디지털논리회로 2 - Assignment 2 보고서

2021202085 전한아솔

1. 과제 설명

이번 과제는 베릴로그를 이용하여 FIFO(First in, First out)의 기능을 구현하는 과제이다.
FIFO는 선입선출의 의미로 먼저 들어간 데이터가 먼저 나오는 형태로 데이터를 처리하는 방식이다. 이는 Queue 자료구조의 동작 방식과 동일하다.

큐의 연산은 Enqueue와 Dequeue가 존재하는데, 먼저 Enqueue는 데이터가 큐에 삽입되면 새 요소가 새로운 tail이 된다. 이때 큐에 더 이상 데이터를 추가할 공간이 없다면 큐는 Full State가 된다. Dequeue 연산은 Head에 존재하는 데이터가 제거되면서 사용자에게 반환된다. 이때 큐에 데이터가 존재하지 않는다면 큐는 Empty State가 된다.

이번 과제에서 구현할 큐는 8개의 32 bits Register로 구성되며 Full이나 empty를 의미하는 Status flag와 write 성공, write 불가, read 성공, read 불가의 signal들을 의미하는 Handshake Signal이 존재한다. 또한 Count vector가 존재하여 현재 존재하는 데이터의 개수를 제공한다.

FIFO는 외부에서 wr_en과 rd_en을 받고 내부에서 현재 상태와 데이터의 개수를 받아, 다음 상태를 출력하는 Next State Logic, 현재 상태, 데이터 개수, head 주소(Read 기능의 포인터 역할), tail 주소(Write 기능의 포인터 역할)를 받아서 다음 상태에 대한 head, tail과 데이터 개수를 계산하는 Calculate Address

Logic, 현재 상태와 데이터 개수를 받아 Status flag와 Handshake signal를 출력하는 Output Logic, 사용자로부터 받은 입력을 저장하거나 알맞은 레지스터의 값을 출력하는 기능을 하는 Register file로 구성된다.

Write의 연산을 진행하면 tail 0부터 증가, Read의 연산을 진행하면 head가 0부터 증가하는 방식으로 구현한다. 따라서 먼저 입력된 데이터는 0, 1, 2... 번째 레지스터에 저장되고 출력되는 데이터 또한 0, 1, 2... 번째 레지스터의 값을 출력하므로 FIFO의 기능으로 동작한다.

2. 설계 과정

State	Encoding
Initial state	000
No operation	111
Read	010
Write	001
Read error	110
Write error	101

과제 설명서의 Testbench를 보고 그대로 인코딩하여 현재 상태를 표현했다.

처음에는 initial 상태이므로 데이터의 개수, head, tail이 모두 0이다. 따라서 resettable D-FF를 통해서 reset이 0이면 0을 내보내도록 하여 처음 상태를 표현했다. Reset이 1이 되고, wr_en이나 rd_en이 1이 되면 현재 상태에 따라서 write 또는 read의 기능을 수행한다.

FIFO는 먼저, Next state logic을 통해 현재 상태와 데이터의 개수를 확인하여 다음 상태를 출력한다. 예를 들어서 데이터의 개수가 8개가 아니라면 write가 가능하므로 상태는 001이 될 것이고, 데이터의 개수가 8개라면 write가 불가능 하므로 상태는 101이 될 것이다.

또한 Calculate logic을 통해, 다음 상태에 대한 head, tail, 데이터의 개수를 계산한다. 데이터의 개수를 확인하여 8개가 아니라면 write가 가능하므로 tail이 하나 증가하여 저장되고 Register file로 연결되는 we(Write enable)은 1이된다. 8개라면 tail은 그대로 유지되며 we가 0이 된다.

이렇게 D Flip Flop으로 넘어간 값은 Output logic으로 전달된다. 아까 인코딩한 상태의 binary값에 따라서 알맞은 signal들을 출력하고 현재 데이터의 개수에 따라 Status flag 또한 출력한다.

Calculate logic에서 계산된 tail과 head는 register file에, 각각 wAddr(write 연산의 주소), rAddr(read 연산의 주소)에 전달된다.

