

# **CSSE4010 – 2025 semester 2 - Lab 4**

## **Data Path and Controller Design with Finite State Machines**

Student name: Adnaan Buksh

Student number: 47435568

Lab Session attended: Wednesday 12-2pm

Submission date: 22/09/2025, 16:00 (UTC+10)

(There is no page limit for reports, however you should not make it overly long)

## Introduction

This lab implements a custom computing hardware architecture for finding the greatest common divisor (GCD) of two 8-bit unsigned numbers using the Euclidean algorithm. The design follows a data path and controller approach with a finite state machine (FSM) controlling the computation process. The system takes inputs from slide switches, processes them through a dedicated data path, converts the result to BCD format, and displays it on seven-segment displays.

## Design Description

The system architecture consists of two main components: the data path figure 1 and the controller (FSM) figure 2. The block diagram below shows the interconnection between these components and all necessary sub-systems.

The data path contains the following components:

- Two 8-bit registers (A and B) to store input values and intermediate results
- Two 2-to-1 multiplexers for selecting between initial inputs and computed values
- Two 8-bit subtractors for performing the Euclidean algorithm operations
- A comparator to determine the relationship between A and B (greater than, less than, or equal)
- A binary-to-BCD converter to prepare the result for display

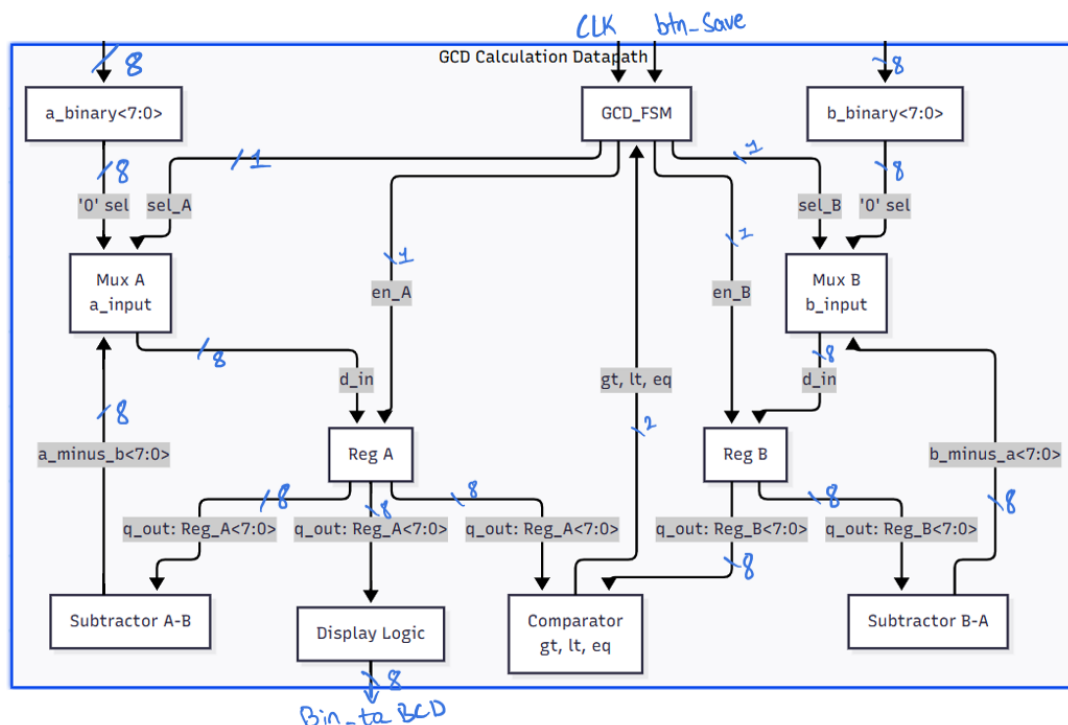


Figure 1 Data Path

The controller is implemented as a finite state machine with the following states:

- IDLE: Waiting for start signal
- LOAD: Loading initial values from switches into registers
- CHECK: Comparing values in registers A and B
- SUB\_A\_B: Performing  $A = A - B$  when  $A > B$
- SUB\_B\_A: Performing  $B = B - A$  when  $B > A$
- FINISH: Computation complete, result available

