



Ch.4-4 The Processor - pipeline hazard(2)

③ Control Hazard → Branch 명령어

Conditional (begin, lone)
Unconditional (J)

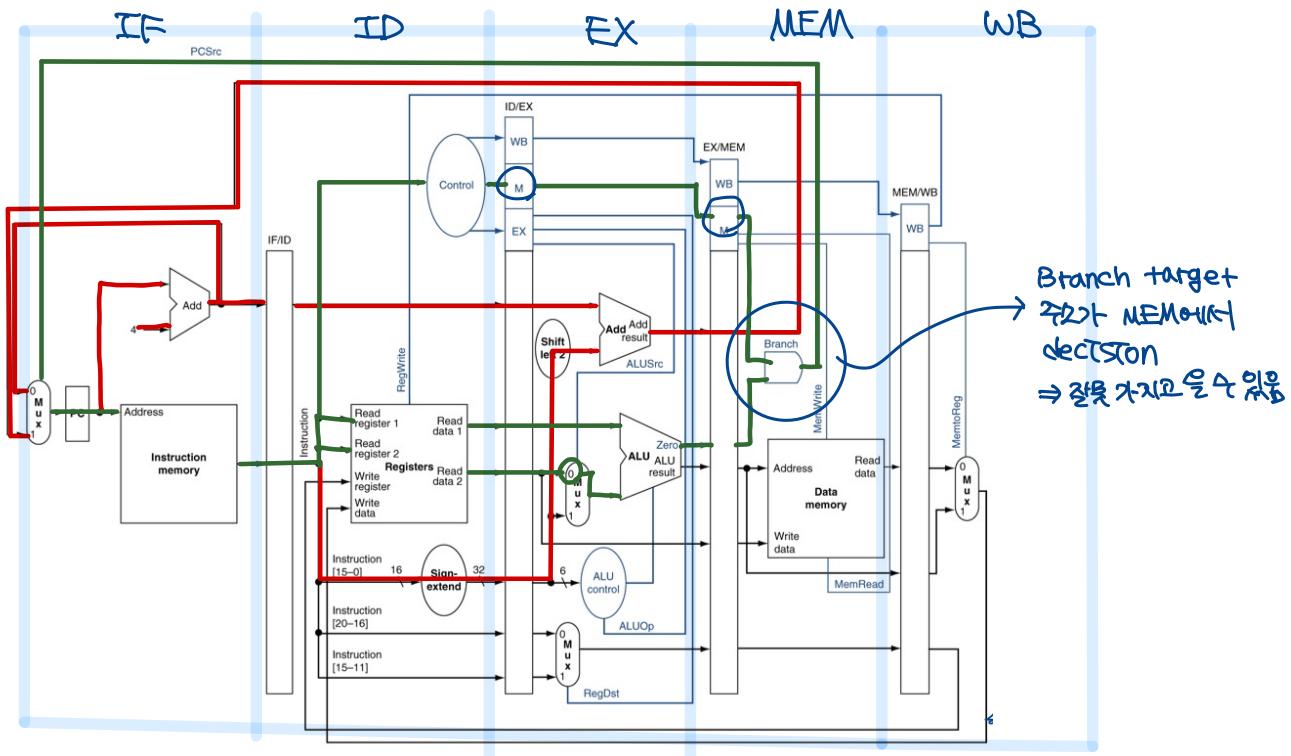
* 언제 발생?

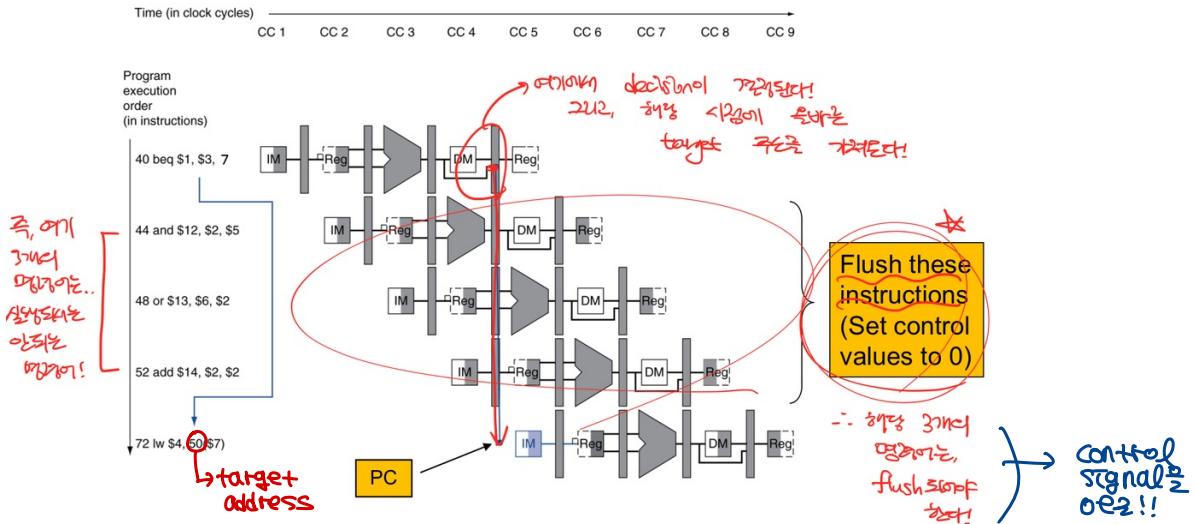
: Instruction이 순차적으로 진행 X, 이것이 change of flow 때문일때!
 (∵ Branched outcome에 따라 다른 instruction 정해짐
 → Pipeline이 해당 정해진 instruction X)

- 해결책 ① :分支가 해결될 때까지 stall
- 해결책 ② : Branch delay \downarrow → pipeline 내부에서 decision point를 더 이르게
- 해결책 ③ : predict
- 해결책 ④ : delay slot