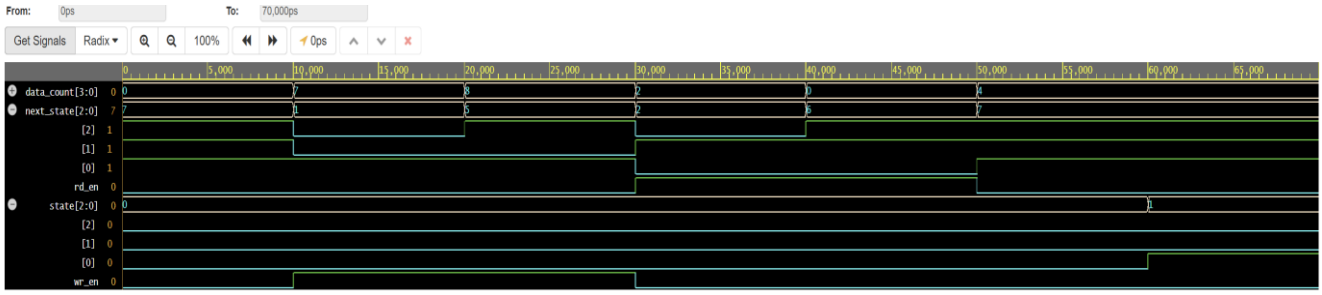
Register file에서는 wAddr 값으로 디코더를 이용하여 어떤 register에 값을 저장할지 선택한다. 예를 들어서 3 to 8 디코더의 input이 010이라면 output은 00000100이므로, 2번째 레지스터가 선택된다는 의미이다. write 연산이 가능하고, 해당 레지스터가 업데이트 될 때만 write 연산을 진행하도록 구현하기 위해서 디코더의 output과 we 값을 AND 연산하여, 각 레지스터의 enable 값으로 전달했다. 각 레지스터는 enable D FF로 구성되므로 enable이 1이 되어야 새로운 input을 업데이트할 수 있다.

따라서 과제에서 구현해야 할 register file의 write 기능을 제대로 수행할 수 있다.

Mux는 selection value가 011이라면 3번째 input이 output으로 나오므로, rAddr을 selection value 값으로 사용하는 8 to 1 Mux를 이용하여 어떤 레지스터에 저장된 값을 출력할지 결정하는 방식으로 register file의 read 기능을 수행할 수 있다.

3. 설계 검증

<Next State Logic>



0 ~ 10,000ps를 보면 처음 state는 000이므로 init 상태이고, we_en과 rd_en이 모두 0이다. 따라서 다음 상태는 111인, No operation이 되는 모습을 볼 수 있다.

10,000 ~ 20,000ps를 보면 현재 state는 000이고 wr_en이 1이며 데이터의 개수는 7개이다. 따라서 write가 가능하므로 다음 상태는 write가 가능한 상태를 의미하는 001이 되는 모습을 볼 수 있다.

20,000 ~ 30,000ps를 보면 wr_en은 1이지만 데이터의 개수가 8개이다. 따라서 wirte가 불가능하므로 다음 상태는 write가 불가

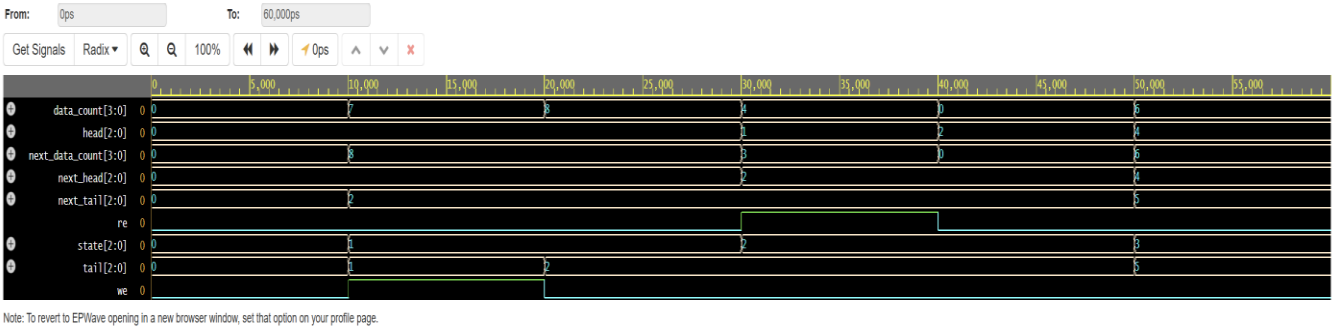
능한 상태를 의미하는 101이 되는 모습을 볼 수 있다.

