<<Basic Assembly language for Reverse code engineering>>

Written by Kyanon
zip43@naver.com
http://kyanon.sur3.org
2010. 10. 17.
(Modify 2010. 10. 17.)

-Contents-

Preface.

-이 문서에 대한 기술적인 내용은 기본적으로
CPU Architecture : IA-32
●O.S : Windows
●High-level language : C
●C compiler : Microsoft Visual Studio 2008, 2010
●Debuger : Microsoft Visual Studio 2010, Ollydbg1.1
를 이용하였으며 문서의 기본전제 사항으로 이용되었음을 알려드립니다.

-본 문서는 리버싱을 공부하기 위한 전반적인 기초 지식만을 기술하였으며 보다 깊은 내용은 포함하지 않습니다.
-본 문서에서는 여러 가지 명칭들을 기본적으로 영문표기를 하였으나 한글로 표기한 경우 영문 발음을 그대로 옮기거나 통용되는 말로 표기를 하였습니다.
-틀린 사항이나 수정해야할 사항은 위의 메일로 연락주시기 바랍니다.

Chapter 1. What is Assembly language?

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 아키텍쳐(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 아키텍쳐 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.)

1.2. Form of Assembly language

-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.)

Chapter 2. How to study it for reverse code engineering?

2.1. Need to know what to study assembly language.

-어셈블리어를 공부하기위해서는 “프로그램은 어떻게 진행되는가?, 함수의 호출은 어떤식으로 이루어지는가?”등과 같은 프로그래밍적 관점에 대한 지식이 많이 요구된다.
다만 이러한 어셈블리어를 직접적으로 작성하거나 공부를 할시에는 고급언어로 작성할 때 보다 메모리와 CPU에 대해 조금 더 상세히 알 필요가 있다.

2.1.1. Memory Structure

-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.

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.

2.1.2. CPU Register.

-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.

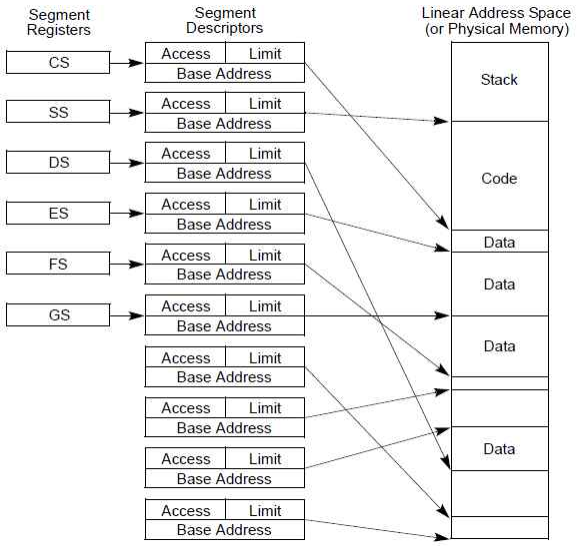
범용 레지스터에는 다음과 같은 종류가 있다.

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.

2.1.3. The relationship between Memory and CPU

-일단 프로그램이 실행되기 위해서는 메모리상에 적제가 되어야 하는데 용량상의 문제, 보안상의 문제 등으로 메모리상에 한꺼번에 모든 프로그램이 올라갈 수는 없으므로 세그먼트라는 이름으로 조각을 내어 당장 필요한 부분 메모리상에 적재를 시킨다. 이러한 작업은 CPU의 Segment Register를 통해 이루어지게 되는데 이러한 과정은 운영체제에 대한 구조와 원리의 이해가 필요하므로 자세한 내용은 생략하도록 하겠다. 중요한 것은 프로그램은 메모리에 필요한 부분만 조각내어져 실행되며, 올라와있는 조각들을 CPU가 읽고 해석한다는 점이다.
아래 그림은 Segment register와 메모리상에 프로그램이 적제되는 과정을 그림으로 나타낸 것이다.

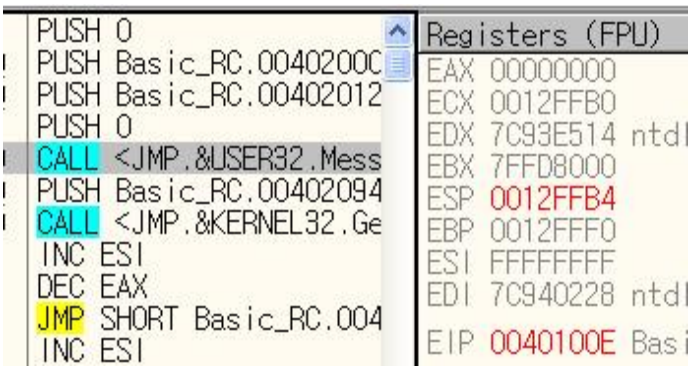


위와 같이 적제된 프로그램들은 다시 각각의 CPU레지스터에 저장된 주소값들을 참조하여 해당 프로그램들의 데이터나 명령들을 읽어오며 실행시키게 된다.

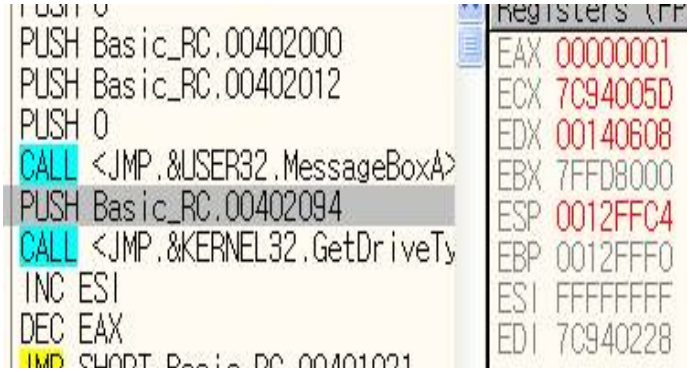
그리고 메모리와 CPU와의 관계에서 알아야 할 중요한 점은 메모리에서 레지스터로, 레지스터에서 레지스터로의 데이터의 복사나 이동은 이루어질 수 있으나 메모리에서 메모리로의 이동은 불가능 하다는 점이다.

