컴퓨터 공학 기초 실험2 보고서

실험제목: Synchronous FIFO

실험일자: 2016년 10월 13일 (목)

제출일자: 2016년 10월 27일 (목)

학 과: 컴퓨터공학과

담당교수: 이혁준 교수님

실습분반: 목요일 0, 1, 2

학 번: 2013722095

성 명: 최재은

1. 제목 및 목적
   1. 제목

Synchronous FIFO

* 1. 목적

이전에 실습에 진행한 register file, 5-way counter 등을 응용하여 fifo machine(Queue)를 구현해본다. 이를 통해서 기존의 c++에서 임의의 메모리 위치에 동적 할당하여 구현하였던 방식과 어떤 식으로 다른지 생각하여 보고 이를 구현하는 데에 적용한다. 또한 Intance한 각 모듈이 어떤 역할을 하는지 이해하고 그 port들이 어떤 역할을 하는지를 생각해 본다.

1. 원리(배경지식)

Next state logic은 Write enable / Read enable 신호와 현재 state, data\_count를 입력값으로 받아 연산하여 다음 state를 출력해주는 역할을 합니다. Calculate address logic은 state, Data\_count, head, tail을 입력으로 받아 다음 state에 대한 head, tail의 증감, 현재 queue에 저장되어 있는 data의 개수를 연산합니다. Output logic은 현재 state와 data\_count를 받아서 현재 queue의 상태(full / empty)와 write/read 에 대한 feedback (handshake) signal을 출력해줍니다. 사용자에 의해 입력되었던 data값들은 기존에 구현하였던 register file 모듈에 의해서 저장되거나 출력되합니다.

전체적인 구조는 fifo\_ns module에서 입력 값을 통해 다음 state를 계산해주고 이를 fifo\_cal\_addr module에서 입력으로 받아 write operation을 진행해도 되는지에 대한 허가

signal인 we과 head, tail pointer, data count vector의 다음 상태값을 register file module로 넘기고, register file은 din, we, tail address, head address를 입력으로 받아 출력할 값을

mux의 입력으로 넣어줍니다. . 이 출력되는 값은 2-to-1 Multiplexer의 입력값으로 바로 들어가게 되며, multiplexer가 받아들인 read enable 신호에 의해서 출력이 될 것인지를 선택하게 됩니다. 이 때의 read enable signal은 fifo\_cal\_addr module에서 출력됩니다.

read했을 때의 최종 출력 값인 dout은 dff으로 들어가 clk에 synchronize되어 rising edge에 출력됩니다.

1. 설계 세부사항

1) fifo\_ns

- 현재 state와 현재 register에 저장되어 있는 data의 개수인 data\_count, read/write

enable을 입력으로 받아서 다음 state로 무엇이 위치해야 하는지를 계산해주는 모듈입니다.

- 각 state에 대한 비트값을 parameter로 선언해두었습니다.

- 아래의 경우의 수에 맞게 always문과 case문, if문을 사용하여 구현하였습니다.

- next state는 parameter로 선언하였습니다.

- 다음과 같은 인자들을 갖습니다.

|  |  |  |  |
| --- | --- | --- | --- |
| 구분 | 이름 | 비트 수 | 설명 |
| Input | wr\_en | 1 | Write enable |
| rd\_en | 1 | Read enable |
| state | 3 | Current state |
| data\_count | 4 | Data count vector |
| Output | next\_state | 3 | Next state |

- 다음과 같은 경우의 수를 갖습니다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| input | | | | output |
| state | wr\_en | rd\_en | data\_count | next\_state |
| IDLE | 0 | 0 | X | IDLE |
| 1 | 0 | <8 | WRITE |
| 0 | ==8 | WR\_ERROR |
| 0 | 1 | >0 | READ |
| 1 | ==0 | RD\_ERROR |
| READ | 0 | 0 | X | IDLE |
| 1 | 0 | <8 | WRITE |
| 0 | 1 | >0 | READ |
| ==0 | RD\_ERROR |
| RD\_ERROR | 0 | 0 | X | IDLE |
| 1 | 0 | <8 | WRITE |
| 0 | 1 | ==0 | RD\_ERROR |
| WRITE | 0 | 0 | X | IDLE |
| 1 | 0 | <8 | WRITE |
| ==8 | WR\_ERROR |
| 0 | 1 | >0 | READ |
| WR\_ERROR | 0 | 0 | X | IDLE |
| 0 | 1 | >0 | READ |
| ==0 | RD\_ERROR |

