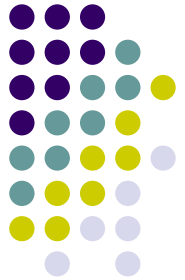
시스템 프로그래밍 (2016)

4번째 강의:
기계어 수준 프로그래밍 – 기초
(데이터 이동 명령)

강의 일정

주	날짜	강의실	날짜	실습실
1	9월 1일(목)	**소개 강의**	9월 6일(화)	**리눅스 개발환경 익히기 (VI, 쉘 기본명령어들)**
2	9월 8일(목)	**정수 표현 방법**	9월 13일(화)	GCC & Make, shell script
3	9월 15일(목)	**추석 휴강**	9월 20일(화)	**C 복습과 GDB 사용하기 1 (소스 수준 디버깅)**
4	9월 22일(목)	**실수 표현 방법**	9월 27일(화)	**Data lab (GDB활용)**
5	9월 29일(목)	**어셈1 - 데이터이동**	10월 4일(화)	**어셈1 – move(실습),**
6	10월 6일(목)	**어셈2 – 제어문**	10월 11일(화)	**어셈2- 제어문 (실습)**
7	10월 13일(목)	**어셈3 – 프로시저**	10월 18일(화)	**어셈3-프로시저(실습)**
8	10월 20일(목)	**어셈보충/중간시험**	10월 25일(화)	**GDB 사용하기2(어셈수준)**
9	10월 27일(목)	**보안**(buffer overflow)	11월 1일(화)	Binary bomb 1 (GDB활용)
10	11월 3일(목)	**프로세스 1**	11월 8일(화)	Binary bomb 2 (GDB활용)
11	11월 10일(목)	**프로세스 2**	11월 15일(화)	Tiny shell 1
12	11월 17일(목)	**시그널**	11월 22일(화)	Tiny shell 2
13	11월 24일(목)	**동적메모리 1**	11월 29일(화)	Malloc lab1
14	12월 1일(목)	**동적메모리 2**	12월 6일(화)	Malloc lab2
15	12월 8일(목)	**기말시험**	12월 13일(화)	Malloc lab3

목차(CONTENTS)



1. IA32→ X64 역사

Intel X86(80x86) processors

- PC 시장의 이제 유일한 프로세서, Intel PC

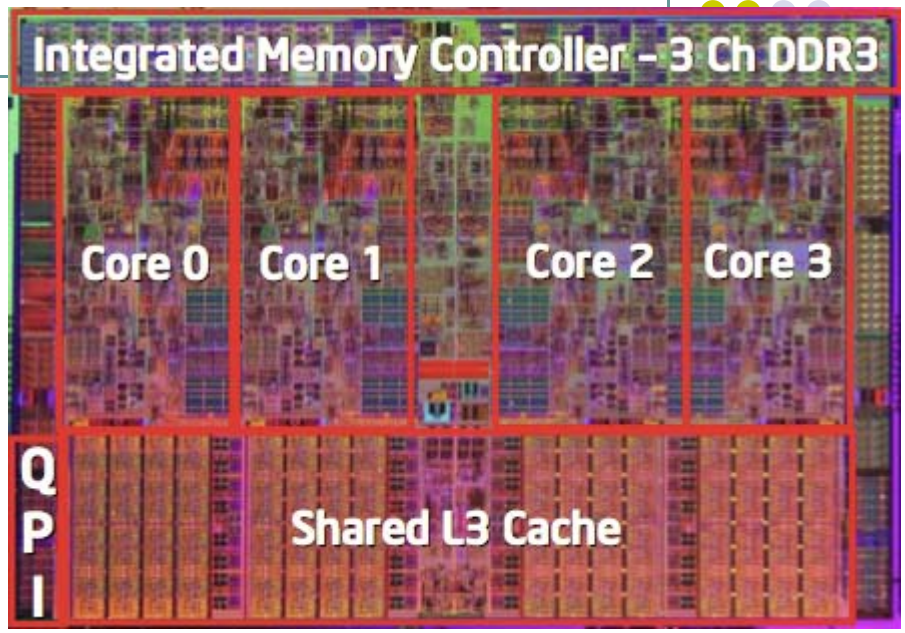
- 혁신적인 설계
  - 1978년에 처음 등장, 8086까지의 호환성(upward compatibility)
  - 시장 지배적 위치라서, 고급 기능들을 추가해가면서, 표준이 되다시피 함.
- Complex Instruction Set Computer (CISC)
  - 다양한 명령어 포맷 - 다양한 명령어
    - 리눅스에서도 small subset만 사용함
  - RISC와 비슷한 성능을 내기 어려움
  - 그러나, 성능은 따라 잡음(pipeline, superscalar 등 통해), 그러나 전력소모(저전력화)에는 별로 성공하지 못함.

Intel x86 Evolution: Milestones

Name	*Date*	*Transistors*	*MHz*
• 8086	1978	29K	5-10
◦ 최초 16비트 프로세서. IBM PC와 DOS의 기초 프로세서			
◦ 1MB 주소공간(20비트)			
• 386	1985	275K	16-33
◦ 최소의 32비트 프로세서, 나중에 IA32라고 …			
◦ 유닉스 수행을 위해 “flat addressing” 사용			
• Pentium 4E	2004	125M	2800-3800
◦ 최소의 64-비트 Intel x86 프로세서, x86-64라고…			
• Core 2	2006	291M	1060-3500
◦ 최소의 멀티코어 프로세서			
• Core i7	2008	731M	1700-3900
◦ 네개의 코어를 포함하는 프로세서			

Intel x86 Processors, cont.

- 진화과정
  - 386 1985 0.3M
  - Pentium 1993 3.1M
  - Pentium/MMX 1997 4.5M
  - PentiumPro 1995 6.5M
  - Pentium III 1999 8.2M
  - Pentium 4 2001 42M
  - Core 2 Duo 2006 291M
  - Core i7 2008 731M