2.2. What is Reverser to know?

-리버싱을 위해서는 메모리와 레지스터들의 값이 언제 어떠한 값으로 변하는지를 잘 파악하는 리버싱을 함에 있어 핵심이다. 특히 함수의 호출전과 후의 변화한 값들은 분석을 함에 있어 매우 유용하게 사용될 것이다.



(함수가 호출되기전)



(호출되고난 후, 빨간색으로 변한 것들이 변화된 값)

이러한 값들의 변화를 인지하기 위해서는 어떠한 내용들을 알아야 할까?

만약 일반적인 윈도우 프로그램이라면 당연히 API에 대해서 알아야 할 것이다.

Visual Basic 으로 작성된 프로그램이라면? VB함수에 대해서 알아야 할 것이다.

이 이외에도 리버싱을 위해서는 깊은 지식과 더불어 다양한 언어들의 함수의 기능을 읽을 수 있는 넓은 지식이 요구된다.

함수 이외에도 거의 대부분의 모든 프로그래밍 언어는 “제어문(Conditionals)”을 포함하고 있다.
이러한 제어문은 언어별로 그 형식이 조금씩 다르긴 하겠지만 Compile을 통해 어셈블리어코드로 변하게 되면 대게 비슷한 모습을 보이게 된다. 이러한 일련의 “패턴(Pattern)”들을 잘 파악하는 것이 중요하다.
(이 문서에서도 이후에는 이러한 패턴들에 대한 내용이 주를 이룰 것이다.)
또한 반대로, 같은 언어로 작성된 프로그램이라 할지라도 Compiler에 따라 달라질 수도 있다.

```
PUSH EBP
MOV EBP,ESP
SUB ESP,18
AND ESP,FFFFFFF0
MOV EAX,0
ADD EAX,0F
ADD EAX,0F
SHR EAX,4
SHL EAX,4
MOV DWORD PTR SS:[EBP-C],EAX
MOV EAX,DWORD PTR SS:[EBP-C]
CALL test.00401720
CALL test.004013C0
MOV DWORD PTR SS:[EBP-4],1
MOV DWORD PTR SS:[EBP-8],2
MOV EDX,DWORD PTR SS:[EBP-8]
LEA EAX,DWORD PTR SS:[EBP-4]
ADD DWORD PTR DS:[EAX],EDX
MOV EAX,0
LEAVE
RETN
```

(Compiled by Dev-C++)

```
#include <stdio.h>

int main()
{
00AD1B20  push        ebp
00AD1B21  mov         ebp,esp
00AD1B23  sub         esp,0D8h
00AD1B29  push        ebx
00AD1B2A  push        esi
00AD1B2B  push        edi
00AD1B2C  lea         edi,[ebp-0D8h]
00AD1B32  mov         ecx,36h
00AD1B37  mov         eax,0CCCCCCCCh
00AD1B3C  rep stos    dword ptr es:[edi]
    int a = 1;
00AD1B3E  mov         dword ptr [a],1
    int b = 2;
00AD1B45  mov         dword ptr [b],2

    a = a + b;
00AD1B4C  mov         eax,dword ptr [a]
00AD1B4F  add         eax,dword ptr [b]
00AD1B52  mov         dword ptr [a],eax

    return 0;
00AD1B55  xor         eax,eax
}
```

(Compiled by Visual Studio 2010)

위의 두 프로그램은 아래의 소스를 똑같이 컴파일 한 것 이다.

```
#include <stdio.h>
int main()
{
        int a = 1;
        int b = 2;

        a = a + b;

        return 0;
}
```

그런데 컴파일러에 따라 수행하는 동작은 똑같은 프로그램이여도 어셈블리어 코드에는 차이가 있음을 발견할 수 있다. 이러한 차이가 발생하는 이유는 컴파일러마다 각기 소스코드를 해석하는 방법이 조금씩 다르며, 기타 여러 가지 이유를 통해 본디 소스코드에 포함되어있지 않는 내용들이 포함되어 들어가기 때문이다.(보안상의 이유가 가장 크다.)

만약 자신이 처음보는 패턴을 가지는 소스, 처음보는 함수 혹은 처음보는 형식의 어셈블리어 코드를 보게된다면 MSDN 혹은 구글의 검색이 매우 많은 도움이 될 것이다.

추가적으로, 일반적인 프로그램에서는 보통 소스코드가 그냥 보이지는 않는다. 대게 “패킹(Packing)”이라는 과정을 통해 소스코드를 압축, 숨기는 과정을 거치게 된다. 이러한 프로그램을 분석하려면 “언패킹(Unpacking)"과정을 거치던가, 소스코드가 압축이 풀리면서 메모리에 적제되는 순간을 찾아낼 줄 알아야한다.
패킹 이외에도 리버싱을 방지하기 위해 Anti-reversing 기술이 많이 도입이 되어있는 추세이다. 이러한 방지기법을 우회하기 위해서는 많은 경험밖에 답이 없다.

Chapter 3. Some Assembly code of frequently found in source

3.1. Basic Assembly code

-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.
대표적인예로 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.

