# Lecture Note 6.
# IA Assembly Programming

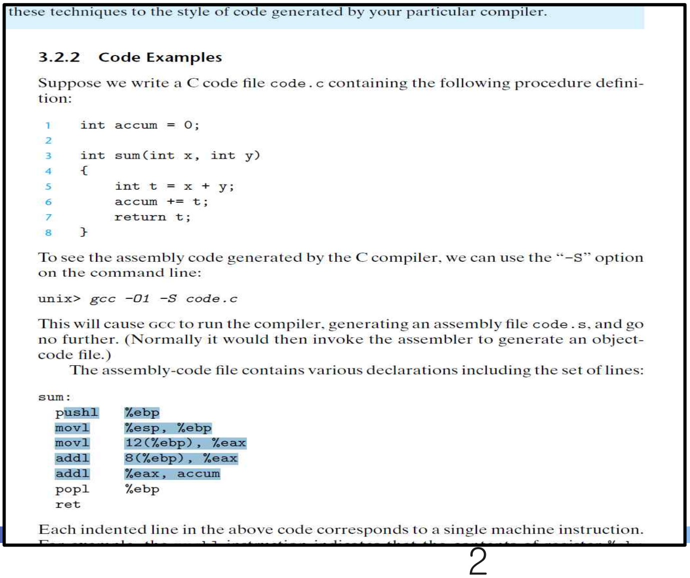
October 12, 2020
Jongmoo Choi
Dept. of Software
Dankook University

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

# Objectives

- Understand various viewpoints about CPU
- Apprehend the concept of ISA (Instruction Set Architecture)
  - ✓ Learn the IA Register model
  - ✓ Learn the IA Memory model
  - ✓ Learn the IA Program model
- Make a program with IA assembly language

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



these techniques to the style of code generated by your particular compiler.

**3.2.2 Code Examples**

Suppose we write a C code file code.c containing the following procedure definition:

```
int accum = 0;

int sum(int x, int y)
{
    int t = x + y;
    accum += t;
    return t;
}
```

To see the assembly code generated by the C compiler, we can use the "-S" option on the command line:

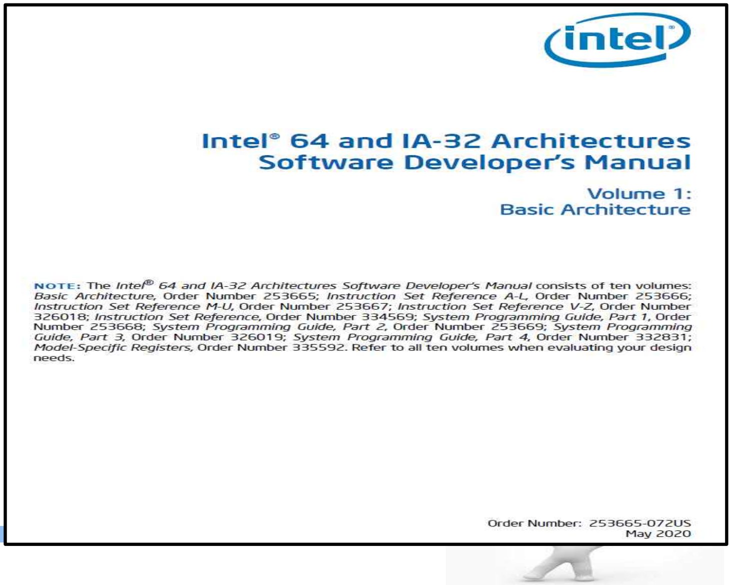
```
unix> gcc -O1 -S code.c
```

This will cause GCC to run the compiler, generating an assembly file code.s, and go no further. (Normally it would then invoke the assembler to generate an object-code file.)

The assembly-code file contains various declarations including the set of lines:

```
sum:
  pushl   %ebp
  movl    %esp, %ebp
  movl    12(%ebp), %eax
  addl    8(%ebp), %eax
  addl    %eax, accum
  popl    %ebp
  ret
```

Each indented line in the above code corresponds to a single machine instruction.



intel

Intel® 64 and IA-32 Architectures
Software Developer's Manual

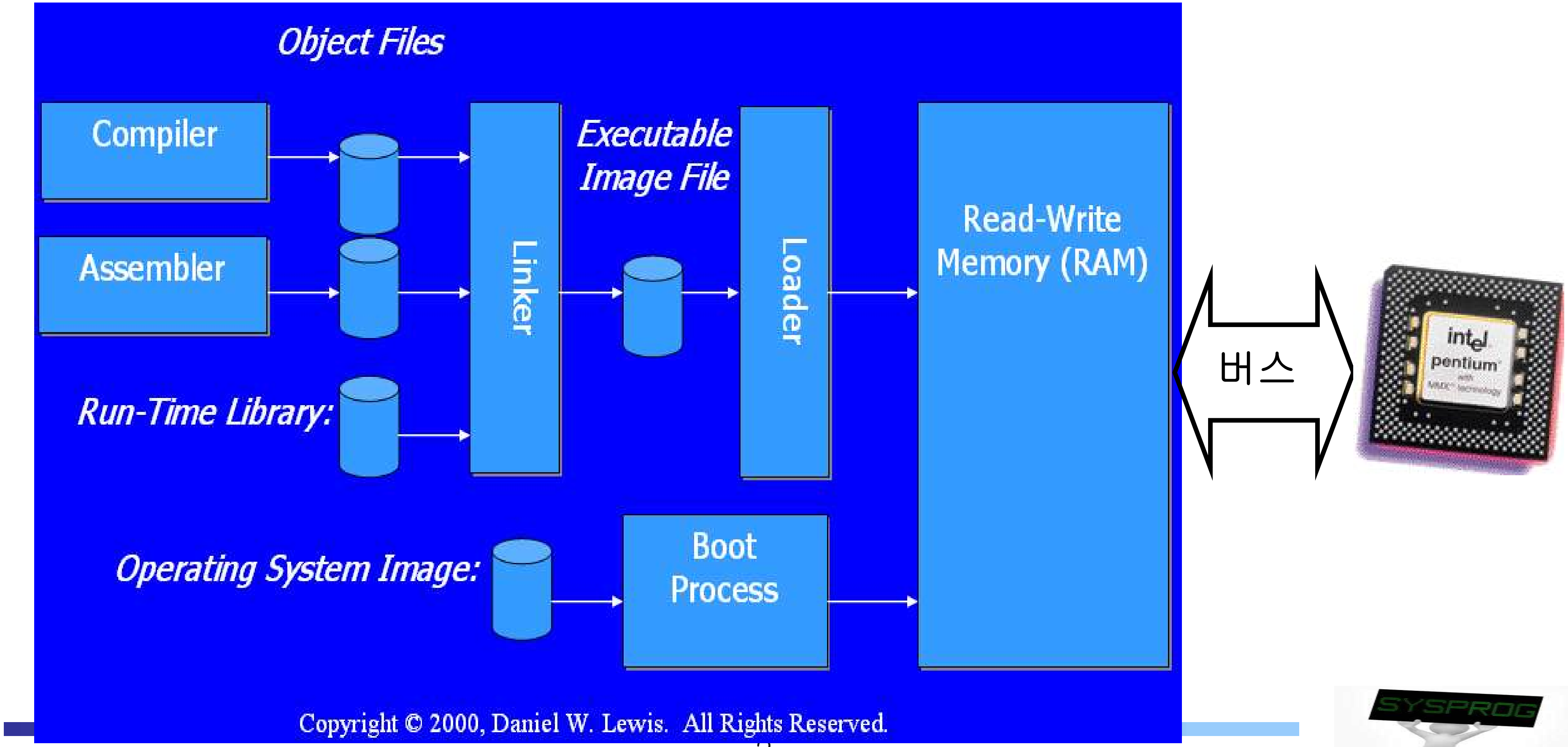
Volume 1:
Basic Architecture

**NOTE:** The *Intel® 64 and IA-32 Architectures Software Developer's Manual* consists of ten volumes: *Basic Architecture*, Order Number 253665; *Instruction Set Reference A-L*, Order Number 253666; *Instruction Set Reference M-U*, Order Number 253667; *Instruction Set Reference V-Z*, Order Number 326018; *Instruction Set Reference*, Order Number 334569; *System Programming Guide, Part 1*, Order Number 253668; *System Programming Guide, Part 2*, Order Number 253669; *System Programming Guide, Part 3*, Order Number 326019; *System Programming Guide, Part 4*, Order Number 332831; *Model-Specific Registers*, Order Number 335592. Refer to all ten volumes when evaluating your design needs.

Order Number: 253665-072US
May 2020

# Introduction (1/2)

- Summarizing what we have learnt
  - ✓ Program development: compile, linking, ELF, …
  - ✓ Program execution: task (text, data, stack), load, fetch, …
    - text: consists of machine instructions



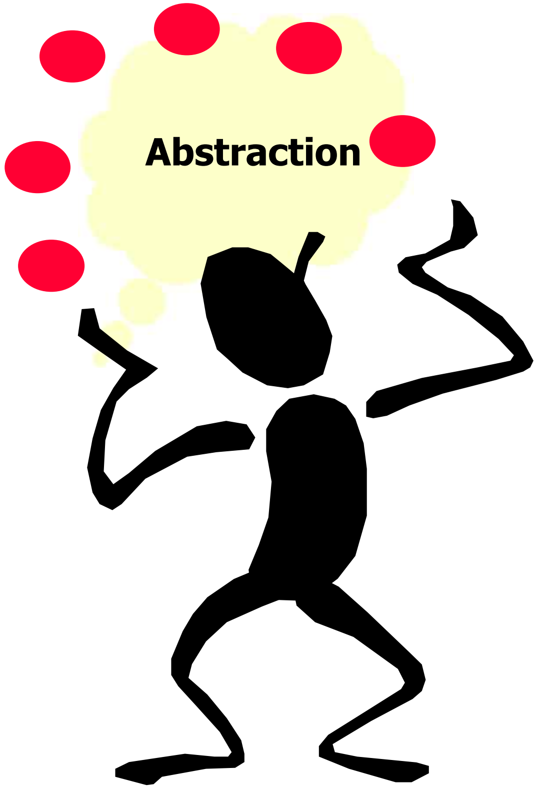
Copyright © 2000, Daniel W. Lewis. All Rights Reserved.

# Introduction (2/2)

- Assembly language
  - ✓ Language hierarchy
    - locate between high-level language and machine language
    - Symbolic (mnemonic) representation of machine language
      - One-to-one mapping, CPU dependent (Not easy)
  - ✓ Application field
    - Hardware control: system initialization, device driver, interrupt handler, embedded systems, IoT, ECU, CPS, Wearable computer, …
    - Vulnerability test (Virus identification, IDS)
    - Optimization
    - SW copyright protection, SW similarity analysis, …
  - ✓ Importance
    - Making a program, debugging, analyzing binary
    - Understand the behavior of hardware (especially CPU)
    - Grape the mechanism how hardware and software are cooperated (hardware software co-design)

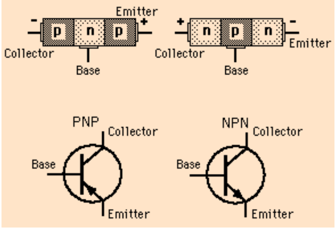
# CPU (1/5)

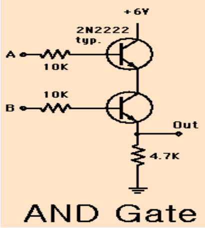
- What is a Processor?

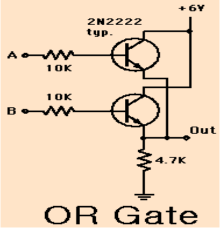


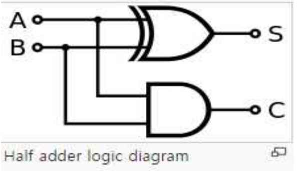
# CPU (2/5)

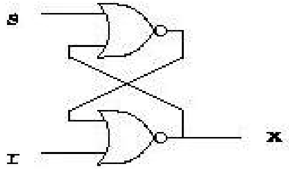
- Various Viewpoints of Processor
  - ✓ 1. Transistor + Gate + Logic + Clock

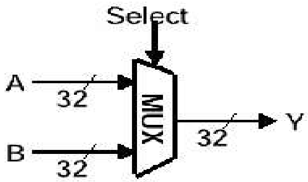


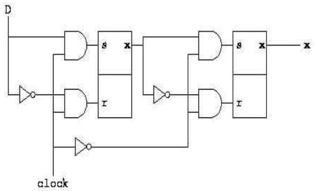












  - ✓ 2. ALU (Arithmetic Logic Unit) + Registers + CU (Control Unit) + BUS

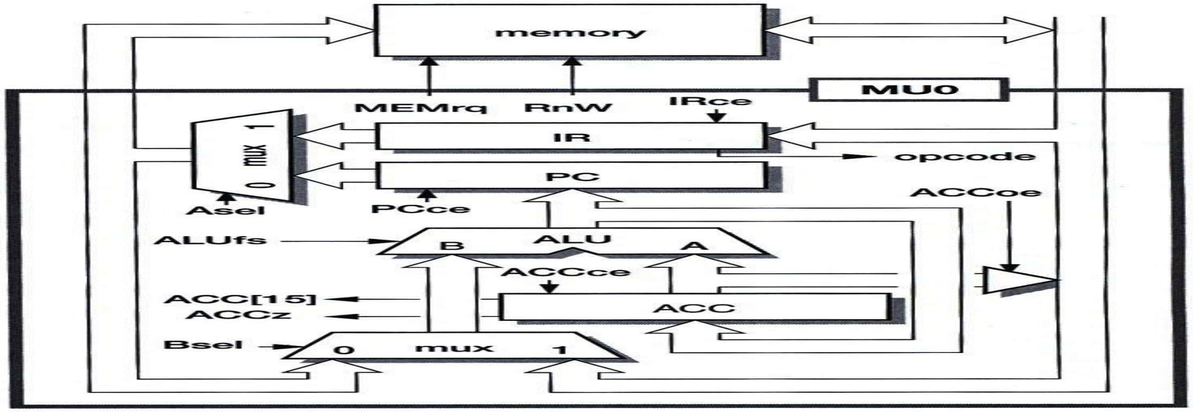


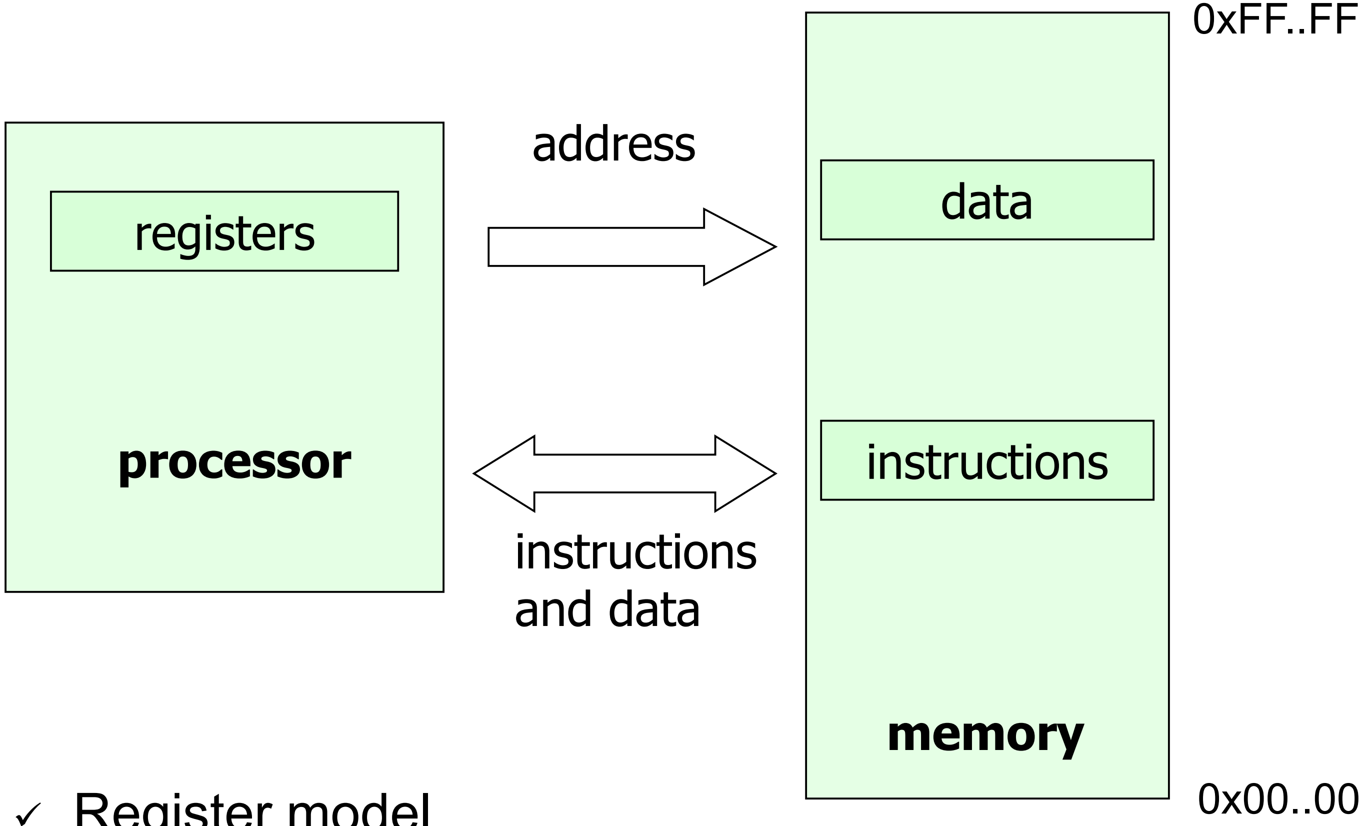
**Figure 1.6** MU0 register transfer level organization.

