Collaborative Scheduling Among Three Elevators

1st Yuanshuo Li
Department of Electrical and Electronic Engineering
Southern University of Science and Technology
Shenzhen, China
12210301@mail.sustech.edu.cn

1st Yulin Liu
Department of Electrical and Electronic Engineering
Southern University of Science and Technology
Shenzhen, China
12211351@mail.sustech.edu.cn

***Abstract*—In this paper, we propose a cooperative multi-elevator control system implemented on the Nexys4 DDR FPGA board using VHDL in the Vivado environment. To achieve efficient elevator scheduling and accurate response to user requests, we design two key software modules: a centralized request handler that assigns external floor requests based on a direction-priority and minimum-distance algorithm, and three elevator control modules that independently manages internal commands and elevator states. Subsequently, we develop a hardware simulation system using external LEDs and push buttons to emulate floor indicators and internal/external call buttons. This ensures that both internal and external requests are processed correctly, and the elevator system behaves in a realistic manner. A VHDL testbench is also constructed to simulate and verify the function of our proposed control module. Finally, the hardware implementation is validated under three typical scenarios: a single external request, multiple concurrent requests, and peak usage conditions. The results demonstrate the robustness and practicality of our FPGA-based elevator control solution.**

***Index Terms*—VHDL, elevator, cooperative system**

I. Introduction

Collaborative elevator systems play a crucial role in the efficiency and responsiveness of modern multi-floor buildings, particularly in commercial or high-density residential environments [1], [2]. Traditional elevator systems often operate independently, handling requests without coordination between multiple elevator units. This lack of collaboration can result in inefficient usage, such as multiple elevators responding to similar floor requests or long waiting times for passengers. Prior studies have pointed out the performance limitations of such decentralized or simple request handling approaches.

In this context, we propose a collaborative elevator scheduling system that enables multiple elevators to operate in a coordinated manner. By analysing the state and position of each elevator in real time, the system dynamically assigns requests to the most suitable elevator, thereby minimizing wait time and travel distance. Our method accounts for both the direction of elevator movement and proximity to the requested floor, improving overall service efficiency.

While many existing implementations rely on software simulations or Verilog-based designs, they often lack real-time hardware deployment or omit practical validation on FPGA platforms [2]. Additionally, many such implementations focus solely on functional correctness, with limited exploration of intelligent dispatching strategies from an algorithmic perspective. To address this gap, our design is implemented in VHDL and targets real-time hardware operation, making it suitable for deployment in embedded elevator control systems.

By combining direction-aware logic, distance-based assignment, and persistent request tracking, our system demonstrates how a hardware-level intelligent elevator dispatcher can significantly improve elevator coordination and responsiveness.

II. Methodology

A. Request Handler Module

In this section, we will detail in the request handler module. This module presents a robust and direction-aware elevator coordination scheme that handles request persistence, visual feedback through LEDs, and smart elevator assignment.

1) External Requests Response: A VHDL-based elevator request handling module designed to coordinate three elevators across six floors. The module accepts external floor requests via up and down buttons and allocates them efficiently to elevators based on their current position and movement direction. Each elevator provides status signals indicating whether it is idle, moving up, or moving down, and its current floor is encoded as a one-hot 6-bit vector.

The system captures external requests using *external_up_request* and *external_down_request* signals, which are derived directly from the debounced input button states. These requests are then latched into *total_up_request* and *total_down_request*, which function as persistent registers ensuring that requests are not lost between clock cycles. The corresponding floor LEDs are driven directly by these registers, illuminating when a request is active.

2) External Requests Assignment: To achieve this objective, three additional functions have been implemented.

```
function is_up_valid(status : std_logic_vector
    (1 downto 0); pos, target : integer)
    return boolean is
begin
```

```
    return (status = "01" and target > pos);
end function;
```

Listing 1. Up elevator detection funcion

This function is used to determine whether a lift is eligible to respond to an upward floor request. It takes the current movement status of the lift, its current floor position, and the target floor as inputs. The function returns true only when the lift is moving upward and the target floor is located above the current position. This condition ensures that the lift only considers upward requests that align with its current direction of travel, thereby improving efficiency and avoiding unnecessary changes in direction.

```
function is_down_valid(status :
    std_logic_vector(1 downto 0); pos, target
    : integer) return boolean is
begin
    return (status = "10" and target < pos);
end function;
```

