어셈블리 프로그래밍

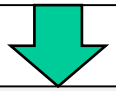
- 어셈블리언어/기계어 프로그래밍
  - ✓ 하드웨어 명령어를 사용한 프로그래밍
- 프로세서: **SPARC**
- 프로그래밍 도구
  - ✓ 컴파일러, 어셈블러: **gcc**
  - ✓ 매크로 프로세서: m4
  - ✓ 디버거: **gdb**

기계언어와 어셈블리언어

- 프로그래밍 모델: 메모리 + CPU
- 기계언어: 이진형태(수치) 언어
- 어셈블리 언어: 사람이 읽고 쓰기 쉽게 기호화된 기계 명령어로 구성된 프로그래밍 언어
- 어셈블러: 어셈블리 프로그램을 기계어 프로그램으로 변환하는 프로그램

원시, 어셈블리, 기계어 프로그램 예

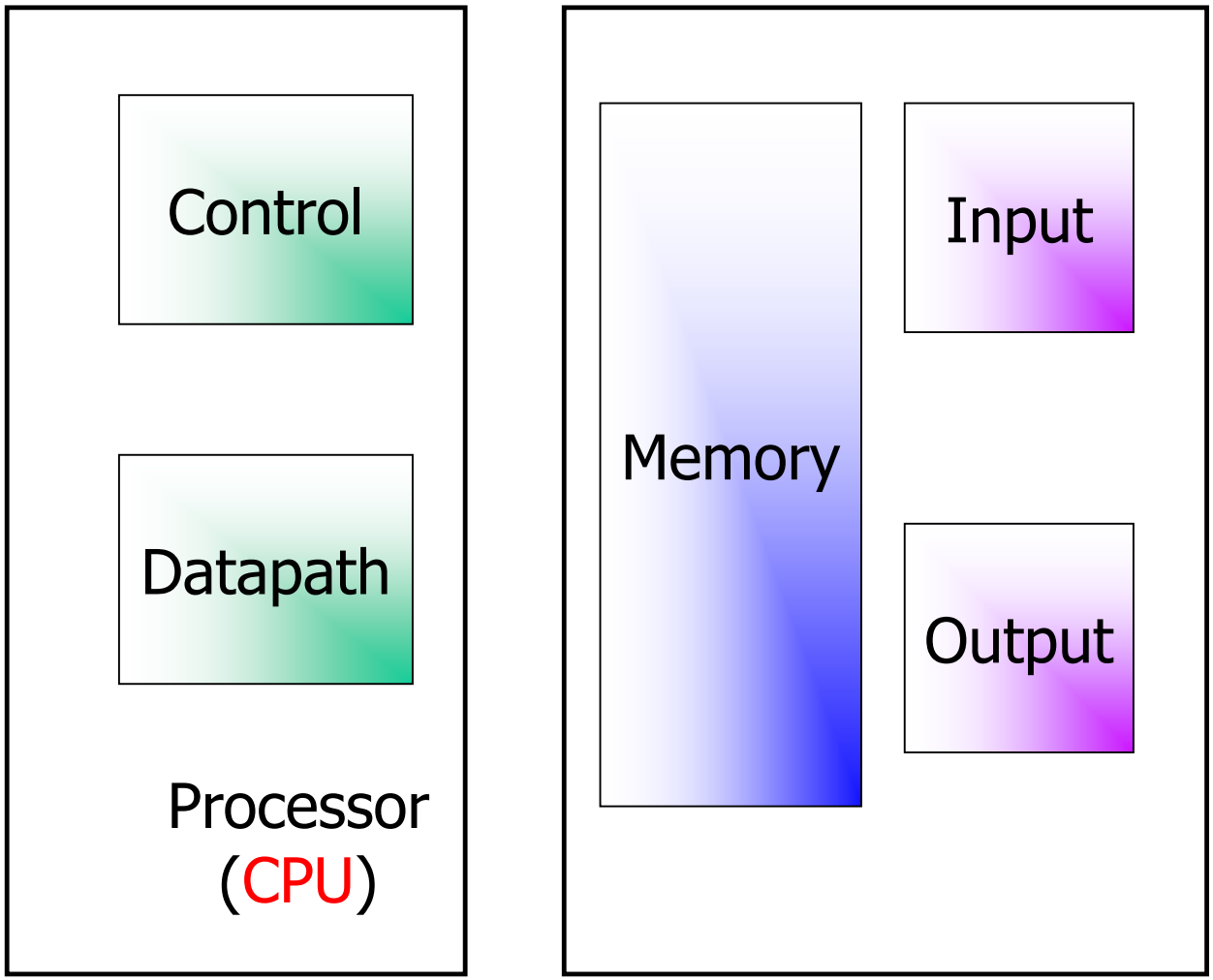
```
void func (int x) {
        x = x + 1;
}
```



```
save    %sp, -112, %sp
st      %i0, [%fp + 0x44]
ld      [%fp + 0x44], %o0
add     %o0, 1, %o1
st      %o1, [%fp + 0x44]
ret
restore
```

```
0x9de3bf90
0xf027a044
0xd007a044
0x92022001
0x81c7e008
0x81c7e008
0x81e80000
```

컴퓨터 구성 원소(1)



컴퓨터 구성 원소(2)

- ALU (Arithmetic Logic Unit)
  - ✓ 계산 기능(+, -, *, /, &등)
- CU (control unit)
  - ✓ 제어 기능
- Memory
  - ✓ 프로그램과 데이타 저장 기능
- I/O devices

컴퓨터 구성 원소(3)

- 기억 소자의 종류
  - memory
    - ✓ cache
    - ✓ main memory
    - ✓ auxiliary memory
  - register
  - stack
  - flip flop

컴퓨터 구성 원소(4)

- **register (레지스터)**
  - ✓ 프로세서에 포함되어 있는 기억 소자
  - ✓ 일반적으로 word 크기
    - SPARC의 경우: 32 bits
  - ✓ 계산의 중간 결과 또는 자주 사용되는 값을 저장
  - ✓ 접근하기 위해서는 레지스터 이름이 필요
  - ✓ 시스템이 사용하는 것도 있음
    - ➢ 예 PC(program counter)
      IR(instruction register)

