UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Design and Implementation of an MMX arithmetic unit

Moruţan Maria

Group: 30433

23.01.2026

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Contents

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

1. Introduction

1.1 Purpose of the Project

This project aims to design and implement an MMX Arithmetic Unit using VHDL. The objective is to reproduce a subset of the MMX instruction set architecture (ISA) to enhance multimedia processing on x86 processors. The system is described in VHDL, verified through functional simulation using Xilinx Vivado, and implemented on the Basys 3 FPGA board (Artix-7).

The purpose of this project is to provide a comprehensive understanding of the internal structure and operation of a SIMD (Single Instruction, Multiple Data) arithmetic unit that performs multiple parallel arithmetic operations in a single clock cycle — a key feature of modern multimedia and signal-processing systems. SIMD arithmetic is fundamental to digital systems that process vectors of data, such as in image processing, audio filtering, and video encoding.

The goal of this project is to replicate this hardware functionality in a simplified, educational context using hardware description language (VHDL).

1.2 Project overview

The designed unit will include:

- PADD (Packed Add) modules implementing parallel addition on multiple subword elements (8-bit packed byte).
- PSUB (Packed Subtract) modules implementing parallel subtraction on multiple subword elements (8-bit packed byte).
- PMUL (Packed Multiplication) modules implementing parallel multiplication on multiple subword elements (8-bit packed byte).
- PINC (Packed Increment) modules implementing parallel incrementation on multiple subword elements (8-bit packed byte).
- PDEC (Packed Decrement) modules implementing parallel decrementation on multiple subword elements (8-bit packed byte).
- PAVG (Packed Average) modules implementing parallel computation of the average of corresponding packed integers.
- A control unit (MMX ALU) which selects the operation based on an opcode and coordinates multi-cycle operations (PMUL).
  The arithmetic unit operates on a 64-bit MMX register, divided into multiple lanes (primarily 8-bit packed bytes in the final implementation).

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

1.3 Project Specifications

- Hardware description language: VHDL
- Simulation environment: Xilinx Vivado 2023.x
- Target platform: Basys 3 (AMD Artix-7 FPGA Trainer Board)
- Supported Operations: PADD (Packed Add), PSUB (Packed Subtract), PMUL (Packed Multiplication), PINC(Packed Increment), PDEC(Packed Decrement), and PAVG(Packed Average) instructions.
- Data width: 64-bit MMX register
- Arithmetic mode: Parallel SIMD integer arithmetic

1.4 Project Objectives

- To understand the MMX arithmetic instruction set and its relevance in SIMD-based arithmetic.
- To design, simulate, and implement packed addition, subtraction, multiplication, incrementation, decrementation, and average computation modules in VHDL.
- To develop a control mechanism capable of managing multiple MMX arithmetic operations.
- To verify and validate the correct functionality of the arithmetic unit.
- To document and analyze simulation results and synthesize the design for FPGA implementation.

1.5 Project Plan

To ensure structured development, the project will be realized incrementally according to the following plan:

- Project meeting 3: Design and simulation of the PADD, PSUB, PINC, and PDEC modules.
- Project meeting 4: Design and simulation of the PMUL, PAVG modules.
- Project meeting 5: Implementation of the control unit and integration of arithmetic modules.
- Project meeting 6: Documentation, testing, validation, and final report preparation.

The present documentation will be updated throughout the project to include the latest design, simulation results, and testing outcomes.

2. Bibliographic research

The project is based on concepts from:

- SIMD processor architectures
- Intel MMX instruction set

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

- Digital arithmetic circuit design
- FPGA-based hardware implementation

The ALU operates as part of the processor's datapath and plays a crucial role in instruction execution, enabling both computational and decision-making capabilities of the system. Due to its fundamental functionality, the ALU is not limited to Central Processing Units (CPUs), but is also an essential building block in specialized processing units, including Graphical Processing Units (GPUs), Floating Point Units (FPUs), and multimedia extensions such as MMX units.

MMX was introduced by Intel to accelerate multimedia workloads by allowing parallel arithmetic on packed data types. Although modern processors use more advanced SIMD extensions (SSE, AVX), MMX remains an excellent educational example due to its simplicity.

VHDL was chosen as the implementation language due to its strong support for structural and behavioral modeling and its suitability for FPGA synthesis.

3. Bibliography

- Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, Vol. 1
- S. Brown, Z. Vranesic, *Fundamentals of Digital Logic with VHDL Design*, McGraw-Hill
- IEEE, *IEEE Standard VHDL Language Reference Manual*
- Xilinx Inc., *Vivado Design Suite User Guide*

4. Analysis

4.1 The SIMD Paradigm (Subword Parallelism)

The fundamental challenge of this project is the transition from Scalar Arithmetic to Vector Arithmetic. In a scalar 64-bit adder, bit **n** depends on the carry from bit **n-1**. In an MMX-style SIMD unit, we implement Subword Parallelism. This requires breaking the carry chain at 8-bit intervals.

If we were to use a standard 64-bit operator **(A + B)**, a carry generated at the 7th bit (the end of the first byte) would "leak" into the 8th bit (the start of the second byte), corrupting the adjacent data lane.

4.2 Lane Isolation and Modular Design

To prevent data leakage between lanes, two design strategies were analyzed:

1. **Masked Addition**: Using a standard 64-bit adder but applying bit-masks to prevent carry propagation. (Discarded due to complexity in handling subtraction and multiplication).
2. **Component Instantiation**: Creating a single 8-bit "Lane Module" and instantiating it 8 times. (Selected for its modularity and ease of debugging).

4.3 Computational Complexity of PMUL

While PADD, PSUB, PINC, and PDEC are combinational (result is available almost instantly), PMUL is sequential.

