



# C언어 소스코드 분석 (x86 Assembly)

---

홍정완

([HTTPS://GITHUB.COM/JUNG9928](https://github.com/JUNG9928))

# # 목차 #

---

1. C언어로 작성된 소스코드 분석 (x86 Assembly).
2. Intel Instruction Architecture의 특징

# # 목적 #

---

1. c언어 코드 분석을 통해 기존에 알고있던 MASM 지식 활용 및 정리.
2. x86 Assembly에 대한 설명을 상세하게 하려는 목적이 아닌 이미 알고 있다는 가정 하에, C언어 코드가 어떻게 x86 Assembly 언어로 변환되고 Memory에는 어떤 값이 저장되는지를 관찰하는 것이 주목적입니다.
3. 기존의 MIPS나 ARM 아키텍처를 사용하는 CORTEX-M 시리즈 등 RISC 방식의 인스트럭션과의 차이점도 복기하자는 목적으로 작성하였습니다.

# # 코드 분석 #

---

```
1  #include <stdio.h>
2
3  int main( )
4  {
5      int a, b, sum;
6      scanf("%d %d", &a, &b);
7
8      sum = a + b;
9
10     printf("%d", sum);
11
12     return 0;
13 }
```

- 왼쪽의 소스코드는 간단한 덧셈을 수행하는 소스코드이다.
- 코드를 line by line으로 분석해보자.

```

1  #include <stdio.h>
2
3  int main()
4  {
5      int a, b, sum;
6      a = 3;
7      b = 4;
8
9      sum = a + b;
10
11     printf("%d", sum);
12
13     return 0;
14 }

```

```
#include <stdio.h>
```

```
int main()
{
```

```

00D21810 push     ebp
00D21811 mov      ebp,esp
00D21813 sub      esp,0E4h
00D21819 push     ebx
00D2181A push     esi
00D2181B push     edi
00D2181C lea      edi,[ebp-0E4h]
00D21822 mov      ecx,39h
00D21827 mov      eax,0CCCCCCCCh
00D2182C rep stos  dword ptr es:[edi]
00D2182E mov      ecx,offset _D9812314_dfs_bfs 연습장\dfs_bfs 연습장\소스코드분석@c (0D2C003h)
00D21833 call     @__CheckForDebuggerJustMyCode@4 (0D21217h)

```

- main 함수의 entry point.
- main 함수의 STACK FRAME이 생성됨.

- main 함수의 stack frame이 생성됨. 현재 stack pointer의 값을 base pointer 값으로 초기화하여 해당 함수의 인자 값을 저장하고 접근하기 위한 기준점(위치)으로 사용하고, stack pointer(esp)는 이동시켜가며 지역변수를 stack frame에 저장.
- ebx, esi, edi, ecx, eax 범용 레지스터들은 main 함수 호출 시, 필요한 stack을 셋업하고 관련 레지스터들을 보존하는 작업을 위해 존재.
- main 뿐만 아니라 모든 함수들에 대한 assembly 코드 초입에 존재한다.

↓ main 함수의 stack frame 영역

```

0x00D21810 55 8b ec 81 ec e4 00 00 00 53 56 57 8d bd 1c ff ff ff b9 39 00 00
0x00D21826 00 b8 cc cc cc cc f3 ab b9 03 c0 d2 00 e8 df f9 ff ff c7 45 f8 03
0x00D2183C 00 00 00 c7 45 ec 04 00 00 00 8b 45 f8 03 45 ec 89 45 e0 8b 45 e0
0x00D21852 50 68 30 7b d2 00 e8 e9 f7 ff ff 83 c4 08 33 c0 5f 5e 5b 81 c4 e4

```

- visual studio에서 d0ebugging 기능을 수행하기 위해 제공하는 함수. 맵글링으로 @\_\_CheckForDebuggerJustMyCode로 사용됨.

