Ho Chi Minh City University of Technology
Faculty of Computer Science and Engineering

Introduction to SoC – Lab Report

Exercises 1–3

Course:	Introduction to SoC
Semester:	251
Student 1:	Nguyen Dinh Khoi - 2352611

Date of submission: November 17, 2025

Contents

Chapter 1

Exercise 1: Traffic Light Control System

1.1 Problem Description

1.2 System Design Requirements

- Use RGB LEDs to simulate red, yellow (blue), and green lights.
- Use 7-segment displays to show the countdown timer for each light.
- Use buttons and switches on the Arty Z7 board to set timing and modes.
- Coordinate two traffic light poles for two intersecting lanes.

1.3 Functional Requirements

- Each pole has three lights: Red, Yellow, Green.
- When one pole is Green/Yellow, the other must be Red.
- Proper sequence: Green $\rightarrow$ Yellow $\rightarrow$ Red for each direction.

1.4 Design Explanation

1. 1 Hz Tick Generator

The FPGA clock runs at 125 MHz, so a 27-bit counter is used to divide the clock and generate a 1 Hz pulse. This "one-second tick" is used as the timing base for changing traffic light phases.

2. Traffic Light State Machine

A four-state FSM controls the behaviour of the intersection:

- `S_MAIN_GREEN`: main road green, side road red.

- `S_MAIN_YELLOW`: main road yellow, side road red.
- `S_SIDE_GREEN`: main road red, side road green.
- `S_SIDE_YELLOW`: main road red, side road yellow.

A 7-bit countdown register `time_left` tracks the remaining seconds for each phase. When the timer reaches zero, the FSM advances to the next state and reloads the appropriate green or yellow duration. The green duration is adjustable using the two switches.

3. RGB LED Control

Two RGB LEDs represent the traffic signals for each road. Each state activates the correct colour:

- Red uses the R channel, Green uses the G channel, and Yellow uses the B channel.

This mapping models a real-world traffic system, ensuring that when one road is green or yellow, the other road stays red.

4. 7-Segment Countdown Display

A 4-digit multiplexed 7-segment display shows the remaining time of the currently active road:

- When the main road is active, digits 3–2 show the timer.
- When the side road is active, digits 1–0 show the timer.

A refresh counter selects which digit to drive, and a BCD-to-7-segment decoder translates the numeric value into segment patterns.

