CSED311: Lab1 RTL Design

컴퓨터공학과 20220302 김지현

컴퓨터공학과 20220455 윤수인

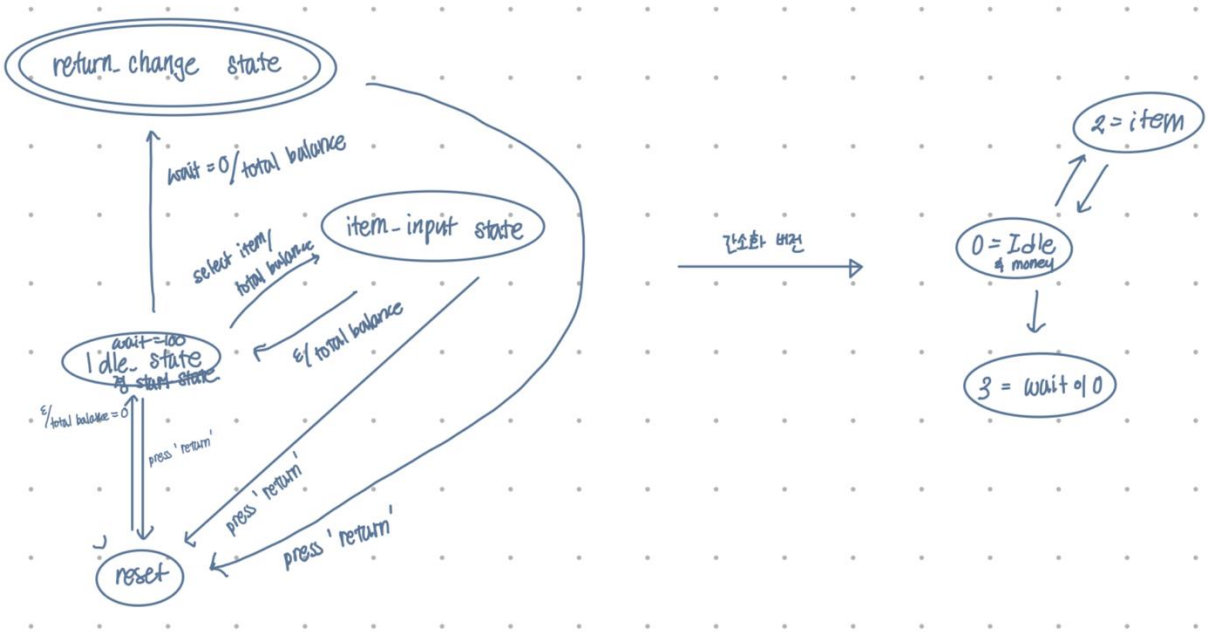
Introduction

- Finite State Machine(FSM)의 정의와 기능, 그 구현에 대해 알아본다.

- Moore Machine, Mealy Machine 의 정의와 형태에 대해 알아본다.

- 공부한 내용을 바탕으로 Combinational Logic 과 Sequential Logic 을 결합하여 'Vending Machine'을 제작해보고, 이를 테스트한다.

Design



<그림 1 : State transition diagram>

구현에 앞서, 현재 Machine 의 상태를 저장하고 Input 과 Output 을 설정하기 위한 State 를 <그림 1>과 같이 설정하였다. 이때, output 이 현재 State 뿐만 아니라 사용자의 i_input_coin 과 i_select_Item 시그널 등에도 영향을 받기 때문에 Mealy Machine 에 해당하는 FSM 으로 구현하였다.

각 변수에 대한 설명

current_total: 위의 그림 1 에서 보여지는 state 들을 나타내는 wire 이다.

current_total_nxt: 다음 state 를 나타내는 레지스터로, 현재 프로그램 내의 여러 값을 토대로 다음 state 를 결정하며, change_state 모듈을 통해 current_total 이 current_total_nxt 로 업데이트된다.

reset_n: 모든 wire, 레지스터의 값을 초기화하는 데 사용되는 신호로, tb 에서 reset_n 의 값이 변경되면 초기화를 수행한다.

i_trigger_return: “Press the return button”에 대한 신호로, 마찬가지로 tb 에서 해당 신호가 들어오면 잔돈을 반환하고 초기화를 수행한다.

i_input_coin: 입력된 코인의 종류(coin_value array 에 따라 순서대로 1000, 500, 100)를 나타낸다.

i_select_item: 입력된 상품의 종류(item_price array 에 따라 각각 가격 2000, 1000, 500, 400)를 나타낸다.

o_available_item: 현재 상태에서 구매 가능한 상품을 비트 1 로 표현하는 시그널에 해당한다.

