Hanbit eBook

Realtime 72

Thinking About CPU 최적화 프로그래밍 노트

심화편

김안석 지음

Thinking About

CPU 최적화 프로그래밍 노트

심화편

Thinking About CPU 최적화 프로그래밍 노트 심화편

초판발행 2014년 6월 30일

지은이 김안석 / **펴낸이** 김태헌
펴낸곳 한빛미디어(주) / **주소** 서울시 마포구 양화로 7길 83 한빛미디어(주) IT출판부
전화 02-325-5544 / **팩스** 02-336-7124
등록 1999년 6월 24일 제10-1779호
ISBN 978-89-6848-667-8 15000 / **정가** 9,900원

책임편집 배용석 / **기획** 김병희, 이중민 / **편집** 안선화
디자인 표지 여동일, 내지 스튜디오 [밈], 조판 김현미
영업 김형진, 김진불, 조유미 / **마케팅** 박상용, 서은옥, 김옥현

이 책에 대한 의견이나 오탈자 및 잘못된 내용에 대한 수정 정보는 한빛미디어(주)의 홈페이지나 아래 이메일로 알려주십시오.
한빛미디어 홈페이지 www.hanbit.co.kr / **이메일** ask@hanbit.co.kr

Published by HANBIT Media, Inc. Printed in Korea
Copyright © 2014 김안석 & HANBIT Media, Inc.
이 책의 저작권은 김안석과 한빛미디어(주)에 있습니다.
저작권법에 의해 보호를 받는 저작물이므로 무단 복제 및 무단 전재를 금합니다.

지금 하지 않으면 할 수 없는 일이 있습니다.
책으로 펴내고 싶은 아이디어나 원고를 메일(ebookwriter@hanbit.co.kr)로 보내주세요.
한빛미디어(주)는 여러분의 소중한 경험과 지식을 기다리고 있습니다.

저자 소개

저자 **김안석**

1990년대 초반 하이텔과 나우누리의 C/C++ 동호회들을 통해 프로그래밍에 입문하여 나우누리 Cezips 동호회 운영진으로 활동했다. 1998년 초반부터 PC 및 콘솔 게임 SW 개발 분야의 국내 IT 업계와 게임 업계에서 일하기 시작하여 현재에 이르기까지 현업 SW 개발자로 종사하고 있다. 최근에는 게임 SW 개발에 주력하고 있으며, uniFlow (삼성SDS:1999), FormXPress ACUBE(삼성SDS:2001)와 같은 일반 IT 프로젝트들 및 Kingdom Under Fire: The Cursaders(블루사이드/Xbox 2004), Ninety-Nine Nights(블루사이드/Xbox360 2006) 등의 게임 개발 프로젝트에 주로 참여했다. 여전히 콘솔 게임 개발에 매력을 느끼는 골수 게임 개발자로, 현재 UTPlus Interactive Inc.에서 스마트폰 게임을 개발 중이다.

저자 서문

'로우-레벨 프로그래밍'은 프로그래머로서의 내 삶에서 사실 애증의 대상과도 같다. 이 애증 관계가 시작된 건 프로그래밍에 대해 잘 알지도 못하던 때였다. BASIC이라는 언어를 이용해 프로그래밍 잡지에 실린 게임 코드를 따라 쳐보는 수준이었던 필자가, 어디선가 프로그램의 속도를 끌어올리려면 기계어를 알아야 한다는 말을 듣고서 시작했던 게 계기였다. 학창 시절엔 DOS게임의 그래픽 라이브러리 제작을 이유로, 사회생활을 하면서는 제한된 시스템상에서 결과물의 요구 조건에 맞추기 위해 때로는 최적화라는 다른 말로 늘 프로그래밍하는 내 옆자리에 있어 왔다. 그리고 지금도 여전히 개발자인 필자를 즐겁게 혹은 힘들게 하는 말로 따라다닌다.

이 책 또한 처음부터 로우-레벨 프로그래밍을 소개하는 책을 펴내고자 집필을 시작했던 것은 아니었다. 프로그래밍을 더 깊이 있게 공부하기 시작하면서 필자는 로우-레벨 프로그래밍에 대해 정리를 해보고 싶었고, 그래서 개인 블로그에 틈틈이 정리해 올리던 것이 시작이었다. 그 글을 꾸준히 보아온 주변 지인들의 격려와 도움으로 이렇게 내 이름을 건 책을 낼 수 있게 되었다.

이런 기회가 한편으로는 설레면서도 다른 한편으로는 사실 두렵기도 하다. 이 책은 개발자를 위한 여타 책들과는 달리 특정 메이커 하드웨어에 열광하는 팬덤과도 같은 성격을 띠고 있어서, 책을 통해 저자가 전달하고자 하는 정보나 지식들이 어느 한쪽으로 편향될까 걱정이 되는 것이다. 하지만 프로그래밍을 사랑하는 독자들이라면, 개발 서적 중엔 이런 책도 있고 이런 저자도 있음을 넓은 마음으로 바라봐주지 않을까. 그런 희망적인 기대를 하며 지금부터 필자가 해보고 싶었던 이야기를 독자들과 나눠볼까 한다.

대상 독자 및 예제 파일

초급	초중급	**중급**	중고급	고급

이 책은 특정 메이커 하드웨어나 컴파일러를 기반으로 이야기를 전개한다. 때문에 일반적으로 C/C++ 중급 이상 개발자, PC 플랫폼상에서 개발 중인 개발자, 인텔 계열 CPU를 사용하고 있는 프로그래머와 같은, 특정 그룹을 직접적인 대상으로 한다.

