Reversing Essential Note

< 목 차 >

1. Compile 된 연산의 이해

1) 산술연산 Flag

(i) Overflow Flag

☑ Carry Flag 와 Overflow Flag 는 모두 Overflow 가 발생했음을 알리기 위한 Flag 이다. 즉, 어떤 산술 연산의 결과가 너무 커서 그것을 인자에 완전히 표현하기 힘들때 Program 에게 그런 상황(Overflow) 를 알리기 위해서 사용된다. 두 Flag 의 차이점은 Program 이 처리하는 Data 의 Type 과 관계가 있다.

☑ 부호 있는 정수 값은 동일한 부호 없는 정수 값 보다 1bit 작으므로(부호bit) 두 정수의 Overflow 는 서로 다르게 인지되어야 한다. 이런 이유로 CF 와 OF 는 각기 다르게 사용된다.

* ADD, SUB 연산은 두 Flag 값 모두에 영향을 미친다.

```
1    mov ax,0x1126        ; 10진수 4390
2    mov bx,0x7200        ; 10진수 29184
3    add ax,bx
```

[그림 1 - 1 - 1] 부호에 따라 다른 결과 값을 내는 Sample Code

➡ 결과 값 0x8326은, AX Register 값이 부호 없는 값이라면 10진수 33574가 되고 부호 있는 값이라면 Overflow 가 발생한다.

* 부호 있는 정수에서 최상위 bit 가 1이면 음수라는 것을 의미하므로 10진수 -31962가 된다.
* 이때 OF (부호 있는 인자에 대한 Overflow) = 1, CF (부호 없는 인자에 대한 Overflow) = 0

(ii) Zero Flag

☑ ZF 는 산술 연산의 결과 값이 0 일때 1이 되고, 그 외의 경우에는 0 이 된다. 이는 매우 다양한 경우에 사용되지만, 두 인자의 값이 같은지 비교하는 경우에 가장 많이 사용된다.

* CMP 명령에서 dst - src 의 결과 값이 0 이면 ZF 를 1로, 0이 아니면 ZF 를 0으로 만든다.

(iii) Sign Flag

☑ SF 는 결과 값의 최상위 bit 값을 나타낸다.

* SF 의 값이 1이면 음수, 0 이면 양수 또는 0을 나타낸다.

2) 기본적인 정수 산술 연산

(i) 더하기와 빼기

☑ ADD 와 SUB 명령은 여러가지 Type 의 인자를 사용할 수 있다. 즉, Register 이름, Hard Coding 된 값이나 Memory 주소를 사용할 수 있다. 그리고 명령의 수행결과는 왼쪽 인자 (dst operand) 에 저장된다.

(ii) 곱하기와 나누기

☑ IA-32 Processor 는 곱하기와 나누기 연산에 Binary Shift 연산을 사용한다. SHL 명령은 2 의 제곱수로 곱하는 것과 동일한 결과를 SHR 명령은 2의 제곱수로 나누는 것과 동일한 결과를 만들어 낸다. 이 후에 Compiler 는 일반적으로 필요한 경우 결과 값을 보정하기 위해서 추가적인 더하기나 빼기 연산을 수행한다.

```
1    lea eax,DWORD PTR [edx+edx]
2    add eax,eax
3    add eax,eax
4    add eax,eax
5    add eax,eax
```

[그림 1 - 2 - 1] Intel 의 LEA 와 ADD 명령을 조합한 32를 곱하는 연산

➡ Code 의 크기는 SHL 명령에 비해 코드가 크지만, 실질적으로 더 빠르다.

