CHUNGNAM NATIONAL UNIVERSITY

시스템 프로그래밍

강의 7 : 3장. 어셈블리어 I
3.1 프로세서의 역사
3.2 프로그램의 코딩
3.3 데이터 이동 명령어
3.4 정보접근하기

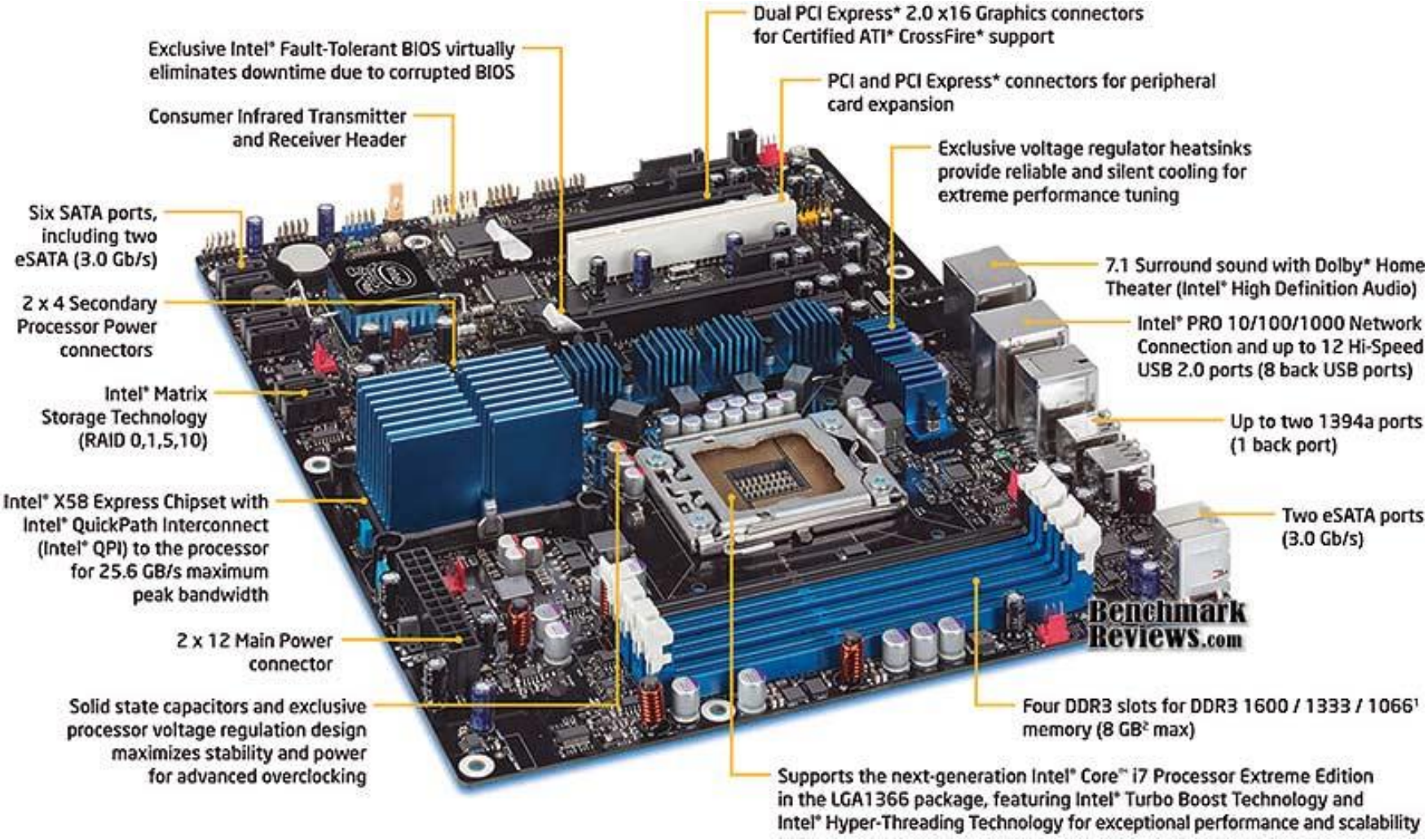
http://eslab.cnu.ac.kr

* 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)

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 – x86-64

C 언어가 어떻게 기계어로 번역되는지 배운다

어셈블리어 프로그래밍 기술과 어셈블리어를 이해하는 방법을 배운다

x86 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

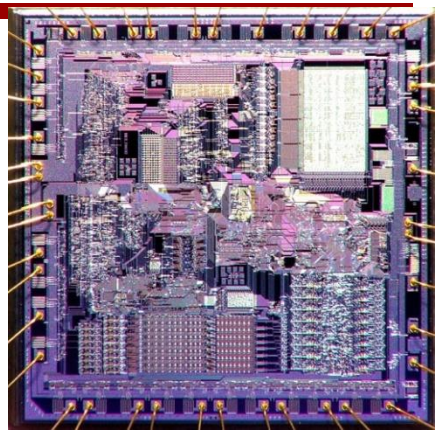
- 16-bit processor. IBM PC & DOS 사용
- 1MB 주소공간 address space. DOS 는 640K만을 허용 (기억하나?)