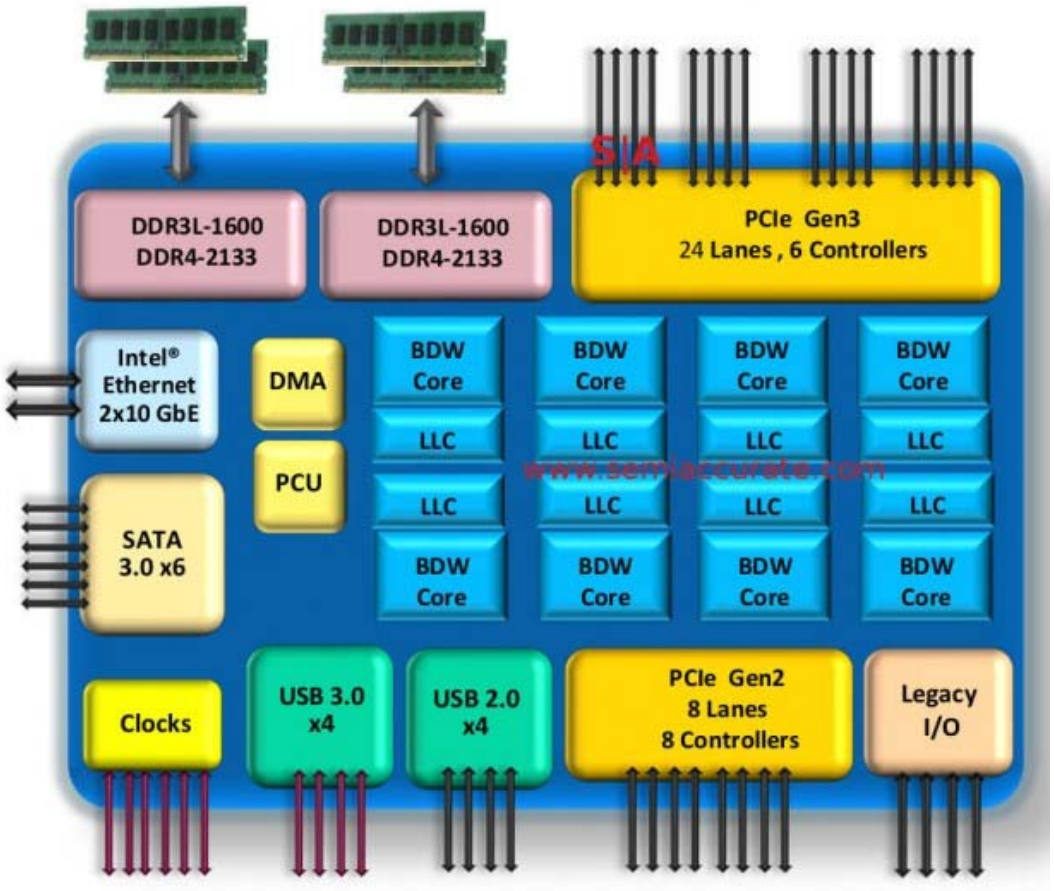


- 추가된 기능들
  - 멀티미디어 동작 지원 명령
  - 효율적인 조건부 동작(분기) 지원 명령
  - 32비트 ➔64비트로 전환, 다중 코어 지원

2015 현재

- Core i7 Broadwell 2015

- 데스크 탑 모델
  - 4 cores
  - Integrated graphics
  - 3.3-3.8 GHz
  - 65W
- 서버모델
  - 8 cores
  - Integrated I/O
  - 2-2.6 GHz
  - 45W



x86 Clones: Advanced Micro Devices (AMD)

- 역사적으로
  - AMD는 Intel 뒤를 쫓아감
  - 좀 느리지만, 좀 싼 …
- 그런데
  - 최고의 회로 설계자들 고용(DEC등 회사에서)
  - Opteron설계: Pentium 4와 경쟁
  - x86-64을 먼저 개발, 나름의 64비트 프로세서
- 최근에는
  - 인텔은 잘 나가는데, AMD는 좀 뒤처지고 있고…

Intel의 64-비트 역사

- 2001에 IA32에서 IA64로 급격한 전환시도
  - 문제는, 전혀다른 아키텍처(Itanium)
  - IA32명령은 legacy 코드를 실행해줌.
  - 성능이 실망스러움
- 2003: AMD는 혁신적인 발걸음 시도
  - x86-64 (요즘에는 “AMD64”라고 더 잘 알려짐)
- Intel은 IA64에 집중하고자 함.
  - 실수 인정 회피, AMD가 제대로 가는데…
- 2004: Intel도 IA32에 대한 EM64T 확장
  - 확장된 64-비트 메모리기술
  - x86-64와 거의 동일…
- 거의 모든 low-end x86프로세서는 x86-64 지원
  - 그러나, 여전히 많은 코드는 IA32 코드

2016년 과정에서는

- IA32는 더 다루지 않으나, 배경이 됨.
- x86-64
  - 표준적으로 다룸.
  - 실습기계는, GCC 명령는 기본적으로 –m64 옵션을 가정함.

2.어셈블리어와 프로그래머의 관점(X64)

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?

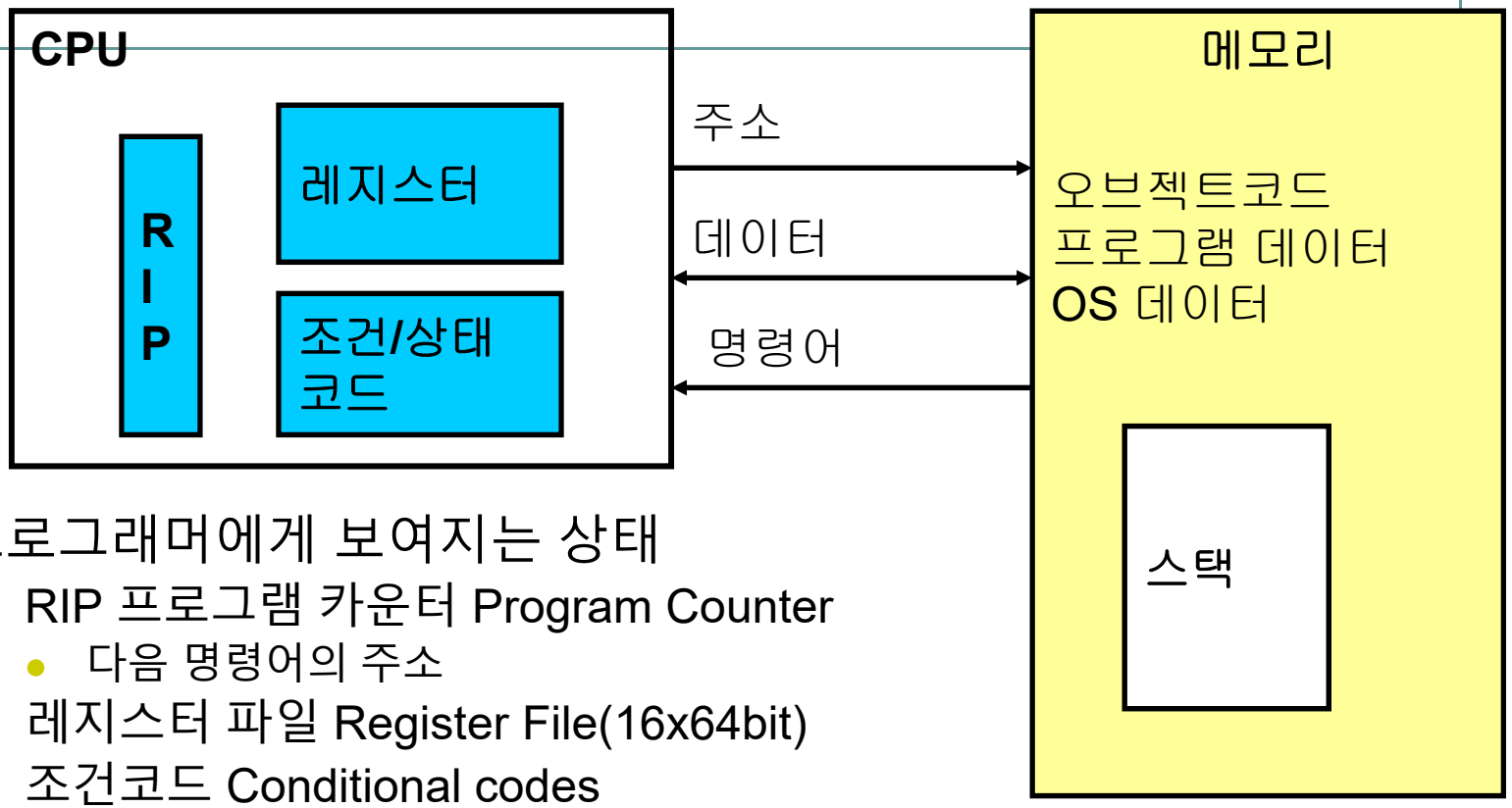
고급언어와 어셈블리어

- 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 (?)