그러나 굳이 위와 같은 협소한 특정 그룹이 아니더라도, 평소 CPU와 로우-레벨 프로그래밍에 대해서 궁금해하거나 자신이 작성한 프로그램이 CPU상에서 어떻게 동작하는지, 어떻게 하면 프로그램의 성능이나 안전성을 개선할 수 있는지 궁금한 학생들 및 초보 개발자들에게도 제법 읽을만한 내용이 되리라 본다.

이 책에서 다루는 내용을 잘 모른다고 해서 프로그래밍을 하는 데 문제가 생기거나 하진 않는다. 단지, 등에 좀 가려운 부분이 있는데 그 부분을 그냥 두자니 신경이 쓰이고 가려워서 그 부분을 조금 긁어준다는 느낌으로 이 책을 보면 되겠다.

또한 필자는 이 책의 내용들을 반드시 따라야 좋다고 강요할 생각이 없으며, 그런 내용도 아니다. 어느 하나가 좋다고 해서 그것만 계속 취하고 골몰하다 보면 반대로 더 안 좋을 수도 있듯, 하드웨어와 시스템에 관한 지식을 고루 취하고자 전체적으로 알아가기 위한 기본 단계라고 봐주기 바란다. 정리하자면 이 책은 하드웨어와 그 위에 동작하는 소프트웨어의 내부를 소개하는 입문서라 볼 수 있겠다.

마지막으로, 익숙하지 않은 책의 주제와 용어, 내용들로 인해 미리 두려움을 가질 필요도 없다고 말하고 싶다. 그보다는 현업 개발자가 프로그램 개발을 하며 느낀 내용들을, 동료 개발자들에게 전해주기 위해 쓴 가볍게 읽어볼 만한 서적이라 봐주길 바란다.

INTRO

이 책에서 다룰 '프로그래밍 최적화'는 설명하기 무거운 주제다. 그래서 처음에는 가볍게 시작할 것이다. 필자가 프로그래밍 최적화에 접근했던 순서대로, 이야기하듯이 설명할 테니 부담 갖지 말고 따라오길 바란다.

필자 또한 IT분야에 입문하여 스스로 만든 프로그램의 성능이 낮고 비효율적일 때, 어떻게 하면 그것을 향상할 수 있을지 고민했었다. 이러한 고민을 주변에 말하니 다들 "최적화하려면 어셈블리어를 해야 한다", "최적화해 주는 전용 프로그램을 이용해서 코드를 변환해야 한다"라고 조언해주었다.

하지만 최적화 프로그램은 가격이 비싸서 개인적으로 구입하기는 무리였다. 그래서 막연히 어셈블리어로 접근하면 더 나을 것이라는 생각으로 어셈블리어를 공부하던 시절이 있었다. 프로그래머로서 다양한 프로젝트를 경험해보고 부족한 부분을 공부하다 보니, 어느 한쪽으로 뻗어 있는 길만이 무조건 옳은 길이라고 할 수 없다는 걸 알게 되었다.

개발하는 프로그래밍의 성능을 높이는 데 어셈블리어가 실제로 도움이 되었으며 때로는 언어적인 문제가 아니라 사용하는 알고리즘에 문제가 생겨서 이를 개선하는 것만으로도 성능을 충분히 높일 수 있었다. 그런가 하면, 맹신했던 어셈블리어를 이용한 프로그래밍을 할 때나 어디선가 주워듣고서 많은 이가 좋다고 한 알고리즘을 사용했다가 그것이 프로그램의 성능에 악영향을 끼치는 반대의 경우도 있었다.

이런 경험들을 통해 필자는 어느 한쪽에만 골몰하거나 얕은 지식으로는 프로그램의 성능을 목표한 만큼 향상하는 데 한계가 있다는 걸 느꼈다. 제대로 알지도 못하고 사용하는 어설픈 기술은 도움이 아니라 오히려 독이 되기도 했다. 그래서 프로그램을 최적화하는 방법을 제대로 알고 접근하고자, 프로그래머가 다룰 수 있는 하드웨어에 관해서 공부하며 하나씩 밟아왔던 길을 되짚어보려고 한다.

한빛 eBook 리얼타임

한빛 eBook 리얼타임은 IT 개발자를 위한 eBook입니다.

요즘 IT 업계에는 하루가 멀다 하고 수많은 기술이 나타나고 사라져 갑니다. 인터넷을 아무리 뒤져도 조금이나마 정리된 정보를 찾는 것도 쉽지 않습니다. 또한 잘 정리되어 책으로 나오기까지는 오랜 시간이 걸립니다. 어떻게 하면 조금이라도 더 유용한 정보를 빠르게 얻을 수 있을까요? 어떻게 하면 남보다 조금 더 빨리 경험하고 습득한 지식을 공유하고 발전시켜 나갈 수 있을까요? 세상에는 수많은 종이책이 있습니다. 그리고 그 종이책을 그대로 옮긴 전자책도 많습니다. 전자책에는 전자책에 적합한 콘텐츠와 전자책의 특성을 살린 형식이 있다고 생각합니다.

한빛이 지금 생각하고 추구하는, 개발자를 위한 리얼타임 전자책은 이렇습니다.

1. eBook Only - 빠르게 변화하는 IT 기술에 대해 핵심적인 정보를 신속하게 제공합니다.

