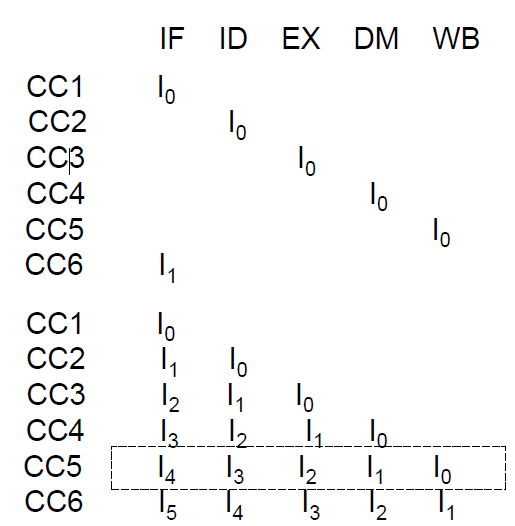
#HW12 2017069598 박상지

multicycle vs pipelining



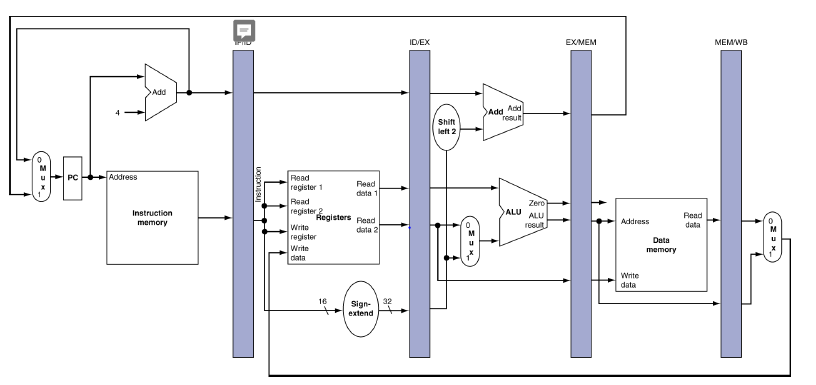
multicycle을 사용할 경우 clock cycle time이 200ps일 때, CPI는 5가 된다.

하지만 Pipeline 5-stage로 돌릴 경우 cct는 똑같지만, CPI는 1이 된다. 하나의 하드웨어로 서로 다른 instruction을 돌리기 때문에 성능이 높아진다. 모든 instruction이 이상적으로 똑 같은 시간이 걸린다면 이상적인 속도는 각 stage의 수와 같게 된다. 하지만 각 stage가 균일한 속도를 갖고 있지 않기 때문에 가장 느린 stage를 speedup 기준을 잡는다. pipeline은 매 cycle 마다 instruction을 fetch를 하는 것이 이상적이지만 hazard 때문에 매 cycle마다 instruction을 fetch할 수 없다.

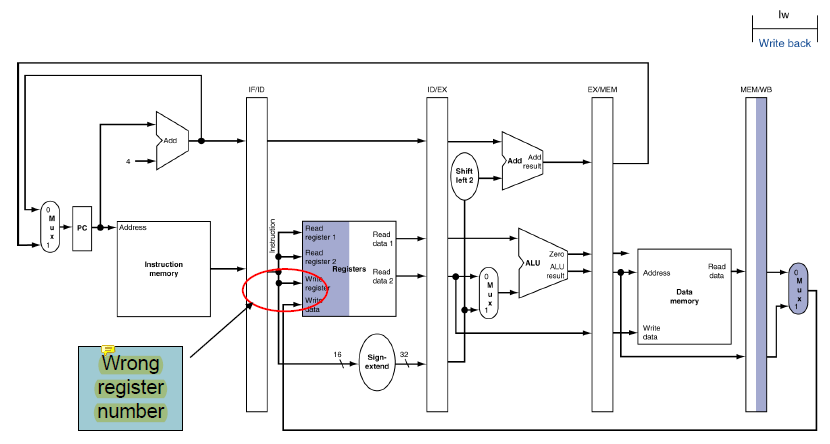
pipelined Datapath and control

파이프라인을 사용할 경우 각 stage마다 서로 다른 instruction간에 데이터와 control signal들이 간섭되면 안된다. 각 instruction은 clock이 올라 갈 때마다 fetch가 되기 때문에 각 stage마다 신호를 insulation을 해주는 역할이 필요하다.

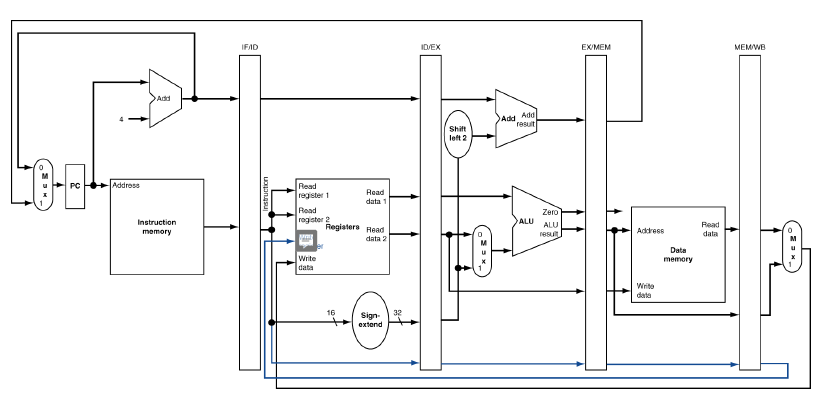
pipeline registers



stage간에 필요한 레지스터를 만들어 준다. instruction이 fetch가 일어나면 다음 clock이 올라오기 전에 해당 register에 결과 값을 저장한다.

