CHUNGNAM NATIONAL UNIVERSITY

시스템 프로그래밍

강의 4 : 3장. 어셈블리어 I
3.1 프로세서의 역사
3.2 프로그램의 코딩
3.3 데이터 이동 명령어
3.4 정보접근하기

2014년 9월 23일
http://eslab.cnu.ac.kr

* Some slides are from Original slides of RBE

전달사항

제주도 세미나에서...

기업에서 원하는 IT 인재
- 전문성 : 많은 전공과목 이수를 기대
- Quick Adaptor with Basics
  - 패러다임 쉬프트에 적응하는 능력 보유
- 수레바퀴형 인재
  - 복합 경쟁력 보유
    - 시스템 전문가 : SW + HW
    - SW 전문가 : App SW + System SW
- Core SW 역량
  - System SW + App SW
- 현재는 시스템 SW 역량 부족상황
  - 자바만 하고 C는 못하는 인재 ?

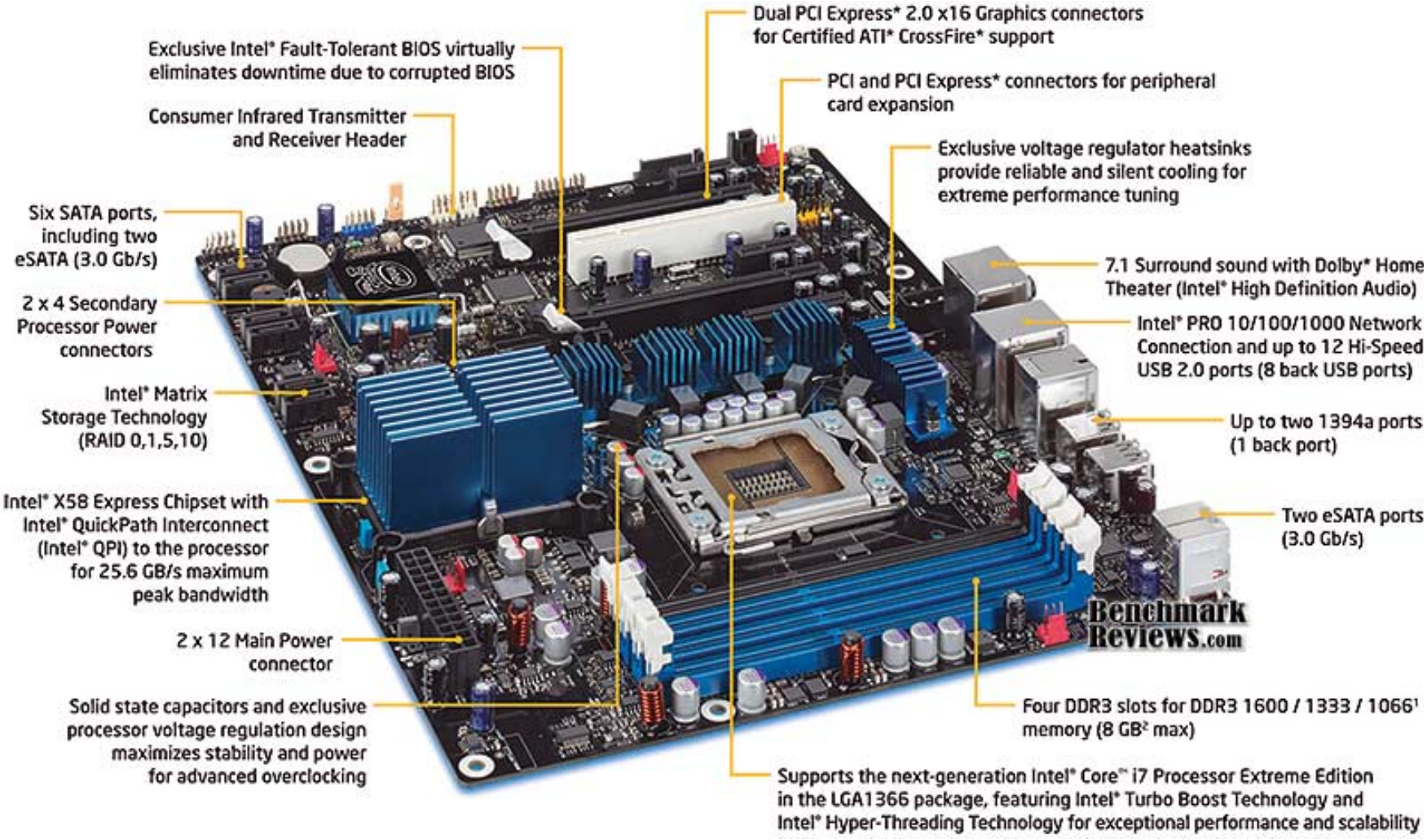
강의 일정

주	날짜	강의실 (화)	날짜	실습실 (목)
1	9월 2일	Intro	9월 4일	리눅스 개발환경 익히기
2	9월 16일	정수	9월 11일	**소수 1**
3	9월 23일	소수2, 어셈1 – move	9월 18일	GCC & Make
4	9월 30일	어셈2 – 제어문	9월 25일	Data lab
5	10월 7일	어셈3 – 프로시져 **I**	10월 9일	한글날/휴강
6	10월 14일	어셈3 – 프로시져 II	10월 16일	개천절(10월5일보강) GDB
7	10월 21일	보안	10월 23일	Binary bomb1
8	10월 28일	시험휴강	10월 30일	Binary bomb 2
9	11월 4일	프로세스 1	11월 6일	**Tiny shell 1**
10	11월 11일	**프로세스 2**	11월 13일	Tiny shell 2
11	11월 18일	시그널	11월 20일	Tiny shell 3
12	11월 25일	동적메모리 1	11월 27일	Malloc lab1
13	12월 2일	동적메모리 2	12월 4일	Malloc lab2
14	12월 9일	**기말고사**	12월 11일	Malloc lab3
15	12월 16일	**Wrap-up/종강**		

오늘 배울 내용

어셈블리어 프로그래밍 개관(교재 3.1~3.2)
데이터이동 명령어(MOV) (교재 3.3)

Intel i7 Quad-Core PC 머더보드



어셈블리어란 ?

어셈블리어란 ?

- 기계어에 1:1 대응관계를 갖는 명령어로 이루어진 low-level 프로그래밍 언어

어셈블리어와 프로그래머

C 언어로 프로그램을 작성할 때는 프로그램이 어떻게 내부적으로 구현되는지 알기 어렵다

어셈블리어로 프로그램을 작성할 때는 프로그래머는 프로그램이 어떻게 메모리를 이용하는지, 어떤 명령어를 이용하는지를 정확히 표시해야 한다.

물론 고급 언어로 프로그램으로 프로그램할 때가 대개의 경우 보다 안전하고, 편리하다

게다가 최근의 Optimizing compiler들은 웬만한 전문 어셈블리 프로그래머가 짠 프로그램보다 더 훌륭한 어셈블리 프로그램을 생성해 준다.

Q. 그렇다면, 왜 어셈블리어를 배워야 할까?

고급언어와 어셈블리어

고급언어의 특성

- 대형 프로그램을 개발하기에 편리한 구조체, 문법을 제공
- 이식성이 높음 High Portability
- 비효율적 실행파일이 생성될 가능성이 높음
- 대형 실용 응용프로그램 개발 시에 이용됨

어셈블리어의 특성

- 대형 프로그램을 개발하기에 불편함
- 속도가 중요한 응용프로그램 또는 하드웨어를 직접제어할 필요가 있는 경우에 이용
- 임베디드 시스템의 초기 코드 개발시에 이용
- 플랫폼마다 새롭게 작성되어야 함. 따라서 이식성이 매우 낮음
- 그러나, 많은 간접적인 응용이 있음 (?)

3장에서는

드디어 어셈블리어를 하나 배운다 – IA32

C 언어가 어떻게 기계어로 번역되는지 배운다

어셈블리어 프로그래밍 기술과 어셈블리어를 이해하는 방법을 배운다

IA32 프로세서 processors

PC 시장의 최강자!

진화형태의 설계 Evolutionary Design

- 1978년 8086 으로부터 시작 – 기억하는가 16비트 IBM PC
- 점차 새로운 기능을 추가
- 그러나, 예전의 기능들을 그대로 유지 (사용하지 않을지라도. 왜?)

