# CPU의 구조와 기능 (1)
# (제 3주 차)

서울사이버대학교

오 창 환

# 학습목표



- CPU의 기본 구조를 설명할 수 있다.
- 명령어 실행을 설명할 수 있다.
- 명령어 파이프라이닝을 설명할 수 있다.

# 학습 내용

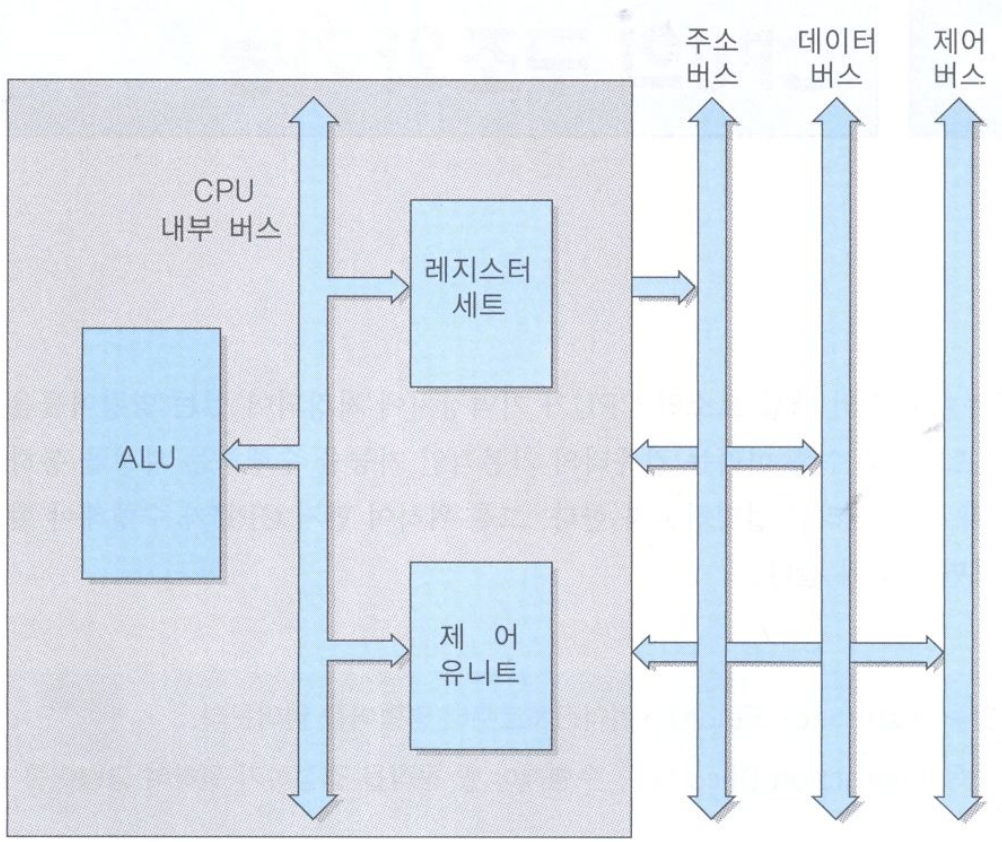


- CPU의 기본 구조
- 명령어 실행
- 명령어 파이프라이닝

# CPU의 기본 구조 (1)



• CPU는 산술논리연산장치(ALU : Arithmetic and Logical Unit), 레지스터 세트, 제어 유니트 등으로 구성됨.




• CPU의 내부 구조 및 시스템 버스와의 접속

# CPU의 기본 구조 (2)



- ALU는 각종 산술 연산들과 논리 연산들을 수행하는 회로들로 이루어진 하드웨어 모듈이며, 여기에서 산술 연산이란 덧셈, 뺄셈, 곱셈, 나눗셈 등을 말하고, 논리 연산으로는 AND, OR, NOT 연산 등이 있음.
- 레지스터는 CPU 내부에 위치한 기억 장치이며, 액세스 속도가 컴퓨터의 기억 장치들 중에서 가장 빠르지만, 레지스터는 내부 회로가 복잡하여 비교적 큰 공간을 차지하기 때문에 많은 수의 레지스터들을 CPU에 포함시키기 어려움. 따라서 지정된 용도로만 사용되는 특수목적용 레지스터들과 적은 수의 일반목적용 레지스터들만이 포함됨.
- 제어 유니트는 프로그램 코드(명령어)를 해석하고, 그것을 실행하기 위한 제어 신호들을 순차적으로 발생하는 하드웨어 모듈임. 즉 명령어 실행에 필요한 각종 정보들의 전송 통로와 방향을 지정해주고, CPU 내부 요소들과 시스템 구성 요소들의 동작 시간도 결정해 줌. CPU가 제공하는 명령어들의 수가 많아질수록 제어 유니트의 내부 회로는 더 복잡해 짐.

# CPU의 기본 구조 (3)



- 이와 같은 복잡도를 줄이기 위해 제어 유니트의 동작을 소프트웨어로 처리해 주는 방법이 바로 마이크로프로그래밍(microprogramming)임.
- 그러나 이 방법을 이용하면 명령어 실행 시간이 길어지기 때문에 최근에는 명령어의 수를 가능한 한 줄이고 명령어 형식을 단순화함으로써 하드웨어만으로 명령어를 실행할 수 있도록 하는 방식이 많이 사용되고 있음.

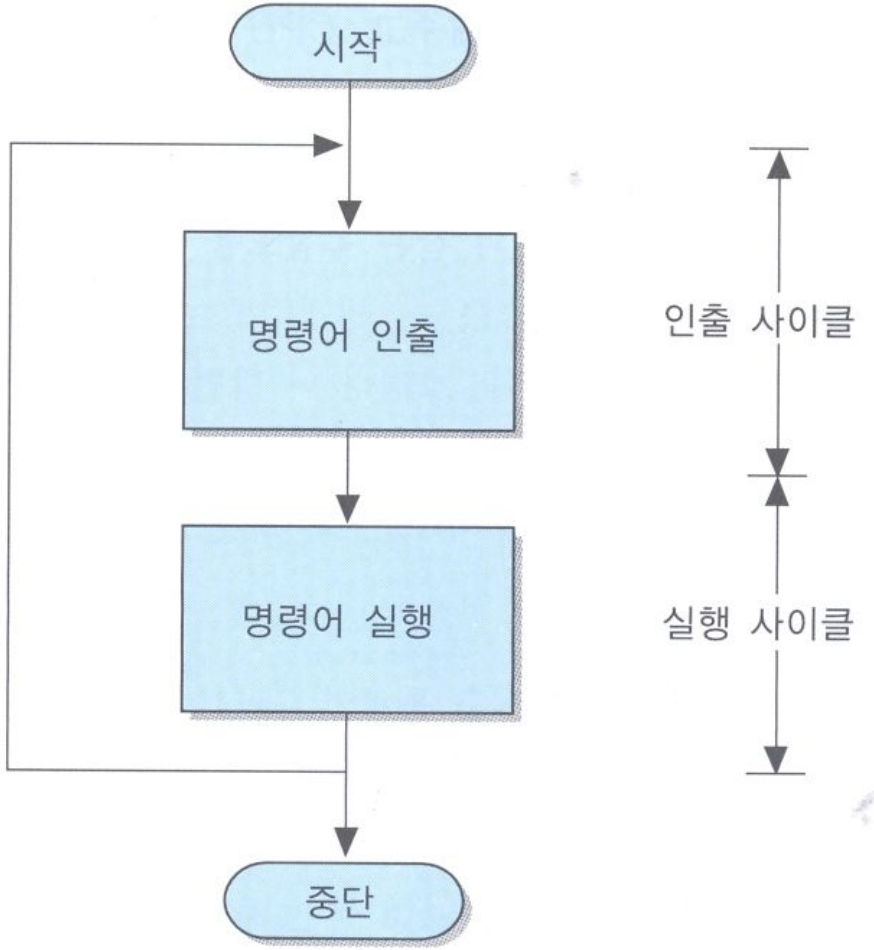
# 명령어 실행 (1)



