Lecture Note 9. Assembler

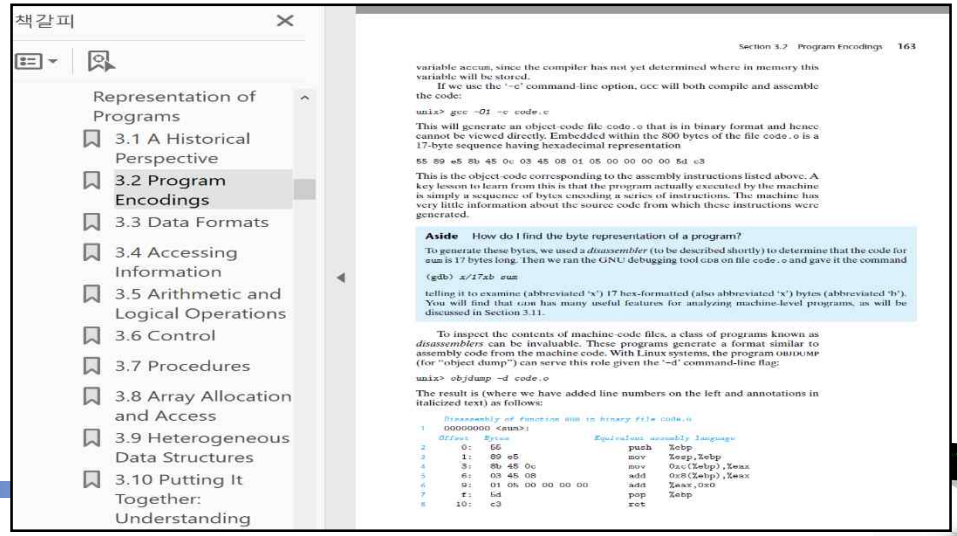
November 20, 2020

Jongmoo Choi
Dept. of Software
Dankook University
http://embedded.dankook.ac.kr/~choijm

Objectives

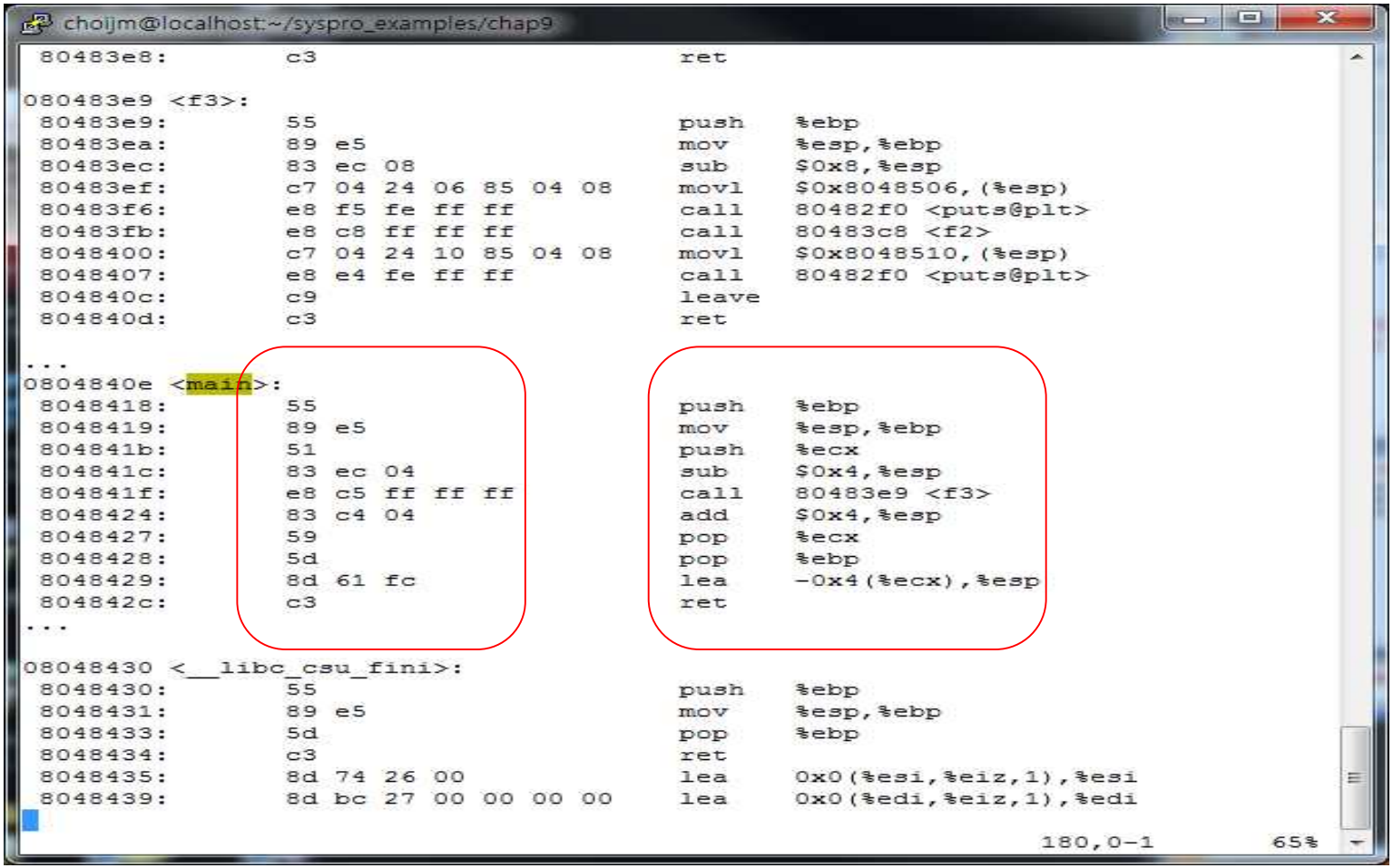
- Understand the role of assembler
- Find out the structure of assembler
- Perceive how a HW designer makes a spec. and how a SW designer makes a program based on the spec.
- Know how to use assembly in a high-level language (inline assembly)