✱ 이는 LEA 명령과 ADD 명령이 모두 저지연, 고속 처리 명령이기 때문이다.

```
1    lea eax,DWORD PTR [esi + esi * 2]
2    sal eax,4
3    sub eax,esi
```

[그림 1 - 2 - 2] GCC 의 LEA 와 SHL (SAL) 명령을 조합한 11을 곱하는 연산

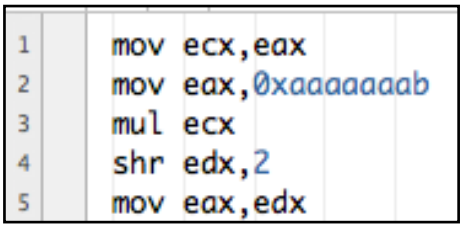
➡ ESI Register 에 3을 곱하기 위해 LEA 명령을 사용하고 있다. 그리고 SAL 명령을 이용해서 4를 더 곱한다. 그 다음에는 곱한 결과의 값에서 원래의 값을 뺀다.

☑ 나누기 연산은 최적화된 나누기 연산 기술인 역곱을 사용한다.

✱ 역곱이란 나누기보다 4배 ~ 6배 정도 빠른 곱하기 연산을 사용해서 나누기 연산을 하는 것이다. 예를들어 30/3 의 경우 1/3 인 0.3333 을 30에 곱한다.

나눗수	**32bit** 역수	역수 값(분수)	나눗수와의 결합
3	0xAAAAAAAB	2/3	2
5	0xCCCCCCCD	4/5	4
6	0xAAAAAAAB	2/3	4

[표 1 - 2 - 1] 나누기 연산을 위한 역곱의 예



```
1    mov ecx,eax
2    mov eax,0xaaaaaaab
3    mul ecx
4    shr edx,2
5    mov eax,edx
```

[그림 1 - 2 - 3] 전형적인 역곱을 수행하는 Code

➡ 이 Code 는 ECX 에 0xAAAAAAAB 를 곱하고 있다. 이는 2/3 을 곱하는 것과 동일하며, 연산 후에 오른쪽으로 2bit Shift 를 시키고 있다. 2/3을 곱한 후 4로 나누는 연산은 결국 6으로 나누는 연산과 동일하다.

* 여기서 결과 값을 EDX 에서 가져 오는 이유는 MUL 명령의 64bit 결과 값중 상위 32bit 는 EDX 에 하위 32bit 는 EAX 에 저장되기 때문이다.
* 역수의 정밀함에는 한계가 있기 때문에 Compiler 는 역수를 한번 더한 후 다시 역수를 뺀다.

3) 64bit 연산

(i) 더하기

☑ ADD 명령과 ADC(Add with Carry) 명령이 모두 사용된다. ADC 명령은 결과 값에 CF 의 값을 더한다는 것이 ADD 와 다르다.

```
1    mov esi,[Operand1_Low]
2    mov edi,[Operand1_High]
3    add eax,[Operand2_Low]
4    adc edx,[Operand2_High]
```

[그림 1 - 3 - 1] 64bit 더하기 연산의 예

➡ 두 인자의 하위 32bit 는 ADD 명령으로, 상위 32bit 는 ADC 명령을 이용한다. 이 경우 ADD에 의한 CF 값도 더한다.

✱ 두 인자의 결과 값은 EDX:EAX 에 저장이 된다.

(ii) 빼기

```
1    mov eax,[Operand1_Low]
2    sub eax,[Operand1_Low]
3    mov edx,[Operand2_High]
4    sbb edx,[Operand2_High]
```

[그림 1 - 3 - 2] 64bit 빼기 연산의 예

➡ CF 의 값이 하위 32bit 와 상위 32bit 를 연결하는 “빌림”의 의미라는 것을 제외하고는 64bit 연산과 동일하다. 두 인자의 하위 32bit 는 SUB 명령을 이용해서 빼고, 상위 32bit 는 SBB 명령을 이용해서 뺀다. 이 경우 CF 값도 함께 뺀다.

(iii) 곱하기와 나누기

☑ 64bit 곱하기 연산은 allmul 이라는 미리 정의된 함수를 사용한다. 이 함수는 두 64bit 값에 대한 곱하기 연산이 수행될 때마다 호출된다.