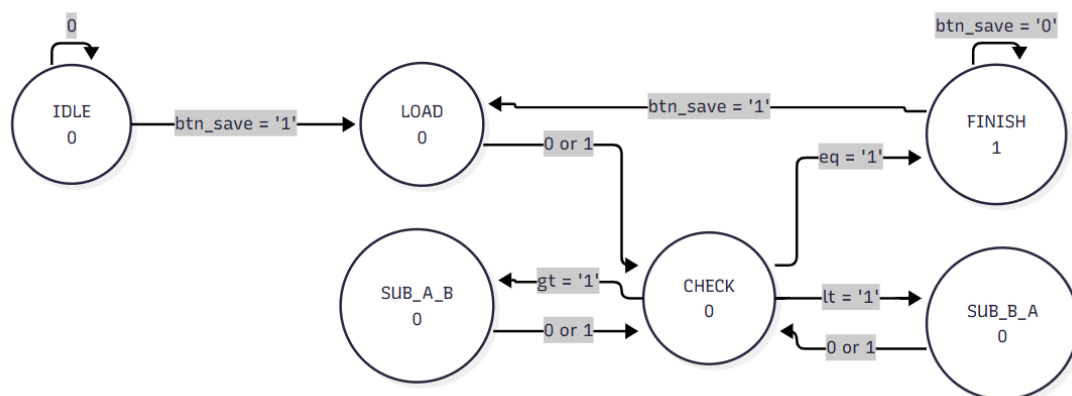


Figure 2 FSM

A significant design decision was the implementation of the arithmetic operations. Rather than building custom ripple-carry subtractors from basic gates, the design leverages the built-in arithmetic operators of VHDL and the numeric\_std package using behavioural instead of structural approach. As last practical highlighted that the FPGA resources don't differ, with small block.

## Simulation Results

In the testbench we define

Four scenarios were exercised:

(i) **A > B: GCD(56,42) = 14**

Input: A = 56 (0x38), B = 42 (0x2A)

Expected Result: GCD = 14 (0x0E)

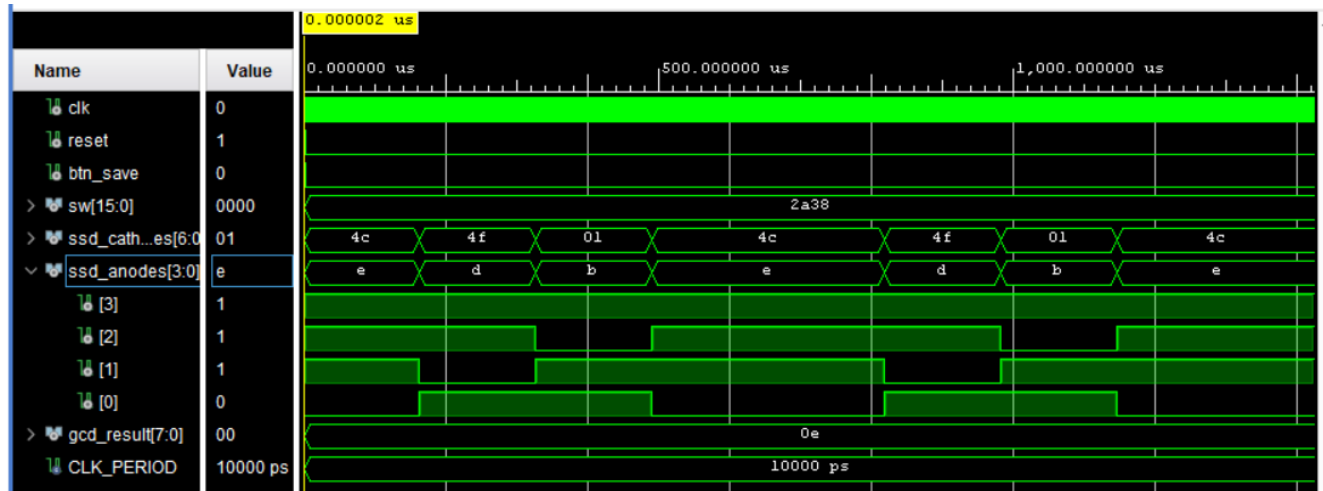


Figure 3 Test A > B

(ii) **A < B: GCD(30, 75) = 15**

Input: A = 30 (0x1E), B = 75 (0x4B)

