SOC 실습 프로젝트 보고서

(Implementation of the FIFO Data Structure)

과목명	SOC설계(ITE4003)
담당 교수님	임재명 교수님
제출일	2022년 05월 14일(토요일)
소속	한양대학교 공과대학 컴퓨터소프트웨어학부
학번	**이름**
2019009261	최가온(CHOI GA ON)

-목차-

Ⅰ. Introduction

1. 프로젝트 주제 및 목적
2. 프로젝트 실행 환경
3. 이론적 배경

Ⅱ. Experiment

1. **문제 1** – Non-circular FIFO data structure
2. **문제 2** – Circular FIFO data structure

Ⅲ. Conclusion

프로젝트 결과 요약 및 의의

Appendix

프로젝트 코드 및 그림자료 GITHUB 링크 첨부

Ⅰ. Introduction

1. 프로젝트 주제 및 목적

위 프로젝트에서는 선입선출 방식(FIFO; First In First Out)을 기반으로 한 자료구조를 Verilog을 이용하여 구현하는 것이 기본적인 목적이다.

2. 프로젝트 실행 환경

아래의 환경에서 프로젝트 개발과 테스트를 진행하였다.

1) Quartus Prime 20.1

2) ModelSim – INTEL FPGA STARTER EDITION 2020.1

3. 이론적 배경

자료구조(Data Structure)는 컴퓨터공학에서 효율적인 접근 및 수정을 가능하게 하는 자료의 조직, 관리, 저장에 대한 개념을 일컫는 말이다. 자료구조는 크게 선형구조, 비선형구조, 파일구조로 나뉘는데, 큐와 스택은 데이터가 연속적으로 연결되어 있는 모양을 띠고 있으므로 이 중 선형구조에 속한다고 할 수 있다.

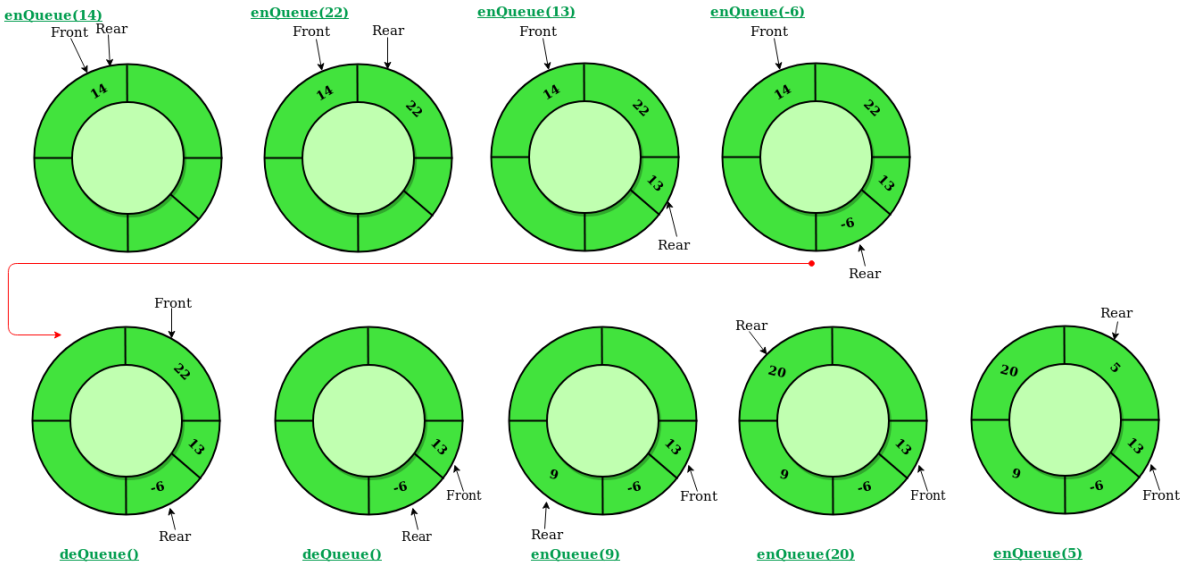
큐(Queue)는 위의 그림에서 볼 수 있듯이 한쪽에서는 삽입 연산이 일어나고, 다른 쪽에서 삭제 작업이 이루어지는 구조이다. 연산이 일어나는 위치적 특성에 따라 가장 먼저 삽입된 데이터가 가장 먼저 삭제되는데 이를 선입선출(FIFO)이라고 한다. C 기반[1]으로 해당 자료구조를 구현하면 데이터를 저장할 배열 1개와 유동적으로 움직일 수 있는 시작 부분(front)과 끝 부분(rear)을 각각 표현할 수 있는 포인터 2개가 필요하다. 큐는 카페 등에서 일어나는 주문과정, 상담원 직원 연결 대기열 등에 사용될