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



Role of Assembler

- Assembler
  - ✓ Translate assembly language into machine language



☞ **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

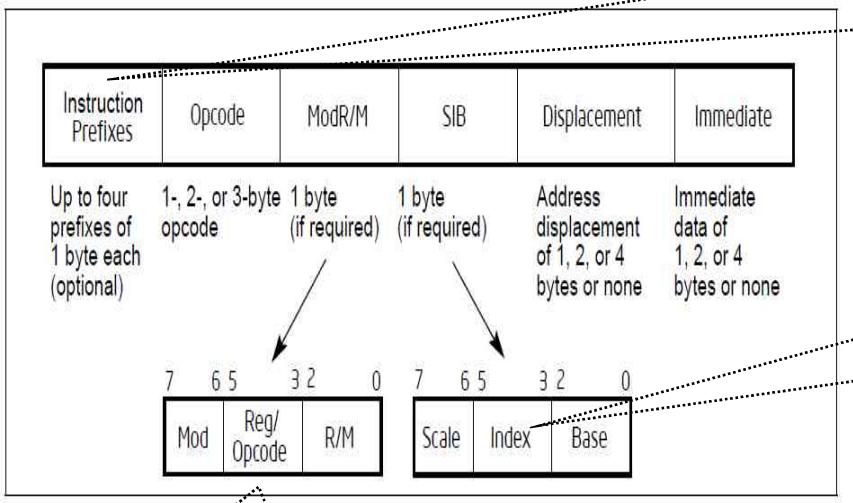


Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format
(from Intel Manual, Volume 2)

Group 1	Lock and repeat prefixes
Group 2	Segment override prefixes, 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.

- 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.

Mod

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

Scale

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

R/M or I/B register

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]

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

- Opcode
  - ✓ Machine format example of MOV opcode

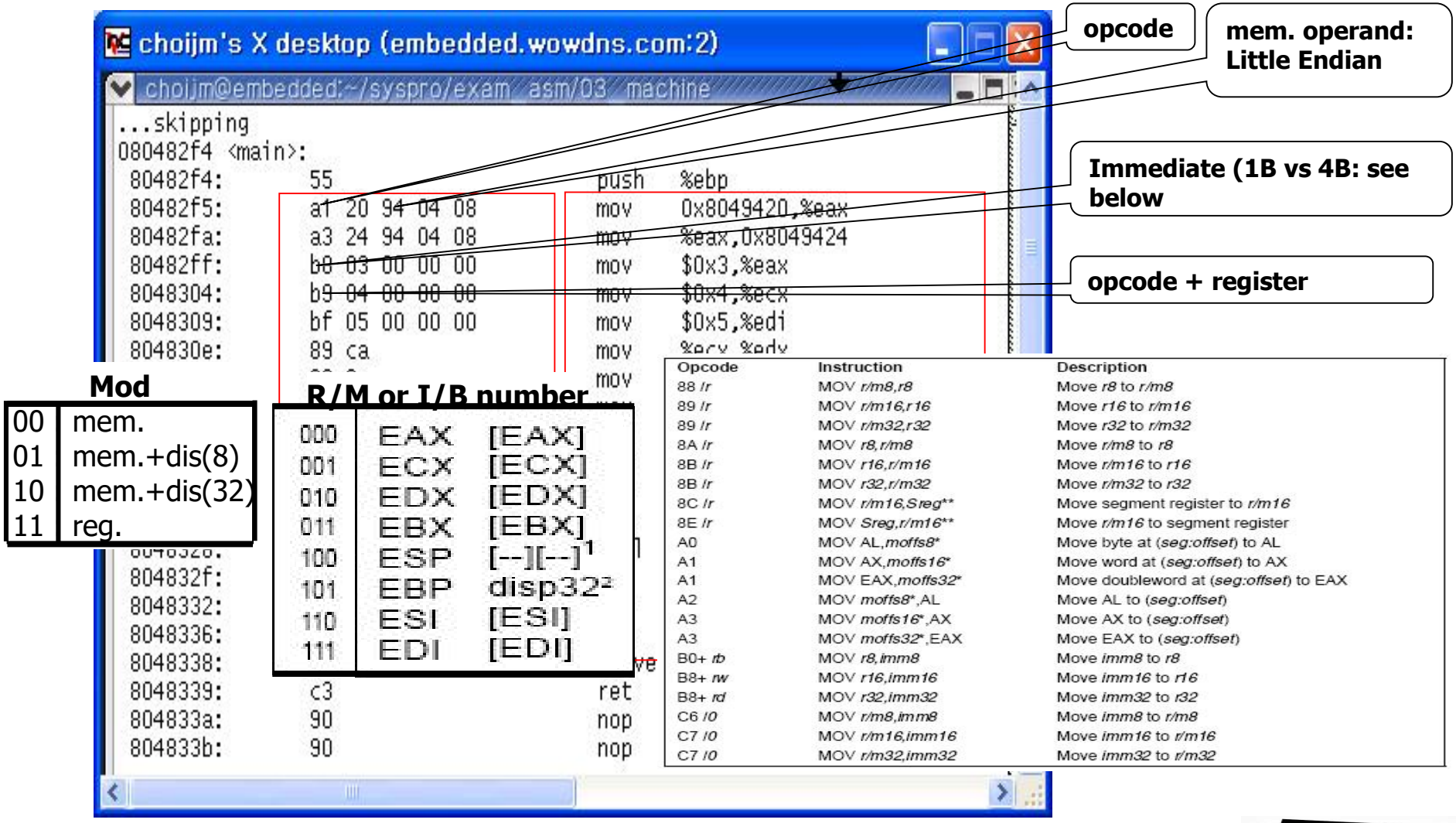
Opcode	Instruction	Description
88 /r	MOV *r/m8,r8*	Move *r8* to *r/m8*
89 /r	MOV *r/m16,r16*	Move *r16* to *r/m16*
89 /r	MOV *r/m32,r32*	Move *r32* to *r/m32*
8A /r	MOV *r8,r/m8*	Move *r/m8* to *r8*
8B /r	MOV *r16,r/m16*	Move *r/m16* to *r16*
8B /r	MOV *r32,r/m32*	Move *r/m32* to *r32*
8C /r	MOV *r/m16,Sreg***	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+ *rb*	MOV *r8,imm8*	Move *imm8* to *r8*
B8+ *rw*	MOV *r16,imm16*	Move *imm16* to *r16*
B8+ *rd*	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)

