

# 컴퓨터 공학 기초 실험2 보고서

실험제목: Synchronous FIFO

실험일자: 2023년 10월 30일 (월)

제출일자: 2023년 11월 08일 (수)

학 과: 컴퓨터공학과

담당교수: 이준환 교수님

실습분반: 월요일 0, 1, 2

학 번: 2022202064

성 명: 최봉규

## 1. 제목 및 목적

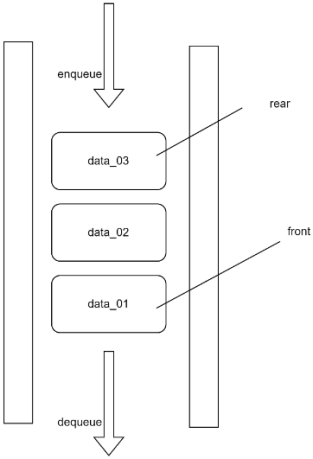
### A. 제목

Synchronous FIFO

### B. 목적

이번 실습에서는 synchronous FIFO를 구현한다. 저번 실습시간에 조사했던 queue를 토대로 이를 구현하는 실습을 진행하며, 4개의 handshake signal(wr\_ack, wr\_err, rd\_ack, rd\_err)을 write와 read요청에 대해 feedback으로 제공한다. 또한 지난 번 실습에 사용했던 register\_file을 이용하여 data를 저장하고 출력하는 것을 실습한다.

## 2. 원리(배경지식)

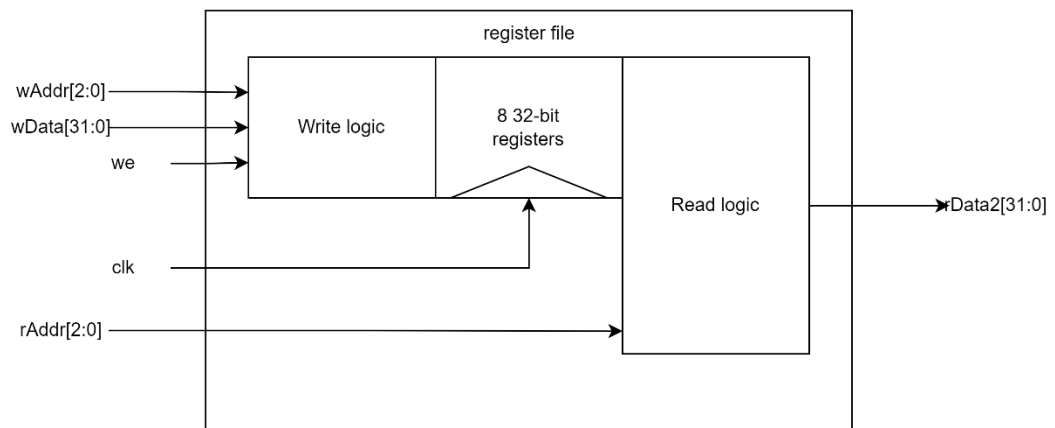
Queue
<p>Queue는 먼저 집어넣은 data가 먼저 빠져나오게 되는 자료구조이다. Stack과 마찬가지로 data는 순서대로 쌓인다. 하지만 stack과 다른 삭제 연산을 시행한다. 가장 먼저 삽입된 data가 가장 먼저 삭제되는 FIFO(first in first out)의 구조를 갖는다. Queue에 대한 예시로는 대기 줄 같은 것이 있다. Queue는 insert하는 것을 enqueue, delete하는 것을 dequeue라는 연산을 갖는다. queue에는 두 가지의 위치를 갖는다. Head의 front, tail의 rear가 그 역할이다. Enqueue 연산을 할 경우에는 rear에서 연산을 수행하고 rear를 한 칸 옮긴다. Dequeue 연산의 경우는 front에서 연산을 수행하고 front를 한 칸 옮기게 된다.</p> <p>아래의 이미지는 queue에 대한 이미지이다. 이 이미지로 예시를 들면, enqueue는 data_01, data_02, data_03의 순서로 진행된다. Dequeue 역시 이와 같은 순서인 data_01, data_02, data_03의 순서로 진행된다. 이때 enqueue가 진행되고 dequeue를 하기 전 front는 data_01의 위치를 rear는 data_03의 위치를 갖는다.</p>


fifo
<p>이번 실습으로 진행하는 fifo는 read &amp; write pointer를 관리하는 구조로 status flags를 생성하고 handshake signals를 사용자에게 제공한다.</p>

