3장. 프로그램의 기계수준 표현

기계어 코드 (Machine Code)

- 3장에서 기계어 코드와 이를 읽기 쉽게 문자로 표현한 어셈블리 코드 (assembly code)를 소개
  - 컴퓨터는 연속적인 바이트로 인코딩된 기계어 코드를 실행
  - 데스크탑, 랩탑, 서버, 슈퍼컴퓨터 등에 널리 사용되는 x86-64 기계어 코드 소개
- 컴파일러는 프로그램 언어의 규칙, 명령어 집합, 운영체제 등을 고려하여 기계어 코드를 생성
  - 과거에는 프로그래머가 작접 어셈블리 프로그램을 작성하였지만, 지금은 컴파일러가 생성한 코드를 이해하는 것이 중요
  - GCC는 어셈블러와 링커를 호출하여 실행 가능한 기계어 코드 생성

기계어 코드를 배워야 하는 이유

- 프로그래머는 컴파일러가 생성한 기계어 코드를 이해하는 것이 중요
  - 컴퓨터의 기본적인 내부 동작 및 구조 이해
  - 컴파일러의 최적화 성능, 생성된 코드의 비효율성을 분석 (5장)
- 프로그램의 런타임(run time) 동작 이해
  - 쓰레드들의 데이터 공유 및 접근 이해 (12장)
  - 프로그램 보안 취약성 이해
    - 악성 프로그램이 프로그램의 런타임 제어 정보를 변경하고 시스템 제어 권을 획득
    - 프로그램의 보안 취약성을 이해하고 방어하려면 프로그램의 기계수준 표현에 대한 지식이 필요

3.1 역사적 관점

인텔 x86 프로세서

- x86으로 불리우는 인텔 프로세서들은 오랜 기간동안 진화되어 개발
  - 1978년 16비트의 8086으로 시작하여 호환성을 유지하며 개발

- CISC (Complex Instruction Set Computer)
  - 다양한 형식의 많은 기계어 명령어
    - 리눅스 프로그램에서는 이 중 일부의 명령어만 자주 사용
  - RISC (Reducded Instruction Set Computer)의 성능을 따라가기 어려웠음
    - 호환성을 위해 CISC 모델 적용
  - 인텔은 다양한 기술적인 개발을 통해 속도 및 전력 소모 등에서 RISC의 성능에 도달

x86 진화 (1)

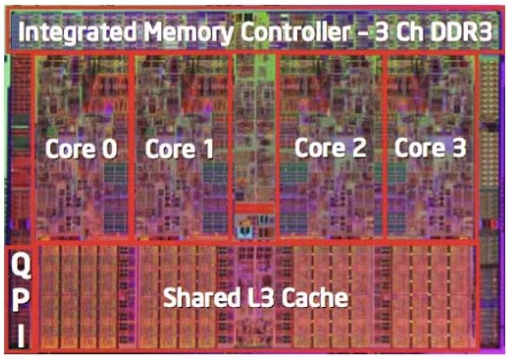
Name	Date	Transistors	MHz
8086	1978	29K	5-10
386	1985	275K	16-33
Pentium 4E	2004	125M	2800-3800
Core2	2006	291M	1060-3500
Core i7	2008	731M	1700-3900

- 8086
  - 최초의 단일칩 16비트 마이크로프로세서, IBM PC (DOS)에 사용
  - 1MB 메모리 공간
- 386
  - 인텔의 첫 32비트 프로세서, IA-32로 불리움
- Pentium 4E
  - 인텔의 첫 64비트 프로세서, x86-64로 불리움
- Core2
  - 인텔의 첫 멀티코아 프로세서
- Core i7
  - 4 코아 프로세서

x86 진화 (2)