**(Source: MU0 in Appendix 1)**

  - ✓ 3. Instruction Set Architecture (CISC, RISC, VLIW, EPIC, …)
  - ✓ 4. Performance Characteristics (Pipeline, Superscalar, Cache, …)

# CPU (3/5)

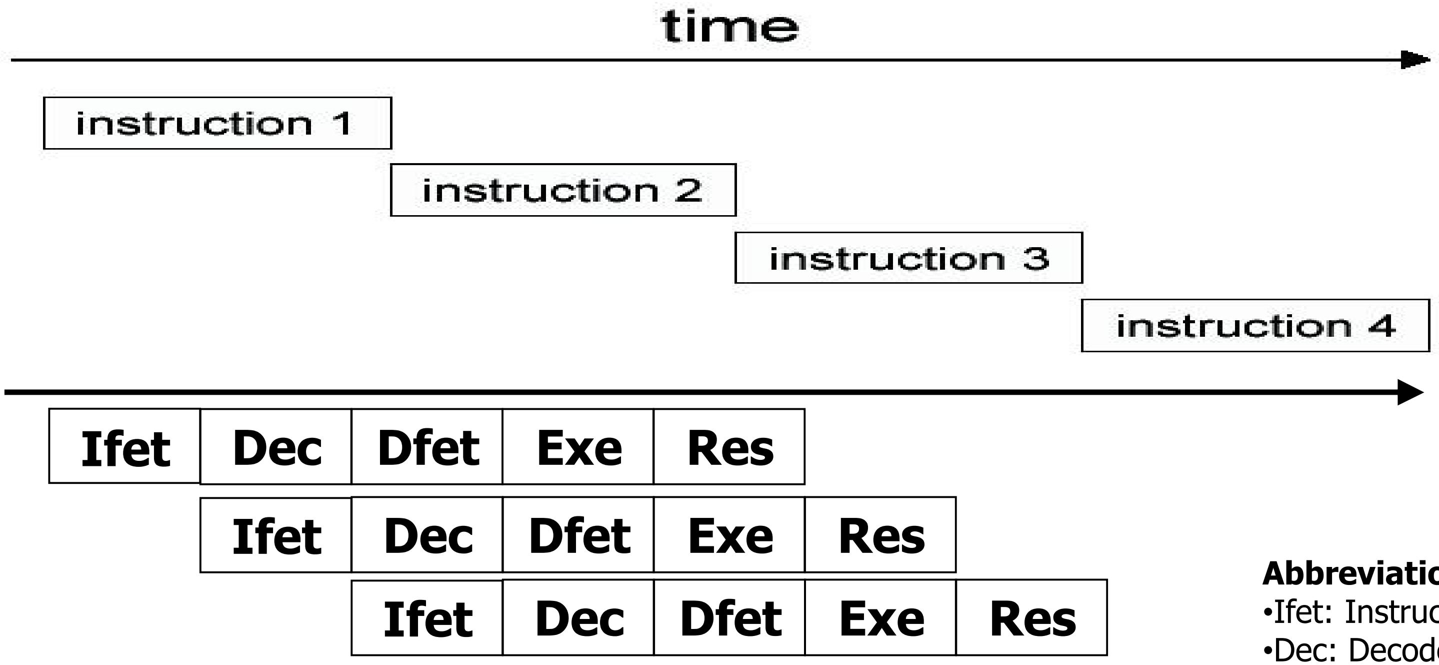
- Instruction Set Architecture: Register + Instructions



- ✓ Register model
- ✓ Memory model
- ✓ Instruction model

# CPU (4/5)

- Performance Characteristics: Pipeline, Superscalar, Cache



time

instruction 1

instruction 2

instruction 3

instruction 4

| | | | | | | |
|---|---|---|---|---|---|---|
| Ifet | Dec | Dfet | Exe | Res | | |
| | Ifet | Dec | Dfet | Exe | Res | |
| | | Ifet | Dec | Dfet | Exe | Res |

**Abbreviation**
- •Ifet: Instruction fetch
- •Dec: Decode
- •Dfet: Data fetch
- •Exe: Execution
- •Res: Results write

✓ For efficient pipeline

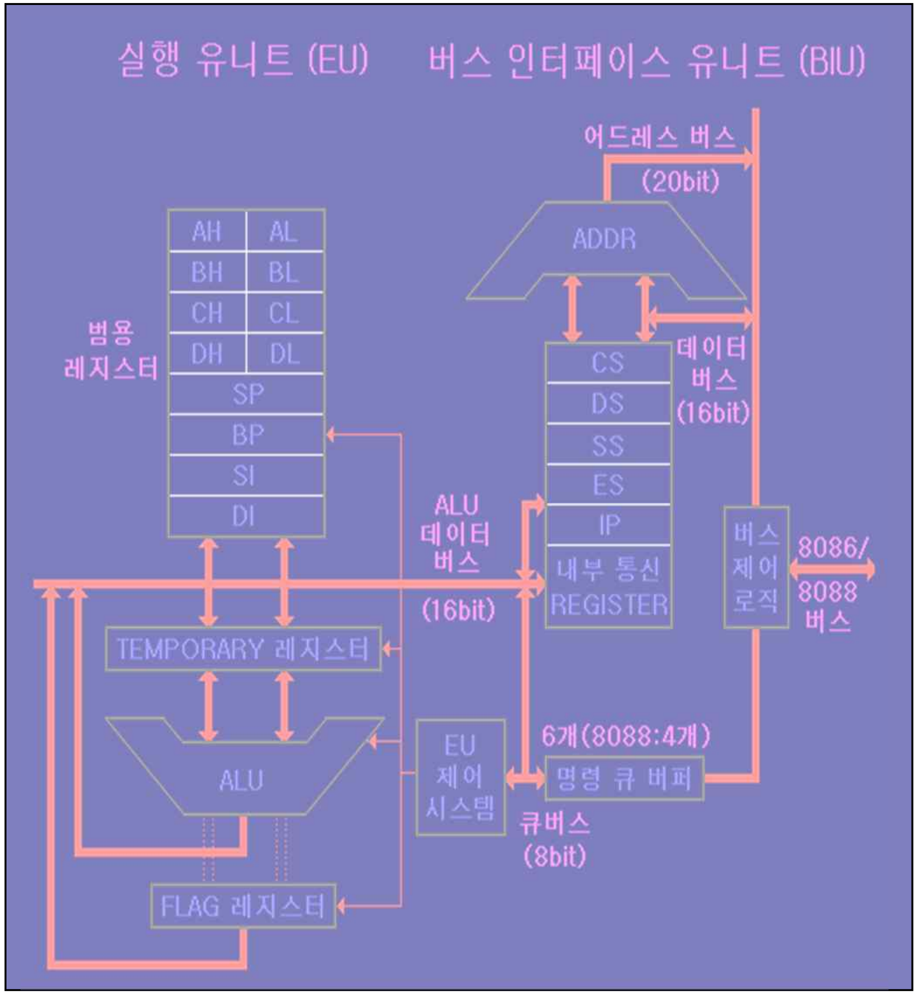
- Similar latency of instructions (not complex)
- Conflict between I. fetch and D. fetch
- Branch prediction, Out-of order executions
- L1, L2 cache …

☞ **Details will be discussed in LN 7**

# CPU (5/5)

- Performance Characteristics: Pipeline, Superscalar, Cache

8086



Pentium

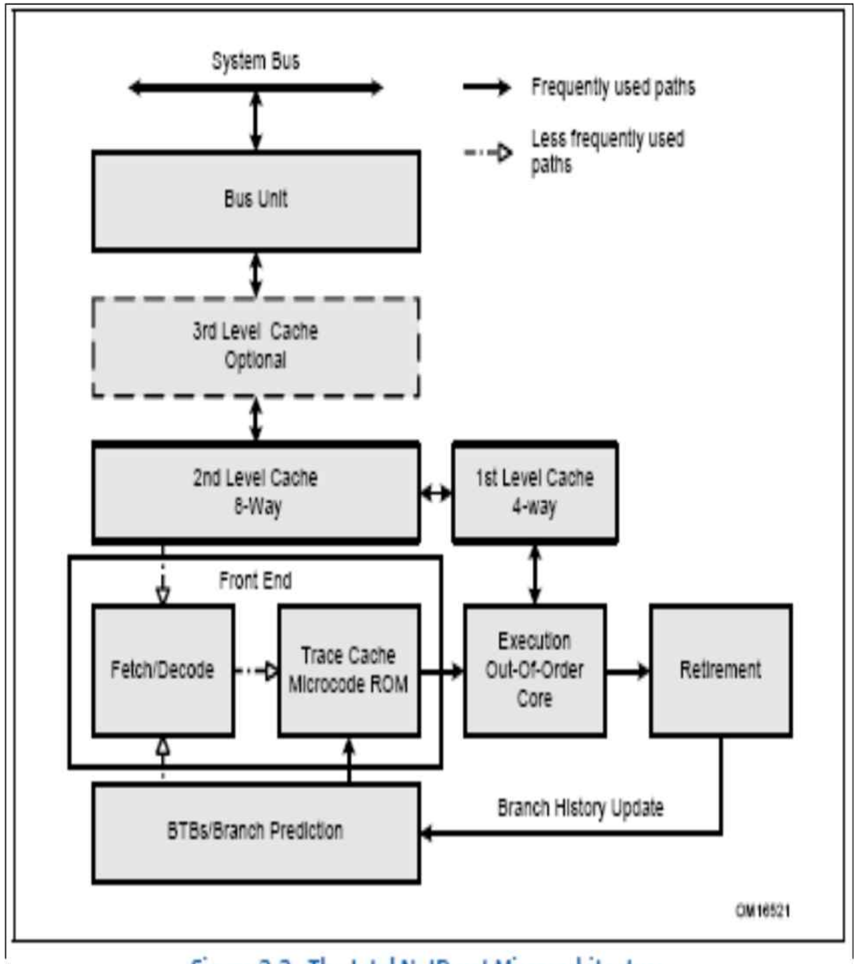


Figure 2-2. The Intel NetBurst Microarchitecture

(Source: Intel SW Developer's Manual, Volume 1: Basic Architecture)

# Register Model (1/3)

- Register definition
  - ✓ A small amount of memory available in a CPU
  - ✓ Can be accessed quickly, compared with main memory
- IA registers

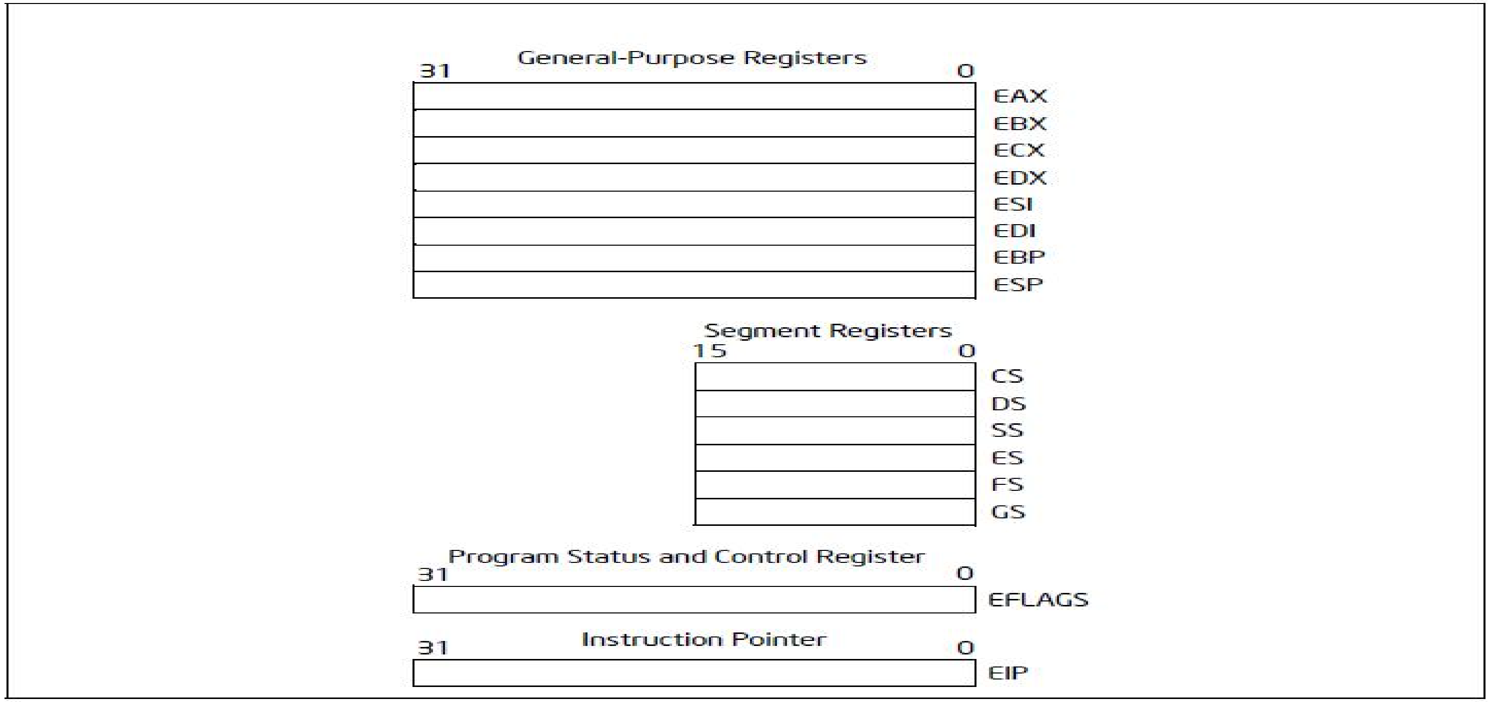


Figure 3-4. General System and Application Programming Registers

**(Source: Intel SW Developer's Manual, Volume 1: Basic Architecture)**

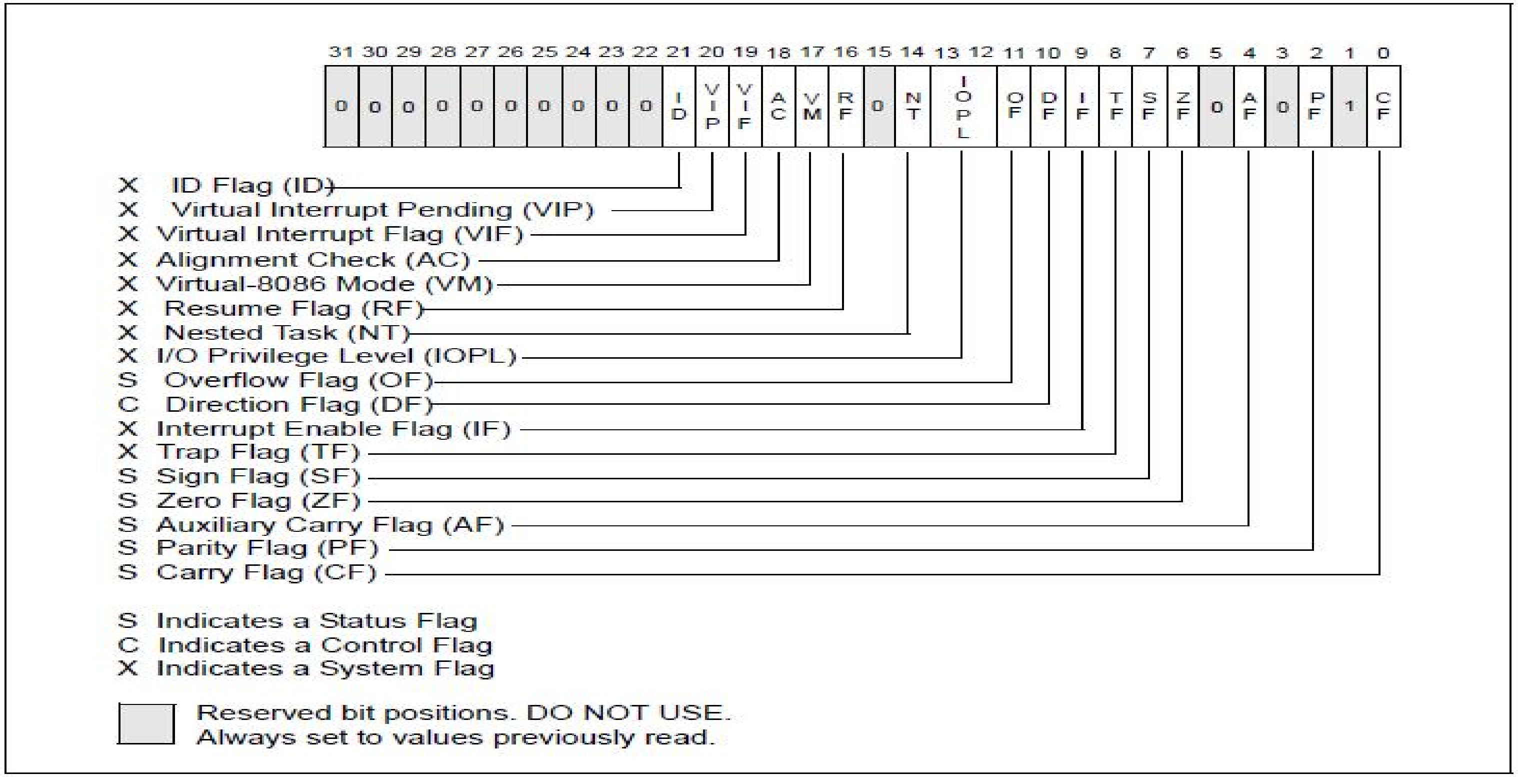
# Register Model (2/3)

- Functionality of each register
  - ✓ Segment register
    - CS(code segment): the base location of all executable instructions
    - DS(data segment): the base location for variables
    - SS(stack segment): the base location of the stack
    - ES(extra segment): an additional base location for variables
  - ✓ General purpose register
    - EAX (accumulator): for arithmetic operation (operand and result data)
    - EBX (base): pointer to data in the DS segment
    - ECX (counter): counter for loop and string operations
    - EDX (data): I/O pointer, a special role in multiply and divide operations
    - ESP (stack pointer): pointer to the top of the stack
    - EBP (base pointer): used as base for accessing variables on the stack (base for stack frame)
    - ESI (source index): source pointer for string operations
    - EDI (destination index): destination pointer for string operations
    - Having its specialty, but commonly being used for general purpose
  - ✓ EIP (instruction pointer): role of PC(Program counter)
  - ✓ EFLAGS: Control and Status Register

☞ **rax, rbx, rip, … for Intel 64**

# Register Model (3/3)

- Details of EFLAGS register
  - ✓ Set of control and status Flags



| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | ID | VIP | VIF | AC | VM | RF | 0 | NT | IOPL | | OF | DF | IF | TF | SF | ZF | 0 | AF | 0 | PF | 1 | CF |

X ID Flag (ID)
X Virtual Interrupt Pending (VIP)
X Virtual Interrupt Flag (VIF)
X Alignment Check (AC)
X Virtual-8086 Mode (VM)
X Resume Flag (RF)
X Nested Task (NT)
X I/O Privilege Level (IOPL)
S Overflow Flag (OF)
C Direction Flag (DF)
X Interrupt Enable Flag (IF)
X Trap Flag (TF)
S Sign Flag (SF)
S Zero Flag (ZF)
S Auxiliary Carry Flag (AF)
S Parity Flag (PF)
S Carry Flag (CF)

S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag

▨ Reserved bit positions. DO NOT USE. Always set to values previously read.