- CPU는 기억장치에 저장되어 있는 명령어들을 인출하여 실행함으로써 실제적인 작업을 수행하게 됨.
- 명령어 처리 과정은 CPU가 기억장치로부터 한 번에 한 개씩 명령어를 읽어오는 명령어 인출(instruction fetch) 단계와 그것을 수행하는 명령어 실행(instruction execution) 단계로 이루어짐.
- 이와 같이 한 개의 명령어를 실행하는 데 필요한 전체 처리 과정을 명령어 사이클 (instruction cycle)이라고 부르고, 위에서 설명한 두 단계를 각각 부사이클(sub cycle)로 구분하여 인출 사이클(fetch cycle)과 실행 사이클(execution cycle)이라고 부름.

# 명령어 실행 (2)






• 기본 명령어 사이클

# 명령어 실행 (3)



• 명령어 실행을 위해 기본적으로 필요한 CPU 내부 레지스터들은 아래와 같음.

* 프로그램 카운터(PC : Program Counter) : 다음에 인출할 명령어의 주소를 가지고 있는 레지스터임. 각 명령어가 인출된 후에는 그 내용이 자동적으로 1 혹은 명령어 길이(바이트 수)만큼 증가되며, 분기(branch) 명령어가 실행되는 경우에는 그 목적지 주소로 갱신됨.
* 누산기(AC : Accumulator) : 데이터를 일시적으로 저장하는 레지스터임. 이 레지스터의 비트 수는 그 CPU가 한 번에 처리할 수 있는 데이터 비트 수, 즉 단어 길이와 같음.
* 명령어 레지스터(IR : Instruction Register) : 가장 최근에 인출된 명령어 코드가 저장되어 있는 레지스터
* 기억장치 주소 레지스터(MAR : Memory Address Register) : PC에 저장된 명령어 주소가 시스템 주소 버스로 출력되기 전에 일시적으로 저장되는 주소 레지스터임.
* 기억장치 버퍼 레지스터(MBR : Memory Buffer Register) : 기억장치에 쓰여질 데이터 혹은 기억장치로부터 읽혀진 데이터를 일시적으로 저장하는 버퍼 레지스터임.

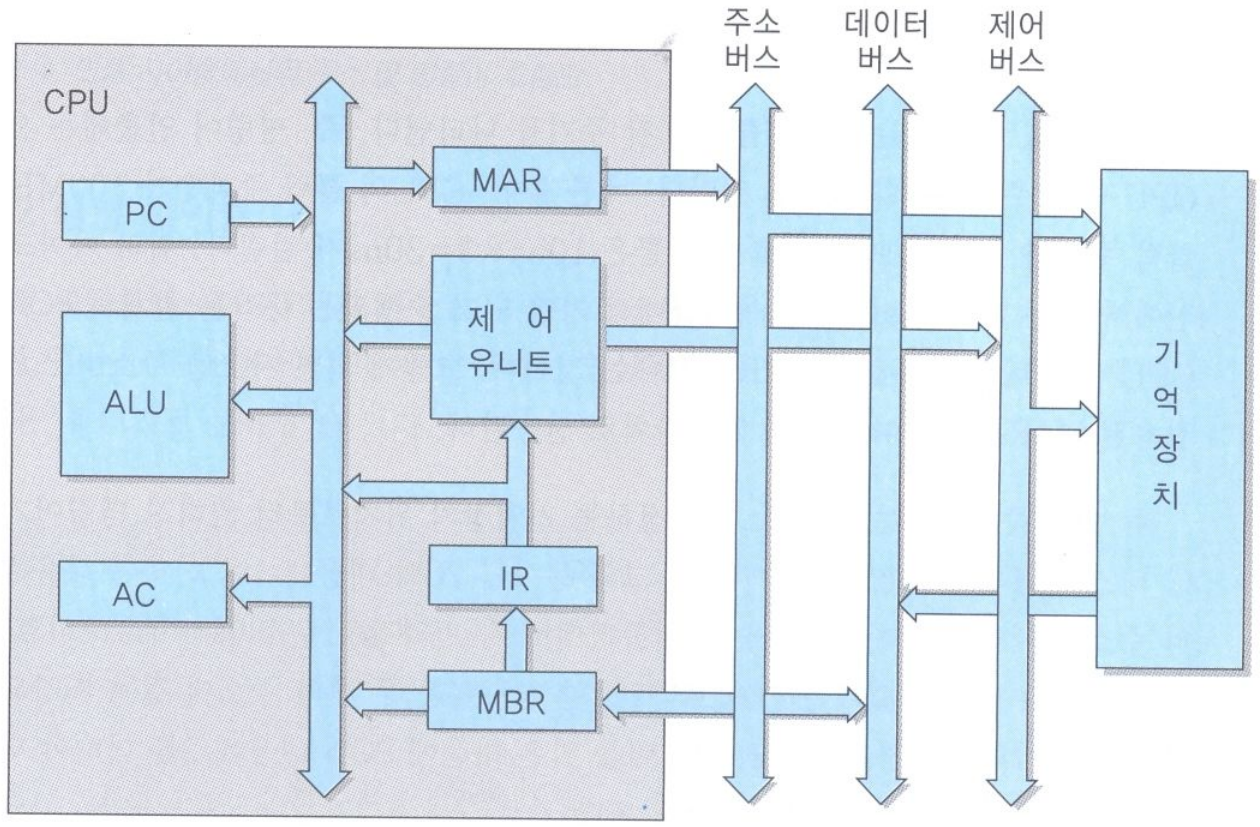
# 명령어 실행 (4)



- 기억장치로부터 인출된 명령어는 MBR을 경유하여 IR에 저장되며,
  실행 사이클에서 제어 유니트로 보내짐.
- MAR은 CPU 내부 주소 버스와 시스템 주소 버스 사이에서
  버퍼 역할을 하는 레지스터이며,
  MBR은 데이터에 대하여 동일한 역할을 하는 버퍼 레지스터임.
- 기억장치로부터 읽혀온 데이터는 MBR을 통해 AC로 적재되는데,
  만약 명령어가 그 데이터에 대해 산술 혹은 논리 연산을 수행하는 것이라면,
  AC의 내용이 ALU로 보내짐.
  그리고 ALU의 연산 결과는 다시 AC에 저장됨.

# 명령어 실행 (5)






• 주요 레지스터들과 데이터 통로가 표시된 CPU 구조

# 2 교시

# 명령어 실행 (6)



(1) 인출 사이클

- 인출 사이클에서 각 단계별로 수행되는 동작을 마이크로-연산(micro-operation)으로 표현하면 다음과 같음.