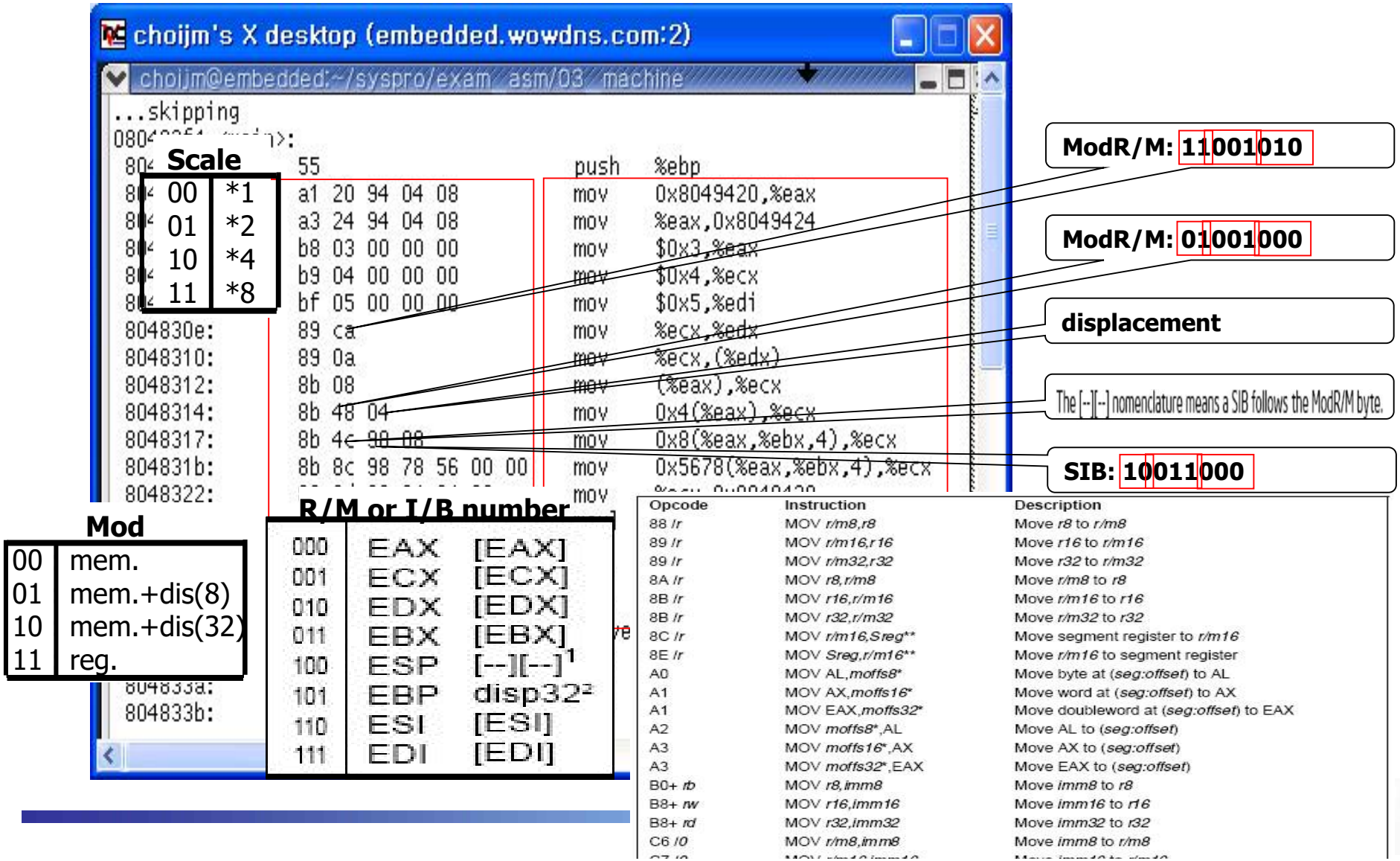
Functionalities of Assembler: 32-bit CPU (3/5)

- Translation example



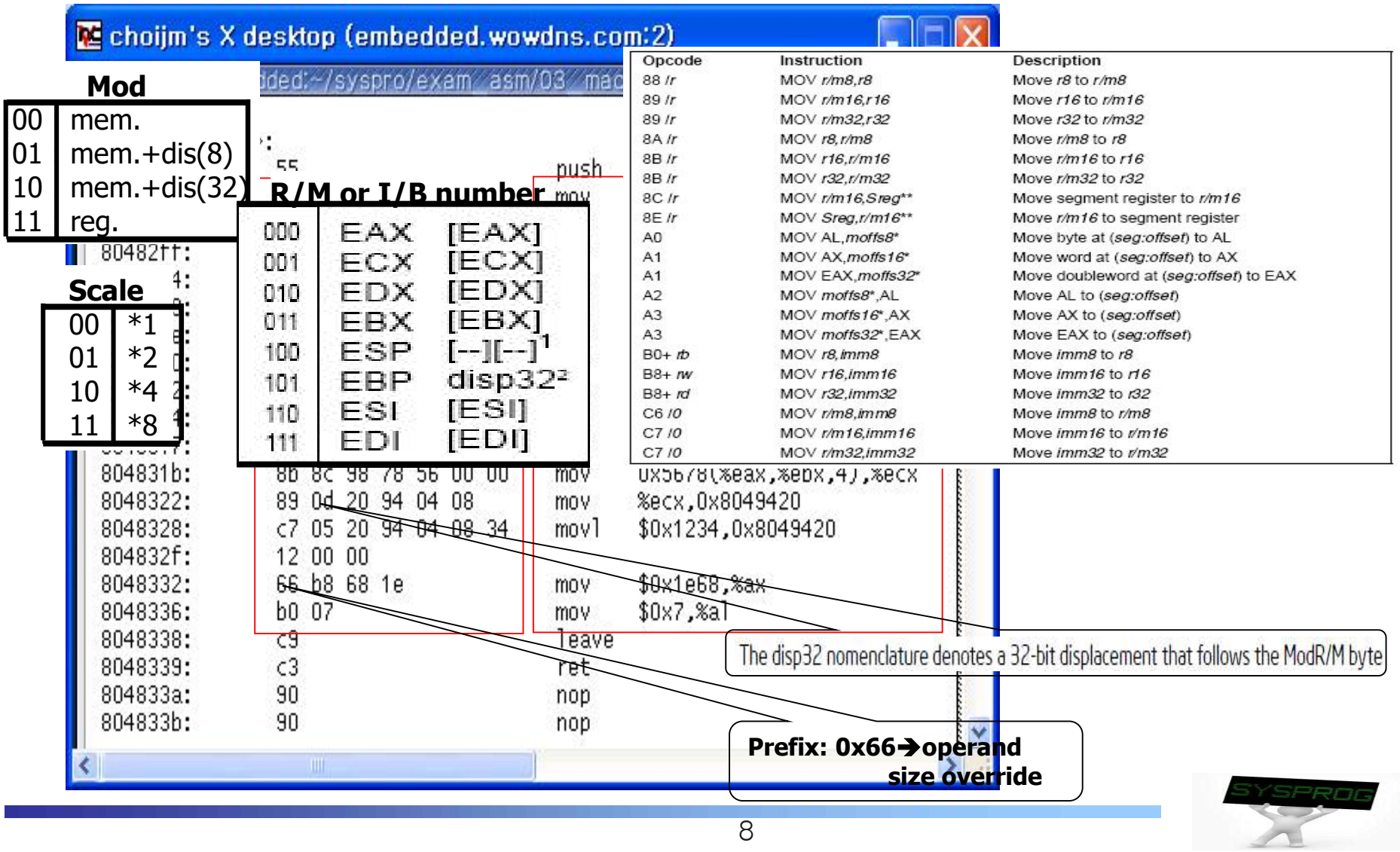
Functionalities of Assembler: 32-bit CPU (4/5)