500페이지 가까운 분량의 잘 정리된 도서(종이책)가 아니라, 핵심적인 내용을 빠르게 전달하기 위해 조금은 거칠지만 100페이지 내외의 전자책 전용으로 개발한 서비스입니다. 독자에게는 새로운 정보를 빨리 얻을 수 있는 기회가 되고, 자신이 먼저 경험한 지식과 정보를 책으로 펴내고 싶지만 너무 바빠서 엄두를 못 내는 선배, 전문가, 고수 분에게는 보다 쉽게 집필할 수 있는 기회가 될 수 있으리라 생각합니다. 또한 새로운 정보와 지식을 빠르게 전달하기 위해 O'Reilly의 전자책 번역 서비스도 하고 있습니다.

2. 무료로 업데이트되는, 전자책 전용 서비스입니다.

종이책으로는 기술의 변화 속도를 따라잡기가 쉽지 않습니다. 책이 일정 분량 이상으로 집필되고 정리되어 나오는 동안 기술은 이미 변해 있습니다. 전자책으로 출간된 이후에도 버전 업을 통해 중요한 기술적 변화가 있거나 저자(역자)와 독자가 소통하면서 보완하여 발전된 노하우가 정리되면 구매하신 분께 무료로 업데이트해 드립니다.

3. 독자의 편의를 위하여 DRM-Free로 제공합니다.

구매한 전자책을 다양한 IT 기기에서 자유롭게 활용할 수 있도록 DRM-Free PDF 포맷으로 제공합니다. 이는 독자 여러분과 한빛이 생각하고 추구하는 전자책을 만들어 나가기 위해 독자 여러분이 언제 어디서 어떤 기기를 사용하더라도 편리하게 전자책을 볼 수 있도록 하기 위함입니다.

4. 전자책 환경을 고려한 최적의 형태와 디자인에 담고자 노력했습니다.

종이책을 그대로 옮겨 놓아 가독성이 떨어지고 읽기 힘든 전자책이 아니라, 전자책의 환경에 가능한 한 최적화하여 쾌적한 경험을 드리고자 합니다. 링크 등의 기능을 적극적으로 이용할 수 있음은 물론이고 글자 크기나 행간, 여백 등을 전자책에 가장 최적화된 형태로 새롭게 디자인하였습니다.

앞으로도 독자 여러분의 충고에 귀 기울이며 지속해서 발전시켜 나가도록 하겠습니다.

지금 보시는 전자책에 소유권한을 표시한 문구가 없거나 타인의 소유권한을 표시한 문구가 있다면 위법하게 사용하고 있을 가능성이 높습니다. 이 경우 저작권법에 의해 불이익을 받으실 수 있습니다.

다양한 기기에 사용할 수 있습니다. 또한 한빛미디어 사이트에서 구입하신 후에는 횟수에 관계없이 다운받으실 수 있습니다.

한빛미디어 전자책은 인쇄, 검색, 복사하여 붙이기가 가능합니다.

전자책은 오탈자 교정이나 내용의 수정·보완이 이뤄지면 업데이트 관련 공지를 이메일로 알려드리며, 구매하신 전자책의 수정본은 무료로 내려받으실 수 있습니다.

이런 특별한 권한은 한빛미디어 사이트에서 구입하신 독자에게만 제공되며, 다른 사람에게 양도나 이전은 허락되지 않습니다.

차례

Part 1. 64비트 환경에서의 최적화

01 **32비트 환경을 넘어 64비트 환경으로** 2

1.1 64비트 운영체제에서 로우 레벨 코드 접근하기 ... 3
1.2 내장 함수를 이용한 접근 ... 6
1.3 어셈블리어를 이용한 접근 ... 8
1.4 AVX 기술에 관한 소개 ... 25

02 **AVX 명령어 살펴보기** 34

2.1 AVX의 부동 소수점 처리 명령어 ... 35
2.2 AVX의 데이터 이동 명령어 ... 35
2.3 그 외 AVX 명령어들 ... 36
2.4 Half 부동 소수점 전환 명령어 ... 45

03 **64비트 모드 프로그래밍 가이드** 47

3.1 32비트 데이터 사용 시 구형 32비트 명령어 사용하기 ... 47
3.2 확장 레지스터 사용하여 레지스터 개수에 대한 스트레스 줄이기 ... 47
3.3 32비트 정수형 곱셈 시 64비트 레지스터 사용하기 ... 48
3.4 정수형과 실수형 간의 변환 시 SSE 명령 적극 사용하기 ... 51
3.5 그 외 나머지들... ... 52

04 **AVX 명령어를 이용한 예제** 53

Part 2. 심도 있는 최적화 가이드

05 **코드의 흐름 제어하기** 59

5.1 분기문 개수 줄이기 59
5.2 정적 분기 예측 알고리즘 사용하기 60
5.3 예측할 수 없는 분기 개선하기 60
5.4 루프 풀어 쓰기 61
5.5 그 밖에 코드의 흐름을 제어하는 방법 61

06 **명령어 실행에 관한 최적화** 63

6.1 명령어 선택하기 63
6.2 명령어 실행 멈춤 피하기 65
6.3 레지스터 참조 시 명령어 멈춤 피하기 67

07 **데이터 정렬하기** 70

7.1 메모리 정렬하기 70
7.2 prefetch 이용하기 70
7.3 캐시 통하지 않고 데이터 저장 및 읽기 71

08 **분기문을 대체할 수 있는 유용한 조건부 명령어** 73

8.1 CMOVcc 73
8.2 FCMOVcc 75
8.3 SETcc 75
8.4 LOOPcc 76
8.5 VMASKMOVxx 77

마치며... 80

Part 1.
64비트 환경에서의 최적화

이번 파트에서는 64비트 환경에서 프로그램의 성능을 높이기 위한 방법에 관해 이야기할 것이다. 먼저, 32비트 운영체제와 64비트 운영체제의 차이점을 알아보고, AVX 명령어의 사용 방법을 살펴볼 것이다. 그리고 64비트 환경에서 어떻게 해야 프로그램의 성능을 높일 수 있는지 알아본 다음, 마지막으로 AVX 명령어를 이용한 예제를 통해 배운 내용을 어떻게 활용할 수 있는지 짚어본다.