1.5 Simulation Results

1.5.1 Waveform

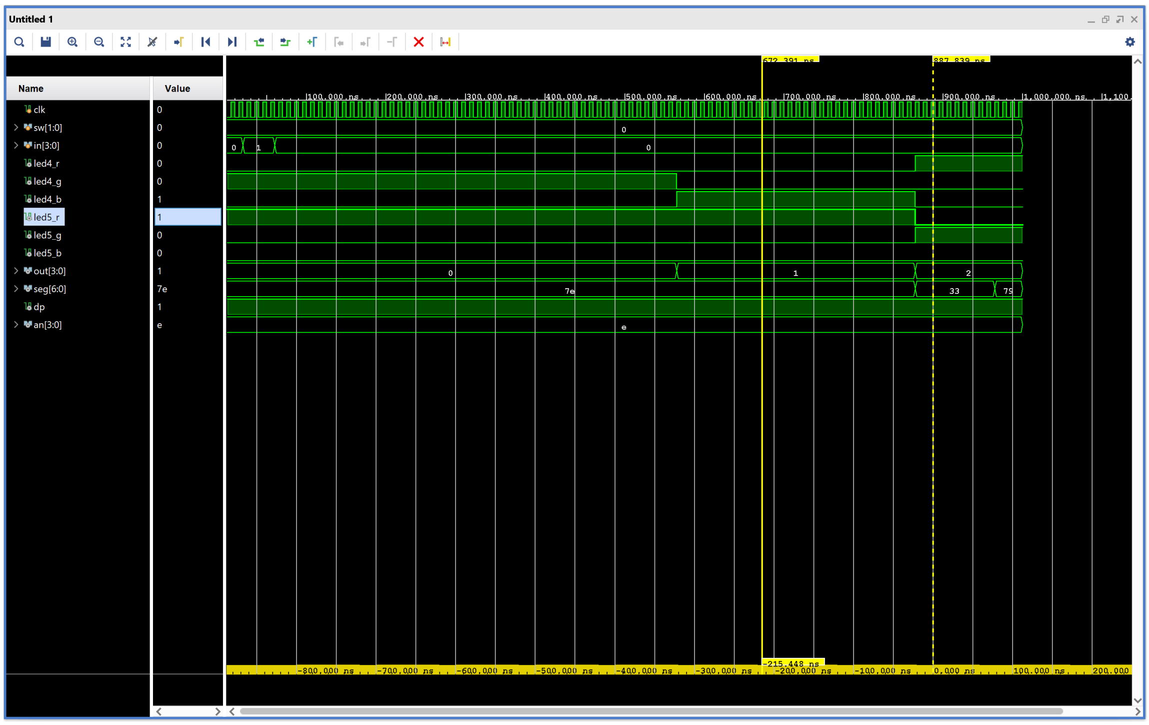


Figure 1.1: Simulation waveform for Exercise 1.

Chapter 2

Exercise 2: Seven-Segment LED Decoration System

2.1 Problem Description

2.2 System Design Requirements

- Use the 7-segment LEDs to display decorative effects.
- Use switches and buttons to modify or adjust system operation.
- Use single-color LEDs as additional indicators (if required).

2.3 Functional Requirements

- Effect 1: Text "CE" scrolls from left to right repeatedly.
- Effect 2: Text "CE" bounces between left and right (shifts left then right).

2.4 Design Explanation

1. 1 Hz Tick Generator

Similar to Exercise 1, the design first divides the 125 MHz system clock down to a 1 Hz pulse. A 32-bit counter counts from 0 up to `SEC_COUNT_MAX = CLK_FREQ - 1` and then generates a single-cycle `one_sec_tick` signal. This tick is used as the time base for moving the "CE" pattern across the digits.

2. Position and Effect Control

The position of the "CE" pattern is stored in a 2-bit register `pos`:

- `pos = 0`: "CE" at digits 3 and 2 (left side),
- `pos = 1`: "CE" at digits 2 and 1,
- `pos = 2`: "CE" at digits 1 and 0 (right side).

A direction bit `dir` is used for the bouncing effect. The switch `sw[0]` selects the display mode:

- **Effect 1 (scroll)**: `sw[0] = 0`, the pattern steps from left to right and wraps around.
- **Effect 2 (bounce)**: `sw[0] = 1`, the pattern moves left–right–left between the edges by changing `dir` when it reaches the leftmost or rightmost position.

On each 1 Hz tick, `pos` and `dir` are updated according to the selected effect. The on-board LEDs `out` are used as indicators: `out[1:0]` show `pos` and `out[2]` reflects the current effect.

3. Character Assignment

The four digits are modelled as an array `digit_char[3:0]`, where each entry encodes one of three possibilities:

- `2'b00`: blank,
- `2'b01`: character "C",
- `2'b10`: character "E".

A combinational block first clears all digits to blank, then places "C" and "E" into the correct two adjacent positions based on the current `pos` value.

4. 7-Segment Multiplexing and Decoding

A separate refresh counter generates a fast 2-bit `refresh_sel` signal, which selects one of the four digits to be driven at a time. Depending on `refresh_sel`, the module:

- Activates the corresponding digit via the active-low `an` lines,
- Selects the encoded character for that digit into `current_digit_code`.

Finally, a combinational decoder maps the character code to the appropriate 7-segment pattern:

- "C": segments `a, f, e, d` on,
- "E": segments `a, f, e, d, g` on,
- Blank: all segments off.