Complex Instruction Set Computer (CISC)

- 다양한 명령어 형태의 다양한 명령어를 가짐
  - 과연 다 배울 수 있을까?
- RISC와 비슷한 성능을 내기 어려움
- 그러나, Intel이 해냈다!

x86 변천사 : 프로그래머의 관점에서

Name	Date	Transistors
8086	1978	29K

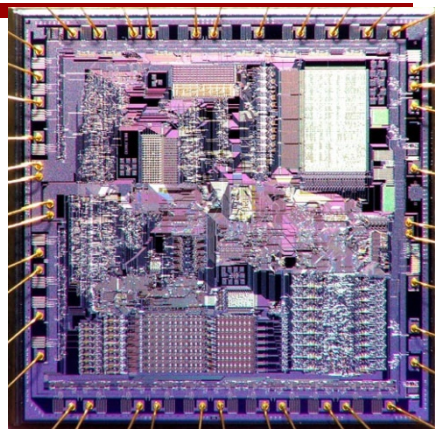
- 16-bit processor. IBM PC & DOS 사용
- 1MB 주소공간 address space. DOS 는 640K만을 허용 (기억하나?)

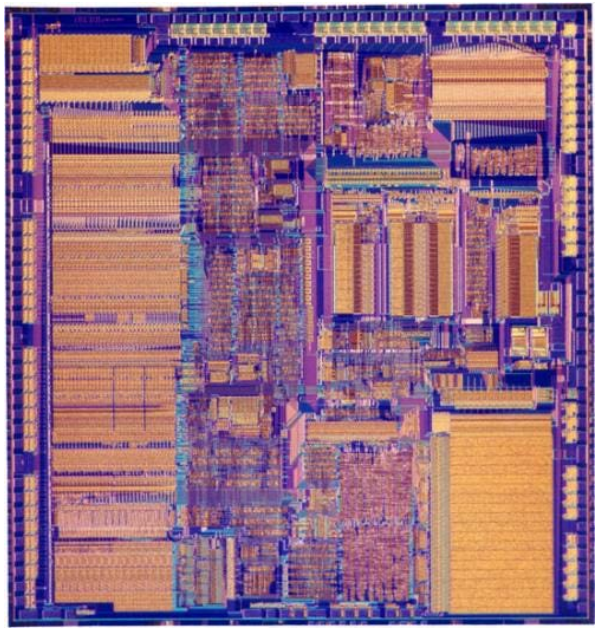
80286	1982	134K

- 다양한 새로운 주소지정 방식 추가. 그러나 별로 쓸데 없음
- IBM PC-AT 와 Windows 에 많이 사용됨

386	1985	275K

- 32 비트 프로세서. “flat addressing” 기능 추가
- Unix 도 사용할 수 있음
- IA32라고 불림

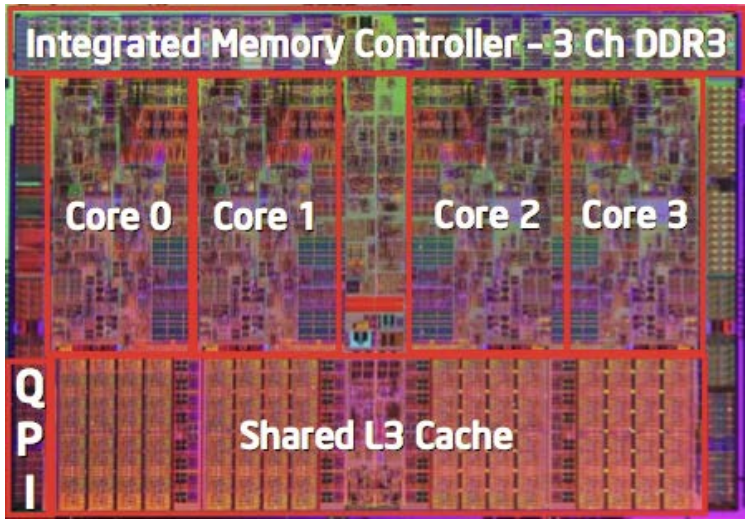




x86 변천사 : 프로그래머의 관점에서

프로세서의 진화

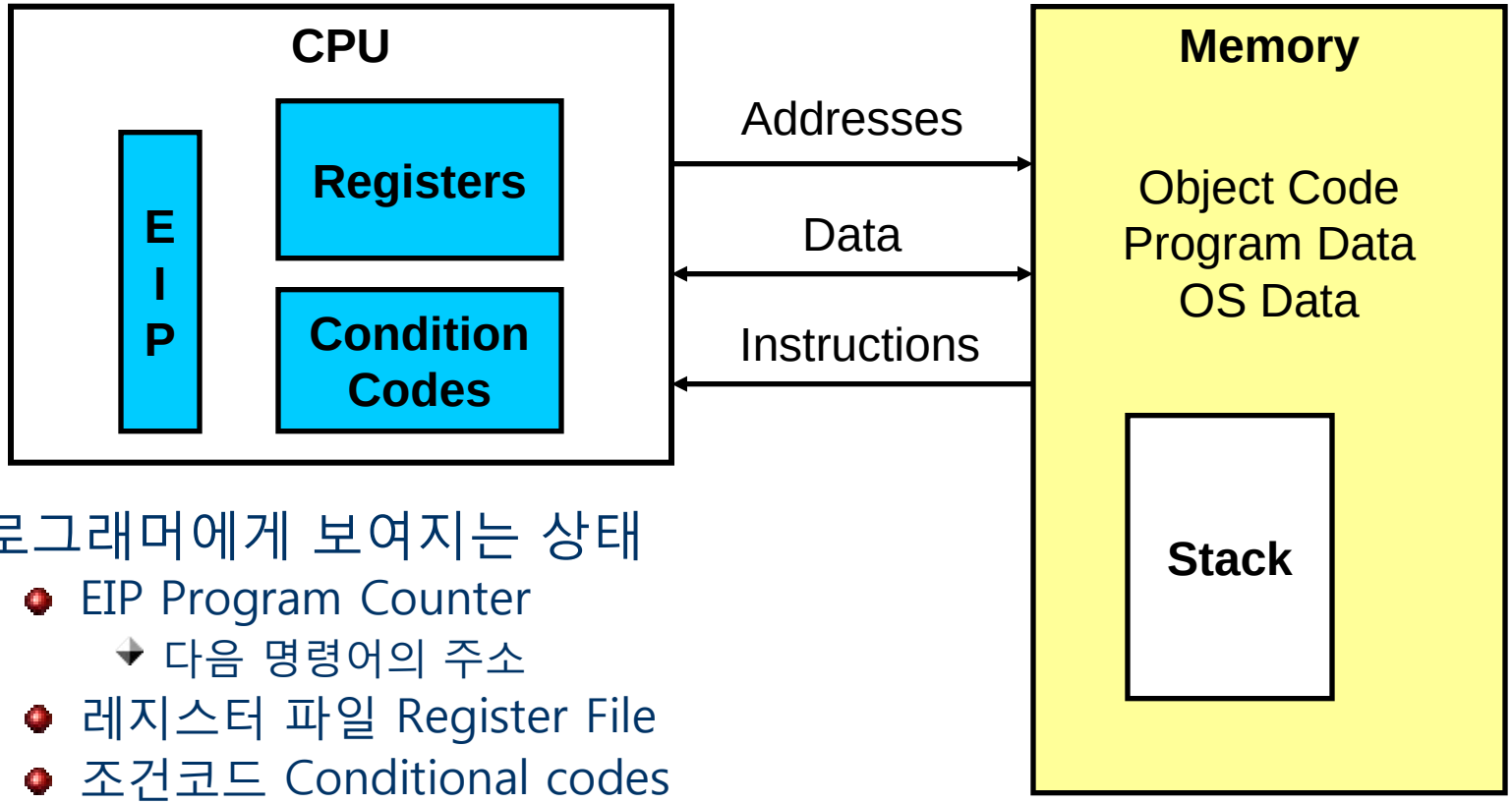
- 486 1989 1.9M
- Pentium 1993 3.1M
- Pentium/MMX 1997 4.5M
- PentiumPro 1995 6.5M
- Pentium III 1999 8.2M
- Pentium 4 2001 42M
- Core Duo 2006 291M – first multicore
- Core i7 2008 781M - Hyperthreading, multicore



추가된 특징

- 멀티미디어 연산 지원 명령어
  - 1, 2, 4바이트 데이터의 병렬 처리 연산 가능
- 효율적인 분기 명령어 => 이럴 필요가 있을까?