⇒ Data Hazards에 비해서 발생 빈도 \downarrow

→ 효율적인 해결책 찾기 X





→ 다음에 실행될 instruction의 주소가 결정되어야 함

Solution ①

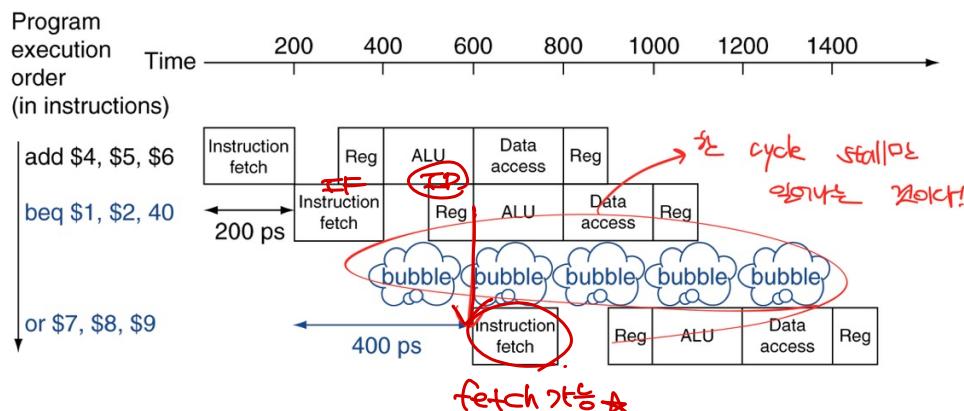
* stall : decision 완료를 때까지 기다림

→ MIPS는 Register를 비교하는 것 + Target을 Pipeline 내부에서 빠르게 계산하는 것 필요

Solution ②

* Branch delay : decision point 끝에 이르게!

→ ID stage : branch decision 가능하도록 H(w)가 가능 ($\frac{\text{Branch}}{\text{ID}}$ 예를 ID로 떼려두)



→ branch发生在 2 clock cycles이 가능

→ 그나마 여전히 느림

~~Solution ③~~

* Branch Prediction : 한 direction을 예측 + 만약 틀리면 back up 수행

i) Background

① pipeline이 걸어지면, branch outcome 일찍 떼려두어서 결정하기 쉽지 X
(stall 문제인 것)

② Branch outcome 예측? 맞으면 본전 찾기...

③ MIPS → branch들이 not taken되는 것 예측 가능

→ delay 없이, Branch 이후에 instruction fetch 가능

ii) Simple prediction

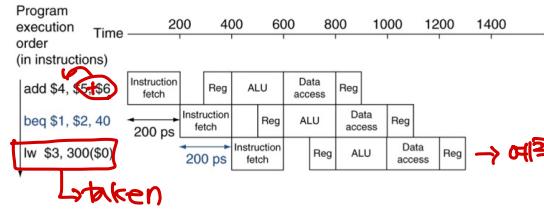
predict not taken

Prediction correct

↳ 1cycle

↳ taken

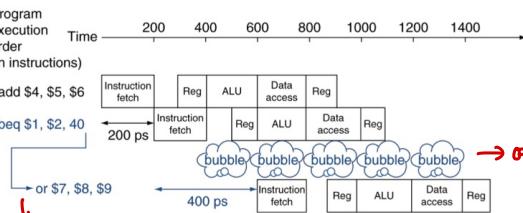
→ dynamic scheme 사용 시 branch 한 번당 90%의 성능 저하



Prediction incorrect

↳ 2cycle

∴ average CPI = 1.5



iii) Static branch prediction (즈즈, compiler support) ⇒ 뒤에 가서 더 자세히 내용

ex) loop and if-statement branches

- ① taken → backward
- ② not " → forward

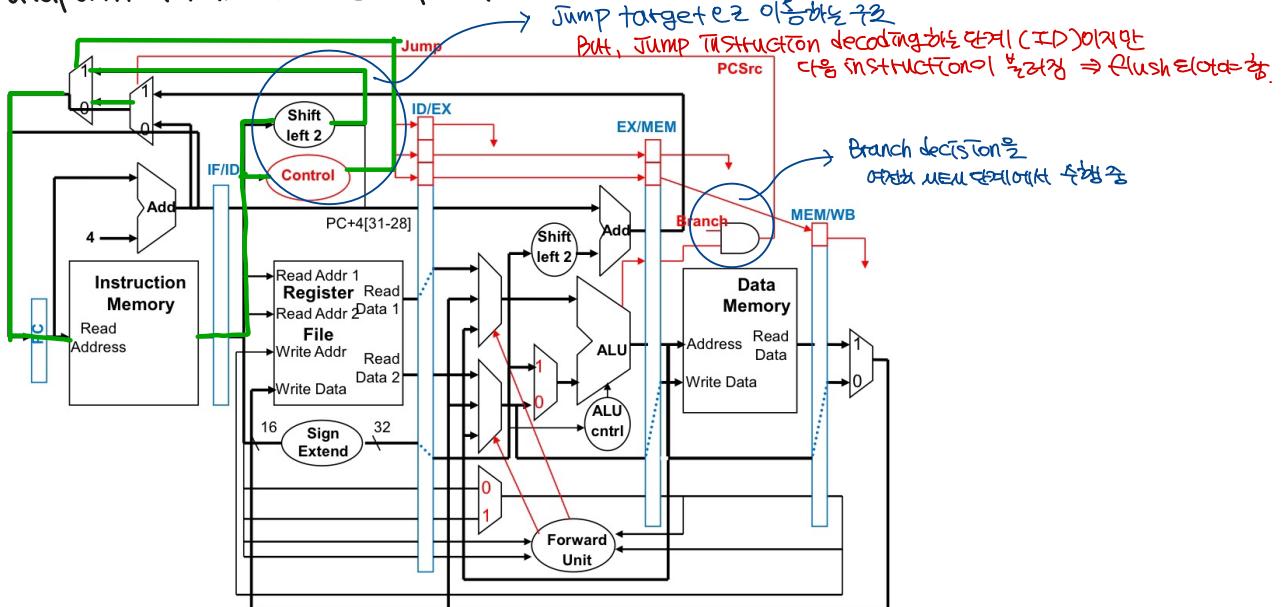
iv) Dynamic branch prediction (즈즈, HW이 support) ⇒ 뒤에 가서 더 자세히 내용