이상의 것들이 대표적으로 어셈블리어 소스코드에서 자주 볼 수 있는 내용들이며, CPU Architecture 마다 조금씩 차이가 나는 것도 유념해두어야 한다. IA-32 OP Code에 대해 조금 더 자세한 설명을 보고싶다면 Intel 홈페이지를

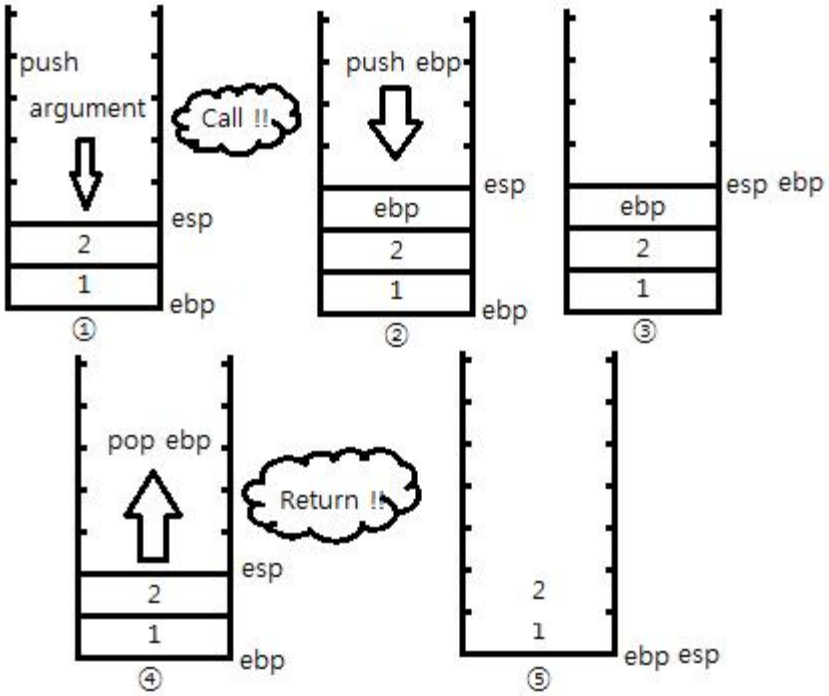
참조하여 IA-32 매뉴얼을 참조하면 된다.

3.2. Funtion

-대부분의 프로그램들은 함수라는 것들의 호출과 리턴을 통해 이루어진다. 사실 C에서의 main함수 조차도 커널영역에서 호출되어 실행되는 함수이다. 그리고 실제로 프로그램들의 많은 동작들이 함수를 통해서 이루어지게 된다. 따라서 이러한 함수의 구조와 원리를 파악하는 것은 리버싱을 함에있어 매우 중요한 작업이라고 할 수 있겠다.

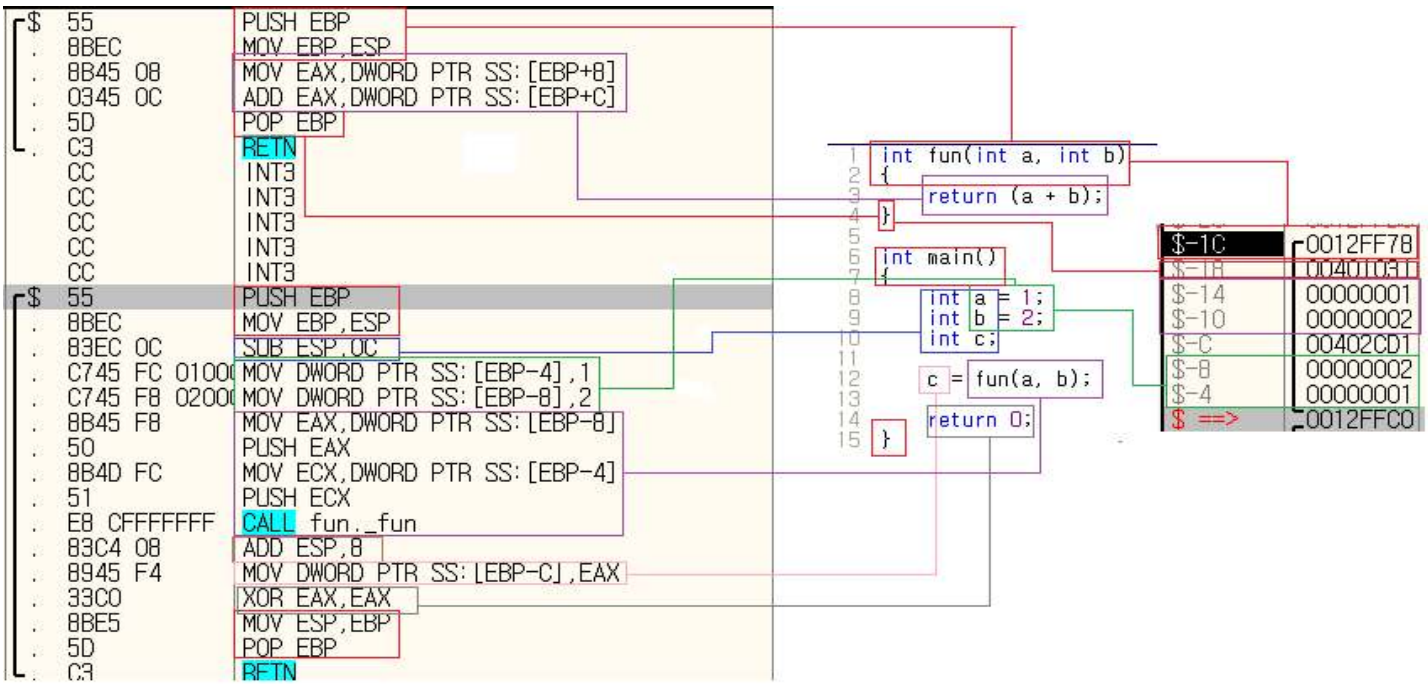
어셈블리어에서는 대게 일정한 형식으로 함수를 시작과 종료를 표현하게 된다.
함수의 시작점에서는 EBP를 STACK에다가 저장을 해두어 Caller함수(함수를 호출한 함수)의 상태를 저장해두고, 새로이 EBP를 ESP의 위치로 저장하는 동작이 이루어진다. 즉, 스택의 맨 꼭대기에 ebp를 위치시킨다는 것이다.
반대로 함수의 종료부에서는 스택에 저장해놨던 EBP를 다시 가져온 뒤, esp를 함수가 호출되던 시점으로 돌린다.
이러한 함수의 시작점을 “Function prologue(함수의 프롤로그)”라 하며, 함수의 종료점을 “Function epilogue(함수의 에필로그)”라 한다.

