EDDI
Electronic Design
Development Institute

에디로봇아카데미

임베디드 마스터 Lv2 과정
[자료구조, 어셈블리]

제 1기

2021. 10. 09

박태인

어셈블리 분석 - 정의

EDDI
Electronic Design Development Institute

1) 어셈블리 언어란?

1.1. Assembly language

-어셈블리어란 컴퓨터(엄밀히는 CPU)가 이해할 수 있는 기계어 명령들을 사람들이 보다 쉽게 이해할 수 있도록 간단한 니모닉 언어(mnemonic symbol, 연상문자)로 나타낸 것. 이 어셈블리어들은 기계어와 1:1로 매칭된다.
(기계어 한 바이트당 하나의 니모닉 언어를 가지나 자주 쓰는 기계어의 경우 둘 이상의 바이트가 매칭되기도 한다.)

기계어(예)	어원	니모닉 언어(어셈블리어)
8B	MOVE	MOV
E9	JUMP	JMP
83 8B	COMPARE	CMP
E8	RETURN	RET

(※기계어는 예제이다. 실제와 같지 않다!)

기계어는 CPU의 아키텍처(CPU Architecture)마다 달라지므로 어셈블리어에 1:1로 매칭되는 기계어 또한 달라진다.
이렇게 달라지는 어셈블리어와 기계어에 대해서는 각 CPU제조사에서 제공하는 매뉴얼 북을 참조해야 한다.
여기서는 IA-32 아키텍처 기준으로 설명을 하겠다.

이러한 어셈블리어를 보고 기계어로 변환시키는 것을 "어셈블러(Assembler)", 거꾸로 기계어를 어셈블리어로 바꾸는 것을 "디스 어셈블러(Disassembler)"라 한다. (cf, (Assembly language)를 어셈블러(Assembler)라고 부르는 경우도 있다.)

어셈블리 분석 - 구성

EDDI
Electronic Design Development Institute

2) 어셈블리 언어 구성

-어셈블리어는 기본적으로 OP code + Operand로 구성되어있다.
OP code란 실행해야하는 기계어 명령어(instruction)를 표현한 것이고, Operand는 명령어의 대상이 되는 것을 말한다. 이러한 OP code와 Operand의 관계의 형태에는 다음과 같은 형식이 있다.

1	OP code	Operand 1	Operand2
2	OP code	Operand 1	.
3	OP code	.	.

1.가장 일반적인 형태의 모습으로 하나의 OP코드에 두 개의 오퍼랜드가 있는 모습이다.
OP Code 의 명령을 Operand2에서 Operand1로 적용하라는 의미이다..
2.하나의 Operand를 가지는 모습이다. OP code 의 명령을 Operand가 수행하라는 의미이다.
3.OP code만을 가지는 경우이다. 단순히 명령만을 수행한다. 내부적으로 보았을 시에는
여러 가지 명령이 복합적으로 실행되는 경우가 많다.

어셈블리어의 형태에는 위의 3가지 형태밖에 존재하지 않는다고 봐도 과언이 아니다.

이러한 OP코드와 오퍼랜드의 관계는 영어의 구문형태를 가지고 있으며 이러한 방식으로 이해를 하면 쉽게 이해할 수 있다.
예를 들어 영어 문장 Give me money를 어셈블리어 관점에서 분석하면, Give는 OP코드, me는 operand1, money는 operand2가 되며, 'money(operand2)를 me(operand1)에게 give(OP code)하라' 라는 의미이다.
(실제 어셈블리어에서는 Give me, money로 오퍼랜드 사이에 콤마(,)를 넣어준다.)

어셈블리 분석 - 메모리 구조(1)

3) 메모리 구조

ㄴ 어셈블리어를 직접적으로 작성하거나 공부를 할시에는 고급언어로 작성할 때 보다 메모리와 CPU에 대해 조금 더 상세히 알 필요가 있다.

-메모리는 실제 프로그램이 실행되기 위해 저장되는 공간을 말하며 CPU는 오직 메모리에 올라와 있는 데이터만을 해석할 수 있다. 따라서 이러한 메모리를 이해한다는 것은 프로그램을 이해하는데 매우 중요하다.
기본적으로 운영체제는 프로그램이 실행됨과 동시에 프로그램별로 최대 4GB까지의 동적으로 할당해주며, 일반적으로 서로 다른 프로그램에서는 절대로 다른 프로그램의 메모리 영역에는 접근할 수 없다.
프로그램별로 할당받은 메모리는 다시 스택 영역, 힙 영역, 데이터 영역, 텍스트 영역으로 나누어 지는데 그 구조는 다음과 같다.

위의 그림은 물리적으로 연속된 공간을 나타내지만 실제로는 연속된 공간이 아니라 논리적으로 연속된 공간이다.
하지만 개념상으로는 위의 그림과 같이 이해하여도 충분하다.