Expected Result: GCD = 15 (0x0F)

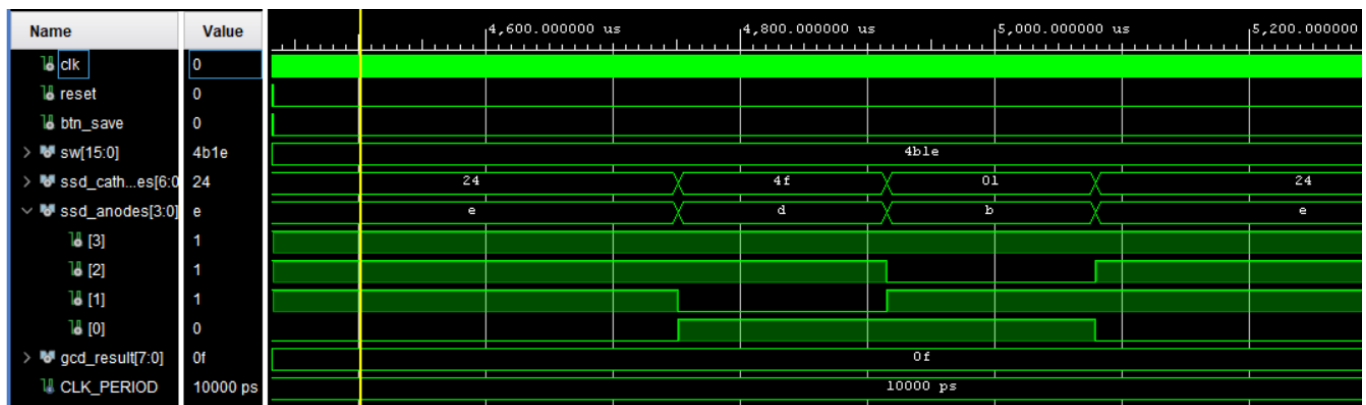


Figure 4 Test A < B

(iii) **A and B Prime:**  $\text{GCD}(13, 17) = 1$

Input: A = 13 (0x0D), B = 17 (0x11)

Expected Result:  $\text{GCD} = 1$  (0x01)

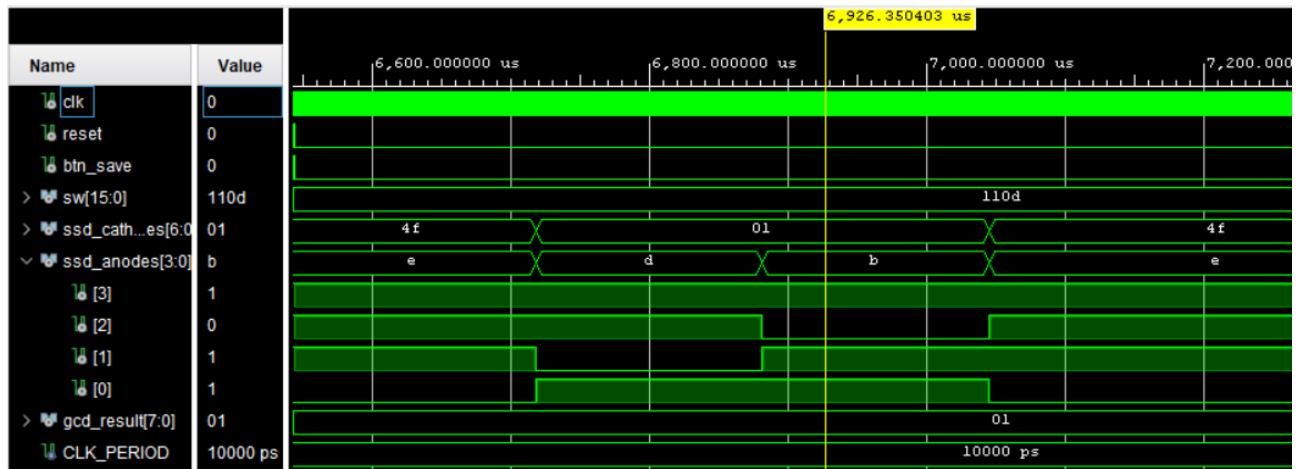


Figure 5 A and B Prime

With the above figures it is proven that the 8-bit binary to 3-digit works.