: HW의 branch behavior 기록 ⇒ 과거의 behavior를 통해 미래 예측
⇒ 예측 결과 틀리면 re-fetching 하는 동안 stall 일시 + history 갱신

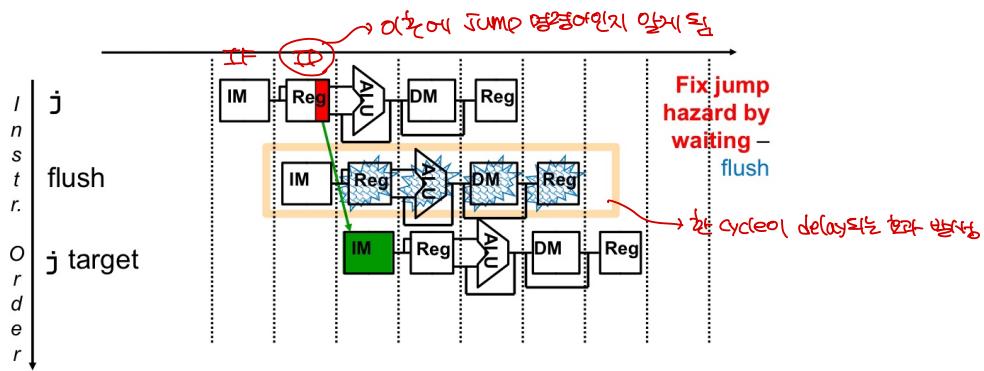
Solution ④

*delay Slot

d) Datapath Branch + Jump H/W



⇒ 한 cycle stalls이 일어나는 경우.



- + Flush를 통해 IF/ID pipeline register에 있는 IF.flush control signal을 0e2 set
→ 해당 instruction을 nop로 바꿔라
- + Jump는 잘 일어나지 X

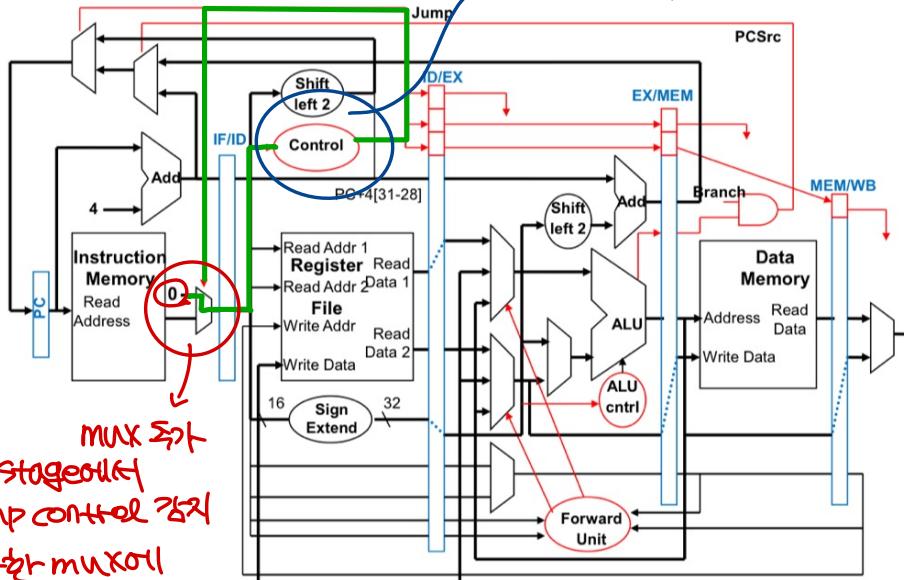
ii) stall 풀기 (참고)

- ① Nop (sw) : NOP 자체를 삽입
- ② Flush (flw) : pipeline 내부에서 멈춰다가 이후에 실행 (nop로 교체)

→ Jump operation인 것을 미리 감지

iii) ID stage Jump

④ mux로 의해 0을 control signal이 0015로
→ nop operation이 실행된다



⇒ 가능해도 어렵고 비효율적이지만
현실적으로 ID stage Jump 지원

iv) delay 를 줄이려는 노력

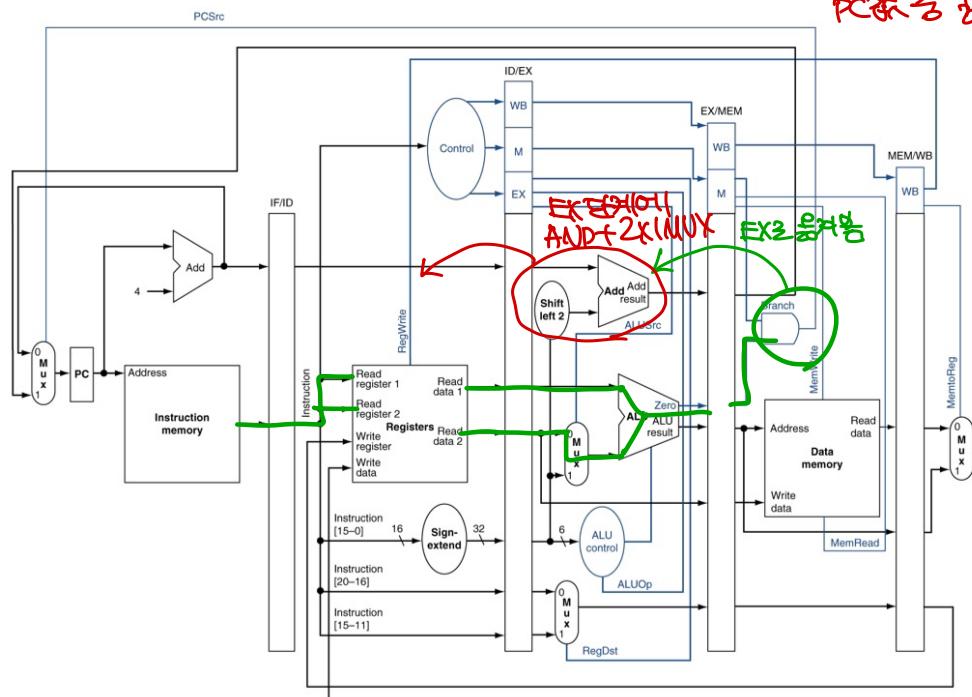
① Branch decision H/w를 EX stage로 옮긴다면?

