컴퓨터 공학 기초 실험2 보고서

실험제목: Synchronous FIFO

실험일자: 2022년 11월 02일 (수)

제출일자: 2022년 11월 08일 (화)

학 과:컴퓨터정보공학부

담당교수: 공진흥 교수님

실습분반: 수요일 0, 1, 2

학 번: 2019202021

성 명: 정 성 엽

1. 제목 및 목적

A. 제목

Synchronous FIFO

B. 목적

Queue에 대해 이해하고 FIFO(First In First Out)을 Verilog를 통해 직접 구현한다. 이때 구조를 FSM을 통해 control logic을 작성하고 read와 write를 진행하고 status flags와 handshake signals을 출력한다.

2. 원리(배경지식)

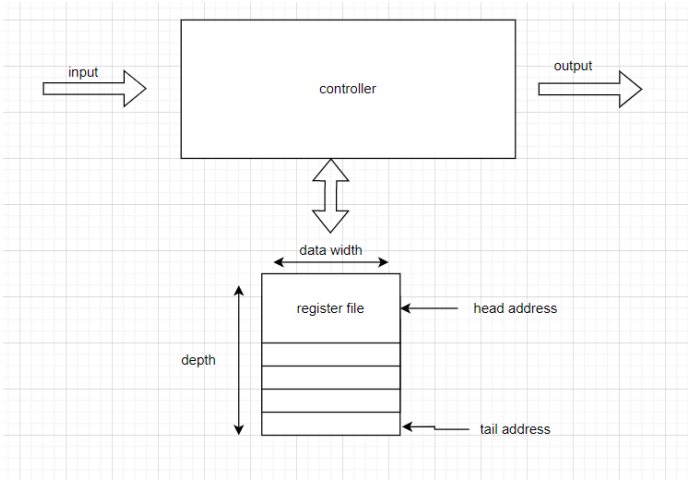
A. Queue

Queue는 데이터를 first in, first out 형식으로 저장하는 자료구조의 일종으로 먼저 들어간 데이터가 가장 먼저 출력된다. 데이터를 순서대로 queue에 저장하고 가장 먼저 삽입한 데이터를 head, 가장 나중에 삽입한 데이터를 tail로 잡아 가장 먼저 저장된 데이터를 불러와 삭제하거나, 제일 처음 데이터를 삽입하는 명령어가 있다. 이는 이번주 실험 FIFO에서 다뤘다. 해당 구조를 그림으로 표현하면 아래와 같다.

스택과 마찬가지로 크기에 제한이 있다면 배열을 통해 쉽게 구현이 가능하고 여기서 배열의 구성은 register과 동일하다고 볼 수 있으며 FSM으로 state를 구현한 controller를 통해 queue의 동작을 논리회로를 통해서 구현할 수 있다.

FIRST IN, FISRT OUT

B. FIFO FSM



FIFO (First In First out)은 위의 그림과 같이 controller를 통해서 register에 저장된 값이 queue와 같이 동작하도록 하는 FSM이다. 이때 head와 tail은 맨 처음으로 저장된 값과 제일 나중에 저장된 값의 각 register file의 주소이고, FSM state에 따라 커지거나 작아진다.

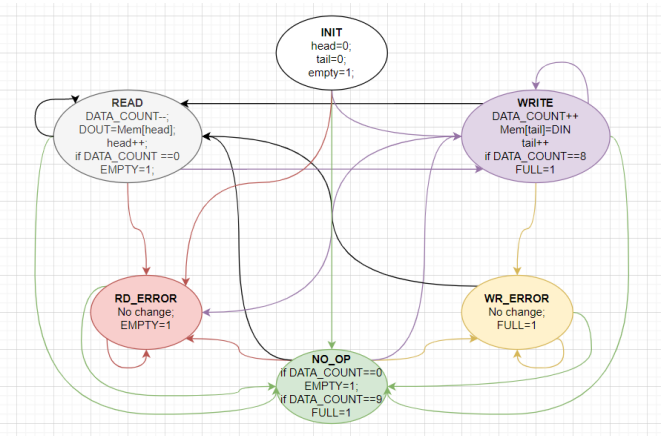
따라서 queue에 정보를 입력하는 경우 tail의 register file의 주소에 값을 write하고, 만약 queue에서 정보를 출력하는 경우 head의 register file의 주소값을 read한다. 또한 register file의 용량이 제한되어 있는 경우 제한된 만큼의 용량을 넘으면 데이터 삽입 시 에러가 발생하고, 아무것도 write 되지 않고 register file을 read하는 경우에도 에러가 발생한다.

