컴퓨터 공학 기초 실험2 보고서

실험제목: Synchronous FIFO

실험일자: 2020년 10월 23일 (금)

제출일자: 2020년 10월 28일 (수)

학 과: 컴퓨터공학과

담당교수: 이준환 교수님

실습분반: 금요일 5,6,7

학 번: 2019202009

성 명: 서여지

1. 제목 및 목적

A. 제목

Synchronous FIFO

B. 목적

지난 실험에서 설계한 register file을 이용하여 FIFO를 작성하는 것을 목표로 한다. FIFO의 구조와 동작에 대해 이해한다. 직접 설계한 FIFO에 적절한 Testbench를 작성하여 FIFO의 동작과 flag 출력을 확인하고 결과를 분석한다.

2. 원리(배경지식)

1) stack과 queue에 대해 조사한다.

1. stack

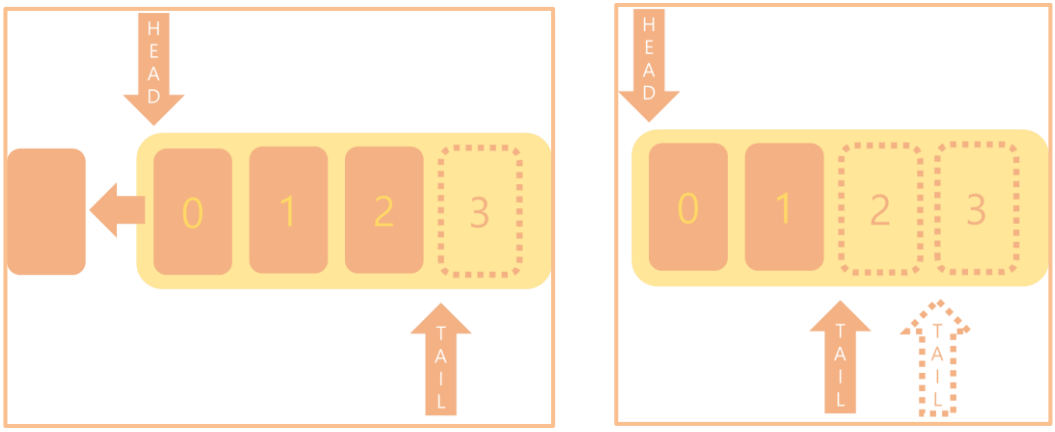
stack은 그 단어의 뜻이 의미하는 바와 같이 데이터를 바닥에서부터 쌓아 올리는 것과 유사하게 처리할 수 있는 구조이다. stack에 데이터가 들어올 때 아래에서 위로 탑을 쌓아 올리듯이 저장되었다면, 반대로 stack에서 데이터를 출력할 때는 위에서 아래방향으로 데이터를 하나씩 출력하게 된다. 결국 첫 번째로 stack에 삽입된 정보는 마지막으로 stack에서 출력되게 된다. 반대로 마지막으로 삽입된 정보가 가장 처음으로 출력되는 구조가 stack구조이다. stack에 가장 처음 삽입되는 정보의 주소를 head라 하고, 마지막에 삽입된 정보를 top이라 한다.