- Translation example



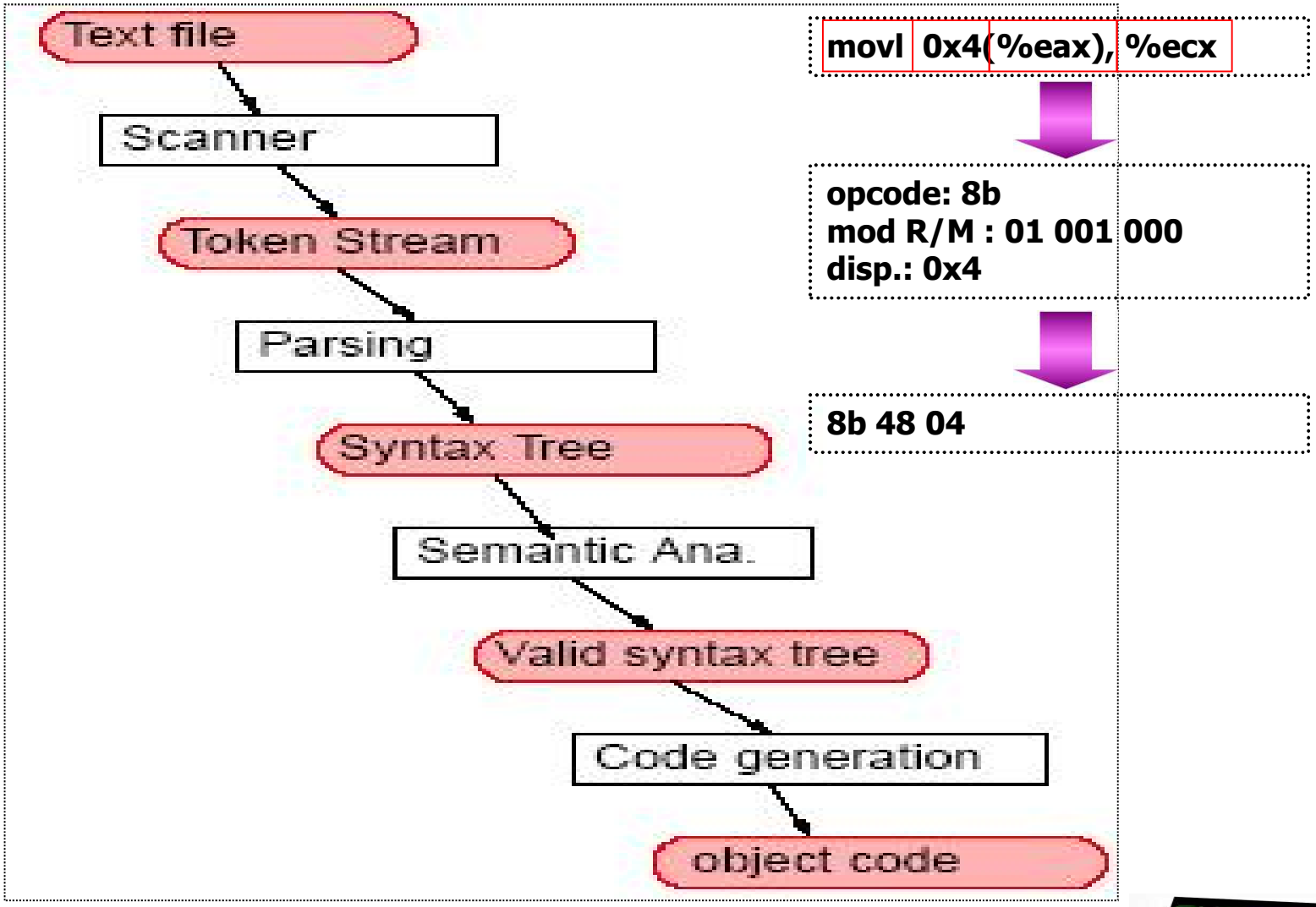
Functionalities of Assembler: 32-bit CPU (5/5)

- Translation example (cont')



