**CSED311 Lab 3: Multi-Cycle CPU**

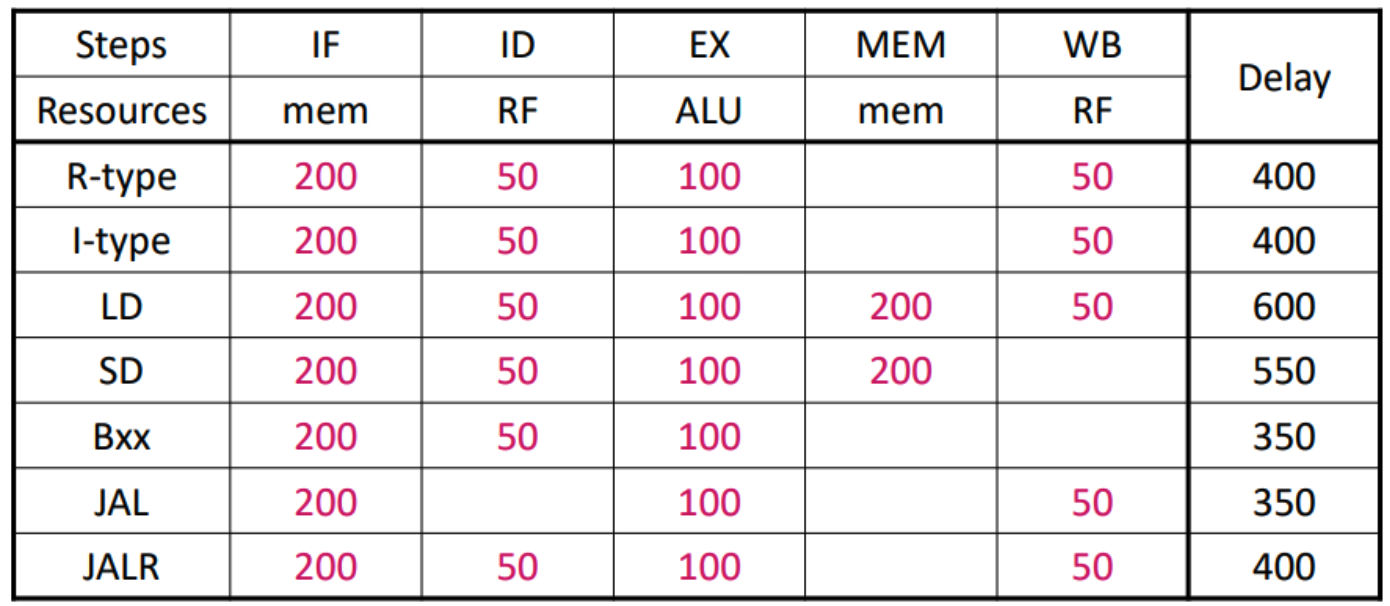
Team number 3

20210643 김현준

20210273 하태혁

1. Introduction

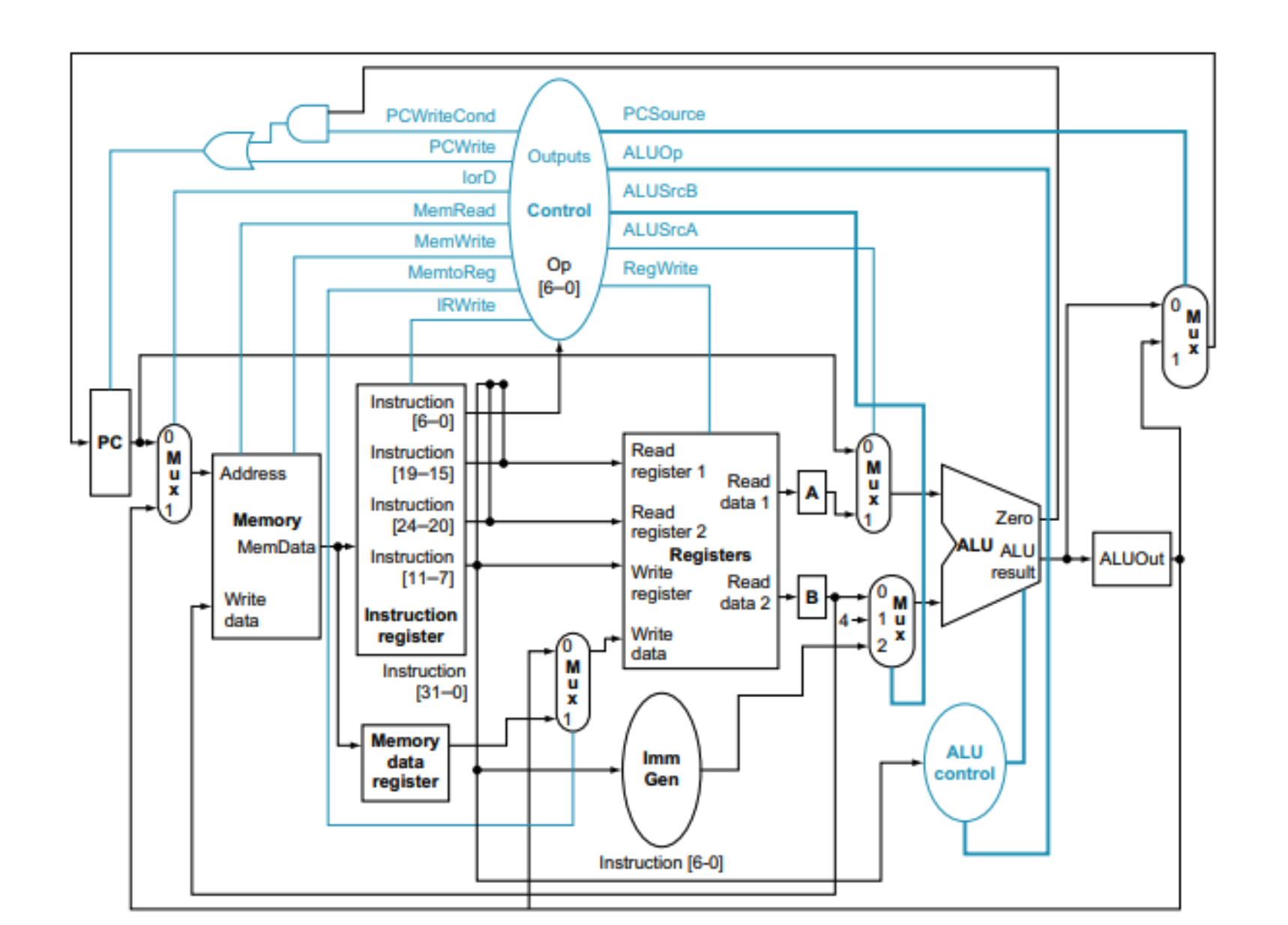
Multi-cycle RISC-V CPU는 RISC-V 명령어를 여러 사이클(Cycle)에 걸쳐 수행하는 CPU의 구현 방법 중 하나이다. 단일 사이클만을 사용하는 Single Cycle CPU는 1개의 사이클 동안 1개의 인스트럭션을 수행할 수 밖에 없기 때문에 가장 느린 인스트럭션을 기준으로 수행되는 한계가 존재한다. 이와 달리 Multi-cycle CPU의 구조는 높은 Clock frequency를 바탕으로 인스트럭션 종류에 따라 서로 다른 사이클 수를 가질 수 있으므로, 간단한 인스트럭션을 더 빠르게 처리하여 전체적인 CPU의 성능을 높일 수 있다.



특히 몇몇 인스트럭션은 모든 단계를 거치지 않고도 결과를 얻을 수 있다. Sequential Control 에서 주어진 state들과 signal에 따라 순서대로의 모든 State를 거치지 않고도 결과를 도출할 수 있다.

1. Design

해당 Multi-cycle CPU의 디자인을 바탕으로 하여 전체 CPU의 Datapath와 Control Unit을 디자인할 수 있었다. 기존 Single-cycle CPU에서 사용되는 동일한 adder를 줄이는 Resource reuse와 메모리가 읽기 전의 인스트럭션을 저장하는 Instruction Register(IR)을 사용하는 Sequential reuse를 통해 Datapath를 줄일 수 있다. 그리고 해당 시점의 신호를 전달하고 관리하기 위한 Control Unit을 정의하고 각 모듈들과 연결할 수 있도록 하였다.



cpu.v에서는 사용되는 모듈을 전부 확인할 수 있으며, Single-cycle CPU와 달리 MUX 등도 모두 모듈로 만들고 사용할 수 있도록 하였다. Datapath에 관여하는 대부분의 모듈은 module.v에 존재하며, Control Unit과 이에 관여하는 Microcode controller는 ControlUnit.v라는 별도의 파일에 정의하고 사용하였다.