레지스터

```

EAX = 008B6EF0 EBX = 005DB000 ECX = 00000001
EDX = 008BD718 ESI = 00D21339 EDI = 00D21339
EIP = 00D21810 ESP = 006FF9CC EBP = 006FF9E8
EFL = 00000206

```

```

1  #include <stdio.h>
2
3  int main()
4  {
5      int a, b, sum;
6      a = 3;
7      b = 4;
8
9      sum = a + b;
10
11     printf("%d", sum);
12
13     return 0;
14 }

```

- 단순 변수 선언문에는 break point가 걸리지 않는다.  
그 이유는, 컴파일러가 컴파일 시, 변수 선언의 경우 메모리를 할당하는 것이 아닌 임의의 메모리 공간을 선정해서 사용할 것이라고 예약한 풀이기 때문.

그렇기에, 위의 어셈블리 코드를 보면 변수 선언문 부분에 instruction이 생성되지 않은 것을 확인할 수 있으며 그렇기에 break point를 걸 수 없는 것이고 실행흐름이 바로 변수 초기화 부분으로 넘어간 것을 확인할 수 있다.

- 변수에 값을 할당하는 부분인 a = 3;에서는 그 이전 변수 선언문에서 컴파일러가 선정(예약)한 메모리 공간에 값을 저장하게 된다.

```

int a, b, sum;
a = 3;
00D21838 mov     dword ptr [a],3
b = 4;
00D2183F mov     dword ptr [b],4

```

**EIP** 는 다음에 실행할 instruction의 주소가 저장된 Register.  
현재 실행될 instruction인 “mov dword ptr [a], 3”의 주소를 가지고 있음.

mov는 메모리 or 레지스터에 값을 옮기기 위한 instruction이며, 지정된 오퍼랜드인 dword ptr [a]와 3 중에서 오른쪽 오퍼랜드 값(3)을 왼쪽 오퍼랜드(dword ptr [a])로 옮기는 역할을 수행.

dword ptr은 우리가 C언어에서 사용하는 int, double, long과 같이 메모리 사이즈를 지정하는 directive임. dword는 32비트 크기 지정.

레지스터

EAX = 0002C003	EBX = 005DB000	ECX = 00D2C003
EDX = 00000001	ESI = 00D21339	EDI = 006FF9C8
<b>EIP = 00D21838</b>	ESP = 006FF8D8	EBP = 006FF9C8
EFL = 00000246		

0x006FF9C0 = CCCCCCCC

```

1 #include <stdio.h>
2
3 int main()
4 {
5     int a, b, sum;
6     a = 3;
7     b = 4;
8
9     sum = a + b;
10
11     printf("%d", sum);
12
13     return 0;
14 }

```

- a와 b를 더한 결과 값을 변수 sum에 저장하는 과정

메모리 2

주소: 0x006FF9C0

0x006FF9C0 03 00 00 00

레지스터

EAX = 00D2C003 EBX = 005DB000 ECX = 00D2C003  
 EDX = 00000001 ESI = 00D21339 EDI = 006FF9C8  
 EIP = 00D21846 ESP = 006FF8D8 EBP = 006FF9C8  
 EFL = 00000246

0x006FF9C0 = 00000003

```

sum = a + b;
00D21846 mov     eax,dword ptr [a]
00D21849 add     eax,dword ptr [b]
00D2184C mov     dword ptr [sum],eax

```

메모리 2

주소: 0x006FF9B4

0x006FF9B4 04 00 00 00

레지스터

EAX = 00000003 EBX = 005DB000 ECX = 00D2C003  
 EDX = 00000001 ESI = 00D21339 EDI = 006FF9C8  
 EIP = 00D21849 ESP = 006FF8D8 EBP = 006FF9C8  
 EFL = 00000246

0x006FF9B4 = 00000004

```

sum = a + b;
00D21846 mov     eax,dword ptr [a]
00D21849 add     eax,dword ptr [b]
00D2184C mov     dword ptr [sum],eax

```

- add는 메모리 오퍼랜드 + 메모리 오퍼랜드 or  
 메모리 오퍼랜드 + 레지스터 or 레지스터 + 메모리 오퍼랜드  
 메모리 오퍼랜드 + 상수를 수행하는 어셈블리어이다.

이전에 초기화한 변수 a, b를 더하기 위해 누산기 역할을 수행하는 레지스터인 eax를 사용한다.

eax 레지스터는 모든 산술연산에서 사용되며, 가장 많이 사용되는 범용 레지스터이다.

