**- 고찰(문제 해결 과정)**

**고찰 1. 어떻게 설계할 것인가?**

도표, 텍스트, 평면도, 기술 도면이(가) 표시된 사진

자동 생성된 설명

기본적으로 **“철저한 모듈화와 엄격한 객체지향적 설계”**를 지향한다. 이는 그림만 봐도 알 수 있듯이 MPU가 각 세션을 모듈화하여 구현하기 좋고 각 모듈의 역할이 명확히 구분된다는 점에서, 객체지향 설계의 이점이 클 것이라 생각했기 때문이다. 이는 또한 중간고사 코딩 과제에서 오류가 날 때마다 객체지향 철학을 조금씩 위배했다가, 후반에 걷잡을 수 없이 꼬여 고생했던 경험에서 비롯된 생각이기도 하다.

객체지향적 설계에서 꼭 지켜야 할 점은 다음과 같다.

* Reg와 ALU, MUX는 지정된 스펙대로 동작하는 본래 자체의 기능을 하는 모듈일 뿐, **자기가 MPU에 들어가는지조차 몰라야 한다**(변수명과 같이 기능 외적인 요소는 예외). 즉, 본래 기능에 불필요한 입력과 출력, 내부 로직이 없어야 한다.
* I/O, CTRL 모듈은 **Reg, MUX, ALU의 내부 값이나 신호, 상태에 관심이 없어야 한다.** 이 셋의 최종 결과인 ALU의 반환값에만 관심을 가진다. 이 둘은 앞서 말한 3개의 모듈에 대해 동작 커맨드를 던져주기만 할 뿐이며, 의미적으로 상위 계층 모듈에 해당한다. ALU의 결과가 I/O에 반환되는 것 외에는 이 상위 계층에 어떠한 피드백도 없는 불합리한 관계여야 한다.

이와 같이 객체지향적 설계를 해 각 모듈의 책임을 모듈 내부로 격리시킴으로써, 오류 수정과 기능 관리와 같은 유지보수 측면에서 큰 이점을 얻을 것이다.

**고찰 2. 어느 모듈이 state를 관리하게 할 것인가?**

텍스트, 스크린샷, 도표, 폰트이(가) 표시된 사진

자동 생성된 설명 텍스트, 도표, 라인, 스크린샷이(가) 표시된 사진

자동 생성된 설명

시스템을 FSM으로 만들어야 함은 당연했는데, FSM으로 만들 때 state 관리를 어느 모듈에서 할 지를 정해야 했다. 상위 계층에 있는 I/O 모듈과 CTRL 모듈 중 하나가 담당하게 하거나, 모든 모듈을 호출하는 최상위 모듈인 MPU 안에서 담당하게 하는 방법이 있는데, 최종적으로 I/O 모듈이 담당하게 하는 것으로 결정되었다. 그 이유는 I/O가 담당하는 사용자 입력과 사용자 출력이 각 state와 매우 밀접하게 연관되어 있으며, Instruction 전달과 Result의 receive 역시 I/O가 담당하는 것으로 되어 있었기에 사실상 시스템 전체에 대한 컨트롤 주체가 I/O모듈이라고 봤기 때문이다.

**고찰 3. stateless 모듈 조합의 무결성?**

텍스트, 도표, 스크린샷, 평면도이(가) 표시된 사진

자동 생성된 설명

I/O모듈에서 state를 관리하기로 정했다고 하더라도, 다른 모듈은 stateless로 구현해도 된다는 보장은 없었기에 이에 대한 고찰도 필요했다. 문제가 될 수 있는 유일한 취약점은 **Register에 값을 쓰는 동작**에 있다. Register에 값을 쓰지 않는 이상은, 과정의 초기에 이상한 값이 나타나더라도 어쨌든 CLK 사이클이 몇 회만 지나고 나면 정상적인 동작을 기대할 수 있다. 그러나 Register에 값을 쓰는 비가역적인 동작을 포함할 경우, 오류 역시 비가역적이게 되어 가만히 둬도 정상 궤도로 돌아오지 않아 주의가 필요하다. 이에 대해 고민한 결과, 아래의 조건만 지키면 I/O 모듈 외에는 모두 stateless로 구현해도 된다는 결론이 나왔다. 그 조건은 다음과 같다.

* CTRL 모듈이 CLK 기반으로 동작하지 않고 wire의 assign으로 이루어진 delay 없는 회로여야 한다.
* Instruction의 15~0번째 비트가 모두 동시에 생성되어 CTRL과 Register에 입력되어야 한다.

첫번째 조건은 Instruction[15:12]와 그에 따른 Reg\_Write신호가 같은 시간대에 동기화되어야 함을 의미한다. 이 조건은 CTRL 모듈이 매우 한정적인 입력과 출력 boundary를 가지기에, 아주 단순한 decoder로 구현될 수 있어 충족 가능한 조건이다.

