컴퓨터 구조

2장 CPU의 구조와 기능3

안형태

anten@kumoh.ac.kr

디지털관 139호

CPU의 구조와 기능

3. 명령어 파이프라이닝

명령어 파이프라이닝

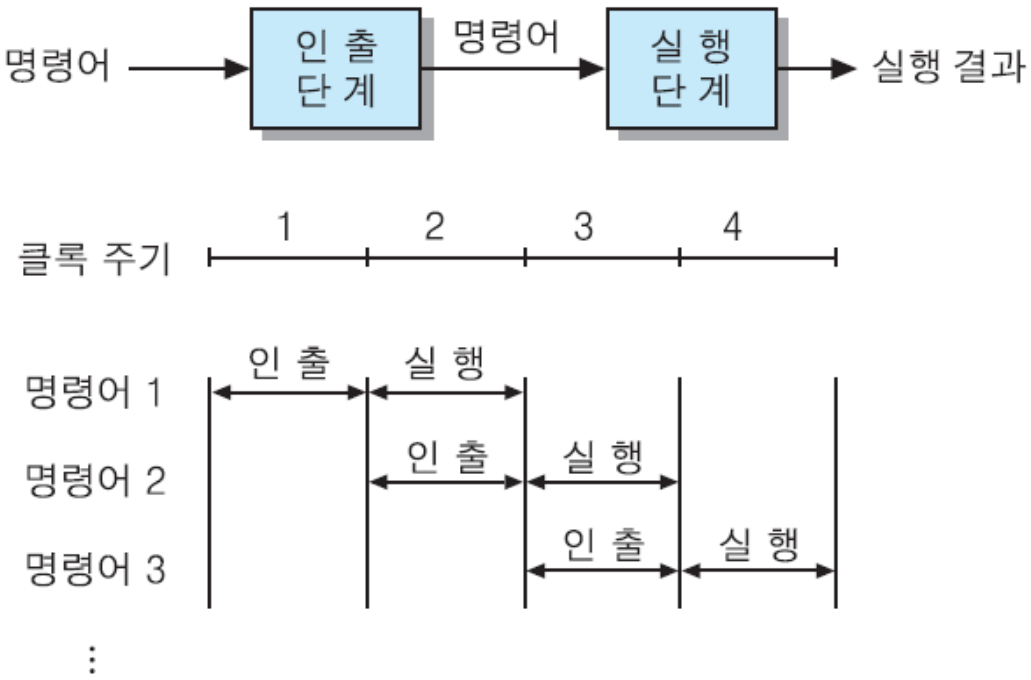
명령어 파이프라이닝(instruction pipelining)

- CPU의 프로그램 처리 속도를 높이기 위하여 CPU 내부 하드웨어를 여러 단계로 나누어 동시에 처리하는 기술

2-단계 명령어 파이프라인(two-stage instruction pipeline)

- 명령어를 실행하는 하드웨어를 **인출 단계(fetch stage)**와 **실행 단계(execute stage)**라는 두 개의 독립적인 파이프라인 모듈로 분리
- 두 단계들에 동일한 클럭으로 동작 시간을 일치시키면,(**[예]** 파이프라인의 1단계 = 1 클럭 주기)
  - 첫 번째 클럭 주기에서는 인출 단계가 첫 번째 명령어를 인출
  - 두 번째 클럭 주기에서는 인출된 첫 번째 명령어가 실행 단계로 보내져서 실행되며, 그와 동시에 인출 단계는 두 번째 명령어를 인출(**명령어 선인출**, instruction prefetch)

