# 컴개실 PPT HW3
# Assembly 언어

13조 백승우 박재완2 배수민 조강현

# 목차


01 Assembly어의 개요

02 프로그램의 동작 과정

03 메모리, 레지스터, 스택, 힙

04 어셈블리 문법

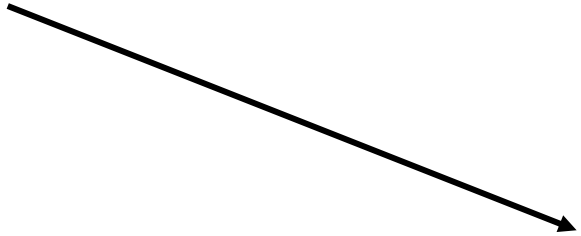
05 어셈블리 예제

# 1. Assembly어의 개요

## 기계어 Original Low-Level Language

(001001 11101 1111111111111000)

- 너무 어려움.
- 0과 1만 사용하여 많은 노력이 필요.



## Assembly Low-Level Language

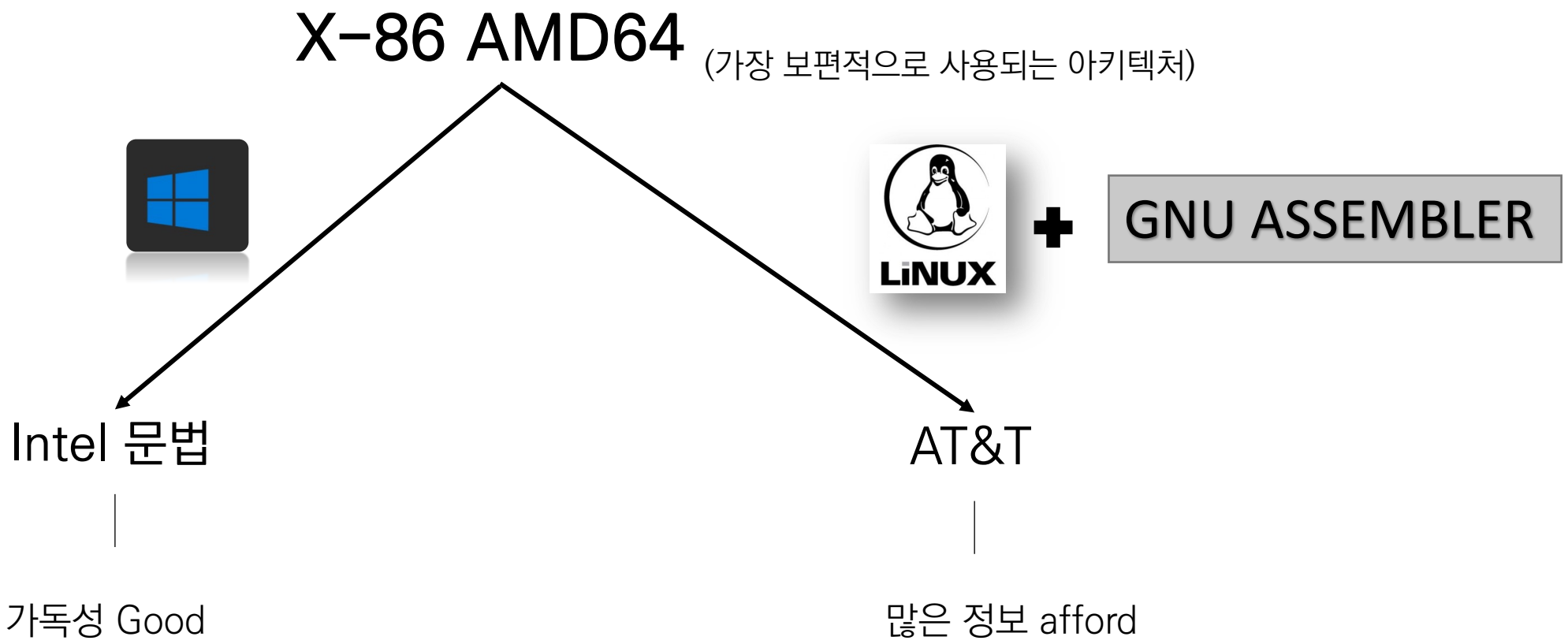
(LBL R 2/P R 3/Y)

- 기계어를 사람의 언어에 최대한 근접하게 기호화한 언어
- 알파벳 기호와 숫자를 이용하여 기계에 대응.
- 특정 프로세스에 맞추어 작용. (1:1대응)