Listing 2. Down elevator detection funcion

This function checks whether the lift can respond to a downward request. It returns ‘true‘ if the lift is currently moving downward (status = ‘”10”‘) and the target floor is below the current position. This helps ensure the lift only handles requests in its current direction.

```
function run_distance(lift_floor :
    std_logic_vector(6 downto 1); req_floor :
    integer; lift_status : std_logic_vector(1
    downto 0)) return integer is
variable lift_pos : integer := 0;
begin
for i in 1 to 6 loop
    if lift_floor(i) = '1' then lift_pos := i;
        end if;
end loop;
if lift_status = "01" then
    if req_floor >= lift_pos then
        return req_floor - lift_pos;
    else
        return (6 - lift_pos) + (6 - req_floor
            );
    end if;
elsif lift_status = "10" then
    if req_floor <= lift_pos then
        return lift_pos - req_floor;
    else
        return (lift_pos - 1) + (req_floor -
            1);
    end if;
else
    return abs(lift_pos - req_floor);
end if;
end function;
```

Listing 3. Running distance caculator.

This function calculates the distance a lift needs to travel to reach a requested floor, based on its current position and movement direction. The lift’s position is determined by scanning the *lift_floor* input, a 6-bit vector where a ”1” indicates the current floor. Depending on the *lift_status*, the function uses different logic to compute the distance. If the lift is moving up and the requested floor is above or equal to its current floor, the distance is the direct difference. If the request is below, the lift wraps around to the top and comes back down. Similarly, when moving down, the function checks whether the request is below or above and calculates accordingly. If the lift is idle, it simply returns the absolute floor difference. This function helps evaluate how far a lift would need to travel to handle a request, which is useful for scheduling decisions.

To assign a floor request to the most appropriate elevator, the design uses a selection algorithm based on direction validation and moving distance measurement. Two helper functions, *is_up_valid* and *is_down_valid*, determine if an elevator can logically respond to a request in its current direction of travel. If an elevator is deemed valid, its distance to the request floor is calculated using the *run_distance* function. This function takes into account the current position and direction of the elevator to estimate the actual number of floors required to reach the request.

During each evaluation cycle, the system iterates over all active requests. For each one, it calculates the distance from each elevator and assigns the request to the elevator with the shortest valid route. If no elevator is valid for the request direction, the system defaults to the nearest available elevator regardless of direction. The chosen elevator’s internal request vector (*up_next* or *down_next*) is updated accordingly. Finally, these request vectors are transferred to the output ports on the rising edge of the clock, signalling the elevator controllers to stop at the requested floors. Though the code includes logic for clearing requests when a response is received, this part is currently commented out. This means once a request is set, it remains active unless manually reset or extended with future clearing logic.

B. Lift Module

For three elevators, they share the same $Request_Handler$ but operate respectively based on the $lift_control$ module. Generally, this module will first receive internal requests and external requests given by $Request_Handler$. After this, base on the **Dual-State-controlling-Algorithm**, the elevator will automatically switch its direction and state to meet all the needs. Finally, the elevator will response to the $Request_Handler$ if an external request is accomplished.

To ensure that the system can respond to external changes promptly and correctly update and display its state, the elevator control module is divided into three parts: sequential logic, combinational logic, and output logic.

1) I/O settings:

- INPUT: up requests, down requests, internal requests from passengers.

