



# American University of Ras Al Khaimah

## CENG 335 – Computer Architecture

### FSM Project

#### Students:

Amir Beshir (2023006017)

Riyad Almasri (2023006329)

Mohamed AbouSaada (2023006306)

Sinishaw Nasa (2023006199)

**Course Section:** Section 1 **Date of Submission:** 22 November 2025

**Instructor:** Prof. Arfan Ghani

### Contents

|                                                                                  |          |
|----------------------------------------------------------------------------------|----------|
| <b>1 Task 1: Memory-less FSM</b>                                                 | <b>1</b> |
| <b>2 Task 2: Memory-Based FSM</b>                                                | <b>1</b> |
| <b>3 Task 3: Waveform Analysis (Testbench)</b>                                   | <b>2</b> |
| <b>4 Task 4: FPGA Implementation, Diagrams &amp; designs and Timing Behavior</b> | <b>3</b> |
| <b>5 Task 5: Discussion and Comparative Analysis</b>                             | <b>5</b> |
| <b>6 Appendix</b>                                                                | <b>6</b> |
| 6.1 Contributions . . . . .                                                      | 6        |
| 6.2 Memory-less FSM . . . . .                                                    | 6        |
| 6.2.1 Code . . . . .                                                             | 6        |
| 6.2.2 Test Bench . . . . .                                                       | 14       |
| 6.2.3 Timing Behavior (VWF) . . . . .                                            | 24       |
| 6.2.4 Practical Output . . . . .                                                 | 26       |
| 6.3 Memory-based FSM . . . . .                                                   | 28       |
| 6.3.1 Code . . . . .                                                             | 28       |
| 6.3.2 Test Bench . . . . .                                                       | 43       |
| 6.3.3 Timing Behavior (VWF) . . . . .                                            | 53       |
| 6.3.4 Practical Output . . . . .                                                 | 56       |
| 6.4 Conclusion . . . . .                                                         | 58       |

## 1. Task 1: Memory-less FSM

We implemented the memory-less finite state machine using a three-entity architecture consisting of a top-level entity, main FSM entity, and testbench entity. The top-level entity `top_level_memory_less` defines all physical I/O ports including the 50 MHz clock, active-low reset, start switch (for FPGA Board) and for the timing behavior we divided the clock to be 1 Hz. The main FSM entity `memory_less_fsm` contains the core state machine logic with five specified states, and a separate testbench entity was developed for comprehensive simulation and verification.

This component works by implementing a Moore-type state machine where the `data_out` signal (which corresponds to the project's specified `out_signal`) displays the sequence "00" or inactive in IDLE, "01" in STEP1, "10" in STEP2, and "11" in STEP3 and DONE states, while the `done` flag asserts exclusively in the FSM\_DONE state. The state transitions follow a deterministic sequence: IDLE  $\rightarrow$  STEP1  $\rightarrow$  STEP2  $\rightarrow$  STEP3  $\rightarrow$  FSM\_DONE  $\rightarrow$  IDLE, with transitions occurring only when the paced `step_en` signal is asserted and the initial transition from IDLE requiring the `start` input to be active.

This implementation satisfies Task 1 requirements because it provides a complete Moore machine implementation with the specified five-state sequence and state-dependent output generation as outlined in the project specification table. The three-entity modular architecture ensures clear separation of concerns between physical interface management, core FSM logic, and verification, while the use of `data_out` as the output signal variable maintains functional equivalence with the required `out_signal` behavior. The design successfully demonstrates the required behavior of waiting for a start signal, performing the three-step operation sequence, and properly asserting the completion flag.

## 2. Task 2: Memory-Based FSM

We implemented the memory-based finite state machine using a four-entity architecture consisting of a top-level entity, main FSM entity, testbench entity, and dedicated RAM component. The top-level entity `top_level_memory_based` defines all physical I/O ports and implements a configurable step enable generator, while the main FSM entity `memory_based_fsm` contains the core state machine logic with five states: IDLE, READ\_STATE, PROCESS\_DATA, INC\_ADDR, and FSM\_DONE. The RAM entity `RAM4x8` provides a 4 $\times$ 8 memory module preloaded with hexadecimal data values through a memory initialization file, and a separate testbench entity was developed for comprehensive simulation and verification.

This component works by implementing a Moore-type state machine that sequentially accesses all four memory locations. The FSM begins in the IDLE state and transitions to



READ\_STATE when the start signal is asserted, where it reads the first 8-bit data value from the RAM at address 0. It then proceeds to PROCESS\_DATA for data manipulation, increments the address in INC\_ADDR, and returns to READ\_STATE to access the next memory location. This read-process-increment cycle continues until all four addresses (0 through 3) have been accessed, at which point the FSM enters FSM\_DONE and asserts the completion flag before returning to IDLE. The design incorporates conditional operation where the FSM only cycles if the start switch remains active, and halts after completion if the switch is deactivated.

This implementation satisfies Task 2 requirements because it successfully combines FSM design with memory addressing and sequential data access as specified in the project. The four-entity modular architecture ensures clear separation between physical interface management, core FSM logic, memory storage, and verification. The memory interface with the 4×8 RAM module demonstrates realistic hardware scenarios, while the state-dependent output generation and address counter management provide clear observability of the memory access sequence. The design properly reads each memory location in sequence, processes the data, and signals completion after the final memory location is accessed, fulfilling all specified functionality for the memory-based FSM implementation.

### 3. Task 3: Waveform Analysis (Testbench)

#### Memory-less testbench

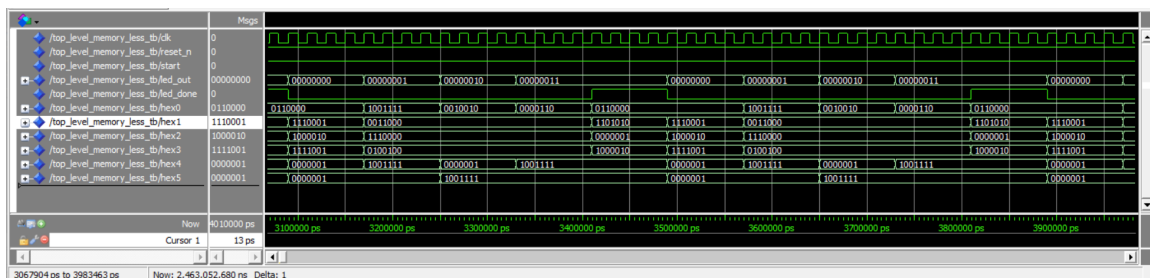


Figure 1: Memory-less and memory-based testbench waveforms

#### Explanation

Memory-less testbench shows the clock and reset sequence putting the FSM into IDLE with LED\_OUT=00; when START is asserted the FSM advances on the next gated step and the LED pattern cycles: first the LSB (LED0) becomes '1' (displayed as '01' on the HEX pair), then on the following gated cycle LED0 returns to '0' and the MSB (LED1) becomes '1' ('10'), the next step sets both bits to '1' ('11') and the DONE signal is asserted, after which the outputs return to '00' as the FSM returns to IDLE. The HEX0–HEX5 traces in the picture mirror these LED changes (state name + 2-bit value), and the left-side

signal list identifies the shown nets (clock, reset, start, LED outputs, done), making it clear which waveform segment corresponds to each registered output and state transition.

## Memory-based testbench

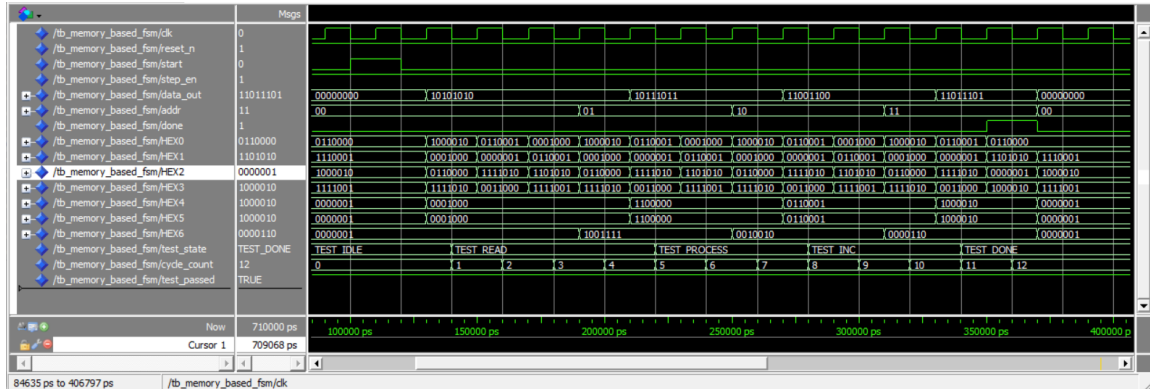


Figure 2: Memory-less and memory-based testbench waveforms

## Explanation

Memory-based testbench shows reset to IDLE with `data_out=0x00` and `addr=00`, then after START the controller reads successive RAM locations: at `addr=00` `data_out` appears as 0xAA (high/low nibbles visible on HEX4/HEX5), the next gated step increments `addr` to 01 and `data_out` becomes 0xBB, then `addr=10` with 0xCC, and `addr=11` with 0xDD; the TEST\_STATE row labels (TEST\_IDLE, TEST\_READ, TEST\_PROCESS, TEST\_INC, TEST\_DONE) align with these cycles and DONE is asserted only after the final memory word is processed. The `addr` trace and HEX6 display the current address, the data trace and HEX4/HEX5 show the RAM contents per-address, and the testbench cycle counter markers demonstrate that reads, processing and address increments occur on successive gated steps as intended.

## 4. Task 4: FPGA Implementation, Diagrams & designs and Timing Behavior

### Memory-less FSM

The memory-less controller is implemented in the top-level entity `top_level_memory_less`, which instantiates the `memory_less_fsm` block and connects it to the DE2-115 board I/O. The 50 MHz clock `FPGA_CLK` is divided using the generic `STEP_TICKS`: a small value is used in simulation, while `STEP_TICKS = 50 000 000` on hardware produces a one-second `step_en` pulse so that state changes are clearly visible. The 2-bit Moore output is mapped to `LED_OUT(1 downto 0)` (other LEDs forced to zero) and the `done` signal drives `LED_DONE`.

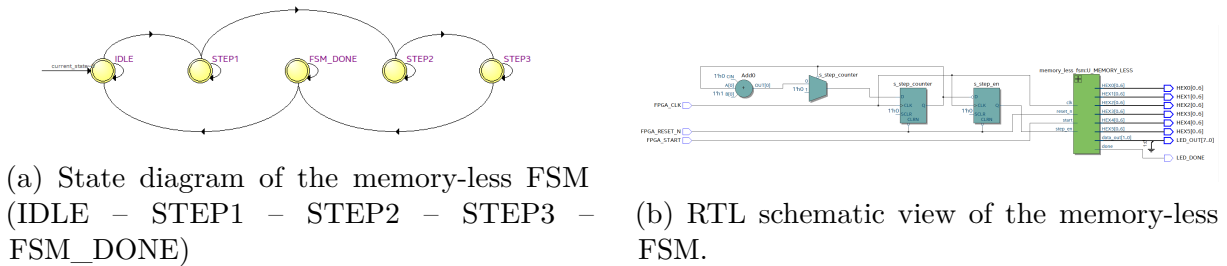


Figure 3: State diagram and RTL view of the memory-Less FSM.

The FSM follows the sequence  $IDLE \rightarrow STEP1 \rightarrow STEP2 \rightarrow STEP3 \rightarrow FSM\_DONE \rightarrow IDLE$ , with HEX3–HEX0 showing the state name and HEX5–HEX4 showing the corresponding 2-bit pattern (“00”, “01”, “10”, “11”), as illustrated in Fig. 4a. The timing waveform and a photograph of the FPGA implementation confirming that the LED and HEX outputs match the expected behaviour are provided in the Appendix.

### Memory-based FSM

The memory-based controller is implemented in the top-level module `top_level_memory_based`, which instantiates the `memory_based_fsm` together with a  $4 \times 8$  RAM (RAM4x8) initialized from `memory_data.mif`. The RAM is configured with  $C\_ADDR\_WIDTH = 2$ ,  $C\_DATA\_WIDTH = 8$ , and  $C\_MEM\_SIZE = 4$ , allowing the FSM to read four 8-bit locations before asserting `done`. As in the memory-less design, the 50 MHz system clock is slowed using the generic `STEP_TICKS`: a small value is used in simulation, while `STEP_TICKS = 50 000 000` on hardware produces a one-second `step_en` pulse for clear observation on the DE2-115 board.