30,000 ~ 40,000ps를 보면 rd_en은 1이고, 데이터의 개수는 2개이다. 따라서 read가 가능하므로 다음 상태는 read가 가능한 상태를 의미하는 010이 되는 모습을 볼 수 있다.

40,000 ~ 50,000ps를 보면 rd_en은 1이지만 데이터의 개수가 0개이다. 따라서 read가 불가능 하므로 다음 상태는 read가 불가능한 상태를 의미하는 110이 되는 모습을 볼 수 있다.

50,000ps 이후부터는 다시 wr_en과 rd_en이 모두 0이므로 다음 상태는 111이 되는 모습을 볼 수 있다.

<Calculate Address Logic>



0 ~ 10,000ps를 보면 현재 state가 000이므로 head, tail, data count가 그대로 유지된다.

10,000 ~ 20,000ps를 보면 현재 state가 001이고 tail은 001, data count는 7개 이므로 Write가 가능하다. 따라서 다음 tail은 010, data count는 8개가 된다. Write를 하므로 head는 변하지 않고 그대로이고 we는 1이 된다.

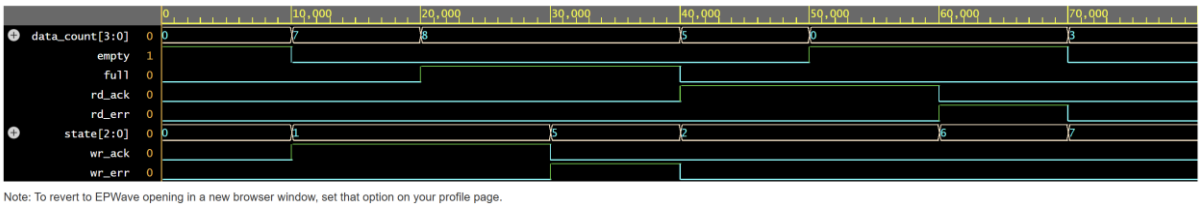
20,000 ~ 30,000ps를 보면 state가 001이지만 현재 tail은 010, data count는 8개이므로 wrtie가 불가능하다. 따라서 tail와 data count, head까지 모두 그대로 유지되며 we는 0이 된다.

30,000 ~ 40,000ps를 보면 state가 010, head는 001이며 data의 개수는 4개이다. 따라서 Read가 가능하므로 다음 head는 010, data의 개수는 3개가 되고 re도 1로 바뀐다. Read를 하므로 tail은 변하지 않고 그대로 값을 유지한다.

40,000 ~ 50,000ps를 보면 state가 010, head는 010이다. 하지만 data의 개수가 0개이므로 Read가 불가능하다. 따라서 data의 개수와 head, tail까지 그대로 유지되며 re는 0이 된다.

50,000 이후로는 state가 011이므로 정의되지 않은 상태이다. 따라서 we, re가 모두 0이되고 현재 data, tail, head도 그대로 값을 유지하며 아무 동작을 하지 않는다.

<Output Logic>



0 ~ 10,000ps를 보면 상태는 000이므로 큐가 비어 있음을 뜻하는 empty flag만 1이고 나머지는 모두 0임을 볼 수 있다.

10,000 ~ 20,000ps를 보면 상태가 001이고 데이터는 7개 이므로 wr_ack flag가 1이 됨을 볼 수 있다.

20,000 ~ 30,000ps를 보면 상태가 001이지만 데이터는 8개 이

므로 큐가 꽉 찼음을 뜻하는 full flag와 wr_ack flag가 1이 됨을 볼 수 있다. wr_err flag가 0인 이유는 해당 로직은 현재 State에 따라서 signal을 출력하기 때문에, 현재 상태는 001 이므로 wr_ack flag가 1이 된다.

30,000 ~ 40,000ps는 이전 케이스에서 state가 101로 바뀐 케이스이다. 101 상태는 큐가 가득 차 있어서 write가 불가능한 상태이므로 해당 케이스에서 wr_err flag가 1이 되는 모습을 볼 수 있다.

