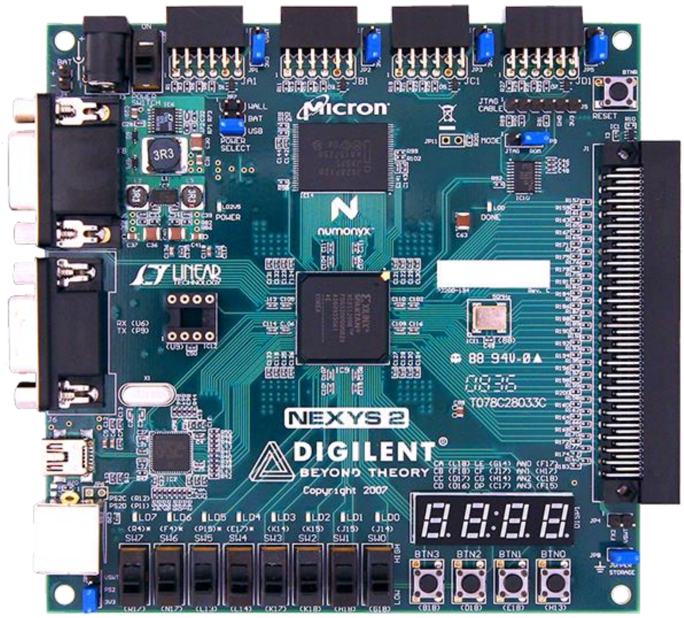
Project 03: Hexadecimal Counter



Prepared by: Muhammad Nuruddin bin Muktaruddin

Date: February 08, 2026

Platform: Digilent Nexys 2 (Spartan-3E FPGA)

Tools: Xilinx ISE 14.7, EDA Playground, Visual Studio, Python 3, Git

Table of Contents

1. Project Aim & Objectives

Aim

To design and implement a 4-bit synchronous counter that counts from 0 to F (Hexadecimal) on the Nexys 2 of 7-segment display, updating automatically every 1 second.

Objectives:

- Hierarchical Design: Break the design into modular components (top, pulse_gen, hex_decoder).
- Sequential Logic: Implement a timing circuit (Pulse Generator) to divide the 50 MHz clock down to 1 Hz.
- IP Reuse: Reuse the verified 7-segment decoder from Project 02.
- Simulation vs. Hardware: Use VHDL Generics to speed up simulation time while maintaining correct hardware timing.
- Verification: Verify the counting sequence using a Python Golden Model and SystemVerilog Testbench.

2. Process Flow (The Engineering Lifecycle)

This project followed a standard industry ASIC/FPGA workflow:

1. **Requirement:** A display that counts 0 → F every second.
2. **Architecture:**
    a. Input: 50 MHz Clock, Reset Button (BTN0).
    b. Logic:
        i. Pulse Gen: Creates a "tick" every 50,000,000 cycles.
        ii. Counter: Increments a 4-bit register when the "tick" happens.
        iii. Decoder: Converts the 4-bit number to 7-segment patterns.
    c. Output: 7-Segment Display (Seg/Anodes).
3. **RTL Design**: Created modular VHDL files (top.vhd, pulse_gen.vhd, hex_decoder.vhd).
4. **Verification:**
    a. Python: Generated the expected truth table (Golden Model).
    b. SystemVerilog: Simulated the design with SIM_SPEED=4 to verify logic transitions.
5. **Implementation:** Synthesized bitstream (top.bit) in Xilinx ISE.
6. **Validation:** Flashed to Nexys 2; verified 1Hz update rate and reset functionality.

3. Directory Structure & File Types

A clean folder structure is critical for collaboration and version control.

Extension	Name	Purpose
.vhd	VHDL Source	Describes the logic circuit behavior.
.ucf	User Constraints	Maps signals (e.g., clk) to physical FPGA pins.
.sv	SystemVerilog	Used for the Testbench (Verify the VHDL).
.py	Python Script	Used for the "Golden Model" (Math verification).
.do	Tcl Script	A macro script to automate the simulator commands.
.bit	Bitstream	The final binary machine code loaded onto the FPGA.