용어들

- 아키텍처: (ISA-instruction set architecture, 명령어의 집합적 구조) 어셈블리나 머신 코드를 이해하는데 필요한 프로세서의 설계 내용.
  - 예: 명령어 집합 스펙, 레지스터 등
- 마이크로 아키텍처: 아키텍처에 대한 구현.
  - 예: 캐쉬 크기, 코어의 주파수 등.
- 코드 형태:
  - 머신 코드: 프로세서가 수행하는 바이트 수준 프로그램
  - 어셈블리 코드: 머신 코드에 대한 텍스트/심볼 표현
- ISA예:
  - Intel: x86, IA32, Itanium, x86-64
  - ARM: 거의 모든 휴대폰에 사용되는 프로세서

어셈블리 프로그래머의 관점(x64)

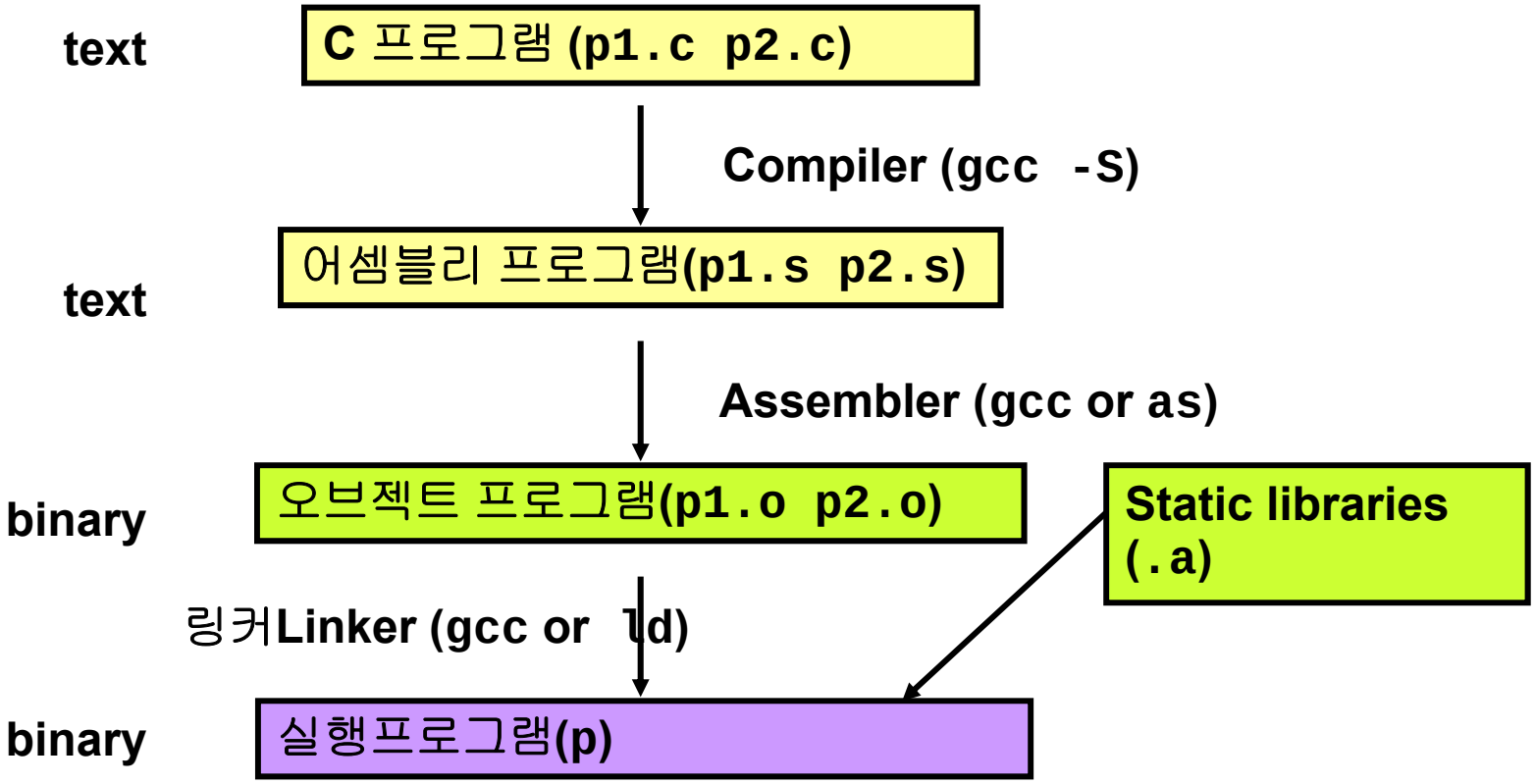


- 프로그래머에게 보여지는 상태
  - RIP 프로그램 카운터 Program Counter
    - 다음 명령어의 주소
  - 레지스터 파일 Register File(16x64bit)
  - 조건코드 Conditional codes
    - 가장 최근의 연산의 결과로 인한 상태정보를 저장
    - 조건형 분기명령에서 이용됨

- **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 데이터(머신코드)를 `p` 에 저장



C 프로그램을 어셈블리어로 컴파일하기

C Code

```
long mult2(long, long);
void multstore(long x, long
y, long *dest) {
    long t = mult2(x,y);
    *dest = t;
}
```

생성된 어셈블리 프로그램

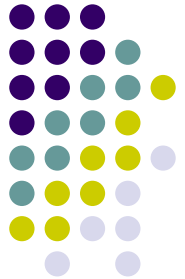
```
multstore:
    pushq %rbx
    movq    %rdx, %rbx
    call    mult2
    movq    %rax, (%rbx)
    popq    %rbx
    ret
```

다음의 명령으로 생성

```
gcc -O -S mult2.c
```

code.s 파일이 만들어짐

기계나, **linux**버전, **OS**, 설정에 따라
다른 코드 생성됨.



3. 어셈블리어의 특성, 목적코드, 역어셈블

어셈블리어의 특성1: 데이터유형

- 정수데이터: 1, 2, 4, 8 바이트
  - 데이터 값
  - 주소 (유형/타입이 없는 포인터)
- 실수형 데이터 : 4, 8, or 10 bytes
- 코드: 바이트들의 나열 (연속된 명령어들이 인코딩된것)
- 배열이나 구조체 유형은 …
  - 메모리에서의 연속적인 바이트들로 표시

어셈블리어의 특성2: 연산

- 산술 연산은 레지스터나 메모리에 대한 연산
- 메모리와 레지스터 간 이동
  - 메모리로부터 레지스터로 데이터를 이동
  - 레지스터의 데이터를 메모리에 저장
- 제어의 이동
  - 무조건형 점프 Unconditional jumps to/from procedures
  - 조건형 분기 Conditional branches

목적코드 Object code

Code for `sum`

Disassembly of section .text:

```
0000000000000000 <multstore>:
   0:    53                   push   %rbx
   1:    48 89 d3             mov    %rdx,%rbx
   4:    e8 00 00 00 00       callq  9
<multstore+0x9>
   9:    48 89 03             mov    %rax,(%rbx)
   c:    5b                   pop    %rbx
   d:    c3                   retq
```

- **Total of 13 bytes**
- **Each instruction 1, 2, 3, 5 bytes**
- **Starts at address `0x0000000`**

