**CSED-311 LAB2 REPORT**

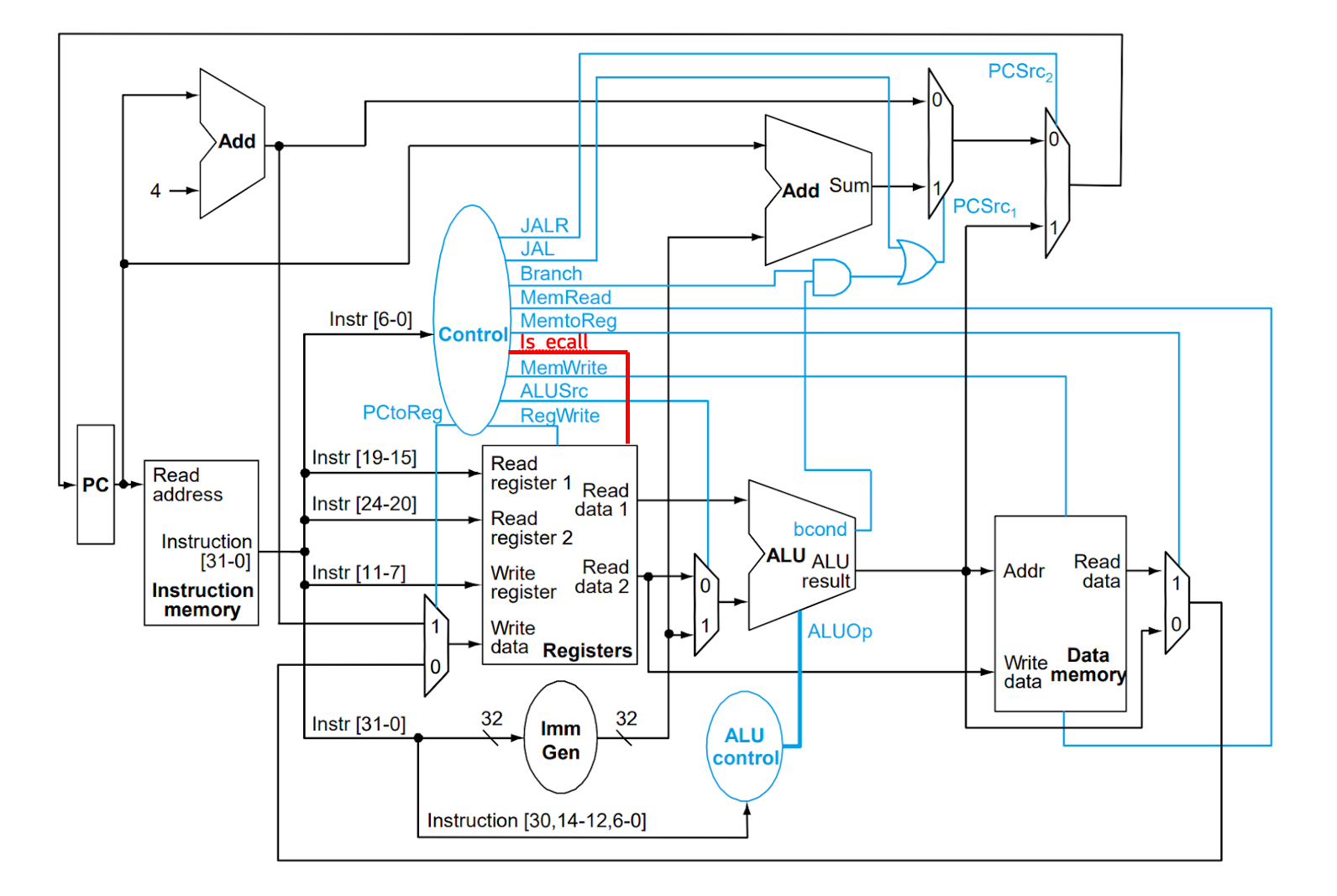
김모세

표승현

**Introduction**

RISC-V instruction을 기반으로 하는 Single cycle CPU를 verilog를 이용해 구현한다. single cycle CPU는 각 명령어를 실행하는 데 하나의 클럭 사이클을 필요로 한다. 각 명령어는 고정된 수의 클럭 사이클을 사용하여 모든 단계를 완료한다. 이는 단순하고 예측 가능한 구조를 제공하지만, 모든 명령어가 동일한 길이의 시간을 소비하기 때문에 효율성과 성능 면에서 비효율적이다.

