Introduction

This report documents the design, simulation, synthesis, and testing of a two-digit BCD comparator system implemented on the Nexys A7 FPGA. The lab exercise required the design of registers, a comparator, validation logic, multiplexed seven-segment display drivers, and RGB LED outputs. The key learning outcomes included experience with structural VHDL, hierarchical design, SSD multiplexing, and on-board debugging. The final design was successfully tested on hardware with only minor implementation issues.

Design Description

The design was implemented using a structural VHDL approach. The top-level block diagram is shown in Figure 1 below.

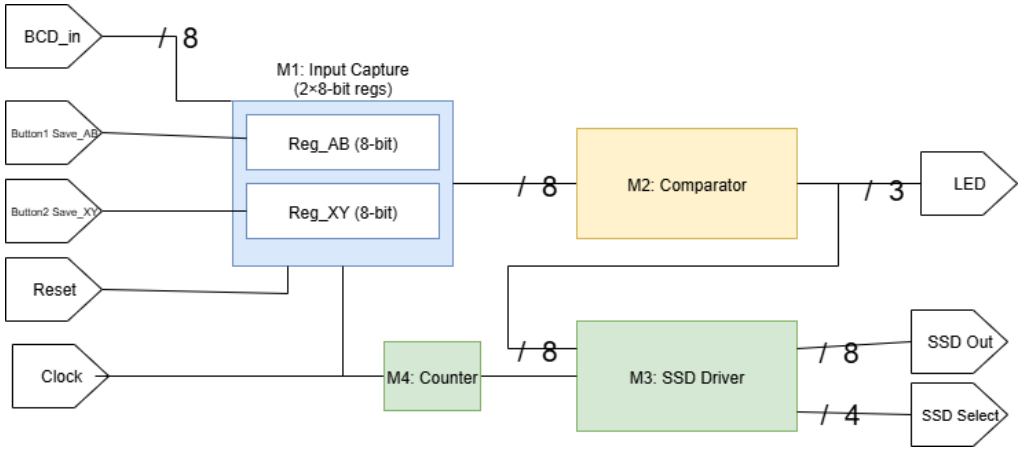

Figure 1 System Block Diagram

Each subsystem corresponds to a VHDL entity:

- EightBitReg: 8-bit register with clock enable and reset
- bcd_validator: checks each 4-bit input for valid BCD (0–9)
- bcd_comparator: compares two 8-bit BCD inputs and reports match type
- display_controller: generates SSD digit outputs and RGB LED signals
- anode_decoder: selects active SSD digit based on counter
- mux4x4: selects which digit to display
- hex_to_ssd_const: maps hex digit to SSD cathode pattern

Which can be further looked at in the appendix (A-G).

As shown in Figure 1, the system has **four main inputs** and **three main outputs**:

Inputs

- clk: the 100 MHz onboard system clock used for registers, counters, and SSD multiplexing.
- reset: active-high reset signal, clearing the registers and resetting the counter.
- btn1 and btn2: pushbuttons used as clock-enables for the registers, allowing values from the switches to be latched.

- switches[7:0]: eight onboard slide switches used to define the two BCD values (two digits each).

Outputs

- ssd_cathodes[6:0]: the seven cathode lines (A–G) for the multiplexed SSDs.
- ssd_anodes[3:0]: the eight anode enable lines, of which only AN0–AN3 (the right-hand display) are active.
- rgb_led[2:0]: a tri-colour LED indicating comparator results (e.g. red = no match, green = full match, yellow = partial match).

Each button (btn1, btn2) is connected to the enable input of one of the two registers, allowing the 8-bit value from the switches to be stored as BCD1 or BCD2. The registers reset to zero when reset is asserted, meaning that immediately after reset the RGB LED indicates a full match (green) and the SSDs display “HHHH” as both registers contain the same value.

Both the upper and lower digits of each register are passed through a validator to check whether the nibble is greater than 9, raising a flag if an invalid number is detected. The comparator takes the two 8-bit BCD values and produces match_type (00 = no match, 01 = partial, 10 = full match) and digit_matches (flags indicating which digits are equal), allowing the system to distinguish partial matches.

The display controller uses the comparator outputs and validator flags to generate the 4-digit output pattern. Valid BCD digits are displayed normally, invalid digits are shown as “E”, and matching digits may be replaced with “H”. The controller also drives the RGB LED colour according to the comparator results.

The digit counter divides the 100 MHz clock and generates a 2-bit count to cycle through the active SSD anodes, ensuring that all four digits are refreshed rapidly enough to appear simultaneously lit. The hex_to_ssd_const module converts each 4-bit nibble into the active-low cathode pattern needed to drive the SSD, while the mux4x4 selects which digit is displayed on the SSD at each time step.

Simulation Results

Simulation was performed in Vivado. Four scenarios were exercised: full match, partial match, no match, and reset. Because the SSD is multiplexed, the **anodes** (active-low) cycle through the four digits, while the **cathodes** (active-low A..G) show the segment pattern for whichever anode is currently active. In our mapping, the letter **H** corresponds to 0x48 on the 7-bit cathode bus (A..G).

