2주차

제2장 CPU의 구조와 기능

2.0 들어가며

CPU는 기본적으로 명령어를 실행하게 되어있음. 그에 따른 구조가 필요함

어떻게 명령어를 효율적으로 빠르게 실행하는지에 대한 파이프라이닝이라는 기술이 있음

명령어 세트의 예시를 보는 순서대로 2장의 내용이 진행될 예정.

CPU의 명령어 사이클

: 명령어를 실행하기 위해 반복하는 사이클이 있다, 5가지가 존재 – 모든 명령어에 5가지가 전부 필요한 것은 아님

명령어 인출 – 메모리에 있던 명령어를 가져옴

명령어 해독 – 수행해야 할 명령어 자체를 분석함

데이터 인출 – 데이터 자체가 기억장치나 I/O장치에 존재하기 때문에 가져오는 단계

데이터 처리 – 데이터를 CPU 내로 가져왔을 때 산술, 논리적인 연산을 실행하는 단계

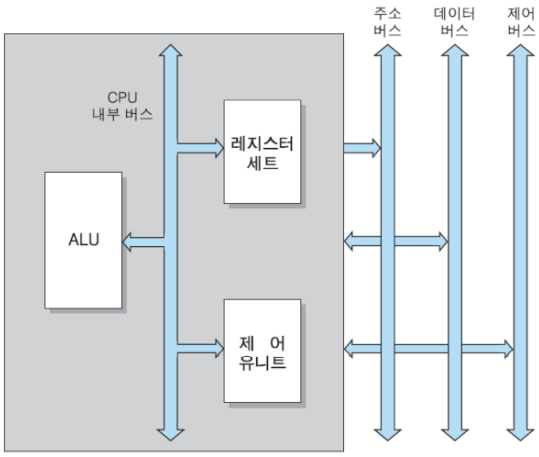
데이터 저장 – 말 그대로 저장

→ 이 과정이 반복되면서 명령어 하나하나씩 수행하는 것임.

2.1 CPU의 기본 구조

크게 산술논리연산장치(ALU), 레지스터 세트, 제어 유니트가 있음.

CPU 내부에도 내부 버스가 존재함.



산술논리연산장치(ALU) – 덧셈, and같은 계산을 함

레지스터 세트 – 메모리, I/O장치에서 데이터를 받아오는데 시간이 걸리는데, 이 때 레지스터에 임시로 데이터를 가져오거나 내보낼 것을 놔두었다 내보낸다. 내부와 외부의 속도 차이를 극복하기 위해 존재한다고 생각하면 될 듯.

제어 유니트 – 전체적으로 어떻게 실행되어야 하는지 제어하는 역할.

CPU 내부 버스 – 내부 구성원끼리 데이터를 주고받는 기능을 하게 해줌.

ALU: 각종 산술, 논리 연산을 하는 회로들로 이루어진 하드웨어 모듈

산술 연산: +, -, ×, ÷

논리 연산: AND, OR, NOT, XOR 등

레지스터: 액세스 속도가 빠른 기억장치, 일종의 메모리 → CPU 내에서 데이터를 주고받을 때 사용하는 메모리

CPU 내에 포함 가능한 레지스터의 수가 제한됨. 고가라는 측면 + CPU 내에서 특정 레지스터를 지정하는 방법이 있어야 하는데, 개수가 많으면 지정하는데도 문제가 생겨서 많으면 안됨.

레지스터의 종류: 특수목적용 레지스터와 일반목적용 레지스터

제어 유니트: 명령어를 해석하고 실행하기 위해 제어신호를 발생시키는 하드웨어 모듈.

CPU 내부 버스: CPU 간의 제어성 같은 것들을 구성하고 연결했을 때 전달하는 통로.

외부 시스템 버스와는 직접 연결되지 않으며, 레지스터 또는 시스템 버스 인터페이스 회로를 통해 연결됨 → 내부 버스와 시스템 버스는 별개임

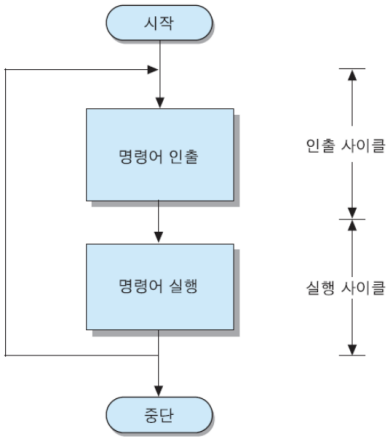
2.2 명령어 실행

명령어 사이클: 명령어 하나를 갖고 와서 실행시키고 다음 거 갖고 와서 실행시키는 과정의 반복. → 하나의 명령어를 실행하는데 필요한 전체 처리 과정, 프로그램 실행 시작 순간부터 전원 끄거나 회복 불가능한 오류 발생 시까지 반복

