어셈블리 튜토리얼 2

HandyPost는 한 도영(HDNua)이 작성하는 포스트 문서입니다.

1. 개요

NASM을 이용하여 어셈블리 언어의 개념과 활용을 학습하고, 이것이 컴파일러에 어떻게 적용되는지를 이해한다. 이 문서의 많은 부분이 'PC 어셈블리어'[1] 문서를 참조하여 작성되었다. 그대로 가져다쓴 부분도 많은데, 이는 저자 및 역자의 너른 양해를 바란다(물론 둘 다 내 문서에 쓸 거라는 얘기는 했다).

2. 프로젝트 준비

여기서는 Netwide Assembler, NASM을 이용하여 어셈블리 프로그래밍을 학습한다. 사실 우리가 만들 컴파일러는 NASM을 이용하여 작성되는 것이 아니라, 우리가 직접 어셈블리를 해석하는 실행기(Runner)를 만들고 실행기에 우리의 어셈블리 소스 파일을 넘길 것이다. 다만 우리가 실행기를 만들 때 어셈블리 언어가 어떤 언어인지를 정확히 알고 가야 하기 때문에, 이 문서에서는 어셈블리 언어가 적어도 어떻게 사용하는 언어인지 이해할 필요가 있다. 예제를 직접 실행하기 위해서는 QuickNASM이라는 프로그램을 이용해야 한다. QuickNASM의 설치 및 사용 방법에 대해서는 필자의 블로그에 있는 포스트[2]에 동영상이 있으니 이를 참조하라.

3. 용어 정의

그럼 어셈블리를 배우기 위해 필요한 용어 및 정의를 알아보자. 참고로 말하자면 정의 하나하나를 영단어처럼 머릿속에 집어넣으려하지는 말고, 일단 교양서적처럼 읽은 다음에 4절을 읽으면서 뜻이 궁금해지면 그때 다시 읽어보는 방법을 추천한다.

3.1) 프로그램(program)과 프로세스(process)

이전 문서에서 컴파일, 링크 및 빌드에 대한 개념을 정리했다. 추가로 말하자면 소스를 컴파일 할 때는 바로 기계어 코드로 번역하는 경우도 있고, 어셈블리 언어로 변환된 다음 이 어셈블리 소스 파일을 기계어로 변역하는 경우도 있다. 컴파일러가 소스 파일을 어셈블리 언어로 변환한다면, 생성한 어셈블리 파일을 기계어로 변환하는 도구를 **어셈블러(Assembler)**라고 한다. 이후에 빌더는 소스 코드를 받으면 컴파일과 링크 과정을 거쳐 실행 가능한 목적 파일을 내놓는다고 했다. 이때 빌더가 생성하는, 이 실행 가능한 목적 파일을 바로 **프로그램(program)**이라고 한다.

하지만 우리가 빌드를 완료했다고 프로그램이 바로 실행되지는 않는다. 프로그램은 그것이 어떻게 동작하는지에 대한 정보를 담고 있을 뿐이다. 프로그램이 실행 '가능'한 목적 파일인 이유가 여기에 있다. 프로그램을 실행하면 운영체제는 프로그램에 기록된 실행 정보를 기반으로 해당 프로그램에 대해 메모리를 확보하고, 곧이어 프로그램의 명령을 메모리에 올린다. 이때 운영체제가 메모리에 올린 프로그램의 실행 정보를 **프로세스(process)**라고 한다. 간단히 말해서, 프로세스란 실행 중인 프로그램이다.

이전 문서에서 프로그램을 위해 운영체제가 메모리를 확보하고, 이 메모리를 네 가지의 영역으로 구분할 수 있다고 했다. 그 때 설명한 내용을 다시 가져와서 이에 대해 간단하게 정리해보자.

- **코드 세그먼트(code segment)**: 실행할 프로그램의 기계어 명령이 올라오는 메모리 영역이다.
- **데이터 세그먼트(data segment)**: 정적 변수 및 전역 변수, 문자열 상수 등 프로그램 실행 시에 정의되고 프로그램이 종료될 때 해제되는 데이터를 저장하는 영역이다.
- **힙 세그먼트(heap segment)**: 동적 할당한 데이터를 저장하는 영역이다.
- **스택 세그먼트(stack segment)**: 지역 변수 등 임시적으로 사용되는 데이터를 저장하는 영역이다.

이전 문서를 기억하는 분이라면, 전에 영역이라고 썼던 부분이 세그먼트라는 단어로 바뀌었음을 알 수 있다. **세그먼트(segment)**란 프로그램의 메모리를 목적에 따라 구분하였을 때 해당 영역을 말한다.

---

1) http://www.drpaulcarter.com/pcasm/, 한국의 이재범 님께서 한국어로 번역하셨다.
2) http://hdnua.tistory.com/9

실제 어셈블리를 사용할 때는 프로그램의 데이터 세그먼트에 대한 정의와 코드 세그먼트에 대한 정의를 별도로 작성해주어야 한다. 이에 대해서는 다음 절에서 다루겠다.

