**CSED311 Lab 4-1: Pipelined CPU**

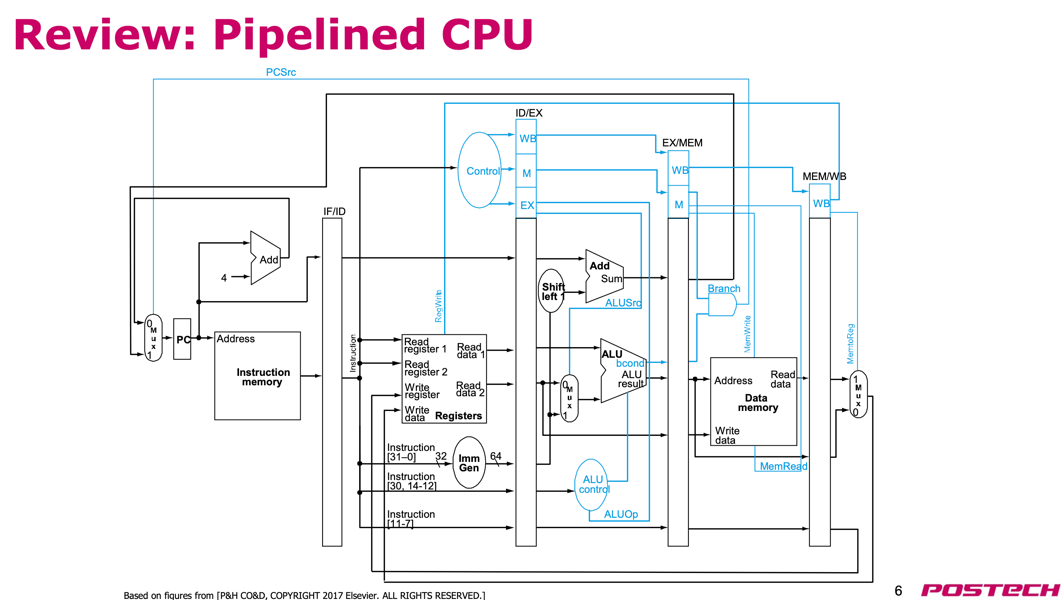
Team number 3

20210643 김현준

20210273 하태혁

1. **Introduction**

Pipelined CPU는 CPU의 throughput, 성능 및 처리량을 향상시키기 위해 5개의 Stage를 각각 병렬적으로 처리하는 기법이다. CPU가 인스트럭션을 처리하는 가장 기본적인 단계인 IF, ID, EX, MEM, WB가 동시에 시행되는 하드웨어 구조를 갖기 위해 각 파이프라인 단계에서 레지스터들이 필요하게 되며, 이는 IF/ID register, ID/EX register, EX/MEM register, MEM/WB register로 나뉜다. 아래 사진은 수업 시간에 다룬 파이프라인 CPU 디자인의 핵심을 알아볼 수 있는 디자인이다. 아래는 강의 노트에서의 pipelined CPU의 개요이다.