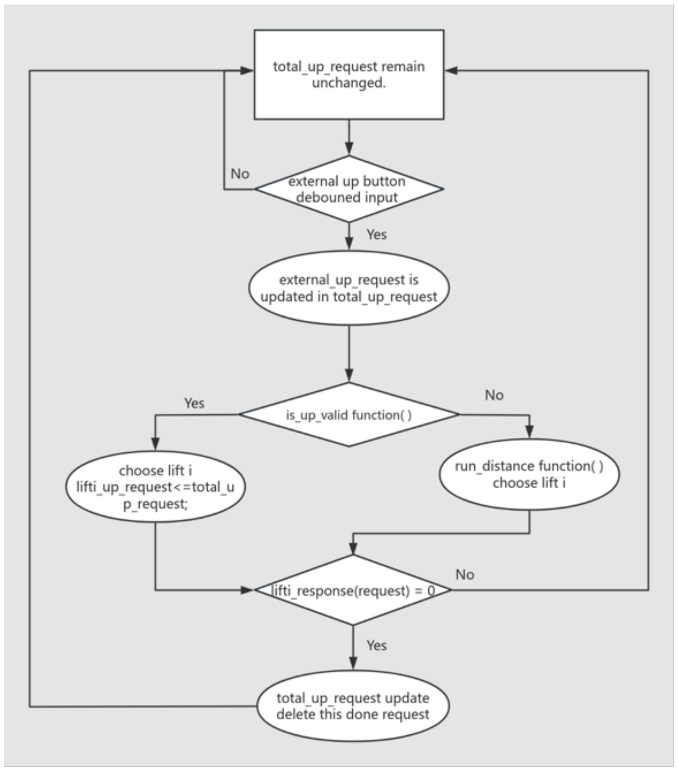

total_up_request remain unchanged.
No
external up button debouned input
Yes
external_up_request is updated in total_up_request
Yes
is_up_valid function( )
No
choose lift i lifti_up_request<=total_up_request;
run_distance function( ) choose lift i
No
lifti_response(request) = 0
Yes
total_up_request update delete this done request

Fig. 1. A flow chart shows the process of external request response and elevator assignment. (Not a ASM chart)

- OUTPUT: completed up requests, completed down requests, LEDs for requests and state visualization, current floor and direction

IN	OUT
`up_req` (std_logic_vector)	`up_resp` (std_logic_vector)
`down_req` (std_logic_vector)	`down_resp` (std_logic_vector)
`internal_req` (std_logic_vector)	`dir` (enumerate)
`clk` (std_logic)	`floor_num` (unsigned)
`reset` (std_logic)	`state_led` (std_logic)

TABLE I
LIFT_CONTROL IO SETTINGS

2) Signal Define: In the sequential logic, the primary operations involve updating registers; therefore, multiple registers are defined as `signals`.

```
type state_type is (IDLE, UP, DOWN, DOOR_OP);
type dir_type is (idle, up, down);
signal state_reg: state_type:= IDLE;
signal dir_reg: dir_type;
signal up_req_list: std_logic_vector(5 downto
   0):= "000000";
signal down_req_list: std_logic_vector(5
    downto 0):= "000000";
signal up_stay_list: std_logic_vector(5
    downto 0):= "000000";
signal up_stay_list_reg: std_logic_vector(5
    downto 0):= "000000";
signal down_stay_list: std_logic_vector(5
    downto 0):= "000000";
signal down_stay_list_reg: std_logic_vector(5
     downto 0):= "000000";
signal internal_req_list: std_logic_vector(5
    downto 0):= "000000";
signal union: std_logic_vector(5 downto 0):=
    "000000";
signal union_reg: std_logic_vector(5 downto
    0):= "000000";
signal floor_num_reg: unsigned(2 downto 0):="
    000";
signal state_led_reg: std_logic:='1';
-- 计时用
signal en_4s_reg: std_logic:='0';
```

Listing 4. Code for Some Signal Define

3) Self-define Functions: Since the elevator needs to simultaneously handle both internal and external requests, and external requests can be categorized into upward and downward directions, separate lists are maintained to record requests according to their direction. The flowing function reads in the current floor and differernt requests. The system identifies the floors where the elevator needs to stop during upward and downward movements by traversing the corresponding request lists.

```
-- 生成 up_stay_list
function generate_up_stay_list(
    floor_num     : integer range 0 to 5;
    up_req        : std_logic_vector(5 downto
        0);
    internal_req  : std_logic_vector(5 downto
        0)
) return std_logic_vector is
    variable result : std_logic_vector(5
        downto 0) := (others => '0');
begin
    for i in 0 to 5 loop
        if i > floor_num then
            result(i) := up_req(i) or
                internal_req(i);
        elsif i = floor_num then
            result(i) := '0';
        else
            result(i) := '0';  -- 低楼层不处理
        end if;
    end loop;
    return result;
end function;

 -- 生成 down_stay_list
function generate_down_stay_list(
    floor_num     : integer range 0 to 5;
    down_req      : std_logic_vector(5 downto
        0);
    internal_req  : std_logic_vector(5 downto
        0)
) return std_logic_vector is
    variable result : std_logic_vector(5
        downto 0) := (others => '0');
begin
    for i in 0 to 5 loop
        if i < floor_num then
            result(i) := down_req(i) or
                internal_req(i);
        elsif i = floor_num then
```

```
                result(i) := '0';
            else
                result(i) := '0';
            end if;
        end loop;
        return result;
    end function;
```

Listing 5. Code for generating up_stay_list and down_stay_list

To determine the update of the elevator state and the direction state, the current floor is examined against the request union list to identify the presence of any higher or lower floor requests. This is achieved via a function that traverses the list and returns a logical value of 1 or 0.