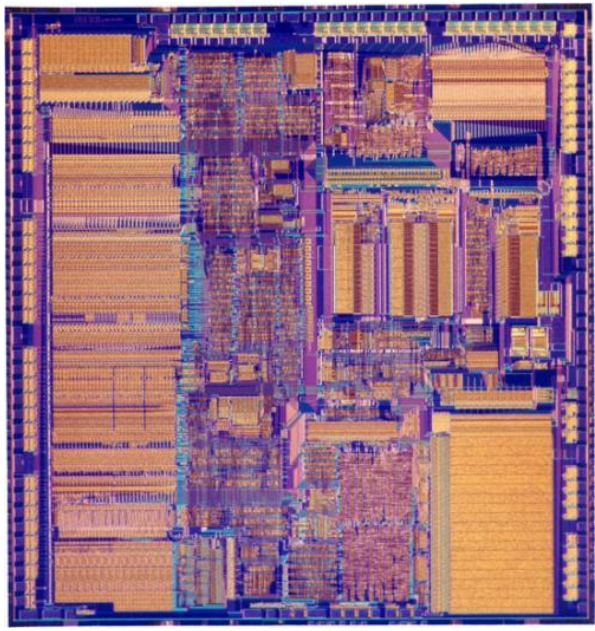
80286	1982	134K

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

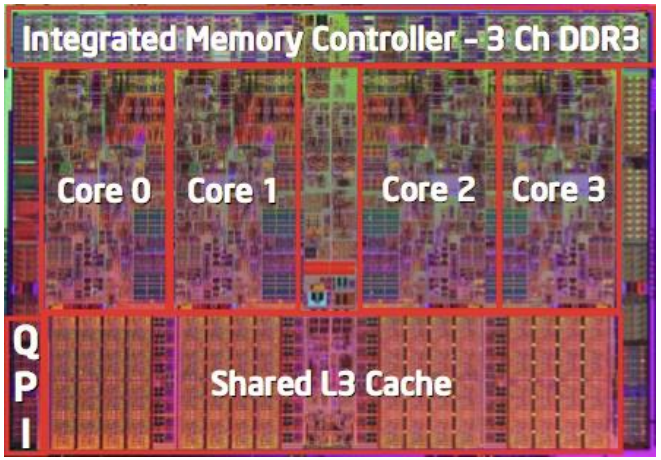
386	1985	275K

- 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 i5 2009 774M - Intel VT-x, 64비트, 멀티코어
- Core i7 2008 781M - Hyperthreading, multicore

추가된 특징

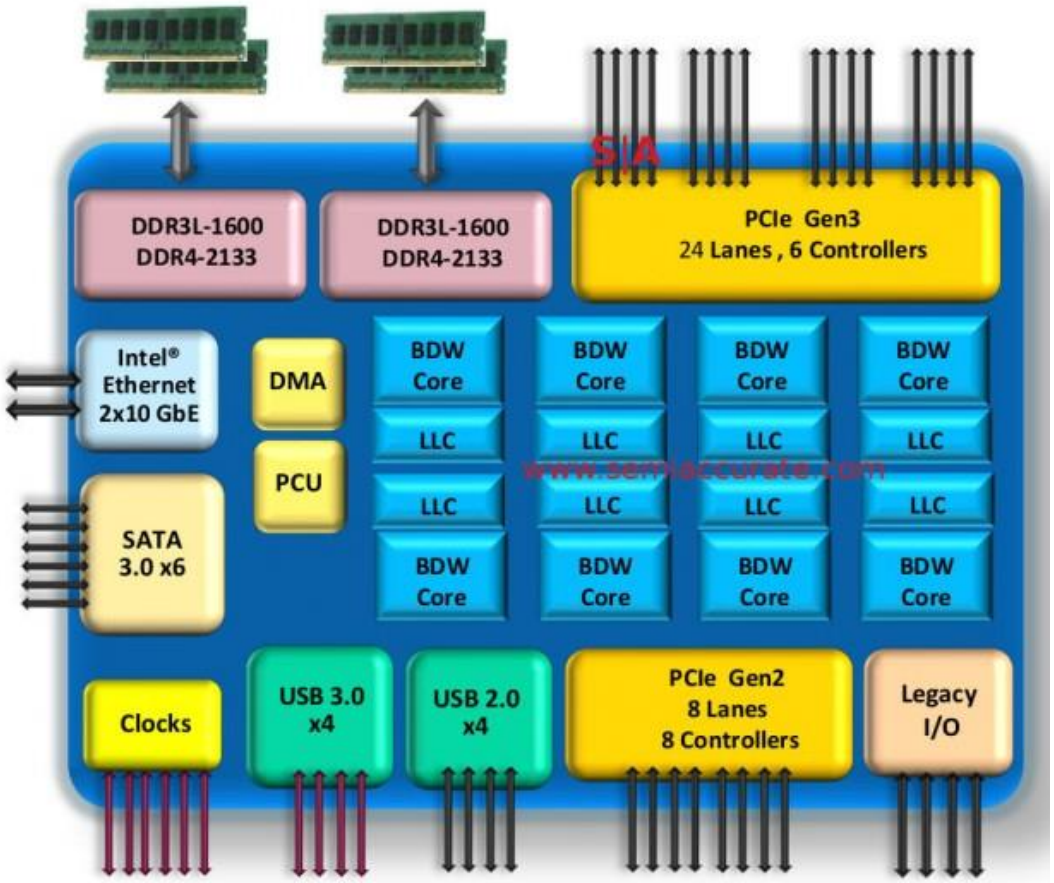
- 멀티미디어 연산 지원 명령어
  - 1, 2, 4바이트 데이터의 병렬 처리 연산 가능
- 효율적인 분기 명령어 => 이럴 필요가 있을까?
- 32비트에서 64비트로 전환

