System Software Laboratory

KU KONKUK UNIVERSITY

# System Programming

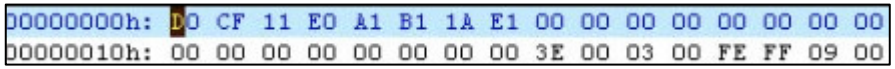
Hyun-Wook Jin
System Software Laboratory
Department of Computer Science & Engineering
Konkuk University
jinh@konkuk.ac.kr

# Inline Assembly

- Assembly 언어란?
- Assembly & C/C++
- Inline Assembly
- Objdump
- Examples

# Assembly 언어란?

- 컴퓨터 언어의 3단계
  - 기계어
  - 어셈블리어
  - 고급언어




```
sub    $0x8,%rsp
mov    $0x1e,%edx
mov    $0x400614,%esi
mov    $0x1,%edi
mov    $0x0,%eax
callq  0x400460 <__printf_chk@plt>
mov    $0x0,%eax
add    $0x8,%rsp
retq
```

```
int main(void) {
        int a = 10;
        int b = 20;
        int r = 0;

        r = a + b;
        printf("result: %d\n", r);
        return 0;
}
```

- 어셈블리어
  - 인간이 이해할 수 있는 문자열
  - 기계어와 1:1 대응

# Assembly 언어의 구성

- 일반적인 어셈블리어 구성
  - 라벨 (Label)
  - 연상기호 (Mnemonic)
  - 연산수 (Operand)

- 일반적인 구조
  - movq $10, %rax
  - rax 레지스터에 10 저장

# Assembly 언어의 구성

- 라벨의 사용
  - 주소만 표현
    - movq $L1 ,%rax: rax에 L1의 주소를 대입
  - 주소에 위치한 데이터를 표현
    - movq L1, %rax: rax에 L1의 주소에 저장된 값을 대입
    - movq %rax, L1 : L1의 주소에 위치한 값에 rax를 대입

# Assembly & C/C++

- 소스 코드 이용
  - Assembly *.s
  - gcc 어셈블리 컴파일 지원
  - gcc c_src.c asm_src.s
- 오브젝트 이용
  - gcc로 각 언어의 *.o 파일을 생성하여 함께 컴파일
  - gcc c_obj.o asm_obj.o
- Inline Assembly 사용
  - C 소스 코드 내부에 직접 어셈블리 코드를 삽입