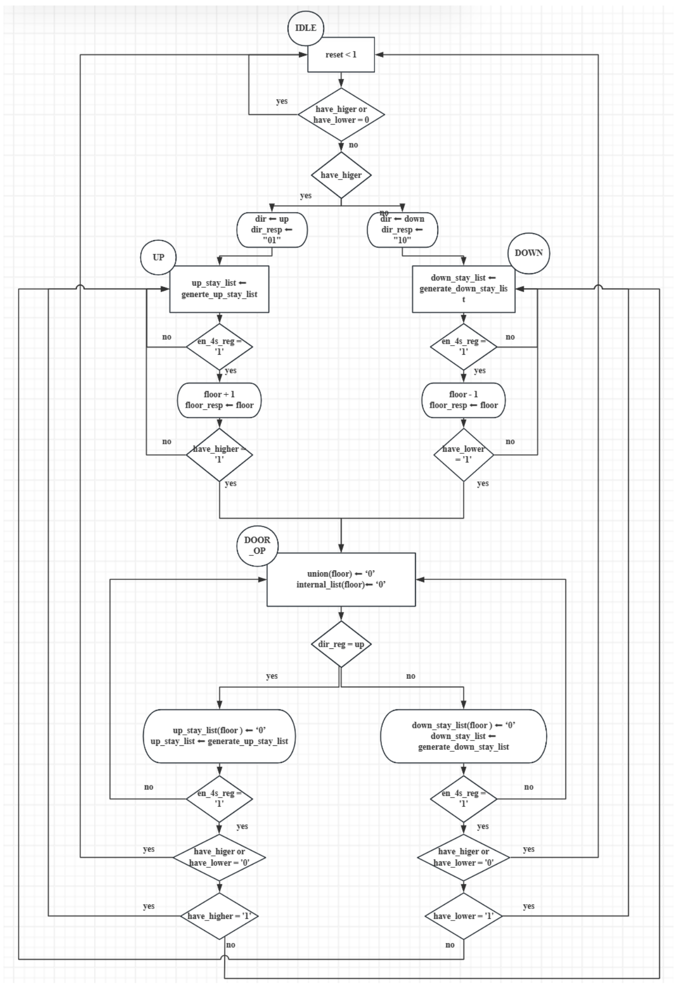
```
function have_higher_req(
    current_floor : integer;
    union         : std_logic_vector
) return std_logic is
    variable result : std_logic := '0';
begin
    for i in union'range loop
        if i > current_floor and union(i) =
            '1' then
            result := '1';
            exit;
        end if;
    end loop;
    return result;
end function;

function have_lower_req(
    current_floor : integer;
    union         : std_logic_vector
) return std_logic is
begin
    -- Boundary check
    if current_floor = 0 then
        return '0';
    end if;
    -- Check lower floors
    for i in union'range loop
        if i < current_floor and union(i) =
            '1' then
            return '1';
        end if;
    end loop;
    return '0';
end function;
```

Listing 6. Code for verifying requests

4) Dual-State-Controlling-Algorithm: The control system of a single elevator can be modeled as a finite state machine with four states: `IDLE`, `UP`, `DOWN`, and `DOOR_OP`. Transitions between these states depend not only on the input and the current state of the elevator, but also on the elevator's direction of movement (denoted as `dir_state`).

The core of the algorithm lies in generating the `up_stay_list` and `down_stay_list` based on the current floor number. In the `UP` state, the elevator prioritizes stopping at floors indicated by the `up_stay_list`, and only considers the `down_stay_list` thereafter. Notably, the direction state is updated exclusively in the `IDLE` and `DOOR_OP` states to ensure that internal and external requests in the same direction are serviced with higher priority.