만약 함수가 인자가 있는 함수라면 인자는 Caller에서 스택에 저장해두고, Callee(호출을 받은 함수)함수에서는 자신의 EBP를 기준으로 아래쪽에 있는 값을 주소를 통해 접근하게 된다.
이상의 과정을 그림으로 표현하면 다음과 같다.



1번 그림에서 전달할 인자 값을 두 개를 스택에 저장한다.
함수를 호출한 뒤 2번 그림에서 기존의 함수의 ebp를 저장한뒤, 3번 그림에서 ebp를 esp로 변경시킨다.
(여기가 함수의 프롤로그이다.)
이후 인자의 접근은 ebp를 기준으로 스택의 주소값을 참조하여 인자값을 가져오게 된다.
(위 그림에서 동작과정은 생략하였다.)
함수의 기능을 모두 수행한 뒤에는 4번 그림과 같이 다시 ebp의 값을 가지고온 뒤, 리턴 후 esp의 값을 호출하기 전의 상태로 되돌리며 스택을 정리한다. 이 때, 값이 사라 진 것이 아니라 접근을 하지 않을 뿐이다.
(만약 4번 그림에서 호출된 함수가 새로운 변수를 사용하였다고 가정을 하면 4번그림에서도 이러한 작업이 일어나게 된다.)
(사실 스택의 정리와 관련되어서는 Calling convention에 따라 달라진다.)

다음 그림은 간단한 기능을 수행하는 소스코드에 대한 전체 어셈블리어 코드와 스택의 모양을 연관지어서 나타낸 그림이다. 천천히 선을 따라가면서 읽다보면 이해가 갈 것이다.

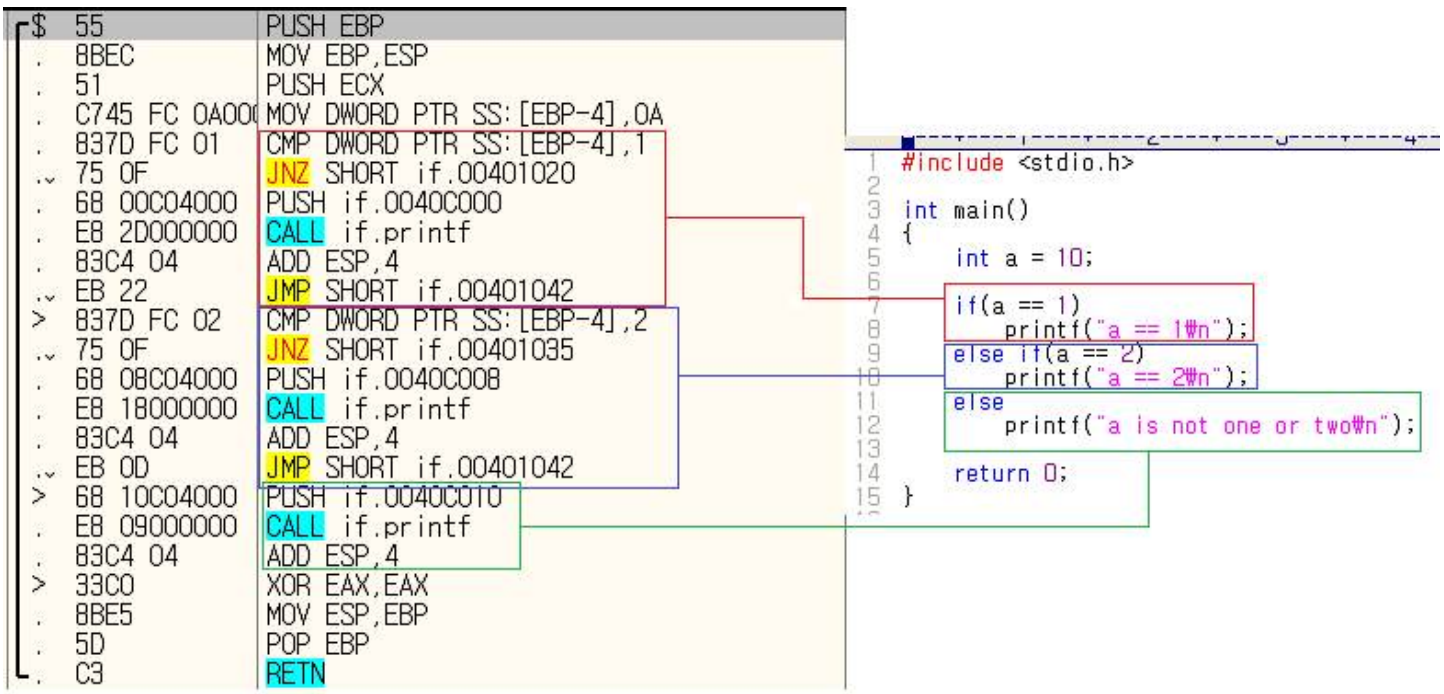


Chapter 4. Pattern example of compiled C source code

4.1. Conditionals

4.1.1. if

-다음 소스코드는 if문의 간단한 예제이다.