어셈블리 분석 - 메모리 구조(2)

3) 메모리 구조

ㄴ 어셈블리어를 직접적으로 작성하거나 공부를 할시에는 고급언어로 작성할 때 보다 메모리와 CPU에 대해 조금 더 상세히 알 필요가 있다.

아래 그림은 어느 프로그램상의 메모리의 일부분을 묘사해낸 것이다.

주소	0	1	2	3	4	5	6	7
메모리	2A	45	B8	20	8F	CD	12	2E

메모리는 데이터가 저장된 부분에 주소라는 값을 지정하고 그 주소를 통해 메모리의 데이터를 구분해 낸다.
프로그램이 메모리의 할당을 요청을 하면 운영체제는 메모리의 빈공간을 체크해보고 빈 공간의 주소값을 프로그램에게 전달해 준다. 프로그램은 그 빈 공간에 다가 자신이 할당받은 공간만큼 데이터를 이용하는 것이다. 할당되는 빈 공간의 크기는 데이터의 크기에 따라 달라지는데 이러한 빈공간의 크기는 변수 선언을 통해 결정되어 진다.

각 데이터의 형태별 변수의 크기는 다음과 같다.

워드(word)	2 바이트
더블워드(double word)	4 바이트
쿼드워드(quad word)	8 바이트
패러그래프(paragraph)	16 바이트

만약 메모리주소 0으로부터 더블워드의 크기만큼 할당받는다고 하면 0~3까지의 주소가 가르키는 메모리 공간을 사용할 수 있게 되는 것이다.

어셈블리 분석 – 레지스터(1)

EDDI
Electronic Design Development Institute

4) CPU 레지스터

ㄴ CPU Register는 CPU가 자체적으로 사용하는 일종의 메모리 공간이라 보면 된다.
CPU는 메모리에 저장된 데이터나 저장된 데이터의 위치를 레지스터에 저장한 후, 이를 읽어들여 연산을 수행한다. CPU 별로 몇몇개의 레지스터가 있으며 이 레지스터들은 각기 하는 용도가 대게 정해져있다.

레지스터의 종류	역할
Accumulator	연산의 대상이 되는 데이터 및 연산후의 데이터 저장
Flag Register	연산 처리 후 CPU의 상태 저장
Program counter	다음에 실행할 명령어가 보관된 메모리의 어드레스 저장
Base Register	데이터용 메모리 영역에서 첫 번째 어드레스 저장
Index Register	베이스 레지스터를 기준으로 한 상대 어드레스 저장
General-purpose register	임의의 데이터 저장
Instruction Register	명령어 자체를 저장. 프로그래머가 프로그램에서 이 레지스터의 값을 읽고 쓰는 것이 아니라 CPU가 내부적으로 사용
Stack Register	스택 영역의 맨 앞의 어드레스를 저장.

(이 이외에도 여러 특수한 레지스터 종류들이 있으나 일반적으로 사용자가 그것까지는 알 필요가 없다.)

여기서 주로 봐야할 것은 General-purose register(범용레지스터)들이다.

범용레지스터는 일반적으로 다양한 용도로 사용되는 레지스터들이다. 어셈블리어를 만나게 되면 가장 자주 보이는 레지스터들의 이름이기도 하다. 실질적으로 프로그램이 구동되기 위한 여러 데이터들은 범용 레지스터들을 통해서 처리된다고 봐도 과언이 아니다.

어셈블리 분석 – 레지스터(2)

EDDI
Electronic Design Development Institute

4) CPU 레지스터

ㄴ CPU Register는 CPU가 자체적으로 사용하는 일종의 메모리 공간이라 보면 된다.
CPU는 메모리에 저장된 데이터나 저장된 데이터의 위치를 레지스터에 저장한 후, 이를 읽어들여 연산을 수행한다. CPU 별로 몇몇개의 레지스터가 있으며 이 레지스터들은 각기 하는 용도가 대게 정해져있다.

범용 레지스터에는 다음과 같은 종류가 있다.

레지스터의 종류	주 역할
EAX, EBX, EDX	일반적인 연산에 모두 이용이 가능하다.
ECX	카운팅이 필요한 곳에 가끔 사용된다.
ESI, EDI	메모리 복사 명령에서 복사 될 대상의 주소와 복사가 수행 될 목적지의 주소 포인터로서 주로 사용된다.
EBP	스택의 베이스 포인터로 주로 사용됨. 스택의 시작주소를 나타냄. 스택포인터와의 조합을 통해 스택프레임을 형성하는데 주로 사용된다.
ESP	스택의 스택포인터로 주로 사용됨. 스택의 가장 상단의 주소를 나타냄. 베이스포인터와의 조합을 통해 스택프레임을 형성하는데 주로 사용된다.

위와 같은 범용레지스터들은 서로 각기 다른 용도로도 사용될 수 있다. 위의 “주 역할”은 말 그대로 주 역할일 뿐이지 반드시 위와 같은 작업을 한다고 볼 수 없으며 실제로 EAX는 범용 레지스터임과 동시에 Accumulator레지스터라고 볼 수 도 있다.

