom (ROM32k)	13	0	0	0	7	13	0	0	0

Image 4: Instance-wise Resource Utilization Summary

The Instruction Memory (ROM), despite being declared as a 32k address space, utilizes only **13 LUTs** (<1% utilization). This is because the synthesis tool optimized the sparse memory array into **combinational logic**. Since the program loaded during synthesis was small (approx. 15 instructions) and the content is constant, the tool pruned the unused address space, implementing the specific instruction set as a compact logic function rather than allocating physical memory resources.



Resource	Utilization	Available	Utilization %
LUT	4751	20800	22.84
LUTRAM	4096	9600	42.67
FF	360	41600	0.87
IO	37	106	34.91

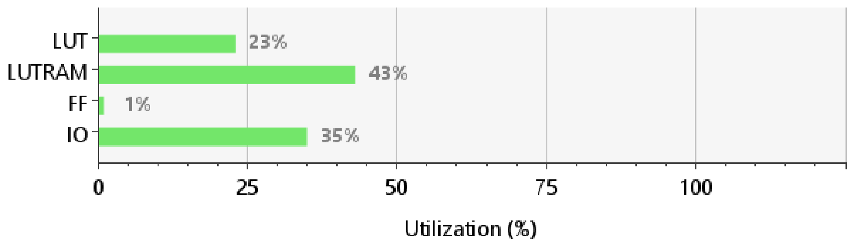
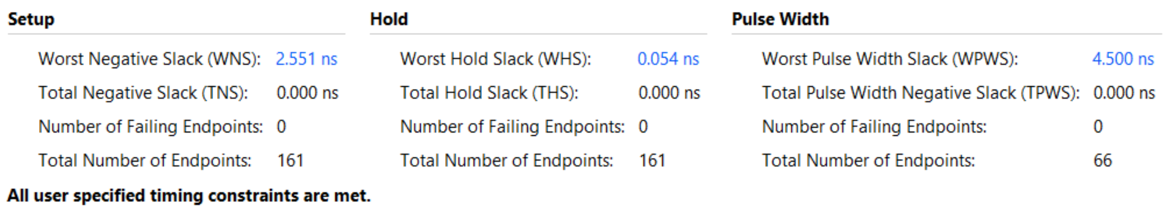


Image 5: Resource Utilization Summary

3.3.2 Timing Performance and Closure

Timing analysis was performed to verify signal integrity and setup/hold compliance. As detailed in the previous section, the system clock was divided to 50 MHz to accommodate the critical path of the single-cycle processor.

- **Timing Constraints Met:** As shown in *Image 6*, the design meets all timing constraints with **0 failing endpoints**.
- **Setup Time Margin:** The **Worst Negative Slack (WNS)** is **+2.551 ns**. This indicates a healthy timing margin; the critical path signal arrives 2.551 ns before the clock edge, confirming that the reduced 50 MHz frequency provides sufficient time for the CPU's fetch-decode-execute cycle to complete reliably.
- **Hold Time Compliance:** The Worst Hold Slack (WHS) is **+0.054 ns**, ensuring that no race conditions occur where data changes too rapidly for the destination registers to capture.



Setup

Worst Negative Slack (WNS):	2.551 ns
Total Negative Slack (TNS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	161

Hold

Worst Hold Slack (WHS):	0.054 ns
Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	161

Pulse Width

Worst Pulse Width Slack (WPWS):	4.500 ns
Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	66

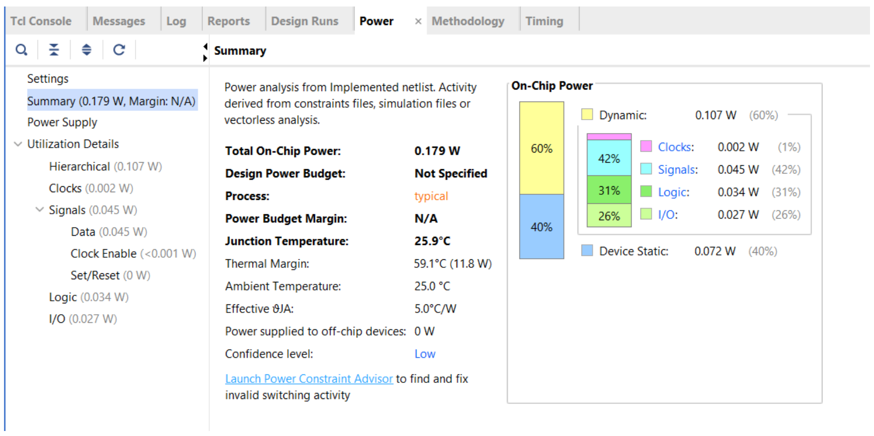
All user specified timing constraints are met.

Image 6: Design Timing Summary showing successful timing closure (Positive Slack).

3.3.3 Power Consumption Profile

Power analysis was conducted on the implemented netlist to estimate the thermal and energy footprint of the system.

- **Total On-Chip Power:** The total estimated power consumption is **0.179 W**.
- **Dynamic vs. Static Power:**
    - **Dynamic Power (60%):** Accounting for **0.107 W**, this represents power consumed by switching activity (signals toggling). The signals themselves account for the largest share (42%), followed by logic (31%), consistent with a design that moves data frequently across a bus-based architecture.
    - **Device Static Power (40%):** The leakage power stands at **0.072 W**, which is the baseline power consumption of the FPGA fabric when powered on.