IDLE
reset < 1
yes
have_higer or have_lower = 0
no
have_higer
yes
no
dir ← up dir_resp ← "01"
dir ← down dir_resp ← "10"
UP
DOWN
up_stay_list ← generte_up_stay_list
down_stay_list ← generate_down_stay_lis t
en_4s_reg = '1'
no
yes
floor + 1 floor_resp ← floor
floor - 1 floor_resp ← floor
have_higher = '1'
have_lower = '1'
DOOR _OP
union(floor) ← '0' internal_list(floor)← '0'
dir_reg = up
up_stay_list(floor) ← '0' up_stay_list ← generate_up_stay_list
down_stay_list(floor) ← '0' down_stay_list ← generate_down_stay_list
have_higer or have_lower = '0'
have_higher = '1'
have_lower = '1'

Fig. 2. ASM Chart of Lift Module

A detailed description of the system's behavior in each operational state is presented below, and most of the functionalities are implemented through user-defined functions mentioned above.

1) IDLE: The direction and elevator state are updated based on the presence of higher or lower floor requests(have_higher_req function), as determined by evaluating the union of all request lists.
2) UP: First, the `up_stay_list` is generated, and the elevator state is updated based on both the upward and downward stay lists. After simulating four seconds of operation, the variable `floor_num` is incremented by one, and the current floor is returned to the `Request_Handler`.
3) DOWN: Same as UP state, but the considering priority of `up_stay_list` and `down_stay_list` switches.
4) DOOR_OP: Based on the current floor, the corresponding positions in the registers are set to zero, and the information is returned to the

`Request_Handler`. After simulating the door opening and closing for four seconds, both the elevator state and direction state are updated.

C. Top Level

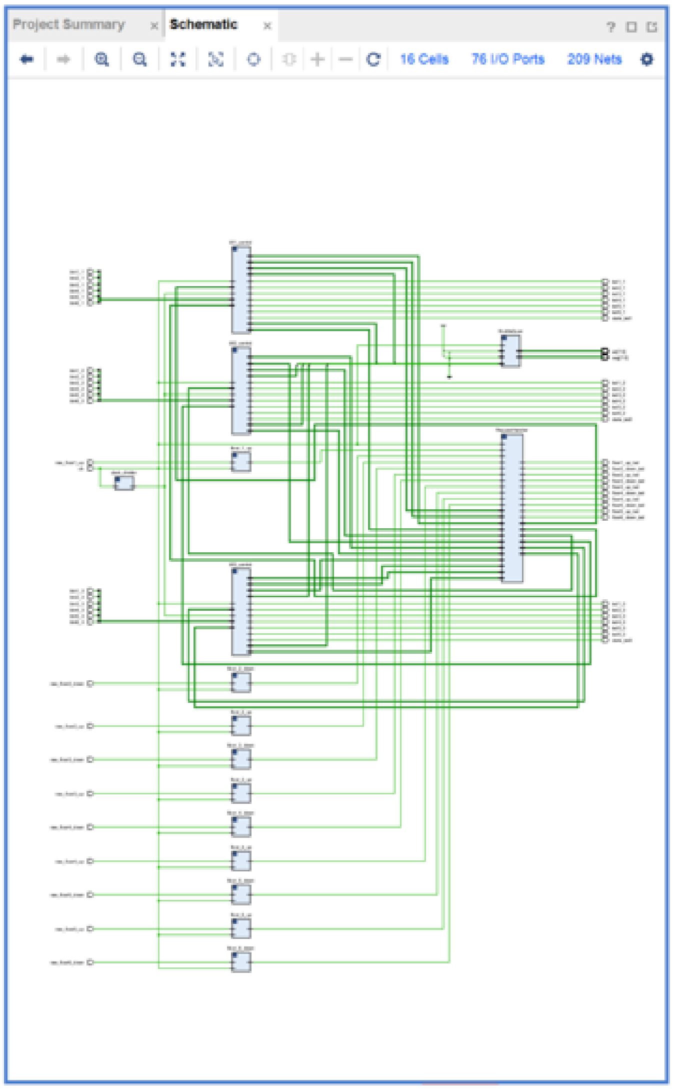


Fig. 3. Schematic

The top level module integrates all submodules to implement a fully functional multi-elevator control system on FPGA hardware. It manages external call buttons, internal requests, elevator control logic, LED indicators, and seven-segment display output. The system simulates a six-floor environment with three independently controlled elevators.