- 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

머신 명령어 예

```
*dest = t;
```

- C 코드
  - **dest 주소에 값 t 저장**

```
movq %rax, (%rbx)
```

- 어셈블리
  - 8바이트값을 메모리로 이동
  - 피연산자:
    - **t:** **레지스터 %rax**
    - **dest:** 레지스터 **%rbx**
    - ***dest:** 메모리 **M[%rbx]**

```
0x40059e:  48 89 03
```

- 오브젝트 코드
  - 3-byte 명령
  - **0x40059e 번지에 저장**

목적코드의 역어셈블(disassembling)

Disassembled

```
0000000000000000 <multstore>:
   0:    53                    push   %rbx
   1:    48 89 d3              mov    %rdx,%rbx
   4:    e8 00 00 00 00              callq  9 <multstore+0x9>
   9:    48 89 03              mov    %rax,(%rbx)
   c:    5b                    pop    %rbx
   d:    c3                    retq
```

- Disassembler(역어셈블러) objdump

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

또 다른 역어셈블

Object

```
0x00000:
    0x53
    0x48
    0x89
    0xd3
    0xe8
    0x00
    0x00
    0x00
    0x00
    0x48
    0x89
    0x03
```

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

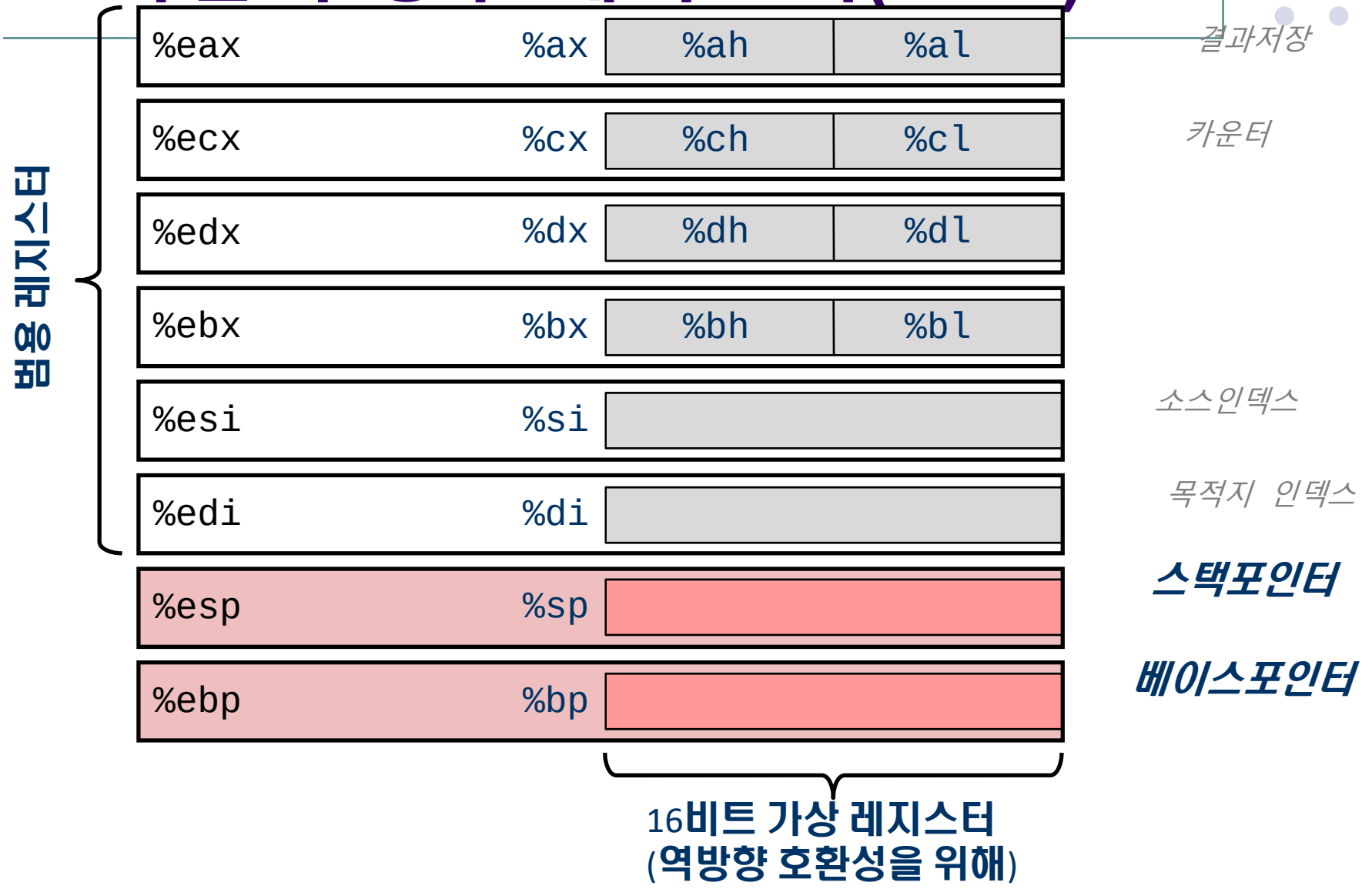
4.레지스터(X64중심)와 MOV명령

x64 registers

64-bit register	Lower 32 bits	Lower 16 bits	Lower 8 bits	설명(나중에 …)
rax	eax	ax	al	Return value
rbx	ebx	bx	bl	Callee saved
rcx	ecx	cx	cl	4th argument
rdx	edx	dx	dl	3rd argument
rsi	esi	si	sil	2nd argument
rdi	edi	di	dil	1st argument
rbp	ebp	bp	bpl	Callee saved
rsp	esp	sp	spl	Stack pointer
r8	r8d	r8w	r8b	5th arugment
r9	r9d	r9w	r9b	6th argument
r10	r10d	r10w	r10b	Caller saved
r11	r11d	r11w	r11b	Caller saved
r12	r12d	r12w	r12b	Callee saved
r13	r13d	r13w	r13b	Callee saved
r14	r14d	r14w	r14b	Callee saved
r15	r15d	r15w	r15b	Callee saved

주소 지정에는 64비트 레지스터만 사용됨!!.

펜티엄의 정수 레지스터(IA32)

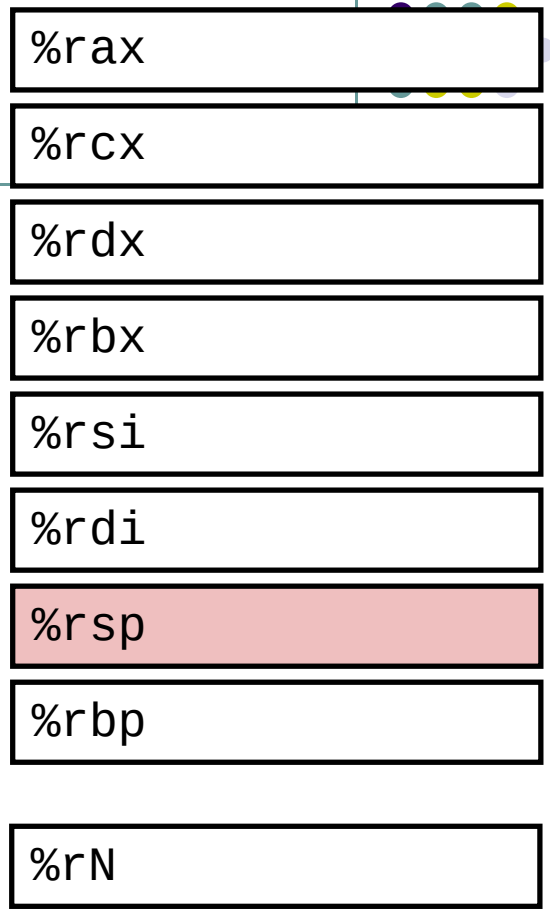


데이터 이동명령 MOV

- 데이터 이동하기

  `movq` *Source*,*Dest*:

- 피연산자(operand) 유형
  - Immediate: 상수 정수 데이터
    - '$' 로 시작함, 예) `$0x400`, `$-533`
    - 1, 2, or 4 바이트 가능
  - Register: 우측의 16개의 레지스터를 이용
    - 예: `%rax`, `%r13`
    - `%rsp` 특정 목적으로만 사용
    - 몇몇은 특별 목적이 있음
  - Memory: 레지스터에 저장된 주소에 있는 연속된 8 바이트
    - 어드레스 모드에 따라 다름 "주소 모드(address modes)"
    - 가장 단순한 형태: (%rax)



movq 피연산자 조합

	Source	Dest	Src,Dest	C Analog
movq	*Imm*	*Reg*	`movq $0x4,%rax`	`temp = 0x4;`
		Mem	`movq $-147,(%rax)`	`*p = -147;`
	Reg	*Reg*	`movq %rax,%rdx`	`temp2 = temp1;`
		Mem	`movq %rax,(%rdx)`	`*p = temp;`
	Mem	*Reg*	`movq (%rax),%rdx`	`temp = *p;`

메모리-메모리 전송은 한 명령어으로는 불가

5. 단순 주소 지정모드 – 이동 명령

단순 메모리 주소지정 모드

- 일반(Normal) (R) Mem[Reg[R]]
  - 레지스터 R 은 메모리 주소 지정
  - 바로! C에서의 포인터로 이동하기 (int *p; int c; c = *p;)
    `movq (%rcx),%rax`
- 변위Displacement D(R) Mem[Reg[R]+D]
  - 레지스터 R은 메모리 영역의 시작 주소 지정
  - 상수 변위 D는 오프셋(시작주소에서 특정위치 까지 차이)

```
movq 8(%rbp),%rdx
```

단순 메모리 주소지정 모드 예

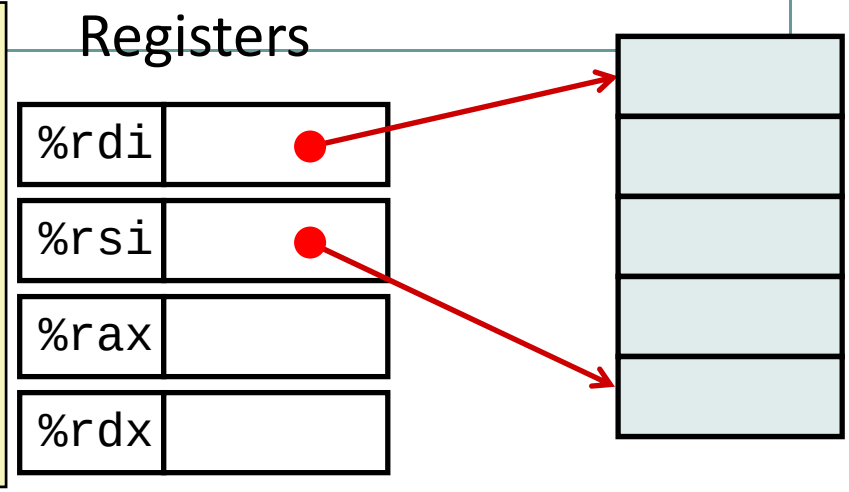
```
void swap
   (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
xp in rdi, yp in rsi

   movq    (%rdi), %rax
   movq    (%rsi), %rdx
   movq    %rdx, (%rdi)
   movq    %rax, (%rsi)
   ret
```

Swap() 함수 이해하기

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



Register	Value
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Swap() 함수 이해하기