# 1. Assembly어의 개요

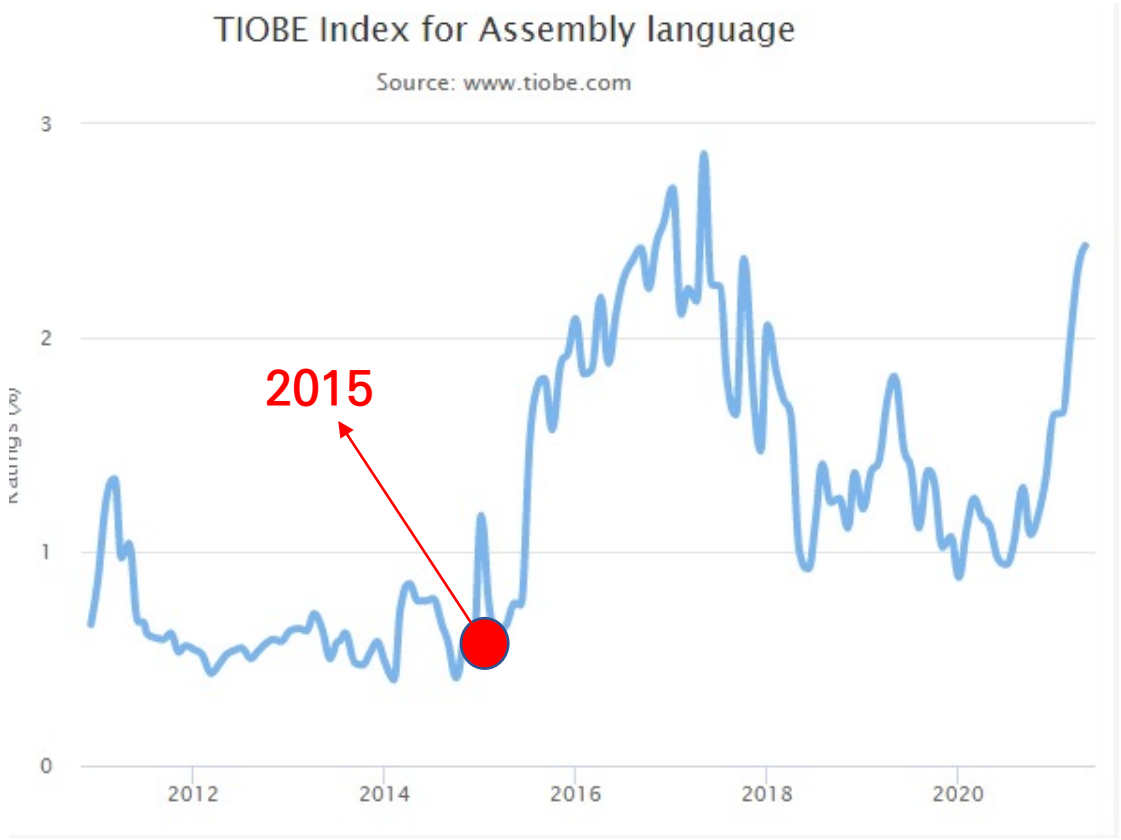
· 기계어와 1:1 **대응**인 Assembly어는 특별한 규격이 X

· 컴퓨터 내의 아키텍처, 어셈블러에 따라 다양한 구조와 문법을 가짐



# 1. Assembly어의 개요 – 사용 부문

· 2015년 이후 스마트워치 등 초소형 사물형기기 인터넷의 사용 ↑




→ **어셈블리어의 사용 빈도 ↑**

**Because..**

→ 어셈블리어는 기계어에 가장 가까운 사람의 언어

→ 프로그램을 최적화하여 기계의 명령을 수행하고 특정 기능들을 수행하는 데에 있어 아주 용이

→ " **초소형 임베디드 시스템** " 에 매우 유용

# 1. Assembly어의 개요 – 장점

1. 가독성 good
2. 전반적으로 프로그래밍 수월
3. 기계어에 가까워 작동속도 매우 빠름
4. 명령어 수가 적고, 지시어 사용 가능
5. 매크로 호출이 쉽고 빠름

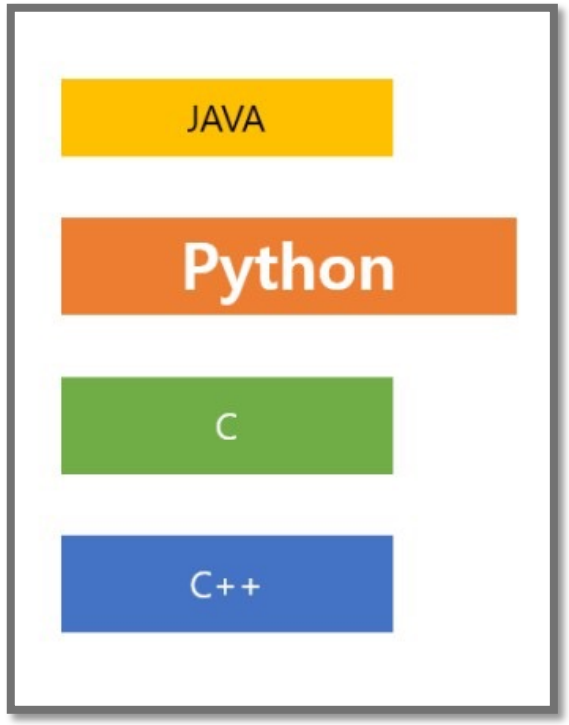

수동기어변속기
( ≒ 어셈블리)

자동기어변속기
( ≒ C, C++)

c.f. 프로그래밍 언어를 자동차 기어변속에 비유하자면,
Assembly어는 수동기어변속기, C나 C++등 고수준 언어는
자동기어변속기에 비유할 수 있다. 수동기어변속기는 운전자의 입맛에 맞게
차를 그때그때 control할 수 있다는 장점이 있는 반면, 자동기어변속기는
운전자의 편리한 운전을 위해 속도에 맞추어 기어를 바꾸어 준다.

# 1. Assembly어의 개요 – 단점

1. C와 같은 고급 언어에 비해 생산성 저하
2. 인간의 관점이 아닌 CPU 아키텍처의 관점에서 서술해야 함.
3. 고성능 CPU로 갈수록 성능을 끌어내는 것이 힘듬.
4. 프로그래머가 직접 지시해야 할 것들이 고급언어보다 많음.
5. 컴파일러의 발전으로 인해 어셈블리어를 이용한 직접적인 코딩이 불필요해짐.




최근에는 CPU와 아키텍처, 컴파일러 등 컴퓨터 내부의 성능이 인간이 도달하지 못할 지경에 이르러 발전하여 굳이 복잡한 Assembly어로 프로그래머가 명령을 내릴 일이 거의 없어졌다.

# 2. 컴퓨터 프로그램의 동작

```
MONITOR FOR 6802 1.4          9-14-80  TSC ASSEMBLER  PAGE    2


C000                    ORG     ROM+$0000 BEGIN MONITOR
C000 8E 00 70  START    LDS     #STACK

               ****************************************
               * FUNCTION: INITA - Initialize ACIA
               * INPUT: none
               * OUTPUT: none
               * CALLS: none
               * DESTROYS: acc A

0013           RESETA   EQU     %00010011
0011           CTLREG   EQU     %00010001

C003 86 13     INITA    LDA A  #RESETA   RESET ACIA
C005 B7 80 04           STA A  ACIA
C008 86 11              LDA A  #CTLREG   SET 8 BITS AND 2 STOP
C00A B7 80 04           STA A  ACIA

C00D 7E C0 F1           JMP    SIGNON    GO TO START OF MONITOR

               ****************************************
               * FUNCTION: INCH - Input character
               * INPUT: none
               * OUTPUT: char in acc A
               * DESTROYS: acc A
               * CALLS: none
               * DESCRIPTION: Gets 1 character from terminal

C010 B6 80 04  INCH     LDA A  ACIA      GET STATUS
C013 47                 ASR A            SHIFT RDRF FLAG INTO CARRY
C014 24 FA              BCC    INCH      RECIEVE NOT READY
C016 B6 80 05           LDA A  ACIA+1    GET CHAR
C019 84 7F              AND A  #$7F      MASK PARITY
C01B 7E C0 79           JMP    OUTCH     ECHO & RTS

               ****************************************
               * FUNCTION: INHEX - INPUT HEX DIGIT
               * INPUT: none
               * OUTPUT: Digit in acc A
               * CALLS: INCH
               * DESTROYS: acc A
               * Returns to monitor if not HEX input

C01E 8D F0     INHEX    BSR    INCH      GET A CHAR
C020 81 30              CMP A  #'0       ZERO
C022 2B 11              BMI    HEXERR    NOT HEX
C024 81 39              CMP A  #'9       NINE
C026 2F 0A              BLE    HEXRTS    GOOD HEX
C028 81 41              CMP A  #'A
C02A 2B 09              BMI    HEXERR    NOT HEX
C02C 81 46              CMP A  #'F
C02E 2E 05              BGT    HEXERR
C030 80 07              SUB A  #7        FIX A-F
C032 84 0F     HEXRTS   AND A  #$0F      CONVERT ASCII TO DIGIT
C034 39                 RTS

C035 7E C0 AF  HEXERR   JMP    CTRL      RETURN TO CONTROL LOOP
```