[stall (flush) cycle] 2 → 2.2 감소

[EX timing path] AND, 2x1 MUX 추가

[AND]: 디지털 회로 만족 때만 branch 발생해야 함.

[MUX]: branch가 발생 혹은 발생하지 않음을 대비 PC Src 중 하나 선택하여 전달



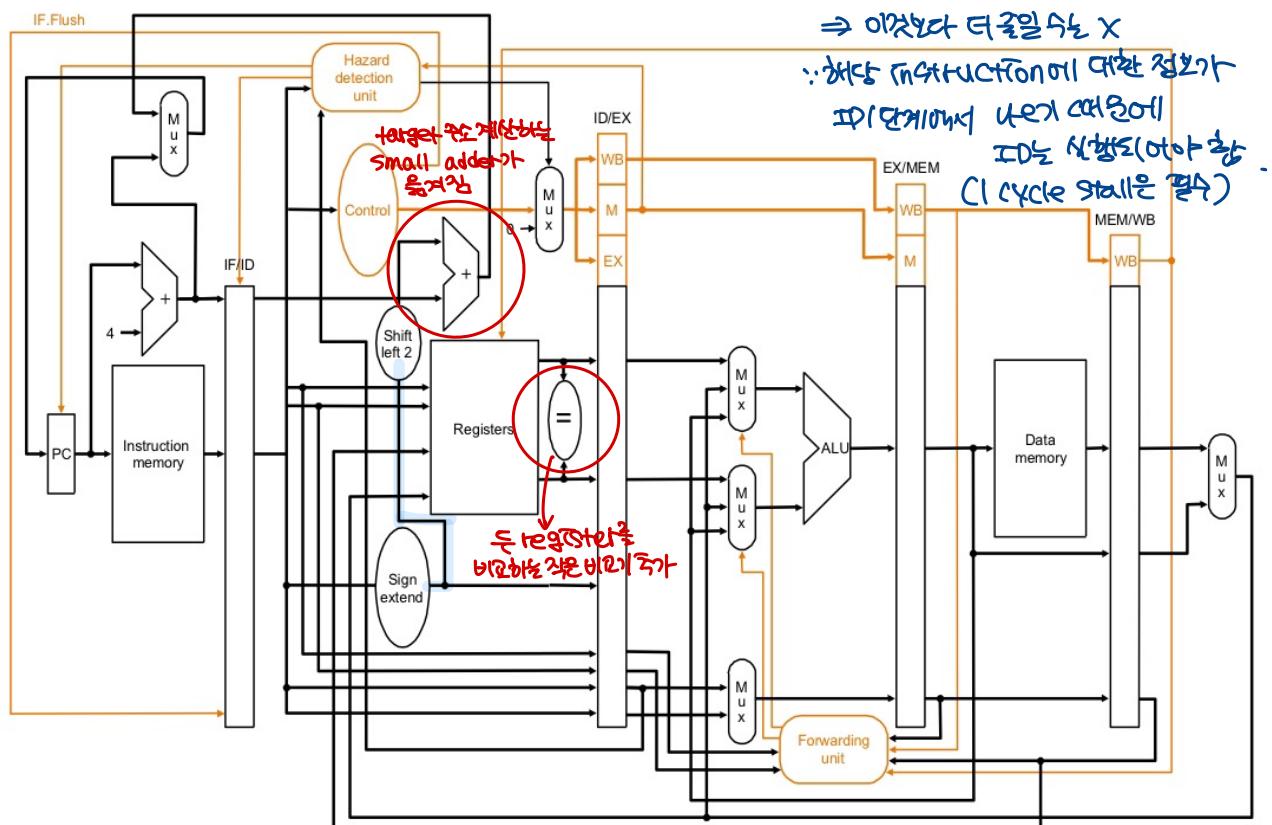
② ID stage에서 branch target address 계산 + branch decision 합침으로 H/w 추가?

[stall (flush) cycle] 2 → 1.3 감소 (but, EX에서 ID3 까지 stage 더 응집함)

⇒ ID stage에서 forwarding H/w가 필요함 (Data forwarding)

[Branch Target address 계산 → 모든 Register Read가 병렬로 가능]