1. Implementation

실행에 있어서 다음과 같은 조건을 만족시켜야 한다.

* Control Unit은 well-implemented state machine으로 구상하여야 하며, 각 state마다 그에 알맞은 control signal을 계산하고 출력할 수 있어야 한다.
* Datapath와 Control Unit에 해당하는 모든 circuit은 clock-synchronous하게 작동해야한다. 각 module에서 다음 동작을 수행하기 전에 이전 동작이 모두 완료되어야 데이터가 불일치하는 경우를 막을 수 있으며, 특정 타이밍에 맞게 명령어의 처리를 수행할 수 있다.
* 모든 데이터 저장 Unit은 Clock’s positive edges하게 동작해야 한다.
* Resourse reuse를 사용해야 하며, 이를 통해 불필요한 Datapath를 줄이고 실행 시간을 단축시킬 수 있다.

위 조건을 만족시키는 각 모듈에 대한 설명은 아래와 같다.

1. Program Counter

Single-cycle CPU와 동일한 기능을 수행하지만, Multi-cycle CPU에서는 pc\_write 값이나 alu\_bcond 값이 1인 경우에 current pc가 next pc 값는 동작을 추가적으로 수행해야 한다. 이를 구현하기 위해 pc\_control이라는 변수를 추가하였으며, 매 클럭마다 동작하여 인스트럭션 값을 확인해야 하므로 clock synchronous하게 동작한다.

1. Immediate Generator

Single-cycle CPU의 모듈과 동일하게 동작한다. 인스트럭션의 각 비트를 분석하여 I, S, B, J-type을 판단하고 imm\_gen\_out을 통해 출력 값을 내보낸다. 각 계산은 입력 시 한번에 진행되면 되므로, Clock과 관련 없이 Asynchronous하게 진행된다.

1. ALU Control Unit

인스트럭션 종류에 따라 ALU가 수행하게 될 연산의 종류를 결정하여 alu\_op로 내보낸다. Single-cycle CPU와 달리 Control Unit 으로부터 받은 ALUop 값이 존재하며 해당 값까지 모두 고려하여 Asynchronous하게 출력 값을 결정한다.

1. ALU

Asynchronous하게 산술 및 논리 연산을 수행하며, ALU Control Unit의 지시에 따라 연산을 수행한 후 결과를 레지스터 파일에 저장한다. 기존 Single-cycle CPU에서는 ALU가 단독적으로 사용되었지만, Multi-cycle CPU에서는 Adder에 대한 Resource reuse 디자인을 사용함으로써 퍼포먼스를 향상시킬 수 있었으며, Adder의 기능을 수행하는 신호를 파악하기 위해 Control Unit에서 출력 신호를 받아서 이를 인식하게 하였다.

1. Register File

Clock synchronous 하게 진행되며, 진행 결과에 따른 rf 배열의 값을 rs1\_dout과 rs2\_dout에 나누어 출력한다. Write\_enable이 1인 경우는 destination register의 값을 저장한다.

1. Memory

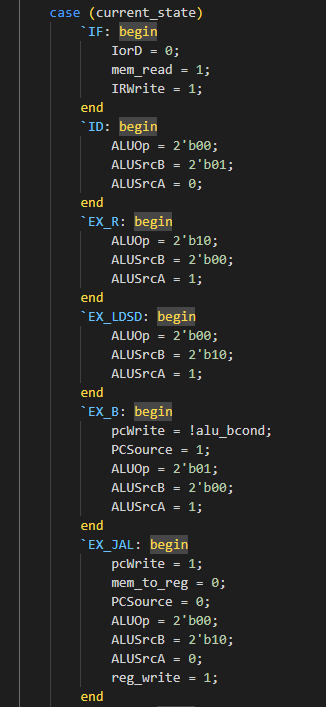
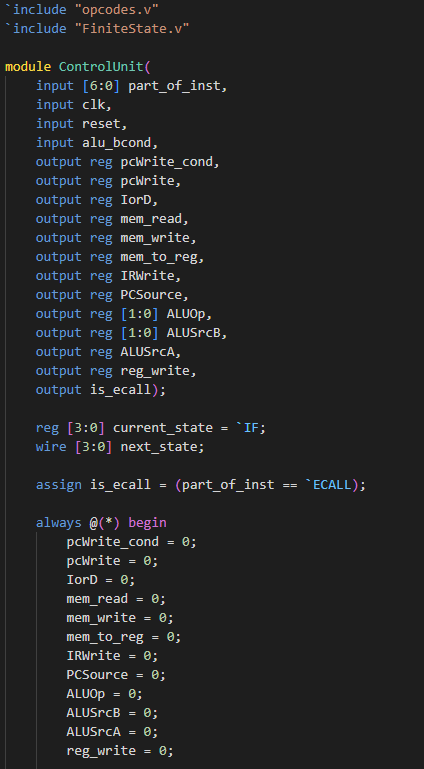
주소 값의 데이터를 읽고 dout을 통해 결과를 내보내는 식으로 동작한다. Single-cycle CPU와 달리 해당 모듈에서 인스트럭션을 읽고(mem\_read) 쓰는(mem\_write) 동작을 모두 수행한다. Reset 값이 1일 때, 경로에 있는 인스트럭션의 포멧을 mem 배열에 저장한다. 이후 Clock synchronous하게 mem 배열의 해당 인덱스 값에 din 값을 입력한다. Memory에서 데이터를 읽는 과정은 asynchronous하게 진행된다.