두번째 조건은 CTRL의 Reg\_Write와 Register의 Write 대상 주소가 같은 시간대에 동기화되어야 함을 의미한다. 이 조건은 사용자 입력을 저장해 두었다가 한번에 Instruction에 대입함으로써 충족할 수 있다.

위 조건들을 통해 결과적으로 Reg\_Write 신호와 Write 대상 주소가 항상 동시에 바뀌도록 하여 엉뚱한 주소에 Write하는 일을 방지하게 된다. Write될 값(Result)은 ALU 연산 delay에 의해 늦게 들어와도 상관없다. 늦도록 둬도 몇 클럭 사이클 안에 결국 옳은 값을 덮어쓰게 될 것이기 때문이다.

**고찰 4. 성능 향상의 여지**

텍스트, 도표, 평면도, 기술 도면이(가) 표시된 사진

자동 생성된 설명

위 3번째 고찰에서의 결론으로 Instruction의 모든 비트가 동시에 생성되어 입력되도록 해야 했다. 이 경우 MPU의 연산은 ‘명령어 입력 4’ state가 끝남과 동시에 시작된다. 그런데 사용자가 명령어를 Opcode부터 Rd1, Rd2, Wr 순서로 입력하기 때문에, 명령어의 앞부분이 준비되는대로 즉각 뒤의 모듈에 전달할 수 있다면 Rd1 혹은 Rd2가 입력된 시점부터 연산을 미리 시작하게 할 수 있다. 이러면 delay가 많이 소요되는 ALU 연산의 결과를 더 일찍 받아 총 연산 시간을 단축시킬 수 있게 된다.

텍스트, 스크린샷, 도표, 라인이(가) 표시된 사진

자동 생성된 설명

다만 이 경우 Reg\_Write 신호와 Write 대상 주소가 동기화되지 않기 때문에 Write 대상 주소가 입력되기 전까지 Register의 Write 동작을 차단할 수 있어야 한다. 이는 CTRL의 Reg\_Write 신호와 I/O의 ‘명령어 입력 4’ state가 완료되었다는 신호를 최상위 모듈에서 AND 게이트로 묶어 Register에 입력되게 함으로써 다시 Reg\_Write 신호와 Write 대상 주소를 동기화하여 해결할 수 있다.

**고찰 5. ‘실행’ state의 필요성?**

텍스트, 폰트, 스크린샷, 라인이(가) 표시된 사진

자동 생성된 설명

지금까지의 설계대로라면 ‘명령어 입력 4’ state와 ‘DONE’ state 사이에 ‘실행’ state를 굳이 넣어야 할까 하는 의문이 생긴다. ‘명령어 입력 2~4’ state에서 이미 부분적으로 연산이 실행된다. 따라서 ‘실행’ state를 따로 구분하여 구현하는 의미가 퇴색된다. 또한 ‘명령어 입력 4’ state 이후 바로 ‘DONE’ state로 전환되어도, 전환 시점에 연산이 끝나 있지 않다 한들 사람이 절대 구분할 수 없을 정도의 매우 빠른 시간 안에 연산이 마저 이뤄져 마치 ‘DONE’ state의 진입과 동시에 올바른 결과가 사용자에게 출력되는 것과 같이 느껴질 것이다. 이것이 ‘실행’ state를 구현하여 사용자에게 연산이 이뤄지고 있음을 강조하고, 불필요한 추가 입력을 요구하기보다 더 좋은 사용 체감을 제공할 것이라고 생각했다. 따라서 ‘실행’ state는 따로 구현하지 않고 ‘명령어 입력 4’ state 이후에 곧바로 ‘DONE’ state에 진입하도록 하였다.

**고찰 5. 버튼 입력 감지의 까다로움**

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

기껏해야 버튼일 뿐이라고 생각했지만, 정작 버튼 입력 감지를 매끄럽게 하는 곳에 시간을 많이 쓰게 되었다. 처음엔 예전 실습 시간에 썼던 방식과 같이 버튼을 debouncer와 synchronzer에 한번씩 거치고, 버튼의 falling edge를 입력 1회로 받아들여 버튼을 꾹 눌렀을 때의 반응을 설계했다. 그러나 결과는 버튼을 한 번 누를 때 아주 많은 횟수가 입력된 것과 같이 동작했고, 꾹 눌렀을 때 역시 빠른 속도로 state가 천이됐다. 분명 debouncer와 synchronzer를 썼음에도 버튼 입력이 상당히 많이 불안정한 듯 했다.

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

이에 이전에 버튼 counter를 만들었던 실습 코드와 현재 코드를 비교해 보았다. 결정적인 차이는 이전 코드의 경우 CLK이 100Hz로 동작하고, 현재 코드는 100MHz로 100만배의 차이가 난다는 것이었다. 이에 현재의 코드도 버튼 필터 모듈 내부에서 freq divider를 사용하여 100Hz까지 떨어진 CLK을 사용하도록 하였고, 결과의 안정성을 확보할 수 있었다.

