어셈블리 실습 1

Tags

어셈블리 프로그래밍 실습

- 이 장에서는 **x86-64 어셈블리 프로그래밍**의 기초를 다룬다.
- 어셈블리 실습의 목표
  - **C++**에서 호출 가능한 간단한 **x86-64 어셈블리 함수 코드**를 **작성**하고, 어셈블리 소스 코드 파일의 **구문**과 **의미**를 이해해보자
  - x86-64 어셈블리 프로그래밍 체험

기본 개념

어셈블러의 역할

- **어셈블러**는 **어셈블리 언어로 작성된 소스 코드**를 **기계어 코드**로 변환하는 역할을 한다.
  - 입력은 소스 코드, 출력은 object **모듈**이며, 이 모듈은 **링커**에 의해 **실행 파일**로 결합
  - **어셈블리 언어**는 CPU 명령어 집합의 기본 구성 요소로, 직접적으로 하드웨어 **레지스터**와 **메모리 관리**를 지원한다.
- **C나 C++과 달리**, **어셈블러**에는 **ISO 표준**이 없다.
  - **프로세서 아키텍처**에 따라 어셈블러 간 **구문적, 의미적 유사성**이 존재하지만, 어셈블러는 특정 프로세서에 의존적이며 **이식성**이 없다.

실습 대상 - Integer Arithmetic

- **32비트 정수 덧셈과 뺄셈**을 시연하는 간단한 프로그램을 예로 든다.
- 이어서 **비트 논리 연산** 및 **시프트 연산**을 다룬다.
- **64비트 정수 덧셈과 뺄셈** 예제를 통해 더 큰 정수에 대한 연산을 설명한다.
- 마지막으로 **정수 곱셈 및 나눗셈**에 대한 예제를 다룬다.

우리가 사용할 프로그램: NASM

NASM 소개

- `NASM (Netwide Assembler)`은 **x86-64 어셈블리어**를 작성하고 컴파일할 수 있는 **어셈블러**이다.
- NASM`은` `GNU/Linux` `와` `Windows`에서 사용할 수 있으며, 간단하고 효율적인 어셈블리 코드 작성을 지원한다.
- NASM은 **어셈블리 언어** 코드를 **기계어**로 변환하여 실행 파일을 생성한다.

NASM 특징

- **자유로운 구문 형식**: NASM은 구문 형식이 간결하고 엄격하지 않다.
- **모듈화 지원**: `include` 지시어로 다른 파일을 포함하여 코드의 재사용성을 높일 수 있다.
- **RIP-relative 메모리 주소 지정**: 64비트 모드에서 **메모리 참조** 시 상대 주소를 사용할 수 있다.
- **호환성**: 다양한 운영체제와 플랫폼에서 호환된다.
-

학생들이 반드시 주시해야 하는 점

어셈블리 언어의 문법은 다음 두가지 주요 스타일이 존재함

- **인텔 문법 (Intel Syntax)**:
  - `목적지(destination)`가 **왼쪽**에, `소스(source)`가 **오른쪽**에 옴.
  - 예시:

    ```
    mov rax, 1  ; rax(목적지) = 1(소스)
    ```

  - 이 문법은 인텔 CPU에서 주로 사용되는 문법으로, **NASM, MASM, TASM** 등이 따름
- **AT&T 문법 (AT&T Syntax)**:
  - `소스(source)`가 **왼쪽**에, `목적지(destination)`가 **오른쪽**에 옵니다.
  - 예시:

```
movl $1, %rax  ; 1(소스) → rax(목적지)
```

- 이 문법은 **리눅스의 GNU 어셈블러(GAS)에서 사용되는 표준 문법**
  - 주로 AT&T 문법은 레지스터 앞에 `%`, 즉시 값(리터럴) 앞에 `$`가 붙음
- 인텔 문법으로 변경하는 방법:
  - GCC에서 어셈블리 출력을 **인텔 문법**으로 보고 싶다면, 컴파일 시 `-masm=intel` 옵션을 사용

```
gcc -S -masm=intel yourfile.c
```

💡 NASM 어셈블러는 **인텔 문법**을 따름을 알아주시길 바랍니다. **목적지가 왼쪽**에 위치

실행 예시 1번

어셈블리 코드 (`ex01_01.asm`):

```
section .data
    msg db 'Hello, NASM!', 0

section .text
    global _start

_start:
    mov rax, 1          ; 시스템 호출 번호 1 = write
    mov rdi, 1          ; 파일 디스크립터 1 = stdout
    mov rsi, msg        ; 출력할 메시지 주소
    mov rdx, 13         ; 메시지 길이
    syscall             ; 시스템 호출 실행

    mov rax, 60         ; 시스템 호출 번호 60 = exit
```

```
    xor rdi, rdi        ; 종료 코드 0
    syscall             ; 시스템 호출 실행
```

