Vending Machine Report

20210115 이세규 20210706 이지원

1. Introduction

이번 장에서는 우리 팀이 구현한 Vending Machine의 동작 방법, 가정 그리고 시스템의 입출력에 대해 설명하겠다.

자판기에는 한 번에 동전 하나를 넣을 수 있고 동전의 종류는 100원, 500원, 1000원으로 총 가지이다. 자판기가 동작한 직후, 제한시간은 100이 되며 일정 시간을 주기로 이 값은 1씩 줄어들고 제한시간이 끝났을 경우 사용자가 넣었던 금액은 모두 반환된다. 동전을 넣으면 제한시간은 다시 100으로 늘어난다. 선택할 수 있는 상품의 종류는 총 4가지로 각각 400원, 500원, 1000원, 2000원이다. 사용자가 넣은 금액으로 구매할 수 있는 경우에만 상품이 표시된다. 사용자가 구매가능한 상품을 선택하였을 경우, 자판기는 해당 상품의 출력을 알려준다. (당시 투입금액 – 선택한 상품의 가격)만큼의 돈은 자판기에 남아 있으며, 제한시간은 다시 100으로 설정된다. 만약 구매할 수 없는 상품을 선택하였다면 아무 일도 일어나지 않는다. 마지막으로, 리턴 버튼을 누르면 현재 자판기에 있는 금액은 모두 반환되고 다시 상품을 선택할 수 있는 상태가 된다. 아이템의 개수와 거스름돈은 무한하다고 가정한다.

최종적으로 구현할 시스템의 입출력은 다음과 같다.

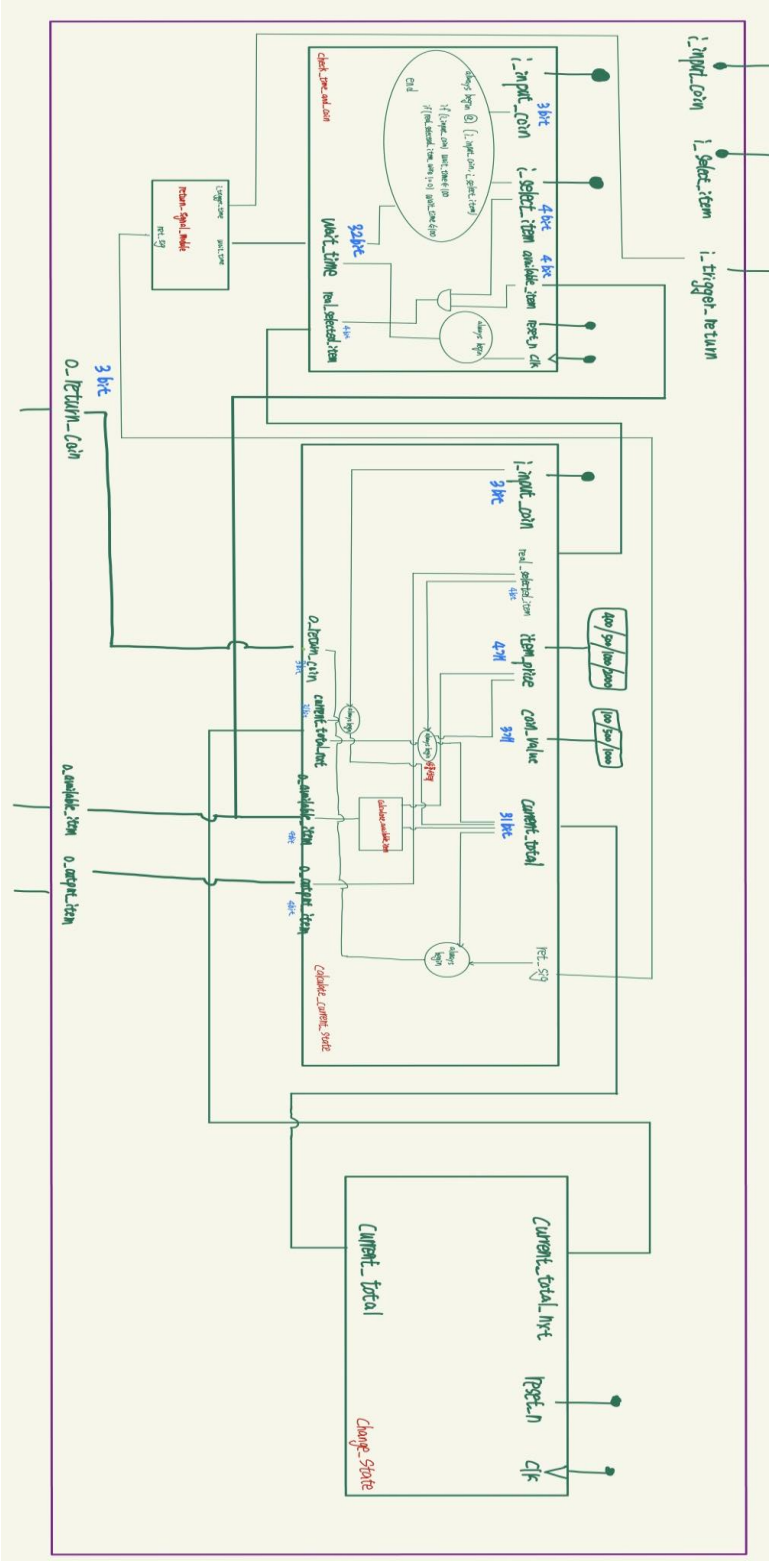
입력: clk(클락), reset_n(register의 초기화를 다루기 위함), i_input_coin(자판기에 넣는 동전 종류), i_select_item(선택된 상품 종류), i_trigger_return(거스름돈 반환 신호)

