CHUNGNAM NATIONAL UNIVERSITY

시스템 프로그래밍

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

2010년 9월 29일
http://eslab.cnu.ac.kr

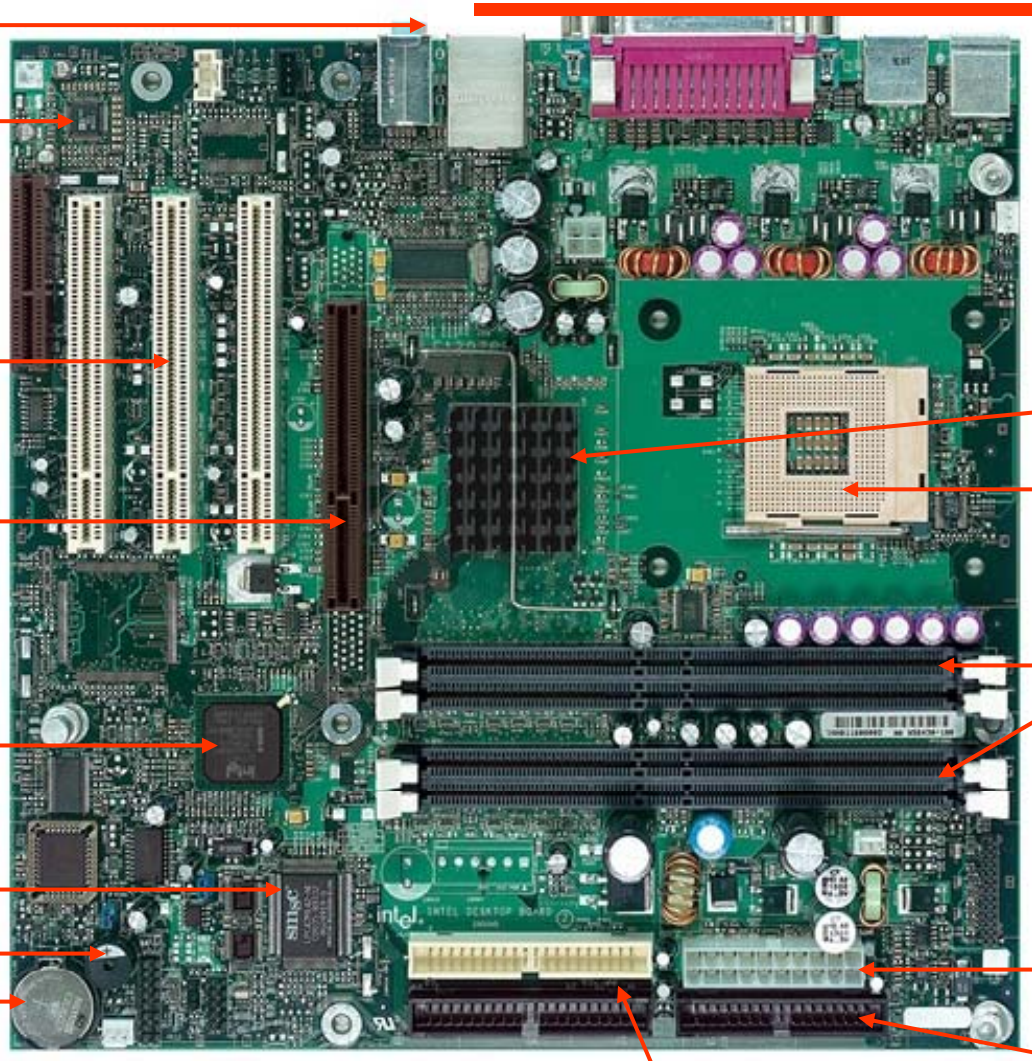
* Some slides are from Original slides of RBE

오늘 배울 내용

어셈블리어 프로그래밍 개관(교재 3.1~3.2)

데이터이동 명령어(MOV) (교재 3.3)

펜티엄 4 컴퓨터 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

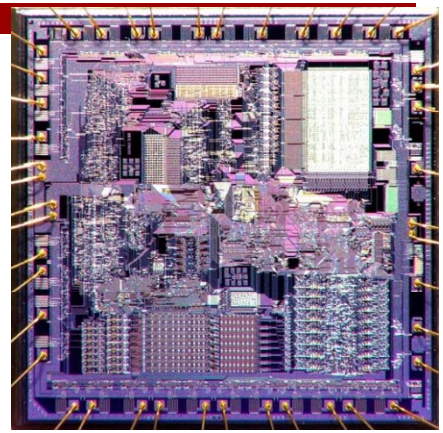
- 16-bit processor. IBM PC & DOS 사용
- 1MB 주소공간 address space. DOS 는 640K만을 허용 (기억하나?)

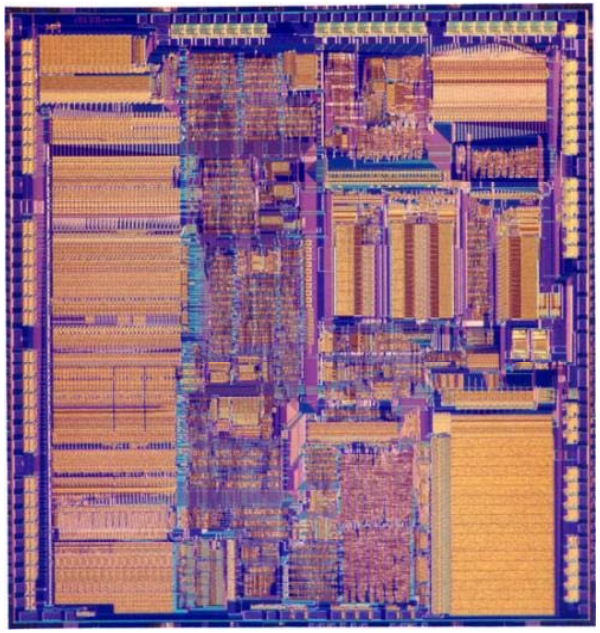
80286	1982	134K

- 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**

추가된 특징

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

새 제품 군: IA64

Name Date Transistors

Itanium 2001 10M

- 64 비트 아키텍춰로 확장
- 완전히 새로운 명령어들을 구현함
- IA32 프로그램들은 여전히 실행가능
- Hewlett-Packard와 공동개발

Itanium 2 2002 221M

- 성능 대폭향상

Itanium 2 Dual-Core 2006 1.7B

Itanium 은 아직 시판되지 않는다

- backward compatibility 가 없다
- 성능이 기대 이하

x86 변천 : 클론 (clone)