실습 명령어:

1. 어셈블리 파일을 컴파일:

```
nasm -f elf64 ex01_01.asm -o ex01_01.o
```

2. 오브젝트 파일을 링킹:

```
ld ex01_01.o -o ex01_01
```

3. 실행:

```
./ex01_01
```

이렇게 하면 `"Hello, NASM!"` 메시지를 출력하는 프로그램을 실행할 수 있음

실행 예시 코드 설명

1. `section .data`:

- **프로그램의 데이터 섹션**을 정의 - 정적 데이터 저장용 메모리 공간
- `msg db 'Hello, NASM!', 0:` **msg 라는 레이블**을 가진 변수를 선언하고, "Hello, NASM!" 문자열과 NULL 문자(0)를 저장 → **NULL 문자는 문자열의 끝을 나타내는 데 사용**.
    - `msg` : 이것은 레이블(label)입니다. 레이블은 메모리 상의 특정 위치를 나타내는 이름입니다. 이 경우 `msg` 는 문자열 "Hello, NASM!"이 저장된 메모리 위치를 가리킵니다.
    - `db` : NASM 에서 사용하는 지시자: "define byte"의 약자로, 1바이트 크기의 데이터를 정의하는 지시자 → 여기서는 문자열의 각 문자를 1바이트씩 저장함을 의미
    - `'Hello, NASM!'` : 저장할 문자열 - 각 문자는 ASCII 코드로 변환되어 메모리에 저장
    - `0` : NULL 문자를 의미함.

2. `section .text`:

- **코드 섹션을 정의 -** 프로그램의 실행 코드가 저장되는 영역

- `global _start`: `_start` 레이블을 전역 심볼로 선언
  - 링커가 프로그램의 시작점을 찾을 수 있도록 함.

3. `_start:`**:**

- 프로그램의 시작점을 나타내는 레이블입니다.
- 우선, 시스템 콜 설명부터

아래는 리눅스에서 system call calling convention을 이야기함

- 시스템 호출이란? **(수업 시간에 한번 언급하겠음)**
  - **운영체제** 커널이 제공하는 서비스에 접근하기 위한 인터페이스
  - 사용자 프로그램이 **커널 기능**을 사용하는 방법
- System call calling convention?
  - 시스템 호출을 수행하기 위한 **약속된 규칙**
  - **목적:** 운영체제와 사용자 프로그램 간의 **효율적인 통신**
  - **주요 내용:**
    - 시스템 호출 번호를 특정 레지스터에 저장
    - 관련 인자들을 정해진 레지스터에 저장
    - `syscall` 명령어 실행
- 다시 위의 예시로 돌아가면...

4. 시스템 호출 `write`**:**

- `mov rax, 1`: 시스템 호출 번호 1 (write)을 `rax` 레지스터에 저장.
- `mov rdi, 1`: 파일 디스크립터 1 (표준 출력, stdout)을 `rdi` 레지스터에 저장.
- `mov rsi, msg`: 출력할 메시지 `msg` 의 주소를 `rsi` 레지스터에 저장.
- `mov rdx, 13`: 메시지의 길이 (13 바이트)를 `rdx` 레지스터에 저장.
- `syscall`: 시스템 호출을 실행하여 운영체제에 write 함수 요청.

5. 시스템 호출 `exit`**:**

- `mov rax, 60`: 시스템 호출 번호 60 (exit)을 `rax` 레지스터에 저장.
- `xor rdi, rdi`: `rdi` 레지스터를 0으로 설정. 종료 코드 0은 프로그램이 정상적으로 종료되었음을 나타냄.

    - **XOR 연산**은 두 **비트가 서로 다를 때 1을, 같을 때 0을 반환**
        - 따라서 같은 값을 XOR 연산하면 항상 0으로 설정함
    - 굳이 `mov rdi, 0` 를 안 쓴 이유는?
        - **크기:** `xor rdi, rdi` 명령어는 `mov rdi, 0` 보다 기계어 코드 크기가 작음.
        - **속도:** 일반적으로 `xor` 명령어가 `mov` 명령어보다 실행 속도가 빠름.
- `syscall`: 시스템 호출을 실행하여 운영체제에 exit 함수 요청.