Registers

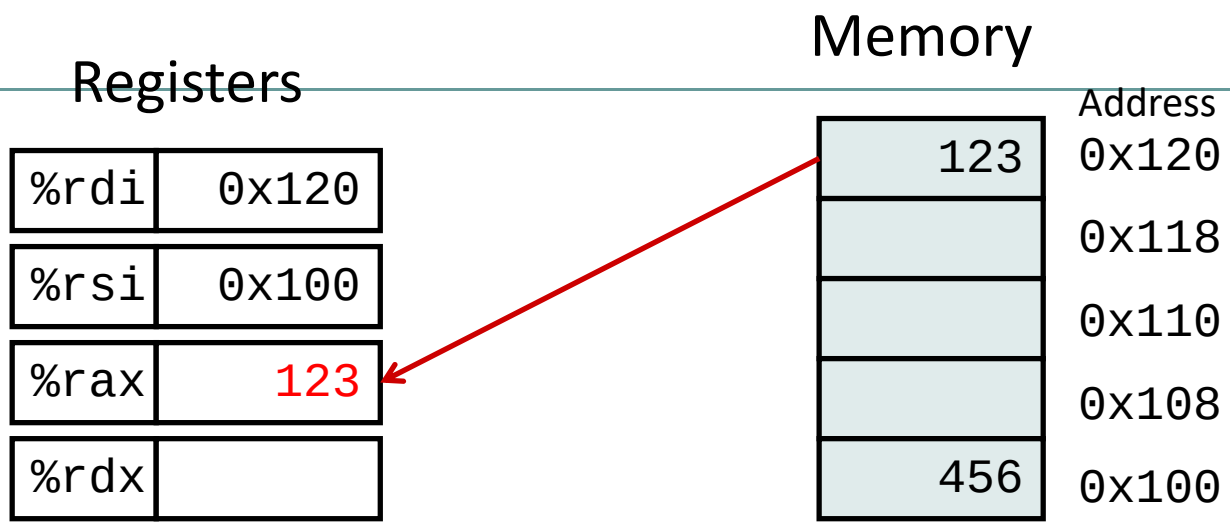
%rdi	0x120
%rsi	0x100
%rax	
%rdx	

Memory

	Address
123	0x120
	0x118
	0x110
	0x108
456	0x100

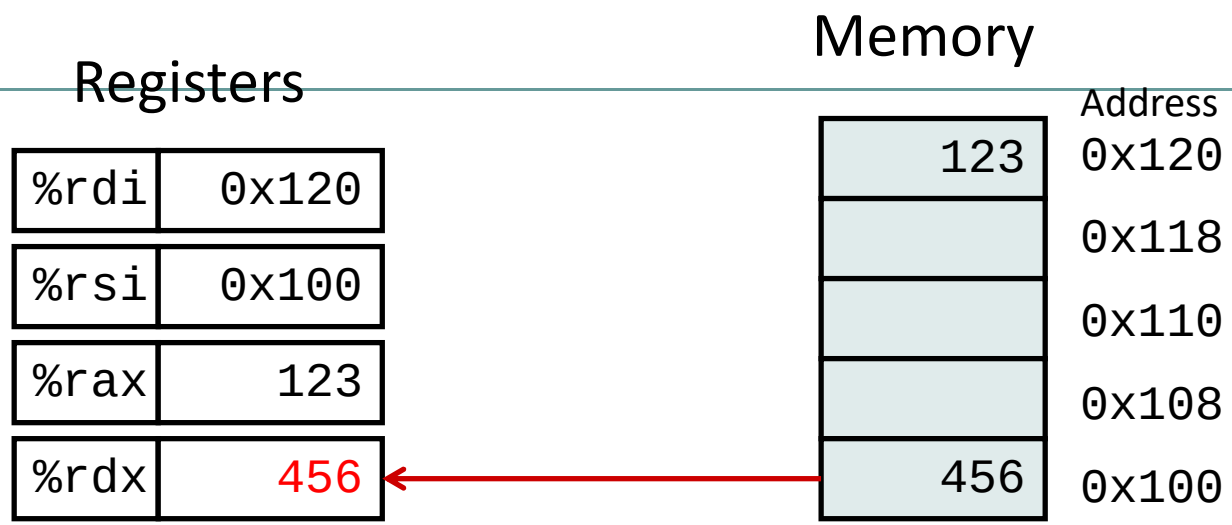
```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Swap() 함수 이해하기



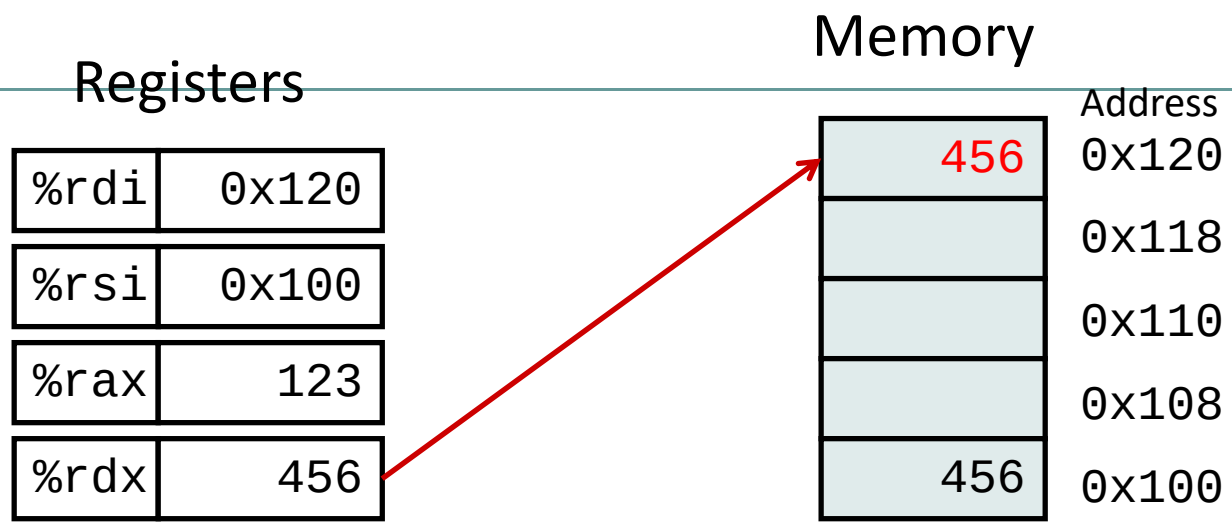

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Swap() 함수 이해하기



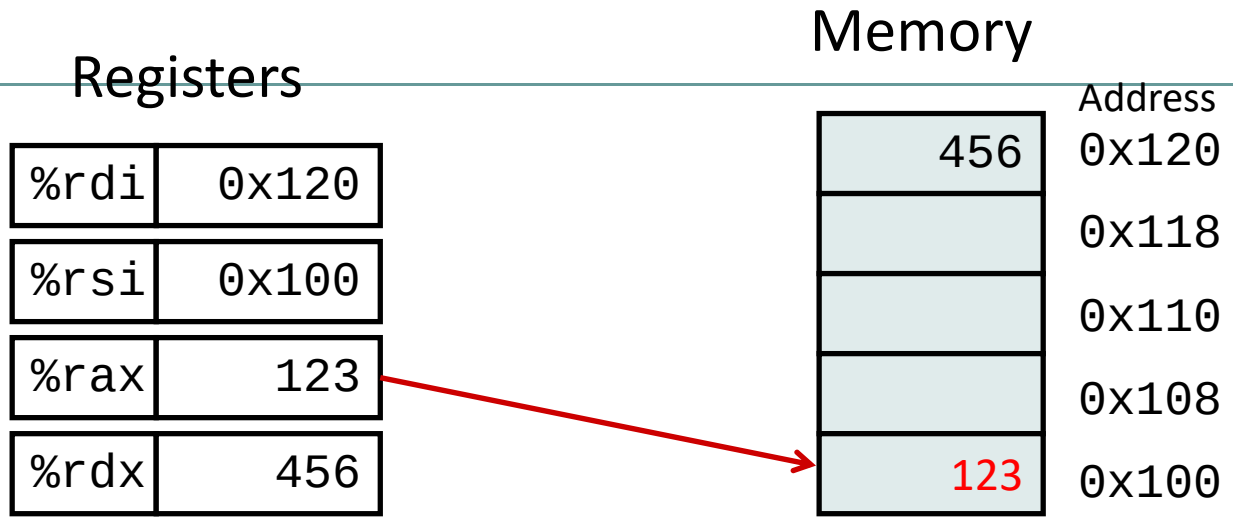

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Swap() 함수 이해하기




```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

Swap() 함수 이해하기




```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

6. 메모리 주소 지정 모드 (일반형)

단순 메모리 주소지정 모드

- 일반(Normal) (R) Mem[Reg[R]]
  - 레지스터 R 은 메모리 주소 지정
  - 바로! C에서의 포인터로 이동하기 (int *p; int c; c = *p;)

```
movq (%rcx),%rax
```

- 변위Displacement D(R) Mem[Reg[R]+D]
  - 레지스터 R은 메모리 영역의 시작 주소 지정
  - 상수 변위 D는 오프셋(시작주소에서 특정위치 까지 차이)