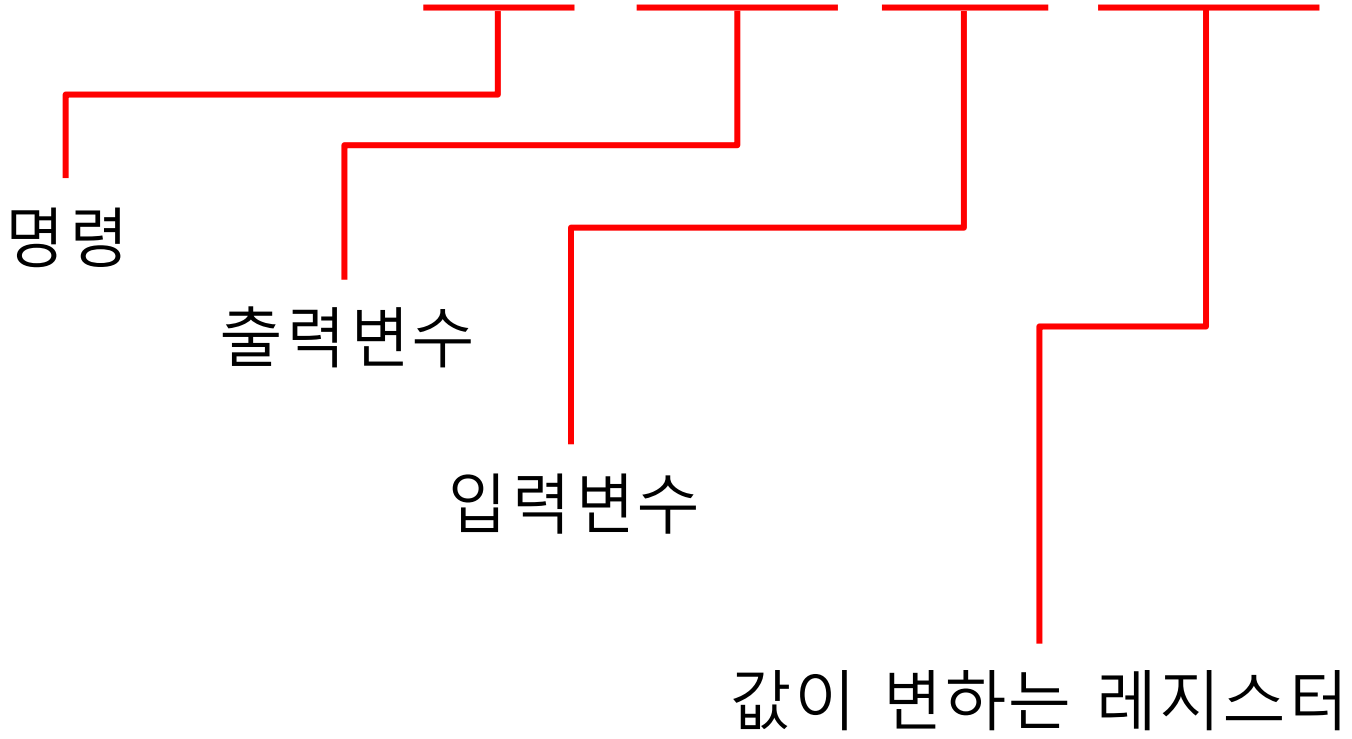
# Inline Assembly

- 사용 목적
  - C/C++ 코드로는 불가능한 작업을 수행하기 위해
    - 프로세서의 특정 명령어
  - 빠른 속도가 필요할 경우
  - 레지스터를 직접 제어하기 위해
  - 장치주소 등의 접근을 위한 커널 코드

# Inline Assembly

- 기본 문법

```
__asm__ __volatile__ (asms : output : input : clobber);
```



# Inline Assembly

- __asm__
  - 다음 코드가 Inline Assembly임을 나타냄
  - asm도 가능, ANSI에 __asm__으로 정의
- __volatile__
  - 컴파일러에 의한 최적화나 위치변화를 회피
  - 버그 발생 가능

# Inline Assembly

- 명령 (asms)
  - 따옴표로 둘러쌓인 어셈블리어 코드 삽입
  - 코드 내에서 %x와 같은 형태로 input, output 파라미터 사용
- 출력변수 (output)
  - “constraint” (variable) 형태의 리스트
- 입력변수 (input)
  - output과 같음

# Inline Assembly

- 값이 변하는 레지스터 (clobber)
  - 어셈블리 코드에서 컴파일러가 모르는 사이에 변할 수 있는 레지스터의 목록
  - 각 항목은 쌍따옴표(“) 안에 위치하며 여러 항목은 쉼표(,)로 구분
  - 메모리에 위치한 변수 값을 수정하는 경우 “memory” 라고 기술

# Inline Assembly

- 변형 문법
  - output, input, clobber는 생략 가능

```
__asm__ __volatile__ (asms : output);
__asm__ __volatile__ (asms : output : clobber);
```

  - ₩n₩t
    - 어셈블리어로 확인할 때 줄을 맞추는 것이 디버깅에 좋음

```
__asm__ __volatile__ (
        "movq %1 %0₩n₩t"
        "addq %2 %0"
        : "=r" (out)
        : "r" (x) "r" (y));
```

# Inline Assembly

- Constraints

| | | |
|---|---|---|
| 'm' | | 아키텍쳐가 지원하는 모든 종류의 메모리 주소를 사용하는 오퍼랜드 |
| 'o' | | 오프셋화가 가능한 주소를 사용하는 메모리 오퍼랜드 |
| 'V' | | 오프셋화가 불가능한 주소를 사용하는 메모리 오퍼랜드 |
| '<' | | 자동 감소(미리 감소하거나 나중에 감소) 주소용 메모리 오퍼랜드 |
| '>' | | 자동 증가(미리 증가하거나 나중에 증가) 주소용 메모리 오퍼랜드 |
| 'r' | | 일반 레지스터 사용 오퍼랜드 |
| 'd', 'a', 'f', ... | | 시스템에 따른 레지스터를 나타내는 오퍼랜드, 각각 68000/68020에서 데이터, 어드레스, 플로팅 포인터 레지스터를 나타냄 |
| 'i' | immediate | 정수 값을 나타내는 오퍼랜드, 심볼로 이루어진 상수도 여기에 해당 |
| 'n' | immediate | 정수 값을 나타내는 오퍼랜드, 많은 시스템이 한 워드 이하의 오퍼랜드용 수를 지원하지 않으므로 'i'보다 'n'을 사용하는 것이 바람직 |
| 'I', 'J' 'K', ... 'P' | | 시스템에 따라 특정 범위의 값을 나타내는 오퍼랜드, 68000에서는 'I'가 1에서 8까지의 값을 나타내며 이는 시프트 명령에서 허용되는 시프트 카운트의 범위임 |
| 'E' | immediate | 플로팅 오퍼랜드, 호스트와 같은 타겟 플로팅 포인트 포맷인 경우에만 사용 가능 |
| 'F' | immediate | 플로팅 오퍼랜드 |
| 'G', 'H' | | 특정 범위의 값을 나타내는 플로팅 오퍼랜드로 시스템에 따라 다름 |
| 's' | | 값이 명확히 정해지지 않은 immediate 정수를 나타내는 오퍼랜드, 'i' 대신 사용하여 좀 더 좋은 코드를 만들어 낼 수 있음 |
| 'g' | | 특수 레지스터를 제외한 일반 레지스터, 메모리 혹은 immediate 정수 중 아무 것이나 나타내는 오퍼랜드 |
| '0', '1', '2', ... '9' | | 같이 사용된 오퍼랜드의 번호를 나타냄 |
| 'p' | | 올바른 메모리 어드레스를 나타내는 오퍼랜드, "load address"와 "push address" 명령에 사용 |
| 'Q', 'B', 'S', ... 'U' | | 시스템에 따라 변하는 여러 다른 오퍼랜드를 나타냄 |