  $t_0$ : MAR ← PC

  $t_1$ : MBR ← M[MAR], PC ← PC + 1

  $t_2$ : IR ← MBR

  여기에서 $t_0$ , $t_1$ , $t_2$ 는 CPU 클럭의 각 주기를 나타냄.

  즉, 명령어 인출에는 세 개의 CPU 클럭 주기만큼의 시간이 걸림.

  예를 들어서 CPU의 클럭 주파수가 100MHz라면 클럭 주기가 10ns이므로,

  인출 사이클은 10ns x 3 = 30ns가 걸림.

- 위의 마이크로-연산에서 보는 바와 같이, 인출 사이클에서 가장 먼저 수행되는 동작은 현재의 PC 내용을 CPU 내부 버스를 통해 MAR로 보내는 것임.

  그렇게 되면 시스템 주소 버스와 직접 접속된 MAR를 통해 주소가 기억장치로 전송됨.

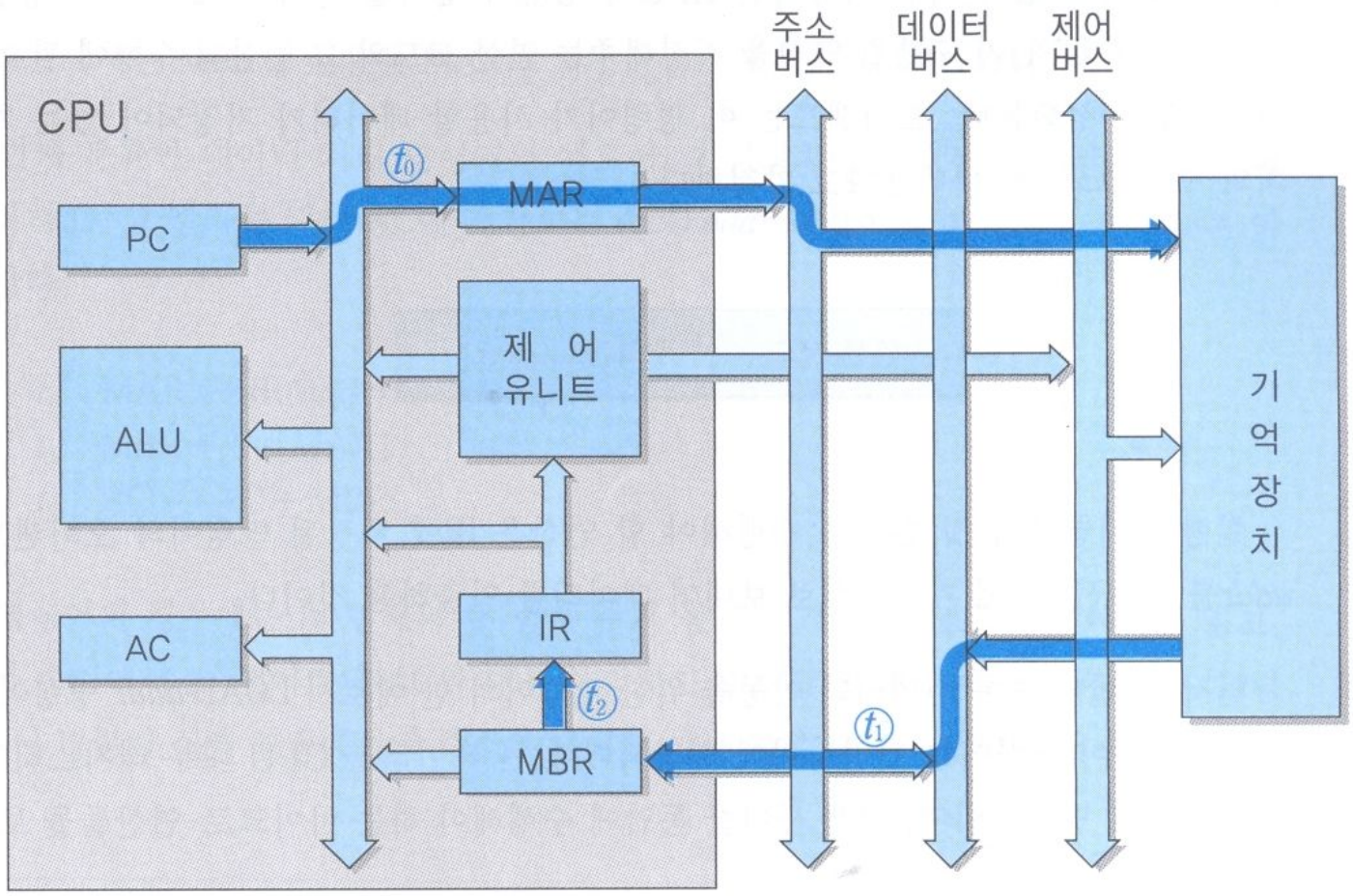
# 명령어 실행 (7)



- 두 번째 주기에서는 그 주소가 지정하는 기억장치 위치로부터 읽혀진 명령어가
  데이터 버스를 통해 MBR로 적재되며,
  그와 동시에 PC의 내용에 1을 더하여 다음 명령어의 주소를 가리키게 됨.
  만약 이 컴퓨터에서 각 기억장치 주소가 바이트 단위로 지정되고
  명령어 길이는 16비트라면,
  한 명령어는 두 개의 주소에 걸쳐 저장되는 것이므로,
  이 경우에는 한 명령어를 읽은 다음에 PC의 내용에 2를 더해야
  다음 명령어가 저장된 위치를 지정할 수 있게 됨.
- 마지막 세 번째 주기에서는
  MBR에 저장되어 있는 명령어 코드가 명령어 레지스터인 IR로 이동됨.

# 명령어 실행 (8)






• 인출 사이클에서의 주소 및 명령어 흐름도

# 명령어 실행 (9)



(2) 실행 사이클

• 실행 사이클에서는 CPU가 인출된 명령어 코드를 해독(decode)하고,
그 결과에 따라 필요한 연산을 수행함.

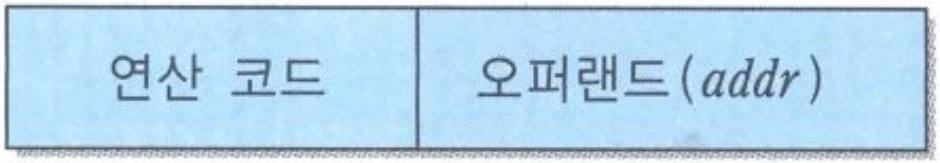
CPU가 수행하는 연산들은 아래와 같이 크게 네 가지로 분류할 수 있음.

* 데이터 이동 : CPU와 기억장치 간 혹은 CPU와 I/O 장치 간에 데이터를 이동함.
* 데이터 처리 : 데이터에 대하여 산술 혹은 논리 연산을 수행함.
* 데이터 저장 : 연산 결과 데이터 혹은 입력장치로부터 읽어 들인 데이터를 기억장치에 저장함.
* 제어 : 프로그램의 실행 순서를 결정함.

# 명령어 실행 (10)



- 명령어는 아래와 같이 CPU가 수행할 연산을 지정해 주는 연산 코드와 그 연산의 수행에 필요한 오퍼런드로 구성되는데, 오퍼런드는 이 명령어가 사용할 데이터가 저장되어 있는 기억장치의 주소(addr)를 나타낸다고 가정함.