Settings
Summary (0.179 W, Margin: N/A)
Power Supply
Utilization Details
Hierarchical (0.107 W)
Clocks (0.002 W)
Signals (0.045 W)
Data (0.045 W)
Clock Enable (<0.001 W)
Set/Reset (0 W)
Logic (0.034 W)
I/O (0.027 W)

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.179 W**
Design Power Budget:	**Not Specified**
Process:	typical
Power Budget Margin:	**N/A**
Junction Temperature:	**25.9°C**
Thermal Margin:	59.1°C (11.8 W)
Ambient Temperature:	25.0 °C
Effective ϑJA:	5.0°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

	Dynamic:	0.107 W	(60%)
	Clocks:	0.002 W	(1%)
	Signals:	0.045 W	(42%)
	Logic:	0.034 W	(31%)
	I/O:	0.027 W	(26%)
	Device Static:	0.072 W	(40%)

Image 7: Power Analysis Summary

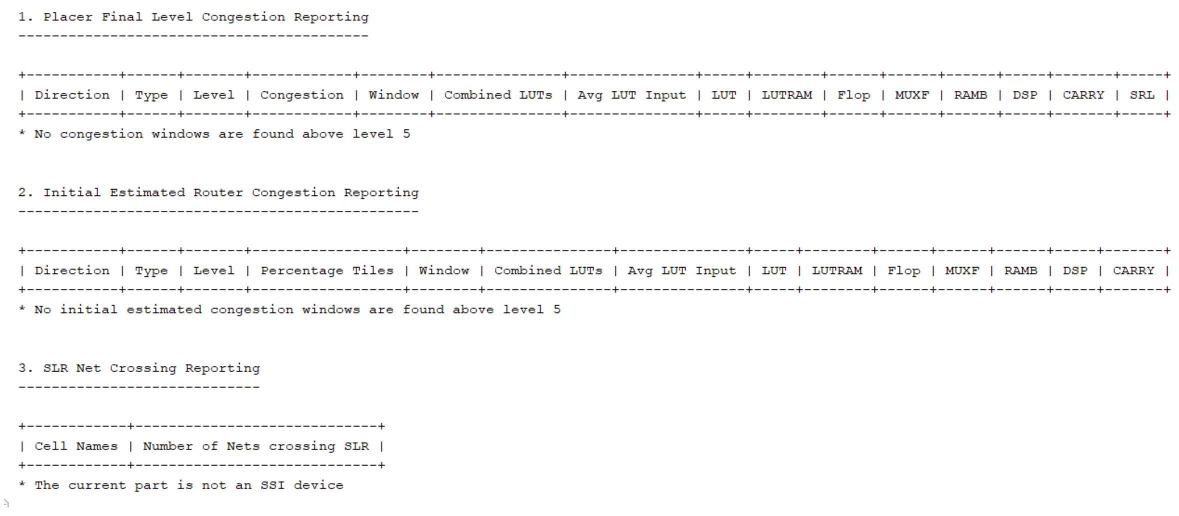
3.3.4 Design Congestion Analysis

A design congestion analysis was performed on the routed design to verify that the placement of logic cells and the routing of signals did not create any high-density "hotspots" that could impede timing or routability.

As summarized in *Image 8* below, the report indicates **zero congestion windows** above Level 5 for both the Placer and the Router.

- **Placer Final Level Congestion:** The absence of congestion suggests that the tool was able to distribute the logic elements (LUTs, Registers, and Muxes) evenly across the FPGA slices without forcing dense clusters.
- **Router Congestion:** The initial routing estimates confirmed that there were no bottlenecks in the interconnect resources (switch matrices). This is consistent with the earlier timing analysis, where routing delays, while significant, were not caused by resource starvation or rerouting due to blockage.

This low congestion profile confirms that the **Basys 3 (Artix-7)** device had ample routing resources to accommodate the custom single-cycle architecture without routing conflict.



```
1. Placer Final Level Congestion Reporting
------------------------------------------

+-----------+------+-------+------------+--------+---------------+---------------+-----+--------+------+------+------+-----+-------+-----+
| Direction | Type | Level | Congestion | Window | Combined LUTs | Avg LUT Input | LUT | LUTRAM | Flop | MUXF | RAMB | DSP | CARRY | SRL |
+-----------+------+-------+------------+--------+---------------+---------------+-----+--------+------+------+------+-----+-------+-----+
* No congestion windows are found above level 5


2. Initial Estimated Router Congestion Reporting
------------------------------------------------

+-----------+------+-------+------------------+--------+---------------+---------------+-----+--------+------+------+------+-----+-------+
| Direction | Type | Level | Percentage Tiles | Window | Combined LUTs | Avg LUT Input | LUT | LUTRAM | Flop | MUXF | RAMB | DSP | CARRY |
+-----------+------+-------+------------------+--------+---------------+---------------+-----+--------+------+------+------+-----+-------+
* No initial estimated congestion windows are found above level 5


3. SLR Net Crossing Reporting
-----------------------------

+------------+-----------------------------+
| Cell Names | Number of Nets crossing SLR |
+------------+-----------------------------+
* The current part is not an SSI device
```

Image 8: Vivado Design Congestion Report

4. Design Evolution: Issues and Resolutions

The development of the Hack Computer was a step-by-step learning process. In the beginning, the main challenge was simply getting used to **Verilog syntax** and understanding which code to use for hardware. As I got comfortable with the language, the challenges became more about the architecture itself, specifically, figuring out how to turn the abstract Hack instructions into real logic gates.