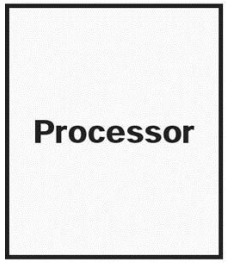
Von Neuman Machine

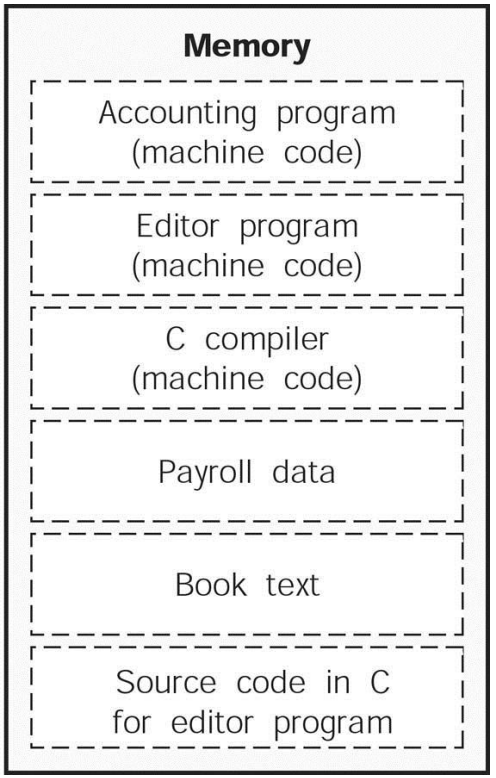
- stored program computer
  - ✓ 프로그램과 데이터를 메모리에 저장하여 실행
- 실행 주기(**instruction cycle**)

```
pc = 0;                          /* 프로그램 카운터 초기화 */
do {
   instruction = memory[pc++];   /* 명령 fetch */
   decode(instruction);          /* decode*/
   fetch(operands);              /* 피연산자 fetch */
   execute;                      /* 실행 */
   store(results);               /* 결과 저장 */
} while (instruction != halt);
```

- 이진코드를 메모리로 부터 하나씩 CPU로 이동하여 실행

- Instruction(명령어)은 수치(numbers) 형태로 표현
- Programs이 수치 형태로 메모리에 기억
- 프로세서가 실행





- 정보
  - 명령어
  - 데이터
    - 수치
    - 비수치

Stored-program concept

instruction (명령어)

- 명령어(instruction)
  - ✓ 명령어 형식(instruction format)

Operation code	Operand	Addressing mode
• 일을 정의 • 예 add(+), sub(-)	• 오퍼랜드 위치 정보	• **operand 필드의 해석 방법** • 예 메모리 주소, 레지스터 이름

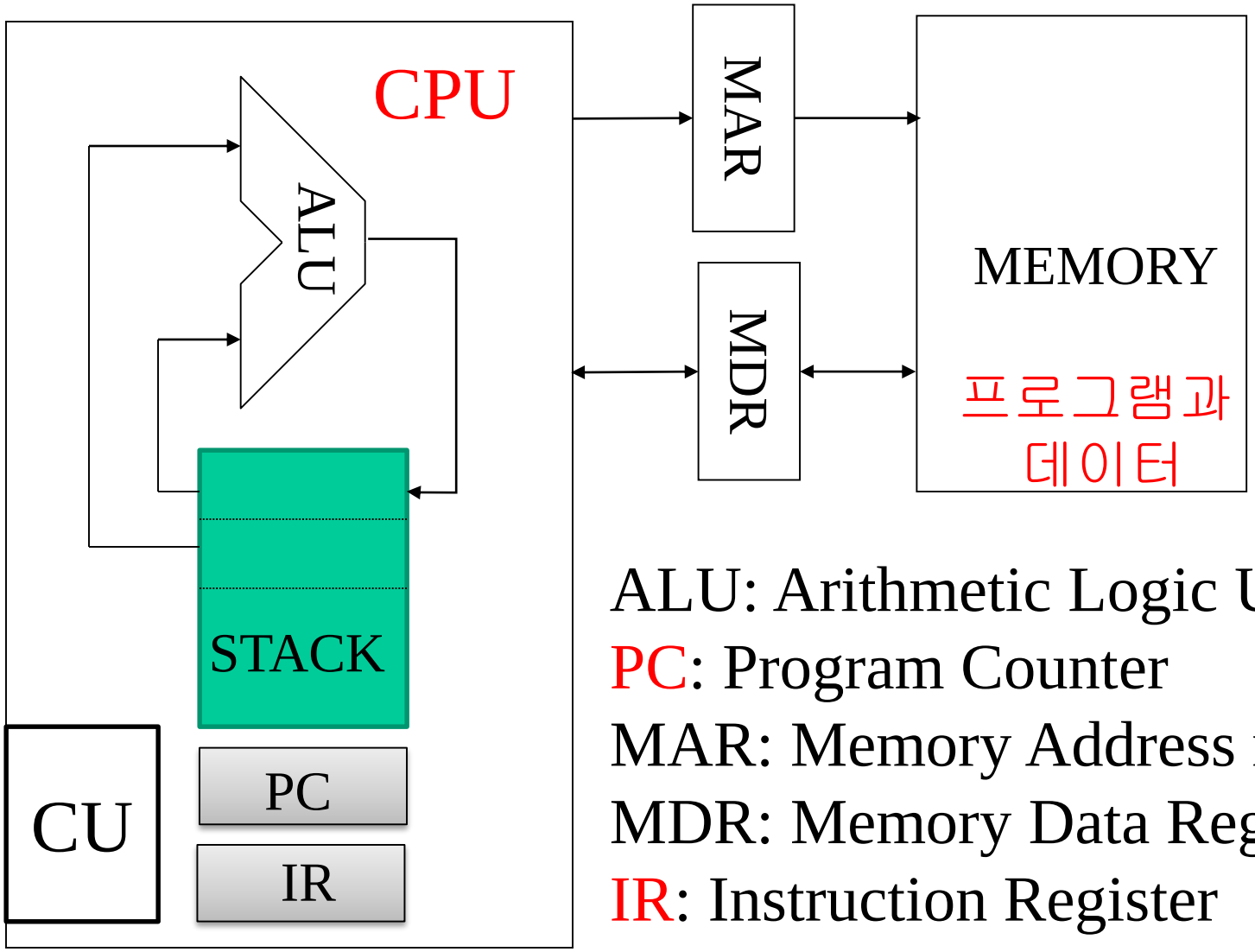
Instruction Set

- 명령어 집합(Instruction set)
  - ✓ 데이터 이동
    - ➢ mov, load, store
  - ✓ 데이터 처리
    - ➢ add, sub, and, or
  - ✓ 프로그램 흐름 제어
    - ➢ call, ba, bl
  - ✓ 입/출력

컴퓨터 구조의 분류

1. 스택 기계(stack machine)
2. 단일 레지스터 기계(accumulator; AC machine)
3. 다중 레지스터 기계(load/store machine)