3.2) 레지스터(register)

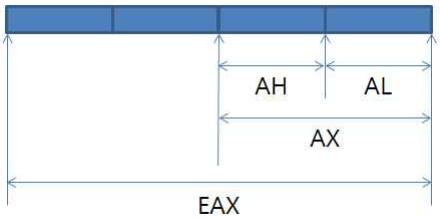
사전에서 레지스터의 정의를 찾으면 굉장히 복잡하게 이를 설명하는데, 그렇게 복잡하게 생각할 필요가 전혀 없다. 컴파일러를 구현하는 우리에게 레지스터란 그저 CPU만 쓸 수 있는 변수일 뿐이다. CIL에서는 사용자가 임의로 변수를 생성할 수 없도록 했는데, 이는 어셈블리와의 사양을 맞추기 위함이었다. 실제 어셈블리 프로그래밍에선, 아까 기본 변수라고 불렀던 변수들은 다음의 레지스터와 대응한다.

- 산술 레지스터: eax, ebx, ecx, edx
- 스택 포인터: ebp, esp
- 명령 포인터: eip
- 플래그: eflags

여기서 각각의 레지스터에 e가 붙은 건 extended를 의미하는데, 레지스터의 크기가 32bit라는 의미로 받아들이면 된다. 레지스터의 종류는 CIL이 제공하던 기본 변수보다 그 종류가 많은데, 실제 32bit 시스템의 레지스터의 목록을 용도에 따라 모두 나열해보면 아래와 같다.

- 범용 레지스터: 특별한 용도 없이 임의로 사용 가능한 레지스터.

> eax, ebx, ecx, edx의 네 가지 레지스터가 있다.
> 일반적으로 eax는 누산기, ecx는 카운터, edx는 데이터의 용도로 사용한다.
> ecx의 경우 loop와 같은 명령이 실제로 카운터로 사용하기 때문에 주의해야 한다.
> 범용 레지스터는 다음과 같이 분리될 수 있으며 각각에 접근할 수 있다.



- 포인터 레지스터: ebp, esp, eip

> CIL에서 사용되는 것과 같으므로 설명은 생략한다.

- 세그먼트 레지스터: 프로세스의 세그먼트를 표현하는 16bit 크기의 레지스터다.

> cs: code segment. 코드 세그먼트의 시작 주소를 가지고 있는 레지스터다.
> ds: data segment. 데이터 세그먼트의 시작 주소를 가지고 있는 레지스터다.
> ss: stack segment. 스택 세그먼트의 시작 주소를 가지고 있는 레지스터다.
> es: extra segment. 여분의 레지스터로 필요하면 사용한다. es, fs, gs의 세 가지가 있다.

- 인덱스 레지스터: esi, edi

> esi: extended source index.
> edi: extended destination index.

- 플래그 레지스터: eflags

> 연산의 결과에 대한 플래그를 저장한다.
> CMP, TEST와 같은 명령어로 값을 설정하고, JZ와 같은 점프 구문에서 자주 활용된다.

이와 같이 레지스터의 종류를 간단하게 알아볼 수 있었다. 이것이 실제로 어떻게 쓰이는지는 코드를 통해 확인할 것이다.

4. 문법

4.1) 뼈대 프로그램(stub program)

그럼 이제 본격적으로 NASM을 이용하여 어셈블리 프로그래밍을 해보자. 포스트에 적힌 방법대로 프로젝트를 생성하고 파일을 처음으로 만들면 다음과 같이 당황스러운 코드를 만나게 된다.

HelloWorld.asm

```
%include 'handy/handy.inc'

segment .data
sHelloWorld    db      'Hello, world!', 10, 0

segment .text
        global _main


_main:
        push    ebp
        mov     ebp,    esp

        push    sHelloWorld
        call    print_string
        add     esp,    4

        mov     eax,    0
        mov     esp,    ebp
        pop     ebp
        ret
```

실행 결과

```
Hello, world!
```

그런데 이게 정말 당황스러운 코드일까? 사실 우리는 이와 비슷한 코드를 이미 본 적이 있다. 이전 문서의 마지막 예제를 다시 가져오겠다.

ProcNaked.c

```
#include "CIL.h"
STRING sHelloWorld = "Hello, world!\n";

PROC(main)
// naked_proc 프로시저를 호출합니다.
CALL(naked_proc)
ENDP

// NAKED 프로시저를 정의합니다.
PROC_NAKED(naked_proc)
PUSH(bp) // 이전 스택 시작 주소를 푸시하여 보관합니다.
MOVL(bp, sp) // 스택 시작 주소를 현재 스택 포인터로 맞춥니다.
```

```
PUSH(sHelloWorld)
INVOKE(print_str)
ADD(sp, 4)

MOVL(sp, bp) // 현재 스택 포인터를 스택 시작 주소로 맞춥니다.
POP(bp) // 보관했던 이전 스택 시작 주소를 불러옵니다.
RET() // 복귀 지점으로 돌아갑니다.
ENDP_NAKED
```

naked_proc의 정의와 _main 레이블 이하의 코드를 유심히 쳐다보면, 두 코드가 거의 비슷하다는 사실을 알 수 있다! 주석을 제거하고 둘을 하나로 합쳐서 보자.