A major learning moment happened during the CPU testing. I chose to integrate the CPU directly into the full system rather than simulating it alone first. While this helped me understand how the components worked together, it also meant that simple wiring errors in the **Control Unit** were harder to spot. When the computer didn't behave as expected, I had to work backward from the system errors to find and fix the logic inside the CPU. This process taught me the importance of patience and structured debugging.

The table below summarizes the key issues I faced and how I solved them.

Sl. No.	Issue	Diagnosis	Resolution
1	**FPGA Memory I/O Mapping:** Unable to map 15-bit Address and 16-bit Data inputs simultaneously on Basys 3 due to limited switches.	The board lacks 32 physical switches.	Implemented a **Time-Shared Input Scheme**. The 16 switches are used for both address and data; BTN_addr latches the switches into the address register, while BTN_write writes the current switch values to that latched address.
2	**CPU Control Logic:** Difficulty determining how to drive the ALU and Registers from the 16-bit instruction.	The raw binary instruction needed decoding.	Reverse-engineered the Hack "C-instruction" format to derive Boolean logic equations for each control bit (dest, comp, jump) using logic gates.
3	**Synthesizability of Memory Initialization:** Mistakenly used an initial block to preload values into the memory registers, which is not valid for hardware synthesis.	'initial' blocks are primarily for simulation and are not synthesizable for dynamic circuit operation.	Replaced the initial block with a synchronous always @(posedge clk) block. Hard-coded values are now loaded into the registers only when the external initialise signal is asserted high.

Sl. No.	Issue	Diagnosis	Resolution
4	**ROM Synthesis Strategy:** Avoiding unsynthesizable $readmem or .hex files for the Instruction Memory.	External file loading is not always reliable for synthesis without BRAM constraints.	Implemented a hard-coded "Loader State." Used an always block triggered by an initialize signal to write instructions directly into the register array (LUTs).
5	**ALU Computation Error:** ALU output was incorrect for valid inputs.	The "Computation" field bits from the instruction were wired to the ALU in reverse order.	Corrected the wire assignments to match the Hack ISA specification.
6	**Data Fetch Timing Error:** CPU was fetching old data or missing data cycles.	The RAM was using Synchronous Read (posedge clk), adding a 1-cycle delay that broke the Single-Cycle architecture.	Converted Memory Read logic to Asynchronous (always @*) to ensure data is available immediately.
7	**Auto-Mode Visualization:** Result values changing too rapidly to observe on LEDs during Auto Mode.	The display was tracking the rapidly changing accumulator.	Implemented a display strategy where Auto Mode ties the LEDs to a specific memory location, while Manual Mode shows RAM[A].
8	**Mode Button Instability:** Mode switching was erratic (detecting multiple presses).	Mechanical switch bounce.	Implemented a parameterized Debouncer. Configured the "Mode" button as a Toggle switch and the "Step" button as a Pulse generator.
9	**Timing Violation:** Design failed timing analysis with WNS of -1.843 ns.	The critical path (Routing Delay + Logic) exceeded the 10ns period (100 MHz).	Implemented a Clock Divider to step down the system frequency to 50 MHz, providing sufficient slack for signal propagation.

5. Key Learnings

This project served as a bridge between my theoretical knowledge of computer architecture and practical hardware implementation. Beyond just getting the design to work, the process taught me several valuable lessons about digital system design.

5.1 Verilog and Hardware Modeling

A major takeaway was understanding that Verilog is fundamentally different from software programming. I learned to view the code not as a set of sequential instructions, but as a model for physical hardware. Understanding **Synthesis**, how the tools translate high-level code into physical primitives like LUTs and Flip-Flops, helped me write more efficient and synthesizable code.

5.2 Engineering Best Practices

Working on a design of this complexity highlighted the importance of code discipline.

- **Documentation:** I learned that detailed commenting is essential, not just for others, but for maintaining my own understanding of the logic during debugging.
- **Readability:** I adopted strict indentation and naming conventions. Although Verilog does not enforce this, I found that organized code was significantly easier to debug and manage, especially when integrating multiple modules.

5.3 The ASIC Design Flow

I gained a better appreciation for the industrial approach to chip design.

- **Role of FPGAs:** I understood why FPGAs are critical in the industry. They act as a verification ground, allowing engineers to test designs extensively at hardware speeds. This ensures that logic errors are caught early, avoiding the high cost of failure during silicon fabrication.
- **Tool Proficiency:** I became more comfortable with the toolchain, understanding the necessary steps from RTL design to Synthesis, Implementation, and Bitstream generation.

5.4 Computer Architecture and Debugging

Building the CPU from the ground up provided insights that textbooks could not offer.