컴퓨터 프로그램
(**Assembly**, **C** 등 언어 이용)

번역

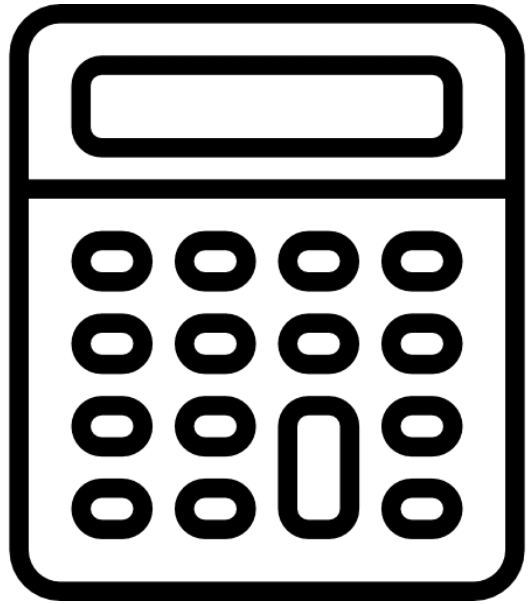
**컴파일러**

```
00000000  01 00 FF FF 00 00 00 00
00000010  0C 00 00 00 00 00 26 01
00000020  65 00 6C 00 65 00 63 00
00000030  6C 00 65 00 00 00 08 00
00000040  20 00 53 00 68 00 65 00
00000050  6C 00 67 00 00 00 00 00
00000060  03 01 A1 50 53 00 3A 00
00000070  FF FF 83 00 00 00 00 00
00000080  03 00 01 50 0E 00 56 00
00000090  FF FF 80 00 26 00 41 00
000000a0  20 00 74 00 6F 00 20 00
000000b0  00 00 00 00 00 00 00 00
000000c0  7E 00 7D 00 32 00 0E 00
000000d0  4F 00 4B 00 00 00 00 00
000000e0  00 00 01 50 B4 00 7D 00
000000f0  FF FF 80 00 43 00 61 00
00000100  00 00 00 00 00 00 00 00
00000110  EA 00 7D 00 32 00 0E 00
00000120  26 00 48 00 65 00 6C 00
00000130  00 00 00 00 00 00 00 00
00000140  3B 00 0E 00 2F 25 00 00
00000150  00 00 00 00 00 00 00 00
00000160  1E 00 08 00 EE 25 00 00
00000170  6C 00 65 00 20 00 54 00
00000180  00 00 00 00 00 00 00 00
00000190  54 00 30 00 2C 00 08 00
000001a0  50 00 61 00 72 00 73 00
000001b0  52 00 75 00 6C 00 65 00
000001c0  00 00 00 00 00 00 00 00
000001d0  1A 01 71 00 ED 25 00 00
000001e0  00 00 00 00 00 00 00 00
000001f0  3E 00 08 00 EC 25 00 00
00000200  6C 00 65 00 63 00 74 00
00000210  65 00 20 00 46 00 6F 00
00000220  6C 00 65 00 00 00 00 00
00000230  80 08 81 50 0E 00 1B 00
00000240  FF FF 81 00 00 00 00 00
00000250  00 00 02 50 19 00 61 00
00000260  FF FF 82 00 00 00 00 00
```

컴퓨터의 ISA 형식에 맞는 **기계어**
(**메모리**에 저장)

명령어 해석
동작 수행

**CPU**

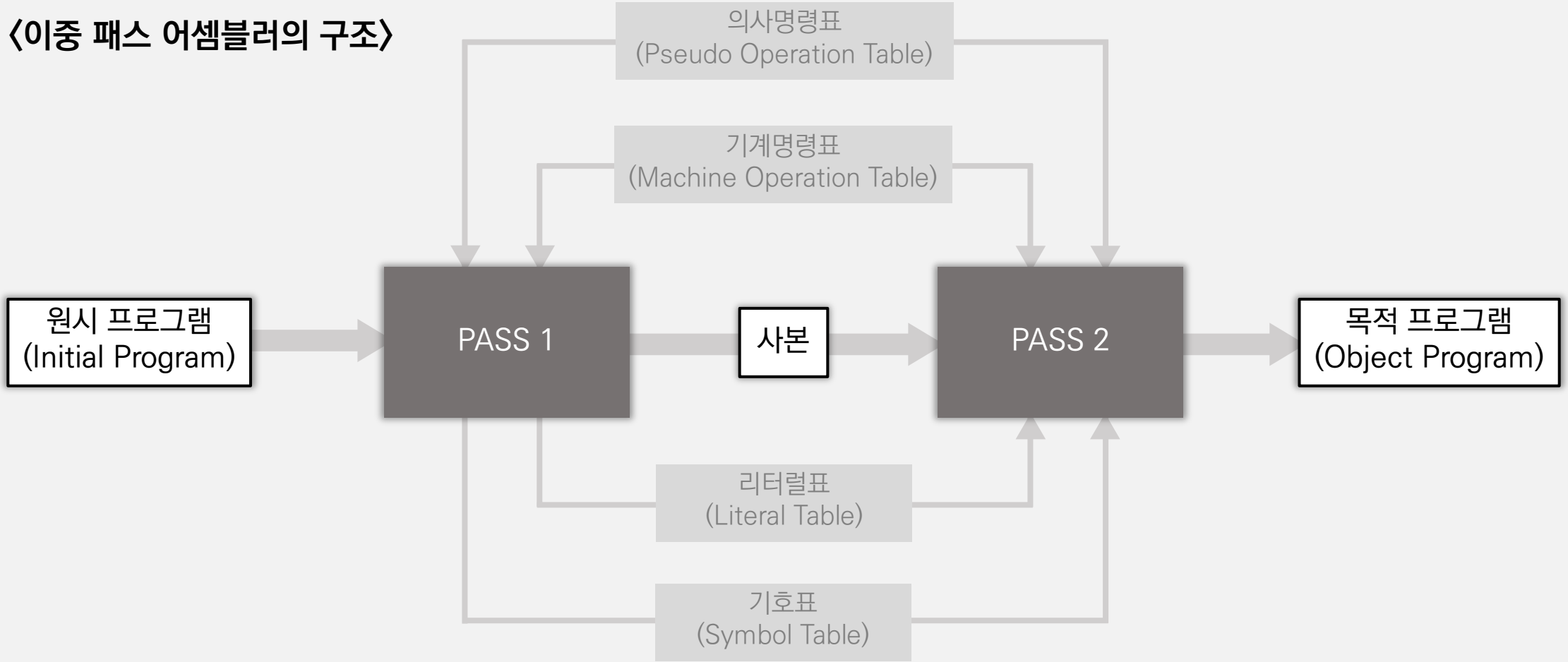


프로그램 실행

# 2. 어셈블러(Assembler)를 이용한 컴파일

어셈블러(Assembler) : 어셈블리어로 작성된 프로그램을 기계어(ISA 명령어)로 된 프로그램으로 번역(assemble)하는 컴파일러
주로 2단계의 작업을 실행하는 이중 패스 구조를 이용함