2) fifo\_cal\_addr

- state, data\_count, head, tail을 입력으로 받아 다음 state에 대한 head, tail(register의 주소 값)과 현재 data의 개수를 계산합니다.

- always, case문을 사용하여 아래와 같은 경우의 수를 계산합니다.

- fifo\_cal\_addr module에서 출력되는 we, re signal은 각각 register file의 input signal과

Mux의 selector로써 입력됩니다.

- 사용되는 인자들은 다음과 같습니다.

|  |  |  |  |
| --- | --- | --- | --- |
| 구분 | 이름 | 비트 수 | 설명 |
| Input | state | 3 | Current state |
| head | 3 | Current head pointer |
| tail | 3 | Current tail pointer |
| data\_count | 4 | Current data count vector |
| Output | we | 1 | Register file write enable |
| re | 1 | Register file read enable |
| next\_head | 3 | Next head pointer |
| next\_tail | 3 | Next tail pointer |
| next\_data\_count | 4 | Next data count vector |

- 다음과 같은 경우의 수를 갖습니다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Case | Setting | | | | |
| Current state | we | re | Next\_data\_count | Next\_head | Next\_tail |
| WRITE | 1 | 0 | Data\_count+1 | head | tail+1 |
| READ | 0 | 1 | Data\_count-1 | head+1 | tail |
| WR\_ERROR | 0 | 0 | data\_count | head | tail |
| RD\_ERROR |
| IDLE |

3) fifo\_out

- current state와 data\_count를 입력으로 받아 full, empty, handshake signal을 출력합니다.

- 아래의 인자들을 사용합니다.

|  |  |  |  |
| --- | --- | --- | --- |
| 구분 | 이름 | 비트 수 | 설명 |
| Input | state | 3 | Current state |
| data\_count | 4 | Current data count vector |
| Output | full | 1 | Data full signal |
| Empty | 1 | Data empty signal |
| wr\_ack | 1 | Write acknowledge |
| wr\_err | 1 | Write error |
| rd\_ack | 1 | Read acknowledge |
| rd\_err | 1 | Read error |

- data\_count를 입력으로 하여 다음과 같은 경우의 수를 갖습니다.

|  |  |  |
| --- | --- | --- |
| Data\_count | Full | empty |
| 0 | 0 | 1 |
| 8 | 1 | 0 |
| 0> && <8 | 0 | 0 |

- state를 입력으로 하여 다음과 같은 경우의 수를 갖습니다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| State | Wr\_ack | Rd\_ack | Wr\_err | Rd\_err |
| IDLE | 0 | 0 | 0 | 0 |
| WRITE | 1 | 0 | 0 | 0 |
| READ | 0 | 1 | 0 | 0 |
| WR\_ERROR | 0 | 0 | 1 | 0 |
| READ\_ERROR | 0 | 0 | 0 | 1 |