두 개의 부사이클

인출 사이클: CPU가 메모리로부터 명령어를 읽어와서 내부 레지스터에 저장하는 단계

실행 사이클: 명령어를 실행하는 단계; 명령에 따라 여러 개로 나뉨



명령어 실행에 필요한 CPU 레지스터들

프로그램 카운터(PC): 명령어가 실행 시에 차례대로 실행되므로 다음 명령어가 어디 있는지에 대한 주소를 가지고 있는지에 대한 레지스터

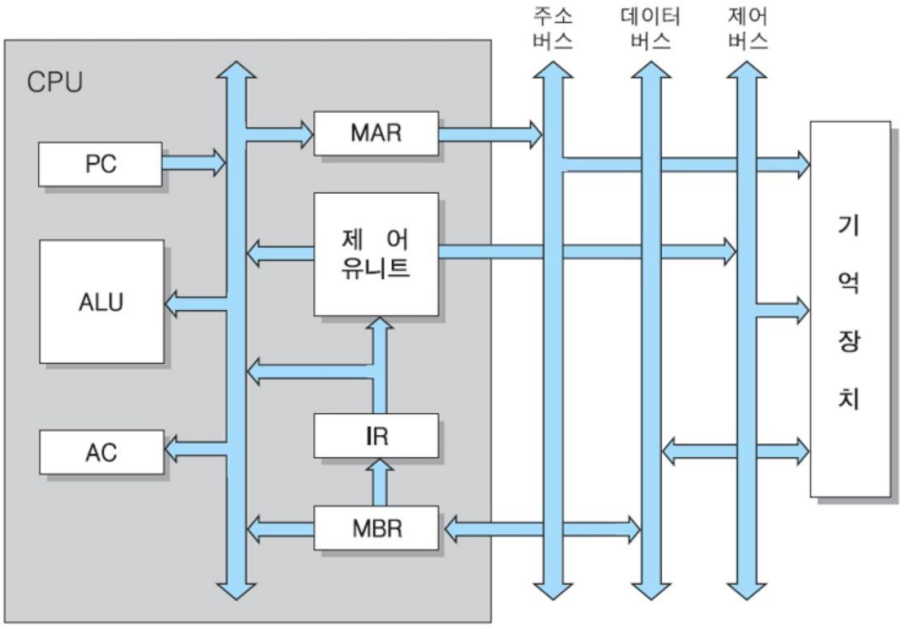
누산기(AC): 계산한 결과 데이터를 일시적으로 저장하는 레지스터

명령어 레지스터(IR): 명령어를 갖고 왔을 때 해석하기 위해 CPU내부에서 가지고 있어야 함, 이 인출된 명령어 코드가 저장되어 있는 레지스터

기억장치 주소 레지스터(MAR): 메모리 주소를 들고 시스템 주소 버스에 출력이 되어야 하는데, 이 데이터를 일시적으로 저장하는 주소 레지스터

기억장치 버퍼 레지스터(MBR): 주소를 통해 기억장치에서 읽어 왔거나 쓸 데이터를 일시적으로 저장하는 레지스터

데이터 통로가 표시된 CPU 내부 구조



2.2.1 인출 사이클

메모리에 있는 명령어를 IR로 옮기는 과정. 단계별 마이크로 연산으로 세분화해서 볼 예정

인출 사이클의 마이크로 연산(t: CPU 클록의 주기)