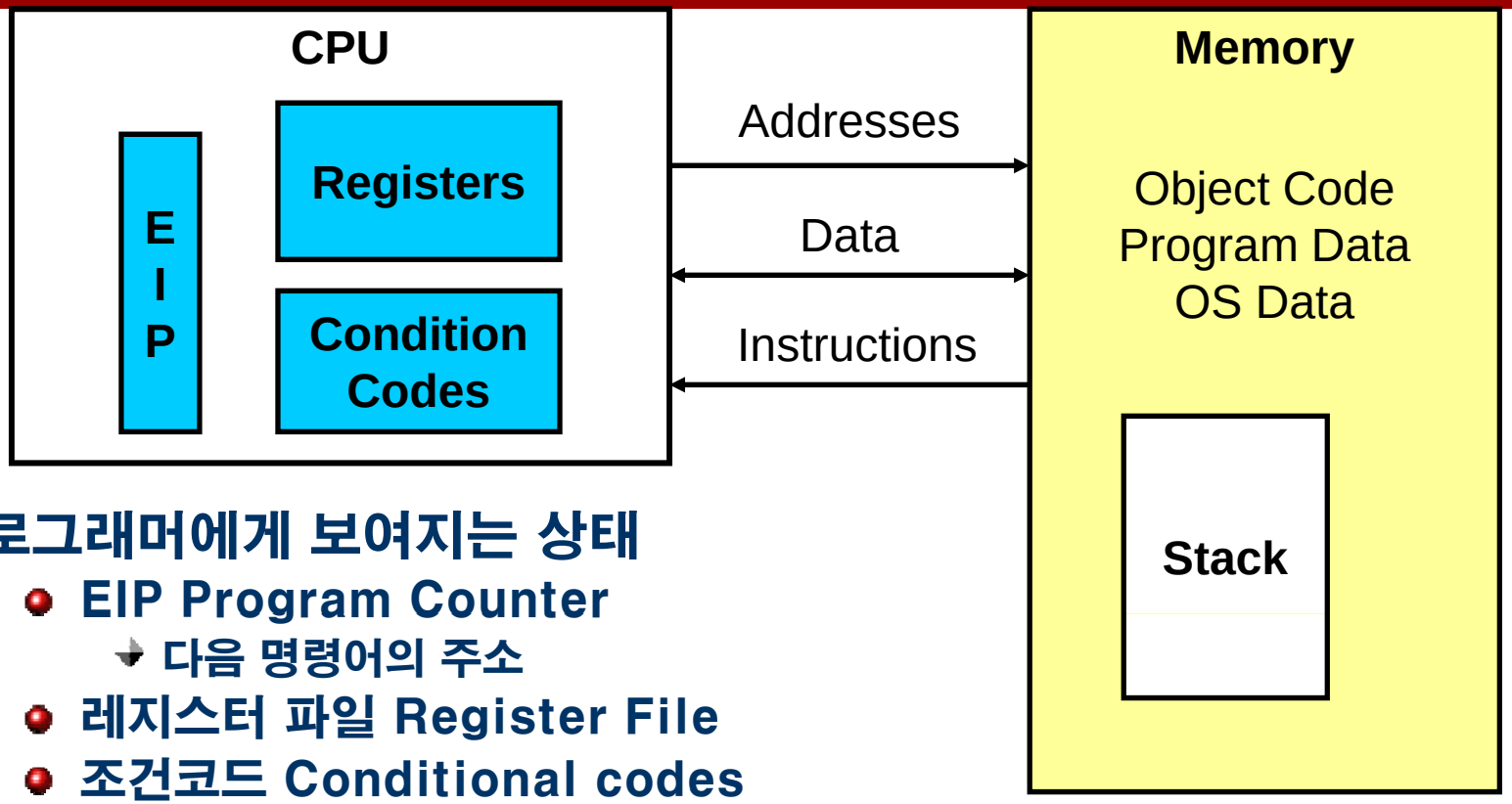
Advanced Micro Devices (AMD)

- 경과
  - AMD 는 Intel의 뒤를 계속 쫓아왔다
  - 인텔보다 조금 성능은 낮지만 값은 엄청 싸다
- 최근 소식
  - Digital Equipment Corp.로부터 고급 설계인력을 고용
  - 이제는 인텔의 가장 강력한 경쟁사
- 자체 64비트 확장 프로세서인 x86-64을 개발
  - 인텔의 고성능 서버 시장을 잠식하기 시작

인텔의 64비트 딜레마

- 인텔은 IA32 로부터 IA64로의 급작스런 변경을 시도함
  - 전혀 다른 새로운 구조임
  - IA32 명령을 원시적인 방법으로 실행함
  - 실망스러운 성능
- AMD 는 x86구조로 부터 점진적인 64비트 진입을 시도
  - x86-64
- 인텔은 IA64에 집중해야 하는 상황이 됨
  - AMD 프로세서가 우수하다는 것을 인정할 수 없음
- 2004: Intel 은 IA32에 EM64T 확장을 개발
  - Extended Memory 64-bit 기술 채택
  - 고성능 Pentium 4에 사용될 예정
  - x86-64과 거의 동일함