스택 기계 구조



ALU: Arithmetic Logic Unit

PC: Program Counter

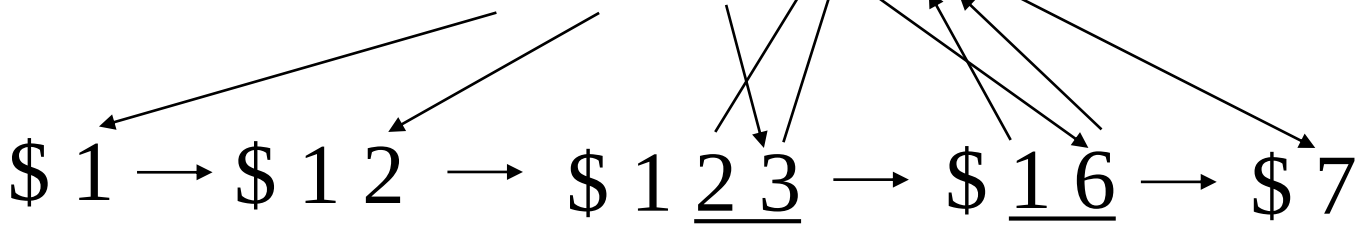
MAR: Memory Address register

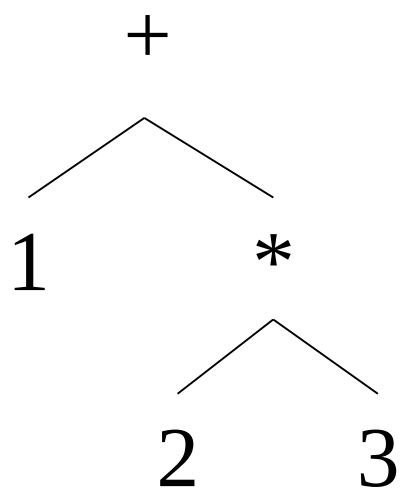
MDR: Memory Data Register

IR: Instruction Register

스택 기계 작동

- 후위 표기식 (postfix notation)
- 1 + 2 * 3 → 1 2 3 * +





스택 기계의 특징

- 스택 연산
  - ✓ push: 메모리에서 스택으로 적재 (상수)
  - ✓ pop: 스택에서 추출하여 메모리로
- **ALU 연산**
  - ✓ 피연산자를 지정하지 않음

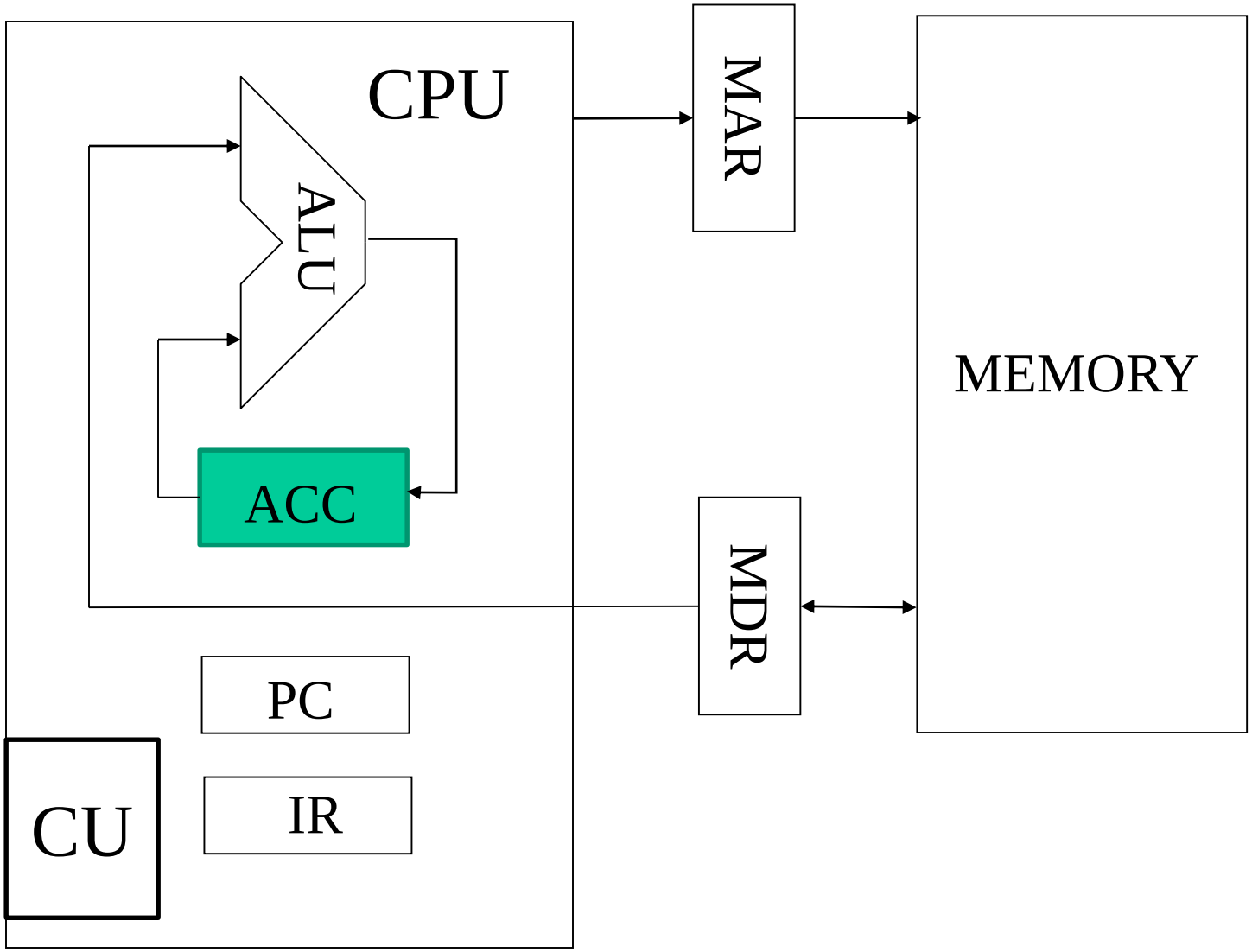
    (**0** address machine)
  - ✓ 피연산자는 항상 스택 상위에 위치
  - ✓ 연산 결과는 스택 상단에 저장

프로그램 예

- X = (A + B) * (C + D)를 위한 프로그램의 예

```
PUSH  A      ! TOS ← A
PUSH  B      ! TOS ← B
ADD          ! TOS ← (A + B)
PUSH  C      ! TOS ← C
PUSH  D      ! TOS ← D
ADD          ! TOS ← (C + D)
MUL          ! TOS ← (C + D) * (A + B)
POP   X      ! M[X] ← TOS
```

누적기(Accumulator) 기계 구조



누적기 기계의 특징

- 단일 레지스터 기계
  - ✓ 계산 결과는 누적기(AC)에 저장
- **ALU 연산**
  - ✓ 피연산자중의 하나는 AC에
  - ✓ 다른 피연산자는? 명령어에서 지정(memory)