2-단계 명령어 파이프라인과 시간 흐름도



- **속도 향상(Speedup, S_p) = 6/4 = 1.5배**
  - 실행되는 명령어 수 증가 시, S_p는 2배에 접근

2-단계 명령어 파이프라인

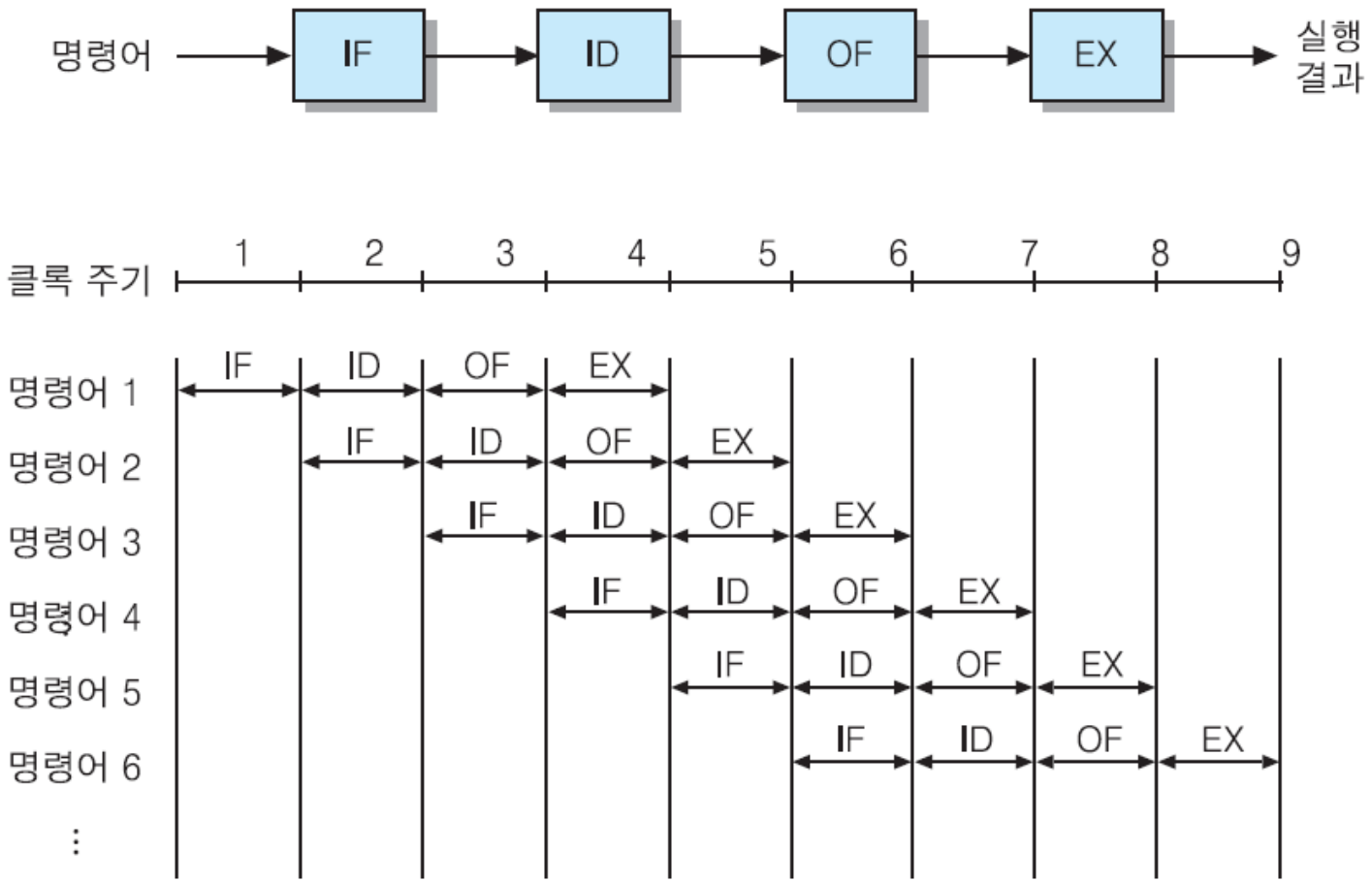
- **문제:** 두 단계의 처리 시간이 동일하지 않으면 파이프라인 속도 향상이 저하됨

- **해결 방안:** 파이프라인 단계를 세분하여, 각 단계의 처리 시간을(거의) 같아지도록 함
  - 파이프라인 단계의 수를 늘리면 전체적으로 속도 향상이 더 높아짐

4-단계 명령어 파이프라인

- **명령어 인출(IF) 단계:** 명령어를 기억장치로부터 인출하는 단계
- **명령어 해독(ID) 단계:** 제어 유니트에서 해독기(decoder)를 이용하여 명령어를 해독하는 단계
- **오퍼랜드 인출(OF) 단계:** 연산에 필요한 오퍼랜드(데이터 등)를 기억장치로부터 인출하는 단계
- **실행(EX) 단계:** 지정된 연산을 수행하고, 결과를 저장하는 단계

4-단계 명령어 파이프라인과 시간 흐름도



파이프라인에 의한 전체 명령어 실행 시간

- 파이프라인 단계 수 = k, 실행할 명령어들의 수 = N, 각 파이프라인 단계가 한 클럭 주기씩 걸린다고 가정
  - 파이프라인에 의한 전체 명령어 실행 시간(T_k):

$$T_k = k + (N - 1)$$

    - 즉, 첫 번째 명령어를 실행하는데 k 주기가 걸리고, 나머지($N-1$) 개의 명령어들은 각각 한 주기씩만 소요
  - 만약, 파이프라인을 사용하지 않을 경우, N 개의 명령어들을 실행 시간(T_1):

$$T_1 = k \times N$$

- 파이프라인에 의한 속도 향상(S_p):

$$S_p = \frac{T_1}{T_k} = \frac{k \times N}{k + (N - 1)}$$

파이프라인에 의한 속도 향상

- **[예제]** 파이프라인 단계 수 = 4, 파이프라인 클럭 = 1GHz(각 단계에서의 소요시간 = 1ns)일 때, 10개의 명령어를 실행하는 경우의 속도 향상은?

