

Lecture Note 9. Assembler

November 29, 2023

Jongmoo Choi
Dept. of Software
Dankook University

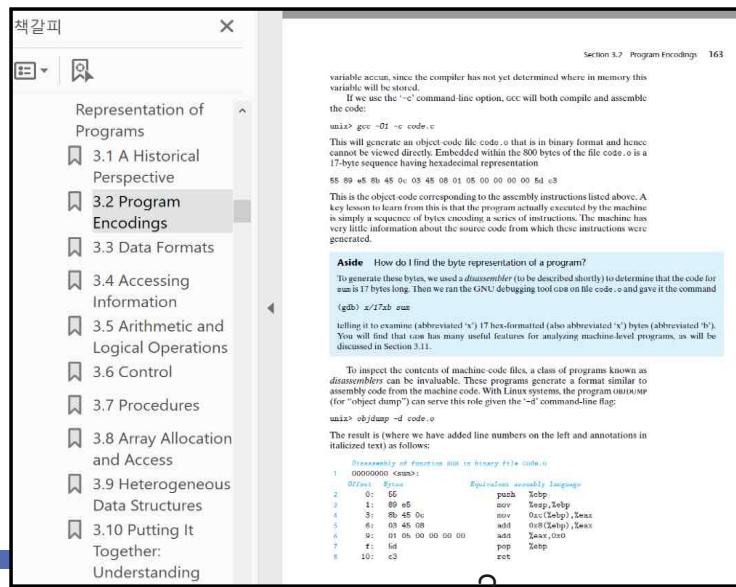
<http://embedded.dankook.ac.kr/~choijm>

(Copyright © 2023 by Jongmoo Choi, All Rights Reserved. Distribution requires permission)

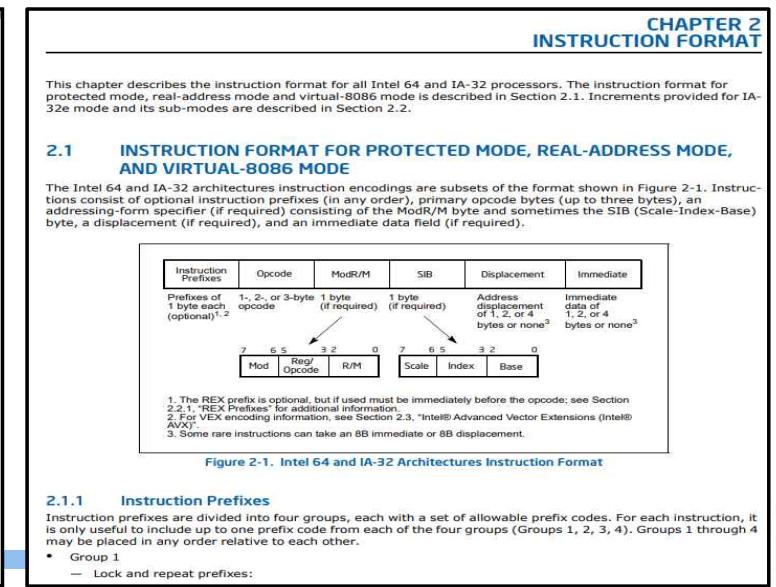
Objectives

- Understand the role of assembler
- Find out the structure of assembler
- Perceive how a HW designer makes a spec. and how a SW designer makes a program based on the spec.
- Know how to use assembly in a high-level language (inline assembly)

- Refer to Chapter 3 in the CSAPP and Intel SW Developer Manual



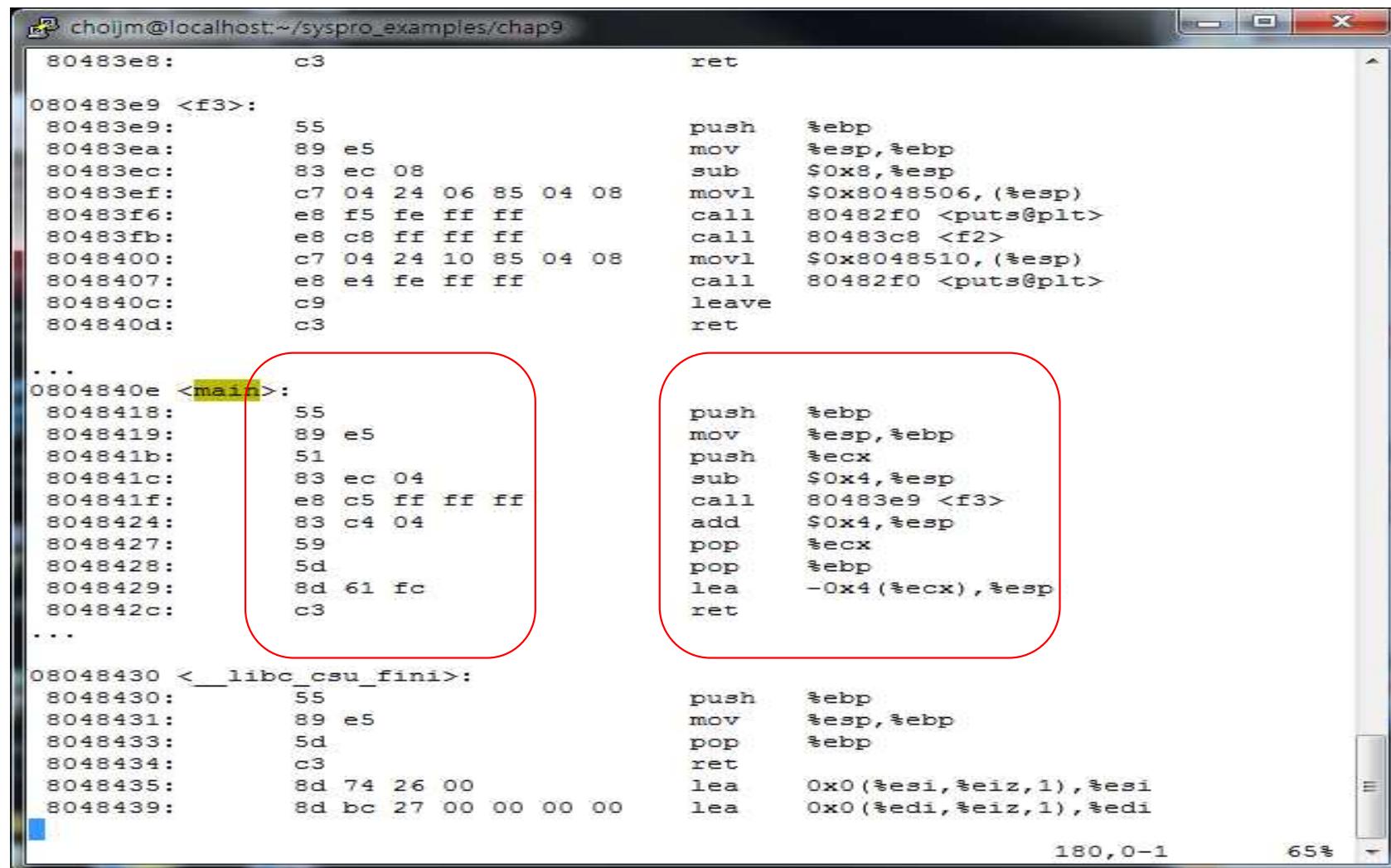
The screenshot shows the 'Program Encodings' section of the CSAPP book. It displays assembly code and its corresponding hex dump. The assembly code includes instructions like `push`, `pop`, and `add`. The hex dump shows the binary representation of these instructions. A sidebar provides an aside on how to find the byte representation of a program.