4. Code Analysis (Line-by-Line)

4.1 Hardware Design (rtl/hex_decoder.vhd)

This is the code that lives inside the FPGA.

A. top.vhd (The Manager)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL; -- Needed for adding +1
entity top is
  Port ( clk : in  STD_LOGIC;              -- 50 MHz Clock
      btn : in  STD_LOGIC_VECTOR (0 downto 0); -- Reset Button (BTN0)
      seg : out STD_LOGIC_VECTOR (6 downto 0); -- Segments
      an  : out STD_LOGIC_VECTOR (3 downto 0); -- Anodes
      dp  : out STD_LOGIC);               -- Decimal Point
end top;

architecture Behavioral of top is
  -- Internal Signals (The "Wires" inside the chip)
  signal enable_tick : STD_LOGIC;              -- The 1 Hz Heartbeat
  signal count_reg   : unsigned(3 downto 0) := "0000";  -- The Current Number (0-15)
begin
  -- 1. INSTANTIATE PULSE GENERATOR (The Heart)
  u_pulse : entity work.pulse_gen
  port map (
    clk   => clk,
    rst   => btn(0),
    pulse => enable_tick
  );
```

```
  -- 2. THE COUNTER (The Brain)
  process(clk)
  begin
    if rising_edge(clk) then
      if btn(0) = '1' then
        count_reg <= (others => '0'); -- Reset to 0
      elsif enable_tick = '1' then      -- Only count when Pulse says "GO"
        count_reg <= count_reg + 1;
      end if;
    end if;
  end process;

  -- 3. INSTANTIATE HEX DECODER (The Face)
  -- We map the "count_reg" (Internal) to "sw" (Decoder Input)
  u_decoder : entity work.hex_decoder
  port map (
    sw  => std_logic_vector(count_reg), -- Typecast: Unsigned -> Std_Logic
    seg => seg,
    an  => an,
    dp  => dp
  );

end Behavioral;
```

B. pulse_gen.vhd (The Timer)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity pulse_gen is
   Port ( clk   : in  STD_LOGIC; -- 50 MHz Input
          rst   : in  STD_LOGIC; -- Reset Button
          pulse : out STD_LOGIC); -- 1 Hz "Tick"
end pulse_gen;

architecture Behavioral of pulse_gen is
   -- 50,000,000 cycles = 1 second
   constant MAX_COUNT : integer := 50000000 - 1;
   signal counter     : integer range 0 to MAX_COUNT := 0;
begin
   process(clk)
   begin
      if rising_edge(clk) then
         if rst = '1' then
            counter <= 0;
            pulse   <= '0';
         else
            if counter = MAX_COUNT then
               counter <= 0;
               pulse   <= '1'; -- Fire for 1 cycle!
            else
               counter <= counter + 1;
```

```
            pulse   <= '0';
          end if;
        end if;
      end if;
    end process;
end Behavioral;
```

C. **hex_decoder.vhd (The Translator)**

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity hex_decoder is
    Port ( sw  : in  STD_LOGIC_VECTOR (3 downto 0); -- 4 Switches (Binary Input)
           seg : out STD_LOGIC_VECTOR (6 downto 0); -- 7 Segments (gfedcba)
           an  : out STD_LOGIC_VECTOR (3 downto 0); -- 4 Anodes (Digit Select)
           dp  : out STD_LOGIC);                    -- Decimal Point
end hex_decoder;

architecture Behavioral of hex_decoder is
begin

    -- 1. ANODE CONTROL (Active Low)
    -- We only want the Rightmost Digit (Digit 0) to be ON.
    -- Pattern: 1110 (Digit3=OFF, Digit2=OFF, Digit1=OFF, Digit0=ON)
    an <= "1110";