메모리 2

주소: &sum

0x006FF9A8 07 00 00 00

레지스터

EAX = 00000007 EBX = 005DB000 ECX = 00D2C003  
 EDX = 00000001 ESI = 00D21339 EDI = 006FF9C8  
 EIP = 00D2184F ESP = 006FF8D8 EBP = 006FF9C8  
 EFL = 00000202

0x006FF9A8 = 00000007

```

sum = a + b;
00D21846 mov     eax,dword ptr [a]
00D21849 add     eax,dword ptr [b]
00D2184C mov     dword ptr [sum],eax

printf("%d", sum);
00D2184F mov     eax,dword ptr [sum]
00D21852 push    eax

```

```

1 #include <stdio.h>
2
3 int main()
4 {
5     int a, b, sum;
6     a = 3;
7     b = 4;
8
9     sum = a + b;
10
11     printf("%d", sum);
12
13     return 0;
14 }

```

- a와 b를 더한 결과 값을 변수 sum에 저장하는 과정

메모리 2	
주소:	0x006FF9C0
0x006FF9C0	03 00 00 00
레지스터	
EAX = 00D2C003	EBX = 005DB000 ECX = 00D2C003
EDX = 00000001	ESI = 00D21339 EDI = 006FF9C8
EIP = 00D21846	ESP = 006FF8D8 EBP = 006FF9C8
EFL = 00000246	
0x006FF9C0 = 00000003	
sum = a + b;	
00D21846	mov eax,dword ptr [a]
00D21849	add eax,dword ptr [b]
00D2184C	mov dword ptr [sum],eax

메모리 2	
주소:	0x006FF9B4
0x006FF9B4	04 00 00 00
레지스터	
EAX = 00000003	EBX = 005DB000 ECX = 00D2C003
EDX = 00000001	ESI = 00D21339 EDI = 006FF9C8
EIP = 00D21849	ESP = 006FF8D8 EBP = 006FF9C8
EFL = 00000246	
0x006FF9B4 = 00000004	
sum = a + b;	
00D21846	mov eax,dword ptr [a]
00D21849	add eax,dword ptr [b]
00D2184C	mov dword ptr [sum],eax

- add는 메모리 오퍼랜드 + 메모리 오퍼랜드 or  
메모리 오퍼랜드 + 레지스터 or 레지스터 + 메모리 오퍼랜드  
메모리 오퍼랜드 + 상수를 수행하는 어셈블리어이다.

이전에 초기화한 변수 a, b를 더하기 위해 누산기 역할을 수행하는 레지스터인 eax를 사용한다.  
eax 레지스터는 모든 산술연산에서 사용되며, 가장 많이 사용되는 범용 레지스터이다.

메모리 2	
주소:	&sum
0x006FF9A8	07 00 00 00
레지스터	
EAX = 00000007	EBX = 005DB000 ECX = 00D2C003
EDX = 00000001	ESI = 00D21339 EDI = 006FF9C8
EIP = 00D2184F	ESP = 006FF8D8 EBP = 006FF9C8
EFL = 00000202	
0x006FF9A8 = 00000007	
sum = a + b;	
00D21846	mov eax,dword ptr [a]
00D21849	add eax,dword ptr [b]
00D2184C	mov dword ptr [sum],eax
printf("%d", sum);	
00D2184F	mov eax,dword ptr [sum]
00D21852	push eax



```

1  #include <stdio.h>
2
3  int main()
4  {
5      int a, b, sum;
6      a = 3;
7      b = 4;
8
9      sum = a + b;
10
11     printf("%d", sum);
12
13     return 0;
14 }

```

- printf함수가 호출되면서 printf 함수  
고유의 stack frame이 생성된다.

```

1  #include <stdio.h>
2
3  int main()
4  {
5      int a, b, sum;
6      a = 3;
7      b = 4;
8
9      sum = a + b;
10
11     printf("%d", sum);
12
13     return 0;
14 }