Full match - LED green, HHHH on SSD (Figure 2).
Both registers were set to `0x34`. First, `btn1` latched the switches into the A/B register; the RGB LED briefly appeared yellow while only one side was stored. When `btn2` latched the same value into the X/Y register (≈15 ms), the LED moved to **green**, indicating a full match. The cathodes hold at `0x48` as each anode is selected, so the multiplexed display reads **H H H H**.

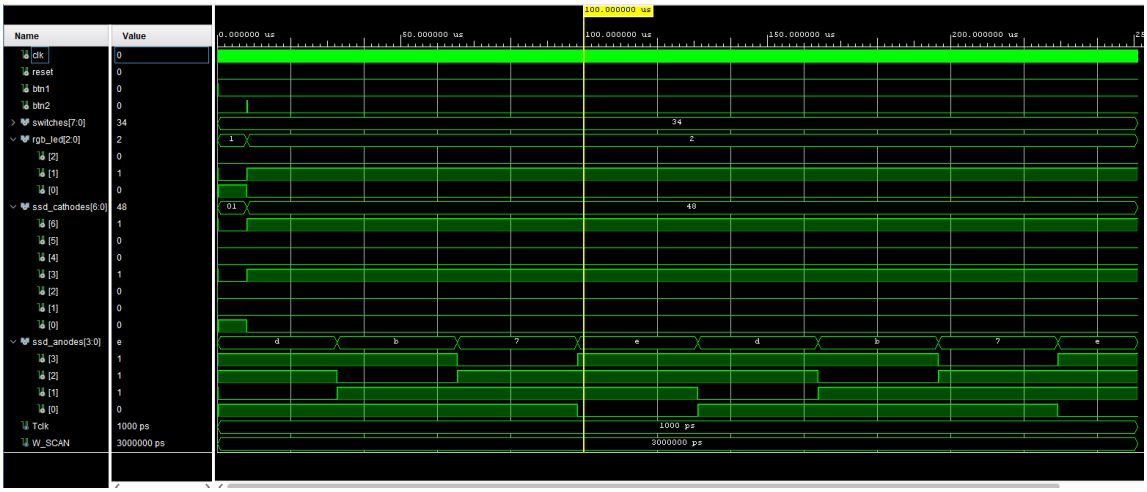

Figure 2 Full match Simulation

Partial match - LED yellow, matched digit shows H (Figure 3).
A/B was set to `0x35`, then X/Y to `0x34`. The LED moved to **yellow**, signalling a partial match (one digit equal). On the active anode slot for the matching digit the cathodes present **H** (`0x48`); on the non-matching slots they present the normal numeral patterns (e.g. `0x24` and `0x4C` for “5” and “4” in this build). The result on the SSD is **H** over the equal digit and the original numerals elsewhere.

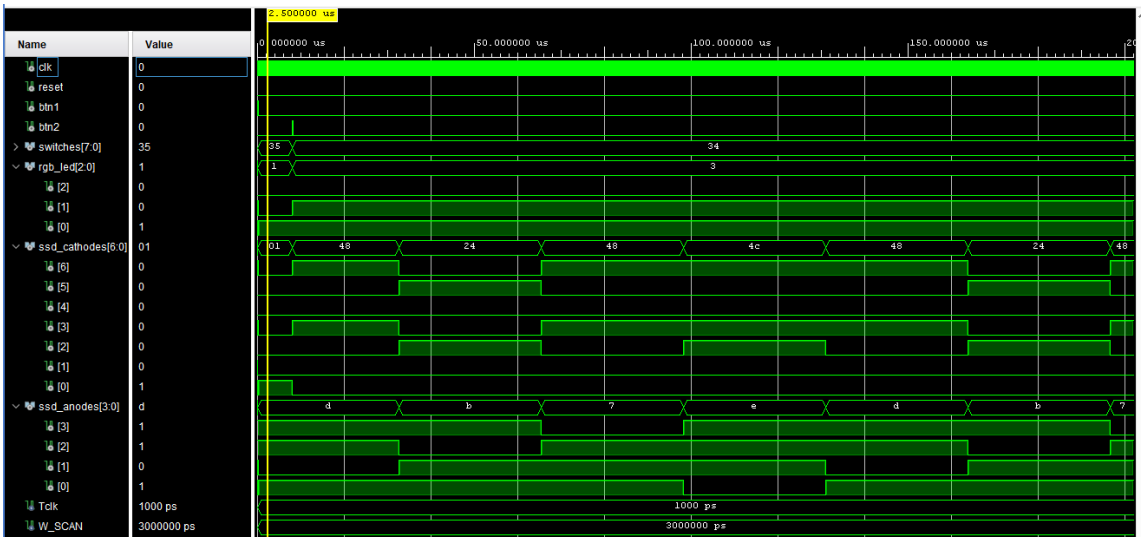

Figure 3 Half match Simulation

No match - LED red, numerals ABXY on SSD (Figure 4).
A/B was set to `0x12` and X/Y to `0x34`. With no equal digits the LED remained **red** and each anode slot showed its corresponding numeral (e.g. a repeating sequence such as `0x06`, `0x12`, `0x4F`, `0x4C` for “1”, “2”, “3”, “4” per the chosen font).