출력: o_available_item(사용자가 선택할 수 있는 상품 종류), o_output_item(자판기가 제공하는 상품 종류), o_return_coin(반환되는 동전 종류)

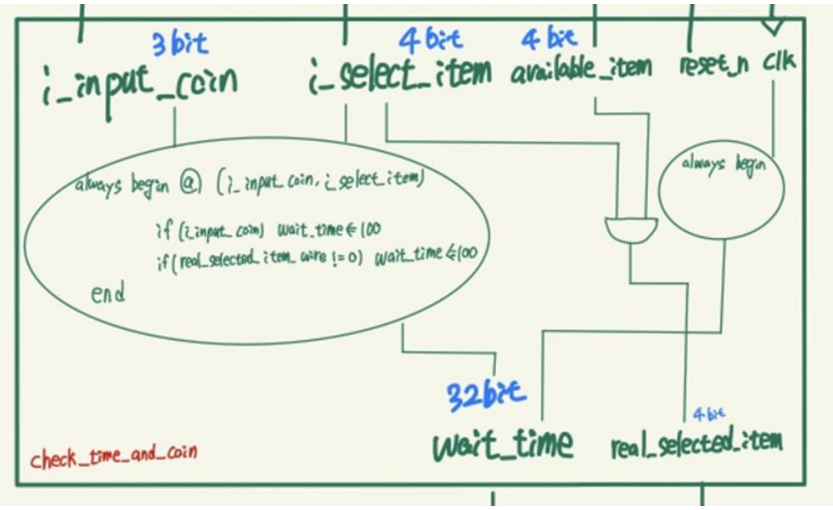
위의 입력과 출력을 이용하여 Vending Mahcine을 설계할 수 있다. 우리 팀은 여러 개의 작은 시스템들을 조합하여 하나의 큰 시스템을 구성하였다.

2. design

스켈레톤 코드에 제공된 모듈은 총 3가지다. 우리 팀은 이 모듈의 입출력을 적절히 변형하여 문제를 해결하였다. 아래에서는 필요한 모듈에 대한 회로도와 모듈 각각에 대한 설명과 모듈 사이의 관계에 대한 설명을 하였다.



