8. 고급 프로세서 구조

8.1 파이프라인 단계별 분석 (IF, ID, EX, MEM, WB)

> IF → ID → EX → MEM → WB

🧠 파이프라이닝이란?

파이프라이닝(Pipelining)은 명령어 실행 단계를 **여러 단계로 나누고**,
여러 명령어를 동시에 병렬로 처리하는 방식이다.
공장 조립라인처럼, 각 스테이지에서 명령어의 일부만 처리하고 다음 스테이지로 넘긴다.

> 단일 사이클에서 하나의 명령어 전체를 처리하는 대신,
> 여러 명령어가 **겹쳐서 실행되기 때문에 처리율(Throughput)이 향상됨**.

🔁 기본 5단계 파이프라인 구조

단계	이름	역할
IF	Instruction Fetch	명령어 메모리에서 명령어를 읽어옴
ID	Instruction Decode / Register Fetch	명령어 해석, 레지스터 읽기
EX	Execute / Address Calculation	연산 수행 또는 메모리 주소 계산
MEM	Memory Access	데이터 메모리 읽기/쓰기
WB	Write Back	결과를 레지스터에 다시 저장

🔍 각 단계 상세 분석

🔹 1. IF (Instruction Fetch)

- **PC (Program Counter)**를 기반으로 명령어 메모리에서 **명령어를 읽음**
- 읽은 명령어는 **파이프라인 레지스터(IF/ID)에 저장**
- **PC += 4** (일반적으로 다음 명령어로 이동)

하드웨어 구성

- PC 레지스터
- 명령어 메모리 (Instruction Memory)
- IF/ID 레지스터

🔹 2. ID (Instruction Decode / Register Fetch)

- 명령어를 해석하여 **Opcode, 피연산자, 제어 신호 추출**
- 필요한 **레지스터 값을 읽음** (ex: rs1, rs2)
- 분기 조건 계산을 위한 초기 비교 수행 가능
- 제어 유닛(Control Unit)에서 **다음 단계용 신호 생성**

하드웨어 구성

- 레지스터 파일 (Register File)
- 제어 유닛 (Control Logic)
- ID/EX 파이프라인 레지스터

🔹 3. EX (Execute)

- **산술 연산 or 주소 계산 or 분기 판단 수행**
- 산술 명령: ALU가 연산 수행
- Load/Store 명령: 메모리 주소 계산
- Branch: 조건에 따라 PC 변경 여부 판단

하드웨어 구성

- ALU
- ALU Control
- 포워딩 유닛 (Hazard 해결용)
- EX/MEM 레지스터

🔹 4. MEM (Memory Access)

- Load: 계산된 주소에서 **데이터 읽음**
- Store: 계산된 주소에 **데이터 저장**
- 산술 연산의 경우 **MEM 단계를 건너뛰고** 결과만 패싱

하드웨어 구성

- 데이터 메모리
- 메모리 제어 로직 (Read/Write)
- MEM/WB 레지스터

🔹 5. WB (Write Back)

- 연산 결과 or 메모리에서 읽은 데이터를
  레지스터 파일에 되돌려 씀

하드웨어 구성

- 레지스터 파일 (쓰기 포트)
- 다중 선택기 (ALU vs Mem 결과 선택)

🎯 파이프라인 명령어 흐름 예시 (5단계)

```
Cycle    IF        ID        EX        MEM       WB
-----------------------------------------------------
1        I1
2        I2        I1
3        I3        I2        I1
4        I4        I3        I2        I1
5        I5        I4        I3        I2        I1
6                  I5        I4        I3        I2
...
```

→ **처리율(Throughput)**: 한 사이클당 1개 명령어 완료 (최적 조건 시)

⚠️ 파이프라인의 위험 요소 (Hazards)

✅ 1. 구조적 위험 (Structural Hazard)

- 하드웨어 자원을 동시에 요구하는 명령어 간 충돌
- 해결: 자원 분리 or 스톨

✅ 2. 데이터 위험 (Data Hazard)

- 이전 명령의 결과가 아직 준비되지 않았는데 다음 명령이 사용함

종류	예시
RAW (Read After Write)	다음 명령이 이전 결과를 사용
해결	포워딩(Forwarding), 스톨 삽입(Bubble)

✅ 3. 제어 위험 (Control Hazard)

