2025 Spring

CSED311

컴퓨터구조

Lab 1 report

Team ID: 15

팀원 1: 20230345 이성재

팀원 2: 20230355 정지성

목차

I. 서론

이 과제에서는 vending machine을 베릴로그로 구현하는 것을 요구한다. 앞으로의 랩들에서 CPU를 구현할텐데, 이번 랩은 이를 위해 베릴로그 및 작업 환경에 익숙해지는 인트로 느낌의 랩이라고 볼 수 있다.

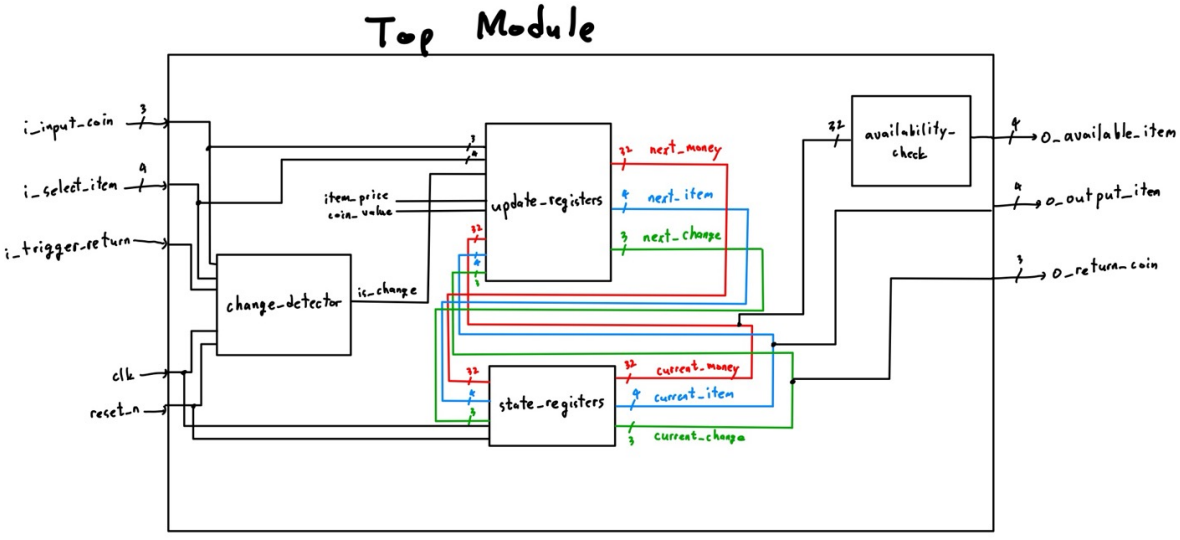
주어진 tb_vending_machine.cpp 테스트벤치의 테스트 케이스들을 최대한 일반적으로 해결할 수 있는 코드를 작성했지만, 테스트벤치에서 다루지 않은 edge case들을 모두 고려하지는 않았으며 이 내용은 IV. 논의사항 부분에서 자세히 서술하도록 하겠다.

II. 디자인

Moore machine으로 구현하였다. 자판기의 현재 재정 상태 (**current_money**), 출력한 아이템 (**current_item**), 반환된 동전(**current_change**)의 정보를 모두 레지스터에 저장했으며 이들이 FSM의 state를 구성한다. 또한 이 레지스터의 값들은 간단한 combinational logic을 통해 전체 모듈의 output으로 가기 때문에 Moore machine이라고 할 수 있다.

(**change_detector**내부의 타이머도 레지스터를 쓰지만 이를 FSM의 state로 보지는 않았다.)

주어진 스켈레톤 코드 대신 아래 그림과 같은 구조의 모듈을 설계했다.



Sequential logic이 적용된 부분은 **change_detector**의 타이머 부분과 **state_registers** 모듈 전체이며, 나머지는 combinational logic이다.

위 디자인의 큰 특징은 **change_detector**을 제외한 다른 모듈이 타이머를 굳이 신경쓰지 않아도 된다는 점이다. **change_detector** 모듈이 타이머 정보를 이용해 **is_change**를 계산하며, 레지스터를 업데이트하는 **update_registers** 모듈은 이 timer 값에 직접 접근하지 않는다. 대신 **change_detector**이 번역해 준 정보를 이용해 값을 업데이트할 수 있게 된다.