어셈블리 프로그래머의 시야



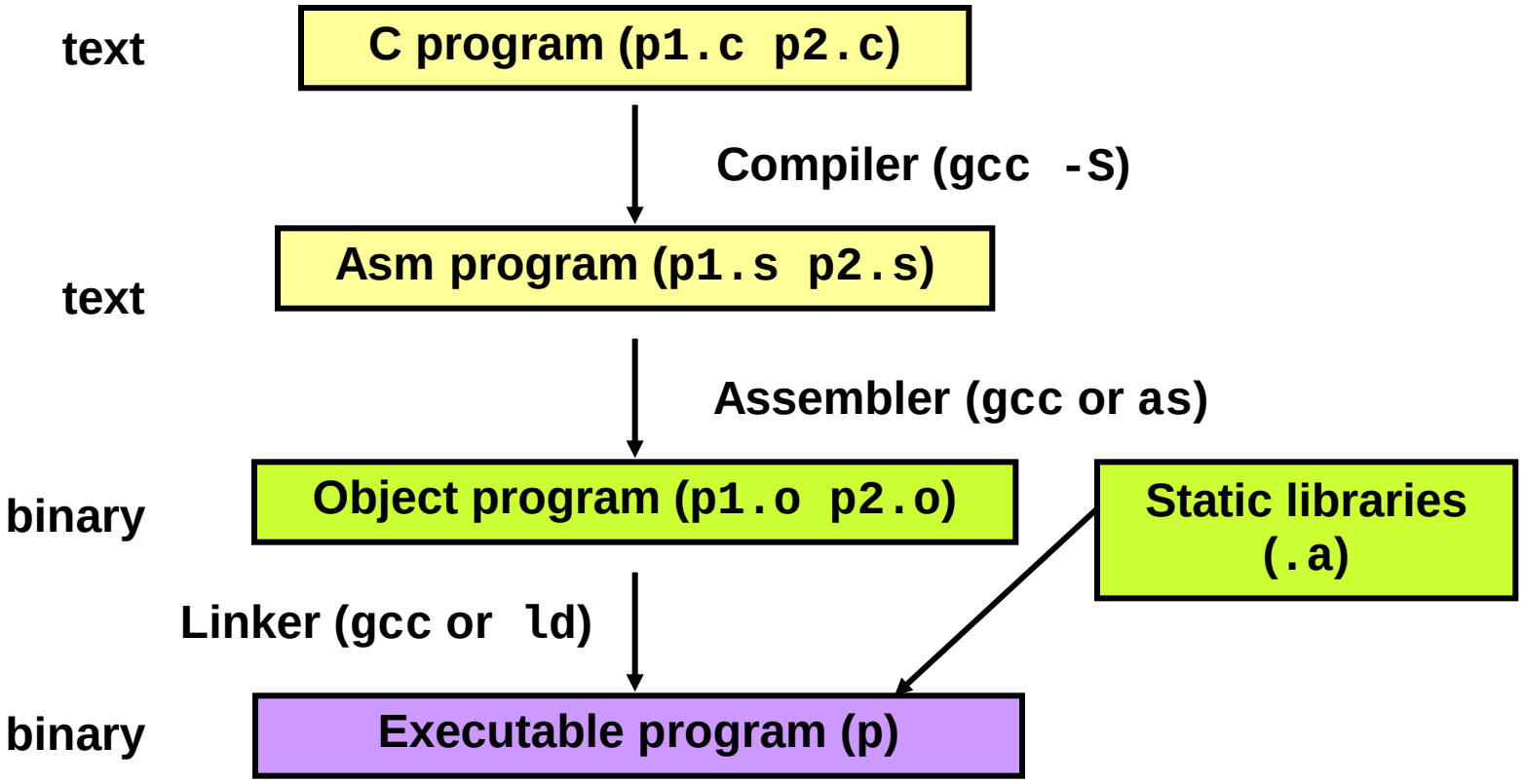
프로그래머에게 보여지는 상태

- EIP Program Counter
  - 다음 명령어의 주소
- 레지스터 파일 Register File
- 조건코드 Conditional codes
  - 가장 최근의 연산의 결과로 인한 상태정보를 저장
  - 조건형 분기명령에서 이용됨

- **메모리 Memory**
  - **바이트 주소 가능 데이터 배열**
  - **명령어, 데이터가 저장**
  - **스택이 위치**

C 프로그램의 목적코드로 변환과정

- 프로그램 파일들 `p1.c p2.c`
- 컴파일 명령: `gcc –O2 p1.c p2.c -o p`
  - 최적화 옵션 optimizations (`-O`)
  - 바이너리 데이터를 `p` 에 저장



C 프로그램을 어셈블리어로 컴파일하기

C Code

```
int sum(int x, int y)
{
  int t = x+y;
  return t;
}
```

생성된 어셈블리 프로그램

```
_sum:
    pushl %ebp
    movl %esp,%ebp
    movl 12(%ebp),%eax
    addl 8(%ebp),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

다음의 명령으로 생성

```
gcc -O -S code.c
```

code.s 파일이 만들어짐

어셈블리어의 특징

데이터 타입이 단순하다
- “Integer” data of 1, 2, or 4 bytes
  - 데이터 값 Data values
  - 주소 Addresses (untyped pointers)
- 실수형 데이터 Floating point data of 4, 8, or 10 bytes
- 배열이나 구조체가 없다 arrays or structures
  - 메모리에서의 연속적인 바이트들로 표시

연산이 기초적이다
- 레지스터나 메모리의 데이터를 이용하여 산술연산을 수행한다
- 레지스터나 메모리간의 데이터를 이동한다
  - 메모리로부터 레지스터로 데이터를 이동
  - 레지스터의 데이터를 메모리에 저장
- 제어기능
  - 무조건형 점프 Unconditional jumps to/from procedures
  - 조건형 분기 Conditional branches

목적코드 Object code

Code for sum

```
0x401040 <sum>:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x89
    0xec
    0x5d
    0xc3
```

- **Total of 13 bytes**
- **Each instruction 1, 2, or 3 bytes**
- **Starts at address `0x401040`**

어셈블러 Assembler

- `.s` 파일을 `.o` 로 번역한다
- 각 명령어들을 이진수의 형태로 변경
- 거의 실행파일과 유사
- 다수의 파일의 경우 연결되지 않은 형태

링커 Linker

- 파일간의 상호참조를 수행
- 정적라이브러리를 연결해줌 static run-time libraries
  - E.g., code for `malloc`, `printf`
- 동적 링크 *dynamically linked*
  - 프로그램 실행시 코드가 연결됨

목적코드의 역어셈블(disassembling)

Disassembled

```
00401040 <_sum>:
   0:      55                   push   %ebp
   1:      89 e5                mov    %esp,%ebp
   3:      8b 45 0c             mov    0xc(%ebp),%eax
   6:      03 45 08             add    0x8(%ebp),%eax
   9:      89 ec                mov    %ebp,%esp
   b:      5d                   pop    %ebp
   c:      c3                   ret
   d:      8d 76 00             lea    0x0(%esi),%esi
```

Disassembler

`objdump -d p`

- 목적코드의 분석에 유용한 도구
- 명령어들의 비트 패턴을 분석
- 개략적인 어셈블리언어로의 번역 수행
- `a.out` (실행파일) or `.o` file 에 적용할 수 있음

또 다른 Disassembly

Object

```
0x401040:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x89
    0xec
    0x5d
    0xc3
```

Disassembled