- **[풀이]** 첫 번째 명령어 실행에 걸리는 시간 = 4ns, 다음부터는 1ns 마다 한 개씩의 명령어 실행 완료
  - 10개의 명령어 실행 시간 = 4 + (10 - 1) = 13ns
  - 속도 향상$(S_p) = \frac{10 \times 4}{13} \approx 3.08$배

파이프라인에 의한 속도 향상

- N: CPU 가 실행하는 명령어 수라면,
  - $N = 100$ 일 때, $S_p = \frac{400}{103} \approx 3.88$
  - $N = 1000$ 일 때, $S_p = \frac{4000}{1003} \approx 3.988$
  - $N = 10000$ 일 때, $S_p = \frac{40000}{10003} \approx 3.9988$
  - $N = \infty$ 일 때, $S_p \approx 4$(**이론적 속도 향상 = 단계 수**)

$$S_p = \lim_{n \to \infty} \frac{T_1}{T_k} = \lim_{n \to \infty} \frac{k \times N}{k + (N - 1)} = K$$

- K-단계 파이프라이닝 기법을 적용하였을 때, 이론적인 성능 향상은 실행 시간에 있어서, 단계 수와 동일한 K배임

파이프라인의 효율 저하 요인들

- 모든 명령어들이 파이프라인 단계들을 모두 거치지는 않음
  - 어떤 명령어는 오퍼랜드 인출이 필요 없지만, 파이프라인의 하드웨어를 단순화시키기 위해서는 모든 명령어가 네 단계들을 모두 통과해야 함

- 파이프라인의 클럭은 처리 시간이 가장 오래 걸리는 단계를 기준으로 결정
  - **[예]** IF, ID, OF의 각 단계에서 1ns 씩 소요되어도, EX 단계에서 1.5ns 소요되면, 각 단계의 처리 시간은 1.5ns으로 설정 됨
  - **해결 방안:** 슈퍼 파이프라이닝(super-pipelining): 명령어 파이프라인의 각 단계를 세분하여, 한 클럭 주기에 여러 단계를 수행하여 명령어를 처리하여 속도를 높이도록 설계된 기술

- **3개의 장애(hazards)**
  - 구조적, 데이터, 제어 등

파이프라인의 효율 저하 요인들

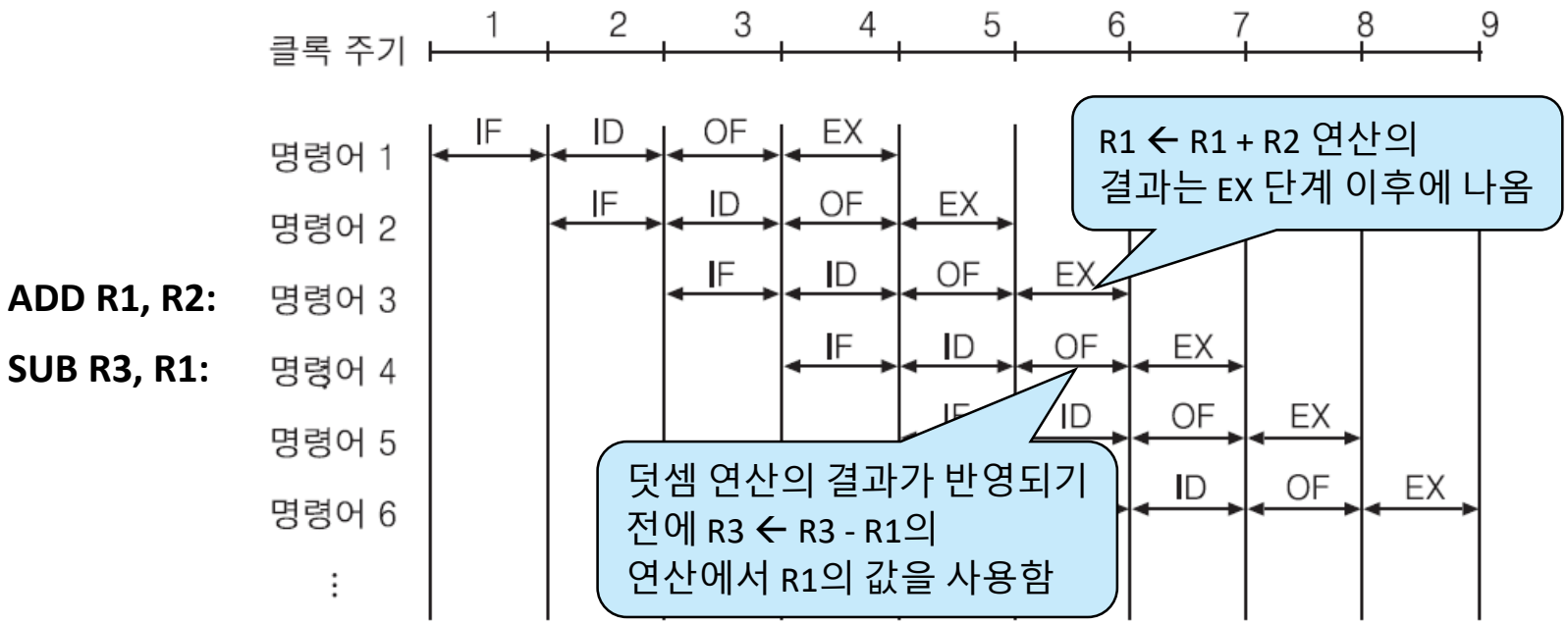
1. 구조적 장애(structural hazards)

