🌙

Ch.4-6 The Processor - ILP

* Instruction-Level Parallelism (ILP)

명령어 수준 병렬성 (pipeline의 병렬성)

- ILP 증가 방법?

① pipeline 깊이 증가 : stage 개수 ↑

각 stage 세분화 → CPI=1 고정, 대신 1 clock cycle 시간 ↓

② 다중 내보내기 (Multiple Issue) : 컴퓨터 내부의 구성 요소 여러 개

매번 stage에서 다수의 명령어 내보내도록 → 명령어 실행 속도 > clock rate

(CPI<1 , IPC>1)

ex) 4Ghz 4-way multiple issue

⇒ 16 BIPS , peak CPI = 0.25 , peak IPC=4

10^9

↳ data dependency 때문에 peak 값이 나오는 것은 힘듦.

Static Multiple Issue : Compiler에 의존 → HW적

- Issue slot에 같이 내보내질 명령어 묶어놓음
- 문제를 회피 + 탐지

Dynamic Multiple Issue : CPU가 역할수행 → SW적

- CPU가 매 cycle마다 함께 보낼 명령어 고름 + 흐름 조절
- Compiler가 명령어의 순서 재조정
- CPU가 run-time에 advanced한 기법 사용

→ 다양한 hazard 해결 시도

⇒ 현대의 computer는 대부분 dynamic multiple issue로 해결

* Speculation (추정)

더 많은 ILP를 찾아내고 이용하는 기법 → 예측을 기반으로 함

⇒ compiler 혹은 processor가 명령어의 특성에 대해 추측하도록 하는 것

- operation을 최대한 빨리 시작
- 추정한 것이 맞는지 check
  - 맞다면 → operation complete
  - 틀리면 → operation roll-back + 올바른 task 유도

(명령어 일찍 수행 가능)

ex) branch outcome에 대해 추정 ⇒ path taken이 다르다면 roll-back

ex2) sw → lw 일 때 동일한 주소를 참조하지 않는다고 추정 ⇒ lw가 sw보다 먼저 실행

→ location이 update되면 roll-back

① compiler / HW speculation

- compiler : 명령어 순서 재배치 가능
  (incorrect한 guess에 대해 다시 roll back 가능하도록 "fix-up" 명령어 필요 가능)
- HW : 선실행 후 결과를 temporary buf에 저장
  → 필요할 때가 될 때까지 대기
  ⇒ 추정한 값이 맞으면 때가 됐을때 실행 / 아니면 flash Buffer

② 만약 추정한 명령어에서 exception 발생?

- Static speculation : 추정 실패 시 사용할 service routin 가지고 있음. ↙ ISA support
- Dynamic Speculation (HW based) : HW가 exception 만든 offering instruction을 buffering
  → 추정 실패시, flush buffer

* Static Multiple Issue

① Issue packets : 한 cycle에 넣을수 있는 instruction group

compiler → 명령어들을 Issue packets으로 groupping

i) group은 single cycle에 처리 가능

ii) 필요한 pipeline resource에 대해 Issue packet 내용이 결정됨

iii) Very Long Instruction Word (VLIW) : 엄청 긴 하나의 명령어로 생각해보기

⇒ compiler의 support가 필요함

② Scheduling 어떻게?

⇒ Compiler : some/all Hazard 제거

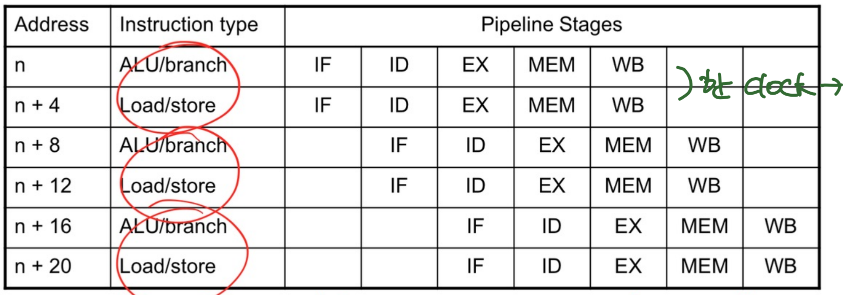
i) hazard가 안 생기도록 명령어 재구성 필요

ii) Issue packet 내부에는 dependency X
   " packet 간에는 dependency 가능 → ISA마다 다름, compiler는 반드시 알아야 함

iii) nop가 필요하다면 사용 → 많이 쓸수록 성능 나빠짐 → 가능하다면 사용X

③ example : Static dual Issue (pipeline이 2개)