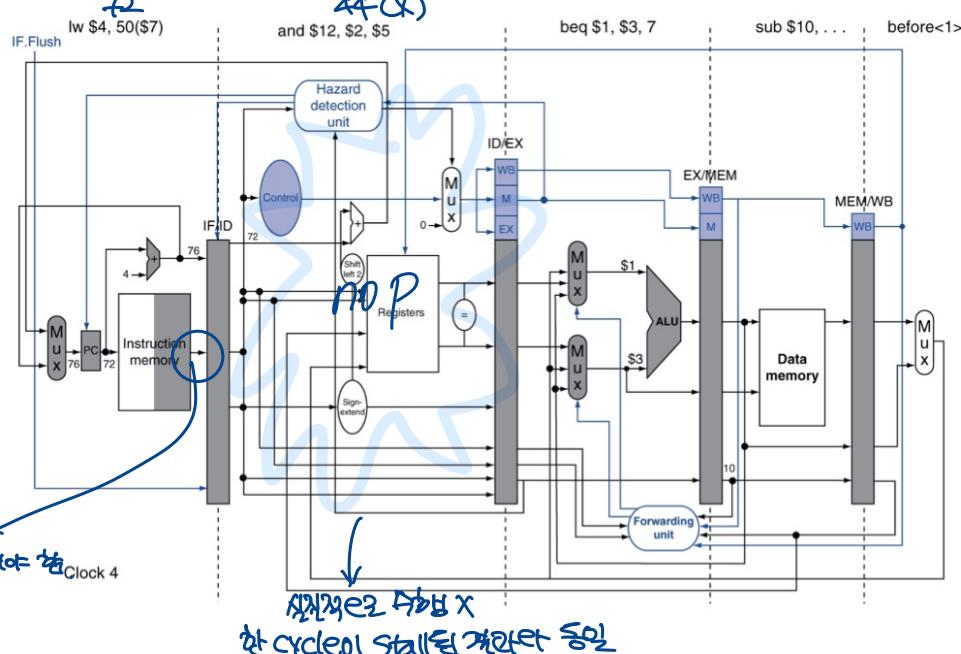
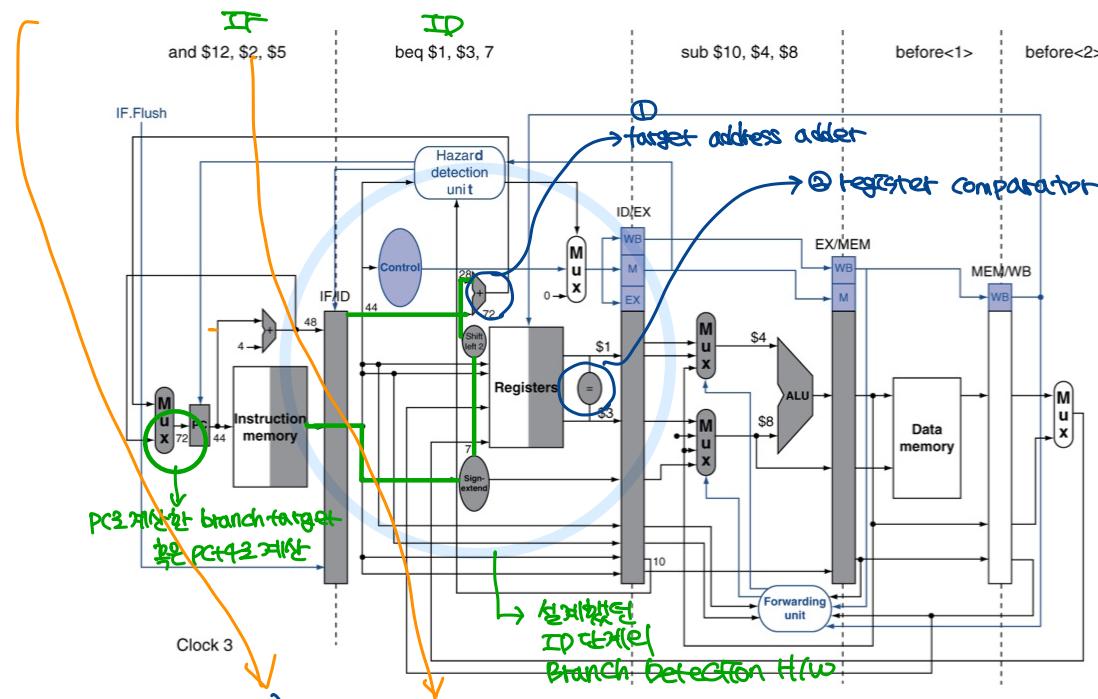
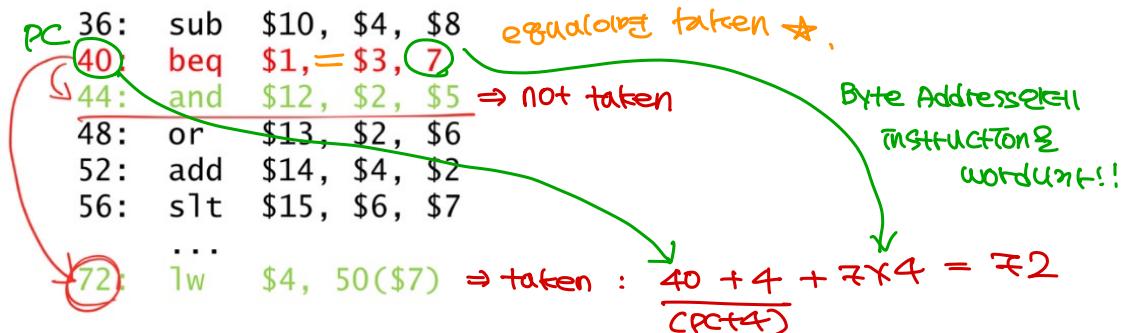
[Register가 되어 있는 Register Read 동작이 끝날 때까지 수행될 수 X ⇒ mux, comparator, AND 필요]



② 결론

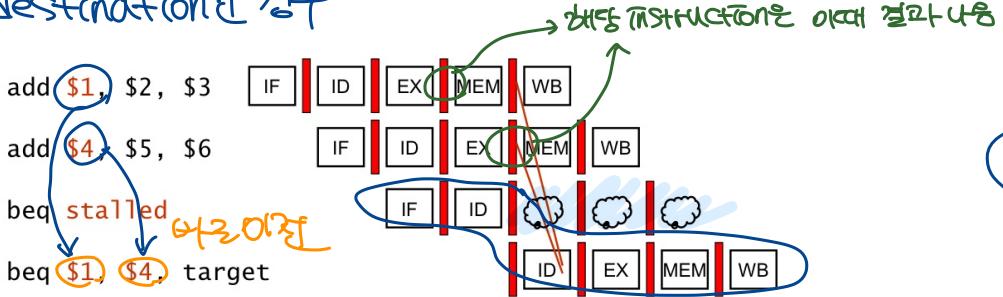
- Branch decisional outcome는 내부적으로 HW² ID stage 끝에로 떨어짐
 - 이가 할 것
 - ① Target Address Adder
 - ② Register Comparator
-) → IP stage에 추가

