파이프라인
(Pipeline)

05 파이프 라이닝

❑ 명령어 단계 병렬 처리

- 프로세서의 제어 장치는 기본적으로 '명령어 인출 → 명령어 해독 → 명령어 실행' 순서로 명령 수행
- 전통적인 프로세서에서는 다음과 같이 순차적으로 실행

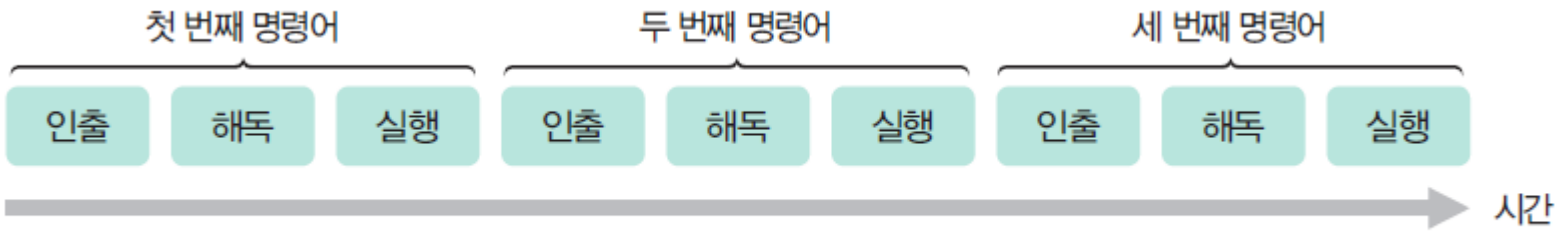


그림 5-22 전통적 PC에서 명령어 실행(순차적 실행)

컴퓨터 설계자 : 처리속도의 향상을 목표로 노력, 클럭 속도를 높여 chip을 빠르게 동작시키는 것은 기술적 한계가 있음, 따라서 병렬화(작업을 한 번에 두개 이상 수행)를 고려

병렬화는 명령어 단계 병렬처리(pipelining)와 프로세서 단계 병렬처리(multiprocessor)가 있음

05 파이프 라이닝

❑ 명령어 단계 병렬 처리

- 현대 대부분의 프로세서에서는 파이프 라이닝(pipelining) 기술로 명령 실행
- 파이프 라이닝은 그림과 같이 명령 하나를 여러 단계로 나누어 각각을 독립적인 장치에서 동시에 실행하는 기술
- 하나의 명령을 3단계로 나누어 실행 하는 예

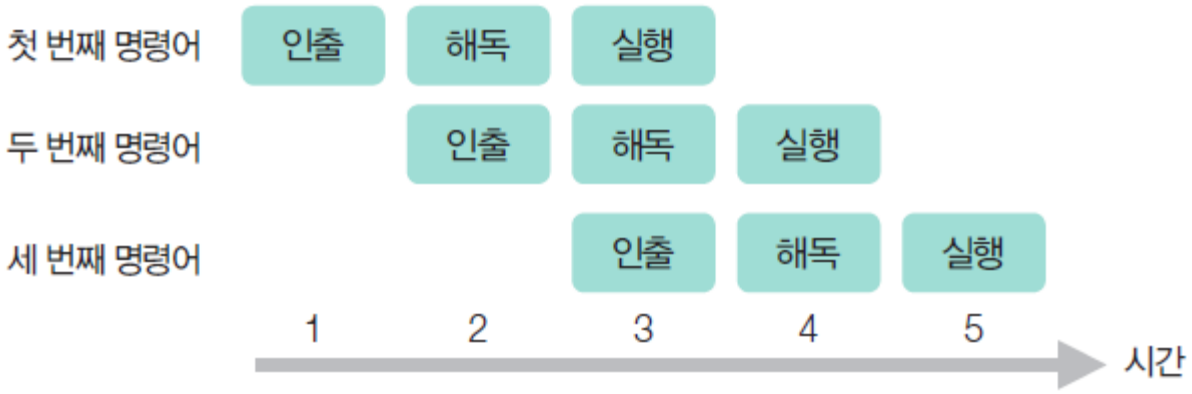
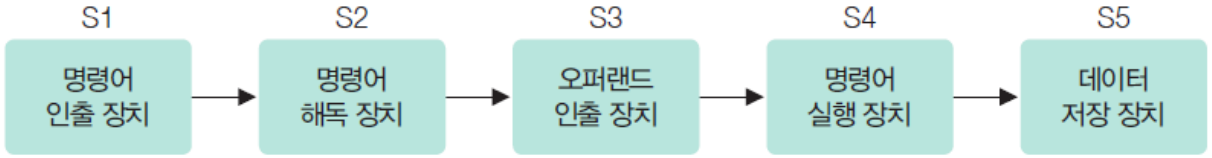