The FSM follows the sequence  $IDLE \rightarrow READ\_STATE \rightarrow PROCESS\_DATA \rightarrow INC\_ADDR \rightarrow FSM\_DONE \rightarrow IDLE$ . In `IDLE` the address and data registers are cleared; in `READ_STATE` the current RAM word is fetched; in `PROCESS_DATA` the value is latched; in `INC_ADDR` the address is incremented when more locations remain; and in `FSM_DONE` the completion flag is asserted for one step before returning to `IDLE`.

For observability on the FPGA, the 8-bit output is mapped to `LED_OUT`, the `done` flag drives `LED_DONE`, HEX3–HEX0 show the state name, HEX5–HEX4 display the RAM data in hex, and HEX6 shows the current address (0–3). This makes it straightforward to confirm that the FSM visits each address in order and that the RAM data and `done` timing match the specification. The timing behaviour and FPGA implementation screenshots are provided in the Appendix.

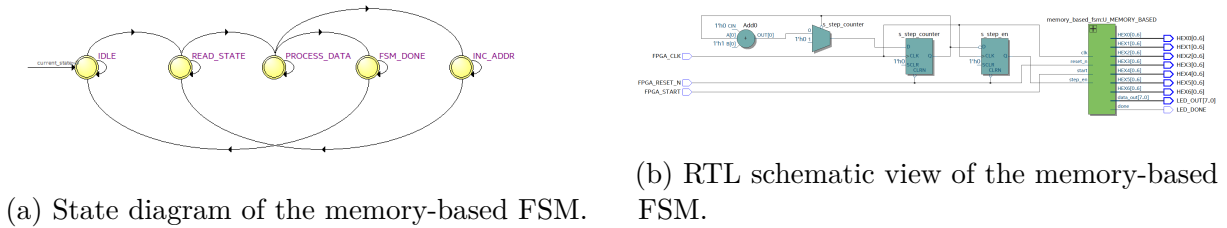


Figure 4: State diagram and RTL view of the memory-based FSM.

## 5. Task 5: Discussion and Comparative Analysis

### Comparative Analysis: Memory-less vs. Memory-based FSM

The comparative analysis reveals significant differences in complexity, outputs, and observability between the two FSM implementations. The memory-less FSM demonstrates lower complexity with only five states and simple combinatorial logic, featuring a hardcoded output sequence where data values are directly embedded in the state machine logic. In contrast, the memory-based FSM introduces additional complexity through memory interfacing using a RAM port, address counters, and sequential data access patterns.

In terms of outputs, the memory-less design generates simple 2-bit state encodings (00, 01, 10, 11) that are predetermined and fixed in the VHDL code, whereas the memory-based version produces 8-bit data values retrieved dynamically from RAM memory through dedicated read ports, with corresponding address outputs that enable flexible data access.

Observability differs substantially: the memory-less FSM provides direct state visibility through LED patterns that reflect the hardcoded sequence, while the memory-based implementation requires monitoring both address progression and corresponding data outputs across multiple displays, offering richer debugging information through its RAM-based architecture but requiring more sophisticated observation techniques.

### Fine-grained vs. Coarse-grained multithreading

In the context of sequential FSM execution, fine-grained multithreading involves switching between multiple threads every clock cycle to maximize processor utilization and hide pipeline stalls. Coarse-grained multithreading, in contrast, switches threads only during significant stall events like cache misses. While our FSMs operate sequentially without true parallel execution, the memory-based FSM exhibits characteristics analogous to fine-grained behavior through its rapid state transitions between memory operations, while the memory-less FSM resembles coarse-grained execution with its longer-duration state occupations. Both approaches represent different trade-offs between throughput and resource utilization, though the FSM's inherent sequential nature limits direct application of multithreading concepts beyond this conceptual comparison.

## 6. Appendix

### 6.1. Contributions

| Student                           | Contributions                                       |
|-----------------------------------|-----------------------------------------------------|
| Amir Beshir<br>(2023006017)       | Worked on Task 2, Task 3, Task 4, and the Appendix. |
| Riyad Almasri<br>(2023006329)     | Worked on Task 1 and Task 5.                        |
| Mohamed AbouSaada<br>(2023006306) | Worked on Task 1 and Task 5.                        |
| Sinishaw Nasa<br>(2023006199)     | Worked on Task 2, Task 3, Task 4, and the Appendix. |

Table 1: Summary of individual contributions to the project.

**Link for Video Implementation:** [Click here to open the video folder](#)

### 6.2. Memory-less FSM

#### 6.2.1 Code

Listing 1: VHDL for the Memory-less FSM Top-Level

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity top_level_memory_less is
  generic (
    STEP_TICKS : integer := 1  -- Use 50,000,000 for board, use 1 or small
    ↪ number for simulation
  );
  port (
    -- Standard FPGA I/O
    FPGA_CLK      : in  std_logic;          -- 50 MHz clock
    FPGA_RESET_N  : in  std_logic;          -- Active-low reset
    FPGA_START    : in  std_logic;          -- Start signal (Switch 0)

    -- Outputs

```

```

        LED_OUT      : out std_logic_vector(7 downto 0); -- Red LEDs
        LED_DONE     : out std_logic;                    -- Done signal (
        ↪ Green LED)

        -- 7-Segment Displays
        HEX0          : out std_logic_vector(0 to 6);
        HEX1          : out std_logic_vector(0 to 6);
        HEX2          : out std_logic_vector(0 to 6);
        HEX3          : out std_logic_vector(0 to 6);
        HEX4          : out std_logic_vector(0 to 6);
        HEX5          : out std_logic_vector(0 to 6)
    );
end entity top_level_memory_less;

architecture structure of top_level_memory_less is

    signal s_step_counter : integer range 0 to STEP_TICKS - 1 := 0;
    signal s_step_en      : std_logic := '0';

    -- now only 2 bits from the FSM
    signal s_data_out : std_logic_vector(1 downto 0);
    signal s_done     : std_logic;

    component memory_less_fsm is
        port (
            clk      : in  std_logic;
            reset_n  : in  std_logic;
            start    : in  std_logic;
            step_en  : in  std_logic;
            data_out  : out std_logic_vector(1 downto 0); -- 2-bit output
            done      : out std_logic;
            HEX0      : out std_logic_vector(0 to 6);
            HEX1      : out std_logic_vector(0 to 6);
            HEX2      : out std_logic_vector(0 to 6);
            HEX3      : out std_logic_vector(0 to 6);
            HEX4      : out std_logic_vector(0 to 6);
            HEX5      : out std_logic_vector(0 to 6)
        );
    end component;

begin

```

```

-- Clock-enable generator: step_en

step_enable_proc : process(FPGA_CLK, FPGA_RESET_N)
begin
    if FPGA_RESET_N = '0' then
        s_step_counter <= 0;
        s_step_en      <= '0';
    elsif rising_edge(FPGA_CLK) then
        if s_step_counter = STEP_TICKS - 1 then
            s_step_counter <= 0;
            s_step_en      <= '1';    -- one-cycle pulse
        else
            s_step_counter <= s_step_counter + 1;
            s_step_en      <= '0';
        end if;
    end if;
end process step_enable_proc;

-- Memory-less FSM instantiation

U_MEMORY_LESS : memory_less_fsm
    port map (
        clk          => FPGA_CLK,          -- main 50 MHz clock
        reset_n      => FPGA_RESET_N,
        start        => FPGA_START,        -- Switch 0
        step_en      => s_step_en,
        data_out     => s_data_out,        -- 2-bit output: 00,01,10,11
        done         => s_done,
        HEX0         => HEX0,
        HEX1         => HEX1,
        HEX2         => HEX2,
        HEX3         => HEX3,
        HEX4         => HEX4,
        HEX5         => HEX5
    );

-- Drive only LEDR(1 downto 0); others are 0
LED_OUT <= (7 downto 2 => '0') & s_data_out;

```

```
LED_DONE <= s_done;

end architecture structure;
```

Listing 2: VHDL code for the Memory-less FSM main body

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity memory_less_fsm is
    port (
        clk          : in  std_logic;  -- 50 MHz clock
        reset_n       : in  std_logic;
        start         : in  std_logic;  -- Switch 0 for continuous cycling
        step_en       : in  std_logic;  -- 1 Hz enable pulse
        data_out      : out std_logic_vector(1 downto 0); -- 2-bit output
        done          : out std_logic;
        HEX0          : out std_logic_vector(0 to 6);
        HEX1          : out std_logic_vector(0 to 6);
        HEX2          : out std_logic_vector(0 to 6);
        HEX3          : out std_logic_vector(0 to 6);
        HEX4          : out std_logic_vector(0 to 6);
        HEX5          : out std_logic_vector(0 to 6)
    );
end entity memory_less_fsm;

architecture rtl of memory_less_fsm is

    -- States for memory-less FSM

    type state_type is (IDLE, STEP1, STEP2, STEP3, FSM_DONE);
    signal current_state, next_state : state_type;

    -- Internal signals

    signal s_data_out_reg : std_logic_vector(1 downto 0) := (others => '0');
    signal s_done_reg     : std_logic := '0';
```



```

signal s_hex0_reg, s_hex1_reg, s_hex2_reg, s_hex3_reg : std_logic_vector(0
↪ to 6);
signal s_hex4_reg, s_hex5_reg : std_logic_vector(0 to 6);

begin

    -- Registered outputs + HEX displays

    reg_proc : process(clk, reset_n)
    begin
        if reset_n = '0' then
            s_data_out_reg <= "00";
            s_done_reg      <= '0';

            -- "IDLE" on HEX3-HEX0
            s_hex3_reg <= "1111001"; -- I
            s_hex2_reg <= "1000010"; -- d
            s_hex1_reg <= "1110001"; -- L
            s_hex0_reg <= "0110000"; -- E

            -- show "00" on HEX5 HEX4
            s_hex5_reg <= "0000001"; -- 0
            s_hex4_reg <= "0000001"; -- 0

        elsif rising_edge(clk) then
            if step_en = '1' then

                -- defaults
                s_data_out_reg <= s_data_out_reg;
                s_done_reg      <= '0';

                -- data_out and done (Moore outputs)

                case current_state is
                    when IDLE =>
                        s_data_out_reg <= "00"; -- IDLE
                        s_done_reg      <= '0';

                    when STEP1 =>

```

```

        s_data_out_reg <= "01";  -- STEP1
        s_done_reg      <= '0';

    when STEP2 =>
        s_data_out_reg <= "10";  -- STEP2
        s_done_reg      <= '0';

    when STEP3 =>
        s_data_out_reg <= "11";  -- STEP3
        s_done_reg      <= '0';

    when FSM_DONE =>
        s_data_out_reg <= "11";  -- DONE
        s_done_reg      <= '1';
end case;

-- HEX0HEX3: state name display

case current_state is
    when IDLE =>
        s_hex3_reg <= "1111001"; -- I
        s_hex2_reg <= "1000010"; -- d
        s_hex1_reg <= "1110001"; -- L
        s_hex0_reg <= "0110000"; -- E

    when STEP1 =>
        s_hex3_reg <= "0100100"; -- S
        s_hex2_reg <= "1110000"; -- t
        s_hex1_reg <= "0011000"; -- P
        s_hex0_reg <= "1001111"; -- 1

    when STEP2 =>
        s_hex3_reg <= "0100100"; -- S
        s_hex2_reg <= "1110000"; -- t
        s_hex1_reg <= "0011000"; -- P
        s_hex0_reg <= "0010010"; -- 2

    when STEP3 =>
        s_hex3_reg <= "0100100"; -- S
        s_hex2_reg <= "1110000"; -- t

```

```

        s_hex1_reg <= "0011000"; -- P
        s_hex0_reg <= "0000110"; -- 3

    when FSM_DONE =>
        s_hex3_reg <= "1000010"; -- d
        s_hex2_reg <= "0000001"; -- 0 (also looks like 0)
        s_hex1_reg <= "1101010"; -- n
        s_hex0_reg <= "0110000"; -- E
    end case;

-- HEX4HEX5: show the 2-bit output as "00,01,10,11"
-- HEX5 = MSB, HEX4 = LSB, both digits 0/1

case current_state is
    when IDLE =>
        -- "00"
        s_hex5_reg <= "0000001"; -- 0
        s_hex4_reg <= "0000001"; -- 0

    when STEP1 =>
        -- "01"
        s_hex5_reg <= "0000001"; -- 0
        s_hex4_reg <= "1001111"; -- 1

    when STEP2 =>
        -- "10"
        s_hex5_reg <= "1001111"; -- 1
        s_hex4_reg <= "0000001"; -- 0

    when STEP3 | FSM_DONE =>
        -- "11"
        s_hex5_reg <= "1001111"; -- 1
        s_hex4_reg <= "1001111"; -- 1
    end case;
end if; -- step_en
end if; -- rising_edge(clk)
end process reg_proc;

-- State register