☑ 64bit 나누기 연산은 alldiv CRT 라는 미리 정의된 함수를 사용한다.

4) 형 변환

(i) Zero 확장

☑ 부호 없는 정수의 크기를 증가 시키고자 할때 주로 사용하는 명령은 MOVZX 명령이다. 이는 작은 인자를 큰 인자에 복사한 후 Zero 확장을 수행한다. 단순히 크기가 큰 목적지 인자로 값을 복사한 후 최상위 bit 을 0 으로 채우는 것이다.

* 이는 일반적으로 복사되는 값이 부호 없는 값이라는 것을 의미한다.

(ii) 부호 확장

☑ 부호 있는 정수를 더 큰 부호 있는 정수로 변환할 때 발생하며, 음수는 2의 보수를 이용해서 표현하므로 부호 있는 정수를 확장하기 위해서는 음수의 모든 상위 bit 들을 1로 만들어 줘야고, 양수는 0으로 만들어 줘야한다.

* 32bit 에서는 MOVSZ 명령을 사용한다.
* 64bit 에서는 CDQ(uard) 명령을 사용해서 EDX:EAX 에 저장하는데 EDX 에는 상위 32bit의 값이, EAX 에는 하위 32bit 의 값이 포함되어 있다. EDX와 EAX 를 하나로 묶어서 하나의 부호 있는 64bit 정수로 취급하며, 취급전에 EDX 의 값이 0 이면 부호 없는 정수임을 알 수 있다.

2. Code 구조 해석

1) Low-level Logic 이해

(i) 인자 비교

dst operand	**src operand**	인자 간의 관계	**Flag** 변화
X >= 0	Y >= 0	X = Y	OF = 0, SF = 0, ZF = 1
X >= 0	Y >= 0	X > Y	OF = 0, SF = 0, ZF = 0
X < 0	Y < 0	X > Y	OF = 0, SF = 0, ZF = 0
X > 0	Y > 0	X < Y	OF = 0, SF = 1, ZF = 0
X < 0	Y >= 0	X < Y	OF = 0, SF = 1, ZF = 0
X < 0	Y > 0	X < Y	OF = 1, SF = 0, ZF = 0
X > 0	Y < 0	X > Y	OF = 1, SF = 1, ZF = 0

[표 2 - 1 - 1] 부호 있는 인자 값에 대한 CMP, SUB 연산 결과

➡ ZF 값이 1 이면 빼기 연산 결과 값이 0 이므로 두 인자 값이 동일
➡ 세 Flag 값이 모두 0인 경우는 연산 결과값이 Overflow 없는 양수이므로 X가 큼
➡ 연산 결과 값이 음수이고 Overflow 가 없으면 Y가 큼
➡ 연산 결과 값이 양수이고 Overflow 가 발생하면 Y가 큼
➡ 연산 결과 값이 음수이고 Overflow 가 발생하면 X가 큼

인자 간의 관계	**Flag** 변화
X = Y	CF = 0, ZF = 1
X < Y	CF = 1, ZF = 0
X > Y	CF = 0, ZF = 0

[표 2 - 1 - 2] 부호 없는 인자 값에 대한 CMP, SUB 연산 결과

➡ ZF 값이 1이면 빼기 연산 결과 값이 0 이므로 두 인자 값이 동일
➡ 두 Flag 값이 모두 0인 경우는 연산 결과 값이 Overflow 없는 양수 이므로 X가 큼
➡ Overflow 가 발생하는 경우는 Y가 X보다 큼

(ii) 조건 Code

mnemonic	Flag	만족 조건
G, NLE	ZF = 0 & ((OF = 0 & SF = 0)	(OF = 1 & SF = 1)	X > Y
GE, NL	(OF = 0 & SF = 0)	(OF = 1 & SF = 1)	X >= Y
L, NGE	(OF = 1 & SF = 0)	(OF = 0 & SF = 1)	X < Y
LE, NG	ZF = 1	((OF = 1 & SF = 0)	(OF = 0 & SF = 1)	X <= Y