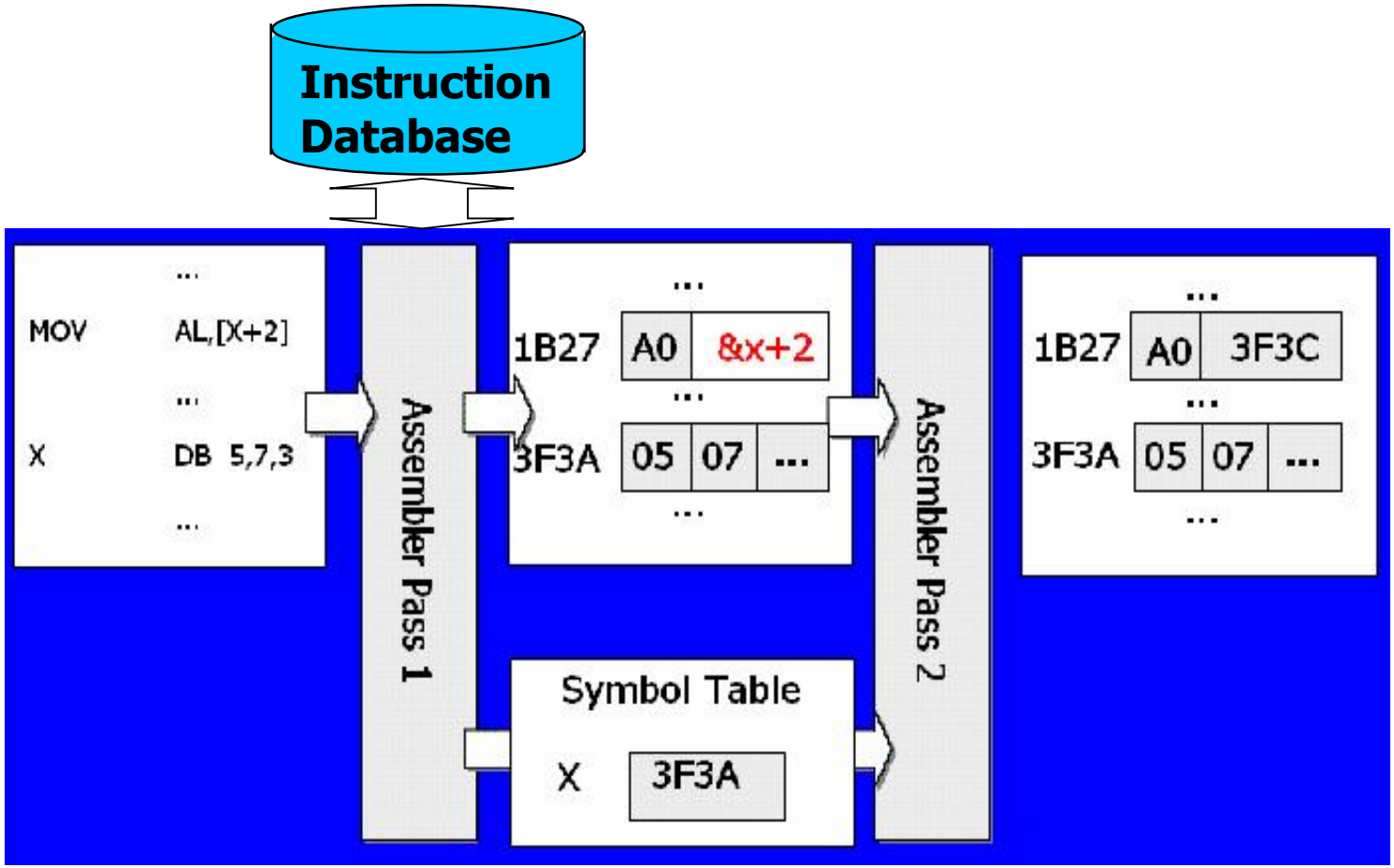
Structure of Assembler (1/2)

- 4 Main Components



Structure of Assembler (2/2)

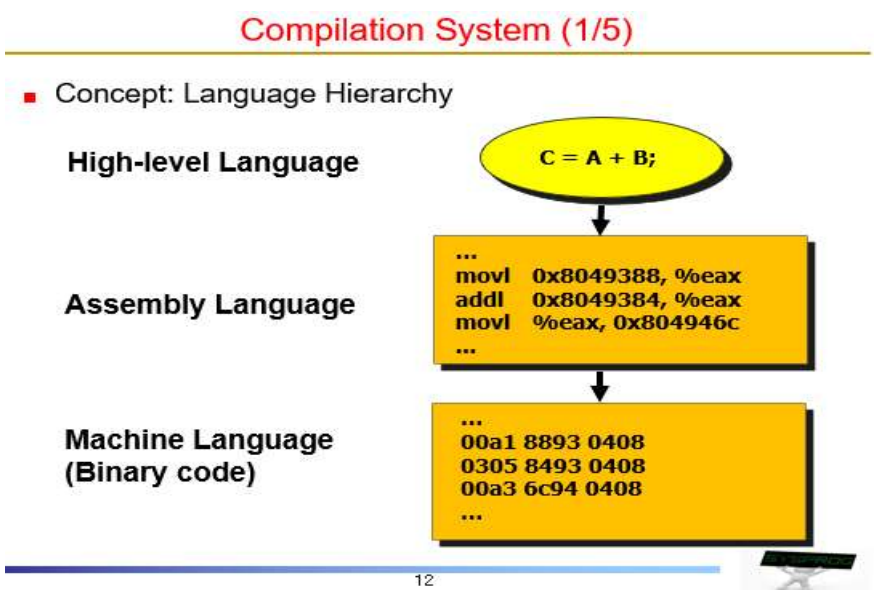
- 2 pass assembler



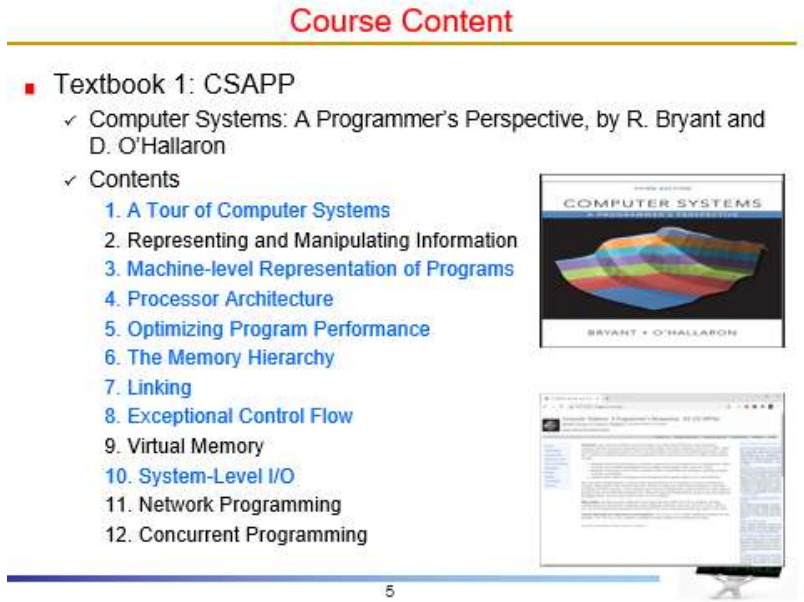
☞ **To sum up, designing an assembler consists of 1) making parser, 2) manipulating DB, 3) managing symbol table, 4) code generating, 5) error handing, 6) optimization and so on.**

Quiz for 14th-Week 2nd-Lesson

- Quiz
  - ✓ 1. Explain the 6 elements in IA machine code format
  - ✓ 2. The below figure is the language hierarchy that we have seen in the LN 1. Now, explain the differences among “C = A + B”, “movl 0x80493bb, %eax” and “a1 8893 0408”.
  - ✓ Bonus) Discuss the little endian using the below figure.
  - ✓ Due: until 6 PM Friday of this week (4th, December)