## On-Board Testing

A > B:  $\text{GCD}(56, 42) = 14$ , where 56 = 0011 1000 and 42 = 0010 1010

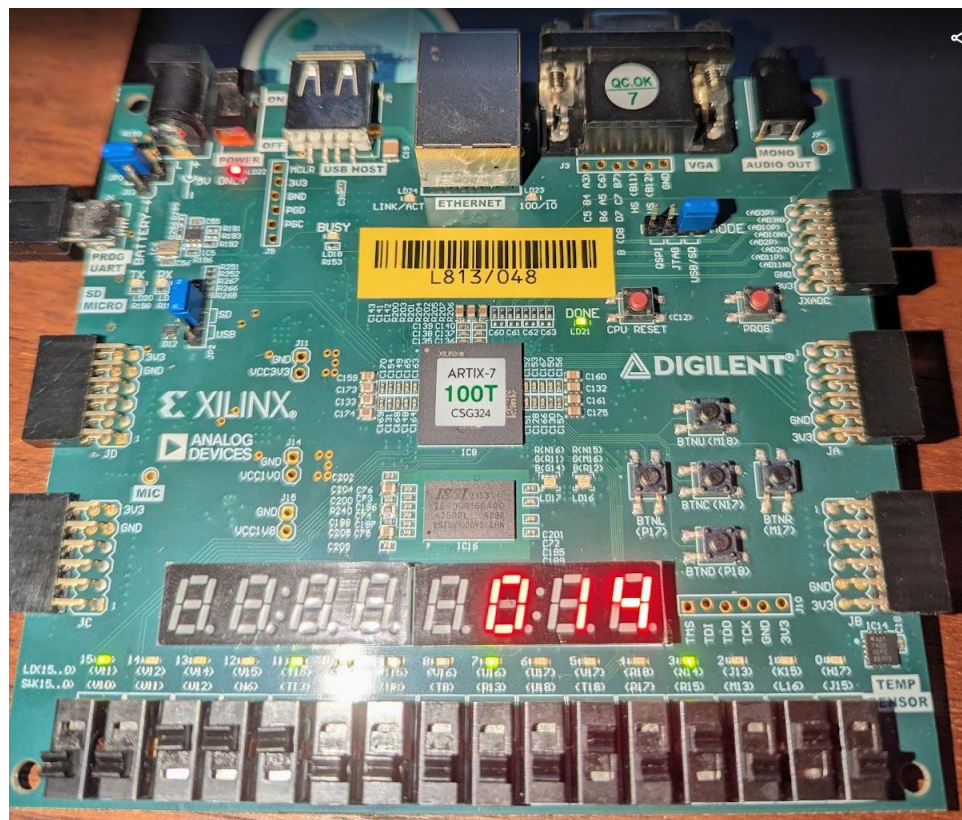


Figure 6 On-Board A > B



**A < B:**  $\text{GCD}(30, 75) = 15$ , where  $30 = 0001\ 1110$  and  $75 = 0100\ 1011$

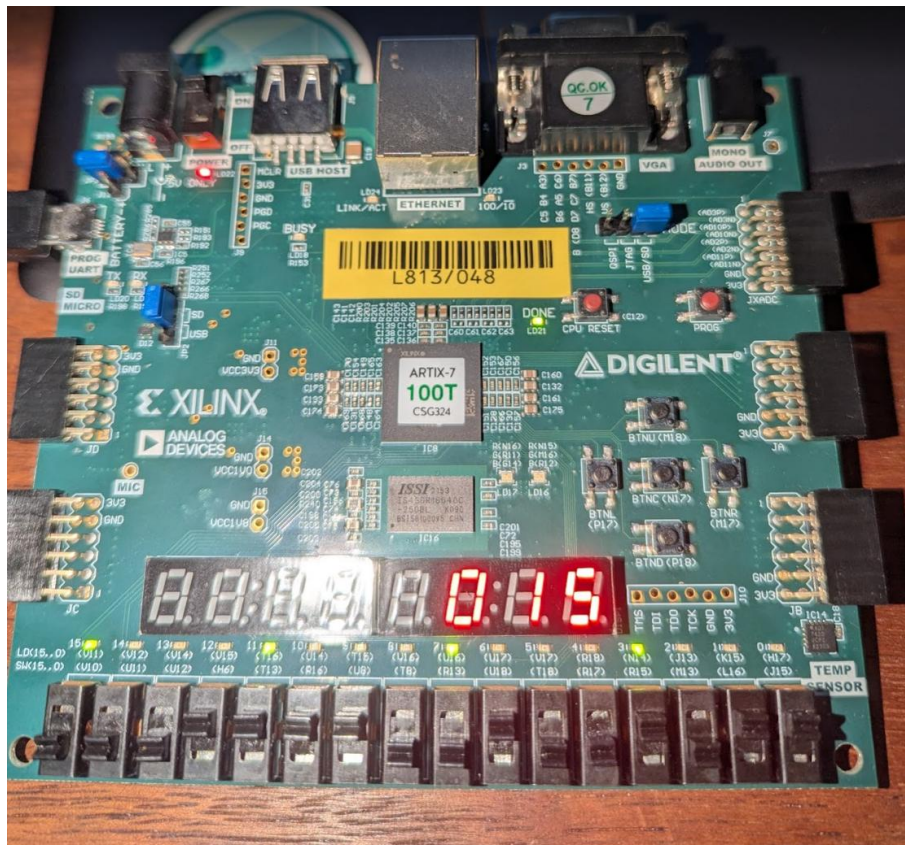


Figure 7 On-Board A < B

**A and B Prime:**  $\text{GCD}(13, 17) = 1$ , where  $13 = 0000\ 1101$  and  $17 = 0001\ 0001$

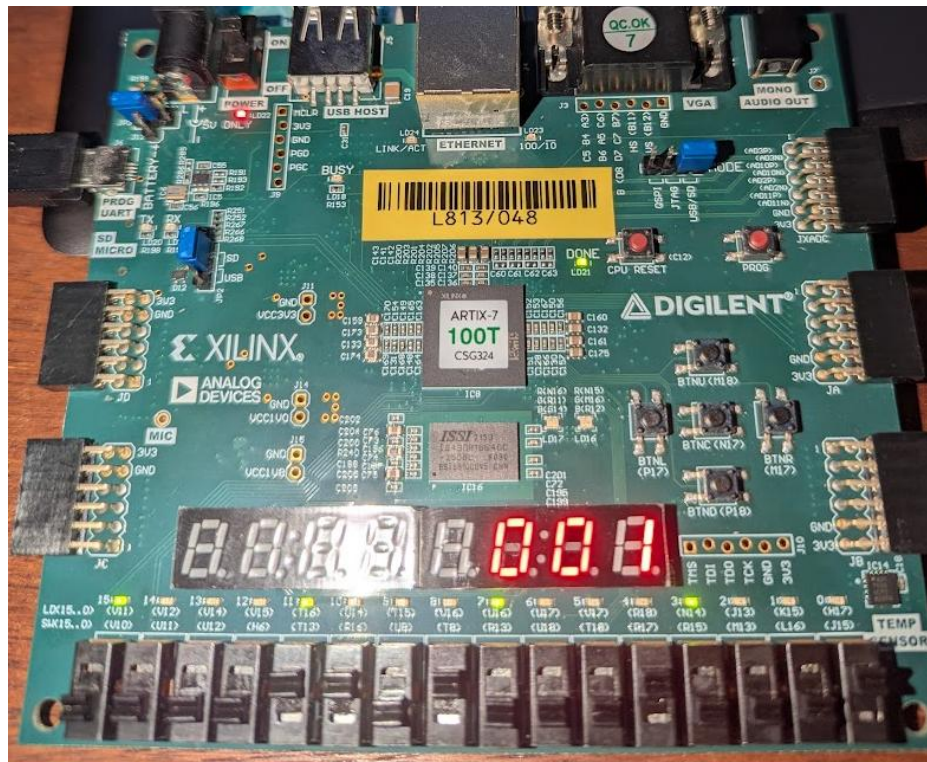


Figure 8 On-Board A and B Prime

## Synthesis/Implementation Results

The resource utilisation demonstrates an efficient implementation of the GCD calculator with display capabilities as seen in figure 9 and 10.

The predominance of LUT6 (33) and LUT4 (30) primitives indicates efficient packing of logic into the FPGA's 6-input LUT architecture.