At its core, the design includes a central RequestHandler module that receives debounced floor call signals (up and down) from each floor, as well as real-time position and direction feedback from each elevator. It computes and assigns each valid request to the most appropriate elevator, based on direction alignment and proximity, and outputs up/down request vectors to each individual elevator module (lift1, lift2, lift3).

Each elevator module receives both internal and external requests and handles them independently with timing controlled by a shared 3Hz clock enable signal from the ClockDivider. Internal requests are captured via pushbuttons, debounced, and registered, while the elevator state (moving up, down, idle) and current floor position are output for system coordination. Each elevator also controls its own set of internal LEDs to indicate active requests and current state.

The top level module further handles the display of elevator positions and states using a multiplexed seven-segment display driven by the SevenSegDisplay module. The display dynamically cycles through each elevator' s floor number and state, using custom segment encodings to represent idle, moving up, and moving down.

To enhance reliability, all physical button inputs go through dedicated debounce modules (ButtonDebounce) before entering the control logic. This ensures accurate response to mechanical pushbuttons, preventing false triggering or repeated detections.

D. Hardware Implementation

Our elevator system includes buttons for both internal and external requests(up and down), request indicator LEDs (for both internal and external calls). Also there's a seven-segment display for showing the current floor and elevator status.

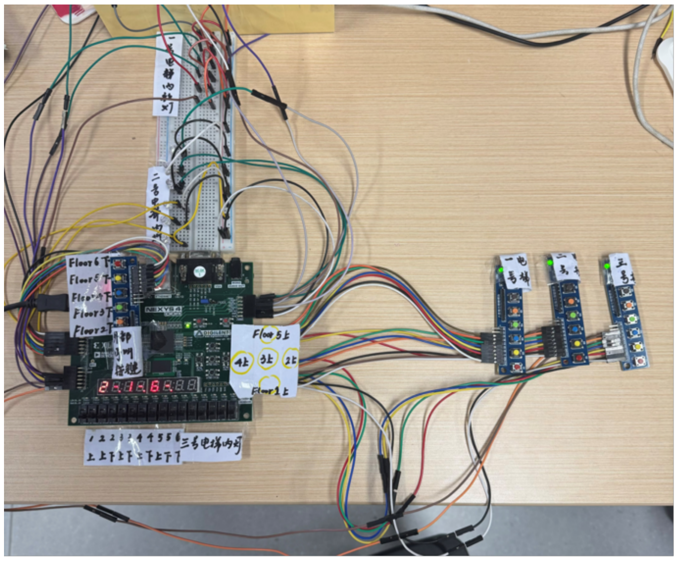


Fig. 4. General Hardware Implementation

III. Results and Analysis

A. Testbench Verification

Since our design includes a large number of button input signals and LED indicator signals, we decided to conduct the test using only a single elevator. This approach allows