```

```
state_register_proc : process(clk, reset_n)
begin
    if reset_n = '0' then
        current_state <= IDLE;
    elsif rising_edge(clk) then
        if step_en = '1' then
            current_state <= next_state;
        end if;
    end if;
end process state_register_proc;

-- Next-state logic

next_state_logic_proc : process(current_state, start)
begin
    case current_state is
        when IDLE =>
            if start = '1' then
                next_state <= STEP1;
            else
                next_state <= IDLE;
            end if;

        when STEP1 =>
            next_state <= STEP2;

        when STEP2 =>
            next_state <= STEP3;

        when STEP3 =>
            next_state <= FSM_DONE;

        when FSM_DONE =>
            next_state <= IDLE;

        when others =>
            next_state <= IDLE;
    end case;
end process next_state_logic_proc;
```

```
-- Output ports

data_out <= s_data_out_reg;
done      <= s_done_reg;

HEX0 <= s_hex0_reg;
HEX1 <= s_hex1_reg;
HEX2 <= s_hex2_reg;
HEX3 <= s_hex3_reg;
HEX4 <= s_hex4_reg;
HEX5 <= s_hex5_reg;

end architecture rtl;
```

### 6.2.2 Test Bench

Listing 3: VHDL code for the Memory-less FSM Test Bench

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity top_level_memory_less_tb is
end entity top_level_memory_less_tb;

architecture testbench of top_level_memory_less_tb is

    -- Constants
    constant CLK_PERIOD : time := 20 ns; -- 50 MHz clock
    constant STEP_TICKS : integer := 4;  -- Same as in design for simulation

    -- Fixed component declaration - STEP_TICKS is a generic, not a port
    component top_level_memory_less is
        generic (
            STEP_TICKS : integer := 4
        );
        port (
```

```

        FPGA_CLK      : in  std_logic;
        FPGA_RESET_N  : in  std_logic;
        FPGA_START    : in  std_logic;
        LED_OUT       : out std_logic_vector(7 downto 0);
        LED_DONE      : out std_logic;
        HEX0          : out std_logic_vector(0 to 6);
        HEX1          : out std_logic_vector(0 to 6);
        HEX2          : out std_logic_vector(0 to 6);
        HEX3          : out std_logic_vector(0 to 6);
        HEX4          : out std_logic_vector(0 to 6);
        HEX5          : out std_logic_vector(0 to 6)
    );
end component;

-- Testbench signals
signal clk      : std_logic := '0';
signal reset_n  : std_logic := '0';
signal start    : std_logic := '0';
signal led_out  : std_logic_vector(7 downto 0);
signal led_done : std_logic;
signal hex0, hex1, hex2, hex3 : std_logic_vector(0 to 6);
signal hex4, hex5 : std_logic_vector(0 to 6);

-- Helper function to convert 7-segment to character for display
function seg_to_char(seg : std_logic_vector(0 to 6)) return character is
begin
    case seg is
        -- Numbers (0-9)
        when "0000001" => return '0';
        when "1001111" => return '1';
        when "0010010" => return '2';
        when "0000110" => return '3';
        when "1001100" => return '4';
        when "0100000" => return '6';
        when "0001111" => return '7';
        when "0000000" => return '8';
        when "0001100" => return '9';

        -- Letters (unique patterns)
        when "0001000" => return 'A';
        when "1100000" => return 'B';
    end case;
end function;

```

```

        when "0110001" => return 'C';
        when "0110000" => return 'E';
        when "0111000" => return 'F';
        when "1111001" => return 'I';
        when "1110001" => return 'L';
        when "1110000" => return 't';
        when "0011000" => return 'P';
        when "1101010" => return 'n';
        when "0100100" => return 'S';
        when "1111111" => return ' ';
        when others     => return '?';
    end case;
end function;

-- Function to decode state from HEX displays
function get_state_from_hex(hex3, hex2, hex1, hex0: std_logic_vector(0 to
↪ 6)) return string is
    variable char3, char2, char1, char0 : character;
begin
    char3 := seg_to_char(hex3);
    char2 := seg_to_char(hex2);
    char1 := seg_to_char(hex1);
    char0 := seg_to_char(hex0);

    if char3 = 'I' and char2 = 'd' and char1 = 'L' and char0 = 'E' then
        return "IDLE";
    elsif char3 = 'S' and char2 = 't' and char1 = 'P' then
        case char0 is
            when '1' => return "STEP1";
            when '2' => return "STEP2";
            when '3' => return "STEP3";
            when others => return "STEP?";
        end case;
    elsif char3 = 'd' and char2 = '0' and char1 = 'n' and char0 = 'E' then
        return "DONE";
    else
        return "UNKNOWN";
    end if;
end function;

-- Function to get data value from HEX4-HEX5

```

```

function get_data_from_hex(hex5, hex4: std_logic_vector(0 to 6)) return
↪ string is
    variable char5, char4 : character;
begin
    char5 := seg_to_char(hex5);
    char4 := seg_to_char(hex4);

    if char5 = ' ' and char4 = ' ' then
        return "OFF";
    elsif char5 = 'A' and char4 = 'A' then
        return "AA";
    elsif char5 = 'B' and char4 = 'B' then
        return "BB";
    elsif char5 = 'C' and char4 = 'C' then
        return "CC";
    else
        return "??";
    end if;
end function;

-- Function to convert std_logic_vector to hex string
function to_hex_string(vec : std_logic_vector) return string is
    variable result : string(1 to 2);
    variable nibble : std_logic_vector(3 downto 0);
begin
    -- Convert high nibble
    nibble := vec(7 downto 4);
    case nibble is
        when "0000" => result(1) := '0';
        when "0001" => result(1) := '1';
        when "0010" => result(1) := '2';
        when "0011" => result(1) := '3';
        when "0100" => result(1) := '4';
        when "0101" => result(1) := '5';
        when "0110" => result(1) := '6';
        when "0111" => result(1) := '7';
        when "1000" => result(1) := '8';
        when "1001" => result(1) := '9';
        when "1010" => result(1) := 'A';
        when "1011" => result(1) := 'B';
        when "1100" => result(1) := 'C';
    end case;
end function;

```



```

        when "1101" => result(1) := 'D';
        when "1110" => result(1) := 'E';
        when "1111" => result(1) := 'F';
        when others => result(1) := 'X';
    end case;

    -- Convert low nibble
    nibble := vec(3 downto 0);
    case nibble is
        when "0000" => result(2) := '0';
        when "0001" => result(2) := '1';
        when "0010" => result(2) := '2';
        when "0011" => result(2) := '3';
        when "0100" => result(2) := '4';
        when "0101" => result(2) := '5';
        when "0110" => result(2) := '6';
        when "0111" => result(2) := '7';
        when "1000" => result(2) := '8';
        when "1001" => result(2) := '9';
        when "1010" => result(2) := 'A';
        when "1011" => result(2) := 'B';
        when "1100" => result(2) := 'C';
        when "1101" => result(2) := 'D';
        when "1110" => result(2) := 'E';
        when "1111" => result(2) := 'F';
        when others => result(2) := 'X';
    end case;

    return result;
end function;

begin

    -- Clock generation using explicit process instead of concurrent assignment
    clk_process : process
    begin
        clk <= '0';
        wait for CLK_PERIOD / 2;
        clk <= '1';
        wait for CLK_PERIOD / 2;
    end process clk_process;

```

```
-- Fixed DUT instantiation - use generic map for STEP_TICKS
DUT : top_level_memory_less
  generic map (
    STEP_TICKS => STEP_TICKS
  )
  port map (
    FPGA_CLK      => clk,
    FPGA_RESET_N  => reset_n,
    FPGA_START    => start,
    LED_OUT       => led_out,
    LED_DONE      => led_done,
    HEX0          => hex0,
    HEX1          => hex1,
    HEX2          => hex2,
    HEX3          => hex3,
    HEX4          => hex4,
    HEX5          => hex5
  );

-- Test process
stimulus : process
begin
  -- Report test start
  report "Starting Task 1 Memory-less FSM Testbench" severity note;

  -- Initial reset
  report "Applying reset..." severity note;
  reset_n <= '0';
  start <= '0';
  wait for CLK_PERIOD * 2;

  -- Release reset
  report "Releasing reset..." severity note;
  reset_n <= '1';
  wait for CLK_PERIOD * 4;

  -- Verify initial state is IDLE
  assert get_state_from_hex(hex3, hex2, hex1, hex0) = "IDLE"
    report "Initial state not IDLE" severity error;
  assert led_done = '0'
```

```
    report "Done signal should be 0 in IDLE" severity error;
assert led_out = x"00"
    report "LED_OUT should be 0x00 in IDLE" severity error;
assert get_data_from_hex(hex5, hex4) = "OFF"
    report "HEX5-HEX4 should be OFF in IDLE" severity error;
report "Initial state verified: IDLE" severity note;

-- Test 1: Single cycle with start released
report "Test 1: Single cycle with start released" severity note;
start <= '1';
wait for CLK_PERIOD * STEP_TICKS;

-- Check STEP1 state
wait until rising_edge(clk);
wait for 1 ns;
assert get_state_from_hex(hex3, hex2, hex1, hex0) = "STEP1"
    report "State not STEP1 after start" severity error;
assert led_out = x"AA"
    report "LED_OUT should be 0xAA in STEP1" severity error;
assert get_data_from_hex(hex5, hex4) = "AA"
    report "HEX5-HEX4 should show AA in STEP1" severity error;
assert led_done = '0'
    report "Done signal should be 0 in STEP1" severity error;
report "State transition to STEP1 verified" severity note;

-- Release start after first step
start <= '0';

-- Wait for STEP2
wait for CLK_PERIOD * STEP_TICKS;
wait until rising_edge(clk);
wait for 1 ns;
assert get_state_from_hex(hex3, hex2, hex1, hex0) = "STEP2"
    report "State not STEP2" severity error;
assert led_out = x"BB"
    report "LED_OUT should be 0xBB in STEP2" severity error;
assert get_data_from_hex(hex5, hex4) = "BB"
    report "HEX5-HEX4 should show BB in STEP2" severity error;
report "State transition to STEP2 verified" severity note;

-- Wait for STEP3
```

```

wait for CLK_PERIOD * STEP_TICKS;
wait until rising_edge(clk);
wait for 1 ns;
assert get_state_from_hex(hex3, hex2, hex1, hex0) = "STEP3"
    report "State not STEP3" severity error;
assert led_out = x"CC"
    report "LED_OUT should be 0xCC in STEP3" severity error;
assert get_data_from_hex(hex5, hex4) = "CC"
    report "HEX5-HEX4 should show CC in STEP3" severity error;
report "State transition to STEP3 verified" severity note;

-- Wait for DONE state
wait for CLK_PERIOD * STEP_TICKS;
wait until rising_edge(clk);
wait for 1 ns;
assert get_state_from_hex(hex3, hex2, hex1, hex0) = "DONE"
    report "State not DONE" severity error;
assert led_out = x"00"
    report "LED_OUT should be 0x00 in DONE" severity error;
assert get_data_from_hex(hex5, hex4) = "OFF"
    report "HEX5-HEX4 should be OFF in DONE" severity error;
assert led_done = '1'
    report "Done signal should be 1 in DONE" severity error;
report "State transition to DONE verified" severity note;

-- Wait for return to IDLE
wait for CLK_PERIOD * STEP_TICKS;
wait until rising_edge(clk);
wait for 1 ns;
assert get_state_from_hex(hex3, hex2, hex1, hex0) = "IDLE"
    report "State not returned to IDLE" severity error;
assert led_done = '0'
    report "Done signal should be 0 in IDLE" severity error;
assert led_out = x"00"
    report "LED_OUT should be 0x00 in IDLE" severity error;
report "State returned to IDLE successfully" severity note;

-- Test 2: Start another cycle
report "Test 2: Starting another cycle" severity note;
start <= '1';
wait for CLK_PERIOD * STEP_TICKS;

```

```

-- Should be in STEP1 again
wait until rising_edge(clk);
wait for 1 ns;
assert get_state_from_hex(hex3, hex2, hex1, hex0) = "STEP1"
    report "Restart failed - not in STEP1" severity error;
assert led_out = x"AA"
    report "LED_OUT should be 0xAA in STEP1" severity error;
report "Restart test passed" severity note;