The 12 CARRY4 primitives demonstrate efficient implementation of the subtraction operations in the GCD datapath, utilizing the FPGA's dedicated carry chain resources for optimal performance.

The resource difference between Register A (27 LUTs) and Register B (9 LUTs) is expected and correct. Register A requires additional LUT resources because it drives both the internal datapath logic and the display output system, necessitating stronger drivers and output buffering to handle the higher fanout requirements.

Name	^1	Slice LUTs (63400)	Slice Registers (126800)	Bonded IOB (210)	BUFGCTRL (32)
▼ N gcd_top		54	40	38	1
I U_CNT (digit_counter)		3	16	0	0
I U_COMP (greaterorlessthan)		0	0	0	0
I U_DEBOUNCE (debounce)		2	2	0	0
I U_DISPLAY (gcd_display)		0	0	0	0
I U_GCD_FSM (gcd_fsm)		17	6	0	0
I U_NUM_REG_A (EightBitReg)		27	8	0	0
I U_NUM_REG_B (EightBitReg_0)		9	8	0	0

Figure 9 Resources summary

```
7. Primitives
-----
+-----+
| Ref Name | Used | Functional Category |
+-----+
| FDRE     | 34  | Flop & Latch      |
| LUT6     | 33  | LUT                |
| LUT4     | 30  | LUT                |
| OBUF     | 19  | IO                 |
| IBUF     | 19  | IO                 |
| CARRY4   | 12  | CarryLogic         |
| LUT5     | 10  | LUT                |
| LUT2     | 10  | LUT                |
| LUT3     | 6   | LUT                |
| FDCE     | 3   | Flop & Latch      |
| LUT1     | 2   | LUT                |
| BUFG     | 1   | Clock              |
+-----+
```

Figure 10 Primitives

## Discussion and Conclusion

This lab successfully designed, implemented, and tested a custom computing hardware architecture for calculating the greatest common divisor (GCD) of two 8-bit unsigned numbers using the Euclidean algorithm. The design followed a structured data path and controller approach, employing a finite state machine to manage the computational workflow.

Behavioural VHDL was effectively employed for describing arithmetic operations and control logic, providing clear and maintainable code while allowing the synthesis tools to generate optimal hardware implementations. This approach demonstrated that for moderate complexity designs, behavioural descriptions can produce area-efficient results comparable to structural implementations.

During testing, some interface limitations were observed. The button debouncing circuit occasionally required multiple activation attempts, and the system relied on proper reset sequencing for reliable operation. These observations highlight the importance of robust input conditioning in practical digital systems.

## References

EightBitReg, digit\_counter, anode\_decoder, mux4x4 and hex\_to\_ssd\_const VHDL descriptions were reused from previous pracs.



## Appendix

### A – FSM

---

```
61      begin
62          -- Default next state to prevent latches
63          next_state <= state;
64
65      case state is
66      when IDLE =>
67          if btn_save = '1' then
68              next_state <= LOAD;
69          else
70              next_state <= IDLE;
71          end if;
72
73      when LOAD =>
74          next_state <= CHECK;
75
76      when CHECK =>
77          if eq = '1' then
78              next_state <= FINISH;
79          elsif gt = '1' then
80              next_state <= SUB_A_B;
81          else
82              next_state <= SUB_B_A;
83          end if;
84
85      when SUB_A_B =>
86          next_state <= CHECK;
87
88      when SUB_B_A =>
89          next_state <= CHECK;
90
91      when FINISH =>
92          if btn_save = '1' then
93              next_state <= LOAD; -- Start a new calculation
94          else
95              next_state <= FINISH; -- hold result
96          end if;
97
98      when others =>
99          next_state <= IDLE;
100      end case;
101  end process;
```

