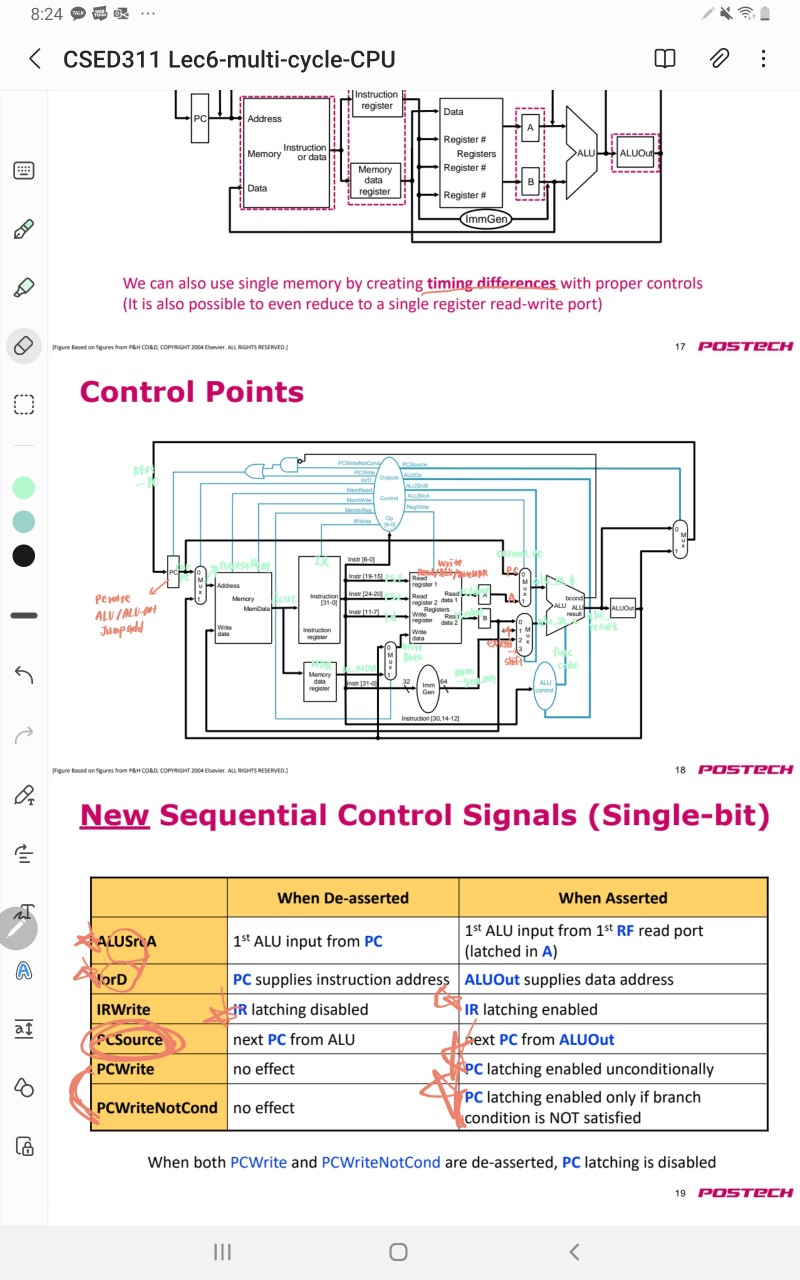
Multi-Cycle CPU

1. Introduction

Multi-Cycle CPU를 구현하는 실습이다. Multi-Cycle CPU는 Single-Cycle CPU보다 효율적으로 동작한다. Single-Cycle CPU는 한 cycle 안에 instruction이 종료되어야 하므로 cycle period를 실행시간이 가장 긴 instruction인 Load instruction을 기준으로 설정한다. 따라서 다른 종류의 instruction에서 불필요한 시간이 낭비된다. 이에 cycle period를 각 instruction 실행시간의 최대공약수 등 특정 길이로 설정하고 각 instruction마다 필요한 cycle만큼 사용하여 실행시간을 줄일 수 있다.

Single-Cycle CPU와 유사하지만 한 instruction에 여러 cycle이 필요하므로 Programmer Visible State(이하 PVS)의 업데이트를 관리하는 새로운 control signal이 필요하다. 또한, 수업시간에 배운 multi-cycle cpu의 큰 특징은 다른 adder를 사용하지 않고, ALU를 통해 연산함으로써 resource를 재사용하는 것이다. ALU를 통해 주소 연산과 명령어의 레지스터 연산 등 여러 연산을 진행하려면 각 연산들이 충돌되지 않도록 연산 사이에 timing difference를 둬야 한다. 따라서 이 두가지에 큰 초점을 두고 실습하는 것이 중요하다.