- 머신 진화

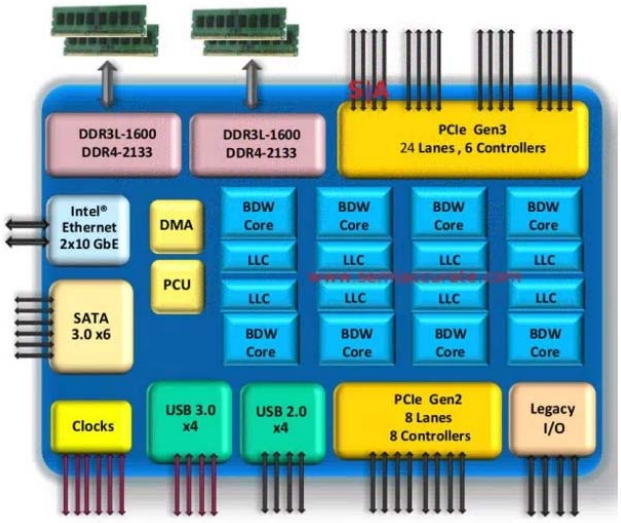
386	1985	0.3M
Pentium	1993	3.1M
Pentium/MMX	1997	4.5M
PentiumPro	1995	6.5M
Pentium III	1999	8.2M
Pentium 4	2001	42M
Core 2 Duo	2006	291M
Core i7	2008	731M



- 추가된 기능
  - 멀티미디어 연산을 지원하는 명령어
  - 효율적인 조건 연산을 지원하는 명령어
  - 32비트에서 64비트로 진화
  - 여러개의 코아

Core i7

- Core i7 Broadwell (2015년 기준)
- 데스크탑 모델
  - 4 코아
  - 그래픽 기능 내장
  - 3.3-3.6 GHz
  - 65W
- 서버 모델
  - 8 코아
  - I/O 제어기 내장
  - 2-2.6 GHz
  - 45W



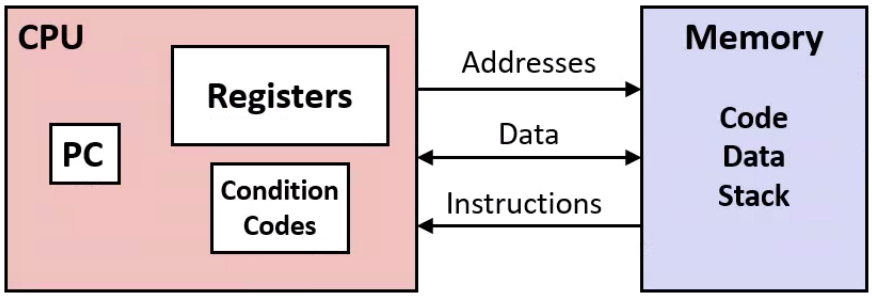
3.2 프로그램의 인코딩

용어 정의

- 아키텍처 (또는 ISA: Instruction Set Architecture)
  - 어셈블리/기계어 코드를 작성 및 이해하는데 필요한 프로세서 설계의 부분
  - 예: 명령어 집합의 사양, 레지스터
- 마이크로아키텍처 (Microarchitecture)
  - 아키텍처의 구현
  - 예: 캐시 크기, 코어의 클럭 주파수
- 코드
  - 기계어 코드: 프로세서가 실행하는 바이트 수준의 프로그램
  - 어셈블리 코드: 머신 코드의 텍스트 형태 표현
- ISA 예
  - 인텔: x86, IA32, Itanium, x86-64
  - ARM

3-1.