Role of Assembler (1/1)

■ Assembler

- ✓ Translate assembly language into machine language



```
chojm@localhost:~/syspro_examples/chap9
80483e8:    c3          ret
80483e9 <f3>:
80483e9:    55          push %ebp
80483ea:    89 e5       mov %esp,%ebp
80483ec:    83 ec 08     sub $0x8,%esp
80483ef:    c7 04 24 06 85 04 08   movl $0x8048506,(%esp)
80483f6:    e8 f5 fe ff ff       call 80482f0 <puts@plt>
80483fb:    e8 c8 ff ff ff       call 80483c8 <f2>
8048400:    c7 04 24 10 85 04 08   movl $0x8048510,(%esp)
8048407:    e8 e4 fe ff ff       call 80482f0 <puts@plt>
804840c:    c9          leave
804840d:    c3          ret

...
0804840e <main>:
08048418:    55          push %ebp
08048419:    89 e5       mov %esp,%ebp
0804841b:    51          push %ecx
0804841c:    83 ec 04     sub $0x4,%esp
0804841f:    e8 c5 ff ff ff       call 80483e9 <f3>
08048424:    83 c4 04     add $0x4,%esp
08048427:    59          pop %ecx
08048428:    5d          pop %ebp
08048429:    8d 61 fc     lea -0x4(%ecx),%esp
0804842c:    c3          ret

...
08048430 <__libc_csu_fini>:
08048430:    55          push %ebp
08048431:    89 e5       mov %esp,%ebp
08048433:    5d          pop %ebp
08048434:    c3          ret
08048435:    8d 74 26 00     lea 0x0(%esi,%eiz,1),%esi
08048439:    8d bc 27 00 00 00 00   lea 0x0(%edi,%eiz,1),%edi
```

☞ Understanding a binary is indispensable for detecting virus, plagiarism and SW refactoring

Functionalities of Assembler: 32-bit CPU (1/5)

■ Machine Code

✓ IA-32 machine code format

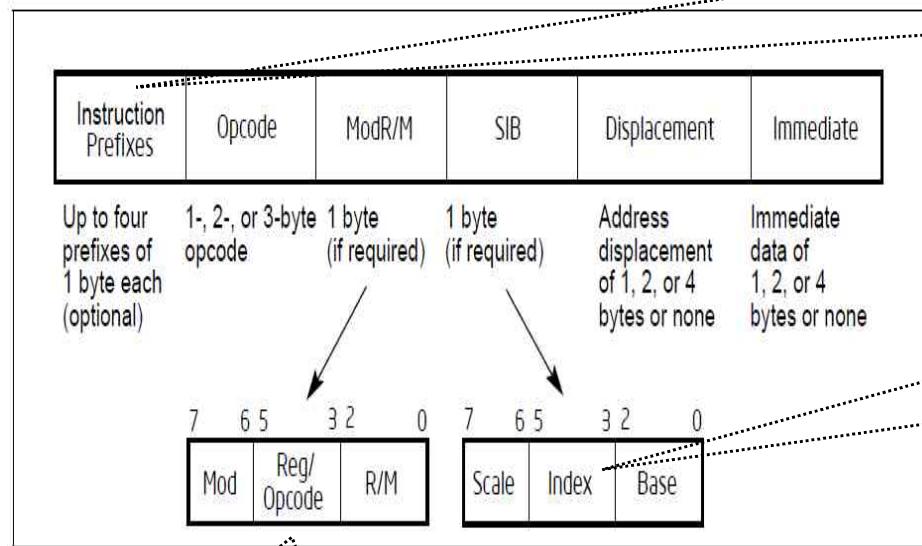


Figure 2.1. Intel 64 and IA-32 Architectures Instruction Format
(from Intel Manual, Volume 2)

Group 1 Lock and repeat prefix:

Group 2 Segment override prefix, Branch hints

Group 3 Operand-size override prefix.

Group 4 Address-size override prefix

- The *scale* field specifies the scale factor.
- The *index* field specifies the register number of the index register.
- The *base* field specifies the register number of the base register.

Mod	R/M or I/B register
00	mem.
01	mem.+dis(8)
10	mem.+dis(32)
11	reg.

Scale	
00	*1
01	*2
10	*4
11	*8

000	EAX	[EAX]
001	ECX	[ECX]
010	EDX	[EDX]
011	EBX	[EBX]
100	ESP	[--][--] ¹
101	EBP	disp32 ²
110	ESI	[ESI]
111	EDI	[EDI]

- The *mod* field combines with the *r/m* field to form 32 possible values: eight registers and 24 addressing modes.
- The *reg/opcode* field specifies either a register number or three more bits of opcode information. The purpose of the *reg/opcode* field is specified in the primary opcode.
- The *r/m* field can specify a register as an operand or can be combined with the *mod* field to encode an addressing mode.

Functionalities of Assembler: 32-bit CPU (2/5)

■ Opcode

- ✓ Machine format example of MOV opcode

Opcode	Instruction	Description
88 /r	MOV r/m8,r8	Move r8 to r/m8
89 /r	MOV r/m16,r16	Move r16 to r/m16
89 /r	MOV r/m32,r32	Move r32 to r/m32
8A /r	MOV r8,r/m8	Move r/m8 to r8
8B /r	MOV r16,r/m16	Move r/m16 to r16
8B /r	MOV r32,r/m32	Move r/m32 to r32
8C /r	MOV r/m16,Sreg**	Move segment register to r/m16
8E /r	MOV Sreg,r/m16**	Move r/m16 to segment register
A0	MOV AL,moffs8*	Move byte at (seg:offset) to AL
A1	MOV AX,moffs16*	Move word at (seg:offset) to AX
A1	MOV EAX,moffs32*	Move doubleword at (seg:offset) to EAX
A2	MOV moffs8*,AL	Move AL to (seg:offset)
A3	MOV moffs16*,AX	Move AX to (seg:offset)
A3	MOV moffs32*,EAX	Move EAX to (seg:offset)
B0+ /b	MOV r8,imm8	Move imm8 to r8
B8+ /w	MOV r16,imm16	Move imm16 to r16
B8+ /d	MOV r32,imm32	Move imm32 to r32
C6 /0	MOV r/m8,imm8	Move imm8 to r/m8
C7 /0	MOV r/m16,imm16	Move imm16 to r/m16
C7 /0	MOV r/m32,imm32	Move imm32 to r/m32

MOV—Move

(from Intel Manual, Volume 2, 4.3 Instructions: move)

Functionalities of Assembler: 32-bit CPU (3/5)

■ Translation example

The screenshot shows a debugger interface with assembly code and its corresponding binary representation. Annotations highlight specific parts of the assembly code and the binary bytes.

Annotations:

- opcode:** Points to the first byte of each instruction (e.g., 55, 89, etc.).
- mem. operand: Little Endian:** Points to the memory operands (e.g., %ebp, %eax).
- Immediate (1B vs 4B: see below):** Points to the immediate values (e.g., 20, 94, 04, 08; 0x8049420, 0x8049424).
- opcode + register:** Points to the combined opcode and register fields (e.g., 55, 89, etc.).