1. check_time_and_coin 모듈

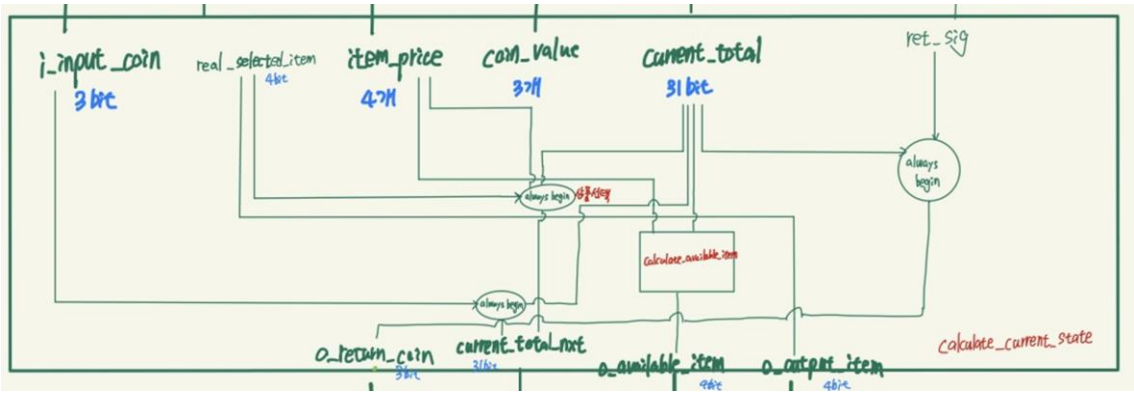


- 클락에 따라 제한시간을 1씩 줄인다.
- 동전이 들어오거나 아이템이 선택되면 제한시간을 100으로 한다.
- 상품에 대한 사용자의 선택의 유효성을 판단한다.

2. return_signal 모듈

- 제한시간 또는 사용자의 반환 신호에 따라 거스름돈의 출력을 결정한다.

3. calculate_current_state 모듈



- check_time_and_coin 모듈에서 처리한 내용을 바탕으로 유효한 상품을 출력한다.
- return_signal_module 모듈에서 처리한 거스름돈 반환 신호에 따라 거스름돈을 출력한다.
- 사용자가 입력한 동전 종류 또는 선택된 상품 종류를 받아 다음 상태의 누적 금액을 계산한다.

4. change_state 모듈

- calculate_current_state에서 계산한 다음 상태의 누적 금액을 바탕으로 다음 상태의 금액을 calculate_current_state로 전달한다.

아래에서는 이러한 기능을 Verilog 를 활용하여 어떻게 구성하였는지 실제 코드를 보며 알아보겠다.

3. implementation

1. check_time_and_coin 모듈

```
assign real_selected_item=i_select_item & o_available_item;
```

real_selected_item은 i_select_item과 o_available_item을 and게이트로 묶어서 구현된다. 사용자가 구매를 원하는 아이템이 현재 판매 가능한 지를 체크하여 실질적으로 물건 판매가 발생하는 지를 보여주는 신호이다.

```
// initiate values
initial begin
    // TODO: initiate values
    wait_time<=100;
end
```

wait_time의 초기값은 100이다.

```
// update coin return time
always @(i_input_coin, i_select_item) begin
    // TODO: update coin return time
    if(i_input_coin!=0) wait_time<=100;
    if(real_selected_item!=0) wait_time<=100;
end
```

coin이나 item에 대해 유의미한 input 신호가 들어오면 시간을 업데이트 해준다. 이때, item의 경우 살 수 없는 물건을 고르는 경우는 제한시간이 초기화되지 않으므로 이를 고려하여 real_selected_item으로 업데이트를 결정한다.

```
always @(posedge clk ) begin
    if (!reset_n) begin
    // TODO: reset all states.
    wait_time<=100;
    end
    else begin
    // TODO: update all states.
      #1 if(wait_time>0) wait_time<=wait_time-1;
    end
end
```

리셋되면 wait_time을 초기값인 100으로 만들어주고, 매초마다 wait_time이 1씩 감소하도록 설정한다.

2. return_signal 모듈

```
module return_signal (
    wait_time,
    i_trigger_return,

    ret_sig
);
    input [31:0] wait_time;
    input i_trigger_return;
    output reg ret_sig;
    always @(*) begin
        if(wait_time==0||i_trigger_return) begin
            ret_sig<=1;
            end
        else begin
            ret_sig<=0;
            end
        end
endmodule
```

return_signal 모듈은 vending_machine.v에 추가로 구현해두었다. input과 output의 구성은 wait_time과 i_trigger_return을 input으로, ret_sig를 output 레지스터로 이루어져 있다. 기능은 간단하게 wait_time이 0이거나(=제한시간이 다 되었거나) i_trigger_return이 1일 때(=사용자가 반환을 요청했을 때) 반환을 요청하는 신호인 ret_sig를 생성하여 calculate_current_state에 보낸다.