```
0x401040 <sum>:      push   %ebp
0x401041 <sum+1>:    mov    %esp,%ebp
0x401043 <sum+3>:    mov    0xc(%ebp),%eax
0x401046 <sum+6>:    add    0x8(%ebp),%eax
0x401049 <sum+9>:    mov    %ebp,%esp
0x40104b <sum+11>:   pop    %ebp
0x40104c <sum+12>:   ret
0x40104d <sum+13>:   lea    0x0(%esi),%esi
```

- **gdb 디버거의 사용**

  `gdb p`

  `disassemble sum`

  - **Disassemble procedure**

  `x/13b sum`

  - `sum` **에서 시작하여** `13`**바이트를 표시하라는 명령**

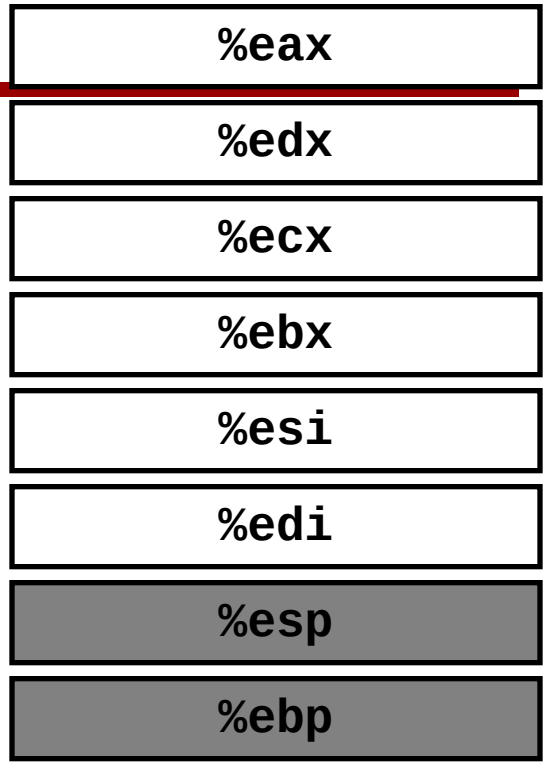
펜티엄의 정수 레지스터(IA32)

데이터 이동명령

명령어	효과	설명
movl S, D movw S, D	D <- S D <- S	Move double word Move word
movb S, D	D <- S	Move byte
movsbl S, D	D <- SignExtend(S)	Move sign-extended byte
movzbl S, D	D <- ZeroExtend(S)	Move zero-extended byte
pushl S	R[%esp] <- R[%esp]-4; M[R[%esp]] <- S	Push
popl D	D <- M[R[%esp]]; R[%esp] <- R[%esp]+4	Pop

스택에 대해 알아야 한다 !!!!

데이터 이동명령 MOV



데이터 이동하기

`movl` *Source*,*Dest*:

- Move 4-byte (“long”) word
- 자주 사용하는 명령어

오퍼랜드 형태 Operand Types

- Immediate: 상수 정수 데이터
  - ‘$’ 로 시작함
  - E.g., `$0x400`, `$-533`
  - 1, 2, or 4 바이트 가능
- Register: 우측의 8개의 레지스터를 이용
  - %esp, %ebp 는 특별한 용도로 사용함 (?)
- Memory: 4 바이트
  - 어드레스 모드에 따라 다름 “address modes”

movl 오퍼랜드 사용

	Source	Dest	Src,Dest	C 언어
movl	*Imm*	*Reg*	`movl $0x4,%eax`	`temp = 0x4;`
		Mem	`movl $-147,(%eax)`	`*p = -147;`
	Reg	*Reg*	`movl %eax,%edx`	`temp2 = temp1;`
		Mem	`movl %eax,(%edx)`	`*p = temp;`
	Mem	*Reg*	`movl (%eax),%edx`	`temp = *p;`

명령어 한 개로 memory-memory 데이터 이동을 할 수 없다

스택 연산 Push & Pop

Initially

%eax	0x123
%edx	0
%esp	0x108

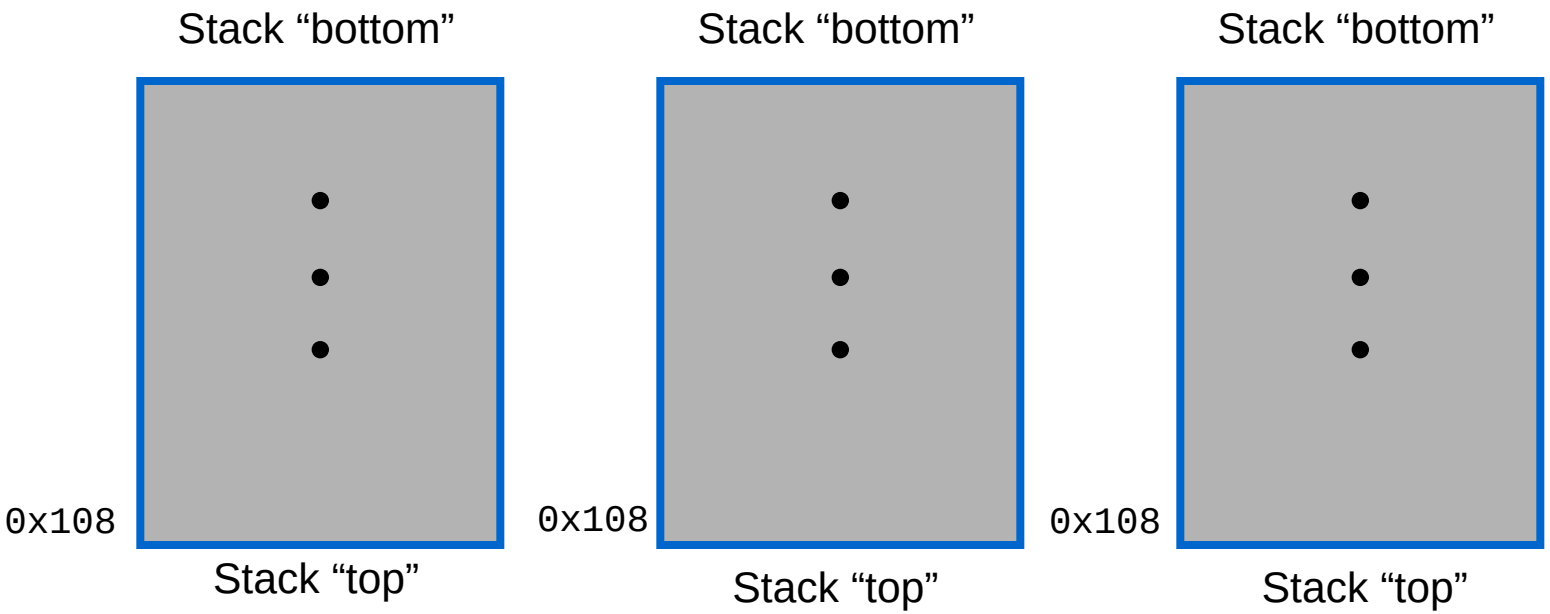
D1

pushl %eax

%eax	
%edx	
%esp	

popl %edx

%eax	
%edx	
%esp	

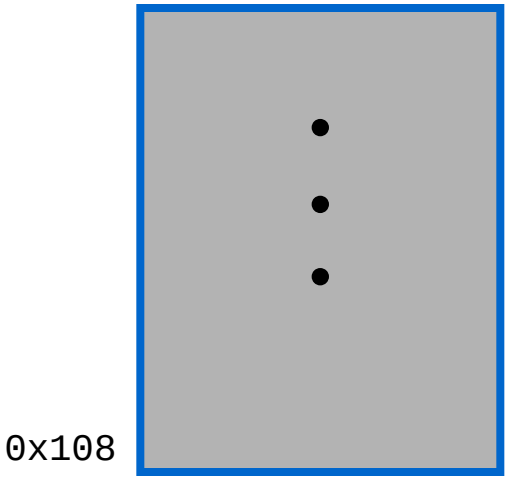


스택 연산 Push & Pop

Initially

%eax	0x123
%edx	0
%esp	0x108

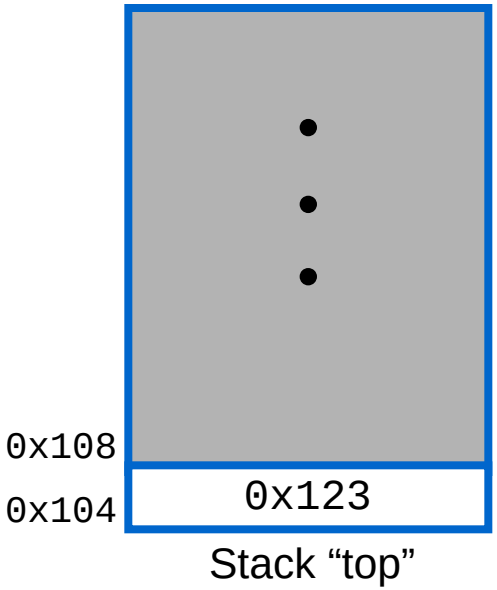
Stack “bottom”



pushl %eax

%eax	0x123
%edx	0
%esp	0x104

Stack “bottom”

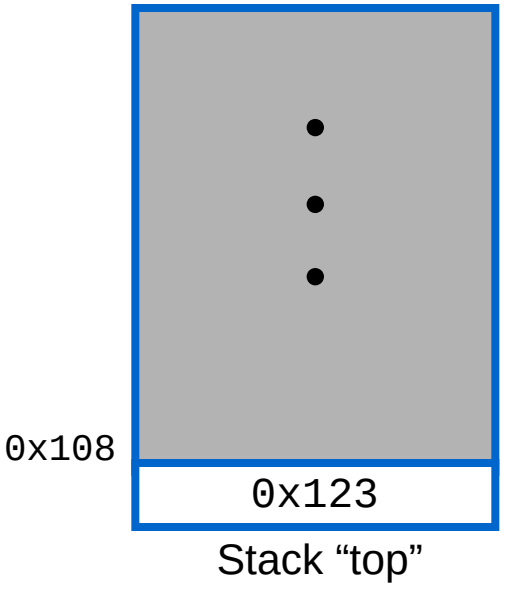


D2

popl %edx

%eax	
%edx	
%esp	

Stack “bottom”

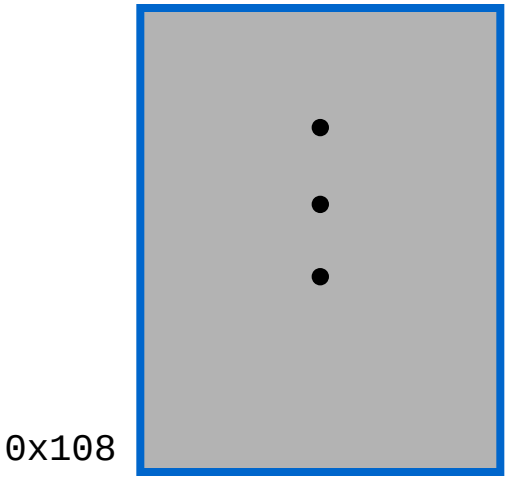


스택 연산 Push & Pop

Initially

%eax	0x123
%edx	0
%esp	0x108

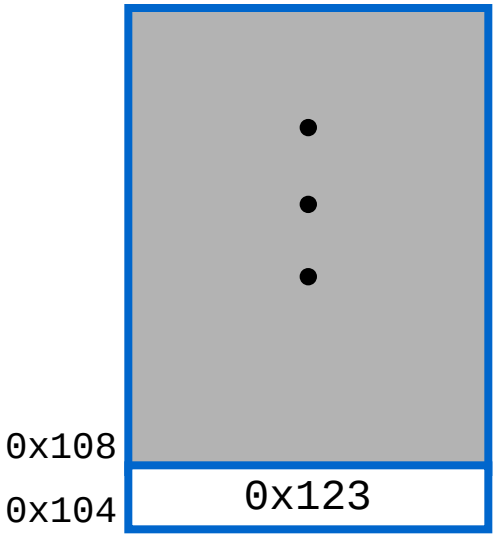
Stack “bottom”



pushl %eax

%eax	0x123
%edx	0
%esp	0x104

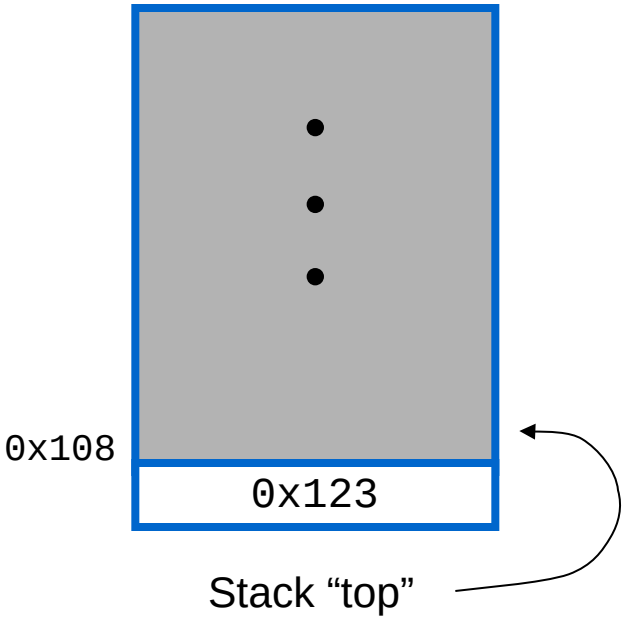
Stack “bottom”



popl %edx

%eax	0x123
%edx	0x123
%esp	0x108

Stack “bottom”



단순 어드레싱 모드 용어의 이해

일반 접근 (R) : Mem[Reg[R]]

- 레지스터 R 은 메모리 주소를 저장하고 있다