[표 2 - 1 - 3] CMP, SUB 에 대한 부호 있는 조건 Code

➡ G, NLE 의 경우 인자가 동일한지 확인하기 위해 ZF 를 사용하며, X > Y 비교를 위해 SF 를 사용해서 결과 값이 Overflow 없는 양수 이거나, Overflow 발생한 음수인지 판단
➡ GE, NL 의 경우 ZF 값이 0인지 확인하는 것을 제외하고는 위의 경우와 동일
➡ L, NGE 의 경우 X < Y 비교를 위해 Overflow 발생한 양수이거나, Overflow 없는 음수인지 판단
➡ LE, NG 의 경우 ZF 값이 0인지 확인하는 것을 제외하고는 위의 경우와 동일

mnemonic	Flag	만족 조건
A, NBE	CF = 0 & ZF = 0	X > Y
AE, NB, NC	CF = 0	X >= Y
B, NAE, C	CF = 1	X < Y
BE, NA	CF = 1	ZF = 1	X <= Y
E, Z	ZF = 1	X = Y
NE, NZ	ZF = 0	X != Y

[표 2 - 1 - 4] CMP, SUB 에 대한 부호 없는 조건 Code

➡ A, NBE 의 경우 Y 가 X 보다 크지 않다는 것을 확인하기 위해서 CF 를 사용며, ZF 를 사용해서 X, Y 가 동일한 지 확인
➡ AE, NB, NC 의 경우 CF 만 확인한다는 것을 제외하고는 위와 동일
➡ B, NAE, C 의 경우 CF 값이 1이면 X < Y 을 알 수 있음
➡ BE, NA 의 경우 ZF 를 확인한다는 것을 제외하고는 위와 동일
➡ E, Z 의 경우 ZF 값이 1이면 결과 값이 0 이고 X = Y 임을 알 수 있음
➡ NE, NZ 의 경우 ZF 값이 0 이면 결과 값이 0 이 아니고 X != Y 를 알 수 있음

2) 제어 흐름과 Program 구조

(i) 함수 해석

☑ IA-32 Processor 에서는 거의 항상 Call 명령으로 함수를 호출한다. 이는 현재의 명령 Pointer 를 Stack 에 저장하고, 해당 함수 주소로 JMP 한다. 따라서 쉽게 구별 가능하다.

☑내부 함수는 구현 Code 를 포함하고 있는 동일한 실행 Binary에서 호출된다. Compiler 는 일반적으로 호출할 함수의 주소를 Code 에 삽입함으로써 내부 함수 호출 Code 를 만들어 낸다. 따라서 Code 내에서 내부 함수 호출을 쉽게 찾아 낼 수 있다.

✱ 일반적으로 " Call CodeSectionAddress " 와 같은 방식을 띈다.

☑ Import 된 함수는 Module 이 다른 실행 Binary 내에 구현된 함수를 호출할 때 발생한다. 이는 IAT (Import Address Table) 을 이용해서 호출을 구현하며, Import Directory 는 실행 시에 해당 실행 Binary 에 있는 함수와 Matching 시키기 위해 사용되며 IAT 는 해당 함수의 실제 주소를 저장한다.

✱ 전형적으로 " Call DWORD PTR [IAT_Pointer] " 와 같은 방식을 띄며, 여기에서 DWORD PTR 이 사용된다는 것은 IAT_Pointer 의 주소가 아닌 IAT_Pointer 가 가리키는 주소로 JMP 하라는 것이다.