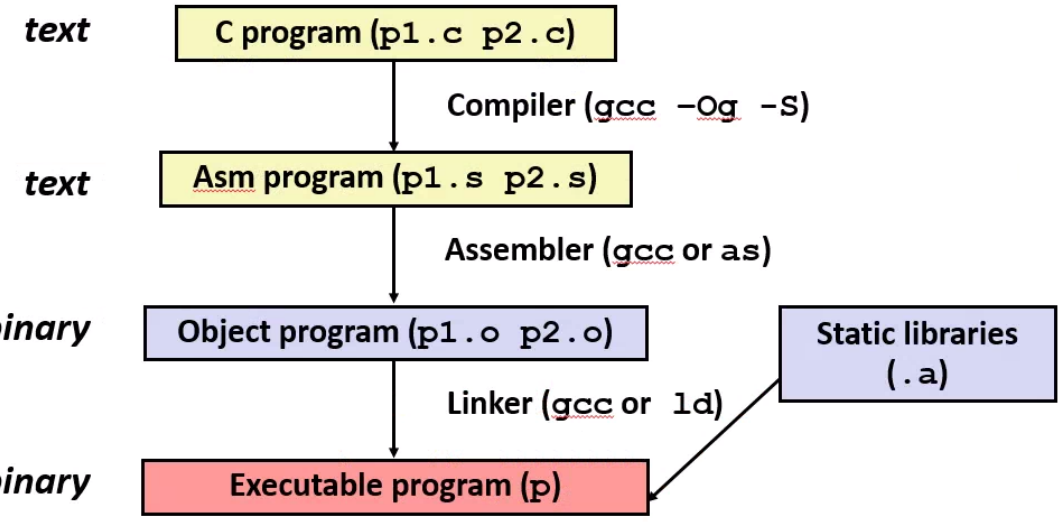
어셈블리/기계어 코드의 관점



- 프로그래머 관점 (programmer-visible)의 상태
  - PC: 프로그램 카운터 (Program Counter)
    - 다음에 실행될 명령어의 주소
    - RIP (x86-64) 라고도 함
  - 레지스터 파일 (Register File)
    - 자주 사용되는 프로그램 데이터 저장
  - 조건 코드 (Condition Codes)
    - 최근에 수행된 산술 및 논리 연산 결과에 대한 상태 저장
    - 조건 분기 명령에서 사용
  - 메모리 (Memory)
    - 바이트 주소지정 배열
    - 프로그램 코드와 데이터
    - 프로시저 호출을 지원하는 스택

GCC 컴파일

- GCC 컴파일러 명령
  - p1.c, p2.c의 소스를 컴파일하여 바이너리 파일 p 생성
    linux> gcc -Og p1.c p2.c -o p
    - 기본 최적화 옵션 (-Og)



어셈블리 코드 생성

- 어셈블리 코드 생성(-S 옵션) 컴파일
  linux> gcc -Og -S mstore.c
  - 생성되는 어셈블리 코드(mstore.s) 는 수행되는 머신, 운영체제에 따라 조금씩 달라짐
  - 아래 sum.s에서 지역 변수 이름이나 데이터 타입 정보는 생략

(mstore.c)

```
long mult2(long, long);

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

(mstore.s)

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


참고: gcc 최적화 옵션

- gcc는 다양한 수준의 최적화 옵션 제공
  - 최적화 수준이 높아질수록 컴파일 시간 및 메모리 사용 증가
- gcc 최적화 옵션
  - -O0 : 최적화하지 않음
  - -O, O1: 코드 크기와 실행 시간을 줄이는 기본적인 최적화
  - -O2: 메모리 공간을 크게 차지하고 속도를 증가시키는 최적화를 제외한 최적화
    - 루프 펼치기(loop unrolling, 인라임 함수 최적화 제외
  - -O3: O2 최적화에 인라인 함수 및 레지스터 할당 최적화 추가
  - -Os: O2 최적화를 사용하지만 코드의 크기 증가는 피함
  - -Og: 디버깅 시에 문제를 일으키는 최적화 (코드 이동 등) 제외

GCC 생성 어셈블리 코드

- GCC 컴파일러가 생성한 어셈블리 코드
  - 명령어 뿐만 아니라 어셈블러와 컴파일러에 지시하기 위한 '.'으로 시작하는 디렉티브(directive. 지시어)도 포함

```
linux> cat mstore.s
        .file   "mstore.c"
        .text
        .globl  multstore
        .type   multstore, @function
multstore:
.LFB0:
        .cfi_startproc
        pushq   %rbx
        .cfi_def_cfa_offset 16
        .cfi_offset 3, -16
        movq    %rdx, %rbx
        call    mult2@PLT
        movq    %rax, (%rbx)
        popq    %rbx
        .cfi_def_cfa_offset 8
        ret
        .cfi_endproc
.LFE0:
        .size   multstore, .-multstore
        .ident  "GCC: (Ubuntu 7.4.0-1ubuntu1~18.04) 7.4.0"
        .section        .note.GNU-stack,"",@progbits
linux>
```


목적코드 생성

- 목적코드 생성(-c 옵션) 컴파일
  linux> gcc -Og -c mstore.c
  - 바이너리 형식의 목적코드(mstore.o) 생성
  - 기계어 파일을 보려면 objdump 명령으로 역어셈블리

