CSED311 Lab01

-RTL Design-

Team ID: 2

20180740 염재후

1. Introduction

이번 Lab의 목표는 크게 Combinational Logic, Register-Transfer Level(a.k.a. RTL), Finite State Machine(a.k.a. FSM)을 이해하고, FSM의 하나인 Vending Machine을 구현하는 것이다. 먼저 각 목표에 대해서 알아보도록 하자.

Combinational Logic은 주어진 input 값들의 Boolean function 값을 output으로 취한다. 이 때, Boolean function이 주어진 input값들로만 결과값을 계산하기 때문에 결과값이 Time-independent하다는 특징을 가지고 있다. 대표적인 Combinational Logic으로는 Multiplexer가 있다.

RTL은 Hardware Description Language(a.k.a. HDL)을 사용하여 synchronous circuit을 디자인하는 방법이다. 이 synchronous circuit은 clock signal에 의해서 synchronized되는 register와 Combinational Logic 두 가지로 이루어져 있다.

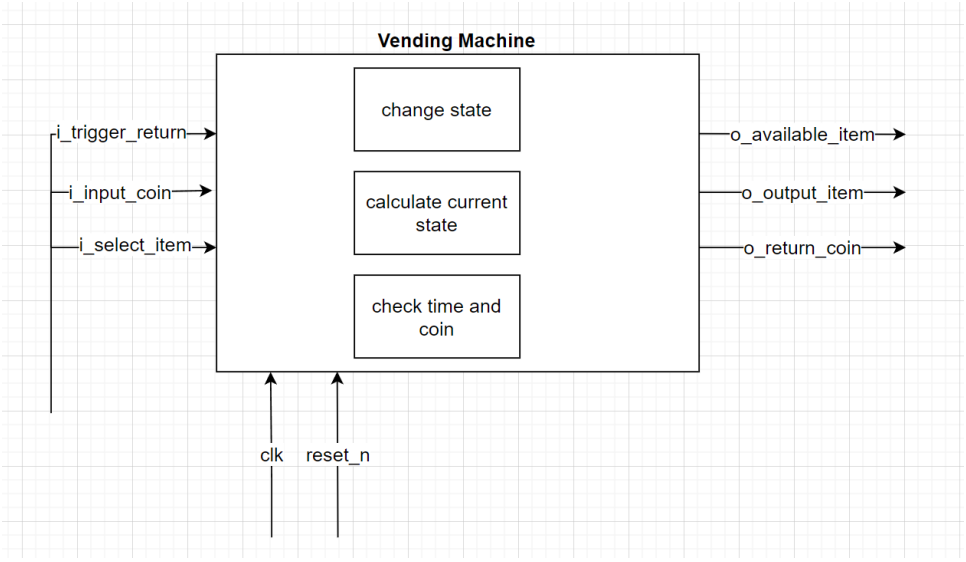
FSM은 유한한 개수의 State를 가지는 Machine으로, Mealy Machine과 Moore Machine이 대표적인 FSM의 예시이다. Moore Machine의 경우, output이 오직 current state에 의해서 결정된다는 특징이 있고, 반면 Mealy Machine의 경우, current state와 current input에 의해서 output이 결정된다. 따라서 Mealy Machine의 경우는 synchronous하게 만들어질 수 있다.

2. Design

이번 랩은 Skeleton code이 주어졌다. Top-level module로는 vending_machine이 있고, 그 내부에 각각 change_state, calculate_current_state, check_time_and_coin 세 개의 모듈이 존재했다.

Input signal은 각 1000원, 500원, 100원의 의미가 있는 i_input_coin, 아이템을 고르는

i_select_item, 잔돈을 반환하는 요청을 하는 i_return_trigger, clk, 초기화 시그널인 reset_n 까지 총 5가지 종류의 input이 있다. Output의 경우는 dispensed item인 o_output_item,



현재 구매가능한 item을 보여주는 o_available_item, 반환되는 잔돈인 o_return_coin 세 종류가 있다. 대략적인 그림은 다음과 같다.

이번에 만들게 되는 Vending machine의 경우, i_select_item의 input값에 의해서 output인 o_output_item값이 결정되기 때문에, mealy machine의 한 종류이다.