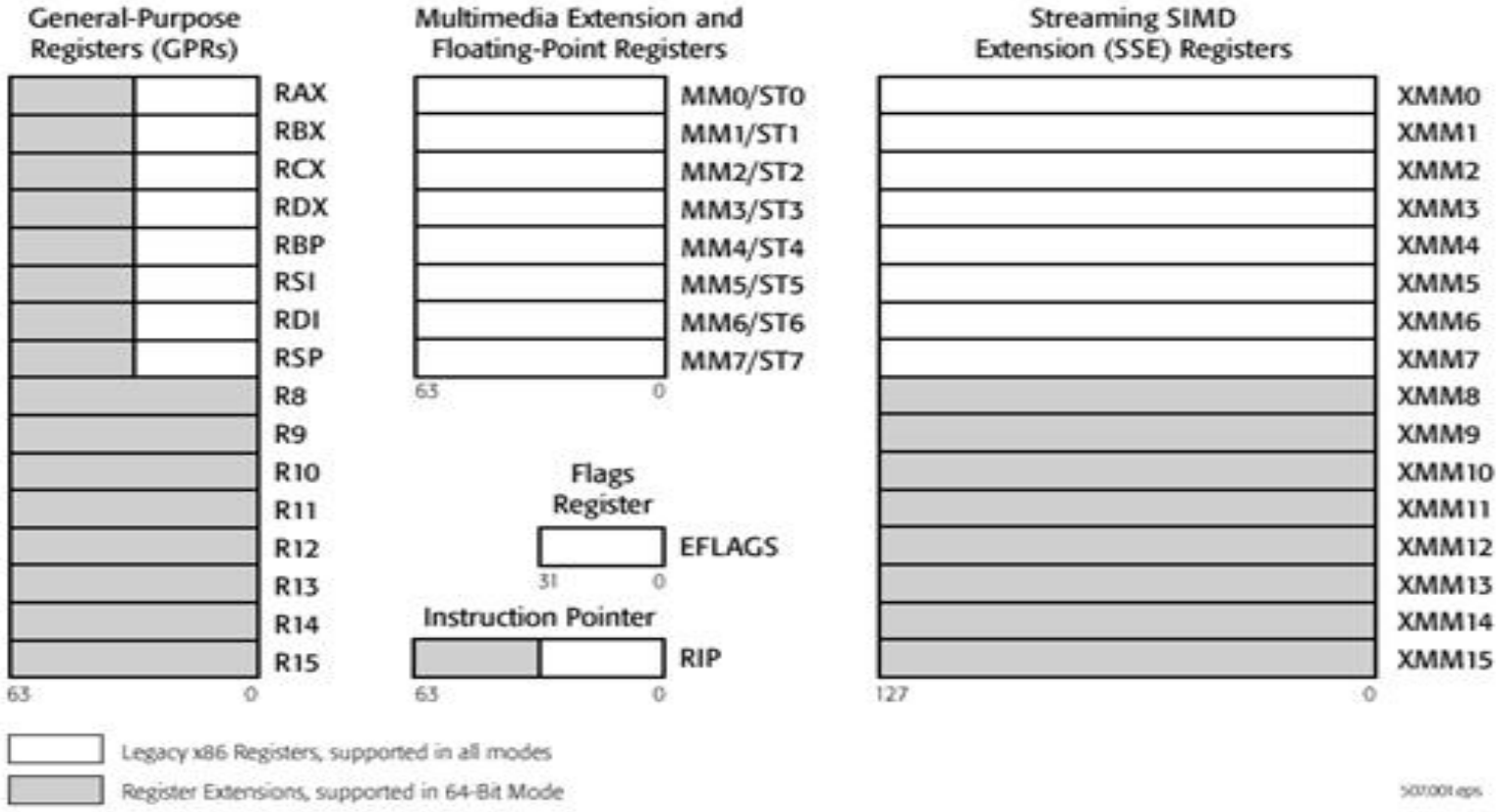
(Source: LN 1 What is System programming?)



(Source: LN 0 Lecture Introduction)