1. Design



pc값을 저장하기 위한 pc register가 존재한다. 실습에서는 pc 모듈에서 pc 값을 업데이트 해준다. 이때, 매 cycle마다 업데이트 해주면 안되고 각 instruction의 실행이 모두 끝났는지 확인하고 업데이트 해주므로 pc\_write의 control signal이 추가되었다. PC, Memory, Register, ALU 사이 timing difference를 주기 위해 각 단계 사이에 레지스터를 추가했다.

Single-cycle CPU와 다르게 memory 모듈을 하나로 합쳐서 사용한다. 따라서 input으로 pc address와 data address를 받으므로 상황에 따라 이를 판단하는 mux가 memory 모듈 앞에 존재한다. Cycle마다 Instruction Register와 Memory Data Register를 업데이트 하며 register write 및 연산에 필요한 값을 저장한다. 이후 register file에 들어온 instruction 정보를 통해 읽은 레지스터 값은 각각 A,B 레지스터에 저장하고 ALU에서 적절한 mux를 통해 연산을 진행한다. ALU에서 레지스터 값들만 연산하는 것이 아니라 next pc도 계산하기 때문에 2-input mux, 4-input mux가 필요하다. ALU에서 연산한 값은 ALUOut에 저장해서 이전 사이클에 계산된 값이 남아있도록 구현한다.

clock synchronous module: ControlUnit의 current\_state 업데이트 부분, cpu, Memory write 부분, pc, register file write 부분

asynchronous module: ALU, ALUControlUnit, ControlUnit의 control signal 설정 및 next\_state 정하는 부분, ImmediateGenerator, Memory read 부분, mux, register file read 부분

Top module> cpu.v

Reset, clk를 입력으로 받고 프로그램 종료 시 is\_halted를 set하여 출력한다.

위의 그림과 같이 각 모듈과 멀티플랙서 사이 wire를 연결했다. 또한 control signal을 위한 ControlUnit과 instruction에 맞는 상수를 만드는 ImmediateGenerator를 연결했다.

매 clock 마다 IR, MDR, A, B, ALUOut을 업데이트 한다. ControlUnit에서 ECALL 신호를 인지하면 RegisterFile에서 rf17값을 읽어와 그 값이 10인지 확인해 is\_haled를 output으로 내보낸다.