o_output_item: 현재 구매 가능하며, 사용자가 선택한 상품을 나타내는 시그널에 해당한다. (i_select_item & o_available_item)

input_total: calculate_current_state.v 모듈에서 선언한 레지스터로, 입력받은 코인 액수의 총합에 해당한다.

output_total: calculate_current_state.v 모듈에서 선언한 레지스터로, 구매한 아이템 가격의 총합에 해당한다.

balance: calculate_current_state.v 모듈에서 선언한 레지스터로, 현재 자판기 내에 있는 잔액을 나타낸다.

return_total: vending_machine.v 모듈에서 선언한 레지스터로, 마찬가지로 현재 자판기 내에 있는 잔액을 나타내며, o_return_coin 을 계산할 때 사용된다.

o_return_coin: 잔돈을 앞서 서술한 코인의 종류로 표현하는 시그널에 해당한다.

reg_return_total: return_total 을 업데이트하기 위해 새로 정의 및 할당한 레지스터이다.

next_return_coin: o_return_coin 을 업데이트하기 위해 새로 정의 및 할당한 레지스터이다.

reg_available_item: o_available_item 을 업데이트하기 위해 새로 정의 및 할당한 레지스터이다.

reg_output_item: o_output_item 을 업데이트하기 위해 새로 정의 및 할당한 레지스터이다.

wait_time: 사용자로부터 입력을 받기 위한 대기 시간으로, 초기 값 10 부터 시작하여 1 씩 감소한다.

State 0 은 초기 상태 및 동전 입력에 대해 대기하는 상태이다. 또한 o_available_item 값을 계산한다.

State 1 는 Vending Machine 이 동작하며 상품 선택, 즉 i_select_item 을 받는 상태로, 사용자가 구매하고자 하는 아이템을 선택하면 Available Item, Output Item 을 출력한다. 또한 시간(wait_time)을 10 으로 설정한 후, 해당 시간동안 추가적인 입력이 없거나 Return 입력(i_trigger_return)이 들어온다면 State 2 로 넘어간다.

State 2 는 잔돈 처리에 관련된 state 로, 현재 vending machine 에 남아 있는 돈을 다시 사용자에게 반환해주며, 잔돈이 0 이 될 때까지 반복해서 한 비트씩 돈을 출력(o_return_coin)하고 완료되고, Return 입력이 들어온 경우에만 State 0 으로 다시 돌아간다.

하나의 모듈로 vending machine 의 모든 기능을 구현하기에는 어려움이 있기 때문에, change_state, calculate_current_state, check_time_and_coin 의 세 가지 Submodule 을 통해 각각 기능을 분리하여 구현하였으며, wire 와 reg 를 통해 각각의 submodule 을 연결하였다.

먼저 calculate_current_state 모듈에서는 다음 state 를 결정하며, 현재 상황에서 구매 가능한 아이템(o_available_item) 및 실제 구매한 아이템(o_output_item)의 값을 결정한다. current_total 과 사용자 입력(i_input_coin 또는 i_select_item)에 따라 다음 state 의 역할을 하는 current_total_nxt 를 combinational logic 을 통해 할당한다. 이 과정에서 사용자가 투입한 동전의 금액 합인 input_total 과 사용자가 선택한 아이템의 가격 합인 output_total 을 계산하고, 이를 통해 현재 자판기에 남아 있는 잔액인 balance 를 업데이트한다. 그리고, 이를 return_total 에 할당하여 check_time_and_coin 모듈에서 사용할 수 있도록 하였다. 또한 state 0 인 경우, 현재 잔액에 따라 구매 가능한 아이템(o_available_item)을 출력하고, 사용자가 그 중 구매할 아이템을 선택하면 이를 출력하였다.

다음으로 change_state 모듈에서는, Sequential logic 을 사용하여 state 를 업데이트하였다. reset_n 이 0 인 경우, 즉 초기화를 해야 하는 경우에는 current_total 을 0 으로 설정하였다. 일반적인 상황에서는 앞선 calculate_current_state.v 에서 할당한 current_total_nxt 의 값을 current_total 에 할당하여 state 를 업데이트하였다.

마지막으로 check_time_and_coin 모듈에서는 return_total(현재 vending machine 에 남아 있는 잔액)과 wait_time 의 변화에 대한 전반적인 처리를 수행하였다. always @(posedge clk)를 사용하여, 매 clk 마다 wait_time 을 1 씩 감소시킨다. 만약 reset_n 이 0 인 경우, 초기화를 위해 wait_time 을 10 으로 설정한다.

Return 입력이 들어온 경우, 즉 i_trigger_return 이 1 인 경우에도 wait_time 을 10 으로 다시 설정하고, 다음 state 를 0 으로 설정한다.

