어셈블리 코드 해석에 필요한 기본 지식

이용일(foryou2008@nate.com)

2008-05-30

컴퓨터 과학을 접하면서, 컴퓨터 엔지니어에게 또는 과학도에게 필요한 마인드 중의 하나가 바로 철저함이고 꼼꼼함이라 생각합니다. 여기서 철저함이란 정확하게 알고 있어야 한다는 것입니다. 컴퓨터에 사용되는 어떠한 기술을 접할 때, 대충 "이렇게 저렇게 하니 되더라" 라는 식의 경험적인 판단보다는, 그 기술과 관련한 매뉴얼을 철저하게 조사하고 공부하고 확인해야 한다는 것이죠. 왜냐하면 경험적인 판단에 의존하다보면, 경험하지 못한 상황에 부딪혔을 때, 올바른 대처를 하지 못하거나 잘못된 결정을 내릴 수 있기 때문입니다.

본 문서에서는 어셈블리 코드 해석에 필요한 몇 가지 기본 지식을 다룰 것입니다. IA-32에서 사용하는 플래그 레지스터에서부터, 컴파일러 최적화 부분까지를 다루었습니다. (문서에 사용된 코드나 그림의 일부는 Reversing: Secrets of Reverse Engineering에서 발췌했음을 밝힙니다.)

산술 플래그(Arithmetic Flags)

컴퓨터 프로그램에서 많이 사용하는 연산 중 하나는 바로 산술 연산이다. 산술 연산의 결과는 더하고, 빼고, 곱하고, 나누는 결과값의 단순한 의미 그 이상이 될 수 있다. 왜냐하면 이러한 산술 연산의 결과가 프로그램의 실행 흐름을 결정하는 경우도 있기 때문이다. 예를 들어 "ax 와 7을 뺄셈한 결과값이 0이면 A로 분기하라." 와 같은 조건부 명령어의 경우처럼 산술 연산의 결과에 따라 A로 분기할 수도 있고, 그렇지 않을 수도 있다.

산술 연산의 결과라는 것은 연산의 결과값 뿐만 아니라 연산 과정에서 발생한 여러 상태 정보를 포함한다. 여기서 상태 정보란 연산 과정에서 오류는 없었는지, 연산 결과값이 양수인지 음수인지, 결과값이 0인지 등에 대해 알려주는 정보이다. 우리가 상태 정보에 대해 관심을 가져야 하는 이유는 바로 프로그램의 실행 흐름을 결정하는 조건부 명령어에서 이러한 상태 정보를 이용하기 때문이다.

이와 같이 연산 결과에 대한 상태 정보는 EFLAGS 플래그 레지스터(IA-32 계열에 한함)에 저장된다. 이 중에서 특히 산술 연산 상태 정보를 저장하고 있는 산술 플래그들에 대해서 살펴볼 것이다.

캐리 플래그(CF)와 오버플로우 플래그(OF)

우리는 산술 연산시 [그림 1]과 같은 경우가 발생했을 때 오버플로우가 발생하였다고 말할 수 있다.

양수 + 양수 = 음수
음수 + 음수 = 양수

[그림 1] 오버플로우 (Signed Operation일 경우)

즉, 연산의 결과값이 너무 크거나 혹은 너무 작아서, 그 값이 오퍼랜드(Destination Operand)가 표현 가능한 수의 범위를 벗어나는 경우이다.

캐리 플래그(CF)와 오버플로우 플래그(OF)는 산술 연산 후 이러한 오버플로우가 발생했는지에 대한 상태 정보를 알려주는 플래그이다. 이 두 플래그의 차이점은 Signed 연산에 대해 오버플로우가 발생했을 때는 OF가 1로 설정되고, Unsigned 연산에 대해 오버플로우가 발생했을 때는 CF가 1로 설정된다는 것이다.

오버플로우 플래그에 대해 더 자세히 알아보기 전에 Signed 연산과 Unsigned 연산에 대한 이해가 필요하다. Signed 연산이란 연산에 사용되는 오퍼랜드를 Signed 데이터 타입으로 간주하고 연산을 한다는 것이며, Unsigned 연산은 연산에 사용되는 오퍼랜드를 Unsigned 데이터 타입으로 간주하고 연산을 한다는 것이다. Signed 데이터 타입이란 양수 혹은 음수를 표현할 수 있는 데이터 타입이며, 이를 위해 최상위 비트가 부호 비트로 사용되는 데이터 타입이다. Unsigned 데이터 타입은 음수를 표현할 수 없고, Signed 데이터 타입에서 부호 비트로 사용되었던 최상위 비트는 수를 표현하기 위한 일반 비트로 사용된다.

덧셈과 뺄셈의 경우에 Signed/Unsigned 연산의 바이너리 결과값이 동일하기 때문에 IA-32에서는 각각 ADD와 SUB 명령어 한 개만을 제공하고 있으며, 곱셈과 나눗셈의 경우

Signed/Unsigned 연산의 바이너리 결과값이 다르기 때문에 Signed 버전 명령어와 Unsigned 버전 명령어를 따로 제공하고 있다.