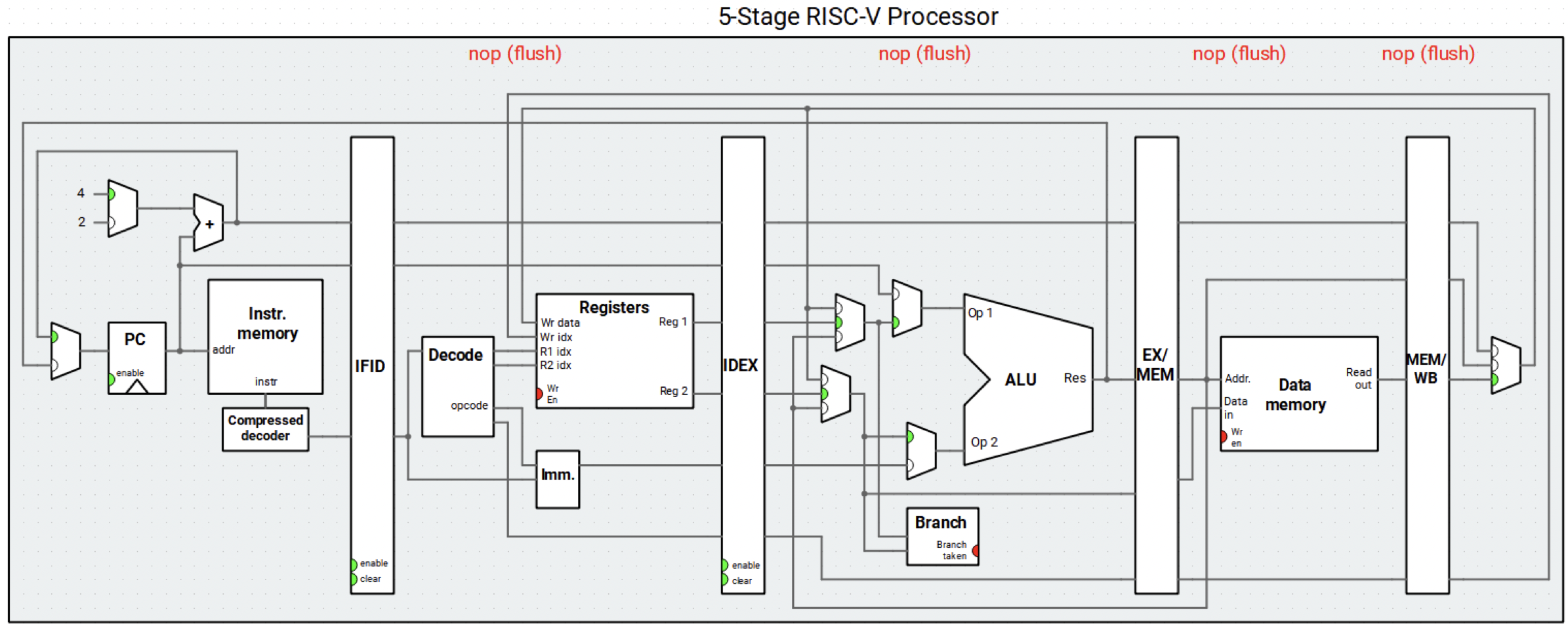


그러나 파이프라인 CPU에서는 각 Stage가 다른 인스트럭션을 병렬적으로 처리하기 때문에 발생하는 문제가 몇 가지 존재하게 된다. 가장 대표적인 문제로는 Structural Hazard, Data Hazard, Control Hazard가 존재하며, 이로 인해 성능이 감소하게 된다. 위 문제들을 해결하기 위해 CPU는 값이 계산되면, WB 전에 즉각적으로 업데이트 하는 등의 Data Forwarding, Branch Prediction 등의 기법이 고안되었다. 이번 Lab4-1에서는 Data Forwarding에 대해서만 다룬다.

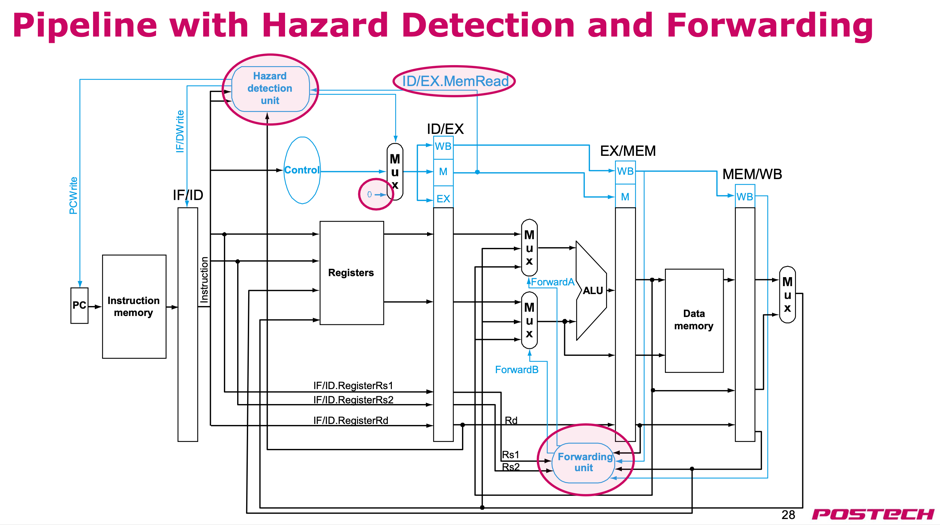
이러한 점들을 고려하여 본 Lab 과제에서는 5 stage pipelined CPU를 구현하고, Data Forwarding까지 구현하여, 이전에 구현했던 CPU보다 성능이 향상된 효율적 CPU를 Verilog로 구현해 보았다.