어셈블리 프로그래머의 시야



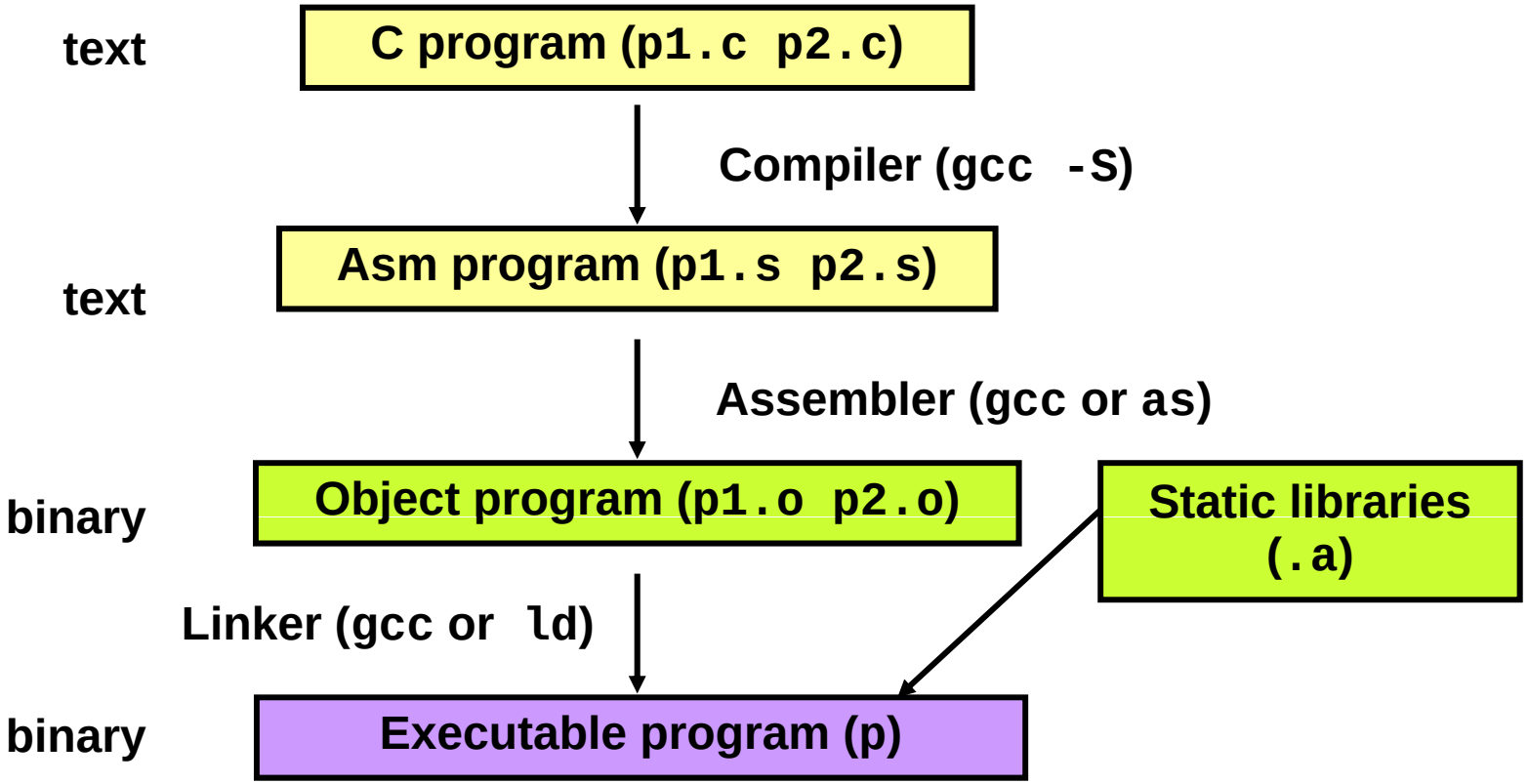
프로그래머에게 보여지는 상태

- 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 디스어셈블링

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 13바이트를 표시하라는 명령

디스어셈블 할 수 있는 파일은?

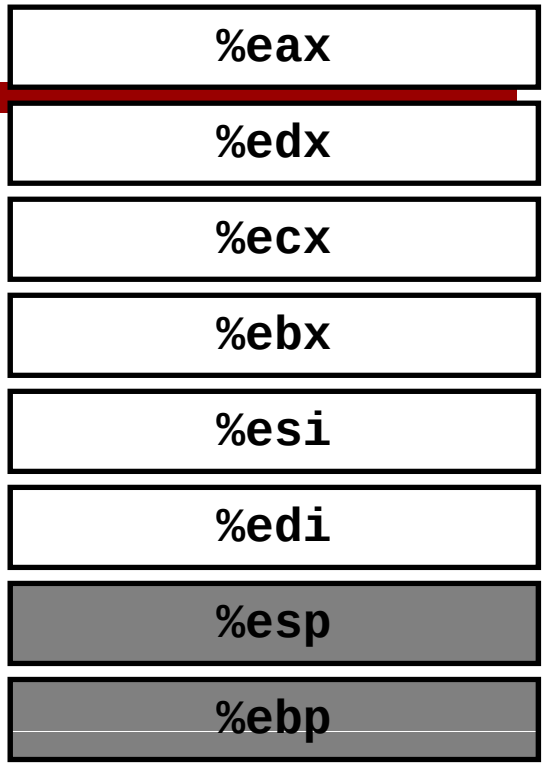
```
% objdump -d WINWORD.EXE

WINWORD.EXE:     file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:  55                   push   %ebp
30001001:  8b ec                mov    %esp,%ebp
30001003:  6a ff                push   $0xffffffff
30001005:  68 90 10 00 30       push   $0x30001090
3000100a:  68 91 dc 4c 30       push   $0x304cdc91
```

- 실행파일이라면 무엇이든 가능
- 디스어셈블러는 바이트들을 읽어 들여서 어셈블리어로 변환해 준다

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

데이터 이동명령 MOV



데이터 이동하기

`movl` ***Source,Dest***:

- Move 4-byte ("long") word
- 자주 사용하는 명령어

오퍼랜드 형태 Operand Types

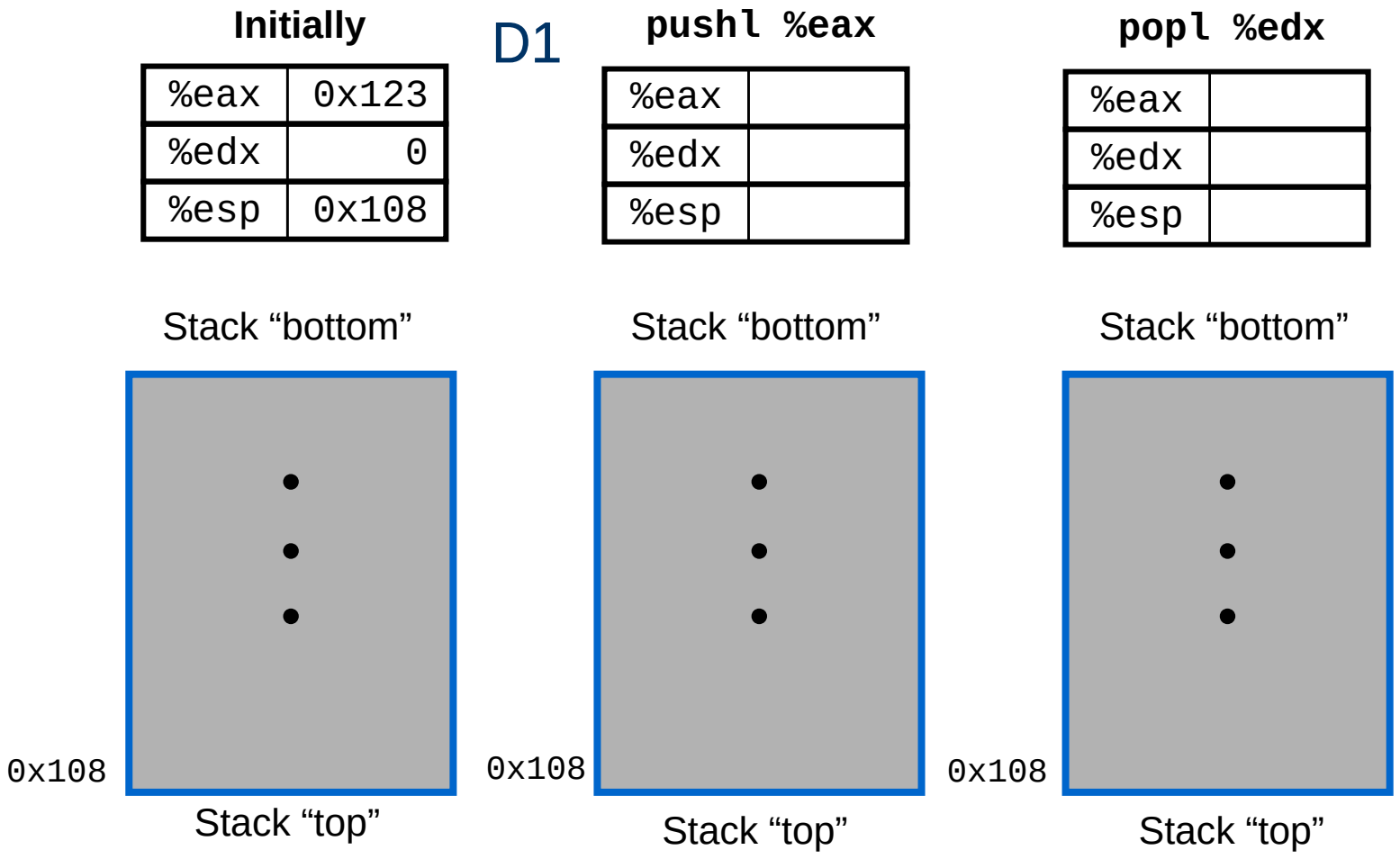
- Immediate: Constant integer data
  - '$' 로 시작함
  - E.g., `$0x400`, `$-533`
  - 1, 2, or 4 바이트 가능
- Register: 우측의 8개의 레지스터를 이용
  - `%esp`, `%ebp` 는 특별한 용도로 사용함 (?)
- Memory: 4 바이트
  - 어드레스 모드에 따라 다름 "address modes"