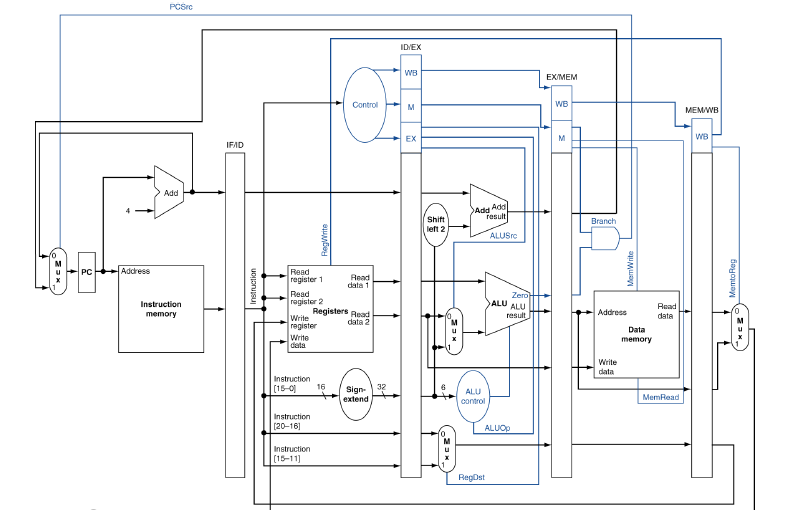


WB에서 쓸 데이터를 주는 것은 5번째 stage에서 주는 것인데 몇 번째 레지스터를 사용할 것인 결정하는 것은 2번째 stage에서 결정한다. 이 부분에서 버그(bug)가 일어난다. 몇 번 레지스터에 어느 데이터를 저장하는 것은 WB에서 결정해야 하는 일이다.



결과 값을 어느 레지스터에 저장할 것인지 write register을 결정하여 각 파이프라인에 계속 저장하면서 마지막 WB에서 사용된다.

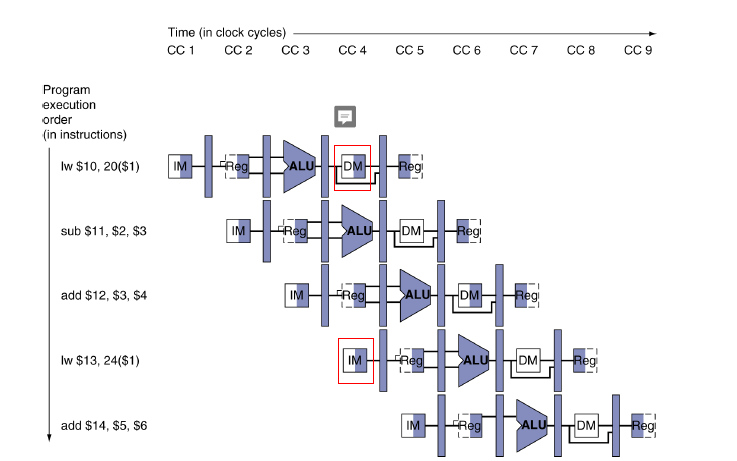
Pipelined Control



IF와 ID는 모든 instruction의 공통 부분이다. decode가 끝나지 않았기 때문이다. 해당 stage에 PCSrc 와 RegWrite 신호들이 있다. 위치는 해당 stage에 있지만 신호를 주는 것은 WB(5stage), Branch(4stage)에서 준다. 즉 instruction이 무엇인지 모르기 때문에 control signal이 있을 수 없다.

Pipeline Hazards

1. Structure Hazards



Structure Hazards는 다른 stage에서 같은 functional unit을 사용할 때 생기는 오류이다. 그림과 같이 Data Memory에서 data access를 하는데 4번째 instruction에서 instruction memory에서도 data fetch를 한다. 둘 중에 하나를 먼저 데이터를 제공을 해줘야 한다. 순서가 빠른 instruction의 stage부터 해결해야 하는데, 그러면 후의 instruction을 한 cycle를 대기를 해야 한다.

질문) 왜 “structural” hazard 인가?

