Lab 2. Vending Machine RTL

CSED311 Computer Architecture Lab

20190065 강두경, 20190316 유병호

Introduction

이번 랩의 목표는 Vending Machine RTL을 구현하는 것이다. FSM 개념을 활용하여 Vending Machine의 분기별 flow를 구현한다. 일반적인 Vending Machine처럼, 번호당 가격이 주어진 동전을 넣으면 안에 금액이 쌓이며, 주어져 있는 아이템의 가격보다 더 큰 양의 금액이 들어오면 그 아이템을 살 수 있게 된다. 사용자가 아이템을 고르면 그 아이템이 dispense되며, 그만큼 금액이 빠진다. 일정 대기 시간 동안 아무 행동도 하지 않거나 반환 버튼을 누르면 Vending Machine 안에 있는 금액은 반환된다.
Vending Machine의 interface는 다음과 같다.

INPUT		
Signal	**Description**	**Number of bit(s)**
i_input_coin	Insert Coin	1 for each type of coins
i_select_item	Select Item	1 for each type of items
i_return_trigger	Return change	1
clk	clock	1
reset_n	reset	1

OUTPUT		
Signal	**Description**	**Number of bit(s)**
o_output_item	Indicate dispensed items	1 for each type of items
o_available_item	Indicate item availability	1 for each type of item
o_return_coin	Indicate type of coin (change)	1 for each type of coins

Vending Machine은 use-case를 모두 만족하도록 설계해야 한다. use-case는 다음과 같다.

Sequence
1. Insert money (available money unit: 100, 500, 1000 won) and initialize waiting time(=100)
2. Vending machine shows all available items where (item cost <= current money)
  2.a. decrease waiting time(-1)

3.a. Insert money within the waiting time
  3.a.1. Go to 2 and initialize waiting time(=100)
3.b. Select an item within the waiting time
  3.b.a. Case1: the item is available
    3.b.a.1. The item is dispensed
    3.b.a.2. Go to 2 and initialize waiting time(=100)
  3.b.b. Case 2: the item is unavailable
    3.b.b.1. Nothing happens. Waiting time is not reset.
3.c. No input within the waiting time
  3.c.1 Return changes

a*. Whenever press the return button
  a*.1. Return changes
  a*.2. Go to 1

FSM은 Finite State Machine의 약자로, 유한 개의 상태를 가지는 머신을 의미한다. Moore Machine과 Mealy Machine의 두 가지 종류가 있는데, Moore Machine은 현재 상태에 의해서만 다음 상태가 결정되는 머신을 의미하며 Mealy Machine은 다음 상태가 현재 상태와 Input 값 둘 다에 의존한다는 것이 차이점이다.

Verilog를 이용해 Vending Machine의 기능을 실행하는 FSM을 구현할 텐데, Combinational Logic과 Sequential Logic이 따로 분리되어 있는 구조를 따를 것이다.

Design

모듈별로 수행하는 일에 차이가 있다.

우선, **check_time_and_coin** 모듈이 담당하는 기능은 다음과 같다.

- 버튼을 누르거나 동전을 넣었을 때, (단 누른 버튼은 available해야 함) wait_time을 100으로 재설정
- 현재 Vending Machine에 있는 금액과 동전 값을 비교하여 o_return_coin 설정
- wait_time이 0보다 클 때 wait_time이 1씩 감소하도록 하기 - sequential logic

다음으로, **calculate_current_state** 모듈이 담당하는 기능은 다음과 같다.

- i_input_coin 값으로부터 input_total 값 계산
- i_select_item 값으로부터 output_total 값 계산
- o-return_coin 값으로부터 return_total 값 계산
- Vending Machine의 잔액과 아이템 가격을 비교하여 o_available item 계산
- o_output_item 계산 (간단히 i_select_item과 o_available_item을 bitwise and 연산하여 구현)

마지막으로, **change_state** 모듈이 담당하는 기능은 다음과 같다.

- current_state에 대한 sequnential logic으로, current_state를 current_next_state로 변경한다. - sequential logic

이 때 주어진 vending_machine 스켈레톤 코드에서 모듈의 인터페이스를 수정하였다. check_time_and_coin에 coin_value와 item_price을 연결했고, calculate_current_state에 o_return_coin을 연결했다.

current_total은 현재 Vending Machine에 있는 잔액의 값을 갖는다. 잔액의 값이 곧 state의 역할을 하고, 이 state와 input 값 (동전, 아이템 등)에 따라 다음 state가 바뀌고 사용자에게 output도 의존하여 바뀌는 Mealy Machine을 구현하였다.