- **System Integration:** I developed a concrete understanding of how the Control Unit, ALU, and Memory interact, particularly regarding timing and memory access cycles.
- **Structured Debugging:** The most significant learning curve occurred during the debugging phase. When my initial CPU wiring was incorrect, I learned to move away from trial-and-error. Instead, I adopted a structured approach, simulating individual modules and verifying waveforms, which ultimately led to a working design. This experience honed my analytical skills and patience.

6. Appendix

6.1 Elaborated circuit

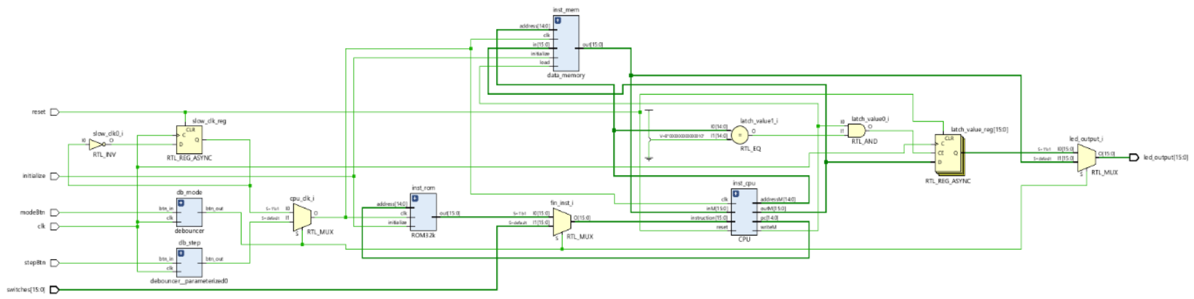


Image 9: Elaborated Design of the Top Module

The schematic *(Image 9)*, illustrates the complete system integration. On the left, input signals pass through debouncers and clock generation logic to drive the central **CPU**, **Instruction Memory (ROM)**, and **Data Memory (RAM)**. The layout confirms the single-cycle architecture's connectivity, showing the 16-bit data paths routing from the input switches, through the processor, to the final LED output multiplexers on the right.

6.2 Simulation Results

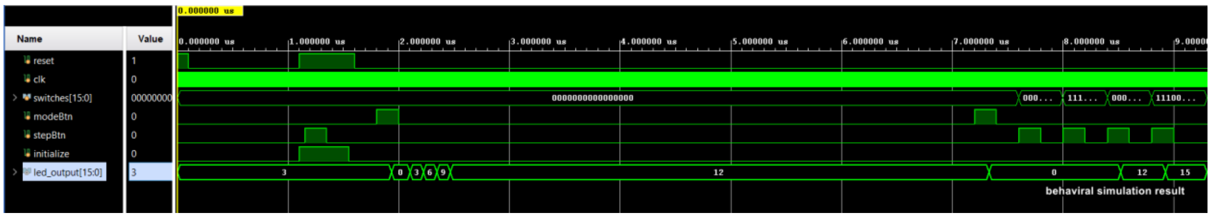


Image 10: Behavioral Simulation of CPU Mode Switching and Arithmetic Execution

This waveform in *Image 10* demonstrates the functional verification of the CPU's dual-mode operation, validating the transition between high-speed automatic execution and manual stepping.

1. **Automatic Multiplication Sequence (2.0 µs – 7.5 µs):**

   Upon asserting the modeBtn signal (High), the system switches to Auto Mode using the 83.3 MHz system clock. The CPU executes the pre-loaded multiplication program (3*4). The led_output bus visualizes the real-time accumulation in the D-register/Memory:

   - **Sequence:** 3 → 6 → 9 → 12
   - **Result Latch:** The final result **12** (0000...1100) remains latched and stable on the output even as the CPU enters its infinite termination loop.

2. **Transition to Manual Mode (7.5 µs):**

When modeBtn is de-asserted (Low), the system instantly reverts to Manual Mode. The led_output transitions from the latched result to the Memory Monitor state. It momentarily displays 0, reflecting the empty data contents of the current instruction address (the infinite loop location).

3. **Manual Verification (8.0 µs – 9.0 µs):**

The user manually inputs instructions via the switches bus and advances the execution using the stepBtn.

- The first manual step retrieves and displays **12** (verifying the previous result stored in memory).
- The second step retrieves and displays **15** (verifying a new memory read operation).

This confirms that the manual clock path (stepBtn) and the memory read logic function correctly after a high-speed automatic run.