- 분기/점프 명령 실행 전, 다음 명령의 흐름 결정 불확실
- 해결: 분기 예측 (Branch Prediction), 딜레이 슬롯

🧠 고급 최적화 기술

기술	목적
Forwarding	결과를 다음 스테이지로 바로 전달
Stall 삽입	해석 불가능한 상황에서 1사이클 쉬기
Branch Prediction	분기 예측으로 IF 단계 계속 진행
Superscalar	여러 명령어 동시 발행
Out-of-order 실행	순서와 관계없이 실행 가능하면 먼저 실행

📌 요약 정리

단계	이름	주요 기능
IF	Instruction Fetch	명령어 메모리에서 명령어 가져옴
ID	Instruction Decode	명령어 해석, 레지스터 읽기
EX	Execute	ALU 연산, 주소 계산
MEM	Memory Access	메모리 읽기/쓰기
WB	Write Back	결과를 레지스터에 저장

8.2 파이프라인 해저드와 해결 방식

> Pipeline Hazards & Resolutions

🧠 해저드(Hazard)란?

파이프라인 해저드란 명령어들이 겹쳐서 실행되는 동안,
정상적인 실행 흐름을 방해하거나 잘못된 결과를 초래할 수 있는 상황을 말한다.

> 해저드가 발생하면 일부 명령어는 **기다리거나 순서를 변경**해야 함
> → 이로 인해 **성능 저하** 발생 가능

📊 해저드의 세 가지 종류

종류	설명
① 구조적 해저드 (Structural Hazard)	하드웨어 자원이 부족하여 충돌
② 데이터 해저드 (Data Hazard)	이전 명령의 결과가 필요하지만 아직 준비되지 않음
③ 제어 해저드 (Control Hazard)	분기 명령 이후 명령 흐름이 결정되지 않음

🔹 1. 구조적 해저드 (Structural Hazard)

✅ 정의

하드웨어 자원이 부족해서 **동시에 실행하려는 명령어들이 자원을 공유**하려 할 때 발생

✅ 예시

- 하나의 메모리를 **명령어 메모리(IF)**와 **데이터 메모리(MEM)**가 동시에 사용하려고 할 때

✅ 해결 방법

- **자원 분리**: 명령어용 메모리와 데이터용 메모리를 분리 (Harvard 구조 등)
- **멀티 포트 메모리**
- **파이프라인 스톨** (대기 사이클 삽입)

🔹 2. 데이터 해저드 (Data Hazard)

✅ 정의

명령어 간의 데이터 의존성으로 인해
다음 명령어가 필요한 데이터를 아직 이전 명령어가 제공하지 못한 상태

📦 대표 유형

유형	의미	예시
RAW (Read After Write)	다음 명령이 이전 명령의 결과를 읽음	`ADD x1, x2, x3` → `SUB x4, x1, x5`
WAW (Write After Write)	두 명령이 같은 레지스터에 쓰려고 함	`ADD x1, ...` → `SUB x1, ...`
WAR (Write After Read)	다음 명령이 레지스터를 덮어쓰려 할 때 이전 명령이 아직 읽지 않음	out-of-order에서 발생

> 보통은 **RAW**만 파이프라인에서 자주 발생
> WAW, WAR는 **동시 실행 프로세서 (Out-of-Order)**에서 주로 나타남

✅ 해결 방법

✔ 1. 포워딩(Forwarding, Bypassing) ⭐

- 아직 레지스터에 저장되지 않았지만,
  ALU 출력 등 중간 결과를 다음 명령어로 직접 전달

```
EX 단계에서 계산한 값을 → 다음 명령의 EX 단계로 바로 전달
```

- 대부분의 현대 프로세서는 하드웨어 포워딩 유닛 포함

✔ 2. 스톨(Stall, Bubble)

- 데이터가 준비될 때까지 파이프라인을 멈추고 **NOP(Bubble)**을 삽입

```
Cycle     I1    I2    I3
-----------------------------
1         IF
2         ID    IF
3         EX    ID    IF
4         MEM   NOP   ID
5         WB    EX    EX
```

- 비효율적이지만 단순한 구조에서는 사용됨

✔ 3. 컴파일러 레벨 스케줄링

- 명령어 간 **의존성이 없는 명령을 중간에 삽입**하여 해저드 회피

```
ADD x1, x2, x3
NOP          ; ← RAW 해저드 회피
SUB x4, x1, x5
```

🔹 3. 제어 해저드 (Control Hazard)