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

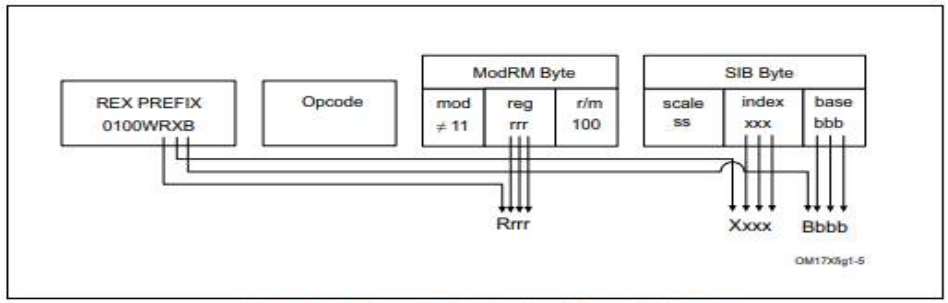


Figure 2-6. Memory Addressing With a SIB Byte

(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

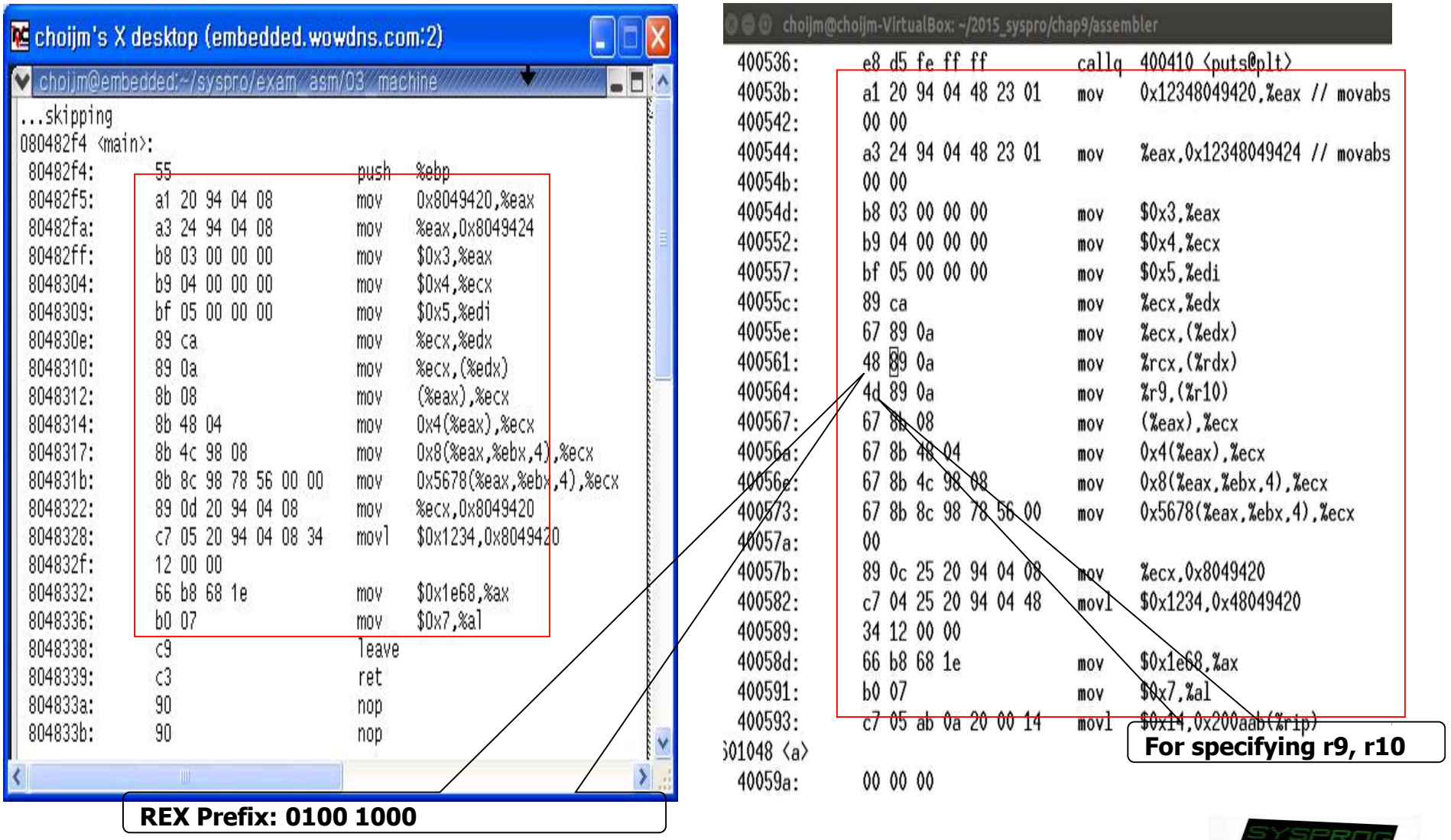
MOV—Move

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*.
8E /r	MOV *Sreg,r/m16***	RM	Valid	Valid	Move *r/m16* to segment register.
REX.W + 8E /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 /0 *ib*	MOV *r/m8, imm8*	MI	Valid	Valid	Move *imm8* to *r/m8*.
REX + C6 /0 *ib*	MOV *r/m8***, imm8*	MI	Valid	N.E.	Move *imm8* to *r/m8*.
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*.

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

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

- Translation example

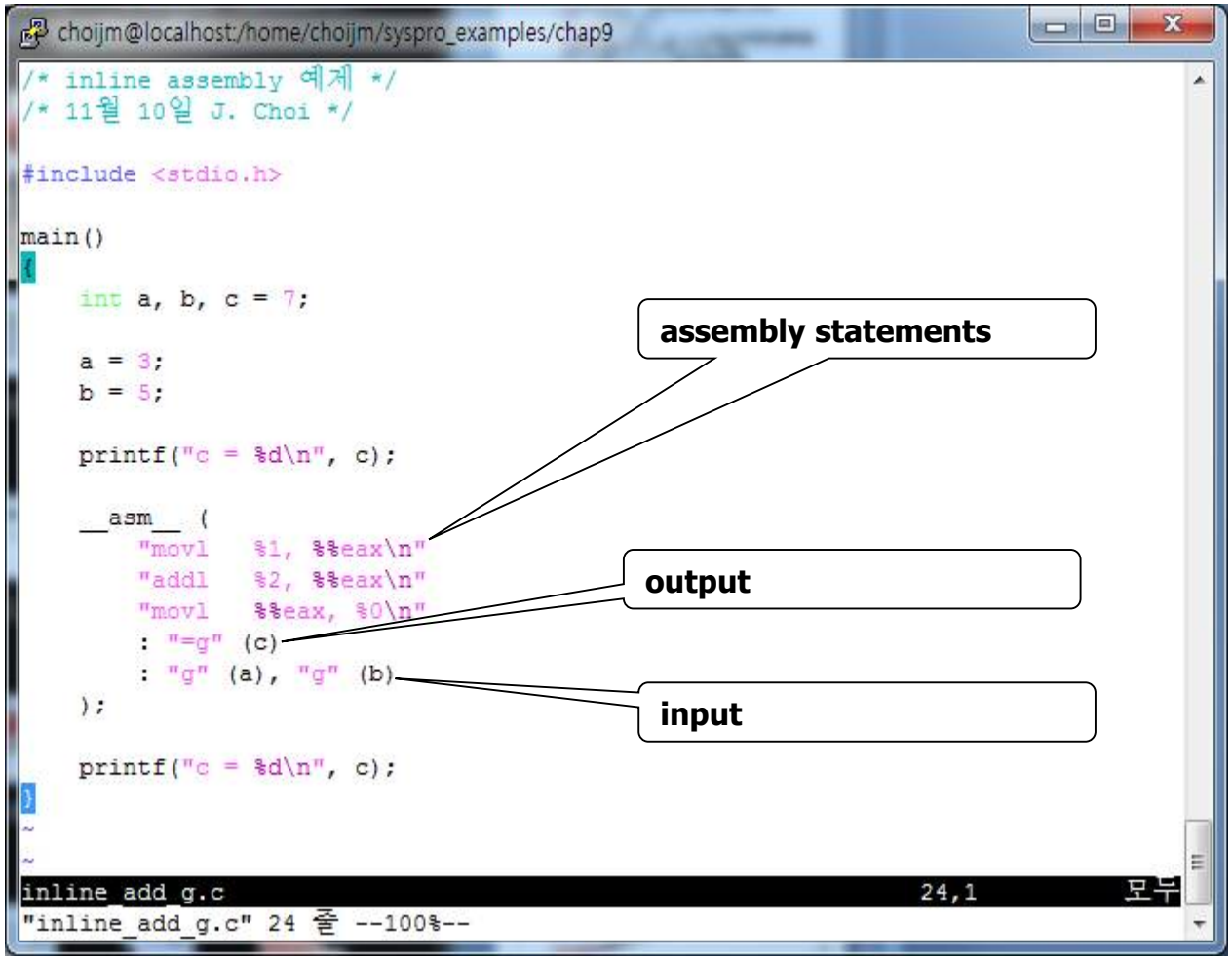


inline Assembly (1/6)