t_0: MAR ← PC

PC의 내용을 CPU 내부 버스 통해 MAR로 옮김

PC – 명령어가 있는 주소, MAR – 메모리에 있는 내용을 갖고 오겠단 의미

t_1: MBR ← M[MAR], PC ← PC + 1

대괄호: 메모리에서 MAR에 있는 주소의 내용을 MBR로 옮김

→ 제어 장치에서 신호를 옮김

그와 동시에 PC += 1 → 다음 명령어의 주소를 가리킨다

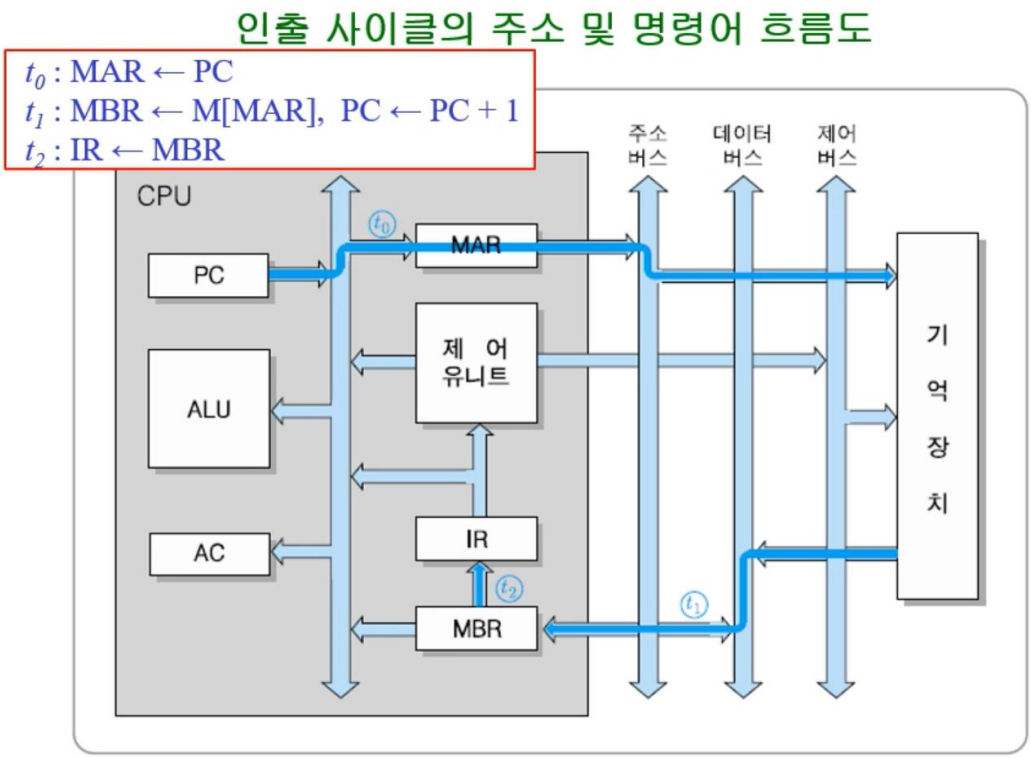
t_2: IR ← MBR

MBR에 있는 명령어 코드를 IR로 이동

예) CPU 클록 주파수 = 1GHz(클럭 주기: 1ns) → 인출 사이클: 1ns * 3 = 3ns

대충 일단 0이 아니라고만 생각하면 된다고 한다

인출 사이클의 주소 및 명령어 흐름도



2.2.2 실행 사이클

인출 사이클에서 메모리 내용을 IR로 옮기면 실행 사이클에서 실행함. 그러기 위해선 명령 코드를 해독해야 하며, 결과에 따라 연산을 수행해야 함.

CPU가 수행하는 연산들의 종류

데이터 이동: CPU와 I/O 또는 기억장치 간 데이터 이동

데이터 처리: 산술, 논리 연산 수행

데이터 저장: 연산 결과 혹은 입력장치로부터 읽어 들인 데이터를 기억장치에 저장

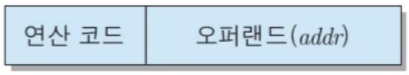
프로그램 제어: 프로그램 실행 순서를 저장 (if문과 같은 것들 실행을 생각하면 편할듯)

실행 사이클에서 수행되는 마이크로 연산들은 명령어의 연산 코드(op-code) 따라 결정됨

기본적인 명령어 형식의 구성

연산 코드: CPU가 수행할 연산을 지정(+같은 것들)

오퍼랜드: 명령어 실행에 필요한 데이터가 저장된 주소(addr)



→ IR의 명령어를 두 개의 필드로 나눠 생각함. 이 두 개가 합쳐 하나의 명령어가 구성됨

사례 1: LOAD addr 명령어

기억장치에 저장된 데이터를 CPU 내부의 AC로 이동하는 명령어

t_0: MAR ← IR(addr)

IR에 있는 명령어의 주소 부분(addr 필드)을 MAR로 전송

t_1: MBR ← M[MAR]

주소가 지정한 기억장소로부터 데이터 인출, MBR로 전송

t_2: AC ← MBR

데이터를 AC에 적재

→ 메모리 인출과 전체적으로 비슷함, 명령어 실행할 때 아주 잘게 나눠준다는 개념

사례 2: STA addr 명령어

AC의 내용을 기억장치에 저장하는 명령어

t_0: MAR ← IR(addr)

데이터 저장할 기억장치의 주소를 MAR로 전송

t_1: MBR ← AC

AC의 저장할 데이터를 MBR로 이동

t_2: M[MBR] ← MBR

MBR의 내용을 MAR이 지정하는 기억장소에 저장

사례 3: ADD addr 명령어

기억장치에 저장된 데이터를 AC의 내용과 더하고 결과를 AC에 저장하는 명령어

t_0: MAR ← IR(addr)