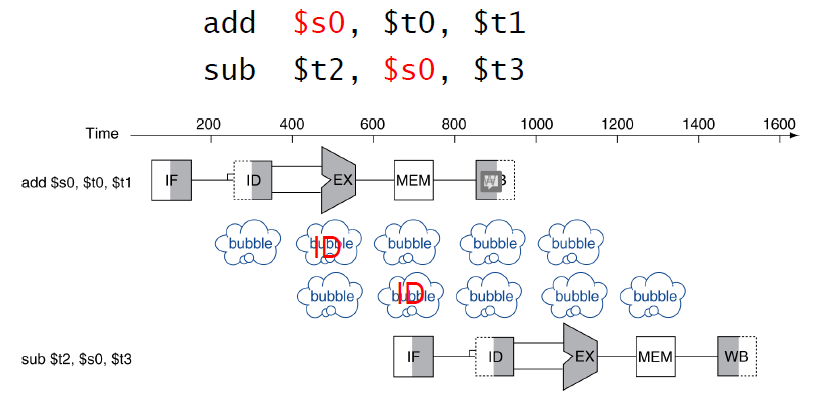
간단히 data path에서 구조만으로 같은 functional unit을 사용한 stage를 쓰는 곳을 한 눈에 볼 수 있기 때문이다.

질문) register access에 대해 어떻게 대처할 것인가?

timing으로 해결을 한다. 다른 instruction은 200ps의 시간이 걸린다. register access는 100ps의 시간이 걸린다. write back stage에서 동작이 끝내는 부분을 instruction memory에서 부분적으로 사용하면 된다.

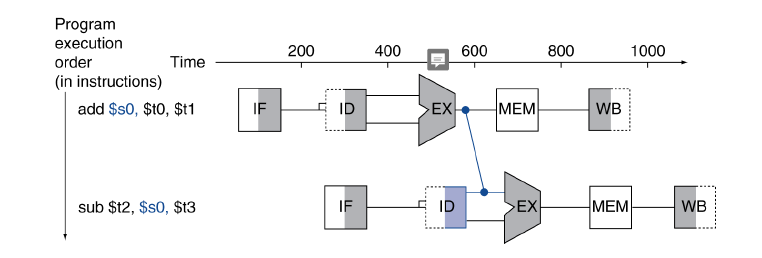
1. Data Hazards

Data Hazards에서는 한 instruction에서 사용했던 레지스터를 다음 instruction에서도 사용할 경우 생기는 hazard이다.



add instruction에서 결과 값을 s0 레지스터에 저장하였다. 하지만 다음 instruction에서 s0의 레지스터의 값을 연산 레지스터로 사용한다. add의 MIPS를 살펴보면 결과 값은 write back stage에서 생긴다. 그렇기 때문에 sub instruction에서는 s0의 레지스터의 값이 저장될 때까지 fetch를 할 수 없는 상태이다. 그렇기 때문에 Pipeline stall, bubble을 사용하여 연산에 무관한 instruction을 추가하여 sub instruction을 늦춰서 실행시킨다. 하지만 CPI는 0.2 늦춰진다.

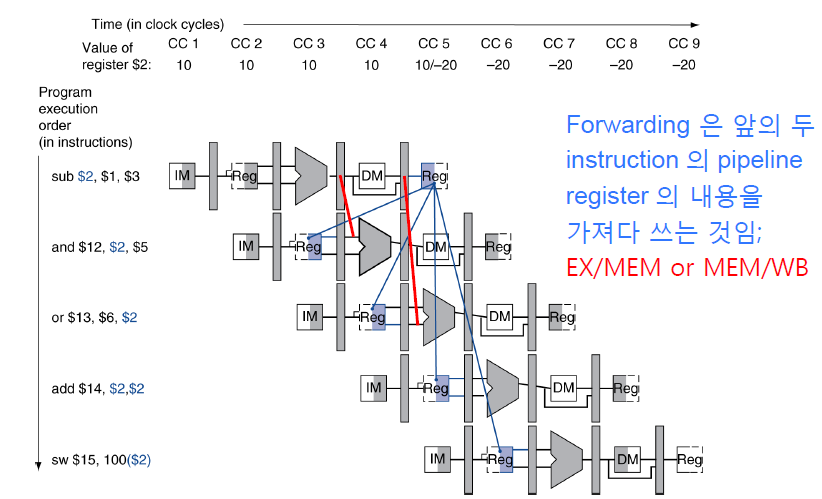
Forwarding



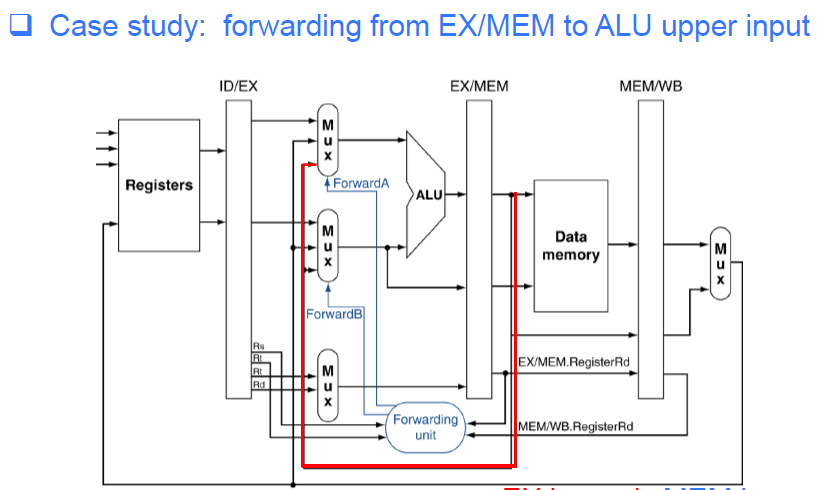
add 연산에서는 두 레지스터의 값을 더한 결과 값이 execute stage가 끝나면 pipeline 레지스터 저장된다. 해당 레지스터의 값을 가져와 sub 연산의 input data에 전달해 준다. forwarding을 사용하면 bubble을 사용할 필요가 없기 때문에 CPI 속도에 지장이 가지 않는다.