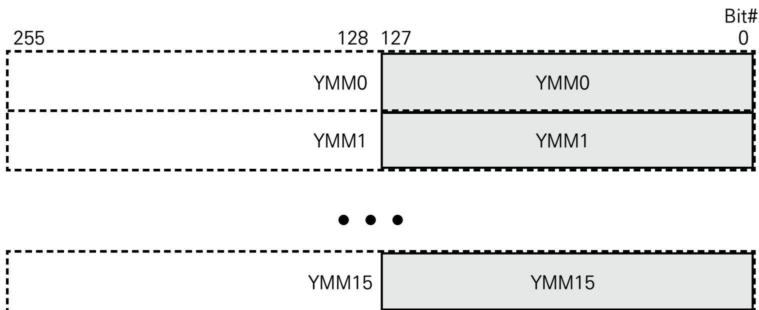
1장 32비트 환경을 넘어 64비트 환경으로
2장 AVX 명령어 살펴보기
3장 64비트 모드 프로그래밍 가이드
4장 AVX 명령어를 이용한 예제

1 | 32비트 환경을 넘어 64비트 환경으로

이번 장에서는 SSE 다음 세대의 명령어인 'AVX Advanced Vector Extension 명령어'를 소개한다. 지금까지는 32비트와 64비트 모두에서 쓸 수 있는 SSE 명령어로 예제와 명령어를 소개했는데, 지금부터 소개할 AVX 명령어는 주로 64비트에서 사용한다.

먼저, 최신 CPU에서 지원하는 256비트 데이터 제어에 대해 살펴보자. 그림 1-1을 보면 이전의 SSE용 데이터와 AVX용 데이터 타입의 차이를 쉽게 알 수 있다.

그림 1-1 AVX 명령어에서 사용되는 YMM 레지스터들



그림에서 보듯 XMM 외에 YMM이라는 새로운 레지스터가 있다. 기존 XMM 레지스터의 개수가 8개였다면 AVX 기술로 넘어오면서 그 수는 8개 확장되어 16개가 되었으며, YMM 레지스터 역시 16개 사용할 수 있다.

또한, YMM 레지스터는 기존 레지스터와 달리 독립되어 있지 않고, XMM 레지스터의 상위 128비트 영역으로 확장되었다. 즉, 동일한 인덱스의 XMM 레지스터에 어떤 값을 기록하면 YMM 레지스터의 하위 128비트 영역과 그 값을 공유한다.

이제, 본격적으로 YMM 레지스터를 사용하는 샘플 코드를 작성해보자.

1.1 64비트 운영체제에서 로우 레벨 코드 접근하기

AVX 프로그래밍을 하기 전에는 본인이 사용하는 CPU가 무엇인지, 그리고 운영체계가 이러한 확장 레지스터를 지원하는지 알아야 한다. 필자는 다음과 같은 코드로 CPU 타입을 읽은 다음 YMM 레지스터를 사용할 수 있는지 체크해보았다.

예제 1-1 CPU 매뉴얼 토대로 만든 AVX 기능 체크 함수

```
int supprts_AVX()
{
     _asm{
         mov eax, 1
         cpuid
         and ecx, 018000000H
         cmp ecx, 018000000H       /// ; OS가 XGETBV, XSAVE, XRSTOR 등 확장
                                   ///   명령어를 사용가능한지 체크한다.

         jne not_supported         /// ; 플래그 세팅이 없다면, 사용 못 함.
         mov ecx, 0                /// ; XCR0 register 리셋
         XGETBV                    /// ; Externded Control Register의 XCR0 인덱스
                                   ///   값을 EDX:EAX에 넣음
         and eax, 06H
         cmp eax, 06H              /// ; OS가 XMM과 YMM 지원할 수 있는지 체크
         jne not_supported

         mov eax, 1                /// ; 여기까지 오면 AVX 기능 사용 가능.
         jmp done

not_supported:
         mov eax, 0

done:
     }
}
```

위 함수를 32비트나 64비트 운영체제에서 Visual Studio로 실행하면, 아쉽게도 둘 다 에러를 잔뜩 내뱉기만 할 뿐, 컴파일되지 않는다. 당연한 결과다. 실행되지 않는 데엔 다음 2가지 이유가 있다.

먼저, AVX 명령어 매뉴얼에 따르면 AVX 기능은 64비트 운영체제에서만 사용할 수 있다. 그러니 Visual Studio가 32비트 형태로 동작할 경우, 위 소스코드를 실행하기 위해서는 컴파일 환경을 64비트 환경으로 바꿔줘야 한다. 이에 관한 자세한 내용은 MSDN의 내용[01]을 참조하길 바란다.

MSDN에 나와 있는 대로 x64 플랫폼으로 변경한 후 컴파일해보자.

그림 1-2 MSDN에서 소개하는 대로 대상 플랫폼을 x64로 변경한다.

이렇게 변경해도 컴파일되지 않고 그림 1-3의 에러코드를 보여줄 것이다. 뭔가 하나 해보기가 정말 쉽지 않다.

