[KISTI 기술문서] ISBN 978-89-294-1309-5-93560

X86_64 어셈블리 프로그래밍 기초

2022. 6. 1.

한국과학기술정보연구원
슈퍼컴퓨팅기술개발센터

저자소개

김상완

한국과학기술정보연구원
국가슈퍼컴퓨팅본부 슈퍼컴퓨팅기술개발센터
책임연구원
sangwan@kisti.re.kr

오광진

한국과학기술정보연구원
국가슈퍼컴퓨팅본부 슈퍼컴퓨팅기술개발센터
책임연구원
koh@kisti.re.kr

본 보고서는 국가과학기술연구회에서 지원한 창의형 융합연구사업(CAP)인 "차세대 초고성능컴퓨터를 위한 이기종 매니코어 하드웨어 시스템 개발" 사업의 결과입니다.

목 차

1. 개요

- 어셈블리어란 사용자가 이해하기 어려운 기계어 대신에 명령 기능을 쉽게 연상할 수 있는 기호를 기계어와 대응시켜 코드화한 언어
- 어셈블리어로 작성한 프로그램은 어셈블러를 통해 오브젝트 코드(기계어)로 변환하는 과정(어셈블)을 거쳐야 함

○ 어셈블리 언어를 배워야 하는 이유는 다음과 같다:[1)]
- 하드웨어 구조에 대한 이해를 돕는다. 기본적인 명령어, 레지스터, 메모리 접근방식, 하드웨어 인터페이싱에 대해 더 잘 이해할 수 있다.
- 툴 체인에 대한 이해: 컴파일러, 어셈블러, 링커, 로더, 디버거와 같은 도구의 세부사항을 이해할 수 있다
- 알고리즘 향상: 저수준 프로그램의 작성을 통해 프로그래머는 세부 사항에 더 많은 생각을 하게 된다.
- 기능/절차에 대한 이해 : 함수/프로시저 호출이 작동하는 방식을 이해하게 되고, 스텍 기반의 인수, 보존 레지스터 및 동적 로컬 변수에 대한 개념을 이해할 수 있음
- 컴파일러의 범위 이해: 컴파일러가 컴퓨터 아키텍처와 관련하여 하는 일과 하지 않는 일에 대해 이해할 수 있음
- 공유메모리, 인터럽트, 스레드 처리, 경쟁 조건과 같은 개념에 대해 이해 할 수 있음

1.1 CISC vs RISC

○ CISC (Complex Instruction Set Computer)
- 복잡한 명령어 집합을 갖는 CPU 아키텍처
- 명령어가 복잡하기 때문에 명령어를 해석하는데 상대적으로 긴 시간이 필요하며 해석에 필요한 회로도 복잡함
- 연산의 대상은 레지스터, 메모리, 또는 상수(immediate value)의 조합으로 다양함
- 피연산자(operand)의 개수는 제한이 없으나, 2개~3개 지정하는 경우가 많음
- 복잡한 명령어 처리를 위해 마이크로프로그램 방식을 채택하는 경우가 많음. 복잡한 명령을 다시 단순한 명령어(micro-instruction)로 나누어 명령어 파이프라인에서 처리함

○ RISC (Reduced Instruction Set Computer)
- 명령어 개수를 줄여 하드웨어 구조를 단순하게 만드는 방식
- CISC에서 지원하는 명령은 많지만, 그중에서 실제로 자주 사용되는 명령어는 몇 개 되지 않는다는 사실을 바탕으로 명령어 집합을 구성함.
- CISC에서 줄어든 제어 로직을 대신하여 레지스터와 캐시를 증가시켜 파이프라이닝(piplining) 기법 등을 적용하여 수행 속도가 전체적으로 향상됨

1) x86-64 Assembly Language Programming with Ubuntu (Jorgensen)

구분	CISC	RISC
하드웨어 복잡도	복잡한 하드웨어 구조	단순한 하드웨어 구조
명령어 개수	많은 명령어	최소 명령어
명령어 길이	다양한 길이	고정된 길이
실행 싸이클	복잡한 명령은 여러 클럭이 필요함	단순한 명령을 단일 클럭에 실행
메모리 참조	대부분의 명령이 메모리를 참조함	소수의 명령만 메모리를 참조함
명령어 실행	마이크로 프로그램이 명령을 실행	하드웨어가 직접 명령을 실행
어드레싱 모드	다양한 어드레싱 모드	단순한 어드레싱 모드
레지스터수	비교적 소수의 레지스터	비교적 레지스터 수가 많음
복잡도	마이크로 프로그램이 복잡	컴파일러 제작이 복잡함
파이프라이닝	파이프라인을 적용하기 힘듦	파이프라이닝이 쉬움
마이크로프로세서	X86	ARM, SPARC, MIPS

표 1 CISC와 RISC의 비교

예시 기능	CISC 방식 어셈블리어	RISC 방식 어셈블리어

덧셈
(메모리 어드레싱)
	`ADD Ma, Mb`	`LOAD Ra, Ma`
		`LOAD Rb, Mb`
		`ADD Ra, Rb`
		`STORE Ma, Ra`

곱셈	`mov ax, 10`
	`mov bx, 5`
	`mul bx, ax`
		`mov ax, 0`
		`mov bx, 10`
		`mov cx, 5`
		`begin:`
		`add ax, bx`
		`loop begin`

그림 1. CISC 와 RISC 명령어 비교

※ VLIW (Very Long Instruction Word)
- 명령어를 선형으로 처리하는 것보다 서로 간에 종속성이 없는 명령어들을 한꺼번에 처리하면 효율적일 것이라는 아이디어를 시작으로 만들어진 아키텍처
- 명령어 스케쥴링의 복잡성이 컴파일러로 옮겨지기 때문에 하드웨어의 복잡성을 줄일 수 있음

1.2 X86 명령어 구조

- X86 64비트 또는 32비트 명령은 다음 그림과 같은 포맷으로 인코딩된다.
- 최대 3바이트의 오피코드(opcode, 명령코드)를 중심으로 접두어가 opcode 앞에 붙을 수 있음. opcode 뒤에는 addressing-form specifier(ModR/M 바이트+SIB 바이트로 구성), 주소변위값, 이미디어트 값이 뒤따름

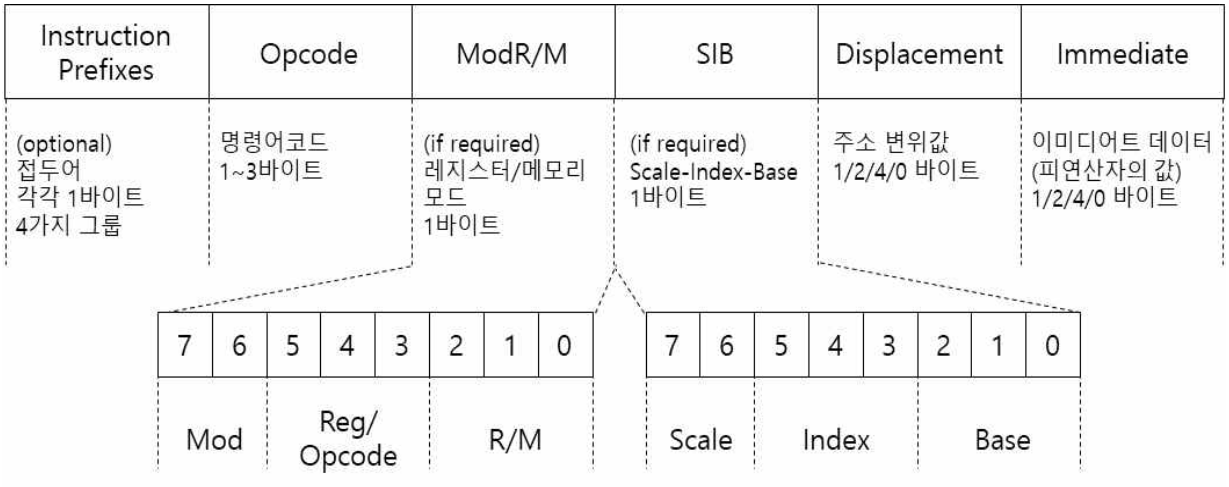


그림 2. Intel 64 and IA-32 명령어 포맷

○ Instruction Prefixes

- 접두어는 4가지 그룹으로 나누어짐.

- Group 1

구분	값	설명
LOCK	F0h	- locked atomic operations[2]
REPNE/REPNZ	F2h	- Repeat-Not-Zero - applies only to string and I/O instructions
REP/REPE/REPZ	F3h	- applies only to string and I/O instructions
BND	F2h	- Intel MPX(memory protection extensions)에 사용됨. - return target should be checked against the bounds specified in the BND0 to BND3 registers[3]

- Group 2

구분	값	설명
Segment override	2Eh	- CS segment override
	36h	- SS segment override
	3Eh	- DS segment override
	26h	- ES segment override
	64h	- FS segment override
	65h	- GS segment override
Branch hints	2Eh	- branch not taken (used only width Jcc instructions)
	3Eh	- branch taken (used only width Jcc instructions)

- Group 3

구분	값	설명
Operrand-size override	66h	- allows a program to switch between 16- and 32-bit operand sizes

- Group 4

구분	값	설명
Address-size override	67h	- allows a program to switch between 16- and 32-bit addressing

2) https://stackoverflow.com/questions/3339141/x86-lock-question-on-multi-core-cpus/3339380#3339380
3) https://stackoverflow.com/questions/43057460/meaning-of-bnd-ret-in-x86

○ Opcode(명령코드)

- primary opcode는 1/2/3 바이트로 구성. 추가적인 3-bit opcode 필드는 ModR/M 바이트에 포함된다.
- 오퍼레이션의 방향, 변위값의 크기, 레지스터 인코딩, 컨디션 코드, 부호 확장 방법을 정의
- 2바이트 opcode 코드 구성: mandatory prefix (66h, F2h, or F3h) + escape byte 0Fh + primary opcode byte

  ※ 예를 들어 `CVTDQ2PD` 명령은 `F3 0F E6`임. (첫번째 바이트 `F3`은 mandatory prefix임)
- 3바이트 opcode 코드 구성: mandatory prefix (66h, F2h, or F3h) + escape byte 0Fh + primary opcode 2 bytes

  ※예를 들어 XMM 레지스터를 위한 `PHADDW` 명령은 `66 0F 38 01`임. (첫번째 바이트 `66`은 mandatory prefix임)

○ ModR/M and SIB (addressing-form specifier, 주소지정 방식 기술자)

- 메모리에 존재하는 피연산자를 참조하는 명령들은 주소지정 방법을 명시함
- 3가지 정보를 명시함

구분	설명
mod field + r/m field	- 32개의 가능한 값 - 8개의 레지스터 + 24개의 주소지정 방식을 결정
reg/opcode field	- 8개의 레지스터 종류 - 또는 3비트의 추가적인 opcode 정보
r/m field	- 레지스터 또는 mod field와 결합하여 주소지정 방식을 기술

- ModR/M 바이트의 특정 인코딩에서는 주소값 표현을 위해 추가적인 정보가 필요함
- base-plus-index와 scale-plus-index 형식의 32비트 주소를 위해 SIB가 필요

구분	설명
scale field	- scale factor
index field	- the register number of the index register
base field	- the register number of the base register

○ Displacement and Immediate

- 주소지정 방식에서 주소 변위(displacement) 값이 필요한 경우 1, 2 또는 4바이트가 추가됨.
- 피연산자로 이미디어트 값이 필요할 경우 사용됨. 1, 2 또는 4바이트가 추가됨.

2. 어셈블리 코딩

※ 이 문서에 작성된 코드는 우분투 리눅스 20.04 LTS 64비트 운영체제에서 테스트함

2.1. Hello World

다음과 같은 hello.asm 파일을 작성한다. 참고[4]

```
section .data
        text db "Hello, World!",10
section .text
        global _start
_start: mov rax, 1
        mov rdi, 1
        mov rsi, text
        mov rdx, 15    ; length of message
        syscall
        mov rax, 60
        mov rdi, 0
        syscall
```

- 빌드를 위한 nasm[5] 어셈블러 설치한다.

```
$ sudo apt-get install nasm
```

컴파일하고, 링크하여 실행파일을 만들고 실행한다.

```
$ nasm -f elf64   -o hello.o  -l hello.lst  hello.asm
$ ld hello.o  -o hello
$ ./hello
Hello, World!
```

- 옵션 -f 는 출력 파일의 형식으로 elf64는 리눅스를 위한 x86_64 포맷을 의미한다. nasm에서 지원하는 출력파일 형식은 nasm -hf **명령을** 참고한다.
- 옵션 -l 은 소스 코드와 매크로 처리 결과와 생성된 코드를 비교할 수 있도록 출력한다.

2.2. 코드 설명

- data 섹션(.data)은 프로그램에서 초기화된 정적 변수를 위한 공간으로, 글로벌 변수와 정적 로컬 변수를 위한 공간이다. 이 섹션의 크기는 런타임에서 변경되지 않는다. data 섹션은 읽기와 쓰기가 가능하나, 읽기 전용을 위한 .rodata 섹션이 존재한다.[6]
- text 섹션(section .text)는 코드를 위한 영역으로 읽기만 가능하다.
- _start 레이블은 프로그램의 엔트리 포인트이다. 이것은 디폴트 값으로 엔트리 포인트를 바꾸고 싶을 경우 ld -e foo 라고 할 수 있다.
- x86_64 아키텍처의 레지스터는 다음과 같다. 자세한 것은 Intel 64 and IA-32 Architectures Software Developer's Manual(이하 SDM)[7]을 참고한다.