[그림 2]의 샘플 코드를 통해 이를 더 자세히 살펴보자.

```
mov    ax, 0x4E20          ; 10진수로 20000
mov    bx, 0x32C8          ; 10진수로 13000
add    ax, bx              ; 20000 + 13000 = 33000
```

[그림 2] 샘플 코드

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Value	1	0	0	0	0	0	0	0	1	1	1	0	1	0	0	0

[그림 3] ax 레지스터 내용

[그림 3]은 [그림 2]의 수행 결과, ax 레지스터(16비트)의 값을 비트단위로 표시한 것이다. 덧셈(ADD) 명령어의 경우 Signed 연산과 Unsigned 연산의 바이너리 결과값은 동일하기 때문에 명령어가 Signed 버전과 Unsigned 버전으로 나뉘지 않는다. 다만 Signed 연산인지 Unsigned 연산인지에 따라 동일한 바이너리 결과값에 대한 해석이 달라질 뿐이다. 이러한 결과값의 해석은 플래그 레지스터를 통해 반영된다. 즉 덧셈 명령어의 실행 후, 동일한 바이너리 결과값에 대해서 Signed 연산으로 해석했을 때 오버플로우가 발생했다면 OF를 1로 설정하고, Unsigned 연산으로 해석했을 때 오버플로우가 발생했다면 CF를 1로 설정하는 것이다.

[그림 3]의 ax 레지스터를 Unsigned 데이터 타입으로 취급하면, 결과값은 33000이며 이는 올바른 결과이다. 하지만 ax 레지스터를 Signed 데이터 타입으로 취급하면, 결과값 33000의 경우 Signed 데이터 타입이 표현할 수 있는 수의 범위(~32767)를 넘어서게 된다. 즉 33000이란 값을 표현하기 위해선 최상위 비트인 15번 비트가 사용되어야 하는데, Signed 데이터 타입에서는 최상위 비트가 부호 비트로 사용되기 때문에 결과값은 음수가 되며, 이는 오버플로우(양수 + 양수 = 음수)가 발생했음을 의미한다. 결과를 정리하면, ax 레지스터의 내용은 [그림 3]과 같을 것이고 상태 정보는 플래그 CF = 0, OF = 1로 설정될 것이다. (그림 4 참조)

분류(16비트)	표현 범위(10진수)	ax 결과값(10진수)	오버플로우
Unsigned	0 ~ 216- 1(65535)	33000	발생안함(CF=0)
Signed	-215 ~ 215-1(32767)	-32536	발생(OF=1)

[그림 4] 덧셈 명령어 수행 결과(CF = 0, OF = 1)

[그림 2]의 ADD 명령어를 보면, Signed 연산인지 Unsigned 연산인지를 특별히 지정해 주는 부분이 존재하지 않는다. 그 이유는 ADD 연산의 경우, Signed 연산과 Unsigned 연산의 바이너리 결과가 [그림 3]과 같이 동일하기 때문이다. (MUL, DIV와 같은 명령어는 Signed 연산를 위한 명령어가 따로 존재한다. Signed 연산과 Unsigned 연산의 바이너리 연산 결과가 다르기 때문이다.)

물론 ADD 연산의 바이너리 결과가 동일하다고 해서, Signed 연산과 Unsigned 연산의 결과가 동일하다는 의미는 결코 아니다. 앞서 설명했듯이, 연산의 결과라는 것은 여기서 말하는 바이너리 결과 뿐만 아니라 상태 정보를 포함하기 때문이다. ADD 연산의 Signed 연산과 Unsigned 연산의 상태 정보는 다를 수 있다. 왜냐하면 바이너리 결과값을 해석하는 관점은 Signed/Unsigned 데이터 타입에 따라 달라지기 때문이다. 따라서 ADD 연산에서 Signed 연산과 Unsigned 연산의 바이너리 결과는 동일하겠지만, Signed 연산에서는

오버플로우가 발생하고 Unsigned 연산에서는 오버플로우가 발생하지 않는 경우가 발생할 수도 있다. 이러한 상태 정보는 조건부 명령어에서 이용하기 때문에, 조건부 명령어도 Signed와 Unsigned로 나뉘며, Signed이냐 Unsigned이냐에 따라 참조하는 상태 정보 플래그가 다르게 된다.

결론은 연산이 Signed/Unsigned에 따라 오버플로우 여부를 판단하는 방법은 다르다. Signed 연산의 경우 오버플로우 여부는 OF 플래그를 확인해야 하고, Unsigned 연산의 경우 오버플로우 여부는 CF 플래그를 확인해야 한다.

제로 플래그(ZF)

산술 연산 결과	0	Not 0
제로 플래그(ZF)	1	0

[그림 5] 제로 플래그(ZF)

ZF는 2개의 오퍼랜드가 같은지를 비교하는 명령어에서 가장 많이 사용된다. 비교 명령어 중의 하나인 CMP는 2개의 오퍼랜드에 대해 서로 뺄셈을 수행한 다음, 결과가 0이면 ZF를 1로 설정한다. CMP 수행 후 ZF가 1로 설정되었다는 것은, CMP에서 사용된 2개의 오퍼랜드가 서로 같은 값을 가지고 있다는 것을 의미한다.

```
cmp    eax, ebx          ; eax와 ebx를 비교(같다면, ZF가 1로 설정된다)
jz     0x1000            ; ZF가 1이면 0x1000으로 분기
```

[그림 6] 제로 플래그(ZF) 사용 예제

사인 플래그(SF)

SF는 연산 결과값에서 최상위 비트의 값을 가진다. 연산에 사용된 오퍼랜드가 Signed 오퍼랜드라면 오퍼랜드의 부호와 SF는 동일한 값을 가진다. Signed 연산에서 SF가 1이라는 것은 결과값이 음수임을 의미하고, 0이라는 것은 양수임을 의미한다.