⑤ 예시) branch taken



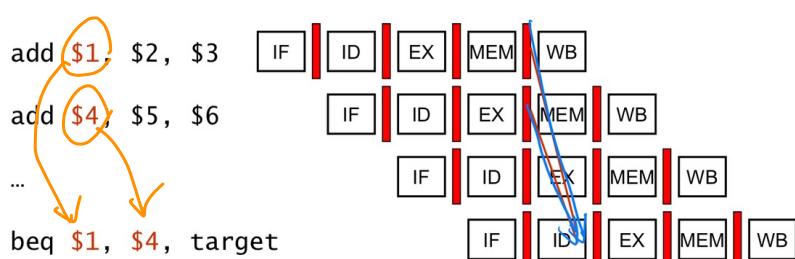
v) Data Hazards for Branches

- ① comparison register (beq 같은 것) 이 바로 이전 ALU instruction의 destination인 경우



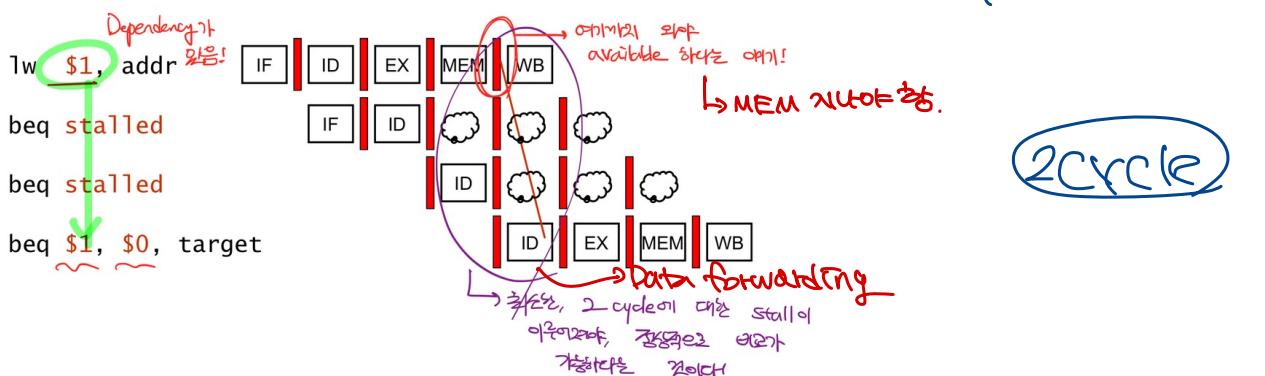
↳ begin decision point : ID stage까지 때간 상태로 신경 챙겼다고 가정
→ 1Cycle Stall을 하야하는 문제 발생
(data forwarding을 통해 data dependency 발생 가능)

②



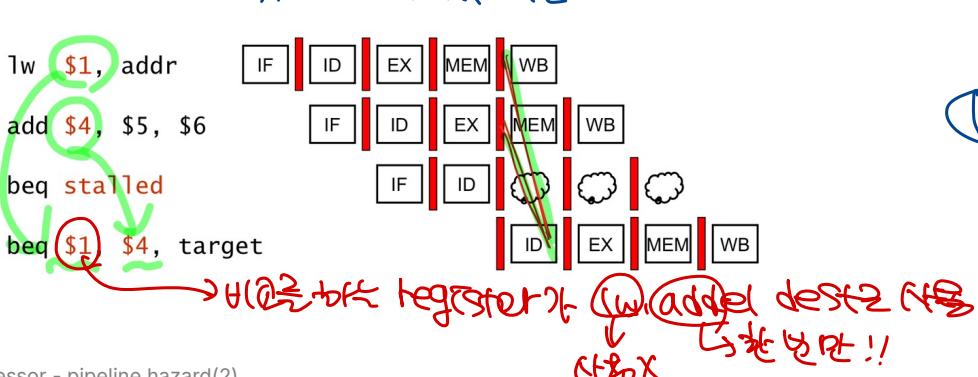
↳ stall 없이 data forwarding을 통해 해결 가능

③



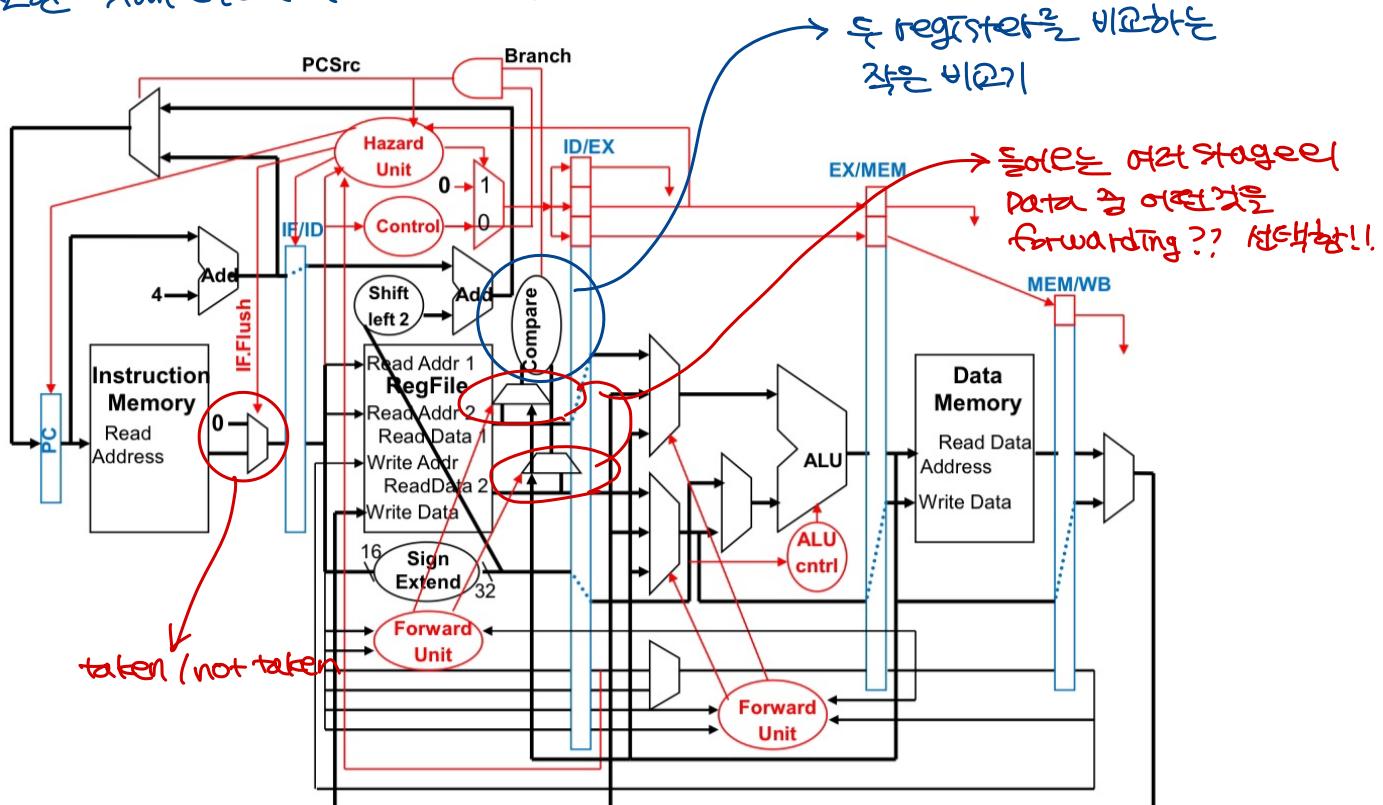
↳ begin decision point : ID stage까지 때간 상태로 신경 챙겼다고 가정
→ 2Cycle stall을 하야하는 문제 발생