4) x86_64 Linux Assembly https://www.youtube.com/watch?v=VQAKkuLL31
5) https://github.com/netwide-assembler/nasm , https://www.nasm.us/
6) https://en.wikipedia.org/wiki/Data_segment
7) https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

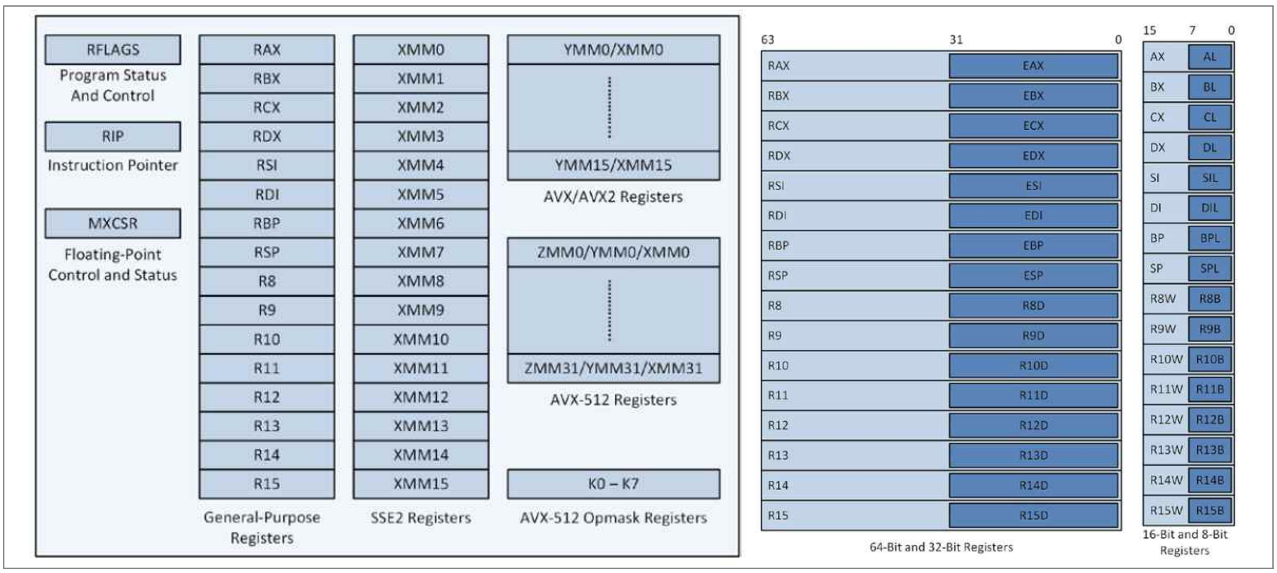


그림 3. X86-64 프로세서 내부 레지스터 구조

- `syscall` 은 시스템콜을 호출하는 명령어로, 리눅스에서 사용하는 시스템 콜 번호는 관련 문서(64비트[8], 32비트[9], ARM[10])를 참고한다. 아키텍처에 따라서 시스템 호출 번호가 서로 다름을 알 수 있다.
- 64비트 리눅스에서 `sys_write` 시스템콜 정의는 다음과 같다.[11]

레지스터	값	설명
%rax	1	sys_write
%rdi	1	unsigned int fd (표준출력:1)
%rsi	text	const char *buf
%rdx	15	size_t count

- 시스템콜 `sys_exit` 의 정의는 다음과 같다.

레지스터	값	설명
%rax	60	sys_exit
%rdi	0	int error_code

(참고) nasm에는 기계어를 어셈블리어로 바꾸는 역어셈블러(disassembler)를 함께 제공한다. `ndisasm` 명령을 이용하여 다음과 같이 확인이 가능하다.

```
$ ndisasm -b64  hello.o > hello.dasm
$ less hello.dasm
...
00000210  B801000000        mov eax,0x1
00000215  BF01000000        mov edi,0x1
0000021A  48BE000000000000  mov rsi,0x0
         -0000
```

8) https://github.com/torvalds/linux/blob/v4.17/arch/x86/entry/syscalls/syscall_64.tbl
9) https://github.com/torvalds/linux/blob/v4.17/arch/x86/entry/syscalls/syscall_32.tbl
10) https://github.com/torvalds/linux/blob/v4.17/arch/arm/tools/syscall.tbl
11) https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/

```
00000224  BA0E000000        mov edx,0xe
00000229  0F05              syscall
0000022B  B83C000000        mov eax,0x3c
00000230  BF01000000        mov edi,0x1
00000235  0F05              syscall
...
```

GNU 컴파일러에서 제공하는 objdump 명령을 이용하여 역어셈블링 하기

```
$ objdump  -d hello.o
...
Disassembly of section .text:
0000000000000000 <_start>:
   0:   b8 01 00 00 00         mov    $0x1,%eax
   5:   bf 01 00 00 00         mov    $0x1,%edi
   a:   48 be 00 00 00 00 00   movabs $0x0,%rsi
  11:   00 00 00
  14:   ba 0e 00 00 00         mov    $0xe,%edx
  19:   0f 05                  syscall
  1b:   b8 3c 00 00 00         mov    $0x3c,%eax
  20:   bf 01 00 00 00         mov    $0x1,%edi
  25:   0f 05                  syscall
```

- hello.lst 파일의 내용은 다음과 같다.

```
     1                                  section .data
     2 00000000 48656C6C6F2C20576F-             text db "Hello, World!",10
     2 00000009 726C64210A
     3                                  section .text
     4                                          global _start
     5 00000000 B801000000              _start: mov rax, 1
     6 00000005 BF01000000                      mov rdi, 1
     7 0000000A 48BE-                           mov rsi, text
     7 0000000C [0000000000000000]
     8 00000014 BA0F000000                      mov rdx, 15    ; length of message
     9 00000019 0F05                            syscall
    10 0000001B B83C000000                      mov rax, 60
    11 00000020 BF00000000                      mov rdi, 0
    12 00000025 0F05                            syscall
```

※ object 파일의 심볼을 조사하고 싶다면 nm 유틸리티를 이용할 수 있음
출력 결과에서 "T"는 text(code) section, "d"는 data section을 의미. (man nm 참고)

```
$ nm hello.o
0000000000000000 T _start
0000000000000000 d text
```

※ strip 명령은 실행 파일이나 오브젝트 코드에서 쓸모없는 정보를 제거한다.
(다음 결과에서 strip 실행 전후에 파일 크기가 변경됨을 확인)

```
$ ls -la hello
-rwxrwxr-x 1 kisti kisti 8888 May  9 09:58 hello
$ strip hello
$ ls -la hello
-rwxrwxr-x 1 kisti kisti 8488 May  9 09:59 hello
```

※ objcopy 명령은 오브젝트 파일을 복사 및 변환하거나, 선택적으로 필요한 부분만 복사할 수 있다. 헤더를 제외한 바이너리만 뽑아내거나, 바이너리 포맷을 바꾸는 등에 이용됨.

2.3. 32비트 버전

- 32비트 아키텍처에서는 다음과 같이 작성한다.

```
;hello32.asm
section .text
   global _start
_start:
   mov  edx,len
   mov  ecx,msg
   mov  ebx,1       ;file descriptor (stdout)
   mov  eax,4       ;system call number (sys_write)
   int  0x80        ;call kernel
   mov  eax,1       ;system call number (sys_exit)
   int  0x80        ;call kernel
section .data
msg db 'Hello, World!', 0xa  ;string to be printed
len equ $ - msg     ;length of the string
```

빌드 및 실행

```
$ nasm -f elf  -o hello32.o   hello32.asm
$ ld -m elf_i386  -s hello32.o  -o hello32
$ ./hello32
Hello, World!
```

64비트 프로그램 버전과의 차이점은 다음과 같다:

- 32비트 환경에서는 `int(interrupt) 0x80`을 이용하여 시스템콜을 수행한다.
- 32비트에서는 `sys_write`의 시스템콜 번호가 4이며, `sys_exit`는 1번이다.
- 64비트 레지스터(`rax, rdi, rsi`)가 아닌 32비트 레지스트(`eax, edx, ecx`)를 사용한다.

2.4. ARM 버전

- ARM 에서는 다음과 같다.[12] (이 코드는 RaspberryPI 에서 테스트함)

```
; hello_arm.asm
.text
.global _start
_start:
  mov r0, #1
  ldr r1, =message
  ldr r2, =len
  mov r7, #4
  swi 0
  mov r7, #1
  swi 0
.data
message:
  .asciz "hello world\n"
```

12) http://kerseykyle.com/articles/ARM-assembly-hello-world

```
len = .-message
```

- ARM에서 sys_write syscall은 4번임. sys_exit 는 1번임
- 어셈블러(as)와 링커(ld)를 이용하여 빌드

```
$ as -o hello_arm.o hello_arm.asm
$ ld hello_arm.o -o hello_arm
$ ./hello_arm
hello world
```

2.5. GNU 어셈블러

- GNU 어셈블러를 사용할 경우 다음과 같이 작성한다.[13)]
- 레지스터 이름에 %가 붙으며, 상수에는 $가 붙음.

```
        .global _start
        .text
_start:
        mov     $1, %rax                # system call 1 is write
        mov     $1, %rdi                # file handle 1 is stdout
        mov     $message, %rsi          # address of string to output
        mov     $13, %rdx               # number of bytes
        syscall                         # invoke operating system to do the write
        mov     $60, %rax               # system call 60 is exit
        xor     %rdi, %rdi              # we want return code 0
        syscall                         # invoke operating system to exit
message:
        .ascii  "Hello, world\n"
```

```
$ as -o hello_gnu.o hello_gnu.asm
$ ld  hello_gnu.o -o hello_gnu
$ ./hello_gnu
Hello, world
```

3. C 코드와 연동

3.1. 따라하기

- 다음 코드를 작성한다.[14)] 파라미터로 전달된 인수중 최대값을 구하는 예제.

```
// callmaxofsix.c
#include <stdio.h>
#include <inttypes.h>
int64_t maxofsix(int64_t, int64_t, int64_t, int64_t, int64_t, int64_t);
int main() {
    printf("%ld\n", maxofsix(1,2,3,4,5,6));
    return 0;
}
```

13) https://cs.lmu.edu/~ray/notes/gasexamples/
14) https://cs.lmu.edu/~ray/notes/gasexamples/

```
// maxofsix.s
.global  maxofsix
.text
maxofsix:
  mov   %rdi, %rax  # rdi  1st argument
  cmp   %rsi, %rax  # rsi  2nd argument
  cmovl %rsi, %rax
  cmp   %rdx, %rax  # rdx  3rd argument
  cmovl %rdx, %rax
  cmp   %rcx, %rax  # rcx  4th argument
  cmovl %rcx, %rax
  cmp   %r8, %rax  # r8  5th argument
  cmovl %r8, %rax
  cmp   %r9, %rax  # r9  6th argument
  cmovl %r9, %rax
  ret
```

- mov 명령(instruction)은 두 레지스터의 값을 비교하고, 그 결과에 따라 EFLAGS 레지스터의 상태 비트(flags)들을 변경시킴. 값의 비교는 첫번째 오퍼랜드에서 두번째 오퍼랜드를 차감(substract)하는 방법으로 이루어짐.
- cmovl (conditional move, CMOVcc)는 EFLAGS 레지스터의 SF(sign flag)가 0F가 아니면 값을 이동함.
- 다음과 같이 컴파일한다.

```
as -g -o maxofsix.o  maxofsix.s
gcc -g -c callmaxofsix.c
gcc  -o maxofsix   callmaxofsix.o  maxofsix.o
```

- 실행결과는 다음과 같다.

```
$ ./maxofsix
6
```

- 오브젝트 파일을 역어셈블링을 하면 다음과 같다.

```
$ objdump  -d maxofsix.o
maxofsix.o:     file format elf64-x86-64
Disassembly of section .text:
0000000000000000 <maxofsix>:
   0:  48 89 f8              mov    %rdi,%rax
   3:  48 39 f0              cmp    %rsi,%rax
   6:  48 0f 4c c6           cmovl  %rsi,%rax
   a:  48 39 d0              cmp    %rdx,%rax
   d:  48 0f 4c c2           cmovl  %rdx,%rax
  11:  48 39 c8              cmp    %rcx,%rax
  14:  48 0f 4c c1           cmovl  %rcx,%rax
  18:  4c 39 c0              cmp    %r8,%rax
  1b:  49 0f 4c c0           cmovl  %r8,%rax
  1f:  4c 39 c8              cmp    %r9,%rax
  22:  49 0f 4c c1           cmovl  %r9,%rax
  26:  c3                    retq
```

※ 프로시저로 전달되는 파라미터를 전달하기 위해서는 레지스터 또는 스택을 통해 전달됨. 파라미터 개수가 6개 이하일 경우는 레지스터를 통해 전달이 가능.[15)]

3.2. GDB를 이용한 디버깅

- objdump 명령으로 바이너리 파일의 디버깅 정보를 확인

```
# objdump  -g   maxofsix

maxofsix:     file format elf64-x86-64

Contents of the .eh_frame section (loaded from maxofsix):
....
```

- gdb (GNU debugger)[16]를 이용하여 디버깅하기

```
$ gdb  maxofsix
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from maxofsix...
```

- 소스코드 리스트 보기

```
(gdb) list
1   // callmaxofsix.c
2   #include <stdio.h>
3   #include <inttypes.h>
4   int64_t maxofsix(int64_t, int64_t, int64_t, int64_t, int64_t, int64_t);
5
6   int main() {
7       printf("%ld\n", maxofsix(1,2,3,4,5,6));
8       return 0;
9   }
```

- 프로그램 시작하기

```
(gdb) start
Temporary breakpoint 1 at 0x1139: file callmaxofsix.c, line 6.
Starting program: /home/kisti/asm/maxofsix

Temporary breakpoint 1, main () at callmaxofsix.c:6
6   int main() {
```

15) Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1, Chapter 6 Procedure Calls 참고

16) https://www.sourceware.org/gdb/documentation/

- 역어셈블 결과 보기

```
(gdb) disas main
Dump of assembler code for function main:
=> 0x0000555555555139 <+0>:   endbr64
   0x000055555555513d <+4>:   push   %rbp
   0x000055555555513e <+5>:   mov    %rsp,%rbp
   0x0000555555555141 <+8>:   mov    $0x6,%r9d
   0x0000555555555147 <+14>:  mov    $0x5,%r8d
   0x000055555555514d <+20>:  mov    $0x4,%ecx
   0x0000555555555152 <+25>:  mov    $0x3,%edx
   0x0000555555555157 <+30>:  mov    $0x2,%esi
   0x000055555555515c <+35>:  mov    $0x1,%edi
   0x0000555555555161 <+40>:  callq  0x555555555181 <maxofsix>
   0x0000555555555166 <+45>:  mov    %rax,%rsi
   0x0000555555555169 <+48>:  lea    0xe94(%rip),%rdi        # 0x555555556004
   0x0000555555555170 <+55>:  mov    $0x0,%eax
   0x0000555555555175 <+60>:  callq  0x555555555030 <printf@plt>
   0x000055555555517a <+65>:  mov    $0x0,%eax
   0x000055555555517f <+70>:  pop    %rbp
   0x0000555555555180 <+71>:  retq
End of assembler dump.

(gdb) disas maxofsix
Dump of assembler code for function maxofsix:
   0x0000555555555181 <+0>:   mov    %rdi,%rax
   0x0000555555555184 <+3>:   cmp    %rsi,%rax
   0x0000555555555187 <+6>:   cmovl  %rsi,%rax
   0x000055555555518b <+10>:  cmp    %rdx,%rax
   0x000055555555518e <+13>:  cmovl  %rdx,%rax
   0x0000555555555192 <+17>:  cmp    %rcx,%rax
   0x0000555555555195 <+20>:  cmovl  %rcx,%rax
   0x0000555555555199 <+24>:  cmp    %r8,%rax
   0x000055555555519c <+27>:  cmovl  %r8,%rax
   0x00005555555551a0 <+31>:  cmp    %r9,%rax
   0x00005555555551a3 <+34>:  cmovl  %r9,%rax
   0x00005555555551a7 <+38>:  retq
   0x00005555555551a8 <+39>:  nopl   0x0(%rax,%rax,1)
End of assembler dump.
```

- 중단점 설정하기 (b *address) (maxofsix 프로시저의 시작과 끝에 중단점을 설정)

```
(gdb) break *0x0000555555555181
Breakpoint 2 at 0x555555555181: file maxofsix.s, line 4.
(gdb) break *0x00005555555551a7
Breakpoint 3 at 0x5555555551a7: file maxofsix.s, line 15.
```

- 중단점 확인하기

```
(gdb) info break
Num     Type           Disp Enb Address            What
2       breakpoint     keep y   0x0000555555555181 maxofsix.s:4
3       breakpoint     keep y   0x00005555555551a7 maxofsix.s:15
```

- 실행하기 (continue, 중단점에서 중지)

```
(gdb) continue
```

```
Continuing.
Breakpoint 2, maxofsix () at maxofsix.s:4
4     mov   %rdi, %rax
```

- 레지스터 정보 출력(info r)

```
(gdb) info register rax rdi rsi rdx rcx r8 r8
rax            0x555555555139      93824992235833
rdi            0x1                 1
rsi            0x2                 2
rdx            0x3                 3
rcx            0x4                 4
r8             0x5                 5
r8             0x5                 5
```

※ info all-registers 는 모든 레지스터를 출력한다.