오퍼랜드 비교(Comparing Operands)

프로그램의 제어 흐름 측면에서 살펴보면, 프로그램은 단순히 코드의 순차적 실행이 아니라 조건(Condition)에 따라 그에 맞는 코드를 실행하도록 제어 흐름이 바뀐다는 것을 알 수 있다.

```
[처음]
명령어 스텝 1;
명령어 스텝 2;
조건이 참이면 [처음]으로 분기;
명령어 스텝 3;
```

[그림 7] 조건에 따른 프로그램의 실행 흐름

프로그램의 제어 흐름을 결정 짓는 것은 대부분 조건이다. 조건이 참인지 거짓인지에

따라 프로그램의 제어 흐름은 완전히 달라지는 것이다. 따라서 조건이 참인지 거짓인지 확인하는 과정, 즉 진리값(Truth Value)을 확인하는 과정을 이해하는 것은 프로그램의 제어 흐름을 파악하는 데 중요하며, 이러한 과정을 IA-32 어셈블리에서는 어떻게 구현하고 있는지 이해하는 것은 상당히 중요하다. IA-32 어셈블리에서는 이와 같은 조건의 진리값 확인을 대부분 오퍼랜드 비교를 통해 구현하고 있다. 여기서 IA-32의 오퍼랜드 비교 메커니즘을 살펴보는 것은 이러한 이유에서이다.

오퍼랜드를 비교할 때 대표적으로 사용하는 명령어가 CMP 명령어이다. CMP 명령어는 첫번째 오퍼랜드로부터 두번째 오퍼랜드를 뺄셈(Operand1 – Operand2)하고, 뺄셈 과정에서 생긴 여러 상태 정보를 프로세서의 플래그에 저장한다. 조건부 명령어(Conditional Instruction)는 CMP 명령어의 상태 정보(플래그)에 따라 적절한 행동을 취하게 된다.

[그림 8]처럼 IF 문이나 WHILE 문 등에서 사용되는 조건이 참인지 거짓인지 판별하는 과정에는 거의 대부분 다음과 같이 오퍼랜드 비교 과정이 포함된다.

if (eax > 0) {
  statement1;
}
statement2;
	cmp eax, 0 ; 오퍼랜드 비교
	jle statement2 ; 조건부 명령어
	statement1:
	…
	statement2:

[그림 8] 오퍼랜드 비교 예

오퍼랜드 비교 연산은 뺄셈을 통해 2개의 오퍼랜드의 크기를 비교한다. 오퍼랜드 비교 연산의 결과를 확인하기 위해서는 상태 정보(플래그)를 확인해야 하는데, Signed 연산인지 Unsigned 연산인지에 따라 확인해야 하는 플래그가 다르다. 따라서 Signed 연산과 Unsigned 연산으로 나눠서 비교 연산을 살펴봐야 한다.

비교 연산(Signed Operation)

Operand 1	Operand 2	Operand 관계	플래그	설명
X >= 0	Y >= 0	X = Y	OF=0 SF=0 ZF=1	ZF가 1이라는 것은 뺄셈 연산의 결과값이 0임을 뜻하며, 따라서 X와 Y는 같다는 것을 의미한다.
X > 0	Y >= 0	X > Y	OF=0 SF=0 ZF=0	SF가 0이라는 것은 뺄셈 연산의 결과값이 양수임을 뜻하며, 따라서 X가 Y보다 크다는 것을 의미한다.
X < 0	Y < 0	X > Y	OF=0 SF=0 ZF=0	위와 동일
X > 0	Y > 0	X < Y	OF=0 SF=1 ZF=0	SF가 1이라는 것은 뺄셈 연산의 결과값이 음수임을 뜻하며, 따라서 이는 Y가 X보다 크다는 것을 의미한다.
X < 0	Y >= 0	X < Y	OF=0 SF=1 ZF=0	위와 동일
X < 0	Y > 0	X < Y	OF=1 SF=0 ZF=0	SF가 0이면, 양수를 나타내고 이는 X > Y를 의미하지만, OF가 1로 설정되었기 때문에 이는 오버플로우가 발생했다는 것을 뜻한다. 따라서 부호의 반전이 일어난 것이며, 따라서 X < Y인 경우라고 볼 수 있다.
X > 0	Y < 0	X > Y	OF=1 SF=1 ZF=0	OF = 1이라는 것은

				오버플로우가 발생했음을 의미하고, 이는 결과값의 부호가 반전되었음을 뜻한다. 결과값의 부호는 SF = 1로써 음수이고, 오버플로우가 발생하였기 때문에 양수에서 음수로 반전된 것이다. 따라서 X > Y인 경우라고 볼 수 있다.

[그림 9] CMP 명령어에 대한 결과 (Signed Operation)

참고로 [그림 9]에서 OF 플래그가 사용되는 이유는 앞서 "산술 플래그"절에서 살펴보았듯이, Signed 연산에 대한 오버플로우 확인은 OF를 확인해야 하기 때문이다. (Signed 오퍼랜드에 오버플로우가 발생하면 OF가 1로 설정된다.)

비교 연산(Unsigned Operation)

Operand 관계	플래그	설명
X = Y	CF=0 ZF=1	ZF = 1이라는 것은 X = Y임을 의미한다.
X < Y	CF=1 ZF=0	CF가 1이라는 것은 오버플로우가 발생했음을 의미하고, X < Y임을 의미한다. (Unsigned 데이터 타입에서는 음수가 존재하지 않는다. 따라서 0보다 작은 결과가 나오면 오버플로우로 판단한다.)
X > Y	CF=0 ZF=0	X가 Y보다 크기 때문에 뺄셈 결과는 0보다 크므로, CF = 0 ZF = 0이다.

[그림 10] CMP 명령어에 대한 결과 (Unsigned Operation)

참고로 [그림 9]에서 CF 플래그가 사용되는 이유는 앞서 "산술 플래그"절에서 살펴보았듯이, Unsigned 연산에 대해서 오버플로우가 발생하면 CF 플래그가 1로 설정되기 때문이다.