```
linux> objdump -d mstore.o

mstore.o:     file format elf64-x86-64


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
linux>
```

목적코드

```
linux> objdump -d mstore.o

mstore.o:     file format elf64-x86-64


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
linux>
```

- 역어셈블된 목적코드
  - 총 14바이트의 명령어 코드
  - x86-64 명령어들은 1에서 15바트의 다양한 크기를 가짐
  - push %rbx 명령의 기계어 코드는 53
  - 많은 명령어에서 접미어 'q' 생략
    - 접미어는 오퍼랜드 데이터의 크기를 나타냄

목적코드 – 기계어 코드 생성 예

- C 코드: *dest = t;
  - dest 포인터가 지정한 위치로 t의 값을 저장
- 어셈블리: movq %rax, (%rbx)
  - 쿼드 워드(wuard word)의 8 바이트 값을 메모리에 저장
  - 오퍼랜드
    - t: 레지스터 %rax
    - dest 레지스터 %rbx
    - *dest 메모리 M[%rbx]
- 목적코드: 9: 48 89 03
  - 3바이트 명령어

실행파일 생성 (1)

- 실제 실행 가능한 코드 생성을 위해서는 목적코드들에 대해 링커(linker)가 수행되어야 함
  - 최소한 한 개의 파일은 main 함수를 포함해야 함
    - mulmain.c 파일

```
#include <stdio.h>
void multstore(long, long, long *);

int main()
{
   long d;
   multstore(2, 3, &d);
   printf("2 * 3 → &ld₩n", d);
}

long mult2(long a, long b)
{
    long s = a * b;
    return s;
}
```


실행파일 생성 (2)

- 실행파일 생성 컴파일
  linux> gcc -Og -o prog mulmain.c mstore.c
  - 실행파일(prog) 생성
  - 실행파일에는 두 프로그램의 목적코드 외에 코드가 더 추가
    - 운영체제와 상호 작용을 위한 코드
    - 프로그램을 시작하고 종료하기 위한 드
  - objdump로 역어셈블하고 multstore 부분 확인
    - 링커가 코드의 위치 주소 지정
    - callq 명령에서 mult2 호출 시에 주소도 채움