이번 실험에서 FIFO의 FSM은 6개의 구조로 INIT, WRITE, WR_ERROR, READ, RD_ERROR, NO_OP으로 구성된다. 해당 FSM의 연결관계는 아래 설계 세부사항에서 더 설명하도록 한다.

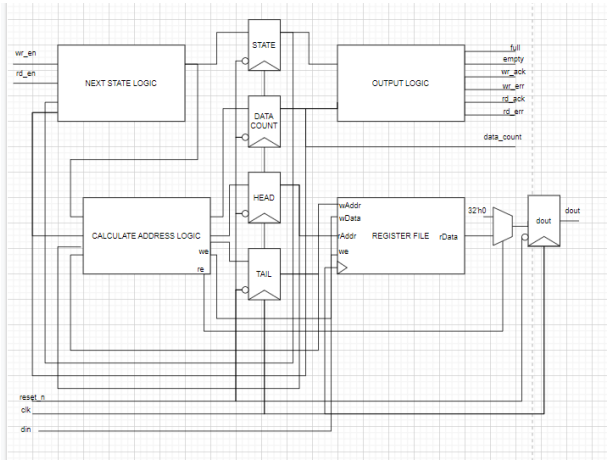
3. 설계 세부사항

Binary encoding	
INIT	000
WRITE	001
WR_ERROR	010
READ	011
RD_ERROR	100
NO_OP	101

Color arrow	
WR_EN & ~FULL	purple
RD_EN & ~EMPTY	black
nop	green
WR_EN & FULL/WR_ERR	yellow
RD_EN & EMPTY / RD_ERR	red



각 state의 3bit encoding 그리고 FSM에서의 관계는 위의 표 그리고 그림과 같다.



전체적인 FIFO 구조로 Moore FSM 파트와 count/head/tail을 계산하는 calculate address logic 파트, 값을 불러오거나 저장하는 register file과 dout 파트, Next 값을 저장

하는 d-flipflop 파트로 나뉘어져 있다.

next state logic에서는 wr_en 그리고 rd_en의 값을 입력받고 현재 저장된 data 수를 저장하는 data_count와 현재 state를 통해 FSM의 next state를 결정한다.

calculate address logic에서는 다음 state와 현재의 head, tail 그리고 data_count 값을 입력 받아 다음 head, tail, data_count를 각 Dflipflop에 저장한다. 또한 Register File에 값을 입력할지 출력할지 we와 re를 통해 결정한다.

output logic에서는 현재 data_count를 입력 받아 register file이 full인지 empty인지 출력한다. 또한 입력과 출력이 잘 되었는지 wr_ack, wr_err, rd_ack, rd_err을 출력한다.

register file에서는 실제 FIFO 값을 저장하고, 값을 입력할 주소는 tail address로 읽을 주소는 head address를 가져온다.

32bit MUX2에서는 rData에 출력된 값을 Calculate address logic의 re의 값에 따라 0 또는 1을 출력하여 Dflipflop에 넘기고 Dflipflop은 clk에 동기화하여 dout을 출력한다.

그리고 state, data_count, head, tail에 대한 Dflipflop에서는 다음 state, data_count, head, tail을 저장한다.

구분	이름	설명
Top module	fifo	FIFO의 top module
Sub module	fifo_ns	fifo의 next state를 정하는 logic module
	fifo_cal_addr	fifo의 head, tail, datacount에 대한 연산 logic module
	fifo_out	Output logic을 다루는 module
	register file	Register file module