6.3 System Verification and Functional Testing

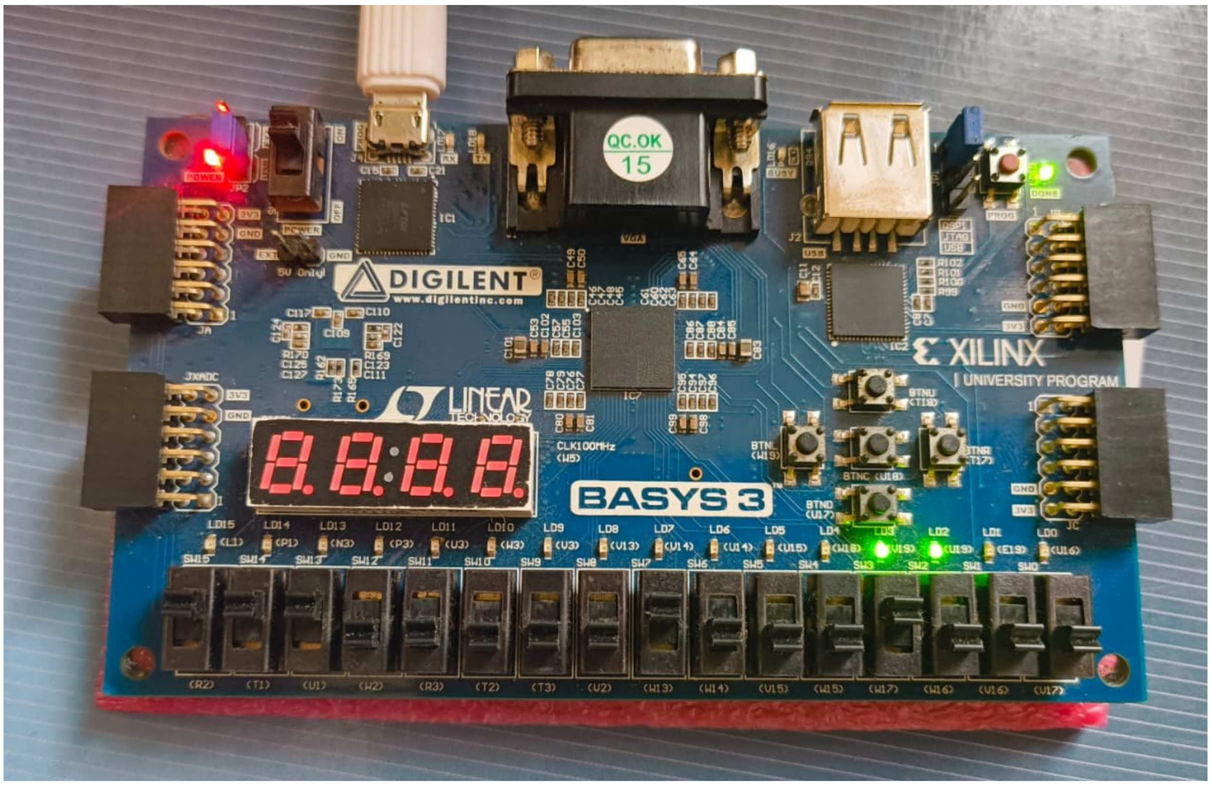


Image 11: Adding inputs 10 and 2. LEDs display binary 12 (00...1100).

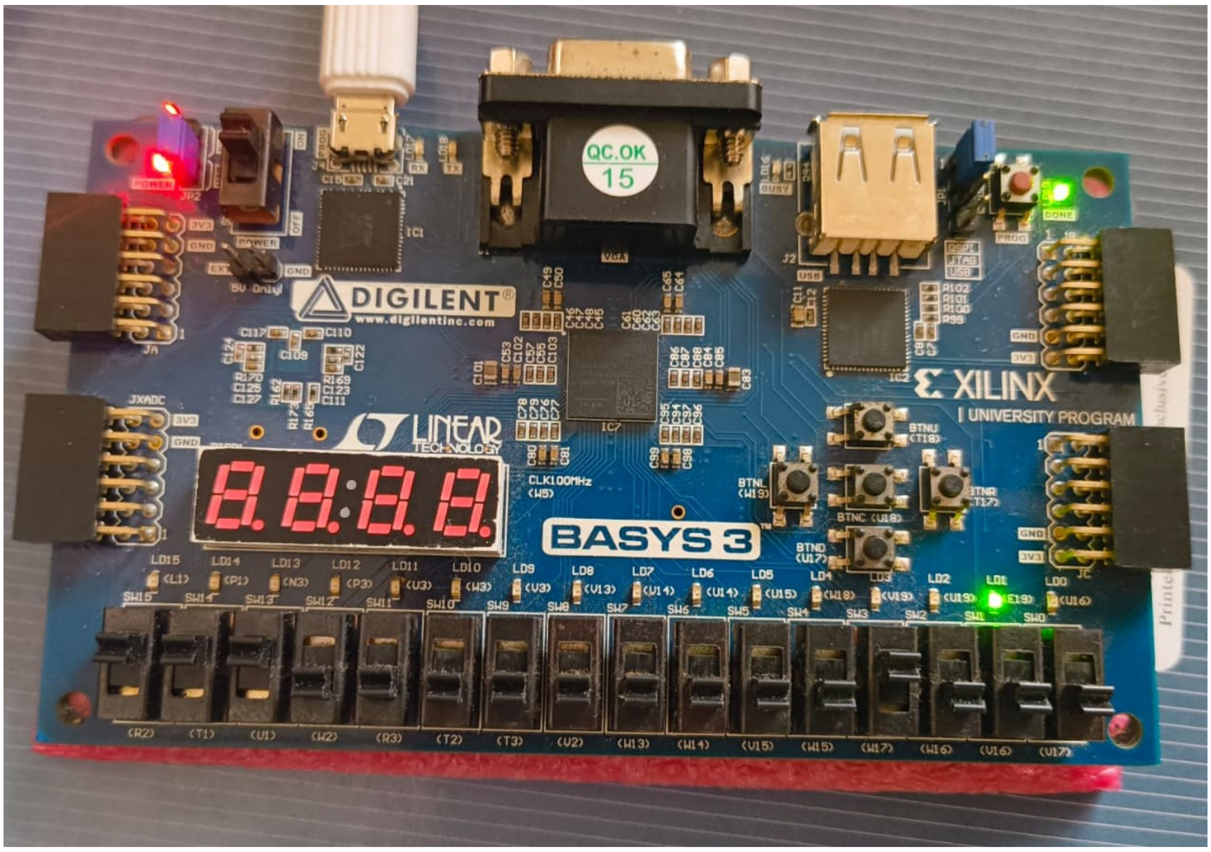


Image 12: Bitwise AND operation between 10 (1010) and 2 (0010). LEDs correctly isolate the overlapping bit (0010).