[1] 연결리스트(linked list) 방식으로 구현한 코드이다.

https://github.com/Gaon-Choi/CSE2010/blob/master/lab3-2/p3-2.c

수 있다.

큐를 구현하는 방식에는 비순환형 방식, 순환형 방식으로 두 가지가 있다. 두 가지 구현 방식 모두 선입선출의 원리를 따르나, 순환형 방식은 비순환형 방식의 한계점을 보완한 구조라고 할 수 있다. 위에서 제시된 그림이 비순환형 방식을 도식적으로 나타낸 예시에 해당한다. 2개의 포인터가 유동적으로 움직이는 구조에서 비순환형 방식은 모든 자료구조가 갖는 공간의 유한성 아래에서 한계점을 갖는다. 큐에서 원소가 하나씩 삽입될 때마다 rear 포인터는 1칸씩 증가하게 되어 마침내 꽉 채워진(full) 상태에 도달한다. 이때 원소를 하나씩 삭제한다면 front 포인터가 1칸씩 증가하게 될 것이다. 이러한 과정이 계속 반복된다면 rear 포인터는 최초로 full이 된 시점으로부터 계속 유한한 공간 내에서 한 지점만을 가리키게 되고 자료구조의 끝 부분을 가리키는 고유 속성을 상실하게 된다. 2개의 포인터의 작동에 따라 큐가 동작한다는 점에서, 결국 큐의 가용 공간의 크기는 사용에 따라 0에 가까워진다. 따라서, full 상태가 되었을 때에는 첫번째 원소부터 메모리 상 0번 인덱스 위치로 시작하게끔 모두 옮겨주는 과정이 필요하며(이 부분에서 O(N) 시간 복잡도가 소요됨) 이것을 기준으로 2개의 포인터의 위치도 재지정해야 한다. 이러한 까닭으로 순환형 방식을 도입하게 되었다.



Ⅱ. Experiment

Structure

비순환형 큐(Non-circular Queue)와 순환형 큐(Circular Queue)를 각각 구현하기 이전에 공통적으로 사용하는 요소를 설명할 것이다.

우선 총 3가지의 모듈 인자(module parameter)를 받는데, DATA_WIDTH는 각각의 원소가 갖는 비트 수를 의미한다. RAM_DEPTH는 2 ** ADDR_WIDTH로 결정한다.

```
module non_circular_fifo_rev1
    #(  parameter   DATA_WIDTH  =   8,
        parameter   ADDR_WIDTH  =   2,
        parameter   RAM_DEPTH   =   1<< ADDR_WIDTH ) //double left shifting means RAM_DEPTH = 4
```

입력 포트(input port)는 아래와 같다. clk는 클락(clock)을 나타내고, reset이 활성화되는 경우 포인터 위치 등을 원상 복귀하는 과정이 시작된다. data_in에 write할 경우 입력할 데이터가 입력으로 들어오게 되며, read와 write 각각에 대해 enable 값이 존재한다.

```
    //------------------------------------------------------------------
    // Input Ports
    //------------------------------------------------------------------
    input                               clk,            //  input clock
    input                               reset,      //  active high reset
    input                               wr_cs,      //  write chip select
    input                               rd_cs,      //  read chip select
    input          [DATA_WIDTH-1:0]     data_in,    //  8bit data_in
    input                                   rd_en,      //  read enable
    input                               wr_en,      //  write enable
```