코드 비교 표

```
%include 'handy/handy.inc'

segment .data
sHelloWorld db 'Hello, world!', 10, 0

segment .text
        global _main

_main:
        push    ebp
        mov     ebp,    esp

        push    sHelloWorld
        call    print_string
        add     esp,    4

        mov     eax,    0
        mov     esp,    ebp
        pop     ebp
        ret
```

```
#include "CIL.h"


STRING sHelloWorld = "Hello, world!\n";



PROC_NAKED(naked_proc)
PUSH(bp)
MOVL(bp, sp)

PUSH(sHelloWorld)
INVOKE(print_str)
ADD(sp, 4)


MOVL(sp, bp)
POP(bp)
RET()
ENDP_NAKED
```

코드가 각각의 줄에 거의 대응하도록 변경하고 살펴보니, 몇몇 차이를 제외하면 두 코드가 아주 비슷함을 알 수 있다. 우리는 CIL을 배웠고, CIL은 어셈블리를 보다 쉽게 익힐 수 있도록 고안된 언어다. 즉, CIL을 이해하고 있는 우리는 어셈블리도 어렵지 않게 익힐 수 있다. 그럼 이 뼈대 프로그램을 먼저 분석하는 것으로 어셈블리 언어의 문법에 대한 감을 잡아보자.

- %include 'handy/handy.inc'

첫 줄은 handy 폴더에 있는 handy.inc 파일을 포함하는 전처리기 지시어(preprocessor directive)다. **지시어(directive)**란 소스 코드 중 실제 기계어로 변환 가능한 명령어가 아니라 소스 코드를 변환하는 프로그램에 전달하는 메시지를 말하며, 이 경우 handy.inc 파일을 포함하는 작업을 전처리기가 수행하기 때문에 %include는 전처리기에 대한 지시어가 된다.

- segment .data

3.1절에서 프로세스의 메모리를 크게 네 단계로 나눌 수 있다고 했다. segment는 소스 코드에 영역

별로 메모리를 정의하고 싶을 때 사용하는 어셈블러 지시어이고, segment 다음에 영역을 넘겨서 해당 영역에 메모리를 정의할 수 있도록 한다. 따라서 이는 이 지시어 다음에 나오는 모든 소스는 데이터 세그먼트를 정의하는 데 사용됨을 나타낸다.

- `sHelloWorld db 'Hello, world!', 10, 0`

문자열을 정의한다. db는 byte 형식의 데이터를 의미한다(후에 자세히 다룬다). 문자열 뒤에 정수 10이 들어가 있는 이유가 궁금할 텐데, 바로 10은 개행 문자의 ASCII 코드 값이기 때문이다. 이 문장은 파고들면 설명할 것이 아주 많이 나오지만, 지금 설명하면 독자가 혼란스럽게 느낄 수 있는 만큼 후에 자세히 다루겠다. 일단은 sHelloWorld는 byte의 배열로 정의된 레이블(label)이고, 개행 문자의 ASCII 코드 값이 10이기 때문에 개행 문자를 표시하기 위해 10을 넣었다는 점, 널 문자(₩0)로 문자열을 끝내기 위해 0을 마지막에 넣었다는 점만 기억하고 있으면 된다.

- `segment .text`

segment에 대해서는 설명했다. '.text'는 해당 지시어 다음에 등장하는 모든 소스가 코드 세그먼트에 대한 것이라고 어셈블러에게 전달하는 어셈블러 지시어다.

- `global _main`

_main 레이블이 전역에 선언된 레이블임을 의미하는 어셈블러 지시어다. 기본적으로 어셈블리 언어의 레이블은 모두 내부 선언되어있다(이는 C 언어에서 함수의 원형을 선언하면 기본적으로 전역에 선언된다는 점과 대조되는 중요한 특성이다). 따라서 이 지시어가 없이 레이블을 정의만 한 상태라면 다른 파일에서 이 레이블에 접근하는 것이 불가능하다. 즉 global은 다른 파일에서 레이블에 접근할 수 있도록 만들어준다.

- `_main:`

_main 프로시저를 정의한다. 어셈블리에서는 프로시저의 정의와 레이블의 정의가 서로 같은데 이에 대해서는 후에 다루겠다.

이와 같이 두 코드는 내부 구조가 완전히 동일하다. 이 프로그램을 이용하여 자신이 원하는 다른 문장을 화면에 출력하는 코드를 작성할 수 있다면 뼈대 프로그램을 사용하는 방법은 완전히 이해한 것이나 다름없다.

4.2) 문법

이제 본격적으로 어셈블리 언어의 문법에 대해 알아보자.