---

## B – FSM – output logic

```
103 |      -- Output logic (Moore FSM)
104 |      process(state)
105 |      begin
106 |          -- default values
107 |          sel_A <= '0';
108 |          sel_B <= '0';
109 |          en_A <= '0';
110 |          en_B <= '0';
111 |          finished <= '0';
112 |
113 |      case state is
114 |          when IDLE =>
115 |              null;
116 |
117 |          when LOAD =>
118 |              sel_A <= '0';
119 |              sel_B <= '0'; -- load switches
120 |              en_A <= '1';
121 |              en_B <= '1';
122 |
123 |          when CHECK =>
124 |              null; -- only look at comparator
125 |
126 |          when SUB_A_B =>
127 |              sel_A <= '1';
128 |              en_A <= '1'; -- load A := A-B
129 |
130 |          when SUB_B_A =>
131 |              sel_B <= '1';
132 |              en_B <= '1'; -- load B := B-A
133 |
134 |          when FINISH =>
135 |              finished <= '1'; -- signal completion
136 |      end case;
137 |  end process;
138 |
139 | end Behavioral;
```

## C – Top Level - Signals

```
18 architecture Structural of gcd_top is
19     -- Internal signals
20     signal d0, d1, d2, d3 : std_logic_vector(3 downto 0); -- Nibbles for SSD display
21     signal cat_full       : std_logic_vector(7 downto 0); -- Full cathode pattern
22     signal sel2           : std_logic_vector(1 downto 0); -- Selection for SSD
23     signal cur_d          : std_logic_vector(3 downto 0); -- Current digit for display
24     signal count          : integer range 0 to 3;         -- Counter for SSD display
25
26     -- Register signals
27     signal Reg_A, Reg_B  : std_logic_vector(7 downto 0); -- Registers to hold A and B
28     signal a_binary, b_binary : std_logic_vector(7 downto 0); -- Binary conversion signals
29
30     -- FSM signals
31     signal gt, lt, eq : std_logic; -- Comparison signals
32     signal sel_A, sel_B : std_logic; -- Mux select signals
33     signal en_A, en_B : std_logic; -- Register enable signals
34     signal finished : std_logic; -- GCD calculation complete
35
36     -- Subtraction signals
37     signal a_minus_b : std_logic_vector(7 downto 0);
38     signal b_minus_a : std_logic_vector(7 downto 0);
39
40     -- Input mux signals
41     signal a_input, b_input : std_logic_vector(7 downto 0);
42
43     -- Button debounce signals
44     signal btn_save_d1, btn_save_d2 : std_logic := '0';
45     signal btn_save_pulse : std_logic;
46
```

## D – Top Level – Structural

```
47   begin
48       -- Initial input binary conversion
49       a_binary <= sw(7 downto 0);    -- A is lower 8 bits
50       b_binary <= sw(15 downto 8);   -- B is upper 8 bits
51       -- Comparator logic
52 ⊕ U_COMP: entity work.greaterorlessthan port map (...
59       -- Subtraction operations
60       a_minus_b <= std_logic_vector(unsigned(Reg_A) - unsigned(Reg_B));
61       b_minus_a <= std_logic_vector(unsigned(Reg_B) - unsigned(Reg_A));
62       -- Input multiplexers
63       a_input <= a_binary when sel_A = '0' else a_minus_b;
64       b_input <= b_binary when sel_B = '0' else b_minus_a;
65       -- Button debounce process
66 ⊕ U_DEBOUNCE: entity work.debounce port map (...
71       -- Instantiate the GCD FSM
72 ⊕ U_GCD_FSM: entity work.gcd_fsm port map (...
85       -- 8-bit registers for storing A and B
86 ⊕ U_NUM_REG_A: entity work.EightBitReg port map (...
93 ⊕ U_NUM_REG_B: entity work.EightBitReg port map (...
100      -- Display logic - show GCD result when finished
101 ⊕ U_DISPLAY: entity work.gcd_display port map (...
108      -- Timebase for cycling SSD digits
109 ⊕ U_CNT: entity work.digit_counter port map (...
114      -- Select the active SSD anode based on the counter value
115      sel2 <= std_logic_vector(to_unsigned(count, 2));
116 ⊕ U_ANODE: entity work.anode_decoder port map (...
120      -- Use mux to select which digit to display
121 ⊕ U_MUX: entity work.mux4x4 port map (...
129      -- Convert the selected hex digit to SSD pattern
130 ⊕ U_HEX: entity work.hex_to_ssd_const port map (...
134
135      -- Output the cathode pattern to the SSD (ignore DP)
136      ssd_cathodes <= cat_full(6 downto 0);
137      ssd_anodes_unused <= "1111";
138      led_used <= "1111";
139
140 ⊖ end Structural;
```