✅ 정의

분기(Branch), Jump 명령어로 인해 **다음 명령어가 무엇인지 확정되지 않은 상태**

✅ 발생 원인

- IF 단계에서 분기 명령어를 읽지만,
  실제로 분기 여부 판단은 **EX 단계 이후**
  → 그 전까지는 다음 명령어가 **실행해도 될지 알 수 없음**

✅ 해결 방법

✔ 1. 분기 지연 슬롯 (Delay Slot)

- 분기 후의 한 사이클은 **무조건 실행되도록 설계**

```
BEQ x1, x2, target
NOP              ; ← 분기 여부와 무관하게 실행됨
```

✔ 2. 정적 분기 예측

- 항상 **분기 안 함** 혹은 **항상 분기함**으로 가정
- 성능은 낮지만 구현은 간단함

✔ 3. 동적 분기 예측 (Branch Prediction) ⭐

- 과거 분기 결과를 기반으로 분기 여부를 예측
- **BTB(Branch Target Buffer)**, **BHT(Branch History Table)** 등 하드웨어 예측기 사용
  → 예측이 틀리면 파이프라인 **플러시(Flush)** 후 다시 실행

✔ 4. 분기 조기 평가

- ID 단계에서 분기 여부를 판단하도록 ALU 위치 조정
- 빠른 분기 결정으로 해저드 최소화

🧪 종합 예시

```
Cycle   ADD x1, x2, x3   SUB x4, x1, x5
----------------------------------------
1       IF
2       ID               IF
3       EX               ID (x1 아직 없음 → RAW)
4       MEM              STALL (bubble 삽입)
5       WB               EX (x1 결과 포워딩됨)
```

📌 요약 정리

해저드	설명	해결 방법
구조적	자원 중복 사용	자원 분리, 스톨
데이터	명령 간 데이터 의존성	포워딩, 스톨, 명령 스케줄링
제어	분기 실행 경로 미확정	분기 예측, 플러시, 딜레이 슬롯

8.3 슈퍼스칼라 아키텍처

> Superscalar Architecture – 병렬 명령 실행의 정점

🧠 슈퍼스칼라란?

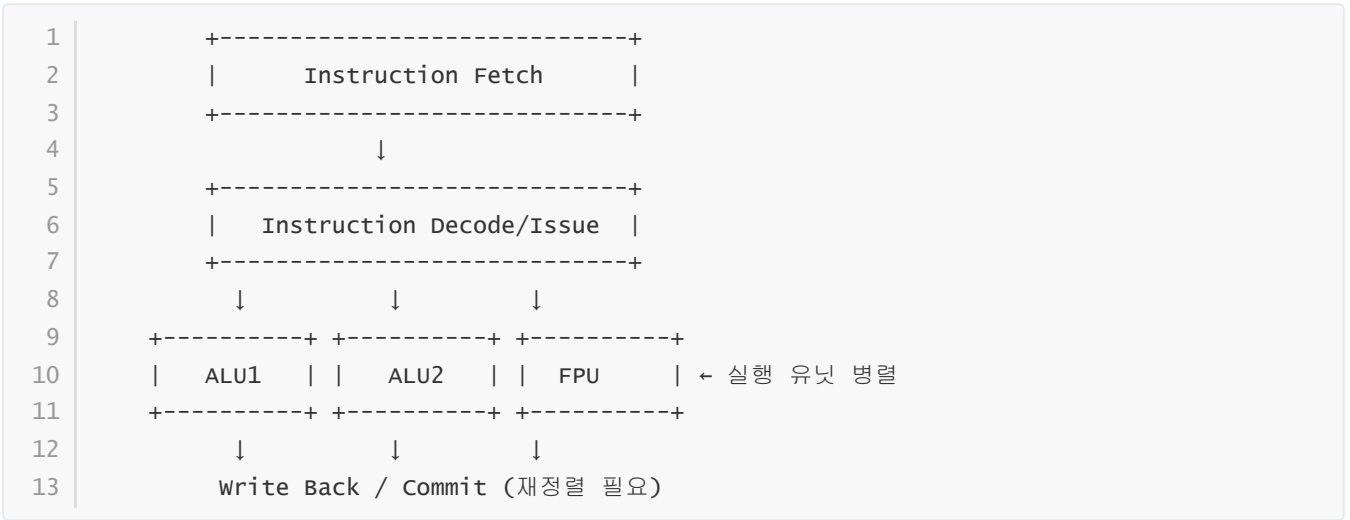
슈퍼스칼라(Superscalar) 아키텍처는 **한 사이클에 여러 개의 명령어를 동시에 실행**할 수 있도록
복수의 실행 유닛을 병렬로 갖춘 CPU 구조이다.