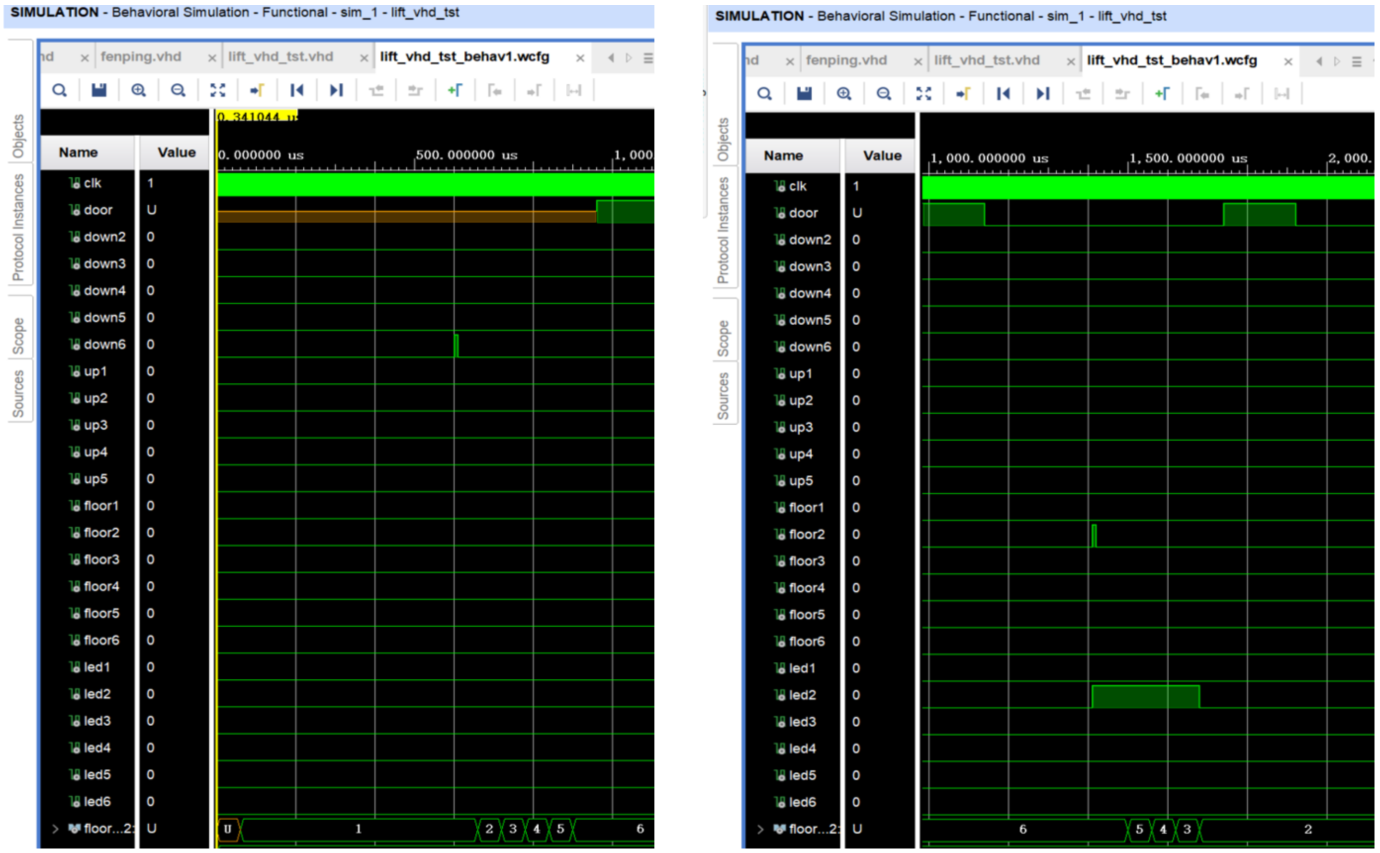


Fig. 5. Testbench result.

us to focus on validating each core functionality—such as floor request handling, direction control, door logic, and LED indication—more clearly and efficiently during the simulation process.

After a downward request was issued from the 6th floor, the elevator began moving upward from the 1st floor to the 6th floor. Upon reaching the 6th floor, the door signal went high, indicating the elevator door opened to allow a passenger to enter. The passenger then selected the 2nd floor as the target. The request light for the 2nd floor remained on during the descent, signalling the active destination. Once the elevator arrived at the 2nd floor, the door signal was set high again to simulate door opening, and the request light turned off, indicating that the service was completed.

B. Hardware Implementation Results

All I/O ports on the FPGA development board are utilized to simulate a six-floor collaborative elevator system. On-board push buttons represent both external floor calls (up/down requests) and internal elevator panel selections across three elevators. Each button press simulates a user request, with corresponding LEDs indicating active calls.

The floor request LEDs illuminate upon request registration, signaling input latching, and turn off once the elevator services the request, accurately reflecting real elevator call indicator behavior.

To demonstrate how the elevator system coordinates scheduling based on real-time conditions and supports user interaction, we present three representative cases, each accompanied by a screenshot from the corresponding demonstration video.

1) *Case1: Single External and Internal Request:* There is an upward request on the 2$^{\text{nd}}$ floor, and the passenger's destination is the 6$^{\text{th}}$ floor. In this case, the LED lights up when the internal request is given, and the elevator shows to go up from the nearest lift. Of course, the LED died out after the lift arrived the 6$^{\text{th}}$ floor.

2) *Case 2: Multiple Internal and External Requests:* Upward requests are made on the 2$^{\text{nd}}$ and 5$^{\text{th}}$ floors, with passengers intending to go to the 4$^{\text{th}}$ and 6$^{\text{th}}$ floors, respectively. A downward request is made on the 4$^{\text{th}}$ floor, with the passenger's destination being the 2$^{\text{nd}}$ floor. In this case, the LED lights up when the external request is given and the first elevator was selected to pick up the first appeard request and the second elevator continued response to the request in the same direction.