그림 5-23 파이프 라인닝 개념

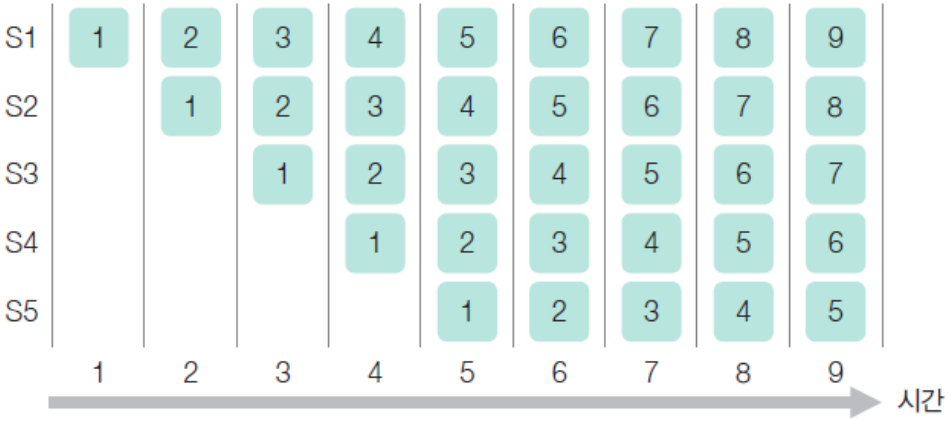
❑ 5단계 파이프라인

- 단계가 S1~S5로, 5단계 파이프 라인
- 1단계 : 메모리에서 명령어를 인출
- 2단계: 명령어를 해독하고 명령어 형태를 결정하며 필요한 피연산자 결정
- 3단계: 레지스터 또는 메모리에서 피연산자 결정
- 4단계: 명령어 연산 수행
- 5단계: 결과를 레지스터에 저장

- 5개의 장치가 서로 독립적으로 작동하고 각각의 명령이 순서대로 각 장치를 이동하며 실행된다면 실행 시간은 훨씬 단축



(a) 5단계 파이프 라인



(b) 시간에 따른 각 단계별 상태

그림 5-24 5단계 파이프 라인에서 프로그램 실행

❑ 5단계 파이프라인 (계속)

- 예: 각 단계가 2ns 소요
  - 전통적인 시스템 : 명령 하나가 완전히 실행되는 데는 10ns 소요
  - 파이프라인 시스템 : 매 클록 사이클(2ns)마다 명령 5개가 동시에 실행되므로 시간은 1/5로 단축
  - 파이프 라인이 없는 컴퓨터에서는 100MIPS
  - 5단계 파이프 라인을 가진 컴퓨터는 500MIPS의 처리 속도
- 파이프라이닝을 사용하면 지연 시간(명령어를 실행하는 데 걸리는 시간)과 프로세서 대역폭(CPU의 MIPS 수)간 균형 유지
  - 한 사이클에 소요되는 시간 : Tns
  - 파이프 라인 : n단계
  - 전체 실행 시간 : nTns
- 클록 사이클이 초당 10^9/T라면 초당 실행되는 명령의 개수는 $10^9/T$개
  - 예를 들어 T=2ns인 경우 매초 5억 건의 명령이 실행
  - MIPS(Million Instructions Per Second) : 초당 실행되는 명령 개수를 100만으로 나눔