그림 1-3 asm 키워드 사용 시 볼 수 있는 에러코드

```
error C4235: 비표준 확장 사용 : 이 아키텍처에서는 '_asm' 키워드를 사용할 수 없습니다.
error C2065: 'mov' : 선언되지 않은 식별자입니다.
error C2146: 구문 오류 : ';'이(가) 'eax' 식별자 앞에 없습니다.
```

그림 1-3에서 보듯 64비트 모드에서는 __asm 키워드를 사용할 수 없다고 나온다. 그래서 MSDN을 살펴보았더니 다음과 같은 구문을 찾을 수 있었다.

01 http://msdn.microsoft.com/ko-KR/library/9yb4317s.aspx

그림 1-4 x64에서는 인라인 어셈블리를 못 쓴다는 안내문

내장 및 인라인 어셈블리

Visual Studio 2013 | 다른 버전 ▾ | 이 항목은 아직 평가되지 않았습니다.- 이 항목 평가

x64 컴파일러의 제약 조건 중 하나는 인라인 어셈블러를 지원하지 않는다는 것입니다. 즉, C 또는 C++로 작성할 수 없는
함수는 컴파일러에서 지원하는 내장 함수나 서브루틴으로 작성해야 합니다. 함수에 따라서는 성능에 큰 영향을 미칠 수
있습니다. 성능에 중요한 부분을 차지하는 함수는 내장 함수로 구현해야 합니다.

보는 바와 같이 사용할 수 없다. '아~ 그렇구나'하고 끝내기에는 그 뒤에 있는 문구가 호기심을 자극한다. 위의 구문을 자세히 보면 '내장 함수나 서브루틴으로 작성해야 합니다'라는 설명이 있다. 즉, 인라인 어셈블리어를 사용할 수 없을 뿐 지금까지 설명했던 내장 함수나 어셈블리어로 따로 만든 서브루틴은 사용할 수 있다는 뜻이다.

표 1-1 MSDN을 살펴보면 x64에서 사용할 수 있는 내장 함수가 잘 정리되어 있다.

기본 이름	함수 프로토타입
__addgsbyte	void __addgsbyte (unsigned long, unsigned char)
__addgsdword	void __addgsdword (unsigned long, unsigned long)
__addgsqword	void __addgsqword (unsigned long, unsigned _int64)
__addgsword	void __addgsword (unsigned long, unsigned short)
__code_seg	void __code_seg (const char *)
__cpuid	void __cpuid (int [4], int)
__cpuidex	void __cpuidex (int [4], int, int)
__debugbreak	void __cdecl _debugbreak(void)

MSDN의 64비트 모드 애플리케이션 개발에 관한 내용을 보면, 표 1-1과 같은 내장 함수 리스트를 볼 수 있다. 필자는 이를 토대로 내장 함수를 이용한 AVX 기능

체크 함수를 만들어보았다.

1.2 내장 함수를 이용한 접근

다음은 내장 함수를 이용해서 만든 AVX 기능 체크 함수다.

예제 1-2 내장 함수를 이용해 만든 AVX 체크 함수

```
int supports_AVX_ByInterinsic()
{
    _int32 reg[4] = { -1 };
    _int64 reg = 0;

    _cpuid(reg, 1);   /// eax에 1을 넣고, 레지스터 읽어옴

    if ((reg[2] & 0x18000000) != 0x18000000)   /// ecx 레지스터 값 체크
        return 0;

    ret = _xgetbv(0);   /// eax에 0을 넣고, xgetbv 명령 호출
    if ((ret & 0x06) != 0x06)
        return 0;
    return 1;
}
```

Visual C에는 예제 1-2에서 사용한 함수가 다 나와 있었다. 참고로 한글 MSDN에는 최신 AVX에 관한 명령어 리스트가 있는데, 함수별 설명이 없어서 이 부분은 CPU 레퍼런스 가이드의 내용을 참고해 작성했다. 다만, 어떤 것은 설명이 명확하지 않아서 임의의 값으로 함수를 직접 호출한 뒤, 디스어셈블리로 어떤 레지스터를 사용하는지 확인한 후 재작성했다.

그림 1-5 뭔지 모를 것들에 대해 하나하나 더듬어 가며 체크하는 과정

```
    int reg[4] = { -1 };
000000013F1F2ED5  mov          dword ptr [rsp+28h],0FFFFFFFFh
000000013F1F2EDD  lea          rax,[rsp+2Ch]
000000013F1F2EE2  mov          rdi,rax
000000013F1F2EE5  xor          eax,eax
000000013F1F2EE7  mov          ecx,0Ch
000000013F1F2EEC  rep stos     byte ptr [rdi]
    __cpuid( reg, 1 );
000000013F1F2EEE  mov          eax,1
000000013F1F2EF3  xor          ecx,ecx
000000013F1F2EF5  cpuid
000000013F1F2EF7  lea          rdi,[rsp+28h]
000000013F1F2EFC  mov          dword ptr [rdi],eax
000000013F1F2EFE  mov          dword ptr [rdi+4],ebx
000000013F1F2F01  mov          dword ptr [rdi+8],ecx
000000013F1F2F04  mov          dword ptr [rdi+0Ch],edx
```

이제 에러도 없고 동작도 잘 한다. 디버그 모드로 한 라인씩 실행시키면 레지스터 창을 통해 함수 안의 내용이 동작하면서 레지스터에 여러 값이 왕래하는 것을 볼 수 있다. X64 환경에서 레지스터 창을 보면 그림 1-6처럼 좀 더 새로운 세계를 볼 수 있다.