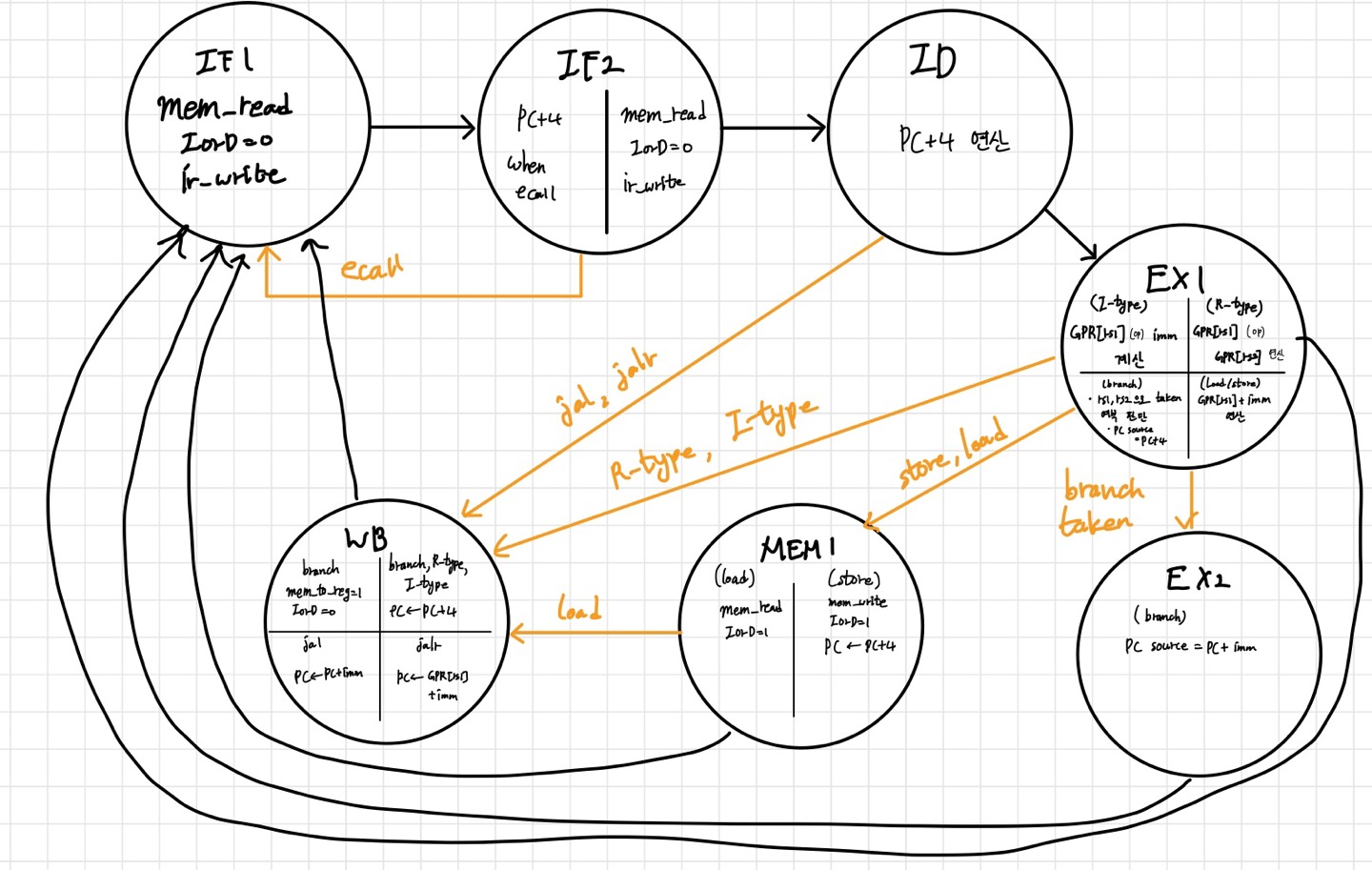
Submodules>

먼저 datapath에 포함되는 모듈들을 소개한다.

1. PC : positive clock마다 current\_pc를 업데이트 한다. 이때 reset이 들어오면 current\_pc를 초기화하고 pc\_write(pc에 write하는 것을 허용하는 signal)가 들어오거나 branch not taken이면서 pc\_write\_not\_cond가 참일 때만 pc 값을 업데이트 한다.
2. Memory: instruction과 data를 저장하는 모듈이다. Write 동작은 synchronous하게 하고 read 동작은 asynchronous하게 한다.
3. RegisterFile: cpu가 reset될 때, positive edge clokc에서 clock synchronous하게 32개의 레지스터 값을 초기화한다. 이후 rs1, rs2와 is\_halted 판단을 위한 register 17의 값을 asynchronous하게 읽는다. Rd에 저장할 입력값이 들어오면 clock synchronous하게 해당 레지스터에 값을 저장한다. Instruction decode를 위해 레지스터에서 값을 읽어온다.
4. ImmediateGenerator: instruction 종류에 따라 적절한 immediate value를 생성하는 모듈이다.
5. ALU: mux에서 결정된 두 값을 operand로 하는 연산 모듈이다. Operator에 따라 add, sub, sll, xor, or, and 등 다양한 연산을 진행한다.
6. Mux: 2개 또는 4개의 input 중에서 control signal에 따라 하나의 값을 output을 내보내는 모듈이다. 아래와 같은 모듈이 이번 실습에 사용됐다.
   1. mem\_selector: current pc와 ALUOut에 저장된 값 중 Memory 모듈에 address로 들어가는 input을 결정한다. 즉, pc에서 나온 instruction address와 ALU에서 연산된 data address 중 하나를 선택해 output으로 내보낸다.
   2. data\_to\_write: 레지스터에 적을 data를 선택하는 mux. 메모리에서 읽어온 데이터와 ALU에서 연산한 값 중 하나를 고른다.
   3. alusrcA\_selector: ALU에 operand로 들어가는 값 중 하나를 결정하는 mux. Current pc와 rs1에서 읽은 값 중 하나를 선택한다.
   4. alusrcB\_selector: ALU에 operand로 들어가는 값을 결정. 이 mux는 input 3개 중 하나를 선택해 output으로 내보낸다. 즉, rs2에서 읽은 값과 상수 4, immediate generator에서 생성한 상수, 상수 0 중 하나를 선택해 output으로 내보낸다.
   5. pcSrc\_selector: 이번 cycle에서 연산한 ALU 결과와 직전 cycle에서 연산한 ALU 결과 중 하나를 pc 값으로 선택한다.
   6. rs1\_selector: ECALL signal이 들어왔을 때, rs17에 있는 값을 불러오기 위한 mux. ECALL signal이 들어왔으면 rs1으로 17을, 아니라면 IR에 있는 rs1 값을 output으로 내보낸다.