    $(10^9/T)/10^6=1000/T$ MIPS
  - 현재는 MIPS 대신 GIPS(Giga(Billion) Instructions Per Second, BIPS)를 쓰는 것이 더 타당

(보충) 파이프라인에서 속도 증가율

- **파이프라인에서 속도증가율 : 비파이프라인 장치일 때, n개의 task에 대한 전체 수행 시간에서 k 세크먼트 파이프라인에서 n task를 완료하는데 필요한 시간을 나눈 것**
- **속도 증가율 Speed_up = $nt_n / (k+n-1)t_p$**
- **n이 충분히 커지면 Speed_up = t_n / t_p**
- **비 파이프라인에서 task 하나를 수행하는데 걸리는 시간과 파이프라인에서 task 하나를 수행하는데 걸리는 시간이 같다고 가정하면 $t_n=kt_p$에서 Speed_up = tn / tp=k t_p / t_p = k 따라서 세그먼트 수 만큼 속도가 증가한다.**

tp:클럭사이클
Task T1수행시간:k x tp
나머지 n-1개의 task 수행시간 : (n-1) x tp
비파이프라인 수행시간: n x tn

펜티엄 프로세서 파이프라인 수

- 클래식 펜티엄: 5개
- 펜티엄 프로 / 펜티엄 II: 12개
- 펜티엄 III: 10개
- 펜티엄 4 (윌라멧, 노스우드): 20개
- 펜티엄 4 (프레스캇, 시더밀): 31개
- 펜티엄 M / 코어 / 코어2 / 코어 i 시리즈: 14개

1 데이터 해저드(data hazards)

- 데이터 의존성(data dependency) : 파이프 라인에서 앞서가는 명령의 ALU 연산 결과를 레지스터에 기록하기 전에 다른 명령에 이 데이터가 필요한 상황
- 앞의 명령 결과가 다음 명령 입력으로 사용될 때 파이프 라인 시스템에서 문제 발생
- 해결 방법

  - 레지스터에 저장되기 전에 ALU 결과를 직접 다음 명령에 직접 전달하는 데이터 포워딩
  - 또는 버블을 명령 사이에 끼워 넣어 프로그램 실행을 1단계 또는 2단계 지연

- WAW(Write After Write)와 WAR(Write After Read) 해저드

  - 레지스터 재명명함(register renaming) : 관련 없는 레지스터로 바꾸어 사용함으로써 쉽게 해결가능

- 따라서 참 의존(True dependency)는 RAW뿐임

파이프라인을 방해하는 요소
1. 데이터 해저드
2. 제어 해저드
3. 구조적 해저드

05 파이프 라이닝

- 예(RAW: read after write): 첫 번째 연산 ADD의 결과(r3)가 두 번째 연산 SUB의 입력으로 사용
  - 1번 명령이 S5단계에서 레지스터에 저장되는데
  - 2번 명령은 S3단계에서 데이터 요구

```
1. ADD    r3, r2, r1    /* r3 = r2 + r1 */
2. SUB    r4, r3, r5    /* r4 = r3 - r5 (HAZARD: r3이 아직 저장되지 않음!) */
```