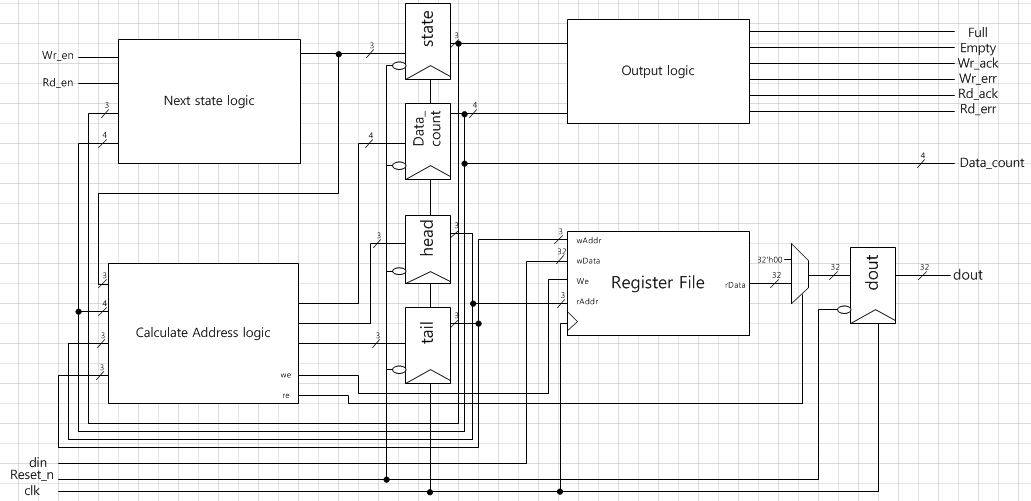
4) fifo

- register file, mux등은 이전에 실험에서 구현한 것을 instance하여 사용하였습니다..

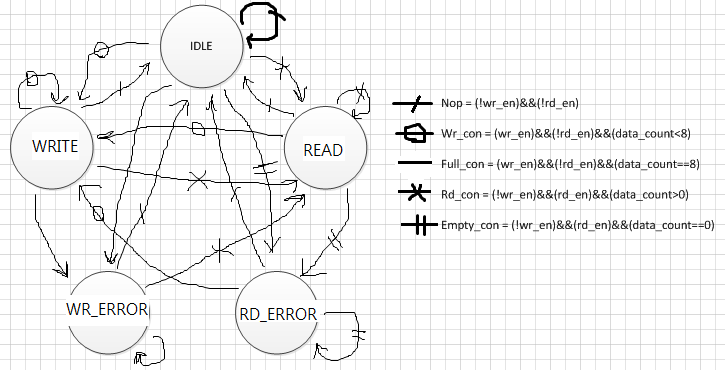
- 단, ppt에서는 register file module이 read enable을 받아서 head data를 출력했으나,

실제 구현에 있어 저는 register file의 head data가 mux의 d1입력으로 들어가게 하고, mux의 selector로 read enable을 주어 read가 정상적으로 되어야 하는 조건에서 출력이 되도록 구현하였습니다.

- 전체적인 모듈의 구성은 다음과 같이 표현됩니다.



- 아래와 같은 state 진행을 갖습니다.



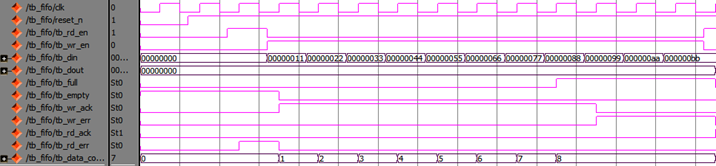
- instance한 모듈은 다음과 같습니다.

|  |  |  |
| --- | --- | --- |
| 구분 | 이름 | 설명 |
| Top module | fifo | FIFO의 top module |
| Sub module | fifo\_ns | Next state module |
| Sub module | fifo\_cal\_addr | Calculate address module |
| Sub module | fifo\_out | Ouput logic module |
| Sub module | Register\_file | Register file module |

- 사용된 인자들은 다음과 같습니다.