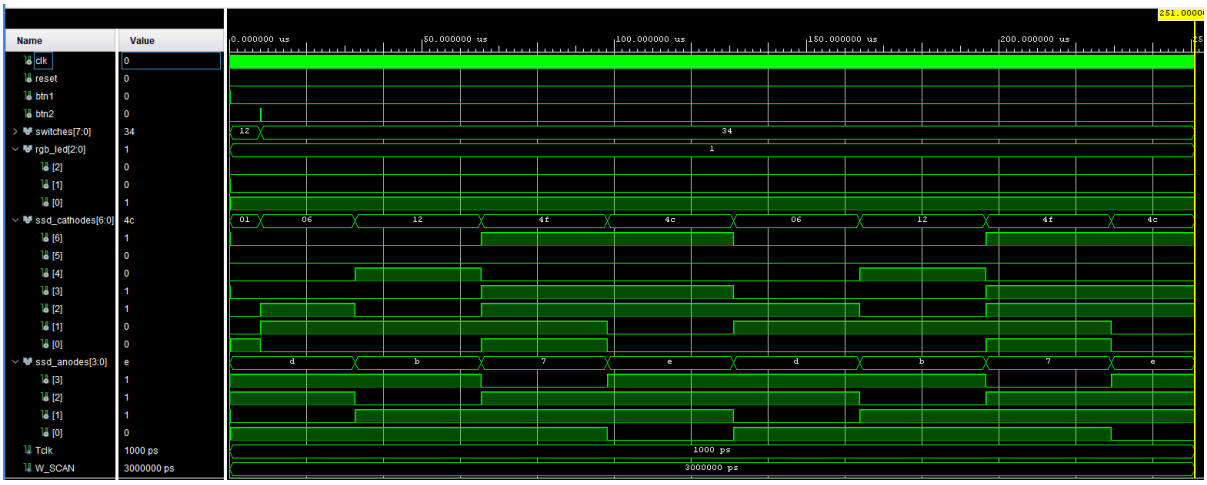

Figure 4 No match Simulation

Reset - registers cleared, LED green, HHHH on SSD (Figure 5).
Asserting `reset` clears both 8-bit registers to `00`. Because the numbers are identical after reset, the LED indicates **green** and all four digits display **H** (`0x48` on the cathodes during each anode window).

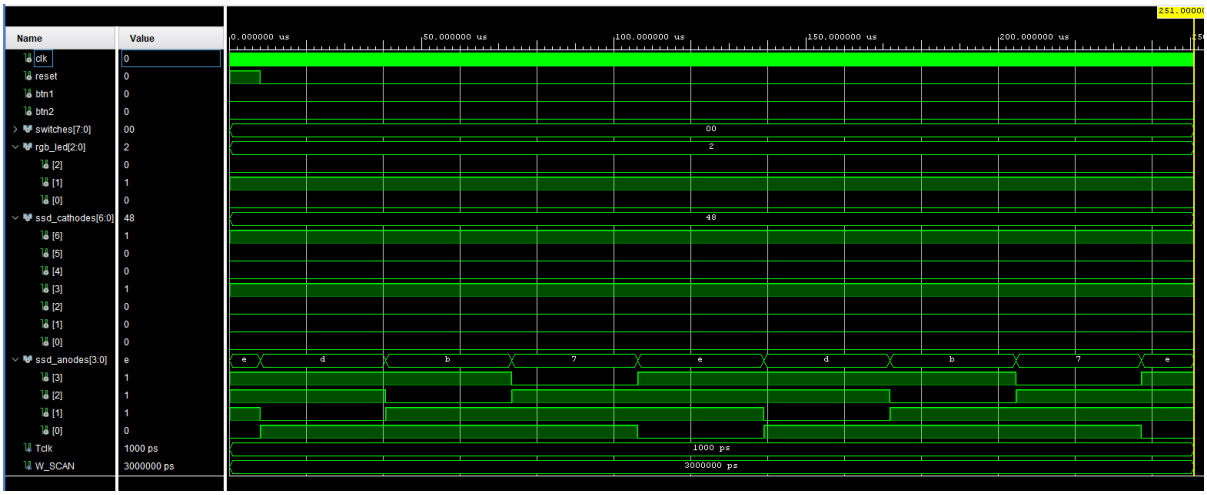

Figure 5 Reset Simulation

Synthesis/Implementation Results

After synthesising the design in Vivado, both the schematic views and the utilisation reports confirm that the system was implemented as expected. After synthesising the design in Vivado, both the schematic views and the utilisation reports confirm that the system was implemented as expected.

FigThe *RTL schematic* (Figure 6) clearly shows the modular structure of the top-level design. Each entity from the structural VHDL (two 8-bit registers, validators, comparator, display controller, digit counter, multiplexer, SSD decoder, and anode decoder) appears as a distinct block with the expected signal connectivity. The *synthesis schematic* (Figure 7) shows the lower-level netlist after synthesis, where each VHDL module is realised in terms of primitive flip-flops (FDREs), look-up tables (LUTs), and buffers. This view highlights how the abstract structural description is mapped onto the FPGA's fabric.