그림 1-6 x64 플랫폼상에서의 레지스터 모습

```
레지스터
 RAX = 000000000FC9F4B0 RBX = 0000000000000000 RCX = 0000000000000001 RDX = 00000000003C2580
 RSI = 0000000000000000 RDI = 0000000000000000 R8  = 00000000003C26B0 R9  = 00000000000000FE
 R10 = 0000000000100000 R11 = 000000000013F408 R12 = 0000000000000000 R13 = 0000000000000000
 R14 = 0000000000000000 R15 = 0000000000000000 RIP = 000000013F7E101A RSP = 000000000013FAD0
 RBP = 0000000000000000 EFL = 00000206

 YMM0 = 5000006300000000003C00000000000000000000000000000000000000000000
 YMM1 = 0000000100000000003C00000000000000000000000000000000000000000000
 YMM2 = 001F66E000000000000000000000000000000000000000000000000000000000
 YMM3 = 3707013100000000000000000000000000000000000000000000000000000000
 YMM4 = 77310000000000000001F00000000000000000000000000000000000000000000
 YMM5 = 001F66E000000000000000000000000000000000000000000000000000000000
 YMM6 = 5000006300000000001F66010000000000000000000000000000000000000000
 YMM7 = 001F66D00000000000001F00000000000000000000000000000000000000000000
 YMM8 = 77379C3800000000001F00000000000000000000000000000000000000000000
 YMM9 = 0000000400000000000000000000000000000000000000000000000000000000
 YMM10 = 0000000000000000000000000000000000000000000000000000000000000000
 YMM11 = 0000000000000000000001010000000000000000000000000000000000000000
 YMM12 = 001F66E000000000000000000000000000000000000000000000000000000000
 YMM13 = 001F66E000000000000000000000000000000000000000000000000000000000
```

그동안 익숙히 보아왔던 레지스터와 이름이 다르다는 걸 분명하게 확인할 수 있다.

'R' 접두사가 붙은 레지스터는 기존의 'E' 접두사가 붙은 CPU의 일반 레지스터가 32비트에서 64비트로 확장된 형태라 보면 된다.

표 1-2 x86과 x64에서 사용되는 레지스터의 이름들

Register Type	Without REX	With REX
Byte Registers	AL, BL, CL, DL, AH, BH CH, DH	AL, BL, CL, DL, DIL, SIL, BPL, SPL, R8L, R15L
Word Registers	AX, BX, CX, DX, DI, SI, BP, SP	AX, BX, CX, DX, DI, SI, BP, SP, R8W – R15W
Doubleword Registers	EAX, EBS, ECX, EDX, EDI, ESI, EBP, ESP	EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, R8D – R15D
Quadword Registers	N.A.	RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8 – R15

표 1-2는 MSDN의 표를 갈무리한 것이다. x86와 x64상 범용 레지스터의 이름이 나와 있으며, 우측이 x64상에서 사용할 수 있는 레지스터의 이름이다.

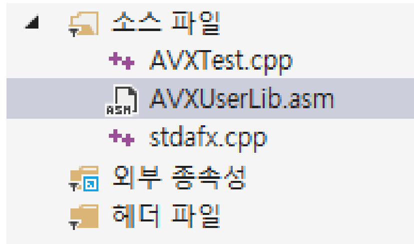
다시 함수로 돌아와, 디스어셈블리 창에서 코드를 보면 내장 함수를 사용하여 만든 어셈블리어 명령어가 잘 연결되어 있음을 확인할 수 있다. 특정 함수에선 불필요한 동작이 몇몇 눈에 띄긴 하지만 결과를 얻는 데에는 차이가 없다.

1.3 어셈블리어를 이용한 접근

두 번째로 살펴볼 방법은 어셈블리어로 '.asm' 코드를 직접 작성하여 Visual Studio에서 제공하는 MASM 어셈블리로 오브젝트 파일을 만든 후, 내부의 사용자 함수를 호출해서 같은 기능을 수행하는 방법이다.

이를 위해서는 일단 asm 파일을 하나 만들어 프로젝트 파일에 넣어야 한다.

그림 1-7 빈 asm 파일을 하나 만들어서 추가한다.



이제 빈 파일을 열어서 코드를 작성해보자. 인라인 어셈블리 코드를 그대로 옮겨왔다.

예제 1-3 어셈블리어로 구성한 AVX 기능 체크 함수

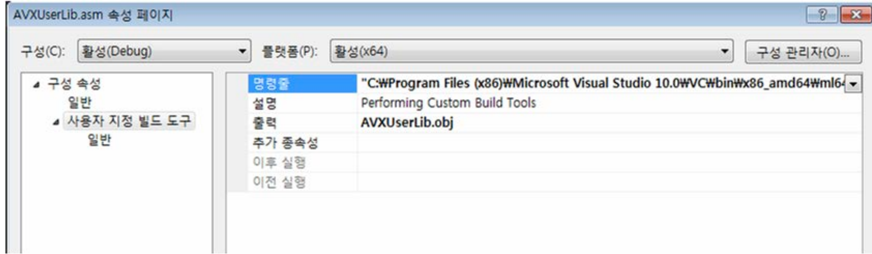
```
;.686P
;.modle flat, stdcall
;option casemap:none

.code

supports_AVX Proc

mov eax, 1
     cpuid
     and ecx, 018000000H
     cmp ecx, 018000000H    ; OS가 XGETBV, XSAVE, XRSTOR 등 확장 명령어를 사용
                              가능한지 체크한다.
     jne not_supported      ; 플래그 세팅이 없다면, 사용 못 함.

     mov ecx, 0             ; XCR0 register 리셋
     XGETBV                 ; Externded Control Register의 XCR0 인덱스 있는 값을
                              EDX:EAX에 넣음
```

```
    and eax, 06H
    cmp eax, 06H           ; OS가 XMM과 YMM 지원할 수 있는지 체크
    jne not_supported

    mov eax, 1             ; 여기까지 오면 AVX 기능 사용 가능.
    jmp done

not_supported:
    mov eax, 0

done:
    ret
support_AVX Endp
End
```