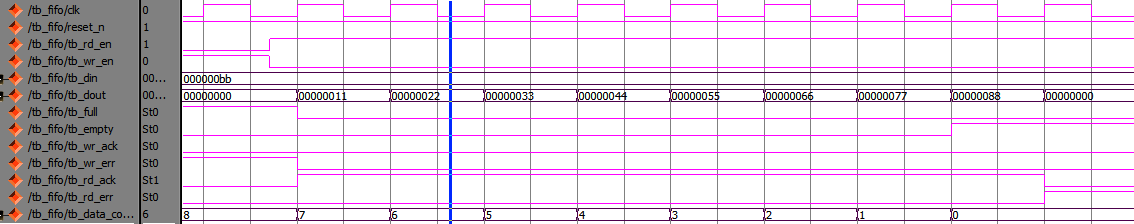
|  |  |  |  |
| --- | --- | --- | --- |
| 구분 | 이름 | 비트 수 | 설명 |
| Input | clk | 1 | Clock |
| reset\_n | 1 | Active low에서 동작하는 reset신호, 값이 인가되면 register의 값을 0으로 초기화함 |
| rd\_en | 1 | Write enable |
| wr\_en | 1 | Read enable |
| d\_in | 32 | Read address |
| Output | d\_out | 32 | Read data |
| full | 1 | Data full signal |
| empty | 1 | Data empty signal |
| wr\_ack | 1 | Write acknowledge |
| rd\_ack | 1 | Write error |
| wr\_err | 1 | Read acknowledge |
| rd\_err | 1 | Read error |
| data\_count | 4 | Data count vector |

1. 설계 검증 및 실험 결과
   1. 시뮬레이션 결과

- 위의 wave form을 보면 22ns에 rd\_en이 1이 되었으나 저장된 data가없으므로 다음 rising edge

에서 rd\_err이 1이 되는 것을 확인할 수 있습니다.

- 이후 data가 입력이 되면서 data\_count가 증가하는 것을 확인할 수 있었습니다.

- data count가 8이 되면서 full signal이 1이 되었고 추가적인 입력을 주자 wr\_err이 1이 되었습니다.

- 이후 rd\_en이 1이되고 wr\_en을 0으로 하자 data\_count가 감소하면서 dout에 head가 출력되는

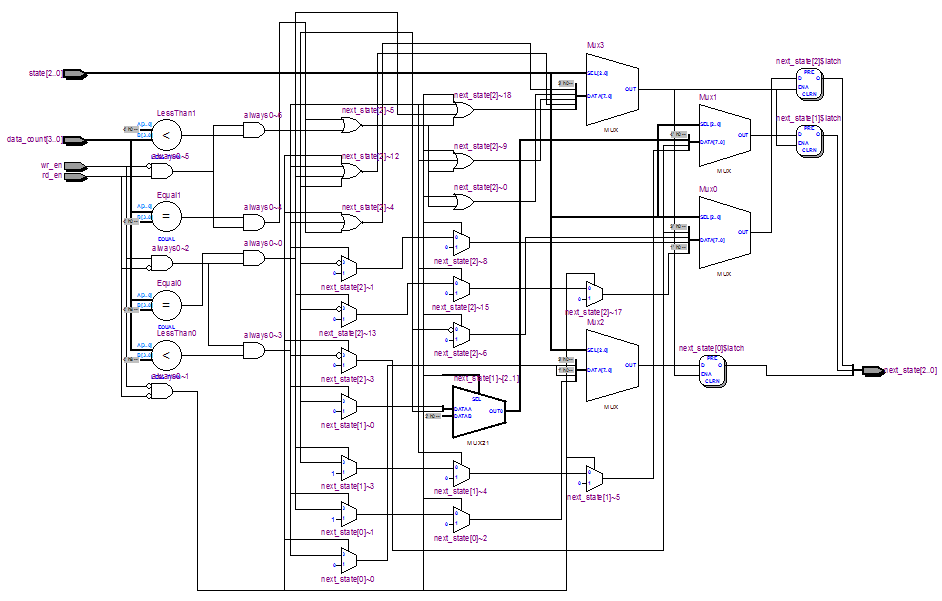
것을 확인할 수 있었습니다.

- 마지막 data를 read하자 empty signal이 1이 되고, 이후 추가적인 read 호출시 rd\_err이 1이

되는 것 확인 할 수 있습니다.