4.2.1) 기본 구문

코드 영역에 작성된 어셈블리 소스 코드를 **명령어(instruction)**라고 한다. 기본 구문은 다음과 같다.

```
[label:] [mnemonic [operands]] [; comment]
```

어셈블리 코드는 모두, 이 단 하나의 규칙만으로 작성된다. 각각의 요소가 무엇인지 설명하기 전에 몇 가지 예를 통해 이를 문법에 대응시켜보자. 일단 꺾쇠괄호('[', ']')가 생략 가능한 기호라는 사실만 기억하고 있으면 된다. 다음은 방금 보였던 코드에 주석을 추가로 달고 불필요한 부분을 정리한 코드이다.

Instruction.asm

```
_main:
        push    ebp
        mov     ebp,    esp

        push    sHelloWorld     ; 명령어와 주석을 조합할 수 있습니다.
        call    print_string    ; print_string 프로시저를 호출합니다.
```

```
        mov     eax,    0       ; 빈 줄에도 주석을 달 수 있습니다.
                                ; 프로세스가 0 이외의 값을 반환하면
                                ; 정상적으로 종료되지 않은 것으로
                                ; 간주하기 때문에, 반환 값을 언제나
                                ; 0으로 맞춰주어야 합니다.

end1:   mov     esp,    ebp     ; 구문의 모든 요소를 적용한 명령입니다.
end2:   pop     ebp
end3:   ret                     ; 프로시저를 반환합니다.
end4:                           ; _main의 경우 프로그램이 종료됩니다.
```

각각에 대해 차근차근 대응시켜보겠다.

`_main:`

label은 레이블이라고 읽는다. 명령어의 주소를 획득하고 싶다면 레이블을 등록하여 가져올 수 있다. 레이블은 생략 가능한 요소이지만, 레이블만으로 구문을 완성할 수 있다.

이 구문은 문법의 'label(_main) :'에 대응한다.

`push    ebp`

push와 같은 요소를 **연상 기호(mnemonic)**라고 한다. 이는 기계어의 명령 코드에 일대일 대응하는 기호로, 실제 레지스터 등에 값을 대입하는 등의 명령을 CPU에 전달한다. 연상 기호는 니모닉이라고 읽는다. ebp와 같이 연상 기호의 인자로 넘어가는 요소를 **피연산자(operand)**라고 한다.

이 구문은 문법의 'mnemonic(push) operands(ebp)'에 대응한다.

`mov     ebp,    esp`

이 구문은 문법의 'mnemonic(mov) operands(ebp, esp)'에 대응한다.

`push    sHelloWorld    ; 명령어와 주석을 조합할 수 있습니다.`

이 구문은 문법의 'mnemonic(push) operands(sHelloWorld) ; comment'에 대응한다.

`call    print_string   ; print_string 프로시저를 호출합니다.`

이 구문 역시 문법의 'mnemonic(call) operands(print_string) ; comment'에 대응한다.

`mov     eax,    0      ; 빈 줄에도 주석을 달 수 있습니다.`

이 구문 마찬가지로, 문법의 'mnemonic(mov) operands(eax, 0) ; comment'에 대응한다.

```
; 프로세스가 0 이외의 값을 반환하면
; 정상적으로 종료되지 않은 것으로
; 간주하기 때문에, 반환 값을 언제나
; 0으로 맞춰주어야 합니다.
```

이 구문은 문법의 '; comment'에 대응한다. 추가하자면 이 주석의 내용은 QuickNASM 프로그램을 사용하는 사람들에게는 중요하므로, QuickNASM 사용자라면 기억하고 있어야 한다.

`end1:   mov     esp,    ebp     ; 구문의 모든 요소를 적용한 명령입니다.`

이 구문은 문법의 'label(end1) : mnemonic(mov) operands(esp, ebp) ;comment'에 대응한다.

`end2:   pop     ebp`

이 구문은 문법의 'label(end2) : mnemonic(pop) operands(ebp)'에 대응한다.

`end3:   ret                     ; 프로시저를 반환합니다.`

이 구문은 문법의 'label(end3) : mnemonic(ret) ; comment'에 대응한다.

`end4:                           ; _main의 경우 프로그램이 종료됩니다.`

이 구문은 문법의 'label(end4) : ; comment'에 대응한다.

이 예제를 통해 NASM의 문법을 대강이나마 짐작할 수 있을 것이다. 정리하면 다음과 같다.

[*label*:] [*mnemonic* [*operands*]] [; *comment*]
- [label:] > 명령어의 주소를 획득하고 싶다면 레이블을 등록하여 가져온다. - mnemonic > 기계어의 명령 코드(operation code, opcode)에 일대일 대응하는 연상 기호이다. - [operands] > 연상 기호의 인자로 넘어가는 피연산자를 말한다. - [; comment] > 주석을 제공하여 가독성을 높일 수 있다.

4.2.2) 피연산자

문법에서 피연산자를 간략하게 설명했는데, 피연산자에도 종류가 있다. 이를 알아보자.

- 레지스터(register)

> CPU의 레지스터에 직접 접근하는 피연산자다. ‘mov eax, 0’ 구문에서 0이라는 값을 eax 레지스터에 직접 접근하여 복사하는 것을 상상하면 된다. 기호로는 reg로 표기한다.