## Register\_File

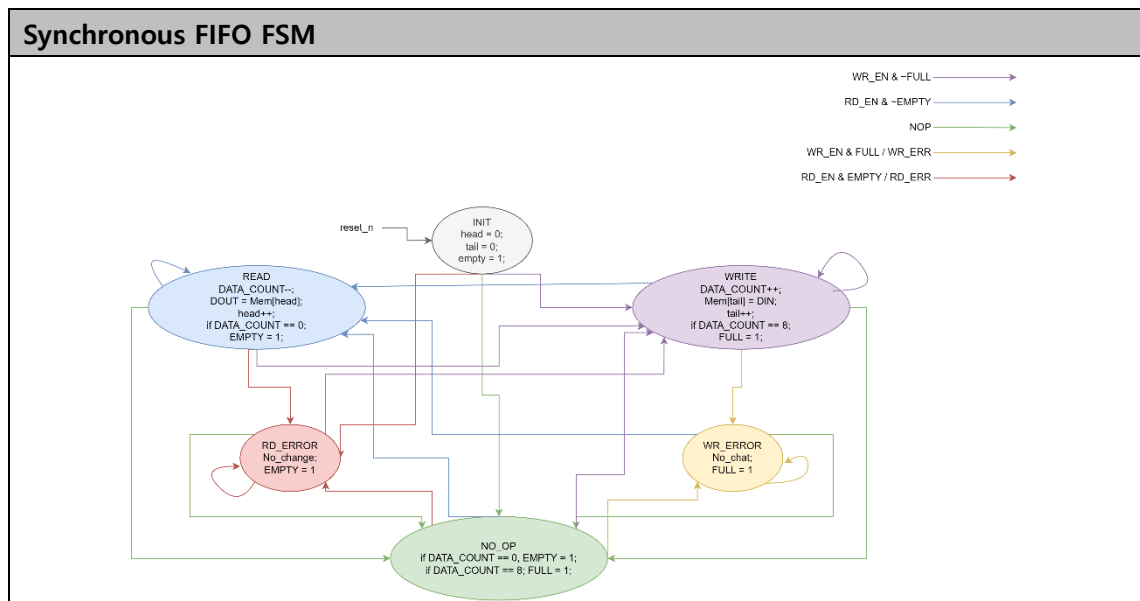
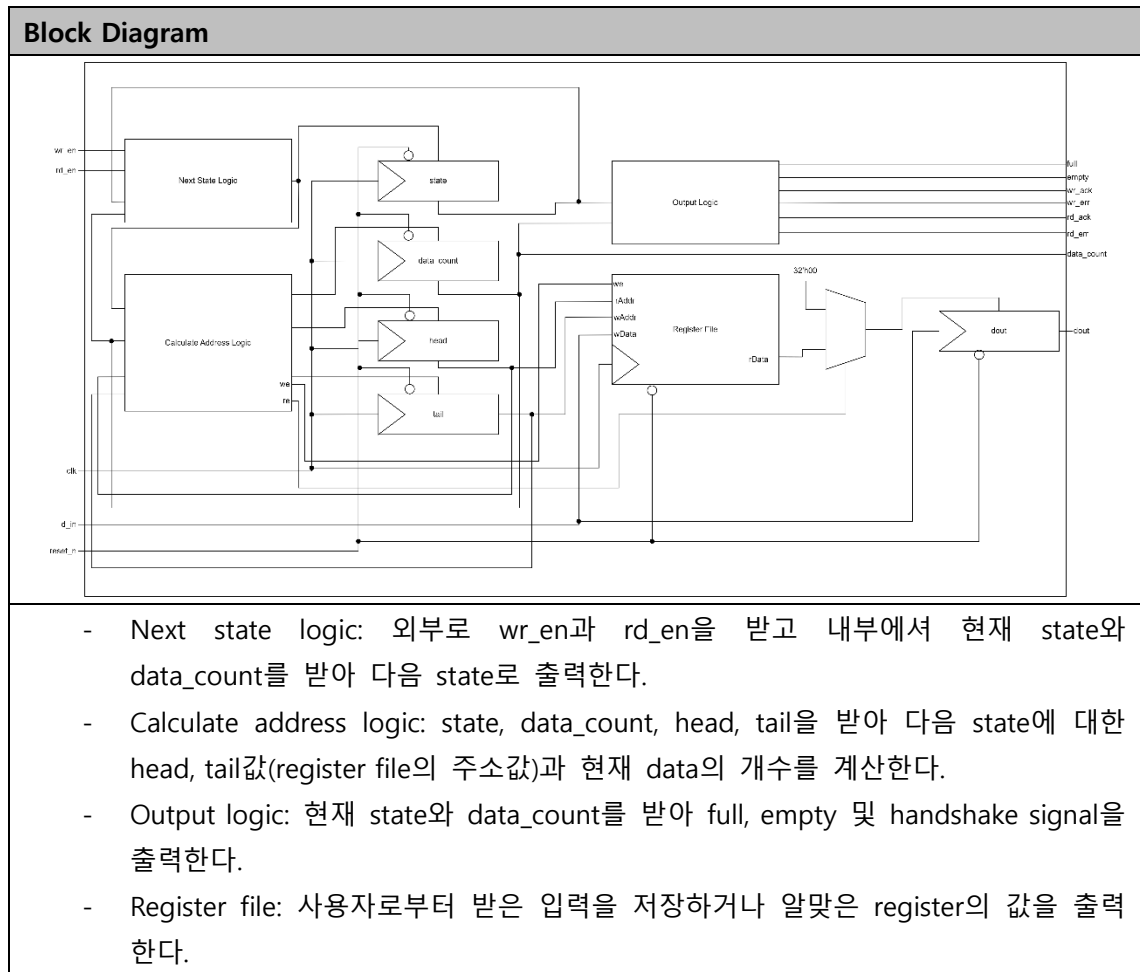
- 32bit register가 8개 instance 된다.
- Write는 write enable(we)에 의해 활성화된다.
- Write operation  
Decoder를 통해 address를 해석하여 해당 register enable
- Read operation  
MUX를 통해 8개의 register 중 한 개를 선택한다.