> 기존 파이프라인이 "세로 분할"이었다면,
> 슈퍼스칼라는 "가로 확장"된 구조.
> 즉, **IF → ID → EX ...** 단계를 여러 명령어가 **같은 시간에 병렬로 통과**함.

🔁 슈퍼스칼라 vs 파이프라인 비교

항목	파이프라인	슈퍼스칼라
처리량	명령어 겹치기 (1사이클 1개)	1사이클 여러 개 가능
실행 유닛	1개씩 순차적 사용	복수 ALU, FPU, Load/Store 등
병렬성	시간적 병렬성	시간 + 명령 병렬성
복잡도	낮음	높음 (의존성 분석 필요)

📦 슈퍼스칼라 기본 구조



- **명령어 발행(Instruction Issue)** 시 여러 개 명령어가 동시에 발행될 수 있음
- 각 명령은 독립된 실행 유닛(ALU, FPU, Load/Store 등)에서 실행

🧩 주요 구성 요소

구성 요소	설명
Instruction Fetch	여러 명령어를 동시에 가져옴 (멀티 Fetch)
Instruction Decoder	병렬로 실행 가능한지 분석
Dispatch/Issue Unit	실행 유닛에 명령어를 할당
Reservation Stations	실행 대기 큐 (동적 명령 발행 시 필요)
Reorder Buffer (ROB)	Out-of-Order 실행 후 순서 복구

🎯 슈퍼스칼라의 실행 방식

✅ Static Superscalar (정적)

- 컴파일러가 병렬 가능 명령을 분석해 스케줄링
- 비교적 단순하지만 **최적화 한계 존재**

✅ Dynamic Superscalar (동적) ⭐

- 하드웨어가 **실행 시점에 명령 간 의존성 분석, 재배치, 발행**
- **Out-of-Order 실행, 동적 스케줄링**, 명령어 윈도우 기반
- 예: **Tomasulo's Algorithm**, **Scoreboarding**

🧠 인스트럭션 발행(Instruction Issue)

방식	설명
단일 발행	한 번에 하나의 명령어만 발행
다중 발행 (Dual/Triple Issue 등)	1사이클에 2~4개 명령어 발행 가능
조건	의존성이 없어야 병렬 발행 가능 (RAW, WAW 등 회피 필요)

🔧 대표적인 실행 유닛 구성

유닛	역할
ALU1, ALU2	정수 연산
FPU	부동소수점 연산
LSU	Load/Store 연산
Branch Unit	분기 판단 처리

유닛	역할
SIMD Unit	병렬 벡터 연산 (AVX, NEON 등)

📊 성능 향상 예시

- 파이프라인 CPU: 1사이클 1명령어
- 듀얼 이슈 슈퍼스칼라: 1사이클 2명령어 → **이론상 2배 Throughput**
- 쿼드 이슈: 1사이클 최대 4개

> 단, **명령 간 의존성, 분기, 캐시 미스** 등이 많으면 이론적인 성능 향상에 못 미침

⚠️ 슈퍼스칼라의 한계점

요소	설명
제어 복잡도	의존성 분석, 예약, 재정렬 회로 필요 → 하드웨어 비용 증가
분기 발생 시 성능 저하	Branch Prediction 실패 시 손해 큼
메모리 병목	명령/데이터 대역폭 부족 시 실행 유닛 놀게 됨
명령어 병렬성 한계	실제 프로그램은 의존성이 많음 → 평균 2~3 Issue가 현실적

💡 대표 CPU 예시

CPU	설명
Intel x86 (Pentium Pro 이후)	Dynamic Superscalar + Out-of-Order + Speculative
ARM Cortex-A 시리즈	2~4 Issue Core, NEON 포함
MIPS R10000	Tomasulo 기반 Out-of-Order 슈퍼스칼라
Apple M1	고급 슈퍼스칼라 + 폭넓은 명령어 윈도우

📌 요약 정리

항목	설명
슈퍼스칼라	1사이클에 여러 명령어를 병렬 실행하는 CPU 구조
핵심	다중 실행 유닛, 명령어 병렬 발행, 의존성 분석
이득	처리율 대폭 증가 (Throughput ↑)
조건	병렬 명령어가 있어야 함 (의존성 낮을수록 유리)
기술	Tomasulo, ROB, 분기 예측, Out-of-Order 등 필수

8.4 VLIW(Very Long Instruction Word)

> 병렬성은 컴파일러가 책임진다

🧠 VLIW란?