- 메모리(memory)

> 메모리에 저장된 데이터를 가리킨다. 메모리의 주소 값은 명령에 직접 사용하거나, 레지스터에 저장하여 사용할 수 있다. 언제나 세그먼트 최상단부터의 오프셋 값으로 나타낸다. 기호로는 mem으로 표기한다.

- 즉시 값(immediate value)

> 명령 자체에 있는 고정된 값을 말한다. 0과 1 같은 상수 리터럴을 생각하면 된다. 기호로는 imm으로 표기한다.

그리고 연상 기호를 설명할 때 이러한 기호를 쓴다. 다음 절에서 알아보자.

4.2.3) mov 명령

mov 명령을 다음과 같이 설명할 수 있다.

mov: copy value from source to destination
mov reg, reg reg, mem reg, imm mem, reg mem, imm

첫 줄에는 mov 명령에 대한 설명이 적혀있다. mov 명령을 포함한 NASM의 대부분의 연상 기호는 첫 번째 피연산자가 목적지(destination), 두 번째 피연산자가 근원지(source)가 된다.

다음 칸을 보면 mov 명령에 대해 가능한 피연산자의 목록을 확인할 수 있다.

- reg, reg: 근원지로 레지스터, 목적지로 레지스터가 옵니다.
- reg, mem: 근원지로 메모리, 목적지로 레지스터가 옵니다.
- reg, imm: 근원지로 즉시 값, 목적지로 레지스터가 옵니다.
- mem, reg: 근원지로 레지스터, 목적지로 메모리가 옵니다.
- mem, imm: 근원지로 즉시 값, 목적지로 메모리가 옵니다.

여기서 중요한 내용이 있다. NASM에서는 한 번에 두 군데 이상의 메모리를 참조할 수 없다. 무슨 말이냐면, mov와 같은 명령의 피연산자로 근원지도 메모리, 목적지도 메모리가 올 수는 없다는 말이다.

따라서 어떤 메모리에 저장된 값을 다른 메모리로 복사하기 위해서는 레지스터에 먼저 보관하는 과정을 거쳐야만 한다.

다음은 mov 명령에 대한 예제이다.

mov.asm

```
%include 'handy/handy.inc'

segment .text
        global _main

_main:
        push    ebp
        mov     ebp,    esp

        mov     esi,    10              ; mov r32, imm (3)
        call    print_int               ; 정수 출력; esi = 출력할 정수
        call    print_newline           ; 개행 문자를 콘솔에 출력합니다.

        mov     eax,    20              ; mov r32, imm (3)
        mov     esi,    eax             ; mov r32, r32 (1)
        call    print_int               ; esi의 값인 10을 출력합니다.
        call    print_newline

        mov     eax,    0
        mov     esp,    ebp
        pop     ebp
        ret
```

r32와 같은 기호를 써서 당황한 분이 있을 것이다. 실제론 간단한데, r32라고 하면 32bit 레지스터를 의미하는 것이다. 마찬가지로, m32라고 하면 32bit 메모리를 의미하는 것이다.

이 예제를 완전히 이해했다면 다음 절로 넘어가자. 메모리를 다루는 방법은 좀 더 나중에 배운다.

4.2.4) 지시어

앞서 지시어를, 소스 코드 중 실제 기계어로 변환 가능한 명령어가 아니라 소스 코드를 변환하는 프로그램에 전달하는 메시지라고 설명한 바 있다. 여기서는 자주 사용되는 지시어들에 대해 알아본다. 참고로 이 절의 내용은 거의 대부분이 “PC 어셈블리어” 문서에서 가져온 것이다.

4.2.4.1) equ 지시어

equ 지시어는 심볼을 정의할 때 사용한다. **심볼(symbol)**이란 어셈블리 프로그래밍을 할 때 사용되는 상수를 말하며, 다음과 같이 사용한다.

symbol equ *value*

한 번 정의된 심볼의 값은 절대로 재정의 될 수 없다.

4.2.4.2) % 지시어

% 지시어 또한 상수 매크로를 정의할 때 사용할 수 있다.

%define *identifier value*

이 구문은 identifier를 value로 정의한다. 매크로는 심볼보다 유연한데, 왜냐하면 매크로는 심볼과 달리 재정의 될 수 있고 단순한 수가 아니어도 되기 때문이다.