다음은 Register File의 structural specification이다.



- Write logic은 사용자로부터 wAddr를 받아 8개의 register 중 한 개를 선택한 뒤 we이 1일 때 register값에 저장한다.
- 8 개의 32bit register 중 write logic에 의해 선택되어진 register는 사용자로부터 받은 data를 저장한다.
- Read logic은 사용자로부터 rAddr를 받아 8개의 register 중 한 개를 선택하여 값을 출력한다.
- 해당 실습에서는 re값을 받기 때문에 rData의 값을 32'h0000\_0000과 mux를 통해 read enable신호를 sel값으로 받아 d\_out으로 출력한다.

### 3. 설계 세부사항



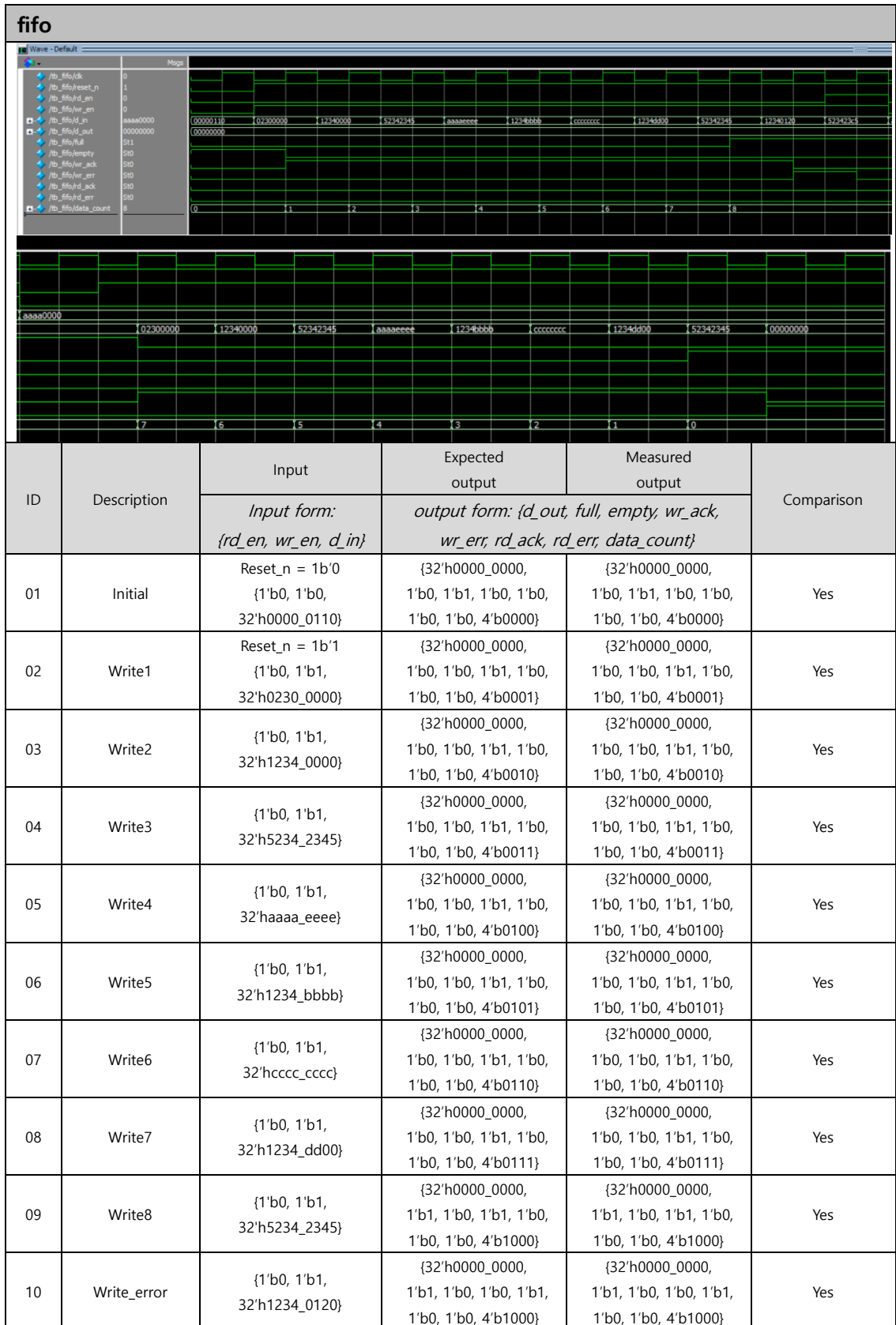
State operations	
<i>State</i>	<i>Operations</i>
INIT	Head = tail = DATA_COUNT = 0;
NO_OP	No change
WRITE	Mem[tail] = DIN; tail++; DATA_COUNT++; If DATA_COUNT == 8; FULL = 1;
WR_ERROR	No change; FULL = 1;
READ	DOUNT = mem[head]; head++; DATA_COUNT--; If DATA_COUNT == 0; EMPTY = 1;
RD_ERROR	No change; EMPTY = 1;

State encoding	
<i>State</i>	<i>Encoding</i>
INIT	3'b000
NO_OP	3'b001
WRITE	3'b011
WR_ERROR	3'b010
READ	3'b110
RD_ERROR	3'b111