그 다음, 세부적으로 각 내부 모듈 디자인을 해보도록 하자. 먼저 change state의 경우, input값으로는 caculate current state에서 계산된 state를 clock의 positive edge에 다음 state로 바꿔주고, reset_n 시그널이 들어올 경우, 즉시 초기화를 해주는 기능을 가진 모듈이다. 따라서, positive edge의 변화를 감지하여, reset_n의 조건을 따진 후, 입력 받은 next state를 넣어줄지, 초기화 시킬지 정하게 될 것이다. 단순히 if-else문을 사용한 구현도 가능하고, mux module을 만든 후, 제작해도 좋을 것이다.

Calculate current state 모듈의 경우, 입출력 값을 사용해 vending machine의 next state를 계산하는 기능과, 동시에 state를 통해 o_available_item과 o_output_item을 계산해주는 기능이 있는 모듈이다. 먼저, vending machine의 next state를 계산하는 방안부터 이야기하고자 한다. vending machine의 다음 state를 계산하기 위해서는 current state와 input coin, output item, return coin 세 가지를 고려해야 한다. 이 때, 각 input coin, output item, return coin이 처리되는 logic이 동일하기 때문에 이를 모듈로 제작한다면, 각 input total, output total, return total로 나누어서 모듈로 처리가 가능할 것이다. 또한 o_available_item을 계산하는 기능 역시, calculate current state module 내부에서 구현하기 보다는, 새로운 모듈로 제작하여, 기능을 대체할 수 있을 것이다.

마지막으로, check time and coin 모듈이 있다. 이 모듈에서 보장해야 하는 기능으로는, 먼저, i_trigger_return 신호가 들어오거나, 대기 시간이 100 unit time이 넘어가면 잔돈을 반환해주는 기능이다. 또 다른 기능으로는 동전이 들어오거나, 아이템이 출력되거나, i_trigger_return 신호가 들어오는 경우, 시간을 다시 리셋해주는 combinational logic이 필요하다. Sequential logic으로는 매 positive edge of clock마다, 시간을 갱신해주고, reset_n 신호가 들어오면 초기화해주는 기능이 필요하다.

3. Implementation

a) Calculate_current_state

Calculate current state module의 변수 및 module instance 선언 부분이다. 해당 모듈 안에서 BitChecker module과 check_available_item module을 사용했다.

```
module calculate_current_state(i_input_coin,i_select_item,item_price,coin_value,current_total,
input_total, output_total, return_total,current_total_nxt,wait_time,o_return_coin,o_available_item,o_output_item);


        input [`kNumCoins-1:0] i_input_coin,o_return_coin;
        input [`kNumItems-1:0]  i_select_item;
        input [31:0] item_price [`kNumItems-1:0];
        input [31:0] coin_value [`kNumCoins-1:0];
        input [`kTotalBits-1:0] current_total;
        input [31:0] wait_time;
        output reg [`kNumItems-1:0] o_available_item,o_output_item;
        output reg  [`kTotalBits-1:0] input_total, output_total, return_total,current_total_nxt;
        integer i;

        wire [`kTotalBits-1:0]wire_input_total;
        wire [`kTotalBits-1:0]wire_output_total;
        wire [`kTotalBits-1:0]wire_return_total;
        wire [`kNumItems-1:0] wire_available_item;


        BitChecker #(.kBitNums(`kNumCoins))Input_Checker(.Bits(i_input_coin), .Values(coin_value), .BitValue(wire_input_total));
        BitChecker #(.kBitNums(`kNumItems))Output_Checker(.Bits(o_output_item), .Values(item_price), .BitValue(wire_output_total));
        BitChecker #(.kBitNums(`kNumCoins))Return_Checker(.Bits(o_return_coin), .Values(coin_value), .BitValue(wire_return_total));

        check_available_item check_available_item_inst(.current_total(current_total), .item_price(item_price), .o_available_item(wire_available_item));
```

다음으로는 logic part이다. 첫 번째 always문 안에서는 Next state를 계산하는 경우, input으로 들어온 current_total과 lower level module인 BitChecker를 통해 각각 value of input coins과 value of output items, value of returning coins를 wire로 받을 수 있었기 때문에 단순한 덧셈 식으로 구현하였다.