1. **Design**

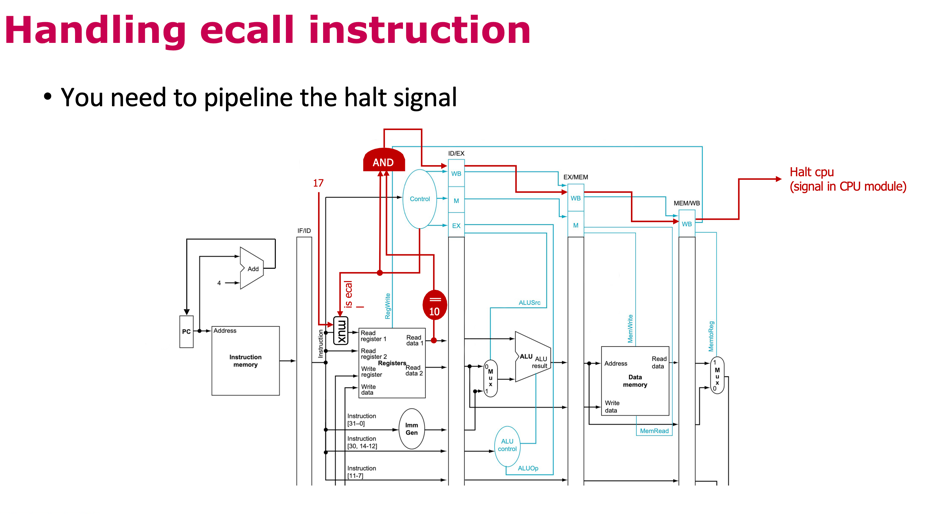
Lips에서 확인할 수 있는 파이프라인 CPU 디자인을 참고하여 Pipeline Stage를 디자인하였으며, Data forwarding은 수업 시간에 배운 자료를 참고하여 모듈을 디자인하였다. Ecall에 대한 처리는 Lab4 자료에 나와있는 디자인을 참고하였다.



각 Step에서 모듈 별로 인스트럭션을 처리하고 각 파이프라인 레지스터를 업데이트하는 non-block 구문이 존재하며, reset 시그널이 발생할 시에는, 0으로 초기화 되도록 구상하였다.



의존성이 존재하는 인스트럭션 사이에서 ALU로 먼저 계산이 완료되는 값들은 다음 EX에서 참조할 수 있도록 EX/MEM, MEM/WB의 값을 EX로 전달하는 Forwarding Unit을 제작하였으며, Forwarding A, B가 각각 시그널로 동작하여 MUX를 통해 ALU로 값이 전달될 수 있도록 하였다. Hazard가 발생하였을 때에는 Hazard Detection Unit에서 이를 인식하고 Stall을 생성할 수 있도록 한다.



의존성이 존재하는 두 인스트럭션에 대해 Dist(i1, i2) 값이 3인 경우에는 EX/MEM으로부터 값을 불러와서 ID에 반영해야 하므로 이는 Forwarding Ecall 모듈에서 처리할 수 있도록 하였다. 특히, Ecall과 의존성이 있는 바로 이전 인스트럭션이 사용되는 경우도 마찬가지로 Data Forwarding이 일어날 수 있도록 하였다.

1. **Implementation**

다음 질문에 만족하는 상황을 생각하며, CPU를 디자인하였다.

* How does your pipelined CPU work?

Data Forwarding을 할 수 있는 Forward 모듈을 구현하고 레지스터 조건에 맞게 값이 전달될 수 있도록 구현하였다. 자세한 설명은 아래 모듈 설명에 첨부하였다.

* How to implement hazard detection?

Hazard detection 모듈을 구현하였고 각 레지스터 조건을 비교하여 data hazard가 발생하였다고 판단되면, stall을 생성하여 인스트럭션의 진행을 막을 수 있게 구현하였다.

* How to implement data forwarding?

Dist(i1,i2) = 1, 2, 3에 따라 수행되는 동작이 다르기 때문에 모듈을 구별하여 dist = 1,2 일때 forwarding을 수행하는 모듈과 dist = 3일때 forwarding을 수행하는 모듈을 제작하였다.

파이프라인 CPU에 사용된 모듈에 대한 설명은 다음과 같다.

1. Program Counter

매 사이클 마다 수행되며, Next\_pc의 값은 Current\_pc + 4로 계산되어 반영된다. Hazard 발생시에 이 과정이 중단되어야 하기 때문에, 이를 나타내는 값이 is\_hazard를 signal로 사용하여, 이에 따라 값이 적용될 수 있도록 하였다.

1. Instruction Memory

Skeleton code와 함께 주어진 모듈로, Instruction memory를 담당하는 모듈이다. 입력으로 reset, clk, 그리고 instruction 주소를 나타내는 addr이 들어가고, output에는 addr에 해당하는 32bit instruction이 나오게 된다. 내부에서는 mem에 readmemh으로 test 파일의 instruction positive clock edge에 따라 reset이 1일 초기에 initializing이 되고, 추후에는 asynchronous하게 해당 주소의 instruction을 출력한다.

1. HazardDetectionUnit

Lab4-1에서 구현한 CPU는 Data Forwarding까지만 구현되어 있기 때문에, 특수한 경우에서는 Hazard가 발생하게 된다. 먼저, 의존성이 있는 두 인스트럭션에서 앞의 인스트럭션이 Load이고 뒤의 인스트럭션이 Load의 dest register를 사용하는 경우 1 stall이 발생하게 된다. 또한, Ecall이 호출된 이후 의존성이 존재하는 인스트럭션이 사용되거나 x17을 dest register로 사용하게 되면, Hazard가 발생하게 된다.