2015 State of the Art

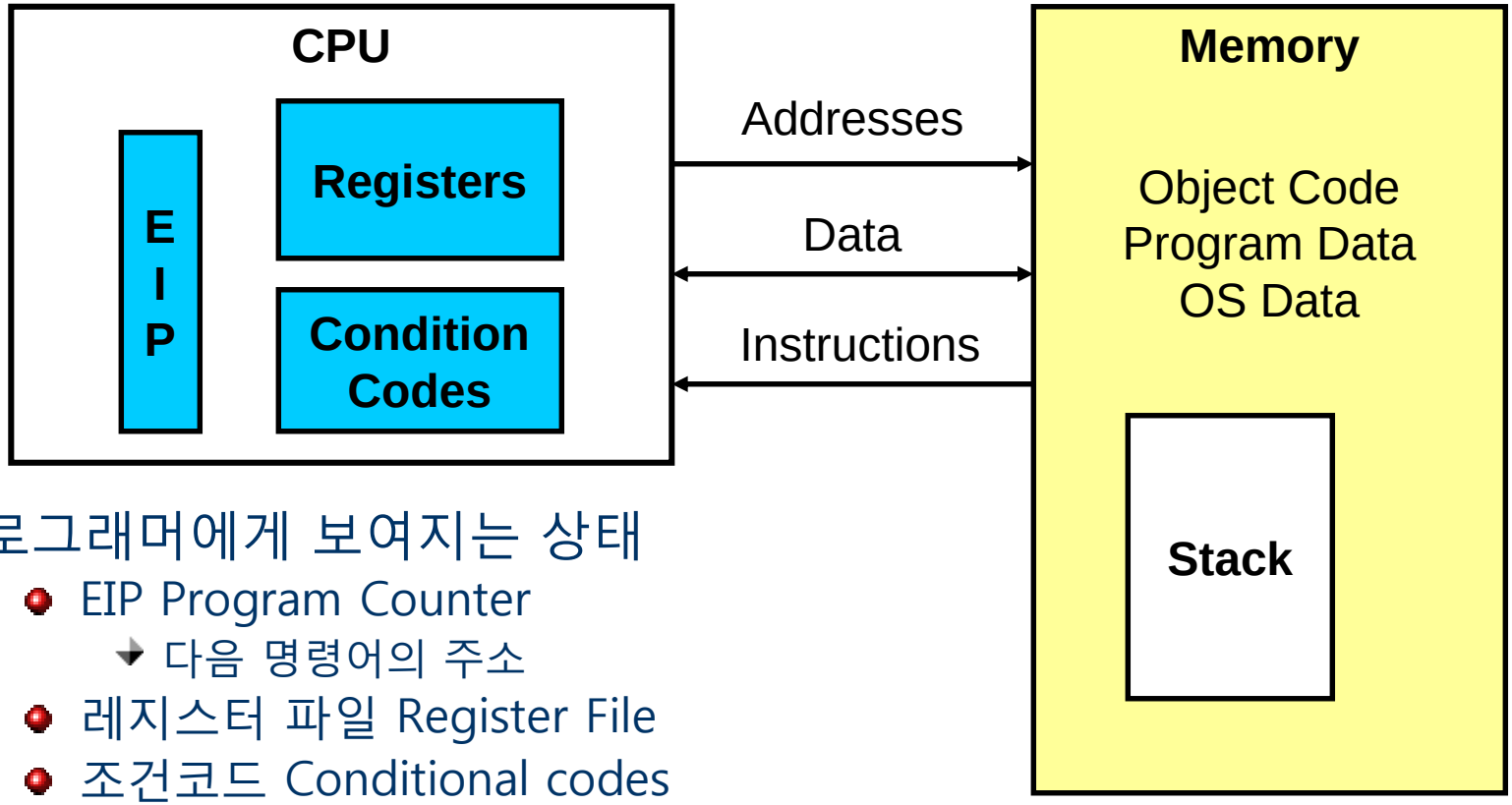
Core i7 Broadwell 2015

데스크탑 모델
- 4 코어
- 그래픽처리기 포함
- 3.3-3.8 GHz
- 65W

서버모델
- 8 코어
- 집적형 I/O
- 2-2.6 GHz
- 45W



어셈블리 프로그래머의 시야



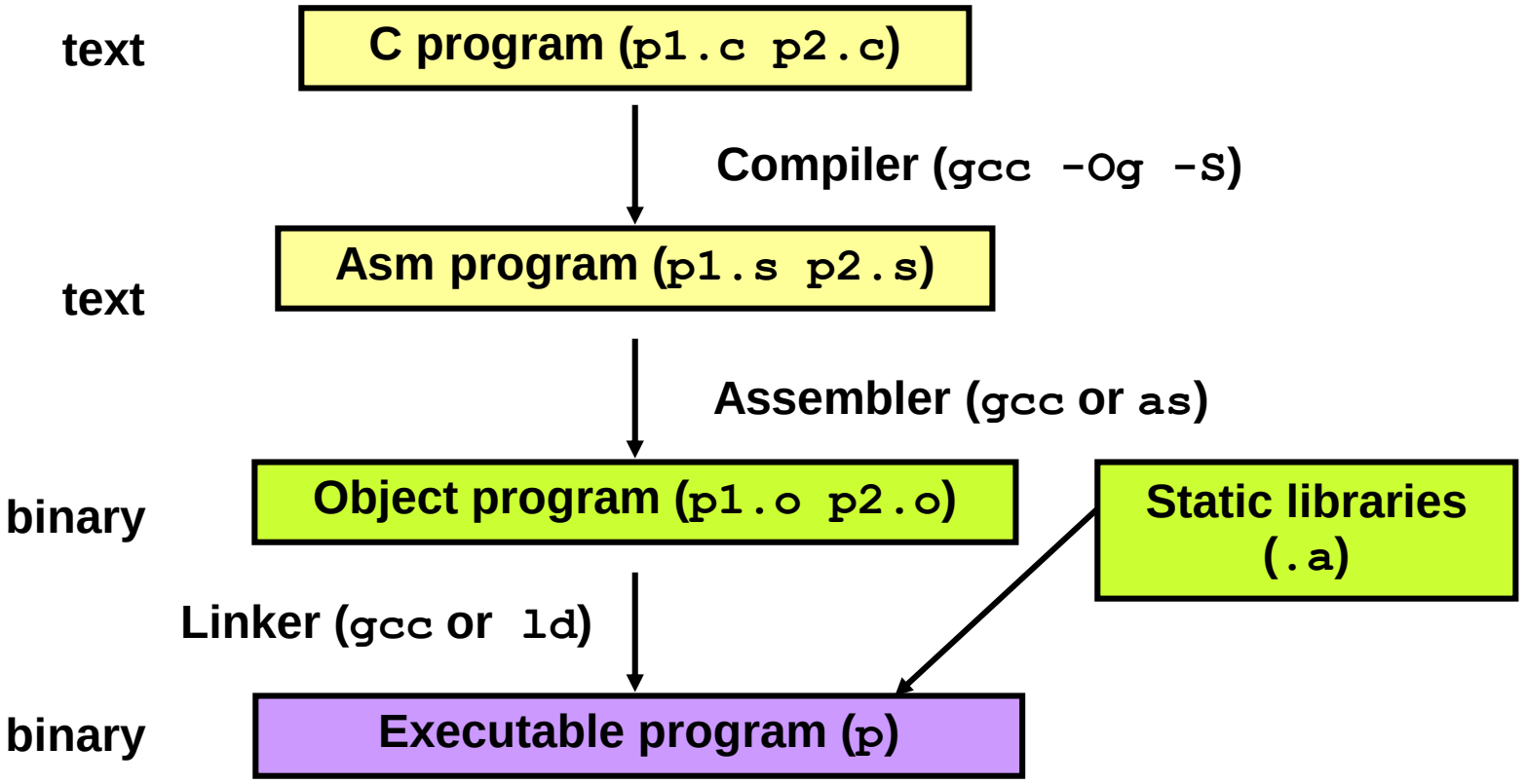
프로그래머에게 보여지는 상태

- 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 –Og p1.c p2.c -o p`
  - 최적화 옵션 optimizations (`-O`)
  - 바이너리 데이터를 `p` 에 저장



C 프로그램을 어셈블리어로 컴파일하기

C Code

```
long plus(long x, long y);

void sumstore(long x, long y,
              long *dest)
{
    long t = plus(x, y);
    *dest = t;
}
```

생성된 어셈블리 프로그램

```
sumstore:
    pushq   %rbx
    movq    %rdx, %rbx
    call    plus
    movq    %rax, (%rbx)
    popq    %rbx
    ret
```

다음의 명령으로 생성

gcc -Og -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)
  - 부동소숫점 데이터 4, 8, or 10 bytes
  - 배열이나 구조체가 없다
    - 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`

```
0x0400595:
     0x53
     0x48
     0x89
     0xd3
     0xe8
     0xf2
     0xff
     0xff
     0xff
     0x48
     0x89
     0x03
     0x5b
     0xc3
```

- **Total of 14 bytes**
- **Each instruction 1, 3, or 5 bytes**
- **Starts at address `0x400595`**

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

```
0000000000400595 <sumstore>:
  400595:  53                   push   %rbx
  400596:  48 89 d3             mov    %rdx,%rbx
  400599:  e8 f2 ff ff ff       callq  400590 <plus>
  40059e:  48 89 03             mov    %rax,(%rbx)
  4005a1:  5b                   pop    %rbx
  4005a2:  c3                   retq
```

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

```
0x0400595:
    0x53
    0x48
    0x89
    0xd3
    0xe8
    0xf2
    0xff
    0xff
    0xff
    0xff
    0x48
    0x89
    0x03
    0x5b
    0xc3
```

Disassembled

```
Dump of assembler code for function sumstore:
 0x0000000000400595 <+0>: push   %rbx
 0x0000000000400596 <+1>: mov    %rdx,%rbx
 0x0000000000400599 <+4>: callq  0x400590 <plus>
 0x000000000040059e <+9>: mov    %rax,(%rbx)
 0x00000000004005a1 <+12>:pop    %rbx
 0x00000000004005a2 <+13>:retq
```