Output Design – Full & Empty		
<i>Data_count</i>	<i>Full</i>	<i>Empty</i>
0	0	1
8	1	0
1 ~ 7	0	0

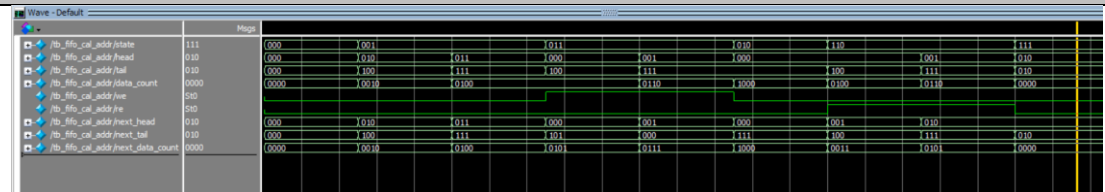
#### 4. 설계 검증 및 실험 결과

##### A. 시뮬레이션 결과



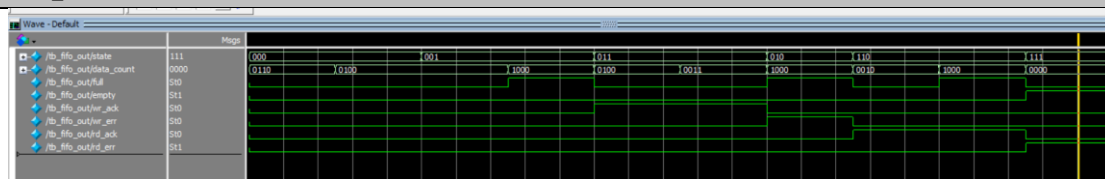
11	No_op1	{1'b1, 1'b1, 32'h5234_23c5}	{32'h0000_0000, 1'b1, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 4'b1000}	{32'h0000_0000, 1'b1, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 4'b1000}	yes
12	No_op2	{1'b0, 1'b0, 32'haaaa_0000}	{32'h0230_0000, 1'b1, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 4'b1000}	{32'h0230_0000, 1'b1, 1'b0, 1'b0, 1'b0, 1'b0, 1'b0, 4'b1000}	Yes
13	Read1	{1'b1, 1'b0, 32'haaaa_0000}	{32'h1234_0000, 1'b0, 1'b0, 1'b0, 1'b0, 1'b1, 1'b0, 4'b0111}	{32'h1234_0000, 1'b0, 1'b0, 1'b0, 1'b0, 1'b1, 1'b0, 4'b0111}	Yes
14	Read2	{1'b1, 1'b0, 32'haaaa_0000}	{32'h5234_2345, 1'b0, 1'b0, 1'b0, 1'b0, 1'b1, 1'b0, 4'b0110}	{32'h5234_2345, 1'b0, 1'b0, 1'b0, 1'b0, 1'b1, 1'b0, 4'b0110}	yes
15	Read3	{1'b1, 1'b0, 32'haaaa_0000}	{32'haaaa_eeee, 1'b0, 1'b0, 1'b0, 1'b0, 1'b1, 1'b0, 4'b0101}	{32'haaaa_eeee, 1'b0, 1'b0, 1'b0, 1'b0, 1'b1, 1'b0, 4'b0101}	Yes
16	Read4	{1'b1, 1'b0, 32'haaaa_0000}	{32'hcccc_cccc, 1'b0, 1'b0, 1'b0, 1'b0, 1'b1, 1'b0, 4'b0100}	{32'hcccc_cccc, 1'b0, 1'b0, 1'b0, 1'b0, 1'b1, 1'b0, 4'b0100}	Yes
17	Read5	{1'b1, 1'b0, 32'haaaa_0000}	{32'h1234_dd00, 1'b0, 1'b0, 1'b0, 1'b0, 1'b1, 1'b0, 4'b0011}	{32'h1234_dd00, 1'b0, 1'b0, 1'b0, 1'b0, 1'b1, 1'b0, 4'b0011}	Yes
18	Read6	{1'b1, 1'b0, 32'haaaa_0000}	{32'h5234_2345, 1'b0, 1'b0, 1'b0, 1'b0, 1'b1, 1'b0, 4'b0010}	{32'h5234_2345, 1'b0, 1'b0, 1'b0, 1'b0, 1'b1, 1'b0, 4'b0010}	Yes
19	Read7	{1'b1, 1'b0, 32'haaaa_0000}	{32'h1234_0120, 1'b0, 1'b0, 1'b0, 1'b0, 1'b1, 1'b0, 4'b0001}	{32'h1234_0120, 1'b0, 1'b0, 1'b0, 1'b0, 1'b1, 1'b0, 4'b0001}	Yes
20	Read8	{1'b1, 1'b0, 32'haaaa_0000}	{32'h5234_2345, 1'b0, 1'b1, 1'b0, 1'b0, 1'b1, 1'b0, 4'b0000}	{32'h5234_2345, 1'b0, 1'b1, 1'b0, 1'b0, 1'b1, 1'b0, 4'b0000}	Yes
21	Read_error	{1'b1, 1'b0, 32'haaaa_0000}	{32'h0000_0000, 1'b0, 1'b1, 1'b0, 1'b0, 1'b0, 1'b1, 4'b0000}	{32'h0000_0000, 1'b0, 1'b1, 1'b0, 1'b0, 1'b0, 1'b1, 4'b0000}	Yes