1. mux\_for\_is\_ecall

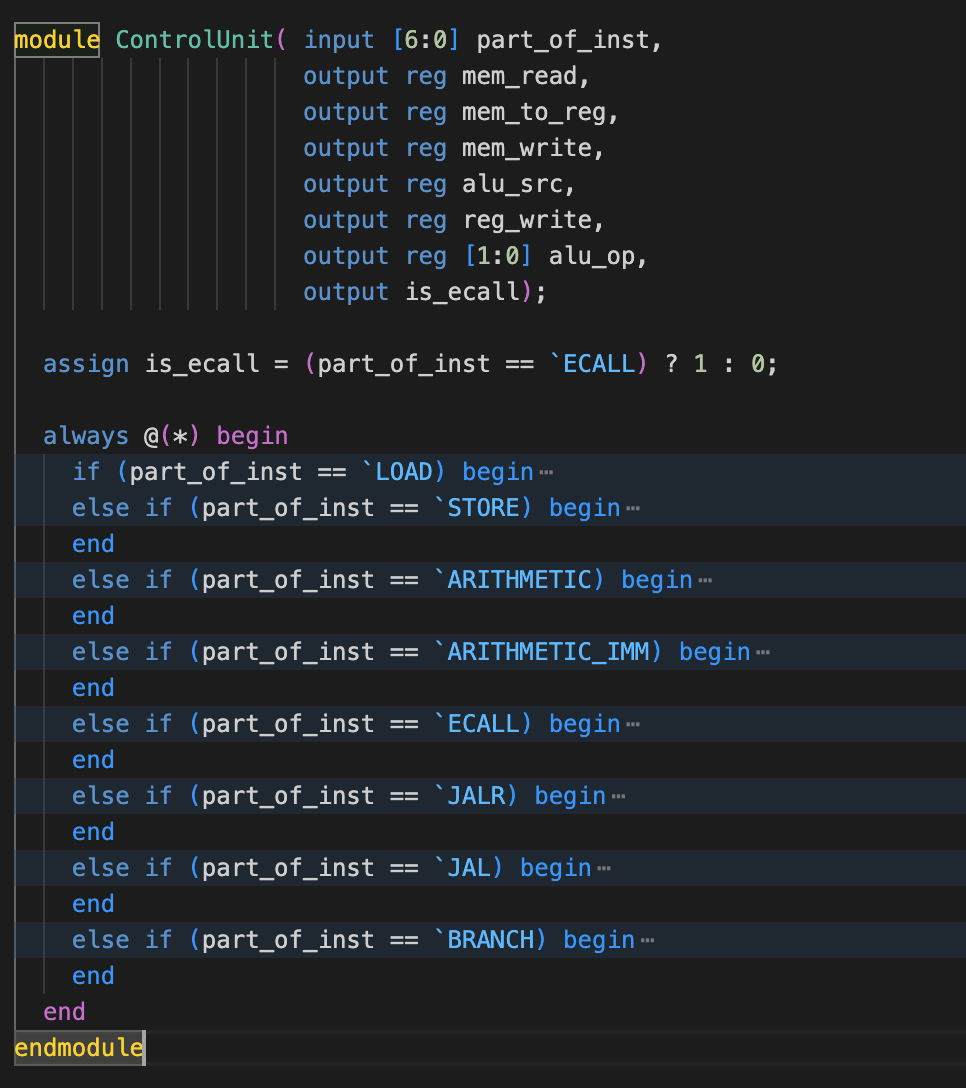
Mux이며, Ecall이 호출되면, 17이 레지스터 파일의 input인 rs1으로 입력될 수 있도록 한다.

1. Register File

Skeleton code와 함께 주어진 모듈로, CPU의 General Purpose Register들을 관리하는 모듈이다. 입력으로는 register unit의 입력들인 rs1, rs2, rd, rd\_din, write\_enable, 그리고 clk과 reset이 있고, 출력으로는 rs1\_dout과 rs2\_dout이 있다. Clock synchronous 하게 negedge에서 rd\_din에 따라 rf값을 업데이트하고(write\_enable이 참이고 rd가 0이 아닐 때), posedge에서 초기에 rf값들을 0으로 초기화해주는 작업을 진행한다. (rf[2]의 경우 stack pointer로, 초기 0x2ffc로 시작)

1. Control Unit

Opcode 값으로부터 control signal을 생성하여 각 stage나 모듈에 보낼 수 있도록 한다. mem\_read, mem\_to\_reg, mem\_write, alu\_src, reg\_write, alu\_op 등의 값을 각 경우에 따라 0또는 1 값을 입력하였으며, synchronous하게 동작한다. 이떄 is\_ecall 값을 block 안에서 할당하면, 잘못된 값으로 진행되는 상황이 있었기 떄문에, always 밖에서 assign으로 값을 할당하였다.