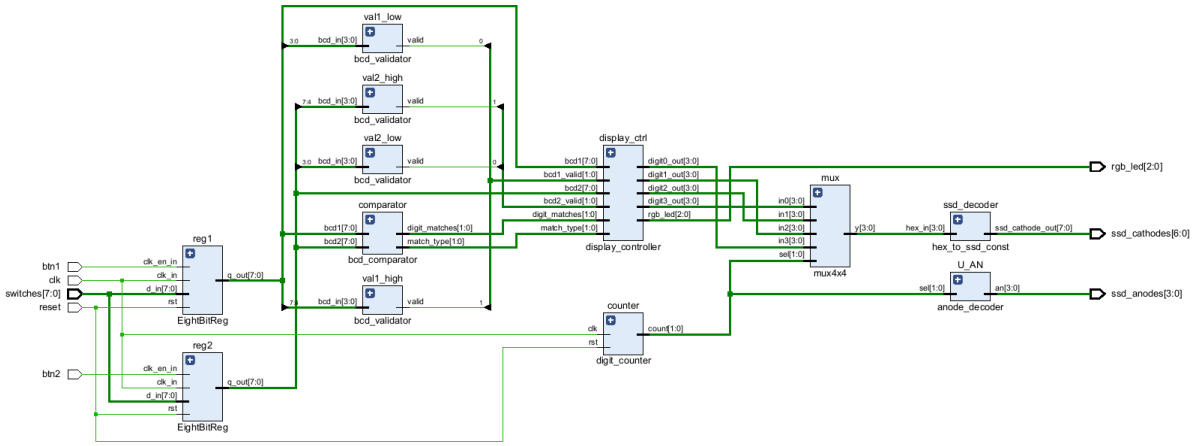

Figure 6 RTL schematic

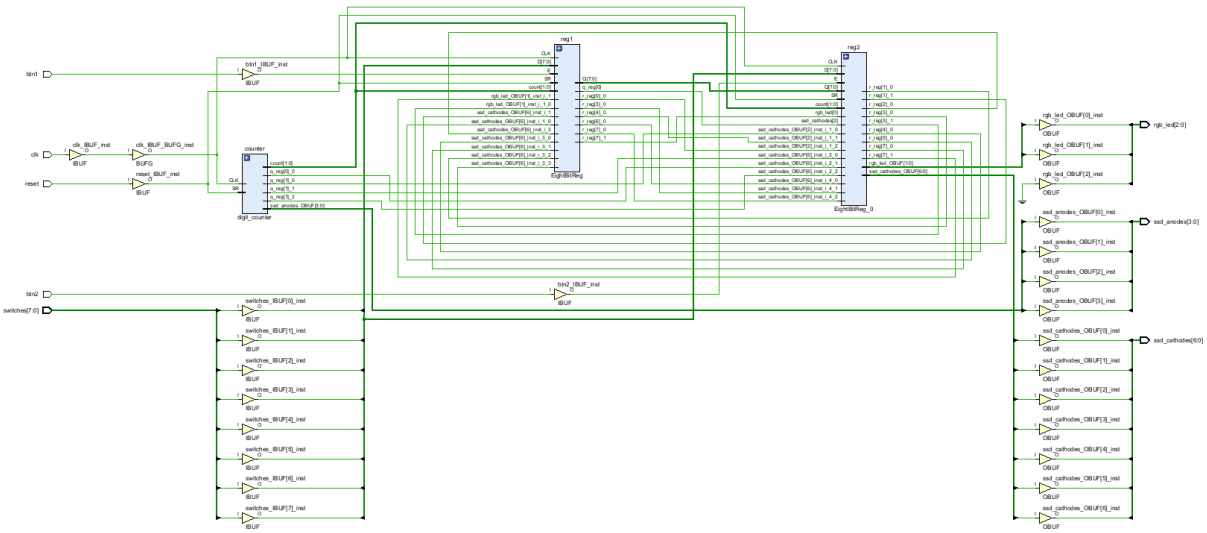

Figure 7 Synthesis Schematic

The primitive report (Figure 8) shows that the design uses 33 FDRE flip-flops, a collection of LUTs of various sizes, 1 BUFG for the system clock, and standard input/output buffers (IBUFs and OBUFs). The presence of CARRY4 elements reflects efficient implementation of arithmetic/comparison logic in the comparator and counter modules.

8. Primitives

Ref Name	Used	Functional Category
FDRE	33	Flop & Latch
LUT6	25	LUT
OBUF	14	IO
IBUF	12	IO
LUT4	9	LUT
LUT3	9	LUT
LUT2	9	LUT
CARRY4	4	CarryLogic
LUT5	3	LUT
LUT1	1	LUT
BUFG	1	Clock

Figure 8 FPGA resource summary (Primitives table)

The utilisation by module (Figure 9) shows that the majority of LUTs are used in the two EightBitReg blocks and the comparator logic, while the counter consumes the largest share of registers (17 out of 33). This matches expectations, since the counter needs a bank of flip-flops to implement the scan divider, and the registers must each hold 8 bits of input data.

Name	Slice LUTs (63400)	Slice Registers (126800)	Bonded IOB (210)	BUFGCTRL (32)
bcd_comparator_top	44	33	26	1
counter (digit_counter)	9	17	0	0
reg1 (EightBitReg)	9	8	0	0
reg2 (EightBitReg_0)	26	8	0	0

Figure 9 Detailed Slice Logic Table

The utilisation summary (Figure 10) demonstrates that the design uses a negligible fraction of the available FPGA resources: The utilisation summary demonstrates that the design uses a negligible fraction of the available FPGA resources. The IO utilisation is higher in relative terms because the design makes heavy use of board peripherals (switches, push-buttons, LEDs, and SSD connections), but the overall logic utilisation is extremely low.