- **The Problem**: Multiplying two 8-bit numbers results in a 16-bit product. However, the MMX standard for PMULL (Packed Multiply Low) requires only the lower 8 bits of the product to be returned to maintain the 64-bit register width.
- **The Hardware Trade-off**: Using the "Shift-and-Add" algorithm saves FPGA area (using registers and a small adder) compared to a large DSP-style multiplier, but it introduces Latency. The system must account for the 8 clock cycles required for the result to stabilize.

4.4 Data Synchronization and the "Busy" State

Because the system blends combinational operations (Add/Sub) with sequential ones (Mul), the Control Unit must manage a **"Busy/Done" handshake**.

- When OPCODE is 101 (PMUL), the top-level entity must ignore the ALU output until the DONE **signal is asserted**.
- The analysis of the "Stuck Busy" bug revealed that without a Transition Trigger (like a pulse when the ROM address changes), the multiplier remains in its terminal state, failing to register new data. This necessitated the implementation of an **"Auto-Reset"** logic in the Design II phase.

4.5 Human-Machine Interface (HMI) Constraints

A 64-bit result cannot be displayed on a 4-digit Seven-Segment Display (SSD) simultaneously.

- Analysis of Multiplexing: The multiplexing was done using a "scrolling" text method versus a "manual slice" selection.

- Decision: The manual slice method (using SW[1:0]) was chosen because it allows the user to pause and verify specific bytes of the result against manual calculations, which is critical for the "Validation" phase of the project.

5. Design I (Initial Design)

5.1 Concept: The "Tape Player" Architecture

The initial design was based on a **linear execution model**. In this phase, the ROM (Read-Only Memory) acted as a fixed script. Each entry in the ROM contained both the operands (A and B) and the instruction (`OPCODE`) to be executed.

- **Logic Flow:** The system was designed to step through memory addresses sequentially.
- **Data Structure:** Each ROM word was 131 bits wide: 3 bits for the Opcode, 64 bits for Operand A, and 64 bits for Operand B.
- **Limitation:** The user had no control over the operation performed on specific data. To test a different operation on the same numbers, the ROM had to be recompiled and reflashed onto the FPGA.

5.2 Hardware Mapping

- **Input:** A single button (`BTN`) was used to increment the `rom_addr`.
- **Control:** The ALU was a "Slave" to the ROM; it simply executed whatever opcode was hardcoded at the current address.
- **Feedback:** The system was "blind" to the user's intent—it simply showed the pre-calculated result.

6. Design II (Updated Design)

6.1 Concept: The "Interactive Menu" Architecture

Based on the analysis of Design I, the architecture was overhauled to provide **User Agency**. The ROM was decoupled from the logic, transforming the project into a functional SIMD calculator.

- **Logic Flow:** The ROM was repurposed to store only a library of "Data Pairs." The control logic (Opcode selection) was moved to the physical switches on the Basys 3 board.

- **The "Menu" System:** This allowed for a M x N testing matrix (where M is the number of ROM data pairs and N is the number of available operations), significantly increasing the verification coverage.

6.2 Structural Improvements

1. **Opcode Injection:** The `OPCODE` signal was mapped to `SW[15:13]`. This required a change in the `mmx_basys` top-level entity to route switch signals directly to the ALU's control port.
2. **Internal Reset Logic (`alu_rst_internal`):**

   A critical design update was the introduction of a composite reset signal:

   alu_rst_internal = BTNRST OR next_addr_pulse

   This ensured that the **PMUL** state machine was reset every time a new data pair was selected, preventing the "Stuck Busy" state encountered in earlier iterations.

3. **Visual Multiplexing:**

   Since 64-bit results cannot fit on the 4-digit SSD, a "View Window" multiplexer was designed. By using `SW[1:0]`, the user can shift a 16-bit "window" across the 64-bit result register.

6.3 Comparison Summary

Feature	Design I	Design II
ROM Content	Data + Opcode	Data Only
Control Source	Hardcoded in Memory	Physical Switches
Flexibility	Low	High
Reset Logic	Global Reset only	Auto-Reset on Data Change
Verification	One test per ROM entry	6 tests per ROM entry

7. Implementation

7.1 Implementation I: The Arithmetic Modules (Low-Level)

The first phase of implementation involved creating the individual "lanes" for the SIMD operations. Each module was designed to process 8-bit inputs to ensure no carry leakage between bytes.

- **Combinational Modules:** `PADD`, `PSUB`, `PAVG`, `PINC`, and `PDEC` were implemented using behavioral VHDL. For example, `PAVG` was implemented by extending the 8-bit inputs to 9 bits (A+B) to capture the carry before performing a logical right shift by 1.

- **Sequential Module (`PMUL`):** To conserve FPGA resources, a shift-and-add multiplier was used. It utilizes an internal state machine (FSM) that cycles through 8 iterations to produce the result.
- **The MMX Wrapper:** All 8 lanes for a specific operation were wrapped into a single 64-bit component (e.g., `padd.vhd`), which instantiates the 8-bit component eight times in the style of the `GENERATE` statement.

7.1.1. PADD & PSUB

These are the foundational SIMD operations. They are implemented using eight independent 8-bit adders/subtractors.