(ii) 조건 분기문

단일	High-Level Logic 의 의도를 유지시키기 위해서 조건에 따른 JMP 문의 의미를 항상 반대로 만들어야 한다.
양 방향	조건이 만족되지 않았을 때의 Code Block 이 실행된 다음에 if-else 문 이후의 Code 가 실행된다는 점이 단일 조건 분기문과 다르다.
다중	각 조건이 만족될 때 실행되는 Code Block 이 많고, 전체 조건 Block 의 끝으로 JMP 하는 명령이 많다는 점이 양 방향 조건 분기문과 다르다.
복합	AND, OR 이 있으며 OR 조건에서 GCC는 직관적이며 가독성이 높지만 성능상의 손실이 있다. MS Compiler 는 첫번째 조건 JMP 는 해당 조건이 만족됐을 때 실행되는 Code 를 가리키게 하고, 두번째 조건식은 반대로 바꾼다.
n 방향	Table 구현과, Tree 구현이 있다.
Loop 문	While, For, Do_while, Break;, Continue 등이 있다.

[표 2 - 2 - 1] 조건 분기문의 종류

3) Working Set Tuning 의 효과

(i) 함수 Level 의 Working Set Tuning

☑ Program 이 실행되면 Working Set Tuner 는 어느 함수가 자주 실행되는지 관찰하고 호출 빈도에 근거해서 Binary 안의 함수 순서를 바꾼다.

✱ Module 의 시작부에 있으면 더 자주 호출되는 함수이며, 끝 부분에 있으면 주로 Error 처리나, 특별한 경우에만 호출되는 함수일 것이다.

Tuning 전	Tuning 후
Function1 (Medium Popularity)	Function3 (High Popularity)
Function1_Condition1 (Frequently Executed)	Function3_Condition1 (Sometimes Executed)
Function1_Condition2 (Sometimes Executed)	Function3_Condition2 (Rarely Executed)
Function1_Condition3 (Frequently Executed)	Function3_Condition3 (Frequently Executed)
Function2 (LowPopularity)	Function1 (Medium Popularity)
Function2_Condition1 (Rarely Executed)	Function1_Condition1 (Frequently Executed)
Function2_Condition2 (Sometimes Executed)	Function1_Condition2 (Sometimes Executed)
Function3 (High Popularity)	Function1_Condition3 (Frequently Executed)
Function3_Condition1 (Sometimes Executed)	Function2 (LowPopularity)
Function3_Condition2 (Rarely Executed)	Function2_Condition1 (Rarely Executed)
Function3_Condition3 (Frequently Executed)	Function2_Condition2 (Sometimes Executed)

[그림 2 - 3 - 1] 실행 Binary 에서의 함수 Level Working Set Tuning

(ii) Line Level 의 Working Set Tuning

☑ 개별 함수 내부의 조건 Code 부분의 위치를 변경함으로써 좀더 효과적인 Tuning 을 수행한다. Working Set Tuner 는 Program 내의 모든 조건에 대한 사용통계를 기록하고, 이를 근거로 재배치를 수행한다.

✱ Reverser 는 어느 조건의 Code 가 많이 실행되는지만 알뿐, 중요한지는 모른다.

Tuning 전	Tuning 후
Function3 (High Popularity)	Function3 (High Popularity)
Function3_Condition1 (Sometimes Executed)	Function3_Condition3 (Frequently Executed)
Function3_Condition2 (Rarely Executed)	Function1 (Medium Popularity)
Function3_Condition3 (Frequently Executed)	Function1_Condition1 (Frequently Executed)
Function1 (Medium Popularity)	Function1_Condition3 (Frequently Executed)
Function1_Condition1 (Frequently Executed)	Function3_Condition1 (Sometimes Executed)
Function1_Condition2 (Sometimes Executed)	Function2 (LowPopularity)
Function1_Condition3 (Frequently Executed)	Function2_Condition1 (Rarely Executed)
Function2 (LowPopularity)	Function2_Condition2 (Sometimes Executed)
Function2_Condition1 (Rarely Executed)	Function3_Condition2 (Rarely Executed)
Function2_Condition2 (Sometimes Executed)	Function1_Condition2 (Sometimes Executed)

[그림 2 - 3 - 2] 동일한 실행 Binary 내에서의 Line Level Working Set Tuning

3. Program Data 해석

1) Stack

(i) Stack Frame

☑ 현재 실행되고 있는 함수가 사용하려고 할당한 Stack 안의 영역이다. 함수에 전달되는 Parameter 와 반환주소가 저장되며, 함수가 사용하는 내부 저장공간 역할을 수행한다.