-- Let it run through full sequence again
report "Running full sequence again..." severity note;
for i in 1 to 3 loop
    wait for CLK_PERIOD * STEP_TICKS;
end loop;

-- Should be back in IDLE
wait until rising_edge(clk);
wait for 1 ns;
assert get_state_from_hex(hex3, hex2, hex1, hex0) = "IDLE"
    report "Final state not IDLE" severity error;

-- Test completed
report "Task 1 Memory-less FSM Testbench COMPLETED SUCCESSFULLY"
    severity note;
report "All state transitions and data outputs verified correctly"
    severity note;

wait;
end process stimulus;

-- Monitoring process for real-time state display
monitor : process
begin
    wait for CLK_PERIOD;
    loop
        wait until rising_edge(clk);
        wait for 1 ns;
        report "Time: " & time'image(now) &
            " | State: " & get_state_from_hex(hex3, hex2, hex1, hex0) &
            " | Data: " & get_data_from_hex(hex5, hex4) &

```

```

    " | LED_OUT: 0x" & to_hex_string(led_out) &
    " | done: " & std_logic'image(led_done);

    -- Exit when stimulus process finishes (at 2 microseconds)
    if now > 2 us then
        exit;
    end if;
end loop;
-- Final report to indicate simulation end
report "Monitoring stopped - simulation complete" severity note;
wait;
end process monitor;

end architecture testbench;

```

## UML Diagram

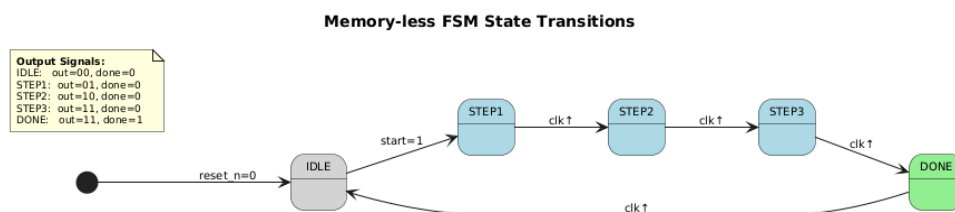


Figure 5: UML Diagram for Memory-less FSM

### Initialization (The Black Circle)

- **Visual Element:** The solid black circle on the left represents the **Initial Pseudo State**.
- **Transition:** The arrow labeled `reset_n=0` points directly to the **IDLE** state.
- **Code Correlation:** This corresponds to the asynchronous reset logic in the VHDL process: `if reset_n = '0' then current_state <= IDLE.`

### The Idle State (Gray)

- **State:** **IDLE** is the waiting state.
- **Transition Condition:** The arrow pointing to **STEP1** is labeled `start=1`.
- **Logic:** The system waits here until the `start` signal (Switch 0) goes high. As long as `start` is 0, the system remains in **IDLE** (implied self-loop) [cite: 81-82].
- **Code Correlation:** `if start = '1' then next_state <= STEP1.`

### The Active Sequence (Blue)

- **States:** **STEP1**, **STEP2**, and **STEP3** represent the active counting phase.
- **Transition Condition:** The arrows connecting these states are labeled **clk ↑** (Rising Edge).
- **Logic:** These are unconditional transitions. Once the FSM enters **STEP1**, it automatically advances to the next state on each subsequent clock cycle (or enable pulse).

- **Code Correlation:**

```

– when STEP1 => next_state <= STEP2.
– when STEP2 => next_state <= STEP3.

```

### Completion (Green)

- **State:** **DONE** indicates the cycle is finished.
- **Transition:** The arrow labeled **clk ↑** points back to **IDLE**.
- **Logic:** After one clock cycle in the **DONE** state, the system automatically resets to **IDLE** to await the next start command.
- **Code Correlation:** when **FSM\_DONE => next\_state <= IDLE**.

**Output Definitions** Our memoryless FSM has multiple outputs, here are the definition of each output we have **reg\_proc** process:

- **IDLE:** out=00, done=0 .
- **STEP1:** out=01, done=0 (Binary 1).
- **STEP2:** out=10, done=0 (Binary 2).
- **STEP3:** out=11, done=0 (Binary 3).
- **DONE:** out=11, done=1. Note that out stays at 11, but done flips to 1.

### 6.2.3 Timing Behavior (VWF)

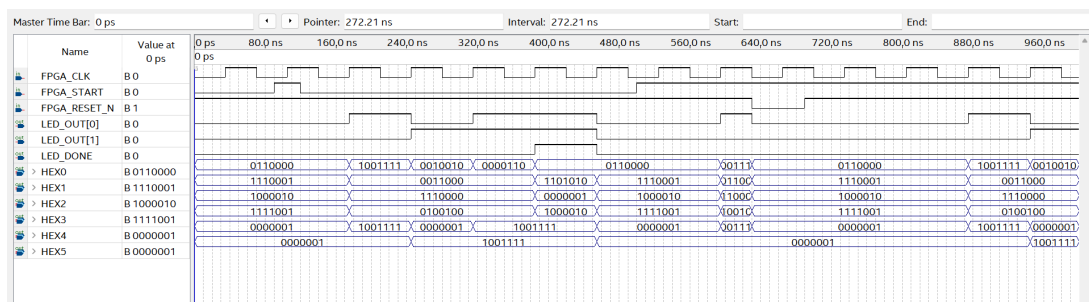


Figure 6: VWF Simulation Output for Memory-less FSM

## Memory-Less FSM Simulation Overview

This waveform provides an overview of a Memory-Less Finite State Machine (FSM) simulation. The FSM is designed to cycle continuously through five distinct states: `IDLE`, `STEP1`, `STEP2`, `STEP3`, and `FSM_DONE`. The simulation is running in “fast mode,” as the generic `STEP_TICKS` is set to 1. This configuration causes state transitions to occur on every clock cycle, bypassing the 1-second delay that would be implemented on physical hardware.

### Initialization and Reset (0 ns to 80 ns)

The simulation begins with an initialization and reset period from 0 to 80 nanoseconds. During this time, the `FPGA_RESET_N` signal is held low, forcing the system to remain in the `IDLE` state. Correspondingly, the `LED_OUT` displays “00” and the `LED_DONE` is off (0). The HEX display is hardcoded to show “IDLE 00,” with the segments for “I” and “E” illuminated accordingly.

### The Trigger (80 ns to 160 ns)

The operational sequence is triggered at approximately 80 nanoseconds. The `FPGA_RESET_N` signal goes high, releasing the system from reset, and is immediately followed by the `FPGA_START` signal going high. While in the `IDLE` state, the FSM logic detects that the start signal is ‘1’, which causes the `next_state` to become `STEP1`. On the very next clock edge, this state transition is executed, initiating the active cycle.

### The State Cycle (Active Operation)

Once the cycle begins, the FSM progresses through its sequence of active states.

In `STEP1`, the `LED_OUT` updates to binary “01” and the HEX display shows “StP1 01”.

The system then advances to `STEP2`, where the outputs change to “10” and “StP2 10”.

This is followed by `STEP3`, where `LED_OUT` becomes “11” and the display reads “StP3 11”.

Finally, the FSM enters the `FSM_DONE` state. Here, while the `LED_OUT` remains at “11”, the crucial `LED_DONE` signal is driven high (1), and the HEX display changes to show “dOnE 11”.

### The Loop (Continuous Operation)

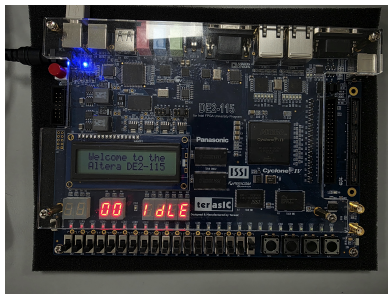
In the waveform, a continuous loop is observed. After reaching `FSM_DONE`, the system’s logic dictates a return to the `IDLE` state. However, because the `FPGA_START` signal is held high throughout the simulation, the FSM does not remain in `IDLE`. Instead, it immediately transitions back to `STEP1` on the subsequent clock cycle. This immediate restart creates the repetitive “sawtooth” pattern visible in the data lines as the FSM cycles endlessly through its states.



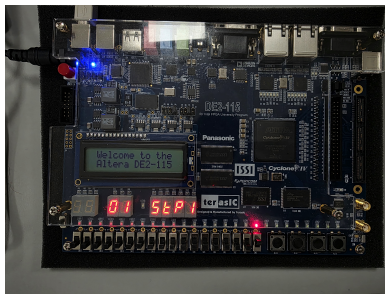
Table 2: Signal Reference Table

| Signal Name  | Function           | Observed Pattern                                                                                            |
|--------------|--------------------|-------------------------------------------------------------------------------------------------------------|
| FPGA_CLK     | System Clock       | A continuous square wave providing the timing reference for the FSM (50 MHz).                               |
| FPGA_RESET_N | Active-Low Reset   | Starts at Logic 0 to initialize the system, then goes High to enable operation.                             |
| FPGA_START   | Enable Switch      | When High (1), it triggers the transition from <i>IDLE</i> to the active stepping states.                   |
| LED_OUT[1:0] | 2-bit Data Counter | Cycles through the binary sequence $00 \rightarrow 01 \rightarrow 10 \rightarrow 11$ repeatedly.            |
| LED_DONE     | Completion Flag    | Pulses High (1) only during the <i>FSM_DONE</i> state to indicate the end of a cycle.                       |
| HEX0         | LSB Char Display   | Changes dynamically to represent the last character of the state name (e.g., 'E' for Idle, 'I' for Step 1). |

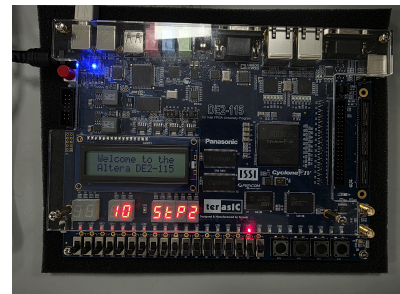
### 6.2.4 Practical Output



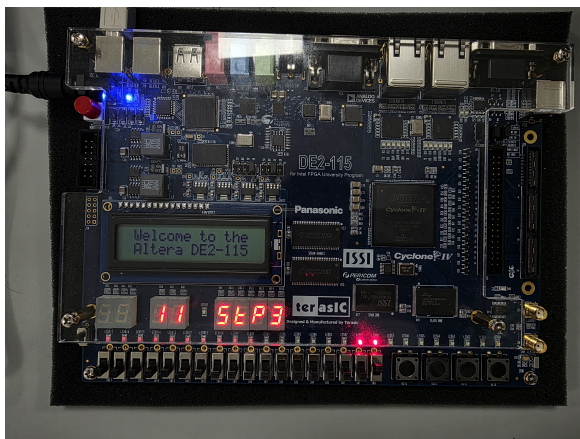
(a) IDLE State



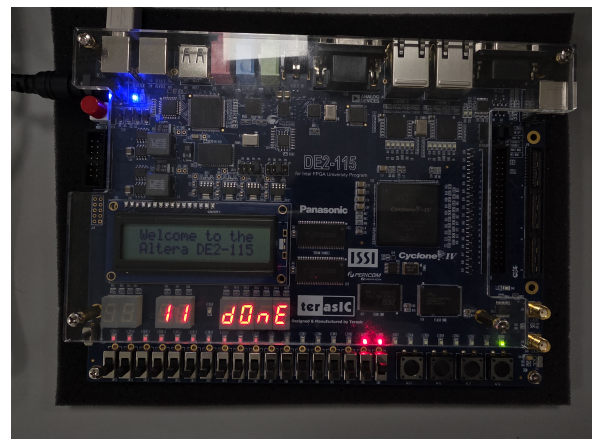
(b) STEP1 State



(c) STEP2 State



(d) STEP3 State



(e) DONE State

Figure 7: FPGA Implementation of Memory-less FSM

### The Idle State (Start)

- **State:** IDLE
- **HEX Display:** Shows “00 IdLE”
  - The code sets HEX5-4 to “00” and HEX3-0 to spell “IdLE”
- **Switches:** Notice Switch 0 (SW0) is in the DOWN position. This corresponds to `start = '0'`, keeping the machine waiting in this state.
- **LEDs:** The Red LEDs show binary 00 (all off).

### Step 1

- **State:** STEP1
- **HEX Display:** Shows “01 StP1”
  - The display logic updates HEX0 to show ‘1’ and the prefix changes to “StP”
- **Switches:** Switch 0 (SW0) is now flipped UP. This `start = '1'` signal triggered the transition out of IDLE.
- **LEDs:** LEDR0 is ON. This represents binary 01 (decimal 1).

### Step 2

- **State:** STEP2
- **HEX Display:** Shows “10 StP2”
  - The display updates HEX0 to show ‘2’
- **LEDs:** LEDR1 is ON and LEDR0 is OFF. This represents binary 10 (decimal 2).

### Step 3

- **State:** STEP3
- **HEX Display:** Shows “11 StP3”
  - The display updates HEX0 to show ‘3’
- **LEDs:** LEDR1 and LEDR0 are BOTH ON. This represents binary 11 (decimal 3).

### The Done State

- **State:** FSM\_DONE
- **HEX Display:** Shows “11 dOnE”
  - The text changes completely to indicate the cycle is finished
  - The number display remains at “11” because the data output holds its value during the done state

- **LEDs:**

- **Red LEDs:** Remain at 11
- **Green LED (LEDG0):** Notice the Green LED is ON above the push buttons. This corresponds to the `done <= '1'` logic in the VHDL code, signaling the completion of the FSM cycle

### 6.3. Memory-based FSM

#### 6.3.1 Code

Listing 4: VHDL for the Memory-based FSM Top-Level

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity top_level_memory_based is
    port (
        -- Standard FPGA I/O
        FPGA_CLK      : in  std_logic;           -- 50 MHz clock
        FPGA_RESET_N  : in  std_logic;           -- Active-low reset (KEY?)
        FPGA_START     : in  std_logic;           -- Start (SW or KEY)

        -- Outputs
        LED_OUT        : out std_logic_vector(7 downto 0);
        LED_DONE       : out std_logic;

        -- 7-Segment Displays
        HEX0            : out std_logic_vector(0 to 6);
        HEX1            : out std_logic_vector(0 to 6);
        HEX2            : out std_logic_vector(0 to 6);
        HEX3            : out std_logic_vector(0 to 6);
        HEX4            : out std_logic_vector(0 to 6);
        HEX5            : out std_logic_vector(0 to 6);
        HEX6            : out std_logic_vector(0 to 6)
    );
end entity top_level_memory_based;

architecture structure of top_level_memory_based is

    constant C_ADDR_WIDTH : integer := 2;