두 번째 always 구문은 available item과 output item bit를 계산하는 구문이다. available item 역시 module을 통해 구현하였기 때문에 간단하게 값을 대입하는 것을 통해서 찾을 수 있었고, output item의 경우, for 문을 사용하여, 선택이 되면서, 동시에 available 한 경우에 output이 가능하도록 logic을 설계했다.

```
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 = current_total + wire_input_total - wire_output_total - wire_return_total;
end



// Combinational logic for the outputs
always @(*) begin
        // TODO: o_available_item
        /*
        o_available_item = `kNumItems'd0;
        for (i=0;i<`kNumItems; i=i+1) begin
                if(item_price[i] <= current_total)
                        o_available_item[i] = 1'b1;
        end
        */
        o_available_item = wire_available_item;
        // TODO: o_output_item
        o_output_item = `kNumItems'd0;
        for(i=0; i<`kNumItems; i=i+1)begin
                if(i_select_item[i] && o_available_item[i])
                        o_output_item[i] = 1'b1;
        end

end
```

i) BitChecker Module

BitChecker Module의 경우, 다음과 같이 Bit를 비교하고, i번째 bit가 1인 경우, value list에서 i번째 값을 대입하는 로직이 반복되었기 때문에 다음과 같은 module을 작성할 수 있었다. Item과 coin의 경우, define된 상수의 값이 달랐지만, parameter를 module instantiation할 때, 인자로 넣어줄 수 있었기 때문에 다음과 같은 구조를 짤 수 있었던 것 같다.

```
module BitChecker #(parameter kBitNums=4)(
        Bits,
        Values,
        BitValue
);
        input [kBitNums-1: 0]Bits;
        input [31: 0]Values[kBitNums-1:0];
        output reg [`kTotalBits -1: 0]BitValue;
        integer i;

        always@(*) begin
                BitValue = `kTotalBits'd0;
                for(i=0; i<kBitNums; i=i+1) begin
                        if(Bits[i]) begin
                                BitValue = Values[i];
                                i=kBitNums;
                        end
                end
        end

endmodule
```

ii) Check_available_item module

Check available item module의 경우, 모든 item list중에서 current state, 즉 현재 vending machine에 넣은 돈보다 적은 아이템에 한해서 bit

를 1로 켜주는 역할을 하는 module이다. 이는 for 문을 통해서 구현되었다.

```
module check_available_item(
        current_total,
        item_price,
        o_available_item
);

        input [31:0] item_price [`kNumItems-1:0];
        input [`kTotalBits-1:0] current_total;
        output reg [`kNumItems-1:0] o_available_item;
        integer i;

        always@(*) begin
                o_available_item = `kNumItems'd0;
                for (i=0;i<`kNumItems; i=i+1) begin
                        if(item_price[i] <= current_total)
                                o_available_item[i] = 1'b1;
                end
        end


endmodule
```

b) Change state module

Change state module의 경우, 타 모듈에 비해서 굉장히 간단한 로직을 가지고 있다. 단 하나의 always 구문을 가지고 있으며, 매 positive edge of clock마다, 새로운 state를 갱신해주는 역할 그리고, reset_n 신호가 들어올 경우, 시간을 초기화해주는 역할을 수행한다.

```
module change_state(clk,reset_n,current_total_nxt,current_total);

        input clk;
        input reset_n;
        input [`kTotalBits-1:0] current_total_nxt;
        output reg [`kTotalBits-1:0] current_total;

        // Sequential circuit to reset or update the states
        always @(posedge clk ) begin
                if (!reset_n) begin
                        // TODO: reset all states.
                        current_total <= `kTotalBits'd0;

                end
                else begin
                        // TODO: update all states.
                        current_total <= current_total_nxt;
                end
        end
endmodule
```

c) Check time and coin module

먼저 필요한 변수를 선언해주고, initial 구문 안에서 시간과 return coin을 초기화 해준다.

```
module check_time_and_coin(i_input_coin,i_select_item,coin_value,clk,reset_n,
                        i_trigger_return, o_output_item, wait_time,o_return_coin, current_total);
        input clk;
        input reset_n;
        input [`kNumCoins-1:0] i_input_coin;
        input [`kNumItems-1:0]  i_select_item;
        input [31:0] coin_value [`kNumCoins-1:0];
        input i_trigger_return;
        input [`kTotalBits-1:0] current_total;
        input [`kNumItems-1:0] o_output_item;
        integer i;

        wire [31:0]wire_wait_time;

        output reg  [`kNumCoins-1:0] o_return_coin;
        output reg [31:0] wait_time;


        // initiate values
        initial begin
                // TODO: initiate values
                wait_time = 'd0;
                o_return_coin = `kNumCoins'd0;
        end
```

다음으로는 combinational logic part이다. 첫 번째 always 문에서는 동전이 들어오거나, 아이템이 나오거나, 혹은 잔돈을 반환하는 신호가 들어올 경우, 시간은 초기화된다.