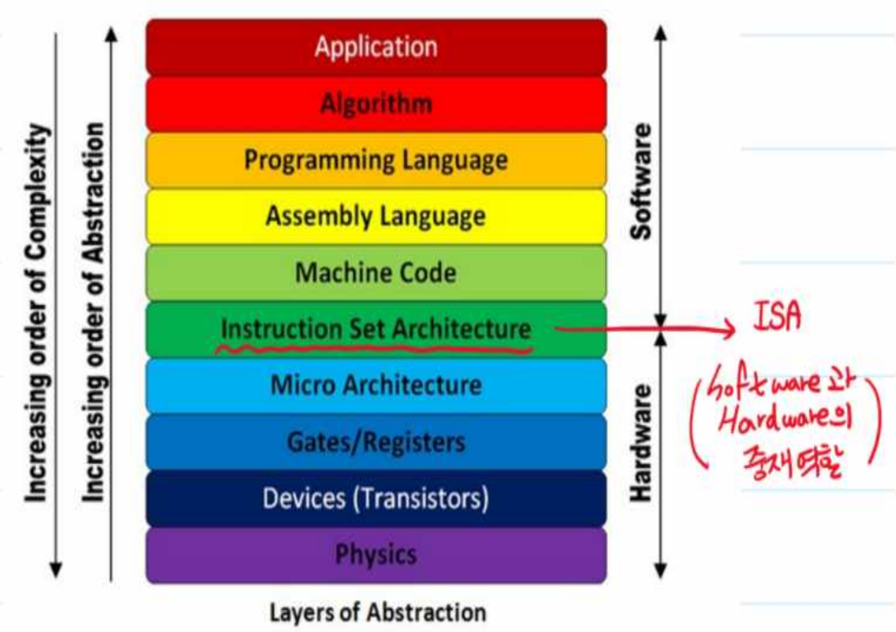
Figure 3-8. EFLAGS Register

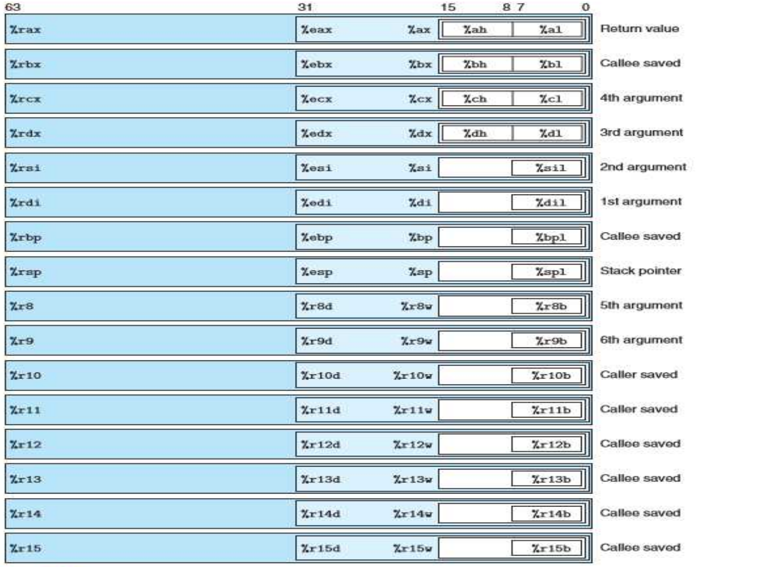
☞ **Refer to the IA-32 Basic Architecture, Chapter 3.4.3 for the role of each bit**

☞ **Intel CPU has several additional registers such as CR0, CR2, CR3, IDTR, GDTR, debugging registers, FPU registers, and MMX registers. (see LN_chapter 7)**

# Quiz for 9th-Week 1st-Lesson

- Quiz
  - ✓ 1. There are various viewpoints regarding CPU. What is the ISA? Explain three components of ISA.
  - ✓ 2. There are 8 GP registers in 32-bit Intel CPU. It increases 16 in 64-bit Intel CPU. Discuss the merit and demerit of larger registers.
  - ✓ Due: until 6 PM Friday of this week (30th, October)





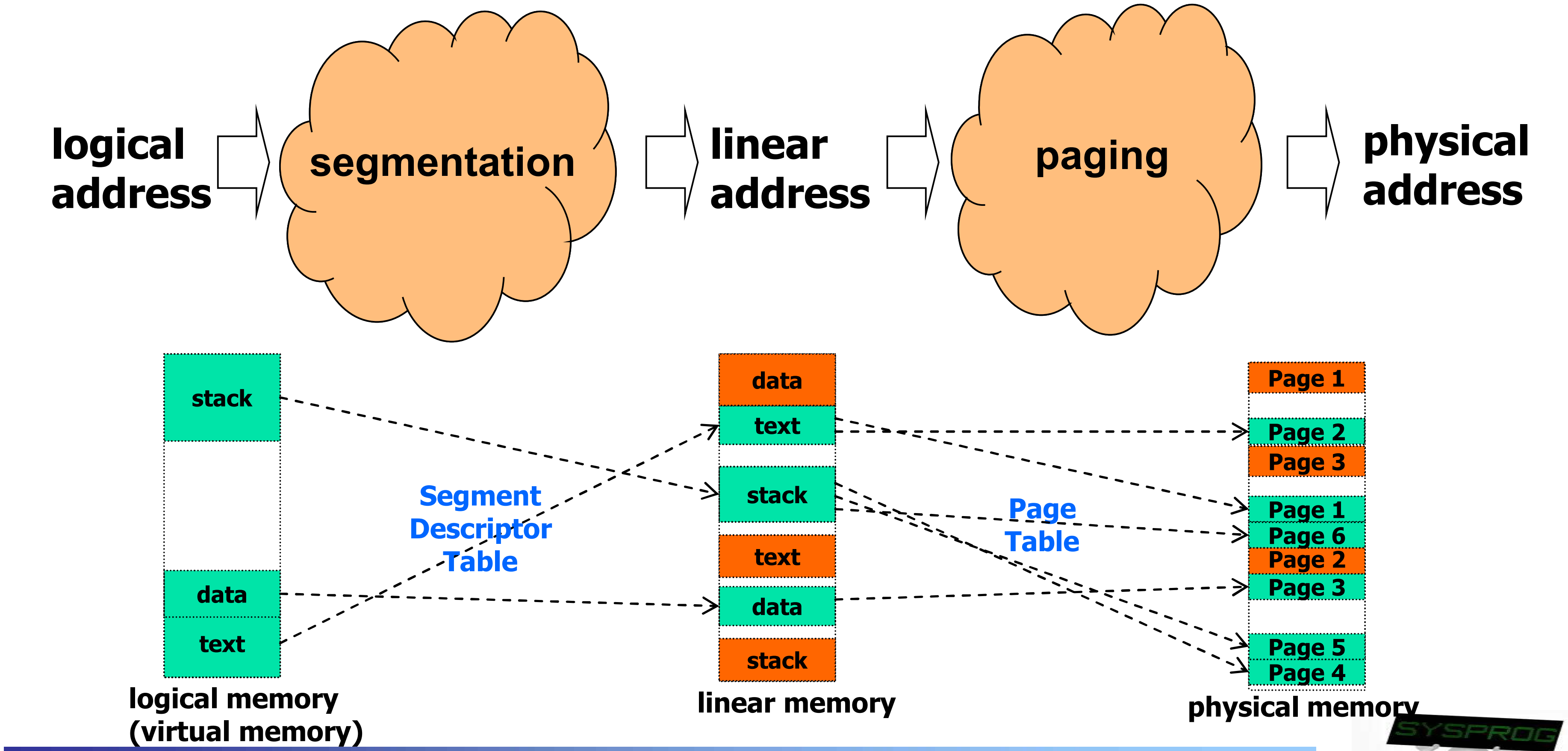
| 63 | 31 | 15 | 8 7 | 0 | |
|---|---|---|---|---|---|
| %rax | %eax | %ax | %ah | %al | Return value |
| %rbx | %ebx | %bx | %bh | %bl | Callee saved |
| %rcx | %ecx | %cx | %ch | %cl | 4th argument |
| %rdx | %edx | %dx | %dh | %dl | 3rd argument |
| %rsi | %esi | %si | | %sil | 2nd argument |
| %rdi | %edi | %di | | %dil | 1st argument |
| %rbp | %ebp | %bp | | %bpl | Callee saved |
| %rsp | %esp | %sp | | %spl | Stack pointer |
| %r8 | %r8d | %r8w | | %r8b | 5th argument |
| %r9 | %r9d | %r9w | | %r9b | 6th argument |
| %r10 | %r10d | %r10w | | %r10b | Caller saved |
| %r11 | %r11d | %r11w | | %r11b | Caller saved |
| %r12 | %r12d | %r12w | | %r12b | Callee saved |
| %r13 | %r13d | %r13w | | %r13b | Callee saved |
| %r14 | %r14d | %r14w | | %r14b | Callee saved |
| %r15 | %r15d | %r15w | | %r15b | Callee saved |

Figure 3.35 **Integer registers.** The existing eight registers are extended to 64-bit versions, and eight new registers are added. Each register can be accessed as either 8 bits (byte), 16 bits (word), 32 bits (double word), or 64 bits (quad word).

**(Source: http://melonicedlatte.com/computerarchitecture/2019/01/30/192433.html)**

# Memory Model (1/6)

- Memory abstraction in IA
  - ✓ logical address (virtual address)
  - ✓ linear address
  - ✓ physical address



# Memory Model (2/6)

- Paging and Segmentation in detail
  - ✓ Segmentation: variable size
    - Address translation: base address + offset, using segment table (segment descriptor table)
  - ✓ Paging: fixed size
    - page start address (PT + index) + offset, using page table (commonly multi-level tables)

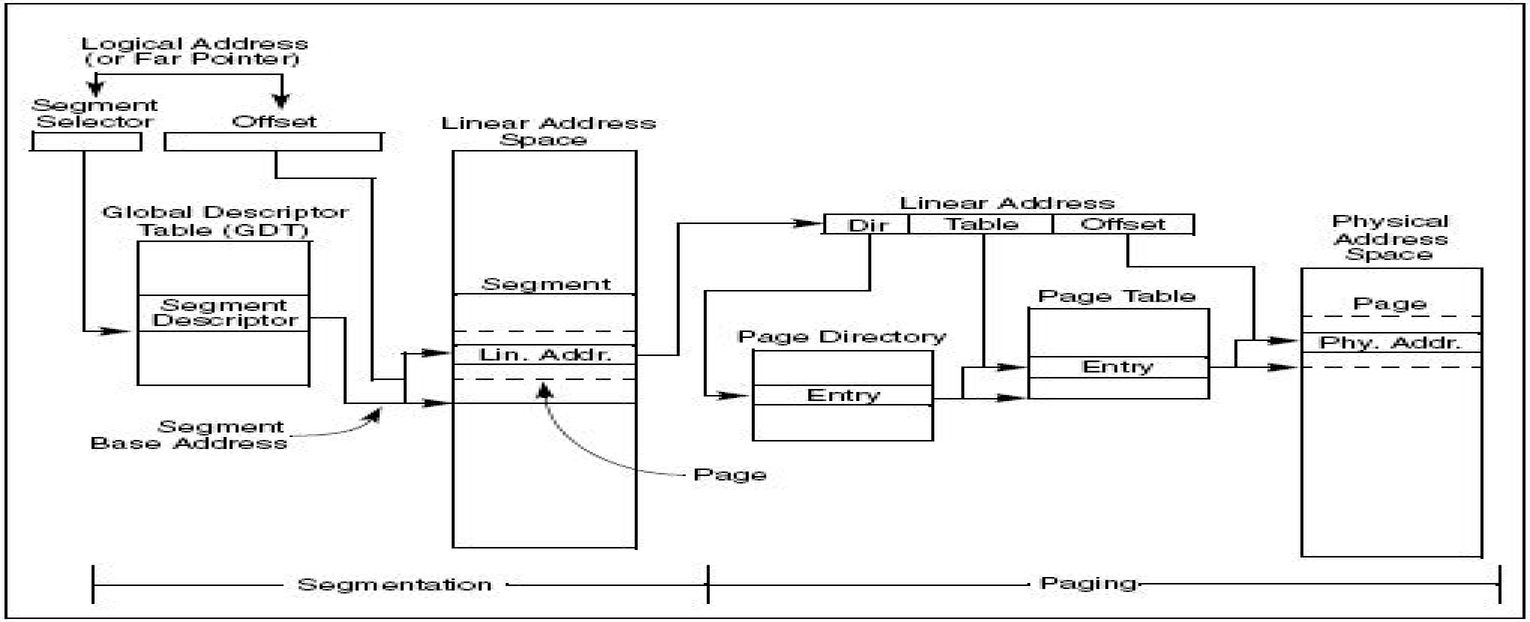


Figure 3-1. Segmentation and Paging

☞ Some CPUs make use of paging only or segmentation only

# Memory Model (3/6)

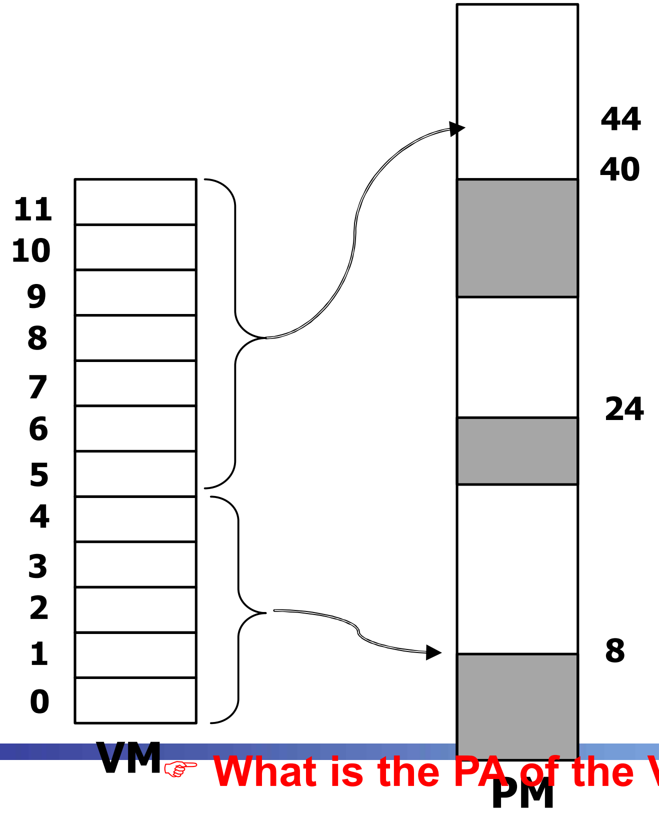
- Segmentation vs Paging example
  - ✓ Assumption
    - Physical memory is fragmented
    - Virtual memory consists of 12 elements
  - ✓ Segmentation vs. Paging
    - Address translation: segment table vs. page table
    - How to: seg # + offset vs. page # + offset

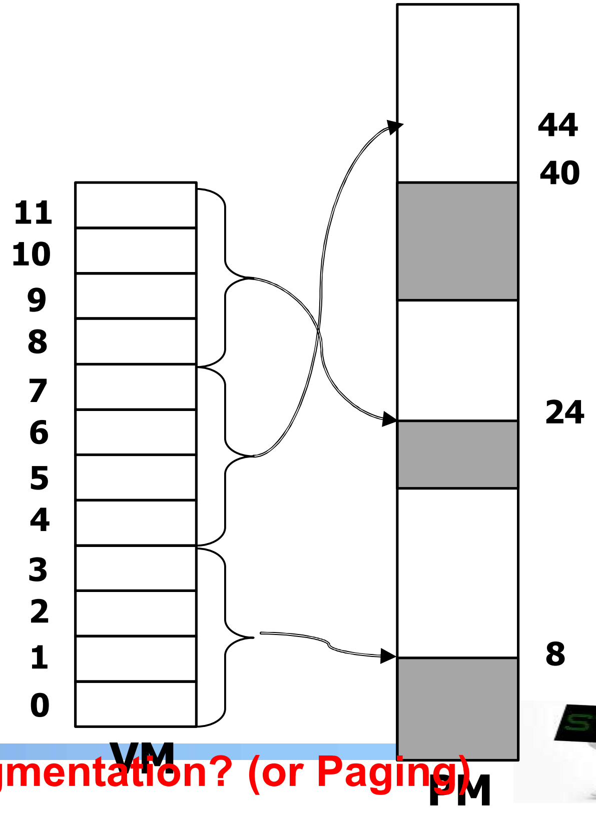
0 → 8, size = 5
5 → 44, size = 7

**Segment table**

0 → 8
4 → 44
8 → 24

**Page table**

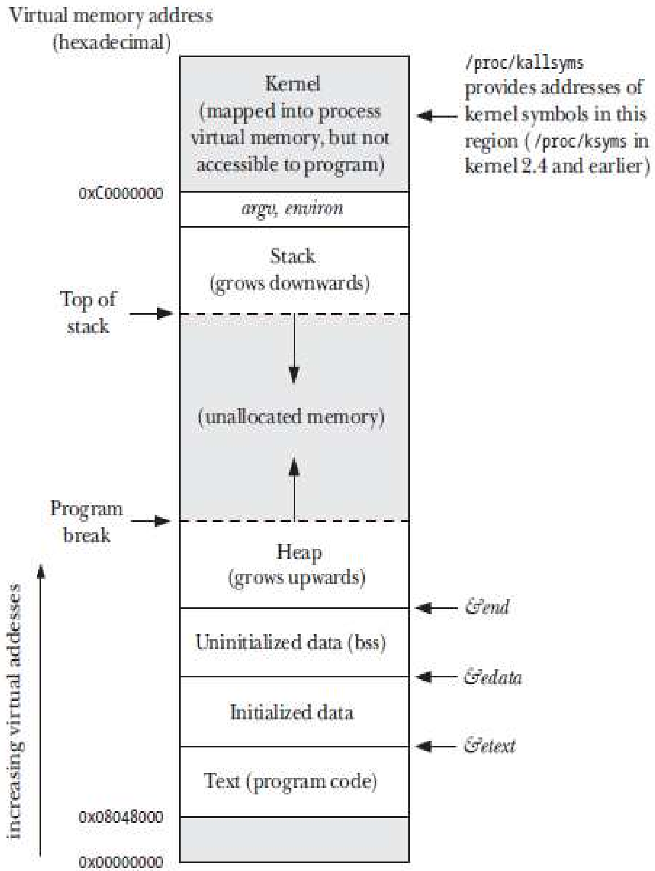




☞ What is the PA of the VA 10 in segmentation? (or Paging)

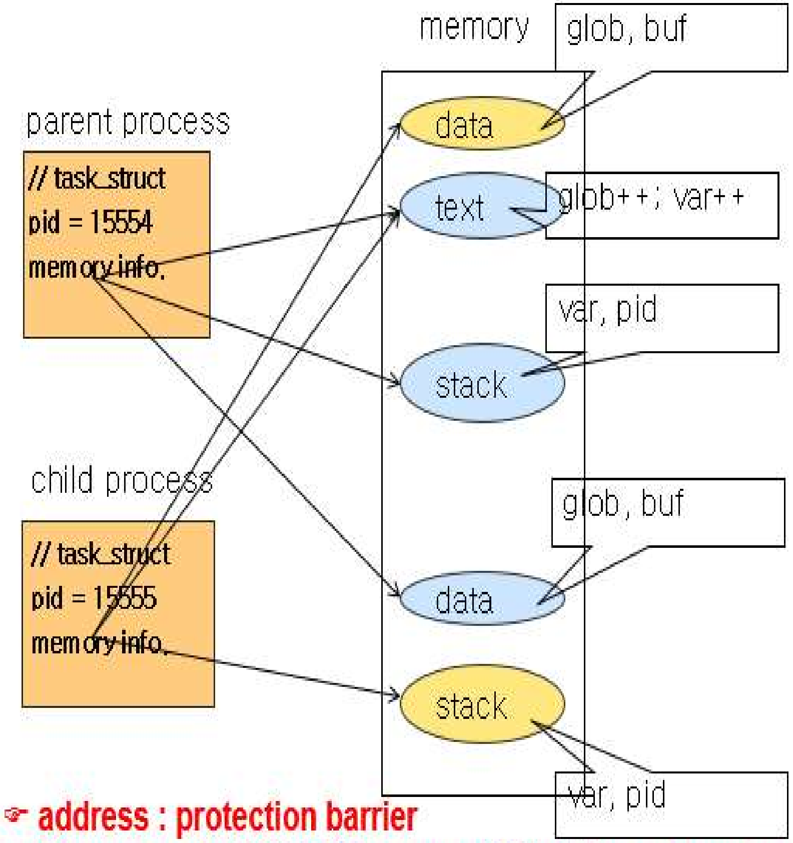
# Memory Model (4/6)

- Revisit
  - ✓ Process structure in LN 4 vs. After fork in LN 5
  - ✓ Virtual memory vs. Using Segmentation



Virtual memory address (hexadecimal)
Kernel (mapped into process virtual memory, but not accessible to program)
/proc/kallsyms provides addresses of kernel symbols in this region ( /proc/ksyms in kernel 2.4 and earlier)
0xC0000000
argv, environ
Stack (grows downwards)
Top of stack
(unallocated memory)
Program break
Heap (grows upwards)
&end
Uninitialized data (bss)
&edata
Initialized data
&etext
Text (program code)
increasing virtual addresses
0x08048000
0x00000000

Figure 6-1: Typical memory layout of a process on Linux/x86-32

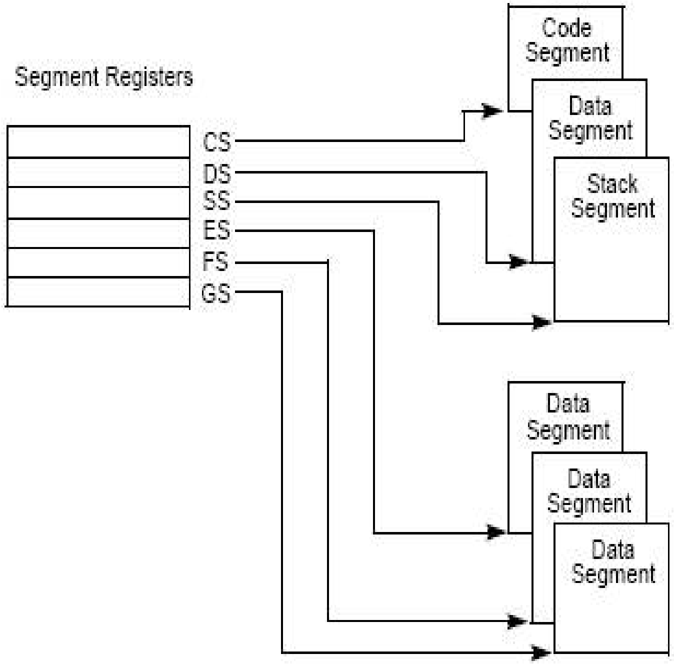


- ☞ address : protection barrier
- ☞ We can exploit "COW(Copy_on_Write)" for enhancing performance
- ☞ We do not consider "Paging" in this slide.