# Inline Assembly

- Modifier

'='
- 오퍼랜드가 쓰기 전용임을 나타냄
- 이전 값은 없어지고 새로운 값으로 교체
- 보통 output으로 사용

'+'
- 읽기, 쓰기 모두 가능함을 나타냄
- input/output 모두 사용 가능,
- 나머지 오퍼랜드는 input 전용으로 간주함

'&'
- "earlyclobber" 오퍼랜드를 나타냄
- input 오퍼랜드를 사용하는 명령이 끝나기 전에 변경됨을 의미
- input 오퍼랜드나 메모리 주소의 일부를 나타내는 레지스터엔 사용 불가
- gcc는 input 변수가 다 사용되고 나면 output에 사용된다고 가정하기 때문에 input에 사용된 변수가 output과 같아지며, 또 output이 input 보다 먼저 사용되는 경우가 발생할 수 있음
- 이런 경우를 막기 위해 output에 사용된 변수가 input으로 모두 사용되기 전에 변경될 수 있다고 알려줘야만 input과 output이 같아져 생기는 에러를 막을 수 있음

'%'
- % 뒤에 따라오는 오퍼랜드로 대체 가능함을 나타낸다. 직접 레지스터를 명시하고 사용할 때 %%eax 등과 같이 사용 가능

'#'
- # 이후의 쉼표가 나올 때까지 모든 문자를 constraints로 취급하지 않음

# Example 1

- 어셈블리 소스 이용
  - gcc ex1.s –o ex1

```
.globl main
main:
        movq    $10, %rax
        imul    %rax, %rax
        ret
```

**ex1.s**

# Example 2

- Inline Assembly 이용
  - gcc ex2.c –o ex2

```c
#include <stdio.h>

int main() {
        long a, sum;
        a = 1;

        __asm__ __volatile__(
                "movq %1, %0\n\t"
                "addq %2, %0\n\t"
                "addq %3, %0\n\t"
                "addq %4, %0\n\t"
                "addq %5, %0\n\t"
                : "=g"(sum)
                : "r"(a), "r"(a+1), "r"(a+2), "r"(a+3), "r"(a+4)
        );

        printf("a = %ld, sum = %ld\n", a, sum);

        return 0;
}
```

ex2.c

```
sub     $0x8,%rsp
mov     $0x5,%edi
mov     $0x4,%ecx
mov     $0x3,%esi
mov     $0x2,%edx
mov     $0x1,%eax
mov     %rax,%rcx
add     %rdx,%rcx
add     %rsi,%rcx
add     %rcx,%rcx
add     %rdi,%rcx
mov     $0x1,%dl
mov     $0x400634,%esi
mov     $0x1,%dil
mov     $0x0,%eax
callq   0x400460 <__printf_chk@plt>
mov     $0x0,%eax
add     $0x8,%rsp
retq
```

ex2.s

# Example 3

- 어셈블리 소스 코드 + C 코드 이용
  - gcc ex3.c myfunc.s –o ex3

```c
#include <stdio.h>

long myprint(long num, long exp, long ret) {
        printf("print func is called\n");
}

int main() {
        long num, exp, ret;

        num = 2;
        exp = 4;

        ret = myprint(num, exp);

        printf("%ld exp %ld = %ld\n", num, exp, ret);

        return 0;
}
```