- 레지스터 값 변경하기

```
(gdb) set $rdi=99
(gdb) info register rdi
rdi            0x63                99
```

- 실행하기 (중단점에서 중지)

```
(gdb) continue
Continuing.
Breakpoint 3, maxofsix () at maxofsix.s:15
15    ret
```

- 다음 명령 실행

```
(gdb) nexti
0x0000555555555166 in main () at callmaxofsix.c:7
```

- 실행하기 (프로그램 종료)

```
(gdb) continue
Continuing.
99
[Inferior 1 (process 493885) exited normally]
```

3.3. 재귀호출 예제

- 피보나치 수열(fibonacci sequence)은 다음과 같이 정의된다.

  F0 = 0, F1 = 1, F(n) = F(n-1) + F(n-2), n > 1

- 피보나치 수열의 n 번째 값을 구하는 C 프로그램은 다음과 같다.

```
uint64_t fibonacci_c(uint64_t n) {
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fibonacci_c(n-1) + fibonacci_c(n-2);
}
int main() {
  uint64_t n = 0;  uint64_t f = 1;
  n = 20; f = fibonacci_c(n); printf("f(%lu) = %lu\n", n, f);
  return 0;
}
```

- 어셈블리어와 연동하기 위한 C 코드

```
#include <stdio.h>
#include <inttypes.h>
uint64_t fibonacci_asm(uint64_t n);
int main() {
  uint64_t n = 0;  uint64_t f = 1;
  n = 20; f = fibonacci_asm(n); printf("f(%lu) = %lu\n", n, f);
  return 0;
}
```

- 어셈블리어 코드

```
; fibonacci_asm.s
    .text
    .globl    fibonacci_asm
fibonacci_asm:
    pushq    %rbp
    movq     %rsp, %rbp
    pushq    %rbx
    subq    $16, %rsp
    movq    %rdi, -16(%rbp)
    cmpq    $0, -16(%rbp)
    jne    .L2
    movl    $0, %eax
    jmp    .L3
.L2:
    cmpq    $1, -16(%rbp)
    jne    .L4
    movl    $1, %eax
    jmp    .L3
.L4:
    movq    -16(%rbp), %rax
    subq    $1, %rax
    movq    %rax, %rdi
    call    fibonacci_asm ;; recursive call
    movq    %rax, %rbx
    movq    -16(%rbp), %rax
    subq    $2, %rax
    movq    %rax, %rdi
    call    fibonacci_asm ;; recursive call
    addq    %rbx, %rax
.L3:
    addq    $16, %rsp
    popq    %rbx
    popq    %rbp
    ret
```

- 컴파일 및 실행

```
gcc -g -o fibo.o  -c fibo.c
as  -g -o fibonacci_asm.o fibonacci_asm.s
gcc -o  fibo  fibonacci_asm.o  fibo.o
```

```
$ ./fibo
f(20) = 6765
```

4. ELF 포맷

- 컴파일된 파일은 ELF(Executable and Linkable Format)[17] 형식으로 저장된다.
- ELF는 유닉스 계열 운영체제에서 ABI(application binary interface)를 위한 표준으로 유닉스 System V Release 4(SVR4, 1988년) 버전에서 처음으로 적용됨.
- ELF 파일은 elf header에 이어서 program header 테이블 또는 section header 테이블(또는 둘 다)이 이어진다.

4.1. ELF 파일 헤더

- ELF 파일은 헤더와 데이터로 이루어진다.
- 자세한 것은 `man elf` 커맨드를 통해 참조가 가능함

표 2. ELF 헤더 형식(64비트)

<table>
<tr><th>오프셋</th><th>크기 (Bytes)</th><th>필드</th><th>설명</th></tr>
<tr><td>0x00</td><td>4</td><td>MAGIC</td><td>0x7F와 'ELF'; 네 바이트. 매직넘버</td></tr>
<tr><td>0x04</td><td>1</td><td>CLASS</td><td>1이면 32비트, 2이면 64비트</td></tr>
<tr><td>0x05</td><td>1</td><td>DATA</td><td>1이면 리틀엔디언, 2이면 빅엔디언</td></tr>
<tr><td>0x06</td><td>1</td><td>VERSION</td><td>오리지널 버전의 ELF인 경우 1로 설정</td></tr>
<tr><td>0x07</td><td>1</td><td>OSABI</td><td>운영체제 ABI(Application binary interface)
<table><tr><th>값</th><th>ABI</th></tr><tr><td>0x00</td><td>System V</td></tr><tr><td>0x01</td><td>HP-UX</td></tr><tr><td>0x02</td><td>NetBSD</td></tr><tr><td>0x03</td><td>Linux</td></tr><tr><td>0x06</td><td>Solaris</td></tr><tr><td>0x07</td><td>AIX</td></tr><tr><td>...</td><td>...</td></tr></table></td></tr>
<tr><td>0x08</td><td>1</td><td>ABIVERSION</td><td>ABI 버전을 명시</td></tr>
<tr><td>0x09</td><td>7</td><td>PAD</td><td>빈 영역</td></tr>
<tr><td>0x10</td><td>2</td><td>e_type</td><td>1=재배치, 2=실행, 3=공유, 4=코어</td></tr>
<tr><td>0x12</td><td>2</td><td>e_machine</td><td><table><tr><th>값</th><th>ABI</th></tr><tr><td>0x00</td><td></td></tr><tr><td>0x02</td><td>SPARC</td></tr><tr><td>0x03</td><td>x86</td></tr><tr><td>0x28</td><td>ARM</td></tr><tr><td>0x3E</td><td>x86_64</td></tr><tr><td>0xB7</td><td>AArch64</td></tr><tr><td>...</td><td>...</td></tr></table></td></tr>
<tr><td>0x14</td><td>4</td><td>e_version</td><td>오리지널 버전의 ELF인 경우 1로 설정</td></tr>
<tr><td>0x18</td><td>8</td><td>e_entry</td><td>프로세스가 실행을 시작하는 엔트리 포인트의 메모리 주소</td></tr>
<tr><td>0x20</td><td>8</td><td>e_phoff</td><td>프로그램 헤더 테이블의 시작. (보통은 파일 헤더 다음에 연속해서 붙음. 64비트 ELF의 경우 0x40)</td></tr>
<tr><td>0x28</td><td>8</td><td>e_shoff</td><td>섹션 헤더 테이블의 시작</td></tr>
<tr><td>0x30</td><td>4</td><td>e_flags</td><td>타겟 아키텍처에서 사용하는 플래그 속성</td></tr>
<tr><td>0x34</td><td>2</td><td>e_ehsize</td><td>elf 헤더의 크기. 64비트 ELF의 경우 64바이트</td></tr>
<tr><td>0x36</td><td>2</td><td>e_phentsize</td><td>프로그램 헤더 테이블 엔트리 크기. (모든 엔트리는 같은 크기를 가짐)</td></tr>
</table>

17) https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

0x38	2	e_phnum	프로그램 헤더 테이블 엔트리 개수
0x3A	2	e_shentsize	섹션 헤더 테이블 엔트리 크기. (모든 엔트리는 같은 크기를 가짐)
0x3C	2	e_shnum	섹션 헤더 테이블 엔트리 개수
0x3E	2	e_shstrndx	섹션 이름들을 포함하는 섹션 헤더 테이블 엔트리의 인덱스
0x40			ELF 헤더의 끝

- 실제로 앞서 1절에서 작성한 hello.o와 hello 를 살펴보면 다음과 같다.

```
$ nasm -f elf64   -o hello.o   hello.asm
$ hexdump -C hello.o
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
          MAGIC       CLASS DATA VER OSABI ABIVERSION   PAD
00000010  01 00 3e 00 01 00 00 00  00 00 00 00 00 00 00 00  |..>.............|
          e_type e_machine e_version     e_entry
00000020  00 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  |........@.......|
          e_phoff                   e_shoff
00000030  00 00 00 00 40 00 00 00  00 00 40 00 07 00 03 00  |....@.....@.....|
          e_flags      e_ehsize e_phentsize e_phnum  e_shentsize e_shnum e-shstrndx
00000040  ....
$ ld hello.o  -o hello
$ hexdump  -C hello
00000000  7f 45 4c 46 02 01 01 00  00 00 00 00 00 00 00 00  |.ELF............|
          MAGIC       CLASS DATA VER OSABI ABIVERSION   PAD
00000010  02 00 3e 00 01 00 00 00  00 10 40 00 00 00 00 00  |..>.......@.....|
          e_type e_machine e_version     e_entry (0x401000)
00000020  40 00 00 00 00 00 00 00  38 21 00 00 00 00 00 00  |@.......8!......|
          e_phoff (0x40)            e_shoff (0x2138)
00000030  00 00 00 00 40 00 38 00  03 00 40 00 06 00 05 00  |....@.8...@.....|
           e_flags      e_ehsize e_phentsize e_phnum  e_shentsize e_shnum  e-shstrndx
00000040 ...
```

- 위에서 프로그램 헤더 엔트리 크기는 0x38(phentsize), 개수는 3개(phnum), 섹션 헤더 엔트리 크기는 0x40(shentsize), 개수는 6개(shnum)임을 알 수 있다.

	오프셋	엔트리 크기	엔트리 개수	전체 크기
프로그램 헤더	0x0040	0x38 (56 bytes)	3	168 bytes
섹션 헤더	0x2138	0x40 (64 bytes)	6	384 bytes

※ file 명령은 ELF 헤더의 요약된 정보를 출력한다.

```
$ file hello.o
hello.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
$ file hello
hello: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, not stripped
```

※ readelf 유틸리티를 사용하면 ELF 파일의 상세한 정보를 조회할 수 있음

```
$ readelf -h hello
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
```

```
Version:                           1 (current)
OS/ABI:                            UNIX - System V
ABI Version:                       0
Type:                              EXEC (Executable file)
Machine:                           Advanced Micro Devices X86-64
Version:                           0x1
Entry point address:               0x401000
Start of program headers:          64 (bytes into file)
Start of section headers:          8504 (bytes into file)
Flags:                             0x0
Size of this header:               64 (bytes)
Size of program headers:           56 (bytes)
Number of program headers:         3
Size of section headers:           64 (bytes)
Number of section headers:         6
Section header string table index: 5
```

4.2. 프로그램 헤더

- 프로그램 헤더 테이블은 시스템이 프로세스 이미지를 어떻게 생성하는지 말해준다. phoff에서 시작하여 phnum 개수의 엔트리로 구성된다.

표 3. ELF 프로그램 헤더 형식(64비트)

<table>
<tr><th>오프셋</th><th>크기
(Bytes)</th><th>필드</th><th>설명</th></tr>
<tr><td>0x00</td><td>4</td><td>p_type</td><td>세그먼트의 타입
<table><tr><th>값</th><th>설명</th></tr><tr><td>0x00000000</td><td>NULL</td></tr><tr><td>0x00000001</td><td>Loadable 세그먼트</td></tr><tr><td>0x00000002</td><td>동적 링크 정보</td></tr><tr><td>…</td><td>…</td></tr></table></td></tr>
<tr><td>0x04</td><td>4</td><td>p_flag</td><td>세그먼트 의존 플래그</td></tr>
<tr><td>0x08</td><td>8</td><td>p_offset</td><td>파일 이미지에서 세그먼트의 오프셋</td></tr>
<tr><td>0x10</td><td>8</td><td>p_vaddr</td><td>메모리에서 세그먼트의 가상 메모리 주소</td></tr>
<tr><td>0x18</td><td>8</td><td>p_paddr</td><td>물리적 주소</td></tr>
<tr><td>0x20</td><td>8</td><td>p_filesz</td><td>파일에서 세그먼트의 크기(바이트 단위)</td></tr>
<tr><td>0x28</td><td>8</td><td>p_memsz</td><td>메모리에서 세그먼트의 크기(바이트)</td></tr>
<tr><td>0x30</td><td>8</td><td>p_align</td><td>p_vaddr = p_offset % p_align 이되는 2의 거듭제곱수. 세그먼트에 맞추기 위함.</td></tr>
<tr><td>0x38</td><td></td><td></td><td>프로그램 헤더의 끝</td></tr>
</table>

- hello 파일의 프로그램 헤더(3개) 내용

```
$ hexdump  -s 0x0040  -n 168 -C hello
00000040  01 00 00 00 04 00 00 00  00 00 00 00 00 00 00 00  |................|
00000050  00 00 40 00 00 00 00 00  00 00 40 00 00 00 00 00  |..@.......@.....|
00000060  e8 00 00 00 00 00 00 00  e8 00 00 00 00 00 00 00  |................|
00000070  00 10 00 00 00 00 00 00  01 00 00 00 05 00 00 00  |................|
00000080  00 10 00 00 00 00 00 00  00 10 40 00 00 00 00 00  |..........@.....|
00000090  00 10 40 00 00 00 00 00  27 00 00 00 00 00 00 00  |..@.....'.......|
000000a0  27 00 00 00 00 00 00 00  00 10 00 00 00 00 00 00  |'...............|
000000b0  01 00 00 00 06 00 00 00  00 20 00 00 00 00 00 00  |......... ......|
```

```
000000c0  00 20 40 00 00 00 00 00  00 20 40 00 00 00 00 00  |. @..... @.....|
000000d0  0e 00 00 00 00 00 00 00  0e 00 00 00 00 00 00 00  |................|
000000e0  00 10 00 00 00 00 00 00                           |........|
```

- 프로그램 엔트리의 개수는 3개임

순서	p_type	p_flag	p_offset	p_vaddr	p_filesz	p_align
1	0x01	0x04	0x0000	0x400000	0xE8	0x1000 (4096)
2	0x01	0x05	0x1000	0x401000	0x27	0x1000 (4096)
3	0x01	0x06	0x2000	0x402000	0x0E	0x1000 (4096)

4.3. 섹션 헤더

- 섹션 헤더 테이블은 shoff에서 시작하여 shnum 개수의 엔트리로 구성된다.