☑ 대부분의 함수는 Stack Frame 을 설정하는 것으로 시작한다. Parameter 영역과 지역변수 영역 사이에 Pointer 를 유지하면서 빠르게 접근하기 위함이다.

* Pointer 는 일반적으로 EBP 에 저장되고, 반면에 ESP 에는 현재의 Stack 위치를 가리키기 위해서 사용된다.

(ii) ENTER 와 LEAVE 명령

☑ 특정 Type 의 Stack Frame 을 구현하기 위해서 CPU 가 제공하는 명령이다.

* ENTER 명령은 EBP 를 Stack 에 PUSH 하고 EBP가 지역 변수 영역의 꼭대기를 가리키게 설정한다. 또한 동일한 함수 내의 중첩 Stack Frame 도 관리할 수 있게 한다. 매우 복잡하므로 성능 문제가 발생하여 사용을 잘 하지 않는다.
* LEAVE 명령은 단순히 ESP 와 EBP 를 저장되기 이전 상태로 복원한다. 단순한 명령이므로 수많은 Compiler 가 함수의 끝 부분에 이 명령을 사용한다.

(iii) 호출 규약

☑ 함수가 Program 내부에서 호출되는 방식을 정의한다. CALL 과 RET 로 기본적인 함수 호출을 하며 CALL 은 명령 Pointer 를 Stack 에 Push 하고 새로운 Code 로 JMP 한다. RET 명령은 반환주소를 EIP 로 POP 하고 EIP 가 가리키는 주소에서의 실행을 계속 수행한다.

cdecl	표준 C, C++ 호출 규약으로, 함수 호출자가 호출 이후의 Stack 복원을 담당하므로 Parameter 수를 동적으로 결정 가능하다. 하나이상의 Parameter 를 받는 함수가 반환할 때 RET 명령을 인자 없이 사용한다면 거의 확실히 cdecl 함수이다.
fastcall	Register 를 이용해 첫번째, 두번째 Parameter 를 함수에 전달하고, 나머지는 Stack 을 통해서 전달한다.
stdcall	가장 많이 사용되는 호출 규약으로, 함수 자체가 Stack 정리를 한다. RET 명령에 전달되는 인자의 값에서 해당 함수에 전달되는 Parameter 의 수를 알아낼 수 있다.
thiscall	함수 호출이 ECX 에 유효한 Pointer 를 Load 하고 Stack 에 Parameter를 Push 하고 EDX 를 사용하지 않는지 확인함으로써 알아낼 수 있다.

[표 3 - 1 - 1] 함수 호출 규약 비교

2) 기본적인 Data 구성 요소

(i) 전역 변수

☑ 일반적으로 전역변수는 실행 Module 의 Data Section 의 고정된 주소에 위치한다. 그리고 전역 변수에 접근할 경우에는 Hard Coding 된 전역 변수 주소를 사용하므로 찾기 쉽다.

* Compiler 는 Hard Coding 된 주소를 전역 변수 이외에는 거의 사용하지 않는다.
* 지역변수에 Static Keyword 를 사용하면 전역변수로 변환되어 모듈의 Data Section 에 저장된다. 이럴 때에는 해당 변수가 하나의 함수 내에서만 접근되는 지 확인함으로 판별 가능하다.

(ii) 지역 변수

☑ 지역변수에는 일반적으로 Counter, Pointer, 다양한 종류의 일시적인 정보들이 들어간다. Compiler 는 지역변수를 Stack 에 저장하거나 Register 에 저장하는 방법으로 관리한다.