```
0000000000000741 <multstore>:
 741:   53                      push   %rbx
 742:   48 89 d3                mov    %rdx,%rbx
 745:   e8 ef ff ff ff          callq  739 <mult2>
 74a:   48 89 03                mov    %rax,(%rbx)
 74d:   5b                      pop    %rbx
 74e:   c3                      retq
 74f:   90                      nop
```

3.3 데이터의 형식

데이터의 형식

- C의 기본 데이터 타입에 대한 x86-64 표시

C 데이터 타입	인텔 데이터 타입	어셈블리 코드 접미어	크기 (바이트)
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

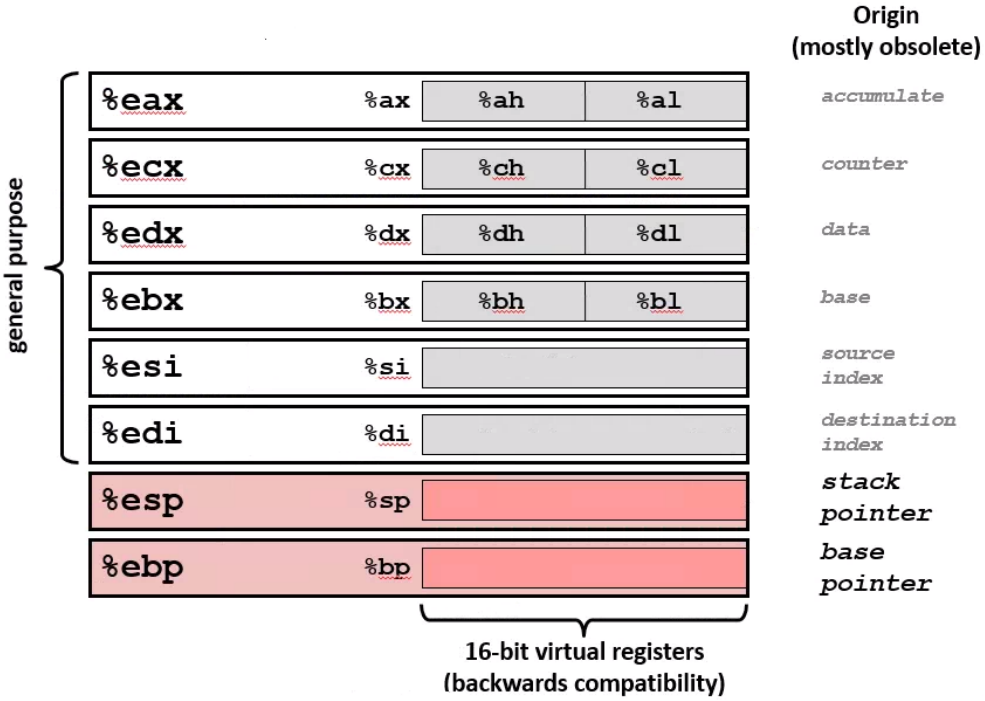
3.4 정보 접근하기

x86-64
정수 레지스터 (Integer Registers)

- 16개의 64 비트 범용 레지스터 (general purpose gegister)
  - 하위 32 비트 (IA32), 16 비트 (8086), 8 비트도 접근 가능

%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

IA32 레지스터



오퍼랜드

- 대부분의 명령어는 하나 이상의 오퍼랜드(operand, 피연산자)를 가짐
  - 오퍼랜드는 즉시값(immediate, 상수)이나 레지스터와 메모리에 저장된 값을 표시
  - 어셈블리 코드에서 상수는 $, 레지스터는 %로 시작, 메모리 주소는 ()안에 표시
- 주소지정방식 (addressing mode)
  - 실제 오퍼랜드의 값이 저장된 메모리의 위치인 유효주소(effective address)를 계산하는 방식
  - 오퍼랜드의 형식을 나타냄

x86-64 오퍼랜드 형식 (주소지정 방식)

타입	형식	오퍼랜드 값	이름
상수	$Imm	Imm	즉시값 (Immediate)
레지스터	r_a	R[r_a]	레지스터
메모리	Imm	M[Imm]	절대값 (Absolute)
메모리	(r_a)	M[R[r_a]]	간접 (Indirect)
메모리	Imm(r_b)	M[Imm+R[r_b]]	베이스+오프셋
메모리	(r_a, r_i)	M[R[r_a]+R[r_i]]	인덱스 (Index)
메모리	Imm(r_a, r_i)	M[Imm+R[r_a]+R[r_i]]	인덱스 (Index)
메모리	(,r_i,s)	M[R[r_i]*s]	스케일 인덱스 (Scaled index)
메모리	Imm(,r_i,s)	M[Imm+R[r_i]*s]	스케일 인덱스 (Scaled index)
메모리	(r_b,r_i,s)	M[R[r_b]+R[r_i]*s]	스케일 인덱스 (Scaled Index)
메모리	Imm(r_b,r_i,s)	M[Imm+R[r_b]+R[r_i]*s]	스케일 인덱스 (Scaled Index)

어셈블리에서 오퍼랜드 사용 (주소지정 방식) 예

- mov %rbx, %rdx — *rdx <= rbx*
- add (%rdx), %r8 — *r8 <= r8 + M[rdx]*
- mul $3, %r8 — *r8 = r8 * 3*
- sub $1, %r8 — *r8 = r8 -1*
- lea (%rdx,%rbx,2), %rdx — *rdx <= rdx + rbx*2*
  - load effective address
  - 메모리에 저장된 값을 참조(dereference)하지 않고 주소만 적재

연습문제 3.1 – 오퍼랜드의 값

- 메모리 주소와 레지스터에 저장된 값 표시

주소	값
0x100	0xFF
0x104	0xAB
0x108	0x13

레지스터	값
%rax	0x100
%rcx	0x1
%rdx	0x3

- %rax, 0x100, 레지스터
- 0x104, 0xAB, 절대값
- $0x108, 0x108, 상수
- (%rax), 0xFF, 주소 0x100
- 4(%rax), 0xAB, 주소 0x104
  - 4+0x100
- 9(%rax,%rdx), 0x11, 주소 0x10C
  - 9+0x100+0x3
- 260(%rcx,%rdx), 0x13, 주소 0x108
  - 260(0x104)+0x1+0x3
- 0xFC(,%rcx,4), 0xFF, 주소 0x100
  - 0xFC+0x1*4
- (%rax,%rdx,4), 0x11, 주소 0x10C
  - 0x100+0x3*4

데이터 이동 명령어

- 데이터를 한 위치에서 다른 위치로 복사 (8바이트, 쿼드워드)

  movq *Src, Dest*
  - *Dest <= Src*
  - 바이트, 2바이트, 4바이트 이동 시에는 movb, movw, movl 명령 사용
  - 메모리에서 메모리로 이동은 할 수 없음

	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;

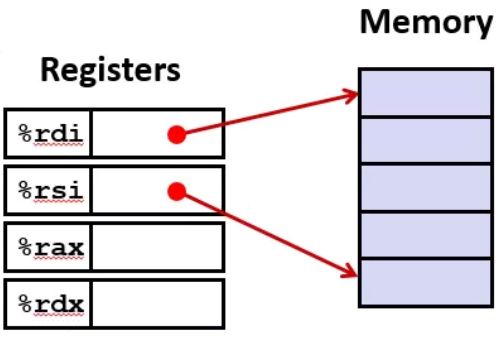
swap() 함수 예 (1)