표 4. ELF 섹션 헤더 형식(64비트)

<table>
<tr><th>오프셋</th><th>크기
(Bytes)</th><th>필드</th><th>설명</th></tr>
<tr><td>0x00</td><td>4</td><td>sh_name</td><td>섹션의 이름 (스트링 테이블에 대한 포인터)</td></tr>
<tr><td>0x04</td><td>4</td><td>sh_type</td><td><table><tr><th>값</th><th>설명</th></tr><tr><td>0x0</td><td>NULL</td></tr><tr><td>0x1</td><td>Program data</td></tr><tr><td>0x2</td><td>Symbol table</td></tr><tr><td>0x3</td><td>String table</td></tr><tr><td>...</td><td>...</td></tr></table></td></tr>
<tr><td>0x08</td><td>8</td><td>sh_flag</td><td>섹션의 속성<table><tr><th>값</th><th>설명</th></tr><tr><td>0x1</td><td>쓰기 가능</td></tr><tr><td>0x2</td><td>실행중 메모리를 점유함</td></tr><tr><td>0x4</td><td>실행가능</td></tr><tr><td>...</td><td>...</td></tr></table></td></tr>
<tr><td>0x10</td><td>8</td><td>sh_addr</td><td>메모리에서 섹션이 로딩되는 가상 메모리 주소</td></tr>
<tr><td>0x18</td><td>8</td><td>sh_offset</td><td>파일 이미지에서 오프셋</td></tr>
<tr><td>0x20</td><td>8</td><td>sh_size</td><td>파일 이미지에서 섹션의 크기</td></tr>
<tr><td>0x30</td><td>8</td><td>sh_addralign</td><td>섹션 얼라인먼트. 2의 거듭제곱</td></tr>
<tr><td>0x38</td><td>8</td><td>sh_entsize</td><td>섹션이 고정크기의 엔트리를 포함할 경우. 엔트리 크기</td></tr>
<tr><td>0x40</td><td></td><td></td><td>섹션 헤더의 끝</td></tr>
</table>

- hello 파일의 섹션 헤더(6개) 내용

```
          $ hexdump  -s 0x2138  -n 384 -v -C hello
(1) =>    00002138  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
64B       00002148  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
          00002158  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
          00002168  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
(2) =>    00002178  1b 00 00 00 01 00 00 00  06 00 00 00 00 00 00 00
64B       00002188  00 10 40 00 00 00 00 00  00 10 00 00 00 00 00 00
          00002198  27 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
          000021a8  10 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
(3) =>    000021b8  21 00 00 00 01 00 00 00  03 00 00 00 00 00 00 00
```

```
 64B      000021c8  00 20 40 00 00 00 00 00  00 20 00 00 00 00 00 00
          000021d8  0e 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
          000021e8  04 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
 (4) =>   000021f8  01 00 00 00 02 00 00 00  00 00 00 00 00 00 00 00
 64B      00002208  00 00 00 00 00 00 00 00  10 20 00 00 00 00 00 00
          00002218  d8 00 00 00 00 00 00 00  04 00 00 00 05 00 00 00
          00002228  08 00 00 00 00 00 00 00  18 00 00 00 00 00 00 00
 (5) =>   00002238  09 00 00 00 03 00 00 00  00 00 00 00 00 00 00 00
 64B      00002248  00 00 00 00 00 00 00 00  e8 20 00 00 00 00 00 00
          00002258  28 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
          00002268  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
 (6) =>   00002278  11 00 00 00 03 00 00 00  00 00 00 00 00 00 00 00
 64B      00002288  00 00 00 00 00 00 00 00  10 21 00 00 00 00 00 00
          00002298  27 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
          000022a8  01 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
```

※ 참고 readelf -t hello

순서	sh_name	sh_type	sh_flag	sh_addr	sh_offset	sh_size
1	0x0000(0)	0x0	0x00	0x00	0x0000	0x00
2	0x001b(27)=.text	0x1(프로그램)	0x06	0x401000	0x1000	0x27 (39)
3	0x0021(33)=.data	0x1(프로그램)	0x03	0x402000	0x2000	0x0e (14)
4	0x0001(1)=.symtab	0x2(심볼)	0x00	0x00	0x2010	0xd8 (216)
5	0x0009(9)=.strtab	0x3(스트링)	0x00	0x00	0x20e8	0x28 (40)
6	0x0011(17)=.shstrtab	0x3(스트링)	0x00	0x00	0x2110	0x27 (39)

- 스트링 섹션의 정보

```
$ hexdump  -s 0x20e8 -n 40 -C hello
000020e8  00 68 65 6c 6c 6f 2e 61  73 6d 00 74 65 78 74 00  |.hello.asm.text.|
000020f8  5f 5f 62 73 73 5f 73 74  61 72 74 00 5f 65 64 61  |__bss_start._eda|
00002108  74 61 00 5f 65 6e 64 00                          |ta._end.|
$ hexdump  -s 0x2110 -n 39 -C hello
00002110  00 2e 73 79 6d 74 61 62  00 2e 73 74 72 74 61 62  |..symtab..strtab|
00002120  00 2e 73 68 73 74 72 74  61 62 00 2e 74 65 78 74  |..shstrtab..text|
00002130  00 2e 64 61 74 61 00                              |..data.|
```

- 심볼 정보

```
$ readelf -s hello
Symbol table '.symtab' contains 9 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000401000     0 SECTION LOCAL  DEFAULT    1
     2: 0000000000402000     0 SECTION LOCAL  DEFAULT    2
     3: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS hello.asm
     4: 0000000000402000     0 NOTYPE  LOCAL  DEFAULT    2 text
     5: 0000000000401000     0 NOTYPE  GLOBAL DEFAULT    1 _start
     6: 000000000040200e     0 NOTYPE  GLOBAL DEFAULT    2 __bss_start
     7: 000000000040200e     0 NOTYPE  GLOBAL DEFAULT    2 _edata
     8: 0000000000402010     0 NOTYPE  GLOBAL DEFAULT    2 _end
```

- .data 섹션

```
$ hexdump  -s 0x2000 -n 14 -C hello
00002000  48 65 6c 6c 6f 2c 20 57  6f 72 6c 64 21 0a        |Hello, World!.|
```

```
0000200e
```

- .text 섹션 (코드)

```
$  hexdump  -s 0x1000 -n 39 -C hello
00001000  b8 01 00 00 00 bf 01 00  00 00 48 be 00 20 40 00  |..........H.. @.|
00001010  00 00 00 00 ba 0e 00 00  00 0f 05 b8 3c 00 00 00  |............<...|
00001020  bf 01 00 00 00 0f 05                              |.......|
00001027
```

4.4. 심볼 테이블

- 오브젝트 파일의 심볼테이블(symbol table)은 프로그램의 심볼릭 정의나 참조를 배치하고 재배치하기 위해 필요한 정보를 포함한다.

표 5. 심볼테이블 형식(64비트)

<table>
<tr><th>크기
(Bytes)</th><th>필드</th><th>설명</th></tr>
<tr><td>4</td><td>st_name</td><td>심볼의 이름. 심볼 스트링 테이블 객체의 인덱스</td></tr>
<tr><td>1</td><td>st_info</td><td>심볼 타입
<table>
<tr><th>값</th><th>설명</th></tr>
<tr><td>STT_NOTYPE(0)</td><td>not defined</td></tr>
<tr><td>STT_OBJECT(1)</td><td>data object</td></tr>
<tr><td>STT_FUNC(2)</td><td>함수 또는 실행 코드</td></tr>
<tr><td>STT_SECTION(3)</td><td>섹션</td></tr>
<tr><td>STT_FILE(4)</td><td>name of source file</td></tr>
<tr><td>STT_LOOS(10)</td><td>os-specific symbol type</td></tr>
<tr><td>...</td><td>...</td></tr>
</table></td></tr>
<tr><td>1</td><td>st_other</td><td>symbol visibility
<table>
<tr><th>값</th><th>설명</th></tr>
<tr><td>STV_DEFAULT(0)</td><td>디폴드 속성</td></tr>
<tr><td>STV_INTERNAL(1)</td><td>processor-specific hidden class</td></tr>
<tr><td>STV_HIDDEN(2)</td><td>로컬 모듈에 종속되어 다른 모듈에 보이지 않음</td></tr>
<tr><td>STV_PROTECTED(3)</td><td>다른 모듈에서 이용할 수 있지만, 로컬 심볼이 우선함</td></tr>
</table></td></tr>
<tr><td>2</td><td>st_shndx</td><td></td></tr>
<tr><td>8</td><td>st_value</td><td>심볼의 내용</td></tr>
<tr><td>8</td><td>st_size</td><td>심볼의 크기</td></tr>
</table>

- hello 파일의 심볼테이블 정보

```
$ hexdump  -s 0x2010 -n 216 -C hello
00002010  [0]00 00 00 00 00 00 00 00    00 00 00 00 00 00 00 00  |................|
00002020    00 00 00 00 00 00 00 00  [1]00 00 00 00 03 00 01 00  |................|
00002030    00 10 40 00 00 00 00 00    00 00 00 00 00 00 00 00  |..@.............|
00002040  [2]00 00 00 00 03 00 02 00    00 20 40 00 00 00 00 00  |......... @.....|
00002050    00 00 00 00 00 00 00 00  [3]01 00 00 00 04 00 f1 ff  |................|
00002060    00 00 00 00 00 00 00 00    00 00 00 00 00 00 00 00  |................|
00002070  [4]0b 00 00 00 00 00 02 00    00 20 40 00 00 00 00 00  |......... @.....|
00002080    00 00 00 00 00 00 00 00  [5]15 00 00 00 10 00 01 00  |................|
00002090    00 10 40 00 00 00 00 00    00 00 00 00 00 00 00 00  |..@.............|
000020a0  [6]10 00 00 00 10 00 02 00    0e 20 40 00 00 00 00 00  |......... @.....|
000020b0    00 00 00 00 00 00 00 00  [7]1c 00 00 00 10 00 02 00  |................|
000020c0    0e 20 40 00 00 00 00 00    00 00 00 00 00 00 00 00  |. @.............|
```

```
000020d0  [8]23 00 00 00 10 00 02 00     10 20 40 00 00 00 00 00  ¦#....... @.....¦
000020e0     00 00 00 00 00 00 00 00                              ¦........¦
```

순서	st_name	st_info	st_other	st_shndx	st_value	st_size
0	0x00	0x00	0x00	0x0000	0x000000	0
1	0x00	0x03(section)	0x00	0x0001	0x401000	0
2	0x00	0x03(section)	0x00	0x0002	0x402000	0
3	0x01(hello.asm)	0x04(file)	0x00	0xfff1	0x000000	0
4	0x0b(text)	0x00	0x00	0x0002	0x402000	0
5	0x15(_start)	0x10	0x00	0x0001	0x401000	0
6	0x10(__bss_start)	0x10	0x00	0x0002	0x40200e	0
7	0x1c(_edata)	0x10	0x00	0x0002	0x40200e	0
8	0x23(_end)	0x10	0x00	0x0002	0x402010	0

5. SSE 확장

5.1 개요

- 스트리밍 SIMD 확장(Streaming SIMD Extension, SSE)는 SIMD(single instruction multiple data, 단일 명령 다중 데이터)를 지원하기 위한 명령
- SSE이전에는 1997년에 발표된 MMX(MultiMedia eXtension) 라는 기술이 있었으며, MMX는 정수 연산만을 지원하였음
- SSE는 1999년 Pentium III에 처음으로 도입되었음[18]. AMD의 3DNow기술에 대응. AMD는 Athlon XP와 Duron(듀론) 이후에 SSE 지원을 추가함
- SSE는 약 70여개의 명령으로 구성. 대부분 단정도 실수연산을 위한 것. XMM0~XMM7의 8개의 128비트(4개의 32비트 실수연산) 레지스터를 추가함. 또한 x86_64에서는 XMM8~XMM15의 8개가 추가로 지원. 32비트 상태 레지스터 MXCSR은 제어를 위해 사용됨
- SSE2는 2개의 64비트 배정도 실수연산, 2개의 64비트 정수 연산, 4개의 32비트 정수 연산, 8개의 16비트 정수연산 또는 16개의 8비트 연산이 가능함.
- SSE에서 추가된 128비트 레지스터들은 타스크 전환시 상태를 유지해야 하기 때문에 추가적인 부담이 될수 있음. 따라서 OS에 의해서 SSE를 가능하게 되게 전까지 기능은 꺼져 있음.