Module	구분	이름	비트 수	설명
fifo	input	clk	1 bit	clock
		reset_n	1 bit	active low에 동작, reset신호 값이 인가되면 register값 0 초기화
		rd_en	1 bit	read enable
		wr_en	1 bit	write enable
		d_in	32 bits	data in
	output	d_out	32 bits	data out
		full	1 bit	data full
		empty	1 bit	data empty
		wr_ack	1 bit	write acknowledge
		wr_err	1 bit	write error
		rd_ack	1 bit	read acknowledge
		rd_err	1 bit	read error
		data_count	4 bits	data count vector
fifo_ns	input	wr_en	1 bit	write enable
		rd_en	1 bit	read enable
		state	3 bits	Current state
		data_count	4 bits	Data count vector
	output	next_state	3 bits	Next state

fifo_cal_addr	input	staete	3 bits	current state
		head	3 bits	current head pointer
		tail	3 bits	current tail pointer
		data_count	4 bits	current data count vector
	output	we	1 bit	register file write enable
		re	1 bit	register file read enable
		next_head	3 bits	next head pointer
		next_tail	3 bits	next tail pointer
		next_data_count	4 bits	next data count vector
fifo_out	input	state	3 bits	current state
		data_count	4 bits	current data count
	output	full	1 bit	data full
		empty	1 bit	data empty
		wr_ack	1 bit	write acknowledge
		wr_err	1 bit	write error
		rd_ack	1 bit	read acknowledge
		rd_err	1 bit	read error

4. 설계 검증 및 실험 결과

A. 시뮬레이션 결과

i. fifo

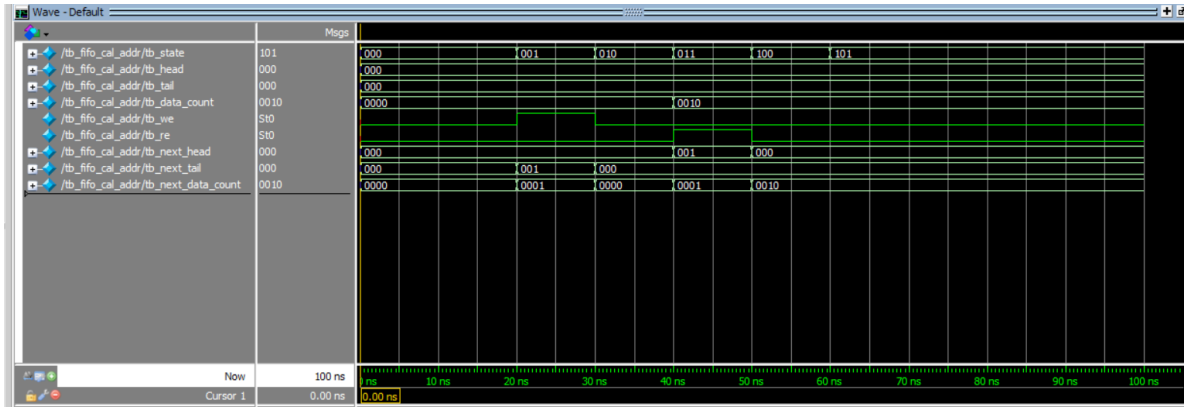
reset_n을 통해 register file과 모든 d-flipflop을 초기화하고 wr_en을 set한 상태에서 0000_0001, 0000_0020 ....8000_0000까지 입력을 진행할때는 정상적으로 진행하고 마지막에 full이 set이 되었고 FFFF_FFFF가 입력되려고 할 때 wr_err이 set이 된 것을 확인할 수 있다. 또한 rd_en

을 set한 상태에서 0000_0001부터 출력됨을 확인할 수 있고, 새로 CCCC_CCCC가 입력되었어도 앞의 head부터 출력되는 것을 확인할 수 있다.

ii. fifo_ns

현재 state와 wr_en, rd_en의 값에 따라 다음 state를 결정하게 되는 것을 waveform을 통해 확인할 수 이다. INIT, WRITE ,WR_ERROR, READ, RD_ERROR, NO_OP에서 각각 2가지 케이스를 확인하였다.

iii. fifo_cal_addr



현재 head, tail, data_count의 값이 현재의 state에 따라 next_head, next tail, next_data_count가 변화한 것을 볼 수 있다. WRITE에서는 data_count가 증가하고 READ에서는 data_count가 감소한다. WRITE에서는 we가 set되고 READ에서는 re가 set 된다.