**Design**

****

**<design of single cycle cpu>**

기본적으로 수업 시간에 공부한 single cycle CPU의 회로 구조를 토대로 구현하였다. control unit에서 is\_ecall 시그널을 활성화하면, register file에서 신호를 받아 X17의 값을 확인한 후 작동을 종료하는 기능을 추가했다. 또한 clk은 positive edge를 기준으로 동작하게 설계하였다.

Single cycle CPU에서는 아래와 같은 5가지 Stages가 있다.

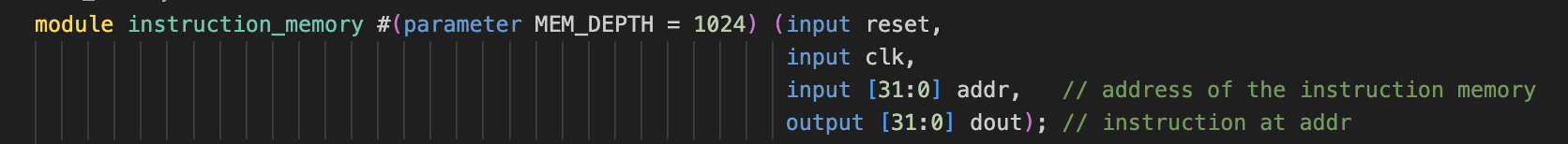
1. Instruction Fetch (IF): 명령어 메모리에서 명령어를 가져오는 단계이다. instruction memory 모듈을 통해 구현했다.
2. Instruction Decode (ID): 명령어를 해독하여 실행할 동작을 결정한다. register file과 control unit 모듈들을 통해 구현했다.
3. Execution (EX): 명령어가 수행된다. 산술 또는 논리 연산 등이 여기에 속하며 ALU 모듈에서 수행한다.
4. Memory Access (MEM): 메모리와의 상호작용이 필요한 명령어인 경우 메모리에 접근한다. Data memory 모듈에서 메모리 상에서의 read 또는 write 동작을 수행할 수 있다.
5. Write Back (WB): 결과를 레지스터에 쓰거나 다음 단계로 전달한다. ALU와 Data memory의 결과를 와이어를 통해 이전 모듈로 전달함으로 구현했다.

**Implementation**

**1)top module**

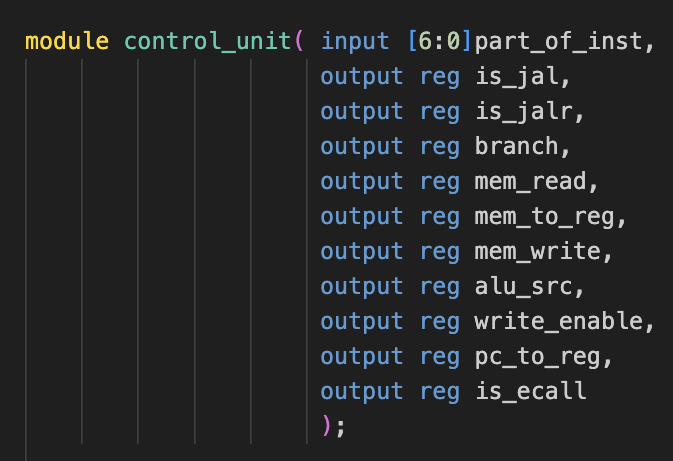
top module은 reset과 clock을 input으로 받고 is\_halted와 reg 32개를 output으로 출력하는 최상위 모듈이다. top module에선 위의 변수들을 cpu module에 input과 output으로 준다.