Mod R/M or T/B number: A legend for the Mod R/M field:

00	mem.
01	mem.+dis(8)
10	mem.+dis(32)
11	reg.

Binary Bytes:

0040320.	000	EAX	[EAX]
804832f:	001	ECX	[ECX]
8048332:	010	EDX	[EDX]
8048336:	011	EBX	[EBX]
8048338:	100	ESP	[--][--] ¹
8048339:	101	EBP	disp32 ²
804833a:	110	ESI	[ESI]
804833b:	111	EDI	[EDI]

Assembly Code:

```
...skipping
080482f4 <main>:
080482f4:    55                      push   %ebp
080482f5:    a1 20 94 04 08          mov    0x8049420,%eax
080482fa:    a3 24 94 04 08          mov    %eax,0x8049424
080482ff:    b8 03 00 00 00          mov    $0x3,%eax
08048304:    b9 04 00 00 00          mov    $0x4,%ecx
08048309:    bf 05 00 00 00          mov    $0x5,%edi
0804830e:    89 ca                mov    %edx,%edx
08048310:    ...
```

Instruction Table:

Opcode	Instruction	Description
88 /r	MOV r/m8,r8	Move r8 to r/m8
89 /r	MOV r/m16,r16	Move r16 to r/m16
89 /r	MOV r/m32,r32	Move r32 to r/m32
8A /r	MOV r8,r/m8	Move r/m8 to r8
8B /r	MOV r16,r/m16	Move r/m16 to r16
8B /r	MOV r32,r/m32	Move r/m32 to r32
8C /r	MOV r/m16,Sreg**	Move segment register to r/m16
8E /r	MOV Sreg,r/m16**	Move r/m16 to segment register
A0	MOV AL,moffs8*	Move byte at (seg:offset) to AL
A1	MOV AX,moffs16*	Move word at (seg:offset) to AX
A1	MOV EAX,moffs32*	Move doubleword at (seg:offset) to EAX
A2	MOV moffs8*,AL	Move AL to (seg:offset)
A3	MOV moffs16*,AX	Move AX to (seg:offset)
A3	MOV moffs32*,EAX	Move EAX to (seg:offset)
B0+ /b	MOV r8,imm8	Move imm8 to r8
B8+ /w	MOV r16,imm16	Move imm16 to r16
B8+ /d	MOV r32,imm32	Move imm32 to r32
C6 /0	MOV r/m8,imm8	Move imm8 to r/m8
C7 /0	MOV r/m16,imm16	Move imm16 to r/m16
C7 /0	MOV r/m32,imm32	Move imm32 to r/m32

Functionalities of Assembler: 32-bit CPU (4/5)

■ Translation example

The screenshot shows a debugger window with assembly code. The assembly code includes SIB bytes and ModR/M bytes, which are highlighted and analyzed with callouts.

ModR/M: **11001010**

ModR/M: **01001000**

displacement

The [-]- nomenclature means a SIB follows the ModR/M byte.

SIB: **10011000**

R/M or I/B number

Mod

00	mem.
01	mem.+dis(8)
10	mem.+dis(32)
11	reg.

804830e: 00 01 *1 Scale 55
8048310: 01 02
8048312: 10 *4
8048314: 11 *8
8048317:
804831b:
8048322:

push %ebp
mov 0x8049420,%eax
mov %eax,0x8049424
mov \$0x3,%eax
mov \$0x4,%ecx
mov \$0x5,%edi
mov %ecx,%edx
mov %ecx,(%edx)
mov (%eax),%ecx
mov 0x4(%eax),%ecx
mov 0x8(%eax,%ebx,4),%ecx
mov 0x5678(%eax,%ebx,4),%ecx
mov %ecx,0x8049420

000	EAX	[EAX]
001	ECX	[ECX]
010	EDX	[EDX]
011	EBX	[EBX]
100	ESP	[--][--] ¹
101	EBP	disp32 ²
110	ESI	[ESI]
111	EDI	[EDI]

Opcode	Instruction	Description
88 /r	MOV r/m8,r8	Move r8 to r/m8
89 /r	MOV r/m16,r16	Move r16 to r/m16
89 /r	MOV r/m32,r32	Move r32 to r/m32
8A /r	MOV r8,r/m8	Move r/m8 to r8
8B /r	MOV r16,r/m16	Move r/m16 to r16
8B /r	MOV r32,r/m32	Move r/m32 to r32
8C /r	MOV r/m16,Sreg**	Move segment register to r/m16
8E /r	MOV Sreg,r/m16**	Move r/m16 to segment register
A0	MOV AL,moffs8*	Move byte at (seg:offset) to AL
A1	MOV AX,moffs16*	Move word at (seg:offset) to AX
A1	MOV EAX,moffs32*	Move doubleword at (seg:offset) to EAX
A2	MOV moffs8*,AL	Move AL to (seg:offset)
A3	MOV moffs16*,AX	Move AX to (seg:offset)
A3	MOV moffs32*,EAX	Move EAX to (seg:offset)
B0+ /b	MOV r8,imm8	Move imm8 to r8
B8+ /w	MOV r16,imm16	Move imm16 to r16
B8+ /d	MOV r32,imm32	Move imm32 to r32
C6 /0	MOV r/m8,imm8	Move imm8 to r/m8
C7 /0	MOV r/m16,imm16	Move imm16 to r/m16
C7 /0	MOV r/m32,imm32	Move imm32 to r/m32

Functionalities of Assembler: 32-bit CPU (5/5)

■ Translation example (cont')

The screenshot shows a debugger interface with assembly code and a reference table.

Mod table:

00	mem.
01	mem.+dis(8)
10	mem.+dis(32)
11	reg.

Scale table:

00	*1
01	*2
10	*4
11	*8

R/M or T/B number dropdown:

000	EAX	[EAX]
001	ECX	[ECX]
010	EDX	[EDX]
011	EBX	[EBX]
100	ESP	[--][--] ¹
101	EBP	disp32 ²
110	ESI	[ESI]
111	EDI	[EDI]

Instruction Table:

Opcode	Instruction	Description
88 /r	MOV r8,r8	Move r8 to r/m8
89 /r	MOV r16,r16	Move r16 to r/m16
89 /r	MOV r32,r32	Move r32 to r/m32
8A /r	MOV r8,r/m8	Move r/m8 to r8
8B /r	MOV r16,r/m16	Move r/m16 to r16
8B /r	MOV r32,r/m32	Move r/m32 to r32
8C /r	MOV r/m16,Sreg**	Move segment register to r/m16
8E /r	MOV Sreg,r/m16**	Move r/m16 to segment register
A0	MOV AL,moffs8*	Move byte at (seg:offset) to AL
A1	MOV AX,moffs16*	Move word at (seg:offset) to AX
A1	MOV EAX,moffs32*	Move doubleword at (seg:offset) to EAX
A2	MOV moffs8*,AL	Move AL to (seg:offset)
A3	MOV moffs16*,AX	Move AX to (seg:offset)
A3	MOV moffs32*,EAX	Move EAX to (seg:offset)
B0+ /b	MOV r8,imm8	Move imm8 to r8
B8+ /w	MOV r16,imm16	Move imm16 to r16
B8+ /d	MOV r32,imm32	Move imm32 to r32
C6 /0	MOV r/m8,imm8	Move imm8 to r/m8
C7 /0	MOV r/m16,imm16	Move imm16 to r/m16
C7 /0	MOV r/m32,imm32	Move imm32 to r/m32