Data Hazards in ALU instructions

프로그램 특성상 데이터는 write 후에 read를 해야 한다. (data dependence) 하지만 연달아 실행되는 instruction에서 연관된 레지스터가 없을 경우 병렬로 실행하여도 결과 값에 지장이 없다. (data parallelism)



5개의 instruction이 2 레지스터를 사용하는데, sub instruction 이후에 3번째 instruction에서는 hazard가 일어나지 않는다. 왜냐하면 이미 write back을 한 상태에서 structural hazard는 timing으로 해결하였기 때문이다. 문제는 1, 2번째 instruction이다.



-condition 1 detecting EX/MEM hazard



바로 앞선 instruction의 결과 레지스터와 ALU upper operand 레지스터 number와 같을 경우 hazard가 일어난다. 그렇기 때문에 해당 레지스터들이 같은 지 확인해야 한다.

-condition 2 detecting EX/MEM hazard

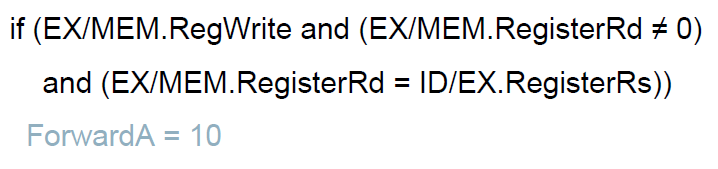


forwarding은 결과 값을 새로운 레지스터에 저장할 때 생기는 hazard를 보완할 때 사용된다. 그렇기 때문에 instruction이 ALU 연산 또는 Load instruction에서 사용된다. 그렇다면 beq 연산 또는 store 연산에서는 결과 값이 나오지가 않는다. beq는 두 개의 레지스터가 같은 지 다른 지만 확인하기 때문에 결과 값이 나오지 않고 store에서는 레지스터를 메모리로 옮기기 때문에 새로운 레지스터 값을 만들지 않는다. 그렇기 때문에 RegWrite control signal이 enable일 경우에만 forwarding을 할 수 있게 설정한다.

-condition 3 detecting EX/MEM hazard

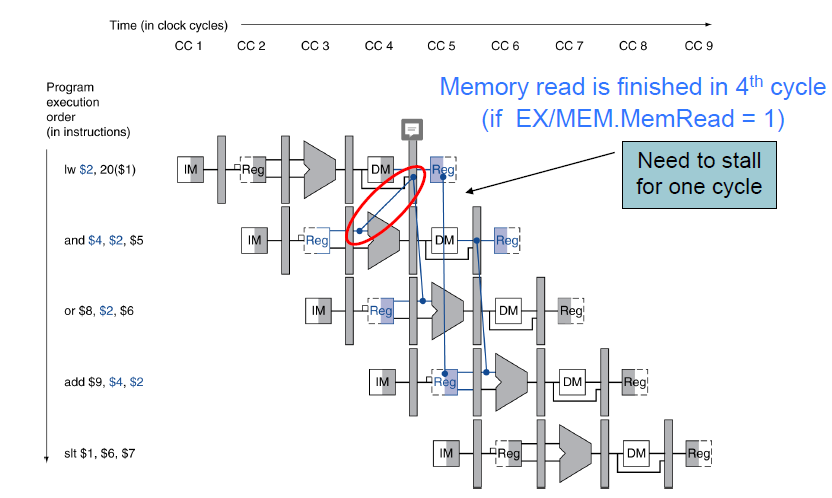


결과 값이 만약에 제로 레지스터를 사용할 경우 forwarding을 할 필요 없다. 제로 레지스터는 사용하지 않는 가짜 레지스터이기 때문이다.



-condition 4 load-use data Hazard

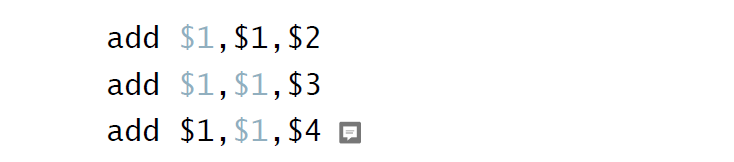




앞선 instruction이 load 연산일 경우 data memory에 결과 값이 레지스터에 저장이 되어야 다음 instruction add 사용할 수 있다. 그러기 때문에 bubble이 1 cycle 들어가야 한다.