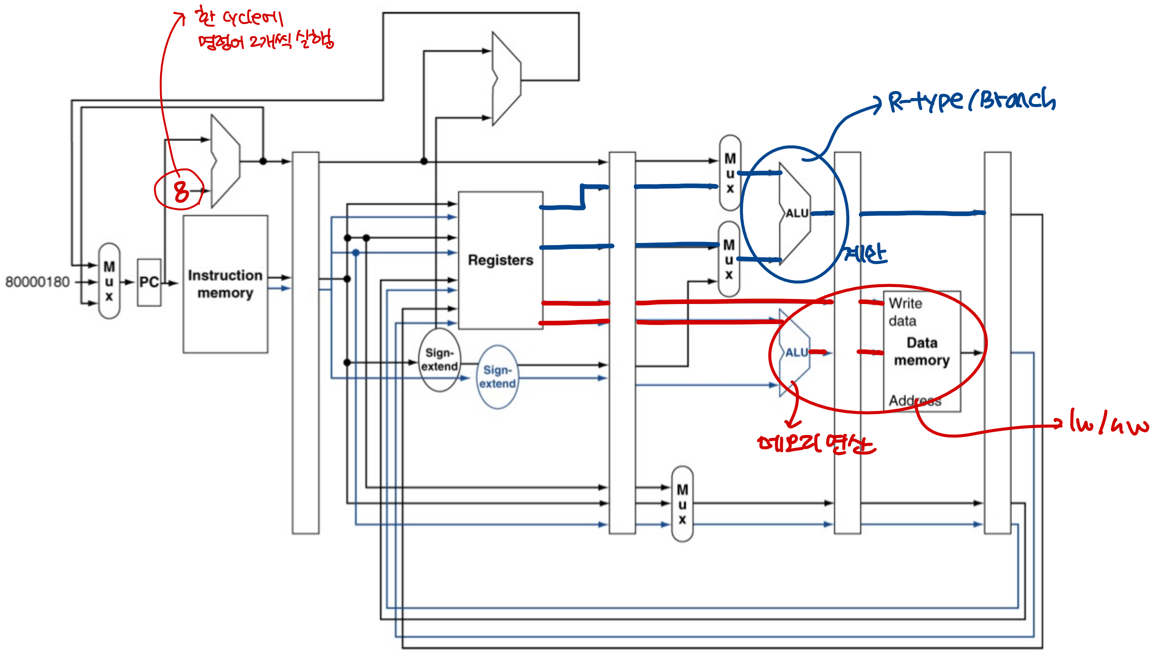
⇒ two Issue packet [ALU/ Branch (32bit) → 하나의 branch
                      lw / sw (32bit) → 하나의 branch.



Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

} 한 clock → 2개의 명령어

⇒ 가장 이상적인 동작 방식.. / CPI = 0.5 / IPC = 2



1) Static dual multiple issue에서 발생할 수 있는 Data Hazard ?

↳ 더 많은 명령어들이 병렬로 돌아감

- EX에서 발생하는 data Hazard

  → 한 cycle stall 발생

```
add $t0, $s0, $s1
load $s2, 0($t0)
```

8차이

dependent

But, 1개의 packet 사용불가

⇒ 동시에 issue할수 X

- Load/use Hazard

  → 여전히 1 cycle의 latency But, 2개의 명령어가 동시에 실행

```
Loop: lw   $t0, 0($s1)       # $t0=array element
      addu $t0, $t0, $s2     # add scalar in $s2
      sw   $t0, 0($s1)       # store result
      addi $s1, $s1,–4       # decrement pointer
      bne  $s1, $zero, Loop  # goto Loop if $s1!=0
```

dependency X 먼저 실행

한 cycle 비워야 함

한cycle 비워야 함

	ALU/branch	Load/store	cycle
Loop:	nop	lw $t0, 0($s1)	1
	addi $s1, $s1,–4	nop	2
	addu $t0, $t0, $s2	nop	3
	bne $s1, $zero, Loop	sw $t0, 4($s1)	4