VLIW(Very Long Instruction Word) 아키텍처는
여러 개의 독립적인 명령어를 하나의 긴 인스트럭션 단위로 묶어서 실행하는 구조이다.

> 슈퍼스칼라가 **하드웨어가 동적으로 명령 병렬성을 분석**한다면,
> **VLIW는 컴파일러가 정적으로 병렬성을 분석해서 미리 묶어줌.**

🔍 핵심 개념

- **"하드웨어는 단순하게, 병렬성은 컴파일러가 책임지자"**는 설계 철학
- 명령어들이 **고정된 포맷으로 병렬 슬롯에 할당됨**
- 하나의 VLIW는 **여러 개의 서브명령어로 구성**되어 한 번에 발행됨

```
| ALU instr | Load instr | FPU instr | Branch instr | ← VLIW 1개
```

🧩 구조 예시

예: 4-슬롯 VLIW 명령어

```
[ADD R1, R2, R3]  → ALU  슬롯
[LOAD R4, 0(R5)]  → Load 슬롯
[MUL F1, F2, F3]  → FPU  슬롯
[BEQ R1, R0, L1]  → Branch 슬롯
```

→ 이 네 개의 명령이 **동시에 실행**됨.
→ 단, **의존성 없어야 함** → 컴파일러가 보장해야 함

🧱 VLIW의 명령 실행 흐름

1. 컴파일 타임에 명령어 간의 병렬성 분석
2. 병렬 실행 가능한 명령어들을 하나의 VLIW로 구성
3. VLIW를 CPU가 **슬롯 단위로 한꺼번에 실행**
4. 결과를 레지스터에 저장

> 하드웨어는 단순한 파이프라인만 있고, **스케줄링 로직 없음**

🔄 VLIW vs 슈퍼스칼라 비교

항목	VLIW	Superscalar
병렬 분석 주체	컴파일러 (정적)	하드웨어 (동적)
제어 논리	단순	복잡 (동적 디스패치, 포워딩 등)

항목	VLIW	Superscalar
코드 크기	큼 (빈 슬롯 포함)	작음
유연성	낮음 (정확한 스케줄 필수)	높음 (분기, 예외 등 대응 유연)
성능	코드가 잘 최적화되면 매우 높음	일반적 상황에 강함
설계 난이도	낮음	높음

📦 장점

장점	설명
하드웨어 단순화	복잡한 동적 스케줄링/포워딩 회로 불필요
고속 실행 가능	잘 최적화된 코드는 Superscalar 이상
전력 효율성	하드웨어 제어 로직이 간단하여 전력 소비 ↓

⚠️ 단점

단점	설명
명령어 패딩 문제	병렬 슬롯을 채우지 못하면 **빈 슬롯 낭비** → 코드 크기 증가
분기/예외 처리 어려움	컴파일러가 실행 흐름 예측 실패 시 낭비 ↑
이식성 낮음	특정 VLIW 폭에 맞춰 컴파일된 코드는 다른 구조와 호환 안 됨
최적화 난이도	병렬성 스케줄링이 컴파일러에 종속됨

🧠 VLIW 구조를 쓰는 대표 예시

아키텍처	설명
Intel Itanium (IA-64)	3-슬롯 VLIW, EPIC(Explicitly Parallel Instruction Computing) 기반
TI C6000 DSP	TI사의 디지털 신호처리기, 고속 FIR/IIR 필터에 최적
HP PA-WideWord	초기 VLIW 연구 프로토타입
LLVM/Clang Backend	VLIW용 코드 생성 백엔드 실험 중 (옵션 기반)

🧪 실전 예: 3-슬롯 Itanium 명령어 구조

```
VLIW:
 [slot 0] ALU 명령
 [slot 1] 메모리 명령
 [slot 2] 분기 명령
```

- 명령 간 종속성 없음 → 병렬 실행 가능
- 컴파일러가 예외 처리, 분기 처리까지 고려해서 정렬해야 함

📌 요약 정리

항목	설명
VLIW	컴파일러가 병렬 명령어를 미리 묶어서 CPU에 공급하는 구조
특징	단순한 하드웨어, 정적 스케줄링, 고정 슬롯
장점	하드웨어 간결, 전력 소모 적음, 고속 실행 가능
단점	코드 낭비, 컴파일러 부담, 이식성 낮음
비교	슈퍼스칼라는 하드웨어가, VLIW는 컴파일러가 병렬성을 관리

8.5 아웃 오브 오더 실행

> 실행 순서를 바꿔서 성능을 높이다

🧠 Out-of-Order란?