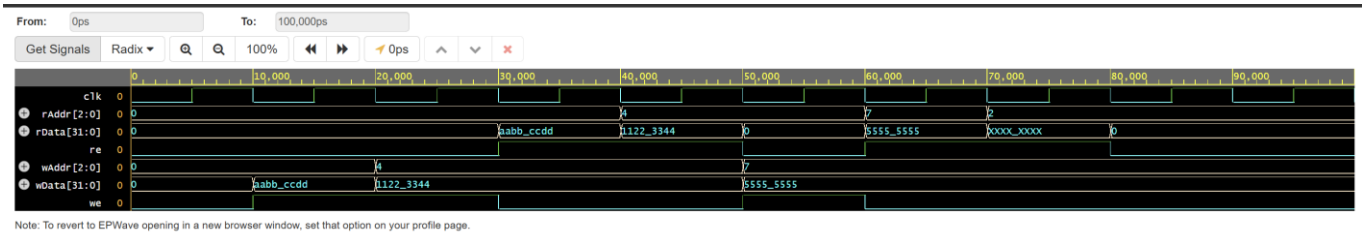
40,000 ~ 50,000ps는 상태가 010이고 데이터의 개수가 5개이다. 따라서 rd_ack가 1이되고 full와 empty 모두 0이 됨을 볼 수 있다.

50,000 ~ 60,000ps는 이전 케이스에서 데이터의 개수가 0이된 케이스이다. 큐가 비어 있으므로 empty flag가 1이 된다. 20,000 ~ 30,000의 케이스와 마찬가지로 현재 상태는 010이므로 rd_ack flag가 1이 됨을 볼 수 있다.

60,000 ~ 70,000ps는 이전 케이스에서 상태가 110으로 바뀐 케이스이다. 110 상태는 큐가 비어 있어서 read가 불가능한 상태이므로 해당 케이스에서 rd_err flag가 1이 되는 모습을 볼 수 있다.

70,000ps 이후는 상태가 111이고 데이터의 개수가 3이므로 아무 작업도 수행하지 않아서 Handshake signal이 모두 0이고 full과 empty 또한 0이다.

<Register file>



0 ~ 10,000ps는 we와 re가 모두 0이므로 Write와 Read가 작동하지 않는다.

10,000 ~ 20,000ps는 we가 1, wAddr은 0이므로 wData의 값인 aabbccdd가 0번째 레지스터에 write 된다.

20,000 ~ 30,000ps는 wAddr이 4이므로 wData의 값인 11223344가 4번째 레지스터에 write 된다.

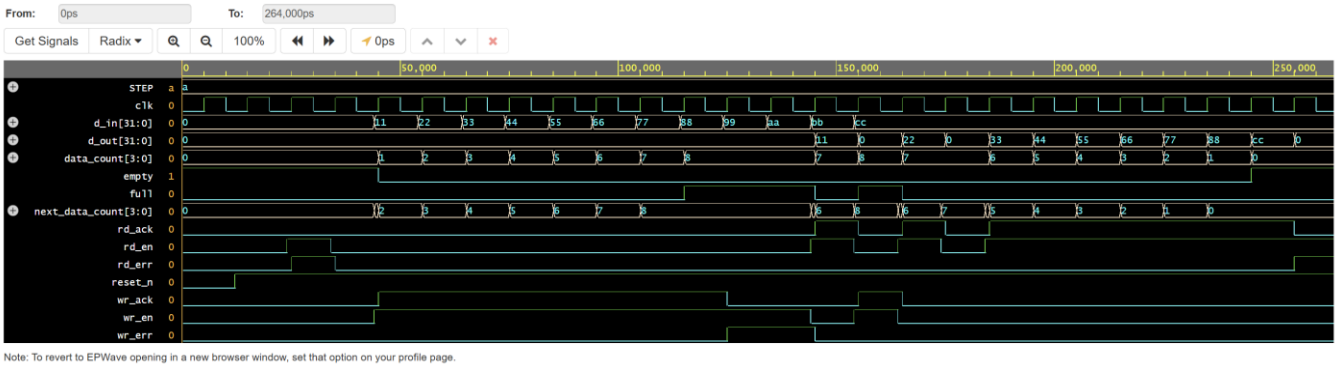
30,000 ~ 40,000ps는 re가 1, rAddr은 0이므로 0번째 레지스터에 저장된 값을 읽어온다. 10,000 ~ 20,000ps의 케이스에서 0번째 레지스터에 aabbccdd를 저장했으므로 결과 값이 aabbccdd가 나오는 것을 볼 수 있다.