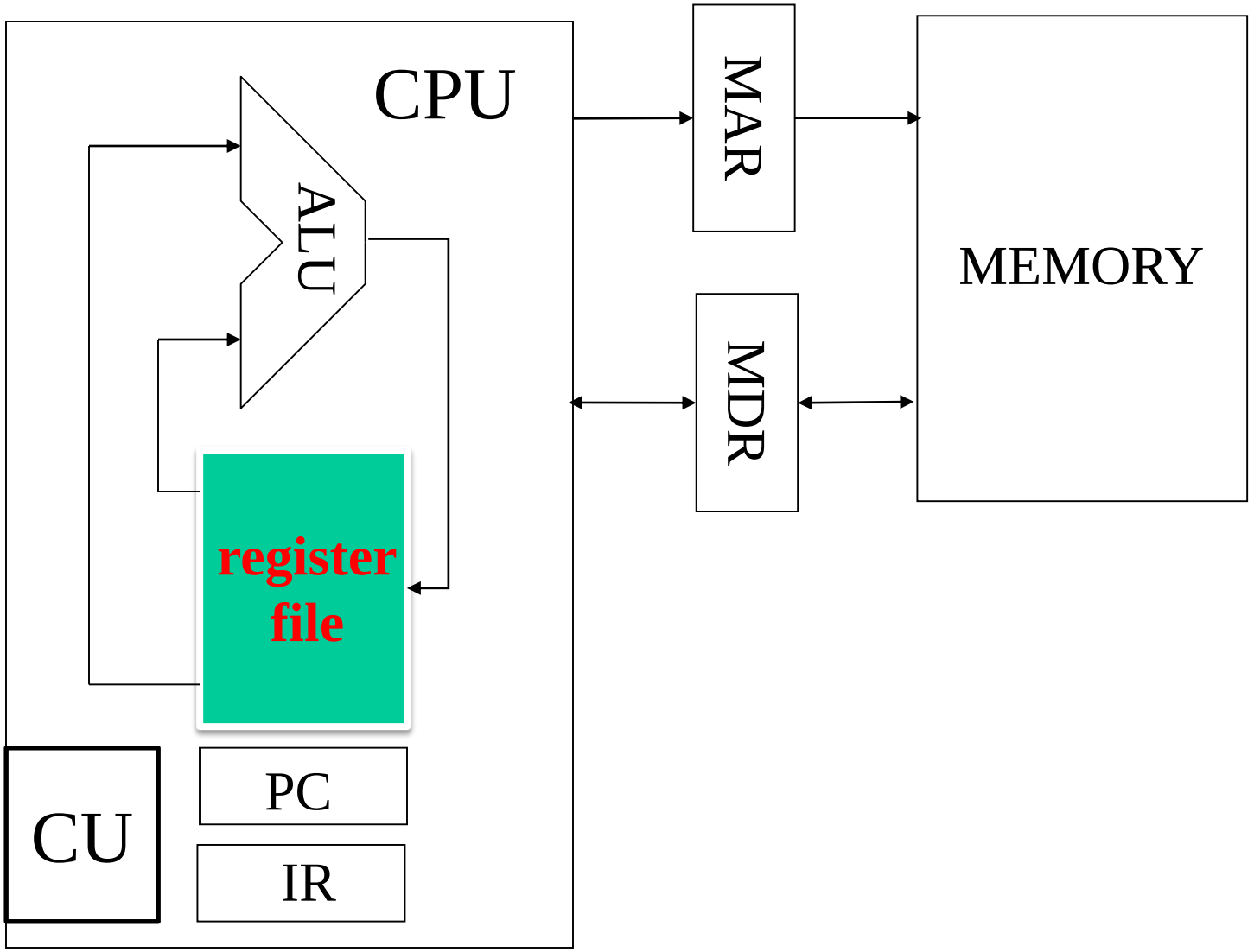
    (**1** address machine)
- 데이타 이동
  - ✓ load: 메모리에서 누적기
  - ✓ store: 누적기에서 메모리

프로그램 예

- X = (A + B) * (C + D)를 위한 프로그램의 예

```
LOAD   A        ! AC ← M[A]
ADD    B        ! AC ← AC + M[B]
STORE  T        ! M[T] ← AC
LOAD   C        ! AC ← M[C]
ADD    D        ! AC ← AC + M[D]
MUL    T        ! AC ← AC * M[T]
STORE  X        ! M[X] ← AC
```

Load/Store 기계 구조



Load/Store 기계의 특징

- CPU에 많은 수의 레지스터(레지스터 화일) 존재
- 메모리 접근
  - ✓ load와 store 명령으로 제한
- **ALU 연산**
  - ✓ 피연산자들은 레지스터 화일로부터 공급
  - ✓ 결과도 레지스터 화일중의 한 레지스터로