- **Logic:** The 64-bit inputs A and B are sliced*: A(7 downto 0)*, *A(15 downto 8)*, etc.
- **Carry Handling:** By using separate 8-bit addition statements in VHDL, the carry from the 7th bit is naturally prevented from propagating to the 8th bit.
- **Status Flags:** A `CARRY_OUT` or `UNDERFLOW_OUT` vector is generated by concatenating the 9th bit (carry) of each individual lane's operation.

```
architecture Behavioral of padd is
signal ifcarry : STD_LOGIC_VECTOR(8 downto 0);
signal local_carry : STD_LOGIC_VECTOR(7 downto 0);
begin
process(A, B)
variable a_byte, b_byte : STD_LOGIC_VECTOR(7 downto 0);
variable temp : STD_LOGIC_VECTOR(63 downto 0);
  begin
    temp := (others => '0');
        for i in 0 to 7 loop
          a_byte := (A(i*8+7 downto i*8));
          b_byte := (B(i*8+7 downto i*8));
          ifcarry <= ('0' & a_byte) + ('0' & b_byte);
          temp(i*8+7 downto i*8) := std_logic_vector(a_byte + b_byte);

          if ifcarry(8) = '1' then
            local_carry(i) <= '1';
            else
            local_carry(i) <= '0';
          end if;
        end loop;
RESULT <= temp;
VALID  <= '1';
end process;
CARRY_FLAG <= local_carry;
end architecture Behavioral;
```

```
architecture Behavioral of psub is
signal ifunder : STD_LOGIC_VECTOR(8 downto 0);
signal local_under : STD_LOGIC_VECTOR(7 downto 0);
begin
process(A, B)
variable a_byte, b_byte : STD_LOGIC_VECTOR(7 downto 0);
variable temp : STD_LOGIC_VECTOR(63 downto 0);
  begin
    temp := (others => '0');
        for i in 0 to 7 loop
          a_byte := (A(i*8+7 downto i*8));
          b_byte := (B(i*8+7 downto i*8));
          ifunder <= ('0' & a_byte) - ('0' & b_byte);
          temp(i*8+7 downto i*8) := std_logic_vector(a_byte - b_byte);

          if ifunder(8) = '1' then
            local_under(i) <= '1';
            else
            local_under(i) <= '0';
          end if;
        end loop;
  RESULT <= temp;
  VALID  <= '1';

  end process;
  UNDERFLOW_FLAG <= local_under;
  end architecture Behavioral;
```

7.1.2. PAVG

The average operation is more complex than a simple addition because it requires an extra bit of precision to prevent data loss.

- **Algorithm: Result = (A + B + 1)** >> 1 or simply (A + B) / 2 depending on the specific MMX version implemented.

- **Implementation:** Each lane performs a 9-bit addition (8 bits + 8 bits = 9 bits). This 9-bit sum is then shifted right by one position (dropping the least significant bit) to achieve the division by two. The rounding chosen is the ceiling rounding for the floating point results.
- **Example:** For `FF` (255) and `01` (1), the sum is `100` (256). Shifting `100` right gives `80` (128).

```
architecture Behavioral of pavg is

begin
process(A,B)
variable a_byte, b_byte: std_logic_vector(7 downto 0);
variable aux_sum: std_logic_vector(8 downto 0);
variable temp: std_logic_vector(63 downto 0);
begin
    temp := (others => '0');
        for i in 0 to 7 loop
          a_byte := (A(i*8+7 downto i*8));
          b_byte := (B(i*8+7 downto i*8));
          --ceil rounding
          aux_sum := ('0' & a_byte) + ('0' & b_byte) + '1';
          --right shift => /2
          temp(i*8+7 downto i*8) := aux_sum(8 downto 1);
        end loop;
RESULT <= temp;
VALID <= '1';
end process;
end Behavioral;
```

7.1.3. PMUL

Multiplication is the most resource-intensive operation. While 8 x 8 bit multiplication yields a 16-bit result, MMX instructions like `PMULL` only return the lower 8 bits to keep the output within the same 64-bit footprint.

- **Algorithm:** A **Sequential Shift-and-Add** multiplier was implemented. This uses an internal counter and an accumulator.
- **The Architectural Setup:**

1. Before the stages begin, each 8-bit lane is equipped with:
2. Multiplier Register (Q): Initialized with Operand B.
3. Multiplicand Register (B): Initialized with Operand A.
4. Accumulator Register (A): Initialized to zero.
5. Counter: A 3-bit register to track the 8 required cycles.

- **Stage 1:**

1. The Initialization (IDLE → START)
2. The Finite State Machine (FSM) stays in an IDLE state until the user_opcode matches 101 (PMUL).
3. Trigger: The FSM detects the start signal.

4. Action: It loads the 8-bit operands from the ROM into the internal registers.
5. Status: The PMUL_BUSY signal is set to high ('1'), which lights up LED 4 on the Basys 3 board.

- **Stage 2**: **The Calculation Loop (8 Cycles)**

1. This is the core of the operation. The FSM enters a CALC state that repeats 8 times. In each cycle, the hardware performs a conditional check and a shift:
2. The "Check" (Add): The hardware looks at the Least Significant Bit (LSB) of the Multiplier (Q).
3. If Q(0) = 1, the Multiplicand is added to the Accumulator.
4. If Q(0) = 0, nothing is added (adding zero).
5. The "Shift": The Accumulator and the Multiplier are treated as a combined register and shifted one bit to the right. This aligns the next bit of the multiplier for the next cycle.