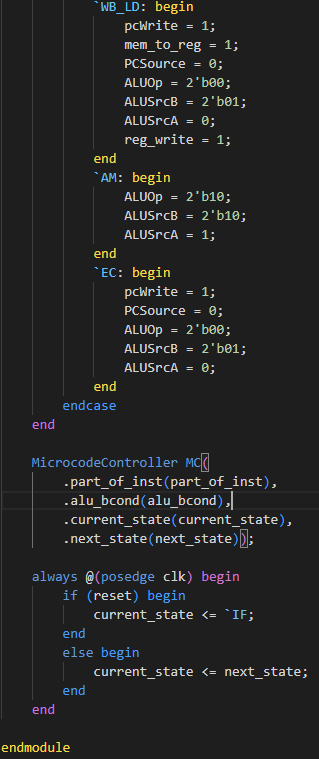
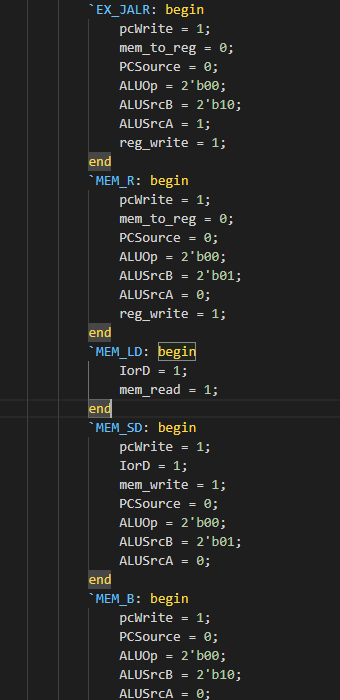
1. MUX

1-bit, 2-bit MUX가 존재하며, Design에 첨부한 그림의 모듈 부분에서 MUX가 사용되는 부분에 적용할 수 있도록 하였다.