- 서로 다른 단계에서 동시에 실행되는 명령이 컴퓨터 내의 장치 하나를 동시에 사용하려고 할 때 발생
- **[예]** IF 단계와 OF 단계가 동시에 기억장치를 액세스하는 경우에, **기억장치 충돌(memory conflict)**이 일어나면 지연이 발생
  - **해결 방안:** 메모리가 하나인 경우 발생 → 하버드 구조 사용 or 명령어 캐시(cache), 데이터 캐시 분리

2. 데이터 장애(data hazards)

- 한 명령어 수행 결과가 다른 명령어의 연산에 사용될 때, 다른 명령어의 파이프라인의 단계가 지연되는 경우 발생
- 명령어 처리 결과 사이에 **데이터 의존성(data dependency)**이 존재
  - **[예]** 파이프라인에서 앞서가는 명령의 ALU 연산 결과를 레지스터에 저장하기 전에 다른 명령에 이 데이터가 필요한 상황

파이프라인의 효율 저하 요인들



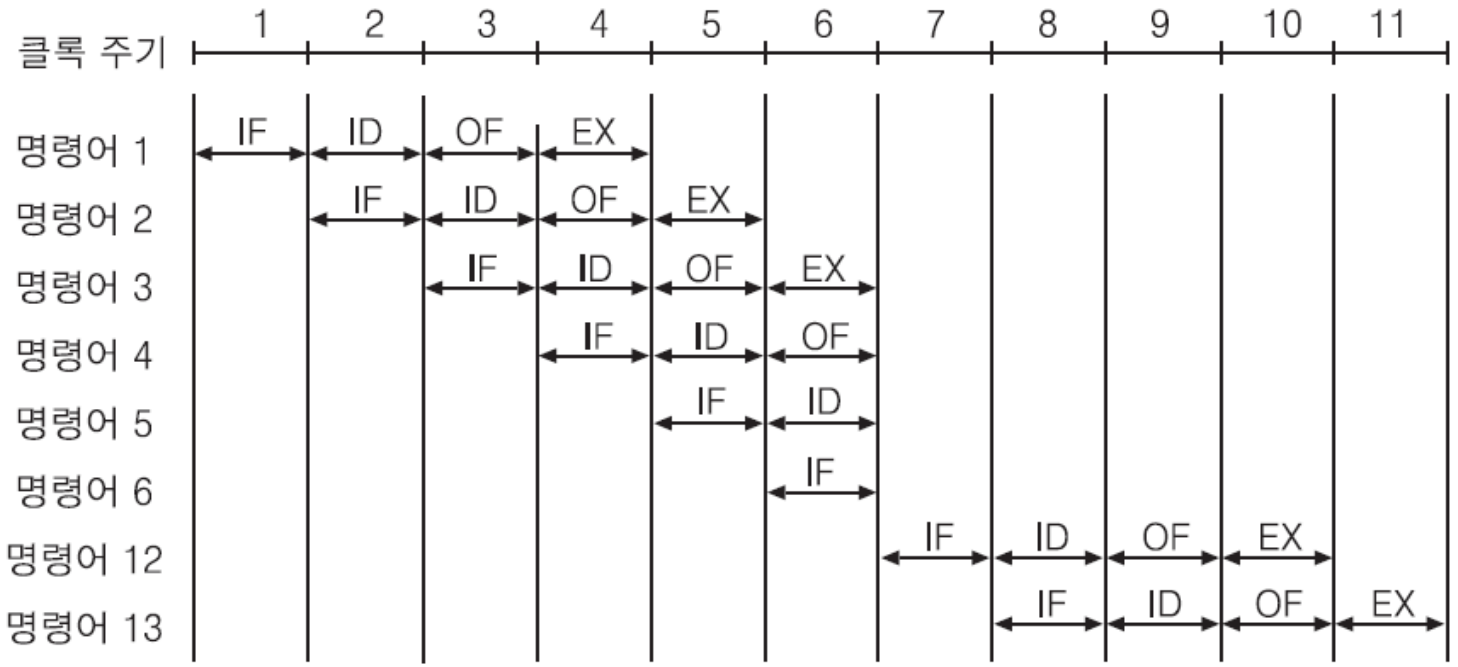
❑ 해결 방안

- ① 파이프라인 **지연 명령(stall)**을 사이에 끼워 넣어 프로그램 실행을 1단계 또는 2단계 지연
- ② **명령어를 재배치**하여 실행 순서를 변경
- ③ 레지스터에 저장되기 전에, ALU 결과를 직접 다음 명령에 직접 전달하는 **데이터 포워딩**

조건 분기가 존재하는 경우의 시간 흐름도

3. 제어 장애(control hazards)

- 조건 분기(conditional branch) 명령어가 실행되면, 미리 인출하여 처리하던 명령어들이 무효화 됨
- **[예]** 명령어 3: JUMP 12; jump(if zero) to address 12