각 모듈의 input/output 및 모듈의 역할을 정리하겠다.

A. change_detector

Input

- **i_input_coin**, **i_select_item** (타이머 초기화 용)
- **i_trigger_return**,(반환 요청)
- **clk**, **reset_n** (타이머 구현 및 초기화)

Output: **is_change** (반환 여부에 관한 정보)

- 반환을 할지 말지 결정하는 **is_change**를 output으로 내놓는 모듈이다. 우리의 자판기가 반환을 해야 하는 경우는 (1) 사용자가 **i_trigger_return**을 통해 직접 반환을 요청한 경우, (2) Timer가 0에 도달한 경우 정도가 있다.
- Timer의 구현이 필요하기 때문에 sequential logic이 구현된 곳이기도 하다. (Timer의 구현은 III. 구현에서 집중해서 다루겠다.)

B. update_registers

Input

- **current_money**, **current_item**, **current_change** (레지스터 현재 상태)
- **i_input_coin**, **i_select_item**, **is_change** (다음상태 계산 용 정보)
- **item_price**, **coin_value** (상수처럼 사용)

Output: **next_money**, **next_item**, **next_change** (레지스터 다음 상태 — 이 모듈에서 계산)

- **state_registers**에 담겨 있는 레지스터의 현재 상태와 여러 정보들을 이용해 레지스터의 다음 상태를 계산하는 combinational logic 모듈이다.

C. state_registers

Input

- **next_money**, **next_item**, **next_change** (combinational logic으로 계산된 레지스터의 다음 상태)
- **clk**, **reset_n** (레지스터 구현 및 초기화)

Output: **current_money**, **current_item**, **current_change** (레지스터의 현재 상태)

- State register들을 담은 모듈이다. 앞선 **update_registers** 모듈을 통해 계산된 next 값들을 input으로 받아 register에 업데이트 시켜 준다.

D. availability_check

Input: **current_money**
Output: **o_available_item**

- 현재 자판기의 재정 상태를 받아서 출력가능한 아이템들을 출력하는 매우 간단한 combinational logic 모듈이다.

III. 구현

A. change_detector

- 우선 타이머의 behavior을 모두 정리해 보자. (타이머의 값은 **wait_time** 레지스터에 저장해 두었다.)
  - 기본값은 0
  - **i_trigger_return** 입력을 받으면 (사용자가 반환요청했을 때), 0으로 초기화
  - 사용자가 정상적인 input을 넣으면 (**i_input_coin** 또는 **i_select_item**) 100으로 초기화
  - **clk**을 받을 때마다 1씩 감소; 0인 경우 감소하지 않음
- 위 정보가 모두 담긴 sequential logic 코드를 작성하면 다음과 같다.

```
always @(posedge clk) begin
    if (!reset_n) begin
        wait_time <= 0;
    end
    else if (i_trigger_return == 1) begin
        wait_time <= 0;
    end
    else if (i_input_coin != 0 || i_select_item != 0) begin
        wait_time <= `kWaitTime;
    end
    else if (wait_time != 0) begin
        wait_time <= wait_time - 1;
    end
end
```

- 최종 출력인 **is_change**는 타이머가 0이거나 **i_trigger_return**을 받은 경우이므로, combinational logic을 이용해 마무리할 수 있다.

```
assign is_change = (wait_time == 0) || i_trigger_return;
```

B. update_registers

1. next_money와 next_change의 계산

- 자판기의 재정 상태가 달라지는 경우 (**next_money**가 업데이트되는 경우)는 다음의 세 가지이다. (c에서는 **next_change**까지 업데이트해 주면 되겠다.)
    a. 사용자가 돈을 넣은 경우 (**i_input_coin != 0**)
    b. 사용자가 아이템을 요청한 경우 (**i_select_item != 0**)
    c. 반환해야 하는 경우 (**is_change == 1**)
- 우선 **next_money**와 **next_change**를 **current_money**와 **current_change**로 initialize한다. 이 상태에서 바꿔야 할 bit들만 추가적인 논리로 바꿔주면 된다.

```
// Combinational logic for next_money & next_change
always @(*) begin
    next_money = current_money;
    next_change = current_change;
```