```

```

constant C_DATA_WIDTH : integer := 8;
constant C_MEM_SIZE   : integer := 4;

-- 50 MHz clock frequency
constant CLK_FREQ_HZ : integer := 50000000;

-- We want one FSM "step" per second:
constant STEP_TICKS : integer := CLK_FREQ_HZ; -- Use CLK_FREQ_HZ =
↪ 50,000,000 for the board and 1 for VWF simulation

signal s_step_counter : integer range 0 to STEP_TICKS - 1 := 0;
signal s_step_en      : std_logic := '0';

signal s_data_out : std_logic_vector(C_DATA_WIDTH - 1 downto 0);
signal s_addr     : std_logic_vector(C_ADDR_WIDTH - 1 downto 0);
signal s_done     : std_logic;

component memory_based_fsm is
    generic (
        C_ADDR_WIDTH : integer := 2;
        C_DATA_WIDTH : integer := 8;
        C_MEM_SIZE    : integer := 4
    );
    port (
        clk      : in  std_logic; -- 50 MHz clock
        reset_n  : in  std_logic;
        start    : in  std_logic;
        step_en  : in  std_logic; -- 1 Hz enable pulse
        data_out : out std_logic_vector(C_DATA_WIDTH - 1 downto 0);
        addr     : out std_logic_vector(C_ADDR_WIDTH - 1 downto 0);
        done     : out std_logic;
        HEX0     : out std_logic_vector(0 to 6);
        HEX1     : out std_logic_vector(0 to 6);
        HEX2     : out std_logic_vector(0 to 6);
        HEX3     : out std_logic_vector(0 to 6);
        HEX4     : out std_logic_vector(0 to 6);
        HEX5     : out std_logic_vector(0 to 6);
        HEX6     : out std_logic_vector(0 to 6)
    );
end component;

```

```

begin

-----
-- Clock-enable generator: 1 pulse per second (step_en)
-----

step_enable_proc : process(FPGA_CLK, FPGA_RESET_N)
begin
    if FPGA_RESET_N = '0' then
        s_step_counter <= 0;
        s_step_en <= '0';
    elsif rising_edge(FPGA_CLK) then
        if s_step_counter = STEP_TICKS - 1 then
            s_step_counter <= 0;
            s_step_en <= '1';    -- one-cycle pulse
        else
            s_step_counter <= s_step_counter + 1;
            s_step_en <= '0';
        end if;
    end if;
end process step_enable_proc;

-----
-- Memory-based FSM instantiation
-----

U_MEMORY_BASED : memory_based_fsm
    generic map (
        C_ADDR_WIDTH => C_ADDR_WIDTH,
        C_DATA_WIDTH => C_DATA_WIDTH,
        C_MEM_SIZE    => C_MEM_SIZE
    )
    port map (
        clk          => FPGA_CLK,          -- main 50 MHz clock
        reset_n      => FPGA_RESET_N,
        start        => FPGA_START,
        step_en      => s_step_en,
        data_out     => s_data_out,
        addr         => s_addr,
        done         => s_done,
        HEX0         => HEX0,
        HEX1         => HEX1,
        HEX2         => HEX2,

```

```
        HEX3    => HEX3,  
        HEX4    => HEX4,  
        HEX5    => HEX5,  
        HEX6    => HEX6  
    );  
  
    LED_OUT  <= s_data_out;  
    LED_DONE <= s_done;  
  
end architecture structure;
```

Listing 5: VHDL code for the Memory-based FSM main body

```
library ieee;  
use ieee.std_logic_1164.all;  
use ieee.numeric_std.all;  
  
entity memory_based_fsm is  
    generic (  
        C_ADDR_WIDTH : integer := 2;  
        C_DATA_WIDTH : integer := 8;  
        C_MEM_SIZE    : integer := 4  
    );  
    port (  
        clk          : in  std_logic;  -- 50 MHz clock  
        reset_n      : in  std_logic;  
        start        : in  std_logic;  
        step_en       : in  std_logic;  -- 1 Hz enable pulse  
        data_out      : out std_logic_vector(C_DATA_WIDTH - 1 downto 0);  
        addr          : out std_logic_vector(C_ADDR_WIDTH - 1 downto 0);  
        done          : out std_logic;  
        HEX0          : out std_logic_vector(0 to 6);  
        HEX1          : out std_logic_vector(0 to 6);  
        HEX2          : out std_logic_vector(0 to 6);  
        HEX3          : out std_logic_vector(0 to 6);  
        HEX4          : out std_logic_vector(0 to 6);  
        HEX5          : out std_logic_vector(0 to 6);  
        HEX6          : out std_logic_vector(0 to 6)  
    );  
end entity memory_based_fsm;
```

```

architecture rtl of memory_based_fsm is

    -----
    -- States
    -----

    type state_type is (IDLE, READ_STATE, PROCESS_DATA, INC_ADDR, FSM_DONE);
    signal current_state, next_state : state_type;

    -----
    -- Internal signals
    -----

    signal s_addr_reg : unsigned(C_ADDR_WIDTH - 1 downto 0) := (others => '0');
    signal s_data_reg : std_logic_vector(C_DATA_WIDTH - 1 downto 0) := (others
    ↪ => '0');
    signal s_ram_data : std_logic_vector(C_DATA_WIDTH - 1 downto 0);

    -- Registered outputs
    signal s_data_out_reg : std_logic_vector(C_DATA_WIDTH - 1 downto 0) := (
    ↪ others => '0');
    signal s_addr_out_reg : std_logic_vector(C_ADDR_WIDTH - 1 downto 0) := (
    ↪ others => '0');
    signal s_done_reg      : std_logic := '0';

    signal s_hex0_reg, s_hex1_reg, s_hex2_reg, s_hex3_reg : std_logic_vector(0
    ↪ to 6);
    signal s_hex4_reg, s_hex5_reg, s_hex6_reg              : std_logic_vector(0
    ↪ to 6);

    -----
    -- RAM component
    -----

    component RAM4x8 is
        port (
            address : in  std_logic_vector(1 downto 0);
            clock    : in  std_logic;
            data     : in  std_logic_vector(7 downto 0);
            rden     : in  std_logic;
            wren     : in  std_logic;
            q        : out std_logic_vector(7 downto 0)
        );
    end component;

```

```

begin

-----
-- RAM instance: clocked at 50 MHz
-----

U_RAM : RAM4x8
  port map (
    address => std_logic_vector(s_addr_reg),
    clock   => clk,
    data    => (others => '0'),
    rden    => '1',
    wren    => '0',
    q       => s_ram_data
  );

-----
-- Sequential process: state-dependent registers + outputs
-- Runs at 50 MHz, but only updates when step_en = '1'
-----

reg_proc : process(clk, reset_n)
begin
  if reset_n = '0' then
    s_addr_reg      <= (others => '0');
    s_data_reg      <= (others => '0');
    s_data_out_reg  <= (others => '0');
    s_addr_out_reg  <= (others => '0');
    s_done_reg      <= '0';

    -- Show "IDLE" and 00, address 0
    s_hex3_reg <= "1111001"; -- I
    s_hex2_reg <= "1000010"; -- d
    s_hex1_reg <= "1110001"; -- L
    s_hex0_reg <= "0110000"; -- E
    s_hex4_reg <= "0000001"; -- 0
    s_hex5_reg <= "0000001"; -- 0
    s_hex6_reg <= "0000001"; -- 0

  elsif rising_edge(clk) then
    if step_en = '1' then
      -----

```



```

-- Internal address/data updates based on current state
-----

case current_state is
  when FSM_DONE =>
    -- After finishing, clear internal regs
    s_data_reg <= (others => '0');
    s_addr_reg <= (others => '0');

  when INC_ADDR =>
    -- Move to next address; RAM will output new data
    -- on a later clock, well before the next step_en.
    s_addr_reg <= s_addr_reg + 1;

  when PROCESS_DATA =>
    -- Place for any "processing" on data
    s_data_reg <= s_ram_data;

  when others =>
    null;
end case;

-----

-- Registered outputs (Moore-style)
-----

s_data_out_reg <= (others => '0');
s_addr_out_reg <= std_logic_vector(s_addr_reg);
s_done_reg      <= '0';

case current_state is
  when IDLE =>
    s_data_out_reg <= (others => '0');
    s_done_reg      <= '0';

  when READ_STATE =>
    -- First visible state for each address's data
    s_data_out_reg <= s_ram_data;
    s_done_reg      <= '0';

  when PROCESS_DATA =>
    s_data_out_reg <= s_ram_data;
    s_done_reg      <= '0';

```

```

when INC_ADDR =>
    -- Address changed this step; RAM output still shows
    -- previous address until the next clock edge.
    s_data_out_reg <= s_ram_data;
    s_done_reg      <= '0';

when FSM_DONE =>
    s_data_out_reg <= s_ram_data;
    s_done_reg      <= '1';
end case;

-----
-- HEX0HEX3: state name display
-----

case current_state is
when IDLE =>
    s_hex3_reg <= "1111001"; -- I
    s_hex2_reg <= "1000010"; -- d
    s_hex1_reg <= "1110001"; -- L
    s_hex0_reg <= "0110000"; -- E

when READ_STATE =>
    s_hex3_reg <= "1111010"; -- r
    s_hex2_reg <= "0110000"; -- E
    s_hex1_reg <= "0001000"; -- A
    s_hex0_reg <= "1000010"; -- d

when PROCESS_DATA =>
    s_hex3_reg <= "0011000"; -- P
    s_hex2_reg <= "1111010"; -- r
    s_hex1_reg <= "0000001"; -- 0
    s_hex0_reg <= "0110001"; -- C

when INC_ADDR =>
    s_hex3_reg <= "1111001"; -- I
    s_hex2_reg <= "1101010"; -- n
    s_hex1_reg <= "0110001"; -- C
    s_hex0_reg <= "0001000"; -- A

when FSM_DONE =>