```
movl (%ecx),%eax
```

이동 접근 D(R) : Mem[Reg[R]+D]

- 레지스터 R 메모리 블럭의 시작주소를 저장하고 있다
- 상수 변위 D 는 오프셋offset을 의미한다

```
movl 8(%ebp),%edx
```

인덱스형 주소지정 모드

가장 일반적인 형태

D(Rb,Ri,S) : <- Mem[Reg[Rb]+S*Reg[Ri]+ D]

- D: 상수값 "변위"
- Rb: 베이스 레지스터 : 8 개의 정수 레지스터중의 한 개
- Ri: 인덱스 레지스터 : %esp를 제외한 모든 레지스터
  - %ebp **도 거의 사용하지 않음**
- S: 배율: 1, 2, 4, or 8

예제

(Rb,Ri)	Mem[Reg[Rb]+Reg[Ri]]
D(Rb,Ri)	Mem[Reg[Rb]+Reg[Ri]+D]
(Rb,Ri,S)	Mem[Reg[Rb]+S*Reg[Ri]]

주소계산 예제

%edx	0xf000
%ecx	0x100

Expression	Computation	Address
0x8(%edx)	0xf000 + 0x8	0xf008
(%edx,%ecx)	0xf000 + 0x100	0xf100
(%edx,%ecx,4)	0xf000 + 4*0x100	0xf400
0x80(,%edx,2)	2*0xf000 + 0x80	0x1e080

연습문제 1. 데이터의 접근

다음과 같은 값들이 표시된 메모리 주소와 레지스터에 저장되어 있다.

Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Register	Value
%eax	0x100
%ecx	0x1
%edx	0x3

다음에 나오는 표에 지시된 오퍼랜드의 값을 채우시오.

Operand	Value
%eax	_____
0x104	_____
$0x108	_____
(%eax)	_____
4(%eax)	_____
9(%eax,%edx)	_____
260(%ecx,%edx)	_____
0xFC(,%ecx,4)	_____
(%eax,%edx,4)	_____

데이터이동 명령예제

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

Q1. t0 = *xp ?
Q2. *xp = t1 ?
Q3. main()**을 작성해 보시오.**

```
swap:
    pushl %ebp
    movl  %esp,%ebp
    pushl %ebx

    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)

    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

Set Up

Body

Finish

데이터이동 명령예제

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    pushl %ebp
    movl  %esp,%ebp
    pushl %ebx

    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)