- 먼저 첫 번째 분류인 데이터 이동을 위한 명령어의 한 예로서 LOAD addr 명령어의 경우를 보자. 이 명령어는 기억장치에 저장되어 있는 데이터를 CPU 내부 레지스터인 AC로 이동하는 명령어이며, 실행 사이클 동안에 수행해야 하는 마이크로-연산들은 아래와 같음.

  $t_0$ : MAR ← IR(addr)

  $t_1$ : MBR ← M[MAR]

  $t_2$ : AC ← MBR

# 명령어 실행 (11)



- 첫 번째 주기에서는 명령어 레지스터 IR에 있는 명령어의 오퍼런드(주소) 부분을 MAR로 보냄. 두 번째 주기에서는 그 주소가 지정한 기억장소로부터 데이터를 인출하여 MBR로 전송하며, 그 데이터를 세 번째 주기 동안에 AC에 적재함으로써 LOAD 명령어의 실행이 완료됨.
- 두 번째 분류인 데이터 저장을 위한 명령어의 예로서, AC 레지스터의 내용을 기억장치에 저장하는 명령어인 STA addr 명령어의 실행 사이클은 다음과 같은 마이크로-연산들로 이루어짐.

  $t_0$ : MAR ← IR(addr)

  $t_1$ : MBR ← AC

  $t_2$ : M[MAR] ← MBR

  여기에서도 첫 번째 주기에서는 데이터를 저장할 기억장치의 주소를 MAR로 보냄.
  다음 주기에서는 저장할 데이터를 버퍼 레지스터인 MBR로 이동시키고, 마지막 세 번째 주기에서는 MBR의 내용을 MAR이 지정하는 기억 장소에 저장함.

# 명령어 실행 (12)



- 세 번째 분류인 데이터 처리 명령어의 예로서, 기억장치에 저장된 데이터를 AC의 내용과 더하고, 그 결과는 다시 AC에 저장하는 ADD addr 명령어의 마이크로-연산들은 다음과 같음.

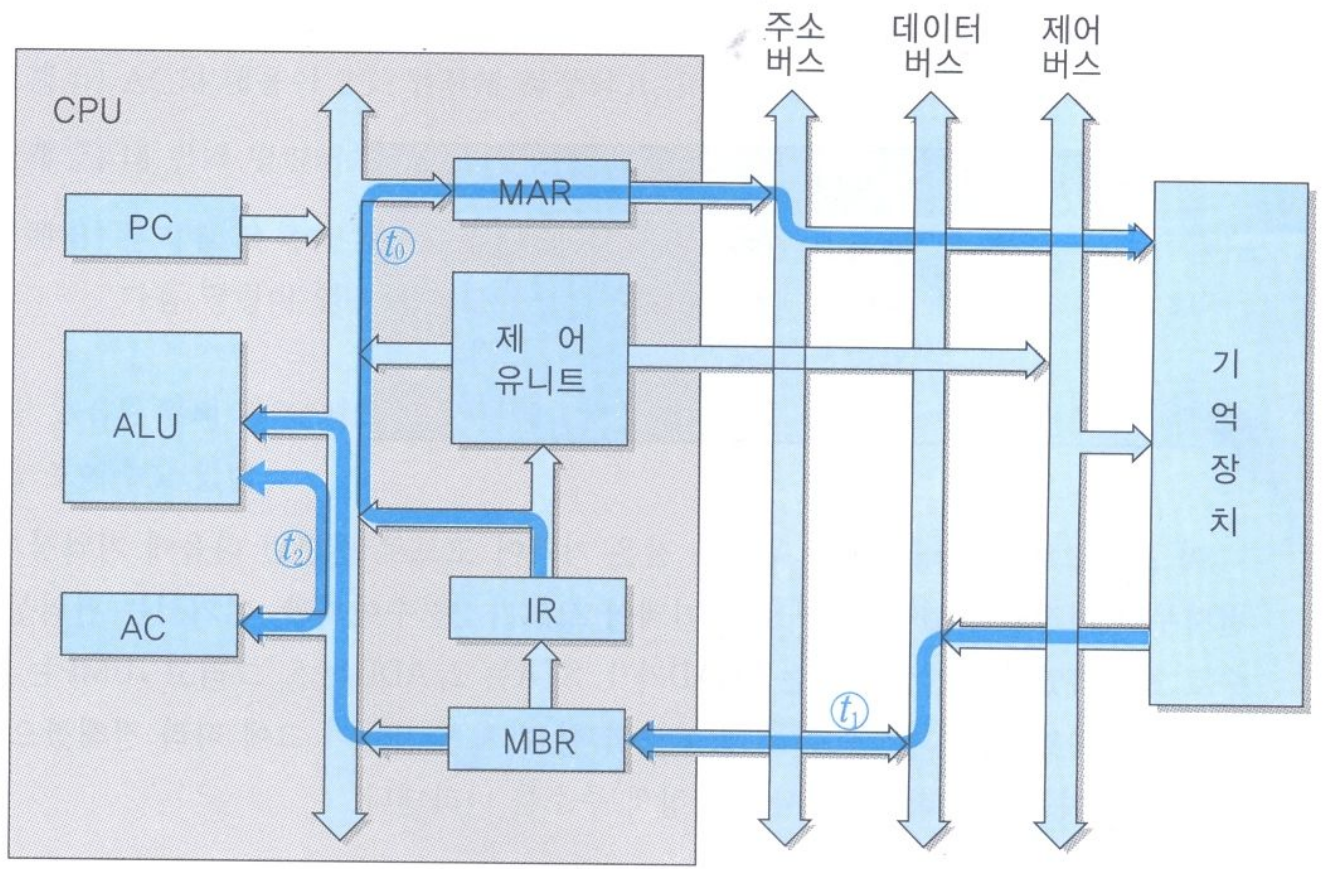
  $t_0$ : MAR ← IR(addr)

  $t_1$ : MBR ← M[MAR]

  $t_2$ : AC ← AC + MBR

# 명령어 실행 (13)






• ADD 명령어 실행 사이클에서의 주소 및 데이터 흐름도

# 명령어 실행 (14)



- 마지막으로, 네 번째 분류인 제어 명령어의 예를 살펴 보자. 일반적으로 명령어들은 기억장치에 저장되어 있는 순서대로 실행됨.
  그러한 경우에는 명령어 실행 순서를 결정하기 위하여 별도의 명령어가 필요하지 않고.
  프로그램 카운터(PC)에 의해 자동적으로 순서가 결정됨.
  그러나 그와 같은 순차적 실행이 아닌,
  전혀 다른 위치에 있는 명령어를 실행해야 하는 경우도 있음.
  그러한 경우를 위하여 현재의 PC 내용이 가리키는 위치가 아닌 다른 위치의 명령어로 실행 순서를 바꾸도록 해 주는 명령어들이 있음. 이러한 명령어들을 분기(branch) 명령어라고 부르는데, 그 예로서 JUMP addr 명령어는 다음과 같이 한 주기만에 실행이 완료됨.
  $t_0$ : PC ← IR(addr)
  즉, 이 명령어가 실행되면 명령어의 오퍼런드(분기할 목적지 주소)가 PC로 적재됨.
  그렇게 되면, 다음 명령어 인출 사이클에서 그 주소가 가리키는 위치의 명령어가 인출되므로 분기가 일어나게 됨.