- **Stage 3: Result Truncation (MMX Specific)**

1. In standard multiplication, 8 x 8 bits produce a 16-bit result. However, MMX "Packed Multiply Low" only cares about the lower 8 bits.
2. Action: Once the 8 cycles are complete, the lower 8 bits of the combined shift register are moved to the **RES_PMUL** signal.

```
architecture Behavioral of mul_8bit is
ty                    else
si                        count <= count + 1;
si                    end if;
si                when DONE =>
si                    VALID <= '1';
si                    state <= DONE;
be            end case;
        end if;
pr    end if;
va end process;
be
   RESULT <= std_logic_vector(Q);

   end Behavioral;
            A <= (others => '0');
            Q <= (others => '0');
            M <= (others => '0');
            count <= 0;
            VALID <= '0';
       else
           case state is
               when IDLE =>
                   VALID <= '0';
                   if START_MUL = '1' then
                       M <= unsigned(A_MCAND);
                       Q <= unsigned(B_MPLIER);
                       A <= (others => '0');
                       count <= 0;
                       state <= RUN;
                   end if;

               when RUN =>
                   if Q(0) = '1' then
                       sum := ('0' & A) + ('0' & M);
                   else
                       sum := ('0' & A);
                   end if;
                   A <= sum(8 downto 1);
                   Q <= sum(0) & Q(7 downto 1);
                   if count = 7 then
                       state <= DONE;
```

7.1.4. PINC & PDEC

These operations are "unary-like" in that they only require one operand from the ROM (A), while the second operand is internally fixed to 1.

- **PINC:** Performs (A + 1) for every 8-bit segment. This is useful for stepping through arrays or adjusting pixel brightness in image processing.
- **PDEC:** Performs (A - 1) for every 8-bit segment.
- **Lane Behavior:** Just like PADD, if a lane contains `FF`, incrementing it results in `00` for that lane only, without affecting the neighbor lane (no carry-out leakage).

```
architecture Behavioral of pinc is

begin
process(A)
variable a_byte: std_logic_vector(7 downto 0);
variable temp: std_logic_vector(63 downto 0);
begin
    temp := (others => '0');
    for i in 0 to 7 loop
        a_byte := (A(i*8+7 downto i*8));
        temp(i*8+7 downto i*8) := a_byte + "00000001";
    end loop;
RESULT <= temp;
VALID <= '1';
end process;
end Behavioral;
```

```
architecture Behavioral of pdec is

begin
process(A)
variable a_byte: std_logic_vector(7 downto 0);
variable temp: std_logic_vector(63 downto 0);
begin
    temp := (others => '0');
    for i in 0 to 7 loop
        a_byte := (A(i*8+7 downto i*8));
        temp(i*8+7 downto i*8) := a_byte - "00000001";
    end loop;
RESULT <= temp;
VALID <= '1';
end process;
end Behavioral;
```

7.1.5. NOP (No Operation)

The NOP is critical for the "Menu" system design.

- **Implementation:** When the switches are set to the NOP opcode (`110`), the ALU output is driven to a default state (`all zeros`).
- **Purpose:** This prevents the 7-segment display from showing old or "ghost" data when the user is transitioning between different operations or memory addresses.

7.2 Implementation II: The ALU & Control Integration (Mid-Level)

The `mmx_alu.vhd` serves as the central processing core. It integrates all the MMX wrappers into a single unit.

- **The Multiplexer Logic:** A large `CASE` statement sensitive to the `OPCODE` signal determines which module's output is routed to the `RES_OUT` port.
- **Sequential Synchronization:** The ALU monitors the `DONE` signal from the `PMUL` module. If the current `OPCODE` is multiplication, the ALU holds the previous result until the multiplier asserts the `DONE` signal, preventing "glitchy" intermediate data from appearing on the display.

- **The Composite Reset:** This was the most critical software fix in the implementation phase. By creating a signal that combines the global reset and the address change pulse, I ensured the ALU never gets stuck in a "Busy" state when the user switches data.

7.3 Implementation III: Top-Level Wrapper & HMI (Board-Level)

The `mmx_basys.vhd` bridges the internal logic with the physical hardware of the Basys 3.

- **Debouncer Logic:** Physical buttons create mechanical noise (bouncing) that can cause a single press to increment the ROM address multiple times. A `debouncer` component was implemented to sample the button state and produce a single-clock-cycle pulse (`next_addr_pulse`).
- **Memory Mapping:** The ROM was implemented as a constant array within the `mmx_rom` component. It stores 128-bit words, which are split into two 64-bit signals (A and B) before being fed into the ALU.
- **Seven-Segment Display (SSD) Driver:** The `ssd` component handles the high-frequency switching required to drive the four hex digits on the board. It takes the 16-bit `view_data` (selected by `SW[1:0]`) and converts it into the appropriate anode and cathode signals.

8. Testing and Validation

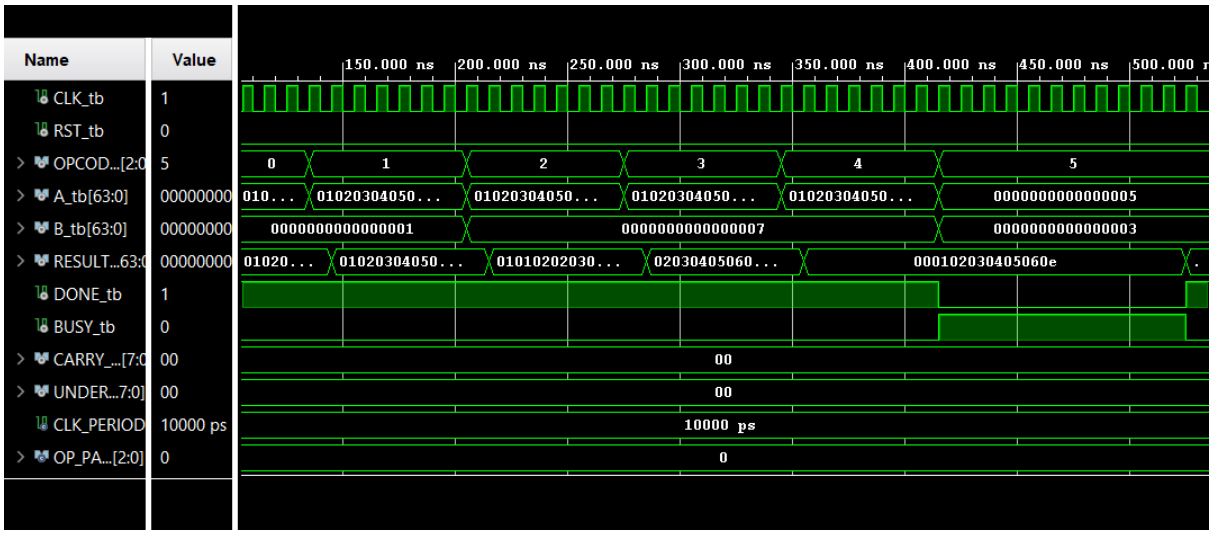
8.1 Simulation-Based Testing (Testbench)