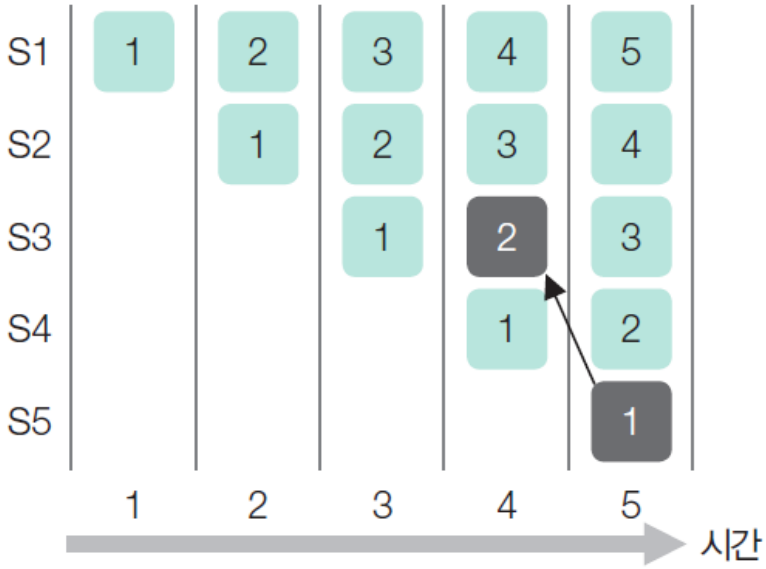


그림 5-25 데이터 의존성(종속)으로 인한 데이터 해저드

1번의 결과를 write후에 2번의 읽기가 진행되어야 하는데 2번이 먼저 요구함
즉, write후 읽어야 하는데 순서가 바뀜

WAR(읽기 후 쓰기 write after read), WAW(write after write)

WAR(읽기 후 쓰기)

1. R2 ← R1+R7
2. R7 ← R1+R3

1번 수행 후 2번이 진행되는 경우와
2번이 수행되고 1번이 진행되는 경우
결과가 달라짐(읽기 후 쓰기가 진행
되어야 하는데 그렇지 않은 경우)

WAW(쓰기 후 쓰기)

1. R2 ← R1+R7
2. R2 ← R1+R3

1번 수행 후 2번이 진행되는 경우와
2번이 수행되고 1번이 진행되는 경우
결과가 달라짐(1번 쓰기 후 2번 쓰기
가 진행되어야 하는데 그렇지 않은
경우)

2 제어 해저드(control hazards)

❖ 파이프 라인 CPU 구조의 분기 명령이 실행될 때 발생

- 이미 파이프 라인에 적재되어 실행되고 있는 이어지는 다른 명령들이 더 이상 필요가 없어지므로 발생
- [그림 5-26]과 같이 3번 명령에서 15번 명령으로 분기가 일어난다면 이미 파이프 라인 단계에 들어와 실행되고 있는 4, 5, 6, 7번 명령은 더 이상 필요가 없으므로 전체 프로그램의 속도 저하 요인이 됨

❖ 분기로 인한 제어 해저드

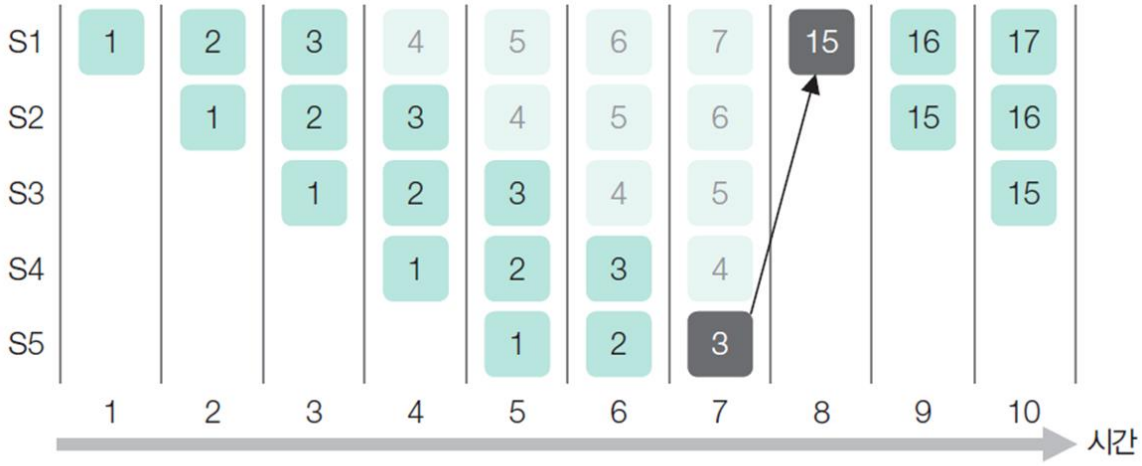


그림 5-26 분기로 인한 제어 해저드

2 제어 해저드(control hazards)