Implementation

```
// Combinational logic for the next states
always @(*) begin
  // TODO: current_total_nxt
  // You don't have to worry about concurrent activations in each input vector (or array).
  // Calculate the next current_total state.
  current_total_nxt = 0;

  if(current_total == 0) begin  // idle state and money input
    if(wait_time == 0) begin
      current_total_nxt = 2;
    end

    if(i_select_item != 0) begin
      current_total_nxt = 1;
    end
  end
  else if(current_total == 1) begin // item input
    if(wait_time == 0) begin
      current_total_nxt = 2;
    end

    if(i_input_coin != 0) begin
      current_total_nxt = 0;
    end

  end
  else if(current_total == 2) begin // return change and exit
  end
end
```

<그림 2 : calculate_current_state 모듈에서 다음 state 를 설정하는 Combinational Logic>

```
// Combinational logic for the outputs
always @(*) begin
  // TODO: o_available_item
  // TODO: o_output_item
  case(current_total)
    0,2: begin  // idle_state
      if(balance >= item_price[3]) begin
        reg_available_item = 4'b1111;
      end

      else if(balance >= item_price[2]) begin
        reg_available_item = 4'b0111;
      end

      else if(balance >= item_price[1]) begin
        reg_available_item = 4'b0011;
      end

      else if(balance >= item_price[0]) begin
        reg_available_item = 4'b0001;
      end

      else begin
        reg_available_item = 4'b0000;
      end

      reg_output_item = reg_available_item & i_select_item;
    end

    default: begin
      reg_available_item = 4'b0000;
      reg_output_item = 4'b0000;
    end
  endcase
end
```

<그림 3 : calculate_current_state 모듈에서 item 을 설정하는 Combinational Logic>

```
always @(posedge clk) begin
  if(!reset_n) begin
    input_total <= 0;
    output_total <= 0;
    return_total <= 0;
    balance_nxt <= 0;
  end
  else begin
    if(current_total == 0 && i_input_coin != 0) begin
      for(i = 0; i < `kNumItems; i = i + 1) begin
        if(i_input_coin[i] == 1) begin
          input_total <= input_total + coin_value[i];
          balance_nxt <= balance + coin_value[i];
        end
      end
    end
    else if(i_select_item != 0) begin
      for(i = 0; i < `kNumItems; i = i + 1) begin
        if(o_output_item[i] == 1) begin
          output_total <= output_total + item_price[i];
          balance_nxt <= balance - item_price[i];
        end
      end
    end
    else begin
      return_total <= input_total - output_total;
    end
  end
end


always @(*)begin
  balance = balance_nxt;
end
```

<그림 4 : calculate_current_state 모듈에서 input_total, output_total, return_total 을 설정하는 Sequential Logic>

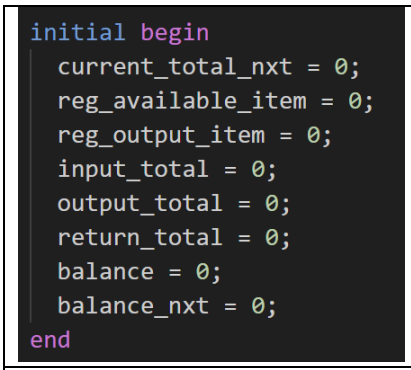
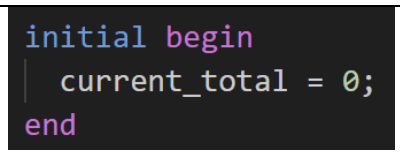
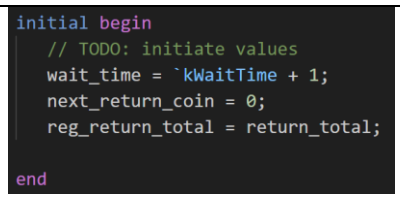
```
always @(posedge clk ) begin
    reg_return_total <= return_total;

    if (!reset_n) begin
    // TODO: reset all states.
        wait_time <= `kWaitTime;
    end
    else begin
    // TODO: update all states.
        if(i_trigger_return) begin // return trigger
            wait_time <= `kWaitTime;
            current_total_nxt_seq <= 0;
        end
        else begin
            wait_time <= wait_time - 1;
        end


        if(reg_return_total >= coin_value[2]) begin
            next_return_coin <= `kNumCoins'b100;
            reg_return_total <= reg_return_total - coin_value[2];
        end
        else if(reg_return_total >= coin_value[1]) begin
            next_return_coin <= `kNumCoins'b010;
            reg_return_total <= reg_return_total - coin_value[1];
        end
        else if(reg_return_total >= coin_value[0]) begin
            next_return_coin <= `kNumCoins'b001;
            reg_return_total <= reg_return_total - coin_value[0];
        end
        else begin
            next_return_coin <= `kNumCoins'b000;
        end
    end
end
```