④



⇒ instruction은什么时候 schedule 될까??

필요한 stall cycle이 달라진다!!



무엇을 주기로 하지 결정짓는다!!

vi) Delayed Branch ⇒ what delay slot!!

Branch가 필요한 명령어의 처리값이 나오는 동안

branch Brancher 관련이 있는 명령어 먼저 실행 ⇒ pipeline 흐름을 사용!!

① 만약 branch hw가 ID stage를 끌어가면 → delayed Branch 사용

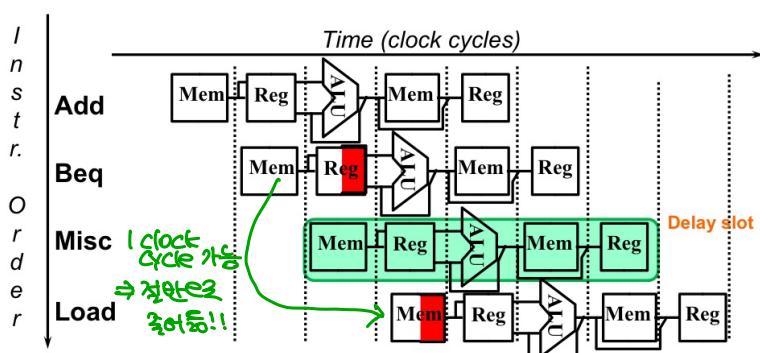
MIPS compiler : branch test 실행 직후 영향 받지 않는 instruction을 즉시 배치
⇒ branch delay 숨길 수 있음

② pipeline의 깊이 + Branch delay + → delay slot 여유가 필요

delayed branch : 값이 더 빨리 올라온다 허용도 X

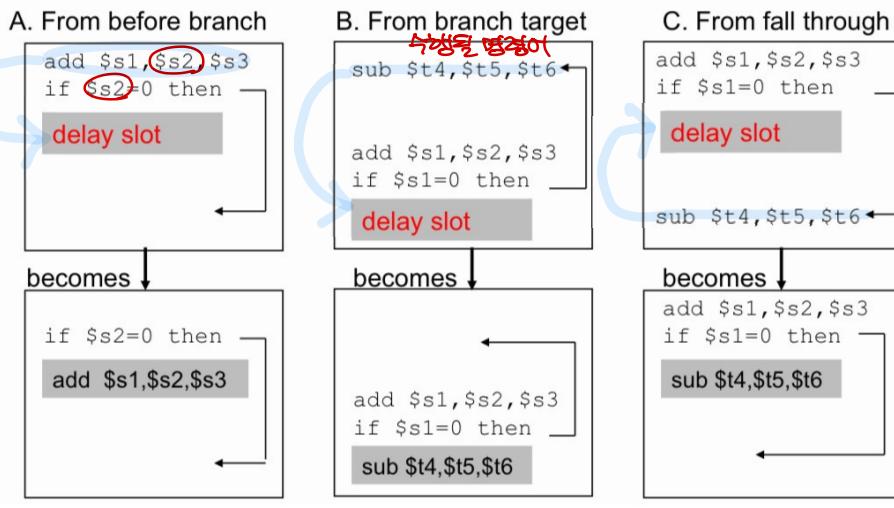
+ dynamic적인 hw가 상대적으로 저정해짐...

③ delay slot : branch instruction 이후의 branch behavior 재정의
⇒ delay slot이 branch 영향받지 않는 다른 instruction 처리되나?



But, 디지털 clock cycle마다 더 많은 INSTRUCTION 수령하기 위해 어떤 branch slot을 채울 수 있는 INSTRUCTION 찾는게 더 어려워짐 \Rightarrow 더 유용

④ Branch delay slot은 어려워지며 schedule 가능 시간 줄임!!



INSTRUCTION 공백 X
IC도 끊어짐 (stall X)
⇒ Best choice

혹시가 발생함!!

- ① not taken 일 때 꼭 실행되어야 하는 INST가 실행 X
- ② sub 명령어 copy & add 가 필요 \Rightarrow 오래 걸리기 때문

\Rightarrow not taken이 되어도 연산을 빙자 않도록!!

⑤ Static Branch Prediction \leftarrow solution ③ : prediction 실행!!