Out-of-Order Execution은 CPU가 **명령어를 프로그램 순서대로 발행(fetch)하되**,
실행 가능한 명령부터 **순서에 상관없이 먼저 실행**하는 방식이야.

> 파이프라인이 자주 멈추는 이유? → 의존성, 분기, 메모리 지연
> → **의존성이 없는 명령어는 먼저 실행해서 CPU 놀리지 말자!**

📜 핵심 원칙

단계	설명
명령어 디코딩은 순서대로	
실행 순서는 유연하게 (병렬성 최대 활용)	
커밋/저장 순서는 반드시 프로그램 순서 보장	

🔁 In-order vs Out-of-Order 실행 흐름

🔹 In-Order

```
ADD → MUL → LOAD → STORE
```

- 순차 실행
- 하나라도 대기 상태면 CPU 정지 (스톨 발생)

🔹 Out-of-Order

```
ADD → (MUL 지연) → LOAD 먼저 실행 → (MUL 완료) → STORE
```

- 병렬로 실행 가능한 명령 먼저 처리
- CPU 활용도 극대화

🧱 핵심 구성 요소

✅ 1. Instruction Queue (Issue Queue)

- 명령어 대기 공간
- 레지스터, 메모리 의존성 분석 후 발행 여부 결정

✅ 2. Reservation Station (RS)

- 실행 유닛 앞에 위치
- 명령어가 필요한 피연산자를 받을 때까지 대기
- 준비되면 즉시 실행 유닛에 투입

✅ 3. Register Renaming (레지스터 이름 재정의) ⭐

- 가상의 **물리 레지스터**를 동적으로 할당
- **WAW(Writes After Writes), WAR(Write After Read)** 해저드를 제거하기 위해 사용

✅ 4. Reorder Buffer (ROB) ⭐

- 실행 순서와 상관없이 처리된 명령어들을
  정해진 순서대로 커밋(commit) 하기 위한 큐
- 예외 처리, 분기 실패 등에도 프로그램 순서 복원 가능

✅ 5. Common Data Bus (CDB)

- 실행 유닛에서 계산 완료된 결과를
  모든 Reservation Station, ROB에 전파하는 버스

📊 동작 단계 요약

단계	설명
1 **Fetch**	명령어를 순서대로 가져옴
2 **Decode & Rename**	가상 레지스터 할당
3 **Issue**	실행 가능한 명령어를 Reservation Station에 투입
4 **Execute**	실행 유닛에서 연산 수행
5 **Write-back**	결과를 CDB를 통해 공유
6 **Commit**	ROB에서 프로그램 순서대로 결과 확정 및 저장

🔧 Tomasulo 알고리즘

Tomasulo 알고리즘은 Out-of-Order 구조의 대표적 구현 방식으로:

요소	역할
Reservation Station	명령 대기 및 피연산자 준비
Register Tagging	레지스터에 '누가 결과를 줄 건지' 표시
CDB	결과를 전파하고 의존성 해소

특징

- RAW 해저드 해결 (결과를 기다림)
- WAW/WAR 해저드 제거 (레지스터 리네이밍 기반)
- 실행 후 프로그램 순서대로 커밋 (ROB 필요)

🧪 실전 예시

```
1: LOAD R1 ← 0(R2)
2: ADD  R3 ← R1 + R4
3: MUL  R5 ← R6 × R7
```

In-Order

- 2번 명령은 1번의 완료까지 기다림
- 3번도 대기 상태

Out-of-Order

- 1번: 메모리 지연 중
- 3번: R6, R7 준비됨 → 먼저 실행 가능
- CPU가 놀지 않음

📉 성능 향상 효과

기술	이점
Out-of-Order 실행	실행 유닛 유휴 시간 최소화
레지스터 리네이밍	가짜 의존성 해소
동적 스케줄링	분기, 지연 없이 지속 실행 가능
커밋 단계 분리	프로그램의 순서 정합성 유지

📌 요약 정리

항목	설명
Out-of-Order Execution	실행 순서를 바꿔서 명령어 병렬 실행
핵심 장치	RS, ROB, 레지스터 리네이밍, CDB
장점	성능 향상, 병렬 처리 ↑, 스톨 최소화
단점	제어 논리 복잡, 전력 소모 증가
대표 구현	Tomasulo 알고리즘 (IBM 360/91), 현대 x86/ARM CPU 대부분 적용

8.6 스레드 수준 병렬성 (SMT, Hyper-Threading)

> SMT, Hyper-Threading으로 CPU 활용도를 극대화하다

🧠 스레드 수준 병렬성이란?