Assembly Code:

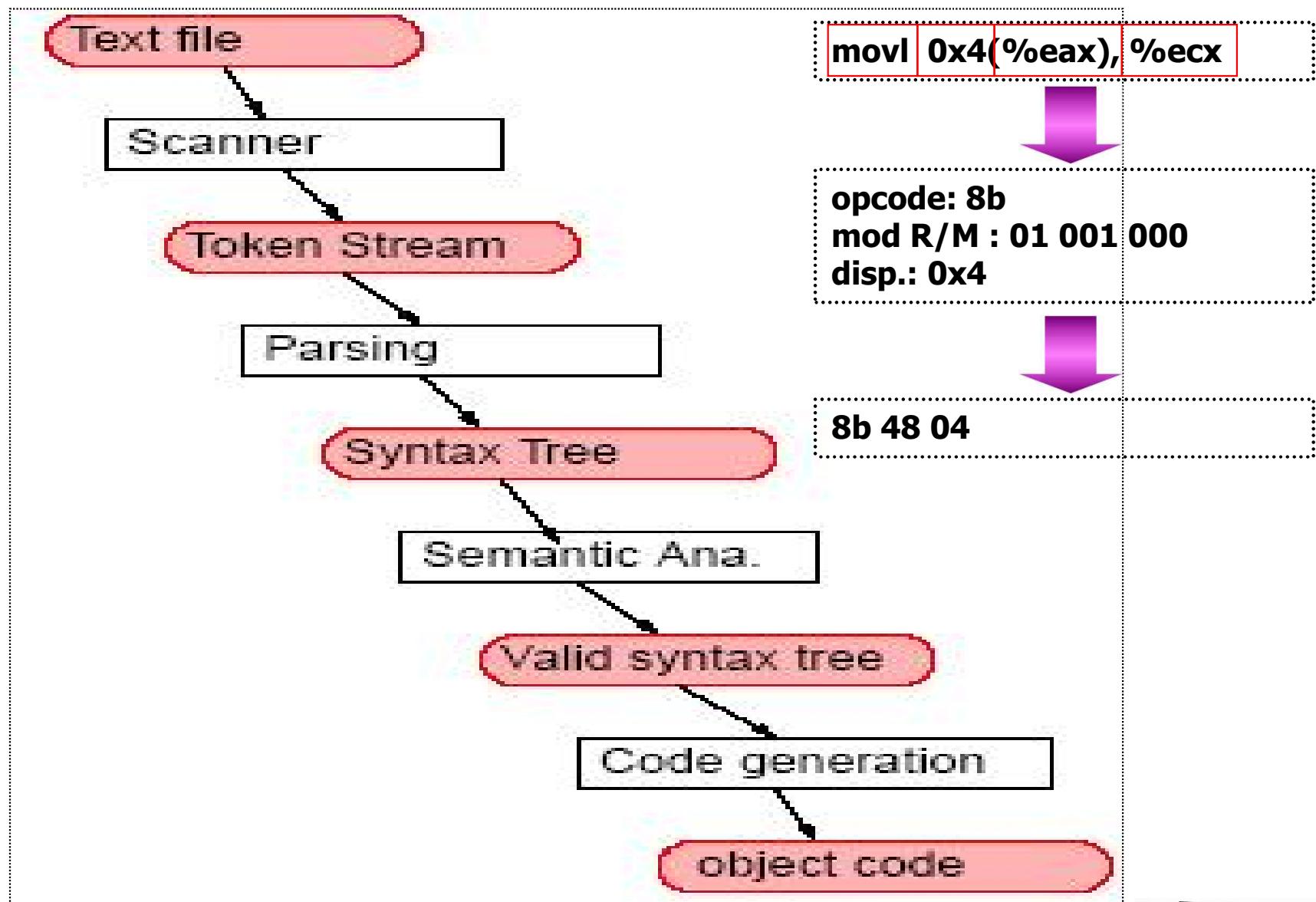
```
8048317: 8b 8c 98 78 56 00 00    mov    0x5678(%eax,%ebx,4),%ecx  
8048322: 89 0d 20 94 04 08    mov    %ecx,0x8049420  
8048328: c7 05 20 94 04 08 34  movl   $0x1234,0x8049420  
804832f: 12 00 00  
8048332: 66 b8 68 1e    mov    $0x1e68,%ax  
8048336: b0 07    mov    $0x7,%al  
8048338: c9          leave  
8048339: c3          ret  
804833a: 90          nop  
804833b: 90          nop
```

Annotations:

- A red box highlights the instruction `mov %ecx,0x8049420`.
- A callout box points to the `disp32` entry in the R/M table, stating: "The disp32 nomenclature denotes a 32-bit displacement that follows the ModR/M byte".
- A callout box points to the `b0 07` instruction, stating: "Prefix: 0x66 → operand size override".

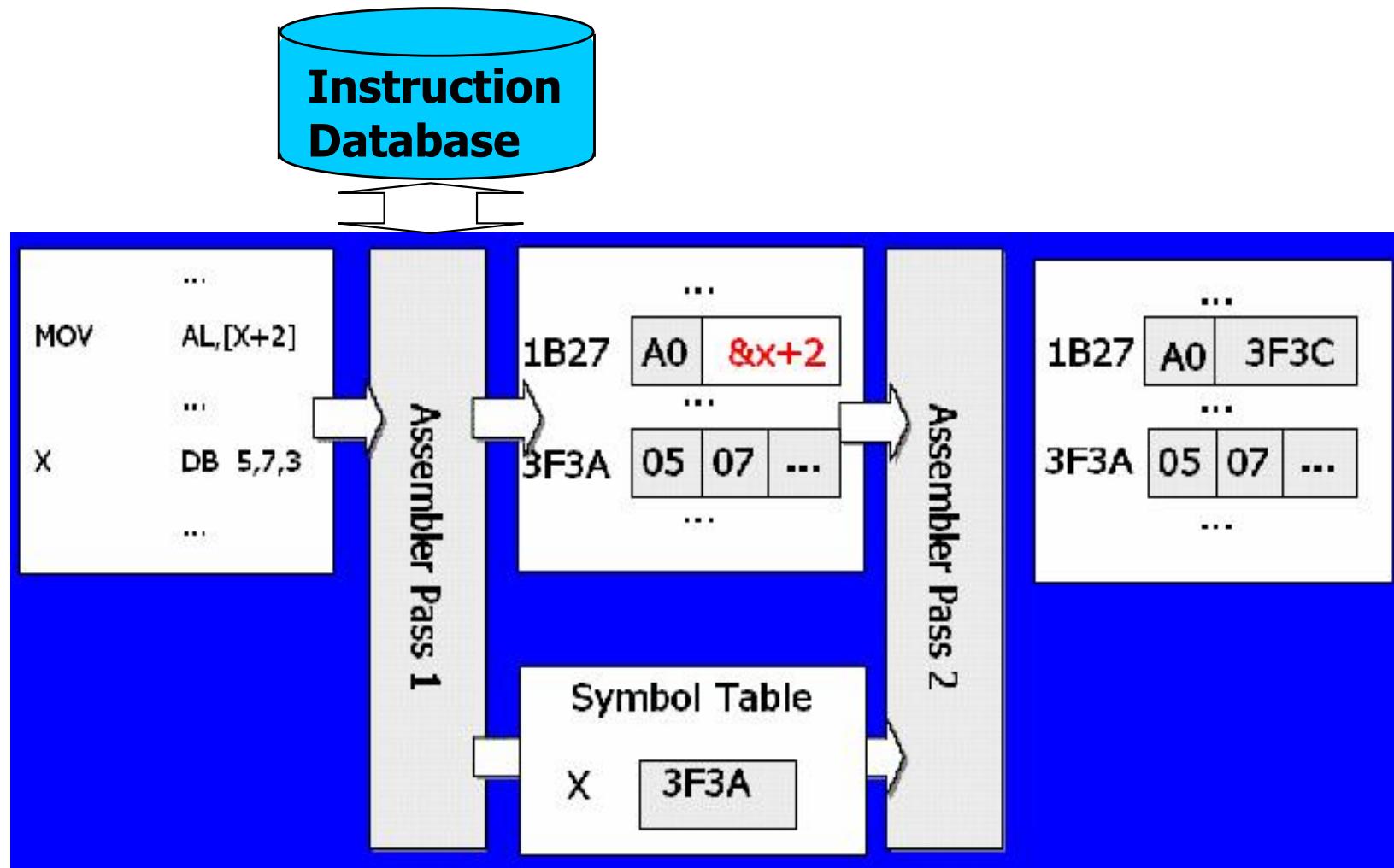
Structure of Assembler (1/2)