여기에서는 MASM 프로시저 관련 키워드 소개를 생략한다. 자세한 내용은 MASM 책자나 MSDN을 참고하길 바란다(구성을 고정 형태로 사용해도 무방하다). 향후에 소개할 함수도 위와 같은 구조의 프로시저를 이용하여 만들 것이니, AVX 기능 구현에만 관심 있는 독자라면 위의 구조를 그대로 사용해도 괜찮다.

일단, 위와 같이 어셈블리어로 코드 파일을 만든다. 그러고서 이를 컴파일하려면 Visual C의 솔루션 탐색기에서 파일을 선택한 후, 그림 1-8과 같이 [속성] 기능에서 [사용자 지정 빌드 도구]를 설정한다.

그림 1-8 속성 페이지에서 사용자 지정 빌드를 설정하는 모습



그리고 명령줄에 다음과 같이 기입한다.

> '"C:\Program Files (x86)\Microsoft Visual Studio 10.0\VC\bin\x86_amd64\ml64.exe" /c /Cx AVXUserLib.asm'.

'/c'나 '/Cx' 같은 약간 생소한 명령어 옵션이 있는데, 명령 프롬프트에서 "ml64.exe /?"하여 살펴보면 관련 옵션이 어떤 기능을 하는지 알 수 있다. 위 명령어에는 특별한 옵션이 없는데, 간단히 설명하자면 해당 파일에 대해 컴파일만 하고 내부에 있는 프로시저를 외부에서 호출하여 사용할 수 있도록 한다는 정도로 알고 있으면 된다.

그림 1-9 기존의 MASM과 별 차이가 없는 옵션들이다.

```
Copyright (C) Microsoft Corporation.  All rights reserved.

        ML [ /options ] filelist [ /link linkoptions ]

/Bl<linker> Use alternate linker          /safeseh Assert all exception
/c Assemble without linking                        handlers are declared
/Cp Preserve case of user identifiers     /Sf Generate first pass listing
/Cu Map all identifiers to upper case     /Sl<width> Set line width
/Cx Preserve case in publics, externs     /Sn Suppress symbol-table listing
/coff generate COFF format object file    /Sp<length> Set page length
/D<name>[=text] Define text macro         /Ss<string> Set subtitle
/EP Output preprocessed listing to stdout /St<string> Set title
/F <hex> Set stack size (bytes)           /Sx List false conditionals
/Fe<file> Name executable                 /Ta<file> Assemble non-.ASM file
/Fl[file] Generate listing                /w Same as /W0 /WX
/Fm[file] Generate map                    /WX Treat warnings as errors
/Fo<file> Name object file                /W<number> Set warning level
/Fr[file] Generate limited browser info   /X Ignore INCLUDE environment path
/FR[file] Generate full browser info      /Zd Add line number debug info
/G<c|d|z> Use Pascal, C, or Stdcall calls /Zf Make all symbols public
/I<name> Add include path                 /Zi Add symbolic debug info
/link <linker options and libraries>      /Zm Enable MASM 5.10 compatibility
/nologo Suppress copyright message        /Zp[n] Set structure alignment
/omf generate OMF format object file      /Zs Perform syntax check only
/Sa Maximize source listing
/errorReport:<option> Report internal assembler errors to Microsoft
    none - do not send report
    prompt - prompt to immediately send report
    queue - at next admin logon, prompt to send report
    send - send report automatically

C:₩Program Files (x86)₩Microsoft Visual Studio 10.0₩VC>
```

이제, 어셈블리어로 만든 코드를 컴파일하여 출력된 obj 파일의 사용자 함수를 호출해보자. 이는 어렵지 않은 작업이다. 일반적인 C 함수를 호출하듯 extern 키워드로 외부 함수를 선언해주면 곧바로 사용할 수 있다.

예제 1-4 외부 함수 선언 후 사용자 함수를 호출하는 예제

```
extern "C" int supports_AVX();

int _tmain(int argc, _TCHAR* argv[])
{
    printf("%d\n", supports_AVX() );
    return 0;
}
```

자, 이제 실행한 후 코드를 하나씩 쫓아가서 해당 함수의 디스어셈블리된 코드가 보이는지 확인해보자.

그림 1-10 디스어셈블리 창을 통해 본 코드의 모습

```
supports_AVX:
000000013F421020  mov      eax,1
000000013F421025  cpuid
000000013F421027  and      ecx,18000000h
000000013F42102D  cmp      ecx,18000000h
000000013F421033  jne      supports_AVX+2Ch (13F42104Ch)
000000013F421035  mov      ecx,0
000000013F42103A  xgetbv
000000013F42103D  and      eax,6
000000013F421040  cmp      eax,6
000000013F421043  jne      supports_AVX+2Ch (13F42104Ch)
000000013F421045  mov      eax,1
000000013F42104A  jmp      supports_AVX+31h (13F421051h)
000000013F42104C  mov      eax,0
000000013F421051  ret
000000013F421052  int      3
```

디스어셈블리 창을 통해 코드를 보면, 입력한 프로시저명도 잘 나와 있고 코드도 컴파일러에 의한 수정 없이 직접 입력한 대로 잘 들어가 있다. 뭔가 당연한 이야기를 하고 있는 듯하다. 그래도 위 코드를 보다 보면 지금까지 보아왔던 코드와는 뭔가 달라 위화감이 드는데, 이제부터는 그 부분에 대해서 하나씩 소개할 것이다.