❖ 제어 해저드 해결 방법

- 지연 슬롯(delay slot)을 넣고 분기 목적지 주소를 계산하는 과정을 파이프 라인 속에 넣는 것
- 지연 슬롯이란 NOP나 분기 명령과 무관한 명령을 끼워 넣는 것
- 이 방법은 컴파일러나 프로그래머가 프로그램 순서를 바꾸는 것
- 또는 분기 예측 알고리즘을 이용하기도 함

3 구조적 해저드(structural hazards)

- 서로 다른 단계에서 동시에 실행되는 명령이 컴퓨터 내의 장치 하나를 동시에 사용하려고 할 때 발생
- 예1 : 명령어 인출과 오퍼랜드 인출이 동시에 발생하는 경우
  - 명령 2개가 동시에 메모리에 가서 명령과 데이터를 가져와야 하는데,
  - 인출 과정이 모두 같은 장치들(bus, memory 등)을 사용하므로 충돌 발생
- 예2 : CPU 내의 장치인 ALU를 명령 2개가 동시에 사용해야 하는 경우

❖ 인출 과정에서 메모리 충돌을 해결 방법

- 하버드 구조(harvard architecture)를 사용
  - 메모리를 프로그램(명령어) 메모리와 데이터 메모리로 완전 분리시킨 구조
  - 명령어 인출과 데이터 인출 과정에서 충돌을 원천적으로 봉쇄
  - RISC 프로세서에서 많이 사용하는 구조
- 분리 캐시를 사용하여 충돌 회피
  - 인텔 계열 프로세서의 L1 캐시에서 사용하는 구조로, L1 명령어 캐시와 L1 데이터 캐시로 분리
  - 그러나 명령어 2개가 동시에 캐시 미스가 발생하면 충돌이 발생할 수 있음
- 동일한 장치를 동시 사용하여 일어나는 충돌 : 장치를 하나 더 둠
  - 예를 들어 CPU 내에 ALU를 2개 둠

4 슈퍼 스칼라

❖ 하나의 프로세서 안에 2개 이상의 파이프 라인 탑재

- 하나의 명령어 인출 장치가 명령어 쌍을 동시에 가져와서 각 명령어를 다른 파이프 라인에 배치하고 개별 ALU를 가지고 병렬 작업
- 명령 2개는 자원(예를 들어 레지스터)을 사용할 때 충돌하지 않아야 함
- 둘 중 어느 명령도 다른 명령의 결과에 의존하지 않아야 함
- 하드웨어를 추가하여 실행 도중에 충돌을 감지 및 제거