제어 장애에 의한 성능 저하의 최소화 방법

- **분기 예측(branch prediction)**
  - 분기가 일어날 것인 지를 예측하고, 그에 따라 명령어를 인출하는 확률적 방법
    - 분기 역사 표(branch history table) 이용하여 최근의 분기 결과를 참조
- **분기 목적지 선인출(prefetch branch target)**
  - 조건 분기가 인식되면, 분기 명령어의 다음 명령어 뿐만 아니라 분기의 목적지 명령어도 함께 인출하여 실행하는 방법
    - 조건 확인 결과에 따라, 유효 명령어 결과를 선택하여 실행
- **루프 버퍼(loop buffer) 사용**
  - 파이프라인의 명령어 인출 단계에 포함되어 있는 작은 고속 기억장치인 루프 버퍼에 가장 최근 인출된 n개의 명령어들을 순서대로 저장해두는 방법
- **지연 분기(delayed branch)**
  - 분기 명령어의 위치를 재배치함으로써 파이프라인의 성능을 개선하는 방법

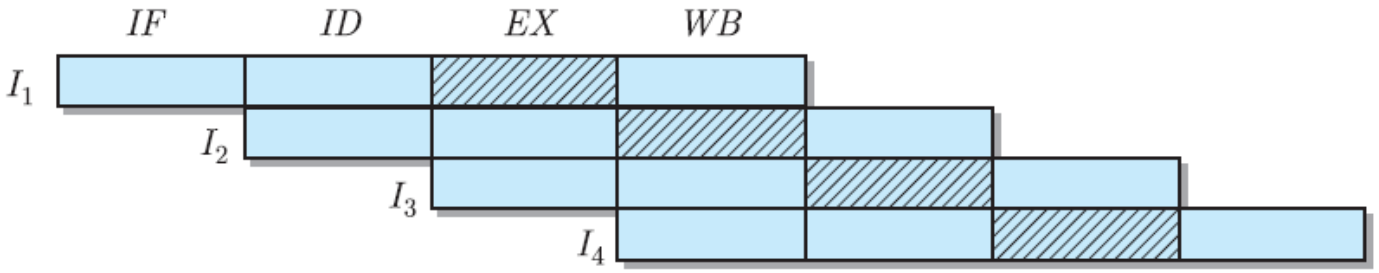
슈퍼스칼라

- **슈퍼스칼라(Superscalar)**
  - CPU의 처리 속도를 더욱 높이기 위하여, 내부에 두 개 혹은 그 이상의 명령어 파이프라인들을 포함시킨 구조
    - 즉, **하나의 프로세서 안에 2개 이상의 파이프라인 탑재**

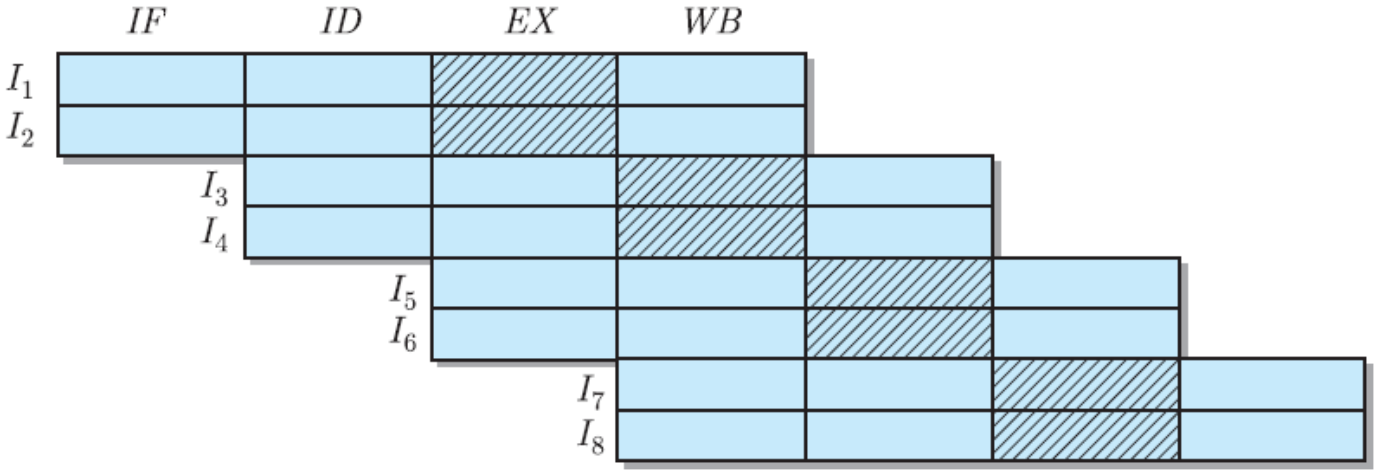
- 매 클럭 주기마다 각 명령어 파이프라인이 별도의 명령어를 인출하여 동시에 실행
  - 이론적으로는 프로그램 처리 속도가 파이프라인의 수만큼 향상 가능