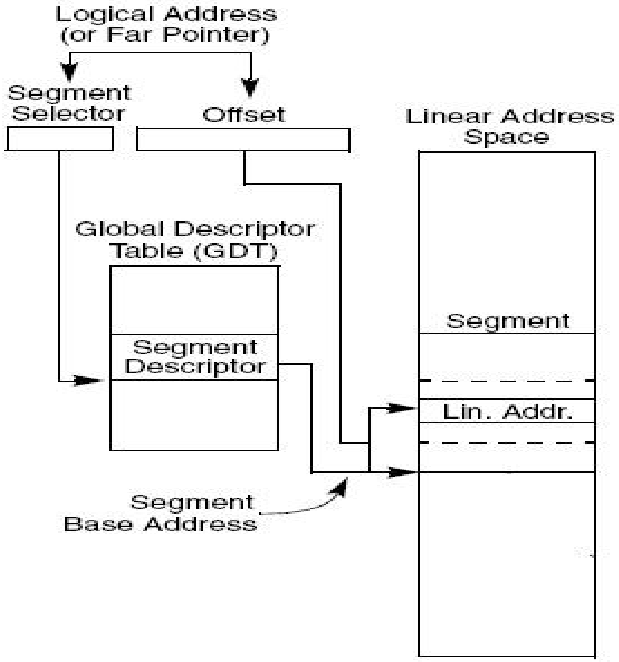
# Memory Model (Optional) (5/6)

- Segmentation on IA
  - ✓ Real Address Model: 8086 compatible, support 1MB (seg.<<4+offset)
  - ✓ Flat Model: protected mode with segment descriptor
  - ✓ Segmented Model: protected mode with segment descriptor table

**real address model**



**segmented model**



# Memory Model (Optional) (6/6)

- Paging on IA
  - ✓ Usually make use of multi-level structure
    - 32 bit: 2-level paging
      - Page directory, page table
    - 64 bit: 4-level paging
      - PML4, page directory pointer, page directory, page table

**32 bit CPU**

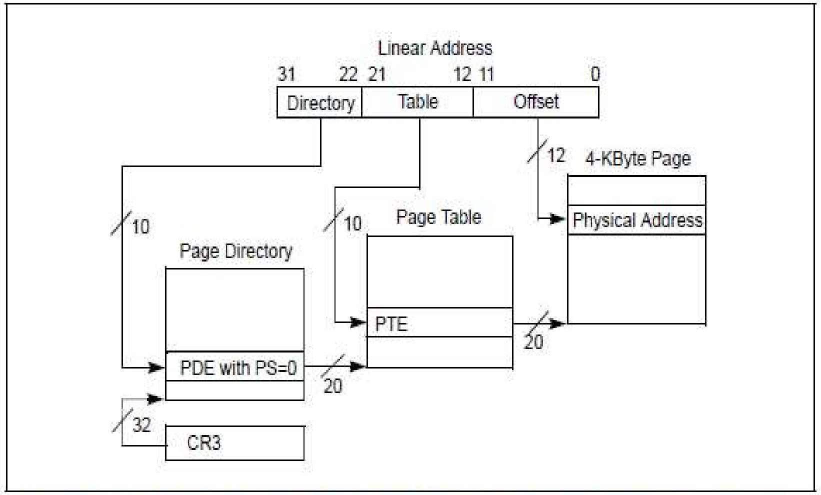


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

**64 bit CPU**

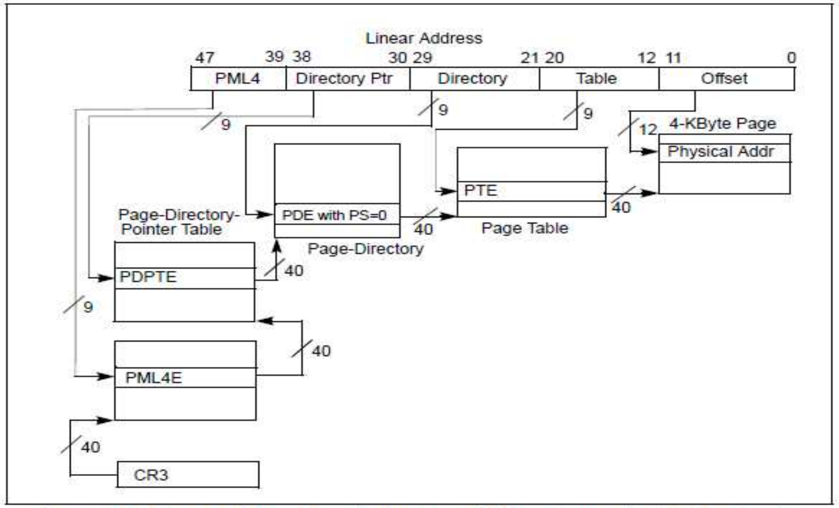


Figure 4-8. Linear-Address Translation to a 4-KByte Page using IA-32e Paging

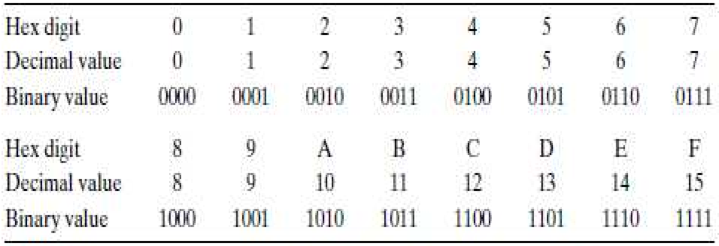
**(Source: Intel SW Developer's Manual, Volume 1: Basic Architecture)**

☞ **The basic concept of address mapping is similar to the indexing in the inode**

# Instruction Model (1/2)

- Instruction format

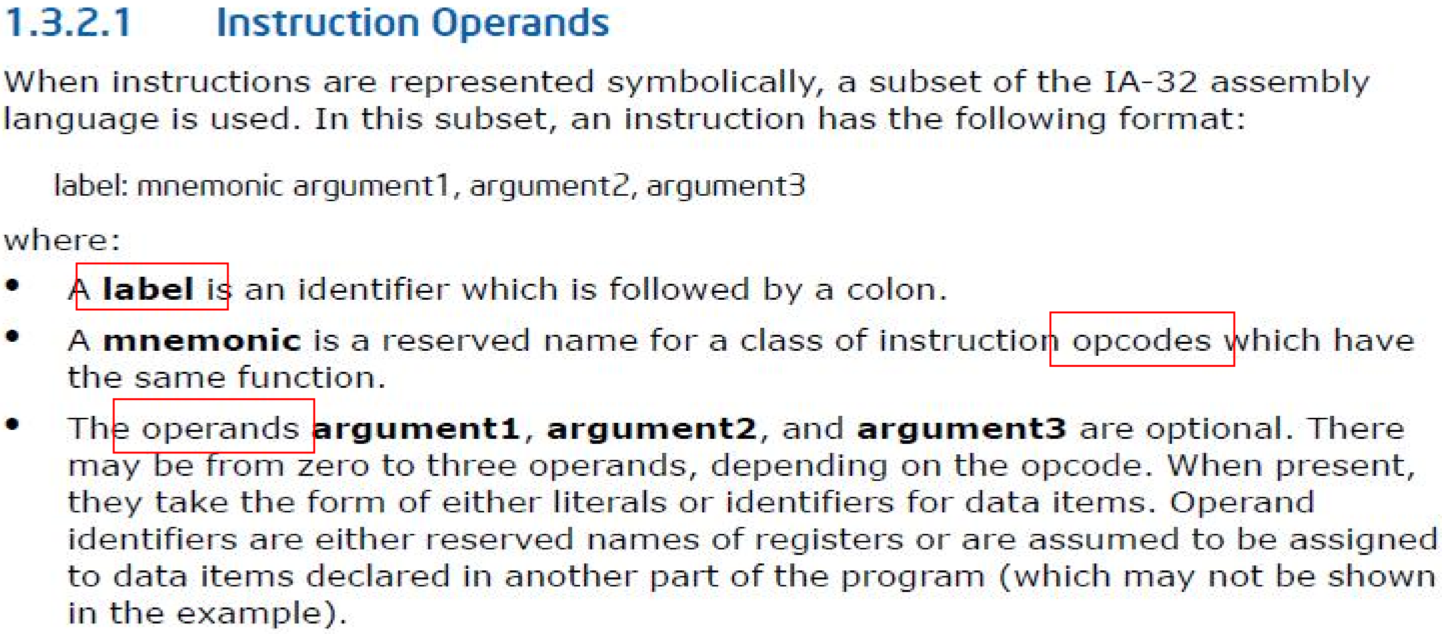
```
here:   movl    0x8049388, %eax
        addl    0x8049384, %eax
        movl    %eax, 0x804946c
```



| Hex digit | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Decimal value | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Binary value | 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 |
| Hex digit | 8 | 9 | A | B | C | D | E | F |
| Decimal value | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| Binary value | 1000 | 1001 | 1010 | 1011 | 1100 | 1101 | 1110 | 1111 |

Figure 2.2 **Hexadecimal notation.** Each Hex digit encodes one of 16 values.

**(Source: CSAPP)**



### 1.3.2.1 Instruction Operands

When instructions are represented symbolically, a subset of the IA-32 assembly language is used. In this subset, an instruction has the following format:

label: mnemonic argument1, argument2, argument3

where:

- A **label** is an identifier which is followed by a colon.
- A **mnemonic** is a reserved name for a class of instruction opcodes which have the same function.
- The operands **argument1**, **argument2**, and **argument3** are optional. There may be from zero to three operands, depending on the opcode. When present, they take the form of either literals or identifiers for data items. Operand identifiers are either reserved names of registers or are assumed to be assigned to data items declared in another part of the program (which may not be shown in the example).

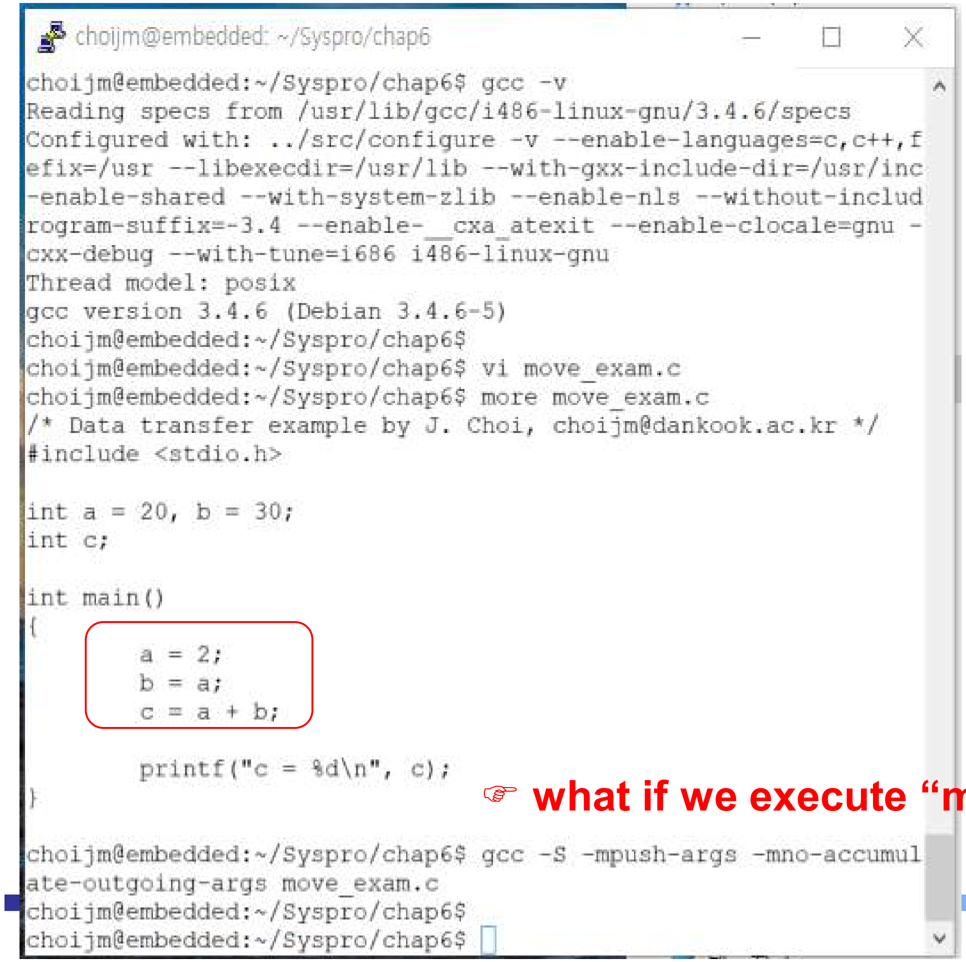
**(Source: Intel SW Developer's Manual, Volume 1: Basic Architecture)**

# Instruction Model (2/2)

- Opcode summary
  - General Purpose
    - Data Transfer Instruction: MOV, CMOVNZ, XCHG, PUSH, POP
    - Arithmetic Instruction: ADD, SUB, MUL, DIV, DEC, INC, CMP
    - Logical Instruction: AND, OR, XOR, NOT
    - Shift and Rotate Instruction: SHR, SHL, SAR, SAL, ROR, ROL
    - Bit and Byte Instruction: BT, BTS, BTC
    - Control Transfer Instruction: JMP, JE, JZ, JNE, LOOP
    - Function related Instruction: CALL, RET, LEAVE
    - String Instruction: MOVS, CMPS, LODS
    - Flag Control Instruction: STC, CLC, STD, CLD, STI, CLI
    - Segment Register Instruction: LDS, LES
    - Miscellaneous: INT, NOP, CPUID
  - Special Purpose
    - FPU Instruction: FLD, FST, FADD, FSUB, FCOM
    - SIMD Instruction (MMX) : MOVD, MOVQ, PADD, PSUB
    - SSE Instruction: MOVSS, ADDSS
    - System Instruction: LGDT, SGDT, LIDT, …

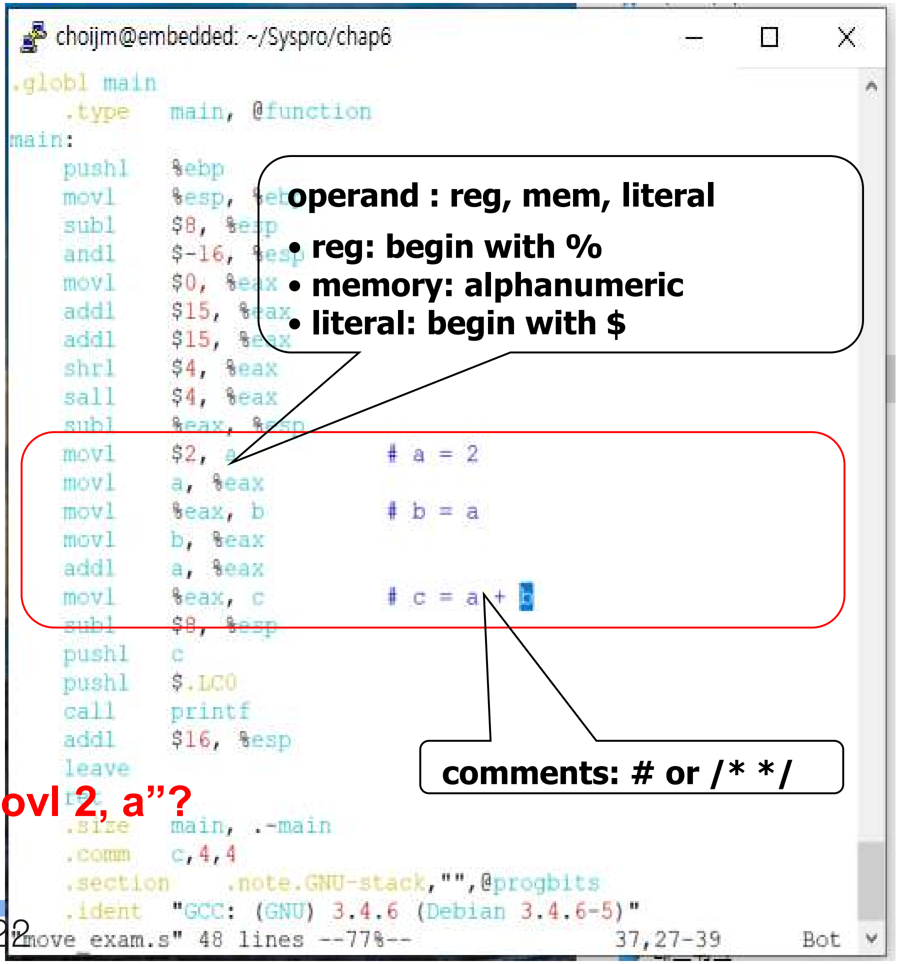
# Instruction Detail: Component (1/11)

- Data Transfer Instruction
  - ✓ Edit move_exam.c and create assembly program using gcc –S
    - Using gcc version 3.4.6 (Since the obfuscation techniques employed in higher gcc version make learning rather complex)



choijm@embedded: ~/Syspro/chap6

```
choijm@embedded:~/Syspro/chap6$ gcc -v
Reading specs from /usr/lib/gcc/i486-linux-gnu/3.4.6/specs
Configured with: ../src/configure -v --enable-languages=c,c++,f
efix=/usr --libexecdir=/usr/lib --with-gxx-include-dir=/usr/inc
-enable-shared --with-system-zlib --enable-nls --without-includ
rogram-suffix=-3.4 --enable-__cxa_atexit --enable-clocale=gnu -
cxx-debug --with-tune=i686 i486-linux-gnu
Thread model: posix
gcc version 3.4.6 (Debian 3.4.6-5)
choijm@embedded:~/Syspro/chap6$
choijm@embedded:~/Syspro/chap6$ vi move_exam.c
choijm@embedded:~/Syspro/chap6$ more move_exam.c
/* Data transfer example by J. Choi, choijm@dankook.ac.kr */
#include <stdio.h>

int a = 20, b = 30;
int c;

int main()
{
        a = 2;
        b = a;
        c = a + b;

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

choijm@embedded:~/Syspro/chap6$ gcc -S -mpush-args -mno-accumul
ate-outgoing-args move_exam.c
choijm@embedded:~/Syspro/chap6$
choijm@embedded:~/Syspro/chap6$
```



choijm@embedded: ~/Syspro/chap6

```
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        movl    $0, %eax
        addl    $15, %eax
        addl    $15, %eax
        shrl    $4, %eax
        sall    $4, %eax
        subl    %eax, %esp
        movl    $2, a                   # a = 2
        movl    a, %eax
        movl    %eax, b                 # b = a
        movl    b, %eax
        addl    a, %eax
        movl    %eax, c                 # c = a + b
        subl    $8, %esp
        pushl   c
        pushl   $.LC0
        call    printf
        addl    $16, %esp
        leave
        ret
        .size   main, .-main
        .comm   c,4,4
        .section        .note.GNU-stack,"",@progbits
        .ident  "GCC: (GNU) 3.4.6 (Debian 3.4.6-5)"
"move_exam.s" 48 lines --77%--                          37,27-39        Bot
```