1. Control Unit

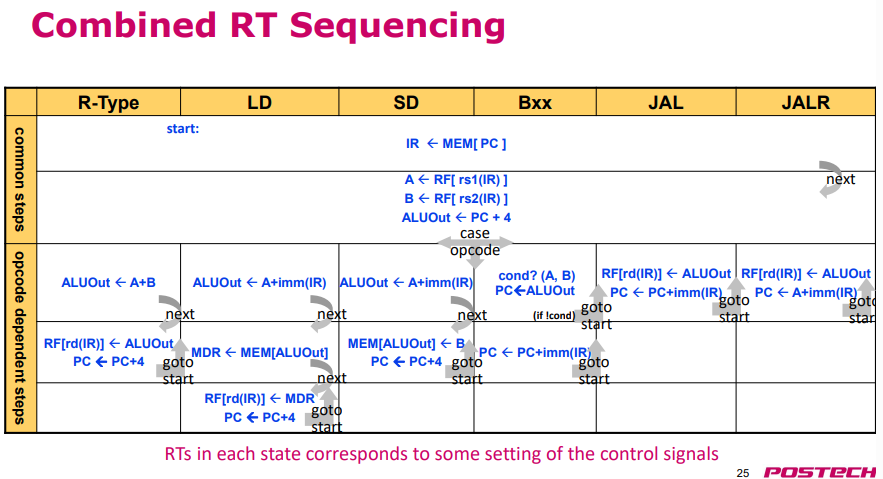
input으로 7비트 part\_of\_inst(opcode) clk, reset, alu\_bcond을 받고, output으로 control 신호들(pcWrite\_cond, pcWrite, IorD, mem\_read, mem\_write, mem\_to\_reg, IRWrite, PCSource, ALUOp, ALUSrcB, ALUSrcA, reg\_Write)을 출력하는 모듈이다. 추가로 is\_ecall 신호로 ecall 신호를 관리하여 프로그램을 종료한다. Is\_ecall 신호는 opcode가 ECALL에 해당하는 경우 1이 되도록 asynchronous하게 처리해 주었다. Control 신호 관리는 0~13 총 14개의 state를 이용하는 Finite State Machine으로 구현하였다. MicorcodController module이 FSM을 관리하며, part\_of\_inst와 alu\_bcond, current\_State를 받아 next\_state를 출력하는 기능을 한다. MicrocodeController 모듈의 결과에 따라, always @(posedge clk) 부분에서 synchronous하게 current\_state를 next\_state로 바꾸어주는 작업을 하며, current\_state에 따라서 각 control 값들을 asynchronous하게 설정해주게 된다. 이로써 현재 state에 따라 알맞은 control signal datapath를 보내는 ControlUnit module을 구현하였다. FSM과 관련해서는 바로 다음 MucrocodeController 부분에서 설명하겠다. 아래는 controlunit 모듈의 코드 내용이다.



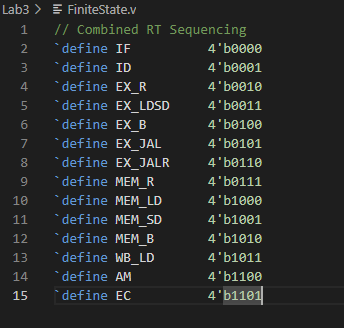


1. Microcode Controller

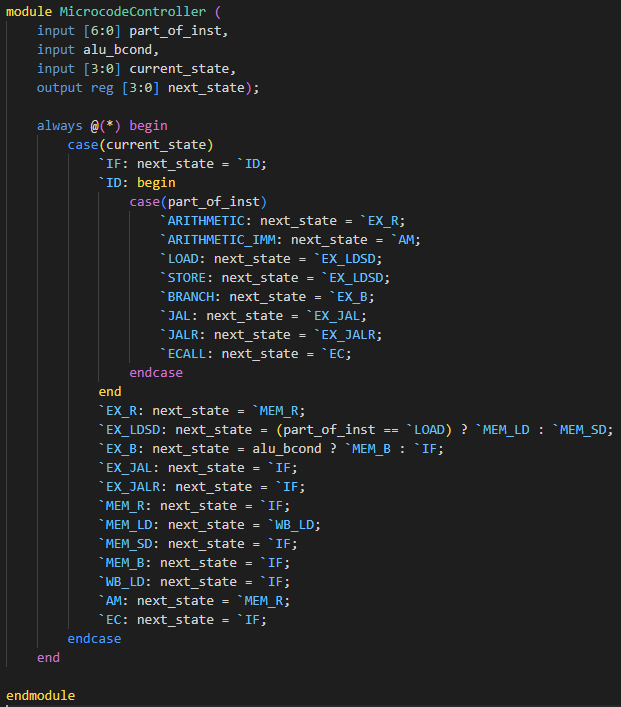
본 모듈은 ControlUnit module의 일종의 submodule이며, FSM을 관리하여 current\_state와 part\_of\_inst, alu\_bcond에 따라 next\_state를 output으로 보내는 역할을 하는 모듈이다. 위에서 state 개수를 총 14개로 잡았다고 했는데, Lecture Slide에서 아래 표에 따라서 state를 설정하였다.



먼저 맨 위 IR <- MEM[PC] 부분을 0, 그 아래 state를 1, 그 아래 opcode dependent steps에서 왼쪽부터 오른쪽, 위에서 아래로 쭉 번호를 매겨 0~11의 state를 설정하였다. 다만 opcode dependent step에서 LD와 SD가 똑같이 ALUOut<-A+imm(IR)을 수행하는 부분은 같은 3번 state로 취급하였다. 나머지 12와 13번 state의 경우는 표에서 나오지 않은 Arithmetic imm type (I-type)와 Ecall을 각각 처리해주기 위해 설정하였다. 이렇게 설정한 state 정보를 FiniteState.v에 정리하였고, 이 파일을 include하여 ControlUnit.v 파일에서 define된 기호를 사용하였다. 아래는 FiniteStae.v의 내용이다. 표에서 가로 축의 instruction, 세로 축을 편의상 IF, ID, EX, MEM, WB으로 생각하여 각 state를 아래와 같이 작명하여 define해주었다.