1. Immediate Generator

본 모듈은 Single-cycle CPU 및 multi-cycle CPU와 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

본 모듈은 single, multi-cycle과 동일한 방식으로 작동하는, ALU를 컨트롤하는 모듈이다. 인스트럭션 종류에 따라 ALU가 수행하게 될 연산의 종류를 결정하여 alu\_op로 내보낸다. Single-cycle CPU와는 다소 다르게, multi cycle과 pipelined CPU은 Control Unit 으로부터 받은 ALUop 값이 존재하며 해당 값까지 모두 고려하여 Asynchronous하게 출력 값을 결정한다. 또한, 이번 Lab 과제에서는 controlflow를 다루지 않기 때문에, 이전 Lab3의 모듈에서 관련된 부분을 모두 빼 주었다. 따라서 output의 bcond가 제거된 형태로 구현되었다.

1. ALU

본 모듈은 single cycle, multi cycle CPU와 거의 동일하다. Asynchronous하게 산술 및 논리 연산을 수행하며, ALU Control Unit의 지시에 따라 연산을 수행한 후 결과를 레지스터 파일에 저장한다. 한 가지 다른 점은 이번 Lab 과제에서는 controlflow를 다루지 않으므로, JAL, JALR, BRANCH 등 관련 부분을 모두 빼고 구현하였따.

1. mux\_for\_alu

Data Forwarding이 발생하게 되면, 기존 값 대신 ForwardingB에서 받아온 값을 사용하여 ALU로 전달하는 Mux 이다.

1. Data Memory

주어진 모듈이다. 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.

1. ForwardingUnit

의존성이 있는 두 인스트럭션 간 거리가 1, 2일 경우( dist(i1, i2) <= 2 ) EX/MEM 또는 MEM/EX 레지스터에 저장된 값을 불러와서 ForwardingA와 ForwardingB에 저장한다.

1. muxForwardA, B

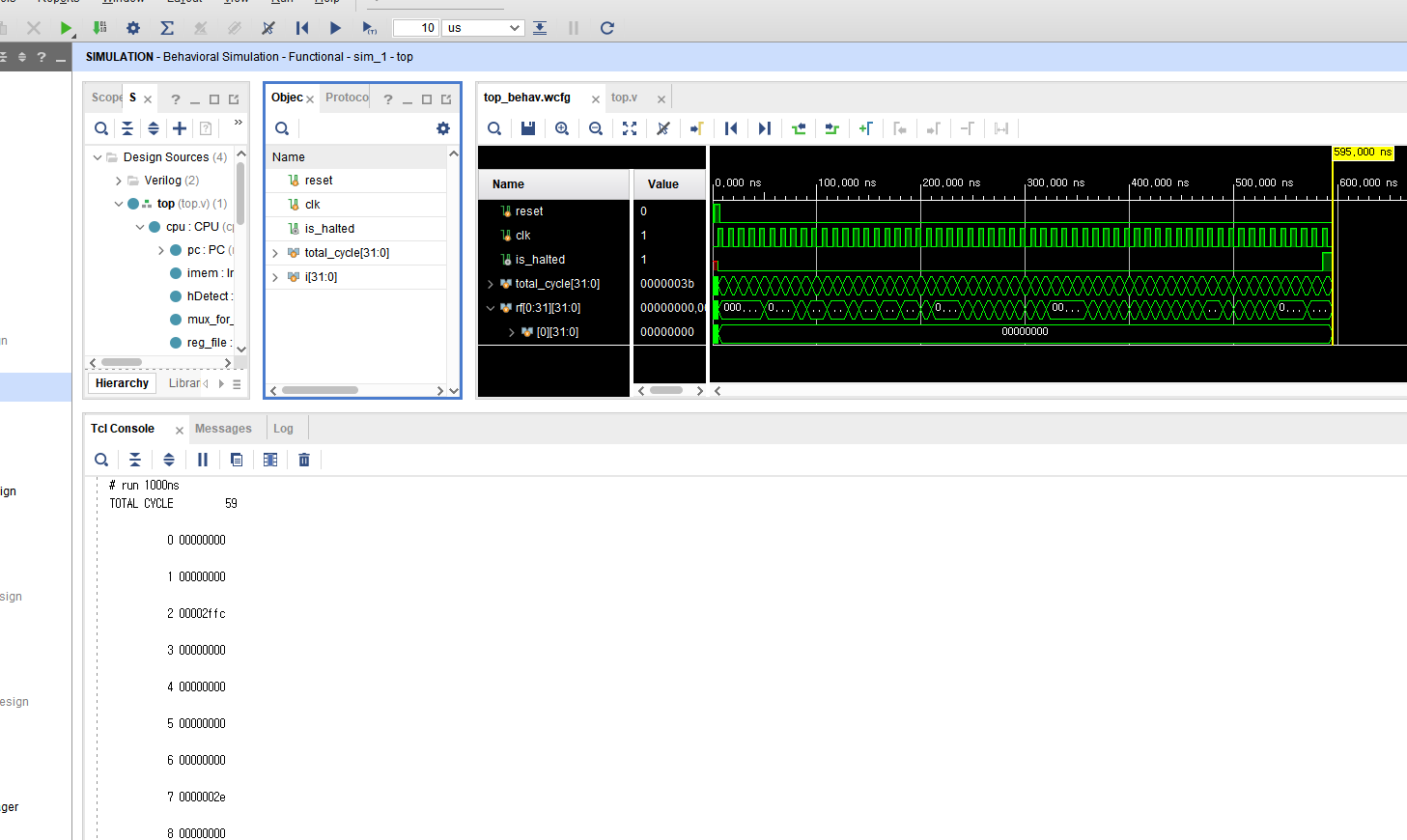
ALU의 input 레지스터로 들어갈 수 있는 값들 중 적절한 값을 선택하는 Mux로 Forwarding Unit에서 계산된 신호를 Select Signal로 사용한다.