〈이중 패스 어셈블러의 구조〉



# 2. 어셈블러(Assembler)를 이용한 컴파일

(EX)

| Label | Opcode | Operand |
|---|---|---|
| | ORG | #100 |
| BeginProg | LDV | #countUP |
| | OUTCH | |
| | CMP | NumA |
| | JMP | Finish |
| | JNE | MoveOn |
| MoveOn | LDD | countUp |
| | INC | |
| | STO | countUp |
| | JMP | BeginProg |
| Finish | LDM | #25 |
| | END | |
| countUp | | 200 |
| | | 21 |
| | | 35 |
| NumA | | 20 |

**PASS 1**

- 각 명령어에 메모리 주소 부여
- 사용된 Label, Literal의 메모리 주소를 각각 Symbol Table, Literal Table에 저장

| Symbol Table | |
|---|---|
| Label | Address |
| … | … |
| … | … |

| Literal Table | |
|---|---|
| Literal | Address |
| … | … |
| … | … |

**PASS 2**

- 각 명령어를 기계어로 번역
- PASS 1에서 전달받은 Symbol Table, Literal Table로부터 Label, Literal의 값을 대체

# 2. 링커(Linker)와 로더(Loader)

원시 프로그램 (Initial Program)

원시 프로그램 (Initial Program)

원시 프로그램 (Initial Program)

컴파일러

목적 프로그램 (Object Program)

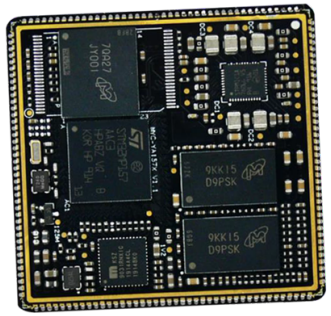
목적 프로그램 (Object Program)

목적 프로그램 (Object Program)

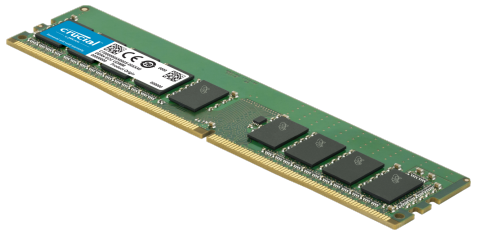
기타 라이브러리, 실행 프로그램

링커

실행 가능 프로그램 (Executable Program)



중앙처리장치(CPU)



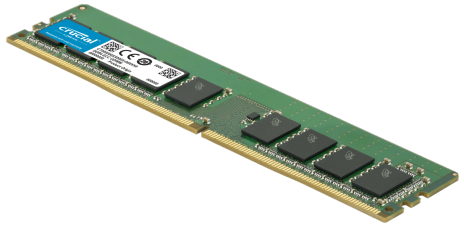
메모리(Memory)

로더

- 프로그램이 사용할 메모리 공간 확보
- 실행 파일의 데이터를 메모리에 적재
- CPU의 제어를 프로그램의 시작 메모리 주소로 이동

# 3. 메모리 주소 (Memory Address)

메모리 : 컴퓨터의 구성 요소 중 데이터가 저장되는 공간



메모리 주소 : 데이터가 저장된 메모리의 위치에 대한 고유한 값, 1바이트 단위로 관리됨

| … | 0x100 | 0x101 | 0x102 | 0x103 | … |
|---|---|---|---|---|---|

각 주소에 특정 데이터가 저장된다. 그러나 1바이트를 넘는 데이터를 입력하려면?

Ex) 메모리에 0x12345678을 저장하는 경우

### Big-Endian 방식

| 주소 | … | 0x100 | 0x101 | 0x102 | 0x103 | … |
|---|---|---|---|---|---|---|
| 데이터 | … | 0x12 | 0x34 | 0x56 | 0x78 | … |

1바이트 크기로 데이터를 나누어 저장하되,
상위 바이트의 값이 먼저 오도록 함

### Small-Endian 방식

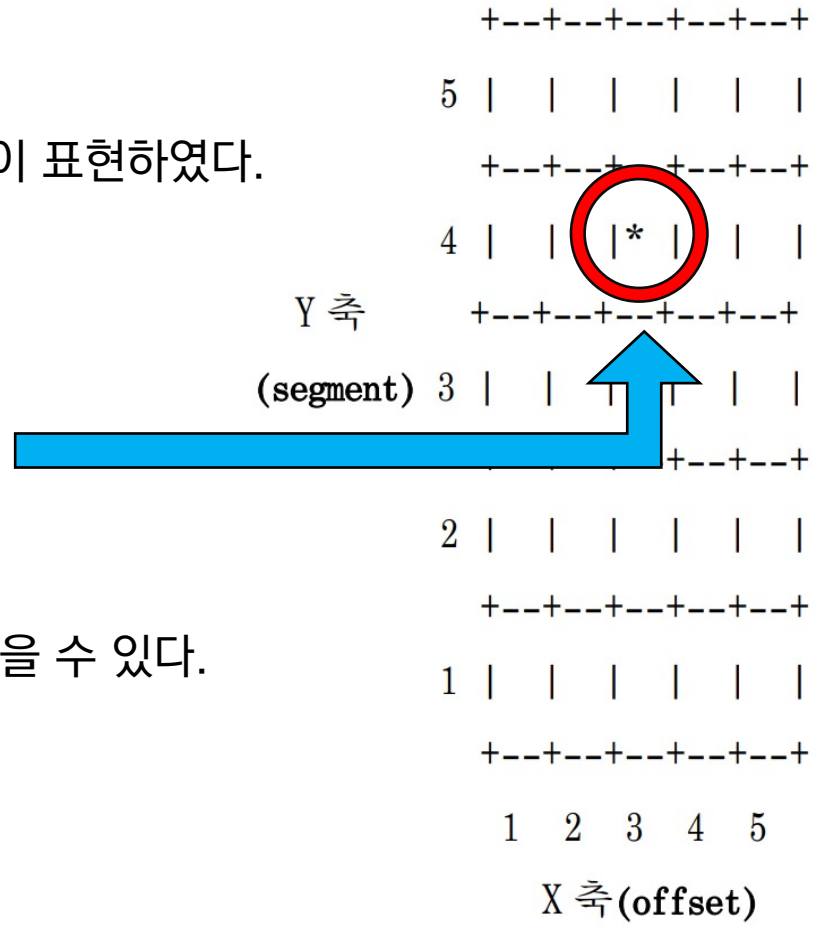
| 주소 | … | 0x100 | 0x101 | 0x102 | 0x103 | … |
|---|---|---|---|---|---|---|
| 데이터 | … | 0x78 | 0x56 | 0x34 | 0x12 | … |

1바이트 크기로 데이터를 나누어 저장하되,
하위 바이트의 값이 먼저 오도록 함

# 3. 메모리 주소 (Memory Address) – 세그먼트:오프셋

컴퓨터는 16bit, 32bit, 64bit 에 적합하게 설계되어왔다.
하지만 1MB($\approx 2^{20}$bit)의 메모리를 디자인하게 되며, 메모리 주소를 다루기 힘들어졌다.