↳ dependency X ⇒ 상관 X

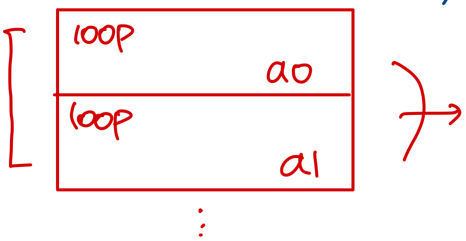
- IPC = 5/4 = 1.25 (c.f. peak IPC = 2)

④ Loop Unrolling

Loop body를 복제하여 더 병렬적인 형태로 만듦 ⇒ Loop-control overhead ↓

↳ Register renaming : 복제본마다 다른 register 사용 (같은 이름 X)

⇒ loop에서 나오는 anti-dependence를 피하게 함 (name dependence)



→ 여러개로 풀어헤침

- example

Loop Unrolling Example

```
Loop: lw   $t0, 0($s1)
      addu $t0, $t0, $s2
      sw   $t0, 0($s1)
      addi $s1, $s1,-4
      bne  $s1, $zero, Loop
```

loop body

loop control

	ALU/branch	Load/store	cycle
Loop:	addi $s1, $s1, -16	lw $t0, 0($s1)	1
	nop	lw $t1, 12($s1)	2
	addu $t0, $t0, $s2	lw $t2, 8($s1)	3
	addu $t1, $t1, $s2	lw $t3, 4($s1)	4
	addu $t2, $t2, $s2	sw $t0, 16($s1)	5
	addu $t3, $t4, $s2	sw $t1, 12($s1)	6
	nop	sw $t2, 8($s1)	7
	bne $s1, $zero, Loop	sw $t3, 4($s1)	8

명령어 4개를 풀어헤쳤을때에 -4 대신에 -16!

→ Register Renaming을 해줬다!

- ❑ IPC = 14/8 = 1.75
  - ▸ Closer to 2, but at cost of registers and code size

* Dynamic Multiple Issue (HW support)

① Superscalar processors Hazard 없다고 판단되면 바로 명령어 issue → 실행

② CPU : Compiler에 다음 → Compiler의 도움 받지 않으려 함.