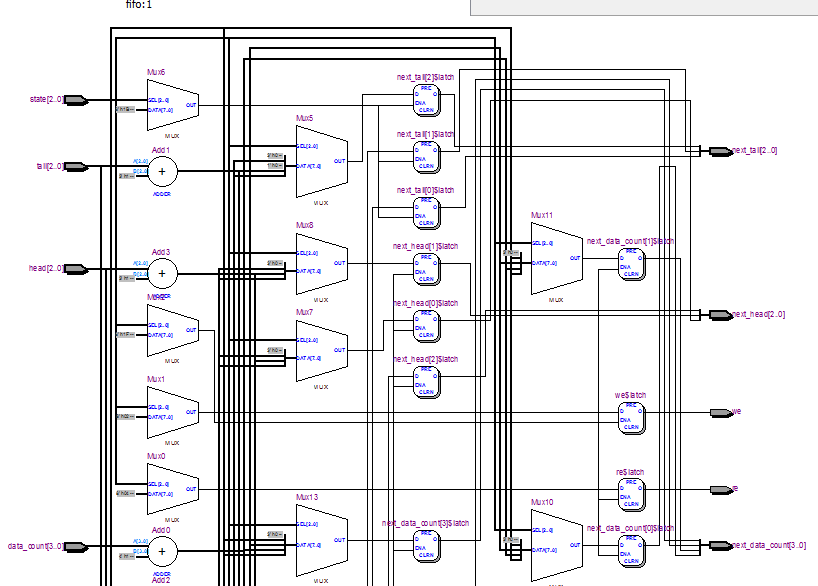
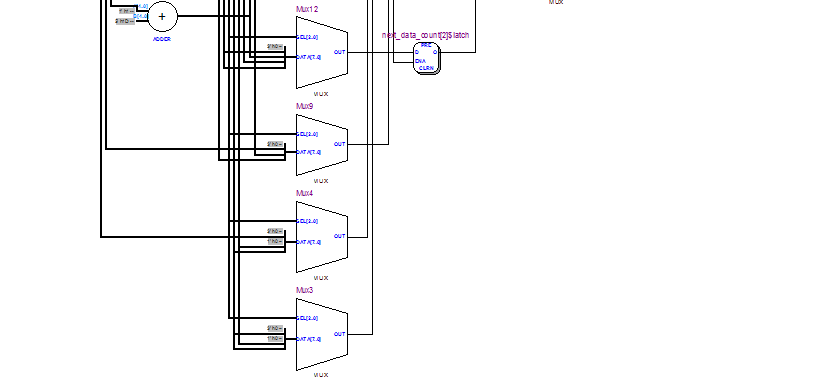
* 1. 합성(synthesis) 결과