Implementation

check_time_and_coin

처음으로 initial begin 절에서 wait_time과 o_return_coin을 초기화한다.

```
initial begin
    wait_time = 0;
    o_return_coin = 0;
end
```

우선 반환 시간을 업데이트하는 코드를 always 절 내부에 작성한다. 동전이 하나라도 들어왔으면 wait_time은 100으로 설정되고, 반환도 재설정된다. 만약 아이템 호출이 들어왔다면, 현재 벤딩 머신의 잔액이 아이템의 금액보다 많을 경우에는 wait_time을 재설정하고 그렇지 않을 경우에는 wait_time을 그대로 둔다.

```
always @(i_select_item,i_input_coin) begin
    if(i_input_coin != 0) begin
        wait_time = 100;
        o_return_coin = 0;
    end

    if(i_select_item != 0) begin
        if (i_select_item[0] == 1 && current_total >= item_price[0]) begin
            wait_time = 100;
        end
        else if (i_select_item[1] == 1 && current_total >= item_price[1]) begin
            wait_time = 100;
        end
        else if (i_select_item[2] == 1 && current_total >= item_price[2]) begin
            wait_time = 100;
        end
        else if (i_select_item[3] == 1 && current_total >= item_price[3]) begin
            wait_time = 100;
        end
        else begin
            wait_time = wait_time;
        end
    end
end
```

다음으로 얼마를 반환할지 결정하는 코드를 아래에 작성한다. current_total과 coin_value의 각 비트를 비교하여, current_total이 더 크다면 o_return_coin의 해당하는 비트를 1로 설정한다. 이 때 else 구문을 사용했는데, 그 이유는 한 번에 하나의 o_return_coin 비트만 1이 되게 하기를, 특히 가장 큰 가치를 갖는 동전에 해당하는 비트만 1이 되기를 의도하기 때문이다.

가령 잔액이 1300원이고 동전의 가치가 1000원, 500원, 100원인 경우 1300원은 1000원보다도, 500원보다도, 100원보다도 크지만 이후 return 연산에서 1000, 500, 100원을 모두 반환하면 잔액은 1300원보다 작아진다. 그렇기 때문에 가장 큰 가치를 갖는 1000원의 비트만 1이 되도록 하고, 반환되어 300원이 남게 되면 그 이후에는 100원에 해당하는 비트가 1이 되도록 한다. 또 다른 이유는 calculate_current_state 모듈에서 i_input_coin과 i_select_item 모두 testbench로부터 오직 하나의 비트만 1이 되도록 강제되는데 이와 parallel한 의미를 갖게 하기 위함도 있다.

```
always @(*) begin
// TODO: o_return_coin
    o_return_coin = 0;

    if(wait_time == 0) begin
    if(current_total >= coin_value[2]) o_return_coin[2] = 1;
    else if(current_total >= coin_value[1]) o_return_coin[1] = 1;
    else if(current_total >= coin_value[0]) o_return_coin[0] = 1;
    else o_return_coin = 0;
    end
end
```

wait_time은 clk이 지날 때마다 1씩 줄어들어야 한다. 물론 clk은 음수가 되어서는 안 되며, 0일 때는 0을 유지해야 한다. 이는 state를 직접적으로 바꾸는 것이므로 sequential logic이어야 하며, non-blocking assignment를 사용하여야 한다. 구현은 다음과 같다. reset_n이 0일 경우에는 wait_time을 0으로 초기화한다.

```
always @(posedge clk ) begin
    if (!reset_n) begin
        wait_time <= 0;
    end
    else begin
        if(wait_time > 0) wait_time <= wait_time -1;
    end
end
```

calculate_current_state

current_total_nxt를 계산하는 combinational logic을 작성한다. current_total은 vending machine의 잔액을 유지해야 한다. vending machine의 잔액에는 현재 들어온 돈이 추가되며, 아이템의 금액과 반환된 금액만큼 차감된다. 들어온 금액을 input_total, 아이템의 금액을 output_total, 반환된 금액을 return_total이라고 하면 current_total_nxt는 current_total + input_total - output_total - return_total과 같다.