```
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() 함수 예 (2)

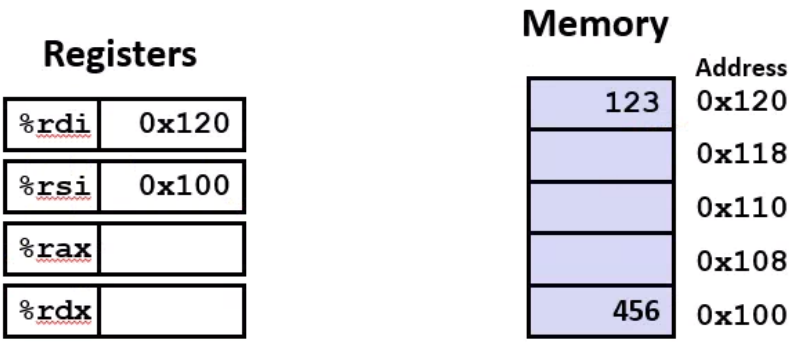
```
void swap
    (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



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

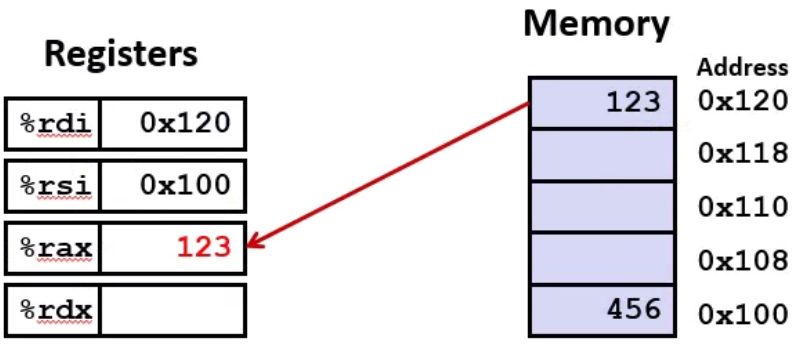
```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

swap() 함수 예 (3)