```

```

        s_hex3_reg <= "1000010"; -- d
        s_hex2_reg <= "0000001"; -- 0
        s_hex1_reg <= "1101010"; -- n
        s_hex0_reg <= "0110000"; -- E
    end case;

    -----
    -- HEX4HEX5: RAM data (high nibble / low nibble)
    -----

    if current_state = IDLE then
        s_hex4_reg <= "0000001"; -- 0
        s_hex5_reg <= "0000001"; -- 0
    else
        -- High nibble
        case s_ram_data(7 downto 4) is
            when "0000" => s_hex4_reg <= "0000001"; -- 0
            when "0001" => s_hex4_reg <= "1001111"; -- 1
            when "0010" => s_hex4_reg <= "0010010"; -- 2
            when "0011" => s_hex4_reg <= "0000110"; -- 3
            when "0100" => s_hex4_reg <= "1001100"; -- 4
            when "0101" => s_hex4_reg <= "0100100"; -- 5
            when "0110" => s_hex4_reg <= "0100000"; -- 6
            when "0111" => s_hex4_reg <= "0001111"; -- 7
            when "1000" => s_hex4_reg <= "0000000"; -- 8
            when "1001" => s_hex4_reg <= "0001100"; -- 9
            when "1010" => s_hex4_reg <= "0001000"; -- A
            when "1011" => s_hex4_reg <= "1100000"; -- B
            when "1100" => s_hex4_reg <= "0110001"; -- C
            when "1101" => s_hex4_reg <= "1000010"; -- D
            when "1110" => s_hex4_reg <= "0110000"; -- E
            when others => s_hex4_reg <= "0111000"; -- F
        end case;

        -- Low nibble
        case s_ram_data(3 downto 0) is
            when "0000" => s_hex5_reg <= "0000001"; -- 0
            when "0001" => s_hex5_reg <= "1001111"; -- 1
            when "0010" => s_hex5_reg <= "0010010"; -- 2
            when "0011" => s_hex5_reg <= "0000110"; -- 3
            when "0100" => s_hex5_reg <= "1001100"; -- 4
            when "0101" => s_hex5_reg <= "0100100"; -- 5

```

```

        when "0110" => s_hex5_reg <= "0100000"; -- 6
        when "0111" => s_hex5_reg <= "0001111"; -- 7
        when "1000" => s_hex5_reg <= "0000000"; -- 8
        when "1001" => s_hex5_reg <= "0001100"; -- 9
        when "1010" => s_hex5_reg <= "0001000"; -- A
        when "1011" => s_hex5_reg <= "1100000"; -- B
        when "1100" => s_hex5_reg <= "0110001"; -- C
        when "1101" => s_hex5_reg <= "1000010"; -- D
        when "1110" => s_hex5_reg <= "0110000"; -- E
        when others => s_hex5_reg <= "0111000"; -- F
    end case;
end if;

-----

-- HEX6: address (03)
-----

case to_integer(s_addr_reg) is
    when 0 => s_hex6_reg <= "0000001"; -- 0
    when 1 => s_hex6_reg <= "1001111"; -- 1
    when 2 => s_hex6_reg <= "0010010"; -- 2
    when 3 => s_hex6_reg <= "0000110"; -- 3
    when others => s_hex6_reg <= "1111111"; -- off
end case;
end if; -- step_en
end if; -- rising_edge(clk)
end process reg_proc;

-----

-- State register: only advances when step_en = '1'
-----

state_register_proc : process(clk, reset_n)
begin
    if reset_n = '0' then
        current_state <= IDLE;
    elsif rising_edge(clk) then
        if step_en = '1' then
            current_state <= next_state;
        end if;
    end if;
end process state_register_proc;

```

```

-----
-- Next-state logic (purely combinational)
-----

next_state_logic_proc : process(current_state, start, s_addr_reg)
    variable v_last_addr_processed : boolean;
begin
    v_last_addr_processed := (s_addr_reg = to_unsigned(C_MEM_SIZE - 1,
↪ C_ADDR_WIDTH));

    case current_state is
        when IDLE =>
            if start = '1' then
                next_state <= READ_STATE;
            else
                next_state <= IDLE;
            end if;

        when READ_STATE =>
            next_state <= PROCESS_DATA;

        when PROCESS_DATA =>
            if v_last_addr_processed then
                next_state <= FSM_DONE;
            else
                next_state <= INC_ADDR;
            end if;

        when INC_ADDR =>
            next_state <= READ_STATE;

        when FSM_DONE =>
            next_state <= IDLE;

        when others =>
            next_state <= IDLE;
    end case;
end process next_state_logic_proc;

-----
-- Output port mappings
-----

```

```

data_out <= s_data_out_reg;
addr      <= s_addr_out_reg;
done      <= s_done_reg;

HEX0 <= s_hex0_reg;
HEX1 <= s_hex1_reg;
HEX2 <= s_hex2_reg;
HEX3 <= s_hex3_reg;
HEX4 <= s_hex4_reg;
HEX5 <= s_hex5_reg;
HEX6 <= s_hex6_reg;

end architecture rtl;

```

Listing 6: VHDL code for the Memory-based FSM RAM module

```

-- megafunction wizard: %RAM: 1-PORT%
-- GENERATION: STANDARD
-- VERSION: WM1.0
-- MODULE: altsyncram

-- =====
-- File Name: RAM4x8.vhd
-- Megafunction Name(s):
--     altsyncram
--
-- Simulation Library Files(s):
--     altera_mf
-- =====
-- *****
-- THIS IS A WIZARD-GENERATED FILE. DO NOT EDIT THIS FILE!
--
-- 20.1.1 Build 720 11/11/2020 SJ Lite Edition
-- *****

--Copyright (C) 2020 Intel Corporation. All rights reserved.
--Your use of Intel Corporation's design tools, logic functions
--and other software and tools, and any partner logic
--functions, and any output files from any of the foregoing
--(including device programming or simulation files), and any

```

```
--associated documentation or information are expressly subject
--to the terms and conditions of the Intel Program License
--Subscription Agreement, the Intel Quartus Prime License Agreement,
--the Intel FPGA IP License Agreement, or other applicable license
--agreement, including, without limitation, that your use is for
--the sole purpose of programming logic devices manufactured by
--Intel and sold by Intel or its authorized distributors. Please
--refer to the applicable agreement for further details, at
--https://fpgasoftware.intel.com/eula.
```

```
LIBRARY ieee;
```

```
USE ieee.std_logic_1164.all;
```

```
LIBRARY altera_mf;
```

```
USE altera_mf.altera_mf_components.all;
```

```
ENTITY RAM4x8 IS
```

```
PORT
```

```
(
    address    : IN STD_LOGIC_VECTOR (1 DOWNTO 0);
    clock      : IN STD_LOGIC      := '1';
    data       : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
    rden       : IN STD_LOGIC      := '1';
    wren       : IN STD_LOGIC ;
    q          : OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
);
```

```
END RAM4x8;
```

```
ARCHITECTURE SYN OF ram4x8 IS
```

```
    SIGNAL sub_wire0 : STD_LOGIC_VECTOR (7 DOWNTO 0);
```

```
BEGIN
```

```
    q    <= sub_wire0(7 DOWNTO 0);
```

```
    altsyncram_component : altsyncram
```

```
    GENERIC MAP (
```

```
        clock_enable_input_a => "BYPASS",
```

```
        clock_enable_output_a => "BYPASS",
```

```

    init_file => "memory_data.mif",
    intended_device_family => "Cyclone IV E",
    lpm_hint => "ENABLE_RUNTIME_MOD=NO",
    lpm_type => "altsyncram",
    numwords_a => 4,
    operation_mode => "SINGLE_PORT",
    outdata_aclr_a => "NONE",
    outdata_reg_a => "UNREGISTERED",
    power_up_uninitialized => "FALSE",
    ram_block_type => "M9K",
    read_during_write_mode_port_a => "NEW_DATA_NO_NBE_READ",
    widthad_a => 2,
    width_a => 8,
    width_byteena_a => 1
)
PORT MAP (
    address_a => address,
    clock0 => clock,
    data_a => data,
    rden_a => rden,
    wren_a => wren,
    q_a => sub_wire0
);

END SYN;

-- =====
-- CNX file retrieval info
-- =====
-- Retrieval info: PRIVATE: ADDRESSSTALL_A NUMERIC "0"
-- Retrieval info: PRIVATE: AclrAddr NUMERIC "0"
-- Retrieval info: PRIVATE: AclrByte NUMERIC "0"
-- Retrieval info: PRIVATE: AclrData NUMERIC "0"
-- Retrieval info: PRIVATE: AclrOutput NUMERIC "0"
-- Retrieval info: PRIVATE: BYTE_ENABLE NUMERIC "0"
-- Retrieval info: PRIVATE: BYTE_SIZE NUMERIC "8"
-- Retrieval info: PRIVATE: BlankMemory NUMERIC "0"
-- Retrieval info: PRIVATE: CLOCK_ENABLE_INPUT_A NUMERIC "0"
-- Retrieval info: PRIVATE: CLOCK_ENABLE_OUTPUT_A NUMERIC "0"

```



```

-- Retrieval info: PRIVATE: Clken NUMERIC "0"
-- Retrieval info: PRIVATE: DataBusSeparated NUMERIC "1"
-- Retrieval info: PRIVATE: IMPLEMENT_IN_LES NUMERIC "0"
-- Retrieval info: PRIVATE: INIT_FILE_LAYOUT STRING "PORT_A"
-- Retrieval info: PRIVATE: INIT_TO_SIM_X NUMERIC "0"
-- Retrieval info: PRIVATE: INTENDED_DEVICE_FAMILY STRING "Cyclone IV E"
-- Retrieval info: PRIVATE: JTAG_ENABLED NUMERIC "0"
-- Retrieval info: PRIVATE: JTAG_ID STRING "NONE"
-- Retrieval info: PRIVATE: MAXIMUM_DEPTH NUMERIC "0"
-- Retrieval info: PRIVATE: MIFfilename STRING "../memory_data.mif"
-- Retrieval info: PRIVATE: NUMWORDS_A NUMERIC "4"
-- Retrieval info: PRIVATE: RAM_BLOCK_TYPE NUMERIC "2"
-- Retrieval info: PRIVATE: READ_DURING_WRITE_MODE_PORT_A NUMERIC "3"
-- Retrieval info: PRIVATE: RegAddr NUMERIC "1"
-- Retrieval info: PRIVATE: RegData NUMERIC "1"
-- Retrieval info: PRIVATE: RegOutput NUMERIC "0"
-- Retrieval info: PRIVATE: SYNTH_WRAPPER_GEN_POSTFIX STRING "0"
-- Retrieval info: PRIVATE: SingleClock NUMERIC "1"
-- Retrieval info: PRIVATE: UseDQRAM NUMERIC "1"
-- Retrieval info: PRIVATE: WRCONTROL_ACLR_A NUMERIC "0"
-- Retrieval info: PRIVATE: WidthAddr NUMERIC "2"
-- Retrieval info: PRIVATE: WidthData NUMERIC "8"
-- Retrieval info: PRIVATE: rden NUMERIC "1"
-- Retrieval info: LIBRARY: altera_mf altera_mf.altera_mf_components.all
-- Retrieval info: CONSTANT: CLOCK_ENABLE_INPUT_A STRING "BYPASS"
-- Retrieval info: CONSTANT: CLOCK_ENABLE_OUTPUT_A STRING "BYPASS"
-- Retrieval info: CONSTANT: INIT_FILE STRING "../memory_data.mif"
-- Retrieval info: CONSTANT: INTENDED_DEVICE_FAMILY STRING "Cyclone IV E"
-- Retrieval info: CONSTANT: LPM_HINT STRING "ENABLE_RUNTIME_MOD=NO"
-- Retrieval info: CONSTANT: LPM_TYPE STRING "altsyncram"
-- Retrieval info: CONSTANT: NUMWORDS_A NUMERIC "4"
-- Retrieval info: CONSTANT: OPERATION_MODE STRING "SINGLE_PORT"
-- Retrieval info: CONSTANT: OUTDATA_ACLR_A STRING "NONE"
-- Retrieval info: CONSTANT: OUTDATA_REG_A STRING "UNREGISTERED"
-- Retrieval info: CONSTANT: POWER_UP_UNINITIALIZED STRING "FALSE"
-- Retrieval info: CONSTANT: RAM_BLOCK_TYPE STRING "M9K"
-- Retrieval info: CONSTANT: READ_DURING_WRITE_MODE_PORT_A STRING "
    ↪ NEW_DATA_NO_NBE_READ"
-- Retrieval info: CONSTANT: WIDTHAD_A NUMERIC "2"
-- Retrieval info: CONSTANT: WIDTH_A NUMERIC "8"
-- Retrieval info: CONSTANT: WIDTH_BYTEENA_A NUMERIC "1"