조건부 코드(Conditional Codes)

조건부 코드는 조건부 명령어의 실질적인 조건을 지정하는 역할을 한다. 조건부 코드는 단독으로 사용될 수 없으며, 조건부 명령어의 접미사 형태로 붙어서 사용된다. 조건부 코드는 EFLAGS 플래그 레지스터의 값을 참조하여 조건의 진리값을 구한다.

```
cmp    eax, 5
je     SomePlace        ; Jcc에 조건부 코드 'e' 가 사용되었다.
```

[그림 11] 조건부 코드 사용 예

[그림 11]의 je 명령어는 조건부 분기 명령어인 Jcc 계열에서 조건부 코드 e (if Equal) 가 접미사로 붙은 형태이다. 이와 같이 조건부 코드가 나타내는 의미가 직관적인 경우도 있지만, 때에 따라서는 그렇지 않은 경우가 있다. 따라서 조건부 코드가 플래그를 어떻게 이용하는지를 이해하는 것은 프로그램 로직을 정확히 이해하는데 중요하다고 할 수 있다.

앞서 설명했듯이, 조건부 명령어가 사용되기 전에, 우선 오퍼랜드의 비교 과정을 수행하게 되는데, 이때 Signed/Unsigned 연산에 따라 영향을 받는 플래그가 다르다. 따라서 조건부 코드 또한 Signed/Unsigned 연산에 따라 확인해야 하는 플래그가 다르게 된다. 따라서 조건부 코드 또한 Signed 연산과 Unsigned 연산으로 나누어서 살펴보아야 한다.

Signed 연산을 위한 조건부 코드

조건부 코드	진리값 확인	조건	설명
G (Greater) NLE (Not Less or Equal)	ZF = 0 AND ((OF = 0 AND SF = 0) OR (OF = 1 AND SF = 1))	X > Y	X > Y일 경우 참을 만족하는 경우이다. 이를 위해 오퍼랜드 비교 연산 과정으로부터 나온 여러 플래그 값을 확인한다. 우선 X != Y 이어야 하기 때문에 ZF = 0을 확인한다. 그리고 오버플로우가 발생하지 않았다면, 뺄셈 연산은 양수가 되어야 하므로 OF = 0 SF = 0이 된다. 오버플로우가 발생했다면, 음수가 되어야 하므로 OF = 1 SF = 1이 된다.
GE (Greater or Equal) NL (Not Less)	(OF = 0 AND SF = 0) OR (OF = 1 AND SF = 1)	X >= Y	위와 비슷하다. 다만 X = Y 조건도 만족하기 위해, ZF = 0 을 제거한다.
L (Less) NGE (Not Greater or Equal)	(OF = 0 AND SF = 1) OR (OF = 1 AND SF = 0)	X < Y	오버플로우가 발생하지 않았다면, 음수가 되므로 OF = 0 SF = 1이 된다. 오버플로우가 발생했다면, 양수가 되므로 OF = 1 SF = 0이 된다.
LE (Less or Equal) NG (Not Greater)	ZF = 1 OR (OF = 0 AND SF = 1) OR (OF = 1 AND SF = 0)	X <= Y	위와 비슷하다. 다만 X = Y인 경우를 만족하기 위해 ZF = 1을 추가하였다.

[그림 12] Signed 연산을 위한 조건부 코드

참고로 X는 오퍼랜드 비교 연산 과정에서 사용된 첫번째 오퍼랜드를 말하고, Y는 두번째 오퍼랜드를 말한다.

Unsigned 연산을 위한 조건부 코드

조건부 코드	진리값 확인	조건	설명
A (Above) NBE (Not Below or Equal)	CF = 0 AND ZF = 0	X > Y	X > Y 경우에 참이 된다. 오퍼랜드 비교 연산 결과 값은 양수가 되어야 하므로 CF = 0과 ZF = 0이 되어야 참을 만족한다.
AE (Above or Equal) NB (Not Below) NC (Not Carry)	CF = 0	X >= Y	X >= Y 인 경우인데, 오버플로우만 발생하지 않는다면 참을 만족한다. (CF = 0 AND ZF = 0) OR (CF = 0 AND ZF = 1) 이므로 이는 CF = 0과 같다.
B (Below) NAE (Not Above or Equal) C (Carry)	CF = 1	X < Y	X < Y 인 경우에 참을 만족한다. 오퍼랜드 비교 연산 결과는 음수가 나오는데, Unsigned 오퍼랜드는 음수를 저장할 수 없기 때문에 오버플로우로 간주한다. 따라서 오버플로우가 발생했다면 X < Y를 만족하는 경우로 생각할 수 있다.
BE (Below or Equal) NA (Not Above)	CF = 1 OR ZF = 1	X <= Y	X < Y 이거나 X = Y이면 참을 만족한다. 따라서 CF = 1 OR ZF = 1 이 된다.
E (Equal) Z (Zero)	ZF = 1	X = Y	X = Y이면 참을 만족한다. 따라서 ZF = 1을 확인한다.
NE (Not Equal) NZ (Not Zero)	ZF = 0	X != Y	X != Y이면 참을 만족한다. 따라서 ZF = 0을 확인한다.

[그림 13] Unsigned 연산을 위한 조건부 코드

프로그램의 제어 흐름(Program Control Flow)

프로그램의 제어 흐름은 조건과 조건부 분기 명령어에 의해 결정된다. 고급 언어에서 사용하는 제어 흐름 관련 문장이 어셈블리 명령어로는 어떻게 표현되는지 알아보는 것은 프로그램의 제어 흐름을 파악하는데 중요하다.

if 문(Single-Branch Conditionals)

어셈블리	고급 언어
mov eax, [SomeVariable] test eax, eax jnz AfterCondition (Reversed)	if (SomeVariable == 0)
call Function	Function();
AfterCondition: ...	...