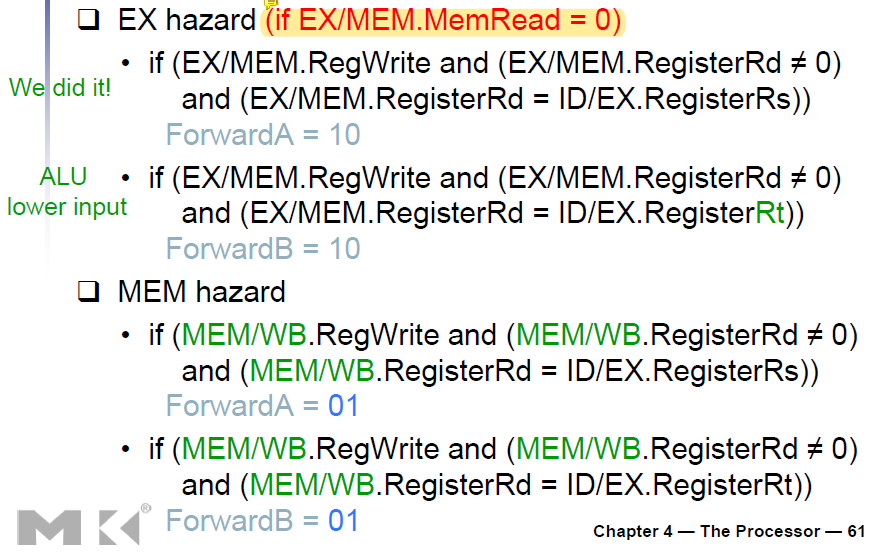
그렇게 때문에 forwarding을 할 수 없기 때문에 앞선 instruction이 load가 아닌 조건이 필요하다. load-use data Hazard에서는 1 cycle의 bubble이 필요하기 때문에 CPI는 1 -> 1.05가 된다.

-condition 5 double data Hazard

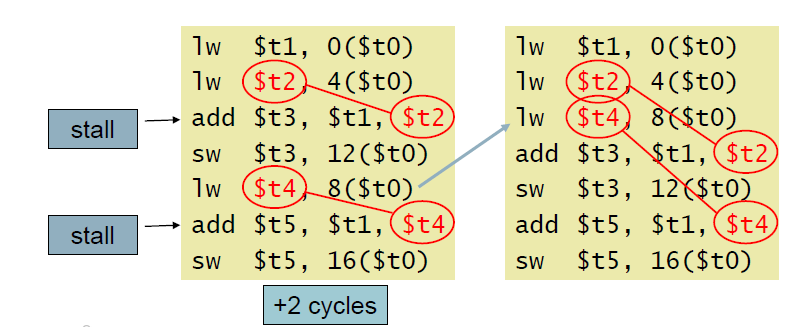


맨 처음 instruction의 결과 값을 1번 레지스터에 저장하면 모든 연산은 forwarding을 맨 처음 instruction의 데이터 값으로 지정했다. 하지만 두 번째 instruction에서 결과 값 레지스터로 1번 레지스터를 한번 더 사용하였다. 이럴 경우 첫 번째 결과 값을 사용하면 오류가 난다. 그렇기 때문에 가장 최근에 사용되었던 결과 값을 사용해야 한다.

hazards가 일어나면 clear control을 해줘야 한다. 모든 control signal을 제로로 만들어 줘서 결과를 레지스터와 메모리에 저장하지 않게 설정하여, 프로세서의 상태를 바꾸지 않았기 때문에 아무 일도 일어나지 않는다.



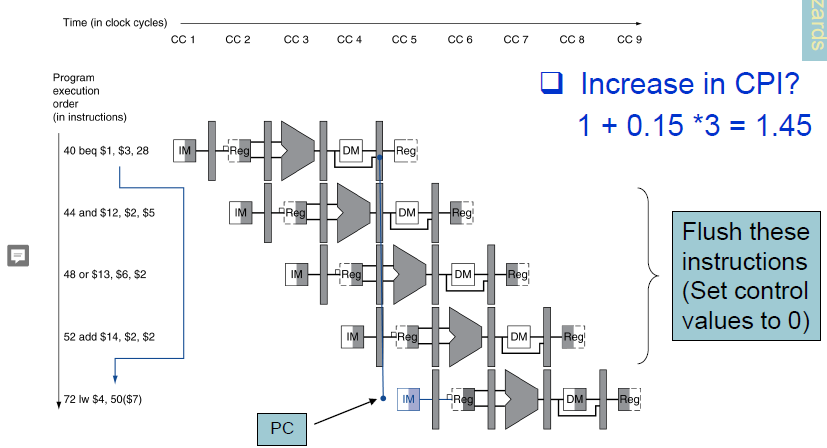
Data Hazards and Compilers



왼쪽 코드를 오른쪽 코드로 고치면 load에 대한 hazard가 사라지게 된다. 하지만 CPU의 implementation을 알고 컴파일러로 load-use hazard를 피했기 때문에 올바른 선택은 아니다. 왜냐하면 사용자는 implementation을 알 필요가 없기 때문이다.