여기까지의 내용을 코드로 먼저 정리해보자.

constant.asm

```
%include 'handy/handy.inc'

; equ 지시어를 이용하여 상수를 정의합니다.
EQU_VALUE      equ            100
; % 지시어를 이용하여 상수 매크로를 정의합니다.
%define        MACRO_VALUE    200
; % 지시어를 이용하여 구문을 정의합니다. equ로는 할 수 없습니다.
%define        MOV_ESI_300    mov    esi,    300
; MOV_ESI_300_EQU     equ    mov    esi,    300

segment .text
        ; _main 프로시저가 전역에서 접근 가능한 프로시저임을
        ; global 지시어를 이용하여 어셈블러에 전달합니다.
        global _main

; _main 레이블 정의입니다.
_main:
        push    ebp
        mov     ebp,    esp

        ; equ 지시어로 정의한 상수 사용 예제
        mov     esi,    EQU_VALUE
        call    print_int
        call    print_newline
        ; % 지시어로 정의한 상수 매크로 사용 예제
        mov     esi,    MACRO_VALUE
        call    print_int
        call    print_newline
        ; % 지시어로 정의한 구문 사용 예제
        MOV_ESI_300
        call    print_int
        call    print_newline

        mov     eax,    0
        mov     esp,    ebp
        pop     ebp
        ret
```

4.2.4.3) 세그먼트 정의 지시어

작성하는 코드가 프로그램이 실행되어 프로세스 메모리에 올라갔을 때 어느 세그먼트에 위치하는지를

어셈블러에 전달하는 역할을 한다. segment 키워드로 시작하고 곧이어 메모리를 위치시킬 세그먼트를 넘긴다. 세그먼트는 다음과 같은 것들이 있다.

- .data
> 초기화된 데이터가 저장되는 데이터 세그먼트다.
- .bss
> 초기화되지 않은 데이터가 저장되는 데이터 세그먼트다.
- .text
> 코드가 저장되는 코드 세그먼트다.

여기에서 data와 bss에 대해 설명이 더 필요할 것 같다. 하지만 지금 당장은 data와 text만 사용하면 된다고 이해했으면 한다. 후에 이 문서에서 이에 대해 더 자세하게 설명할 것이다.

4.2.4.4) 데이터 지시어

데이터 지시어는 데이터 세그먼트에서 메모리상의 공간을 정의하는 데 사용된다. NASM에서 지원하는 데이터 지시어에 대해 알아보자. NASM에서는 메모리 공간을 정의하는 방법이 2가지 있다. 하나는 메모리 공간만을 정의하는 resx 계열 지시어고, 다른 하나는 메모리 공간과 초기 값을 지정하는 dx 계열 지시어다. x에는 다음과 같은 문자(letter)가 들어갈 수 있다.

문자(Letter)	dx	resx	단위(Unit)	another name
b	db	resb	1바이트	바이트(byte)
w	dw	resw	2바이트	워드(word)
d	dd	resd	4바이트	더블 워드(double word, dword)
q	dq	resq	8바이트	쿼드 워드(quard word, qword)
t	dt	rest	10바이트	텐 바이트(ten byte, tbyte)

혹시 첫 예제에서 문자열 sHelloWorld의 오른쪽에 db라는 단어가 있던 것을 기억하는가? 그것이 바로 여기서 설명한 데이터 지시어의 일종이다.

이와 같이 NASM의 지시어를 알아볼 수 있었다.

4.2.5) 데이터 세그먼트

4.2.5.1) 데이터 세그먼트의 레이블

데이터 세그먼트도 레이블(label)을 갖는다. 그런데 이 경우 레이블이라는 이름보다는 라벨이라고 읽는 편이 이해하기 수월할 수 있다. 왜냐하면 데이터 세그먼트의 레이블이 실제 메모리에 붙어서 해당 메모리를 가리키는 라벨의 역할을 하기 때문이다(사실 코드 세그먼트의 레이블도 코드의 주소 값에 대한 라벨로 보는 편이 옳다). 즉 라벨 그 자체는 가리키는 메모리의 주소 값을 나타낸다. 코드 세그먼트와 다른 점은, 데이터 세그먼트의 데이터에 라벨을 붙일 때는 콜론(:)을 사용하지 않는다는 점이다. 예를 들어 값이 1인 바이트에 lbl이라는 별명, 즉 라벨을 붙이려면 다음과 같이 한다.

```
lbl     db      1
```

이제 데이터 세그먼트에서 데이터를 정의하는 방법을 알아보자. 앞으로 'lbl 라벨이 붙은 데이터 세그먼트의 메모리'를 단순히 lbl이라고 하겠다.

4.2.5.2) 데이터를 정의하고 사용하기

데이터 정의문(data definition statement)은 메모리에 데이터를 정의할 때 사용한다. 구문은 다음과 같다.

```
[label] directive initializer[, initializer]...
```

이는 말로 설명하는 것보다 예제를 보는 것이 이해가 빠르다. 다음은 “PC 어셈블리어” 문서에 소개된 예제를 발췌하여 정리한 것이다. 바로 위의 절과 함께 설명하겠다.

dataseg.asm

```
%include 'handy/handy.inc'

; 데이터 세그먼트의 시작을 알리는 segment .data 지시어입니다.
segment .data
L1      db      0               ; l1의 바이트 값을 0으로 설정
L2      dw      1000            ; l2의 워드 값을 1000으로 설정
L3      db      110101b         ; l3의 바이트 값을 110101_2로 설정
L4      db      17o             ; l4의 바이트 값을 17_8로 설정
L5      dd      1A92h           ; l5의 더블워드 값을 1A92_16으로 설정
L6      resb    1               ; l6을 1바이트 메모리로 정의하고 초기화하지 않음
L7      db      "A"             ; l7의 바이트 값을 문자 A의 ASCII 값으로 설정
L8      db      0, 1, 2, 3              ; 4바이트를 정의
L9      db      "w", "o", "r", "d", 0 ; C 문자열 "word"를 정의
L10     db      'word', 0               ; l10과 같음
L11     times   100     db      0       ; 100개의 db 0을 나열한 것과 같다

...
```