- 파이프라인의 수 = ***m* : *m*-way 슈퍼스칼라**

2-way 슈퍼스칼라의 명령어 실행 흐름도



(a) 일반적인 파이프라인의 명령어 실행 시간도



(b) 2-way 슈퍼스칼라의 명령어 실행 시간도

슈퍼스칼라의 속도 향상

- 단일 파이프라인에 의한 실행 시간(N: 실행할 명령어 수)

$$T(1) = k + N - 1$$

- ***m*-way** 슈퍼스칼라에 의한 실행 시간

$$T(m) = k + \frac{N-m}{m}$$

- 속도 향상(S_p)

$$S_p = \frac{T(1)}{T(m)} = \frac{k+N-1}{k+(N-m)/m} = \frac{m(k+N-1)}{N+m(k-1)}$$

- 명령어 수 $N \rightarrow \infty$, $S_p \rightarrow m$
  - 슈퍼스칼라 프로세서의 속도는 파이프라인을 사용하지 않는 프로세서에 비해서 이론상 최대 mk배 성능 향상 가능

슈퍼스칼라의 속도 저하 요인

- 명령어들 간의 **데이터 의존성**
  - 한 명령어를 실행한 다음에, 그 결과값을 보내주어야 다름 명령어의 실행이 가능한 관계

- 하드웨어(레지스터, 캐시, 기억장치 등) 이용에 대한 경합 발생
  - 동시 실행 가능한 명령어 수 $< m$

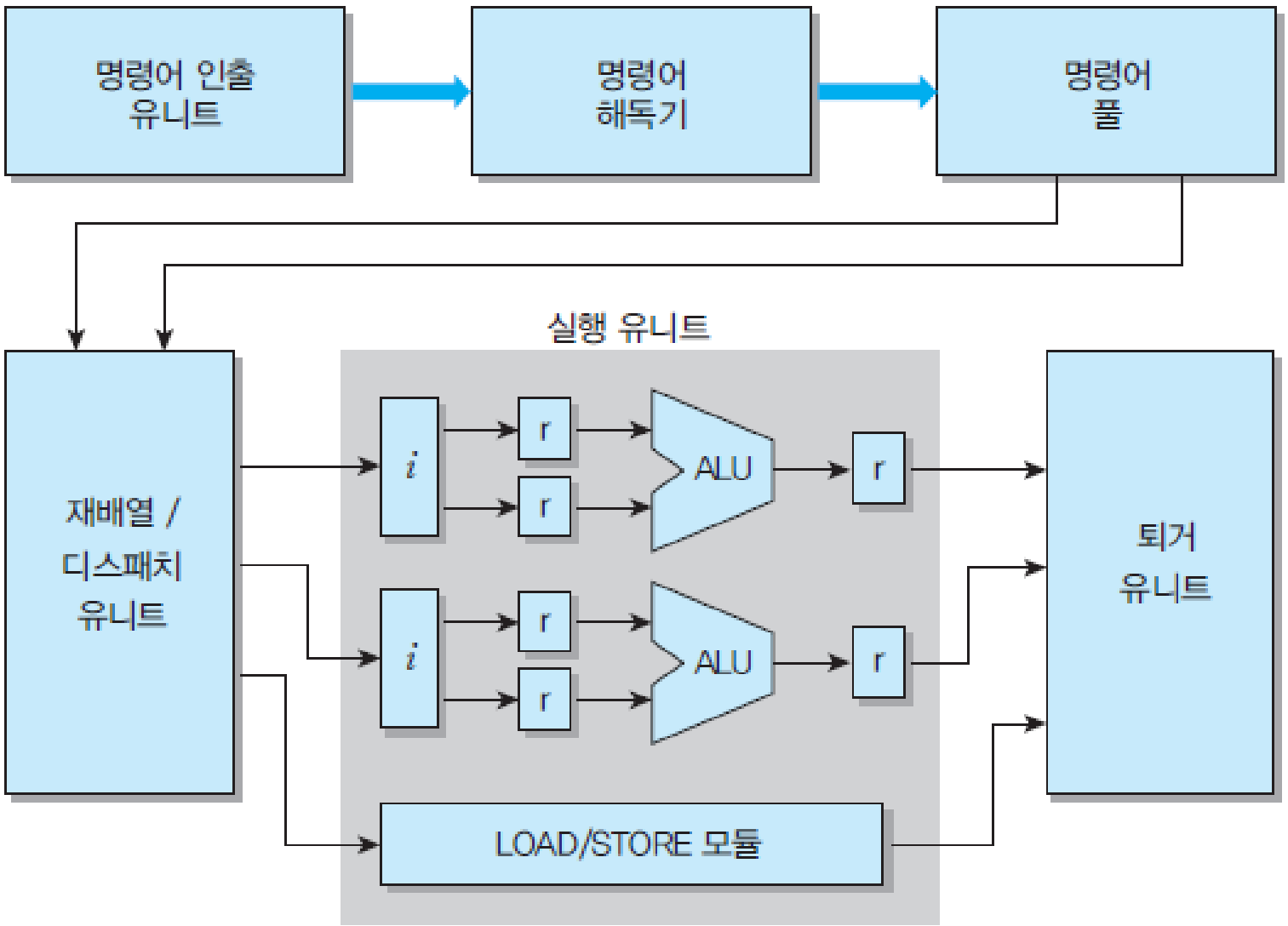
- **해결 방안:**
  - 명령어 실행 순서 재배치 → 명령어들 간의 데이터 의존성 제거
  - 하드웨어 추가(중복) 설치 → 하드웨어(ALU, 레지스터, 캐시 등)에 대한 경합 감소