텍스트, 스크린샷, 폰트이(가) 표시된 사진

자동 생성된 설명

그러나 CLK을 100Hz까지 낮춰도, 여전히 버튼을 꾹 누르고 있었을 때 여러 번으로 인식되는 상황은 계속됐다. 이는 debouncer 내부에서 이유를 찾을 수 있었다. debouncer는 N회 연속으로 High가 감지되면 **단 1회** High를 출력하는 회로이다. 이론적으론 더더욱 버튼이 꾹 눌려있을 때 여러 회 High로 받아들이는 것과는 거리가 있다. 그러나 N회를 세는 counter가 Kbit로 이뤄져 overflow를 일으키며 순회한다는 점에 주목해야 한다. 이에 따라 counter에서 N회가 감지되는 횟수가 1회가 아니게 되며, 결과적으로 버튼이 꾹 눌린 상태에서 counter가 순회하는 주기와 함께 High를 감지하게 되는 것이다.



따라서 K의 값을 충분히 높여 counter가 쉽게 순회하지 않도록 하였고, 언제나 debouncer가 총 1회의 High를 출력하도록 하였다. N의 값은 버튼 입력 감지에 장애를 주지 않는 수준에서 반응성을 향상시키기 위해 적절한 수인 3으로 조였다. 이로써 버튼 입력에 대해 올바른 횟수를 셀 수 있게 되었다.

**고찰 6. 버튼 필터 내부 CLK 주기**

위 고찰의 결과로 버튼 필터 내부에서 100Hz CLK을 사용하게 하였다. 그러나 과제에서 요구사항으로 **“100MHz로 동작하도록 구성한다.”**라고 명시되어 있었기에, 이왕이면 100MHz CLK을 사용하며 문제를 해결할 수 있으면 좋았다. debouncer의 동작과 문제의 원인을 정확히 알고 난 뒤였기에 얼마든지 그렇게 할 수 있었다. 그러나 100MHz CLK을 사용하도록 debouncer를 손보면, 그에 따른 오버헤드가 너무 커진다는 문제가 있었다. CLK 주기가 곧 debouncer의 counter 값 증가 주기와 같은데, debouncer가 쉽게 overflow를 내지 못하게 하려면 K의 값을 너무 크게 잡아야만 한다. 그리고 N 역시 값을 크게 높여야 하는데, 예를 들어 N이 3만이라고 하면 3만 회의 counting 중에 1회라도 Low가 입력될 때 이전의 count를 버리게 되므로 버튼 입력의 안정성이 크게 떨어지게 된다. 이를 위해 적은 수의 Low 입력에 대한 초기화 유예를 위한 counter를 따로 또 만들어 도입해야 할 수도 있게 된다.

따라서 버튼 필터 내부에서만큼은 100Hz CLK을 사용하게 하기로 했다. 모듈화를 통해 100Hz CLK은 철저히 버튼 필터 내부에서만 사용되는 신호가 되며, freq\_divider를 통해 원래 CLK으로부터 파생된 신호이기에 시스템에 별다른 무리를 주지도 않을 것이라 생각한다.

**고찰 7. 버튼과 I/O의 CLK 동기화 문제**

텍스트, 스크린샷, 도표, 폰트이(가) 표시된 사진

자동 생성된 설명

위 고찰 결과에 따라 버튼은 100Hz CLK 기반으로 동작하게 했다. 그러나 이 때 한가지 문제가 발생했다. btn[1]이 눌리고 있는 상황에서 display 값이 바뀌어야 하는데, 이 기능이 동작하지 않는 것이었다. 해당 상황에서 HW를 자세히 살펴보면, btn[1]이 눌리고 있을 때 segment의 불빛이 아주 미세하게 떨리는 것처럼 느껴졌다. **I/O 모듈과 버튼 사이의 CLK 불일치로 인한 문제였던 것이다.** 이에 따라 btn[1]이 I/O모듈과 같이 100MHz CLK 기반으로 동작할 필요가 생겼다.

그런데 사실 100Hz CLK을 쓸 수 밖에 없었던 이유는 버튼 필터를 이용해 입력 횟수를 정확히 세기 위한 것이었지, btn[0]을 제외한 다른 버튼은 그런 필터가 필요한 상황은 아니었다. 왜냐하면 다른 버튼들은 몇 회가 됐건 ‘버튼이 눌렸다는 사실’만 중요(btn[3])하거나, 버튼이 눌려있는 과정에서 노이즈가 있어도 상관없기(btn[1]) 때문이다. 그렇기에 버튼 필터를 통과하는 버튼은 btn[0]뿐인 것으로 하고, 다른 btn은 버튼 필터를 통과하지 않은 채 100MHz CLK 기반으로 그대로 시스템에 연결되도록 하였다. 그 결과, 의도한 동작이 제대로 이뤄짐을 확인할 수 있었다.