iv. fifo_out

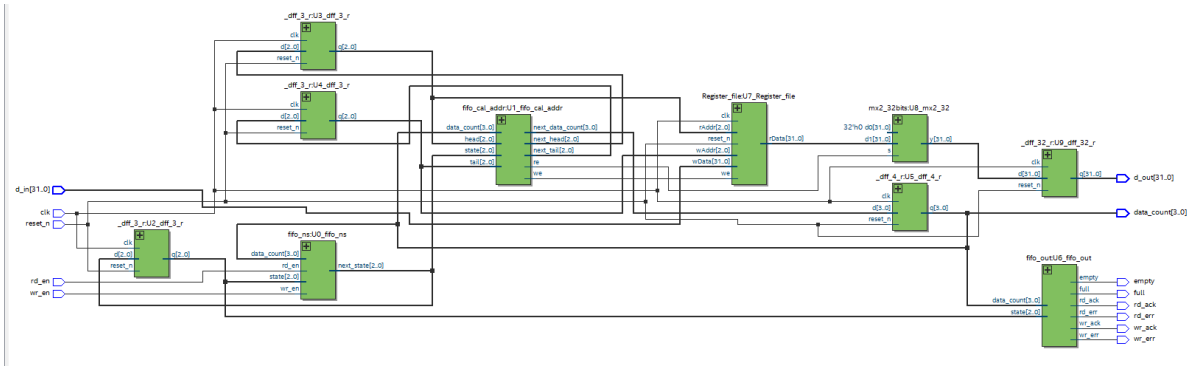
fifo_out module에서 data_count에 따른 empty, full을 확인하고 각 state 별로 wr_ack, wr_err,

rd_ack, rd_err 값이 변화하여 set 되는 것을 위의 testbench의 waveform을 통해 알 수 있다.

B. 합성(synthesis) 결과

i. fifo

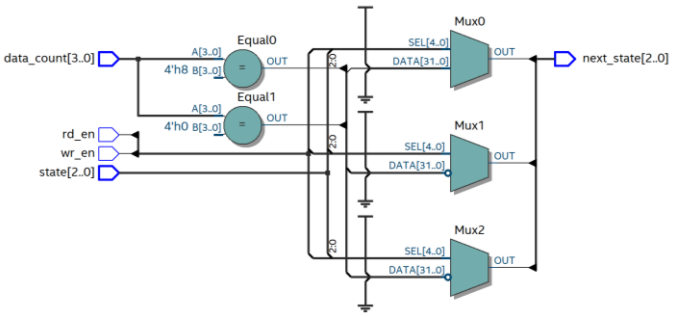
Flow Summary	
Flow Status	Successful - Tue Nov 08 22:37:31 2022
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	fifo
Top-level Entity Name	fifo
Family	Cyclone V
Device	5CSXFC6D6F31C6
Timing Models	Final
Logic utilization (in ALMs)	150 / 41,910 (< 1 %)
Total registers	301
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 %)



Register file 그리고 dff_3_r 그리고 dff_4_r 그리고 dff_32_r 모두 clk에 동기화 되어 작동하는 logic이며 reset_n이 active low일 때 모두 초기화 된다. 새로운 module block의 세부사항은 아래서 확인한다.

ii. fifo_ns

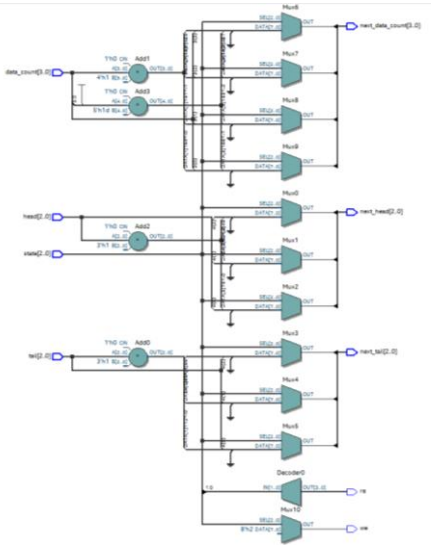
Flow Summary	
Flow Status	Successful - Tue Nov 08 22:28:20 2022
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	fifo
Top-level Entity Name	fifo_ns
Family	Cyclone V
Device	5CSXFC6D6F31C6
Timing Models	Final
Logic utilization (in ALMs)	5 / 41,910 (< 1 %)
Total registers	0
Total pins	12 / 499 (2 %)
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 %)



state, re_en, wr_en의 값에 따라 각 case를 두어 다음 state를 결정하는 logic이다. READ state나 WRITE state가 정상 작동할 때 data_count가 empty이거나 full인 경우를 따져서 다음 state를 결정한다.