이렇게, 정리한 state 번호에 따라, 그리고 current\_state와 opcode, alu\_bcond에 따라 next\_state를 결정할 수가 있게 되었고, 이를 MicrocodeController 모듈에 asynchronous하게 아래와 같이 구현할 수 있었다. Current state에 따라 case문으로 구현하였고, IF인 경우 바로 ID로, ID인 경우 opcode에 따라서 각 state로, 그리고 나머지 state들은 위의 표와 같이 opcode와 bcond 신호를 이용하여 다음 state를 설정해주었다. 또한, 추가된 AM과 EC의 경우도 AM은 바로 MEM\_R으로, EC의 경우는 초기 상태인 IF로 이동하도록 구현해주었다.



1. Discussion

각 모듈에 필요한 wire와 register를 선언하는 부분에서 is\_harted을 고려하지 않고 연산을 수행했기 때문에, 시뮬레이션에서 시간이 무한대로 나오고 종료되지 않는 경우가 많았다. 해당 연산을 처리하기 위해서 Register File에서 x17 레지스터 값을 저장하는 wire를 선언하고 이후 control unit에서 is\_ecall 값을 계산하여 ecall을 호출하는 경우를 만들어 에러를 해결할 수 있었다.

ControlUnit 구현에서도 약간의 어려움이 있었다. 초기 버전에서는 따로 Microcontroller를 두지 않고 하나의 모듈에서 구현을 시도하였는데, 계속해서 state 전이가 중간에 멈추고 halt되지 않는 이슈가 발생하였고, 결국에는 이를 해결하지 못하여 microcontroller가 FSM을 관리하는 현재의 버전으로 수정하였다. 수정한 결과 코드가 훨씬 직관적이게 되었고, 추가로 FiniteState.v 헤더 파일을 만들어 각 state를 define해준 결과 가독성 높은 코드가 되었다. 따라서 디버깅이 쉬워졌고, 적절히 작동하는 controlunit 모듈을 구현 완료할 수 있었다.

5개의 testbench 파일을 모두 돌려본 결과 적절한 최종 RF 값들을 보이는 것을 확인할 수 있었다. 아래는 5개 testbench의 cycle 수이다.