- **gdb 디버거의 사용**

  `gdb p`

  `disassemble sum`

  - **프로시져 역어셈블하기**

  `x/14xb sumstore`

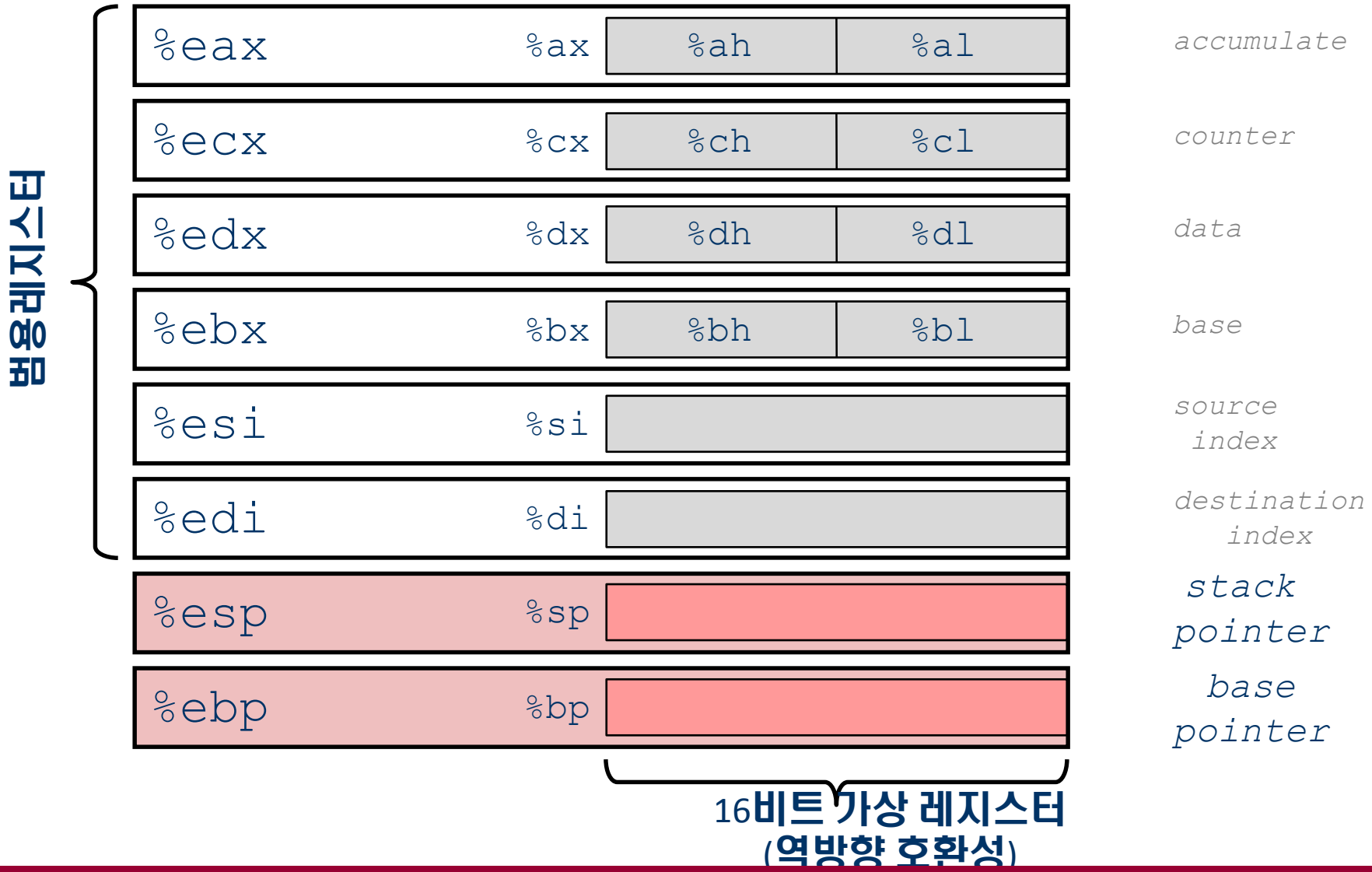
  - `sumstore` **에서 시작하여** `14`**바이트를 표시하라는 명령**

x86-64의 정수 레지스터

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

16개 레지스터는 바이트, 워드(16비트), 더블워드, 쿼드워드로 접근할수 있다

역사적인 고찰 : IA32 레지스터



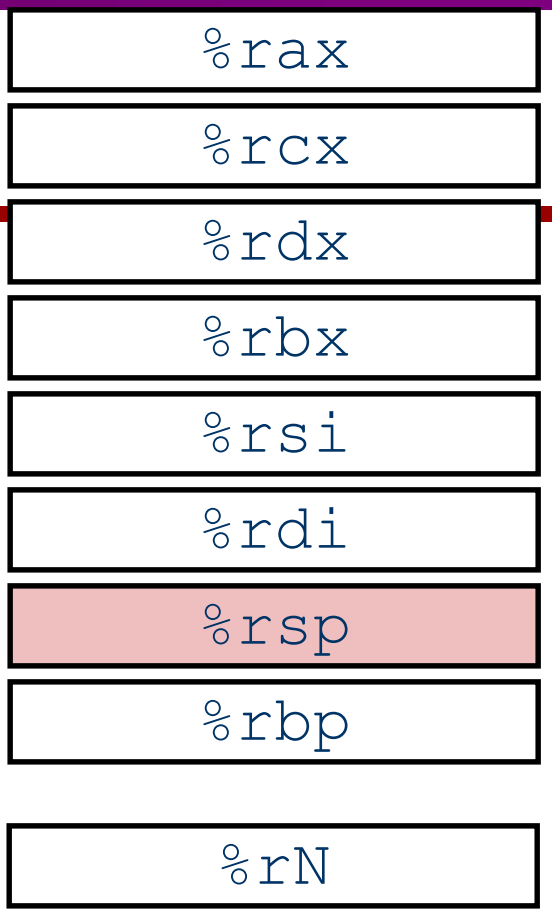
데이터의 이동

Moving Data

movq ***Source, Dest:***

오퍼랜드 유형

- ***Immediate:*** 상수 정수 데이터
  - Example: **$0x400, $-533**
  - C의 상수와 유사, 그러나 **'$'로 시작**
  - 1, 2, 4바이트로 인코드 됨
- ***Register:*** 16개 레지스터 중 한개
  - Example: **%rax, %r13**
  - 그러나 **%rsp는 특별한 목적에 이용**
  - 다른 레지스터들은 특수 인스트럭션에 이용될 수 있음
- ***Memory:*** 레지스터로 지정되는 주소에 저장된 8개의 연속적인 메모리 바이트
  - 가장 간단한 예: **(%rax)**
  - 여러가지 주소지정모드 존재"address modes"



movq 오퍼랜드 조합

	Source	Dest	Src,Dest	C 프로그램
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;`

memory-memory 데이터 이동은 한 개의 명령으로 불가능

데이터 형식

C declaration	Intel data type	Assembly-code suffix	Size (bytes)
`char`	Byte	`b`	1
`short`	Word	`w`	2
`int`	Double word	`l`	4
`long`	Quad word	`q`	8
`char *`	Quad word	`q`	8
`float`	Single precision	`s`	4
`double`	Double precision	`l`	8