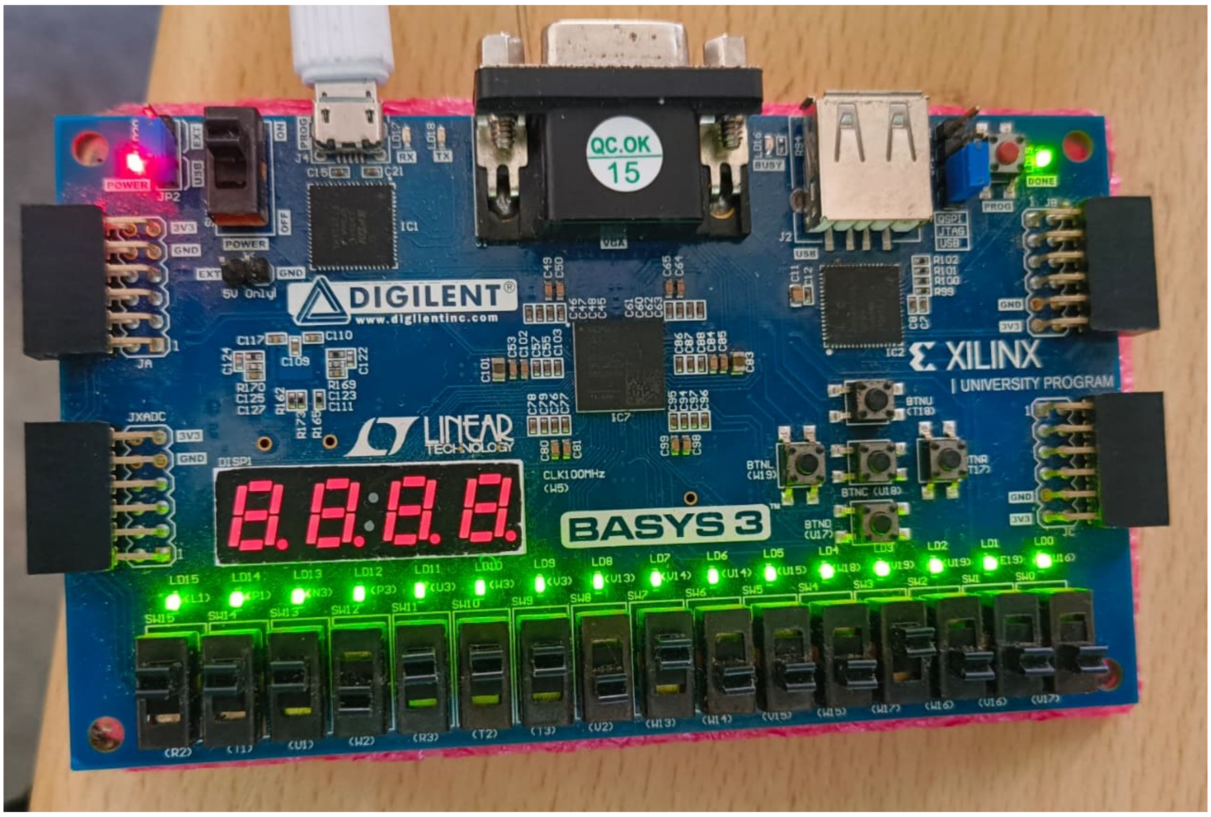


Image 13: Executing 'D=-1'. All 16 LEDs illuminate, confirming the Two's Complement representation of -1 (1111 1111 1111 1111).