주어진 결과를 가정 + 실제 bbranch outcome을 확인하기 위해 기억하지 않고 진행

\Rightarrow branch hazard 해결

(cycle stall X)

• Predict not taken

branch가 예상한 not taken 된다고 예측 \Rightarrow sequential하게 IF에서도

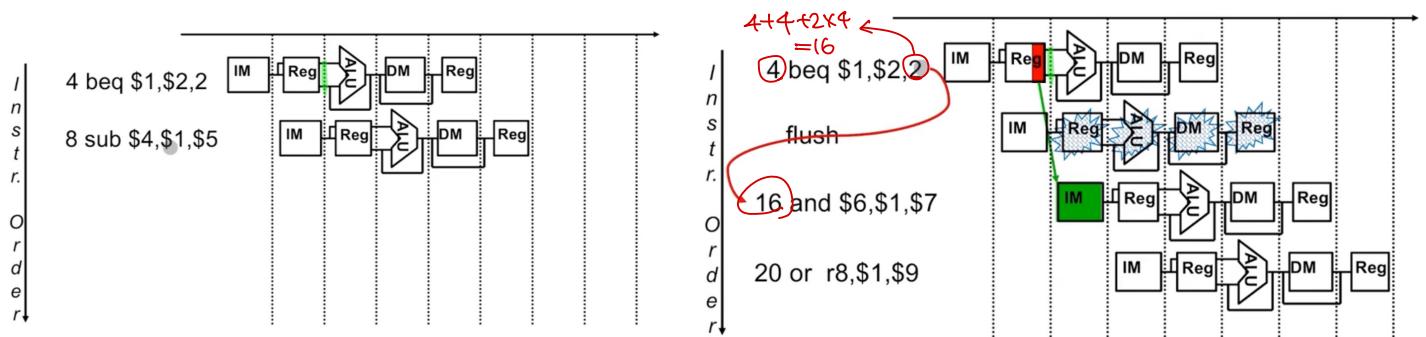
① taken되는 경우에만 stall 진행

\Rightarrow branch 이후에 있는 INSTRUCTION flush

- Branch \rightarrow [MEM] : flush \rightarrow [IF], [ID], [Ex]
- Branch \rightarrow [Ex] : flush \rightarrow [IF], [ID]
- Branch \rightarrow [ID] : flush \rightarrow [IF]

② flushed된 instructional machine state는 허용 X

③ branch destination에서 pipeline restart (호출자로 운영)



④ loop example

⇒ "predict not taken" of bottom of the loop

예측은 성능이 좋을까?

① : top of the loop

⇒ branch not taken 경우가 loop를 실행

→ stall 회피

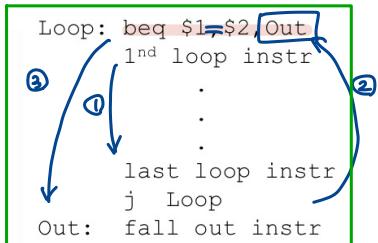
(taken은 loop 경우가 적어짐)

② : Bottom of the loop

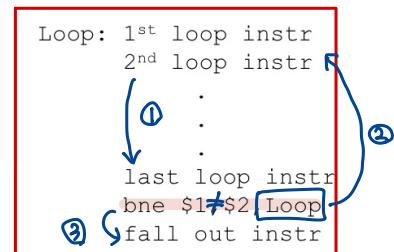
⇒ branch taken "

→ stall ↗

①



②



• predict taken

branch가 무조건 taken 된다고 예측 ⇒ 한정 one stall cycle 됨!

(∴ 항상 IF는 flush해야 함)

⇒ 경로선택, pipeline이 깊어질수록 branch penalty가 커짐

⇒ 추가적인 HW로 Dynamically 하지 수행?

⑥ Dynamic Branch Prediction

Run-time 이용하여 branch 예측 → 무드 H/w 사용?

BHT

* Branch - prediction buffer (branch history table)

Branch	Target	History
≡	≡	≡
⋮	↓	↓

특정 Branch instruction 주소들에 의한 indexing됨 ⇒ outcome 저장 (taken / not taken)

bz로 저장

① Branch 명령어 수행 어려움?

i) Table check → table에 적힌 outcome과 동일한 outcome 찾기

ii) fetching 뒤집

iii) 만약 outcome이 다르다면 [pipeline flush
+ prediction 결과 뒤집 바꿈]

② IF에 위치

[Index : PC addressel lower bits]

[taken 여부 [0 : not taken
1 : taken]]

③ 예측의 정밀도 → program 성능에 영향을 미치지 X

④ branch가 taken되는 시점(when) 알 예측 가능

taken 되어서 어디로 (where) 가능지 예측 불가

⇒ BTB 사용!

*Branch Target Buffer (BTB)

① IF 앤드 위치 → Branch Target Address를 catching

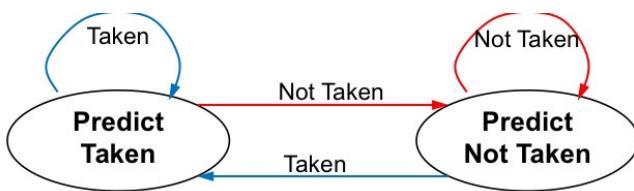
② 마지막, next sequential instruction을 fetch 필요

③ prediction bit → IF(ID register에 저장되어 있음.
→ next clock edge에서
다음 instruction 예상일지 결정

→ 다음 방법도 생각해보자

- ④ I-MEM : next sequential instruction을 fetch하는 동안
Cache : branch taken instruction 속행하기!
→ Avoid stall

• 1bit Prediction Accuracy



Loop: 1st loop instr
2nd loop instr
⋮
last loop instr
bne \$1,\$2,Loop
fall out instr

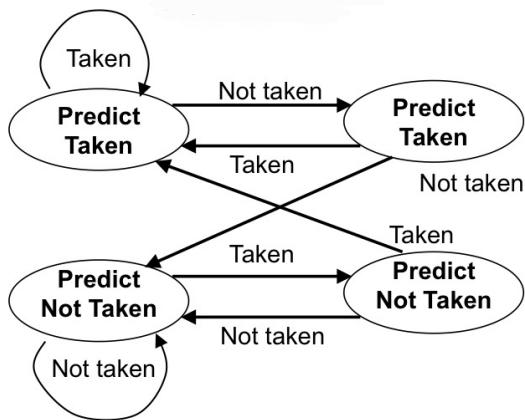
2번의 misprediction 발생 가능

① 1st : 이전에 등장한 적 있는 branch
→ mIS 가능

② last : 이전까지 loop가 계속 등장 → branch taken이거나 예측
→ mIS 발생

⇒ 10번 중 약 2번 mIS → 80%의 정확도

• 2bit Predictors



Loop: 1st loop instr
2nd loop instr
⋮
last loop instr
bne \$1,\$2,Loop
fall out instr

⇒ 10번 중 약 1번 mIS → 90%의 정확도