소스코드를 살펴보면 각각의 조사식마다 cmp 명령을 통해 값을 비교함을 확인할 수 있다.

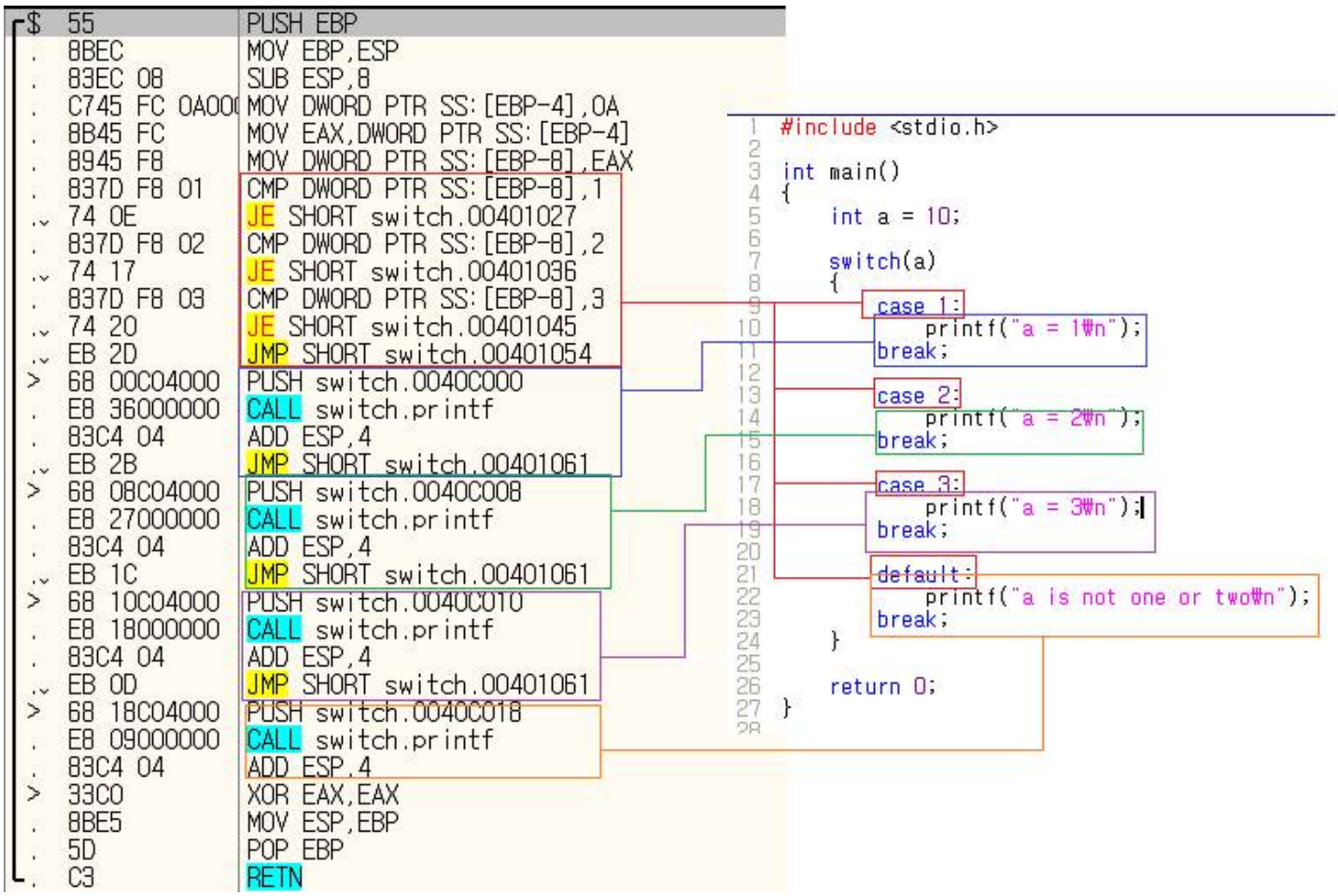
만약 같지 않으면 다음 조사식으로 jmp를 수행하고 같으면 그냥 아래쪽의 문장을 실행한 후, jmp명령을 통해 if문을 빠져나온다.

if을 한마디로 정리하자면 한번 검사하고 수행하고 한번 검사하고 수행하고의 패턴이 나타난다는 점이다.

4.1.2. switch

-다음 소스 코드는 switch문의 간단한 예제이다.