1. Branch hazards

주어진 cycle에 실행될 instruction이 그 cycle에 fetch되지 못해서, 실행되지 못하는 경우에 생기는 hazard이다. fetch된 instruction이 실행해야 할 instruction이 아니고 instruction address flow가 예상과 다르게 되어 기존의 fetch된 instruction 폐기하고 다시 fetch해야하는 상황, 또는 instruction address flow를 알 수 없어서 적당한 instruction을 fetch를 할 수 없는 상황에 생긴다.



-reducing the delay of branches

1. branch execution이 이루어지는 pipeline stage

MEM stage에서 branch에 대한 next PC값이 결정된다고 가정했을 때, next PC 값의 결정을 더 빨리 함으로 써 flush instruction 수를 감소시킨다. branch execution을 ID stage로 이동시키면 flush instruction이 3개에서 1개로 감소되는데, 즉 1 clock cycle로 감소된다.

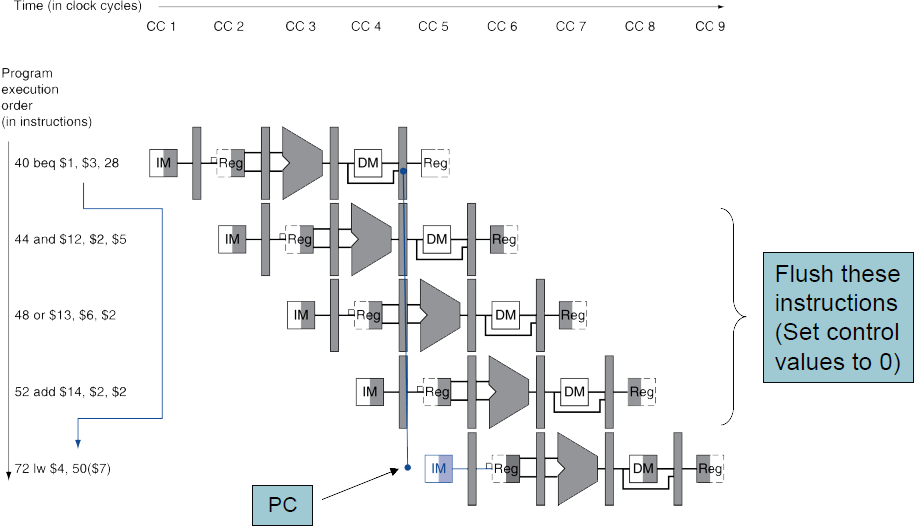
2.branch execution을 ID stage에서 수행하기 위한 요구사항

ID stage에서 branch target address를 계산을 한다. branch target address를 계산하기 위한 adder를 EX stage에서 ID stage로 이동하고 증가된 PC는 IF/ID. PC에 이미 존재한다. 이전처럼 branch target address는 모든 instruction에 대해 계산되고 필요 없게 되면 버린다.

ID stage에서 branch decision을 한다. register file에서 Rs, Rt를 읽어서 바로 comparison을 하고 ID stage에 comparator을 추가하여 각 bit끼리 XOR 한 32bit를 NOR로 바꾼다. ALU 또는 adder와 비교해서 하드웨어 비용이 작으면, comparison에 소요되는 time도 작아진다.

3. ID stage operation의 변화

＊branch의 결과가 MEM 단계에서 결정된다면??(원래 ALU에서 zero인지 아니까 EX아닌가?? 말 그대로 가정인가??)



＊Control hazard 파트에서,

　 branch 결과를 알기 위해 기다리는 것 대신 "아닌 것으로 가정"하여 연속된 명령어를 수행하다가

　 아니면 그대로, 맞으면 제대로 명령어 가져와서 수행한다.

　 만약 결과가 맞으면 그동안 미리 예측하여 수행해둔 명령어는 버려야 한다.

　 제어 신호를 0으로 하여 비워야(flush) 한다.