- 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



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

choijm@localhost:/home/choijm/syspro_examples/chap9

```
/* 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%--
```

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__ (
        "mull   %%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                                11,2-5        모두
```

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: 스 택  구 조  분 석  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: \t%p, \t%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                    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 7: Make an assembler**

✓ Requirements

- build an assembler that can translate assembly codes into the IA machine codes shown in slides 6~7.

- manipulate DB and do error handling

- shows student’s ID and date (using whoami and date)

- Make a report that includes a snapshot and discussion.

1) Upload the report to the e-Campus (pdf format!!, 6pm. 18th December)

2) Send the report and source code to TA (이성현: wwbabaww@gmail.com)

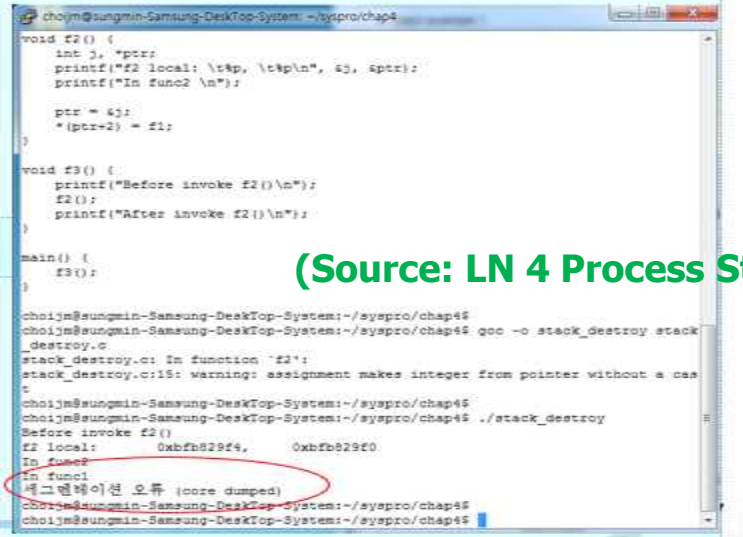
Quiz for 15th-Week 1st-Lesson

- Quiz
  - ✓ 1. Explain how x86-64 maintain the backward compatibility.
  - ✓ 2. In page 19, 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.
  - ✓ Due: until 6 PM Friday of this week (11th, December)

Stack Details (6/6)

- Stack example 2

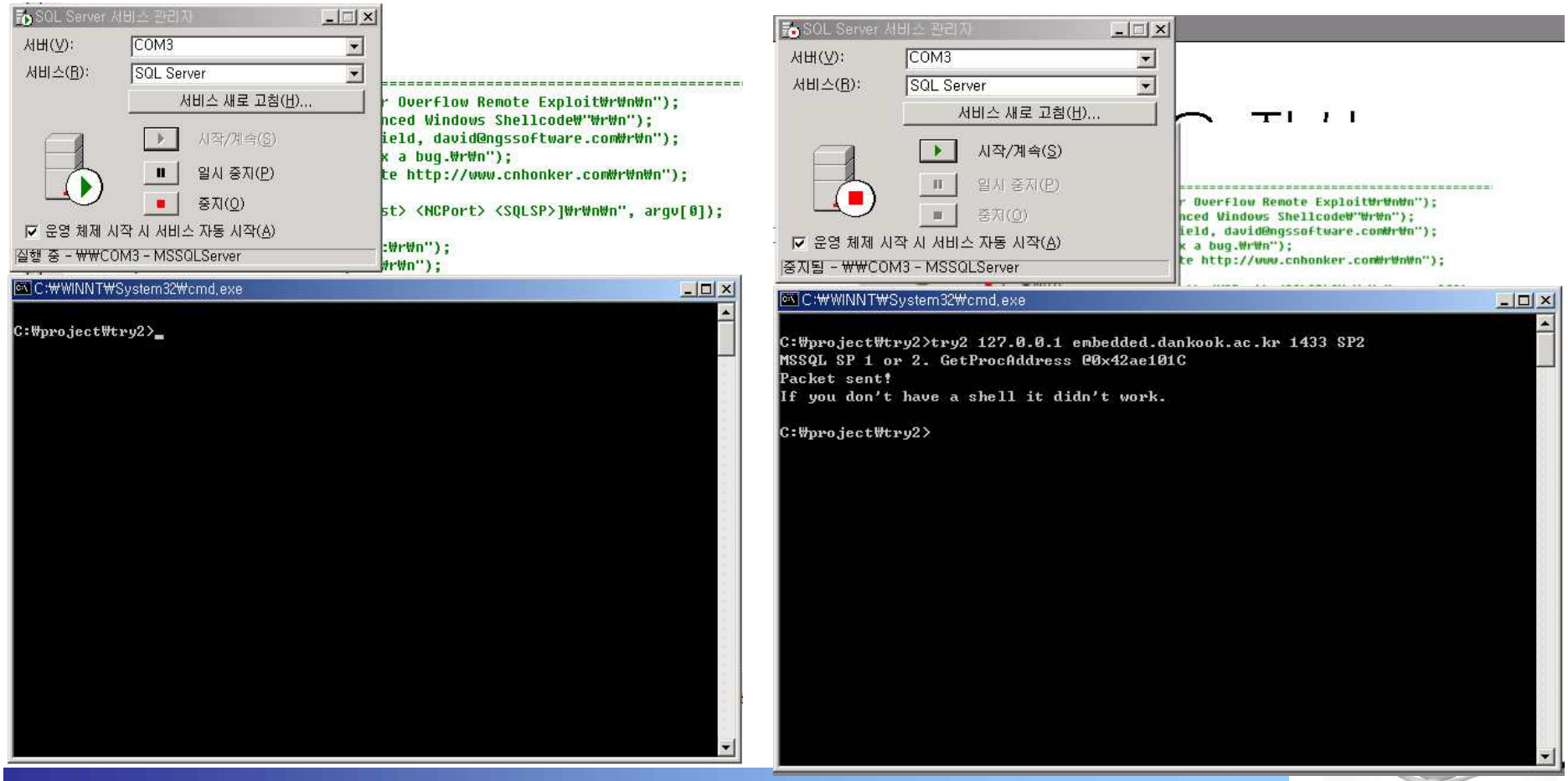
```
/* stack_destroy.c: 스택 구조 분석 2, 9월 19일, choijm@dku.edu */
#include <stdio.h>

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

void f2() {
   int j, *ptr;
   printf("f2 local: \t%p, \t%p\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();
}
```



(Source: LN 4 Process Structure)

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