1. ForwardEcall

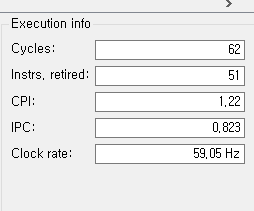
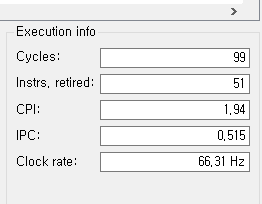
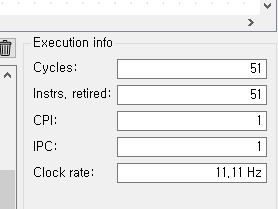
의존성이 있는 두 인스트럭션 간 거리가 3인 경우( dist(i1, i2) = 3) EX/MEM의 결과를 ID stage로 forwarding하여 반영할 수 있게 하는 모듈이다. 레지스터 파일의 input으로 들어갈 rs1, rs2 레지스터의 값을 최종적으로 결정하여 f\_rs\_1\_dout, f\_rs\_2\_dout으로 내보내며, Ecall 호출이 발생한 경우에는 EX/MEM의 alu 결과값이 반영될 수 있도록 하였다.

1. **Discussion**

가장 먼저, Lab 설명 pdf에 나와 있는 single cycle cpu와의 cycle 수 비교에 대해 설명하겠다. 이번 Lab 과제를 통해서 pinelined cpu를 구현하였는데, data forwarding을 적절히 구현하여 stall을 줄이도록 구현하였다. 그 부분을 담당하는 것이 forwardingUnit과 hazardDetectionUnit으로, 각각 data forwarding, hazard 발생 여부 감지를 담당하도록 하였다. 자세한 구현은 앞의 각 모듈 설명에 작성하였다. 이렇게, forwarding을 구현한 결과 아래와 같이 non\_control\_flow.txt testbench에 대해 아래와 같은 시뮬레이션 결과를 얻을 수 있었다.



59cycle이라는 결과가 나왔다. 반면, Ripes를 이용하여 작동을 시켜보니, 아래와 같이 99cycle이라는 결과가 나왔다. (각각 single cycle, 5-stage RISC-V processor w/o Forwarding Unit와 5-stage RISC-V processor)