movl 오퍼랜드 사용

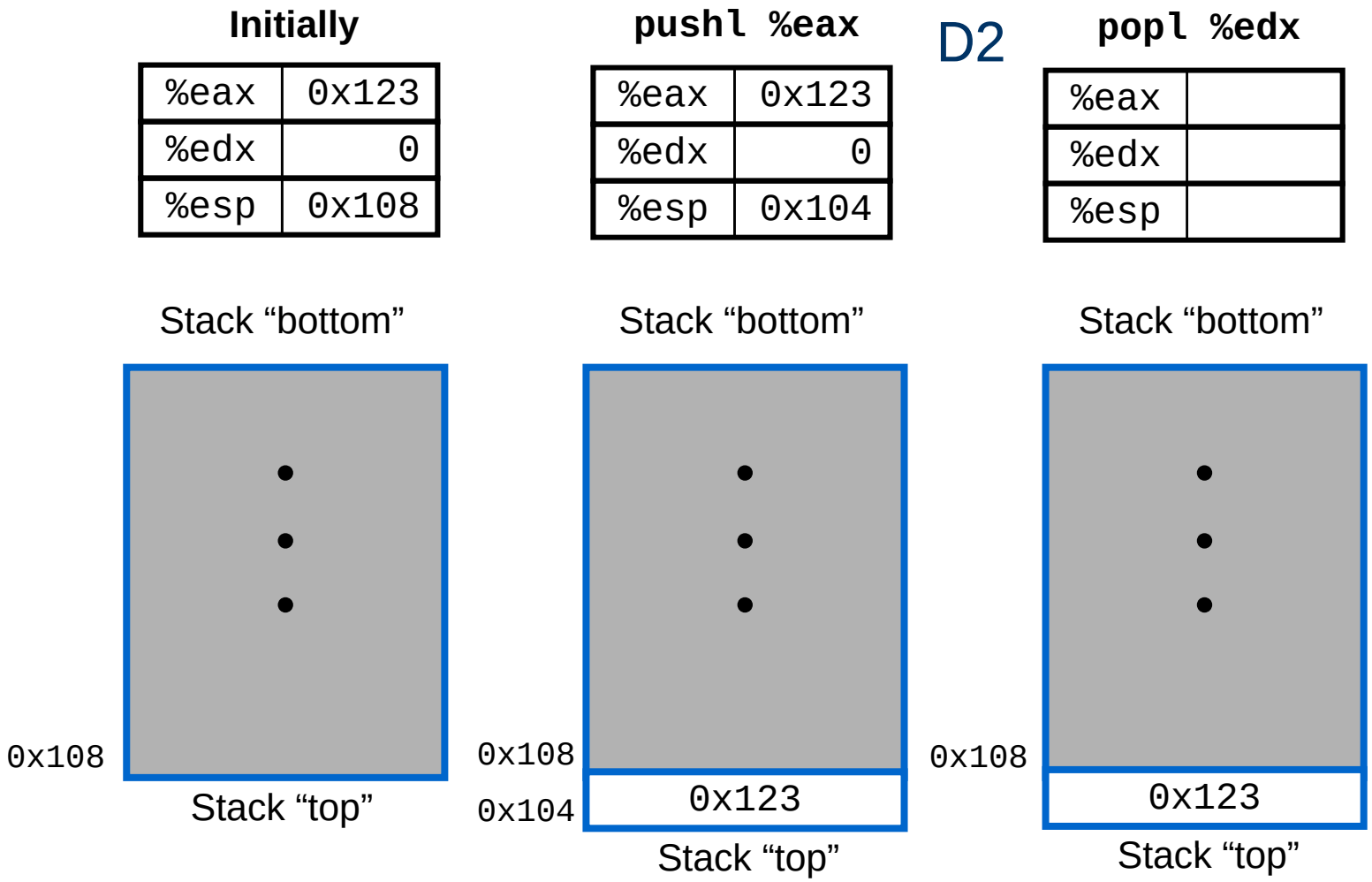
	Source	Dest	Src,Dest	C 언어
movl	*Imm*	*Reg*	`movl $0x4,%eax`	`temp = 0x4;`
		Mem	`movl $-147,(%eax)`	`*p = -147;`
	Reg	*Reg*	`movl %eax,%edx`	`temp2 = temp1;`
		Mem	`movl %eax,(%edx)`	`*p = temp;`
	Mem	*Reg*	`movl (%eax),%edx`	`temp = *p;`