**2)instruction\_memory module**



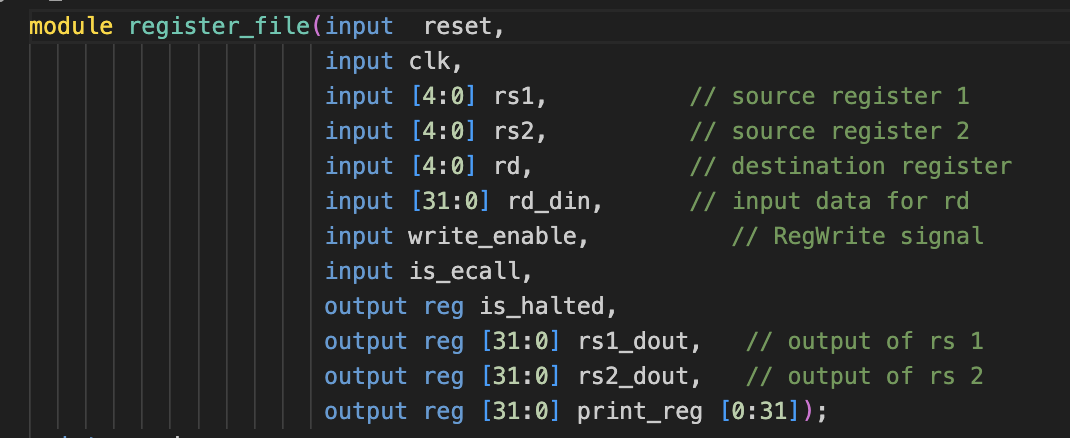
instruction memory module에선 input으로 reset, clk, addr(현재 실행해야 할 instruction의 주소)를 받아 instruction memory에서 addr에 있는 instruction을 읽어 output으로 내보낸다. asynchronous하게 input이 바뀔떄마다 dout을 insturction메모리에서 새로 읽어 할당해준다. 또 synchronous하게 posedge clk마다 reset이 1이라면 메모리를 모두 0으로 초기화해준다.

**3) control\_unit module**



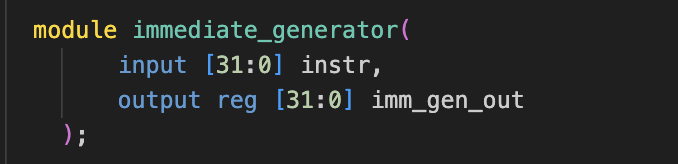
control\_unit에선 instruction 메모리에서 읽은 instruction의 opcode부분(part\_of\_instruction)을 input으로 받으며 part\_of\_instruction이 변할때마다 asynchronous하게 알맞은 control signal들을 설정해준다. 매번 모듈이 불릴때마다 먼저 모든 output signal들을 0으로 해준 뒤, if, else if, else구조를 이용해 part\_of\_inst과 opcodes.v에 정의해둔 값들을 비교하며 필요한 signal에 1을 넣어준다. part\_of\_istruction이 ‘ARITHMETIC이면 연산 후 register에 값을 써야 하므로 write\_enable을 1로 하고, ‘ARITHMETIC\_IMM이라면 regster에 값을 쓰는 동시에 ALUSrc mux에서 변위를 골라야 하므로 write\_enable과 alu\_src를 1로 한다. part\_of\_instruction이 ‘STORE일 경우 변위를 이용해 계산한 data 메모리의 주소에 값을 써야하므로 mem\_write와 alu\_src를 1로 한다. ‘LOAD일 경우 변위를 이용해 계산한 data메모리의 주소에서 값을 읽어 레지스터에 써야하므로 alu\_src, mem\_read, mem\_to\_reg, write\_enable을 1로 한다. ‘Branch의 경우 branch signal을 1로 한다. ‘JAL의 경우 is\_jal을 1로 해주고, register에 다음 instruction의 주소를 저장해야 하기에 pc\_to\_reg와 write\_enable을 1로 해준다. ‘JALR의 경우 ‘JAL과 비슷하지만 추가로 변위를 이용해 분기할 주소를 찾기에 alu\_src를 1로 해준다.(is\_jalr도 1로 함) ‘ECALL의 경우 is\_ecall을 1로 해준다.