[그림 14] if 문장에 대한 어셈블리 코드

test 명령어는 2개의 오퍼랜드에 대해 bitwise-AND 연산을 수행하여 그 상태 정보를 플래그 레지스터에 반영하는 명령어이다. 만약 eax 레지스터의 값이 0이라면, bitwise-AND 연산 결과값은 당연히 0이 되며, 따라서 제로 플래그인 ZF = 1로 설정된다. 이러한 test 명령어의 특징 때문에, test 명령어는 오퍼랜드에 사용된 레지스터의 값이 0인지 확인하는 데에 자주 사용된다. 따라서 test eax, eax 라는 문장은 eax 레지스터의 값이 0인지 아닌지를 판단하기 위해 사용된 문장이라 볼 수 있다. 여기서 사용된 jnz 명령어는 ZF = 0이면 특정

위치(AfterCondition)로 분기하는 명령어이며, ZF = 1이면 바로 아래 문장(call Function)을 수행한다. 여기서 눈여겨 보아야 할 점은, 고급 언어에서는 SomeVariable이 0인지를 비교하지만, 어셈블리 명령어에서는 0이 아님을 비교한다는 것이다. 이와 같이 고급 언어 비교 문장에서 사용되는 조건이 어셈블리 명령어에서 반전(Reversed)되는 이유는 무엇일까? 만약 조건을 반전시키지 않고 고급 언어에 사용된 문장과 동일하게 조건을 검사한다면 [그림 15]와 같은 어셈블리 코드가 구성될 것이다.

어셈블리
mov eax, [SomeVariable] test eax, eax jz CallFunction (not reversed)
AfterCondition: … jmp NextBlock
CallFunction: call Function
NextBlock: …

[그림 15] Not Reversed

조건에 대해 반전을 사용하지 않았기 때문에, 고급 언어에서 사용된 소스 코드와 대응하는 어셈블리 코드의 위치 순서가 일치하지 않는다. 반전을 사용하는 이유는 바로 반전을 사용하였을 경우에 무조건 분기 명령어(jmp NextBlock)가 추가 되기 때문이다. (물론 이러한 단순한 이유 이외에도 여러가지 이유가 있을 것이라 생각한다.)

if-else 문(Two-Way Conditionals)

어셈블리	고급 언어
cmp [SomeVariable], 7 jne ElseBlock (Reversed)	if (SomeVariable == 7)
call SomeFunction	SomeFunction();
jmp AfterConditionalBlock	else
ElseBlock: call SomeOtherFunction	SomeOtherFunction();
AfterConditionalBlock: …	…

[그림 16] if-else 문장

고급 언어에서 사용한 소스 코드에 대응하는 어셈블리 코드의 순서가 일치한다는 사실을 알고 있다면, [그림 16]의 어셈블리 코드도 쉽게 이해할 수 있을 것이라 생각한다. 여기서 살펴볼 부분은, 무조건 분기 명령어(jmp AfterConditionalBlock)가 사용된 부분이다. 만약 조건이 true를 만족하여 if 이하 블록을 실행하였다면, else 블록 내의 문장에 해당하는 어셈블리 코드 블록은 건너뛰어야 하기 때문에 무조건 분기 명령어를 사용하는 것이다.

if-else if(Multiple Alternative Conditionals)

어셈블리	고급 언어
cmp SomeVariable, 10 jae AlternateBlock (reversed)	if (SomeVariable < 10)
call SomeFunction jmp AfterIfBlock	SomeFunction();
AlternateBlock: cmp SomeVariable, 345 jne AfterIfBlock (reversed)	else if (SomeVariable == 345)
call SomeOtherFunction	SomeOtherFunction();
AfterIfBlock: …	…

[그림 17] if-else 문

논리 연산자(Logical Operators)

앞서 if, if-else, if-else if 문을 살펴 보았다. 그렇다면 다음과 같이 &&(AND)와 ||(OR)와 같은 논리 연산자가 사용될 경우에는 어떻게 어셈블리로 표현할 것인가?

```
if (Var1 == 100 && Var2 == 20)
        function();
...
```

[그림 18] AND 논리 연산자(&&)가 사용된 경우

어셈블리	고급 언어
cmp [var1], 100 jne AfterCondition (reversed)	if (var1 == 100 &&
cmp [var2], 20 jne AfterCondition (reversed)	var2 == 20)
call function	Function();
AfterCondition: ...	...

[그림 19] AND 논리 연산자에 대한 어셈블리 구현

AND 논리 연산자가 사용된 문장은 AND 논리 연산자에 사용된 조건이 모두 참이어야 참이 된다. 즉 조건이 하나라도 참이 되지 않으면 그 문장은 거짓이 된다. 위 [그림 19]에서 조건을 반전시킨 것에 대해서 한번 생각 해보기 바란다.

다음은 OR 논리 연산자(||)가 사용된 경우를 살펴보자.

```
if (Var1 == 100 || Var2 == 20)
        function();
...
```

[그림 20] OR 논리 연산자(||)가 사용된 경우

OR 논리 연산자가 사용된 문장의 경우, 조건이 하나라도 참이면 그 문장은 참이 된다.

어셈블리	고급 언어
cmp var1, 100 je ConditionalBlock (Not reversed)	if (var1 == 100 ||
cmp var2, 50 je ConditionalBlock (Not reversed)	var2 == 50)
jmp AfterConditonalBlock	
ConditionalBlock: call function	function();
AfterConditionalBlock: ...	...

[그림 21] OR 논리 연산자에 대한 어셈블리 구현

AND 문장의 경우 조건이 하나라도 거짓이면 조건부 블록을 빠져나와야 하지만, OR 문장의 경우 조건이 모두 거짓일 경우에 조건부 블록을 빠져 나온다. 즉 다시 말하면 OR 문장을 구현하기 위해서는 조건이 하나라도 참인 경우에는 조건부 블록을 실행하는 것이다.