출력 포트(output port)는 아래와 같다. read 시 읽어온 데이터는 data_out에 표시된다. 할당된 메모리 공간에 모두 데이터가 차 있을 경우 full이 1로 활성화되며, 반대로 할당된 공간에 데이터가 아무것도 없는 경우 empty가 활성화된다.

```
    //------------------------------------------------------------------
    // Output Ports
    //------------------------------------------------------------------
    output  reg     [DATA_WIDTH-1:0]    data_out,   //  8bit data out
    output  wire                        full,       //  FIFO full
    output  wire                        empty,      //  FIFO empty
```

full과 empty의 정의는 아래와 같다. 기본적으로 큐에 들어있는 원소의 개수는 data_counter에 저장된다.

```
//  output port
assign full=(data_counter==(RAM_DEPTH)) ? 1'b1 : 1'b0;//data_counter 4 means data full
assign empty = ( data_counter == 1'b0 ) ? 1'b1 : 1'b0;//data_counter 0 means data empty
```

1. 문제1 – Non-circular FIFO data structure

Source Code

아래의 부분은 write 포인터를 움직이는 부분이다. reset이 활성화된다면 모든 부분을 초기화해야 하므로 write pointer를 0으로 위치시키고(0은 배열의 첫 부분 인덱스를 나타낸다.) wr_full을 0으로 설정한다. else문에는 평상시에 작동해야 하는 과정이 명시되어 있다. write가 enable되고 현재 큐가 full인 상태가 아니며 write pointer가 끝단에 위치한 것이 아니라면 write 포인터를 1만큼 증가시킨다. 원소를 하나씩 삽입한다면 write 포인터가 RAM_DEPTH-1에 도달하는데 이는 현재 큐가 full인 상태임을 의미하므로 wr_full을 1로 활성화하여 큐가 가득 차있음을 표시한다.

```
//  write pointer move
always @ ( posedge clk or posedge reset )
begin: WRITE_POINTER
    if ( reset )                                    //if reset=1
    begin
        wr_pointer  <=  'd0;                        // wr_pointer=0
        wr_full     <=  1'b0;                       //wr_full=0
    end
    else
    begin
        // check write chip selct, enable signal and write pointer
        if ( wr_cs && wr_en && !full && (wr_pointer != RAM_DEPTH-1))
            wr_pointer  <=  wr_pointer + 'd1; // increase the write pointer
        else
            wr_pointer  <=  wr_pointer;

        // if wr_pointer== RAM_DEPTH-1, wr_full=1
        if ( wr_pointer == RAM_DEPTH-1 )
            wr_full     <=  1'b1; //it means data is full
        else
            wr_full     <=  wr_full; // if not, keep its value
    end
end
```

다음은 read 포인터를 움직이는 부분이다. write 포인터와 마찬가지로 reset이 활성화되면 read 포인터를 0으로 위치시킨다. 유사한 구조로 rd_en이 활성화되고 현재 큐가 비어있지 않으면서 rd_pointer가 끝단을 가리키는 것이 아니라면 rd_pointer를 1만큼 증가시킨다.

```
//  read pointer move
always @ ( posedge clk or posedge reset )
begin: READ_POINTER
    if ( reset )
    begin
        rd_pointer  <=  'd0;   //if reset=1, rd_pointer=0
    end
    else
    begin
        if ( rd_cs && rd_en && !empty && ( rd_pointer!=RAM_DEPTH-1 ) )  //rd_pointer increments by 1
in this condition
            rd_pointer  <=  rd_pointer + 'd1;
        else
```

```
            rd_pointer  <=  rd_pointer;     //if not, keep its value
    end
end
```

아래는 실제 데이터를 write하는 부분에 대한 구현이다. wr_en이 활성화되어 있고 현재 큐에 들어있는 데이터의 수가 RAM_DEPTH에 도달하지 않았으며 wr_full이 0인 경우에 삽입이 가능하다. 이 조건들을 만족하면 입력받은 데이터인 data_in을 메모리 상에서 wr_pointer가 가리키는 인덱스(위치)에 삽입한다.

```
//  data write
always @ ( posedge clk )
begin: DATA_WRITE
    if ( wr_cs && wr_en  && ( data_counter != RAM_DEPTH ) && !wr_full )
        // blank
        // --> write data_in into memory location that is pointed by wr_pointer
        memory[wr_pointer] <= data_in;
    else
        memory[wr_pointer] <= memory[wr_pointer];
end
```