Summary

Resource	Utilization	Available	Utilization %
LUT	44	63400	0.07
FF	33	126800	0.03
IO	26	210	12.38

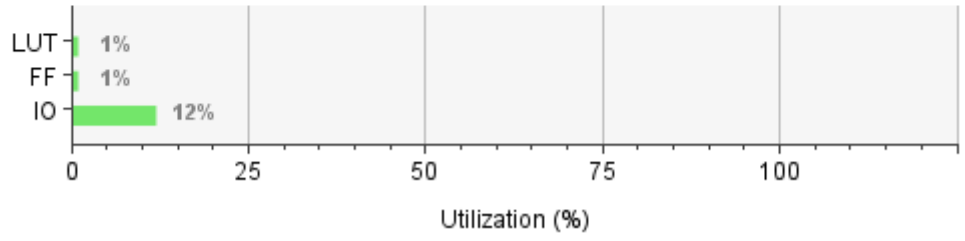

Figure 10 Utilisation Summary Chart

On-Board Testing

The design was tested on the Nexys A7 FPGA board. The right-hand four-digit SSD was used.

Testing confirmed that:

- Full match → LED green, both digits show H

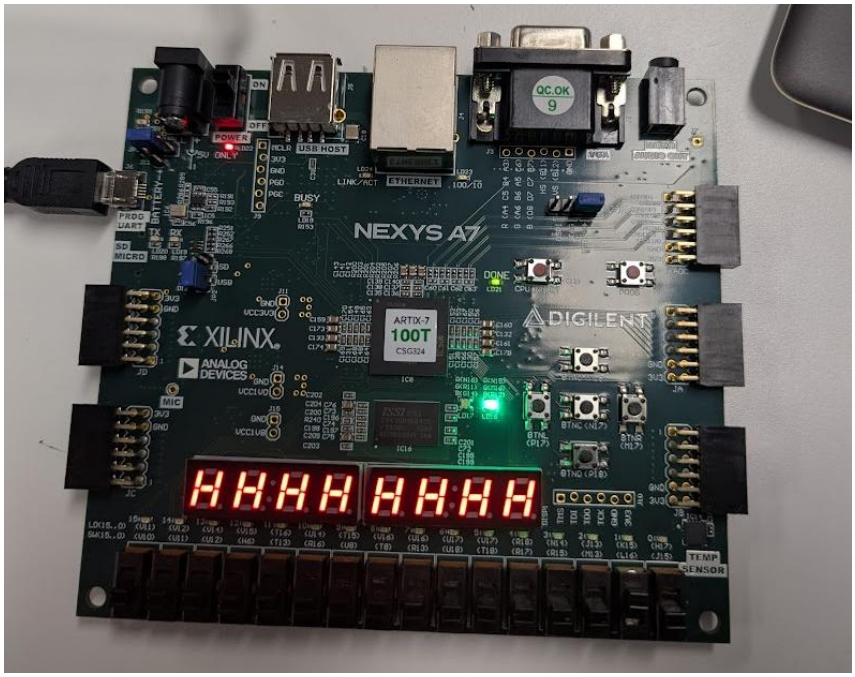

Figure 11 Full match on board

- Partial match → LED yellow, only the matching digit shows H

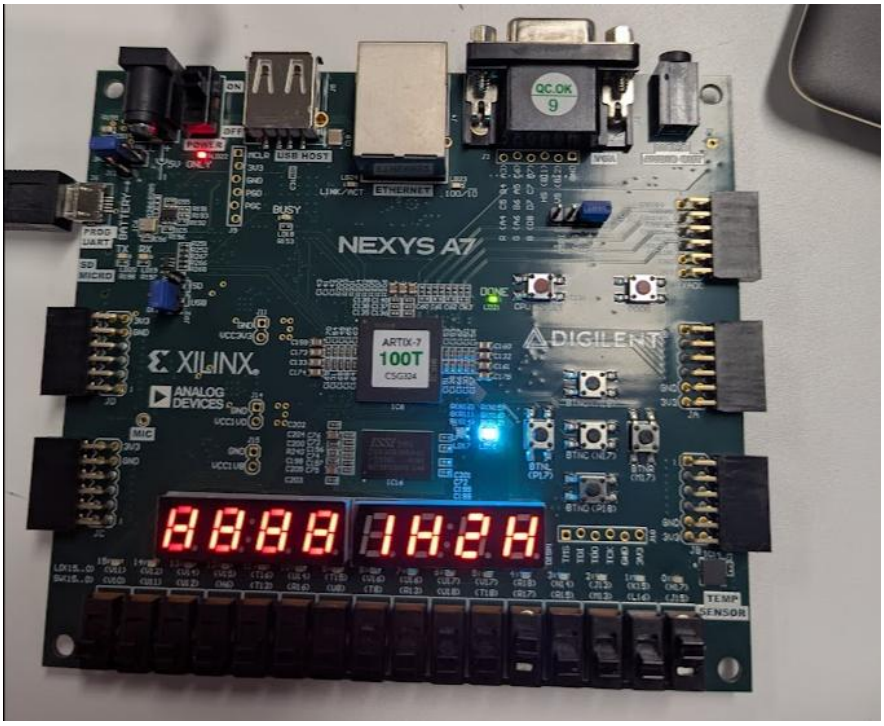

Figure 12 Half match on board

- No match → LED red, digits show ABXY, where Y is an digit more than 9 showing “E”