2. queue

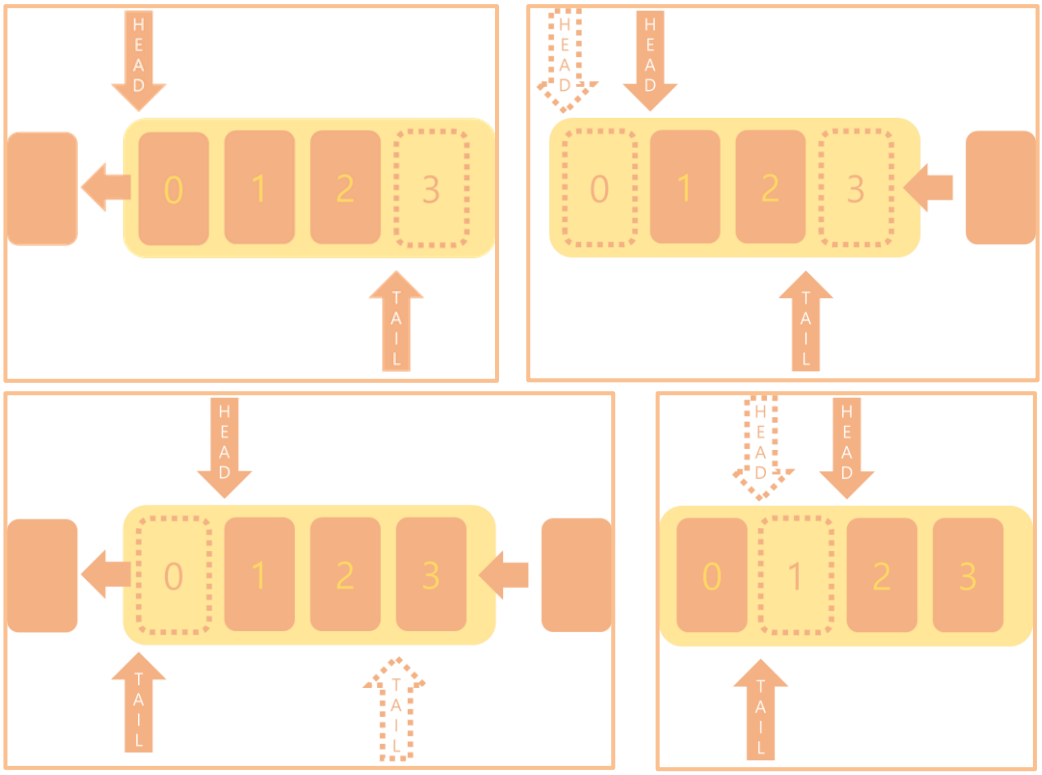
queue는 stack과 달리 아래에서 위로 쌓인 데이터가 출력될 때도 아래에서 위 방향으로 출력되는 구조이다. 첫 번째로 queue에 삽입된 정보는 첫 번째로 출력되고, 마지막으로 삽입된 정보는 마지막으로 출력된다. 이러한 구조를 First In, First Out을 줄인 FIFO 혹은 선입선출이라 부르기도 한다.

2) FIFO의 순환

FIFO에서 head pointer는 다음에 출력되는 data의 주소값이고, tail은 다음에 입력되는 data가 저장되는 위치의 주소값이다. FIFO의 head를 첫 번째 register로 고정한다면 FIFO에 저장된 data의 개수에 따라 tail의 값만 변화하고, 출력은 항상 첫 번째 data가 출력된다. 이후 첫 번째 data가 저장되어있던 register의 내용이 비게 되기 때문에 뒤의 모든 data를 한 칸씩 옮겨 재정렬하는 과정이 필요하다.



이때 마지막 register의 뒤에 첫 번째 register가 연결되어 순환하는 형태로 modeling 하면 위와 같이 register를 재정렬하는 과정을 생략할 수 있다. 이번 실험에서 설계하는 FIFO는 다음과 같이 순환하는 형태로 설계하였다.



3. 설계 세부사항

이번 실험에서 구현하는 FIFO는 write, read 동작을 하고, FIFO에 삽입된 data의 수인 Data_count와, 이것을 통해 FIFO가 비었을 때와 가득 찼을 때를 나타내는 것에 Empty flag와 Full flag를 이용한다. write와 read의 결과에 따라 각 명령이 정상적으로 처리되었음을 나타낼 때 wr_ack, rd_ack signal을 이용하고, 반대로 명령의 실행 중 오류가 발생하였음을 나타내는 것에 wr_err, rd_err을 사용하여 결과를 출력한다. full과 empty signal은

Data_count의 값을 이용하여 구할 수 있다. 실험에서 구현할 FIFO는 32bit의 data가 8개 저장될 수 있는 register file을 이용하므로, Data_count가 8일 때 full이고, 0일 때 empty이다. 위의 동작을 실행하기 위해 FIFO는 clk, reset_n, rd_en, wr_en, d_in신호를 입력받고, d_out, full, empty, wr_ack, wr_err, rd_ack, rd_err, data_count를 출력한다.

FIFO는 크게 5부분으로 나누어져 구성된다. FIFO의 state, data count, head, tail을 저장하는 register와, FIFO에 write된 data를 저장하는 Register_file module, next state를 계산하는 fifo_ns module, output을 계산하는 fifo_out module외에도 주소값을 계산하는 fifo_cal module을 설계하여 사용한다.