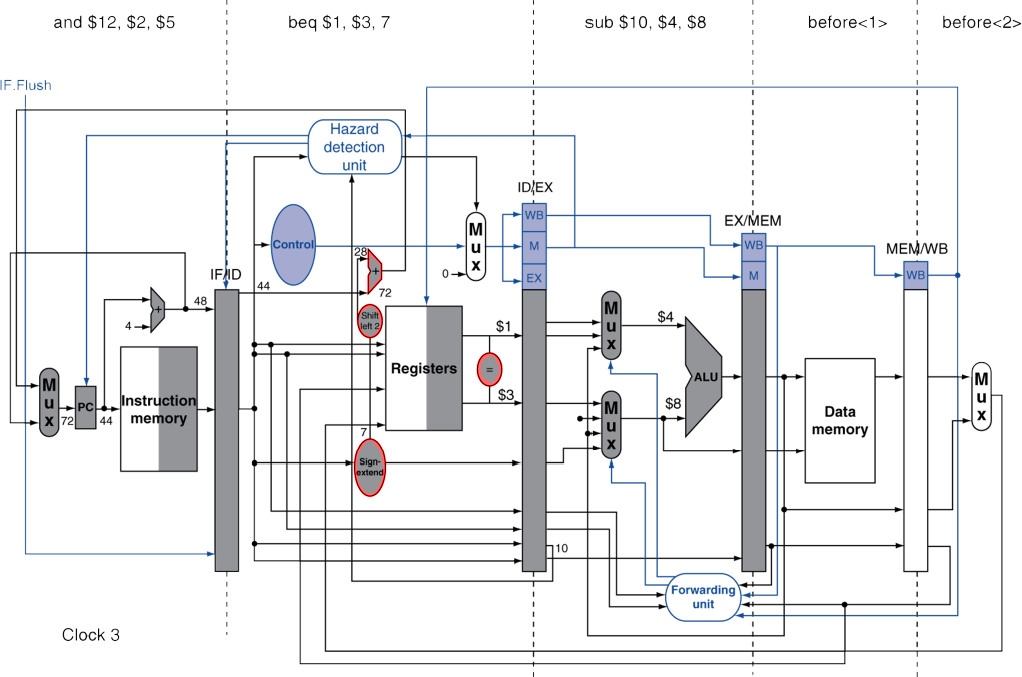
Reducing Branch Delay

예측에 실패하더라도 버려지는 단계를 줄여보자

branch 결과를 구하는 걸 ID stage에서 할 수 있도록 하드웨어를 옮긴다.

　 target address adder (대상 주소를 PC+4+offset(address\*4)해주던 ALU 및 left shift, sign extend)

　 register comparator (ALU로 비교해서(빼서) zero인지 봤던)

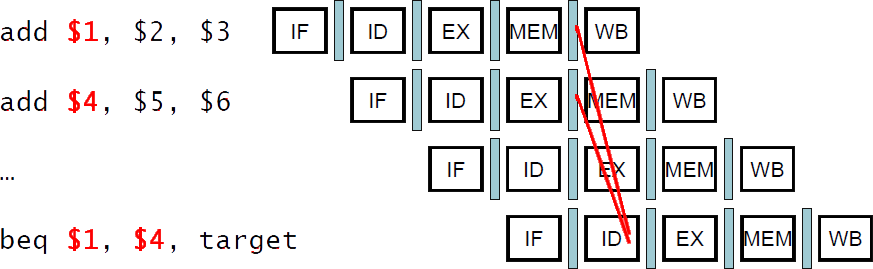
ID stage에 값비교와 주소계산이 모두 가능하도록 장치를 옮기거나 추가함.

　 be는 다음 stage들도 넘어가도 딱히 할 일은 없음.

　 예측이 맞다 면 상관없이 다음 명령어 계속 수행한다.

　 예측이 틀리고 정말 분기로 이동해야 한다면, 분기 대상 주소를 PC에 넣어 해당하는 명령어를 IF할 수 있도록 하고, 직후 명령어(IF였던)는 파기(이제서야 ID로 들어오지만)한다.

　 기존 모델에서는 예측으로 실행해둔 여러 instruction을 버리면서 여러 cycle의 bubble이 발생하지만, ID stage에서 분기 결과값을 구함으로써, 단 1번(IF 단계였던 직후 명령어)의 bubble만으로 줄일 수 있다.



forwarding을 사용하여 해결할 수 있다.

　 이전에 본 forwarding은 현재 cycle이 EX stage였다(ALU에 필요한 피연산자).

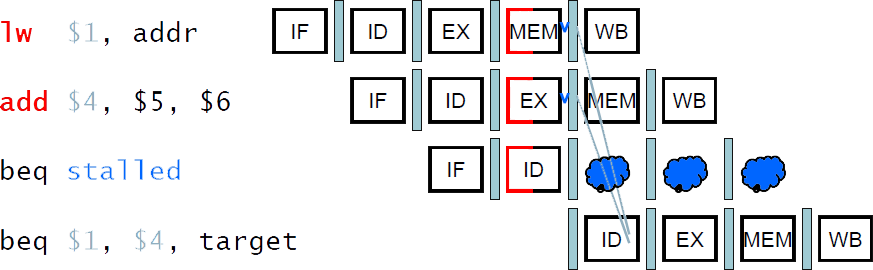
　 이제 branch의 비교 ALU는 ID stage에 있기 때문에, 1, 2 전 단계가 아닌, 2, 3 전 단계에 대해 forwarding할 수 있다.

　　(1 전 명령어는 이제 막 EX를 수행하는 중이고, 2 전 명령어는 EX/MEM, 3 전 명령어가 MEM/WB)

만약 비교하려는 레지스터가

　 1 cycle 전(바로 직전)의 ALU instruction의 결과(ID/EX. RegisterRd)이거나,

　 2 cycle 전의 load로 불러온 값을 쓸 레지스터(EX/MEM.RegisterRt)라면, forwarding할 수 없고, stall이 발생할 수밖에 없다.