- a & b: input의 값에 따라 돈을 **next_money**에 적절하게 더하거나 빼면 된다.

```
    // Adding logic for next_money
    if (i_input_coin != 0) begin
        for (i = 0; i < `kNumCoins; i++) begin
            next_money += coin_value[i] * i_input_coin[i];
        end
    end

    // Subtracting logic for next_money
    else if (i_select_item != 0) begin
        for (i = 0; i < `kNumItems; i++) begin
            next_money -= item_price[i] * i_select_item[i];
        end
    end
```

- c: **next_change**를 적절히 계산하고, 그 과정에서 **next_money**를 0으로 만드는 과정을 생각할 수 있다.
    - 현재 자판기에 들어있는 돈을 100, 500, 1000원으로 표현하면 되는데, 큰 단위의 돈부터 확인하여 빼면 된다.
    - 예를 들어 1100원을 거슬러야 한다고 하자. 1100>1000이므로 1000원을 거스르고 100원이 남는다. 100<500이므로 500원을 거슬러주지 않는다. 100=100이므로 100원을 거슬러 주면 반환이 완료된다.

```
    // if change is requested, next_money should be 0 and change should exist
    // Assumption: money can always be represented with one of 100, 500, 1000 each
    else if (is_change == 1) begin
        for (i = 0; i < `kNumCoins; i++) begin
            if (next_money >= coin_value[`kNumCoins-1 - i]) begin
                next_money -= coin_value[`kNumCoins-1 - i];
                next_change[`kNumCoins-1 - i] = 1'b1;
            end
        end
    end
```

2. **next_item**의 계산: **current_item**의 정보에 **i_select_item**으로 사용자가 선택한 아이템을 누적시키는 느낌으로 생각했다. 따라서 bitwise or을 사용한다.

```
// Combinational logic for next item
// Assumption: always possible to dispense
always @(*) begin
    // Used bitwise OR so that it doesn't get overwritten
    next_item = i_select_item | current_item;
end
```

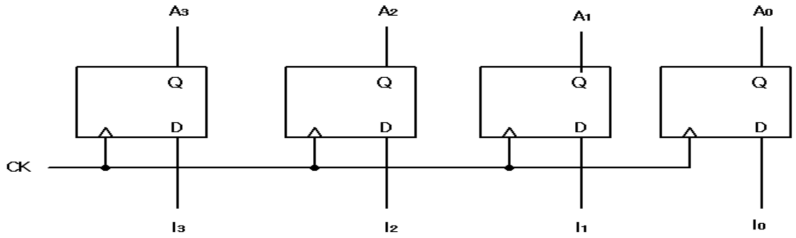
C. state_registers

- **current_money**, **current_item**, **current_change**의 register들을 저장하고 있는 sequential logic 모듈이다.
- **clk** 사이클이 돌 때마다, 앞선 **update_registers** 모듈이 계산한 새로운 값(**next_***)들로 레지스터 값을 업데이트해 준다.

```
always @(posedge clk) begin
    if (!reset_n) begin
        current_money <= 0;
        current_item <= 0;
        current_change <= 0;
    end
    else begin
        current_money <= next_money;
        current_item <= next_item;
        current_change <= next_change;
    end
end
```

- 이 모듈은 그 자체로 기다란 parallel load register이라고 생각해줄 수 있다.

(ex) 4-bit parallel load register



(image from CSED273 lecture slides)

D. availability_check

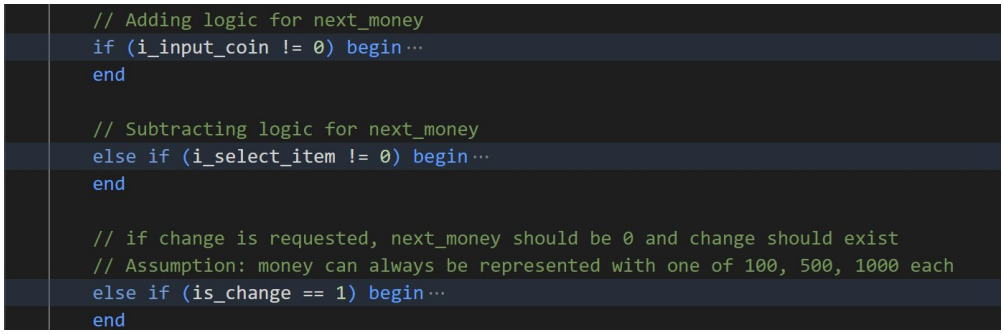
- 현재 자판기의 재정상태가 i번째 아이템의 가격보다 높다면 **o_available_item**의 i번째 비트가 1이 되게 하는 간단한 combinational logic이다.

```
// Combinational logic to convert current money -> available items
always @(*) begin
    o_available_item = 0;
    for (i = 0; i < `kNumItems; i++) begin
        if (item_price[i] <= current_money) begin
            o_available_item[i] = 1'b1;
        end
    end
end
```