데이터 저장할 기억장치의 주소를 MAR로 전송

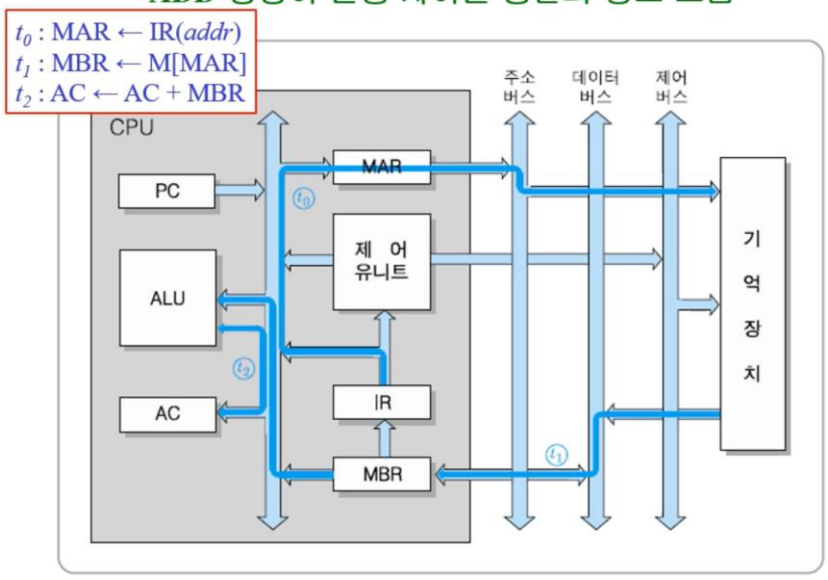
t_1: MBR ← M[MAR]

저장할 데이터를 MBR로 이동

t_2: AC ← AC + MBR

그 데이터와 AC의 내용을 더하고 결과를 AC에 저장

ADD 명령어 실행 사이클 동안의 정보 흐름



사례 4: JUMP addr 명령어

오퍼랜드가 가리키는 위치의 명령어로 실행 순서를 변경하는 명령어

t_0: PC ← IR(addr)

명령어의 오퍼랜드를 PC에 저장

데이터도 메모리에, 프로그램 자체도 메모리에 있다. 주소를 지정해주면 주소에 가서 가져오게 되는 형태임

어셈블리 프로그램 실행과정의 예

연산 코드에 임의의 정수 배정

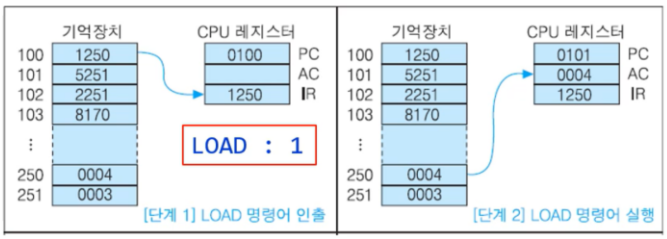
LOAD: 1, STA: 2, ADD: 5, JUMP: 8

[어셈블리 프로그램의 예]

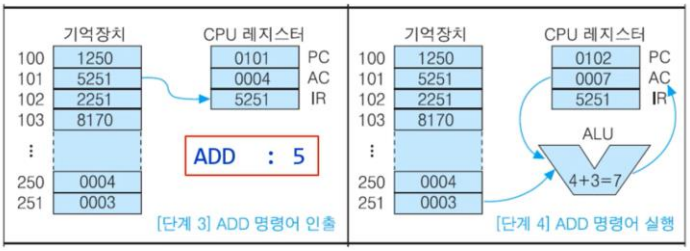
주소	명령어	기계 코드
100	LOAD 250	1250
101	ADD 251	5251
102	STA 251	2251
103	JUMP 170	8170

프로그램 실행 과정의 예

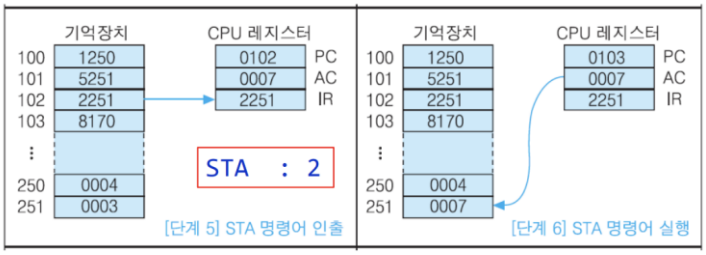
- 100 번지의 첫 번째 명령어 코드가 인출되어 IR에 저장
- 250 번지의 데이터를 AC로 이동
- PC = PC + 1 = 101



- 두 번째 명령어가 101번지로부터 인출되어 IR에 저장
- AC의 내용과 251 번지의 내용을 더하고, 결과를 AC에 저장
- PC의 내용은 102로 증가