표 6. SSE 명령어 집합

	Scalar 명령	Packed 명령
이동	MOVSS	MOVAPS, MOVUPS, MOVLPS, MOVHPS, MOVLHPS, MOVHLPS, MOVMSKPS
연산(실수)	ADDSS, SUBSS, MULSS, DIVSS, RCPSS, SQRTSS, MAXSS, MINSS, RSQRTSS	ADDPS, SUBPS, MULPS, DIVPS, RCPPS, SQRTPS, MAXPS, MINPS, RSQRTPS
비교	CMPSS, COMISS, UCOMISS	CMPPS
셔플링		SHUFPS, UNPCKHPS, UNPCKLPS
형변환	CVTSI2SS, CVTSS2SI, CVTTSS2SI	CVTPI2PS, CVTPS2PI, CVTTPS2PI
비트연산		ANDPS, ORPS, XORPS, ANDNPS
연산(정수)	PMULHUW, PSADBW, PAVGB, PAVGW, PMAXUB, PMINUB, PMAXSW, PMINSW	
이동	PEXTRW, PINSRW	
제어	LDMXCSR, STMXCSR	
캐시	MOVNTQ, MOVNTPS, MASKMOVQ, PREFETCH0, PREFETCH1, PREFETCH2, PREFETCHNTA, SFENCE	

18) Pentium III 초기에는 Katmai New Instructions(KNI)라는 코드네임으로 시작됨

※ x87 이라고 불렸던, 부동소수점 연산에 관련 명령어 집합은 NPX(Numeric Processor eXtension)라고 함. x87은 초기에는 x86 프로세서에 대응하는 코프로세서 형태로 지원되었으니, 80486 모델에서 부터는 CPU 자체에서 x87명령어를 지원하기 시작함[19]

- 32비트의 Control-Status Register인 MXCSR은 다음과 같다.

31 ~ 16	15	14 13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved	FIZ	RC	PM	UM	OM	ZM	DM	IM	DAZ	PE	UE	OE	ZE	DE	IE

Bit	Name	Bit	Name
IE	Invalid operation flag	IM	Invalid operation mask
DE	Denormal flag	DM	Denormal mask
ZE	Divide-by-zero flag	ZM	Divide-by-zerok mask
OE	Overflow flag	OM	Overflow mask
UE	Underflow flag	UM	Underflow mask
PE	Precision flag	PM	Precision mask
DAZ	Denormals are zero	RC	Rounding control
-	-	FIZ	Flush to zero

그림 4. MXCSR 레지스터

- denormal number(비정규값)[20]이란 IEEE 754 floating-point value 표현에서 exponent가 0이고, significand bits이 non-zero인 경우를 말한다. DAZ를 1로 설정하면, denomal 값을 0으로 취급함
- RC: round to nearest(00b), round down toward +∞(01b), round up toward +∞(10b), and round toward zero or truncate(11b).

5.2 예제

- SSE 기능을 사용하여 4개의 실수 연산을 동시에 수행하는 예제

```
; add_four_floats.asm
; void add_four_floats(float x[4], float y[4])
; x[i] += y[i] for i in range(0..3)

        global   add_four_floats
        section  .text
add_four_floats:
        movdqa   xmm0, [rdi]            ; all four values of x
        movdqa   xmm1, [rsi]            ; all four values of y
        addps    xmm0, xmm1             ; do all four sums in one shot
        movdqa   [rdi], xmm0
        ret
```

```
// test.c
#include <stdio.h>
```

19) https://en.wikipedia.org/wiki/X87
20) https://en.wikipedia.org/wiki/Subnormal_number

```
void add_four_floats(float[], float[]);

int main() {
    float x[] = {-29.750, 244.333, 887.29, 48.1E22};
    float y[] = {29.750,  199.333, -8.29,  22.1E23};
    add_four_floats(x, y);
    printf("%f\n%f\n%f\n%f\n", x[0], x[1], x[2], x[3]);
    return 0;
}
```

```
# do compile
nasm -f elf64 -g  add_four_floats.asm
gcc -g -c test.c
gcc -o test -no-pie  add_four_floats.o test.o
```

```
$ ./test
0.000000
443.665985
879.000000
269100007271845562469580 8.000000
```

- GDB를 이용하여 트레이스를 해보면 다음과 같다.

```
$ gdb ./test
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2
...
(gdb) start
Temporary breakpoint 1 at 0x401150: file test.c, line 6.
Starting program: /home/kisti/asm/nasm/test

Temporary breakpoint 1, main () at test.c:6
6   int main() {
```

```
(gdb) list
1
2   // test.c
3   #include <stdio.h>
4   void add_four_floats(float[], float[]);
5
6   int main() {
7       float x[] = {-29.750, 244.333, 887.29, 48.1E22};
8       float y[] = {29.750,  199.333, -8.29,  22.1E23};
9       add_four_floats(x, y);
10      printf("%f\n%f\n%f\n%f\n", x[0], x[1], x[2], x[3]);
(gdb) disas add_four_floats
Dump of assembler code for function add_four_floats:
   0x0000000000401140 <+0>:   movdqa (%rdi),%xmm0
   0x0000000000401144 <+4>:   movdqa (%rsi),%xmm1
   0x0000000000401148 <+8>:   addps  %xmm1,%xmm0
   0x000000000040114b <+11>:  movdqa %xmm0,(%rdi)
   0x000000000040114f <+15>:  retq
End of assembler dump.
```

- 중단지점을 설정하고 continue

```
(gdb) b *0x0000000000401140
Breakpoint 2 at 0x401140
(gdb) c
```

```
Continuing.

Breakpoint 2, 0x0000000000401140 in add_four_floats ()
```

- xmm0, xmm1 레지스터 내용 (변경전)

```
(gdb) info registers xmm0 xmm1
xmm0           {v4_float = {0xffffffff, 0x0, 0x0, 0x0}, v2_double = {0x0, 0x0}, v16_int8 = {0x2
c, 0xfe, 0xe9, 0x67, 0x0 <repeats 12 times>}, v8_int16 = {0xfe2c, 0x67e9, 0x0, 0x0, 0x0, 0x0, 0x
0, 0x0}, v4_int32 = {0x67e9fe2c, 0x0, 0x0, 0x0}, v2_int64 = {0x67e9fe2c, 0x0}, uint128 = 0x67e9f
e2c}
xmm1           {v4_float = {0x0, 0xffffffff, 0xc85, 0x0}, v2_double = {0x7fffffffffffffff, 0x0},
v16_int8 = {0x61, 0x73, 0x6d, 0x2f, 0x74, 0x65, 0x73, 0x74, 0x0, 0x53, 0x48, 0x45, 0x4c, 0x4c, 0
x3d, 0x2f}, v8_int16 = {0x7361, 0x2f6d, 0x6574, 0x7473, 0x5300, 0x4548, 0x4c4c, 0x2f3d}, v4_int3
2 = {0x2f6d7361, 0x74736574, 0x45485300, 0x2f3d4c4c}, v2_int64 = {0x747365742f6d7361, 0x2f3d4c4c
45485300}, uint128 = 0x2f3d4c4c45485300747365742f6d7361}
```

- xmm0 에 값이 로드됨 (movdqa   xmm0, [rdi])

```
(gdb) stepi
0x0000000000401144 in add_four_floats ()
(gdb) info registers xmm0 xmm1
xmm0           {v4_float = {0xfffffffe3, 0xf4, 0x377, 0xffffffff}, v2_double =....., v2_int64 =
{0x4374553fc1ee0000, 0x66cbb620445dd28f}, uint128 = 0x66cbb620445dd28f4374553fc1ee0000}
xmm1           {v4_float = {0x0, 0xffffffff, 0xc85, 0x0},...
```

- xmm1 에 값이 로드됨 (movdqa   xmm1, [rsi])

```
(gdb) stepi
0x0000000000401148 in add_four_floats ()
(gdb) info registers xmm0 xmm1
xmm0           {v4_float = {0xfffffffe3, 0xf4, 0x377, 0xffffffff}, v2_double = .....
xmm1           {v4_float = {0x1d, 0xc7, 0xffffffff8, 0xffffffff}, v2_double = {0x2eaa7e83dc0000,
0x7fffffffffffffff}, v16_int8 = {0x0, 0x0, 0xee, 0x41, 0x3f, 0x55, 0x47, 0x43, 0xd7, 0xa3, 0x4,
0xc1, 0x2c, 0xfe, 0xe9, 0x67}, v8_int16 = {0x0, 0x41ee, 0x553f, 0x4347, 0xa3d7, 0xc104, 0xfe2c,
0x67e9}, v4_int32 = {0x41ee0000, 0x4347553f, 0xc104a3d7, 0x67e9fe2c}, v2_int64 = {0x4347553f41ee
0000, 0x67e9fe2cc104a3d7}, uint128 = 0x67e9fe2cc104a3d74347553f41ee0000}
```

- 연산결과 xmm0 값이 변경됨 (addps   xmm0, xmm1)

```
(gdb) stepi
0x000000000040114b in add_four_floats ()
(gdb) info registers xmm0
xmm0           {v4_float = {0x0, 0x1bb, 0x36f, 0xffffffff}, v2_double = {0x7754fc0000000000, 0x7
fffffffffffffff}, v16_int8 = {0x0, 0x0, 0x0, 0x0, 0x3f, 0xd5, 0xdd, 0x43, 0x0, 0xc0, 0x5b, 0x44,
0xda, 0x75, 0xe, 0x68}, v8_int16 = {0x0, 0x0, 0xd53f, 0x43dd, 0xc000, 0x445b, 0x75da, 0x680e}, v
4_int32 = {0x0, 0x43ddd53f, 0x445bc000, 0x680e75da}, v2_int64 = {0x43ddd53f00000000, 0x680e75da4
45bc000}, uint128 = 0x680e75da445bc00043ddd53f00000000}
(gdb) info registers mxcsr
mxcsr          0x1fa0             [ PE IM DM ZM OM UM PM ]
```

- SSE 를 사용한 경우와 사용하지 않은 경우 속도 비교를 위해 test.c 를 다음과 같이 수정

```
// test.c
#include <stdio.h>
#include <limits.h>
void add_four_floats(float[], float[]);
void dummy() { return; }
int main() {
  unsigned long i = 0;
```

```
  float x[] = {-29.750, 244.333, 887.29, 48.1E22};
  float y[] = {29.750,  199.333, -8.29,  22.1E23};
  for (i = 0; i < INT_MAX; i++) { // INT_MAX 횟수 반복
    dummy(); add_four_floats(x, y);
  }
  return 0;
}
```

- 비교를 위해 SSE를 사용하지 않은 경우의 코드

```
// test2.c
#include <stdio.h>
#include <limits.h>
void add_four_floats(float *x, float *y) {
  x[0] += y[0];  x[1] += y[1];  x[2] += y[2];  x[3] += y[3];
}
int main() {
  unsigned long i = 0;
  float x[] = {-29.750, 244.333, 887.29, 48.1E22};
  float y[] = {29.750,  199.333, -8.29,  22.1E23};
  for (i = 0; i < INT_MAX; i++) { // INT_MAX 횟수 반복
    add_four_floats(x, y);
  }
  return 0;
}
```

- 긴 배열을 연산하는 경우 SSE 연산을 사용하지 않고 위와 같이 작성하게 되면, 불필요한 함수 호출이 발생하여 오버헤드가 발생하게 된다. 이 점을 고려하여 SSE를 적용한 test.c 에서는 dummy() 함수를 호출하게 하여 최대한 같은 조건으로 비교하려 하였음.
- time 명령으로 실행시간을 보면 실제로 SSE를 적용한 경우 실행 시간이 짧게 측정됨

SSE 적용 (6.1초)	SSE 적용하지 않음 (9.7초)
$ time ./test	$ time ./test2
real 0m6.184s	real 0m9.762s
user 0m6.180s	user 0m9.762s
sys 0m0.005s	sys 0m0.001s

- 실제로 여러번 반복 실행 후 정리한 결과는 다음과 같다. (5회 실행한 평균 실행시간 기준으로 SSE를 사용한 경우 27%정도 실행 시간이 감소함)

구분	1회	2회	3회	4회	5회	min	average
SSE 사용	6.213	6.148	5.731	5.732	5.734	5.731	5.9116 (63%)
SSE 사용안함	9.275	9.294	9.285	9.289	9.714	9.275	9.3714 (100%)

6. AVX

6.1 개요

- AVX(Advanced Vector Extensions, 고급 백터 확장)[21]는 2008년 인텔의 개발자 포럼에서 발표되었고, 2011년 Sandy Bridge 프로세서에 처음으로 적용됨. AVX2는 2013년 Haswell 에서 도입됨. AVX-512는 AVX를 512비트 지원으로 확장한 것으로 2013년에 발표되고 2016 년 출시한 Knights Landing 프로세서에 처음 적용됨.
- AVX는 16개의 256비트 YMM 레지스터를 지원. 각 레지스터는 8개의 32비트 단정도 실수연산, 또는 4개의 64비트 배정도 실수 연산을 지원
- 3-operand SIMD 연산을 지원하며, 기존 명령에 VEX(vector extension) coding scheme을 적용하여 확장
- AVX-512는 EVEX prefix 인코딩을 적용하여 기존의 명령을 확장. 4-operand 연산을 지원. 레지스터 폭은 512비트로 증가. 레지스터 개수도 64비트모드에서 32개까지 지원(ZMM0~ZMM31)

표 7. 백터연산 확장 레지스터

연산 확장	512비트	256비트	128비트(하위)	레지스터 개수
SSE/SSE2	-	-	XMM	16
AVX/AVX2	-	YMM	XMM	16
AVX-512	ZMM	YMM	XMM	32

표 8. 2항 연산과 3항 연산의 비교

구분	SSE2	AVX
연산방식	`paddw xmm0, xmm1` `(xmm0 += xmm1)`	`vpaddw xmm2, xmm0, xmm1` `(xmm2 = xmm0 + xmm1)`
C 언어 표현	`b += a`	`c = a + b`

표 9. SSE와 AVX 명령어 차이

명령어	SSE	AVX	비고
실수 데이터 이동	`movups xmm0, [eax+esi]`	`vmovups ymm0, [eax+esi]`	256비트 레지스터
실수형 덧셈	`addups xmm0, xmm1`	`vaddups ymm2, ymm1, ymm0`	256비트 레지스터
실수형 곱셈	`mulps xmm0, xmm1`	`vmulps ymm2, ymm1, ymm0`	256비트 레지스터
정수 데이터 이동	`movdqu xmm0, [eax+esi]`	`vmovdqu xmm0, [eax+esi]`	-
정수형 곱셈	`pmullw xmm0, xmm1`	`vpmullw xmm2, xmm0, xmm1`	3항 연산
정수형 뺄셈	`psubw xmm2, xmm0`	`vpsubw xmm2, xmm2, xmm0`	3항 연산
정수형 덧셈	`paddw xmm2, xmm1`	`vpaddw xmm2, xmm2, xmm1`	3항 연산