■ 4 Main Components



Structure of Assembler (2/2)

■ 2 pass assembler

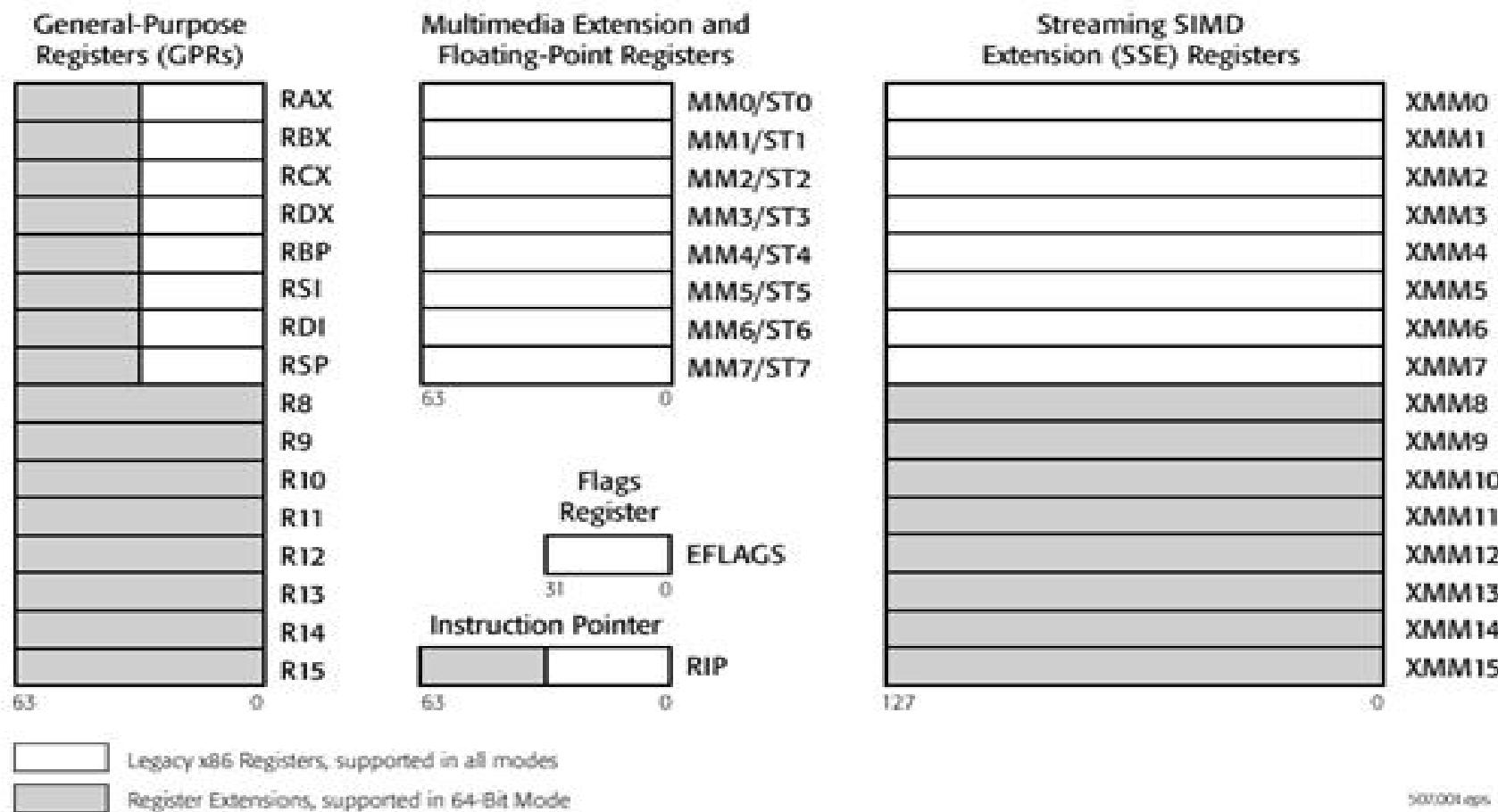


☞ To sum up, designing an assembler consists of 1) making parser, 2) manipulating DB, 3) managing symbol table, 4) code generating, 5) error handing, 6) optimization and so on.

Functionalities of Assembler: 64-bit CPU (1/4)

■ Machine Code with 64-bit extension

- ✓ Need to encode new registers (GPRs) and 64-bit addressing
- ✓ Need to maintain backward compatibility



Functionalities of Assembler: 64-bit CPU (2/4)

■ Machine Code with 64-bit extension

- ✓ Code format

Legacy Prefixes	REX Prefix	Opcode	ModR/M	SIB	Displacement	Immediate
Grp 1, Grp 2, Grp 3, Grp 4 (optional)	(optional)	1-, 2-, or 3-byte opcode	1 byte (if required)	1 byte (if required)	Address displacement of 1, 2, or 4 bytes	Immediate data of 1, 2, or 4 bytes or none

Figure 2-3. Prefix Ordering in 64-bit Mode

- REX prefix
 - Specify GPRs (rax, rbx, ..., rdi, r8, r9, ... r15) and SSE registers
 - Specify 64-bit operand size