40,000 ~ 50,000ps는 rAddr이 4이므로 4번째 레지스터에 저장된 값을 읽어온다. 20,000 ~ 30,000ps의 케이스에서 4번째 레지스터에 11223344를 저장했으므로 결과 값이 11223344가 나오는 것을 볼 수 있다.

50,000 ~ 60,000ps에서 7번째 레지스터에 55555555의 값을 저장했고 60,000 ~ 70,000ps에서 7번째 레지스터에 저장된 값을 가져온 결과, 55555555임을 볼 수 있다.

70,000 ~ 80,000ps에서 2번째 레지스터에 저장된 값을 가져오는데, 2번째 레지스터에는 write한 값이 없으므로 xxxxxxxx가 출력되는 모습을 볼 수 있다.

<FIFO>



0 ~ 50,000ps에서 처음 FIFO가 실행됐으므로 empty flag가 1이 되고 나머지 flag는 모두 0이다. 또한 data의 개수와 d_out도 0임을 볼 수 있다. 또한 rd_en이 1이 되었을 때, 큐가 비어 있으므로 rd_err flag가 1이 됨을 볼 수 있다.

50,000 ps 이후로 wr_en이 1이 되어서 Write를 수행하기 시작한다. Wave를 보면 data의 개수가 점점 증가하면서 we_ack도 1인 상태이므로 next_data의 개수도 점점 증가하는 결과를 볼 수 있다. Data의 개수가 8개가 된 이후, 큐가 꽉 차 있으므로 we_err flag가 1이 되면서 더 이상 data의 개수가 증가하지 않고 full flag 또한 1이 된 결과를

볼 수 있다.

이제, rd_en이 1이 되었으므로 rd_ack flag가 1이 되면서 000에 저장되었던 값인 11이 d_out으로 11이 나온 결과를 볼 수 있다. 또한 data의 개수가 7개로 줄었고, 현재 read 상태이므로 next_count의 개수가 6개임을 볼 수 있다.

다시 wr_en이 1이 되었는데, data의 개수가 7개이므로 write가 가능하다. 따라서 wr_ack flag가 1이되고 d_in으로 입력된 cc를 레지스터에 write한다.

이후 rd_en이 1이 되면서 Read 기능을 수행한다. d_out으로 22, 33, 44...의 값이 출력되면서 data의 개수도 점점 줄어든다. Read가 계속 수행되며 Data의 개수가 0이 되면 empty flag가 1이 되면서 rd_err flag 또한 1이 된다. 해당 테스트 케이스를 통해서 FIFO의 기능을 정상적으로 수행함을 알 수 있다.

wr_en과 rd_en이 모두 0이 되면 No operation 상태가 되므로 d_out으로 0이 출력된다.

<고찰>

제안서를 잘못 읽고 mux나 D-FF 같은 다른 서브 모듈을 만들지 않고 FIFO를 구성해야 한다고 생각했다. 그래서 과제를 완성한 후, 테스트 벤치를 수행하는데 모듈끼리 클럭이 맞지 않는 상황이 많이 발생했다.

이는 제안서에 추가로 필요한 모듈은 따로 구현해도 된다는 구문을 읽고 추가로 모듈을 구현하고 클럭을 메인 클럭과 연결하여 같은 클럭에 동작하도록 수정했다.

초기에 값은 대부분 0으로 설정되어 있다. 따라서 탑 모듈에서 reset_n의 값을 보고 판단하여 값을 초기화 시키는 방식으로 구현하려 했으나 서브 모듈을 인스턴스하는 과정에서 변수의 자료형이 일치하지 않아서 에러가 발생했다. 그래서 reset_n에 따라서 0을 내보내거나 입력을 내보내는 resettable D-FF를 사용하여 초기화 하도록 구현했다.

mux를 사용하여 register file의 결과와 0을 re에 따라 선택하도록 구현하는 과정에서, register file의 결과는 32 bits 이므로 8'h0을 사용하면 32'b0과 같은 값으로 인스턴스가 진행될 줄 알았으나 테스트 벤치로 결과를 검증해보니, 8 bits만 0이 되고 나머지 24 bits는 Z가 나왔다. 따라서 32'b0의 값으로 인스턴스하여 에러를 수정했다.