**operand : reg, mem, literal**

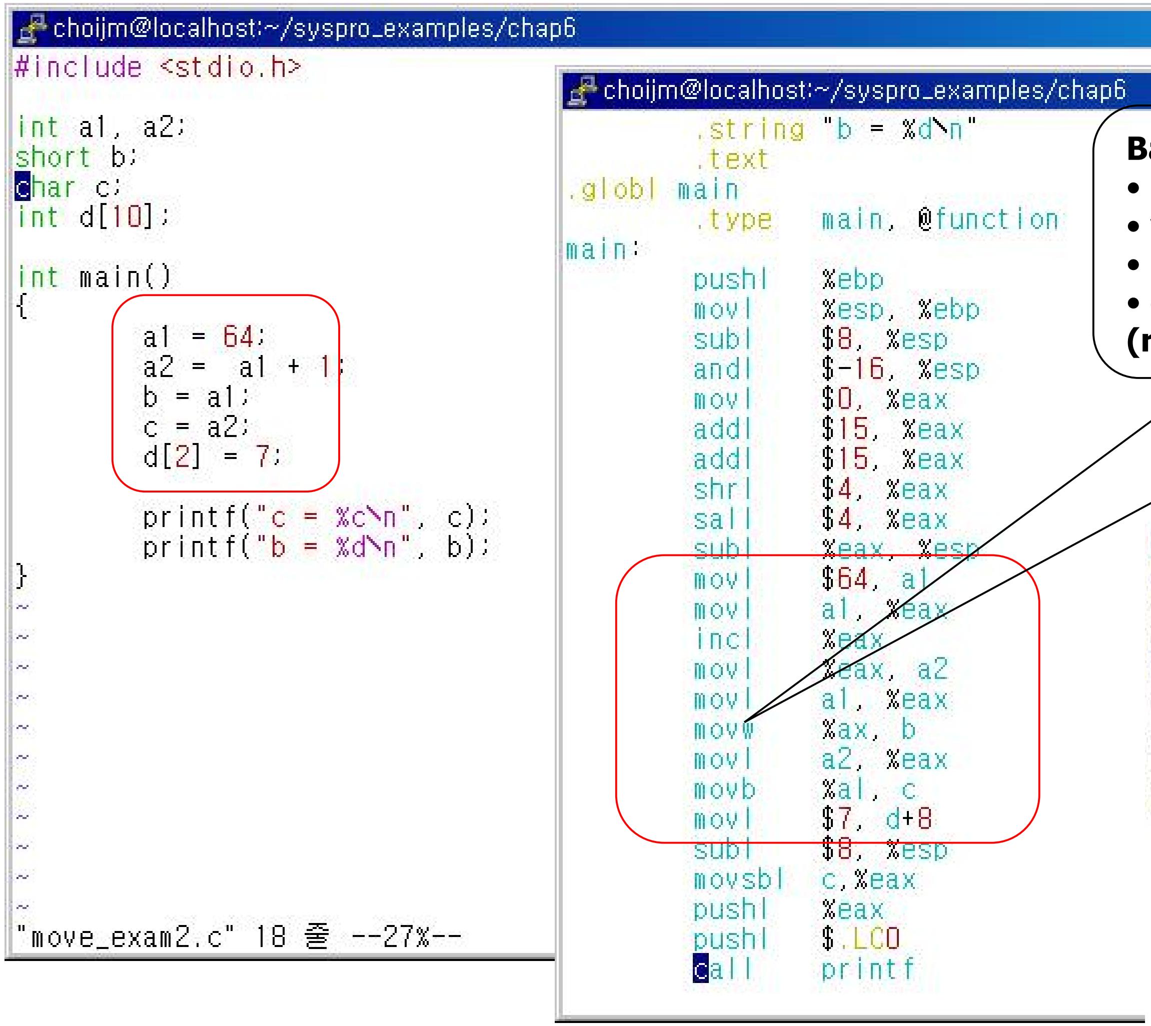
- **reg: begin with %**
- **memory: alphanumeric**
- **literal: begin with $**

**comments: # or /* */**

☞ **what if we execute "movl 2, a"?**

# Instruction Detail: Component (2/11)

- Data Transfer Instruction (cont')



choijm@localhost:~/syspro_examples/chap6

```
#include <stdio.h>

int a1, a2;
short b;
char c;
int d[10];

int main()
{
        a1 = 64;
        a2 =  a1 + 1;
        b = a1;
        c = a2;
        d[2] = 7;

        printf("c = %c\n", c);
        printf("b = %d\n", b);
}
~
~
~
~
~
~
~
~
~
~
~
"move_exam2.c" 18 줄 --27%--
```

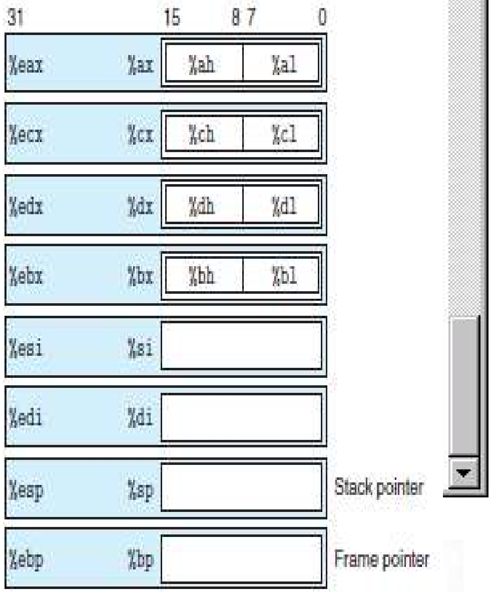
choijm@localhost:~/syspro_examples/chap6

```
        .string "b = %d\n"
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        movl    $0, %eax
        addl    $15, %eax
        addl    $15, %eax
        shrl    $4, %eax
        sall    $4, %eax
        subl    %eax, %esp
        movl    $64, a1
        movl    a1, %eax
        incl    %eax
        movl    %eax, a2
        movl    a1, %eax
        movw    %ax, b
        movl    a2, %eax
        movb    %al, c
        movl    $7, d+8
        subl    $8, %esp
        movsbl  c,%eax
        pushl   %eax
        pushl   $.LC0
        call    printf
```

**Basic opcode(mov) + suffix [l|w|b|q]**
- **b: byte (1 byte)**
- **w: word (2 bytes)**
- **l: long (double) word (4 bytes)**
- **q: quad word (8 byte)**

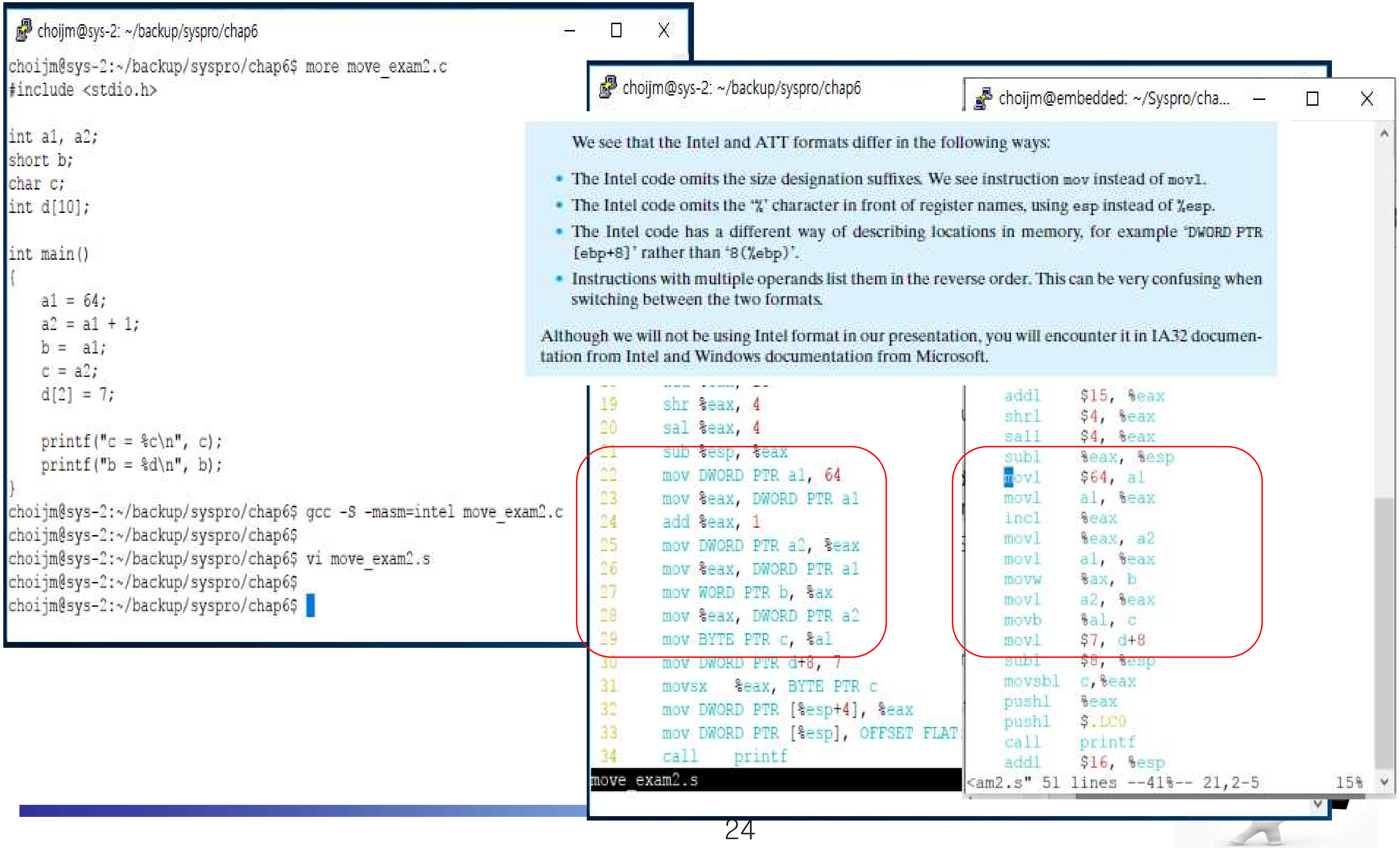
**(refer to Figure 3.1 in CSAPP)**

Figure 3.2
**IA32 integer registers.** All eight registers can be accessed as either 16 bits (word) or 32 bits (double word). The 2 low-order bytes of the first four registers can be accessed independently.



# Instruction Detail: Component (3/11)

- AT&T vs. Intel (cf. Microsoft ASM)



choijm@sys-2: ~/backup/syspro/chap6

```
choijm@sys-2:~/backup/syspro/chap6$ more move_exam2.c
#include <stdio.h>

int a1, a2;
short b;
char c;
int d[10];

int main()
{
    a1 = 64;
    a2 = a1 + 1;
    b =  a1;
    c = a2;
    d[2] = 7;

    printf("c = %c\n", c);
    printf("b = %d\n", b);
}
choijm@sys-2:~/backup/syspro/chap6$ gcc -S -masm=intel move_exam2.c
choijm@sys-2:~/backup/syspro/chap6$
choijm@sys-2:~/backup/syspro/chap6$ vi move_exam2.s
choijm@sys-2:~/backup/syspro/chap6$
choijm@sys-2:~/backup/syspro/chap6$
```

choijm@sys-2: ~/backup/syspro/chap6

We see that the Intel and ATT formats differ in the following ways:

- The Intel code omits the size designation suffixes. We see instruction `mov` instead of `movl`.
- The Intel code omits the '%' character in front of register names, using `esp` instead of `%esp`.
- The Intel code has a different way of describing locations in memory, for example '`DWORD PTR [ebp+8]`' rather than '`8(%ebp)`'.
- Instructions with multiple operands list them in the reverse order. This can be very confusing when switching between the two formats.

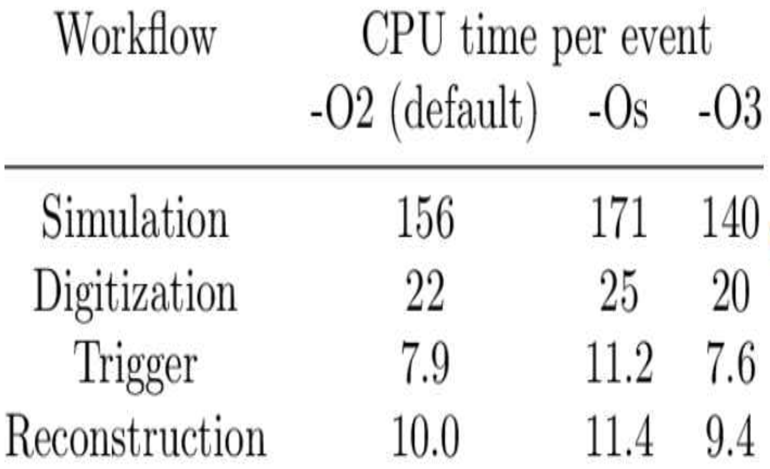
Although we will not be using Intel format in our presentation, you will encounter it in IA32 documentation from Intel and Windows documentation from Microsoft.

```
    shr %eax, 4
    sal %eax, 4
    sub %esp, %eax
    mov DWORD PTR a1, 64
    mov %eax, DWORD PTR a1
    add %eax, 1
    mov DWORD PTR a2, %eax
    mov %eax, DWORD PTR a1
    mov WORD PTR b, %ax
    mov %eax, DWORD PTR a2
    mov BYTE PTR c, %al
    mov DWORD PTR d+8, 7
    movsx   %eax, BYTE PTR c
    mov DWORD PTR [%esp+4], %eax
    mov DWORD PTR [%esp], OFFSET FLAT:
    call    printf
move_exam2.s
```

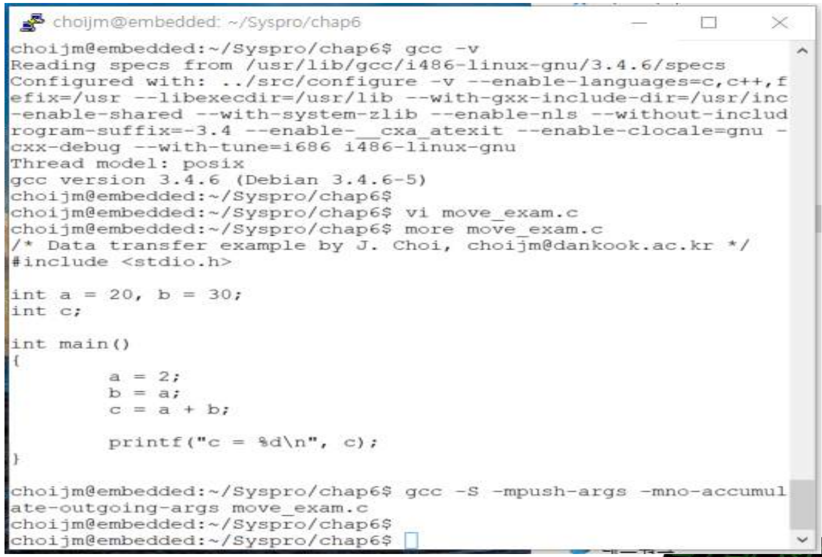
```
    addl    $15, %eax
    shrl    $4, %eax
    sall    $4, %eax
    subl    %eax, %esp
    movl    $64, a1
    movl    a1, %eax
    incl    %eax
    movl    %eax, a2
    movl    a1, %eax
    movw    %ax, b
    movl    a2, %eax
    movb    %al, c
    movl    $7, d+8
    subl    $8, %esp
    movsbl  c,%eax
    pushl   %eax
    pushl   $.LC0
    call    printf
    addl    $16, %esp
<am2.s" 51 lines --41%-- 21,2-5        15%
```

# Quiz for 9th-Week 2nd-Lesson

- Quiz
  - ✓ 1. Explain the three components of an IA instruction format.
  - ✓ 2. There are various optimization options in gcc such as “O0, O2, O3 and Os”. What if we create an assembly program using O3 when we create the move_exam.s in slide 22?
  - ✓ Bonus. What if we create an assembly program using O3 when we declare the a, b, c as local variables?
  - ✓ Due: until 6 PM Friday of this week (30th, October)



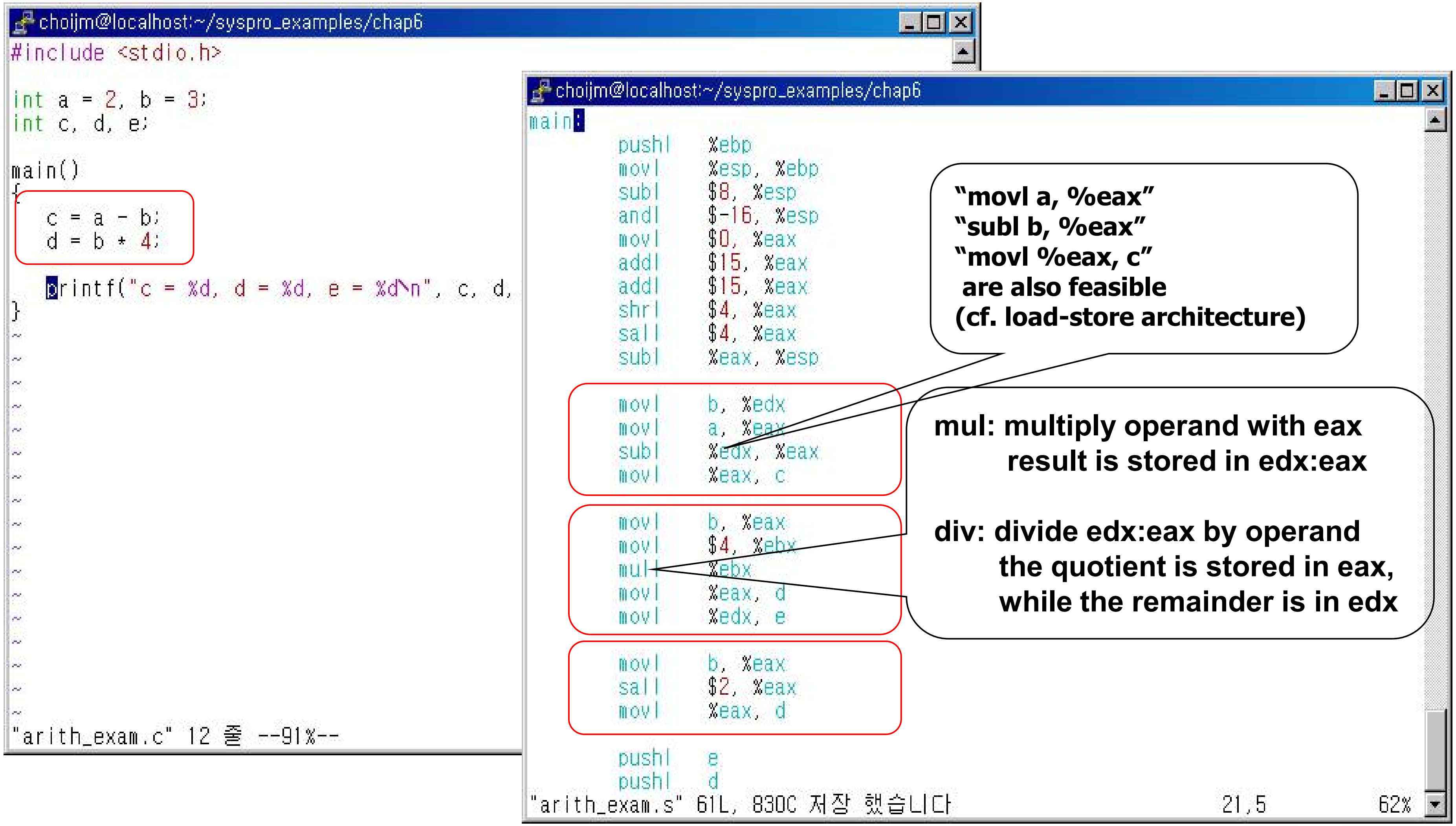
| Workflow | CPU time per event | | |
|---|---|---|---|
| | -O2 (default) | -Os | -O3 |
| Simulation | 156 | 171 | 140 |
| Digitization | 22 | 25 | 20 |
| Trigger | 7.9 | 11.2 | 7.6 |
| Reconstruction | 10.0 | 11.4 | 9.4 |



```
choijm@embedded:~/Syspro/chap6$ gcc -v
Reading specs from /usr/lib/gcc/i486-linux-gnu/3.4.6/specs
Configured with: ../src/configure -v --enable-languages=c,c++,f
efix=/usr --libexecdir=/usr/lib --with-gxx-include-dir=/usr/inc
-enable-shared --with-system-zlib --enable-nls --without-includ
rogram-suffix=-3.4 --enable-__cxa_atexit --enable-clocale=gnu -
cxx-debug --with-tune=i686 i486-linux-gnu
Thread model: posix
gcc version 3.4.6 (Debian 3.4.6-5)
choijm@embedded:~/Syspro/chap6$
choijm@embedded:~/Syspro/chap6$ vi move_exam.c
choijm@embedded:~/Syspro/chap6$ more move_exam.c
/* Data transfer example by J. Choi, choijm@dankook.ac.kr */
#include <stdio.h>

int a = 20, b = 30;
int c;

int main()
{
        a = 2;
        b = a;
        c = a + b;

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

choijm@embedded:~/Syspro/chap6$ gcc -S -mpush-args -mno-accumul
ate-outgoing-args move_exam.c
choijm@embedded:~/Syspro/chap6$
choijm@embedded:~/Syspro/chap6$
```