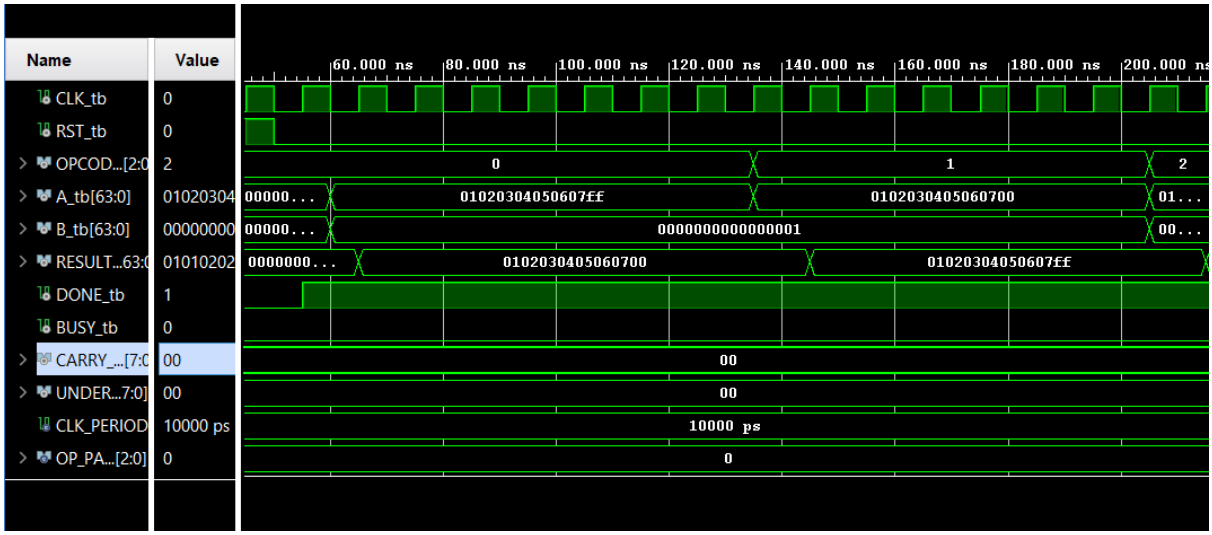
Before deploying to the Basys 3 hardware, a VHDL Testbench was developed to verify timing and logic accuracy. The testbench instantiated the `mmx_alu` and applied a series of stimulus vectors.

- **Timing Verification:** The simulation was critical for the **PMUL** operation. I verified that the `DONE_OUT` signal asserted exactly 8 clock cycles after the `BUSY` signal went high.
- **Corner Case Testing:** I simulated "Packed Carry" scenarios, such as adding `x00000000000000FF` and `x0000000000000001`. The simulation confirmed that the result was `x0000000000000100` in standard math, but for MMX, it correctly resulted in `x0000000000000000` with the `CARRY_OUT(0)` bit set, proving lane isolation.

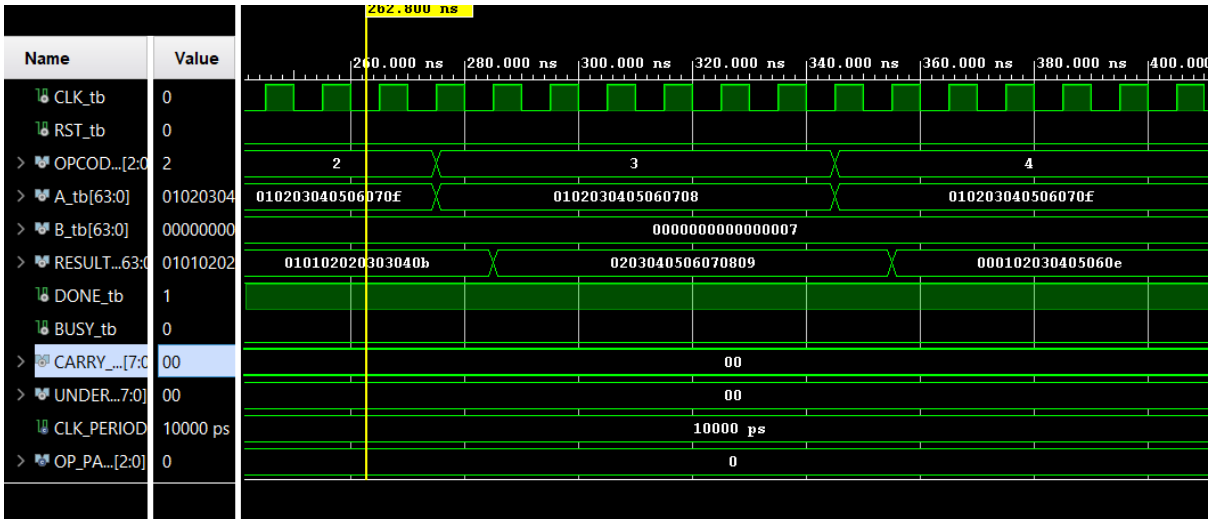
UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