- 세 번째 명령어가 102 번지로부터 인출되어 IR에 저장
- AC의 내용을 251 번지에 저장
- PC의 내용은 103으로 증가



- 네 번째 명령어가 103 번지로부터 인출되어 IR에 저장
- 분기될 목적지 주소, 즉 IR의 하위 부분(170)이 PC로 적재
  (다음 명령어 인출 사이클에서는 170 번지의 명령어 인출)

2.2.3 인터럽트 사이클

인터럽트: 프로그램이 실행되고 있을 때 하나 실행하고 다음을 실행하는 순차적인 방식으로 진행되는데, 현재 처리 순서를 중단하고 다른 일을 시작하게 하는 시스템 동작

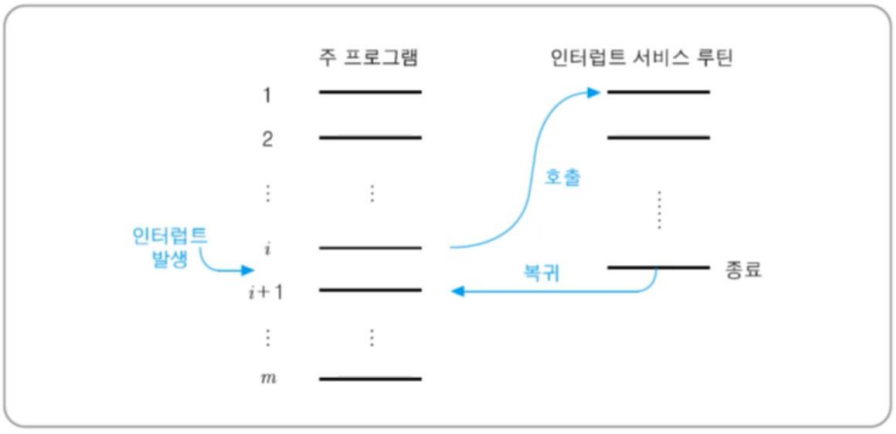
어떨 때 필요한가? → 전원이 나갔을 때 짧은 시간이라도 뭔가를 할 수 있는 시간이 존재하는데, 이때 아주 중요한 데이터를 세이브한다든가/혹은 사람의 키보드 입력을 기다리고 있을 때 속도 차이 매우 발생하는데, 이 때 키보드에서 입력이 들어왔을 때도 기다리지 않고 인터럽트를 사용할 수도 있다.

외부로부터 인터럽트 요구가 들어오면

원래 프로그램 수행 중단 → 요구된 인터럽트를 위한 서비스 프로그램을 먼저 수행

인터럽트 서비스 루틴(ISR): 인터럽트를 처리하기 위하여 수행되는 프로그램 루틴

인터럽트에 의한 제어의 이동



인터럽트 처리 과정

인터럽트가 들어왔을 때 CPU는

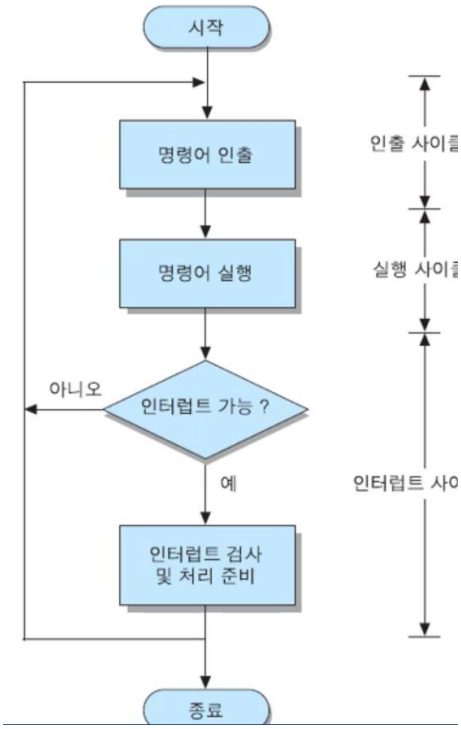
어떤 장치가 인터럽트를 요구했는지 확인, 해당 ISR 호출

서비스 종료 후 중단되었던 원래 프로그램의 수행 계속

CPU 인터럽트 처리의 세부 동작

1. 현재 명령어 실행을 끝낸 즉시, 다음 실행할 명령어 주소(PC의 내용)를 스택에 저장 → 일반적으로 스택은 주 기억 장치의 특정 부분
2. ISR을 호출하기 위해 루틴의 시작 주소를 PC에 적재. 시작 주소는 미리 지정되어 있거나 인터럽트를 요구하는 장치에서 전송됨

인터럽트 사이클이 추가된 명령어 사이클



인터럽트 사이클의 마이크로 연산

t_0: MBR ← PC

현재의 PC를 MBR 내에 옮김

t_1: M[MAR] ← SP, PC ← ISR의 시작 주소

CPU 내에 스택 포인터라는 레지스터가 하나 더 있다고 가정함, 스택의 최상위 주소(TOS)를 저장하고 있는 레지스터임, 저장 후 1 감소 (스택으로 사용하고 있는 메모리의 주소를 말하는 것)