3. calculate_current_state 모듈

```
always @(*) begin
    if(current_total >= item_price[3])
        o_available_item <= 4'b1111;
    else if(current_total >= item_price[2])
        o_available_item <= 4'b0111;
    else if(current_total >= item_price[1])
        o_available_item <= 4'b0011;
    else if(current_total >= item_price[0])
        o_available_item <= 4'b0001;
    else o_available_item <= 4'b0000;
end
```

o_available_item을 상시로 업데이트해주는 코드이다. 단순히 현재 가지고 있는 돈인 current_total과 아이템들과의 가격인 item_price와의 비교를 통해 살 수 있는 아이템들의 비트를 1로 설정한다.

```
always @(*) begin//real_selected_item
    case(real_selected_item)
        4'b1000: begin
          current_total_nxt<=current_total-item_price[3];
          o_output_item<=real_selected_item;
          end
        4'b0100: begin
          current_total_nxt<=current_total-item_price[2];
          o_output_item<=real_selected_item;
          end
        4'b0010: begin
          current_total_nxt<=current_total-item_price[1];
          o_output_item<=real_selected_item;
          end
        4'b0001: begin
          current_total_nxt<=current_total-item_price[0];
          o_output_item<=real_selected_item;
          end

        default: begin
          current_total_nxt<=current_total;
          o_output_item<=real_selected_item;
          end
    endcase
end
```

real_selected_item이 유효한 값으로 들어왔을 때에 대한 코드이다. real_selected_item이 평소에 0으로 되어 있는 경우 물건을 구매하지 않는 상황이기에 현재 돈의 다음 상태는 현재 돈 그대로이며, (current_total_nxt <= current_total) 이것은 default에 구현되어 있다. 그 외에 real_selected_item이 유효한 값이 되었을 경우

다음 상태의 돈은 현재 가지고 있는 돈에서 물건 값만큼 빼서 결제가 일어난 돈이 되고, 해당 상품의 출력(o_output_item <= real_selected_item)도 진행한다.

```
always @(*) begin
    case(i_input_coin)
        3'b100: current_total_nxt<=current_total+coin_value[2];
        3'b010: current_total_nxt<=current_total+coin_value[1];
        3'b001: current_total_nxt<=current_total+coin_value[0];
        default: current_total_nxt<=current_total;
    endcase
end
```

코인의 입력이 들어온 경우 단순히 현재 돈에서 코인 종류에 따른 금액만큼을 늘려주는 식으로 코드를 구성하였다.

```
always @(*) begin
    if(ret_sig) begin
      if(current_total>=coin_value[2])begin
          o_return_coin<=3'b100;
          current_total_nxt<=current_total-coin_value[2];
      end
      else if(current_total>=coin_value[1])begin
          o_return_coin<=3'b010;
          current_total_nxt<=current_total-coin_value[1];
      end
      else if(current_total>=coin_value[0])begin
          o_return_coin<=3'b001;
          current_total_nxt<=current_total-coin_value[0];
      end
      else
          o_return_coin<=0;
    end
end
```

ret_sig를 체크하여 만약 1이 되면 현재 금액을 모두 반환하여야 한다. 현재 금액에서 반환할 수 있는 가장 큰 금액의 코인이 무엇인지를 체크하여 o_return_coin을 그것으로 설정하고, 현재 금액에서 반환되는 코인의 금액만큼을 빼서 current_total_nxt에 반영한다. ret_sig가 1인 동안에는 이 코드가 반복적으로 진행되어 결국 현재 금액이 0이 될 때까지 진행할 것이다.

4. change_state 모듈

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

change_state의 경우 정말 간단하다. 리셋이 될 때는 current_total을 0으로 초기화해주고, 평소에는 클락에 따라 calculate_current_state에서 계산한 다음 상태의 누적 금액을 현재 상태의 금액으로 인식하여 상태를 변화하는 과정을 거치고, 이를 다시 calculate_current_state 모듈에 전달한다.