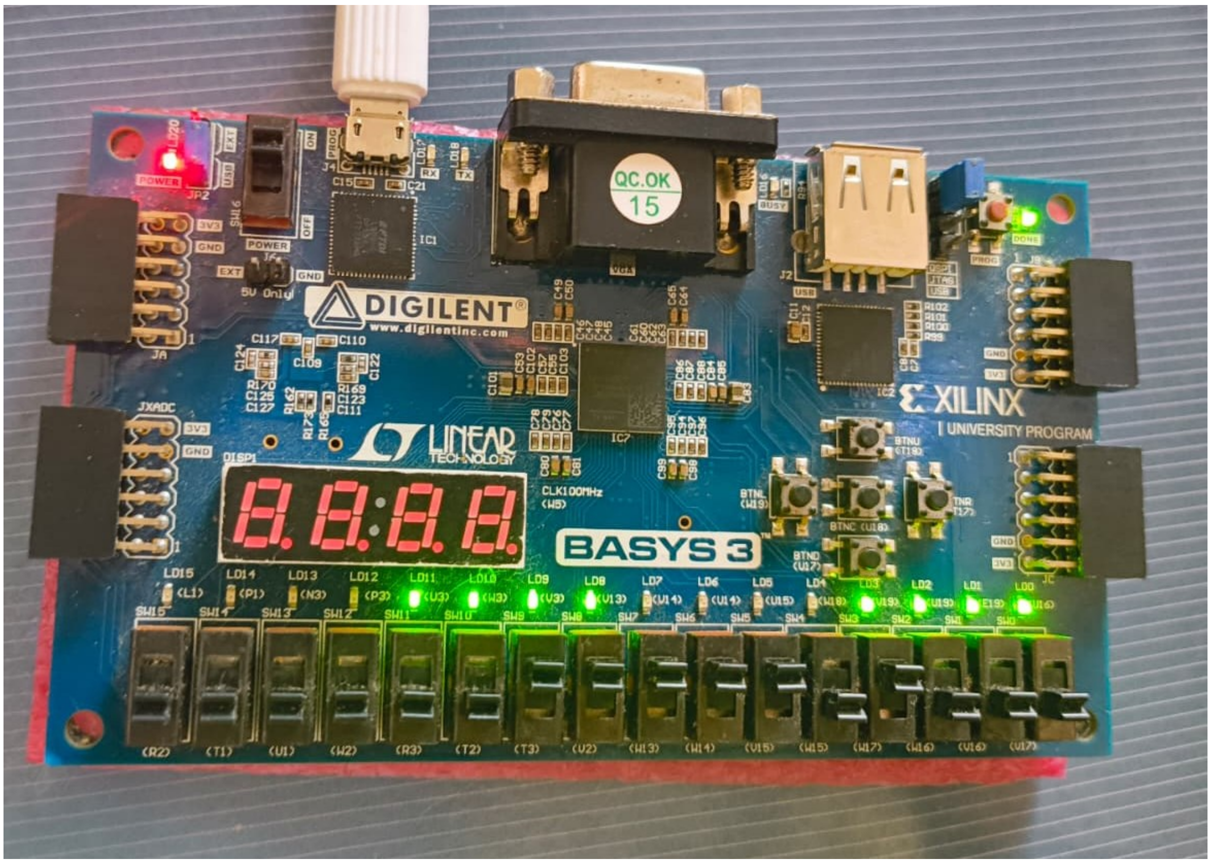


Image 14: Successfully writing to high memory address 1000 and retrieving the data, confirming successful read-write operations.

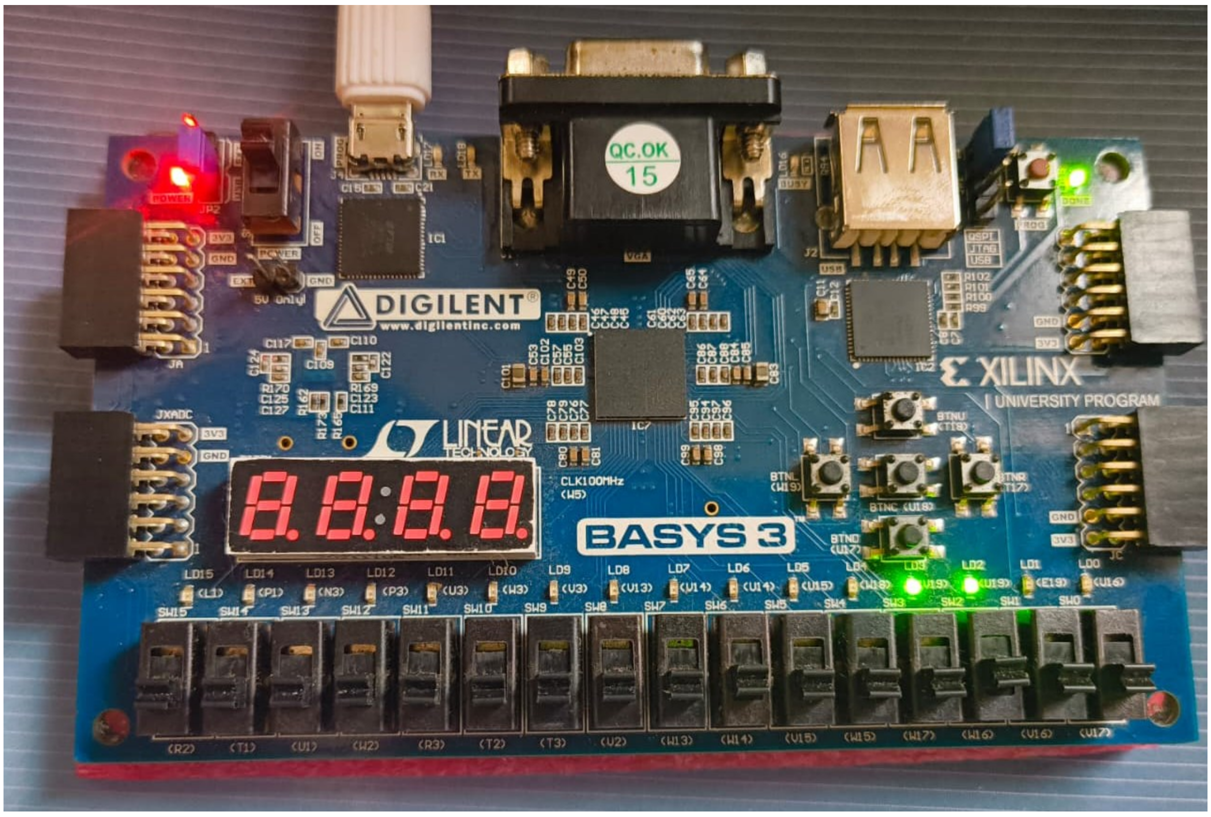


Image 15: Hardware execution of a multiplication algorithm (3 × 4). Showing result 12 (1100)