input_total, output_total, return_total은 각각 i_input_coin, i_select_item, o_return_coin에서 1인 비트의 index에 해당하는 가격을 대입한다. 예를 들어 i_input_coin이 100이면 두 번째 코인의 금액인 1000원이 input_total에 대입된다. output_total의 경우 현재 금액과 대소비교를 하여 현재 금액이 더 많을 경우에만 계산한다. 구현은 다음과 같다.

```
always @(*) begin
    input_total = 0;
    output_total = 0;
    return_total = 0;
    current_total_nxt = 0;

    if(o_return_coin[2] == 1 ) begin
        return_total = coin_value[2];
    end
    else if(o_return_coin[1] == 1 ) begin
        return_total = coin_value[1];
    end
    else if(o_return_coin[0] == 1 ) begin
        return_total = coin_value[0];
    end
    else begin
        return_total = 0;
    end

    if (i_input_coin[0] == 1 ) begin
        input_total = coin_value[0];
    end
    else if (i_input_coin[1] == 1 ) begin
        input_total = coin_value[1];
    end
    else if (i_input_coin[2] == 1 ) begin
        input_total = coin_value[2];
    end
    else input_total = 0;
```

```
    if(i_select_item[0] == 1 && current_total >= item_price[0]) begin
        output_total = item_price[0];
    end
    else if(i_select_item[1] == 1 && current_total >= item_price[1]) begin
        output_total = item_price[1];
    end
    else if(i_select_item[2] == 1 && current_total >= item_price[2]) begin
        output_total = item_price[2];
    end
    else if(i_select_item[3] == 1 && current_total >= item_price[3]) begin
        output_total = item_price[3];
    end
    else begin
        output_total = 0;
    end

    current_total_nxt = input_total + current_total - output_total -
return_total;
end
```

Output에 대해서 o_available_item과 o_output_item을 계산해야 한다. o_available_item은 간단히 각 index에 대해 아이템의 가격이 현재 금액보다 작은지 여부로 값을 결정한다. o_output_item은 선택한 아이템이 available해야 하는데 이 계산값은 정확히 i_select_item과 o_available_item을 정확히 bitwise AND 연산한 값과 같다. 구현은 다음과 같다.

```
always @(*) begin
    o_available_item = 0; o_output_item = 0;

    if(current_total >= item_price[0]) o_available_item[0] = 1;
    if(current_total >= item_price[1]) o_available_item[1] = 1;
    if(current_total >= item_price[2]) o_available_item[2] = 1;
    if(current_total >= item_price[3]) o_available_item[3] = 1;

    o_output_item = i_select_item & o_available_item;
end
```

change_state

마지막으로 current_total을 current_total_nxt로 바꿔주는 sequential logic을 change_state 모듈에 작성한다.

```
always @(posedge clk ) begin
    if (!reset_n) begin
        current_total <= 0;
    end
    else begin
        current_total <= current_total_nxt;
    end
end
```

Discussion

가장 어려웠던 부분은 current_total이 제대로 값이 반영되지 않는 버그를 수정하는 과정이었다. modelsim wave로 확인한 결과 current_total이 0에서 100원이 추가되어 100(110100)이 되는 것이 기대되지만, 실제로는 1이 되어야 할 부분이 xx0x00이 되는 오류를 맞이했다. 대입 과정에 문제가 있는지 한참 동안 살펴봤지만 문제의 원인은 current_total을 다른 모듈에서 0으로 대입하고 있는 것이었다. 동시에 서로 다른 모듈에서 한 쪽은 0으로, 한 쪽은 1로 대입하고 있다 보니 겹치는 부분에서 문제가 생긴 것이다. 따라서 이번 랩을 경험으로 하나의 레지스터를 서로 다른 모듈 혹은 서로 다른 always에서 assign하면 안 된다는 것을 깨달았다.

한 가지 의문인 부분은 본 Verilog RTL은 FSM Mealy Machine을 바탕으로 만드는데, state를 잔액으로 설정하는 것을 Finite한 state라고 할 수 있을지가 의문이다. 사용자가 언제 돈을 추가하고 아이템을 고를지 알 수 없는 상황에서 Vending Machine의 잔액은 반드시 어떤 변수 혹은 레지스터에 저장되어야만 하는데, 이는 사실상 무한해질 수 있기 때문에 FSM의 Finite 조건을 만족하는지 잘 모르겠다. 여러 차례 상의했지만 결국 현재 잔액을 state로 설정하는 것이 가장 합리적이라는 결론을 내렸다.

Verilog에서 for loop은 루프를 도는 횟수가 상수가 아니라 변동 가능할 경우 synthesize 단계에서 몇 번을 돌지 예측 불가능하기 때문에 synthesizable하지 않다. 아이템의 종류의 갯수만큼, 혹은 동전의 종류의 갯수만큼 비트별로 연산을 해 주는 반복 가능한 코드가 많았지만 synthesizable 이슈 때문에 루프문을 따로 사용하지 않았다. 다만 실제 vending machine은 아이템의 종류가 언제든 바뀔 수 있고, 동전의 가격과 종류 또한 언제든 추가될 수 있기에 조금 더 일반화된 RTL을 디자인하는 방법이 있을 것 같다.

Conclusion

Verilog의 기본적인 syntax나 문법을 이해하는 것에서 벗어나, FSM 기반의 RTL을 Combinational Logic과 Sequential Logic의 차이점을 이해하여 구분지어 쓰는 방법을 익혔다. 다른 Machine의 경우도 FSM 기반으로 설계를 해 보면 실력 향상에 큰 도움이 될 것이라 확신한다.