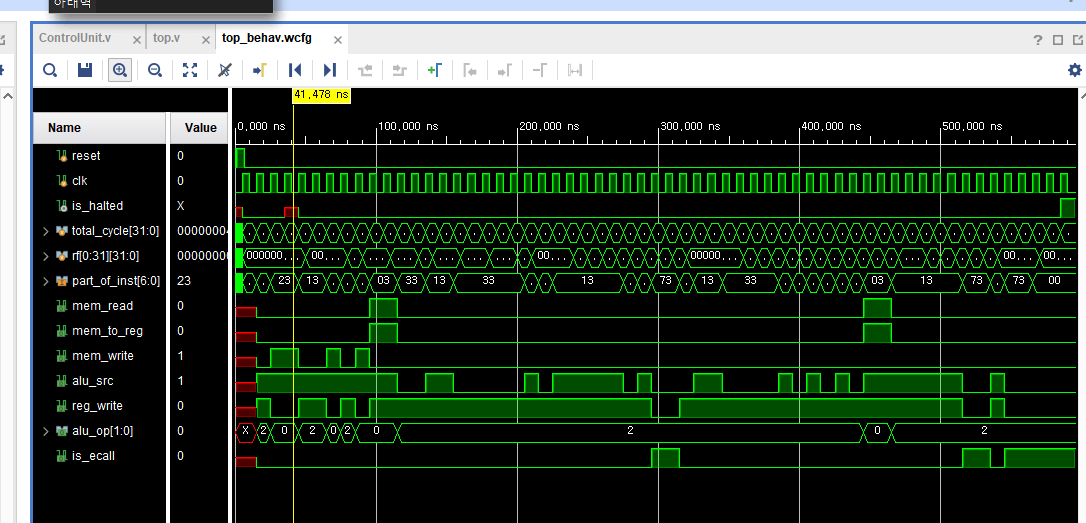


위와 같이 각각 51cycles, 99cycles, 62cycles가 나온 것을 확인할 수 있었다. 먼저, single cycle과 비교해보면 cycle 수 자체는 single cycle CPU가 더 적은 것을 확인할 수 있다. 다만, 5 stage pipelined CPU의 경우 중간에 추가되는 logic과 register들로 인한 시간 지연을 고려하지 않는다면 single의 5cycle이 pipelined의 1cycle에 해당하고, 지연을 고려하더라도 몇 배의 차이가 난다. 또한 본 과제에서 forwarding을 하였기 때문에 stall이 적어 pipeline의 이점을 충분히 사용할 수 있었다. 즉, 51cycles과 59cycles의 결과이지만 실질적으로 pipelined cpu가 훨씬 빠를 것이다. 실제로 위 첫번째 사진과 두번째 사진에서의 clock rate를 비교해본다면 single cycle cpu보다 pipelined 버전이 5배가량 빠른 것도 확인할 수 있다.

또한 이제 두 번째 사진과 비교해보자면, 이번 구현에서 forwarding unit을 적절히 구현하였기 때문에, branch가 없는 경우 stall을 줄여서 cycle 수가 상당히 감소하였다는 것을 확인할 수 있었다. Without forwarding unit의 경우 거의 59cycle보다 상당히 큰 값의 cycle 수를 보였고, 5-stage processor의 경우 59cycle과 거의 같은 62cycles를 보였다. Ripes환경과 vivado simulation 상 약간의 환경 차이를 고려한다면 거의 똑 같은 cycles 수는 비슷한 방식으로 적절히 구현되었다는 것을 나타낼 수 있을 것이다.

추가로, 이번 Lab에서 internal forwarding을 구현하지 않고 Register module에서 negedge로 수정이 되었던 상황이 있었는데, 처음 수정된 사항을 늦게 확인하여 forwarding unit을 거의 완성해가던 시점에 이를 발견하였다. 따라서, 만들던 대로 일단 구현을 완료하였고, negedge로 변경된 register unit 파일로 기존 파일을 대체를 하여도 구현한 cpu 작동에는 문제가 없을 것으로 추측하였는데, 실제로 파일을 대체하기 전과 같은 시뮬레이션 결과가 나오는 것을 확인할 수 있었다. 따라서 따로 forwarding 부분을 제거하지 않고 그대로 놔둔 상태로 제출을 완료하였다.

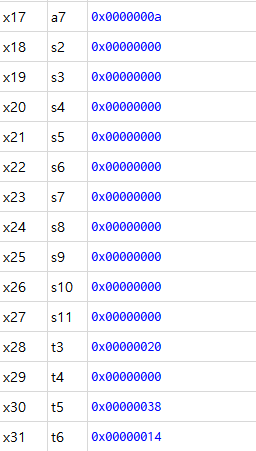
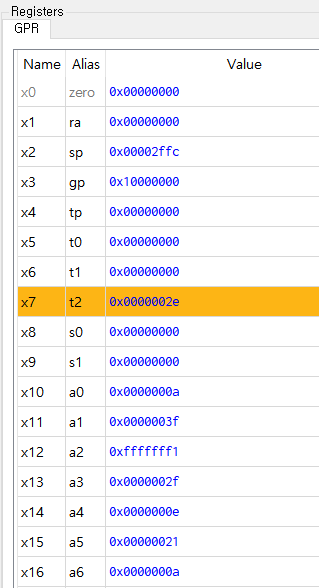
마지막으로, 본 Lab을 구현하면서 Forwarding과 Hazard detection 관련해서 상당히 고전을 하였다. 다양한 문제들이 있었지만, hazard detection 관련된 문제가 가장 컸다. 아래는 문제의 시뮬레이션 결과이다.

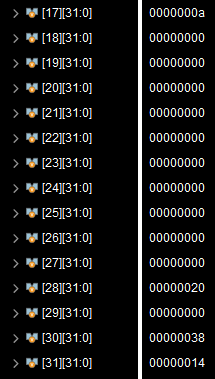
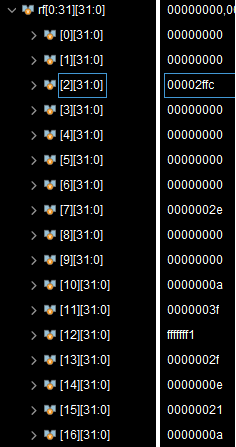