**4) register\_file module**

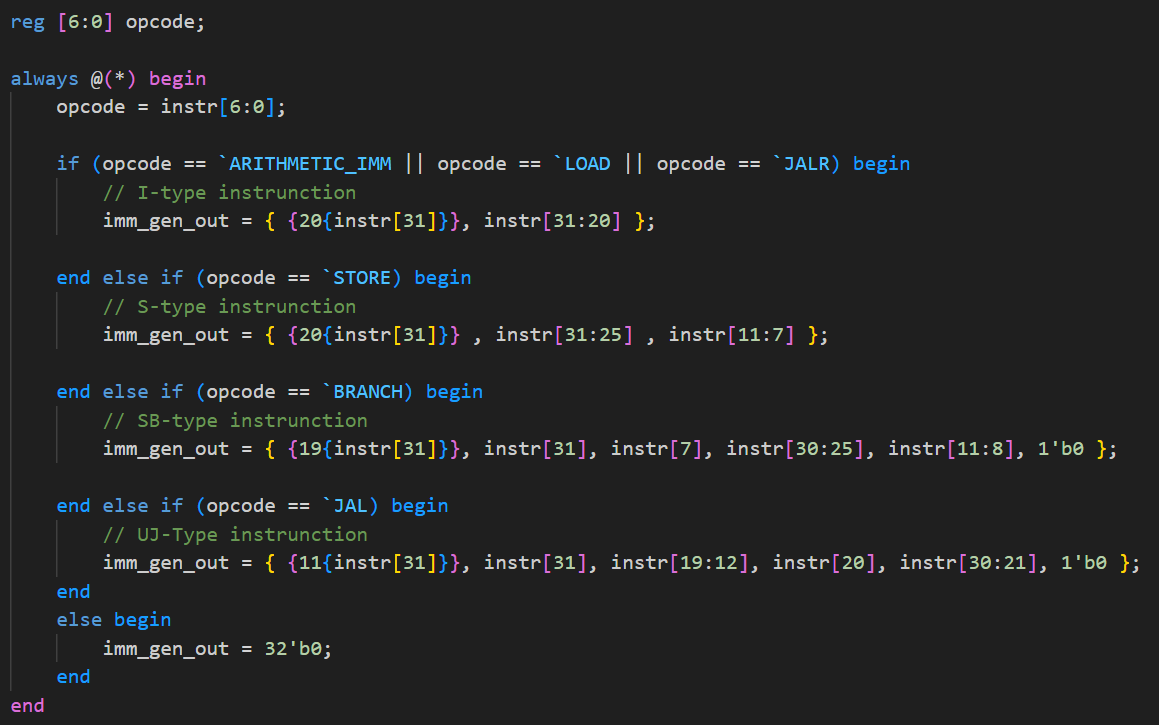


register\_file은 위와 같이 input, output 포트가 선언되어 있다. asynchronous하게 선언해둔 resister array에서 rs1과 rs2 index에 맞는 값을 읽어 rs1\_dout, rs2\_dout에 할당한다. 또한 synchronous하게 write\_enable이 1일 경우 input으로 들어온 rd\_din을 rd index를 가지는 resister에 저장하고 reset이 1일 경우엔 resister에 모두 0을 넣는다. 마지막으로 asynchronous하게 is\_ecall이 1일 경우 is\_halted를 1로 만든다.