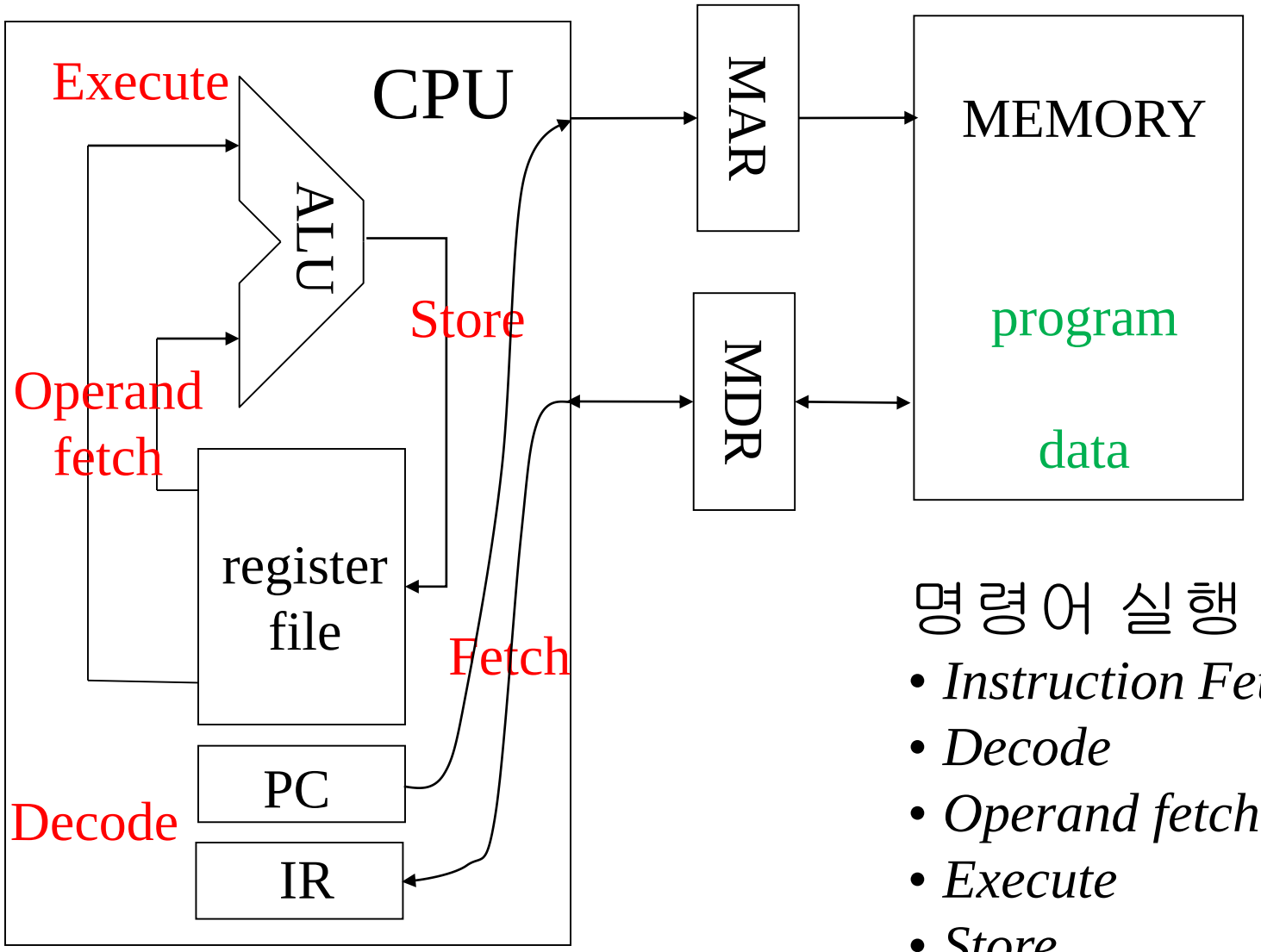
    (2 or **3** address machine)

프로그램 예

- X = (A + B) * (C + D)를 위한 프로그램의 예

```
LOAD  A, R1        ! R1 ← M[A]
LOAD  B, R2        ! R2 ← M[B]
ADD   R1, R2, R3   ! R3 ← R1 + R2
LOAD  C, R1        ! R1 ← M[C]
LOAD  D, R2        ! R2 ← M[D]
ADD   R1, R2, R4   ! R4 ← R1 + R2
MUL   R3, R4, R5   ! R5 ← R3 * R4
STORE R5, X        ! M[X] ← R5
```

Load/Store 프로세서 실행 모델



명령어 실행 단계

- *Instruction Fetch*
- *Decode*
- *Operand fetch*
- *Execute*
- *Store*

파이프라이닝(pipelining)

- 순차적인 명령어 스트림에 있는 명령어들간에 병렬성을 이용
- 여러 개의 명령어가 중첩(overlap) 실행되어 처리 성능(throughput)을 높임
  - ✓ 1 CPU에서의 병렬 처리
- 각각의 명령어 실행시간(execution time)은 변화 없음
- 명령어 처리 과정을 일정한 단계(stage)로 나누어 처리

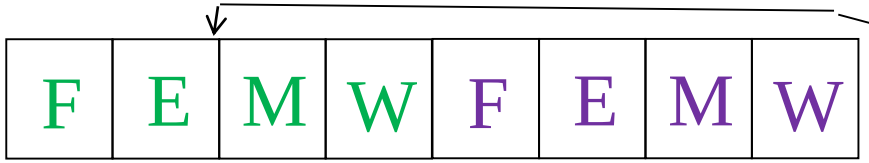
(각 단계의 진행 시간은 같다)

- ✓ F: Fetch and Decode, getting operands
- ✓ E: Execute
- ✓ M: Memory access
- ✓ W: Write back

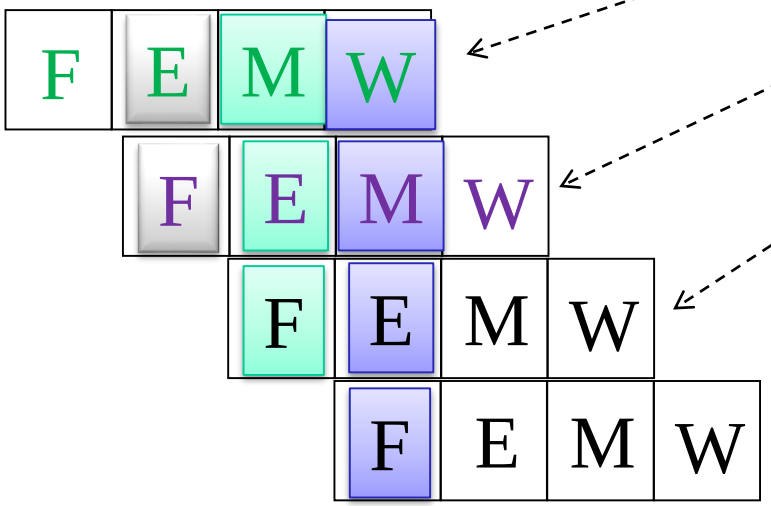
- F： instruction fetch & decoding,
  obtain operands from register files
- E：execute arithmetic instruction,
  compute branch target address,
  compute the memory address for load/store
- M：access memory for load/store,
  fetch the instruction at the target of
  a branch instruction
- W：write the results back to the register files