이제 control과 관련된 모듈을 소개한다.

1. ALUControlUnit: instruction에서 opcode와 funct3 부분을 이용해 어떤 연산이 필요한지 판단하고, 이를 alu\_op로 내보내서 ALU에 보낸다.
2. ControlUnit: opcode를 받아서 그에 맞는 control signal을 발생시킨다. 이때, 한 instruction이라도 microcode control에 의해 state를 변경한다. 이렇게 instruction과 state에 따라 맞는 control signal을 생성한다. 아래는 이 유닛에서 내보내는 control signal과 대략적인 microcode control 구조다.
   1. pc\_write\_not\_cond: branch가 taken 되지 않은 것을 표현하는 신호
   2. pc\_write: pc에 값을 쓸 수 있는지 없는지를 표현하는 신호
   3. IorD: Memory 모듈 앞에 있는 mux를 control 함. Instruction address인지 data address인지를 알려주는 신호
   4. mem\_read: memory를 읽을 수 있다는 신호
   5. mem\_to\_reg: 메모리에서 읽어온 값을 레지스터에 쓸 때 발생하는 신호
   6. mem\_write: 메모리를 읽어도 될 때 발생하는 신호
   7. ir\_write: Instruction Register에 값을 써도 될 때 발생하는 신호
   8. reg\_write: 레지스터에 값을 써도 될 때 발생하는 신호
   9. pc\_source: pc값을 이번 cycle에 연산한 ALU 값인지 이전 cycle에서 발생한 값인지 알려주는 신호
   10. ALU\_op: ALU에서 진행할 연산 종류를 알려주는 신호
   11. ALU\_SrcB: ALU에서 operand로 사용할 값을 결정하는 mux의 control signal
   12. ALU\_SrcA: ALU에서 operand로 사용할 값을 결정하는 mux의 control signal
   13. is\_ecall: rf17을 확인하라는 control signal



위 그림은 microcode control 모습을 대략적으로 그린 그림이다.

나머지 파일에는 instruction이나 ALU에서 실행되는 operation 종류를 가리키는 상수를 정의해두었다.

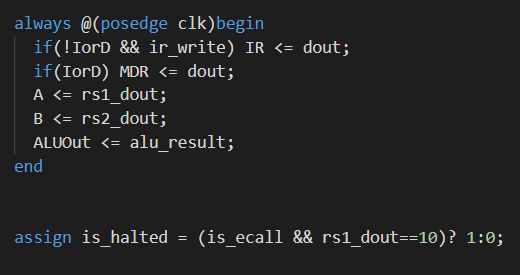
* AluOps.v: ALU에서 수행할 연산의 종류에 대해 상수 값을 정의함
* opcodes.v: RISC-V instrction들의 opcode, funct3 부분에 대해 상수 값을 정의함

1. Implementation: microcode controller, control unit, register file code를 스캔해서 넣기

Top module> cpu.v

Input: reset, clk

Logic: 2)의 그림에서 선언한 것과 동일한 이름으로 wire를 선언하고 PC, RegisterFile, Memory, ALU와 여러 레지스터를 연결한다. ECALL 신호가 발생했을 때, RegisterFile에서 17번 레지스터의 값을 읽어와 그 값이 10인지 확인하고 is\_halted를 set하여 모듈을 종료한다. 또한, 매 cycle마다 PVS의 값을 저장하는 레지스터들의 값을 업데이트 한다.



Output: is\_halted

CPU에서 돌린 프로그램이 종료되었음을 나타내는 신호를 출력한다.