다음은 data read에 대한 부분이다. rd_cs와 rd_en이 활성화되는 경우에 데이터 읽기 작업을 할 수 있다. 읽기 작업은 큐 상에서 rd_pointer가 가리키는 인덱스 값을 읽어오는 것이다. 따라서 memory[rd_pointer]의 값을 data_out에 넣는 것으로 해당 기능을 구현하였다.

```
//  data read
always @ ( posedge clk )
begin: DATA_READ
    if ( rd_cs && rd_en )
        // blank
        // --> read data from memory location that is pointed by rd_pointer
        data_out <= memory[rd_pointer];
    else
        data_out <= 'hx;
end
```

마지막 부분은 data_counter 값을 갱신하는 부분이다. data_counter는 현재 큐에 들어있는 element의 개수를 의미한다. else if 부분의 조건과 else 부분을 통해 각각의 부분이 의미하는 것을 추론할 수 있다.

i) 1번째 case(if문): no write and read

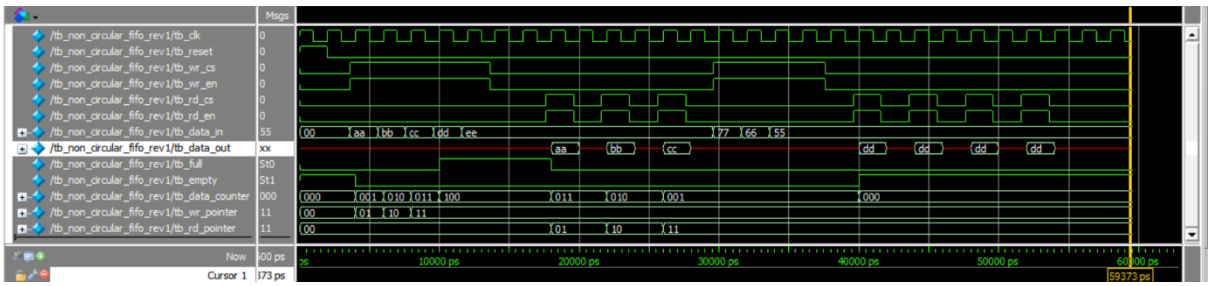
ii) 2번째 case(else if문): write and no read

iii) 3번째 case(else문): no write and no read

```
//  data counter move
always @ ( posedge clk or posedge reset )
begin: DATA_COUNTER
    if ( reset )  //data counter reset
    begin
        data_counter    <=  'd0;
    end
```

```
    else
    begin
        if (!( wr_cs && wr_en ) && ( rd_cs && rd_en ) && !empty)          // blank
            // decrement data_counter
            // : due to the read operation of one element
            // : without any writing new element
            data_counter <= data_counter - 1;
        else if ( ( wr_cs && wr_en ) && ! ( rd_cs && rd_en ) && ( data_counter != RAM_DEPTH )
&& !wr_full )    //  write and no read
            // increment data_counter
            // : due to the insertion of one element
            data_counter <= data_counter + 1;
        else
            // if not, keep it's original value
            data_counter <= data_counter;
    end
end
```