2.5 Simulation Results

2.5.1 Waveform

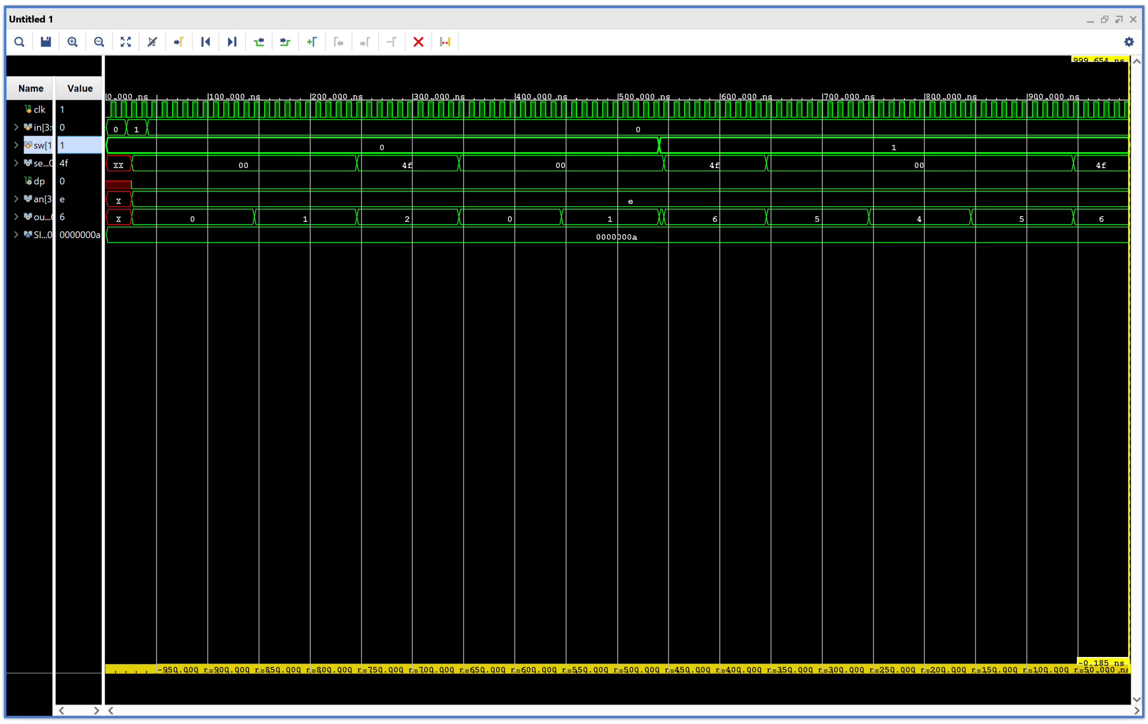


Figure 2.1: Simulation waveform for Exercise 2.

Chapter 3

Exercise 3: Change Mode String Bit LED Circuit

3.1 Problem Description

3.2 System Design Requirements

- Use Verilog to model a state machine that changes the display mode of a 4-bit LED string.
- Use a reset signal to load a default 4-bit pattern (e.g., 0011).
- Use 4 buttons on the board to select modes.

3.3 Functional Requirements

According to the lab description:

- Button 0: **Mode Reset** – show the default 4-bit string on LEDs.
- Button 1: **Mode Circular Shift Left Ring** – shift the 4-bit string left in a ring every 1s.
- Button 2: **Mode Circular Shift Right Ring** – shift the 4-bit string right in a ring every 1s.
- Button 3: **Pause** – pause the current shifting string.

3.4 Design Explanation

1. Modes and LED String

The core of the design is a 4-bit register `led_reg` that holds the current LED pattern. A parameter `DEFAULT_STR` defines the default bit string (e.g. `4'b0011`). The buttons `in[3:0]` control the display mode:

- **BTN0** (`in[0]`): Reset mode — load `DEFAULT_STR`.

- **BTN1** (`in[1]`): Circular shift left every second.
- **BTN2** (`in[2]`): Circular shift right every second.
- **BTN3** (`in[3]`): Pause — hold the current pattern.

A small 2-bit `mode` register encodes these four behaviours. At each rising edge of `clk`, if reset is pressed, the mode is set to `MODE_RESET` and `led_reg` is restored to the default string. Otherwise, button presses update the mode with a simple priority (left, then right, then pause). On each 1 Hz tick, a case statement updates `led_reg` according to the selected mode using circular shifts for left and right. The current pattern is continuously driven to the LED outputs `out[3:0]`.

2. 1 Hz Tick Generator

As in the previous exercises, the design generates a 1 Hz timing signal from the 125 MHz board clock. A counter runs from 0 to `SEC_COUNT_MAX = CLK_FREQ - 1` and then asserts `one_sec_tick` for one clock cycle before restarting. This tick is used both for updating the LED pattern and for scheduling UART transmissions.

3. UART Transmission of LED Pattern

To send the current 4-bit string to the PC, the design uses a separate `uart_tx` module configured with the system clock and a baud rate (e.g. 115200 bps). The `stringbit` module includes a small UART FSM that, every second, when the transmitter is idle, sends five bytes in sequence:

1. ASCII character for bit 3 (`'0'` or `'1'`),
2. ASCII character for bit 2,
3. ASCII character for bit 1,
4. ASCII character for bit 0,
5. A newline character `'\n'`.