# 명령어 실행 (15)

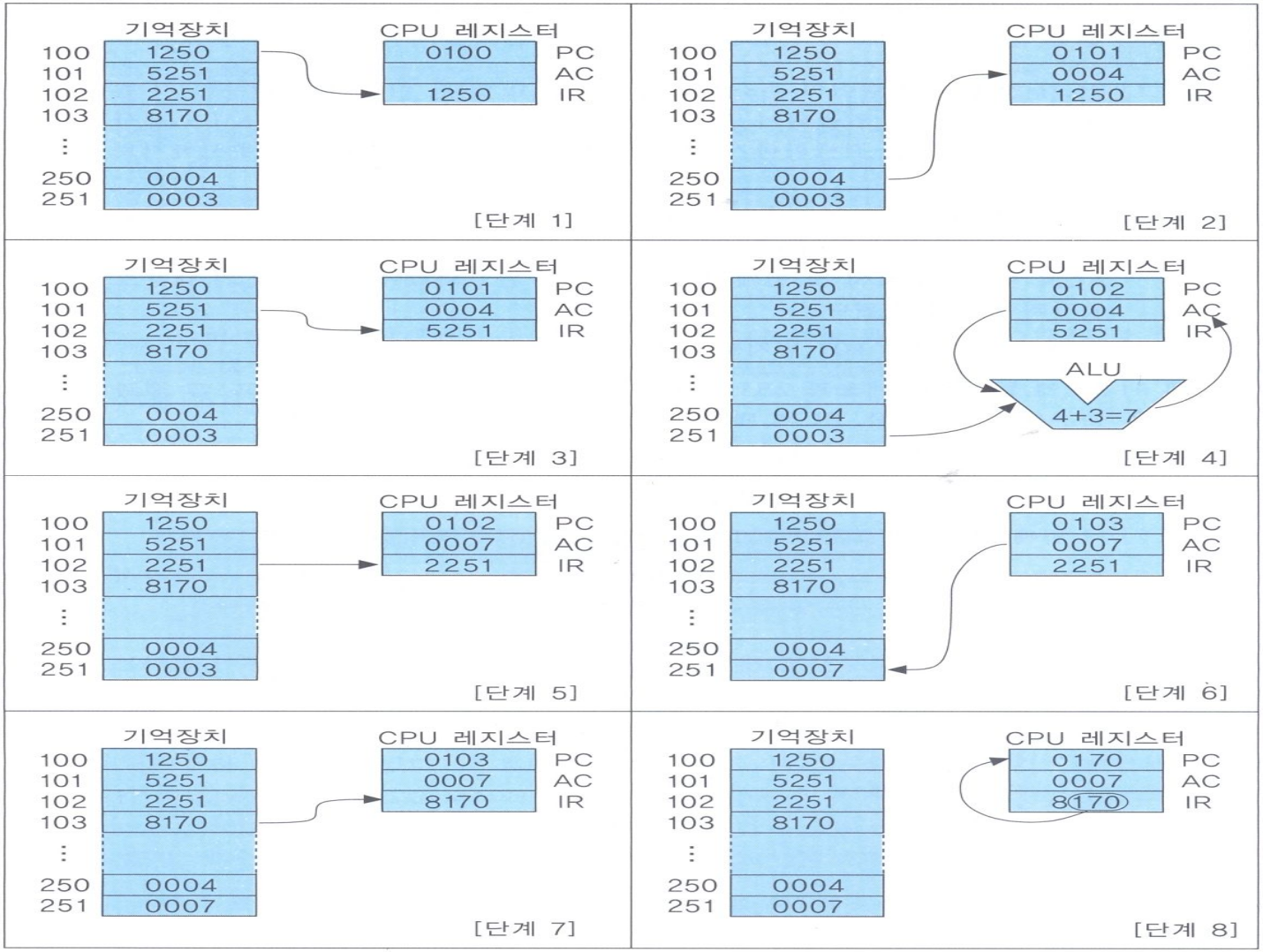


| 주소 | 명령어 | | 기계 코드 |
|---|---|---|---|
| 100 | LOAD | 250 | 1250 |
| 101 | ADD | 251 | 5251 |
| 102 | STA | 251 | 2251 |
| 103 | JUMP | 170 | 8170 |

- 상기 프로그램의 명령어들은 우측과 같이 기계어 코드로 변환된 다음에
  기억장치 100번지부터 저장되어 있음.
  실제로는 기계어 코드가 2진수인데, 여기에서는 편의상 10진수로 표시하였음.
  연산 코드는 LOAD가 1, STA는 2, ADD는 5, 그리고 JUMP는 8인 것으로 가정하였음.