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

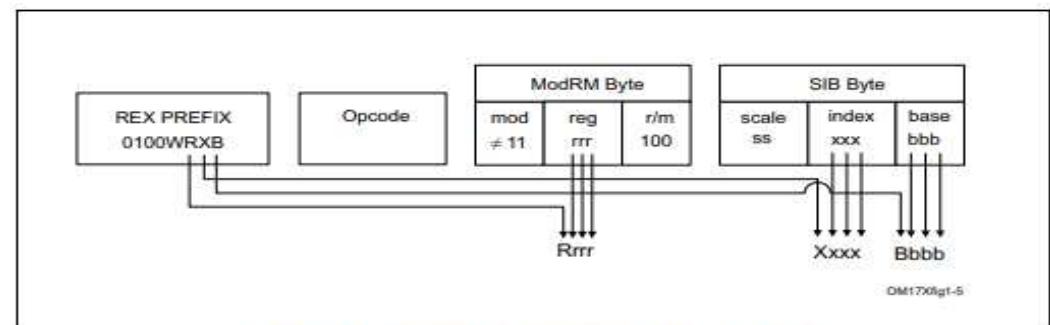


Figure 2-6. Memory Addressing With a SIB Byte
(from Intel Manual, Volume 2, 2.2 IA-32e Mode)

Functionalities of Assembler: 64-bit CPU (3/4)

- Machine Code including 64-bit extension
 - ✓ Machine format example of MOV opcode
 - 64bit addressing → REX prefix

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.
REX + 8A /r	MOV r8***,r/m8***	RM	Valid	N.E.	Move r/m8 to r8.
8B /r	MOV r16,r/m16	RM	Valid	Valid	Move r/m16 to r16.
8B /r	MOV r32,r/m32	RM	Valid	Valid	Move r/m32 to r32.
REX.W + 8B /r	MOV r64,r/m64	RM	Valid	N.E.	Move r/m64 to r64.
8C /r	MOV r/m16,Sreg**	MR	Valid	Valid	Move segment register to r/m16.
8C /r	MOV r16/r32/m16, Sreg**	MR	Valid	Valid	Move zero extended 16-bit segment register to r16/r32/m16.
REX.W + 8C /r	MOV r64/m16, Sreg**	MR	Valid	Valid	Move zero extended 16-bit segment register to r64/m16.
BE /r	MOV Sreg,r/m16**	RM	Valid	Valid	Move r/m16 to segment register.
REX.W + BE /r	MOV Sreg,r/m64**	RM	Valid	Valid	Move lower 16 bits of r/m64 to segment register.
A0	MOV AL,moffs8*	FD	Valid	Valid	Move byte at (seg:offset) to AL.
REX.W + A0	MOV AL,moffs8*	FD	Valid	N.E.	Move byte at (offset) to AL.
A1	MOV AX,moffs16*	FD	Valid	Valid	Move word at (seg:offset) to AX.
A1	MOV EAX,moffs32*	FD	Valid	Valid	Move doubleword at (seg:offset) to EAX.
REX.W + A1	MOV RAX,moffs64*	FD	Valid	N.E.	Move quadword at (offset) to RAX.
A2	MOV moffs8,AL	TD	Valid	Valid	Move AL to (seg:offset).
REX.W + A2	MOV moffs8**,AL	TD	Valid	N.E.	Move AL to (offset).
A3	MOV moffs16*,AX	TD	Valid	Valid	Move AX to (seg:offset).
A3	MOV moffs32*,EAX	TD	Valid	Valid	Move EAX to (seg:offset).
REX.W + A3	MOV moffs64*,RAX	TD	Valid	N.E.	Move RAX to (offset).
B0+ rb ib	MOV r8,imm8	OI	Valid	Valid	Move imm8 to r8.
REX + B0+ rb ib	MOV r8**,imm8	OI	Valid	N.E.	Move imm8 to r8.
B8+ rw iw	MOV r16,imm16	OI	Valid	Valid	Move imm16 to r16.
B8+ rd id	MOV r32,imm32	OI	Valid	Valid	Move imm32 to r32.
REX.W + B8+ rd io	MOV r64,imm64	OI	Valid	N.E.	Move imm64 to r64.
C6 /O ib	MOV r/m8,imm8	MI	Valid	Valid	Move imm8 to r/m8.
REX + C6 /O ib	MOV r/m8***,imm8	MI	Valid	N.E.	Move imm8 to r/m8.
C7 /O iw	MOV r/m16,imm16	MI	Valid	Valid	Move imm16 to r/m16.
C7 /O id	MOV r/m32,imm32	MI	Valid	Valid	Move imm32 to r/m32.
REX.W + C7 /O id	MOV r/m64,imm32	MI	Valid	N.E.	Move imm32 sign extended to 64-bits to r/m64.

(from Intel Manual, Volume 2, 4.3 Instructions: move)

Functionalities of Assembler: 64-bit CPU (4/4)

■ Translation example

choijm's X desktop (embedded.wowdns.com:2)

choijm@embedded:~/syspro/exam/asm/03/machine

```
...skipping
080482f4 <main>:
080482f4:    55          push %ebp
080482f5:    a1 20 94 04 08    mov 0x8049420,%eax
080482fa:    a3 24 94 04 08    mov %eax,0x8049424
080482ff:    b8 03 00 00 00    mov $0x3,%eax
08048304:    b9 04 00 00 00    mov $0x4,%ecx
08048309:    bf 05 00 00 00    mov $0x5,%edi
0804830e:    89 ca           mov %ecx,%edx
08048310:    89 0a           mov %ecx,(%edx)
08048312:    8b 08           mov (%eax),%ecx
08048314:    8b 48 04         mov 0x4(%eax),%ecx
08048317:    8b 4c 98 08         mov 0x8(%eax,%ebx,4),%ecx
0804831b:    8b 8c 98 78 56 00 00    mov 0x5678(%eax,%ebx,4),%ecx
08048322:    89 0d 20 94 04 08    mov %ecx,0x8049420
08048328:    c7 05 20 94 04 08 34    movl $0x1234,0x8049420
0804832f:    12 00 00
08048332:    66 b8 68 1e         mov $0x1e68,%ax
08048336:    b0 07           mov $0x7,%al
08048338:    c9              leave
08048339:    c3              ret
0804833a:    90              nop
0804833b:    90              nop
```

REX Prefix: 0100 1000

choijm@choijm-VirtualBox: ~/2015_syspro/chap9/assembler

400536:	e8 d5 fe ff ff	callq 400410 <puts@plt>
40053b:	a1 20 94 04 48 23 01	mov 0x12348049420,%eax // movabs
400542:	00 00	
400544:	a3 24 94 04 48 23 01	mov %eax,0x12348049424 // movabs
40054b:	00 00	
40054d:	b8 03 00 00 00	mov \$0x3,%eax
400552:	b9 04 00 00 00	mov \$0x4,%ecx
400557:	bf 05 00 00 00	mov \$0x5,%edi
40055c:	89 ca	mov %ecx,%edx
40055e:	67 89 0a	mov %ecx,(%edx)
400561:	48 89 0a	mov %rcx,(%rdx)
400564:	4d 89 0a	mov %r9,(%r10)
400567:	67 8b 08	mov (%eax),%ecx
40056a:	67 8b 48 04	mov 0x4(%eax),%ecx
40056e:	67 8b 4c 98 08	mov 0x8(%eax,%ebx,4),%ecx
400573:	67 8b 8c 98 78 56 00	mov 0x5678(%eax,%ebx,4),%ecx
40057a:	00	
40057b:	89 0c 25 20 94 04 08	mov %ecx,0x8049420
400582:	c7 04 25 20 94 04 48	movl \$0x1234,0x48049420
400589:	34 12 00 00	
40058d:	66 b8 68 1e	mov \$0x1e68,%ax
400591:	b0 07	mov \$0x7,%al
400593:	c7 05 ab 0a 20 00 14	movl \$0x14,0x200aab(%rip)
40059a:	00 00 00	