### fifo\_cal\_addr



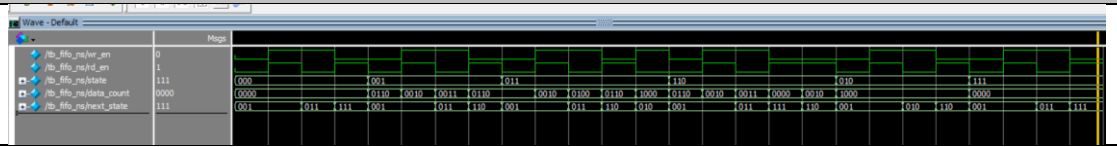
we값이 1'b1일 때는 next\_tail의 값이 1증가, re값이 1'b1일 때는 next\_head의 값이 1증가하는 것을 확인할 수 있다. Data\_count는 we = 1'b1일 때 1 증가, re = 1'b1일 때 1감소하는 것도 확인할 수 있다.

### fifo\_out

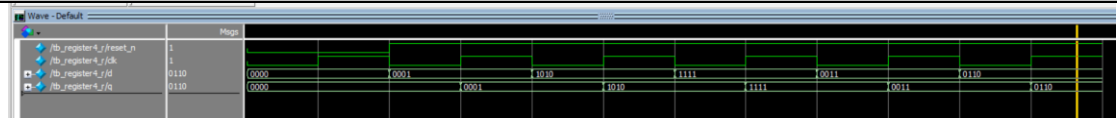


각 state와 data\_count에 대해서 full, empty, wr\_ack, wr\_err, rd\_ack, rd\_err 값이 정해지는 output state에 대한 정의를 담고 있는 module이다. Data\_count가 4'b0000이면 empty =