# 명령어 실행 (16)






서울사이버대학교 SEOUL CYBER UNIVERSITY

- 프로그램 실행 과정의 예

# 3 교시

# 명령어 실행 (17)

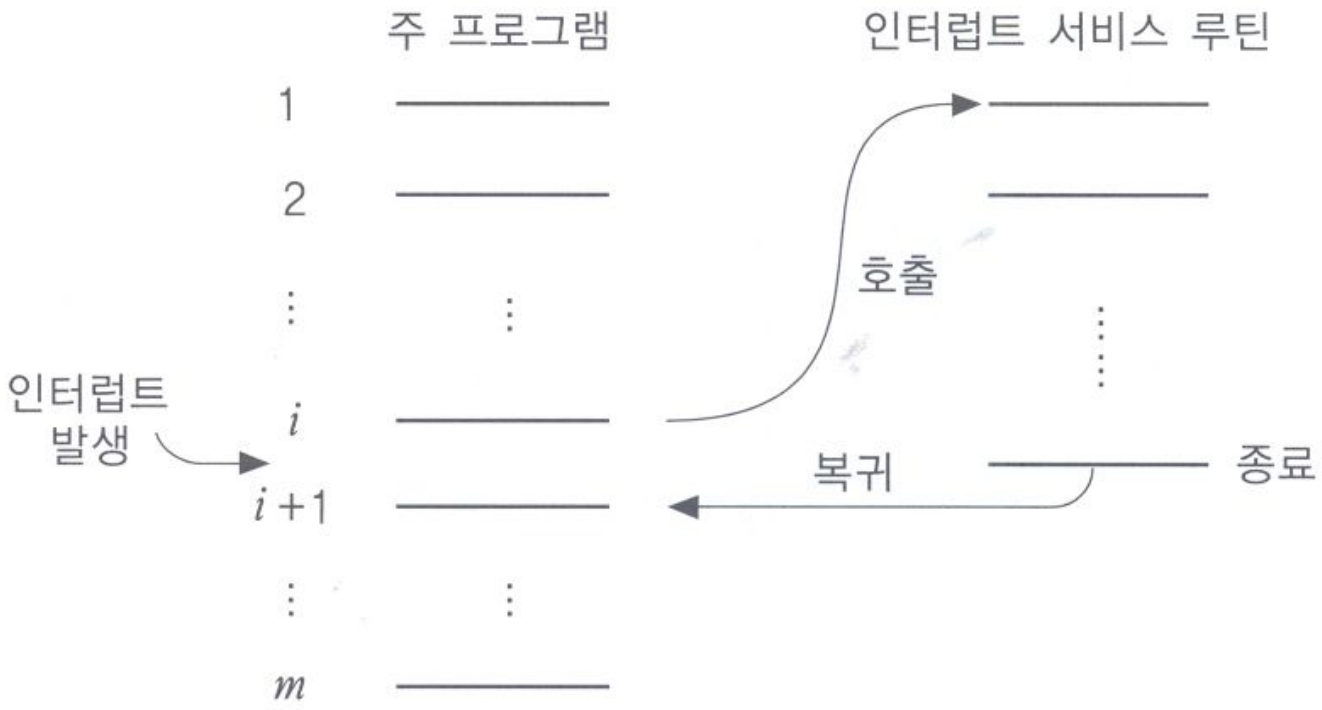


(3) 인터럽트 사이클

- 대부분의 컴퓨터들은 프로그램 실행 중에 CPU의 정상적인 처리를 방해,
  즉 인터럽트 (interrupt)할 수 있는 메커니즘을 제공함.
- CPU가 어떤 프로그램을 순차적으로 수행하는 도중에
  외부로부터 인터럽트 요구가 들어 오면,
  CPU는 원래의 프로그램 수행을 중단하고,
  요구된 인터럽트를 위한
  서비스 프로그램을 먼저 수행하게 되는데,
  그러한 서비스 프로그램을 인터럽트 서비스 루틴(interrupt service routine: ISR)이라고 부름.
  그리고, 인터럽트에 대한 처리가 끝나면,
  원래의 프로그램으로 복귀(return)하여 수행을 계속함.

# 명령어 실행 (18)






• 인터럽트에 의한 제어의 이동

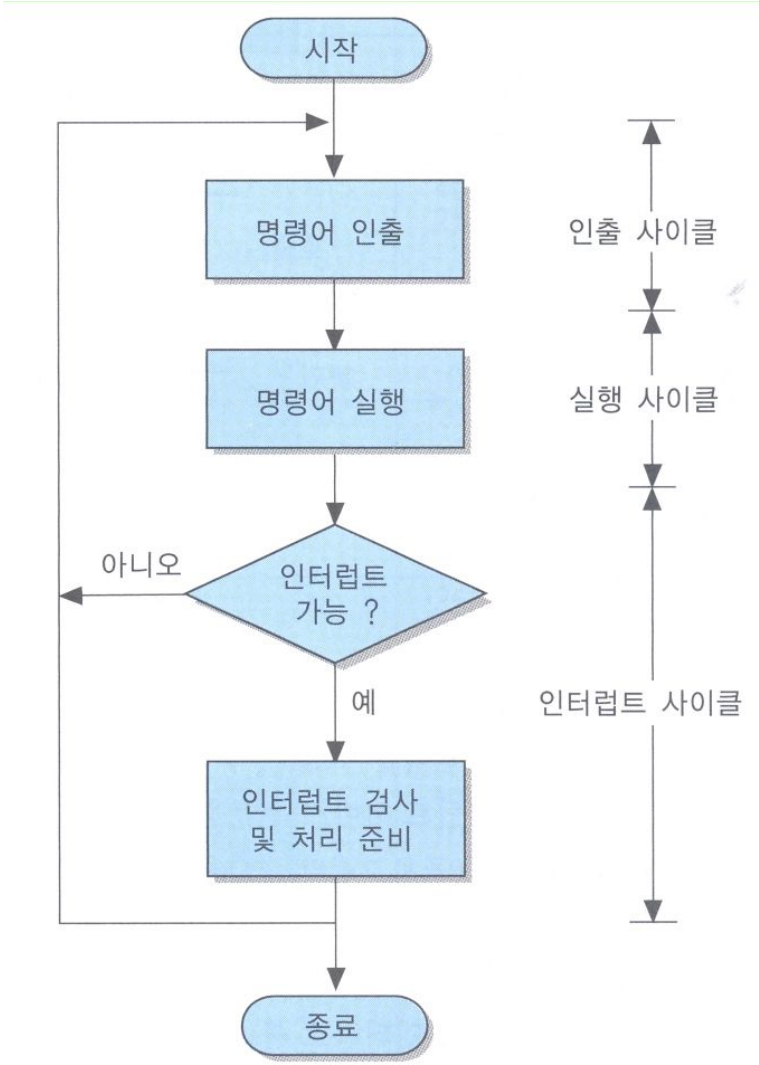
# 명령어 실행 (19)



- 인터럽트가 들어왔을 때 CPU는 그것을 인식할 수 있어야 하며, (1) 어떤 장치가 인터럽트를 요구하였는지 확인하여 해당 인터럽트 서비스 루틴을 호출하여 수행하고, (2) 서비스가 종료된 다음에는 중단되었던 원래 프로그램의 수행을 계속할 수 있어야 함. 이를 위하여 CPU는 한 명령어의 실행 사이클을 종료하고 다음 명령어를 위한 인출 사이클을 시작하기 전에 인터럽트 요구 신호가 대기 중인 지를 반드시 점검해야 함. 이 과정은 중요한 프로그램 실행을 위해 CPU가 인터럽트 불가능(interrupt disable) 상태로 세트 되어 있는 경우에는 생략함.
- 만약 인터럽트 요구가 들어왔다면 CPU는 다음과 같은 동작을 수행함. (1) 현재의 명령어 실행을 끝낸 즉시, 다음에 실행할 명령어의 주소(PC의 내용)를 스택(stack)에 저장함. 일반적으로 스택은 주기억장치의 특정 부분이 됨. (2) 인터럽트 서비스 루틴을 호출하기 위해 그 루틴의 시작 주소를 PC에 적재함. 이때 시작 주소는 인터럽트를 요구한 장치로부터 전송되거나 미리 정해진 값으로 결정됨.

# 명령어 실행 (20)








- 인터럽트 사이클이 추가된 명령어 사이클

# 명령어 실행 (21)



- 인터럽트 요구가 들어온 경우에 인터럽트 사이클 동안 수행되는 동작들을 마이크로-연산으로 표현하면 다음과 같음.

  $t_0$ : MBR ← PC

  $t_1$ : MAR ← SP, PC ← ISR의 시작 주소

  $t_2$ : M[MAR] ← MBR

  여기에서 SP는 CPU 내부에 있는 특수목적용 레지스터들 중의 하나인 스택 포인터(stack pointer)를 의미하는데, 그 내용은 항상 스택의 최상위의 주소를 가리킴.

  첫 번째 주기에서 PC의 내용이 MBR로 전송됨.

  두 번째 주기에서는 SP의 내용이 MAR로 전송되며,

  PC의 내용은 인터럽트 서비스 루틴의 시작 주소로 바뀌게 됨.

  마지막 주소에서는 MBR에 저장되어 있던 원래 PC의 내용이 스택에 저장됨.