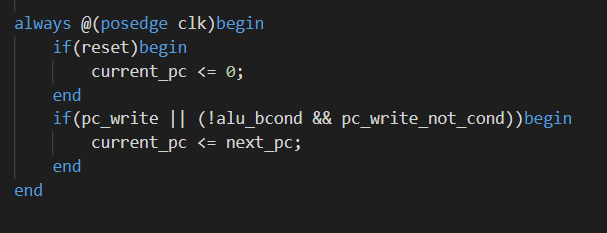
Submodules>

1. PC

Input: reset, clk, pc\_write, alu\_bcond, pc\_write\_not\_cond, next\_pc

Logic: 매 positive clock edge마다 pc\_write가 1이거나 alu\_bcond이 0이고 pc\_write\_not\_cond가 1인지 확인하고 current\_pc를 next\_pc로 업데이트 한다. 이때 reset이 1이면 current\_pc를 0으로 초기화한다.

Output: current\_pc



1. Memory

Input: reset, clk, addr, din, mem\_read, mem\_write

Logic: 메모리에 대해 asynchronous read, synchronous write한다. Positive clock edge일 때 reset이 1이면, 메모리를 초기화하고 파일에서 instruction을 읽어온다.

Output: dout에 읽은 명령어를 저장해서 내보낸다.

1. RegisterFile

Input: reset, clk, rs1, rs2, rd, rd\_din, write\_enable

Logic: register에 대해 asynchronous read, synchronous write 한다. Positive clock edge에서 reset이 1이면 모든 register를 초기화한다.

Output: rs1 레지스터의 데이터 rs1\_dout, rs2 레지스터의 데이터 rs2\_dout

1. ImmediateGenerator

Input: inst

Logic: Single-cycle CPU와 동일한 code를 사용했다. opcode를 이용해 instruction 종류를 확인하고 이에 맞는 상수를 sign-extend하여 32비트 상수를 발생시킨다.

Output: imm\_gen\_out

1. ALU

Input: alu\_op, alu\_in\_1, alu\_in\_2

Logic: single-cycle CPU와 동일한 code를 사용했다. alu\_op가 instruction의 종류를 알려준다. Branch instruction이면 이에 맞는 연산을 통해 branch taken, not taken을 결정하는 alu\_bcond를 생성한다. Arithmetic operation이면 operand인 alu\_in\_1, alu\_in\_2를 이용해 alu\_result를 연산한다.

Output: alu\_result, alu\_bcond

1. Mux
   1. mux2

Input: mux\_in1, mux\_in2, control

Logic: mux\_out에 control이 0이면 mux\_in1을, control이 1이면 mux\_in2를 assign한다.

Output: mux\_out

1. mem\_selector

input: current\_pc, ALUOut(alu에서 계산한 data address)

control: IorD

output: accessMem / 메모리에 접근하는 주소가 무엇인지 내보냄

1. data\_to\_write

input: ALUOut, MDR

control: mem\_to\_reg

output: writeData / alu 연산 결과와 메모리에서 읽은 데이터 중 레지스터에 쓸 데이터를 내보냄

1. alusrcA\_selector

input: current\_pc, A

control: ALU\_SrcA

output: alu\_in\_1 / pc와 레지스터에서 읽은 A 중 하나를 alu operand로 내보냄

1. pcSrc\_selector

input: alu\_result, ALUOut

control: pc\_source

mux\_out: next\_pc

1. rs1\_selector

input: IR[19:15], 17

control: is\_ecall

output: rs1 / ecall이 발생했을 때, 17을 rs1에 입력해 레지스터를 읽어옴

* 1. mux4

Input: mux\_in1\_1, mux\_in2\_1, mux\_in3, mux\_in4, control

Logic: control이 00이면 mux\_in1\_1, 01이면 mux\_in2\_1, 10이면 mux\_in3, 11이면 mux\_in4를 내보낸다.

Output: mux\_out

1. alusrcB\_selector

input: B, 4, imm\_gen\_out, 0

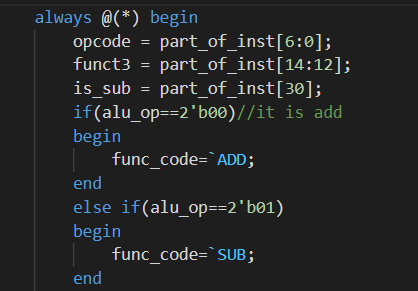
control: ALU\_SrcB

output: alu\_in\_2

1. ALUControlUnit

Input: part\_of\_inst, alu\_op

Logic: single-cycle cpu와 동일한 logic을 갖는다. Single-cycle cpu에서 사용한 코드에서 load, store, branch와 같은 instruction에서 주소를 연산할 때 필요한 add를 따로 지정해주는 아래 부분을 추가하고 나머지 코드는 동일하다.