    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

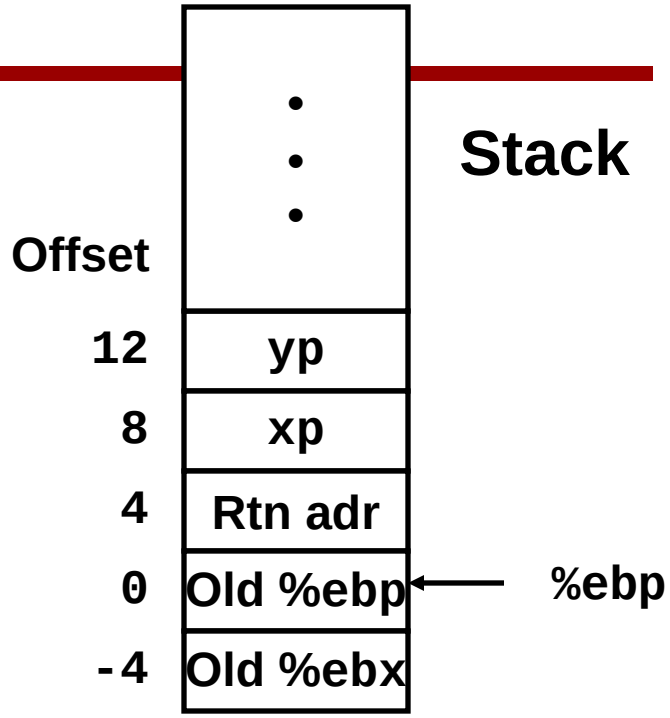
Set Up

Body

Finish

Swap 함수의 이해

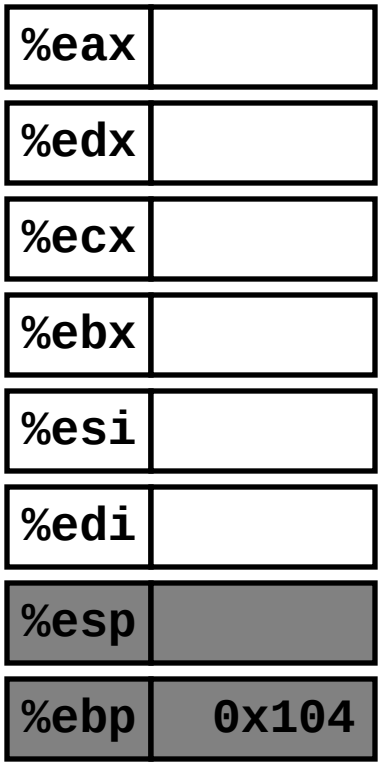
```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

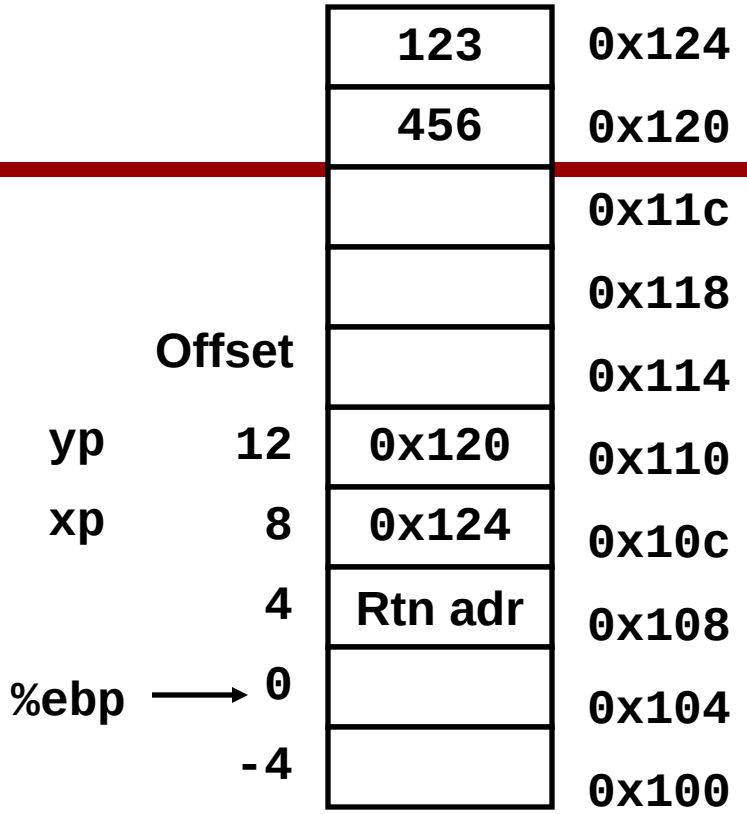


Register	Variable
%ecx	yp
%edx	xp
%eax	t1
%ebx	t0

```
movl 12(%ebp),%ecx  # ecx = yp
movl 8(%ebp),%edx   # edx = xp
movl (%ecx),%eax    # eax = *yp (t1)
movl (%edx),%ebx    # ebx = *xp (t0)
movl %eax,(%edx)    # *xp = eax
movl %ebx,(%ecx)    # *yp = ebx
```

Swap 함수의 이해

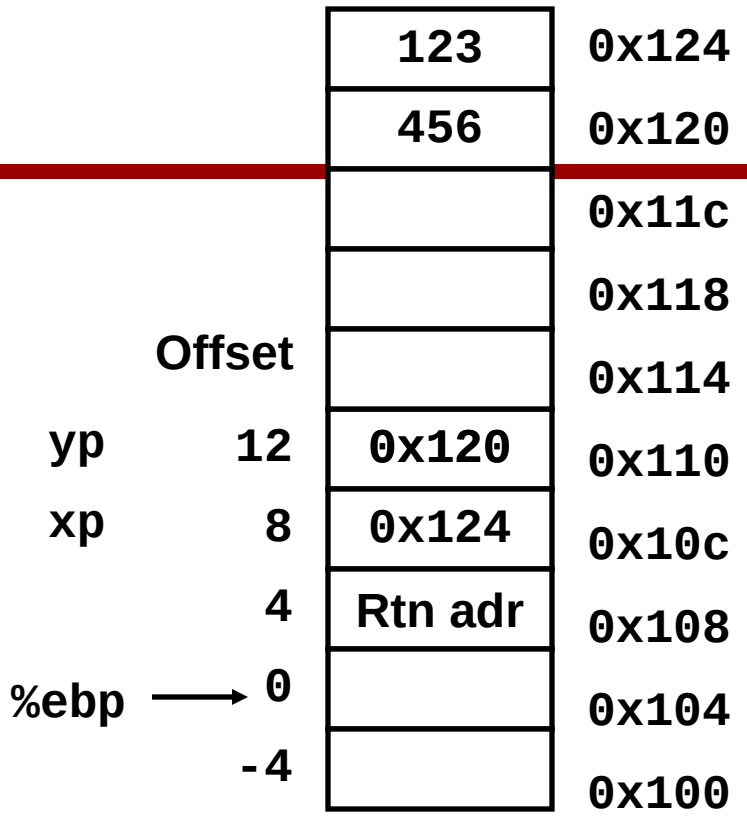





```
movl 12(%ebp),%ecx  # ecx = yp
movl 8(%ebp),%edx     # edx = xp
movl (%ecx),%eax      # eax = *yp (t1)
movl (%edx),%ebx      # ebx = *xp (t0)
movl %eax,(%edx)      # *xp = eax
movl %ebx,(%ecx)      # *yp = ebx
```

Swap 함수의 이해

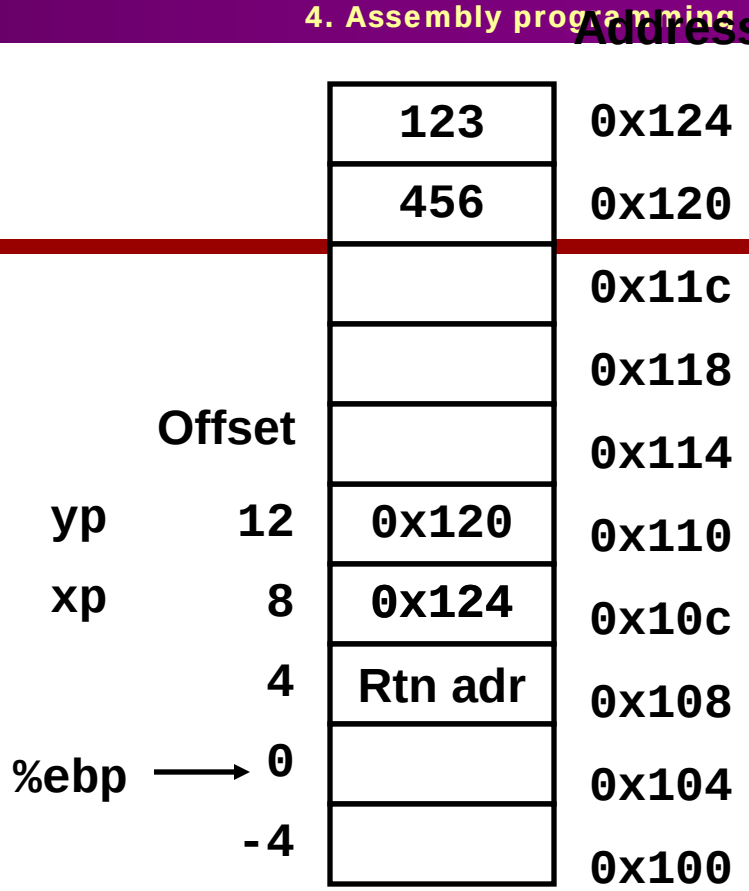
%eax	
%edx	
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104