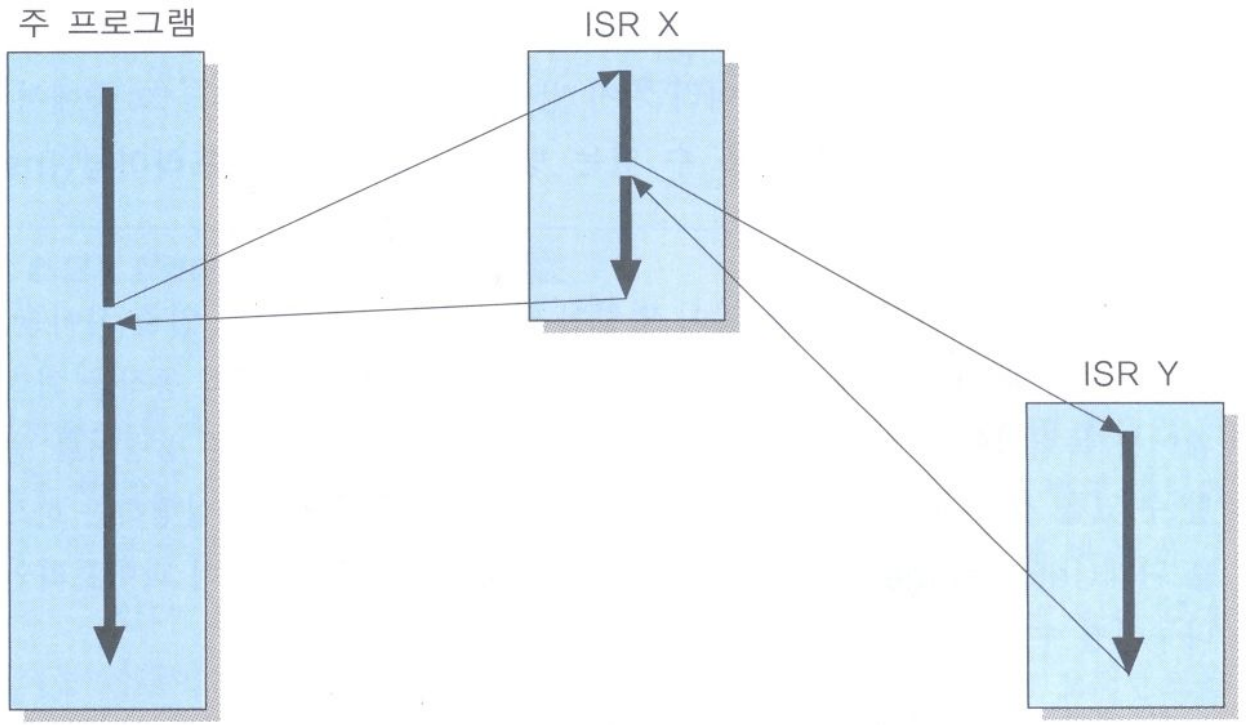
# 명령어 실행 (22)



- 인터럽트 서비스 루틴의 명령어들을 실행하는 것도
  일반적인 명령어 실행 과정과 동일하기 때문에
  그 동안에 다른 인터럽트가 발생할 수도 있는데 이것을 다중 인터럽트라 함.
- 다중 인터럽트를 처리하는 방법으로는 두 가지가 있는데, 첫 번째 방법은 CPU가
  인터럽트 서비스 루틴을 처리하고 있는 도중에는 새로운 인터럽트 요구가 들어오더라도
  CPU가 인터럽트 사이클을 수행하지 않도록 하는 것인데, 이는 인터럽트 서비스 루틴에서
  인터럽트 불가능(interrupt disable) 상태로 세트 시키면 됨.
  이렇게 되면 그 루틴을 처리하는 동안에 발생한 인터럽트 요구는 대기 상태로 남아 있다가,
  CPU가 다시 인터럽트 가능(interrupt enable) 상태로 바뀐 후에 처리됨.
  두 번째 방법은 인터럽트의 우선 순위를 정하고, 우선 순위가 낮은 인터럽트가 처리되고
  있는 동안에 우선 순위가 더 높은 인터럽트가 들어오면 현재의 인터럽트 서비스 루틴의
  수행을 중단하고 새로운 인터럽트를 처리하도록 하는 것임.

# 명령어 실행 (23)






• 다중 인터럽트에서 프로그램이 실행되는 순서

# 명령어 파이프라이닝 (1)

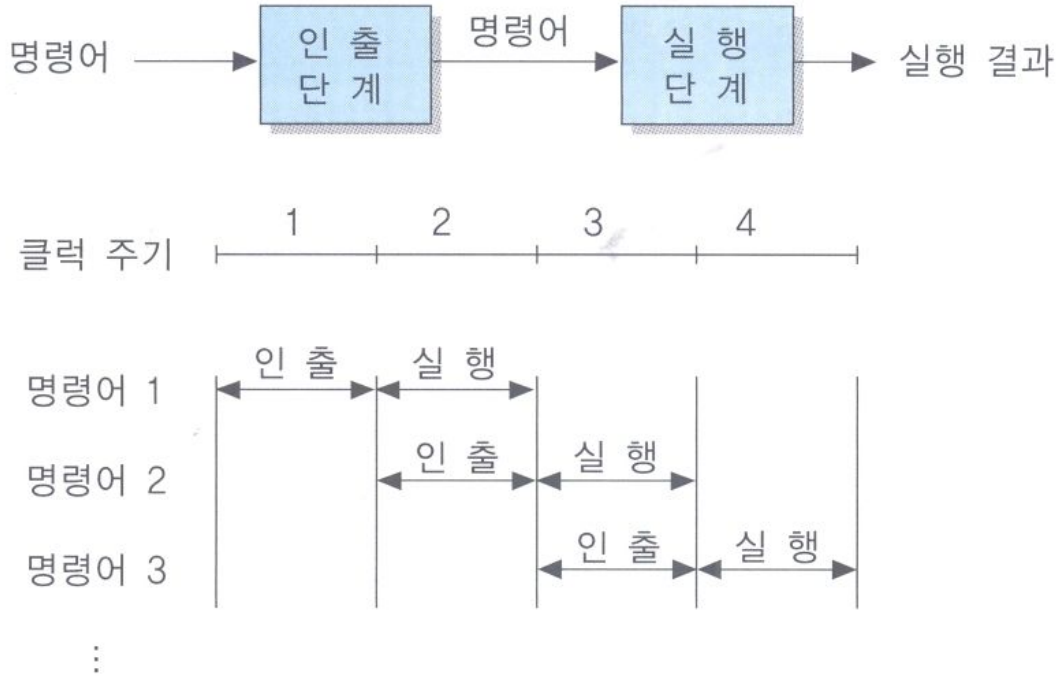


- CPU의 속도는 컴퓨터시스템의 프로그램 처리 시간에 직접 영향을 주기 때문에 CPU의 속도를 향상시키기 위하여 여러 가지 방법들이 사용되고 있는데 그 중에서 가장 간단하면서도 분명한 효과를 얻을 수 있는 방법이 명령어 파이프라이닝(instruction pipelining)임.
- 명령어 사이클은 인출 사이클과 실행 사이클이라는 두 개의 단계로 이루어지는데, 이들 각 사이클의 동작을 처리하는 하드웨어를 독립적인 모듈로 구성할 수 있다면, 각 모듈이 서로 다른 명령어를 동시에 처리할 수 있을 것임.
- 두 파이프라인 단계들에 동일한 클럭을 가하면 그 단계들의 동작 시간을 일치시킬 수 있음.
  * 첫 번째 클럭 주기 동안에 인출 단계가 첫 번째 명령어를 인출함.
  * 두 번째 주기에서는 그 명령어가 실행 단계로 보내져서 실행되며, 그와 동시에 인출 단계는 두 번째 명령어를 인출함.
  * 세 번째 주기에서는 두 번째 명령어의 실행과 세 번째 명령어의 인출이 동시에 이루어짐.

# 명령어 파이프라이닝 (2)



- 이와 같이 명령어 실행 하드웨어를 두 단계로 분리시킨 것을 2-단계 명령어 파이프라인이라고 부름.
- 만약 인출과 실행에 같은 길이의 시간이 걸린다면, 파이프라인을 이용하여 명령어 처리 속도를 두 배 높일 수 있지만, 실제로 실행 단계의 처리시간이 인출 단계보다 더 김.