1. Basic\_mem: 116 cycles



1. Non-controlflow\_mem: 157 cycles



1. Loop\_mem: 977 cycles



1. Recursive\_mem: 3686 cycles



1. Ifelse\_mem: 139 cycles

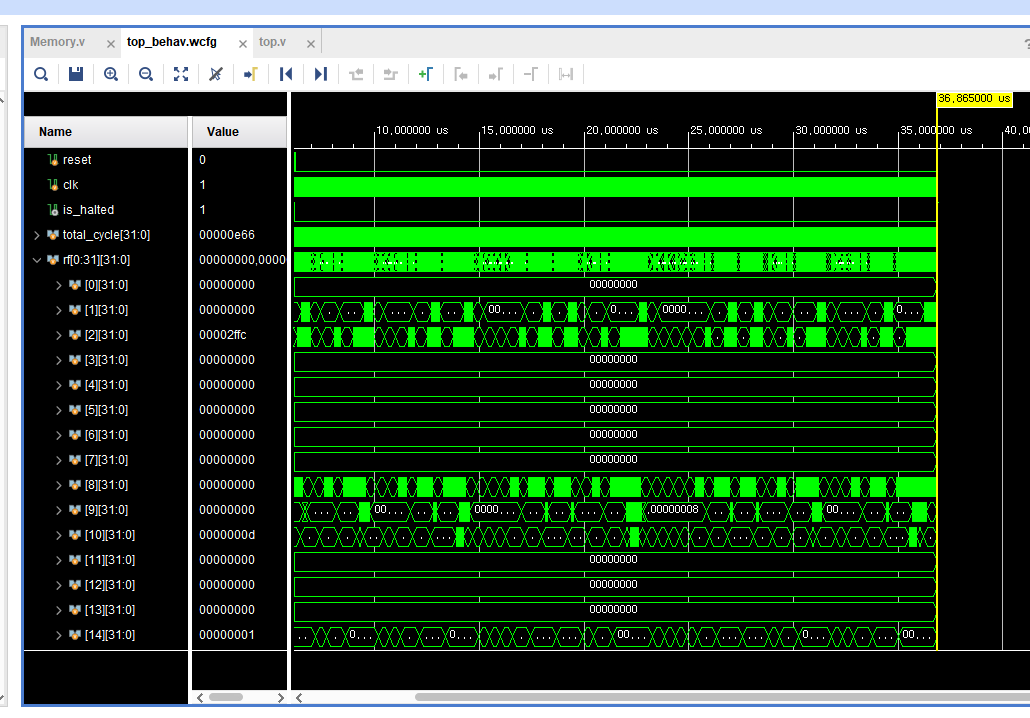


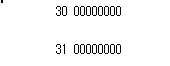
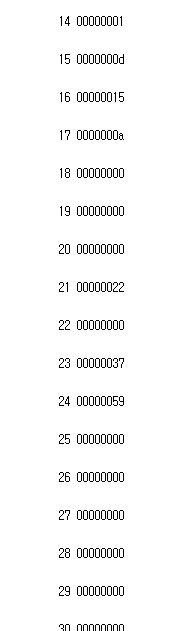
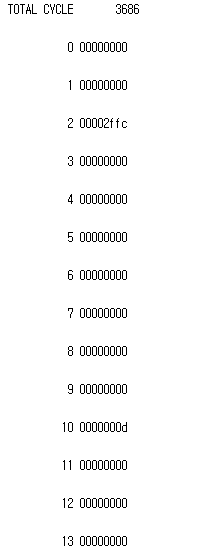
추가로, single cycle cpu에 비해 성능이 얼마나 향상되었는지 체크해보고자 하였고, 체크를 위한 척도로 single cycle cpu의 clock 수 \* 5의 값보다 얼마나 multi-cycle cpu의 clock 수가 작은지 확인해보고자 하였다. 확인 결과 모두 single cycle cpu의 clock\*5의 값보다 작은 cycle 수를 보였다. (예: non-controlflow 기준 single: 39cycle, multi: 157cycle => 39\*5=195, 195/157=1.242의 성능 향상, loop 기준 single: 222cycle, multi: 977cycle => 222\*5=1110, 1110/977=1.136의 성능 향상)