A helper function `bit_to_ascii` converts a single bit into its ASCII representation. The FSM only asserts `tx_start` when `tx_busy` is low, ensuring that each byte is launched at the proper time.

4. UART Transmitter Module

The `uart_tx` module implements a simple 8N1 UART transmitter. It calculates a baud-rate divider `BAUD_DIV = CLK_FREQ / BAUD` and uses:

- a baud counter to step at the correct bit rate,
- a 10-bit shift register `shifter` that holds the frame `[stop, data(8), start]`,
- a bit index to count the 10 transmitted bits.

When `start` is asserted and the transmitter is idle, `shifter` is loaded with the start bit (0), the 8 data bits (LSB first), and a stop bit (1). On each baud tick, the least significant bit is driven onto `tx` and the shift register is advanced. After all 10 bits are sent, `busy` is cleared and the line returns to the idle high level.

3.5 Simulation Results

3.5.1 Waveform

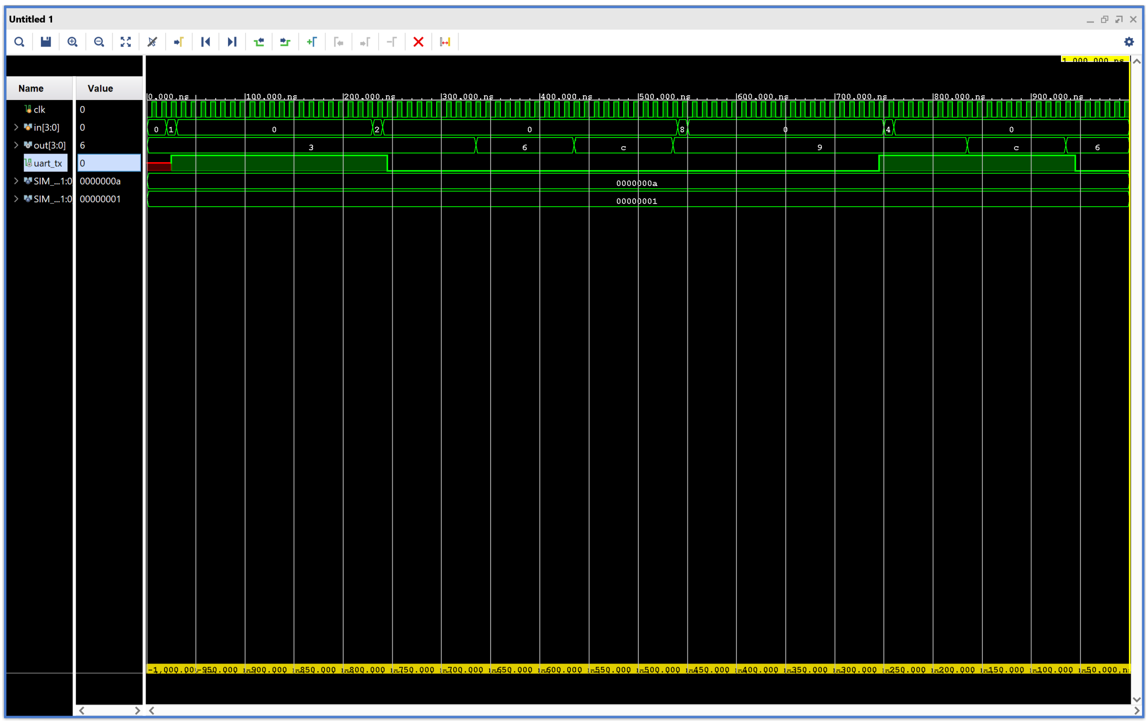


Figure 3.1: Simulation waveform for Exercise 3.

3.5.2 UART

```
Time      buttons(in)      LEDs(out)
0    0000         0011
20000    0001         0011
30000    0000         0011
230000   0010         0011
240000   0000         0011
335000   0000         0110
435000   0000         1100
535000   0000         1001
540000   1000         1001
550000   0000         1001
750000   0100         1001
760000   0000         1001
835000   0000         1100
935000   0000         0110
```

Figure 3.2: Simulation UART for Exercise 3.