SP의 내용을 MAR로 전송, PC 내용을 ISR의 시작 주소로 함

t_2: M[MAR] ← MBR

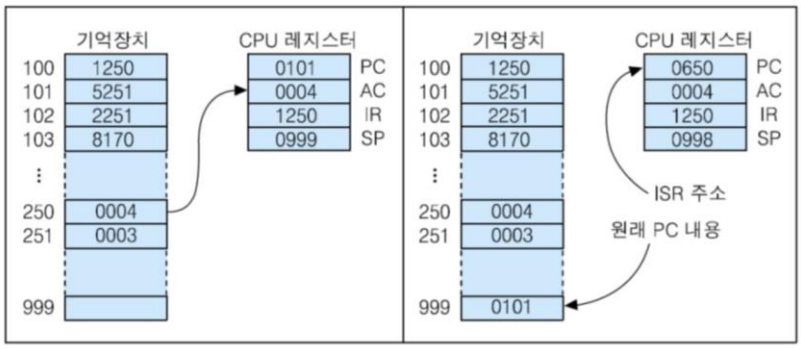
MBR에 저장되어 있던 원래 PC의 내용을 스택에 저장

인터럽트 사이클의 마이크로 연산 [예]

❑ 프로그램의 첫 번째 명령어인 **LOAD 250** 명령어가 실행되는 동안에 인터럽트가 들어왔으며, 현재 **SP = 999**이고, 인터럽트 서비스 루틴의 시작 주소는 **650** 번지라고 가정

```
100   LOAD  250
101   ADD   251
102   STA   251
103   JUMP  170
```

인터럽트 요구가 들어온 경우의 상태 변화



(a) LOAD 명령어의 실행 사이클이 종료된 상태 (b) 인터럽트 사이클이 종료된 상태

100번지 LOAD 250 명령어가 실행되는 동안에 인터럽트가 들어왔으며, 현재 SP = 999이고, ISR의 시작 주소는 650 번지라고 가정.

다중 인터럽트

인터럽트를 처리하는 동안 또 인터럽트가 일어남

다중 인터럽트의 처리방법 – 크게 두 가지

1. CPU가 인터럽트 서비스 루틴을 처리하고 있는 동안은 다른 인터럽트가 발생하지 않게 함

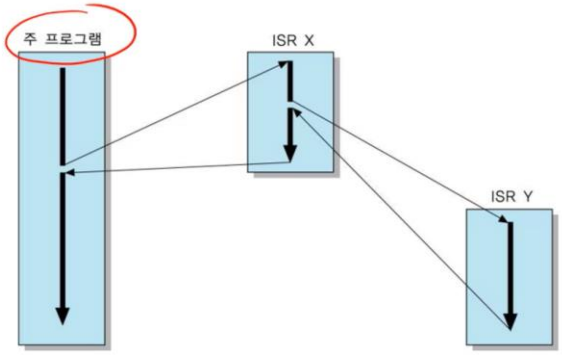
   인터럽트 플래그: 시스템 내 레지스터를 하나 더 추가함, 0일 경우 인터럽트 불가능 상태

   시스템 운영상 중단할 수 없는 경우에 사용할 수 있음

2. 인터럽트의 우선순위를 정하고, 실행되는 인터럽트보다 우선순위가 높은 인터럽트가 들어온다면 일반 프로그램처럼 일시중단하고 새로운 인터럽트를 처리함

다중 인터럽트 처리 방법

❑ 장치 X를 위한 ISR X를 처리하는 도중에 우선 순위가 더 높은 장치 Y로부터 인터럽트 요구가 들어와서 먼저 처리되는 경우에 대한 제어의 흐름



2.2.4 간접 사이클

명령어에 포함되어 있는 주소를 이용하여 그 명령어 실행에 필요한 데이터의 주소를 인출하는 사이클 → 간접 주소지정 방식에서 사용

인출 사이클과 실행 사이클 사이에 있는 사이클

간접 사이클에서 수행될 마이크로 연산

t_0: MAR ← IR(addr)

t_1: MBR ← M[MAR]