따라서 조건을 반전시키지 않고, 고급 언어에서 사용한 조건을 그대로 사용하며, 모든 조건이 거짓일 경우에 무조건 분기 명령어를 통해 조건부 블록을 빠져 나오도록 하고 있다. 따라서 어셈블리 코드를 분석할 때, OR 문장이 사용되었다는 것을 확인하는 방법은 모든 분기 명령어가 똑같은 조건부 블록을 가리키고, 조건부 블록 바로 이전에 무조건 분기 명령어가 조건부 블록을 건너뛰는 것을 확인하는 것이다.

지금까지 살펴본 OR 구현 방식은 GCC 컴파일러에서 사용하는 방식이다. 이 방식은 코드를 해석하는 입장에서는 읽기 편하고 직관적이지만, 성능상의 단점이 존재하는데, 그것은 바로 OR 조건이 모두 거짓이었을 때 실행되는 무조건 분기 명령어(JMP) 때문이다. 이러한 무조건 분기 명령어는 조건 반전을 통해 제거할 수 있으며, 이러한 방식을 마이크로소프트 컴파일러에서는 사용하고 있다.

어셈블리	고급 언어
cmp var1, 100 je ConditionalBlock (Not reversed)	if (var1 == 100 ||
cmp var2, 50 jne AfterConditonalBlock (reversed)	var2 == 50)
ConditionalBlock: call function	function();
AfterConditionalBlock: …	…

[그림 22] 무조건 분기 명령어가 제거된 OR 구현 방법

Switch (n-way Conditional)

Switch 문장을 구현하는 방법에는 크게 2가지가 있다. 바로 테이블 방식과 트리 방식이다.

Switch 구현 (테이블 방식)

테이블 방식은 Switch 문장에서 사용된 각각의 코드 블록을 컴파일하고 각각의 코드 블록에 대한 주소를 포인터 테이블에 저장한다. 이렇게 포인터 테이블을 구성하고 나서, Switch 문장이 실행되면, Switch 문장에서 사용되는 오퍼랜드는 코드 블록 포인터를 저장하고 있는 테이블의 인덱스로 사용된다. 따라서 이러한 방식은 포인터 테이블을 위한 공간적인 요소는 부담이 되지만, 실행은 굉장히 빠르다. 오퍼랜드가 인덱스로 직접 사용되므로, 오퍼랜드 비교 과정이 불필요하기 때문이다.

포인터 테이블은 보통 스위치 블록을 포함하고 있는 함수 코드의 바로 뒷 부분에 위치하지만, 사용된 컴파일러에 따라 그렇지 않을 수도 있다. 만약에 함수 포인터 테이블이 코드 섹션의 중간쯤에 위치해 있다면, 이는 switch 를 위한 테이블일 가능성이 아주 높다.

```
movzx eax, BYTE PTR [ByteValue]
add eax, -1
cmp eax, 4
ja DefaultCase_Code
jmp DWORD PTR [PointerTableAdd + eax * 4]
```

[그림 23] Switch 구현(테이블 방식)

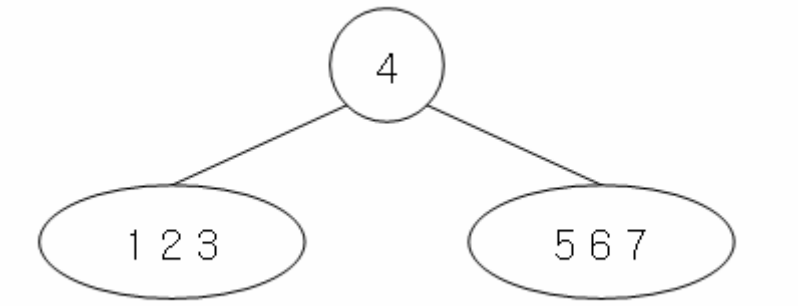
Switch 구현 (트리 방식)

Switch를 구현하기 위해 이진 트리 검색 알고리즘을 적용하는 방식이다. 이진 트리 검색을 위해서는 먼저 이진 트리를 구성하여야 한다. 이진 트리를 구성하는 방법은 간단한 예를 통해 알아보자. 먼저 다음과 같은 키의 집합 A가 존재한다고 가정하자.

1	2	3	4	5	6	7

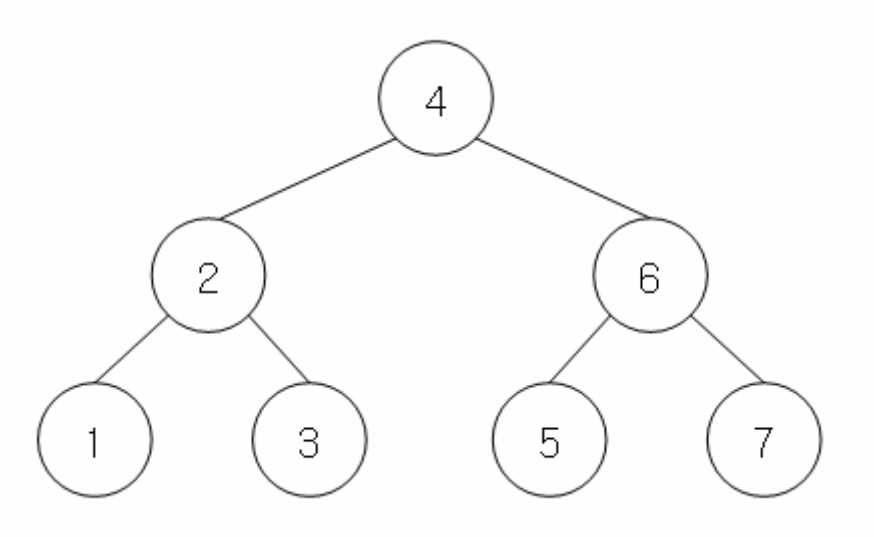
[그림 24] 집합 A

집합 A가 가진 원소 중에서 중간에 해당 하는 값을 취하고, 그 값보다 작은 원소의 집합과 큰 원소의 집합으로 나누게 된다.