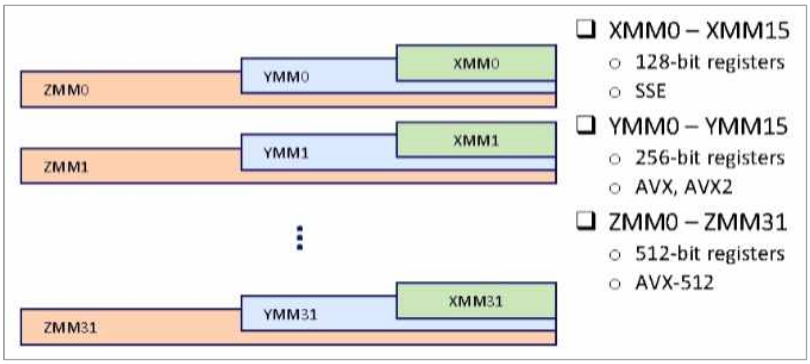


그림 5. SIMD 레지스터

21) https://en.wikipedia.org/wiki/Advanced_Vector_Extensions

6.2 예제

```
// main.c
#include <stdio.h>
#include <inttypes.h>

// copy float vector 32bit * 8 = 256
void copy_value_256b(float *src, float *dst);

  __attribute__((aligned(32))) float SrcAligned[8] = {1,2,3,4,5,6,7,8};
  __attribute__((aligned(32))) float DstAligned[8] = {0};

int main() {
  copy_value_256b(SrcAligned, DstAligned);
  for (int i = 0; i < 8; i++) {
    printf("%d  %f  %f \n", i, SrcAligned[i], DstAligned[i]);
  }
  return 0;
}
```

```
// func.s
    .text
    .globl    copy_value_256b
copy_value_256b:
        .cfi_startproc
        endbr64
        vmovss  (%rdi), %xmm0
        vmovss  %xmm0, (%rsi)
        vmovss  4(%rdi), %xmm0
        vmovss  %xmm0, 4(%rsi)
        vmovss  8(%rdi), %xmm0
        vmovss  %xmm0, 8(%rsi)
        vmovss  12(%rdi), %xmm0
        vmovss  %xmm0, 12(%rsi)
        vmovss  16(%rdi), %xmm0
        vmovss  %xmm0, 16(%rsi)
        vmovss  20(%rdi), %xmm0
        vmovss  %xmm0, 20(%rsi)
        vmovss  24(%rdi), %xmm0
        vmovss  %xmm0, 24(%rsi)
        vmovss  28(%rdi), %xmm0
        vmovss  %xmm0, 28(%rsi)
        ret
        .cfi_endproc
```

```
// do compile
gcc -g -o main.o  -c main.c
as -g -o func.o func.s
gcc -g -o exe  func.o  main.o
```

```
$ ./exe
0  1.000000  1.000000
1  2.000000  2.000000
2  3.000000  3.000000
3  4.000000  4.000000
4  5.000000  5.000000
5  6.000000  6.000000
6  7.000000  7.000000
7  8.000000  8.000000
```

7. 인스트럭션 디코딩

7.1 디코딩 예제

- 위에서 작성된 main.o의 오브젝트 코드를 보자

```
$ objdump  -d main.o
main.o:     file format elf64-x86-64
Disassembly of section .text:
0000000000000000 <main>:
   0:  f3 0f 1e fa           endbr64
   4:  55                    push   %rbp
   5:  48 89 e5              mov    %rsp,%rbp
   8:  48 83 ec 10           sub    $0x10,%rsp
   c:  48 8d 35 00 00 00 00  lea    0x0(%rip),%rsi        # 13 <main+0x13>
  13:  48 8d 3d 00 00 00 00  lea    0x0(%rip),%rdi        # 1a <main+0x1a>
  1a:  e8 00 00 00 00        callq  1f <main+0x1f>
  1f:  c7 45 fc 00 00 00 00  movl   $0x0,-0x4(%rbp)
  26:  eb 54                 jmp    7c <main+0x7c>
  28:  8b 45 fc              mov    -0x4(%rbp),%eax
  2b:  48 98                 cltq
  2d:  48 8d 14 85 00 00 00  lea    0x0(,%rax,4),%rdx
  34:  00
  35:  48 8d 05 00 00 00 00  lea    0x0(%rip),%rax        # 3c <main+0x3c>
  3c:  f3 0f 10 04 02        movss  (%rdx,%rax,1),%xmm0
  41:  f3 0f 5a c8           cvtss2sd %xmm0,%xmm1
  45:  8b 45 fc              mov    -0x4(%rbp),%eax
  48:  48 98                 cltq
  4a:  48 8d 14 85 00 00 00  lea    0x0(,%rax,4),%rdx
  51:  00
  52:  48 8d 05 00 00 00 00  lea    0x0(%rip),%rax        # 59 <main+0x59>
  59:  f3 0f 10 04 02        movss  (%rdx,%rax,1),%xmm0
  5e:  f3 0f 5a c0           cvtss2sd %xmm0,%xmm0
  62:  8b 45 fc              mov    -0x4(%rbp),%eax
  65:  89 c6                 mov    %eax,%esi
  67:  48 8d 3d 00 00 00 00  lea    0x0(%rip),%rdi        # 6e <main+0x6e>
  6e:  b8 02 00 00 00        mov    $0x2,%eax
  73:  e8 00 00 00 00        callq  78 <main+0x78>
  78:  83 45 fc 01           addl   $0x1,-0x4(%rbp)
  7c:  83 7d fc 07           cmpl   $0x7,-0x4(%rbp)
  80:  7e a6                 jle    28 <main+0x28>
  82:  b8 00 00 00 00        mov    $0x0,%eax
  87:  c9                    leaveq
  88:  c3                    retq
```

각 명령어 별로 SDM문서를 참조하여 분석해 보자.

```
  0:  f3 0f 1e fa           endbr64
```

- endbr64 명령은 Intel Control-Flow Enforcement Technology(CET) 를 위한 명령으로 SDM Volume 1-Ch18을 참고한다.
- SDM Volume 2을 참고한다. endbr64는 64비트 모드에서 indirect branch를 종료하는 명령이며 opcode는 F3 0F 1E FA 임을 알 수 있고, operand는 필요하지 않음을 알 수 있다.

ENDBR64—Terminate an Indirect Branch in 64-bit Mode

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 1E FA ENDBR64	ZO	V/V	CET_IBT	Terminate indirect branch in 64 bit mode.

Instruction Operand Encoding

Op/En	Tuple Type	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA	NA

```
4:   55                      push   %rbp
```

- SDM을 참고하면 다음과 같다. `55`는 `50+5`으로 아래 표에서 `50+rw` 또는 `50+rd`에 해당됨
- `Op/En`(operation encoding) 컬럼은 `O`인데, `opcode+rd(r)` 즉, opcode의 하위 3비트가 레지스터를 지정하는데 사용됨.

PUSH—Push Word, Doubleword or Quadword Onto the Stack

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
FF /6	PUSH *r/m16*	M	Valid	Valid	Push *r/m16*.
FF /6	PUSH *r/m32*	M	N.E.	Valid	Push *r/m32*.
FF /6	PUSH *r/m64*	M	Valid	N.E.	Push *r/m64*.
50+*rw*	PUSH *r16*	O	Valid	Valid	Push *r16*.
50+*rd*	PUSH *r32*	O	N.E.	Valid	Push *r32*.
50+*rd*	PUSH *r64*	O	Valid	N.E.	Push *r64*.
6A *ib*	PUSH *imm8*	I	Valid	Valid	Push *imm8*.

- Table 3-1을 참고하면 +rd(5)에 해당하는 레지스터는 EBP임.

Table 3-1. Register Codes Associated With +rb, +rw, +rd, +ro

byte register +rb			word register +rw			dword register +rd			quadword register (64-Bit Mode only) +ro		
Register	REX.B	Reg Field	Register	REX.B	Reg Field	Register	REX.B	Reg Field	Register	REX.B	Reg Field
AL	None	0	AX	None	0	EAX	None	0	RAX	None	0
CL	None	1	CX	None	1	ECX	None	1	RCX	None	1
DL	None	2	DX	None	2	EDX	None	2	RDX	None	2
BL	None	3	BX	None	3	EBX	None	3	RBX	None	3
AH	Not encodable (N.E.)	4	SP	None	4	ESP	None	4	N/A	N/A	N/A
CH	N.E.	5	BP	None	5	EBP	None	5	N/A	N/A	N/A
DH	N.E.	6	SI	None	6	ESI	None	6	N/A	N/A	N/A
BH	N.E.	7	DI	None	7	EDI	None	7	N/A	N/A	N/A
SPL	Yes	4	SP	None	4	ESP	None	4	RSP	None	4
BPL	Yes	5	BP	None	5	EBP	None	5	RBP	None	5

```
5:   48 89 e5                mov    %rsp,%rbp
```

- `48`은 REX prefix로 64-비트에서만 사용됨. (SDM 2.2.1절 참고). `48`은 `40+8`임. `8H=1000b` 즉 `W=1` 인 상태, `REX.W`를 의미(64비트 operand)[22]

Table 2-4. REX Prefix Fields [BITS: 0100WRXB]

Field Name	Bit Position	Definition
-	7:4	0100
W	3	0 = Operand size determined by CS.D
		1 = 64 Bit Operand Size
R	2	Extension of the ModR/M reg field
X	1	Extension of the SIB index field
B	0	Extension of the ModR/M r/m field, SIB base field, or Opcode reg field

- `89`는 `MOV` 명령의 opcode이며, SDM을 참고하면 다음과 같다. `Op/En` 은 `MR`로 Operand1은 `ModRM:r/m(w)`, Operand2는 `ModRM:reg(r)`, 즉 operand2인 레지스터에서 read후 operand1인 레지스터/메모리에 write하는 것을 의미.

22) https://wiki.osdev.org/X86-64_Instruction_Encoding#Encoding

MOV—Move

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
88 /r	MOV *r/m8,r8*	MR	Valid	Valid	Move *r8* to *r/m8*.
REX + 88 /r	MOV *r/m8***,r8****	MR	Valid	N.E.	Move *r8* to *r/m8*.
89 /r	MOV *r/m16,r16*	MR	Valid	Valid	Move *r16* to *r/m16*.
89 /r	MOV *r/m32,r32*	MR	Valid	Valid	Move *r32* to *r/m32*.
REX.W + 89 /r	MOV *r/m64,r64*	MR	Valid	N.E.	Move *r64* to *r/m64*.
8A /r	MOV *r8,r/m8*	RM	Valid	Valid	Move *r/m8* to *r8*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
MR	ModRM:r/m (w)	ModRM:reg (r)	NA	NA
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA
FD	AL/AX/EAX/RAX	Moffs	NA	NA
TD	Moffs (w)	AL/AX/EAX/RAX	NA	NA
OI	opcode + rd (w)	imm8/16/32/64	NA	NA
MI	ModRM:r/m (w)	imm8/16/32/64	NA	NA

- e5 (1110 0101b)는 ModR/M 값으로, Mod=11, R/M=101이므로, Table2-2에 의하면 Operand1=EBP, Operand2=ESP(64비트 모드이므로 각각 RBP, RSP) 레지스터를 의미함.

Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

r8(/r) r16(/r) r32(/r) mm(/r) xmm(/r) (In decimal) /digit (Opcode) (In binary) REG =			AL AX EAX MM0 XMM0 0 000	CL CX ECX MM1 XMM1 1 001	DL DX EDX MM2 XMM2 2 010	BL BX EBX MM3 XMM3 3 011	AH SP ESP MM4 XMM4 4 100	CH BP EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111
Effective Address	**Mod**	**R/M**	**Value of ModR/M Byte (in Hexadecimal)**							
[EAX] [ECX] [EDX] [EBX] [--][--]1 disp32[2] [ESI] [EDI]	00	000 001 010 011 100 101 110 111	00 01 02 03 04 05 06 07	08 09 0A 0B 0C 0D 0E 0F	10 11 12 13 14 15 16 17	18 19 1A 1B 1C 1D 1E 1F	20 21 22 23 24 25 26 27	28 29 2A 2B 2C 2D 2E 2F	30 31 32 33 34 35 36 37	38 39 3A 3B 3C 3D 3E 3F
[EAX]+disp8[3] [ECX]+disp8 [EDX]+disp8 [EBX]+disp8 [--][--]+disp8 [EBP]+disp8 [ESI]+disp8 [EDI]+disp8	01	000 001 010 011 100 101 110 111	40 41 42 43 44 45 46 47	48 49 4A 4B 4C 4D 4E 4F	50 51 52 53 54 55 56 57	58 59 5A 5B 5C 5D 5E 5F	60 61 62 63 64 65 66 67	68 69 6A 6B 6C 6D 6E 6F	70 71 72 73 74 75 76 77	78 79 7A 7B 7C 7D 7E 7F
[EAX]+disp32 [ECX]+disp32 [EDX]+disp32 [EBX]+disp32 [--][--]+disp32 [EBP]+disp32 [ESI]+disp32 [EDI]+disp32	10	000 001 010 011 100 101 110 111	80 81 82 83 84 85 86 87	88 89 8A 8B 8C 8D 8E 8F	90 91 92 93 94 95 96 97	98 99 9A 9B 9C 9D 9E 9F	A0 A1 A2 A3 A4 A5 A6 A7	A8 A9 AA AB AC AD AE AF	B0 B1 B2 B3 B4 B5 B6 B7	B8 B9 BA BB BC BD BE BF
EAX/AX/AL/MM0/XMM0 ECX/CX/CL/MM1/XMM1 EDX/DX/DL/MM2/XMM2 EBX/BX/BL/MM3/XMM3 ESP/SP/AH/MM4/XMM4 EBP/BP/CH/MM5/XMM5 ESI/SI/DH/MM6/XMM6 EDI/DI/BH/MM7/XMM7	11	000 001 010 011 100 101 110 111	C0 C1 C2 C3 C4 C5 C6 C7	C8 C9 CA CB CC CD CE CF	D0 D1 D2 D3 D4 D5 D6 D7	D8 D9 DA DB DC DD DE DF	E0 E1 E2 E3 E4 E5 E6 E7	E8 E9 EA EB EC ED EE EF	F0 F1 F2 F3 F4 F5 F6 F7	F8 F9 FA FB FC FD FE FF

```
 8:   48 83 ec 10            sub    $0x10,%rsp
```

- 48은 앞에서 설명하였으며 REX.W (64-bit operand size)를 의미
- 83은 SUB(subtract)의 opcode임. Op/En 은 MI 로 Operand1은 ModRM:r/m(r,w), Operand2는 imm8/16/32임

83 /5 *ib*	SUB *r/m16, imm8*	MI	Valid	Valid	Subtract sign-extended *imm8* from *r/m16*.
83 /5 *ib*	SUB *r/m32, imm8*	MI	Valid	Valid	Subtract sign-extended *imm8* from *r/m32*.
REX.W + 83 /5 *ib*	SUB *r/m64, imm8*	MI	Valid	N.E.	Subtract sign-extended *imm8* from *r/m64*.
2D *id*	SUB *RAX, imm32*	I	Valid	Valid	Subtract *imm32* from *RAX*.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
I	AL/AX/EAX/RAX	imm8/16/32	NA	NA
MI	ModRM:r/m (r, w)	imm8/16/32	NA	NA
MR	ModRM:r/m (r, w)	ModRM:reg (r)	NA	NA
RM	ModRM:reg (r, w)	ModRM:r/m (r)	NA	NA

- ec(1110 1100b)는 Mod=11, R/M=100이므로, Table2-2에서 ESP(64비트 모드이므로 RSP)임
- 10은 이미디어트값(imm8)임.

```
 c:   48 8d 35 00 00 00 00   lea    0x0(%rip),%rsi        # 13 <main+0x13>
```

- 48은 앞에서 설명하였으며 REX.W (64-bit operand size)를 의미
- 8D는 LEA(Load Effective Address)의 opcode.