따라서 16bit 수 두 개를 이용하여 메모리 주소를 세그먼트 : 오프셋과 같이 표현하였다.

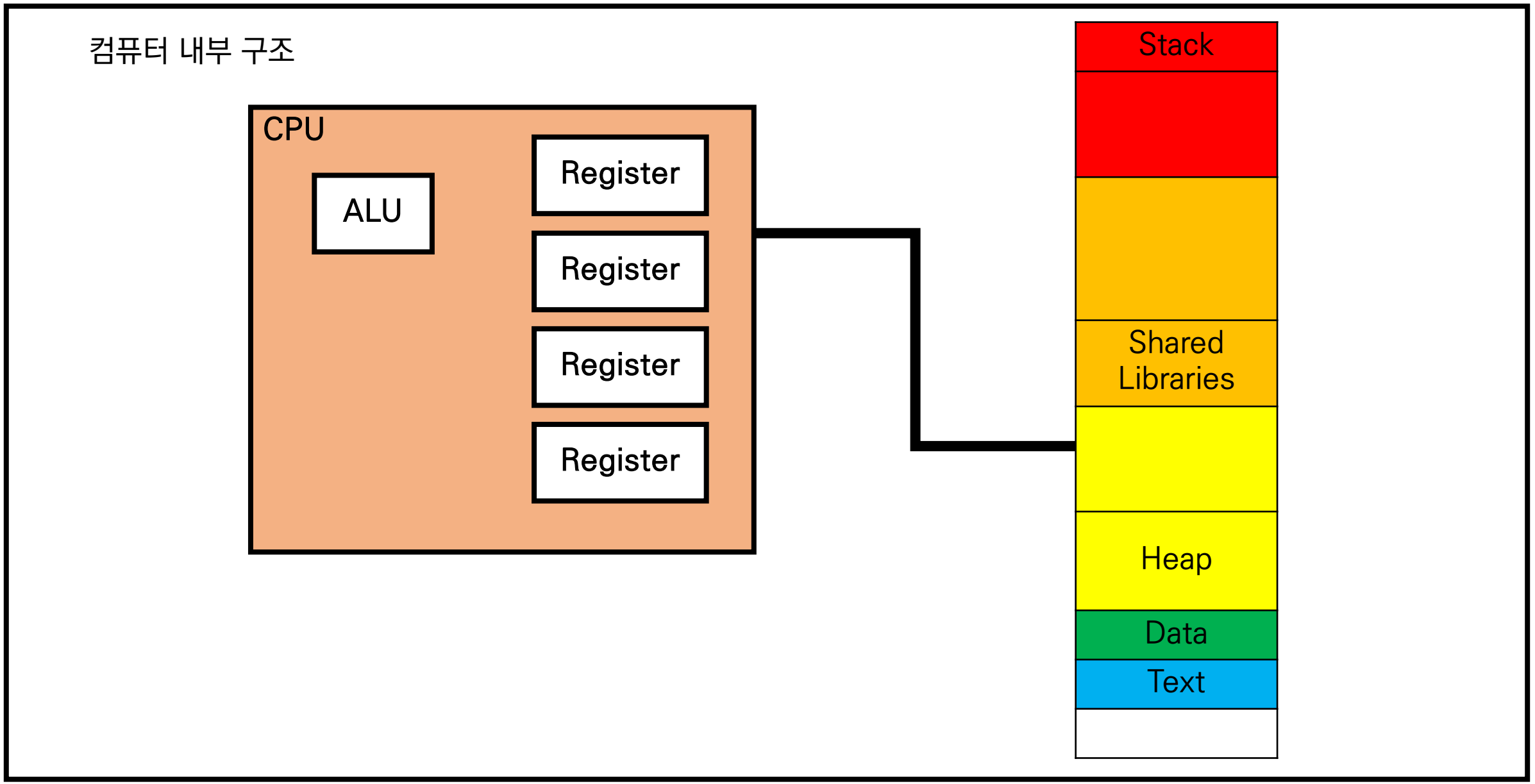



물리적 주소의 예시
(3,4)-> 4×10 +3=43

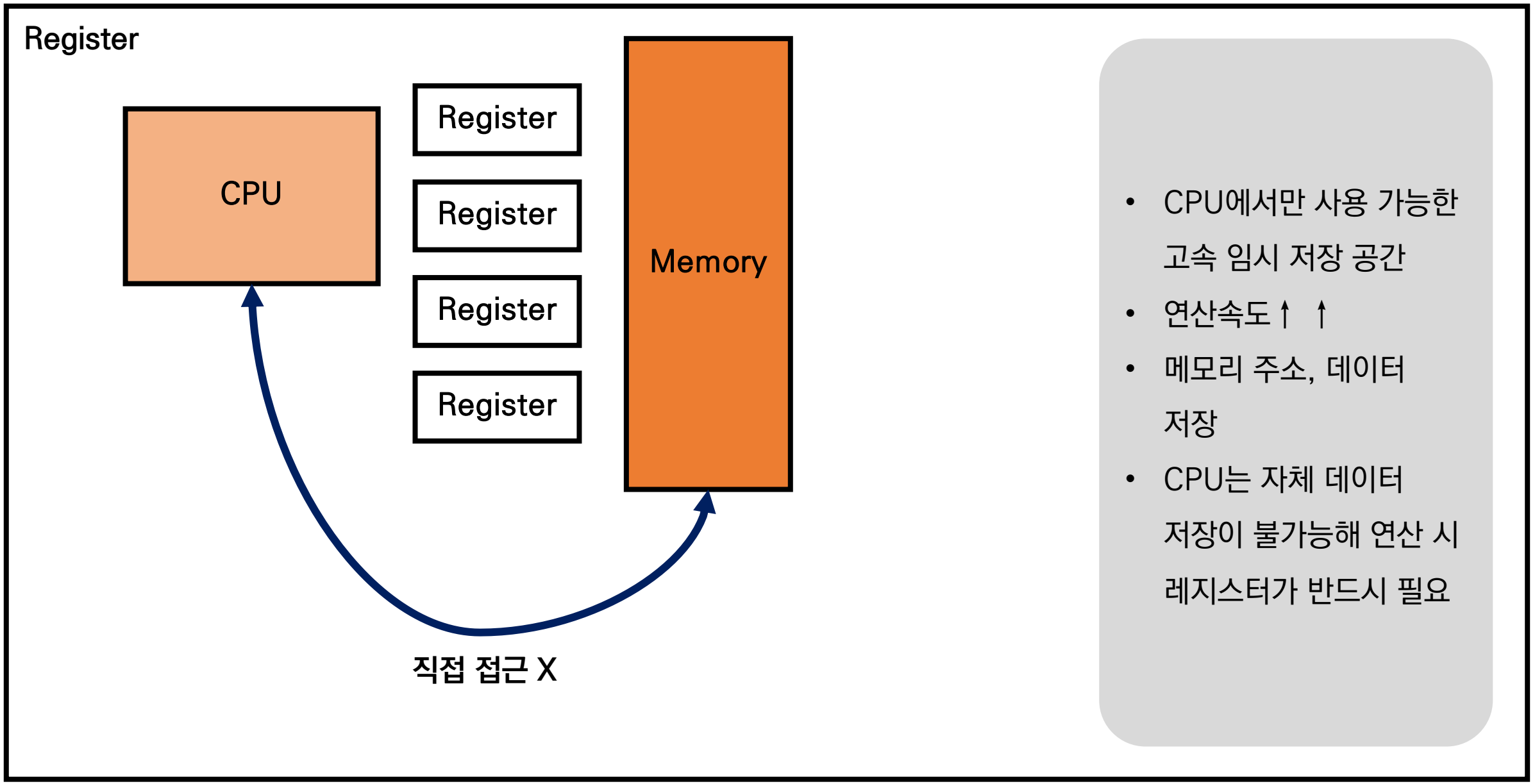
Segment 번호와 Offset 번호가 주어지면 다음과 같이 물리적 주소를 얻을 수 있다.

물리적 주소(실제 주소) = Segment×10h+Offset
1234h:4321h = 16661h