Chapter 4

Homework: Divider Unit

4.1 Problem Description

In this homework, we design a key component of the RISC-V processor: the **divider unit**. RISC-V contains four division and remainder instructions:

- `div`, `divu` – signed and unsigned division,
- `rem`, `remu` – signed and unsigned remainder.

In this assignment, we focus on the **unsigned** case. The hardware takes two 32-bit inputs:

- **dividend** (32-bit),
- **divisor** (32-bit),

and produces two 32-bit outputs:

- **quotient**,
- **remainder**.

4.2 Division Algorithm (Software Model)

The hardware is based on the given C algorithm (sequential restoring division), which iterates 32 times to compute quotient and remainder.

The Verilog implementation must compute the same result (quotient and remainder) but using combinational logic only, without using the division operators.

4.3 Corner Case and Restrictions

- **Divide-by-zero**: If the divisor is zero, this special case does not need to be handled in this homework.
- **Verilog restriction**: The use of `/` or `%` operators is not allowed. The design must explicitly implement the algorithm using shifts, comparisons, and subtraction.

4.4 Design Strategy

The implementation is split into two modules:

1. `divu_1iter` – a module that performs *one* iteration of the division algorithm.
2. `divider_unsigned` – a top-level combinational module that instantiates `divu_1iter` 32 times to implement the full unsigned divider.

4.4.1 `divu_1iter` Module

4.4.2 `divider_unsigned` Module

4.5 Source Code

4.5.1 `divu_1iter.v`

```
module divu_1iter (
    input  [31:0] i_dividend_in,
    input  [31:0] i_divisor,
    input  [31:0] i_quotient_in,
    input  [31:0] i_remainder_in,

    output [31:0] o_dividend_out,
    output [31:0] o_quotient_out,
    output [31:0] o_remainder_out
);

    // remainder' = (remainder << 1) | MSB(dividend)
    wire [31:0] rem_shift;
    assign rem_shift = {i_remainder_in[30:0], i_dividend_in[31]};

    // dividend' = dividend << 1
    assign o_dividend_out = {i_dividend_in[30:0], 1'b0};

    // so sánh remainder' vi divisor
    wire rem_lt_div;
    assign rem_lt_div = (rem_shift < i_divisor);

    // quotient' = shift left, thêm bit 0 hoc 1 tùy theo so sánh
    assign o_quotient_out =
        rem_lt_div ?
            (i_quotient_in << 1) :
            ((i_quotient_in << 1) | 32'h1);

    // remainder' = rem_shift hoc rem_shift - divisor
    assign o_remainder_out =
        rem_lt_div ?
            rem_shift :
            (rem_shift - i_divisor);

endmodule
```

4.5.2 divider_unsigned.v

```
module divider_unsigned (
    input  [31:0] i_dividend,     // was: dividend
    input  [31:0] i_divisor,      // was: divisor
    output [31:0] o_quotient,     // was: quotient
    output [31:0] o_remainder     // was: remainder
);

    // Internal stage arrays (0..32)
    wire [31:0] dividend_stage  [0:32];
    wire [31:0] quotient_stage  [0:32];
    wire [31:0] remainder_stage [0:32];

    // Initial values before iteration 0
    assign dividend_stage[0]  = i_dividend;
    assign quotient_stage[0]  = 32'b0;
    assign remainder_stage[0] = 32'b0;

    genvar i;
    generate
        for (i = 0; i < 32; i = i + 1) begin : iter
            divu_1iter u_iter (
                .dividend_in  (dividend_stage[i]),
                .divisor      (i_divisor),
                .quotient_in  (quotient_stage[i]),
                .remainder_in (remainder_stage[i]),

                .dividend_out (dividend_stage[i+1]),
                .quotient_out (quotient_stage[i+1]),
                .remainder_out(remainder_stage[i+1])
            );
        end
    endgenerate

    // Final results after 32 rounds
    assign o_quotient  = quotient_stage[32];
    assign o_remainder = remainder_stage[32];

endmodule
```