동적 실행

동적 실행(dynamic execution)

- 프로그램의 실행 시, 명령어들을 컴파일된 순서 그대로 실행하지 않고, 데이터 의존성을 고려하여 순서를 바꿔 실행하는 기법
  - 데이터 의존성 때문에 발생하는 유휴 사이클(Idle Cycle) 을 줄여, 슈퍼스칼라 구조의 실행 효율을 높임
- 동작 과정
  - ① 명령어 인출(Fetch): 기억장치에서 명령어들을 원래 순서대로 인출
  - ② 명령어 해독(Decode): 해독된 정보를 명령어 풀(Instruction Pool)에 저장
  - ③ 의존성 검사(Dependency Check): 순서상 연속된 명령어 중 데이터 의존성 때문에 동시에 실행 불가 시, 의존성 없는 명령어를 풀에서 선택
  - ④ 디스패치(Dispatch): 동시 처리 가능한 명령어들을 실행 유닛(ALU, Load/Store Unit 등)에 할당하여 실행
  - ⑤ 퇴거(Retire): 실행이 끝난 명령어의 결과를 프로그램 순서대로 확정(commit)하여 레지스터·메모리에 반영하고, 명령어 풀에서 제거

[예] 동적 실행 가능한 2-way 슈퍼스칼라 구조



[예] 동적 실행 가능한 2-way 슈퍼스칼라 구조

I1: LOAD R1, X ; R1 ← M[X]

I2: ADD R1, R2 ; R1 ← R1 + R2

I3: SUB R3, R4 ; R3 ← R3 - R4

I4: STOR Y, R3 ; M[Y] ← R3

- 실행 순서
  - ① 매 사이클마다 두 명령어 씩(***I1***과 ***I2***, ***I3***과 ***I4***)을 순서대로 인출하여, 해독하고, 명령어 풀에 저장
  - ② 재배열/디스패치 유니트가 동시 실행 가능한 명령어들 검색
    - ***I1***과 ***I2*** 간 데이터 의존성 존재: 동시 실행 불가 → 원래 순서와 다르게, (***I1***과 ***I3***) (***I2***와 ***I4***)를 순차적으로 동시 실행
  - ③ 퇴거 유니트는 연산 처리 결과를 저장하고, 실행 완료된 명령어들 제거

듀얼-코어 및 멀티-코어

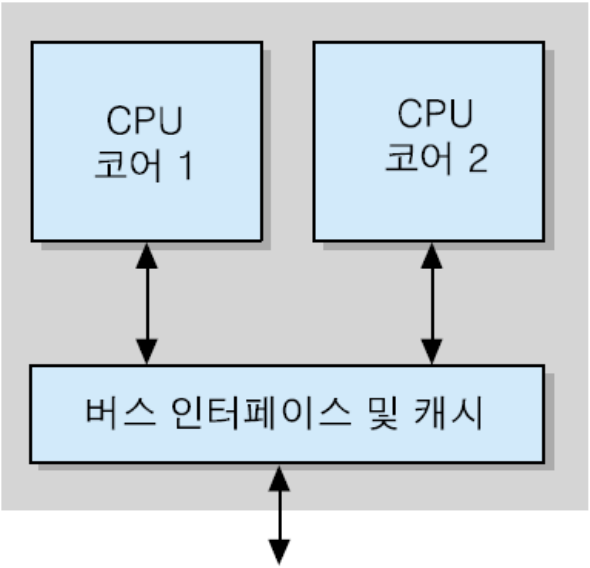
- **CPU 코어(core):** 명령어 실행에 필요한 CPU 내부의 핵심 하드웨어 모듈
  - 슈퍼스칼라 실행 모듈, 산술논리연산장치, 부동소수점연산장치(Floating Point Unit, FPU), 레지스터 세트, 제어장치, 온 칩 캐시(on-chip cache) 등

- **멀티-코어 프로세서(multi-core processor):** 여러 개의 CPU 코어들을 하나의 칩에 포함시킨 프로세서
  - 칩-레벨 다중프로세서(chip-level multiprocessor) 혹은 단일-칩 다중프로세서(multiprocessor-on-a-chip)이라고도 부름
  - 듀얼-코어(dual-core): CPU 코어 2개 포함
  - 쿼드-코어(quad-core): CPU 코어 4개 포함
  - 헥사-코어(hexa-core): CPU 코어 6개 포함
  - 옥타-코어(octa-core): CPU 코어 8개 포함