간단한 메모리 주소지정 모드

일반형 (R) Mem[Reg[R]]

- 레지스터 R은 메모리 주소를 저장하고 있다
- 아하! 이것은 C언어에서 포인터 역참조와 동일

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

변위형 D(R) Mem[Reg[R]+D]

- 레지스터 R은 메모리 영역의 시작주소를 저장하고 있다
- 상수 변위 D는 오프셋을 표시

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

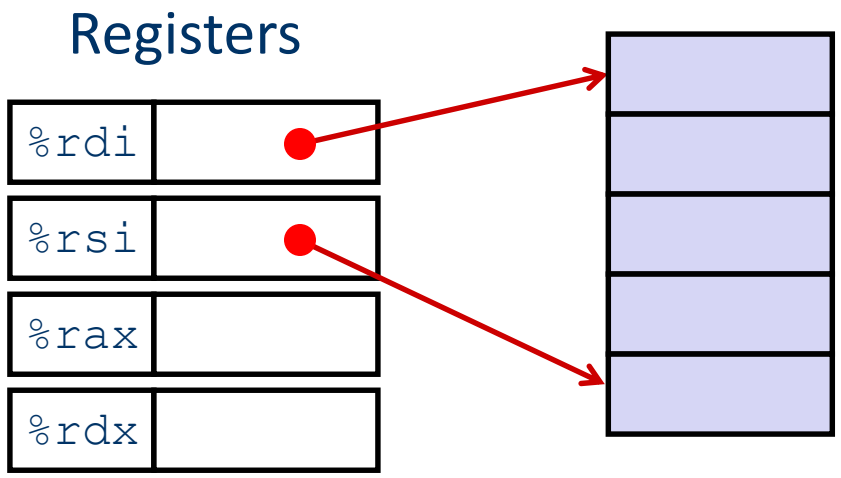
단순 주소지정 모드 예제

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

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

Swap()의 이해

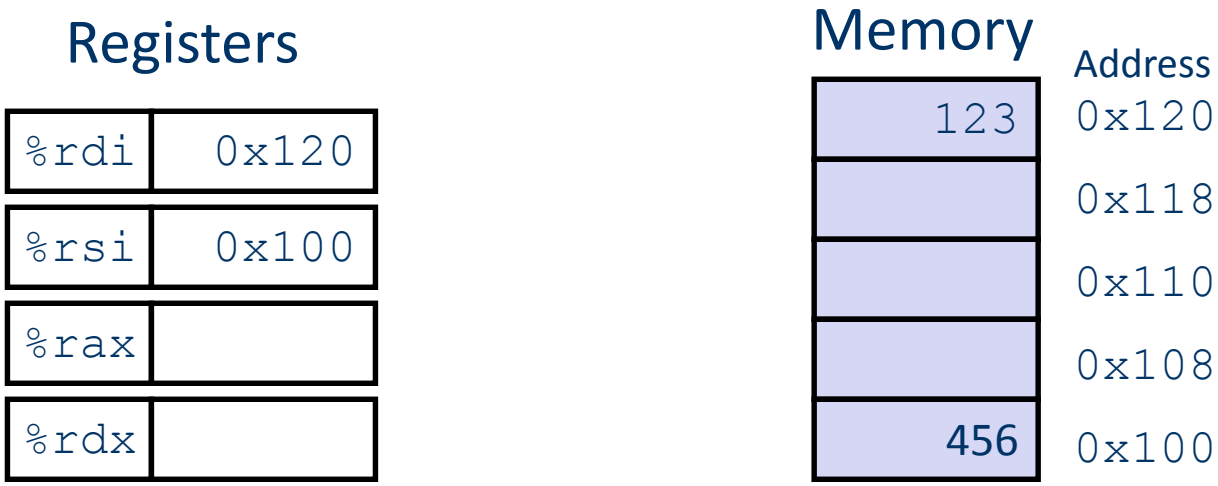
```
void swap
    (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