4. discussion

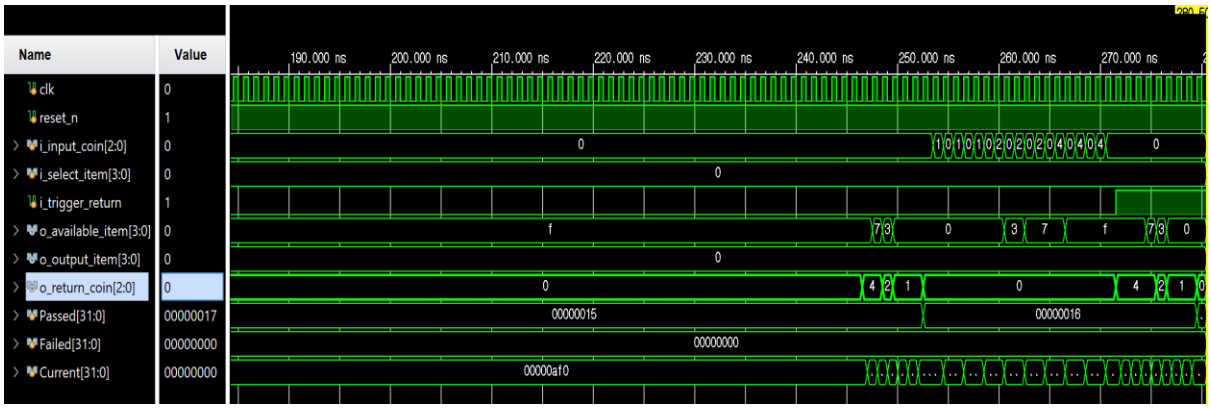
코딩을 하면서 가장 많이 소통하며 고민했던 부분은 output과 output reg의 구분이었다. output으로 쓰이면 무작정 output reg로 쓰자는 의견도 나왔고, 레지스터를 함부로 쓰면 오류가 발생하기 굉장히 쉬워지기 때문에 상황에 맞게 꼭 필요한 때만 레지스터를 사용하자는 의견도 나왔다. 우리 팀은 많은 토론 끝에 후자가 옳다는 결론을 내렸다. 단순히 and게이트나 or게이트로 묶어서 표현할 수 있는 것은 output을 사용하고, always를 통해 변화하는 값에 대해 산술적인 연산으로 값이 대입을 통해 정해져야 하는 것은 output reg를 사용하였다. 이는 구현의 편의를 위해 reg를 사용한 경우이며, 신호가 끝난 이후에도 값을 저장하고 있어야 하는 경우에도 레지스터를 사용하였다.

always문에 대해서 처음 코딩을 할 때에는 유의미하게 변화한 신호에 대해서만 변화를 처리하자고 여기며 코딩하였다.

```
always @(*) begin
    case(i_input_coin)
        3'b100: current_total_nxt<=current_total+coin_value[2];
        3'b010: current_total_nxt<=current_total+coin_value[1];
        3'b001: current_total_nxt<=current_total+coin_value[0];
        default: current_total_nxt<=current_total;
    endcase
end
```

calculate_current_state 모듈의 i_input_coin의 입력에 따른 current_total_nxt를 변화시켜 주는 코드 부분을 예시로 가져왔다. 이 코드의 경우 결국 i_input_coin이 사용자에 의해 변화할 때만 실행시켜 주는 코드이기에 always @(i_input_coin) begin 이런 식으로 코딩을 했었다. 하지만 이러한 상태로 테스트를 해보자

current_total이 계속 제대로 반영이 안되는 듯한 결과를 보여주며 결국 테스트를 제대로 수행하지 못하였다. 이를 실험적으로 *로 바꾸어 넓은 범위에 대해 계속 체크해주는 방식으로 바꾸었는데 결과가 성공적으로 나왔다. 결과가 다르게 나오는 이유를 완벽한 논리로 파악하지는 못했으나, 값을 수시로 업데이트 시켜주는 것이 수치 보존에서 유의미하기 때문으로 해석하였다. 실제로 코딩이 끝나고 얼마의 시간이 흐른 뒤 공지 메일에서 꼭 always @(*)를 사용하라고 하였다.



테스트를 모두 성공적으로 완수하였다.

5. conclusion

Vending machine을 구현하며 동기/비동기, register의 사용성, 회로 구성 방법 등에 대한 이해를 높일 수 있었다. discussion에서는 다루지 않았지만, 동기/비동기의 구현 차이가 출력을 완전히 뒤바꿔버리는 경우를 경험하며 회로를 디자인할 때 데이터 흐름에 대한 설계뿐 아니라 동기/비동기에 대한 고려, register의 사용 여부에 대한 고려도 함께 해야함을 알게 되었다.