그 전에 이번 파트를 시작할 때 잠깐 언급했던 범용 레지스터에 관해서 다시 한 번 소개하려고 한다. 표 1-2에 있는 레지스터를 유심히 본 독자는 알겠지만, 새롭게 추가된 레지스터가 몇 개 있다. 단순히 레지스터의 접두사만 바뀐 게 아니라, 64비트 모드에서는 일반 레지스터의 개수까지 2배 늘어났다.

32비트 모드에서는 일반적으로 EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP라고 부르는 총 8개의 기본 레지스터가 있었다. EAX는 AX 레지스터에서 16비트 확장된 32비트 크기의 레지스터며, 한 레지스터 안에서 유저가 어떤 크기로 사용하느냐에 따라 32비트, 16비트, 8비트 형태로 사용할 수 있다.

32비트라면 앞에서 소개한 것과 같이 EAX 형태로, 16비트라면 AX 형태로, 8비트라면 AH와 AL 형태로 상하위 8비트씩 나눠서 사용할 수 있다. 또한 64비트 모드일 때는 RAX라는 이름으로 사용된다고 위에서 간단히 소개했다. 단, 32비트 모드 이상일 경우 EAX에서 상위 16비트, RAX에서 상위 32비트만 따로 사용할 수는 없다. 여기까지는 기존과 유사하다고 보면 된다.

64비트 모드는 기존 8개의 레지스터에 더해, 일반 레지스터와 형태가 동일한 R8~R15까지의 레지스터를 추가적으로 제공한다. 그리고 독립적인 이름이 없기 때문에 이들을 사용할 때는 비트 타입에 따른 알파벳을 접미사로 추가해준다. 예를 들어 R8 레지스터의 8비트 타입을 사용하고자 한다면 R8B, 16비트 타입을 사용하고자 한다면 R8W, 32비트 타입을 사용하고자 한다면 R8D, 64비트 타입은 그냥 그대로 R8 레지스터로 사용하면 된다.[02]

이렇게 64비트 모드에서는 기본적인 레지스터의 크기뿐만 아니라 개수도 늘었기에, 특정 사용자 함수 내에서 기본 연산을 처리할 때 레지스터가 부족해서 겪는 이전의 어려움을 쉽게 해결할 수 있다. 더 이상 스택이나 코드 배치에 따라 기존 레지스터 돌려 사용하기 등을 하지 않아도 된다. 물론, 필자 개인적인 생각으로는 이게 얼마나 갈는지 모르겠다. 지금은 일반 레지스터 수가 2배로 늘어서 풍족하다(?) 느낄지 몰라도, 조만간 이것조차 부족하다고 느끼게 될 것 같다.

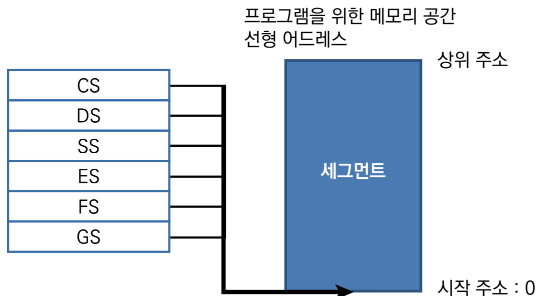
여기까지 살펴본 64비트 모드의 레지스터는 64비트 모드에 맞춰 크기와 개수에 변화가 있었다. 세그먼트 레지스터들(CS, DS, SS, ES, FS, GS)은 일반 레지스터들과 달리 16비트 모드 이후로 크기가 변하지 않았는데, 64비트 모드에서 모든 세그먼트 레지스터의 시작 주소는 기본적으로 0번지로 되어 있다. 따라서 모든 세그먼트가 같은 메모리 시작점을 가리키고 있다고 보면 된다.

---

02 CPU 레퍼런스 가이드에 따르면 8비트 타입의 경우 접미사로 R8L~R15L을 사용하도록 하고 있지만, 실제로는 R8B~ R15B로 사용을 할 수 있다. 단순 매뉴얼 표기 오류인지, 내용이 변경된 것인지는 확인하지 못했다.

어찌 보면 64비트 모드에서는 세그먼트 레지스터를 사용할 필요가 없지만, 하위 버전과의 호환을 위해 해당 레지스터를 유지하는 것 같다. 반대로 생각해보면, 확장된 레지스터나 앞으로 소개할 AVX 기능을 사용하여 사용자 함수를 작성하는 순간 하위 버전과 호환되지 않는다고 봐야 한다.

그림 1-11 64비트 모드에서 세그먼트 레지스터들이 사용되는 모습



새롭게 확장된 레지스터를 코드에서 사용해보려고 필자는 간단한 사용자 함수를 만들어보았다. 단순히 2개의 int64 변수값을 더하는 함수를 만들어서 호출했다.

예제 1-5 단순히 64비트 값 두 개를 더하는 어셈블리 함수

```
testAdd PROC

push  rbp
mov  rbp, rsp

; 매개변수 세팅.
mov  rax, QWORD PTR [rbp+8]  ; 첫 번째 64비트 파라미터.
mov  rcx, QWORD PTR [rbp+16] ; 두 번째 64비트 파라미터.

add  rax, rcx
pop  rbp
ret
```