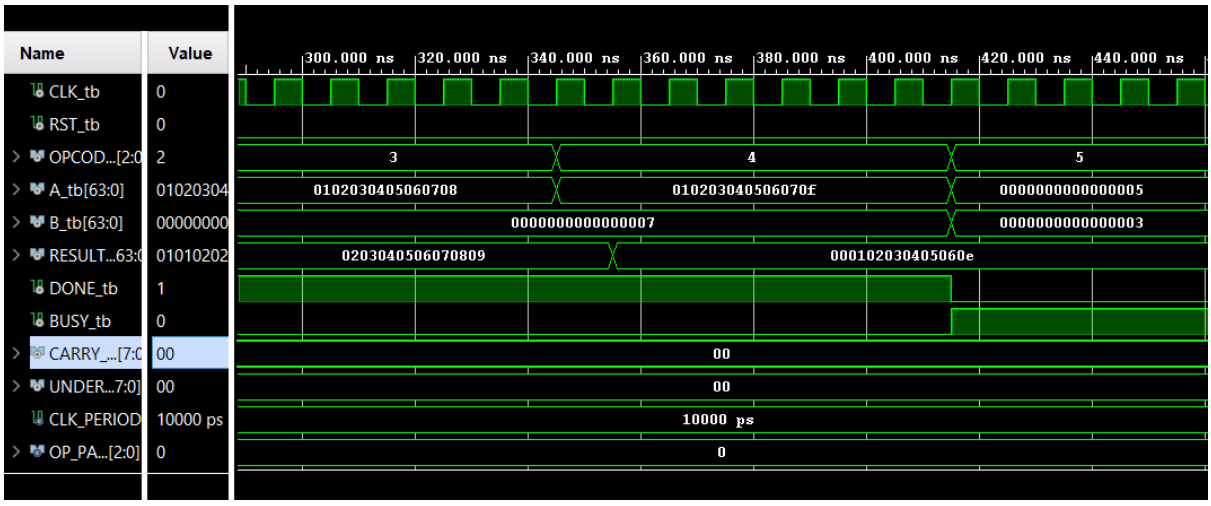
```
-- TEST 6: PMUL
-----------------------------------------------------
report "--- TEST 6: PMUL (Multi-Cycle 5*3=0F) ---" severity NOTE;
OPCODE_tb <= OP_PMUL;
A_tb <= (others => '0'); A_tb(7 downto 0) <= X"05";
B_tb <= (others => '0'); B_tb(7 downto 0) <= X"03";

wait until rising_edge(CLK_tb);

assert DONE_tb = '0'
    report "Test 6 Failed: DONE asserted too early."
    severity ERROR;

wait for CLK_PERIOD * 7;

wait until rising_edge(CLK_tb);

assert DONE_tb = '1'
    report "Test 6 Failed: PMUL (Multi-Cycle) did not assert DONE."
    severity ERROR;
assert RESULT_tb(7 downto 0) = X"0F"
    report "Test 6 Failed: PMUL result (5*3) not 0F."
    severity ERROR;
assert BUSY tb = '0'
```





UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA





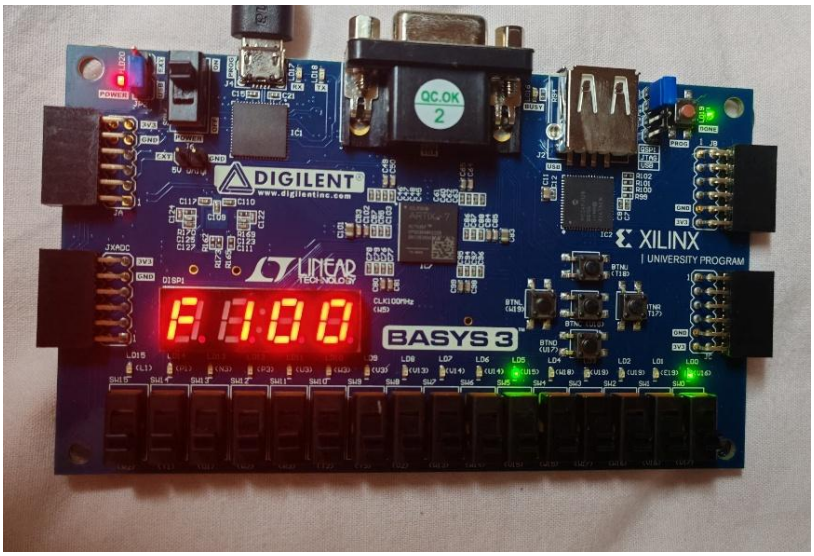
8.2 Actual Board Results

Hardware validation was performed on the Basys 3 FPGA. Unlike simulation, board testing accounts for physical phenomena like button bounce and display flicker. The most important cases (the edge cases) are showcased in the table below:

Operation	**Lane A (Hex)**	**Lane B (Hex)**	**Expected Result**	**Board Result**	**Status / Logic Note**
PADD	FF	01	00	00	**Pass**: Carry generated; lane wraps to 0.
PSUB	00	01	FF	FF	**Pass**: Underflow generated; lane wraps to 255.
PAVG	FF	FE	FE	FE	**Pass**: Truncated integer average (254.5 -> 254).
PMUL	FF	02	FE	FE	**Pass**: 255 x 2 = 510 (01FE). Low byte FE kept.
PINC	FF	N/A	00	00	**Pass**: Max byte wraps to zero.
PDEC	00	N/A	FF	FF	**Pass**: Zero byte wraps to max (255).
PMUL (Zero)	FF	00	00	00	**Pass**: Multiplication by zero validated.