# 3. 레지스터, 스택, 힙


컴퓨터 내부 구조
CPU
ALU
Register
Register
Register
Register
Stack
Shared
Libraries
Heap
Data
Text

# 3. 레지스터, 스택, 힙

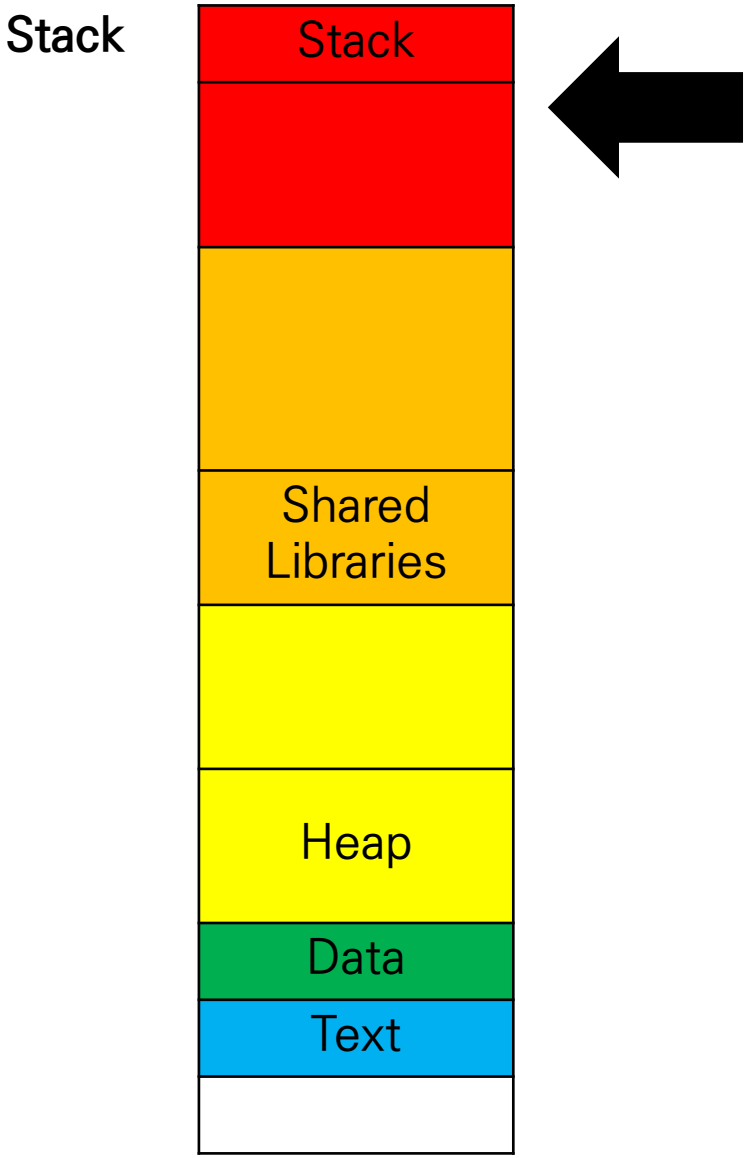



- CPU에서만 사용 가능한 고속 임시 저장 공간
- 연산속도 ↑ ↑
- 메모리 주소, 데이터 저장
- CPU는 자체 데이터 저장이 불가능해 연산 시 레지스터가 반드시 필요