**fifo\_ns**

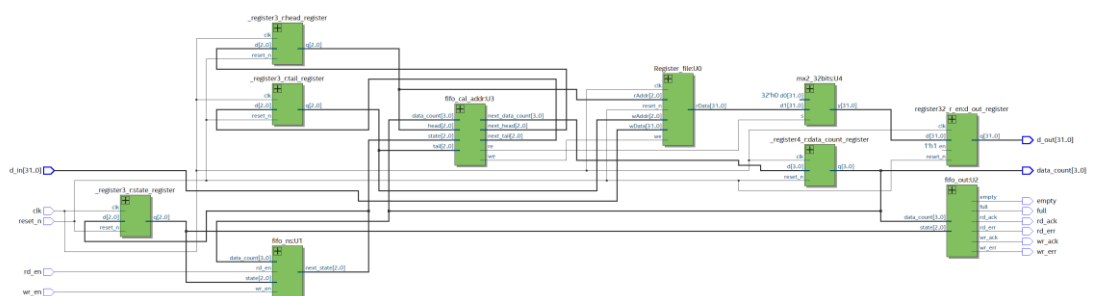


\_register4\_r



## B. 합성(synthesis) 결과

**fifo**



해당 그림은 fifo의 RTL map으로 3bits resettable register 3개, 4bits resettable register, fifo\_cal\_addr, fifo\_ns, Register\_file, mx2\_32bits, fifo\_out, register32\_r\_en으로 구성된 것을 확인할 수 있다. register32\_r\_en의 사용용도는 단순한 resettable register이나 새로운 submodule을 사용하여 테스트 벤치를 하는 것보다 en 값을 1으로 고정하면 같은 결과가 나올 것을 생각하여 구상한 레지스터이다.



Flow Summary	
<<Filter>>	
Flow Status	Successful - Sun Oct 29 07:55:57 2023
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	fifo
Top-level Entity Name	fifo
Family	Cyclone V
Device	5CSXFC6D6F31C6
Timing Models	Final
Logic utilization (in ALMs)	161 / 41,910 (< 1 %)
Total registers	301
Total pins	78 / 499 (16 %)
Total virtual pins	0
Total block memory bits	0 / 5,662,720 (0 %)
Total DSP Blocks	0 / 112 (0 %)
Total HSSI RX PCSs	0 / 9 (0 %)
Total HSSI PMA RX Deserializers	0 / 9 (0 %)
Total HSSI TX PCSs	0 / 9 (0 %)
Total HSSI PMA TX Serializers	0 / 9 (0 %)
Total PLLs	0 / 15 (0 %)
Total DLLs	0 / 4 (0 %)

해당 flow summary는 top module을 보았을 때 fifo의 flow summary로 flow status를 보았을 때, Successful을 확인할 수 있다. 또한 Logic utilization은 161, Total register는 301개, Total pins는 78임을 확인할 수 있다.

## 5. 고찰 및 결론

### A. 고찰

해당 실습을 진행하면서 submodule에 대해 뭔가 제대로 알려진 것이 없는 것 같아 이전 실습에 비해 구현이 조금 힘들었던 것 같다. Submodule을 자유롭게 구현할 수 있다는 점이 이전에는 정형화된 것을 구현하다 보니 조금 당황스러웠다. 큰 문제없이, 해당 실습을 마칠 수 있었다. 최종적으로 fifo의 test bench를 작성하고 wave form을 보던 중 이미 값이 we에 의해 썼는데, data\_count가 조금 늦게 움직이는 것을 확인했다. Submodule의 test bench는 이미 작성하고 확인한 상태로 top module인 fifo의 wire에 대해 state와 next\_state에 대해 조금 조정해주니 원하는 값대로 나오는 것을 확인할 수 있었다.

### B. 결론

해당 실습을 진행하면서 queue의 fifo형태를 제작할 수 있었다. Stack에 대해서도 lifo형식을 head와 tail의 값만 수정하면 구현 가능하다고 생각한다. Register file을 이전 실습에서 구현했던 register file이 이번 실습에서 submodule로 들어가고 이를 제외하고도 이전에 사용했던 코드들이 submodule로 들어갔다. 하나의 submodule이 이상하기만 하더라도 전체에 대한 설계가 틀어질 수 있음을 알게 됐으며, 앞으로 구현하는 모든 모듈에 대해서도 테스트 벤치를 통한 철저한 검증이 필요하다고 생각이 들었다.

## 6. 참고문헌

심동규 교수님/객체지향프로그래밍설계/광운대학교(컴퓨터정보공학부)/2023

이기훈 교수님/데이터구조설계/광운대학교(컴퓨터정보공학부)/2023

이준환 교수님/디지털논리회로2/광운대학교(컴퓨터정보공학부)/2023

이준환 교수님/컴퓨터공학기초실험2/광운대학교(컴퓨터정보공학부)/2023