```
movl 12(%ebp),%ecx  # ecx = yp
movl 8(%ebp),%edx       # edx = xp
movl (%ecx),%eax        # eax = *yp (t1)
movl (%edx),%ebx        # ebx = *xp (t0)
movl %eax,(%edx)        # *xp = eax
movl %ebx,(%ecx)        # *yp = ebx
```

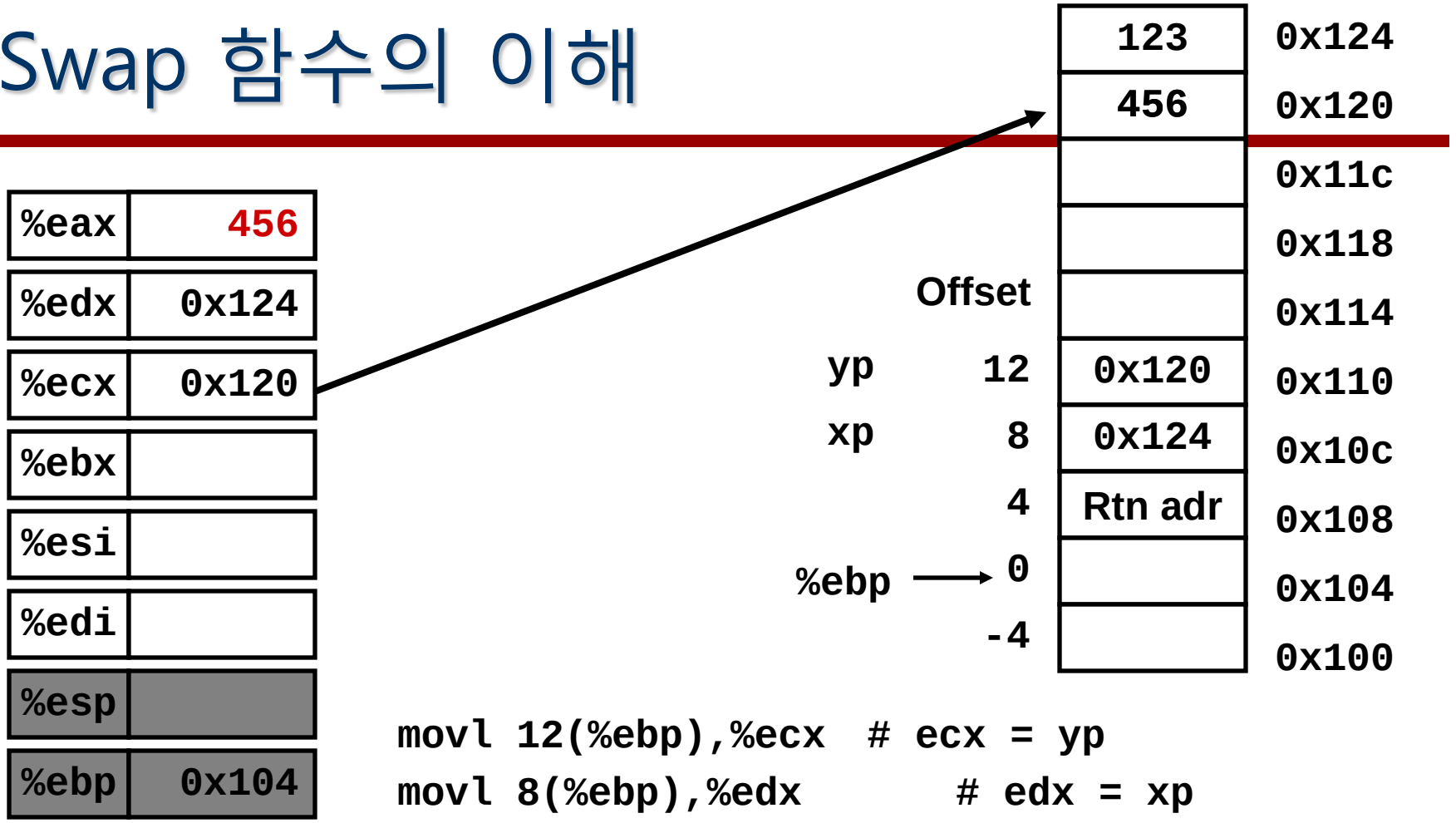
Swap 함수의 이해

%eax	
%edx	0x124
%ecx	0x120
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104



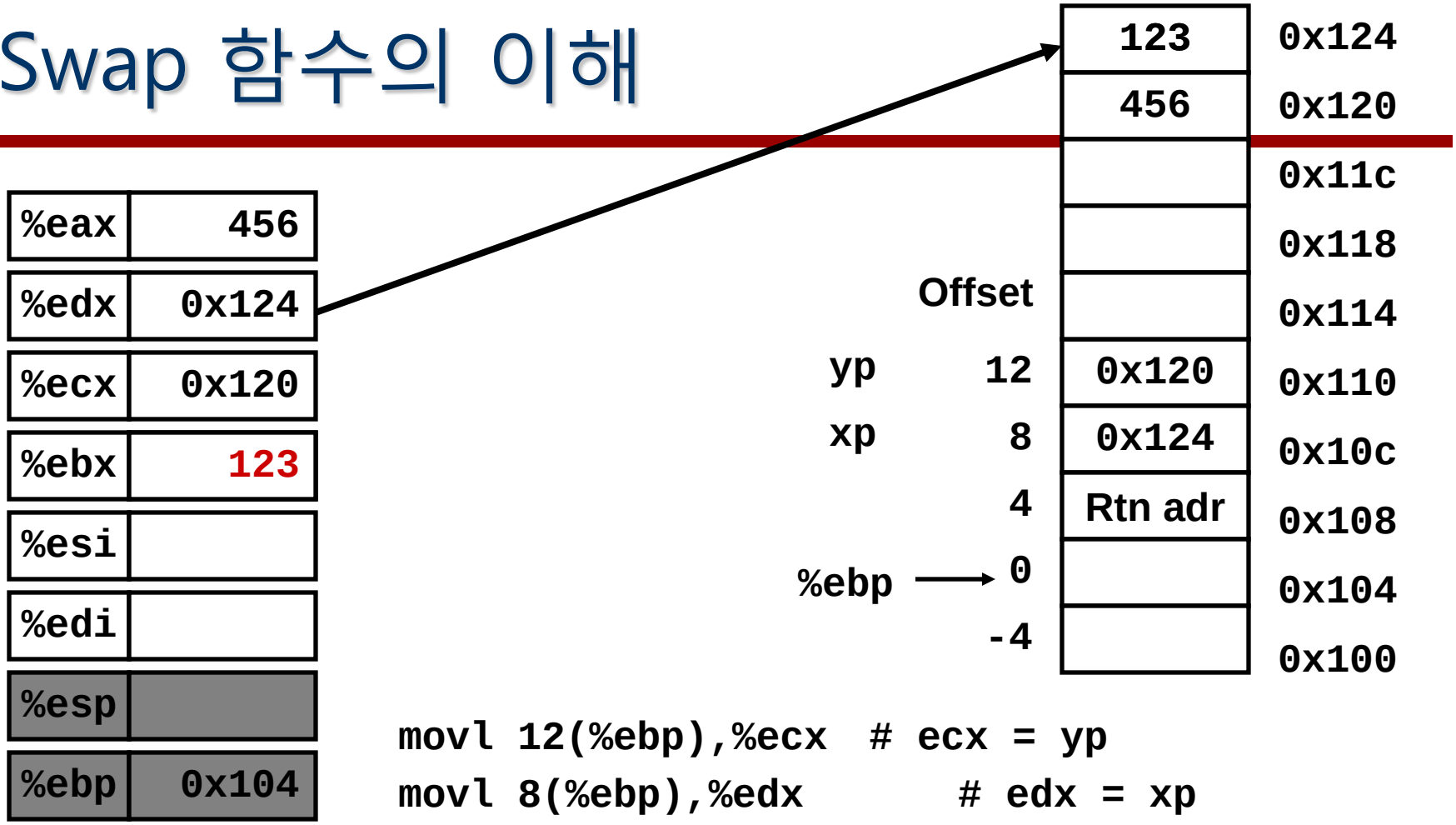

```
movl 12(%ebp),%ecx  # ecx = yp
movl 8(%ebp),%edx     # edx = xp
movl (%ecx),%eax      # eax = *yp (t1)
movl (%edx),%ebx      # ebx = *xp (t0)
movl %eax,(%edx)      # *xp = eax
movl %ebx,(%ecx)      # *yp = ebx
```

Swap 함수의 이해