# 3. 레지스터, 스택, 힙

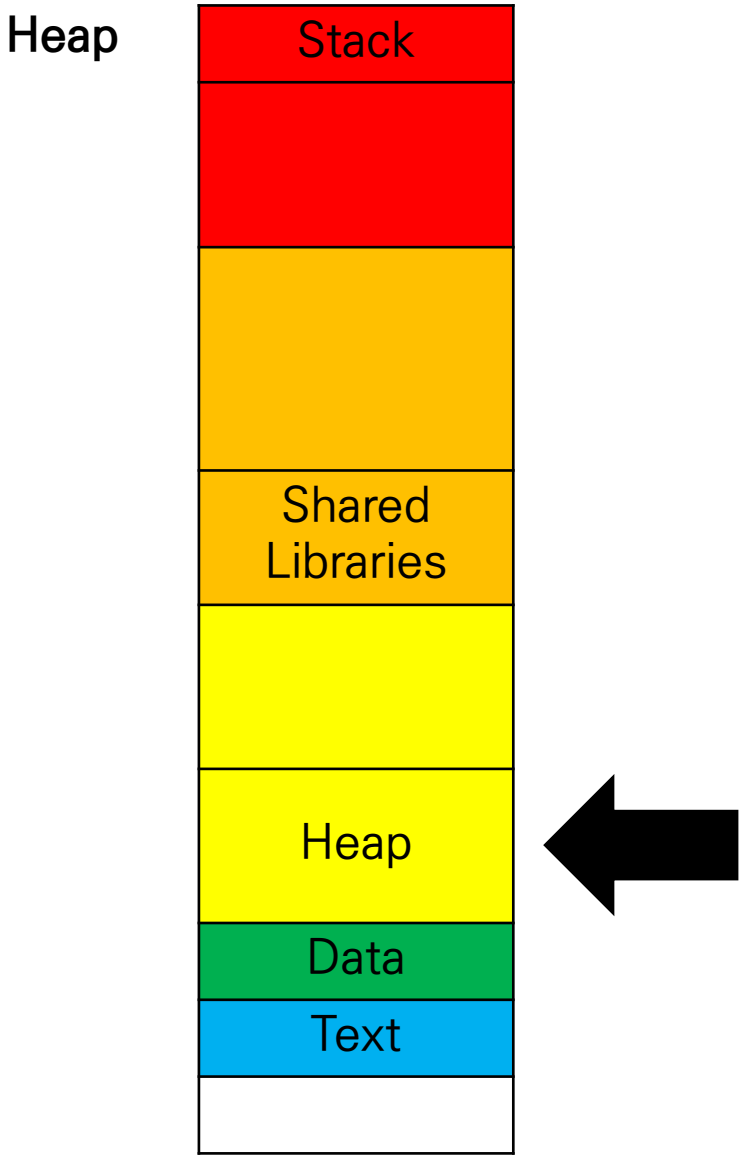
여러가지 Register

# 3. 레지스터, 스택, 힙




- LIFO(Last In First Out) : 마지막에 들어간 항목이 제일 먼저 나옴
- 지역 변수, 매개 변수, 리턴 값, 주소 저장된 공간
- 높은 주소에서 낮은 주소 방향으로 할당
- 함수 호출과 함께 할당, 함수 호출 완료 시 소멸
- 컴파일 타임에 크기가 결정됨
- 스택 영역에 차례대로 저장되는 함수의 호출 정보를 스택 프레임이라 함

# 3. 레지스터, 스택, 힙




- 스택 메모리보다 할당할 수 있는 메모리 공간 ↑
- 사용자가 직접 관리 가능한 메모리 영역(동적으로 할당, 해제)
- 낮은 주소에서 높은 주소 방향으로 할당
- 메모리 해제 X, 주소 잃어버리면 memory leak 발생
- 런타임에 크기가 결정됨
- Malloc함수를 통해 직접 힙 영역에 메모리 할당
- Free함수를 통해 할당 받은 메모리 공간 반환
- calloc, realloc, malloc, free

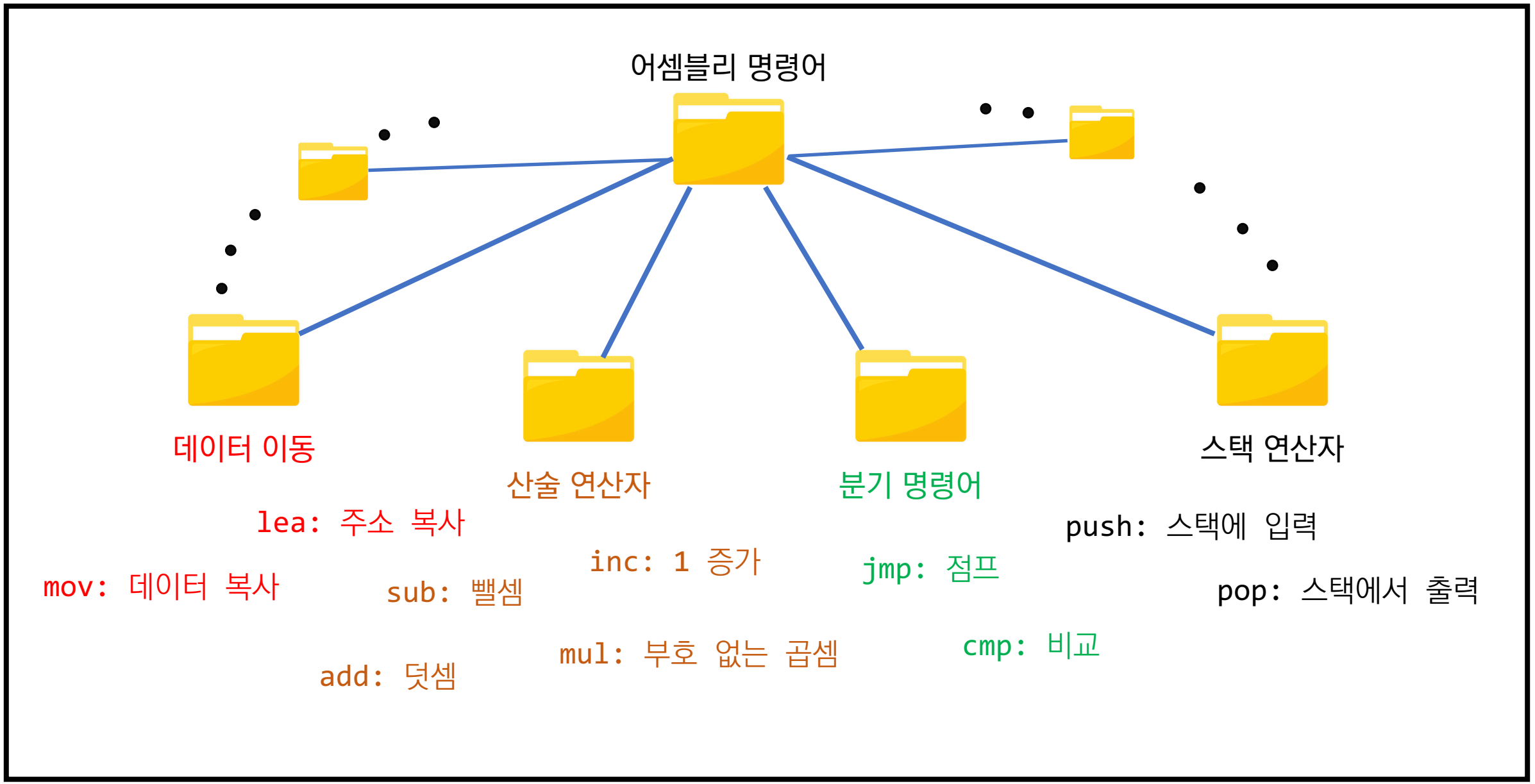
# 4. 어셈블리 문법 – 구조

x64 Intel 문법의 구조


ex )