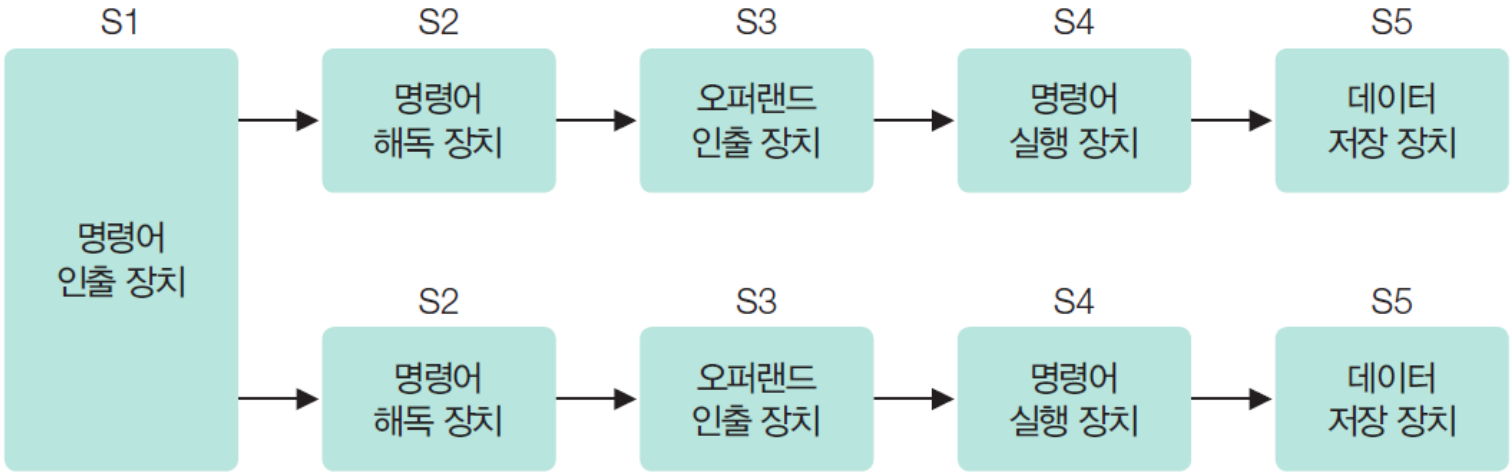


그림 5-27 슈퍼 스칼라에서 명령어 실행

05 파이프 라이닝

❖ **단일 또는 2중 파이프 라인 : RISC 머신이 원조**

❖ **486부터는 인텔이 데이터 파이프 라인 도입**

- 486은 파이프 라인을 하나
- 펜티엄은 [그림 5-27]과 비슷한 5단계 파이프 라인을 2개 가지고 있음

  - U 파이프 라인(메인 파이프 라인)은 임의의 펜티엄 명령을 실행
  - V 파이프 라인이라고 하는 두 번째 파이프 라인은 단순한 정수 명령어나 간단한 부동 소수점 명령어 하나만 실행

❖ **실행 규칙**

- 명령어 한 쌍이 서로 호환되어 병렬로 실행될 수 있는지 여부를 결정
- 명령어 한 쌍이 충분히 단순하지 않거나 호환되지 않는 경우, 첫 번째 명령만 실행
- 두 번째 명령은 멈추어 있다가 다음 명령과 짝을 지어 실행
- 명령어는 항상 순서대로 수행: 따라서 호환 가능한 쌍을 생성한 펜티엄 전용 컴파일러는 구형 컴파일러보다 빠르게 실행되는 프로그램을 생성 - 측정 결과 최적화된 펜티엄 실행 코드는 동일한 클록 속도로 실행되는 486보다 정수 프로그램에서 정확히 2배 빠름 – 속도 향상은 두 번째 파이프 라인에 의한 것

05 파이프 라이닝

- 파이프 라인을 4개 도입하는 것도 가능하지만, 너무 많은 하드웨어가 소요
- 대신 고급 CPU에서는 다른 접근 방식 사용
  - 기본 개념은 파이프 라인은 하나지만 기능 장치를 여러 개 제공
  - 인텔 코어 아키텍처는 아래 그림과 유사한 구조