For specifying r9, r10

inline Assembly (1/6)

■ inline Assembly

- ✓ Assembly code embedded in a high level language like C
- ✓ structure
 - **`__asm__(assembly statement : output : input : modified register)`**
 - **Each parts are separated by :**
 - **output, input, modified register are optional**
 - **assembly statement:** using “ ”, add a prefix % to each register
 - **output:** “=g”(variable name)
 - **input:** “g”(variable name)
 - **modified register (clobber):** notify to compiler which registers are modified by inline assembly (to prevent the side effect of inline assembly)
 - **Output and input are accessed using the notation of %0, %1, %2, ...**

inline Assembly (2/6)

■ inline Assembly practice 1: add

The screenshot shows a terminal window with the following C code:

```
/* inline assembly 예제 */
/* 11월 10일 J. Choi */

#include <stdio.h>

main()
{
    int a, b, c = 7;

    a = 3;
    b = 5;

    printf("c = %d\n", c);

    __asm__ (
        "movl    %1, %%eax\n"
        "addl    %2, %%eax\n"
        "movl    %%eax, %0\n"
        : "=g" (c)
        : "g" (a), "g" (b)
    );

    printf("c = %d\n", c);
}

~
~
```

Annotations with arrows point from labels to specific parts of the code:

- A box labeled **assembly statements** points to the `__asm__` block.
- A box labeled **output** points to the `": "=g" (c)` part of the assembly code.
- A box labeled **input** points to the `: "g" (a), "g" (b)` part of the assembly code.

The terminal status bar at the bottom shows: `inline add g.c`, `"inline_add_g.c" 24`, `--100%--`, `24, 1`, and `모두`.

input/output passing

a	eax
b	ebx
c	ecx
d	edx
S	esi
D	edi
q	general register
m	memory
g	general register and memory
A	edx: eax (64 bits)
f	FP register
i	immediate
0	first parameter
..	

inline Assembly (3/6)

■ inline Assembly practice 2: register input

The screenshot shows a terminal window titled "choijm@localhost:/home/choijm/syspro_examples/chap9". The code in the terminal is:

```
/* inline assembly 예제 */
/* 11월 10일 J. Choi */

#include <stdio.h>

main()
{
    int a, b, c = 7;

    a = 3;
    b = 5;

    printf("c = %d\n", c);

    __asm__ (
        "addl    %%ebx, %%eax\n"
        : "=a" (c)
        : "a" (a), "b" (b)
    );

    printf("c = %d\n", c);
}

inline_add.c 12,0-1 모두
"inline_add.c" 22 줄 --54%--
```

The terminal shows the assembly code generated by the inline assembly directive:

```
addl    %%ebx, %%eax\n
```

The status bar at the bottom indicates the file is "inline_add.c", has 22 lines, and is 54% complete.

inline Assembly (4/6)

■ inline Assembly practice 3: clobber

```
choijm@localhost:~/public_html/syspro/exam_inline/3_clobber
/* inline assembly 예제: clobber */
/* 11월 15일 최종무 */
#include <stdio.h>

main()
{
    int a, b, c, d;
    a = 0x40000002;
    b = 4;

    __asm__ (
        "null    %%ebx\n"
        "movl    %%eax, %0\n"
        "movl    %%edx, %1\n"
        : "=g" (c), "=g" (d)
        : "a" (a), "b" (4)
        : "%edx"
    );
    printf("a = %x, b = %d, c = %d, d = %d\n", a, b, c, d);

    __asm__ (
        "divl    %%ebx\n"
        "movl    %%eax, %0\n"
        "movl    %%edx, %1\n"
        : "=g" (c), "=g" (d)
        : "a" (a), "b" (4), "d" (0)
    );
    printf("a = %x, b = %d, c = %x, d = %d\n", a, b, c, d);
}

~ ~ ~
"clobber1.c" 30L, 516C
```

To notify that a register is used internally in inline assembly

inline Assembly (5/6)

■ inline Assembly practice 4: stack again

```
choijm@embedded: ~/syspro18/chap9
/* stack_destroy.c: 스택 구조 분석 2, 11월 25일, choijm@dku.edu */
#include <stdio.h>

void f1() {
    int i;
    printf("In func1\n");
}

void f2() {
/*
    int j, *ptr;
    printf("f2 local: %p, %p\n", &j, &ptr);
    printf("In func2 \n");

    ptr = &j;
    *(ptr+2) = f1;
*/
    printf("In func2 \n");
    __asm__ (
        "movl    %0, 4(%ebp)\n"
        :
        : "g" (f1)
    );
}

void f3() {
    printf("Before invoke f2()\n");
    f2();
    printf("After invoke f2()\n");
}

main() {
    f3();
}
"stack_destroy_inline.c" 35L, 499C 19 1,1 Top
```

inline Assembly (6/6)

■ inline Assembly practice 5: define

```
choijm@choijm-VirtualBox: ~/2015_syspro/chap9/inline
#include <stdio.h>

#define rep_movsl(src, dest, numwords) \
__asm__ __volatile__ ( \
    "cld\n" \
    "rep\n" \
    "movsb" \
    : \
    : "S" (src), "D" (dest), "c" (numwords) \
)

main()
{
    char a[] = "hello";
    char b[16];

    rep_movsl(a, b, sizeof(a));
    printf("dest = %s\n", b);
}
```

Prevent a compiler from moving these codes to other place for the optimization purpose.

Summary

- Apprehend the role of assembler (“as” in Linux)
 - ✓ Assembly language → Machine language
- Understand the structure of assembler
 - ✓ Token analysis, Parsing, Syntax analysis, Semantic Analysis, Symbol table, Code generation, Optimization
 - ✓ 2 pass assembler
- Make a program with inline assembly

☞ Homework 8: Make an assembler

1.1 Requirements

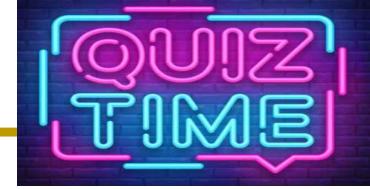
- build an assembler that can translate assembly codes into the IA machine codes shown in slides 6~8.
- manipulate DB and do error handling
- shows student's ID and date (using whoami and date)

1.2 Write a report

- 1) Introduction, 2) Source code, 3) Snapshots, 4) Discussion

1.3 How to submit? Send report to mgchoi@dankook.ac.kr

1.4 Deadline: a week later (same time)



Quiz for this Lecture

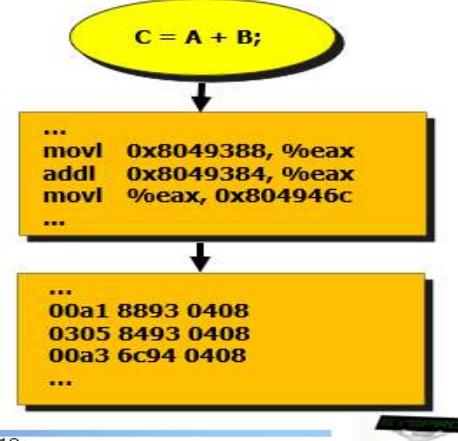
■ Quiz

1. Discuss 4 main components of assembler.
2. The below left figure is the language hierarchy that we have seen in the LN 1. Now, explain what is “movl”, “0x8049388” and “a1”. Explain the little endian in this figure.
3. Explain how x86-64 maintain the backward compatibility.
4. In LN 9, we make a program that can destroy stack using inline assembly. Discuss the differences between this program and the program we have learnt in LN 4.