두 번째 always 문에서는 반환 신호가 들어오거나 혹은 대기 시간이 끝난 경우, 현재 vending machine 안에 남아있는 돈을 기반으로 하여, 잔돈을 반환하는 logic이 있다.

마지막 always 문은 positive edge of clock일 때 작동하는데, 매 positive edge마다, 대기 시간을 감소시키거나, 혹은 reset_n 신호가 들어왔을 경우에는 시간을 초기화 시켜주는 역할을 한다.

```
always @(*) begin
        // TODO: update coin return time
        // Check if select is valid
        if(i_input_coin || o_output_item || i_trigger_return)
                wait_time <= 'd0;
end

always @(*) begin
        // TODO: o_return_coin
        o_return_coin = `kNumCoins'b000;
        if(wait_time > `kWaitTime || i_trigger_return) begin
                if(current_total >= coin_value[2])
                        o_return_coin[2] = 1'b1;
                else if(current_total >= coin_value[1])
                        o_return_coin[1] = 1'b1;
                else
                        o_return_coin[0] = 1'b1;
                //else if(current_total >= coin_value[0])
                        //o_return_coin[0] = 1'b1;
                //else
                        //o_return_coin = `kNumCoins'b000;
        end
end


always @(posedge clk ) begin
        if (!reset_n) begin
        // TODO: reset all states.
                wait_time <= 'd0;
        end
        else begin
        // TODO: update all states.
                wait_time <= wait_time + 1;
        end
end
```

4. Discussion

가장 먼저 wiat_time의 경우, pdf에 기술된 대로, 100에서 discounting 되는 방식을 채택하여 구현을 시도했었다. 하지만, 이 경우, wait_time에 0에서 더 내려가는 경우가 발생하였고, 이 경우, always에서 감소된 wait_time을 비교하는 과정이 생략되어, 시간이 제대로 감소되는 효과가 나오지 않았다. 따라서 100에서 0으로 가는 대신 역순으로 0에서 100으로 시간을 더해가는 방향을 채택하게 되었고, 그 결과 정상적으로 구현이 가능했다.

그 다음으로는, BitChecker module이다. 이런 module을 구현하는 경우, parameter를 종종 쓴다고 배웠다. 하지만, module을 정의할 때, 이 parameter의 값을 정해놨음에도 불구하고, module이 instantiation되는 경우에 새롭게 인자를 전달할 수 있다는 것을 이번 랩을 통해서 알 수 있었다.

BitChecker module의 경우, 내가 생각했던대로 바람직하게 modularization이 가능했다. 하지만, check available item module의 경우는 조금 성급하게 modularization이 된 것 같아서 아쉽다. 그 이유는 이 모듈은 available item과 output item 두 가지를 대체하고자 설계한 모듈이었으나, 한계 때문에 available item만을 나타내게 되었기 때문이다. 내가 마주한 한계는 available item을 check하는 과정에서, current state >= item_price[i]라는 논리식이 필요했는데, 이런 논리식을 argument로 넘길 수 없다는 것이었다. 만약 논리식을 값으로 넘겨주는 경우도 생각해봤지만, 이 경우에는 for문의 index i에 의해서 값이 바뀔 수 있기 때문에 그럴 수 없었다. 따라서 아쉽게도 내가 원했던 대로는 할 수 없었다. 하지만, 객체지향언어의 inheritance처럼, basic한 module을 형성한 후에, 그 안에 논리식만을 추가하는 방안을 하는 방안이나, 혹은 그와 비슷한 Verilog 문법을 찾아보고자 한다.

5. Conclusion

이번 랩을 통해서 RTL을 통해서 Combinational Logic과 sequential logic으로 synchronous circuit을 구성하는 방법에 대해서 배울 수 있었다. 또한 두 가지 이상의 module을 만드는 과정을 통해서 Verilog를 사용한 module 설계 방법과 wire의 사용 방법에 대해서 더 자세히 알 수 있었다.