# Instruction Detail: Component (4/11)

- Arithmetic Instruction



choijm@localhost:~/syspro_examples/chap6

```
#include <stdio.h>

int a = 2, b = 3;
int c, d, e;

main()
{
    c = a - b;
    d = b * 4;

    printf("c = %d, d = %d, e = %d\n", c, d,
}
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"arith_exam.c" 12 줄 --91%--
```

choijm@localhost:~/syspro_examples/chap6

```
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        movl    $0, %eax
        addl    $15, %eax
        addl    $15, %eax
        shrl    $4, %eax
        sall    $4, %eax
        subl    %eax, %esp

        movl    b, %edx
        movl    a, %eax
        subl    %edx, %eax
        movl    %eax, c

        movl    b, %eax
        movl    $4, %ebx
        mull    %ebx
        movl    %eax, d
        movl    %edx, e

        movl    b, %eax
        sall    $2, %eax
        movl    %eax, d

        pushl   e
        pushl   d
"arith_exam.s" 61L, 830C 저장 했습니다                    21,5          62%
```

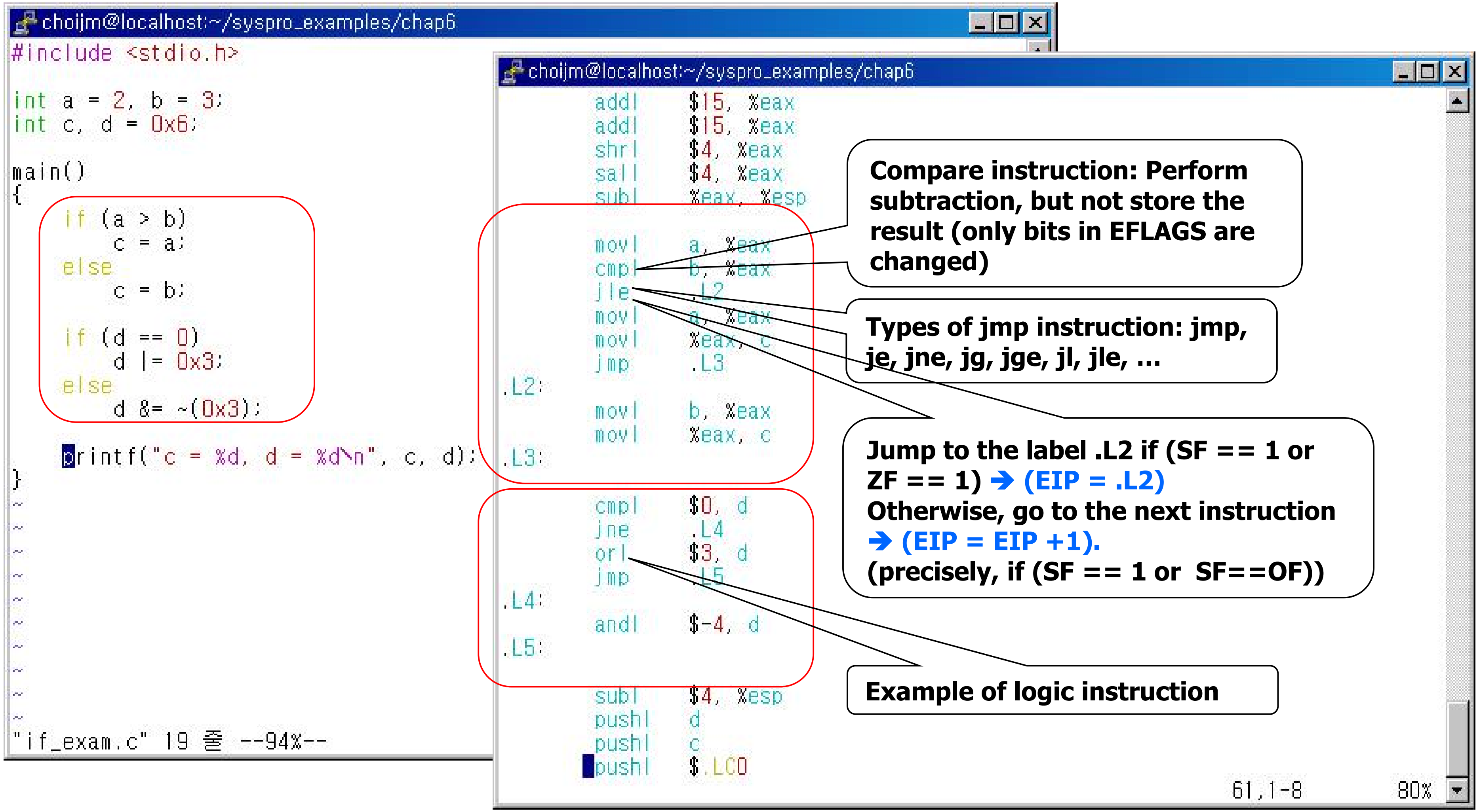
**"movl a, %eax"**
**"subl b, %eax"**
**"movl %eax, c"**
**are also feasible**
**(cf. load-store architecture)**

**mul: multiply operand with eax**
**result is stored in edx:eax**

**div: divide edx:eax by operand**
**the quotient is stored in eax,**
**while the remainder is in edx**

# Instruction Detail: Component (5/11)

- Control Transfer Instruction: if



choijm@localhost:~/syspro_examples/chap6

```
#include <stdio.h>

int a = 2, b = 3;
int c, d = 0x6;

main()
{
    if (a > b)
        c = a;
    else
        c = b;

    if (d == 0)
        d |= 0x3;
    else
        d &= ~(0x3);

    printf("c = %d, d = %d\n", c, d);
}
~
~
~
~
~
~
~
~
~
~
"if_exam.c" 19 줄 --94%--
```

choijm@localhost:~/syspro_examples/chap6

```
        addl    $15, %eax
        addl    $15, %eax
        shrl    $4, %eax
        sall    $4, %eax
        subl    %eax, %esp

        movl    a, %eax
        cmpl    b, %eax
        jle     .L2
        movl    a, %eax
        movl    %eax, c
        jmp     .L3
.L2:
        movl    b, %eax
        movl    %eax, c
.L3:
        cmpl    $0, d
        jne     .L4
        orl     $3, d
        jmp     .L5
.L4:
        andl    $-4, d
.L5:
        subl    $4, %esp
        pushl   d
        pushl   c
        pushl   $.LC0
                                        61,1-8          80%
```

**Compare instruction: Perform subtraction, but not store the result (only bits in EFLAGS are changed)**

**Types of jmp instruction: jmp, je, jne, jg, jge, jl, jle, …**

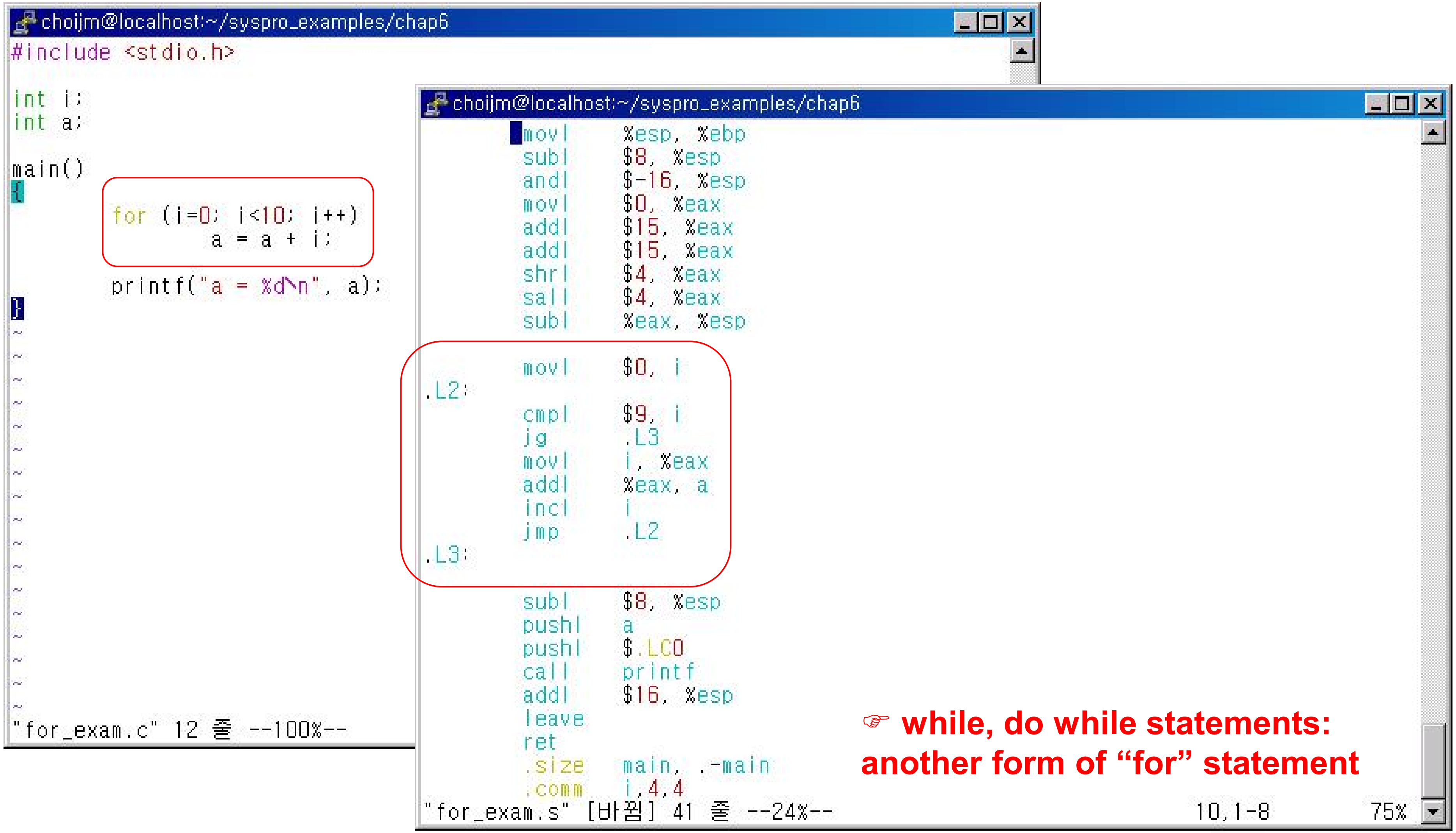
**Jump to the label .L2 if (SF == 1 or ZF == 1) ➔ (EIP = .L2)**
**Otherwise, go to the next instruction ➔ (EIP = EIP +1).**
**(precisely, if (SF == 1 or SF==OF))**

**Example of logic instruction**

☞ **switch statement: extension of "if else" statement**

# Instruction Detail: Component (6/11)

- Control Transfer Instruction: for



choijm@localhost:~/syspro_examples/chap6

```
#include <stdio.h>

int i;
int a;

main()
{
        for (i=0; i<10; i++)
                a = a + i;

        printf("a = %d\n", a);
}
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"for_exam.c" 12 줄 --100%--
```

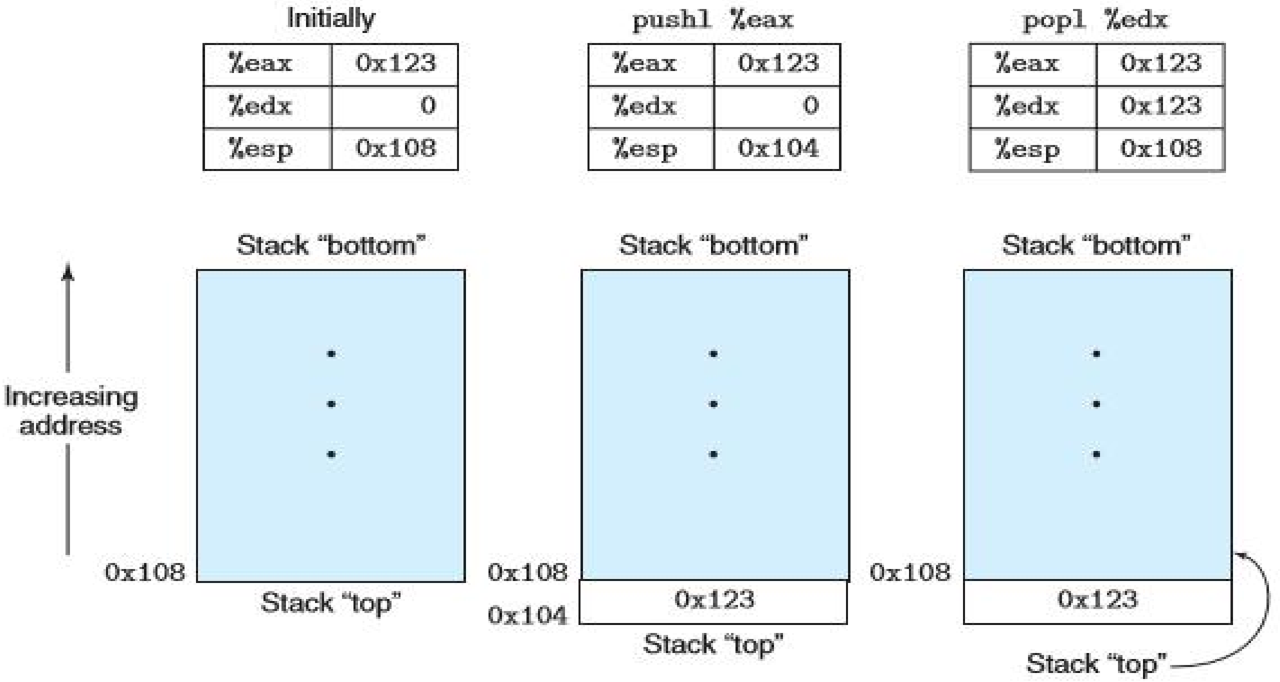
choijm@localhost:~/syspro_examples/chap6

```
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        movl    $0, %eax
        addl    $15, %eax
        addl    $15, %eax
        shrl    $4, %eax
        sall    $4, %eax
        subl    %eax, %esp

        movl    $0, i
.L2:
        cmpl    $9, i
        jg      .L3
        movl    i, %eax
        addl    %eax, a
        incl    i
        jmp     .L2
.L3:

        subl    $8, %esp
        pushl   a
        pushl   $.LC0
        call    printf
        addl    $16, %esp
        leave
        ret
        .size   main, .-main
        .comm   i,4,4
"for_exam.s" [바뀜] 41 줄 --24%--                  10,1-8          75%
```

☞ **while, do while statements: another form of “for” statement**

# Instruction Detail: Component (7/11)

- Function-related Instruction: stack revisit
  - ✓ Stack operation: push and pop
  - ✓ Stack management: bottom and top (SS and esp)



Initially

| | |
|---|---|
| %eax | 0x123 |
| %edx | 0 |
| %esp | 0x108 |

pushl %eax

| | |
|---|---|
| %eax | 0x123 |
| %edx | 0 |
| %esp | 0x104 |

popl %edx

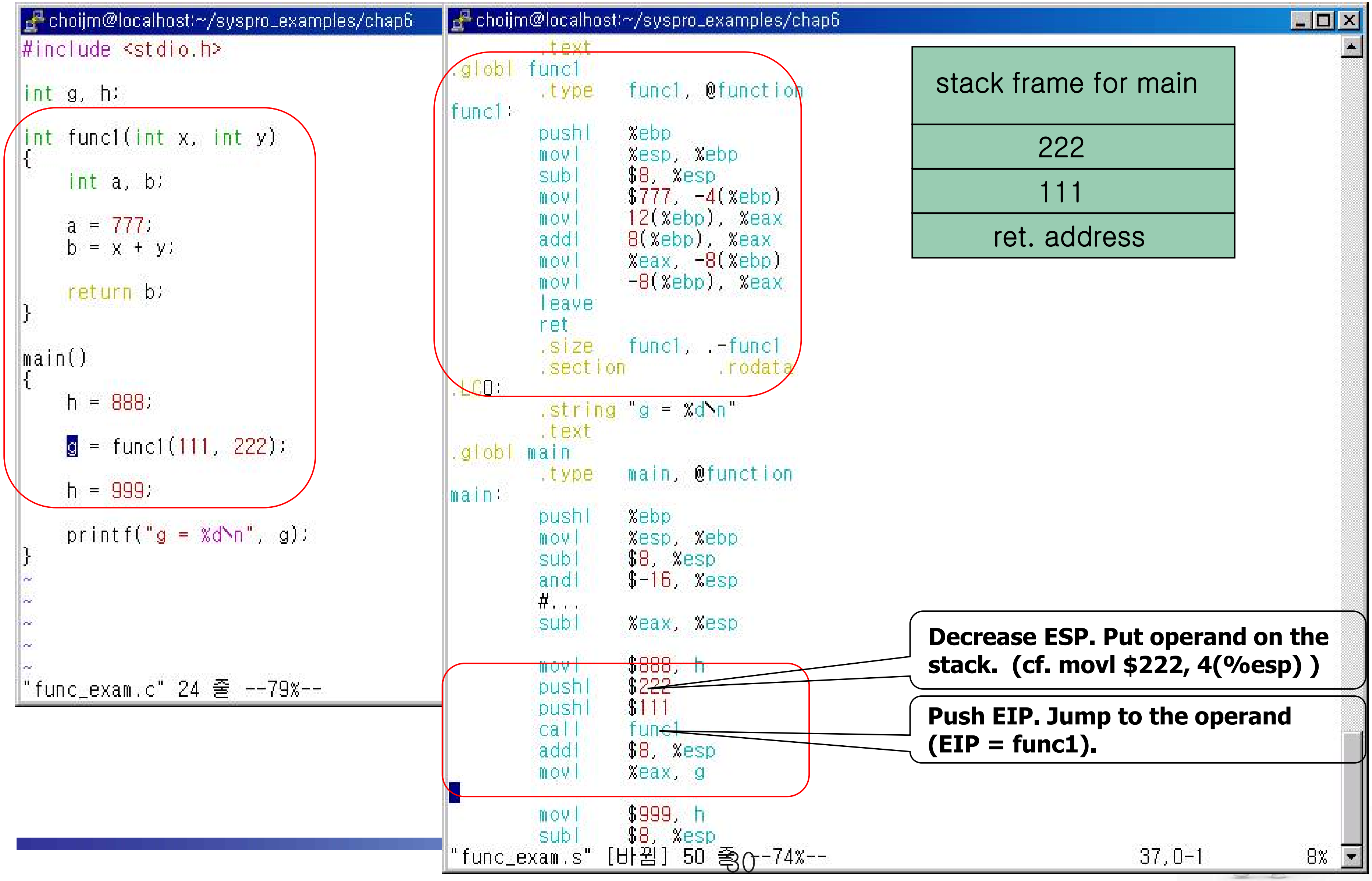
| | |
|---|---|
| %eax | 0x123 |
| %edx | 0x123 |
| %esp | 0x108 |

Figure 3.5 **Illustration of stack operation.** By convention, we draw stacks upside down, so that the "top" of the stack is shown at the bottom. IA32 stacks grow toward lower addresses, so pushing involves decrementing the stack pointer (register %esp) and storing to memory, while popping involves reading from memory and incrementing the stack pointer.