[그림 25] 4를 기준으로 나눔

나뉘어진 1 2 3 하위 트리와 5 6 7 하위 트리에 대해서 다시, 스플릿(Split) 과정을 재귀적으로 수행하면, 최종적으로는 다음과 같은 이진 탐색 트리를 구성할 수 있다.



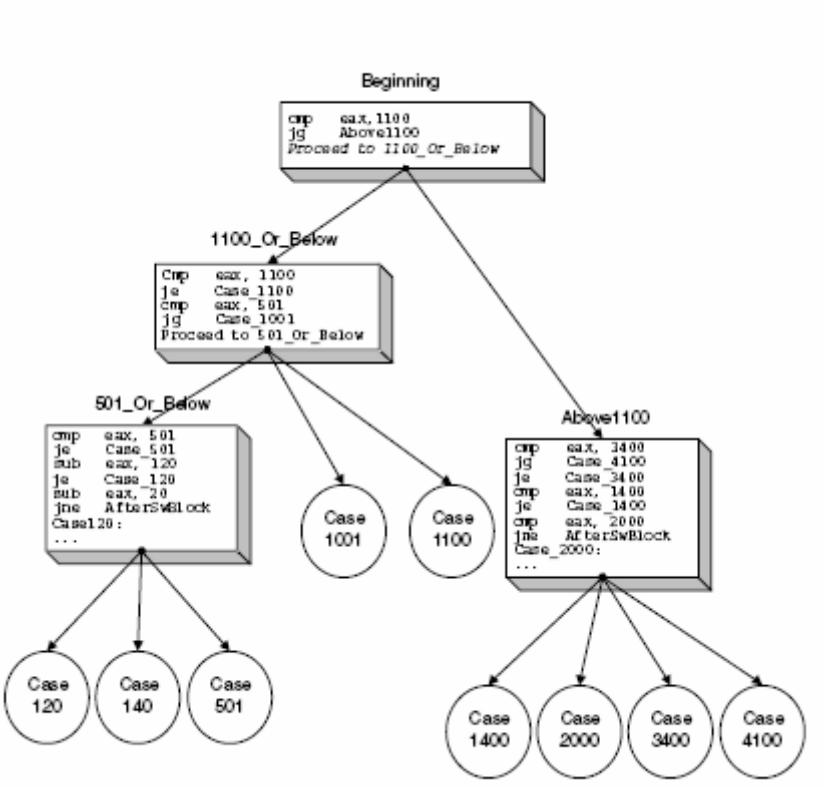
[그림 26] 이진 탐색 트리 구성

그렇다면 탐색 과정은 어떻게 이루어질까? 먼저 트리의 루트 노드와 검색값을 비교하여 같다면 검색을 종료하고, 같지 않았을 경우에 검색값이 루트 노드보다 작다면 루트 노드의 왼쪽 하위 트리를 검색하고, 검색값이 루트 노드보다 크다면 루트 노드의 오른쪽 하위 트리를 검색한다. 이러한 과정을 재귀적으로 수행한다.

```
void bintree_search(Tree root, int search) {
        if (root == null)
                // 찾지 못함. 검색 종료.
        if (root.value == search)
                // 찾았다!
        if (root.value < search)     bintree_search(root.rightSubtree, search);
        if (root.value > search)     bintree_search(root.leftSubtree, search);
}
```

[그림 27] 이진 탐색 트리의 탐색 과정에 대한 의사 코드

Switch 문은 이러한 이진 탐색 트리의 알고리즘에 기초하여, Switch 문에 사용된 각 조건 블록의 코드를 이진 탐색 트리로 구성한 후에, Switch 문에 사용된 오퍼랜드의 값을 기준으로 탐색 과정을 거치고 나서 최종적으로 조건 블록의 코드를 수행하게 된다.



[그림 28] Reversing Secrets of Reverse Engineering에서 발췌

while (Pretested Loops)

```
c = 0;
while (c < 1000)
{
        array[c] = c;
        c++;
}
```

[그림 29] while 문 사용 예제

While 문의 구현을 위해서는 분기 명령어가 2개가 필요한데, 첫번째는 while 문에서 사용하는 조건 분기 명령어이고, 두번째는 while 코드 블록의 실행이 끝나고 나서 되돌아가는 무조건 분기 명령어이다. 하지만 실제 컴파일러에서는 최적화를 위해 [그림 30]과 같이 posttested 루프 형태로 변경한다.

```
mov ecx, DWORD PTR [array]
xor eax, eax
LoopStart:
mov DWORD PTR [ecx + eax * 4], eax
add eax, 1
cmp eax, 1000
jl LoopStart
```

[그림 30] posttested 루프 형태

[그림 29]에서 c는 0으로 초기화되어 있기 때문에, 루프 초기에 c < 1000을 비교할 필요 없이 바로 루프 블록을 진입하는 것이 가능하기 때문에, 루프 초기에 위치해야 할 조건 분기 명령어를 루프 블록의 마지막으로 옮길 수 있게 되고, 무조건 분기 명령어를 제거할 수 있는 것이다. 만약 c의 값을 컴파일 과정에서 알 수 없다면, 지금과 같이 무조건 루프로 진입해서는 안되며 루프 진입 초기에 c의 값을 검사하는 과정을 포함해야 한다.

```
mov eax, DWORD PTR [c]
mov ecx, DWORD PTR [array]
cmp eax, 1000
jge EndOfLoop
LoopStart:
mov DWORD PTR [ecx + eax * 4], eax
add eax, 1
cmp eax, 1000
jl LoopStart
EndOfLoop:
```