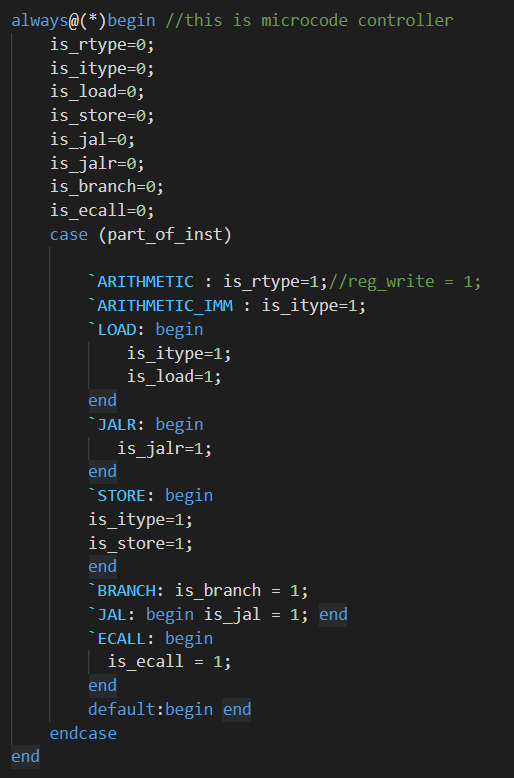
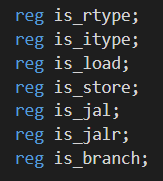
Output: func\_code

1. ControlUnit

Input: clk, reset, part\_of\_inst, alu\_bcond

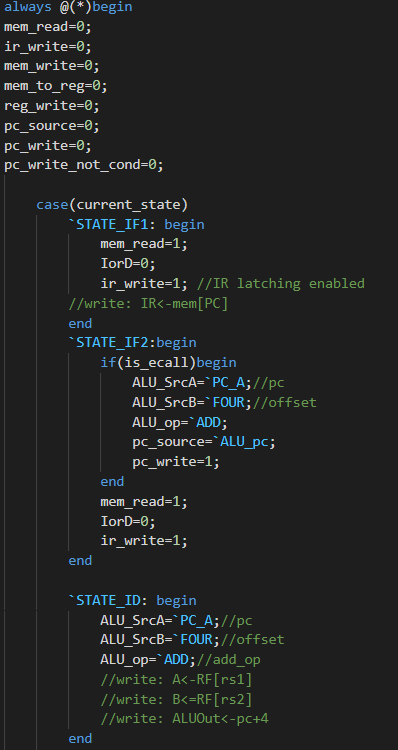
Logic: 이 모듈은 4개의 always 문이 있다.

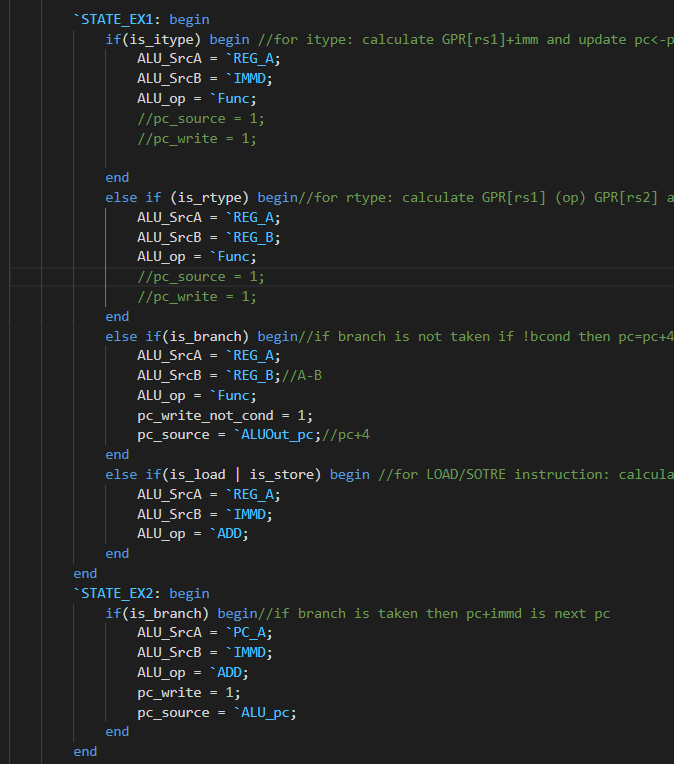
첫번째 always 문은 아래와 같다.