```
movl 12(%ebp),%ecx  # ecx = yp
movl 8(%ebp),%edx       # edx = xp
movl (%ecx),%eax        # eax = *yp (t1)
movl (%edx),%ebx        # ebx = *xp (t0)
movl %eax,(%edx)        # *xp = eax
movl %ebx,(%ecx)        # *yp = ebx
```

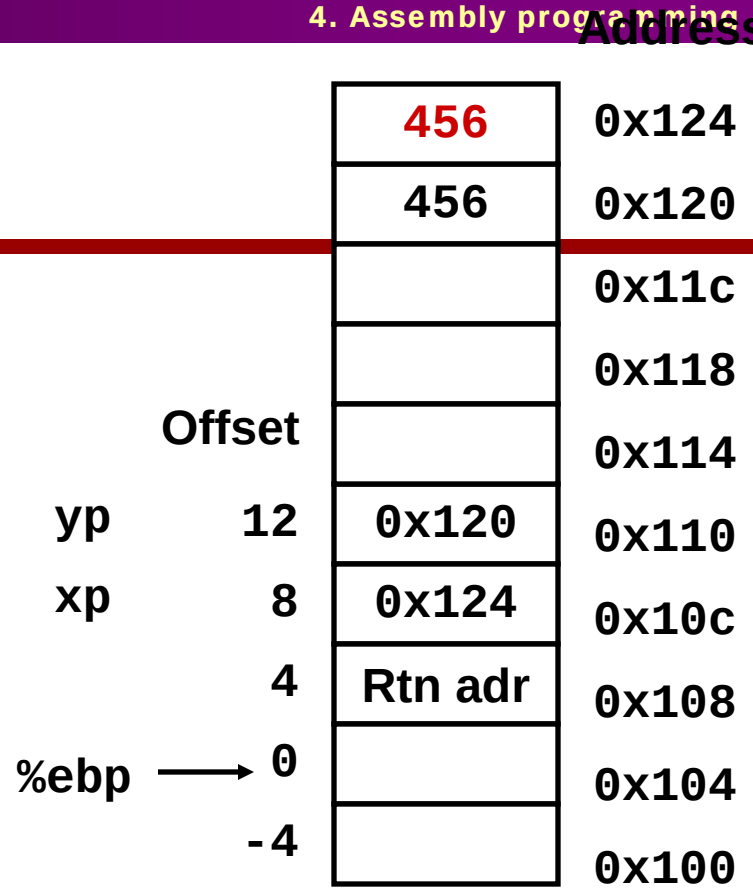
Swap 함수의 이해




```
movl 12(%ebp),%ecx  # ecx = yp
movl 8(%ebp),%edx     # edx = xp
movl (%ecx),%eax      # eax = *yp (t1)
movl (%edx),%ebx      # ebx = *xp (t0)
movl %eax,(%edx)      # *xp = eax
movl %ebx,(%ecx)      # *yp = ebx
```

Swap 함수의 이해

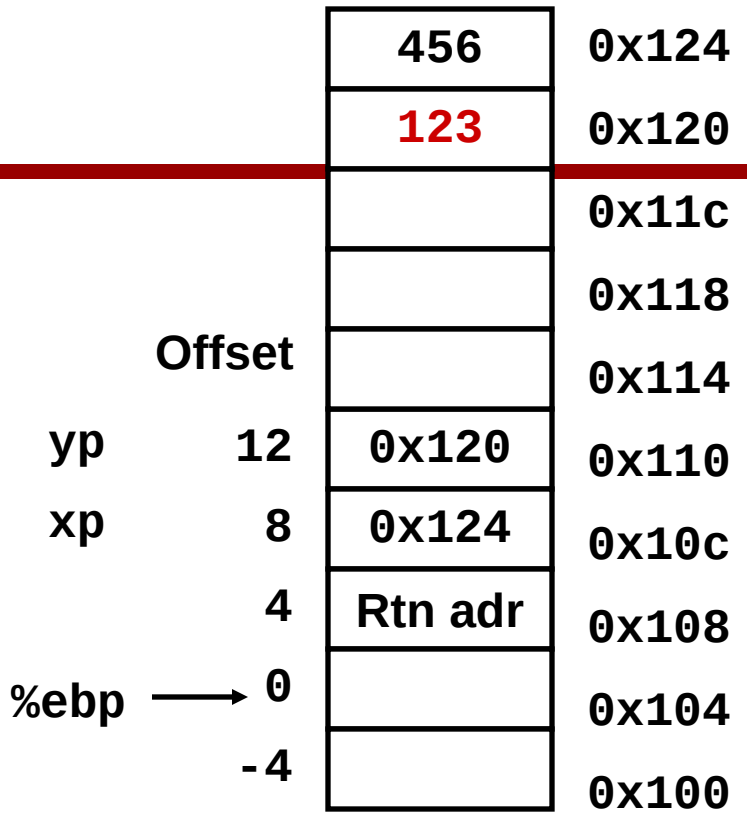
%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104




```
movl 12(%ebp),%ecx  # ecx = yp
movl 8(%ebp),%edx     # edx = xp
movl (%ecx),%eax      # eax = *yp (t1)
movl (%edx),%ebx      # ebx = *xp (t0)
movl %eax,(%edx)      # *xp = eax
movl %ebx,(%ecx)      # *yp = ebx
```

Swap 함수의 이해

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104




```
movl 12(%ebp),%ecx  # ecx = yp
movl 8(%ebp),%edx     # edx = xp
movl (%ecx),%eax      # eax = *yp (t1)
movl (%edx),%ebx      # ebx = *xp (t0)
movl %eax,(%edx)      # *xp = eax
movl %ebx,(%ecx)      # *yp = ebx
```

주소 계산을 위한 명령어

`leal` *Src*,*Dest* ; load effective address

- *Src* 어드레스 모드 수식
- *Dest* 어드레스 모드 계산에 의한 최종 주소값이 기록됨

용도

- 메모리 접근 없이 주소를 계산하는 경우
  - E.g., `p = &x[i];`
- x + k*y 형태의 수식을 계산하게 됨
  - k = 1, 2, 4, or 8.

연산 명령어

Instruction		Effect	Description
`leal`	S, D	$D \leftarrow \&S$	Load effective address
INC	D	$D \leftarrow D + 1$	Increment
DEC	D	$D \leftarrow D - 1$	Decrement
NEG	D	$D \leftarrow -D$	Negate
NOT	D	$D \leftarrow \sim D$	Complement
ADD	S, D	$D \leftarrow D + S$	Add
SUB	S, D	$D \leftarrow D - S$	Subtract
IMUL	S, D	$D \leftarrow D * S$	Multiply
XOR	S, D	$D \leftarrow D \hat{} S$	Exclusive-or
OR	S, D	$D \leftarrow D \mid S$	Or
AND	S, D	$D \leftarrow D \& S$	And
SAL	k, D	$D \leftarrow D << k$	Left shift
SHL	k, D	$D \leftarrow D << k$	Left shift (same as SAL)
SAR	k, D	$D \leftarrow D >>_A k$	Arithmetic right shift
SHR	k, D	$D \leftarrow D >>_L k$	Logical right shift

연습문제 2. leal 명령

eax = x, ecx = y 일때, edx를 x, y의 수식으로 표시하시오

```
leal    9(%eax, %ecx, 2), %edx
```

연습문제 3. 연산명령어

```
int shift_left2_rightn(int x, int n)
{
    x <<= 2;
    x >>= n;
    return x;
}
```

```
1    movl    8(%ebp), %eax    Get x
2    _____                    x <<= 2
3    movl    12(%ebp), %ecx   Get n
4    _____                    x >>= n
```

우측의 어셈블리코드는 좌측의 c 프로그램을 컴파일하는 과정에서 나온 코드이다. 빈 곳에 들어갈 어셈블리코드를 주석을 참조하여 작성하시오. 단, 우측 시프트는 산술시프트를 해야함.

연습문제 3. 데이터의 길이

다음 어셈블리 프로그램의 각 줄에서 오퍼랜드를 고려해서 인스트럭션의 접미어를 적절히 결정하시오.(movb, movw, movl 중에 선택)

```
1       mov   %eax, (%esp)
2       mov   (%eax), %dx
3       mov   $0xFF, %bl
4       mov   (%esp,%edx,4), %dh
5       push  $0xFF
6       mov   %dx, (%eax)
7       pop   %edi
```

요약

우리는 IA32 프로세서의 어셈블리어를 배운다

디스어셈블러를 이용하면 실행파일로부터 어셈블리를 볼 수 있다

IA32 mov 데이터 이동명령어를 사용할 수 있다

다음주 예습숙제 : 3.6~3.6.2, pp.219~223