(iii) Import 된 변수

☑ 다른 Binary Module (DLL) 에 저장되어 있는 전역 변수이다. Import 된 변수는 기타 다른 변수와는 다르게 이름을 가지므로 Reverser 에게 중요하다. 이는 가독성을 향상시킨다.

```
1    mov     eax,DWORD PTR [IATAddress]
2    mov     ebx,DWORD PTR [eax]
```

[그림 3 - 2 - 1] Import 변수에 접근하는 일반적인 Code

➡ 다른 Pointer 를 가리키는 Pointer 를 통해 Data 를 간접적으로 읽어 들이고 있다. Import 된 변수인지 여부는 IATAddress 값으로 알 수 있다. 결국 첫번째 Pointer 가 IAT 를 가리키는 Pointer 가 되고 두번째가 Import 된 변수를 가리키는 Pointer 가 된다.

(iv) 상수

☑ C 와 C++ 에서는 const Keyword 를 사용해서 전역 변수를 정의 할 수 있다. 이렇게 하면 일반 전역 변수에 접근하는것 처럼 상수에 접근하는 Code 가 만들어진다.

* 다른 개발 tool 은 이러한 형태의 상수를 다른 전역 변수와 함께 Data Section 에 위치시킨다.

(v) TLS (Thread-Local Storage)

☑ Window OS 는 TLS API (TlsAlloc, TlsGetValue, TlsSetValue) 로 TLS 저장소를 할당할 수 있다.

* 또 다른 방법으로는 전역 변수를 declspec (thread) 속성으로 정의함으로써, 실행 Image 의 Thread-local Section 에 해당 변수를 위치시킨다.

3) Data 구조체

(i) 일반적인 Data 구조체

☑ 여러 Data Type 의 Field 들이 모인 Memory Block 으로 각 Field 는 Memory Block 의 시작 위치에서부터 연속적으로 위치한다.

✱ 구조체의 마지막 Member 로 유동적인 Data 구조체를 만드는 것이 가능하며, 실행시에 특정 크기의 Data 구조체를 동적으로 할당하는 Code 를 만드는 것도 가능하다.

☑ 일반적으로 Processor 의 Word 크기로 정렬하여 성능 저하를 예방한다.

(ii) 배열

☑ Data 가 Memory 에 연속적으로 저장된 Data List 이다. Compiler 는 배열에 접근하기 위해서 거의 항상 어떤 종류의 변수를 객체의 Base 주소에 더하므로 배열에 접근 하는 Code 를 쉽게 찾을 수 있다.

✱ Source Code 에 Hard Coding 된 배열의 Index 가 포함되어 있다면 구조체와 구별이 불가능

☑ 배열의 경우 구조체와는 달리 정렬을 수행하지 않는다. (Memory 낭비, 순차접근 때문)

```
1    mov     eax, DWORD PTR [ebp-0x20]
2    shl     eax, 4
3    mov     ecx, DWORD PTR [ebp-0x24]
4    cmp     DWORD PTR [ecx+eax+4], 0
```

[그림 3 - 3 - 1] Loop 문에서 발췌한 Data 구조체 배열에 접근하는 일반적인 형태의 Code

➡ 지역 변수 ebp - 0x20 을 EAX 에 Load 해서 왼쪽을 4bit Shift 시키므로 ebp - 0x20 은 Loop 의 Counter 이다. 주로 배열의 Index 에 이와 같은 연산을 수행한다.
➡ Shift 연산을 수행 후 ECX 에 배열의 Base Pointer 로 보이는 ebp - 0x24 를 Load 한 후 곱하기를 수행한 값과 4를 더한다. 이는 전형적인 Data Type 구조체에 대한 접근 Code 이다.
➡ 첫 번째 변수 (ECX) 는 배열의 Base Pointer 이고, 두 번째 변수 (EAX) 는 배열에서의 Byte Offset 값이다. 이 값은 현재의 Index 값에 배열의 Item 크기를 곱해서 구한다. 마지막으로 4를 더함으로써 우리는 Data 구조체의 2번째 Item 에 접근함을 알 수 있다.