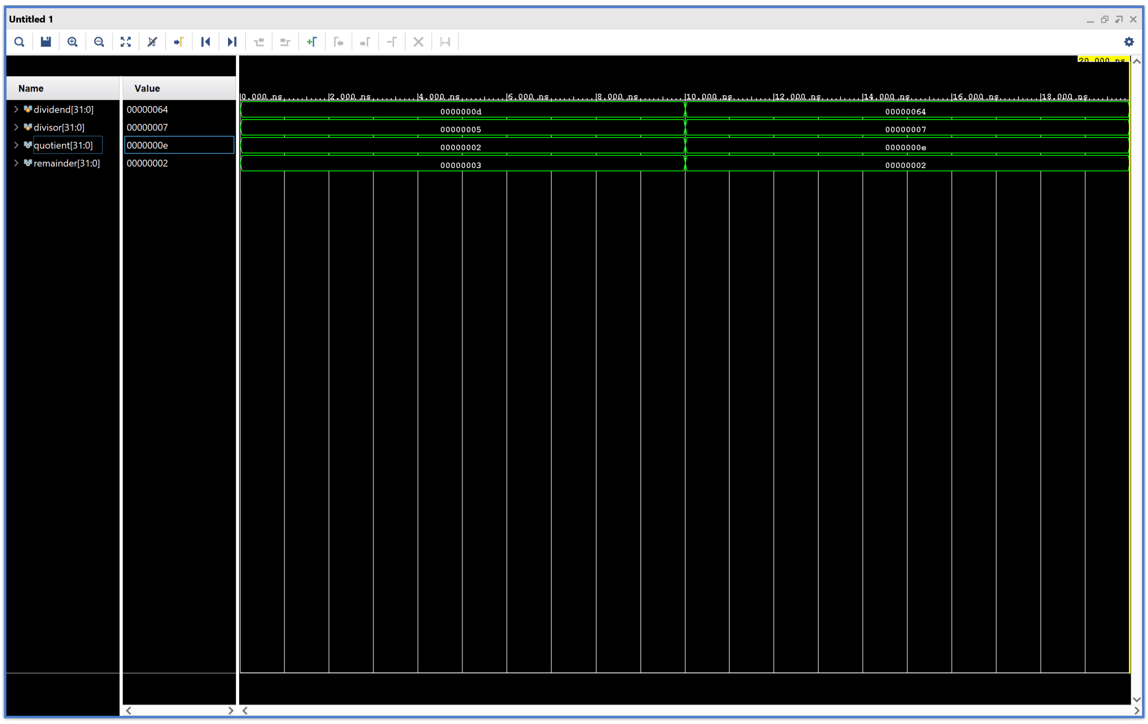
4.6 Simulation and Testing

4.6.1 tb_divider_unsigned.v

```
module tb_divider_unsigned;

    reg  [31:0] dividend, divisor;
    wire [31:0] quotient, remainder;

    divider_unsigned dut (
        .dividend(dividend),
        .divisor (divisor),
        .quotient(quotient),
        .remainder(remainder)
    );

    initial begin
        // ví d 1: 13 / 5
        dividend = 32'd13;
        divisor  = 32'd5;
        #10;   // combinational nên ch cn chút thi gian

        $display("13 / 5: q = %0d, r = %0d", quotient, remainder);

        // ví d 2: 100 / 7
        dividend = 32'd100;
        divisor  = 32'd7;
        #10;
        $display("100 / 7: q = %0d, r = %0d", quotient, remainder);

        $finish;
    end

endmodule
```

Here i generated a test case with (13 / 5) and (100 / 7).

4.6.2 Waveform



Name	Value	0–10 ns	10–20 ns
dividend[31:0]	00000064	0000000d	00000064
divisor[31:0]	00000007	00000005	00000007
quotient[31:0]	0000000e	00000002	0000000e
remainder[31:0]	00000002	00000003	00000002

Figure 4.1: Simulation waveform for the unsigned divider.

4.6.3 UART Testcase

```
source tb_divider_unsigned.tcl
# set curr_wave [current_wave_config]
# if { [string length $curr_wave] == 0 } {
#   if { [llength [get_objects]] > 0} {
#     add_wave /
#     set_property needs_save false [current_wave
#   } else {
#      send_msg_id Add_Wave-1 WARNING "No top le
#   }
# }
# run 1000ns
13 / 5: q = 2, r = 3
100 / 7: q = 14, r = 2
```

Figure 4.2: Simulation UART for the unsigned divider.

Source Code Submission

All source code used in this lab assignment is publicly available at the following GitHub repository:

https://github.com/khoi14412352345/SoCs_lab