```

```

-- Retrieval info: USED_PORT: address 0 0 2 0 INPUT NODEFVAL "address[1..0]"
-- Retrieval info: USED_PORT: clock 0 0 0 0 INPUT VCC "clock"
-- Retrieval info: USED_PORT: data 0 0 8 0 INPUT NODEFVAL "data[7..0]"
-- Retrieval info: USED_PORT: q 0 0 8 0 OUTPUT NODEFVAL "q[7..0]"
-- Retrieval info: USED_PORT: rden 0 0 0 0 INPUT VCC "rden"
-- Retrieval info: USED_PORT: wren 0 0 0 0 INPUT NODEFVAL "wren"
-- Retrieval info: CONNECT: @address_a 0 0 2 0 address 0 0 2 0
-- Retrieval info: CONNECT: @clock0 0 0 0 0 clock 0 0 0 0
-- Retrieval info: CONNECT: @data_a 0 0 8 0 data 0 0 8 0
-- Retrieval info: CONNECT: @rden_a 0 0 0 0 rden 0 0 0 0
-- Retrieval info: CONNECT: @wren_a 0 0 0 0 wren 0 0 0 0
-- Retrieval info: CONNECT: q 0 0 8 0 @q_a 0 0 8 0
-- Retrieval info: GEN_FILE: TYPE_NORMAL RAM4x8.vhd TRUE
-- Retrieval info: GEN_FILE: TYPE_NORMAL RAM4x8.inc FALSE
-- Retrieval info: GEN_FILE: TYPE_NORMAL RAM4x8.cmp TRUE
-- Retrieval info: GEN_FILE: TYPE_NORMAL RAM4x8.bsf FALSE
-- Retrieval info: GEN_FILE: TYPE_NORMAL RAM4x8_inst.vhd FALSE
-- Retrieval info: LIB_FILE: altera_mf

```

### 6.3.2 Test Bench

Listing 7: VHDL code for the Memory-based FSM Test Bench

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity tb_memory_based_fsm is
end entity tb_memory_based_fsm;

architecture testbench of tb_memory_based_fsm is

    -- Component Declaration for the Unit Under Test (UUT)
    component memory_based_fsm is
        generic (
            C_ADDR_WIDTH : integer := 2;
            C_DATA_WIDTH  : integer := 8;
            C_MEM_SIZE    : integer := 4
        );
        port (
            clk          : in  std_logic;

```

```

    reset_n    : in  std_logic;
    start      : in  std_logic;
    step_en    : in  std_logic;  -- NEW

    data_out   : out std_logic_vector(7 downto 0);
    addr       : out std_logic_vector(1 downto 0);
    done       : out std_logic;

    HEX0       : out std_logic_vector(0 to 6);
    HEX1       : out std_logic_vector(0 to 6);
    HEX2       : out std_logic_vector(0 to 6);
    HEX3       : out std_logic_vector(0 to 6); -- NEW
    HEX4       : out std_logic_vector(0 to 6); -- NEW
    HEX5       : out std_logic_vector(0 to 6); -- NEW
    HEX6       : out std_logic_vector(0 to 6)  -- NEW
  );
end component;

-- Testbench Signals
signal clk      : std_logic := '0';
signal reset_n  : std_logic := '0';
signal start    : std_logic := '0';
signal step_en  : std_logic := '0';  -- NEW

signal data_out : std_logic_vector(7 downto 0);
signal addr     : std_logic_vector(1 downto 0);
signal done     : std_logic;

signal HEX0, HEX1, HEX2 : std_logic_vector(0 to 6);
signal HEX3, HEX4, HEX5, HEX6 : std_logic_vector(0 to 6);  -- NEW

-- Clock period definitions
constant CLK_PERIOD : time := 20 ns; -- 50 MHz clock for testbench

-- Test sequence tracking
type test_state_type is (TEST_IDLE, TEST_READ, TEST_PROCESS, TEST_INC,
  ⇨ TEST_DONE);
signal test_state : test_state_type := TEST_IDLE;
signal cycle_count : integer := 0;
signal test_passed : boolean := true;

```

```
begin

-----
-- Instantiate the Unit Under Test (UUT)
-----

uut: memory_based_fsm
  generic map (
    C_ADDR_WIDTH => 2,
    C_DATA_WIDTH => 8,
    C_MEM_SIZE   => 4
  )
  port map (
    clk      => clk,
    reset_n  => reset_n,
    start    => start,
    step_en  => step_en,    -- NEW

    data_out => data_out,
    addr     => addr,
    done     => done,
    HEX0     => HEX0,
    HEX1     => HEX1,
    HEX2     => HEX2,
    HEX3     => HEX3,      -- NEW
    HEX4     => HEX4,      -- NEW
    HEX5     => HEX5,      -- NEW
    HEX6     => HEX6       -- NEW
  );

-----
-- Clock generation process (50 MHz)
-----

clk_process : process
begin
  while true loop
    clk <= '0';
    wait for CLK_PERIOD/2;
    clk <= '1';
    wait for CLK_PERIOD/2;
  end loop;
end process;
```

```

-----
-- For simulation: let the FSM advance EVERY clock (step_en = '1')
-- This makes your old test logic (20 cycles) still valid.
-----

step_en <= '1';

-----

-- Stimulus process
-----

stimulus_process : process
    variable test_step : integer := 0;
begin
    -- Initialize and reset
    report "Starting Testbench for Memory-Based FSM";
    reset_n <= '0';
    start    <= '0';
    wait for CLK_PERIOD * 2;

    -- Release reset
    reset_n <= '1';
    wait for CLK_PERIOD * 2;

    -- Test 1: Verify IDLE state
    report "Test 1: Verifying IDLE state";
    assert data_out = "00000000" report "IDLE state data_out should be 0"
    ↪ severity error;
    assert addr      = "00"          report "IDLE state addr should be 00"
    ↪ severity error;
    assert done      = '0'          report "IDLE state done should be 0"
    ↪ severity error;
    wait for CLK_PERIOD;

    -- Test 2: Start the FSM
    report "Test 2: Starting FSM operation";
    start <= '1';
    wait for CLK_PERIOD;
    start <= '0';

    -- Wait for sequence to complete
    report "Test 3: Monitoring FSM sequence";

```

```

-- Monitor the sequence through all 4 memory locations
for i in 1 to 20 loop -- Allow enough cycles for complete sequence
    wait for CLK_PERIOD;
    cycle_count <= cycle_count + 1;

    -- Track state based on outputs
    if done = '1' then
        test_state <= TEST_DONE;
        report "FSM reached DONE state at cycle " & integer'image(
↪ cycle_count);
        exit;
    elsif data_out /= "00000000" then
        if addr = "00" and data_out = "10101010" then -- 170 (AA)
            test_state <= TEST_READ;
        elsif addr = "01" and data_out = "10111011" then -- 187 (BB)
            test_state <= TEST_PROCESS;
        elsif addr = "10" and data_out = "11001100" then -- 204 (CC)
            test_state <= TEST_INC;
        elsif addr = "11" and data_out = "11011101" then -- 221 (DD)
            test_state <= TEST_DONE;
        end if;
    else
        test_state <= TEST_IDLE;
    end if;
end loop;

-- Give outputs time to settle after DONE is detected
wait for CLK_PERIOD / 2;

-- Verify completion
assert done = '1'
    report "FSM should have reached DONE state"
    severity error;

assert data_out = "11011101"
    report "Final data should be 221 (DD)"
    severity error;

assert addr = "11"
    report "Final address should be 11"

```

```

        severity error;

-- Test 4: Verify return to IDLE
report "Test 4: Verifying return to IDLE";
wait for CLK_PERIOD * 3; -- Wait for return to IDLE

assert data_out = "00000000" report "Should return to 0 in IDLE"
↪ severity error;
    assert done      = '0'          report "Done should be 0 after completion"
↪ severity error;

-- Test 5: Start second sequence
report "Test 5: Starting second sequence";

-- Reset debug counters for clarity
cycle_count <= 0;
test_state  <= TEST_IDLE;

start <= '1';
wait for CLK_PERIOD;
start <= '0';

-- Monitor the second sequence in exactly the same way
for i in 1 to 20 loop
    wait for CLK_PERIOD;
    cycle_count <= cycle_count + 1;

    if done = '1' then
        test_state <= TEST_DONE;
        exit;
    elsif data_out /= "00000000" then
        if addr = "00" and data_out = "10101010" then -- 170 (AA
↪ )

            test_state <= TEST_READ;
        elsif addr = "01" and data_out = "10111011" then -- 187 (BB
↪ )

            test_state <= TEST_PROCESS;
        elsif addr = "10" and data_out = "11001100" then -- 204 (CC
↪ )

            test_state <= TEST_INC;

```

```

        elsif addr = "11" and data_out = "11011101" then    -- 221 (DD
↪ )

            test_state <= TEST_DONE;
            end if;
        else
            test_state <= TEST_IDLE;
            end if;
        end loop;

        -- Check that the second run really completed
        assert done = '1' report "Second sequence should complete" severity
↪ error;

        -- Final report
        if test_passed then
            report "ALL TESTS PASSED - Memory-Based FSM is working correctly!";
            report "Sequence: 170(AA) -> 187(BB) -> 204(CC) -> 221(DD) -> DONE"
↪ ;
        else
            report "SOME TESTS FAILED - Check FSM implementation" severity
↪ error;
        end if;

        report "Testbench completed successfully";

        -- Cleanly stop the simulation
        report "Simulation completed, stopping simulation now" severity failure
↪ ;

        wait;
    end process;

    -----
    -- Monitor process for detailed state tracking
    -----

    monitor_process : process(clk)
        variable prev_data : std_logic_vector(7 downto 0) := (others => '0');
        variable prev_addr : std_logic_vector(1 downto 0) := "00";
    begin
        if rising_edge(clk) then
            -- Report state changes

```



```

        if data_out /= prev_data or addr /= prev_addr then
            case data_out is
                when "10101010" =>
                    report "Cycle " & integer'image(cycle_count) & ":
↪ Address " &
                                integer'image(to_integer(unsigned(addr))) & " =
↪ 170 (AA)";
                when "10111011" =>
                    report "Cycle " & integer'image(cycle_count) & ":
↪ Address " &
                                integer'image(to_integer(unsigned(addr))) & " =
↪ 187 (BB)";
                when "11001100" =>
                    report "Cycle " & integer'image(cycle_count) & ":
↪ Address " &
                                integer'image(to_integer(unsigned(addr))) & " =
↪ 204 (CC)";
                when "11011101" =>
                    report "Cycle " & integer'image(cycle_count) & ":
↪ Address " &
                                integer'image(to_integer(unsigned(addr))) & " =
↪ 221 (DD)";
                when "00000000" =>
                    report "Cycle " & integer'image(cycle_count) & ": IDLE
↪ state (0)";
                when others =>
                    report "Cycle " & integer'image(cycle_count) & ":
↪ Unknown data " &
                                integer'image(to_integer(unsigned(data_out)));
            end case;

            prev_data := data_out;
            prev_addr := addr;
        end if;
    end if;
end process;