**(Source: CSAPP)**

# Instruction Detail: Component (8/11)

- Function-related Instruction: before function call



choijm@localhost:~/syspro_examples/chap6

```
#include <stdio.h>

int g, h;

int func1(int x, int y)
{
    int a, b;

    a = 777;
    b = x + y;

    return b;
}

main()
{
    h = 888;

    g = func1(111, 222);

    h = 999;

    printf("g = %d\n", g);
}
~
~
~
~
~
"func_exam.c" 24 줄 --79%--
```

choijm@localhost:~/syspro_examples/chap6

```
        .text
.globl func1
        .type   func1, @function
func1:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        movl    $777, -4(%ebp)
        movl    12(%ebp), %eax
        addl    8(%ebp), %eax
        movl    %eax, -8(%ebp)
        movl    -8(%ebp), %eax
        leave
        ret
        .size   func1, .-func1
        .section        .rodata
.LC0:
        .string "g = %d\n"
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        #...
        subl    %eax, %esp

        movl    $888, h
        pushl   $222
        pushl   $111
        call    func1
        addl    $8, %esp
        movl    %eax, g

        movl    $999, h
        subl    $8, %esp
"func_exam.s" [바뀜] 50 줄 --74%--                    37,0-1        8%
```

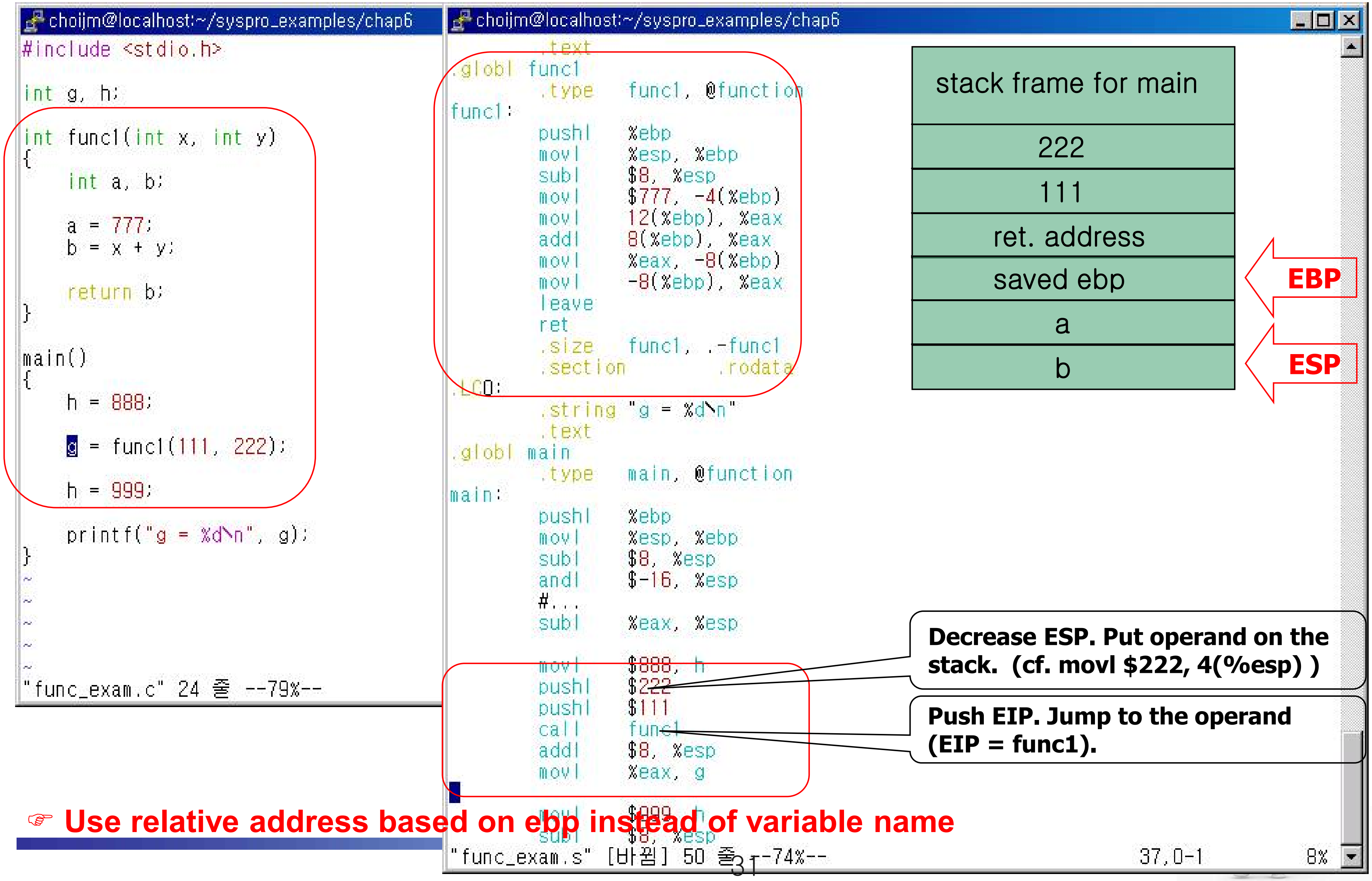
| stack frame for main |
|---|
| 222 |
| 111 |
| ret. address |

**Decrease ESP. Put operand on the stack. (cf. movl $222, 4(%esp) )**

**Push EIP. Jump to the operand (EIP = func1).**

# Instruction Detail: Component (9/11)

- Function-related Instruction: in function



choijm@localhost:~/syspro_examples/chap6

```
#include <stdio.h>

int g, h;

int func1(int x, int y)
{
    int a, b;

    a = 777;
    b = x + y;

    return b;
}

main()
{
    h = 888;

    g = func1(111, 222);

    h = 999;

    printf("g = %d\n", g);
}
~
~
~
~
~
"func_exam.c" 24 줄 --79%--
```

choijm@localhost:~/syspro_examples/chap6

```
        .text
.globl func1
        .type   func1, @function
func1:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        movl    $777, -4(%ebp)
        movl    12(%ebp), %eax
        addl    8(%ebp), %eax
        movl    %eax, -8(%ebp)
        movl    -8(%ebp), %eax
        leave
        ret
        .size   func1, .-func1
        .section        .rodata
.LC0:
        .string "g = %d\n"
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        #...
        subl    %eax, %esp

        movl    $888, h
        pushl   $222
        pushl   $111
        call    func1
        addl    $8, %esp
        movl    %eax, g

        movl    $999, h
        subl    $8, %esp
"func_exam.s" [바뀜] 50 줄 --74%--                    37,0-1        8%
```

| |
|---|
| stack frame for main |
| 222 |
| 111 |
| ret. address |
| saved ebp |
| a |
| b |

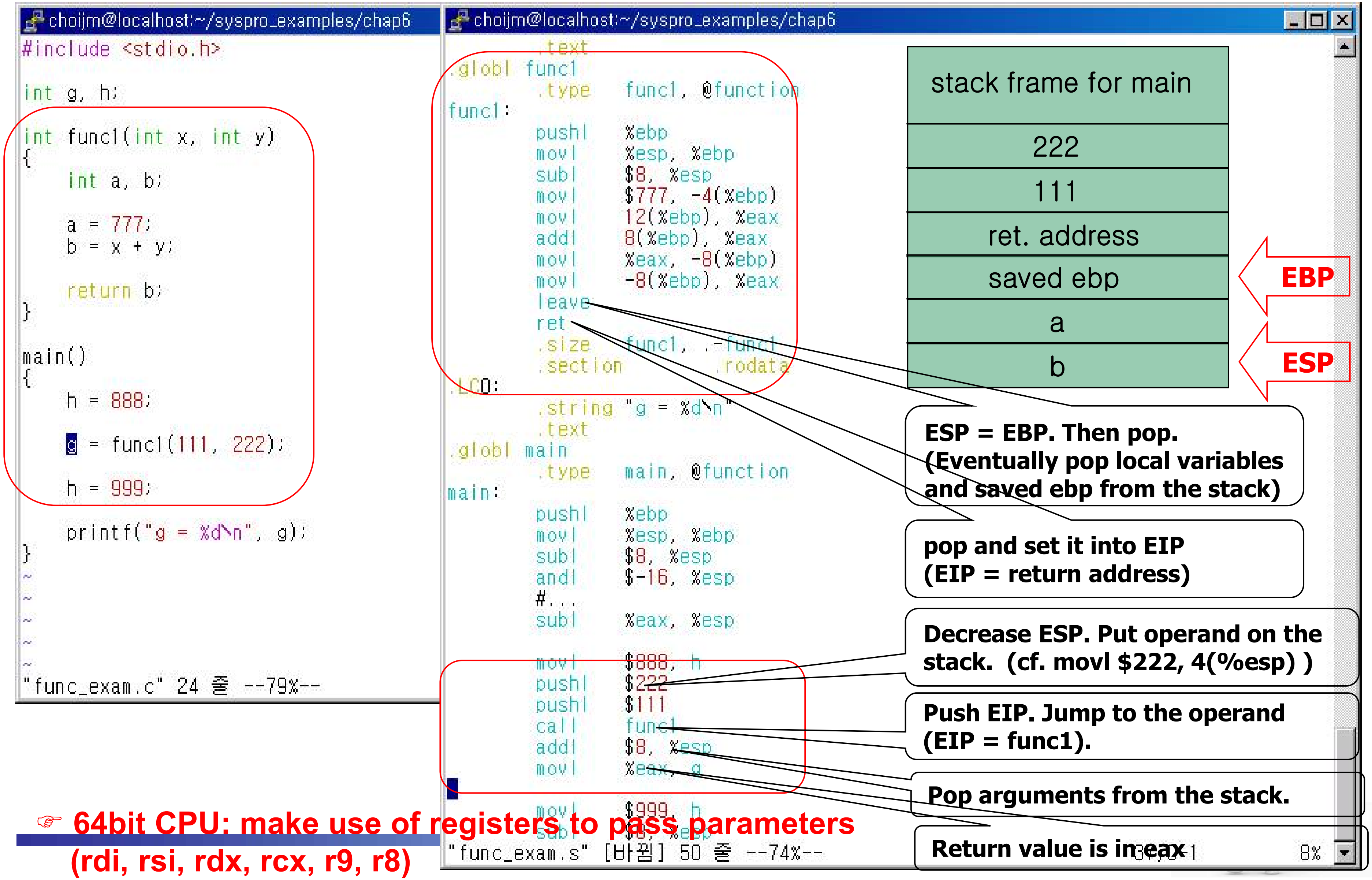
**Decrease ESP. Put operand on the stack. (cf. movl $222, 4(%esp) )**

**Push EIP. Jump to the operand (EIP = func1).**

☞ **Use relative address based on ebp instead of variable name**

# Instruction Detail: Component (10/11)

- Function-related Instruction: after function



choijm@localhost:~/syspro_examples/chap6

```
#include <stdio.h>

int g, h;

int func1(int x, int y)
{
    int a, b;

    a = 777;
    b = x + y;

    return b;
}

main()
{
    h = 888;

    g = func1(111, 222);

    h = 999;

    printf("g = %d\n", g);
}
~
~
~
~
~
"func_exam.c" 24 줄 --79%--
```

choijm@localhost:~/syspro_examples/chap6

```
        .text
.globl func1
        .type   func1, @function
func1:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        movl    $777, -4(%ebp)
        movl    12(%ebp), %eax
        addl    8(%ebp), %eax
        movl    %eax, -8(%ebp)
        movl    -8(%ebp), %eax
        leave
        ret
        .size   func1, .-func1
        .section        .rodata
.LC0:
        .string "g = %d\n"
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        #...
        subl    %eax, %esp

        movl    $888, h
        pushl   $222
        pushl   $111
        call    func1
        addl    $8, %esp
        movl    %eax, g

        movl    $999, h
        subl    $8, %esp
"func_exam.s" [바뀜] 50 줄 --74%--                31,2-1        8%
```

| stack frame for main |
|---|
| 222 |
| 111 |
| ret. address |
| saved ebp |
| a |
| b |

**ESP = EBP. Then pop. (Eventually pop local variables and saved ebp from the stack)**

**pop and set it into EIP (EIP = return address)**

**Decrease ESP. Put operand on the stack. (cf. movl $222, 4(%esp) )**

**Push EIP. Jump to the operand (EIP = func1).**

**Pop arguments from the stack.**

**Return value is in eax**

☞ **64bit CPU: make use of registers to pass parameters (rdi, rsi, rdx, rcx, r9, r8)**

# Instruction Detail: Component (11/11)

- Function-related Instruction: stack frame illustration

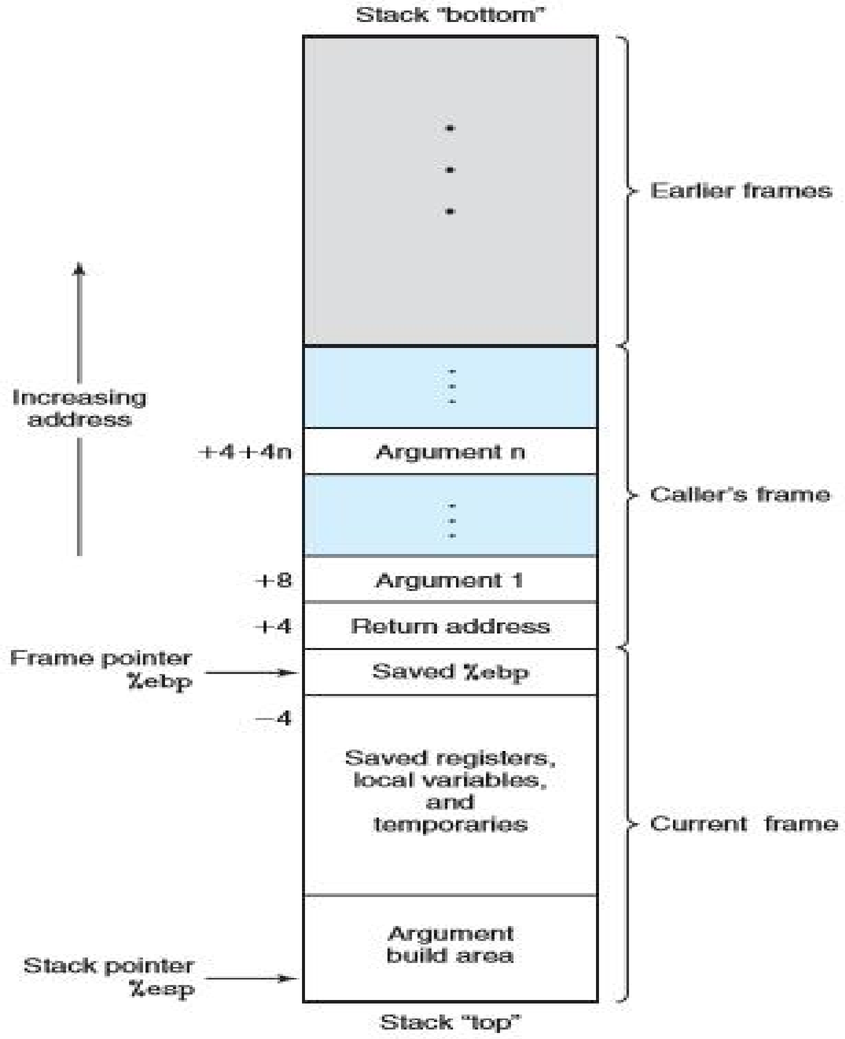


Figure 3.21
**Stack frame structure.** The stack is used for passing arguments, for storing return information, for saving registers, and for local storage.