명령어 한 개로 memory-memory 데이터 이동을 할 수 없다

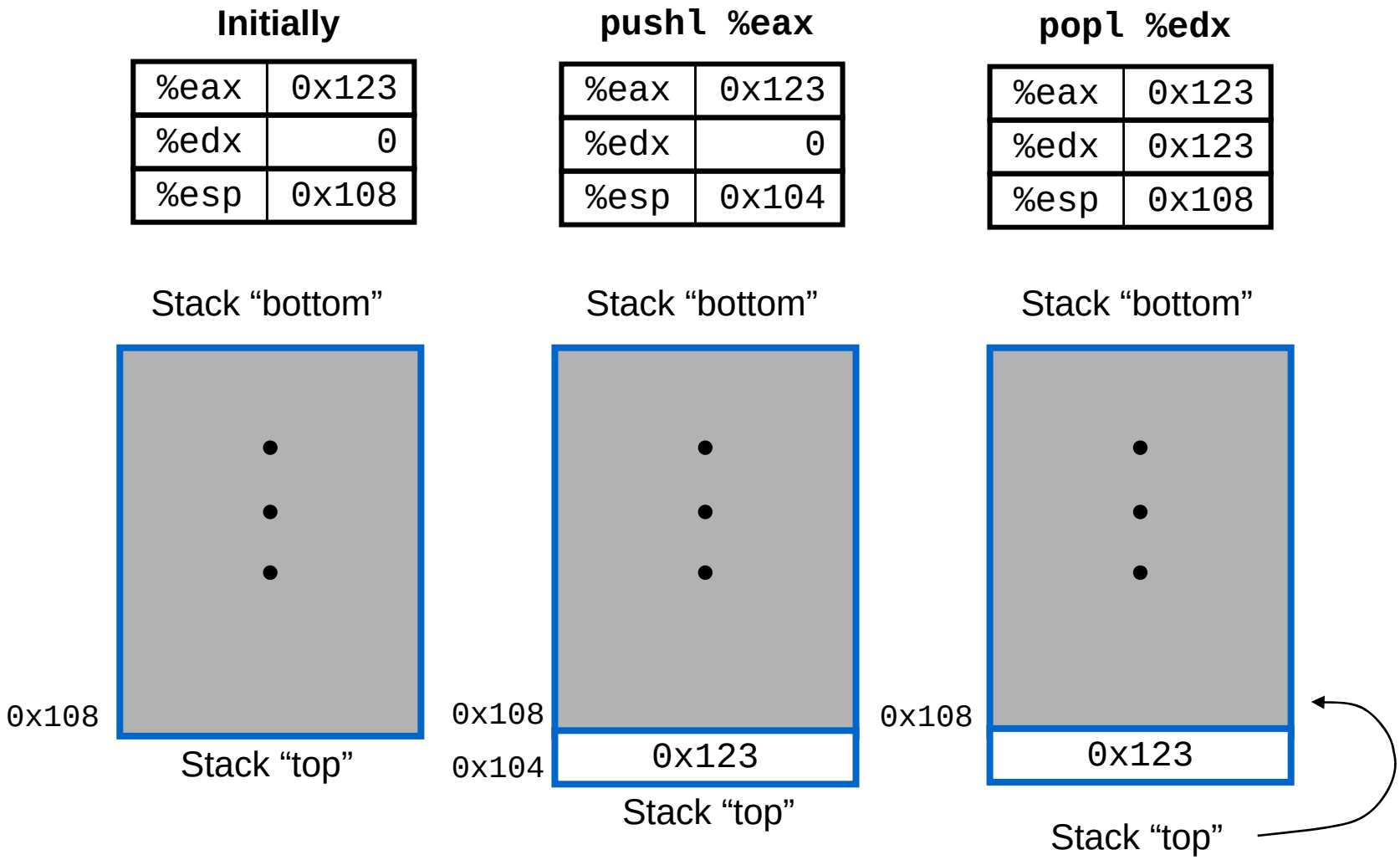
스택 연산 Push & Pop



스택 연산 Push & Pop



스택 연산 Push & Pop



단순 어드레싱 모드 용어의 이해

Normal (R) Mem[Reg[R]]

- **Register R 은 메모리 주소를 의미한다**

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

Displacement D(R) Mem[Reg[R]+D]

- **레지스터 R 메모리 블럭의 시작주소를 의미한다**
- **상수 변위 D 는 offset을 의미한다**

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

단순 어드레싱 모드의 이용

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

Q1. t0 = *xp ?
Q2. *xp = t1 ?
D3. 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

단순 어드레싱 모드의 이용

```
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
```

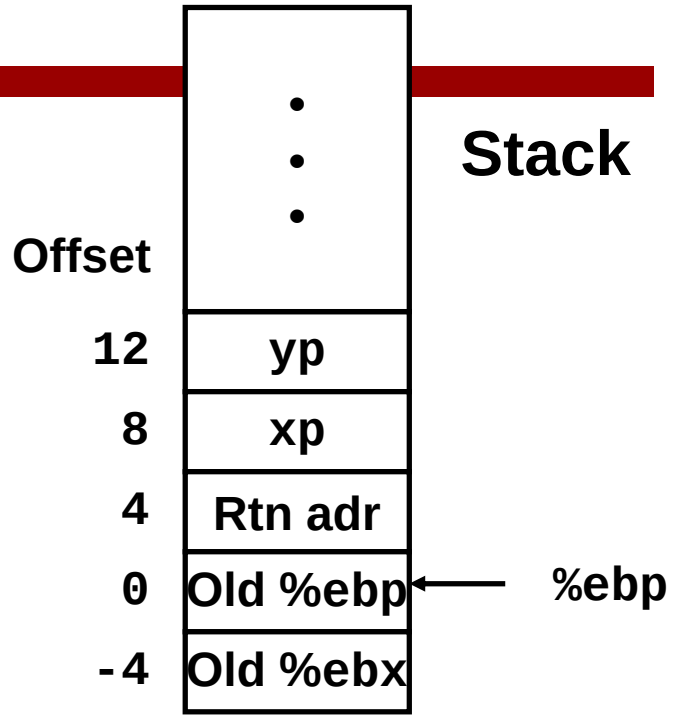
Set Up

Body

Finish

Swap 함수의 이해

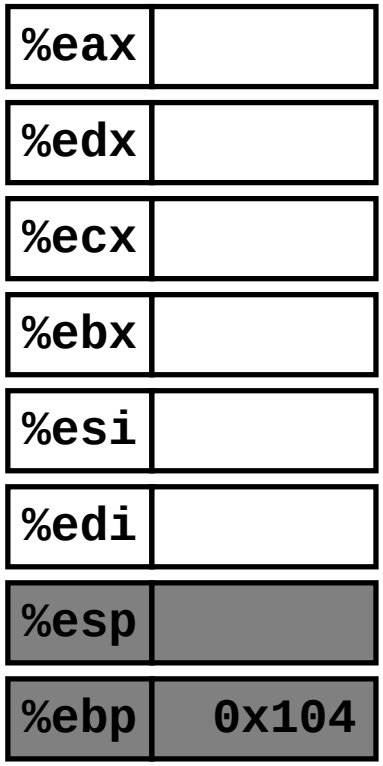
```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

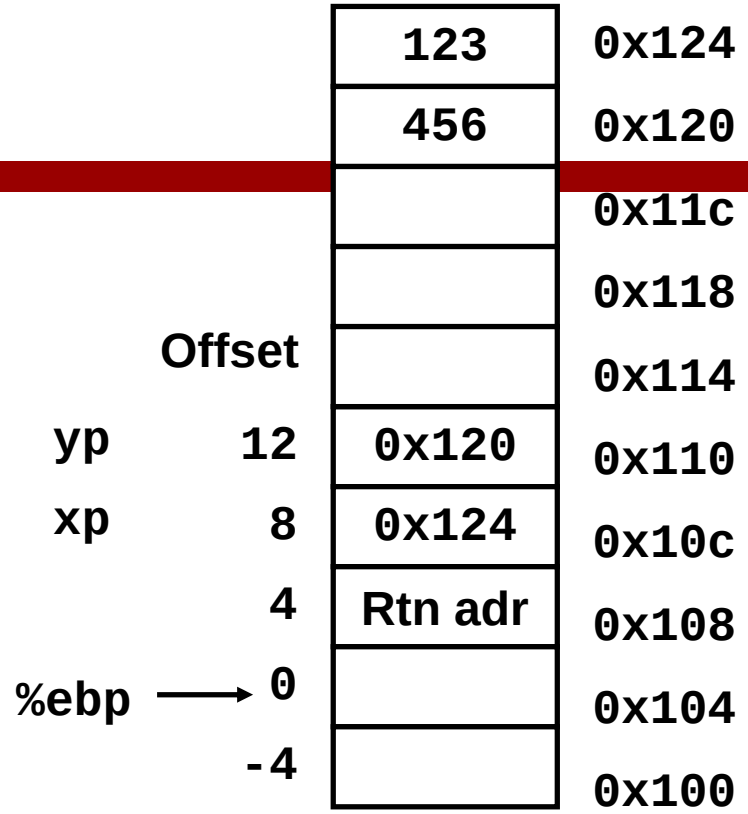


Register	Variable
%ecx	yp
%edx	xp
%eax	t1
%ebx	t0