**1) fifo\_ns**

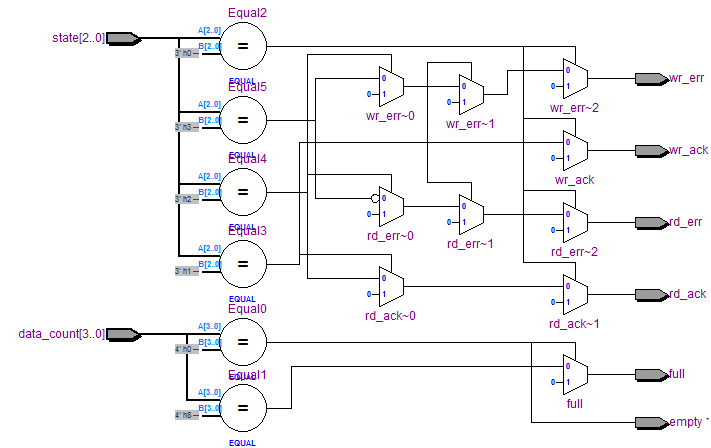
**<RTL Viewer>**

**2) fifo\_cal\_addr**

**<RTL Viewer>**

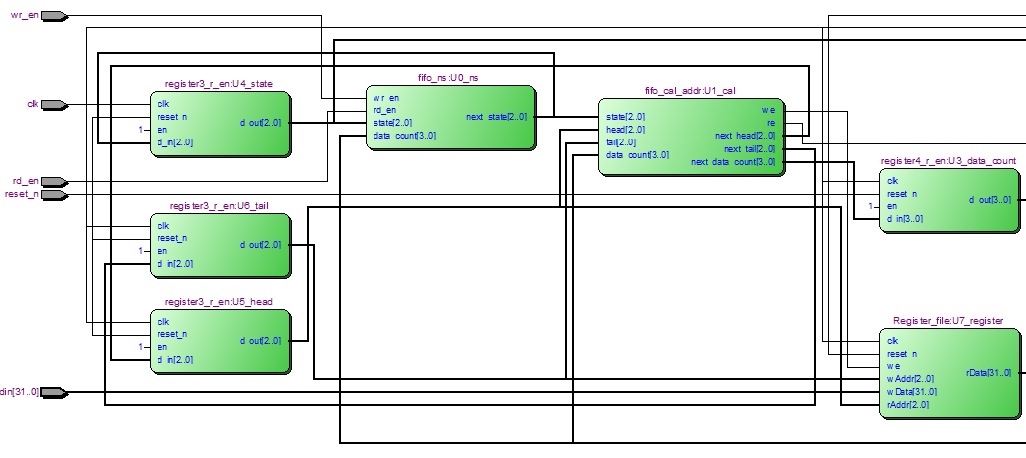


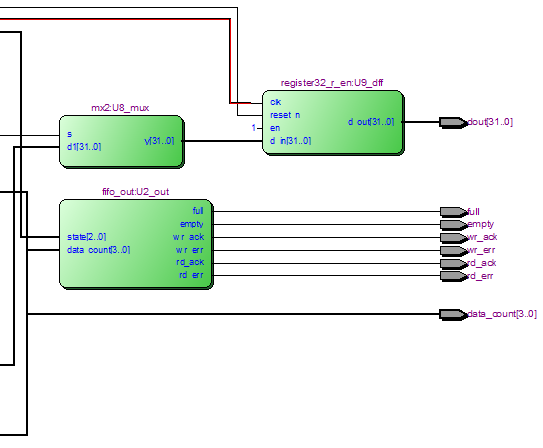
**3) fifo\_out**



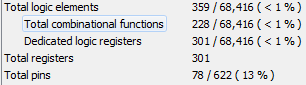
**4) fifo**

**<RTL Viewer>**





- 배경원리에 설명한 것처럼 각 입출력이 잘 이어졌음을 확인할 수 있었습니다.

**<Flow summary>**

**-** 359개의 logic이 사용되었습니다.

- top module에서 input으로 clk, reset\_n, rd/wr\_en, 32bit din, output으로 32bit dout,

Full, empty, wr/rd\_ack, wr/rd\_err, 4bit data\_count 총 78개의 핀이 사용되었습니다.

- top module에서 instance한 module에서 output reg로 총 301개의 register가 사용되었습니다.

1. 고찰 및 결론
   1. 고찰 및 결론

이번 실험을 하면서 기존에 구현하였던 module을 instance하면서 요구하는 입력 값이

충족되지 않는 module은 직접 signal을 1로 설정하여주었습니다. 또한 c언어에서는 임의의 메모리 공간에 동적 할당하여 이를 pointer로 link해줌으로써 queue를 구현하였는데 지정된 메모리 공간의 주소 값을 가리키는 pointer의 값을 변경함으로써 queue를 짜는 것이 신기했습니다. 기본 개념 자체는 같은데 왜 그리 이질적으로 느껴졌는지…..

처음에 구현하였을 당시에 data\_count가 0이 되고도 쭉 아래로 내려갔었는데, 이를 fifo\_ns module에서 case 설정을 빼먹은 부분이 있다는 것을 찾아내어 수정하였습니다. 이외에 큰 어려움은 없었지만 다음에 project를 할 때는 이런 자료를 주지 않고서 혼자 모든 것을 해야 한다니 걱정이 앞섭니다.

1. 참고문헌