NASM에선 큰따옴표와 작은따옴표는 서로 같은 것으로 취급된다. 데이터가 연속적으로 정의되면 그 데이터들은 메모리에 연속해서 존재하게 된다. 따라서 L2는 L1의 바로 다음 메모리에 위치하게 된다. 배열을 정의하려면 L11과 같이 times 지시어를 이용한다.

4.2.5.1절에서 라벨이 메모리의 주소 값이라고 말했다. C에서는 주소 값을 이용해 해당 주소가 가리키는 값을 * 연산자를 이용해 획득할 수 있었다. NASM에서는 * 대신 대괄호(‘[’, ‘]’)를 이용한다. 그 방법은 다음과 같다.

dataseg.asm

```
...

; 코드 세그먼트의 시작을 알리는 segment .text 지시어입니다.
segment .text
        global _main
_main:
        push    ebp
        mov     ebp,    esp

        mov     al,     [L1]    ; al에 L1에 위치한 데이터를 대입한다
        mov     eax,    L1      ; eax = L1에 위치한 바이트의 주소
        mov     [L1],   ah      ; L1에 위치한 바이트에 ah를 대입한다
```

```
        mov     eax,    [L6]    ; L6에 위치한 더블워드를 eax에 대입한다
        add     eax,    [L6]    ; eax = eax + L6에 위치한 더블워드
        add     [L6],   eax     ; L6에 위치한 더블워드 += eax
        mov     al,     [L6]    ; L6에 위치한 더블워드의 하위 비트를
                                ; al에 대입한다

        mov     eax,    0
        mov     esp,    ebp
        pop     ebp
        ret
```

NASM의 중요한 특징이 이 예제에서 드러난다. 바로 어셈블러가, 라벨이 어떠한 데이터를 가리키고 있는지 전혀 신경을 쓰지 않는다는 점이다. 이는 C 컴파일러가 컴파일 시에 자료형을 검사하는 것과는 대조적이다. 후에는 데이터의 주소 값을 레지스터에 저장하고 C의 포인터 연산을 하듯 코드를 작성하게 되는데 이때도 포인터가 정확하게 사용되는지를 어셈블러가 검사하지 않는다. 이 때문에 어셈블리 프로그래밍은 C언어를 이용한 프로그래밍보다도 오류가 잦아지게 된다.

4.2.5.3) data와 bss

다음과 같은 C 코드를 생각하자.

```
char *str_ptr = "Hello, world!";
char str_arr[] = "Hello, world!";
int main(void) {
    char *p = &str_arr[0]; // &str_ptr[0];
    p[0] = 'h'; // p가 가리키는 메모리의 첫 번째 바이트를 'h'로 변경합니다.
    return 0;
}
```

이 코드는 정상적으로 실행되는 코드다. p가 str_arr을 가리키고 있을 때는 첫 번째 문자는 잘 변경된다. 그렇다면 p가 str_ptr을 가리키도록 예제를 수정해보자. 이때는 컴파일 시에는 오류가 발생하지 않지만 실행 시에 오류가 발생한다. 무엇이 문제인가?

결론부터 말하자면, 데이터 세그먼트도 수정 가능한 데이터 세그먼트와 수정 불가능한 데이터 세그먼트가 별도로 존재한다. 기본적으로 C의 전역 변수는 수정 가능한 데이터 세그먼트에 들어간다. const와 같은 한정자를 걸어놓지 않는 한 우리가 마음대로 수정할 수 있으니 당연한 것이다. 따라서 str_ptr 포인터 변수와 str_arr 배열 변수 모두 수정 가능한 데이터 세그먼트에 자리하게 된다.

문제는 str_ptr가 가리키는 “Hello, world!" 문자열은 수정 불가능한 데이터 세그먼트에 자리한다는 점이다. str_arr의 경우는 위와 같이 초기화를 진행하면 str_arr 배열을 위한 별도의 공간을 수정 가능한 데이터 세그먼트에 생성하고 문자열을 복사하므로 당연히 수정이 가능하다. 하지만 str_ptr은? 이 포인터 변수는 수정 불가능한 메모리에 자리한 문자열의 주소를 가리키고 있으니, 위와 같이 값을 수정하려고 하면 오류가 발생하는 것이다.

바로 여기서 data와 bss의 차이를 확인할 수 있다. 수정 가능한 데이터가 들어가는 세그먼트는 bss 세그먼트다. data 세그먼트에는 수정 불가능한 데이터가 들어간다. 즉 이 경우 str_ptr, str_arr는 모두 bss 세그먼트게 저장되고, str_ptr이 가리키는 문자열 "Hello, world!"는 data 세그먼트에 들어간다. 따라서 우리가 컴파일러를 개발할 때는 전역 변수와 정적 변수는 모두 bss 세그먼트에 넣어야 한다.