FIFO는 INIT, NO_OP, READ, RE_ERROR, WRITE, WR_ERROR로 총 6개의 state를 갖는다. 6가지 이상의 state를 표현할 수 있는 3bit로 binary encoding하면 다음과 같다.

state	Encoding	state	Encoding
INIT	000	RE_ERROR	011
NO_OP	001	WRITE	100
READ	010	WR_ERROR	101

1) fifo_ns

wr_en과 rd_en, data_count 신호가 변경될 때 마다 next state를 계산한다. next state는 current state의 영향을 받지 않는다. re_en은 wr_en에 우선하고, data_count값이 적절하지 않은 경우 RD_ERROR이나 WR_ERROR이 next state가 된다. data_count가 적절할 때 READ, WRITE가 next state가 되고, wr_en과 rd_en이 모두 0인 경우 NO_OP가 next state가 된다.

2) fifo_cal_addr

state, data_count, head, tail값이 변경될 때 마다 we, re, next_data_count, next_head, next_tail값을 계산한다. 해당 모듈의 output은 next state와 달리 current state의 영향을 받는다. INIT state에서 모든 값은 0으로 초기화 되고, NO_OP에서 we,re는 0이고 다른 값은 current 값과 동일하다. READ state에서 re=1이 되고, next_data_count는 현재의 data_count에서 1이 줄어든 값이 된다. 반면 WRITE state에서 we=1이고, next_data_count는 current data_count +1의 값이다. READ state와 WRITE state에서 head와 tail의 값을 적절히 조정하여 전달한다. RD_ERROR state와 WR_ERROR state에서는 NO_OP와 같은 결과를 갖는다.

3) fifo_out

state 또는 data_count값이 변경될 때 마다 full, empty, wr_ack, wr_err, rd_ack, rd_err 출력을 계산한다. handshack signal은 state에 따라 다르게 결정된다. INIT, NO_OP의 경우 4개의 handshack signal은 모두 0이고, READ, WRITE에서 각각 rd_ack, wr_ack가 1이 된다. 반면 RD_ERROR, WR_ERROR에서 rd_err, wr_err이 1이된다. empty와 full은

state와 관계없이 data_count의 값에 따라 결정된다. data_count가 8일 때 full, 0일 때 empty가 1이 된다.

4) Register_file

register file은 8개의 32bit data를 저장할 수 있다. 다음 read동작에서 읽을 수 있는 data의 주소값이 head이고, write동작에 의해 다음 data가 들어오는 공간의 주소값이 tail이다. 배경지식 부분에서 살펴본 것과 같이, register file을 순환하는 형태로 생각하여 이용한다.

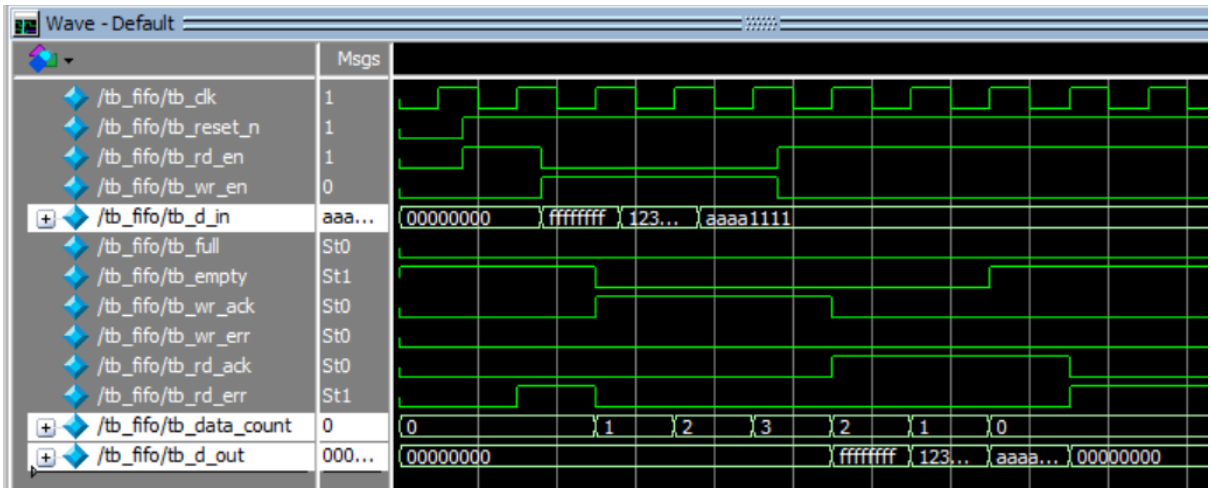
5) register

각각의 module에서 구한 값을 저장하고, clk에 동기화시켜서 출력할 수 있도록 한다.