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

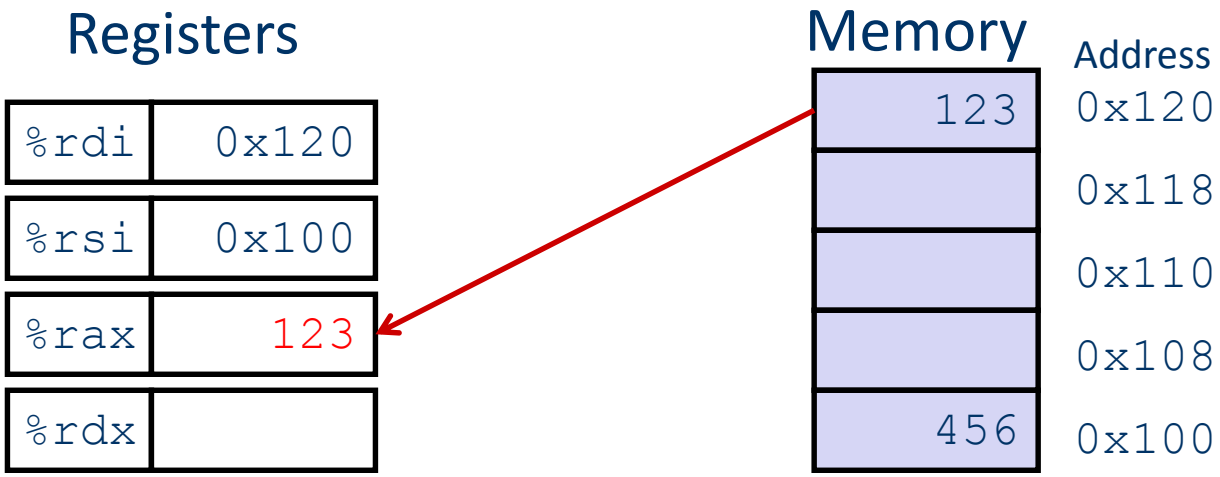
```
swap:
        movq    (%rdi), %rax  # t0 = *xp
        movq    (%rsi), %rdx  # t1 = *yp
        movq    %rdx, (%rdi)  # *xp = t1
        movq    %rax, (%rsi)  # *yp = t0
        ret
```

Swap()의 이해



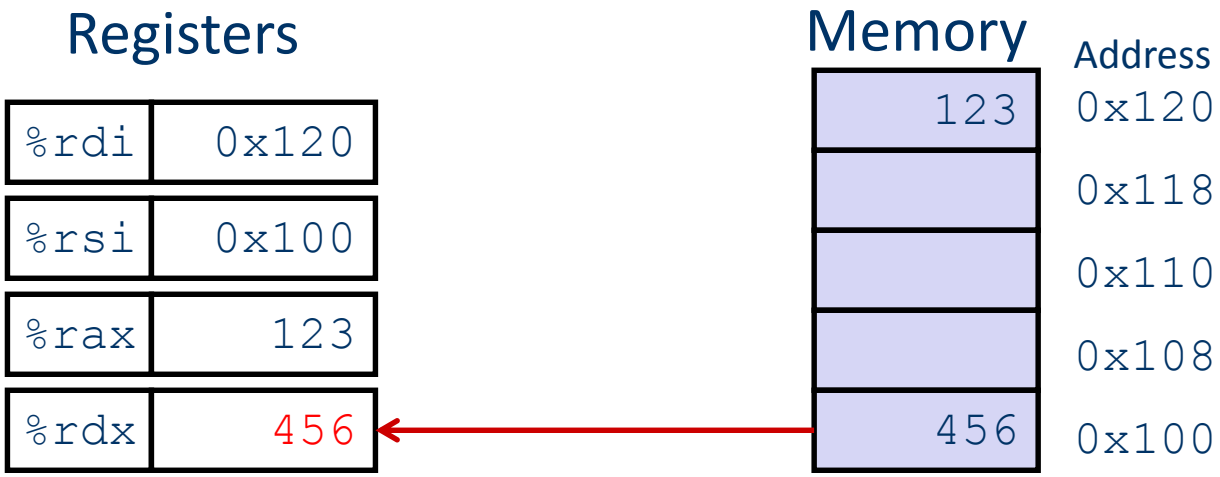

```
swap:
        movq    (%rdi), %rax  # t0 = *xp
        movq    (%rsi), %rdx  # t1 = *yp
        movq    %rdx, (%rdi)  # *xp = t1
        movq    %rax, (%rsi)  # *yp = t0
        ret
```

Swap()의 이해



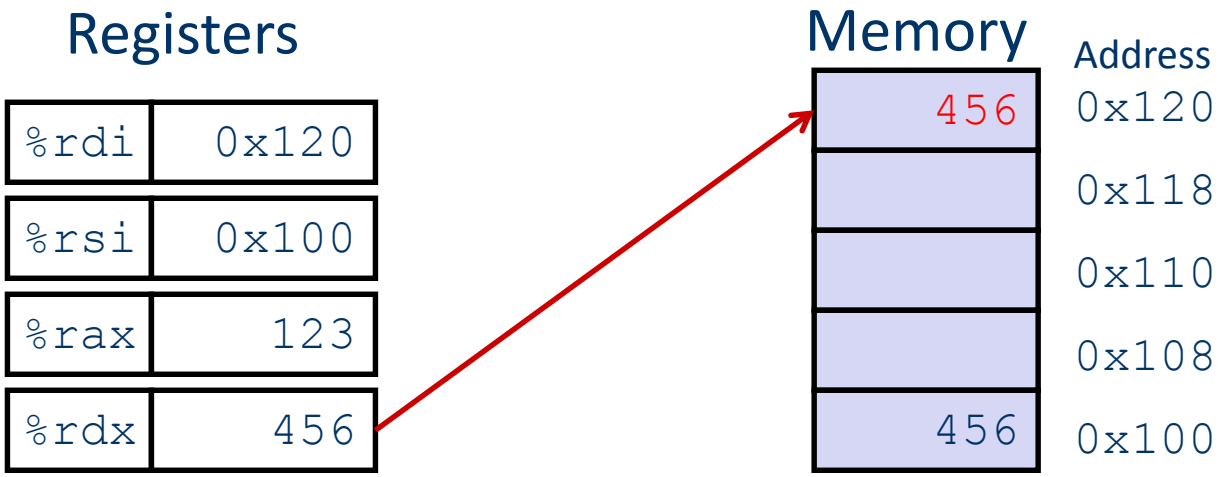