iii. fifo_cal_addr

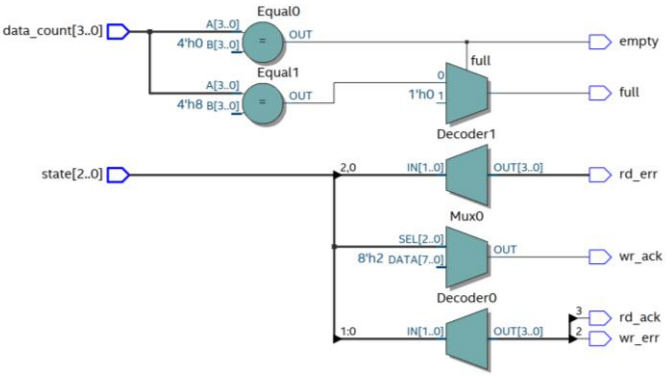
Flow Summary	
<<Filter>>	
Flow Status	Successful - Tue Nov 08 22:19:56 2022
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	fifo
Top-level Entity Name	fifo_cal_addr
Family	Cyclone V
Device	5CSXFC6D6F31C6
Timing Models	Final
Logic utilization (in ALMs)	9 / 41,910 (< 1 %)
Total registers	0
Total pins	25 / 499 (5 %)
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 %)



data_count , head, tail 값들이 state에 따라 다음 data_count, head, tail의 값이 결정되는 logic이다. READ 때는 re가 set되고 WRITE 에서는 we가 set 된다.

iv. fifo_out

Flow Summary	
<<Filter>>	
Flow Status	Successful - Tue Nov 08 20:26:46 2022
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	fifo
Top-level Entity Name	fifo_out
Family	Cyclone V
Device	5CSXFC6D6F31C6
Timing Models	Final
Logic utilization (in ALMs)	4 / 41,910 (< 1 %)
Total registers	0
Total pins	13 / 499 (3 %)
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 %)



data_count에 따라 empty 또는 full이 결정되는 logic과 state에 따라 re_err, wr_ack, rd_ack, wr_err 이 결정되는 logic을 보이고 있다.

5. 고찰 및 결론

A. 고찰

이번 실험에서는 FIFO를 구현하였으며, 이를 FSM 구조로 설계하여서 제작하였다. 이전에 사용한 register file 그리고 mux 등 전에 사용한 module을 다시 사용했으며 bit에 따른 사용이 다른 Dflipflop 새로 구현하여 사용하였다. 이번엔 코드에 대한 예시가 없이 제작하는 것이기 때문에 기본적인 틀을 잡는데 어려움을 겪었다.

다만 FSM의 관계를 ppt에서 자세히 설명해주었고 testbench 결과를 미리 보여주며 실험이 진행되어야 할 방향을 알려줘서 구현을 할 수 있었다. 또한 FSM에서 if else 문이 아닌 case 문을 써서 각 encoding 별로 진행될 수 있도록 하고, wr_en이나 rd_en의 조합

에서 예외가 발생하는 경우 NO_OP state로 이동하도록 하였다.

fifo_ns에서 WRITE와 READ에서는 정상적으로 작동할 때, DATA_COUNT가 full인지 empty인지에 따라 다르게 진행하도록 if 문을 통해 구현하였다.

B. 결론

이번 실험을 통해 FIFO르 구현하면서 FSM을 이용하여 controller를 제작하면서 약간 함수 또는 어셈블리의 branch를 만들어 사용하는 느낌을 받았다. 이번 실험처럼 각 상관관계를 이해하면 약간 수정하여 Last In First Out으로 stack 구조를 구현할 수 있을 것으로 본다.

그리고 verilog의 always@문의 활용을 submodule에서 적극 활용하다 보니 더욱 익숙해졌고 다음에도 always@문을 사용해야하는 실험에서 적절히 잘 사용할 수 있을 것이란 자신감을 가지게 되었다.

6. 참고문헌

1) 공진흥 교수님/ 컴퓨터정보공학실험2/ 광운대학교 컴퓨터정보공학부/ 2022
2) 공영호 교수님/ 디지털논리회로2 FIFO, FSM design/ 광운대학교 컴퓨터정보공학부/2022