**5) immediate\_generator module**

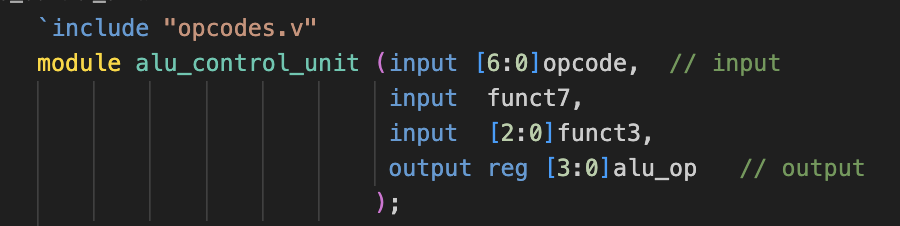


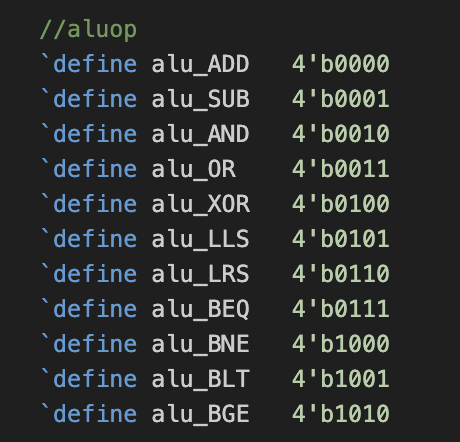
immediate\_generator 모듈에선 instruction 메모리에서 읽은 instr전체를 input으로 받고 asynchronous하게 알맞은 변위를 계산해 내보낸다.



instruction의 0~6비트인 opcode만을 확인하여 필요한 immediate value를 결정할 수 있으므로 7비트의 opcode 레지스터를 선언하여 사용했다. 각 opcode에 따라 instruction에서 immediate value로 활용하는 비트의 자리가 다르기 때문에 if문을 이용해 적절하게 값을 설정해주었다. 본 Lab에서 immediate value는 sign extension을 필요로 하기 때문에 각각의 경우의 MSB로 남은 자리를 채웠다.

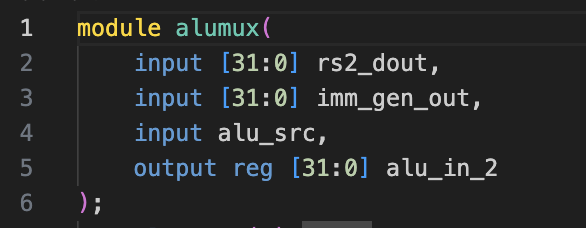
**6)alu\_control unit**



(새롭게 정의한 alu\_op)

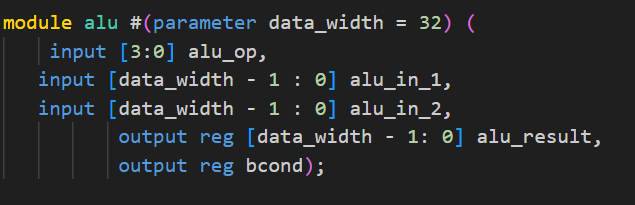
alu\_control unit에선 instruction의 opcode 부분과, funct7의 7bit 중 상위 2번째 1비트, funct3를 input으로 받아 asynchronous하게 4bit의 alu\_op를 output으로 내보낸다. if, else if구조를 써서 구현했으며 먼저 opcode가 ‘ARITHMETIC이고 funct7이 1일 경우 alu\_op에 ‘alu\_SUB를 넣어준다. opcode가 ‘JAL, ‘JALR, ‘LOAD, ‘STORE이면 주소를 연산하는 과정에서 addition을 이용하므로 alu\_op에 ‘alu\_ADD를 넣어준다. 또한 opcode가 ‘ARITHMETIC 이나 ‘ARITHMETIC\_IMM 이면서 동시에 funct3가 FUNCT3\_ADD일 때에도 alu\_op에 ‘alu\_ADD를 넣는다. opcode가 ‘BRANCH일 경우 funct3를 읽어 해당하는 연산을 alu\_op에 넣는다. opcdoe가 ‘ARITHMETIC\_IMM이거나 ‘ARITHMETIC인 경우에도 funct3를 읽어 해당하는 연산을 alu\_op에 넣는다.(ADD는 이전 else if문에서 처리되므로 여기서 처리되지 않는다.)

**7) alumux module**



alumux는 asynchronous하게 alu\_src conrol signal을 받아 레지스터에서 읽은 rs2\_dout과 immediate\_generator에서 나온 imm\_gen\_out 중 하나를 골라 alu\_in\_2에 넣는다. if else문을 이용해 alu\_src가 1일 때 imm\_gen\_out을, 0일때 rs2\_dout을 alu\_in\_2로 해준다.