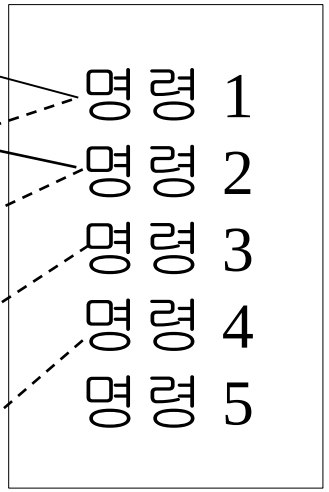
파이프라인에 의한 병렬성

- 파이프라인을 사용하지 않을 때

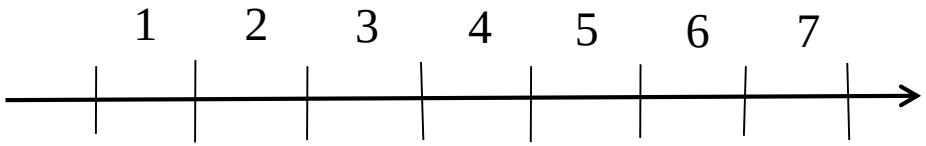


- 파이프라인을 사용할 경우

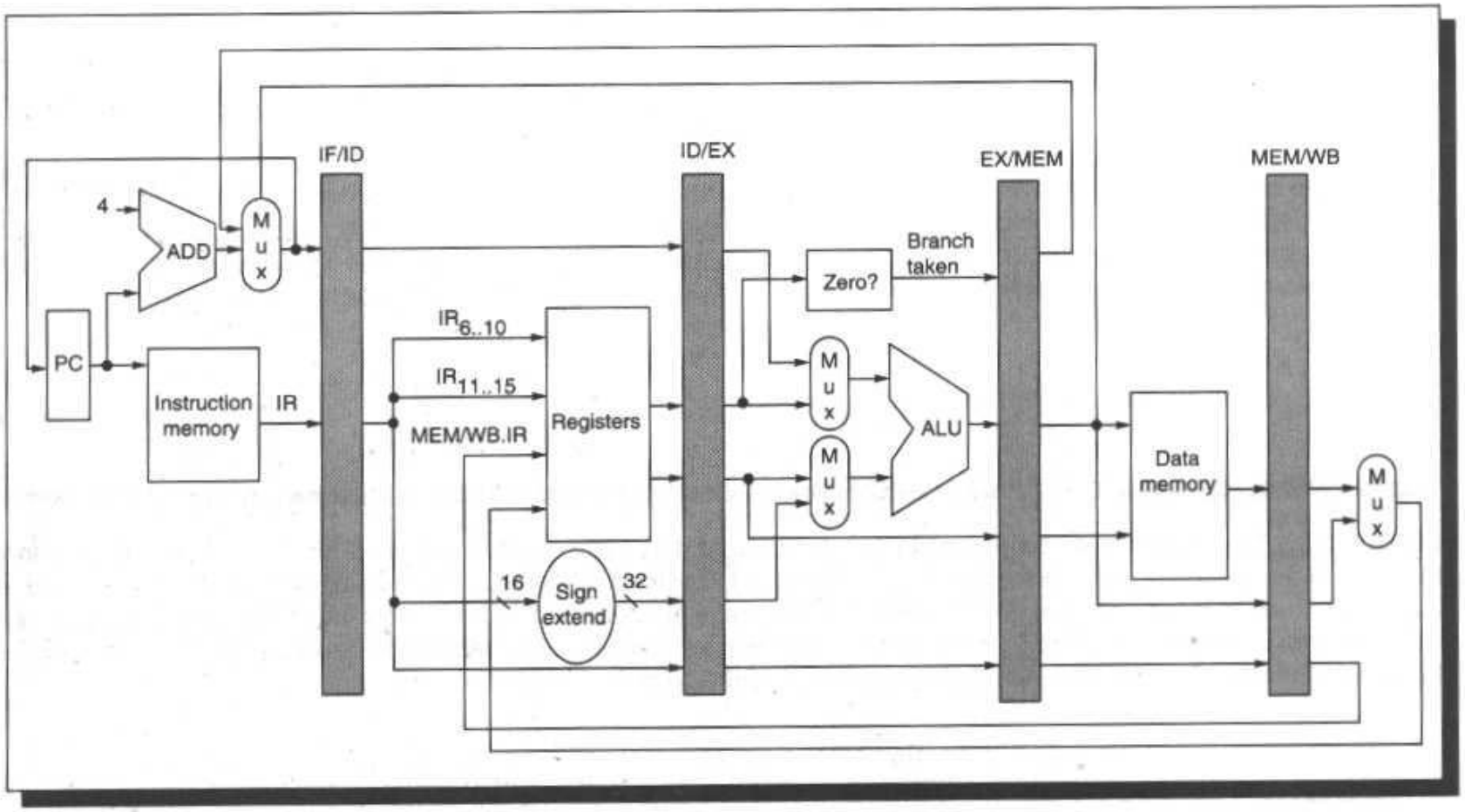




성능 향상은?



파이프라인 구조의 예


IF/ID
ID/EX
EX/MEM
MEM/WB
4
ADD
M u x
PC
Instruction memory
IR
IR6..10
IR11..15
MEM/WB.IR
Registers
16
Sign extend
32
Zero?
Branch taken
M u x
M u x
ALU
Data memory
M u x

문제점: 해저드 (hazard)

- 후속 명령어가 후속 클럭 사이클에서 실행될 수 없는 상황

① 구조적 해저드(structural hazard)

✓ 하나의 하드웨어를 같은 클럭에서 두 명령어가 사용하려는 경우

즉 하드웨어가 충분하지 않음 예: 명령어 인출과 메모리 데이터 접근

✓ 해결책: 하드웨어 추가

② 제어 해저드(control hazard)

✓ 분기(branch) 명령어를 만나는 경우

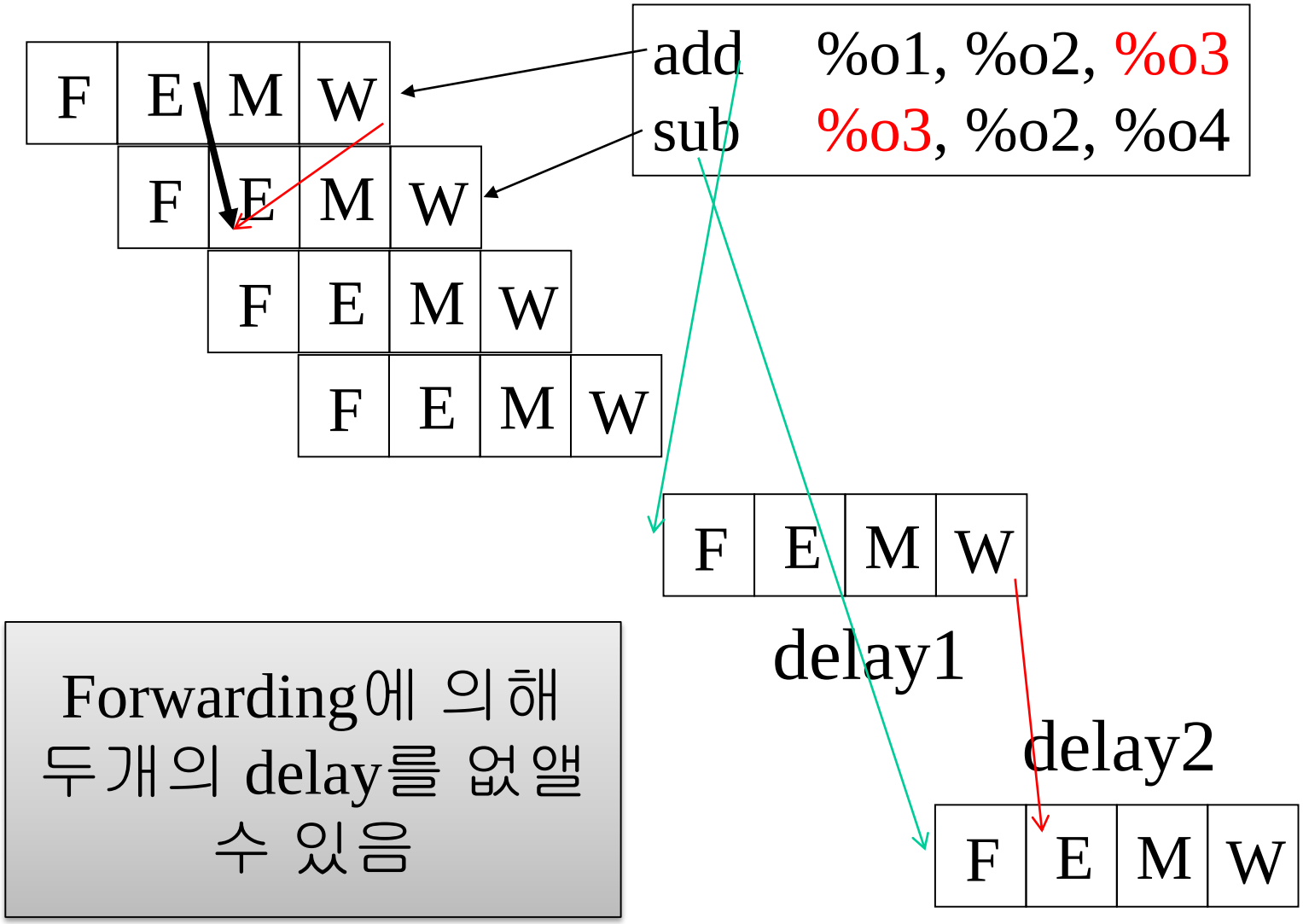
✓ 해결책: 파이프라인 지연(stall), 분기 예측, 지연분기(delayed branch)