중간에 is\_halted가 갑자기 xxxx값이 되는 것을 확인할 수 있었다. 또한 이전에도 중간에 캡쳐해놓지는 않았지만 is\_halted 값이 마구 올라가고 register 값들이 제대로 반영이 안 되는 것을 확인할 수 있었다. 그러한 경우에서, is\_halted에 영향을 미치는 신호를 찾고, 그 신호에 영향을 미치는 신호와 모듈을 찾아서 원인을 찾아 올라가는 방식으로 디버깅을 진행하였고, 그 결과 첫 번째 register 값들이 제대로 반영이 되지 않는 문제는 구현한 MUX 모듈의 문제임을 확인하였다. 따라서 다른 방식으로 MUX를 구현하여 이를 해결하였다. 다음으로 발생했던 위의 오류의 경우 hazardDetection과 forwarding 모듈의 문제인 것으로 파악하였다. HazardDetection의 경우 강의 노트를 점검하면서 검토한 결과 다른 register를 모듈로 보내준 것으로 파악되어 이를 수정하였고, forwarding 모듈의 경우 문제를 겪으면서 forwardingEcall 모듈 부분을 따로 분리하여 구현을 하게 되었다. 이렇게 디버깅과 구현 방법 수정을 거치면서 제대로 작동하는 pipelined CPU를 완성할 수 있었다.

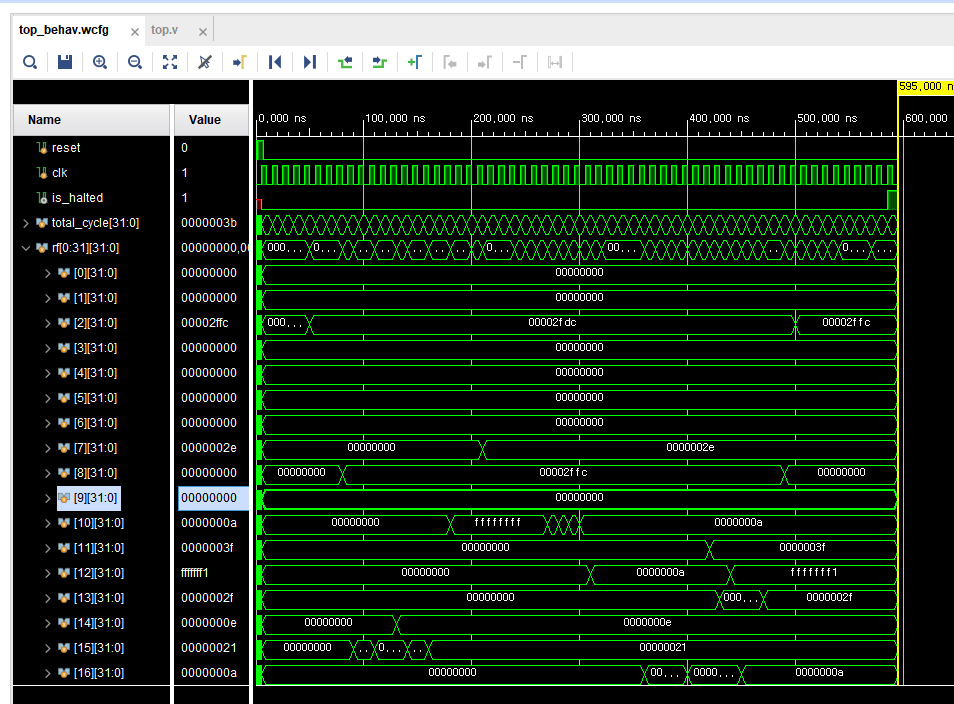
1. **Conclusion**

본 Lab 과제를 통하여 주어진 skeleton code를 바탕으로 pipelined CPU를 디자인해볼 수 있었다. 또한, Lab3에서 사용하였던 모듈을 일부 수정하여 pipelined CPU에 맞도록 구현하였다. 그러한 과정에서 5-stage pipelined CPU의 작동 방식에 대해 다시금 공부할 수 있었고, data가 어떠한 방식으로 Forwarding되는 지 모듈을 구현하여 직접 알아볼 수 있었다. 구현한 내용을 바탕으로 non\_control\_flow\_mem.txt testbench 파일로 시뮬레이션에 성공하였고, reference인 Ripes에서 구현한 값과 일치하는 register 값이 나옴을 확인할 수 있었다. 아래는 Ripes에서의 register 값과 직접 simulation한 register값이다.





이렇게 같은 결과를 확인하였고, 구현한 pipelined cpu가 잘 작동하는 것을 확인할 수 있었다. 또한, 전체 cyle 수는 59라는 결과가 나왔으며, 아래는 전체 시뮬레이션 사진이다.



이렇게, 성공적으로 구현을 완료하며 pipelined cpu에 대해 공부할 수 있었다. 본 Lab 과제를 기반으로 non-controlflow를 넘어서 branch detection 모듈을 구현하여 branch도 처리할 수 있는 cpu를 Lab4-2에서 구현하고자 한다.