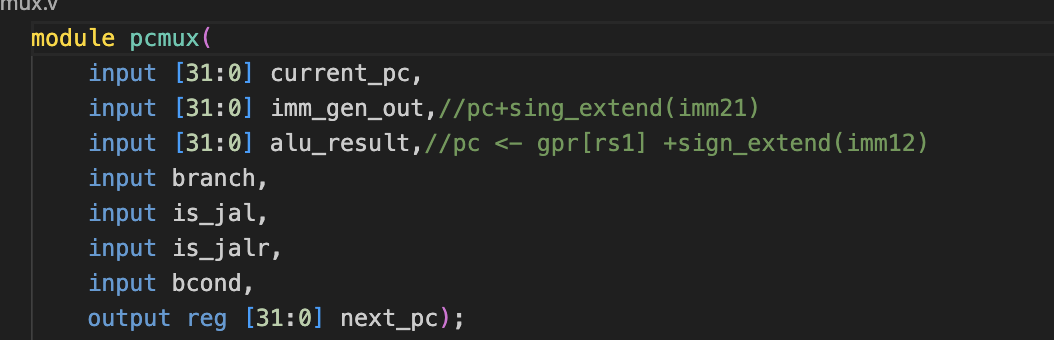
**8) alu module**



single cycle CPU를 구현하기 위해 작성한 ALU 코드이다. 주어진 두 입력값에 대해 필요한 ALU 연산을 수행한다. 본 실습에서는 add, sub 및 논리 연산, 비트 shift 그리고 비교연산만(크기 비교)을 활용했다. 데이터 너비는 기본적으로 32비트로 설정했다.

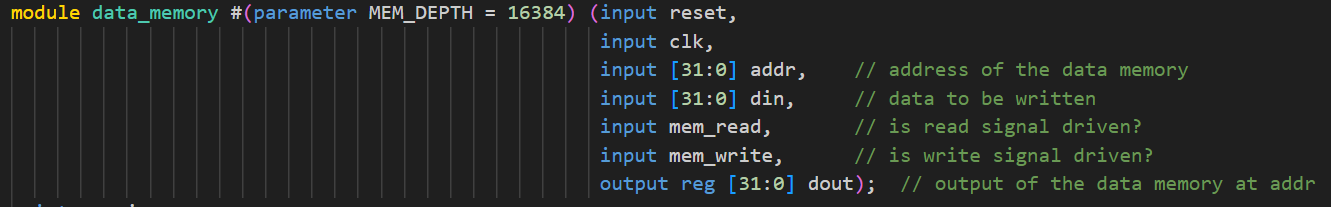
alu\_op, alu\_in\_1, alu\_in\_2가 변할 때마다 ALU 동작이 업데이트된다. 주어진 alu\_op에 따라 적절한 연산이 선택되고 결과가 alu\_result에 할당된다. 비교 연산의 결과는 bcond에 할당된다. 각각의 연산에 대해 if 조건문을 활용하여 주어진 alu\_op에 따라 알맞은 연산을 수행할 수 있도록 하였다.

**9) pcmux module**



pcmux에서는 branch, is\_jal, is\_jalr, bcond를 signal로 받아 asynchronous하게 next\_pc값을 계산한다. if, else if, else 문을 이용하는데 if문에서 branch가 1인 동시에 bcond가 1일 경우와 is\_jal이 1일 경우 next\_pc에 current\_pc +imm\_gen\_out값을 넣어 다음 instruction의 주소를 지정한다. else if문에선 Is\_jalrdl 1일 경우 next\_pc에 ALU연산 결과로 나온 주소를 넣어준다. else문에선 jump나 branch하지 않는 상황이므로 next\_pc에 current\_pc+4를 넣는다.

**10) data\_memory module**



이 모듈은 주어진 주소에서 데이터를 읽거나 쓰는 데 사용된다. mem\_read와 mem\_write 신호가 활성화될 때마다 데이터 메모리 상의 read, write 동작을 적절하게 작동하며, 리셋 신호가 활성화될 때는 초기화된다.

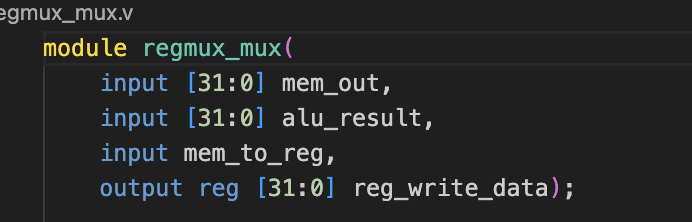


1. 비동기 메모리 읽기(Asynchronously read data from the memory):

mem\_read 신호가 활성화되면, 해당 주소에서 데이터를 읽고, 이를 dout에 할당한다.