③ 데이터 해저드(data hazard)

✓ 이전 단계의 결과를 사용하려는 경우 즉 데이터 종속(data dependence)

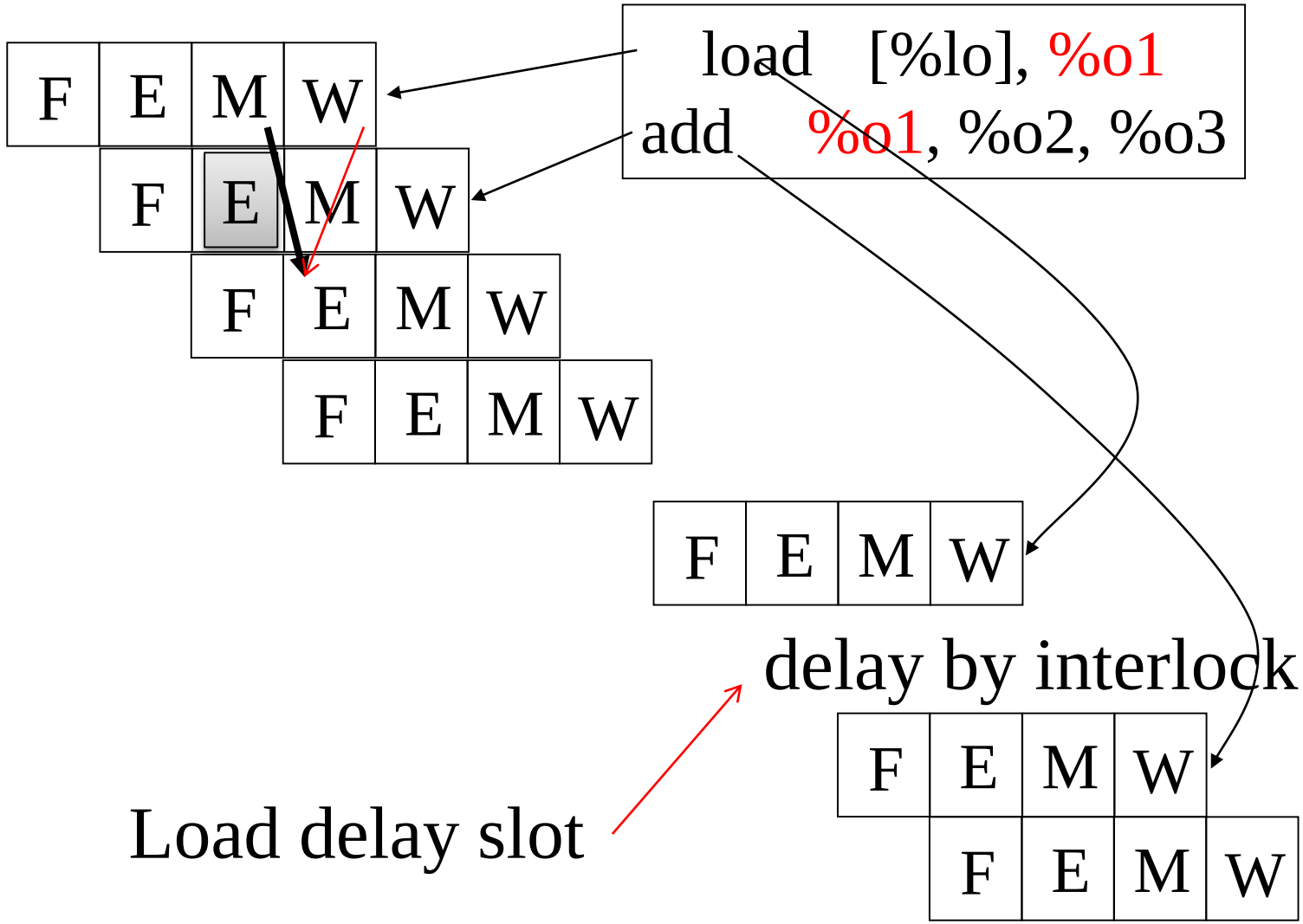
✓ 해결책: 파이프라인 지연(stall) 또는 전방전달(forwarding or bypassing)

전방전달(forwarding)



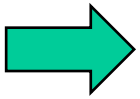
Forwarding에 의해 두개의 delay를 없앨 수 있음

Load Delay


load [%lo], %o1
add %o1, %o2, %o3
F E M W
F E M W
F E M W
F E M W
F E M W
delay by interlock
F E M W
F E M W
Load delay slot