TLP(Thread-Level Parallelism)은
하나의 CPU 코어에서 **여러 개의 소프트웨어 스레드를 동시에 실행**하는 기술이다.

> 파이프라인, Out-of-Order, Superscalar는 **한 스레드 내 병렬성(ILP)**을 높이는 것
> → TLP는 **여러 스레드 간 병렬성**을 활용해서 CPU 활용도를 높이는 전략

🔁 개념 비교

종류	설명	병렬 단위
ILP (Instruction-Level Parallelism)	한 스레드 내에서 명령어를 병렬 처리	명령어
TLP (Thread-Level Parallelism)	여러 스레드를 동시에 실행	스레드
DLP (Data-Level Parallelism)	같은 명령을 여러 데이터에 병렬 적용	데이터

🔹 SMT (Simultaneous Multithreading)

✅ 정의

하나의 Superscalar CPU에서 **여러 스레드의 명령어를 동시에 발행하고 실행**하는 기술

> → 다중 명령어 발행 + 다중 스레드 발행
> → 실행 유닛이 놀고 있는 시간을 다른 스레드가 차지하도록 구성

🧱 SMT 구조 예시

```
          +------------------------+
          | Superscalar Pipeline   |
          | (Issue Width: 4)       |
          +------------------------+
              ↑ ↑ ↑ ↑
          [스레드 A] [스레드 B]
```

- 명령어 발행 단계에서 **여러 스레드의 명령어를 섞어서 발행**
- 실행 유닛은 **스레드 구분 없이 사용**

🔧 Hyper-Threading (Intel SMT 기술)

항목	설명
도입 시기	Intel Pentium 4 (NetBurst 아키텍처)부터
핵심 특징	1개의 물리 코어 → 2개의 논리 코어로 OS에 표시
공유 자원	ALU, FPU, L1/L2 캐시, TLB 등은 공유
분리 자원	레지스터 파일, 명령 큐, 스레드 컨텍스트는 분리

💡 예시: 1코어, 2스레드 Hyper-Threading

자원	구성
명령어 디코더	공유
파이프라인	공유
레지스터 셋	분리 (Thread A, Thread B 따로)
캐시	공유 (L1~L3)
ROB, RAT 등	공유 또는 파티셔닝

> 즉, **논리적으로는 2코어처럼 보이지만,**
> **물리적으로는 1코어를 효율적으로 사용하는 기술**

📈 왜 SMT/Hyper-Threading이 성능을 올릴까?

이유	설명
스톨 회피	한 스레드가 메모리 접근으로 멈췄을 때 다른 스레드 실행
유휴 유닛 활용	ALU, FPU 등 사용률이 낮을 때 다른 스레드 명령어로 채움
컨텍스트 스위칭 오버헤드 없음	스레드 전환 시 OS 개입 필요 없음

⚠️ SMT의 한계

단점	설명
자원 경합	L1 캐시, TLB 등의 공유로 오히려 성능 저하 가능
전력 소비 증가	논리 코어가 증가해 전력 소모 증가 가능
보안 이슈	Spectre, Meltdown 등의 측면채널 공격 경로 제공
성능 향상 폭 제한	CPU가 이미 바쁠 경우 SMT 효과 없음

📊 SMT vs 멀티코어 vs Superscalar 비교

항목	SMT	멀티코어	Superscalar
병렬성 수준	스레드	스레드	명령어
하드웨어 비용	낮음	높음	중간
캐시 공유	많음	코어 단위로 분리 가능	코어 내부
성능 확장성	제한적	좋음	한정적
예시	Intel Hyper-Threading	AMD Ryzen, Apple M1	대부분의 현대 CPU

📌 요약 정리

항목	설명
TLP	여러 스레드를 동시에 실행하여 CPU 자원 활용도 향상
SMT	Superscalar CPU가 여러 스레드 명령어를 동시 실행
Hyper-Threading	Intel의 대표 SMT 기술, 1코어 → 2 논리코어
장점	스톨 회피, 유휴 유닛 활용, 성능 향상
단점	자원 경합, 보안 이슈, 성능 향상 한계 존재

8.7 멀티코어 및 NUMA 구조

> CPU 다중화와 메모리 계층화의 병행 설계

🧠 멀티코어란?