Waveform

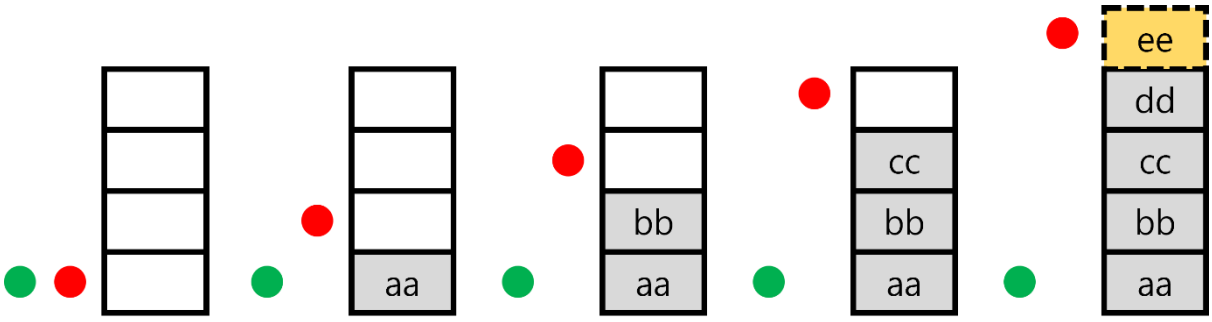


Explanation

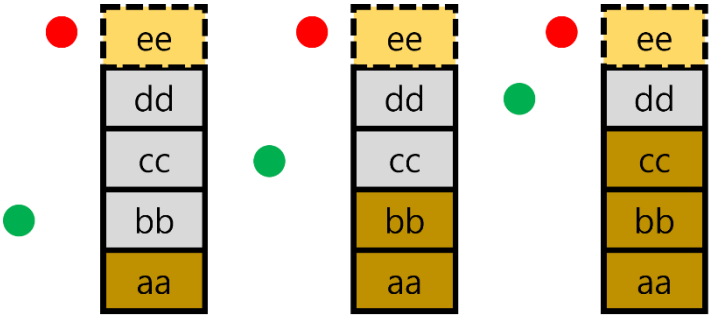
Testbench를 기반으로 각각의 연산 과정에서 큐의 내부에서 일어나는 결과들을 모식적으로 나타낸 것이다. 각 요소가 갖는 의미는 아래와 같다.

	write point
	read point
	read once
	invalid bound (for representation)
	inserted space
	read more than once

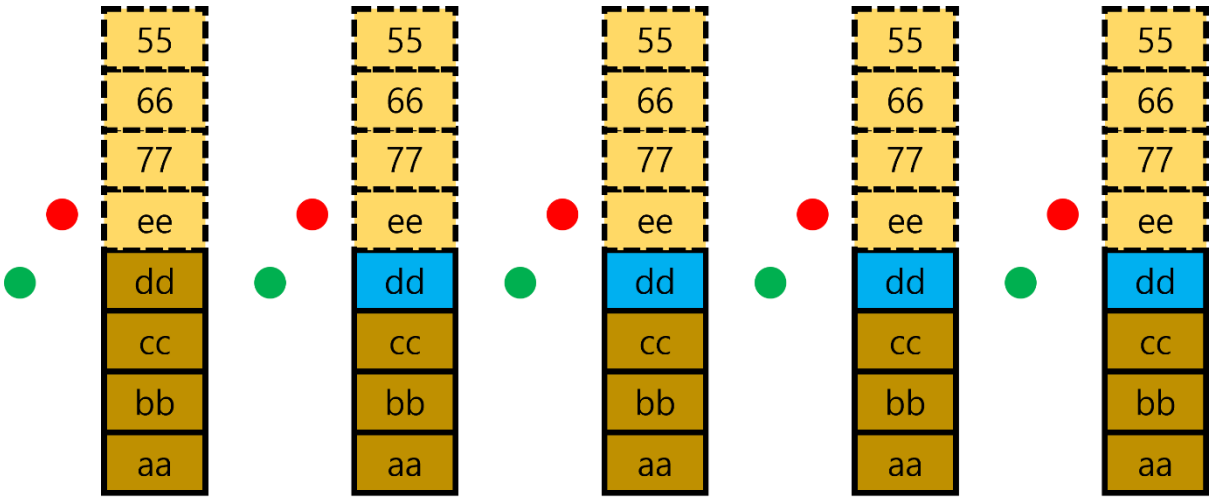
[step1] aa, bb, cc, dd, ee를 각각 write하는 부분이다. 노란색으로 칠해진 부분은 실제 큐 상에서는 존재하지 않으나, 이해를 돕기 위해 표시하였다.



[step2] read 연산을 3번 수행하는 모습이다.



[step3] write 연산을 3번 수행하여 차례대로 77, 66, 55를 삽입하는 부분이다. Non-circular FIFO 구조에서는 삽입이 되지 않는 것을 확인할 수 있다. 이후 read 연산을 4번 수행하게 되는데 각각 위의 자료에 해당하는 dd만 출력됨을 확인할 수 있다.



2. 문제2 – Circular FIFO data structure

Source Code

- Non-circular와 겹치는 부분은 생략한다.
- Non-circular FIFO의 Verilog 파일을 기반으로 작성하였고, 필요한 부분만을 변형하여 Circular FIFO 구조를 구현하였다.