```
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
```

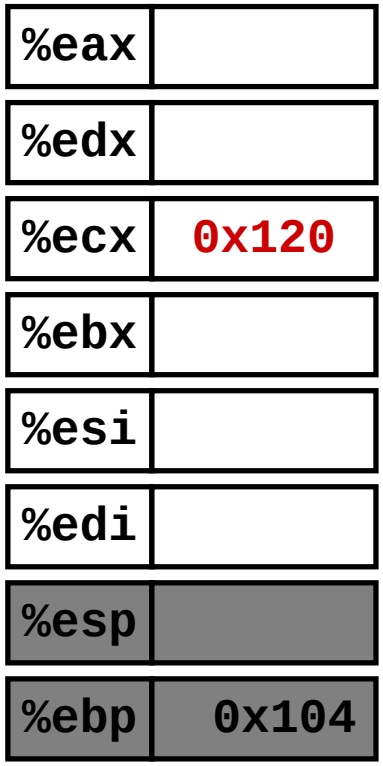
Swap 함수의 이해

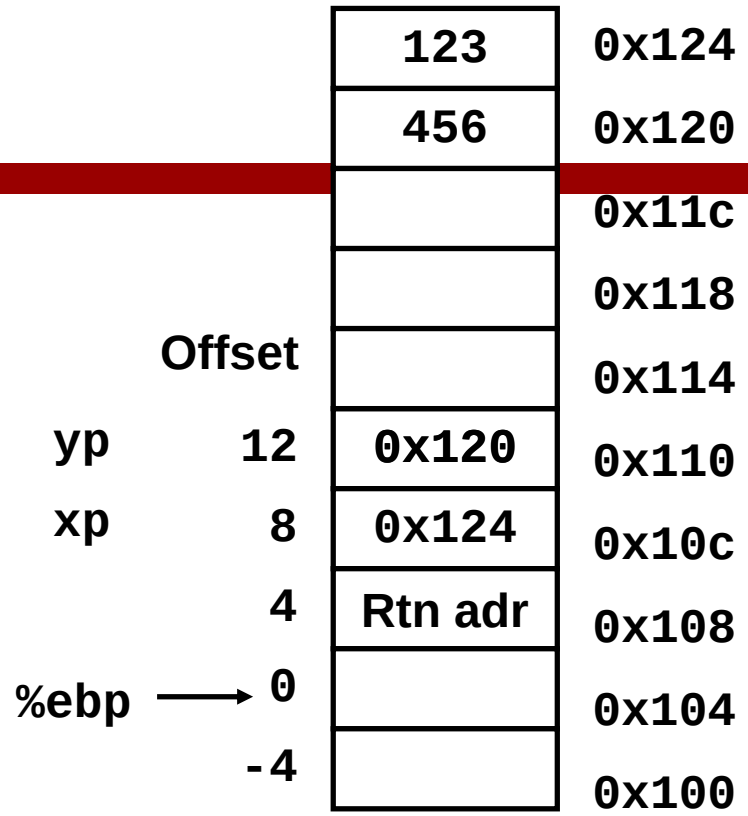





```
movl 12(%ebp),%ecx  # ecx = yp
movl 8(%ebp),%edx     # edx = xp
movl (%ecx),%eax      # eax = *yp (t1)
movl (%edx),%ebx      # ebx = *xp (t0)
movl %eax,(%edx)      # *xp = eax
movl %ebx,(%ecx)      # *yp = ebx
```

Swap 함수의 이해

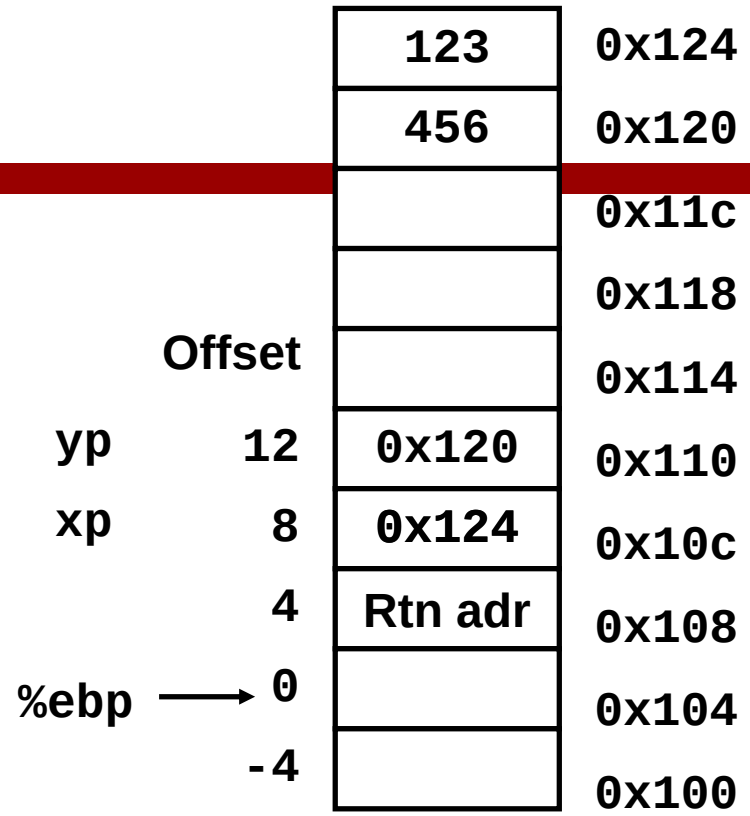