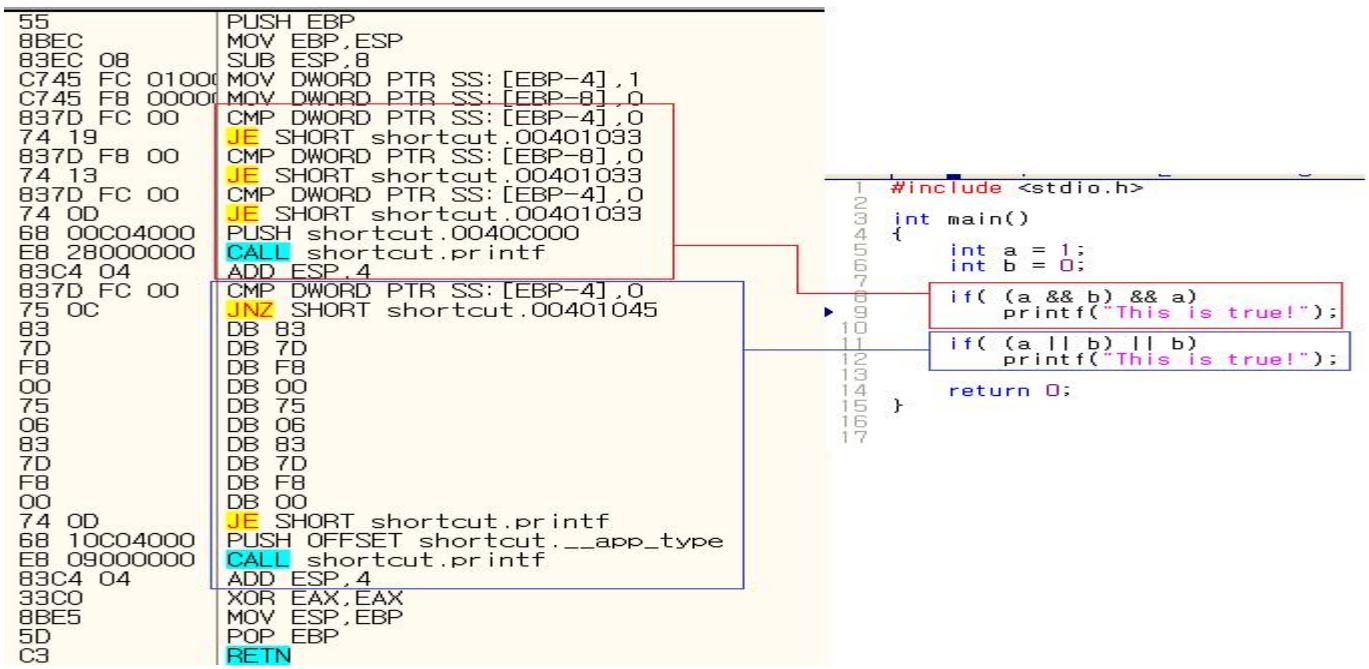
소스의 내용은 if문과 크게 다르지 않다.



if문과의 큰 차이점은 switch문은 검사를 조건문 상단에서 수행한다는 점이다. 즉 검사를 미리 한 뒤에 해당되는 소스코드의 내용으로 점프를 뛰는 것이다.
엄밀히 말해서 조건문의 길이가 길어질수록 if문 보다는 switch문의 속도가 더 빠르다는 것을 예상할 수 있다.
(사실 그 차이는 굉장히 미미하다.)

4.1.3. shortcut

-우선 shortcut이란, 검사를 안해도 넘어가도 되는 상황을 말한다. 예를 들어 'a는 거짓', 'b는 참' 이라는 두 변수가 있고 두 개의 변수에서 and연산을 수행한다고 가정해 보자. a & b의 식에서 a 는 거짓이므로 뒤의 b의 참/거짓의 관계없이 무조건 거짓이 된다. 반대로 'a = 참', 'b = 거짓'에서, or 연산을 하게되면 a | b에서 a가 참이므로 뒤의 b의 참/거짓에 관계없이 무조건 참이 된다. 이렇게 뒤의 항목을 보지 않고 참/거짓을 판단할 수 있을 경우 쇼컷이 수행되며 이러한 경우에는 약간 다른 모습의 어셈블리어 코드를 볼 수 있게 된다.



소스코드를 살펴보면 처음 조건문에서 각 변수들을 0과 비교를 해서 0이 존재하게 되면 더 이상 뒤의 문장을 검색하지 않고 바로 JE를 실행시켜 조건문을 나오는 모습을 볼 수 있다.
두 번째 문장에서는 0과 비교해서 0이 아니라면 뒤의 내용은 검사하지 않고 바로 조건문의 문장을 수행하는 경우를 볼 수 있다.

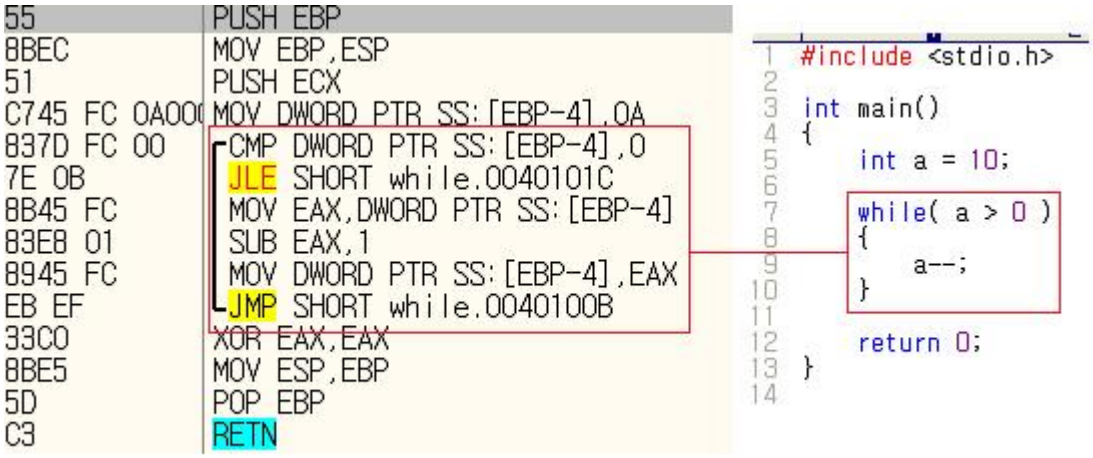
추가적으로 첫 번째 검사를 보면 참인 경우가 아니라 거짓인 경우, 즉 반대를 찾음을 볼 수 있다.
이것은 컴파일러의 해석에 따라 다르긴 하지만 조건문의 경우 조건식을 만족하는 경우를 찾아서 모든 경우의 수를 찾는 것 보다는 하나의 거짓을 찾는 것이 좀 더 연산횟수를 줄일 수 있기 때문에(하나가 거짓이면 그 이후는 연산은 실시하지 않아도 된다) 이러한 형식으로 나오게 된다.
(만약 위 소스에서 만약 참인 경우를 조사를 하였으면 식이 거짓이 됨에도 불구하고 뒤의 연산을 수행하게 된다.)