가장 먼저 변형을 준 부분은 wr_full이다. wr_full 부분을 아예 삭제하였는데, Circular FIFO 구조에서는 wr_pointer가 다시 원형으로 돌아와 순환할 수 있도록 하기 위함이다.

```
//-----------------------------------------------------------------
// internal variables declaration
//-----------------------------------------------------------------
reg     [ADDR_WIDTH-1:0]    wr_pointer;            // 2bit regs that can represent 4 address
reg     [ADDR_WIDTH-1:0]    rd_pointer;            // 2bit regs
reg     [DATA_WIDTH-1:0]    memory [0:RAM_DEPTH-1]; // 8bit*4addr memory declaration
reg     [ADDR_WIDTH:0]      data_counter;          // 3bit regs
// reg                        wr_full;              // 1bit regs
```

write 포인터가 순환하여야 한다는 점으로 인해 기존의 write 포인터가 움직이는 부분을 수정해야 했다. Non-circular FIFO 구조에서는 wr_pointer를 1만큼 증가시키기 위한 조건으로 wr_pointer가 끝단을 가리키지 않아야 한다는 것이 있었으나, wr_pointer는 00 → 01 → 10 → 11 → 00 → … 과 같이 순환해야 하므로 해당 조건을 주석으로 처리하여 삭제하였다. 또한, wr_full 부분을 사용하지 않기로 하였으므로 해당 부분도 삭제하였다.

```
//  write pointer move
always @ ( posedge clk or posedge reset )
begin: WRITE_POINTER
    if ( reset )                                    //if reset=1
    begin
        wr_pointer  <=  'd0;                        // wr_pointer=0
        // wr_full     <=  1'b0;                       // wr_full=0
    end
    else
    begin
        if ( wr_cs && wr_en && !full /* && (wr_pointer != RAM_DEPTH-1) */) //check write chip selct,
enable signal and write pointer
            wr_pointer  <=  wr_pointer + 'd1; // increase the write pointer
        else
            wr_pointer  <=  wr_pointer;

            /*
        if ( wr_pointer == RAM_DEPTH-1 )        // if wr_pointer== RAM_DEPTH-1, wr_full=1
            wr_full     <=  1'b1; //it means data is full
        else
            wr_full     <=  wr_full; // if not, keep its value
            */
    end
end
```

read 포인터도 write 포인터와 마찬가지로 순환성(circularity)이 부여되어야 하므로 read 포인터가 큐의 가장 끝단이 아니어야 한다는 조건을 삭제하였다.

```
//  read pointer move
always @ ( posedge clk or posedge reset )
begin: READ_POINTER
    if ( reset )
    begin
        rd_pointer  <=  'd0;   //if reset=1, rd_pointer=0
    end
    else
    begin
        if ( rd_cs && rd_en && !empty /* && ( rd_pointer!=RAM_DEPTH-1 ) */ )  //rd_pointer
increments by 1 in this condition
            rd_pointer  <=  rd_pointer + 'd1;
        else
            rd_pointer  <=  rd_pointer;     //if not, keep its value
    end
end
```

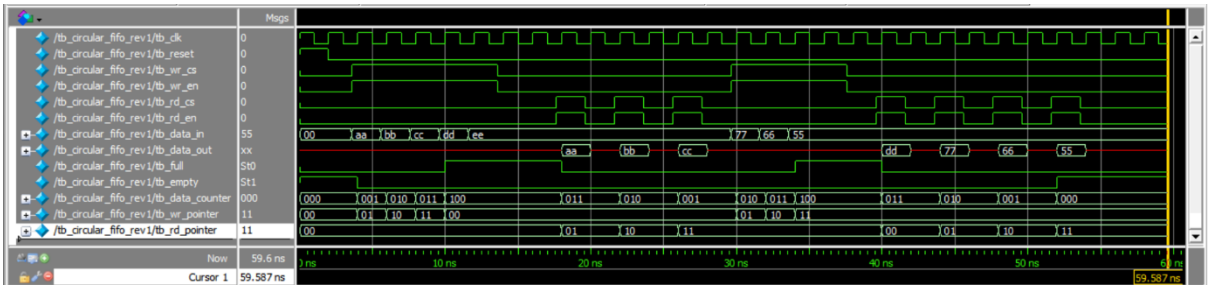
데이터 write하는 부분에서 wr_full 조건 또한 삭제하였다. 다만, Circular FIFO의 경우에도 데이터가 저장되는 공간이 유한하다는 점에는 Non-circular FIFO와 차이가 없다는 점에서 덮어쓰기 등을 허용하지 않기 때문에 data_counter 값을 검사하는 부분은 그대로 두었다.

```
//  data write
always @ ( posedge clk )
begin: DATA_WRITE
    if ( wr_cs && wr_en && (  data_counter != RAM_DEPTH  ) /* && !wr_full */ )
```