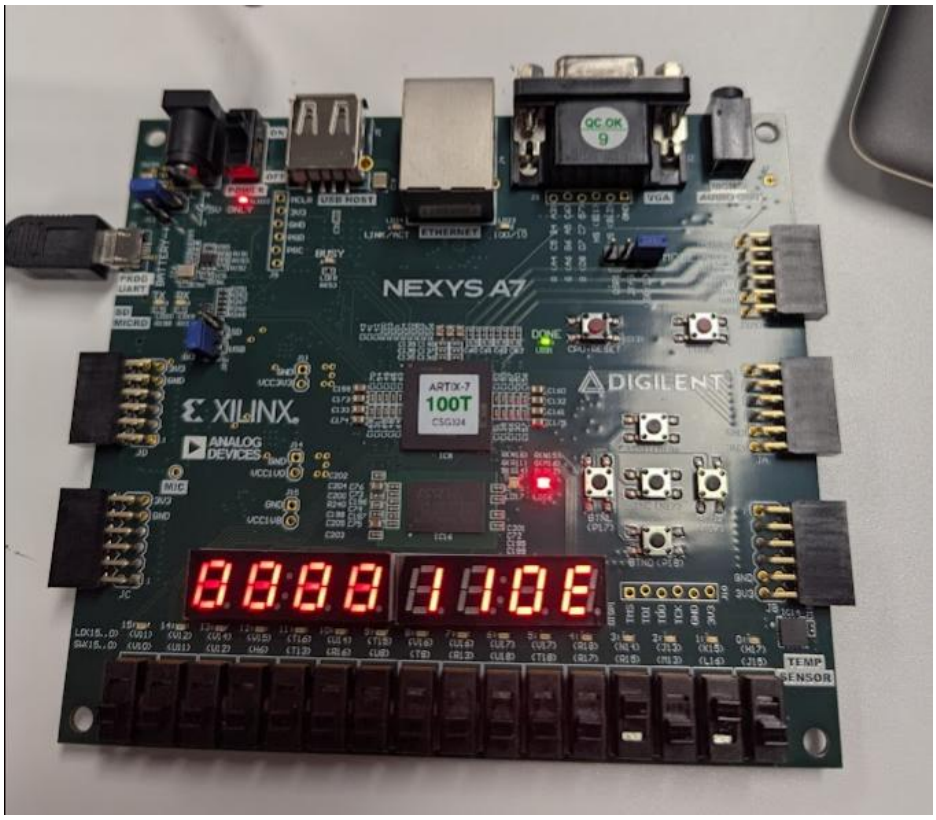

Figure 13 No match on board

- Reset clears registers

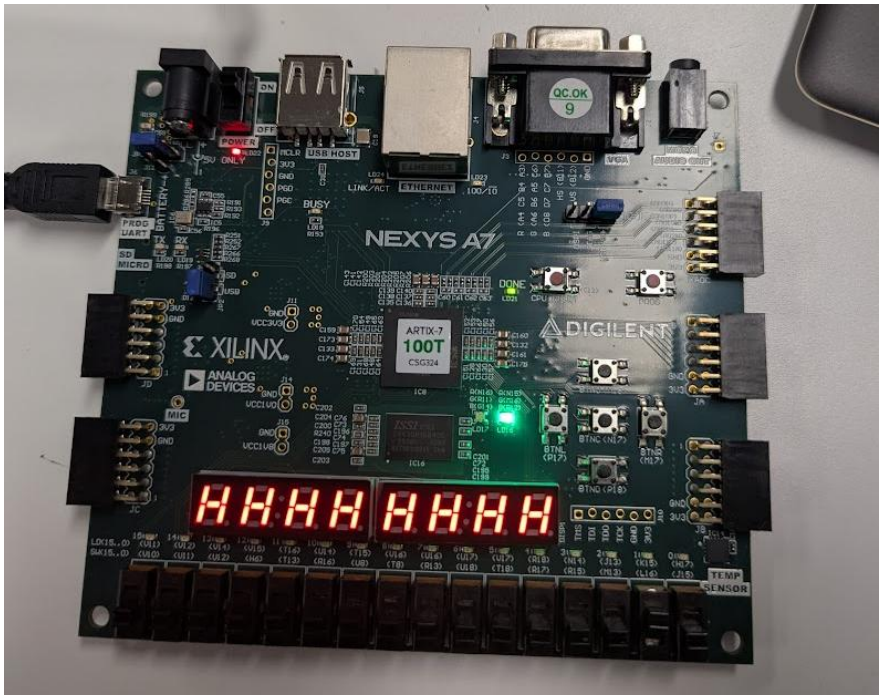

Figure 14 Reset on board

Discussion and Conclusion

The lab successfully demonstrated the use of a modular VHDL design to implement a multi-component digital system. By structuring the system into separate entities such as registers, validators, comparator, display controller, multiplexer, and SSD decoder the design process was simplified and each module could be independently developed, simulated, and tested. This modular approach aligns with good hardware engineering practice and reflects how large FPGA designs are normally partitioned.