LEA—Load Effective Address

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
8D /r	LEA *r16,m*	RM	Valid	Valid	Store effective address for *m* in register *r16*.
8D /r	LEA *r32,m*	RM	Valid	Valid	Store effective address for *m* in register *r32*.
REX.W + 8D /r	LEA *r64,m*	RM	Valid	N.E.	Store effective address for *m* in register *r64*.

- 35(0011 0101b)는 Mod=00, R/M=101이므로, Table2-2에서 disp32를 의미.
- 00 00 00 00 은 32-bit displacement 값이다.

```
1a:   e8 00 00 00 00         callq  1f <main+0x1f>
```

- e8은 CALL(call procedure)의 opcode임. opcode 뒤의 cd는 code offset double word(4-byte)를 의미. code segment register의 값을 의미.

CALL—Call Procedure

Opcode	Instruction	Op/ En	64-bit Mode	Compat/ Leg Mode	Description
E8 *cw*	CALL *rel16*	D	N.S.	Valid	Call near, relative, displacement relative to next instruction.
E8 *cd*	CALL *rel32*	D	Valid	Valid	Call near, relative, displacement relative to next instruction. 32-bit displacement sign extended to 64-bits in 64-bit mode.

- 00 00 00 00 은 rel32 값임.

```
1f:   c7 45 fc 00 00 00 00   movl   $0x0,-0x4(%rbp)
```

- C7은 MOVE 명령의 opcode. ib/iw/id/io 는 1-byte(ib), 2-byte(iw), 4-byte(id), 8-byte(io) immediate operand 를 의미

C7 /0 *iw*	MOV *r/m16, imm16*	MI	Valid	Valid	Move *imm16* to *r/m16*.
C7 /0 *id*	MOV *r/m32, imm32*	MI	Valid	Valid	Move *imm32* to *r/m32*.
REX.W + C7 /0 *id*	MOV *r/m64, imm32*	MI	Valid	N.E.	Move *imm32 sign extended to 64-bits* to *r/m64*.

- 45(0100 0101b)는 Mod=01, R/M=101이므로, [EBP]+disp8를 의미.
- FC(1111 1100b)는 signed int 값으로 -4 임.
- 00 00 00 00 는 imm32 값임
- 앞에서 rbp 레지스터의 값을 스택에 보관(push)한 후 스택 포인터(rsp)의 값을 rbp에 복사한 후 스택 포인터를 0x10만큼 감소함으로써 스택의 크기를 증가하였다. rbp에는 스택 포인터가 감소하기 전의 값이 저장되어 있으므로 rbp를 기준으로 0x10바이트 만큼의 영역을 local variable로 사용하기 위함이다. (C코드에서 로컬 변수 i에 0을 초기화함)

```
26:   eb 54                  jmp    7c <main+0x7c>
```

- EB는 JMP(Jump) 명령의 opcode 임. RIP(instruction pointer)를 기준으로 8비트 변위.

JMP—Jump

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
EB *cb*	JMP *rel8*	D	Valid	Valid	Jump short, RIP = RIP + 8-bit displacement sign extended to 64-bits

- 54는 imm8 값으로 JMP의 다음 명령 주소(28h)를 기준으로함. 28h+54h=7Ch 이므로, 이 위치에 존재하는 cmpl 명령으로 점프함.

- copy_value_256b 함수의 역어셈블러 결과는 다음과 같다.

```
$ objdump  -d  func.o
func.o:     file format elf64-x86-64
Disassembly of section .text:
0000000000000000 <copy_value_256b>:
   0:   f3 0f 1e fa             endbr64
   4:   c5 fa 10 07             vmovss (%rdi),%xmm0
```

```
 8:   c5 fa 11 06           vmovss %xmm0,(%rsi)
 c:   c5 fa 10 47 04        vmovss 0x4(%rdi),%xmm0
11:   c5 fa 11 46 04        vmovss %xmm0,0x4(%rsi)
16:   c5 fa 10 47 08        vmovss 0x8(%rdi),%xmm0
1b:   c5 fa 11 46 08        vmovss %xmm0,0x8(%rsi)
20:   c5 fa 10 47 0c        vmovss 0xc(%rdi),%xmm0
25:   c5 fa 11 46 0c        vmovss %xmm0,0xc(%rsi)
2a:   c5 fa 10 47 10        vmovss 0x10(%rdi),%xmm0
2f:   c5 fa 11 46 10        vmovss %xmm0,0x10(%rsi)
34:   c5 fa 10 47 14        vmovss 0x14(%rdi),%xmm0
39:   c5 fa 11 46 14        vmovss %xmm0,0x14(%rsi)
3e:   c5 fa 10 47 18        vmovss 0x18(%rdi),%xmm0
43:   c5 fa 11 46 18        vmovss %xmm0,0x18(%rsi)
48:   c5 fa 10 47 1c        vmovss 0x1c(%rdi),%xmm0
4d:   c5 fa 11 46 1c        vmovss %xmm0,0x1c(%rsi)
52:   c3                    retq
```

```
 4:   c5 fa 10 07           vmovss (%rdi),%xmm0
```

- C5 FA 10은 VMOVSS(move or merge scalar single-precision floating-point value)명령의 opcode임.

MOVSS—Move or Merge Scalar Single-Precision Floating-Point Value

Opcode/ Instruction	Op / En	64/32 bit Mode Support	CPUID Feature Flag	Description
F3 0F 10 /r MOVSS xmm1, xmm2	A	V/V	SSE	Merge scalar single-precision floating-point value from xmm2 to xmm1 register.
F3 0F 10 /r MOVSS xmm1, m32	A	V/V	SSE	Load scalar single-precision floating-point value from m32 to xmm1 register.
VEX.LIG.F3.0F.WIG 10 /r VMOVSS xmm1, xmm2, xmm3	B	V/V	AVX	Merge scalar single-precision floating-point value from xmm2 and xmm3 to xmm1 register
VEX.LIG.F3.0F.WIG 10 /r VMOVSS xmm1, m32	D	V/V	AVX	Load scalar single-precision floating-point value from m32 to xmm1 register.

- C5 FA는 VEX Prefix임.[23] SDM Vol2A 2.3.5절을 참고하면 2-byte VEX prefix의 처음 바이트는 C5(1100 0101b)임. 두번째 바이트인 FA(1111 1010b)는 R/vvvv/L/pp로 구성됨. R=1은 REX.R=0 임을 의미. vvvv는 source or dest 레지스터로 Vol 2A 2.3.5.6절(Table 2-8)을 참고. vvvv=1111은 XMM0/YMM0에 맵핑됨. L=0은 vector length 128비트를 의미. pp=10은 opcode extension 으로 SIMD prefix F3에 해당함.
- 07(0000 0111b)는 Mod=00, R/M=111이므로, [EDI], xmm0 에 해당

```
 8:   c5 fa 11 06           vmovss %xmm0,(%rsi)
```

- C5 FA 10은 VMOVSS명령의 opcode임.
- 06(0000 0110b)는 Mod=00, R/M=110이므로, [ESI], XMM0 에 해당

```
 c:   c5 fa 10 47 04        vmovss 0x4(%rdi),%xmm0
```

- C5 FA 10은 VMOVSS명령의 opcode임.
- 47(0100 0111b)는 Mod=01, R/M=111이므로, [EDI]+disp8, XMM0 에 해당. 04는 disp8 임

```
52:   c3                    retq
```

- C3는 RET(return from procedure) 명령의 opcode 임

RET—Return from Procedure

Opcode*	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
C3	RET	ZO	Valid	Valid	Near return to calling procedure.

23) https://en.wikipedia.org/wiki/VEX_prefix

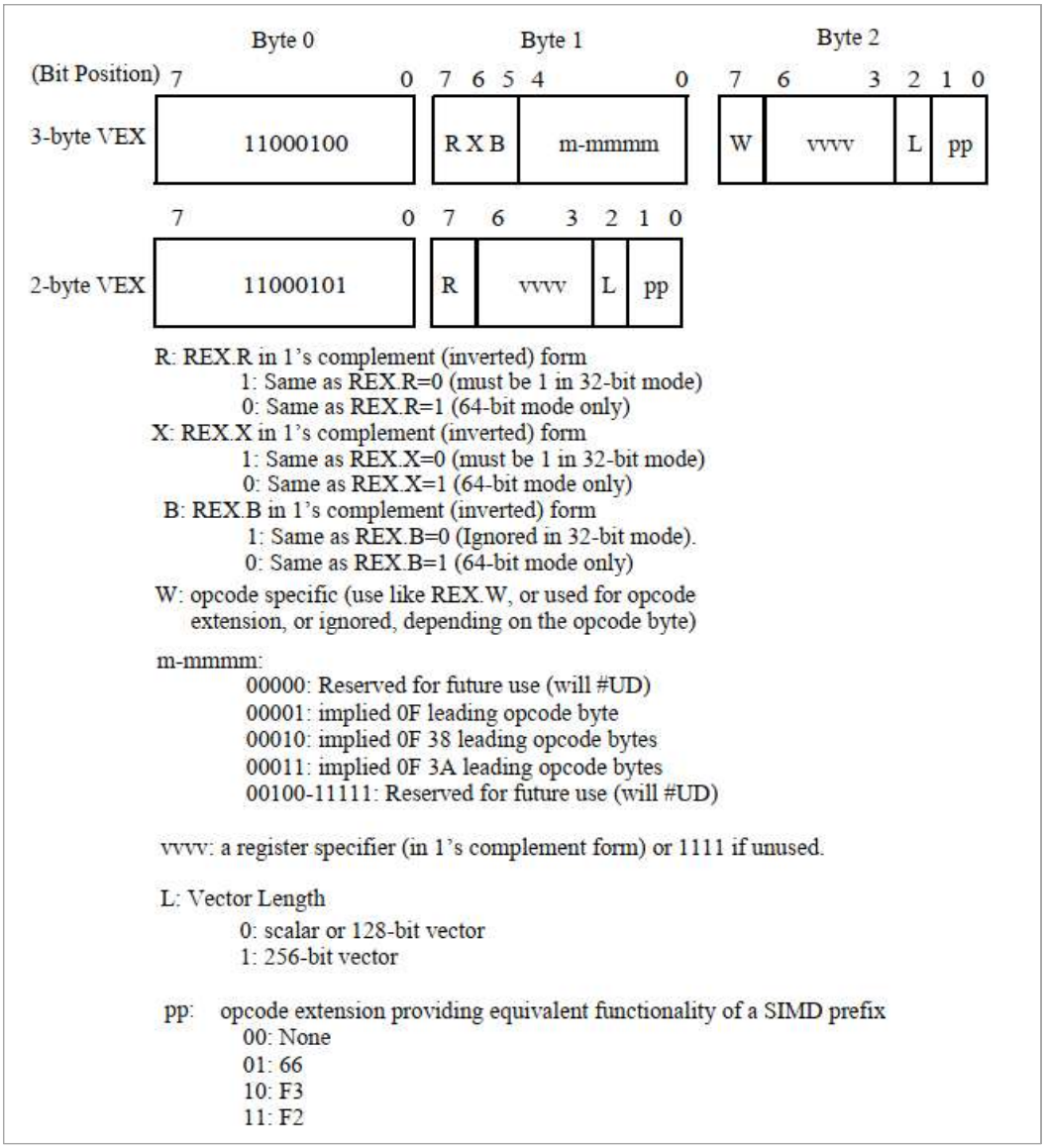
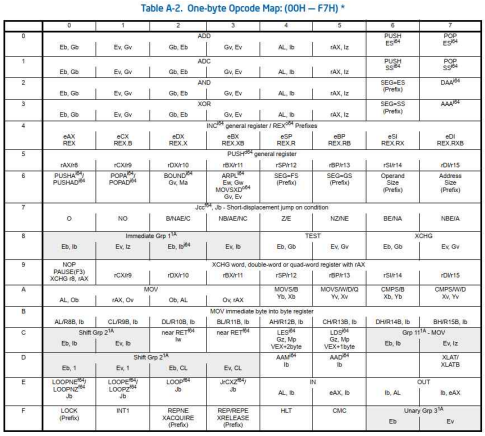


그림 6. VEX Extension Prefix

7.2 OPCODE 테이블 참조

- SDM 문서 Vol 2D A에는 Opcode Table이 있음



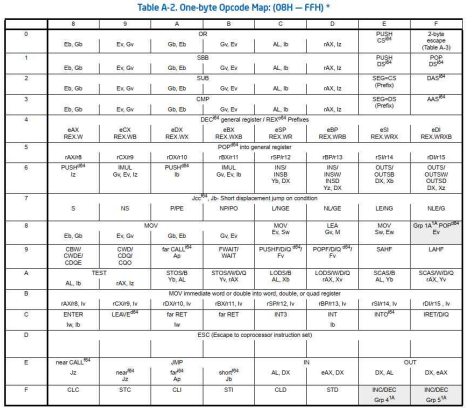


그림 7. One-byte Opcode Map

- 예를 들어 위에서 push명령의 코드는 다음과 같다.

```
4:  55                      push   %rbp
```

- opcode map의 row, column 에서 55를 찾으면 다음과 같다. 해당되는 명령은 PUSH general register이며 rBP/r13 즉 RBP 또는 R13 레지스터를 스텍 푸쉬하는 명령이다. REX.B 속성값이 있으면 R13이 대상이 되나, 이 명령에는 REX prefix 가 없으므로 RBP가 operand가 됨.