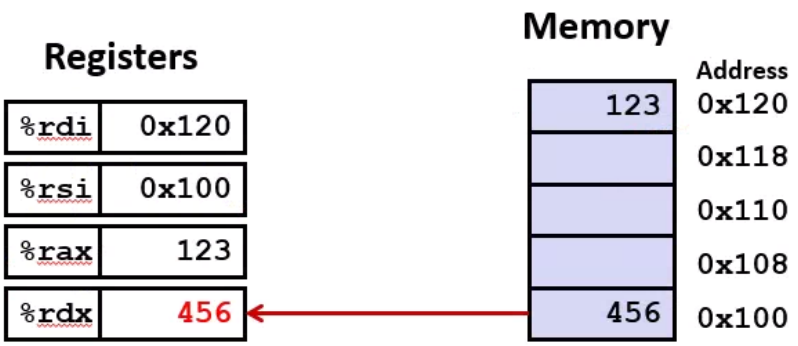

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```


swap() 함수 예 (4)



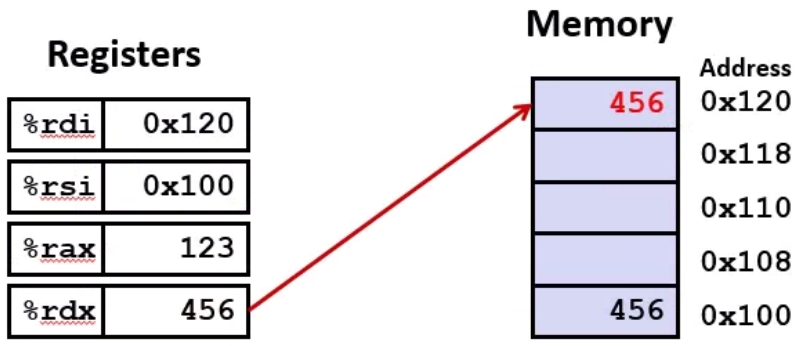

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```

swap() 함수 예 (5)



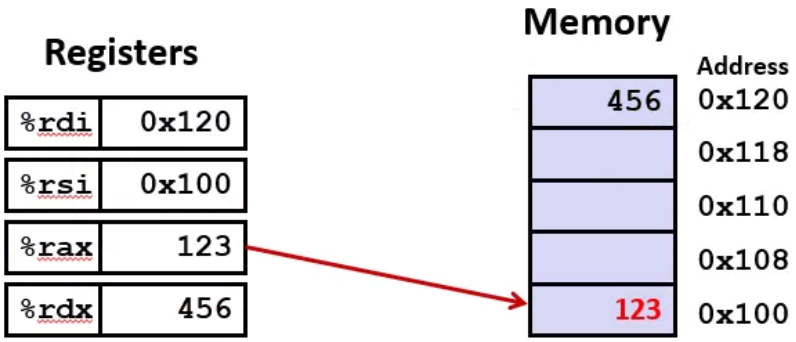

```
swap:
    movq    (%rdi), %rax  # t0 = *xp
    movq    (%rsi), %rdx  # t1 = *yp
    movq    %rdx, (%rdi)  # *xp = t1
    movq    %rax, (%rsi)  # *yp = t0
    ret
```


swap() 함수 예 (6)




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

swap() 함수 예 (7)




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


과제 3-1: swap() 함수의 gdbgui 실행 (실습과제)

- 가상머신에서 앞의 swap() 함수를 아래와 같이 C 프로그램을 작성하고 컴파일 한 후 실행파일을 gdbgui로 실행하여 메모리 및 레지스터 등의 상태 추적
  - 실행파일 생성을 위해 main() 함수 추가
  - 두 가지 버전을 컴파일하여 swap() 함수의 어셈블리 코드도 비교 분석
    - 최적화 버전: -Og -g 옵션으로 컴파일
    - 비최적화 버전: -g 옵션
  - 주요 단계 실행 및 상태를 추적 캡쳐하고 설명
    - 함수의 어셈블리 언어는 실행 후 함수 진입하면 나타남
  - 실습 강의노트 2 참조