-----
-- 7-Segment display verification (optional)
-----

hex_monitor : process(clk)

```

```

    variable hex0_prev, hex1_prev, hex2_prev : std_logic_vector(0 to 6) :=
    ↪ (others => '1');
    begin
        if rising_edge(clk) then
            if HEX0 /= hex0_prev or HEX1 /= hex1_prev or HEX2 /= hex2_prev then
                -- Optional: add checks on HEX patterns if needed
                hex0_prev := HEX0;
                hex1_prev := HEX1;
                hex2_prev := HEX2;
            end if;
        end if;
    end process;

end architecture testbench;

```

## UML Diagram

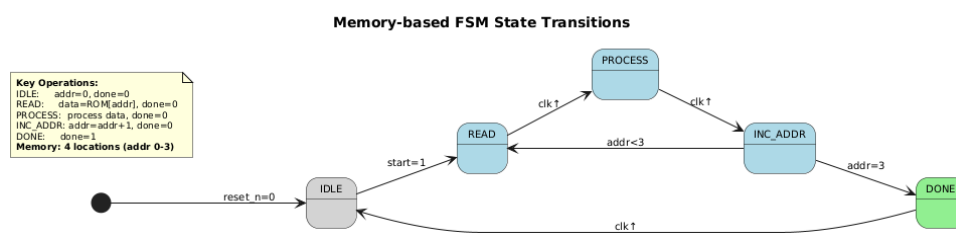


Figure 8: UML Diagram for Memory-based FSM

## Memory-Based FSM Control System Overview

This UML diagram represents a control system that iterates through a memory block (RAM) containing 4 locations. Unlike the previous “memory-less” version, this FSM uses an internal address counter to fetch specific data values (AA, BB, CC, DD) from a RAM block instantiated in the code.

### Initialization (IDLE State)

- **Visual:** The process begins at the Initial Pseudo State (black circle) and transitions to IDLE on `reset_n=0`.
- **Code Logic:** The system waits in IDLE with the address register `s_addr_reg` cleared to 0 and the `done` signal set to 0.
- **Transition:** The machine remains here until `start=1`. When the start signal is received, it moves to the READ state.

### The READ State (READ\_STATE)

- **Visual:** The state labeled READ.
- **Operation:** In this state, the FSM exposes the data retrieved from the RAM (`s_ram_data`) to the output `data_out`.
- **Hardware:** The RAM component (RAM4x8) is constantly reading based on the current address `s_addr_reg`.
- **Transition:** On the next clock edge (or step enable), the system unconditionally transitions to PROCESS.

### The PROCESS State (PROCESS\_DATA)

- **Visual:** The state labeled PROCESS.
- **Operation:** This state is a placeholder for data manipulation. In the code, it holds the RAM data on the output bus.
- **Branch Logic:**
  - The VHDL code actually evaluates the “Next State” logic here. It checks if the current address is the last one (`v_last_addr_processed`).
  - If the address is not the last one ( $< 3$ ), it prepares to increment.
  - If the address is the last one ( $= 3$ ), it prepares to finish.

### The Increment State (INC\_ADDR)

- **Visual:** The state labeled INC\_ADDR.
- **Operation:** The address register is incremented: `s_addr_reg <= s_addr_reg + 1`.
- **The Loop:**
  - **Path A (`addr < 3`):** The UML shows an arrow looping back to READ. This creates the cycle to fetch the next memory location.
  - **Path B (`addr = 3`):** The UML shows a transition to DONE. Note: The VHDL logic actually anticipates this in the PROCESS\_DATA state, but the visual flow in the UML represents the logical sequence of “Process → Increment → Check Address”.

### Completion (DONE State)

- **Visual:** The green state labeled DONE.
- **Operation:**
  - The `done` signal goes High (1).
  - The internal address and data registers are cleared/reset to prepare for a new

run.

- **Return:** On the next clock cycle, the system automatically transitions back to IDLE to await the next start command.

### 6.3.3 Timing Behavior (VWF)

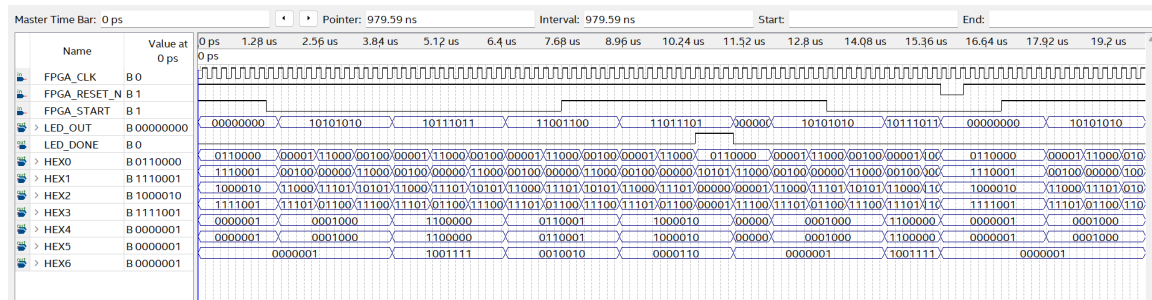


Figure 9: VWF Timing Behavior for Memory-based FSM

## Simulation Overview

This waveform demonstrates the FSM retrieving data from a 4-word RAM block. Unlike the previous memory-less example, the transitions here are governed by a slightly slower step enable rate (`STEP_TICKS = 4` in the generic configuration). This allows the internal states to stabilize for 4 clock cycles before moving to the next logic step, creating wider, more distinct data blocks in the waveform. The FSM iterates through memory addresses 0 to 3, retrieves the values stored there, and outputs them to the LEDs.

## Signal Descriptions

| Signal Name | Value/Pattern                                     | Explanation                                                                                        |
|-------------|---------------------------------------------------|----------------------------------------------------------------------------------------------------|
| FPGA-CLK    | Square Wave                                       | 50 MHz System Clock.                                                                               |
| FPGAS-TART  | High (1)                                          | Input switch held high to enable continuous looping of the memory read sequence.                   |
| LEDOUT      | Data cycles: 00 (Idle) → AA → BB → CC → DD (Data) | Sequence                                                                                           |
| LED-DONE    | Pulse                                             | Goes High (1) only after the DD (Address 3) block is finished processing.                          |
| HEX6        | 0, 1, 2, 3                                        | Address Indicator. Displays the current RAM address being accessed.                                |
| HEX5 / HEX4 | Hex                                               | These display the data split into nibbles (e.g., for 10101010, HEX5 shows 'A' and HEX4 shows 'A'). |

## Sequence of Operation

### Reset and Idle ( $0\mu s$ to $\approx 1.5\mu s$ )

- The simulation begins with `FPGA_RESET_N` low, forcing the system into the **IDLE** state.
- The `LED_OUT` is cleared to 00000000, and the `LED_DONE` signal is 0.
- The HEX displays show “IDLE” on HEX3-0 and the address indicator HEX6 shows ‘0’ (0000001).

### The Memory Loop (Active Processing)

- Once `FPGA_START` goes high, the FSM enters the main loop.
- It cycles through the internal states **READ** → **PROCESS** → **INC\_ADDR** for each memory location.
- You can see four distinct data plateaus on the `LED_OUT` bus:

- **Address 0 (Logic AA):** The first valid output is binary 10101010. This corresponds to Hex AA (Decimal 170). HEX6 remains at 0.
- **Address 1 (Logic BB):** The output changes to 10111011. This is Hex BB (Decimal 187). HEX6 updates to 1 (1001111).
- **Address 2 (Logic CC):** The output changes to 11001100. This is Hex CC (Decimal 204). HEX6 updates to 2 (0010010).
- **Address 3 (Logic DD):** The final output is 11011101. This is Hex DD (Decimal 221). HEX6 updates to 3 (0000110).

### Completion and Restart

- After processing Address 3 (the DD block), the FSM detects it has reached the end of the memory (`C_MEM_SIZE - 1`).
- It transitions to the **FSM\_DONE** state.
- **The Done Flag:** Notice the `LED_DONE` signal pulses High (1) briefly at the end of the DD data block.
- **The Restart:** Because `FPGA_START` is still held High in the simulation, the system briefly goes to **IDLE** (where `LED_OUT` drops to 0) and immediately restarts the sequence at Address 0 (AA), creating the repeating pattern seen in the second half of the waveform.

### 6.3.4 Practical Output



Figure 10: FPGA Implementation of Memory-based FSM - Complete Memory Access Sequence

#### FSM Memory Access Sequence

The system follows the cycle: READ → PROCESS → INCREMENT, iterating through addresses 0 to 3.

#### Phase 1: Initialization

- **State:** IDLE

- **Display:** “0 00 IdLE”
- **Logic:** The system is reset. The Address (HEX6) is 0. The Data (HEX5-4) is cleared to 00. The State (HEX3-0) spells “IdLE”.

**Phase 2: Address 0 (Data = AA)** The cycle begins when the Start switch is flipped.

- **READ State:**
  - **Display:** “0 AA rEAd”
  - **Action:** The FSM reads memory address 0. The RAM returns 10101010 (Hex AA).
- **PROCESS State:**
  - **Display:** “0 AA PrOC”
  - **Action:** The FSM stabilizes the data. The logic checks if this is the last address (It is not).

**Phase 3: Address 1 (Data = BB)** The address has been incremented ( $0 \rightarrow 1$ ).

- **READ State:**
  - **Display:** “1 bb rEAd”
  - **Action:** RAM Address 1 is accessed. Data is 10111011 (Hex BB).
- **PROCESS State:**
  - **Display:** “1 bb PrOC”
  - **Action:** Data processing placeholder.
- **INCREMENT State:**
  - **Display:** “1 bb InCA”
  - **Action:** The FSM enters the INC\_ADDR state. The text spells “InCA” (Increment Address). The internal register `s_addr_reg` is updating from  $1 \rightarrow 2$ .

**Phase 4: Address 2 (Data = CC)** The address is now 2.

- **READ State:**
  - **Display:** “2 CC rEAd”
  - **Action:** Data 11001100 (Hex CC) is retrieved.
- **PROCESS State:**
  - **Display:** “2 CC PrOC”
  - **Action:** Processing CC.



- **INCREMENT State:**

- **Display:** “2 CC InCA”
- **Action:** The display still shows Address 2 and Data CC, but the state text is “InCA”. This indicates the system is currently calculating the next address ( $2 + 1 = 3$ ) to prepare for the final step.

**Phase 5: Address 3 (Data = DD)** The address is now 3 (The final memory location).

- **READ State:**

- **Display:** “3 dd rEAd”
- **Action:** Data 11011101 (Hex DD) is retrieved.

- **PROCESS State:**

- **Display:** “3 dd PrOC”
- **Action:** The system processes the final data packet.
- **Crucial Logic:** In this state, the logic checks `v_last_addr_processed`. Since `s_addr_reg` is 3 (which equals `C_MEM_SIZE - 1`), the `next_state` is set to `FSM_DONE` instead of `INC_ADDR`.

**Phase 6: Completion**

- **State:** `FSM_DONE`
- **Display:** “3 dd dOnE”
- **Visuals:**
  - **Text:** HEX3-0 spells “dOnE”.
  - **Data:** HEX5-4 holds the last value “dd”.
  - **LED:** The Green LED (above the push buttons) is ON. This is the hardware confirmation of the `done <= '1'` signal assignment in the VHDL code.

## 6.4. Conclusion

This project successfully implemented and analyzed two distinct finite state machine architectures: a memory-less FSM and a memory-based FSM. Through comprehensive design, simulation, and FPGA implementation, we demonstrated the fundamental differences between these approaches and their respective applications in computer architecture.

### Comparative Analysis Summary

The memory-less FSM proved to be a straightforward sequential controller with five states (IDLE, STEP1, STEP2, STEP3, `FSM_DONE`) that generated simple 2-bit output patterns

(00, 01, 10, 11). Its simplicity made it ideal for basic control applications requiring minimal hardware resources and predictable timing behavior. The design demonstrated excellent performance for applications where output sequences are fixed and predetermined.

In contrast, the memory-based FSM introduced significant complexity through its integration with a 4×8 RAM module, featuring a five-state sequence (IDLE, READ\_STATE, PROCESS\_DATA, INC\_ADDR, FSM\_DONE) that dynamically retrieved data values (AA, BB, CC, DD) from memory. This architecture showcased the power of combining sequential logic with memory access, enabling more flexible and data-driven operation patterns.

### Key Technical Differences

- **Complexity:** The memory-less FSM required only combinatorial logic and state registers, while the memory-based FSM incorporated address counters, memory interfaces, and data path management.
- **Output Generation:** Memory-less outputs were hardcoded state encodings, whereas memory-based outputs were data-dependent, retrieved from external storage.
- **Observability:** The memory-less design provided direct state visibility through simple LED patterns, while the memory-based implementation required monitoring both address progression and corresponding data outputs across multiple displays.
- **Resource Utilization:** The memory-based FSM consumed significantly more FPGA resources due to the RAM block and additional control logic for memory interfacing.

### Practical Implications

The memory-less FSM represents the foundation of digital control systems, suitable for applications like sequence generators, timing controllers, and simple protocol handlers. Its deterministic behavior and minimal resource requirements make it ideal for embedded systems with strict constraints.

The memory-based FSM, however, bridges the gap between pure control logic and data processing systems. This architecture is essential for applications requiring data storage, retrieval, and processing - such as buffer management, data stream processing, and memory-mapped I/O controllers. The ability to separate control flow from data values provides tremendous flexibility in system design.

## Educational Value

This project effectively demonstrated core computer architecture concepts including state machine design, memory hierarchy, address decoding, and the von Neumann architecture principle of stored-program execution. The hands-on implementation on FPGA hardware provided valuable insights into real-world digital system design constraints and verification methodologies.

Both implementations successfully met their respective specifications, with comprehensive testbench verification confirming correct functionality across all operating conditions. The modular design approach using multiple entities (top-level, FSM core, memory, testbench) proved effective for managing complexity and facilitating code reuse.

The successful completion of this project underscores the importance of finite state machines as fundamental building blocks in digital system design, while highlighting the trade-offs between simplicity and functionality in architectural decisions.