Several challenges were encountered and resolved during development. One challenge was the correct ordering of SSD digits: the logical digit outputs from the controller did not initially align with the physical anode scan order, which caused characters to appear scrambled across the display. This was resolved by deliberately cross-mapping the multiplexer inputs so that each digit was displayed in the intended position. Another issue was the mapping of the RGB LED bits, where red/green/blue ordering needed to be confirmed against the board documentation to ensure that “green” for a full match and “yellow” for partial match displayed as expected. A further practical challenge was the presence of two banks of SSD anodes on the Nexys A7; only the right-hand bank (AN0–AN3) was used, while the left-hand bank remained unused but was initially being unintentionally driven.

In addition, the reset functionality can be refined. In the current implementation, asserting reset clears both registers to zero, which causes the LED to indicate a full match (green) and all SSD digits to display “H”. A more intuitive behaviour would be for reset to instead drive the LED red to indicate “no data loaded”, while simultaneously blanking the SSD cathodes to ensure no misleading characters are displayed. This change would make the system state on reset clearer to the user. Furthermore, the unused left-hand SSD bank should be explicitly disabled so that it remains blank at all times, improving clarity and avoiding potential confusion during demonstrations.

In conclusion, the design met its functional requirements by correctly detecting full, partial, and no matches, driving the RGB LED and SSD outputs accordingly. The lab reinforced practical FPGA skills in structural modelling, simulation, and hardware verification. The suggested improvements enhanced reset behaviour, explicit disabling of unused SSD digits, debouncing of inputs, and scalability of the comparator represent natural next steps that would increase robustness, usability, and reusability of the system.

References

Chat-GPT was used to guided me through coding, debugging, and report writing, with particular emphasis on SSD multiplexing and digit alignment, simulation best practices in Vivado, and clearer documentation and report structure.

https://chatgpt.com/share/68aae1b7-cd5c-8009-a377-91558378b06d

Appendix

A – Eight bit register VHDL description

```
34  entity EightBitReg is
35    port (
36      d_in      : in  std_logic_vector(7 downto 0);
37      clk_in    : in  std_logic;
38      clk_en_in : in  std_logic;
39      rst       : in  std_logic;                      -- active-high, synchronous
40      q_out     : out std_logic_vector(7 downto 0)
41    );
42  end EightBitReg;
43
44  architecture Behavioral of EightBitReg is
45    signal r : std_logic_vector(7 downto 0) := (others => '0');
46  begin
47    process(clk_in)
48    begin
49      if rising_edge(clk_in) then
50        if rst = '1' then
51          r <= (others => '0');          -- clear register
52        elsif clk_en_in = '1' then
53          r <= d_in;                     -- latch input
54        end if;
55      end if;
56    end process;
57
58    q_out <= r;
59  end Behavioral;
```

B – bcd Validator VHDL description

```
5   entity bcd_validator is
6       Port (
7           bcd_in : in STD_LOGIC_VECTOR (3 downto 0);
8           valid : out STD_LOGIC
9       );
10  end bcd_validator;
11
12  architecture Behavioral of bcd_validator is
13  begin
14      process(bcd_in)
15      begin
16          if unsigned(bcd_in) <= 9 then
17              valid <= '1';
18          else
19              valid <= '0';
20          end if;
21      end process;
22  end Behavioral;
```

C – bcd Comparator VHDL description

```vhdl
entity bcd_comparator is
    Port (
        bcd1 : in STD_LOGIC_VECTOR (7 downto 0);  -- First 2-digit BCD (AB)
        bcd2 : in STD_LOGIC_VECTOR (7 downto 0);  -- Second 2-digit BCD (XY)
        match_type : out STD_LOGIC_VECTOR (1 downto 0); -- 00=no match, 01=partial, 10=full
        digit_matches : out STD_LOGIC_VECTOR (1 downto 0) -- bit 1: high digit match, bit 0:
    );
end bcd_comparator;

architecture Behavioral of bcd_comparator is
    signal high_digit_match, low_digit_match : STD_LOGIC;
begin
    -- Compare high digits (A vs X)
    high_digit_match <= '1' when bcd1(7 downto 4) = bcd2(7 downto 4) else '0';

    -- Compare low digits (B vs Y)
    low_digit_match <= '1' when bcd1(3 downto 0) = bcd2(3 downto 0) else '0';

    digit_matches <= high_digit_match & low_digit_match;

    -- Determine match type
    process(high_digit_match, low_digit_match)
    begin
        if high_digit_match = '1' and low_digit_match = '1' then
            match_type <= "10"; -- Full match
        elsif high_digit_match = '1' or low_digit_match = '1' then
            match_type <= "01"; -- Partial match
        else
            match_type <= "00"; -- No match
        end if;
    end process;
end Behavioral;
```

D – Anode decoder VHDL description

```vhdl
entity anode_decoder is
  port (
    sel : in  std_logic_vector(1 downto 0);
    an  : out std_logic_vector(3 downto 0)   -- active-low
  );
end;

architecture rtl of anode_decoder is
begin
  with sel select
    an <= "1110" when "00",
          "1101" when "01",
          "1011" when "10",
          "0111" when others;
end;
```