멀티코어(Multicore)는 하나의 CPU 칩에 **두 개 이상의 독립된 실행 유닛(코어)**을 포함하는 구조이다.

> 각 코어는 독립된 파이프라인, 레지스터 파일, L1 캐시를 갖고 있으며
> **L2/L3 캐시 및 메모리 버스는 공유하거나 분리할 수 있다.**

🧱 멀티코어 기본 구조 예시

```
+-------------------------------------------------+
|          SoC / CPU Die (하나의 패키지)             |
|  +--------+  +--------+  +--------+  +--------+  |
|  | Core 0 |  | Core 1 |  | Core 2 |  | Core 3 |  |
|  +--------+  +--------+  +--------+  +--------+  |
|          ↘      ↙        ↘      ↙               |
|           +---------------------------+          |
|           |  Shared L3 Cache (or NoC) |          |
|           +-------------+-------------+          |
|                         ↓                        |
|               DRAM Memory Controller             |
+-------------------------------------------------+
```

🔁 SMP (Symmetric Multiprocessing) 구조

✅ 특징

- 모든 코어가 **공통된 메모리**에 접근 가능
- 메모리 접근 속도 **균일(Uniform)**
- OS 입장에서는 **코어 구분 없이 동일한 자원처럼 취급**

> 예: 듀얼코어, 쿼드코어 대부분이 SMP 구조로 작동

🔥 NUMA (Non-Uniform Memory Access) 구조

✅ 정의

NUMA는 여러 CPU 노드가 있고, 각 노드에 **자기 메모리 뱅크가 붙어있는 구조**이다.
→ **로컬 메모리는 빠르고**, 다른 노드의 메모리는 **느리게 접근**됨

✅ 구조 예시

```
[ CPU 0 ]---[ MEM 0 ]
     |
     └──────────────┐
                    ▼ (Interconnect)
[ CPU 1 ]---[ MEM 1 ]
```

> 각 CPU는 자신의 메모리(MEM 0, MEM 1)에 빠르게 접근
> 다른 CPU의 메모리 접근은 **원격 접근 (Remote Access)**
> → **메모리 접근 속도 ≠ 일정하지 않음**

📊 NUMA vs UMA (SMP) 비교

항목	UMA (SMP)	NUMA
메모리 접근 속도	동일	노드마다 다름 (로컬 vs 원격)

항목	UMA (SMP)	NUMA
구조	단일 메모리 공유	CPU마다 메모리 영역 보유
확장성	제한적	우수 (서버급 수십 코어 이상)
성능 최적화	신경 쓸 것 적음	**NUMA-aware 배치 필수**
예시	일반 PC, 저가형 서버	고성능 서버, 워크스테이션

📍 NUMA에서 중요해지는 개념

개념	설명
NUMA 노드(Node)	각 CPU와 그에 연결된 메모리 영역
로컬 메모리	현재 CPU에 직접 연결된 메모리
원격 메모리	다른 NUMA 노드에 있는 메모리
NUMA-aware OS 스케줄링	프로세스가 자신과 가까운 메모리만 사용하도록 배치

🧠 성능 영향

✅ NUMA 미지원 시스템의 문제

- 스레드 A가 CPU 0에서 실행되는데
  메모리는 CPU 1의 MEM 1에 할당되면
  → **필요할 때마다 원격 접근 → 지연 발생**

✅ 해결 방법

- **NUMA-aware 메모리 배치** (`numactl`, `mbind()`, `taskset`)
- **NUMA-aware 스레드 스케줄링** (CPU affinity 설정)
- **메모리 locality 최적화 알고리즘 도입**

🧪 실전 명령어 예시 (Linux 기준)

```
lscpu      # NUMA 노드 정보 확인
numactl --hardware   # 노드별 메모리 확인
numactl --cpunodebind=0 --membind=0 ./myapp
```

→ 해당 앱을 NUMA 노드 0의 CPU + 메모리에서 실행

📌 요약 정리

항목	설명
멀티코어	하나의 CPU 패키지에 여러 실행 유닛
SMP (UMA)	메모리 접근 속도 균일, 구조 단순

항목	설명
NUMA	노드 간 메모리 접근 속도 차이 존재
NUMA 최적화 필요	스레드, 메모리 배치에 주의해야 성능 발휘
현대 서버 구조	대부분 NUMA 기반 (인텔 Xeon, AMD EPYC 등)