**ADD EAX, EBX ; EBX의 값을 EAX에 더한다**

# 4. 어셈블리 문법 – 명령어


어셈블리 명령어
데이터 이동
산술 연산자
분기 명령어
스택 연산자
lea: 주소 복사
mov: 데이터 복사
sub: 뺄셈
inc: 1 증가
jmp: 점프
push: 스택에 입력
pop: 스택에서 출력
add: 덧셈
mul: 부호 없는 곱셈
cmp: 비교

# 5. 예제를 통해 알아보는 어셈블리 문법 – Hello World

- x64 Intel 문법 사용

**HelloWorld.asm**

```
section .data
        text db "Hello, World!", 0x0A

section .text
        global _start

_start:
        call _printHello

        mov rax, 60
        mov rdi, 0
        syscall

_printHello:
        mov rax, 1
        mov rdi, 1
        mov rsi, text
        mov rdx, 14
        syscall
```

- 어셈블리 프로그램은 세 개의 section 으로 분리 되어있다.
- section .data :
  전역, 정적 변수를 선언하는 공간
  상수, 버퍼 사이즈를 선언
- section .text :
  실행할 코드를 작성하는 공간
- section .bbs :
  추가적인 변수를 선언하는 공간

# 5. 어셈블리 예제 – Hello World

**HelloWorld.asm**

```
section .data
        text db "Hello, World!", 0x0A

section .text
        global _start

_start:
        call _printHello

        mov rax, 60
        mov rdi, 0
        syscall

_printHello:
        mov rax, 1
        mov rdi, 1
        mov rsi, text
        mov rdx, 14
        syscall
```

- db 명령어로 메모리에 1byte씩 문자를 나열, 0x0A는 개행문자
- text는 데이터가 위치한 메모리 주소에 할당한 이름이다.

- global label_name : label_name을 전역에 선언하는 어셈블러 지시어
- _start는 프로그램의 시작 지점이다.

- 프로그램 시작 시 _printHello 라벨을 호출

# 5. 어셈블리 예제 – Hello World

**HelloWorld.asm**

```
section .data
        text db "Hello, World!", 0x0A

section .text
        global _start

_start:
        call _printHello

        mov rax, 60
        mov rdi, 0
        syscall

_printHello:
        mov rax, 1
        mov rdi, 1
        mov rsi, text
        mov rdx, 14
        syscall
```

- rax, rdi, rsi, rdx는 레지스터의 종류
- System call을 통해 기본적으로 실행 가능한 함수들의 parameter가 될 수 있다.
- rax : 함수의 반환값을 저장, ID를 저장한 후 syscall이 선언되면 ID에 해당하는 함수 번호를 가진 함수를 호출한다.
- rdi : description, 값에 따라 함수의 세부 역할이 바뀐다.
- rsi : source, 메모리 주소를 저장해놓다가 함수가 시작하는 지점으로 사용할 수 있다.
- rdx : 다른 레지스터들을 도와주는 역할

# 5. 어셈블리 예제 – Hello World

**HelloWorld.asm**

```
section .data
        text db "Hello, World!", 0x0A

section .text
        global _start

_start:
        call _printHello

        mov rax, 60
        mov rdi, 0
        syscall

_printHello:
        mov rax, 1
        mov rdi, 1
        mov rsi, text
        mov rdx, 14
        syscall
```

- 프로그램 실행 종료를 의미하는 함수 호출
- rdi에 0이 들어간 것은 정상 종료를 의미

- rax가 1 : sys_write가 호출
- rdi가 1 : standard output
- rsi가 text : 출력할 문자열의 위치
- rdx가 14 : 문자열의 길이가 14(개행문자 포함)

# 5. 예제를 통해 알아보는 어셈블리 문법 – 반복문

- x64 Intel 문법 사용

**loop.asm**

```
section .data
        msg db "IWantHome "

section .text
        global _start

_start:
        mov rax, 1
        mov rdi, 1
        mov rsi, msg
        mov rdx, 10
        mov r10, 1

again:
        cmp r10, 10 ; 원하는 횟수만큼
        je done
        syscall
        mov rax, 1
        inc r10
        jmp again

done:
        mov rax, 60
        mov rdi, 0
        syscall
```

- 두 Operand를 비교한다.

- je = jump on equal
- 결과가 0일 경우에 분기하는 명령어
- again을 10번째 돌 때 실행

- 출력 후 inc 명령어로 r10을 증가
- 분기 명령어 jmp로 again을 다시 실행

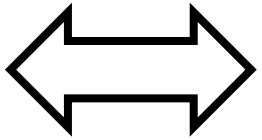
- 반복문이 끝나면 프로그램을 종료

# 5. 어셈블리 예제 – 고급 언어와의 비교

**HelloWorld.asm**

```
section .data
        text db “Hello, World!”, 0x0A

section .text
        global _start

_start:
        call _printHello

        mov rax, 60
        mov rdi, 0
        syscall

_printHello:
        mov rax, 1
        mov rdi, 1
        mov rsi, text
        mov rdx, 14
        syscall
```

어셈블리어는 저급 언어,
파이썬, C는 고급 언어로
어셈블리로는 C와 파이썬 정도의
추상화가 불가능하다.



**HelloWorld.py**

```
print(“Hello, World!”)
```

**HelloWorld.c**

```
#include <stdio.h>

int main() {
   printf(“Hello, World!”);
   return 0;
}
```

# 추가 4. 어셈블리 문법 – Intel vs AT&T


| | | |
|---|---|---|
| 레지스트리 표현 | eax | %eax |
| 상수 표현 | h(16진수), b(2진수), o(6진수) | $숫자 |
| Operands 위치 | destination, source | source, destination |
| 메모리 주소 참조 | [eax] | (%eax) |
| 레지스터+offset 위치 | [eax+숫자] | 숫자(%eax) |

| | | | |
|---|---|---|---|
| ex ) | 명령 : | ADD EAX, EBX | ADD %EAX, %EBX |
| | 해석 : | EBX의 값을 EAX에 더한다 | EAX의 값을 EBX에 더한다 |

# 참고자료


- https://en.wikipedia.org/wiki/Assembly_language
- https://en.wikipedia.org/wiki/Processor_register
- https://en.wikipedia.org/wiki/Stack-based_memory_allocation
- https://en.wikipedia.org/wiki/Memory_management#HEAP
- https://en.wikipedia.org/wiki/Central_processing_unit
- https://en.wikipedia.org/wiki/Offset_(computer_science)
- https://en.wikipedia.org/wiki/Linker_(computing)
- https://www.brainbox.co.kr/bbs/board.php?bo_table=review&wr_id=7033
- https://en.wikipedia.org/wiki/Assembly_language
- https://sites.google.com/site/cyberworksprogramminglanguages/machine-code-language
- http://clipart-library.com/clip-art/calculator-transparent-23.htm
- https://kr.mouser.com/new/myir-tech/myir-myc-ya157c-cpu-module/