3) *Coordination of Three Elevators During Peak Usage:* Complex situation with multiple internal and external requests, both up and down requests appeared in 3,4 and 5 floor at the same time. In this case, the three elevators per-

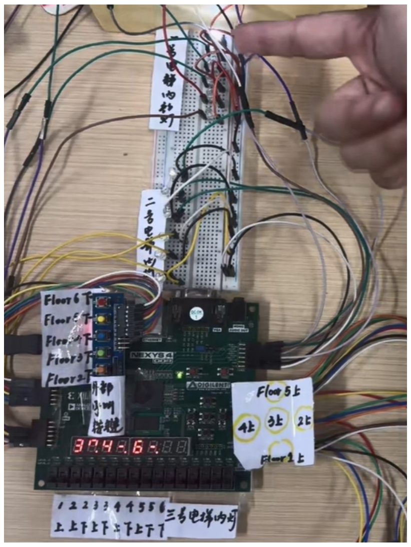


Fig. 6. Single External Request

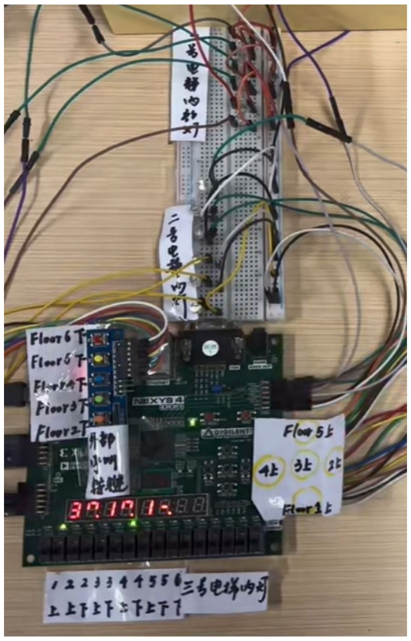


Fig. 7. Multiple Internal and External Requests

fectly coordinate to respond to complex external requests. When a request button is pressed, the corresponding LED lights up, and the LED turns off when the elevator arrives at the requested floor.

IV. Conclusion

We presented a cooperative multi-elevator control system implemented on the Nexys4 DDR FPGA using VHDL. Our design integrates a centralized request handler and independent elevator control modules, enabling efficient scheduling and realistic operation across multiple elevators. The system has been validated through both simulation and hardware testing under various usage

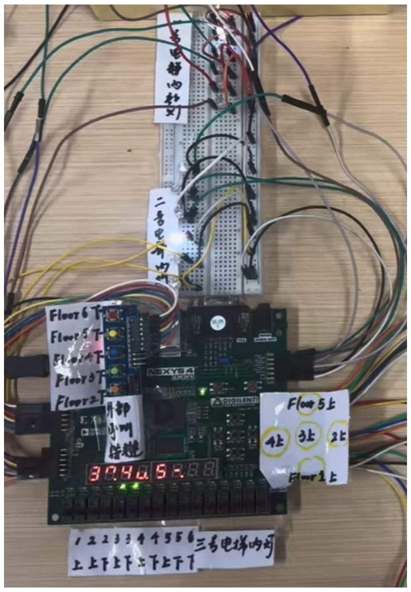


Fig. 8. Coordination of Three Elevators During Peak Usage

scenarios, demonstrating accurate request handling and coordinated elevator behaviour.

The modular architecture of our implementation allows easy scalability. The number of elevators and floors can be increased with minimal changes to the core logic with sufficient I/O ports available. This flexibility makes our design adaptable to more complex building environments and larger-scale elevator systems.

In the future, several enhancements could be explored. Integrating emergency alarm mechanisms and overweight detection systems would greatly improve user safety and fault handling. Features such as graphical interfaces for monitoring could also enhance usability and control. These extensions would bring the system closer to real-world deployment standards and further increase its practical value.

References

[1] T. Yue, "Elevator control chip design," in *2021 International Conference on Computer Information Science and Artificial Intelligence (CISAI)*, 2021, pp. 712–716.

[2] J. Sun, Q. Zhao, and P. Luh, "Optimization of group elevator scheduling with advance information," *IEEE T. Automation Science and Engineering*, vol. 7, pp. 352–363, 04 2010.