이 부분은 우리가 실제로 고급언어를 통해 소스코드를 작성할 때와 상반이 되는 부분 이므로 쉬이 착각할 수 있는 부분이기도 하다.

4.2. Routine

4.2.1. while

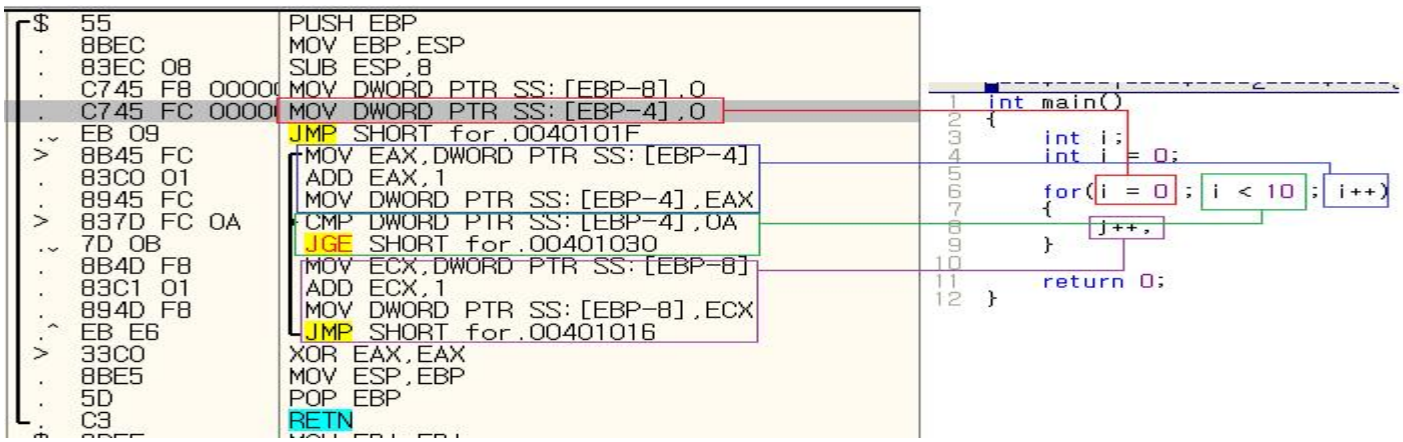
-다음은 while문의 간단한 예제이다.



while문의 특징은 매 반복이 시작될 때 마다 반복문 상단에 검사하는 내용이 존재한다는 것이다.
위 소스에서는 a > 0 큰 경우가 아니라, a <= 0 인 경우를 조사하여 반복문을 탈출함을 볼 수 있는데, 반복문의 경우 참인 횟수가 더 많으므로 반대인 경우—즉 참이 안되는 경우—를 검사하게 되는 것이다.
(만약 위 소스에서 a > 0의 경우를 조사하였다면 매 참이 되는 경우 마다 실행문으로 점프가 일어났을 것이다. 이것은 매우 비효율 적인 일이다.)