4. 설계 검증 및 실험 결과

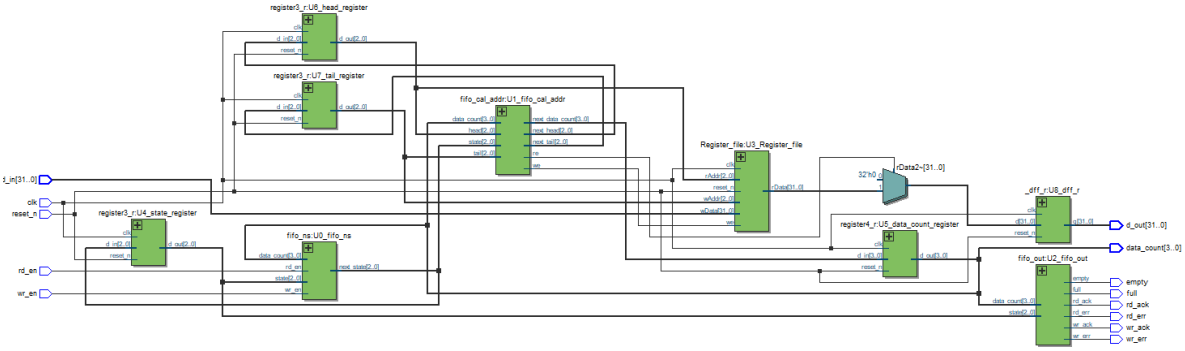
A. 시뮬레이션 결과

FIFO에 3개의 data를 write하고, read하는 testbench를 작성하여 실험하였다. data_count가 0일 때 read를 실행하면 rd_err가 1이 되는 것을 확인할 수 있고, 정상적으로 data가 write, read된 경우 data_count가 적절히 증가/감소하는 것을 볼 수 있다. 4개의 wr_ack, wr_err, rd_ack, rd_err도 정상적으로 출력된 것을 확인할 수 있다.



B. 합성(synthesis) 결과

RTL Viewer에는 module과 register가 다음과 같이 나타났다.



flow summary는 다음과 같다. pin은 총 78개 이용되었고, 사용한 register는 모두 301개인 것을 확인할 수 있다.

Compilation Report - fifo | fifo.v | fifo_cal_addr.v

Flow Summary	
Flow Status	Successful - Wed Oct 28 01:11:34 2020
Quartus Prime Version	15.1.0 Build 185 10/21/2015 SJ Lite Edition
Revision Name	fifo
Top-level Entity Name	fifo
Family	Cyclone V
Device	5CSXFC6D6F31C6
Timing Models	Final
Logic utilization (in ALMs)	147 / 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 %)

5. 고찰 및 결론

A. 고찰

프로그램을 작성하는 과정에서 input으로 입력된 state에 따라 다른 결과를 만들기 위하여 case문을 이용할 때, case문을 always문 밖에서 이용하여 컴파일 에러가 발생하였다. case문을 always문 내부에 위치시켜서 오류를 해결하였다. 또한 sensitive list의 작성 과정에서 필요한 요소를 누락시켜 프로그램이 기대한 것과 다르게 동작하기도 했다. 해당 조건을 확인하고, sensitive list에 추가하여 문제를 해결하였다.

B. 결론

이번 과제는 지난 실험에서 작성한 register file을 이용하여 FIFO를 설계하는 것이었다. 이번 실험에서 설계한 FIFO는 read, write기능뿐만 아니라 empty, full등의 flag와 4개의 handshack signal을 지원한다. 전체적으로 FSM의 설계방법을 이용하여

FIFO를 설계할 수 있었고, 구체적인 module은 Behavioral implementation 하여 구현할 수 있었다.

6. 참고문헌

이준환교수님, 디지털논리회로2 강의자료, 광운대학교 컴퓨터정보공학과,2020

이준환교수님, 컴퓨터공학기초실험2 강의자료, 광운대학교 컴퓨터정보공학과,2020