3.5 산술연산과 논리연산

유효주소 계산 명령어

- leq *Src, Dst*
  - *Dst <= Src의 유효주소*
  - *Src*는 주소지정 모드의 표현
  - 메모리를 참조하지 않고 주소만 계산 시에 사용
    - 예: p = &x[i];
  - x + k*y 형식의 산술식 계산에도 사용
    - k = 1,2,4,8
    - 예

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

Converted to ASM by compiler:

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

산술연산

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	
incq	Dest	Dest = Dest + 1	
decq	Dest	Dest = Dest – 1	
negq	Dest	Dest = – Dest	
notq	Dest	Dest = ~Dest	

- 오퍼랜드 크기에 따라 연산의 접미어는 b, w, l, q (이 후 동일)

단항 산술연산

incq	Dest	Dest = Dest + 1
decq	Dest	Dest = Dest – 1
negq	Dest	Dest = – Dest
notq	Dest	Dest = ~Dest

산술연산 예 (1)

```
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
```

Interesting Instructions

- leaq: address computation
- salq: shift
- imulq: multiplication

산술연산 예 (2)

```
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

과제 3-2: arith() 함수의 gdbgui 실행 (실습과제)

- 가상머신에서 앞의 arith() 함수를 아래와 같이 C 프로그램을 작성하고 컴파일 한 후 실행파일을 gdbgui로 실행하여 메모리 및 레지스터 등의 상태 추적
  - 실행파일 생성을 위해 main() 함수 추가
  - 두 가지 버전을 컴파일하여 swap() 함수의 어셈블리 코드도 비교 분석
    - 최적화 버전: -Og -g 옵션으로 컴파일
    - 비최적화 버전: -g 옵션
  - 주요 단계 실행 및 상태를 추적 캡쳐하고 설명
    - 함수의 어셈블리 언어는 실행 후 함수 진입하면 나타남
  - 실습 강의노트 2 참조

요 약

요약

- C, 어셈블리, 기계어 코드
  - 컴파일러는 문장, 식, 프로시저 등을 저수준의 기계어 명령어들로 변환
  - 프로그래머 관점에서 프로그램 카운터, 레지스터, 메모리 등의 상태를 볼 수 있음
- 어셈블리 명령어는 하나 이상의 오퍼랜드(operand,피연산자)를 가짐
  - 오퍼랜드는 즉시값(immediate, 상수)이나 레지스터와 메모리에 저장된 값을 표시
    - 메모리 참조의 주소지정방식 (addressing mode)은 실제 오퍼랜드의 값이 저장된 메모리의 위치인 유효주소(effective address)를 계산하는 방식
- 데이터 이동 명령어, 산술 명령어

과 제

과제 3-1: swap() 함수의 gdbgui 실행 (실습과제)

- 가상머신에서 앞의 swap() 함수를 아래와 같이 C 프로그램을 작성하고 컴파일 한 후 실행파일을 gdbgui로 실행하여 메모리 및 레지스터 등의 상태 추적
  - 실행파일 생성을 위해 main() 함수 추가
  - 두 가지 버전을 컴파일하여 swap() 함수의 어셈블리 코드도 비교 분석
    - 최적화 버전: -Og -g 옵션으로 컴파일
    - 비최적화 버전: -g 옵션
  - 주요 단계 실행 및 상태를 추적 캡쳐하고 설명
    - 함수의 어셈블리 언어는 실행 후 함수 진입하면 나타남
  - 실습 강의노트 2 참조

과제 3-2: arith() 함수의 gdbgui 실행 (실습과제)

- 가상머신에서 앞의 arith() 함수를 아래와 같이 C 프로그램을 작성하고 컴파일 한 후 실행파일을 gdbgui로 실행하여 메모리 및 레지스터 등의 상태 추적
  - 실행파일 생성을 위해 main() 함수 추가
  - 두 가지 버전을 컴파일하여 swap() 함수의 어셈블리 코드도 비교 분석
    - 최적화 버전: -Og -g 옵션으로 컴파일
    - 비최적화 버전: -g 옵션
  - 주요 단계 실행 및 상태를 추적 캡쳐하고 설명
    - 함수의 어셈블리 언어는 실행 후 함수 진입하면 나타남
  - 실습 강의노트 2 참조