```
movq 8(%rbp),%rdx
```

완전 메모리 주소지정 모드

- 가장 일반적인 형태

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

  - D: 상수 “변위 " 1, 2, or 4 bytes
  - Rb: 베이스 레지스터: 16 정수 레지스터
  - Ri: 인덱스 레지스터: **`%rsp`를 제외하고 모두**
  - S: 배수: 1, 2, 4, or 8

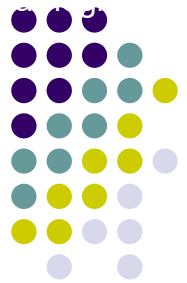
- 일반적인 형태의 특정 사례

  (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]]

주소 계산 예제



%rdx	0xf000
%rcx	0x0100

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

x86-64의 특징 포함. 제로/부호 확장 이동 명령,

7. 데이터 이동 명령2

데이터 이동 명령어1(X86-64특징)

명령어	동작	설명
mov S, D	D ← S	Move
movb		Move byte
movw		Move word
movl		Move double word
movq		Move quad word
movabsq I, R	R ← I	Move absolute quad word

1. movl 명령의 D가 registe인 경우, 예를 들어 eax인 경우, rax의 하위4바이트인 movl $0x12345678, %eax ➔ %eax는 0x12345678로 채워지고, %rax의 상위 4바이트는 0x0로 채워짐. ➔ x86-64의 특징임.
즉, 32 register에 32비트값을 채우는 명령은 하위 바이트 외에
상위 4바이트는 0으로 설정
2. movabsq I, R은 64-bit immediate(상수)값을 64bit register에 저장하는 명령임.
3. movq I, R 에서 I는 32-bit이고, 그 결과는 R(64bit)에 부호확장으로 저장됨.
Movq $0xff345678, %rax ➔ %rax에는 0xffffffffff345678이 저장됨.

이동명령어 사례

명령어들	설명
movl $0x4050, %eax	Immediate → register, 4 byte, x64특징반영!!
movw %bp, %sp	Register→register , 2 byte
movb (%rdi,%rcx), %al	Memory → register, 1 byte
movb $-17, (%esp)	Immediate → memory, 1 byte
movq %rax, -12(%rbp)	Register → memory, 8 byte