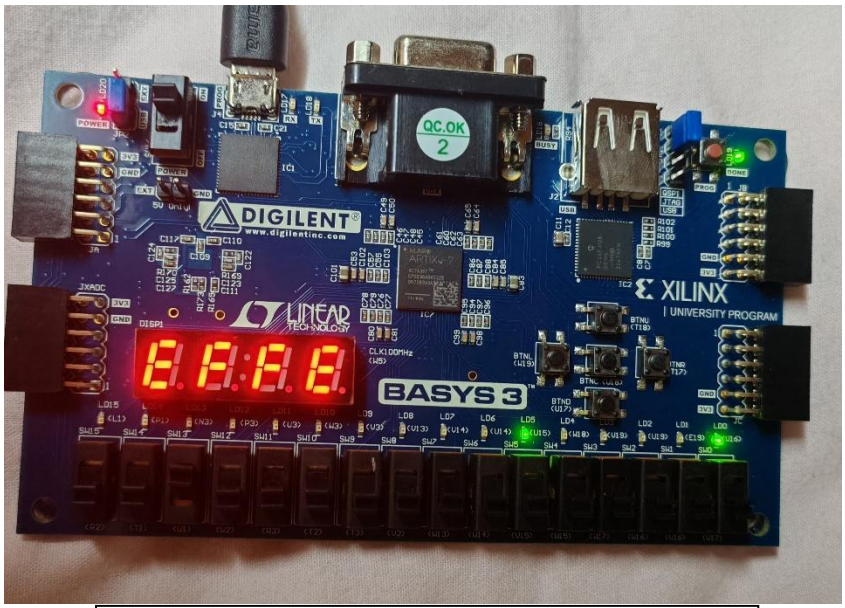
UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

The inputs x"000000000000F0FF" and x"0000000000000101" are used as example for the six operations on the Basys 3 board with the corresponding results as follows:



PADD: x"000000000000F0FF" +
x"0000000000000101" =>

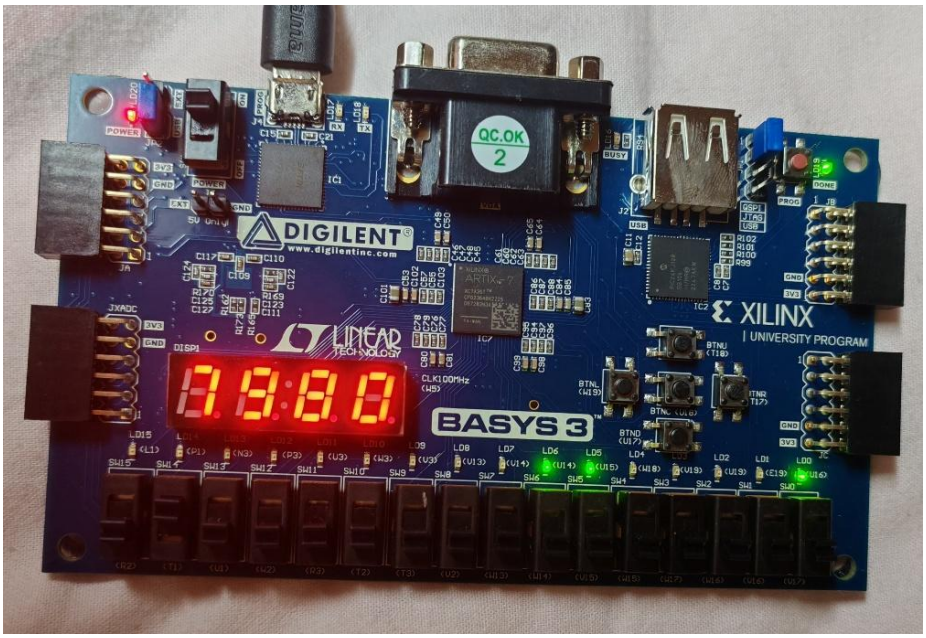
Expected: x"000000000000F100"



PSUB: x"000000000000F0FF" -
x"0000000000000101" =>

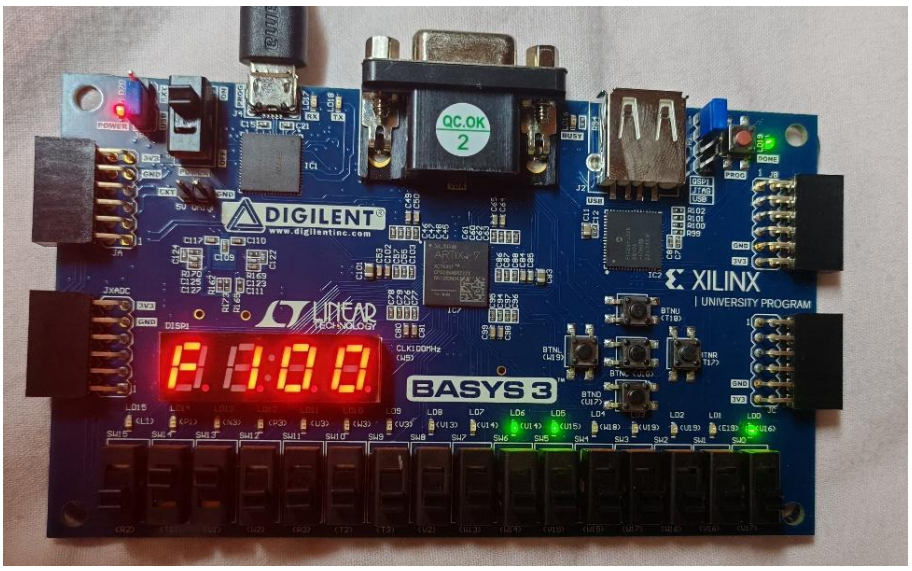
Expected: x"000000000000EFFE"