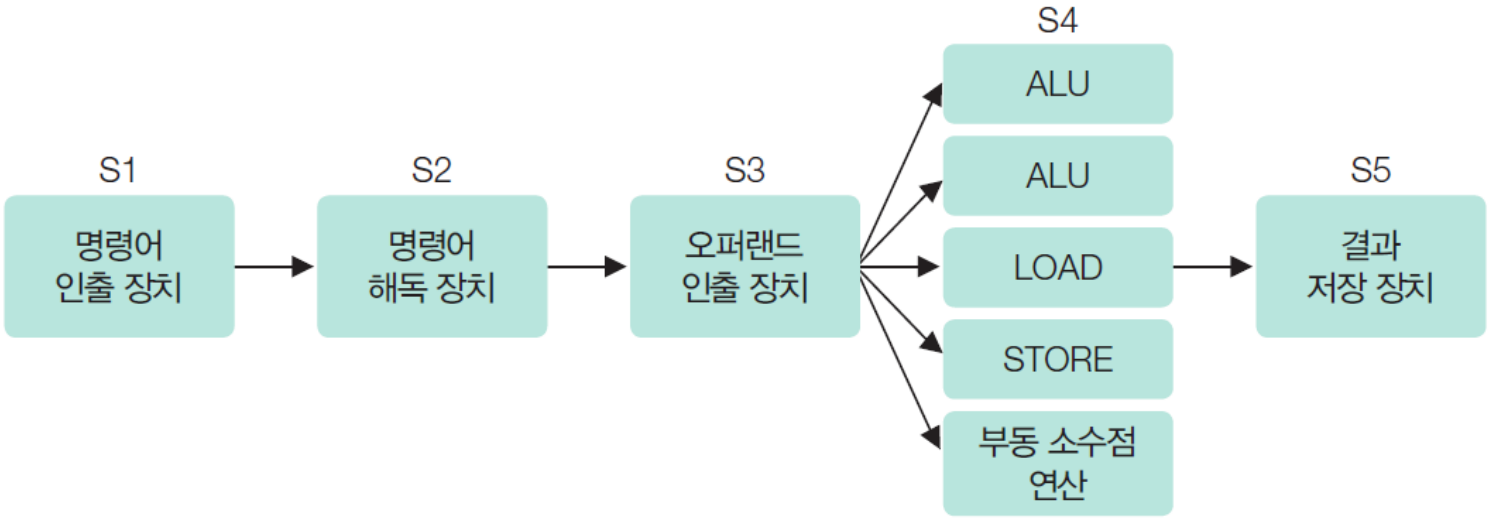


그림 5-28 기능 장치를 여러 개 가진 슈퍼 스칼라

05 파이프 라이닝

- 슈퍼 스칼라 프로세서라는 아이디어: S3이 S4 단계보다 훨씬 빠르게 명령을 실행 가능
- S3 단계에서 매 10ns마다 명령을 수행하고 모든 기능 유닛에서 10ns에 작업을 수행할 수 있다면, 작업을 한번에 하나 이상 수행할 수 없으므로 의미 없음
- S4 단계가 시간이 더 오래 걸린다면 의미 있음
  - 실제로 S4 단계의 기능 장치인 LOAD 및 STORE의 메모리 액세스 및 부동 소수점 연산은 실행에 1 클록 사이클보다 오래 걸림
  - 그림에서 알 수 있는 바와 같이 S4 단계에서 다수의 ALU를 가짐

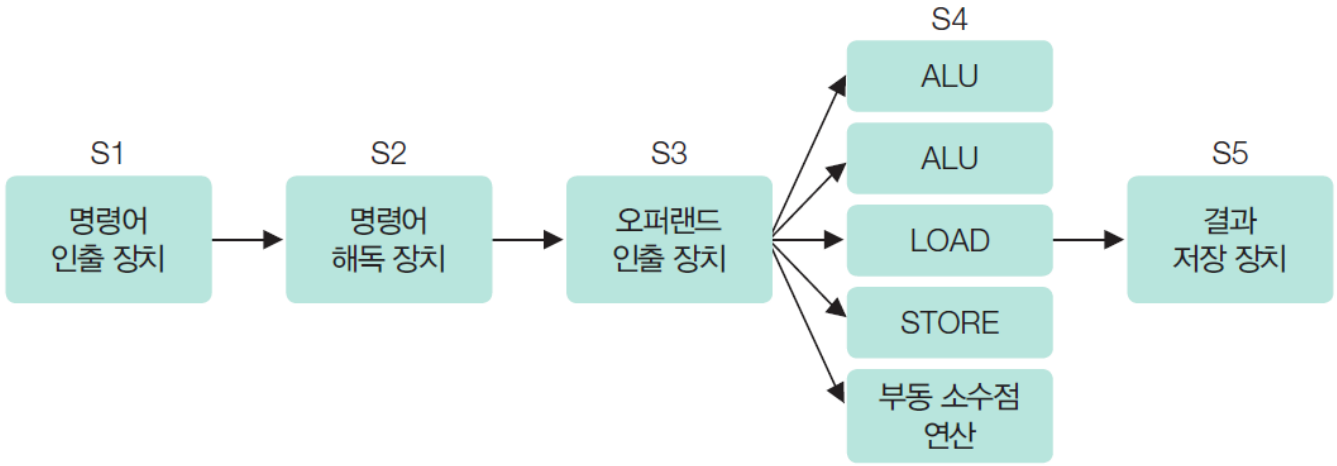


그림 5-28 기능 장치를 여러 개 가진 슈퍼 스칼라

Thank You