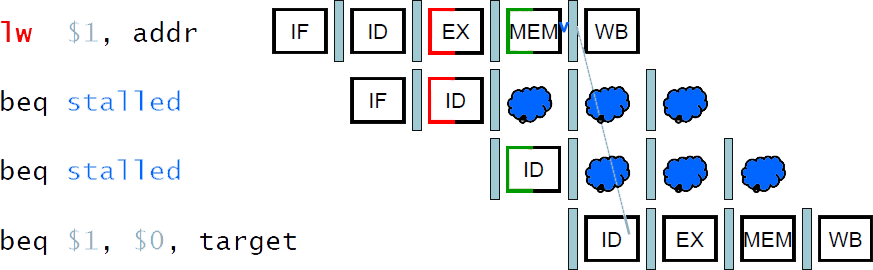


-직전 명령어가 산술일 때, 이제 EX 들어가려는 데 값이 있을 리가 없다.

　　(branch 피연산자로서 ALU 결과는 한 칸의 여유 필요)

-2-cycle 전 명령어가 load일 때, 이제 MEM 들어가려는 데 값이 있을 리가 없다.

　　(branch 피연산자로서 MEM read는 두 칸의 여유 필요)



Dynamic Branch Prediction

초대형(deeper and superscalar(stage 아주 많음)) pipeline일수록, branch 페널티(stall 발생)는 더 중요하게 여겨진다(크다). stage 수가 많으면 나눠서 일을 하니 속도가 올라가지만, 그만큼 stall도 한꺼번에 여러 cycle이 발생할 수 있다(여러 stage가 쉬어야 하니까). 또한 복잡하기 때문에, branch의 결과도 ID stage가 아닌, 좀 더 진행이 된 상태에서 알 수 있게 된다. 그러면 예상이 틀렸을 때, 미리 진행한 명령어들을 flush하면서 stall도 더 많이 발생함.

dynamic prediction(동적 예측)을 사용하자

　 branch prediction buffer(aka. branch history table)를 둔다.

　 거쳐온 branch instruction의 주소들을 색인화(index)한다.

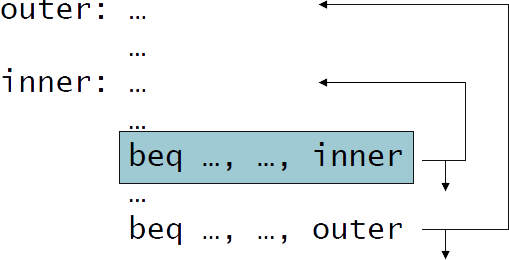
　 분기 결과(outcome)를 저장해 둔다(store). taken/not taken(조건 맞음/조건 틀림)

　 branch를 수행하려면, 테이블을 살펴보고 추론한 예상을 바탕으로, 이러하길 바라면서 진행하기로 한다. 분기 대상이든, 아니라서 연속으로 진행하든, 예상한 대로의 명령어를 가져와서 진행한다. 만약 틀렸다면, 예상대로 했던 명령어를 비우고(flush pipeline) 예측을 뒤엎는다(반대의 명령어를 가져온다)

-1-Bit Predictor: Shortcoming(결점, 단점)

1-Bit. 바로 전의 결과만 사용하는 경우.

다중 loop에서, 안쪽 loop branch는 예측실패(mis predict)를 두 번 한다(처음, 마지막)



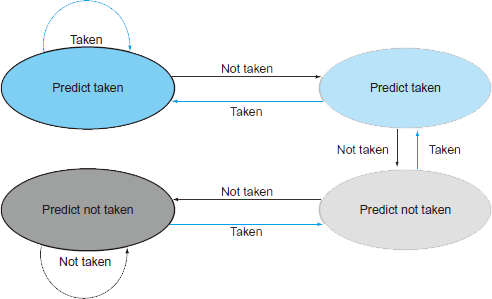
　 안쪽 loop의 반복 중 마지막에서, 예측실패(mis predict)는 일어날 것(taken)으로 나타난다.

　　 안쪽 loop는 항상 일어나서(taken) 내부가 반복되었기 때문에, 이번에도 그럴 것으로 예측해서

　 안쪽 loop의 반복 중 첫번째에서, 예측실패(mis predict)는 일어나지 않을 것(not taken)으로 나타난다. 안쪽 loop의 가장 최근은 일어나지 않고(not taken), 밖으로 나가서 바깥 loop가 다시 안쪽을 시작시킨 것이라, 이번에도 안쪽은 최근대로 일어나지 않을 것으로 예측한다.

2-Bit Predictor

두번의 연속된 예측실패(misprediction)이 일어나야만 예측기준의 결과(predictor가 판단할)를 역전한다.



Diagram

　 11↔10↔01↔00

Calculation the Branch Target(address)

branch 명령어를 위해 기존의 EX stage에서였든, ID stage로 옮겼든, 대상 주소를 계산(Rs(PC+4) + address\*4)해야 한다.

predictor를 사용하더라도, 여전히 대상 주소는 계산해야 한다. branch마다 1-cycle의 페널티

branch target buffer

　 반복될 branch라면 아예 대상 주소(바뀌지 않고 고정된 메모리주소)를 기억해두고 재활용하자.

　 대상 주소의 캐시(cache)

　 instruction fetch 때 PC에 의해 색인화(index)됨

　　 만약 예측이 적중하면, 대상 주소의 명령어를 즉시 fetch할 수 있다.

　　　 (이미 buffer에 있는 주소라면, 주소 계산의 오버헤드가 사라짐.)