③ Dynamic Pipeline scheduling

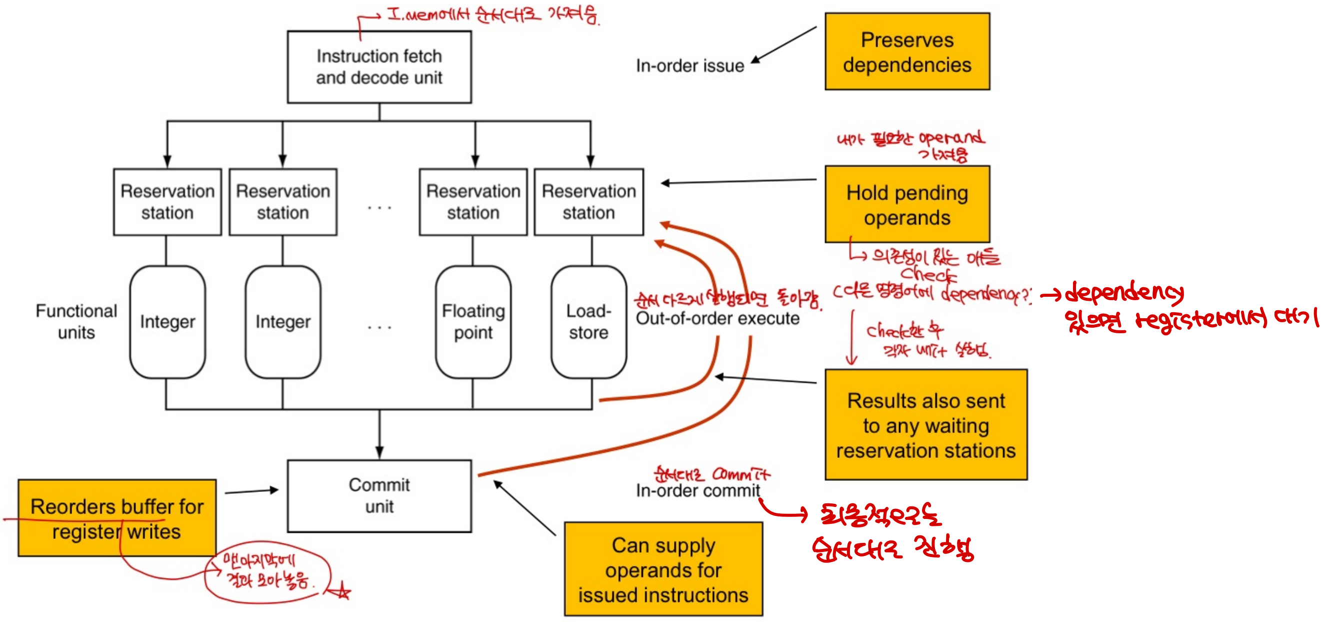
⇒ CPU가 임의적으로 instruction order 바꿔서 실행

→ But, result를 commit(=write) 하는 작업은 작성된 순서대로 실시

```
lw     $t0, 20($s2)
addu   $t1, $t0, $t2
sub    $s4, $s4, $t3
slti   $t5, $s4, 20
```

→ addu가 lw을 기다리는 동안 sub 먼저 실행 가능

⇒ data dependency 타격 x



I.mem에서 순서대로 가져옴.
Instruction fetch and decode unit
In-order issue
Preserves dependencies
내가 필요한 operand 가져옴
Reservation station
Reservation station
Reservation station
Reservation station
Hold pending operands
의존성이 있는 애들 check
Functional units
Integer
Integer
Floating point
Load-store
순서 다르게 실행되면 돌아감.
Out-of-order execute
(다른 명령어에 dependency?) → dependency 있으면 register에서 대기
check한 후 각자 바로 실행.
Results also sent to any waiting reservation stations
Commit unit
순서대로 commit
In-order commit
→ 최종적으로는 순서대로 진행
Reorders buffer for register writes
맨마지막에 결과 모아놓음.
Can supply operands for issued instructions

④ Register Renaming → Reservation Station / Reorder buffer에서도 효과적

Reservation Station으로 instruction issue 해오는 경우

- operand 결과가 만들어진 상태
  - i) operand를 reservation station에 copy
  - ii) dependency 발생 X (overwrite도 가능)
- operand 결과 만들어지지 않은 경우
  - i) function unit에서 결과 생성 ⇒ Data Forwarding ⇒ reservation station에 제공
  - ii) register update 필요X (temporary result)

⑤ speculation ⇒ 무슨 일??

- Predict Branch + continue issuing : branch outcome이 결정될 때까지 commit X
- Load speculation
  - i) Load & Cache miss delay 피하려 예측
  - ii) Speculation이 clear 될 때까지 lw 실행 X
    - → clear되면 mark 모두 지움
    - ⇒ 순차적으로 결과 commit

⇒ Dynamic Scheduling 하는 이유? Compiler에서 모두 다 하지 않는 이유?

① 모든 stall을 compiler에서 predict X

ex) cache misses ⇒ run time으로만 파악 가능

② Branches 관련하여 언제나 scheduling 가능한 것은 X

↳ outcome 동적으로 결정됨

→ ISA마다 다르게 compile + execution

⇒ Data dependency → 성능 제한, 몇몇은 제거 어려움, 병렬성 확장 곤란

Memory delays & limited bandwidth 문제

⇒ speculation은 굉장히 큰 도움

Fallacies

- Pipelining은 굉장히 쉬운 개념이다(?)
    - Basic Idea 자체는 매우 쉽다.
    - 그런데, 디테일하게 들어가면 굉장히 머리 아프다는 거다.

      ex) Data Hazards를 detecting하는 것 등등..
- Pipelining 기법의 원리 자체는, technology와 independent하다.

  그러면, 우리는 왜 항상 pipelining을 사용하지 않는 것일까?
    - Transistor가 많이 필요하고, 이에 따라 더 advanced한 테크닉들이 필요하고.. 하여튼 실현하기 위해서 많은 수고가 든다.
    - CISC는.. pipeline 구현이 너무 힘든데, RISC의 등장으로 pipeline 구현 난이도가 많이 내려갔다.

      ex) predicated instructions (조건부 실행 명령 / 명령어가 특정 조건에 따라 실행되거나 무시 되도록 하는 방법)

Pitfalls

- ISA design이 쓰레기면, pipelining을 구현하는 게 어려워진다.

  ex) complex instruction sets(VAX, IA-32)

  → pipelining이 동작하기 위해 상당한 overhead가 발생한다.

  → IA-32 micro-op approach (micro-operation approach / AMD에서는 ROP이라고 부른다)

  ex2) complex addressing modes

  → Register update를 하며 side effects, memory indirection 등..

  ex3) delayed branches

  → Advanced pipelines에는 긴 delay slots가 있다.