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA



PAVG: x"000000000000F0FF" avg x"0000000000000101" =>

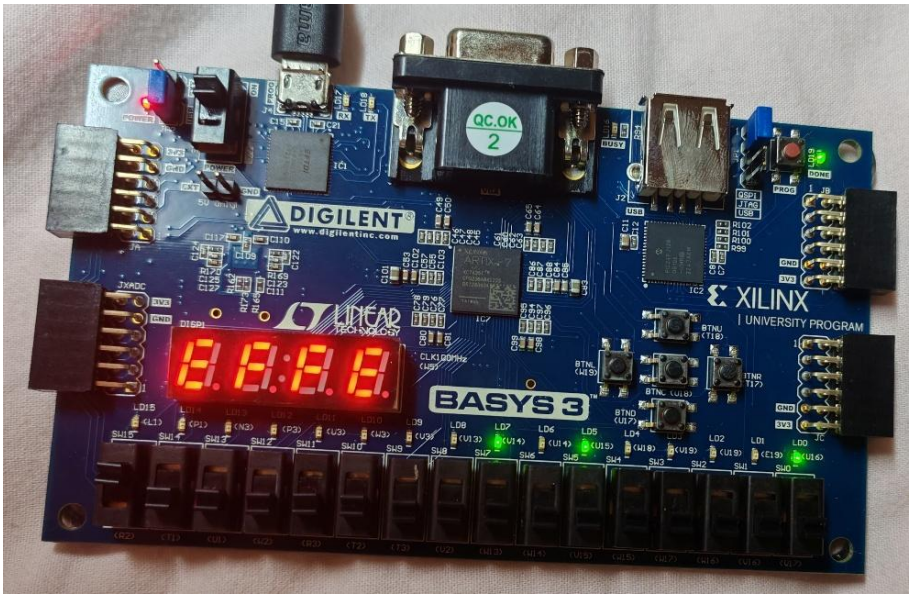
Expected: x"0000000000007980"



PINC: x"000000000000F0FF" =>

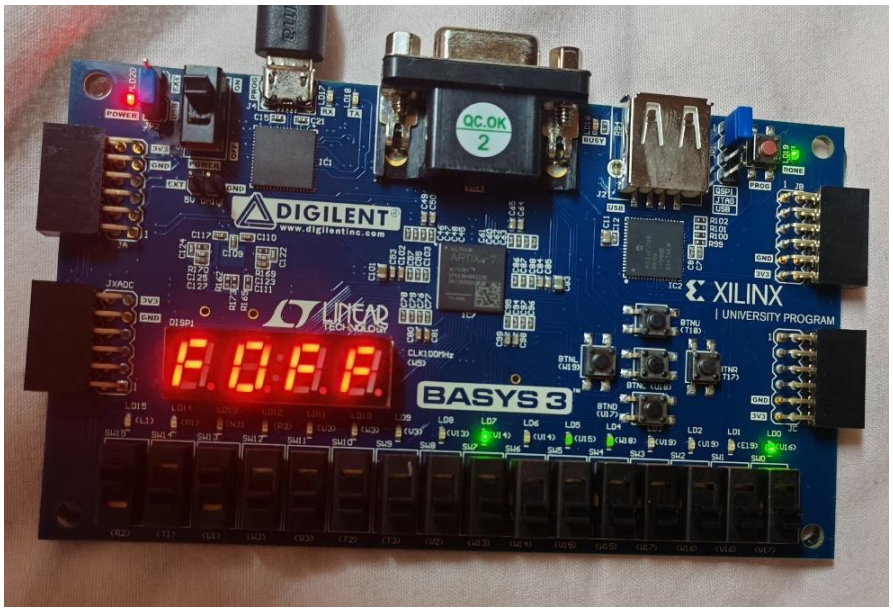
Expected: x"000000000000F100"

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA



PDEC: x"0000000000000F0FF" =>

Expected: x"0000000000000EFFE"



PMUL: x"0000000000000F0FF" *
x"0000000000000101" =>

Expected: x"0000000000000F0FF"

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

9. User Manual: Operation on Basys 3

This manual describes how a user interacts with the MMX Arithmetic Unit on the FPGA.

9.1 Hardware Interface Mapping

- **Inputs:**
  - **BTNC (Center):** Cycles to the next data pair in the ROM.
  - **BTNRST (Right):** Resets the ROM address and ALU state.
  - **SW[15:13]:** Opcode Select (The "Function" dial).
  - **SW[1:0]:** Result Window Select (The "Scroll" dial).
- **Outputs:**
  - **7-Segment Display:** Shows 16 bits of the 64-bit result.
  - **LEDs [3:0]:** Display the current ROM Address in binary.
  - **LED 4:** PMUL Busy indicator.
  - **LED 5:** Operation Done indicator.

9.2 Step-by-Step Operation

1. **Initialize:** Press **BTNRST** to ensure you are at ROM Address 0.
2. **Select Data:** Press **BTNC** to find the pair of numbers you wish to process. Observe the address change on **LEDs 0-3**.
3. **Select Function:** Toggle the leftmost switches to the desired operation:
   - `000`: Addition
   - `001`: Subtraction
   - `010`: Average
   - `011`: Incrementation
   - `100`: Decrementation
   - `101`: Multiplication
4. **Read Result:** Look at the 7-segment display.
   - If the result is larger than 4 hex digits, toggle **SW[0]** and **SW[1]** to see the higher-order bytes.
5. **Multiplication Note:** When selecting `101`, notice **LED 4** flicker briefly as the 8-cycle calculation completes, followed by **LED 5** indicating the result is stable.