E – Display Controller VHDL description

```vhdl
architecture Behavioral of display_controller is
    constant H_PATTERN : STD_LOGIC_VECTOR(3 downto 0) := "1111"; -- Use F to represent 'H'
    constant E_PATTERN : STD_LOGIC_VECTOR(3 downto 0) := "1110"; -- Use E for error
begin
    process(bcd1, bcd2, match_type, digit_matches, bcd1_valid, bcd2_valid)
    begin
        -- Default RGB LED to red
        rgb_led <= "001"; -- Red

        -- Check for invalid BCD inputs
        if bcd1_valid /= "11" or bcd2_valid /= "11" then
            -- Error case - show 'E' for invalid digits
            rgb_led <= "001"; -- Red
            -- Show E for invalid digits, actual digits for valid ones
            if bcd1_valid(1) = '0' then
                digit3_out <= E_PATTERN;
            else
                digit3_out <= bcd1(7 downto 4);
            end if;

            if bcd1_valid(0) = '0' then
                digit2_out <= E_PATTERN;
            else
                digit2_out <= bcd1(3 downto 0);
            end if;

            if bcd2_valid(1) = '0' then
                digit1_out <= E_PATTERN;
            else
                digit1_out <= bcd2(7 downto 4);
            end if;

            if bcd2_valid(0) = '0' then
                digit0_out <= E_PATTERN;
            else
                digit0_out <= bcd2(3 downto 0);
            end if;

        else
            -- Valid BCD inputs
            case match_type is
                when "10" => -- Full match
                    rgb_led <= "010"; -- Green
                    digit3_out <= H_PATTERN;
                    digit2_out <= H_PATTERN;
                    digit1_out <= H_PATTERN;
                    digit0_out <= H_PATTERN;

                when "01" => -- Partial match
                    rgb_led <= "011"; -- Yellow (red + green)
                    -- Show H for matching digits, actual digits for non-matching
                    if digit_matches(1) = '1' then
                        digit3_out <= H_PATTERN; -- A matches X
                        digit1_out <= H_PATTERN;
                    else
                        digit3_out <= bcd1(7 downto 4); -- Show A
                        digit1_out <= bcd2(7 downto 4); -- Show X
                    end if;

                    if digit_matches(0) = '1' then
                        digit2_out <= H_PATTERN; -- B matches Y
                        digit0_out <= H_PATTERN;
                    else
                        digit2_out <= bcd1(3 downto 0); -- Show B
                        digit0_out <= bcd2(3 downto 0); -- Show Y
                    end if;

                when others => -- No match
                    rgb_led <= "001"; -- Red
                    digit3_out <= bcd1(7 downto 4); -- A
                    digit2_out <= bcd1(3 downto 0); -- B
                    digit1_out <= bcd2(7 downto 4); -- X
                    digit0_out <= bcd2(3 downto 0); -- Y
            end case;
        end if;
    end process;
end Behavioral;
```

F – 4x4 Mux VHDL description

```
34  entity mux4x4 is
35    port (
36      in3,in2,in1,in0 : in  std_logic_vector(3 downto 0);
37      sel             : in  std_logic_vector(1 downto 0);
38      y               : out std_logic_vector(3 downto 0)
39    );
40  end;
41
42  architecture rtl of mux4x4 is
43  begin
44    with sel select y <=
45      in3 when "00",
46      in2 when "01",
47      in1 when "10",
48      in0 when others;   -- "11"
49  end;
```

G – Hex to ssd VHDL description

```
40
41  architecture Dataflow of hex_to_ssd_const is
42    subtype cathode_out_t is std_logic_vector(7 downto 0); -- [7]=A ... [0]=DP
43
44    -- ACTIVE-LOW patterns for common-anode displays:
45    -- These match the usual "segments on = 0", and bit order A..G..DP.
46    constant ZERO : cathode_out_t := B"0000_0011";
47    constant ONE : cathode_out_t := B"1001_1111";
48    constant TWO : cathode_out_t := B"0010_0101";
49    constant THREE : cathode_out_t := B"0000_1101";
50    constant FOUR : cathode_out_t := B"1001_1001";
51    constant FIVE : cathode_out_t := B"0100_1001";
52    constant SIX : cathode_out_t := B"0100_0001";
53    constant SEVEN : cathode_out_t := B"0001_1111";
54    constant EIGHT : cathode_out_t := B"0000_0001";
55    constant NINE : cathode_out_t := B"0001_1001";
56    constant EE    : cathode_out_t := B"0110_0001"; -- 'E'
57    constant HH    : cathode_out_t := B"1001_0001"; -- 'H' (segments B, C, E, F, G on)
58  begin
59    ssd_cathode_out <=
60        ZERO  when hex_in = X"0" else
61        ONE   when hex_in = X"1" else
62        TWO   when hex_in = X"2" else
63        THREE when hex_in = X"3" else
64        FOUR  when hex_in = X"4" else
65        FIVE  when hex_in = X"5" else
66        SIX   when hex_in = X"6" else
67        SEVEN when hex_in = X"7" else
68        EIGHT when hex_in = X"8" else
69        NINE  when hex_in = X"9" else
70        EE    when hex_in = X"E" else
71        HH    when hex_in = X"F" else  -- 'H' pattern for hex value F
72        (others => '1');
73  end Dataflow;
74
```