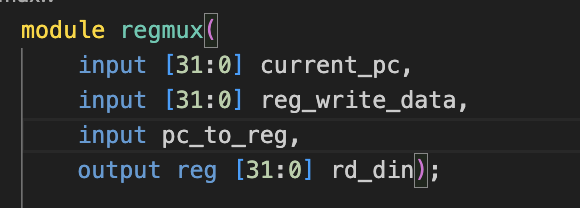
1. 동기 메모리 쓰기(Synchronously write data to the memory):  
   mem\_write 신호가 활성화되면, din에 있는 데이터가 해당 주소에 저장된다.

**11)regmux\_mux module**



regmux\_mux 에선 asynchronous하게 mem\_to\_reg signal을 받아 alu 연산결과와 data 메모리에서 읽은 값 중 무엇을 reg\_wirte\_data에 넘길지 결정한다. (reg\_write\_data는 regmux module에서 pc\_to\_reg가 0이라면 register에 저장된다). if, else문을 이용해 mem\_to\_reg 가 1일 경우 mem\_out을, 0일 경우 alu\_result를 reg\_write\_data에 넣는다.(WB를 위한 부분)

**12)regmux module**



regmux 모듈에선 asynchronous하게 pc\_to\_reg signal에 따라 register에 쓸 값으로 current\_pc를 쓸지 reg\_write\_data를 쓸지 결정한다. if else문을 이용해 pc\_to\_reg가 1이면 current\_pc를 rd\_din으로 하고, 0이면 regmux\_mux에서 할당한 reg\_write\_data를 rd\_din에 넣어준다.

**Discussion**

1. 한정적 ALU 기능 활용
   1. Lab1에서 구현한 alu의 코드를 활용했다. 그러나 single cycle CPU에서는 지난 lab에서 구현한 alu의 연산 기능을 전부 필요로 하지는 않아 필요한 연산 기능만을 추려냈다. add, sub, and, or, xor과 같은 기본적인 연산과 logical shift 그리고 비교 연산만을 활용하였다. 때문에 alu opcode를 4비트 내에서 모두 표현할 수 있었다.
2. alu control unit 구현
   1. alu control unit은 instruction의 opcode와 funct3 비트를 읽고 적절하게 alu의 연산을 선택 및 활성화하는 역할을 한다. 이를 구현하는 과정에서 각 instruction이 어떤 연산을 필요로하는지 탐구하고, 직접 실행시켜 결과를 확인할 수 있었다. 아무리 다양한 instruction이라도 결국에는 alu의 연산 기능을 조합하여 처리할 수 있음을 깨달았다.
3. tb 실행 문제
   1. 이번 lab에서 주어진 tb가 window 환경에서는 문제 없이 실행되었지만, Mac 환경에서는 알 수 없는 오류를 발생시키는 것을 확인하였다. #include <sstream>을 추가하여 해결할 수 있었다. 이후 똑같은 일이 발생할 시 라이브러리 함수를 다시 한 번 살펴볼 것이다.

**Conclusion**

verilog를 이용해 RISC-V instruction을 기반으로 하는 single cycle CPU를 구현하였다. single cycle CPU는 구조가 단순하고 하나의 instruction이 고정된 한 사이클만을 소요하기 때문에 구현하는데 큰 어려움은 없었다. 다만 코딩 과정 중 실수를 하여 기대했던 결과가 나오지 않는 경우에는 디버깅하는 일이 힘들었다. 보통 특정 레지스터에만 예상하지 않은 값이 저장되어 있어 실패하는 경우였는데, 어셈블리 코드를 해석해 해당 레지터스에 관여하는 연산 또는 모듈을 추적하는 방식을 취했다. 이후 배운 multi cycle CPU나 pipeline CPU에 비해 효율성이 부족한 CPU지만 직접 구현하는 과정에서 CPU의 기본적인 작동 원리를 익힐 수 있었다.