[그림 31] 루프에 사용되는 조건을 예측할 수 없는 경우

Loop Break

```
do
{
        if (array[c]) break;
```

```
        array[c] = c;

        c++;

} while (c < 1000);
```

[그림 32] break 사용 예제

```
mov eax, DWORD PTR [c]

mov ecx, DWORD PTR [array]

LoopStart:

cmp DWORD PTR [ecx + eax * 4], 0

jne AfterLoop

mov DWORD PTR [ecx + eax * 4], eax

add eax, 1

cmp eax, 1000

jl LoopStart

AfterLoop:
```

[그림 33] 그림 22에 대한 어셈블리 코드

순수 산술 연산을 통한 Branchless Logic의 구현

컴파일러에서 최적화를 수행할 때, 조건 분기 명령어의 수를 가능한 줄이기 위해 조건 분기 명령어를 대체할 수 있는 다른 방법이 있다면 그것으로 대체하고 있다. 이러한 대체 방법 중에서 산술 연산을 통해 구현할 수 있는 방법을 알아보자. 우선 다음 코드는 어떤 의미를 나타내고 있을까? 해석이 가능한가?

```
mov eax, [ebp – 10]
and eax, 0x00001000
neg eax
sbb eax, eax
neg eax
ret
```

[그림 34] 순수 산술 연산을 통해 Branchless Logic을 구현한 코드

어셈블리 코드를 분석하다 보면, [그림 34]와 비슷한 코드를 종종 보게 된다. 이러한 코드는 실제 if 문과 같은 조건 분기 코드로부터 변환된 것인데, 어떻게 코드로 변환된 것일까? 우선 어셈블리 라인을 하나하나 분석해보자. 먼저 지역 변수를 0x1000으로 and 연산하고 나서, neg 명령어를 통해 부정(negation) 연산을 수행한다. 부정 연산이란 오퍼랜드의 부호를 바꾸는 것이다. 이는 단순히 최상위 비트를 스위치하는 것이 아니라, 2의 보수 과정을 수행한다. 수학적으로는 다음과 같은 연산을 수행하는 것이다.

Result = -(Operand)

다음 살펴보아야 할 명령어가 sbb 명령어인데, sbb 명령어는 다음과 같은 연산을 수행한다.

Operand1 = Operand1 – (Operand2 + CF)

예제에서 sbb의 Operand1과 Operand2에 동일한 eax 레지스터가 사용되었기 때문에, Operand1의 결과값은 CF에 의해 결정된다. CF = 1이면 eax는 -1이 될 것이고, CF = 0이면 eax는 0이 될 것이다. 그렇다면 CF는 어디서 설정하는 것일까? 바로 neg 명령어인데 neg 명령어는 명령어에 사용된 오퍼랜드의 부호가 바뀌면 CF를 1로 설정하고 부호가 바뀌지 않으면 CF를 0으로 설정하도록 하고 있다. 따라서 neg 명령어에 사용된 eax 레지스터의 값이 0이면 CF는 0이 될 것이고, eax 레지스터의 값이 0이 아니라면 CF는 1로 설정되는 것이다. 이와 같이 neg 명령어는 오퍼랜드의 값이 0인지 아닌지를 판단하는 데 사용될 수

있다. 이를 종합해 보면 다음 [그림 35]와 같은 고급 언어로 작성된 코드를 생각해 볼 수 있다.

```
EAX = EAX & 0x1000
if (EAX)
        CF = 1
else
        CF = 0
EAX = EAX – (EAX + CF)
EAX = -(EAX)
return EAX
```

[그림 35] 그림 34로부터 추출한 소스 코드

EAX가 0x1000을 가졌다면, 최종 EAX 값은 1이 되며 그렇지 않으면 0을 가진다. 따라서 다음과 같은 소스 코드를 유추해 볼 수 있다.

```
if (Variable == 0x1000)
        return TRUE;
else
        return FALSE;
```

[그림 36] 그림 35로부터 유추한 최종 소스 코드

소스 코드에서 if-else 문이 사용되었기 때문에, 어셈블리 코드에서 이를 구현하기 위해 조건 분기 명령어가 필요하겠지만, 컴파일 최적화 과정을 통해 산술 연산으로 대체할 수

있는 방법을 소개하였다.

SETcc(Set Byte On Condition)를 통한 Branchless Logic의 구현

SETcc 명령어 셋은 조건부 분기 명령어인 Jcc 명령어 셋과 같이 플래그에 대해 논리 테스트를 수행하는 것은 동일하지만, Jcc 처럼 분기를 수행하지는 않고, 논리 테스트 결과를 오퍼랜드에 저장하는 명령어이다.

```
return result;
```

만약 Branchless Logic을 적용하지 않는다면, 위의 코드는 다음 [그림 38]과 같은 어셈블리 코드로 표현할 수 있다.

```
cmp [result], 0
jne NotEquals
mov eax, 0
ret
NotEquals:
mov eax, 1
ret
```

[그림 37] Branchless Logic을 적용 안 한 경우

SETcc 계열의 명령어를 사용하면, 위의 코드를 Brancless Logic으로 구현할 수 있다.

```
xor eax, eax
cmp [result], 0
setne al
ret
```

[그림 38] SETcc 명령어를 통해 Branchless Logic을 구현

Branchless Logic을 위해 IA-32 프로세서에서는 SETcc 명령어 셋 외에도, CMOVcc 명령어 셋을 제공하고 있다.