```
movl 12(%ebp),%ecx  # ecx = yp
movl 8(%ebp),%edx     # edx = xp
movl (%ecx),%eax      # eax = *yp (t1)
movl (%edx),%ebx      # ebx = *xp (t0)
movl %eax,(%edx)      # *xp = eax
movl %ebx,(%ecx)      # *yp = ebx
```

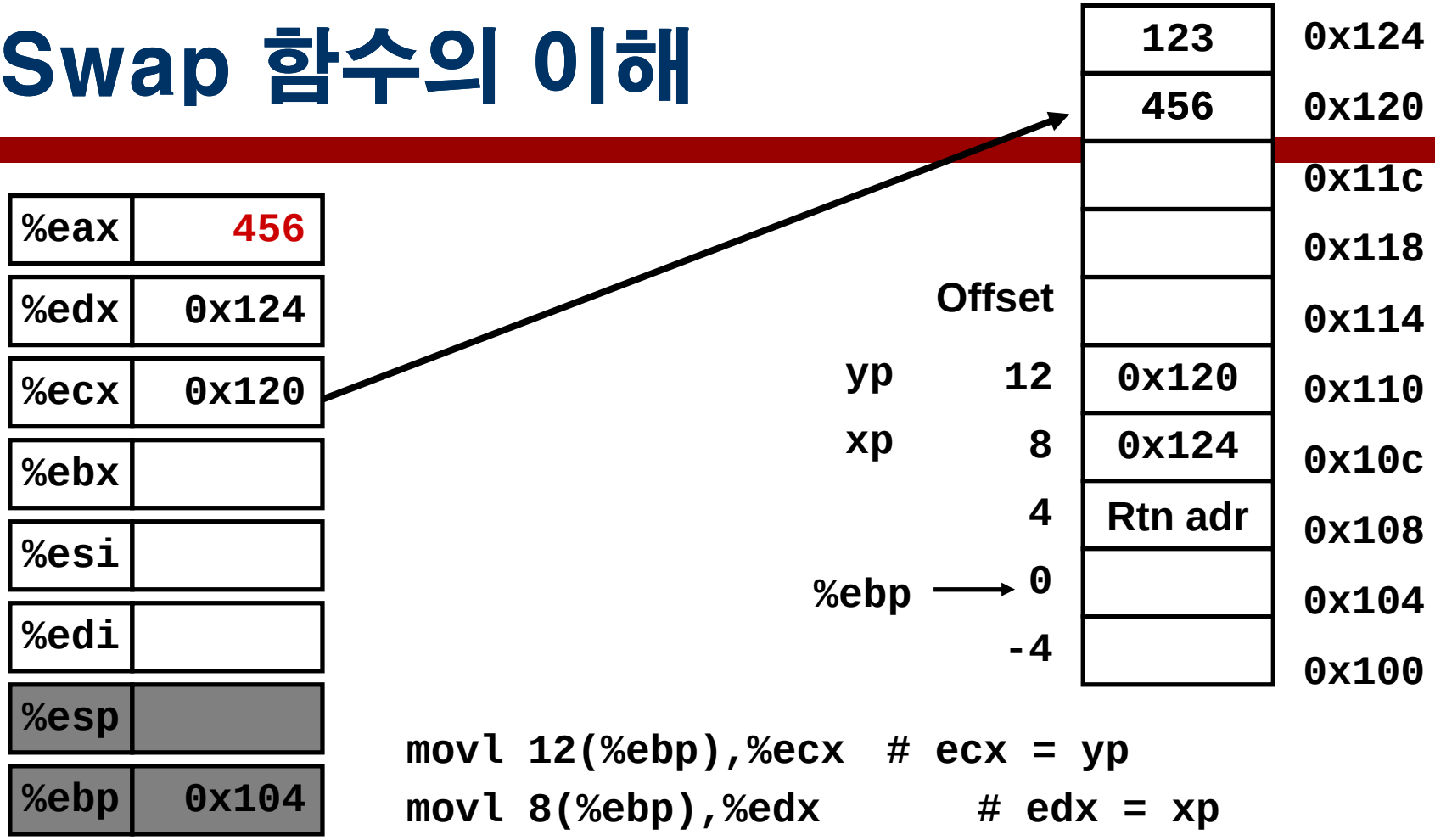
Swap 함수의 이해

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



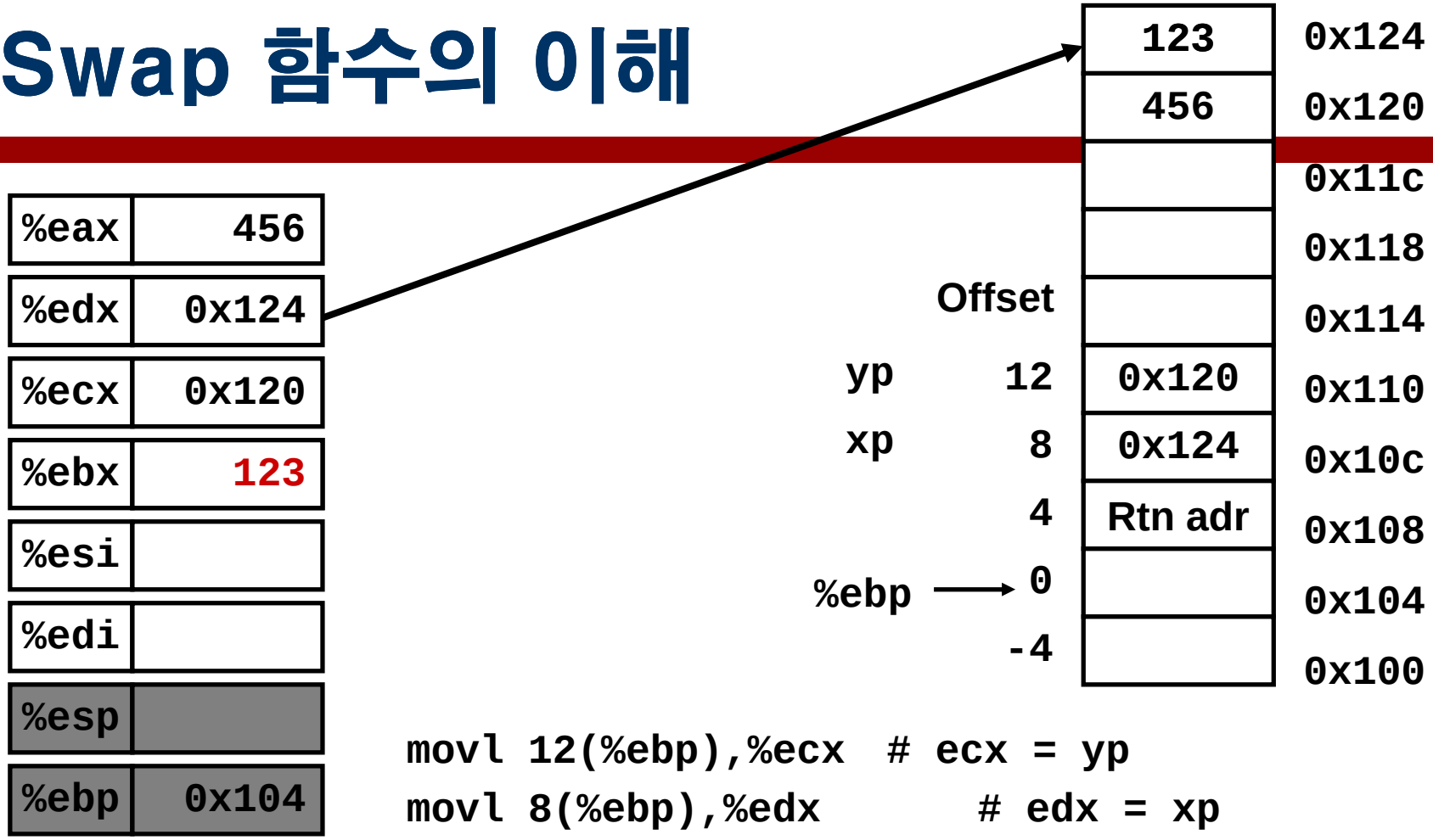

```
movl 12(%ebp),%ecx  # ecx = yp
movl 8(%ebp),%edx     # edx = xp
movl (%ecx),%eax      # eax = *yp (t1)
movl (%edx),%ebx      # ebx = *xp (t0)
movl %eax,(%edx)      # *xp = eax
movl %ebx,(%ecx)      # *yp = ebx
```

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
```