t_2: IR(addr) ← MBR

인출된 명령어의 주소 필드 내용을 이용하여 기억장치로부터 데이터의 실제 주소를 인출하여 IR의 주소 필드에 저장

2.3 명령어 파이프라인

CPU의 처리 속도를 높이기 위한 방법, CPU 내부의 하드웨어를 여러 단계로 나누어 동시에 처리 가능하게 하는 기술

비유하면 구내식당에서 밥, 반찬, 국 주는 사람 따로 있을 때 같은 시간동안 처리 능력이 3배

2단계 명령어 파이프라인

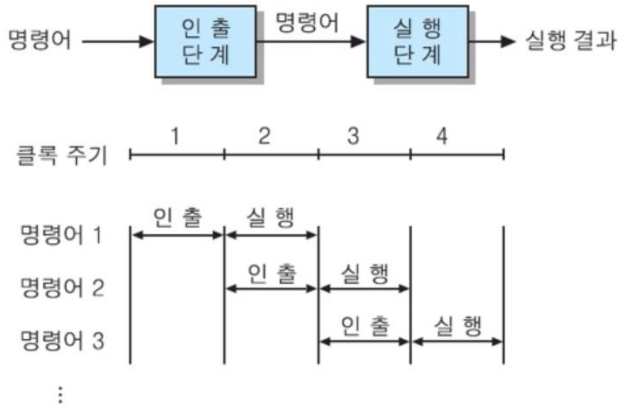
명령어를 실행하기 위해서 두 단계를 거쳤는데, 인출 단계와 실행 단계로 나눠져 있음. 이 두 단계를 각각 하나의 파이프라인이라고 행각하면 됨

두 단계로 되어있는데, 동일한 클록을 가하여 동작 시간을 일치시킴

첫 번째 클록 주기에서 첫 번째 명령어의 인출 단계

두 번째 클록 주기에선 첫 번째 명령어가 실행 단계, 두 번째 명령어는 인출 단계 → 두 번째 명령어를 먼저 가져온다는 뜻에서 prefetch라고 함

2단계 명령어 파이프라인과 시간 흐름도



속도향상(Sp) = 6/4 = 1.5배. 실행되는 명령어 수 증가 시, Sp = 2배에 접근

2단계 파이프라인을 이용하면 명령어 처리 속도가 두 배 향상

문제점: 두 단계의 처리 시간이 동일하지 않으면 두 배의 효과를 얻지 못함(효율 저하)

해결책: 파이프라인 단계를 세분화함 – 각 단계의 처리 시간을 거의 같아지도록 함

4-단계 명령어 파이프라인

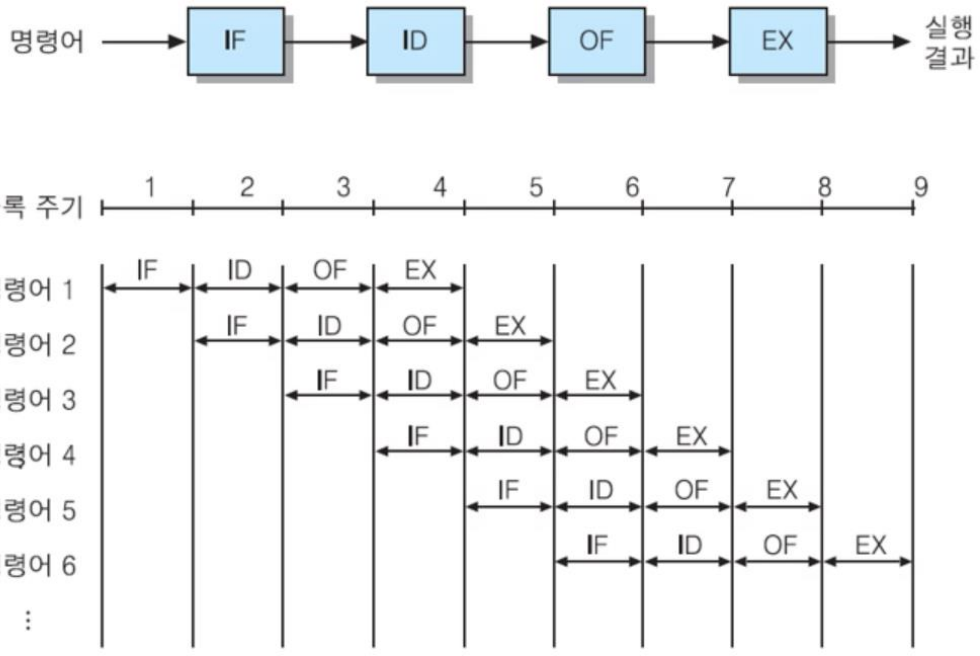
명령어 인출(IF) 단계: 다음 명령어를 기억장치로부터 인출

명령어 해독(ID) 단계: 해독기(decoder)를 이용하여 명령어를 해석

오퍼랜드 인출(OF) 단계: 기억장치로부터 오퍼랜드를 인출

실행(EX) 단계: 지정된 연산을 수행

4-단계 명령어 파이프라인과 시간 흐름도



파이프라인에 의한 속도 향상

파이프라인 단계 수 = k, 실행할 명령어들의 수 = N

각 파이프라인 단계가 한 클록 주기씩 걸린다고 가정한다면

파이프라인에 의한 전체 명령어 실행 시간: $T_k = k + (N - 1)$

→ 즉, 첫 번째 명령어를 실행하는데 k 주기가 걸리고, 나머지 (N - 1) 개의 명령어들은 각각 한 주기씩만 소요

파이프라인 되지 않은 경우의 N 개의 명령어들을 실행 시간: $T_1 = k \times N$

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

→ N이 커질수록 값이 k에 근사해짐 (lim N → inf 적용해서)