6.4 Simulation Results of Clock Divider and Debouncer

1. The simulation confirms the frequency divider logic. The generated slow_clk (bottom trace) toggles state only after a full period of the system clk (top trace), resulting in a clock period exactly twice as long (20ns) as the source, producing the required 50 MHz signal.

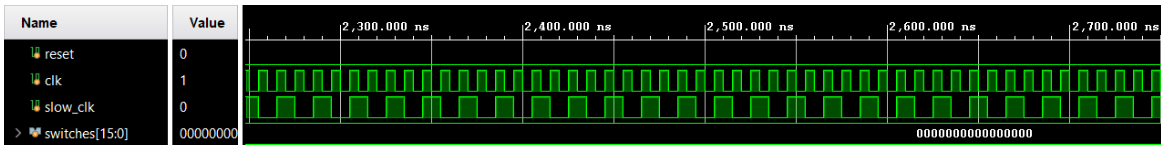


Image 16: Clock Frequency Division

2. This waveform illustrates the pulse generation for the Step button. The debounced output clean_step (orange line) does not rise immediately; it waits for the raw stepBtn input to remain stable for the timer duration before generating a clean positive edge, effectively filtering out noise.

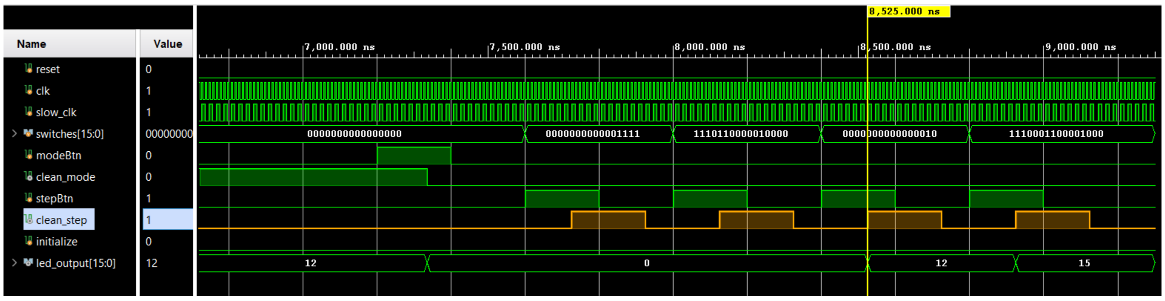


Image 17: Step Button Signal Conditioning

3. The areas marked in red *(Image 18)*, highlight the latching logic of the parameterized debouncer. Unlike the Step button, the clean_mode signal acts as a toggle switch: it latches HIGH upon the first valid press and holds that state stably until the button is pressed a second time to toggle it back LOW.

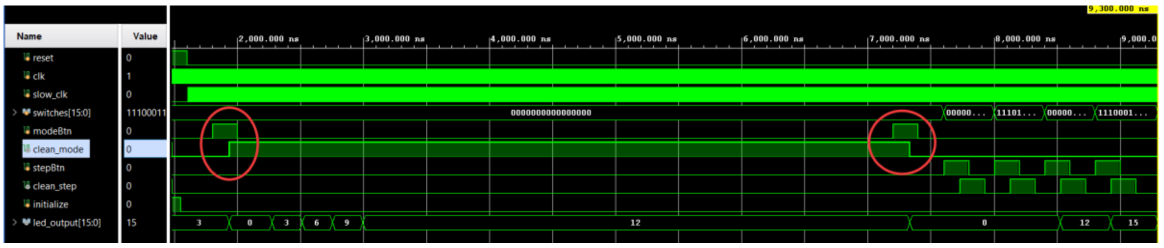


Image 18: Mode Button Toggle Behavior

6.5 Links

GitHub Repository Link: https://github.com/rishtirkulkarni/CPU-on-FPGA/tree/main/CPU%20Implementation

CPU Test Video Link: https://youtu.be/oumZqjgryk8

7. References

1. N. Nisan and S. Schocken, *The Elements of Computing Systems: Building a Modern Computer from First Principles*. MIT Press, 2005.
2. Samir Palnitkar, *Verilog HDL: A Guide to Digital Design and Synthesis*, 2nd ed. Prentice Hall, 2003.
3. Digilent Inc., "Basys 3 FPGA Board Reference Manual," 2016. [Online]. Available: https://digilent.com/reference/programmable-logic/basys-3/reference-manual
4. N. Nisan and S. Schocken, "The NAND2Tetris Course," *NAND2Tetris.org*. [Online]. Available: https://www.nand2tetris.org/
5. Xilinx Inc., "Vivado Design Suite User Guide: Implementation," UG904, 2021. https://docs.amd.com/r/en-US/ug904-vivado-implementation/Preparing-for-Implementation