    -- 2. DECIMAL POINT
    -- Turn it OFF (High Voltage = OFF for Common Anode)
    dp <= '1';
```

```
    -- 3. DECODER LOGIC (The Lookup Table)
    -- Sensitivity List: process(sw) means "Wake up whenever 'sw' changes"
    process(sw)
    begin
        case sw is
            -- Input (Switch)      Output (gfedcba) - Active Low
            when "0000" => seg <= "1000000"; -- 0
            when "0001" => seg <= "1111001"; -- 1
            when "0010" => seg <= "0100100"; -- 2
            when "0011" => seg <= "0110000"; -- 3
            when "0100" => seg <= "0011001"; -- 4
            when "0101" => seg <= "0010010"; -- 5
            when "0110" => seg <= "0000010"; -- 6
            when "0111" => seg <= "1111000"; -- 7
            when "1000" => seg <= "0000000"; -- 8 (Check: All Zeros)
            when "1001" => seg <= "0010000"; -- 9
            when "1010" => seg <= "0001000"; -- A
            when "1011" => seg <= "0000011"; -- b
            when "1100" => seg <= "1000110"; -- C
            when "1101" => seg <= "0100001"; -- d
            when "1110" => seg <= "0000110"; -- E
            when "1111" => seg <= "0001110"; -- F
            when others => seg <= "1111111"; -- OFF (Safety)
        end case;
    end process;

end Behavioral;
```

4.2 Constraints (constraints/nexys2.ucf)

This bridges the code to the physical world.

```
# --- CLOCK ---

NET "clk" LOC = "B8"; # 50 MHz Oscillator

# --- BUTTONS ---

NET "btn<0>" LOC = "B18"; # BTN0 (Used for Reset)

# --- 7-SEGMENT DISPLAY ---

NET "seg<0>" LOC = "L18"; # a

NET "seg<1>" LOC = "F18"; # b

NET "seg<2>" LOC = "D17"; # c

NET "seg<3>" LOC = "D16"; # d

NET "seg<4>" LOC = "G14"; # e

NET "seg<5>" LOC = "J17"; # f

NET "seg<6>" LOC = "H14"; # g

NET "dp"    LOC = "C17"; # Decimal Point

# --- ANODES ---

NET "an<0>" LOC = "F17"; # Digit 0

NET "an<1>" LOC = "H17"; # Digit 1

NET "an<2>" LOC = "C18"; # Digit 2

NET "an<3>" LOC = "F15"; # Digit 3
```

Output:

The design was programmed onto the Nexys 2 board.

1. Default State: Upon programming top.bit, the rightmost 7-segment digit (Digit 0) displays "0".

2. Counting Sequence:

- The digit increments automatically every 1 second (1 Hz).
- Sequence: 0 → 1 → 2 → ... → 9 → A → b → C → d → E → F.
- Rollover: After F, the counter correctly resets to 0 and continues.

3. Reset Logic (BTN0):

- Action: Pressing and holding BTN0.
- Result: The display immediately resets to "0" and stops counting.
- Release: Upon releasing BTN0, the counter resumes counting from 0.

4.3 Automation Script (verification/run.do)

Used in EDA Playground to run the simulation automatically.

TCL

```
vlib work        #Creates a workspace.
vcom -work work ../rtl/hex_decoder.vhd     #Compiles the VHDL files (Design).
vcom -work work ../rtl/pulse_gen.vhd       #Compiles the VHDL files (Design).
vcom -work work ../rtl/top.vhd       #Compiles the VHDL files (Design).
vlog -work work testbench.sv       #Compiles the SystemVerilog file (Testbench).
vsim +access+r -c testbench       #Runs the simulation.
run -all
```

4.4 Python Golden Model (verification/blinky_model.py)

This script calculates the expected behaviour using software logic.

Python Code:

```python
print("--- Golden Model Output ---")
print("Count | Hex Code (seg) | Display")
print("-" * 35)
# Dictionary of expected Hex values (Active Low)
# 0 = ON, 1 = OFF
patterns = {
  0: "40", 1: "79", 2: "24", 3: "30",
  4: "19", 5: "12", 6: "02", 7: "78",
  8: "00", 9: "10", 10:"08", 11:"03",
  12:"46", 13:"21", 14:"06", 15:"0E"
}

for i in range(16):
  print(f"  {i:2}  |    {patterns[i]}    |  {i:X}")
```

Output:

```
--- Golden Model Output ---
Count | Hex Code (seg) | Display
----------------------------------
  0 |    40    |  0
  1 |    79    |  1
  2 |    24    |  2
  3 |    30    |  3
  4 |    19    |  4
  6 |    02    |  6
  8 |    00    |  8
  9 |    10    |  9
 10 |    08    |  A
 11 |    03    |  B
 12 |    46    |  C
 14 |    06    |  E
```

Remarks:

The Python script predicts the 7-segment values.

- Row 0: Shows 40. This corresponds to segments 1000000 (Active Low), which visually forms a "0".
- Row 1: Shows 79. This corresponds to "1".
- This confirms our look-up table logic is mathematically correct.

4.5 UVM Testbench (verification/testbench.sv)

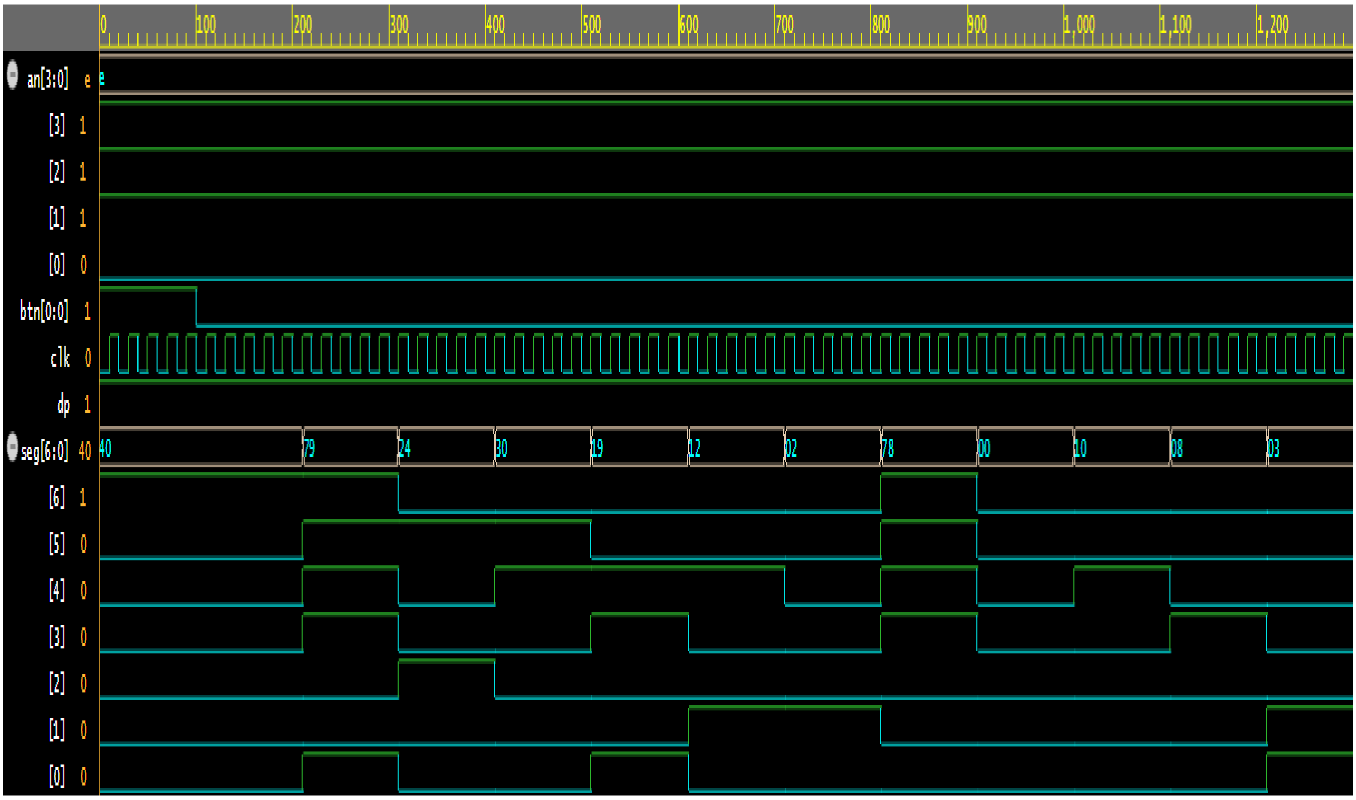
The SystemVerilog testbench wraps around the VHDL to simulate it.

Code snippet:

```
module testbench;

 // 1. Signals
 logic clk;
 logic [0:0] btn;
 logic [6:0] seg;
 logic [3:0] an;
 logic dp;

 // 2. Instantiate DUT (Design Under Test)
 // CRITICAL: We override the Generic to 4 for super-fast simulation!
 top #(.SIM_SPEED(4)) u_top (
  .clk(clk),
  .btn(btn),
  .seg(seg),
  .an(an),
  .dp(dp)
 );

 // 3. Clock Generation (50 MHz = 20ns period)
 initial begin
  clk = 0;
  forever #10 clk = ~clk; // Toggle every 10ns
 end
```

```
 // 4. Test Sequence
 initial begin
  // Dump waves for EPWave
  $dumpfile("dump.vcd");
  $dumpvars(0, testbench);

  $display("Starting Simulation...");

  // Reset
  btn[0] = 1;
  #100;
  btn[0] = 0;
  $display("Reset Released. Counting begins.");

  // Wait for 20 counts (should see 0 -> F -> 0 -> 4)
  // Since SIM_SPEED=4, one count = 5 clocks = 100ns.
  // 20 counts * 100ns = 2000ns.
  #2500;

  $display("Simulation Finished.");
  $finish;
 end

endmodule
```

Output:



- Cyan Trace (seg[6:0]): We see the value transition from 40 → 79 → 24. Comparing this to the Python table: 40="0", 79="1", 24="2".
- Transition Timing: The values change rapidly (every 5 clock cycles) because we set SIM_SPEED = 4.
- Conclusion: The logic works perfectly. The counter increments correctly and rolls over from 0E (F) to 40 (0).

5. Software & Tools Settings

5.1 EDA Playground (Simulation)

- **Simulator:** Aldec Riviera-PRO 2025.04.
- **Top Entity:** testbench.sv
- **Open EPWave:** Checked (to view waveforms).
- **Files:** All 3 VHDL files + 1 SV file must be in the project root.

5.2 Xilinx ISE 14.7 (Implementation)

- **Family:** Spartan3E
- **Device:** XC3S500E
- **Package:** FG320
- **Speed:** -4
- **Top Module:** top.vhd
- **Rtl files:** top.vhd, pulse_gen.vhd, hex_decoder.vhd
- **Process:** Synthesize → Implement Design → Generate Programming File.
- **Note:** Use "Add Copy of Source" to prevent modifying the original repository files during experimentation.

6. Troubleshooting & Lessons Learned

Throughout Project 3, several critical engineering lessons were learned:

1. **Hierarchy Management:** Learned that top.vhd is just a wiring diagram. If logic is wrong, check the sub-modules (pulse_gen).
2. **Simulation Timeout:** Initially, simulation would "hang" because 50 million cycles take too long to simulate. Solution: Used VHDL Generics to reduce the count limit during simulation.
3. **Git hygiene:** Learned to commit .vhd files before experimenting in ISE to avoid losing the generic parameters.