data_counter의 값을 1만큼 증가시키는 부분에서도 wr_full 조건을 삭제하였다.

```
//  data counter move
always @ ( posedge clk or posedge reset )
begin: DATA_COUNTER
    if ( reset )  //data counter reset
    begin
        data_counter    <=  'd0;
    end
    else
    begin
        if (!( wr_cs && wr_en ) && ( rd_cs && rd_en ) && !empty)         // blank
            // decrement data_counter
            // : due to the read operation of one element
            // : without any writing new element
            data_counter <= data_counter - 1;
        else if ( ( wr_cs && wr_en ) && ! ( rd_cs && rd_en ) && ( data_counter != RAM_DEPTH  )
/* !wr_full */ )    //  write and no read
```

Waveform

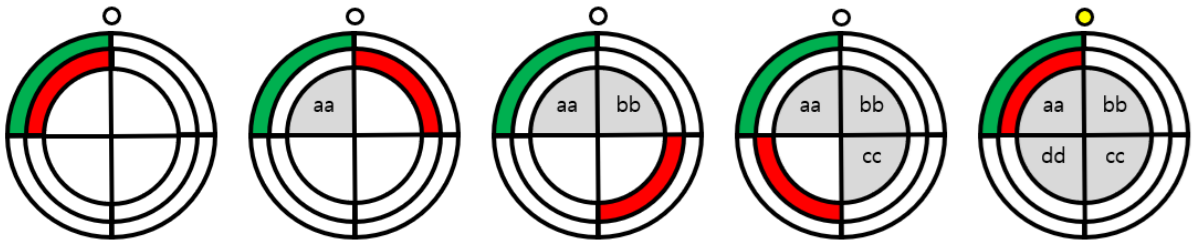


Explanation

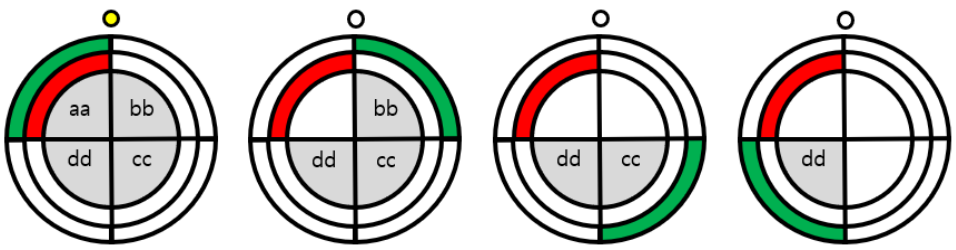
Testbench를 기반으로 각각의 연산 과정에서 큐의 내부에서 일어나는 결과들을 모식적으로 나타내었다. 각 요소가 갖는 의미는 아래와 같다.

	write point
	read point
	inserted space
	write more than once
	full (white if not full)

[step1] aa, bb, cc, dd, ee를 각각 write하는 부분이다.

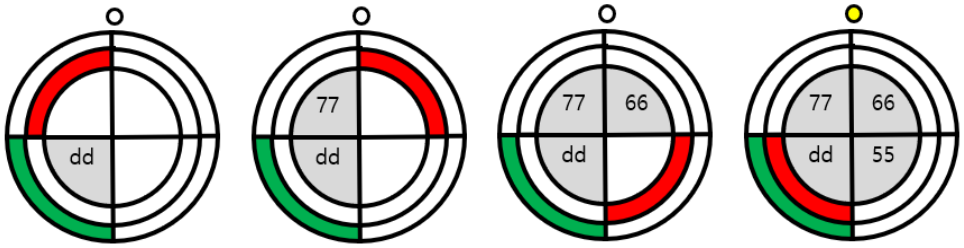


[step2] read 연산을 3번 수행하는 모습이다.