이와 같이 데이터 세그먼트에 대해 알 수 있었다. 우리는 CIL을 이미 배웠으므로, 이 정도만 이해하면 나머지는 jscc를 개발하면서 찾아보면 된다.

5. 단원 마무리

지금까지 봐온 어셈블리 입문 서적들은 모두 처음부터 어셈블리 언어의 스펙을 가르치지 않고, 대신 정수의 연산과 이진수 및 컴퓨터 공학의 개론적인 내용을 가르쳤다. 그것이 물론 기본적인 내용이긴 하지만, 어셈블리 언어 책을 샀는데 시작부터 어셈블리 언어와 다른 내용이 나오는 것은 매번 혼란스럽기 그지없었다(실제 주소 모드니 80x86이니). 그런 점에서 이 튜토리얼은 보다 빠르게 어셈블리 언어에 적응할 수 있도록 도와준다는 점에서 훌륭하게 작성되었다고 스스로 평가해본다.

그 외에 불안한 점이라면, 맨 위에도 밝혔고 문서 중간에도 밝히지만 이 문서는 사실상 기존에 존재하던 문서를 짜깁기하여 만든 문서라는 점이다(이전 문서까지는 그럴 필요가 없어서 하지 않았다). 혹 원저작자들이 불편하게 느껴 글을 내려달라는 요청이 들어오면 나는 지체 없이 문서를 내릴 것이고, 그러면 나는 이 강의자료 대신 그 분들의 자료를 보고 학습하라고밖에 할 수 없다. 혹 이 글을 보게 된다면 너른 마음으로 양해를 부탁드린다(메일, 댓글은 남겼는데 안 된다고 안 하셔서 씁니다).

어셈블리 튜토리얼 문서가 지나치게 짧다고 느끼실지 모르겠다. 사실 맞다. 어셈블리는 고작 13페이지만에 그 내용을 모두 설명할 수 있는 언어가 아니다. 하지만 문법만 따지고 본다면, 어셈블리의 문법은 아주 쉽고 간단해서 이 정도로도 설명을 충분히 마칠 수 있다. 원래는 프로그램 흐름 제어와 같은 부분을 더 넣고 싶었는데, 이는 좀 더 고민해본 다음 이 문서를 수정하거나 NASM 요리책(cookbook) 같은 것을 만들어서 레퍼런스를 참조하도록 하는 것이 좋을지, 아니면 스스로 검색해볼 수 있도록 하는 것이 옳은지 고민하고 있어서 일단 이 정도로 글을 마무리하려 한다.

다음에 배울 내용은 HTML, CSS, JavaScript를 이용한 웹 에디터 HandyHtmlMaker를 제작하는 것인데, 좋은 웹 에디터가 이미 많음에도 불구하고 역시 신뢰가 안 가는 이런 프로그램을 만들어서 쓰려는 것 역시 이유가 있다.

1. 필자가 이거보다 편한 웹 에디터를 못 찾았다. 아무렴 내가 편하라고 만든 건데.
2. HandyHtmlMaker를 만들 때 사용하는 기술 중 일부를 JSCC에 적용한다.
3. 디버깅이 어려운 환경에서 코딩할 때의 경험은 후에 디버깅 능력에 도움이 될 거라고 생각하니까.

이때 HandyHtmlMaker, JSCC 모두 파일을 생성하기 때문에, 악의 축이라고 불리는 ActiveX를 이용하여 파일에 접근하고 프로그래밍 할 것이다. 즉 Internet Explorer 9 또는 그 이상의 버전을 이용하여 프로그래밍 한다. Windows 이외의 환경에서 불만을 가질 수 있는데, 사실 한국에서 Windows 이외의 운영체제를 사용하는 사람이라면 어느 정도 컴퓨터에 대한 지식이 있지 않을까 싶은 마음에, 꼭 IE가 아니더라도 알아서 방법을 찾지 않을까 기대하고 있다(글을 쓰는 시점에서 새롭게 든 생각인데, 조만간 IE 지원이 종료되고, Chrome 브라우저에서 FileWriter API를 지원한다는 사실을 알게 되면서 이 생각은 바뀌고 있다).

참고로 말하자면 앞으로 진행하는 프로젝트는 아주 고통스러운데, 익혀야 하는 개념이 어려운 게 아니라 코드를 잘못 작성했을 때 디버깅이 정말로 쉽지 않기 때문이다. 서문에도 말했지만 JavaScript를 처음 배울 때 필자는 267번 줄에서 오류가 발생했다면 1번 줄부터 267번 줄까지를 모두 하나하나 세서 오류를 찾아냈다. 혹 좋은 도구가 있다면 그걸 쓰는 건 말리지 않겠다. 솔직히 말하면 이 프로그램에서 제공하는 기능이 지원되는 더 편한 도구라면 나라도 당장 그걸 쓸 것이다.