1. Conclusion

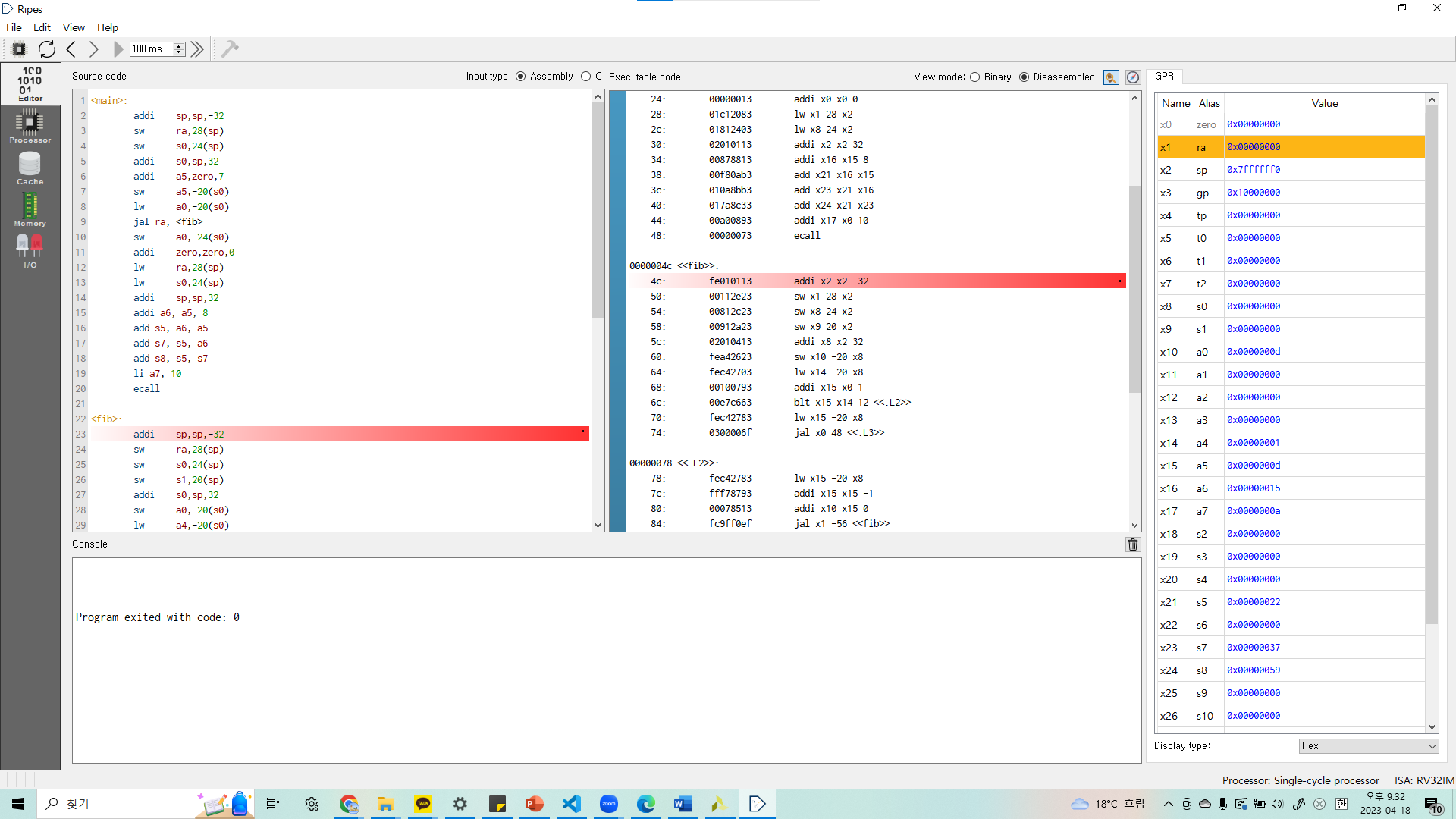
보고서 내용과 같이 구현한 결과, 주어진 5개의 testbench 파일(non-controlflow\_mem.txt, basic\_mem.txt, loop\_mem.txt, recursive\_mem.txt, ifelse\_mem.txt) 모두 적절히 작동하는 것을 확인할 수 있었다. Register File의 값이 같으면 정확히 작동한 것으로 생각하였고, 이전 Lab에서 구현하였던 Single-Cycle CPU와 주어진 Ripes 프로그램, Multi-Cycle CPU의 최종 RF 결과값을 5개 testbench에 대해 비교해본 결과 일치하는 것을 확인할 수 있었다. Ripes 프로그램의 경우 아쉽게도 Multi-Cycle CPU로 돌리는 것은 불가능해 cycle 수 비교는 해볼 수 없었다. 다만 구현 방식에 따라 cycle 수는 달라질 수 있기 때문에 RF파일 위주로만 점검해보았다.

추가로, 확인할 수 있었던 점은 single Cycle CPU에 비해 Multi Cycle CPU의 Total Cycle 수가 적게는 3~4배, 많게는 4~5배 정도 많았던 것이었다. 이는 이번에 구현한 Multi-cycle CPU가 IF, ID, EX, MEM, WB를 각 1 사이클씩으로 하여 돌아가는 것이기 때문에 각 instruction에 따라 4~5배 정도 cycle 수 자체는 많아질 수 있기 때문이었고, 실제 작동에서도 이것을 확인할 수 있었다. 다만 clock cycle 수는 많아도 clock frequency가 훨씬 빨라질 수 있기 때문에 결과적으로는 Multi-Cycle CPU의 성능이 좋을 것이다. 아래는 시뮬레이션 결과의 일부이다(recursive\_mem.txt).





아래는 Ripes로 recursive\_mem.txt를 돌려본 결과이다. 초기 2번 register 세팅이 다른 점을 감안했을 때, 나머지 부분은 모두 일치함을 확인할 수 있었다.



이렇게, 주어진 skeleton code를 바탕으로 Multi-Cycle CPU를 구현함으로써 Single-Cycle CPU와의 차이점을 느낄 수 있었다. 또한, testbench를 통해, Total Cycle 수 자체는 늘어났지만 성능은 multi-cycle cpu가 single-cycle cpu에 비해 향상되었음을 체크해볼 수 있었다.