```
swap:
        movq    (%rdi), %rax  # t0 = *xp
        movq    (%rsi), %rdx  # t1 = *yp
        movq    %rdx, (%rdi)  # *xp = t1
        movq    %rax, (%rsi)  # *yp = t0
        ret
```

Swap()의 이해



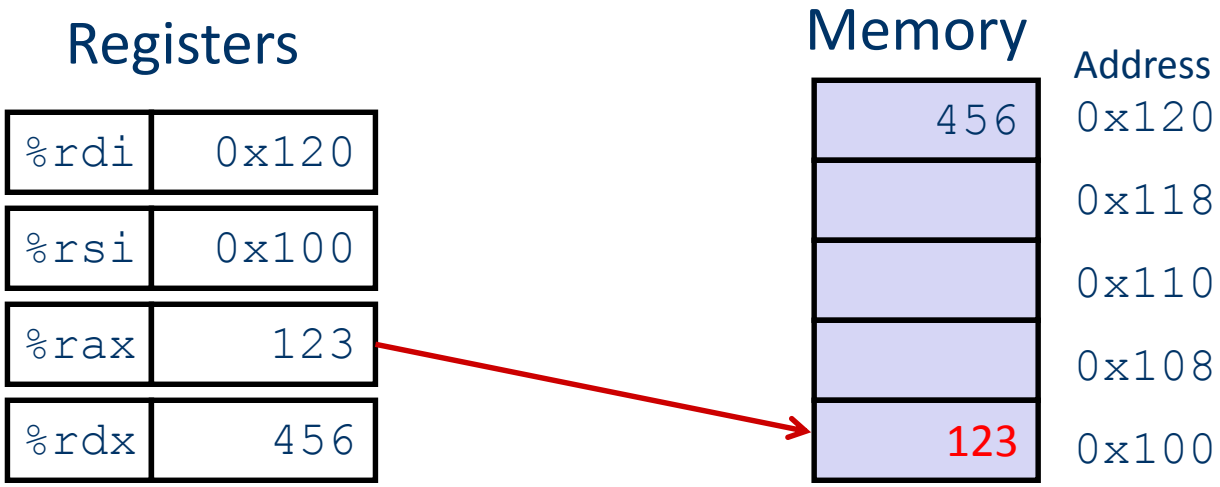

```
swap:
        movq    (%rdi), %rax  # t0 = *xp
        movq    (%rsi), %rdx  # t1 = *yp
        movq    %rdx, (%rdi)  # *xp = t1
        movq    %rax, (%rsi)  # *yp = t0
        ret
```

Swap()의 이해




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

Swap()의 이해




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

완전한 메모리 주소지정모드

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: 16개 레지스터 아무거나 가능
- Ri: 인덱스 레지스터: **`%rsp`를 제외한 아무거나 가능**
- S: 배율: 1, 2, 4, or 8 (하필 왜 이런 숫자들일까?)

특수형태

(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	
(%rdx,%rcx,4)		
0x80(,%rdx,2)	2*0xf000 + 0x80	0x1e080

주소계산 명령어

`leaq` *Src*, *Dst*

- *Src* 주소 모드 수식
- *Dst* 는 수식으로 표현된 주소값이 저장됨

용도

- 메모리 참조하지 않고 주소를 계산할 때
  - E.g., **`p = &x[i];`**
- x + k*y 형태의 계산을 할 때
  - k = 1, 2, 4, or 8

Example

```
long m12(long x)
{
  return x*12;
}
```

컴파일러가 생성한 코드:

```
leaq (%rdi,%rdi,2), %rax # t <- x+x*2
salq $2, %rax            # return t<<2
```

산술연산 명령어

2오퍼랜드 명령어:

Format	*Computation*		
`addq`	*Src,Dest*	Dest = Dest + Src	
`subq`	*Src,Dest*	Dest = Dest – Src	
`imulq`	*Src,Dest*	Dest = Dest * Src	
`salq`	*Src,Dest*	Dest = Dest << Src	***Also called shlq***
`sarq`	*Src,Dest*	Dest = Dest >> Src	***Arithmetic***
`shrq`	*Src,Dest*	Dest = Dest >> Src	***Logical***
`xorq`	*Src,Dest*	Dest = Dest ^ Src	
`andq`	*Src,Dest*	Dest = Dest & Src	
`orq`	*Src,Dest*	Dest = Dest | Src	

인자들의 순서에 주의할 것!

부호형과 비부호형 정수간에 차이가 없음 (why?)

산술연산 명령어

1오퍼랜드 명령어

`incq`	*DestDest = Dest + 1*
`decq`	*DestDest = Dest – 1*
`negq`	*DestDest = – Dest*
`notq`	*DestDest = ~Dest*

산술연산 예제

```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
   leaq    (%rdi,%rsi), %rax
   addq    %rdx, %rax
   leaq    (%rsi,%rsi,2), %rdx
   salq    $4, %rdx
   leaq    4(%rdi,%rdx), %rcx
   imulq   %rcx, %rax
   ret
```

유의해야 하는 명령어

- **leaq**: 주소계산
- **salq**: shift
- **imulq**: multiplication
  - 단 한번만 사용되었음

산술연산 수식 이해하기

```
long arith
(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
   leaq    (%rdi,%rsi), %rax    # t1
   addq    %rdx, %rax           # t2
   leaq    (%rsi,%rsi,2), %rdx
   salq    $4, %rdx             # t4
   leaq    4(%rdi,%rdx), %rcx   # t5
   imulq   %rcx, %rax           # rval
   ret
```

Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rdx	Argument **z**
%rax	**t1**, **t2**, **rval**
%rdx	**t4**
%rcx	**t5**