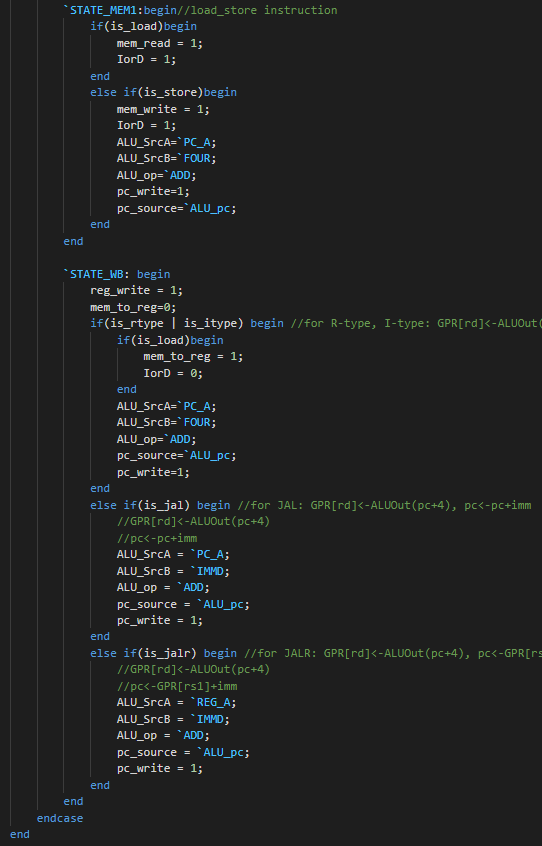
 

여기서는 input으로 받은 part\_of\_inst를 이용해 instruction 종류를 파악하고 이를 위에서 선언한 reg에 저장한다.

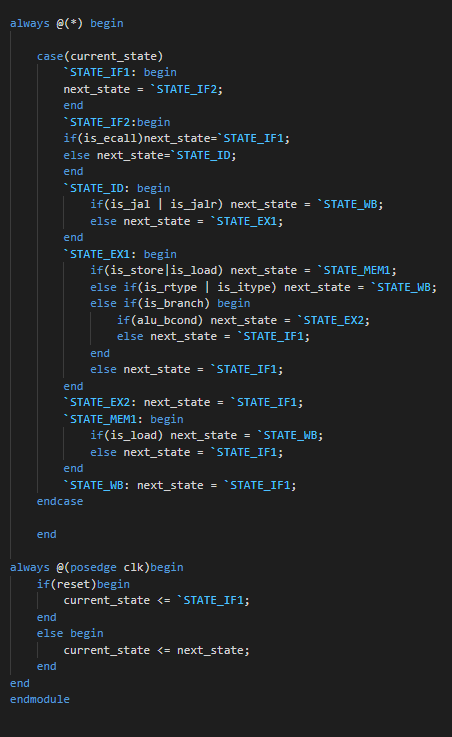
두번째 always 문은 아래와 같다. State와 instruction 종류에 맞는 control signal을 만든다. 대략적인 그림은 2) Design에서 ControlUnit 설명에 첨부한 그림과 같다.



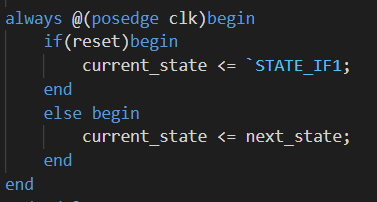




세번째 always 문은 next\_state를 업데이트 하며 아래와 같다. State 업데이트 흐름은 2) Design의 ControlUnit 설명에 첨부한 그림과 같다.

****

네번째 always 문은 synchronous하게 state를 업데이트한다.



Output: pc\_write\_not\_cond, pc\_write, IorD, mem\_read, mem\_to\_reg, mem\_write, ir\_write, reg\_write, pc\_source, ALU\_op, ALU\_SrcB, ALU\_SrcA, is\_ecall

1. Discussion

Resource reuse design and implementation

Microcode controller state design

1. Conclusion

테스트로 basic\_mem을 사용하면 143 cycle이 나온다.



테스트로 loop\_mem을 사용하면 1198 cycle이 나온다.