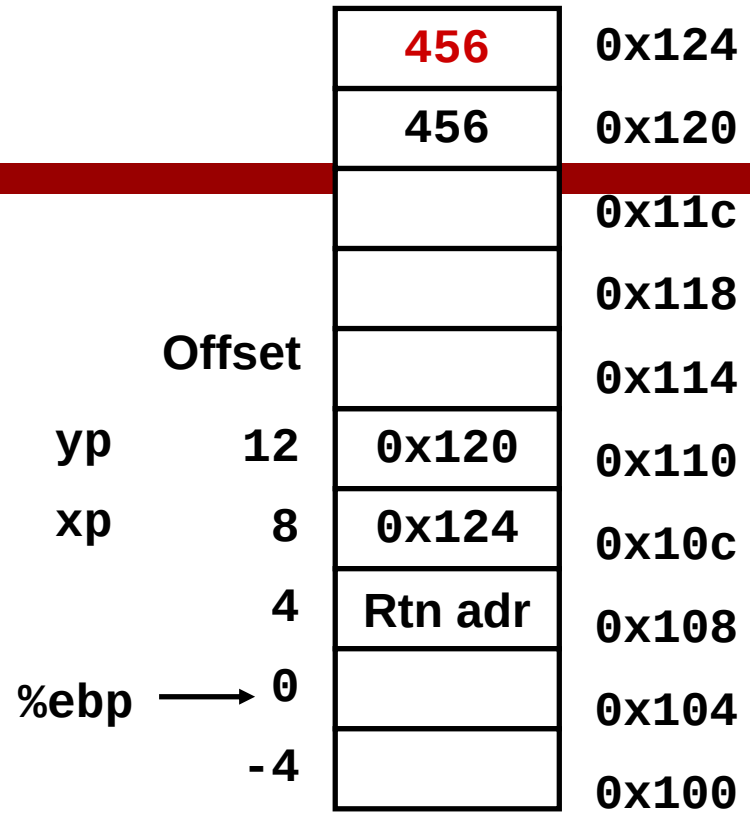
Swap 함수의 이해




```
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

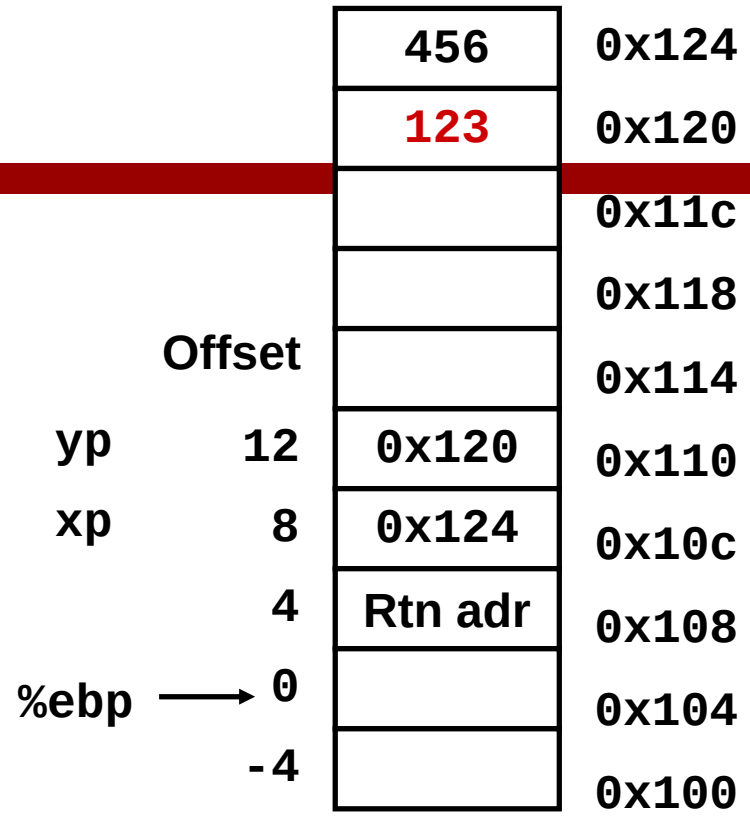
%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104




```
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

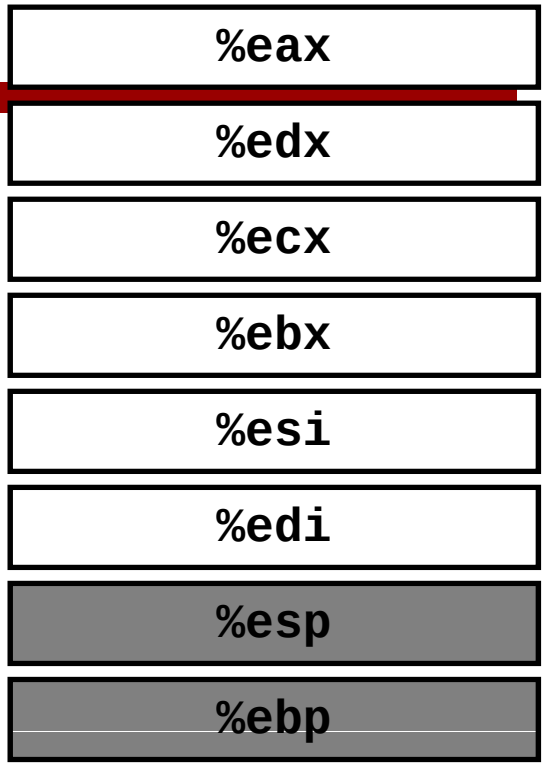



```
movl 12(%ebp),%ecx  # ecx = yp
movl 8(%ebp),%edx     # edx = xp
movl (%ecx),%eax      # eax = *yp (t1)
movl (%edx),%ebx      # ebx = *xp (t0)
movl %eax,(%edx)      # *xp = eax
movl %ebx,(%ecx)      # *yp = ebx
```

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

데이터 이동명령 MOV



데이터 이동하기

`movl` ***Source,Dest***:

- Move 4-byte ("long") word
- 자주 사용하는 명령어

오퍼랜드 형태 Operand Types

- Immediate: Constant integer data
  - '$' 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"

swap 함수에서

```
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

Indexed Addressing Modes

가장 일반적인 형태

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

- D: 상수값 "변위"
- Rb: Base register: 8 개의 정수 레지스터중의 한 개
- Ri: Index register: %esp를 제외한 모든 레지스터
  - %ebp 도 거의 사용하지 않음
- S: Scale: 1, 2, 4, or 8

Special Cases

(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

주소 계산을 위한 명령어

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

- ***Src*** 어드레스 모드 수식임
- ***Dest*** 에 어드레스 모드에 의한 최종 주소값을 기록

용도

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

D4. eax = x, edx = y 일때, edx를 표시하시오

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

요약

- 우리는 IA32 프로세서의 어셈블리어를 배운다
- 디스어셈블러를 이용하면 실행파일로부터 어셈블리를 볼 수 있다
- IA32 mov 데이터 이동명령어를 사용할 수 있다