어셈블리 분석 – 코드

EDDI
Electronic Design Development Institute

5) 기본 어셈블리 명령어

-아래의 내용은 디버깅을 실시하면 자주 발견할 수 있는 어셈블리어 코드들이다.

OP Code	Operand	기능	OP Code 어원
mov	a, b	b의 값을 a로 저장	move
add	a, b	b의 값을 a에다가 더함	add
sub	a, b	b의 값을 a에서 뺌	subtraction
push	a	a의 값을 스택에 저장	push
pop	a	스택에서 값을 꺼내 a에 저장	pop
call	a	함수 a를 호출	call
ret	(null)	함수를 호출한곳으로 리턴	return
cmp	a, b	a와 b를 비교. 비교 결과값은 flag에 저장	compare
jmp	a	a로 점프	jump
and, or, xor...	a, b	a와 b의 비트연산	.

이 이외에도 약간씩 다른모양의 OP Code들이 있을 수 도 있다. 예를 들어 ret의 경우에는 retn 으로 나오는 경우도 있으나 이것은 소스코드가 다른 것이 아니라 디버거의 해석에 따라 달라지는 경우이므로 큰 문제는 없다.

그리고 기본적인 OP Code에서 변형된 OP Code들이 있다.
대표적인에로 jmp관련 코드들이 있는데 몇가지 예를 들자면 jnz(jump not zero), je(jump equal) jge(jump grater eqaul)등이 있으며 이와같은 내용들은 flag register의 값들을 참조하며 jump 여부를 결정하게 되는 것이다.

또한 어셈블리어에서는 대부분의 데이터의 접근은 주소값을 참조하여 접근하게 된다. 이러한 주소값참조를 표현하는 방식은 "(참조할 데이터의 크기) ptr[주소]" 로 나타내어진다. 예를들면 "dword ptr[ebp-4]" 같은 값으로 나타내어진다.

어셈블리 분석 – C 예제 분석(1)

EDDI
Electronic Design Development Institute

```c
#include <stdio.h>

int mult(int n1, int n2)
{
        return n1 * n2;
}


int main(void)
{
        int num1 = 3, num2 = 2;
        int res;

        res = mult(num1, num2);
        printf("res = %d\n", res);

        return 0;
}

// gcc를 활용해서 컴파일을 할 때 -g 옵션을 주면 디버깅 메타 정보를 기록하게 된다.
// 추가적으로 -O0 옵션을 추가하면 어떠한 최적화 기법도 적용되지 않는다.
// -O2는 최대한의 보수성을 지키며 최적화를 수행하며
// -O3는 문제가 발생하던 말던 일단 최고의 효율을 뽑겠다라는 옵션이라 문제가 발생할 우려가 있다.
```

왼쪽 C 코드는
Num1 과 Num2 의 값에 따라
곱셈 계산 연산 결과가
출력되는 코드 이다.

아래 주석은 gdb 분석(어셈블리)시
컴파일 코드상에 옵션을 정하여
컴파일 할 수 있다는 설명이다.

Ex) gcc -g -O0 function.c -o function

어셈블리 분석 – C 예제 분석(2)

```
=> 0x0000555555555160 <+0>:     endbr64
   0x0000555555555164 <+4>:     push   %rbp
   0x0000555555555165 <+5>:     mov    %rsp,%rbp
   0x0000555555555168 <+8>:     sub    $0x10,%rsp
   0x000055555555516c <+12>:    movl   $0x3,-0xc(%rbp)
   0x0000555555555173 <+19>:    movl   $0x2,-0x8(%rbp)
   0x000055555555517a <+26>:    mov    -0x8(%rbp),%edx
   0x000055555555517d <+29>:    mov    -0xc(%rbp),%eax
   0x0000555555555180 <+32>:    mov    %edx,%esi
   0x0000555555555182 <+34>:    mov    %eax,%edi
   0x0000555555555184 <+36>:    callq  0x555555555149 <mult>
   0x0000555555555189 <+41>:    mov    %eax,-0x4(%rbp)
   0x000055555555518c <+44>:    mov    -0x4(%rbp),%eax
   0x000055555555518f <+47>:    mov    %eax,%esi
   0x0000555555555191 <+49>:    lea    0xe6c(%rip),%rdi        # 0x555555556004
   0x0000555555555198 <+56>:    mov    $0x0,%eax
   0x000055555555519d <+61>:    callq  0x555555555050 <printf@plt>
   0x00005555555551a2 <+66>:    mov    $0x0,%eax
   0x00005555555551a7 <+71>:    leaveq
   0x00005555555551a8 <+72>:    retq
```

Function.c의 컴파일 후

- Gdb funtion (gdb 모드 function 실행)
- B main (brake main)
- R (reset)
- Disas (dis assembly)

명령어 실행시의 어셈블리 분석 모습

메모리 구조 분석