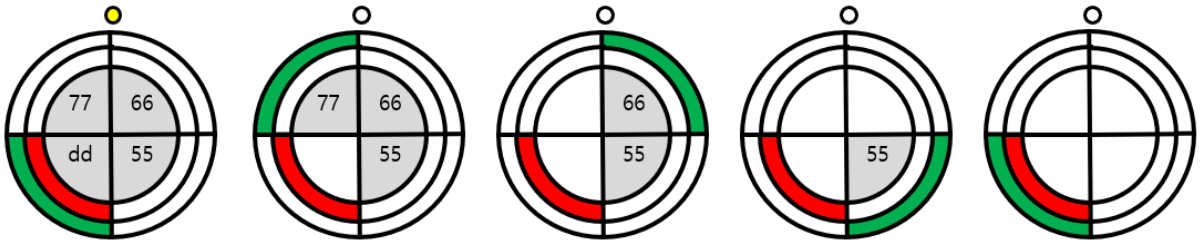


[step3] write 연산을 3번 수행하여 차례대로 77, 66, 55를 삽입하는 부분이다.

Circular FIFO 구조에서는 write 포인터가 순환하며 움직이므로 Non-circular FIFO에서와 다르게 실제로 삽입되는 것을 볼 수 있다.



[step4] read 연산을 4번 연속으로 수행하는 모습이다. read 포인터 또한 write 포인터와 마찬가지로 순환하며 움직인다는 것을 알 수 있다.



Ⅲ. Conclusion

프로젝트 결과 요약 및 의의

이번 프로젝트에서는 선입선출 원리로 작동하는 큐(Queue)를 구현하였다. Non-circular FIFO 방식과 Circular FIFO 방식으로 각각 구현하여 둘 사이의 작동상에서의 차이점을 확인할 수 있었다. Circular FIFO에서는 두 개의 포인터가 순환성(circularity)를 가지기 때문에 삽입과 삭제가 일정 부분 일어나면 더 이상 사용할 수 없어 reset이 필연적으로 필요한 Non-circular FIFO 방식에 비해 지속적으로 사용가능한 자료구조임을 실험적으로 증명할 수 있었다.

Appendix

Author: 최가온(Gaon Choi)

E-mail: choigaon1028@hanyang.ac.kr

Github: https://github.com/Gaon-Choi

1. 프로젝트 각 요소별 코드파일

1-1. Non-circular FIFO data structure Verilog File

https://github.com/Gaon-Choi/ITE4003/blob/main/Project_FIFO_Queue/project_fifo/non_circular_fifo_rev1.v

1-2. Non-circular FIFO data structure Testbench Verilog File

https://github.com/Gaon-Choi/ITE4003/blob/main/Project_FIFO_Queue/project_fifo/tb_non_circular_fifo_rev1.v

1-3. Circular FIFO data structure Verilog File

https://github.com/Gaon-Choi/ITE4003/blob/main/Project_FIFO_Queue/project_fifo/circular_fifo_rev1.v

1-4. Circular FIFO data structure Testbench Verilog File

https://github.com/Gaon-Choi/ITE4003/blob/main/Project_FIFO_Queue/project_fifo/tb_circular_fifo_rev1.v

2. 다이어그램

2-1. Non-circular structure Diagram PNG File

https://github.com/Gaon-Choi/ITE4003/tree/main/Project_FIFO_Queue/images/non_circular_FIFO

2-2. Circular structure Diagram PNG File

https://github.com/Gaon-Choi/ITE4003/tree/main/Project_FIFO_Queue/images/circular_FIFO

3. Report (PDF File)

https://github.com/Gaon-Choi/ITE4003/blob/main/Project_FIFO_Queue/SOC_Practice_Project_FIFO_2019009261_%EC%B5%9C%EA%B0%80%EC%98%A8.pdf