[예제 2-1]파이프라인에 의한 속도 향상

❑ 파이프라인 단계 수 = 4,
파이프라인 클록 = 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

➔속도향상(speedup: *Sp*) = (10 × 4) / 13 ≒ 3.08 배 (참고: 1GHz 역수가 1ns)

파이프라인의 효율 저하 요인들

모든 명령어들이 파이프라인 단계들을 모두 거치지는 않는다

→ 어떤 명령어에서는 오퍼랜드를 인출할 필요가 없지만, 파이프라인의 하드웨어를 단순화시키기 위해선 모든 명령어가 네 단계들을 모두 통과하도록 해야 함

파이프라인의 클록은 처리 시간이 가장 오래 걸리는 단계를 기준으로 결정됨

→ 다른 것을 강제로 빠르게 할 수는 없으니까

IF단계와 OF단계가 동시에 기억장치를 액세스하는 경우, 기억장치 충돌이 일어나면 지연이 발생함

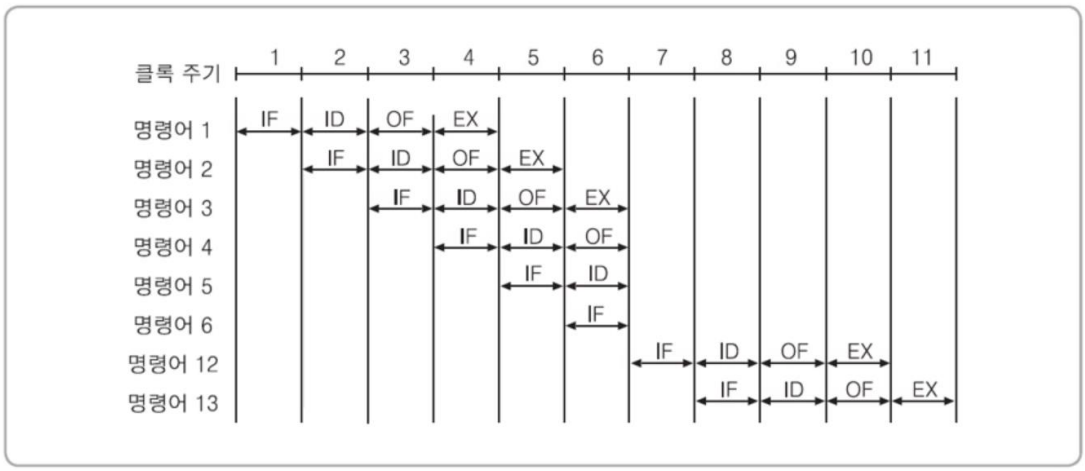
→ 둘 다 메모리에서 갖고 오는 것이기 때문에 메모리를 동시에 사용하려 하면 충돌 발생함, 교통정리 하다 보면 느려짐

조건 분기 명령어가 실행되면, 미리 인출하여 처리하던 명령어들이 무효화됨

→ 참일 때와 거짓일 때 실행하는 내용이 다르니까

조건 분기가 존재하는 경우의 시간 흐름도

[예] 명령어 3: JZ 12 ; jump (if zero) to address 12



상태 레지스터

명령어 실행 결과에 따른 조건 플래그들 저장

조건 플래그의 종류

부호(S) 플래그: 직전 수행된 산술연산 결과값의 부호 비트 저장(양수: 0, 음수: 1)

영(Z) 플래그: 연산 결과값이 0 이면, 1로 세트

올림수(C) 플래그: 덧셈/뺄셈에서 올림수/빌림수가 발생한 경우에 1로 세트

동등(E) 플래그: 두 수를 비교한 결과가 같게 나왔을 경우에 1로 세트

오버플로우(V) 플래그: 산술 연산 과정에서 오버플로우 발생한 경우에 1로 세트

인터럽트(I) 플래그

인터럽트 가능(interrupt enabled) 상태이면, 0으로 세트

인터럽트 불가능(interrupt disabled) 상태이면, 1로 세트

슈퍼바이저(P) 플래그

CPU의 실행 모드가 슈퍼바이저 모드(supervisor mode)이면, 1로 세트

사용자 모드(user mode)이면, 0으로 세트