	0	1	2	3	4	5	6	7
0	ADD Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	PUSH ES^{i64}	POP ES^{i64}
1	ADC Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	PUSH SS^{i64}	POP SS^{i64}
2	AND Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	SEG=ES (Prefix)	DAA^{i64}
3	XOR Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz	SEG=SS (Prefix)	AAA^{i64}
4	INC^{i64} general register / REX^{o64} Prefixes							
	eAX REX	eCX REX.B	eDX REX.X	eBX REX.XB	eSP REX.R	eBP REX.RB	eSI REX.RX	eDI REX.RXB
5	PUSH^{o64} general register							
	rAX/r8	rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
6	PUSHA^{i64}/	POPA^{i64}/	BOUND^{i64}	ARPL^{i64}	SEG=FS	SEG=GS	Operand	Address

8. X87 FPU 제어

8.1 소개

- x87 FPU(floating-point unit)는 x86 시스템에서 사용되는 부동소수점 연산 장치임. 80486 (1989년출시) 이전의 프로세서에는 CPU와 별도로 분리된 프로세서(8087, 80287, 80487 등) 였으나, 이후에는 CPU에 통합됨.
- 부동소수점 연산 뿐만 아니라, 루트, 삼각함수 연산을 자체적으로 지원.
- x87 FPU는 8개의 ST레지스터(ST(0)~ST(7))를 사용함. ST레지스터는 80비트이며, ST레지스터의 하위 64비트를 MMX에서 사용함.
- ST(x) 레지스터는 스텍과 같이 동작하며, 인덱스를 통한 직접 접근도 가능.
- X87 FPU 기본 연산 명령 (SDM V1 5.2절, 8장)

	실수	정수	비고
데이터이동	FLD FST FSTP	FILD FIST FISTP	-P는 pop
	FBLD FBSTP		BCD(binary-coded decimal)
	FXCH		exchange registers
	FCMOVE FCMOVENE FCMOVEB FCMOVBE FCMOVNB FCMOVNBE FCMOVU FCMOVNU		conditional move E (equal), NE(not equal) B (below), BE(below or equal) U (unordered), NU(not unordered)

연산

구분	실수	정수	비고
덧셈	FADD FADDP	FIADD	-P는 pop
뺄셈	FSUB FSUBP FISUBRP	FISUB FISUBR	-R 은 reverse
곱셈	FMUL FMULP	FIMUL	
나누기	FDIV FDIVP FDIVR FDIVRP	FIDIV FIDIVR	
기타	FPREM FPREM1	partial remainder	
	FABS	absolute value	
	FCHS	change sign	
	FRNDINT	round to integer	
	FSCALE	sacale by power of two	
	FSQRT	sqare root	
	FXTRACT	extract exponent and significand	

비교

구분	실수	정수	비고
비교	FCOM FCOMP FCOMPP	FICOM FICOMP	-P는 pop
	FUCOM FUCOMP FUCOMPP		U는 unordered compare
	FCOMI FCOMIP FUCOMI FUCOMIP		비교후 EFLAGS설정
기타	FTST	Test floating-point	
	FXAM	examine floating-point	

기타

구분	명령
load constant	FLD1 (+1.0) FLDZ (0.0) FLDPI (PI) FLDL2E ($\log_2 e$) FLDL2T ($\log_2 10$) FLDLG2 ($\log_{10} 2$)
삼각함수	FSIN (sin) FCOS (Cosine) FSINCOS (sine and cosine) FPTAN (partial tangent) FPATAN (arctangent) F2XM1 (2^x-1) FYL2X ($y*\log_2 x$) FYL2XP1 ($y*\log_2(x+1)$)

8.2 예제

- gcc에서 -mfpmath=387 옵션을 사용하면 x87 FPU를 사용한다.
- 다음 C 코드를 작성하고, 어셈블리어로 변환

```
#include <stdio.h>
int calc(int n, double *x, double *y, double *sum) {
  int i;
  double s = 0;
  for (i = 0; i < n; i++) {
     s += x[i] * y[i];
  }
  *sum = s;
  return 0;
}
int main() {
  int i, r;
  int n = 4;
  double x[] = { 1,2,3,4 };
  double y[] = { 6,5,7,10 };
  double sum;
```

```
  r = calc(n, x, y, &sum);
  for (i = 0; i < n; i++) printf(" %f %f \n", x[i], y[i]);
  printf(" r=%d  %f  \n", r, sum);
  return 0;
}
```

```
# -mfpmath=387 옵션을 사용
gcc -mfpmath=387 -S main.c
gcc -g -o main  main.s
```

```
$ ./main
 1.000000 6.000000
 2.000000 5.000000
 3.000000 7.000000
 4.000000 10.000000
 r=0  77.000000
```

- 어셈블리 코드는 다음과 같다. (일부분으로 FPU 스택 명령은 파란색으로 표시함)

```
calc:
.LFB0:
    .cfi_startproc
    endbr64
    pushq  %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq   %rsp, %rbp
    .cfi_def_cfa_register 6
    movl   %edi, -20(%rbp) # n
    movq   %rsi, -32(%rbp) # x
    movq   %rdx, -40(%rbp) # y
    movq   %rcx, -48(%rbp) # sum
    fldz
    fstpl  -8(%rbp) # s = 0
    movl   $0, -12(%rbp) # i = 0
    jmp .L2
.L3:
    movl   -12(%rbp), %eax # i
    cltq
    leaq   0(,%rax,8), %rdx
    movq   -32(%rbp), %rax # x
    addq   %rdx, %rax # x[i]
    fldl   (%rax)   # x[i] -> st(0)  # STACK : x[i]
    movl   -12(%rbp), %eax # i
    cltq
    leaq   0(,%rax,8), %rdx
    movq   -40(%rbp), %rax # y
    addq   %rdx, %rax # y[i]
    fldl   (%rax) # y[i] -> st(0)  # STACK : y[i], x[i]
    fmulp  %st, %st(1)  # STACK : y[i]*x[i]
    fldl   -8(%rbp) # s
 # STACK : s, y[i]*x[i]
    faddp  %st, %st(1)
 # STACK : s + y[i]*x[i]
    fstpl  -8(%rbp) # s += y[i]*x[i]
    addl   $1, -12(%rbp)
```

```
.L2:
    movl   -12(%rbp), %eax
    cmpl   -20(%rbp), %eax # compare i < n
    jl  .L3
    movq   -48(%rbp), %rax # sum
    fldl   -8(%rbp) # s
    fstpl  (%rax)  # *sum = s
    movl   $0, %eax # return 0
    popq   %rbp
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
```

- 위 코드를 gdb로 트레이스 함.

```
$ gdb main
(gdb) start
Temporary breakpoint 1 at 0x11e0: file main.s, line 65.
Starting program: /home/kisti/asm/main

Temporary breakpoint 1, main () at main.s:65
65      endbr64
(gdb) disas calc
Dump of assembler code for function calc:
   0x0000555555555169 <+0>:   endbr64
   0x000055555555516d <+4>:   push   %rbp
   0x000055555555516e <+5>:   mov    %rsp,%rbp
   0x0000555555555171 <+8>:   mov    %edi,-0x14(%rbp)
   0x0000555555555174 <+11>:  mov    %rsi,-0x20(%rbp)
   0x0000555555555178 <+15>:  mov    %rdx,-0x28(%rbp)
   0x000055555555517c <+19>:  mov    %rcx,-0x30(%rbp)
   0x0000555555555180 <+23>:  fldz
   0x0000555555555182 <+25>:  fstpl  -0x8(%rbp)
   0x0000555555555185 <+28>:  movl   $0x0,-0xc(%rbp)
   0x000055555555518c <+35>:  jmp    0x5555555551c8 <calc+95>
   0x000055555555518e <+37>:  mov    -0xc(%rbp),%eax
   0x0000555555555191 <+40>:  cltq
   0x0000555555555193 <+42>:  lea    0x0(,%rax,8),%rdx
   0x000055555555519b <+50>:  mov    -0x20(%rbp),%rax
   0x000055555555519f <+54>:  add    %rdx,%rax
   0x00005555555551a2 <+57>:  fldl   (%rax)
   0x00005555555551a4 <+59>:  mov    -0xc(%rbp),%eax
   0x00005555555551a7 <+62>:  cltq
   0x00005555555551a9 <+64>:  lea    0x0(,%rax,8),%rdx
   0x00005555555551b1 <+72>:  mov    -0x28(%rbp),%rax
   0x00005555555551b5 <+76>:  add    %rdx,%rax
   0x00005555555551b8 <+79>:  fldl   (%rax)
   0x00005555555551ba <+81>:  fmulp  %st,%st(1)
   0x00005555555551bc <+83>:  fldl   -0x8(%rbp)
   0x00005555555551bf <+86>:  faddp  %st,%st(1)
   0x00005555555551c1 <+88>:  fstpl  -0x8(%rbp)
   0x00005555555551c4 <+91>:  addl   $0x1,-0xc(%rbp)
   0x00005555555551c8 <+95>:  mov    -0xc(%rbp),%eax
   0x00005555555551cb <+98>:  cmp    -0x14(%rbp),%eax
```

```
   0x00005555555551ce <+101>: jl     0x55555555518e <calc+37>
   0x00005555555551d0 <+103>: mov    -0x30(%rbp),%rax
   0x00005555555551d4 <+107>: fldl   -0x8(%rbp)
   0x00005555555551d7 <+110>: fstpl  (%rax)
   0x00005555555551d9 <+112>: mov    $0x0,%eax
   0x00005555555551de <+117>: pop    %rbp
   0x00005555555551df <+118>: retq
(gdb) b *0x00005555555551a2          // 위의 fld  명령 앞에서 break 설정 후 continue
Breakpoint 2 at 0x5555555551a2: file main.s, line 28.
(gdb) c
Continuing.

Breakpoint 2, calc () at main.s:28
28      fldl   (%rax)
(gdb) info r st0 st1   // FPU stack register 출력
st0            0                   (raw 0x00000000000000000000)
st1            0                   (raw 0x00000000000000000000)
(gdb) si   // 위의 fld (%rax) 명령을 실행  x[0] 값이 st0 에 로드됨
29      movl   -12(%rbp), %eax
(gdb) info r st0 st1
st0            1                   (raw 0x3fff8000000000000000)
st1            0                   (raw 0x00000000000000000000)
(gdb) si
30      cltq
(gdb) si
31      leaq   0(,%rax,8), %rdx
(gdb) si
32      movq   -40(%rbp), %rax
(gdb) si
33      addq   %rdx, %rax
(gdb) si
34      fldl   (%rax)
(gdb) si  // y[0] 값이 st0 에 로드됨
35      fmulp  %st, %st(1)
(gdb) info r st0 st1  // y[0]=6 이 st0에 로드됨. stack 이므로 push됨
st0            6                   (raw 0x4001c000000000000000)
st1            1                   (raw 0x3fff8000000000000000)
(gdb) si  // fmulp 실행   st1*st0 계산 결과를 st0에 저장. stack pop 이 발생
36      fldl   -8(%rbp)
(gdb) info r st0 st1   // 계산결과 6이 저장됨
st0            6                   (raw 0x4001c000000000000000)
st1            0                   (raw 0x00000000000000000000)
(gdb) si   // 변수 s의 값이 st0 에 로드됨
37      faddp  %st, %st(1)
(gdb) info r st0 st1   // 변수 s 의 값 0 이 로드됨
st0            0                   (raw 0x00000000000000000000)
st1            6                   (raw 0x4001c000000000000000)
(gdb) si   // faddp 실행   st0+st1 계산 결과를 st0에 저장. stack pop 이 발생
38      fstpl  -8(%rbp)
(gdb) info r st0 st1    // 0+6 계산 결과가 st0 에 저장.
st0            6                   (raw 0x4001c000000000000000)
st1            0                   (raw 0x00000000000000000000)
 // 이부분에 break 를 걸고 몇번 continue 한후 레지스터 값을 출력
(gdb) b *0x00005555555551d7
```

```
Breakpoint 3 at 0x5555555551d7: file main.s, line 46.
// ... continue, continue, continue, ... 최종결과인 77이 저장됨을 확인
(gdb)  info r st0 st1
st0            77                  (raw 0x40059a00000000000000)
st1            0                   (raw 0x00000000000000000000)
```

[참고 자료]

- Intel 64 and IA-32 Architectures Software Developer's Manual https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html
- Intel Advanced Vector Extensions Programming Reference, June 2011, https://www.intel.com/content/dam/develop/external/us/en/documents/36945
- x64 Assembly and C++ Tutorial, Youtube - Creel, https://www.youtube.com/watch?v=fHE0txCjGgI&list=PL0C5C980A28FEE68D
- NASM - The Netwide Assembler manual, https://www.nasm.us/doc/
- NASM Tutorial, https://cs.lmu.edu/~ray/notes/nasmtutorial/
- System V Application Binary Interface AMD64 Architecture Processor Supplement (With LP64 and ILP32 Programming Models) Version 1.0, https://raw.githubusercontent.com/wiki/hjl-tools/x86-psABI/x86-64-psABI-1.0.pdf
- AMD64 Architecture Programmer's Manual https://developer.amd.com/resources/developer-guides-manuals/
- x86 and amd64 instruction reference, https://www.felixcloutier.com/x86/
- X86-64 명령어 레퍼런스, https://modoocode.com/316
- Modern X86 Assembly Language Programming: 32-bit, 64-bit, SSE, and AVX 1st Edition, Daniel Kusswurm, Apress, https://www.amazon.com/Modern-X86-Assembly-Language-Programming/dp/1484200659
- Compiler Explorer https://godbolt.org/
- Microsoft Macro Assembler reference https://docs.microsoft.com/en-us/cpp/assembler/masm/microsoft-macro-assembler-reference
- Using as The GNU Assembler, https://ftp.gnu.org/old-gnu/Manuals/gas-2.9.1/html_chapter/as_toc.html

본 보고서는 국가과학기술연구회에서 지원한
창의형 융합연구사업(CAP)인 "차세대 초고성능컴퓨터를 위한 이기종
매니코어 하드웨어 시스템 개발" 사업의 결과입니다.

비매품/무료
93560
9 788929 413095
ISBN 978-89-294-1309-5