ex3.c

```
 1 .globl  myfunc
 2 myfunc:
 3     movl    4(%esp), %eax
 4     movl    8(%esp), %ebx
 5     movl    $1, %ecx
 6     movl    $1, %edx
 7 L1: cmpl    %edx,   %ebx
 8     je L2
 9     imull   %eax,   %ecx
10     pushl   %ecx
11     pushl   %edx
12     pushl   %eax
13     call    myprint
14     popl    %eax
15     popl    %edx
16     popl    %ecx
17     incl    %edx
18     jmp L1
19 L2: imull   %eax,   %ecx
20     movl    %ecx,   %eax
21     ret
```

myfunc.s

# Objdump

- GNU 바이너리 유틸리티의 일부
- 바이너리 파일들의 정보를 보여주는 프로그램
- 역어셈블러로 사용 가능

# Objdump

- Option

| 옵션 \| 긴 옵션 | 설명 |
|---|---|
| -d \| --disassemble | 오브젝트 파일을 기계어로 역어셈블 |
| -D \| --disassemble-all | 모든 섹션을 대상으로 역어셈블 |
| --[no-]show-raw-insn | 코드와 바이트열 제거/출력 |
| --prefix-address | 코드의 주소를 심볼에서의 상대주소로 표시 |
| -j section \| --section=section | 특정 섹션 지정 |
| -l \| --line-numbers | 각각의 코드에 대응하는 소스코드의 행에 관한 정보 출력 |
| -S \| --source | 행 번호에 해당하는 소스코드가 그 위치에 삽입되어 출력 |

# Example 1

- Objdump
  - gcc -g –c ex4.c
  - objdump –d –S ex4.o

```c
#include <stdio.h>

int main() {
        unsigned int ret;

        __asm__ __volatile__("rdtsc" : "=A"(ret));

        printf("clock time: %d\n", ret);

        return 0;
}
```

**ex4.c**

```
int main() {
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   48 83 ec 10             sub    $0x10,%rsp
        unsigned int ret;

        __asm__ __volatile__("rdtsc" : "=A"(ret));
   8:   0f 31                   rdtsc
   a:   89 45 fc                mov    %eax,-0x4(%rbp)

        printf("clock time: %d\n", ret);
   d:   8b 45 fc                mov    -0x4(%rbp),%eax
  10:   89 c6                   mov    %eax,%esi
  12:   bf 00 00 00 00          mov    $0x0,%edi
  17:   b8 00 00 00 00          mov    $0x0,%eax
  1c:   e8 00 00 00 00          callq  21 <main+0x21>

        return 0;
  21:   b8 00 00 00 00          mov    $0x0,%eax
}
  26:   c9                      leaveq
  27:   c3                      retq
```

# Example 2

- Objdump
  - gcc -g –c ex5.c
  - objdump –d –S ex5.o

```c
#include <stdio.h>

int max(int x, int y) {
        if (x > y) {
                return x;
        } else {
                return y;
        }
}

void main(void) {
        int x = 4;
        int y = 6;
        int ret = 0;
        ret = max(x, y);
}
```

**ex5.c**

```
int max(int x, int y) {
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   89 7d fc                mov    %edi,-0x4(%rbp)
   7:   89 75 f8                mov    %esi,-0x8(%rbp)
        if (x > y) {
   a:   8b 45 fc                mov    -0x4(%rbp),%eax
   d:   3b 45 f8                cmp    -0x8(%rbp),%eax
  10:   7e 05                   jle    17 <max+0x17>
                return x;
  12:   8b 45 fc                mov    -0x4(%rbp),%eax
  15:   eb 03                   jmp    1a <max+0x1a>
        } else {
                return y;
  17:   8b 45 f8                mov    -0x8(%rbp),%eax
        }
}
  1a:   5d                      pop    %rbp
  1b:   c3                      retq

000000000000001c <main>:

void main(void) {
  1c:   55                      push   %rbp
  1d:   48 89 e5                mov    %rsp,%rbp
  20:   48 83 ec 10             sub    $0x10,%rsp
        int x = 4;
  24:   c7 45 f4 04 00 00 00    movl   $0x4,-0xc(%rbp)
        int y = 6;
  2b:   c7 45 f8 06 00 00 00    movl   $0x6,-0x8(%rbp)
        int ret = 0;
  32:   c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
        ret = max(x, y);
  39:   8b 55 f8                mov    -0x8(%rbp),%edx
  3c:   8b 45 f4                mov    -0xc(%rbp),%eax
  3f:   89 d6                   mov    %edx,%esi
  41:   89 c7                   mov    %eax,%edi
  43:   e8 00 00 00 00          callq  48 <main+0x2c>
  48:   89 45 fc                mov    %eax,-0x4(%rbp)
}
  4b:   c9                      leaveq
  4c:   c3                      retq
```