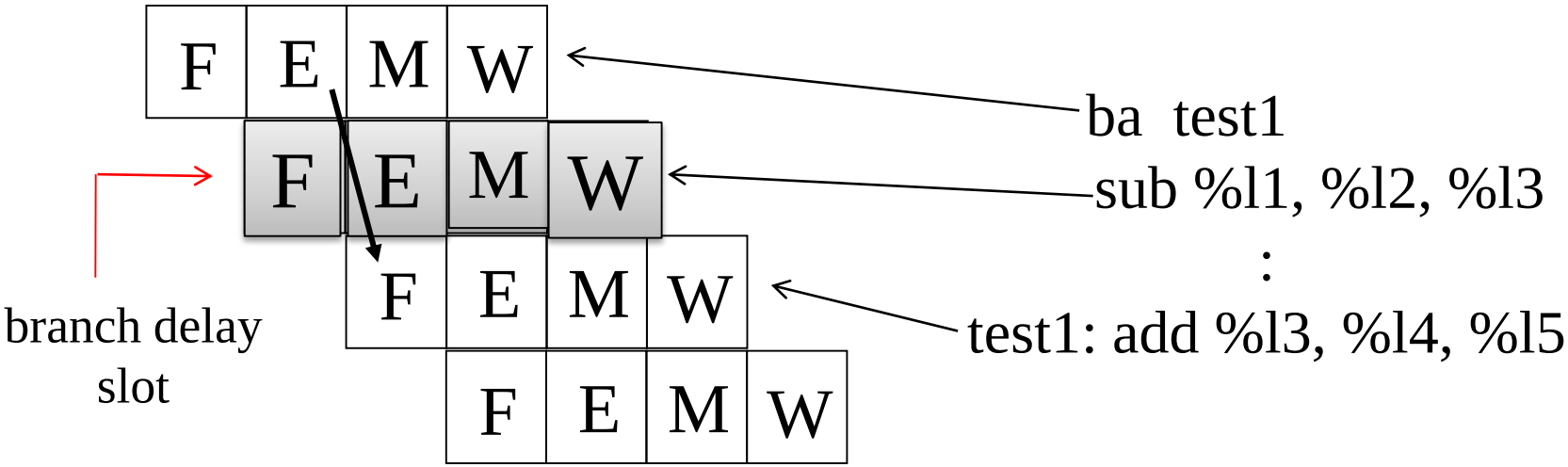
- 코드 재배치

```
load  [R0], R1
add   R1, R2, R3
sub   R4, R2, R5
  :
```



```
load [R0], R1
sub  R4, R2, R5
add  R1, R2, R3
   :
```

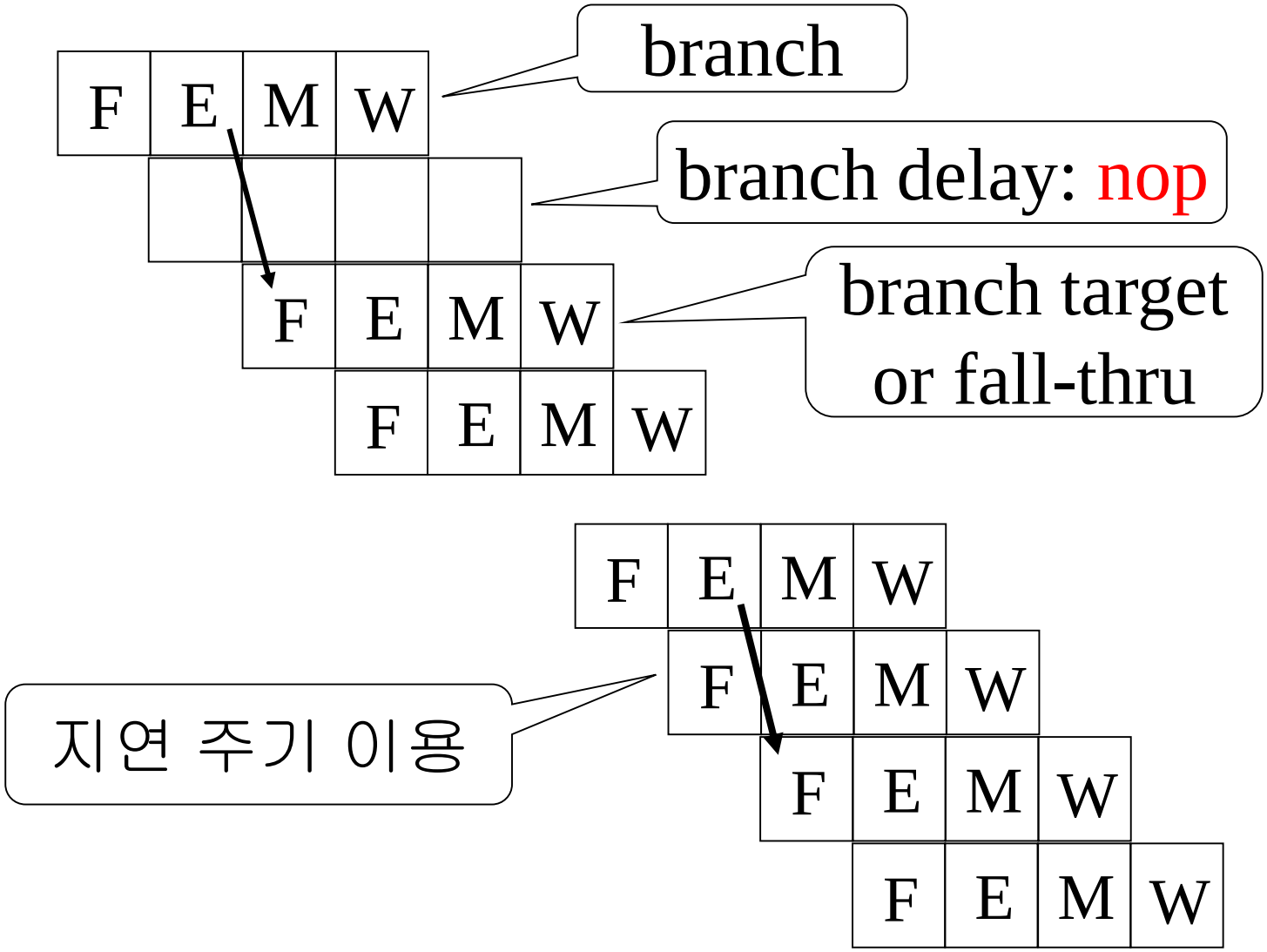
Branch Delay(1)



- 분기 명령 다음(branch delay slot)에 위치한 명령은 실행

- 분기 명령어 목적지에 있는 명령어를 반입하기 전에 파이프라인 청소 필요 → 또는 nop 명령 사용

Branch Delay(2)



- 코드 재배치

```
        :                       :
    mov R1, R2                  :
    ba   t                   ba t
    nop                      mov R1, R2
       :                        :
       :                        :
t:  add R3, R4, R5      t:  add R3, R4, R5
    sub R3, R6, R7          sub R3, R6, R7
```