• 2-단계 명령어 파이프라인과 시간 흐름도

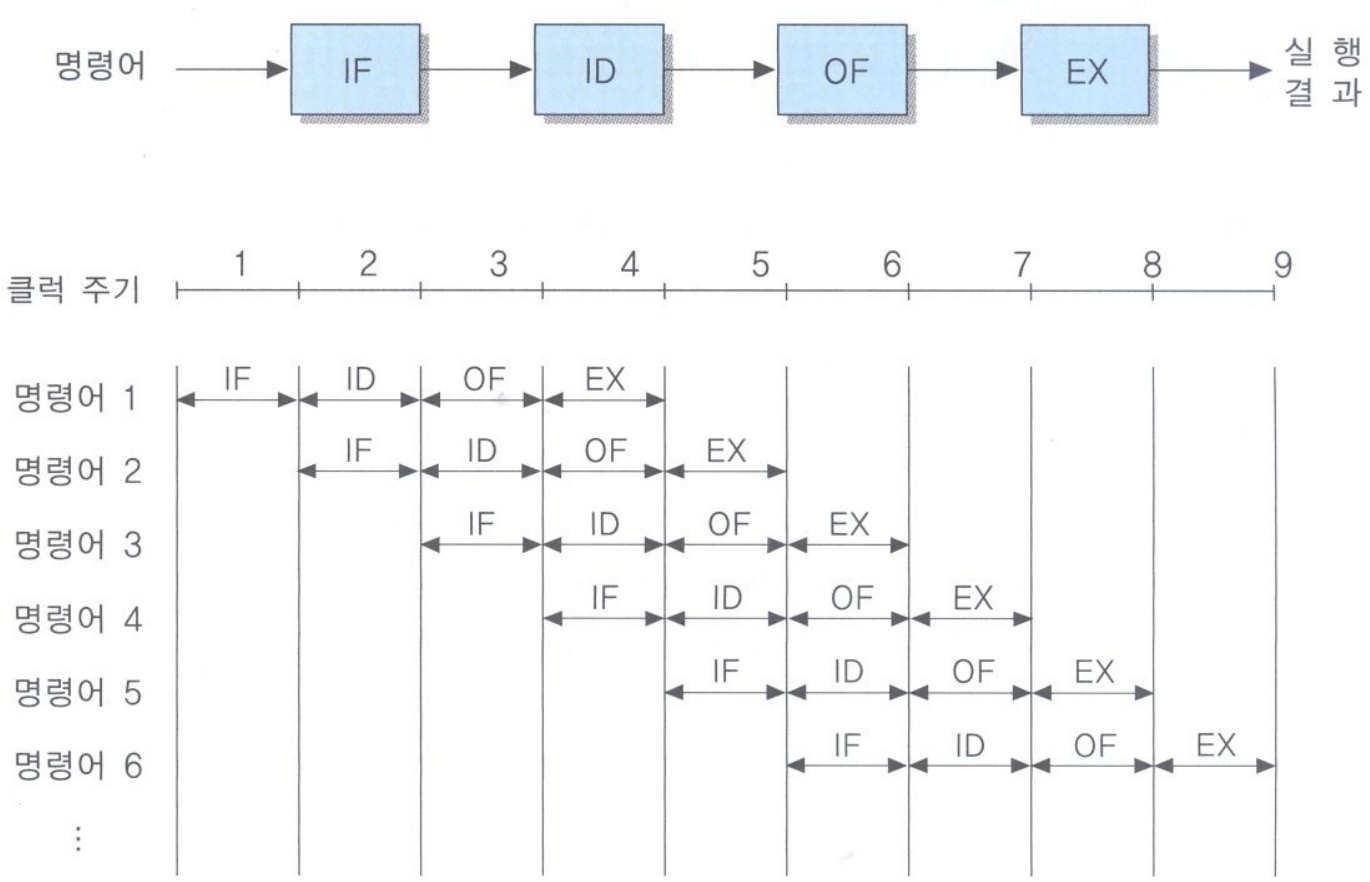
# 명령어 파이프라이닝 (3)



- 파이프라인 단계들의 처리 시간이 동일하지 않음으로 인해 발생하는 효율 저하를 방지하는 방법은 처리 시간이 더 긴 파이프라인 단계를 더 작게 분리하여 거의 같아지도록 하는 것임.
- 이와 같이 파이프라인 단계의 수를 늘리면 전체적으로 속도 향상이 더 높아짐.
- 앞에서 살펴본 2-단계 파이프라인의 실행 단계를 더 분리하여 다음과 같이 파이프라인을 네 단계로 구성하는 경우를 고려해 볼 수 있음.
  * 명령어 인출(IF) : 다음 명령어를 기억장치로부터 인출함.
  * 명령어 해독(ID) : 해독기(decoder)를 이용하여 명령어를 해석함.
  * 오퍼런드 인출(OF) : 기억장치로부터 오퍼런드를 인출함.
  * 실행(EX) : 지정된 연산을 수행함.

# 명령어 파이프라이닝 (4)






• 4-단계 명령어 파이프라인과 시간 흐름도

# 셀프 테스트



• CPU 내부에 위치한 기억 장치는 무엇인가?

* 레지스터

해설) 레지스터는 플립플롭으로 구성되며 CPU 내부에 위치하여 속도가 가장 빠른 기억장치에 해당함.

• 인출 사이클에서 CPU 내부 버스를 통해 현재의 PC 내용이 저장되는 레지스터는 무엇인가?

* MAR

해설) MAR은 메모리 주소 레지스터로서 CPU 외부로 나가는 주소버스의 내용을 저장하는 레지스터이므로 프로그램 카운터도 MAR에 저장됨.

• 명령어 사이클의 각 사이클의 동작을 처리하는 하드웨어를 독립적인 모듈로 구성하여 각 모듈이 서로 다른 명령어를 동시에 처리함으로써 CPU 속도를 향상시키는 방법을 무엇이라고 하는가?

* 명령어 파이프라이닝

해설) 명령어 파이프라이닝은 각 사이클의 동작을 구분하여 모듈을 형성하고 각 모듈을 동시에 처리함으로써 CPU 속도를 증가시킬 수 있는 방법임.

# 요점 정리



- CPU는 산술논리연산장치(ALU : Arithmetic and Logical Unit), 레지스터 세트, 제어 유니트 등으로 구성됨.
- 명령어 처리 과정은 CPU가 기억장치로부터 한 번에 한 개씩 명령어를 읽어오는 명령어 인출(instruction fetch) 단계와 그것을 수행하는 명령어 실행(instruction execution) 단계로 이루어짐.
- 명령어 실행을 위해 기본적으로 필요한 CPU 내부 레지스터에는 PC, AC, MAR, IR, MAR, MBR 등이 있음.
- 대부분의 컴퓨터들은 프로그램 실행 중에 CPU의 정상적인 처리를 방해, 즉 인터럽트(interrupt)할 수 있는 메커니즘을 제공함.
- CPU의 속도는 컴퓨터시스템의 프로그램 처리 시간에 직접 영향을 주기 때문에 CPU의 속도를 향상시키기 위하여 여러 가지 방법들이 사용되고 있는데 그 중에서 가장 간단하면서도 분명한 효과를 얻을 수 있는 방법이 명령어 파이프라이닝(instruction pipelining)임.