movq $0xff345678, %rax	Immediate → register, 8 byte(부호확장) Immediate는 32bit 임.

점검문제1: 일련의 이동 명령어 수행 결과?

순서대로 수행됨.	%rax에 저장된 값의 변화
movabsq $0x0011223344556677,%rax	
movb $-1,%al	
movw $-1,%ax	
movl $-1,%eax	
movq $-1,%rax	

제로/부호 확장 이동 명령어

명령어	효과	설명
movz/s S, R	R ← zero/signExtend(S)	제로/부호확장이동
movz/sbw		바이트→ 워드
movz/sbl		바이트→더블워드
movz/swl		워드→더블워드
movz/sbq		바이트→쿼드워드
movz/swq		워드→ 쿼드워드

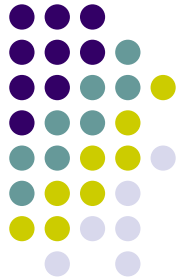
더블워드→ 쿼드워드??

일련의 movs/z 이동 명령어 수행 결과?

순서대로 수행됨.	%rax에 저장된 값의 변화?
movabsq $0x0011223344556677,%rax	
movb $0xAA,%dl	
movb %dl,%al	
movsbq %dl,%rax	
movzbq %dl,%rax	

점검 문제2: 어떤 오류가 있나 ?

명령어들	어셈블러 오류 내용은?
Movb $0xF, (%ebx)	
Movl %rax, (%rsp)	
Movw (%rax), 4(%rsp)	
Movb %al, %sl	
Movq %rax, $0x1234	
Movl %eax, %rsi	
Movb %si, %di	



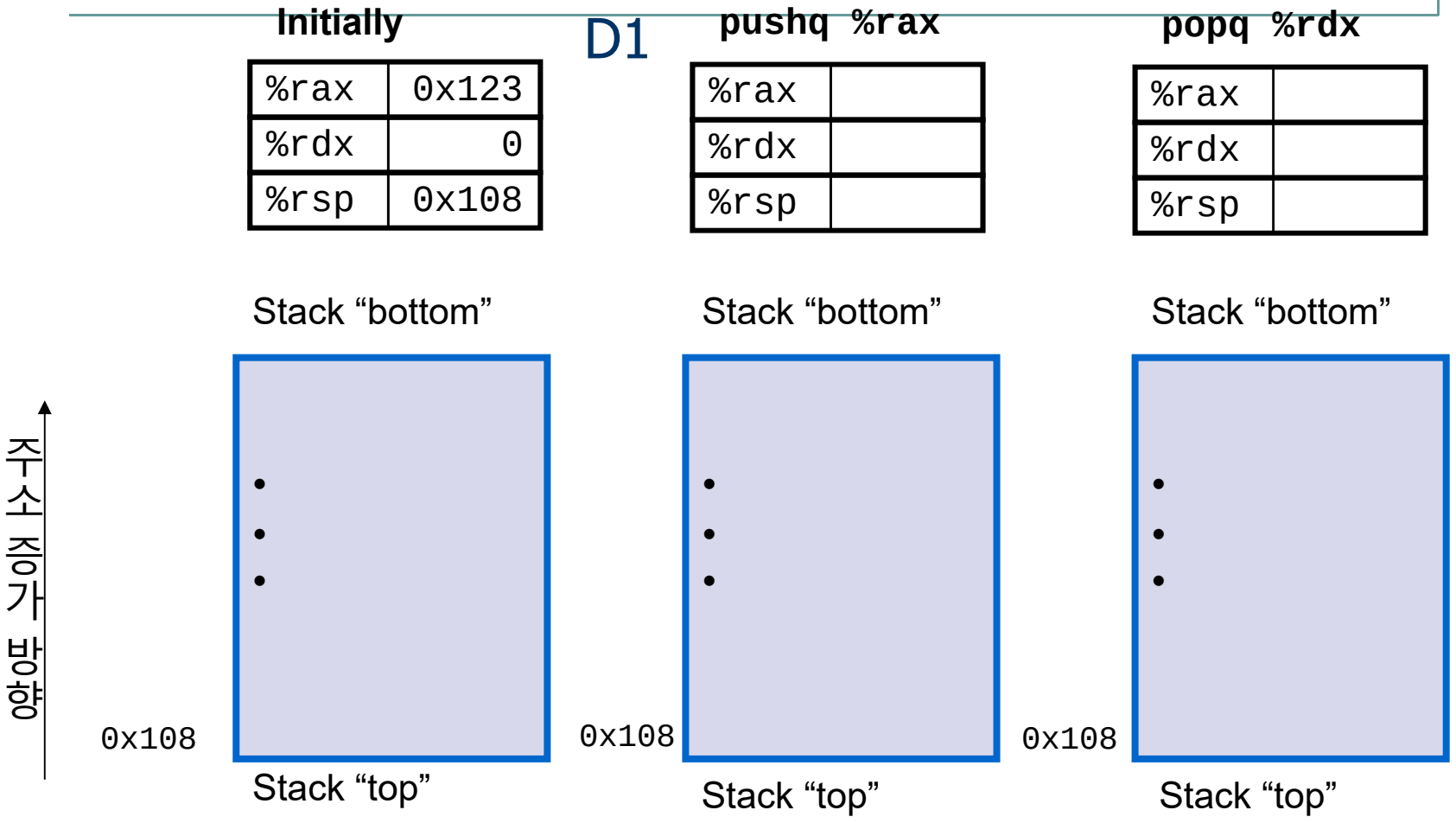
8. 스택 데이터의 이동(PUSHQ, POPQ와 레지스터 RSP)

PUSHQ, POPQ 명령

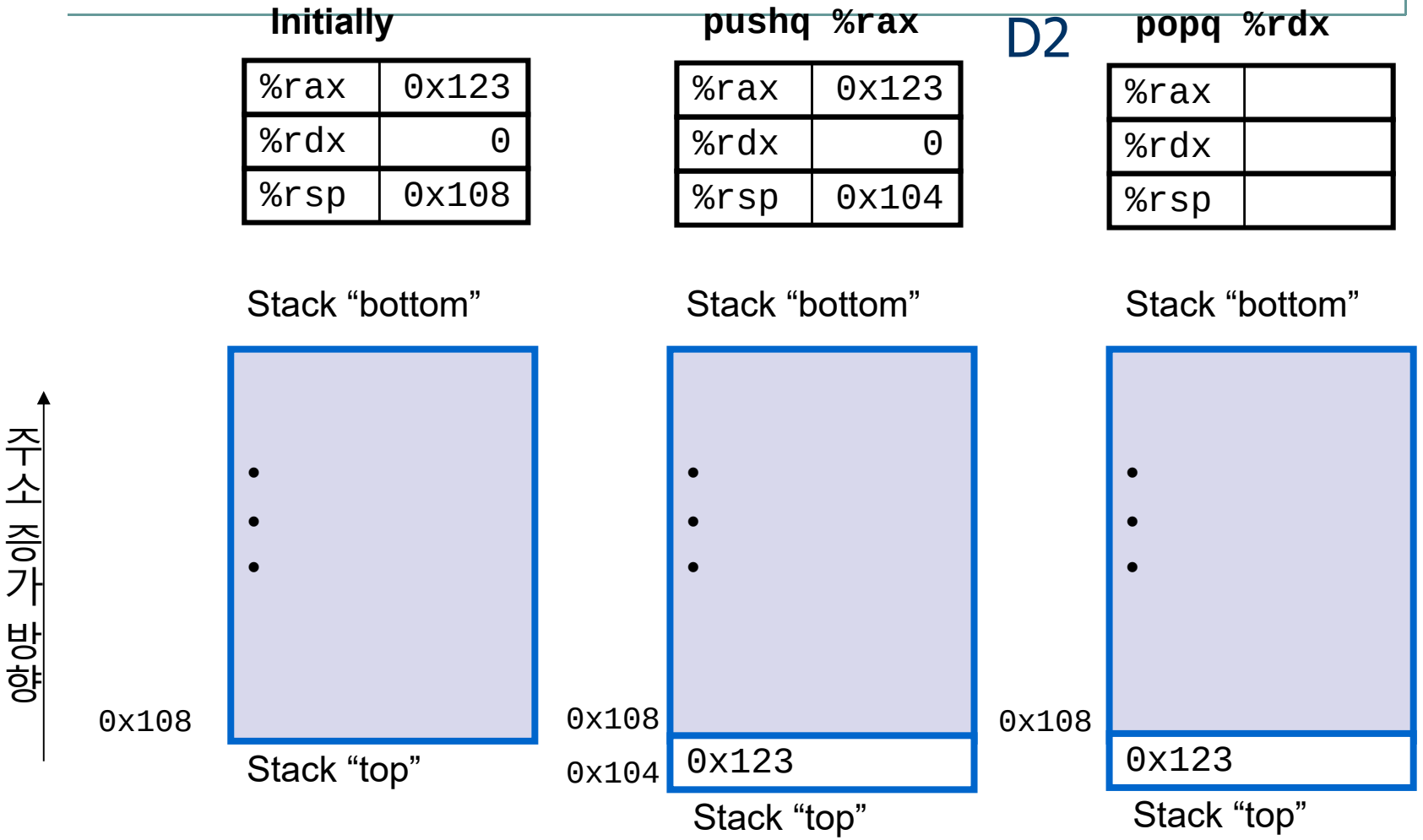
명령어	효과	설명
pushq S	R[%rsp] ←R[%rsp]-8 M[R[%rsp]]← S	Push quad word
popq D	D ← M[R[%rsp]] R[%rsp] ←R[%rsp]+8	Pop quad word

스택에서 push하면, 주소가 감소하는 방향으로 커짐. Pop는 반대로…
%rsp는 stack top에 저장되어 있는 값(8byte)의 주소를 가짐.
Stack의 top은 유효한 8바이트 저장소를 가리킴: stack top과 %rsp는 동일함.

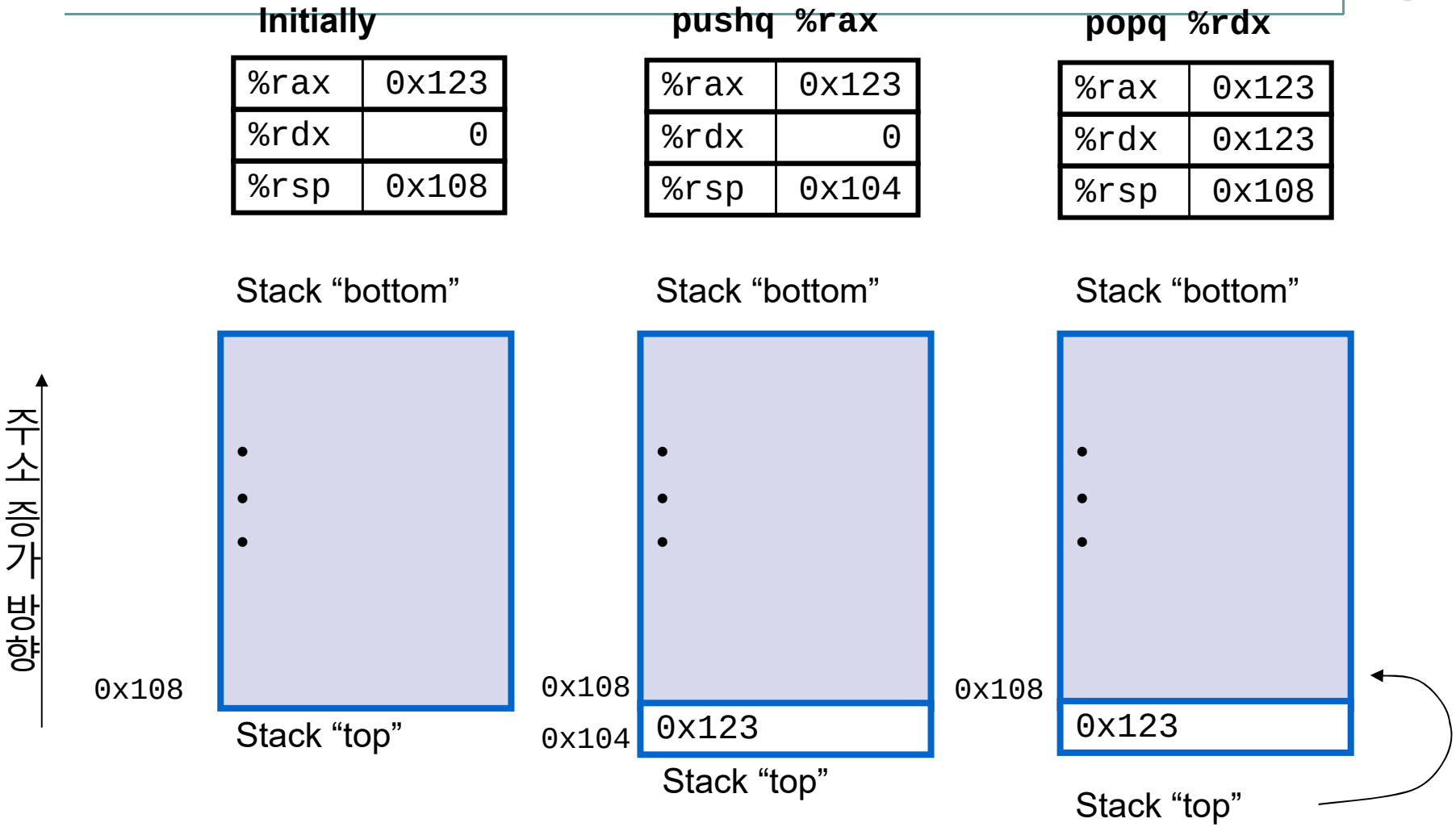
스택 연산 Push & Pop



스택 연산 Push & Pop



스택 연산 Push & Pop



요약

- 이동 명령
- 산술 논리 연산 명령

실습과 다음 주 준비

- 이어지는 실습: 어셈1 – move(실습)

- 다음주 강의 동영상: 어셈블리어 프로그래밍2 – 제어문
- 예습 질문은 개인과제로 올림.