IV. 논의사항

이 과제는 주어진 테스트 케이스를 통과하는 것에 중점을 두고 있어 실제 일어날 수 있는 다양한 상황에 대한 처리는 불필요했다. 또한 몇 가지 가정 하에 코드를 작성했다.

1. 세가지 종류의 입력(**i_input_coin**, **i_select_item**, **i_trigger_return**)이 동시에 들어오는 경우는 없다고 가정했다.
   - 그 가정하에 **update_registers**부분을 if, else if 문으로 작성하였다.
     - 예를 들어, 만약 **i_input_coin**과 **i_select_item**의 입력이 동시에 들어왔다면 if 문에서 다루는 **i_input_coin** 부분은 정상적으로 처리가 되지만 else if 문에서 다루는 **i_select_item** 부분은 처리가 되지 않을 것이다.



   - 현실적으로 자판기에 돈을 넣으면서 상품을 선택하는 것이 말이 안 되기도 하고 테스트 케이스에서도 그런 edge case는 검사하지 않기 때문에 위와 같은 가정을 하며 모듈들을 구현했다.

2. 반환해야 하는 동전은 종류당 최대 1개임을 가정했다.
   - 100원짜리 동전을 여러 번 넣을 수는 있지만, 반환을 해야 하는 상황에선 100, 500, 1000원짜리 동전을 최대 하나씩만 사용해서 반환해야 한다고 가정했다.
   - Testbench를 보면 반환 요청 후 6 cycle 이후에 **o_return_coin** output의 전체를 읽는 방식으로 반환 코인을 확인하기 때문이다.
   - 예를 들어 200원을 반환해야 하는 상황은 구현하지 않았다.

3. 사용자가 선택한 item은 항상 available하다고 가정했다.
   - 투입한 금액에 따라 무엇이 available 한지는 **availability_check** 모듈에서 확인하지만 **i_select_item**으로 표현되는 사용자의 입력이 available한 상품인지를 체크하는 모듈이나 로직은 구현하지 않았다.
     - ppt의 "Assumption: infinite item, change"에 의한 결정이다.

V. 결론

이 과제를 통해 기본적인 베릴로그의 문법과 FSM을 디자인하는 방법을 익힐 수 있었다. 처음엔 일반적인 케이스들을 모두 다룰 수 있는 자판기를 설계해야 한다고 생각해서 막막했지만 테스트 케이스가 많은 edge case를 고려하지는 않았기 때문에 구현하기에 복잡하지 않았다.

또한 스켈레톤 코드를 그대로 이용하지 않은 것이 결과적으로 더 좋은 경험이 되었던 것 같다. 문제를 보고 필요한 모듈들이 무엇일지부터 생각해보는 과정에서 이 과목에서 길러야 할 소양을 기를 수 있었다고 생각한다.

체감상 소프트웨어 프로그래밍 언어보다 더 꼼꼼하게 작성해야 한다는 점을 느꼈다. always block을 사용할 때의 combinational logic과 sequential logic의 syntax 차이도 익숙하지 않았다. (combinational logic을 작성할 때에는 blocking, sequential logic을 작성할 때에는 nonblocking assignment을 사용하는 것) 또한 always block에서 combinational logic을 짜려고 할 때 왼쪽의 variable은 (실제로는 latch가 없더라도) reg으로 define되어야 한다는 점, if문으로 모든 bit을 정의해 주지 않는다면 latch가 infer되어 오류가 뜨는 점 등이 익숙하지 않았다. 그렇지만 이번에 연습해 보며 베릴로그에 훨씬 익숙해졌다. 덕분에 다음 랩들부터는 베릴로그 관련 이슈에 대해 신경을 덜 쓰고 CPU 디자인에 온전히 집중할 수 있을 것 같아서 좋다.