```

```

printf("%d", sum);
0002184F mov     eax,dword ptr [sum]
00021852 push    eax
00021853 push    offset string "%d" (0027B30h)
00021858 call    _printf (0021046h)
0002185D add     esp,8

printf("%d", sum);
0002184F mov     eax,dword ptr [sum]
00021852 push    eax
00021853 push    offset string "%d" (0027B30h)
00021858 call    _printf (0021046h)
0002185D add     esp,8

printf("%d", sum);
0002184F mov     eax,dword ptr [sum]
00021852 push    eax
00021853 push    offset string "%d" (0027B30h)
00021858 call    _printf (0021046h)
0002185D add     esp,8

printf("%d", sum);
0002184F mov     eax,dword ptr [sum]
00021852 push    eax
00021853 push    offset string "%d" (0027B30h)
00021858 call    _printf (0021046h)
0002185D add     esp,8

return 0;
00021860 xor     eax,eax
}

```

호출 규칙      \_cdecl(/Gd)

- 1) printf의 stackframe에 인자 값인 sum을 저장하기 위해 eax 레지스터로 데이터를 이동.
- 2) stack frame에 sum 값이 저장된 eax 레지스터 값을 push
- 3) 출력 값을 저장할 공간을 stack에 push.
- 4) 실제 printf함수를 구성하는 \_printf 함수를 호출.
- 5) printf 동작을 마친 후, esp에 할당된 메모리 크기 값을 더하여 stack frame을 해제. 이것을 통해 cdecl 호출 규약을 사용한다는 것을 알 수 있음.  
  
만약 ret 8 이었다면 stdcall 호출 규약을 사용한 것.
- 6) main 함수의 stack frame이 소멸되면서 main 함수 종료.

xor 연산을 통해 return 값이 0인지 아닌지 판별.

```

int main()
{
00661920 push      ebp
00661921 mov       ebp,esp
00661923 sub       esp,0E4h
00661929 push      ebx
0066192A push      esi
0066192B push      edi    경과 시간 1ms 이하
0066192C lea       edi,[ebp-0E4h]
00661932 mov       ecx,39h
00661937 mov       eax,0CCCCCCCCh
0066193C rep stos   dword ptr es:[edi]
0066193E mov       ecx,offset _D9812314_dfs_bfs 연습장\dfs_bfs 연습장\소스코드분석@c (066C003h)
00661943 call     @__CheckForDebuggerJustMyCode@4 (066121Ch)

    int a, b, sum;
    scanf("%d %d", &a, &b);
00661948 lea       eax,[b]
0066194B push     eax
0066194C lea       ecx,[a]
0066194F push     ecx
00661950 push     offset string "%d %d" (0667B30h)
00661955 call     _scanf (0661098h)
0066195A add       esp,0Ch

    sum = a + b;
0066195D mov       eax,dword ptr [a]
00661960 add       eax,dword ptr [b]
00661963 mov       dword ptr [sum],eax

    printf("%d", sum);
00661966 mov       eax,dword ptr [sum]
00661969 push     eax
0066196A push     offset string "%d" (0667B38h)
0066196F call     _printf (0661046h)
00661974 add       esp,8

    return 0;
00661977 xor       eax,eax
}

```

왼쪽은 디스 어셈블리한 결과임.

빨간 박스에 있는 숫자들은 각 x86 명령어들의 instruction 값을 의미함.

보면 불규칙적으로 값이 늘어가는 것을 볼 수 있는데 이것이 Intel Instruction Architecture의 특징임.

Intel은 Instruction set의 초기 설계 방식을 CISC로 했기 때문에 각각의 instruction들의 크기가 다르다.

그래서 instruction의 주소가 규칙적으로 증가하지 않는 것.

그러나 시간이 지남에 따라 RISC 방식과 합쳐 CISC + RISC 방식을 도입하였고 이것은 하위 호환성을 고려하여 CISC 방식을 완전히 버리지 못한 것 같다.

이러한 원초적으로 한계가 있을 수 밖에 없는 CISC 방식을 개선하기 위해 슈퍼스칼라 방식 또한 도입하여 성능 개선을 계속해서 해오고 있으며

이것이 MIPS와 ARM에서 사용되는 RISC 방식과는 대비되는 Intel Instruction Architecture의 특징이다

감사합니다.

더 많은 자료를 보고 싶으시다면  
Github : <https://github.com/Jung9928>  
를 방문해주세요.