4.2.2. for

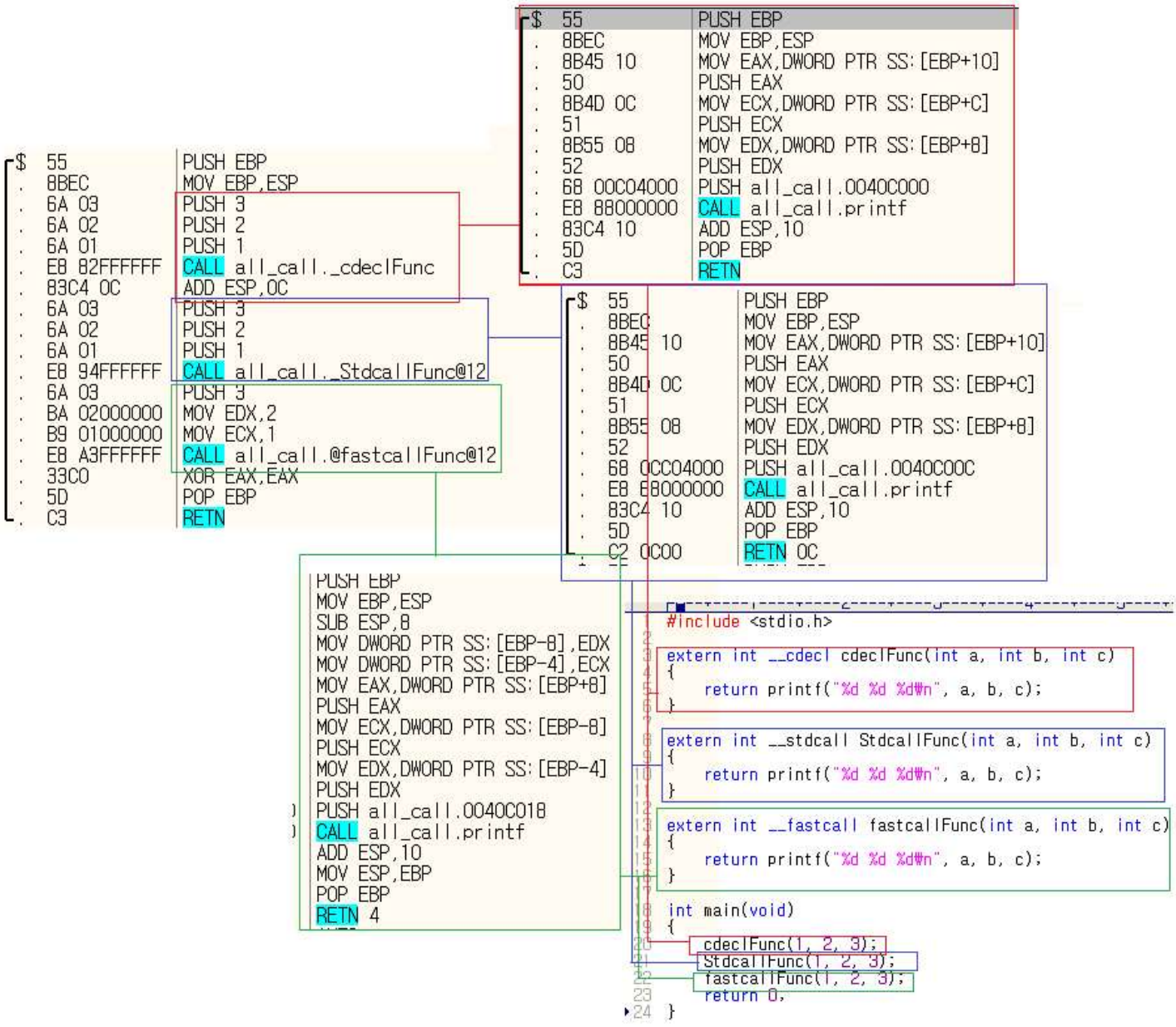
-다음은 for문의 예제이다.



for문의 특징은 우선 가장 앞의 초기화 하는 소스가 들어가는 것이고 그다음 조건식으로 점프를 한다.
조건문이 참이 되면(위 소스에서는 역시 반대의 경우(거짓)를 조사함을 볼 수 있다)아래 반복문의 내용을 수행하고 반복문의 내용이 끝나면 증감식으로 jmp를 수행, 증감문을 수행하고 다시 조건식을 검사하게 된다.
즉, while문과의 큰 차이점이라면 조건식이 반복문 중간에 끼어있게 된다는 점이다
(물론 증감문을 입력하지 않으면 while문과 크게 다르지 않다)

4.3. Calling convention

-Calling convention이란 함수가 호출되는 규약을 말하는 것인데 Argument 전달방법, 전달순서, 스택의 정리, 리턴값의 전달등이 규정되어 있는 것이다. 이는 사용자가 특별히 지정하지 않는 이상 컴파일러별로, 언어별로, 함수별로 다르다. 그리고 사실 어셈블리어 코드를 보기전까지는 큰 차이를 느끼기 어렵다.
아래 그림은 몇 가지의 함수 calling convention을 나타낸 것이다.



우선 맨 위의 함수부터 각각 C Calling convention, Standard Calling convention, Fast Calling convention이며 다음과 같은 차이점들이 있다.

Calling Convention	매개변수 전달방향	매개변수 전달방법	스택정리
cdecl	right -> left	stack	caller
stdcall	right -> left	stack	callee
fastcall	right -> left	register + stack	callee

우선 cdecl은 C 언어에서 기본으로 쓰이는 Calling convention이다. 매개변수가 오른쪽의부터 왼쪽으로 3, 2, 1 의

순서로 전달이 되며, 매개변수는 스택에 저장되어 사용되며 함수가 종료가 되고 난 뒤에는 caller쪽에서 add명령을 수행해 스택을 정리함을 볼 수 있다. 스택정리를 caller에서 하는 경우는 함수로 전달하는 인자의 개수가 정확하지 않을 때 주로 사용된다. (printf 함수가 대표적인 예이다.)

stdcall은 WinAPI등에서 사용되는 calling convention으로 cdecl과 거의 흡사하나 스택정리를 callee에서 수행한다. 넘어가야할 인자가 정확하게 필요한 경우 혹은 프로그램의 전체 크기를 줄일 때 사용된다.
(만약 함수 호출이 10번이 이루어 진다고하면 cdecl에서는 스택 정리 소스가 10줄이 추가 되지만, stdcall에서는 단 한 줄만 있으면 된다.)

fastcall은 매개변수를 register와 stack을 함께 사용해서 넘기는 방식인데 register를 이용하는 만큼 조금더 빠른속도를 기대할 수 있는 호출방식이다. 스택정리는 callee에서 함을 볼 수 있다.
빠르게 처리해야할 함수를 호출할 때 사용되는 방식이나 사실 오늘날의 컴퓨터에서는 그 효과가 매우 미미하다.

Etc.

-참고한 자료

- ●성공과 실패를 결정하는 1%의 프로그래밍의 원리(성안당)
- ●리버싱, 리버스 엔지니어링 비밀을 파헤치다(에이콘)
- ●리버스 코어(http://www.reversecore.com/)
- ●위키피디아(http://ko.wikipedia.org)
- ●기타 구글검색