듀얼-코어 및 멀티-코어

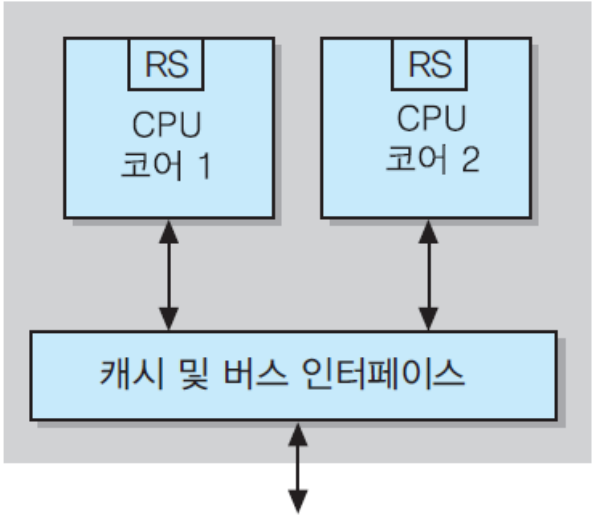
듀얼-코어 프로세서

- 단일 코어 슈퍼스칼라 프로세서 대비, 이론적으로 최대 2배 성능 향상 가능
- 코어들은 내부 캐시와 시스템 버스 인터페이스만 공유
- 코어 별로 독립적 프로그램 실행하여 멀티-태스킹(multi-tasking)과 멀티-스레딩(multi-threading) 지원
- 멀티-태스킹: 여러 CPU 코어들을 사용하여 독립적인 프로세스를 동시에 실행



듀얼-코어 및 멀티-코어

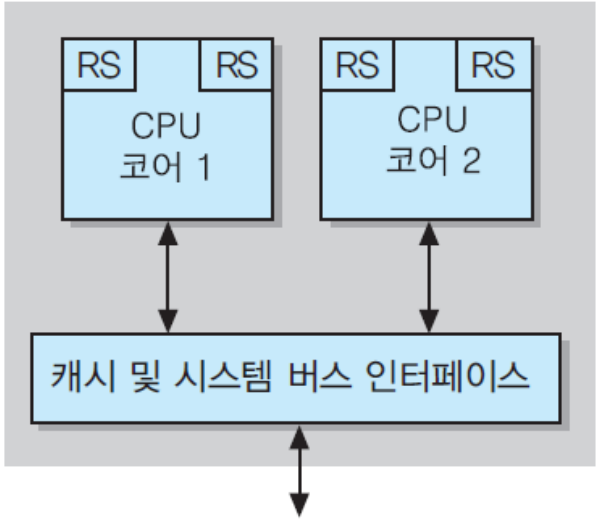
- **멀티-스레딩:** 하나의 CPU 코어가 여러 스레드를 동시에 실행
  - **프로세스(process):** 운영체제가 자원을 할당하는 독립적인 작업 단위
  - **스레드(thread):** 독립적으로 실행될 수 있는 최소 크기의 프로그램 단위
    - 여러 스레드가 프로세스 자원을 공유하면서 독립적으로 실행하여 동시성 구현
  - **단일-스레드 모델(그림(a)):** 각 코어가 스레드를 하나씩 처리
    - 레지스터 세트(Register Set, **RS**)에서 스레드의 실행 상태를 저장하며, 프로그램 카운터(PC), 스택 포인터(SP), 상태 레지스터 등을 포함



(a) 단일-스레드 모델

듀얼-코어 및 멀티-코어

- **멀티-스레드 모델(그림(b)):**
  - 각 코어는 두 개의 RS를 보유하여, 최대 두 개의 스레드를 동시에 처리
  - 두 스레드들이 CPU 코어의 H/W 자원들([예] ALU, FPU, 온-칩 캐시 등)을 공유
  - 스레드별 시스템 상태(PC, SP, 상태 레지스터 등)는 각자의 RS에 저장되어 독립성 유지
- **듀얼-코어 멀티-스레드 프로세서:** 두 개의 물리적 프로세서(physical processor)들이 네 개의 논리적 프로세서(logical processor)들로 구성



(b) 멀티-스레드 모델

혼합형 멀티-코어 프로세서

- 혼합형 멀티-코어 프로세서(Hybrid Multi-core Processor)
  - 성능과 전력 효율이 다른 CPU 코어들을 하나의 칩에 포함한 구조
    - 복잡한 태스크는 고성능(일반적인) 코어가 처리
    - 단순한 태스크는 단순 구조의 저전력 코어가 처리
  - 저전력이 중요한 랩탑·스마트폰 등에서, 단순 태스크(작업)과 복잡 태스크이 혼재된 환경을 효율적으로 처리
  - **[예]** Intel사의 혼합형 멀티-코어 프로세서 구조(최대 20개 스레드 동시 처리)
    - P-코어(performance-core): 고성능 CPU 코어, 복잡한 태스크 처리, 멀티-스레딩 지원
    - E-코어(efficient-core): 단순·저전력 CPU 코어, 작은 태스크·백그라운드 태스크 처리, 멀티-스레딩 미지원

<table>
<tr><td rowspan="2">P-코어</td><td rowspan="2">P-코어</td><td rowspan="2">P-코어</td><td>E-코어</td><td>E-코어</td></tr>
<tr><td>E-코어</td><td>E-코어</td></tr>
<tr><td rowspan="2">P-코어</td><td rowspan="2">P-코어</td><td rowspan="2">P-코어</td><td>E-코어</td><td>E-코어</td></tr>
<tr><td>E-코어</td><td>E-코어</td></tr>
</table>

Q & A

End!