<그림 5 : check_time_and_coin 모듈에서 wait_time 및 잔돈에 대해 처리하는 Sequential Logic>

```
// Sequential circuit to reset or update the states
always @(posedge clk ) begin
  if (!reset_n) begin
    // TODO: reset all states.
    current_total <= 0; // start_state
  end
  else begin
    // TODO: update all states.
    current_total <= current_total_nxt;
  end
end
```

<그림 6 : change_state 모듈에서 state 를 적용해주는 Sequential Logic>

추가) 각 모듈에서 사용하는 레지스터들을 다음과 같이 initialize 하였다.

		
calculate_current_state.v	change_state.v	check_time_and_coin.v

<표 1 : 각 모듈에서 사용한 레지스터 초기화>

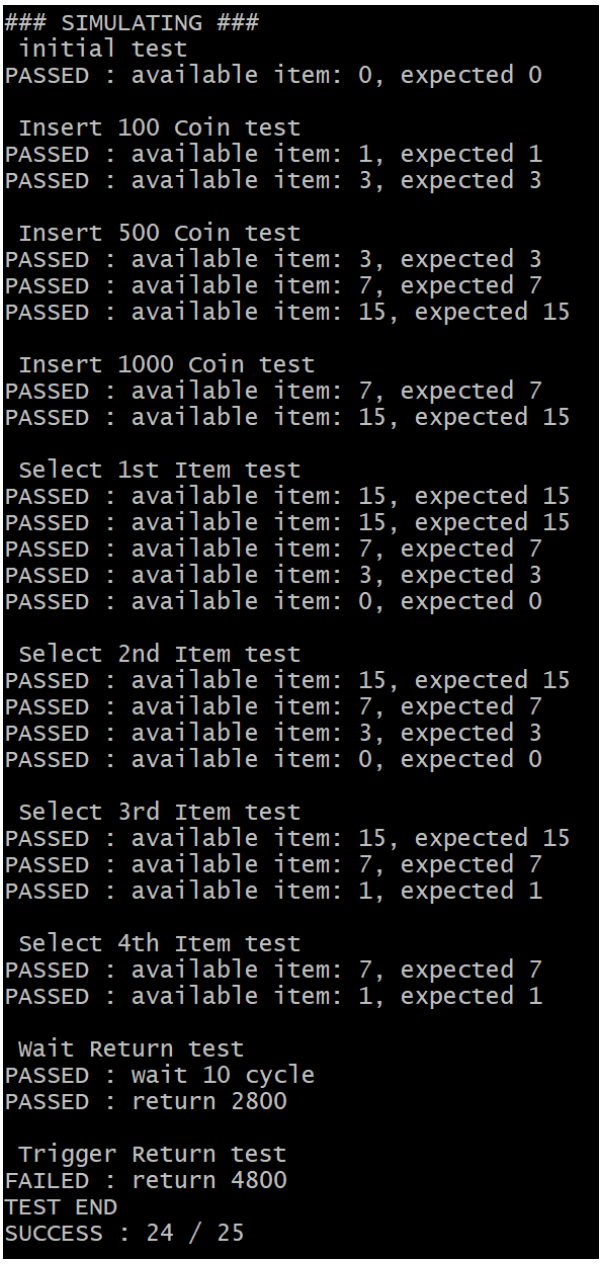
Discussion

Verilog RTL(Register Transfer Level) Design 이기 때문에, Combinational logic 으로 다음 상태를 설정한 후 Register 에 값을 Transfer 하여야 하는데, 과제 초반에는 wire 또는 register 의 다음 값을 나타내는 register 를 따로 선언하지 않고 원래의 값을 직접 업데이트하는 방식으로 구현하여 오류가 많이 발생하였다. 이후 다음 상태를 나타내는 register(ex. reg_return_total)를 사용함으로써 해당 오류를 해결할 수 있었다.

Design 상에서는 Return 입력이 들어왔을 때 i_trigger_return 의 값을 변경하여 wait_time 을 초기화하고 State 0 을 이동할 수 있도록 하였으나, 실제 implementation 에서는 return_coin 의 계산 과정과 몇 clk 씩 차이가 발생하여 Trigger Return Test 를 실패하였다.

Conclusion

Testbench 를 통해 Simulation 해본 결과, 25 개의 Test Case 중 24 개의 case 에 대하여 Passed 를 받았고, 마지막 Trigger Return Test 는 실패하였다.



<그림 7 : Simulation 결과>