C++ 코드와 어셈블리 혼합 실행 예시 2번

1. C++ 코드 (`ex01_main.cpp`)

```
// ex01_main.cpp
#include <iostream>#include "ex01.h"void DisplayResults(int
a, int b, int c, int d, int r1, int r2)
{
    constexpr char nl = '\n';
    std::cout << "----- Results for ex01 -----\n";
    std::cout << "a = " << a << nl;
    std::cout << "b = " << b << nl;
    std::cout << "c = " << c << nl;
    std::cout << "d = " << d << nl;
    std::cout << "r1 = " << r1 << nl;
    std::cout << "r2 = " << r2 << nl;
    std::cout << nl;
}

extern "C" int AddSubI32_a(int a, int b, int c, int d); //
(a + b) - (c + d) + 7

int main()
{
    int a = 10;
    int b = 40;
    int c = 9;
    int d = 6;
    int r1 = (a + b) - (c + d) + 7;
```

```
    int r2 = AddSubI32_a(a, b, c, d);

    DisplayResults(a, b, c, d, r1, r2);
    return 0;
}
```

2. 어셈블리 코드 (`ex01_addsub.asm`)

```
; ex01_addsub.asm
section .text
    global AddSubI32_a

AddSubI32_a:
    ; int AddSubI32_a(int a, int b, int c, int d)
    ; Calculate (a + b) - (c + d) + 7
    ; rdi = a, rsi = b, rdx = c, rcx = d
    mov eax, edi    ; eax = a
    add eax, esi    ; eax = a + b
    sub eax, edx    ; eax = a + b - c
    sub eax, ecx    ; eax = a + b - c - d
    add eax, 7      ; eax = a + b - c - d + 7
    ret
```

```
// ex01.h
#pragma once

void DisplayResults(int a, int b, int c, int d, int r1, int
r2);
extern "C" int AddSubI32_a(int a, int b, int c, int d);
```

4. 컴파일 및 실행 방법

1. 먼저 어셈블리 코드를 컴파일

   ```
   nasm -f elf64 ex01_addsub.asm -o ex01_addsub.o
   ```

2. C++ 파일을 컴파일하고 어셈블리 오브젝트 파일과 링크

```
g++ -no-pie -o ex01_main ex01_addsub.o ex01_main.cpp
```

3. 프로그램을 실행

```
./ex01_main
```

실습 문제: 비트 연산을 사용하는 어셈블리 코드 작성

문제 설명:

- 두 정수 `a`와 `b`에 대해 비트 AND, OR, XOR, 그리고 NOT 연산을 수행하는 어셈블리 코드를 작성하시오
- 각 연산 결과를 C++ 코드에서 출력할 수 있도록 어셈블리어로 비트 연산을 처리하는 함수를 작성하고, C++에서 호출해 결과를 출

요구사항:

1. `a = 0x55` (01010101)와 `b = 0xAA` (10101010)로 설정
2. 비트 연산 결과:
   - `a AND b`
   - `a OR b`
   - `a XOR b`
   - `NOT a`
3. 각 연산의 결과는 C++ 함수에서 출력되도록 어셈블리 코드`(ex02_bitwise.asm)`를 작성하시오.

C++ 코드 (`ex02_main.cpp`):

```
cpp
Copy code
#include <iostream>// 어셈블리 함수 선언
extern "C" int BitwiseAnd(int a, int b);
extern "C" int BitwiseOr(int a, int b);
extern "C" int BitwiseXor(int a, int b);
extern "C" int BitwiseNot(int a);
```

```
int main()
{
    int a = 0x55;  // 01010101
    int b = 0xAA;  // 10101010

    // 비트 연산 결과 출력
    std::cout << "a AND b = " << std::hex << BitwiseAnd(a,
b) << std::endl;
    std::cout << "a OR b = " << std::hex << BitwiseOr(a, b)
<< std::endl;
    std::cout << "a XOR b = " << std::hex << BitwiseXor(a,
b) << std::endl;
    std::cout << "NOT a = " << std::hex << BitwiseNot(a) <<
std::endl;

    return 0;
}
```

제출 문서

> 💡 **서버 실행 결과 캡쳐 시 학번 보이게** 캡쳐해주세요. 아래 내용을 **하나의 pdf 파일**로 제출해주세요

1. `"Hello, NASM!"` 메시지를 출력 - **서버 실행 결과** 캡쳐
2. **(a + b) - (c + d) + 7** c++ 과 asm 혼합 실행 예시 - **서버 실행 결과** 캡쳐
3. 비트 오퍼레이션을 수행하는 **어셈블리 코드**와 **실행 결과 캡쳐**