Compilation System (1/5)

■ Concept: Language Hierarchy

High-level Language



Assembly Language

Machine Language
(Binary code)

12

Stack Details (6/6)

■ Stack example 2

```
/* stack_destroy.c: 스택 구조 분석 2, 9월 19일, choijm@dku.edu */  
#include <stdio.h>  
  
void f1() {  
    int i;  
    printf("In func1\n");  
}  
  
void f2() {  
    int j, *ptr;  
    printf("f2 local: %tp, %tp\n", &j, &ptr);  
    printf("In func2 \n");  
  
    ptr = &j;  
    *(ptr+2) = f1;  
}  
  
void f3() {  
    printf("Before invoke f2()\n");  
    f2();  
    printf("After invoke f2()\n");  
}  
  
main() {  
    f3();  
}
```

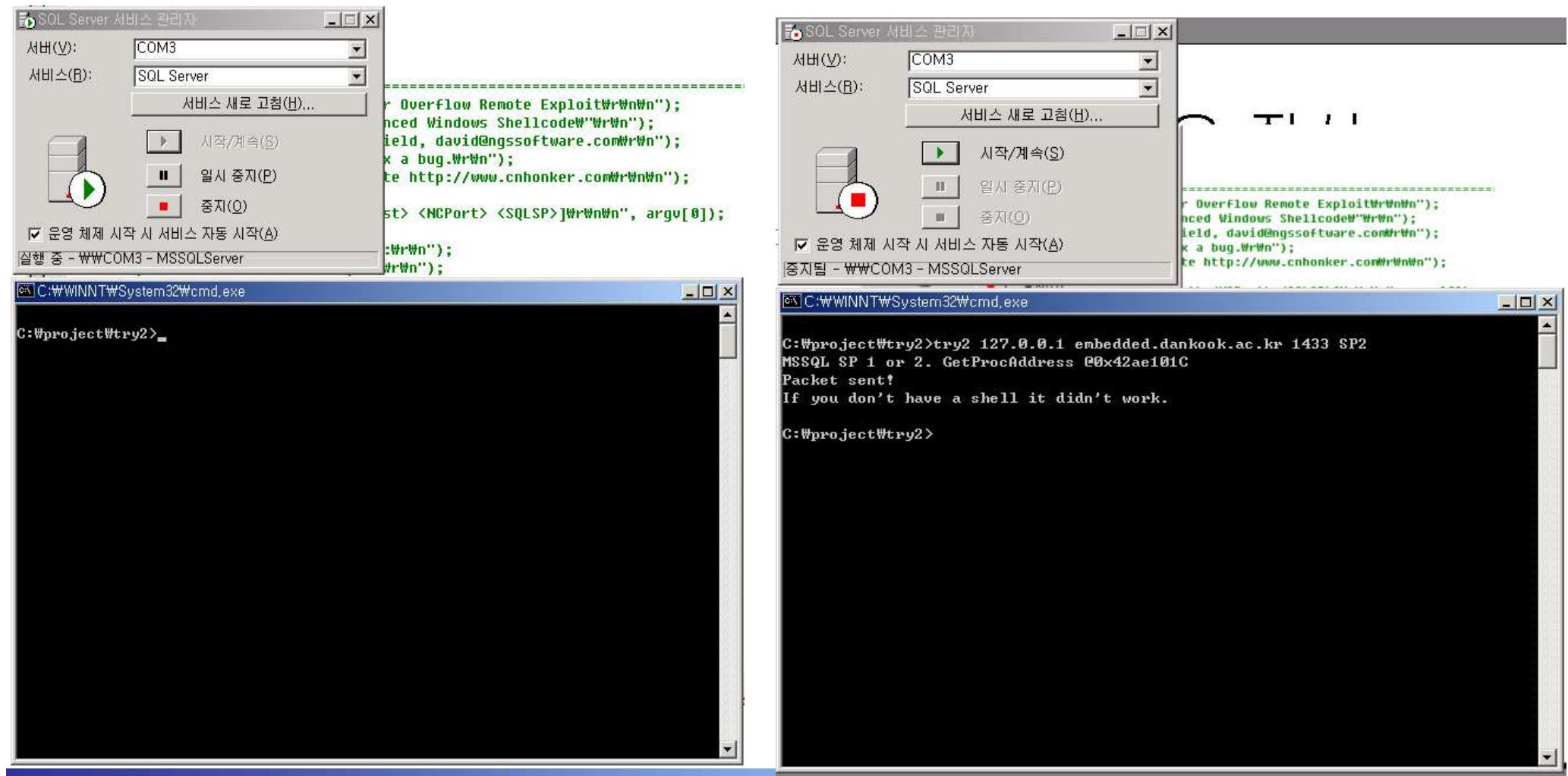
A screenshot of a terminal window showing the execution of the program. The output shows the stack layout and the modification of the stack by the function f2().

18

Appendix: Exploit code (1/2)

■ Exploit code

- ✓ A code that attacks the vulnerabilities of program
 - System down, obtain a shell with root privilege



Appendix: Exploit code (2/2)

SQL Exploit code

- ✓ Copy a request into stack in a SQL internal function (vulnerable point)
- ✓ Make a larger request might destroy stack (buffer overflow)
- ✓ Modify the return address of stack so that it executes an exploit code

push %ebp

mov %esp, %ebp

push immediate

```
char exploit_code[] =
```

```
"\x55\x8B\xEC\x68\x18\x10\xAE\x42\x68\x1C"
```

```
"\x10\xAE\x42\xEB\x03\x5B\xEB\x05\xE8\xF8"
```

```
"\xFF\xFF\xFF\xFF\xBE\xFF\xFF\xFF\xFF\x81\xF6"
```

```
"\xAE\xFE\xFF\xFF\x03\xDE\x90\x90\x90\x90"
```

```
"\x90\x33\xC9\xB1\x44\xB2\x58\x30\x13\x83"
```

```
"\xEB\x01\xE2\xF9\x43\x53\x8B\x75\xFC\xFF"
```

```
"\x16\x50\x33\xC0\xB0\x0C\x03\xD8\x53\xFF"
```

```
"\x16\x50\x33\xC0\xB0\x10\x03\xD8\x53\x8B"
```

```
..."
```

```
"\xFF\xD0\x90\x2F\x2B\x6A\x07\x6B\x6A\x76"
```

```
"\x3C\x34\x34\x58\x58\x33\x3D\x2A\x36\x3D"
```

```
"\x34\x6B\x6A\x76\x3C\x34\x34\x58\x58\x58"
```

```
"\x58\x0F\x0B\x19\x0B\x37\x3B\x33\x3D\x2C"
```

```
"\x19\x58\x58\x3B\x37\x36\x36\x3D\x3B\x2C"
```

```
"\x58\x1B\x2A\x3D\x39\x2C\x3D\x08\x2A\x37"
```

```
"\x3B\x3D\x2B\x2B\x19\x58\x58\x3B\x35\x3C"
```

```
"\x58";
```

jmp 0x03

pop %eax