# Revisit Stack Destroy in LN4

- 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();
}
```

# Quiz for 10th-Week 1st-Lesson

- Quiz
  - ✓ 1. Explain two ways how the C statement “d = b * 7” is translated into assembly language.
  - ✓ 2. Describe how arguments and local variables are accessed in CPU.
  - ✓ Due: until 6 PM Friday of this week (6th, November)

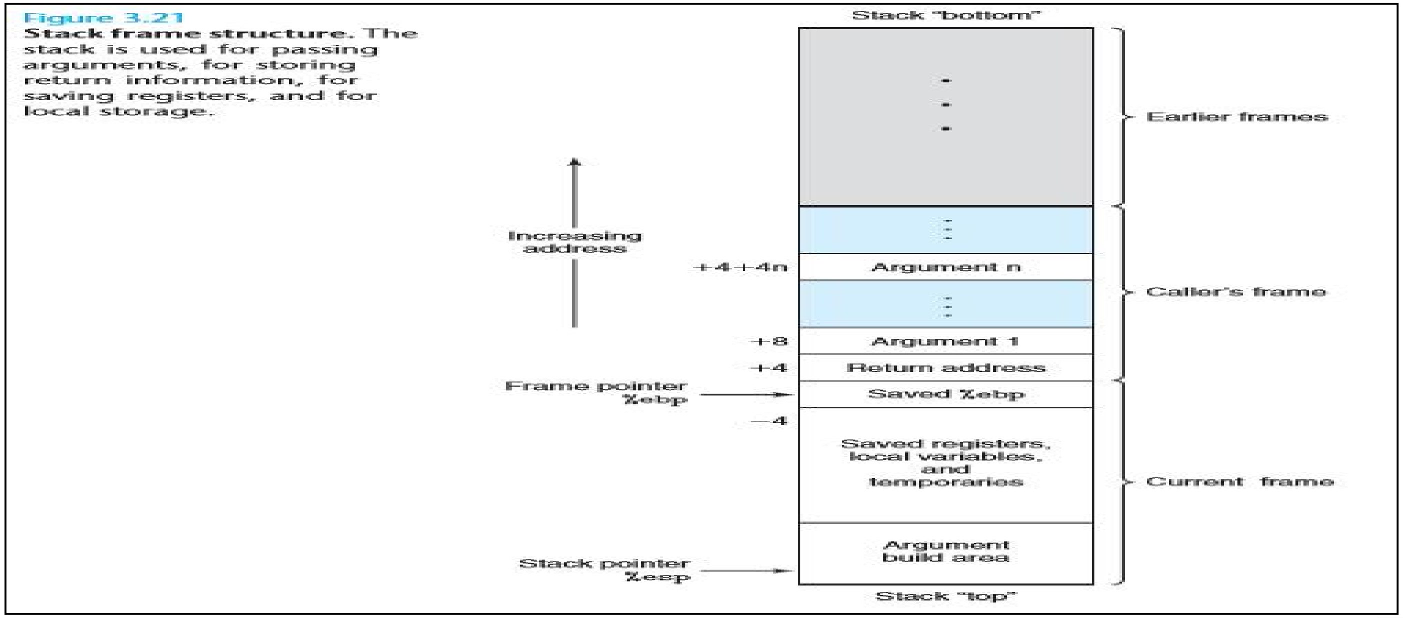
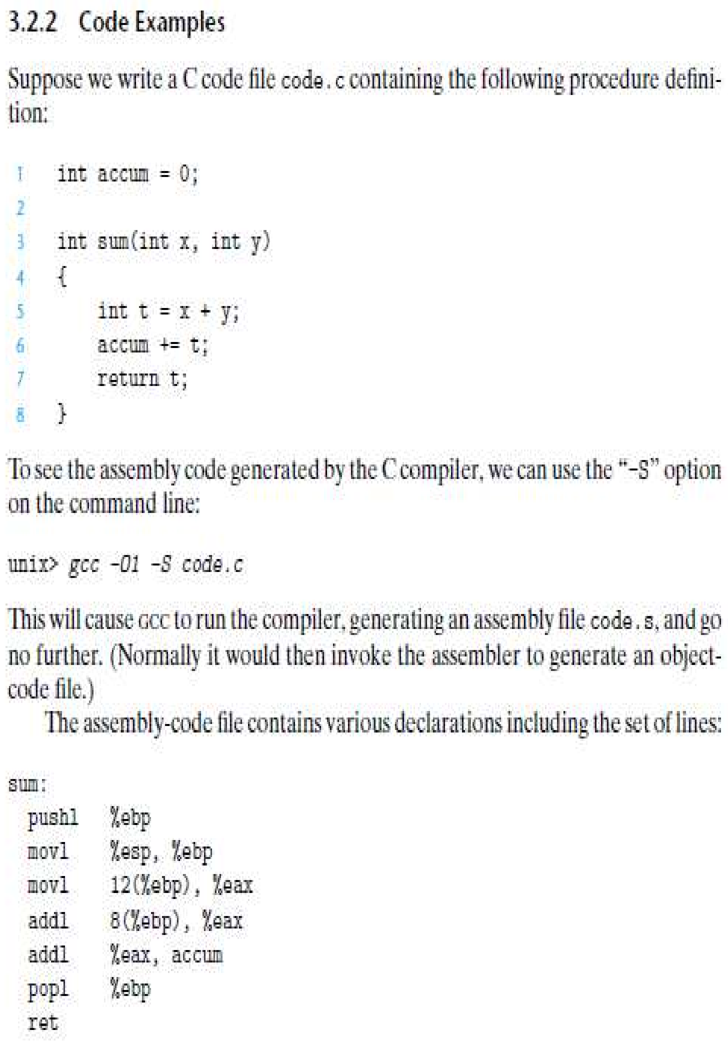


Figure 3.21
Stack frame structure. The stack is used for passing arguments, for storing return information, for saving registers, and for local storage.

# Revisit CSAPP

- Assembly code example from CSAPP



### 3.2.2 Code Examples

Suppose we write a C code file `code.c` containing the following procedure definition:

```c
int accum = 0;

int sum(int x, int y)
{
    int t = x + y;
    accum += t;
    return t;
}
```

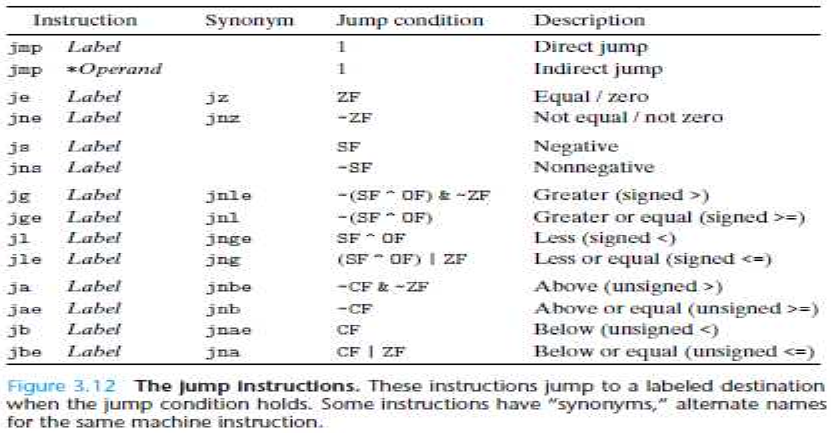
To see the assembly code generated by the C compiler, we can use the "-S" option on the command line:

```
unix> gcc -O1 -S code.c
```

This will cause GCC to run the compiler, generating an assembly file `code.s`, and go no further. (Normally it would then invoke the assembler to generate an object-code file.)

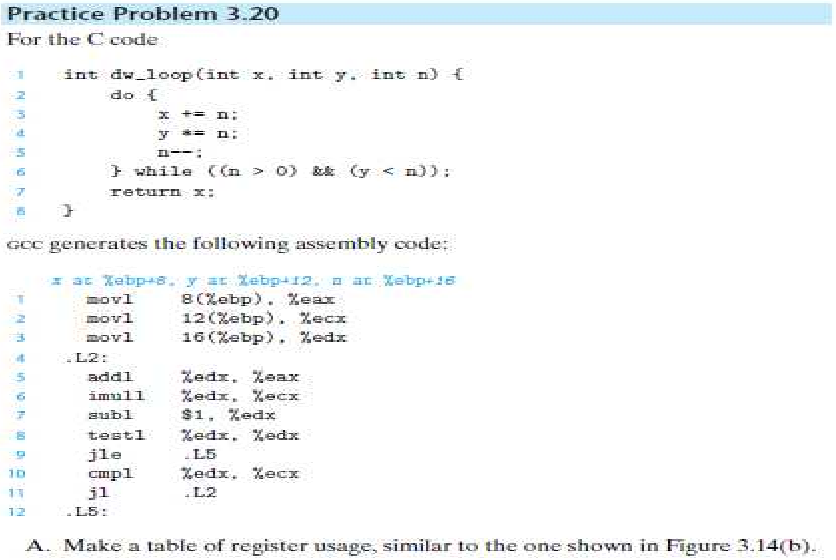
The assembly-code file contains various declarations including the set of lines:

```
sum:
  pushl   %ebp
  movl    %esp, %ebp
  movl    12(%ebp), %eax
  addl    8(%ebp), %eax
  addl    %eax, accum
  popl    %ebp
  ret
```



| Instruction | | Synonym | Jump condition | Description |
|---|---|---|---|---|
| `jmp` | *Label* | | 1 | Direct jump |
| `jmp` | **Operand* | | 1 | Indirect jump |
| `je` | *Label* | `jz` | ZF | Equal / zero |
| `jne` | *Label* | `jnz` | ~ZF | Not equal / not zero |
| `js` | *Label* | | SF | Negative |
| `jns` | *Label* | | ~SF | Nonnegative |
| `jg` | *Label* | `jnle` | ~(SF ^ OF) & ~ZF | Greater (signed >) |
| `jge` | *Label* | `jnl` | ~(SF ^ OF) | Greater or equal (signed >=) |
| `jl` | *Label* | `jnge` | SF ^ OF | Less (signed <) |
| `jle` | *Label* | `jng` | (SF ^ OF) \| ZF | Less or equal (signed <=) |
| `ja` | *Label* | `jnbe` | ~CF & ~ZF | Above (unsigned >) |
| `jae` | *Label* | `jnb` | ~CF | Above or equal (unsigned >=) |
| `jb` | *Label* | `jnae` | CF | Below (unsigned <) |
| `jbe` | *Label* | `jna` | CF \| ZF | Below or equal (unsigned <=) |

Figure 3.12 **The jump instructions.** These instructions jump to a labeled destination when the jump condition holds. Some instructions have "synonyms," alternate names for the same machine instruction.



**Practice Problem 3.20**

For the C code

```c
int dw_loop(int x, int y, int n) {
    do {
        x += n;
        y *= n;
        n--;
    } while ((n > 0) && (y < n));
    return x;
}
```

GCC generates the following assembly code:

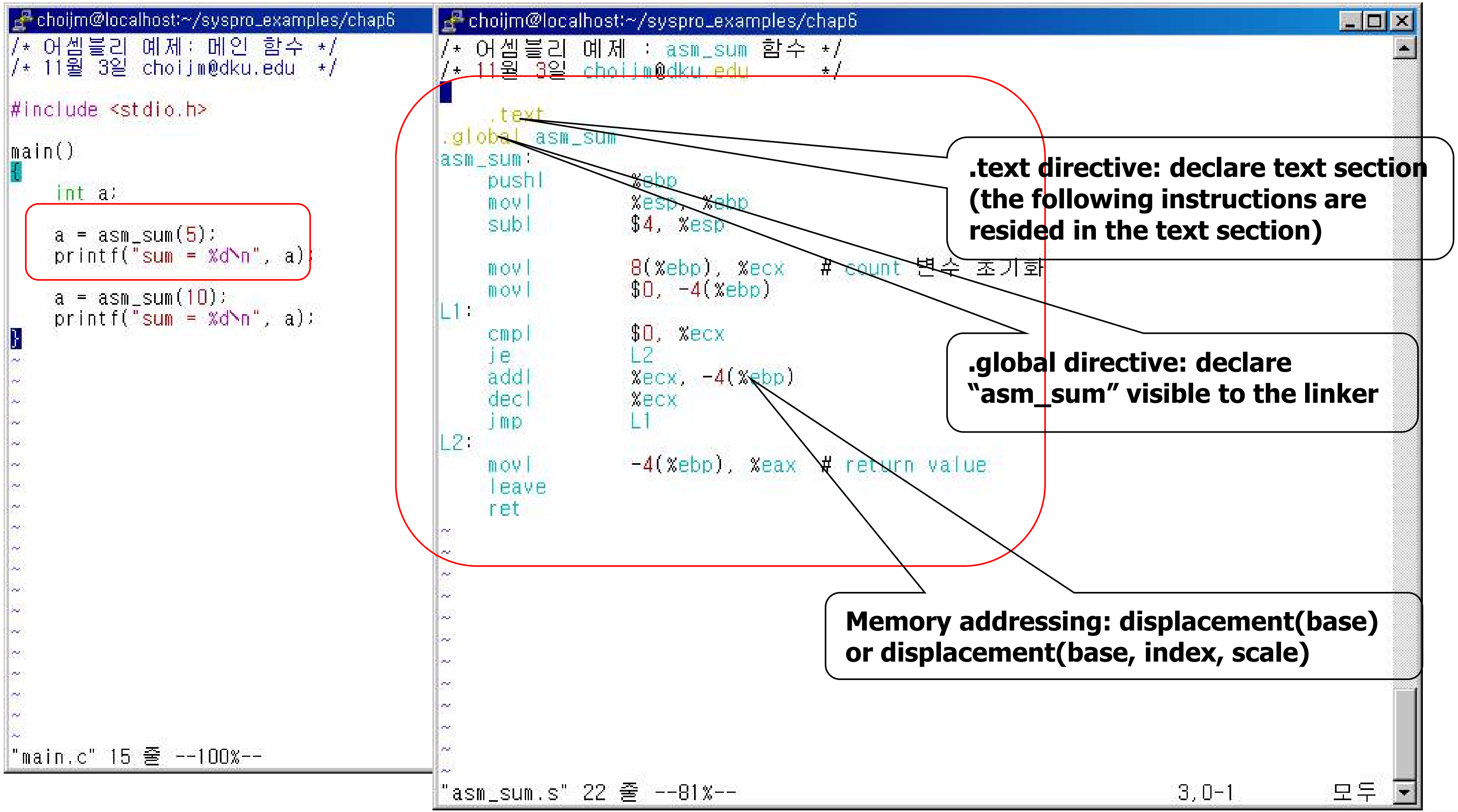
```
    x at %ebp+8, y at %ebp+12, n at %ebp+16
      movl    8(%ebp), %eax
      movl    12(%ebp), %ecx
      movl    16(%ebp), %edx
    .L2:
      addl    %edx, %eax
      imull   %edx, %ecx
      subl    $1, %edx
      testl   %edx, %edx
      jle     .L5
      cmpl    %edx, %ecx
      jl      .L2
    .L5:
```

A. Make a table of register usage, similar to the one shown in Figure 3.14(b).

☞ **See Chapter 3 in CSAPP for more examples**

# Instruction Detail: Make a Program (1/6)

- Practice1: function example
  - ✓ result = asm_sum(final_number), written by assembly language



choijm@localhost:~/syspro_examples/chap6

```
/* 어셈블리 예제: 메인 함수 */
/* 11월 3일 choijm@dku.edu  */

#include <stdio.h>

main()
{
        int a;

        a = asm_sum(5);
        printf("sum = %d\n", a);

        a = asm_sum(10);
        printf("sum = %d\n", a);
}
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"main.c" 15 줄 --100%--
```

choijm@localhost:~/syspro_examples/chap6

```
/* 어셈블리 예제 : asm_sum 함수 */
/* 11월 3일 choijm@dku.edu      */

        .text
.global asm_sum
asm_sum:
        pushl           %ebp
        movl            %esp, %ebp
        subl            $4, %esp

        movl            8(%ebp), %ecx   # count 변수 조기화
        movl            $0, -4(%ebp)
L1:
        cmpl            $0, %ecx
        je              L2
        addl            %ecx, -4(%ebp)
        decl            %ecx
        jmp             L1
L2:
        movl            -4(%ebp), %eax  # return value
        leave
        ret
~
~
~
~
~
~
~
~
~
~
~
~
"asm_sum.s" 22 줄 --81%--                                    3,0-1          모두
```

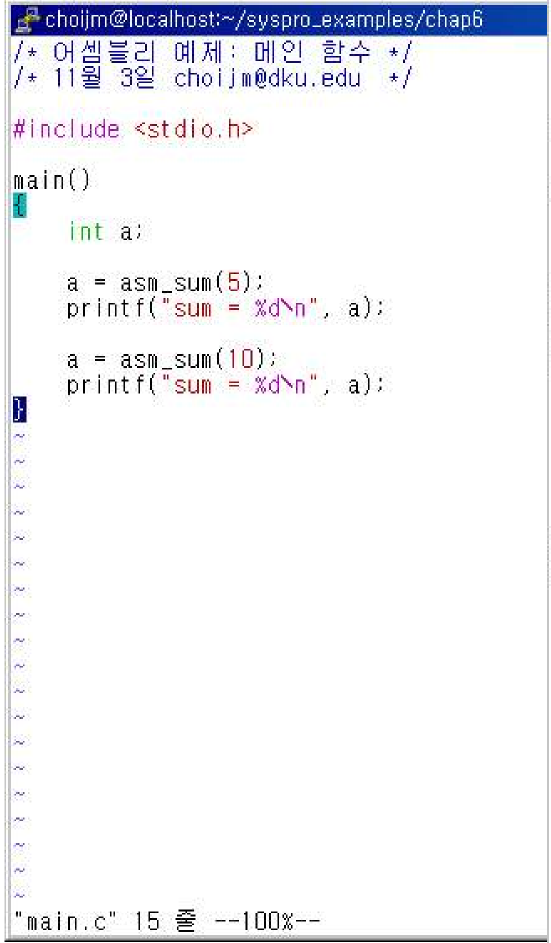
**.text directive: declare text section (the following instructions are resided in the text section)**

**.global directive: declare "asm_sum" visible to the linker**

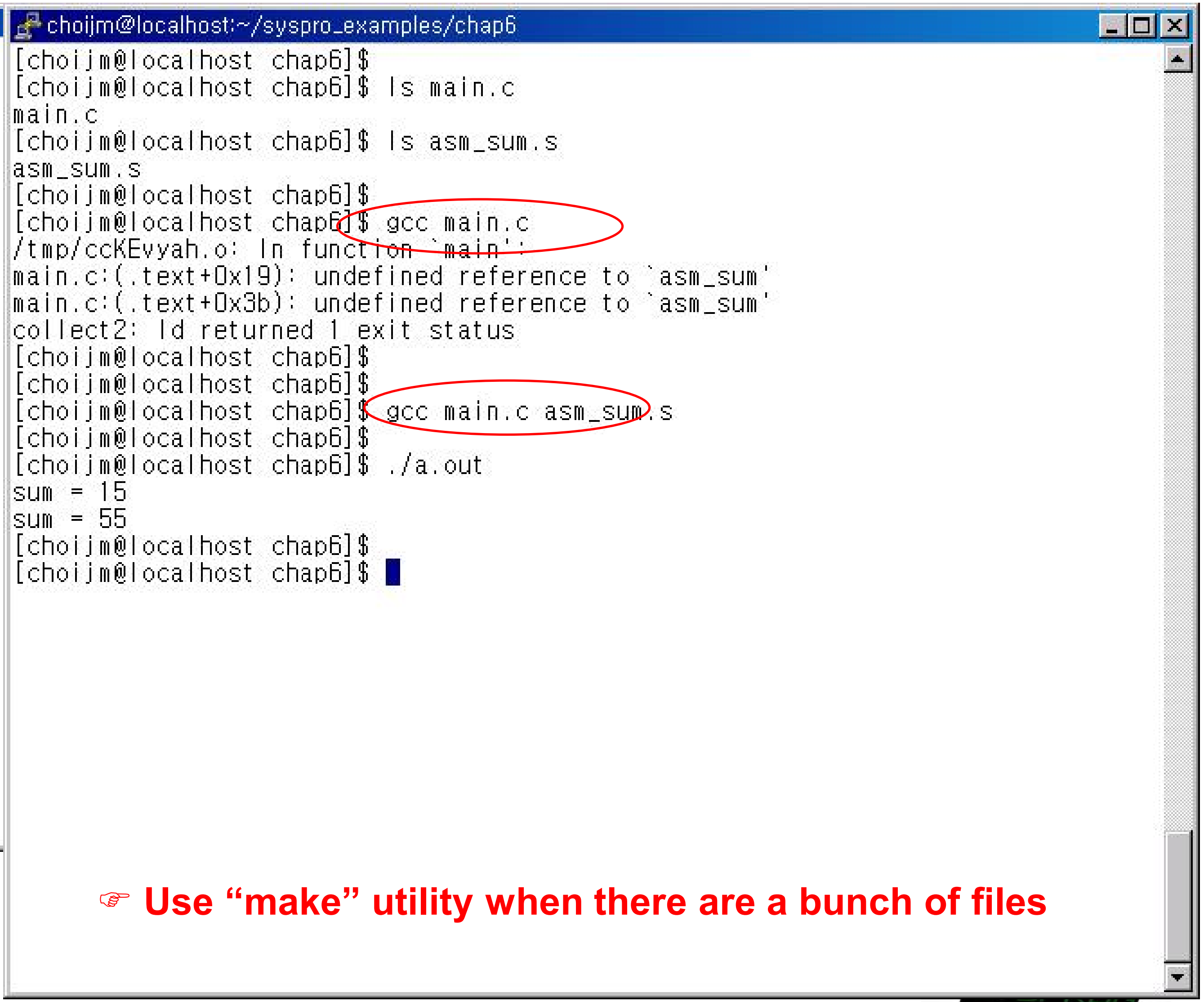
**Memory addressing: displacement(base) or displacement(base, index, scale)**

# Instruction Detail: Make a Program (2/6)

- Execution results of Practice 1



```
/* 어셈블리 예제: 메인 함수 */
/* 11월 3일 choijm@dku.edu  */

#include <stdio.h>

main()
{
    int a;

    a = asm_sum(5);
    printf("sum = %d\n", a);

    a = asm_sum(10);
    printf("sum = %d\n", a);
}
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
"main.c" 15 줄 --100%--
```

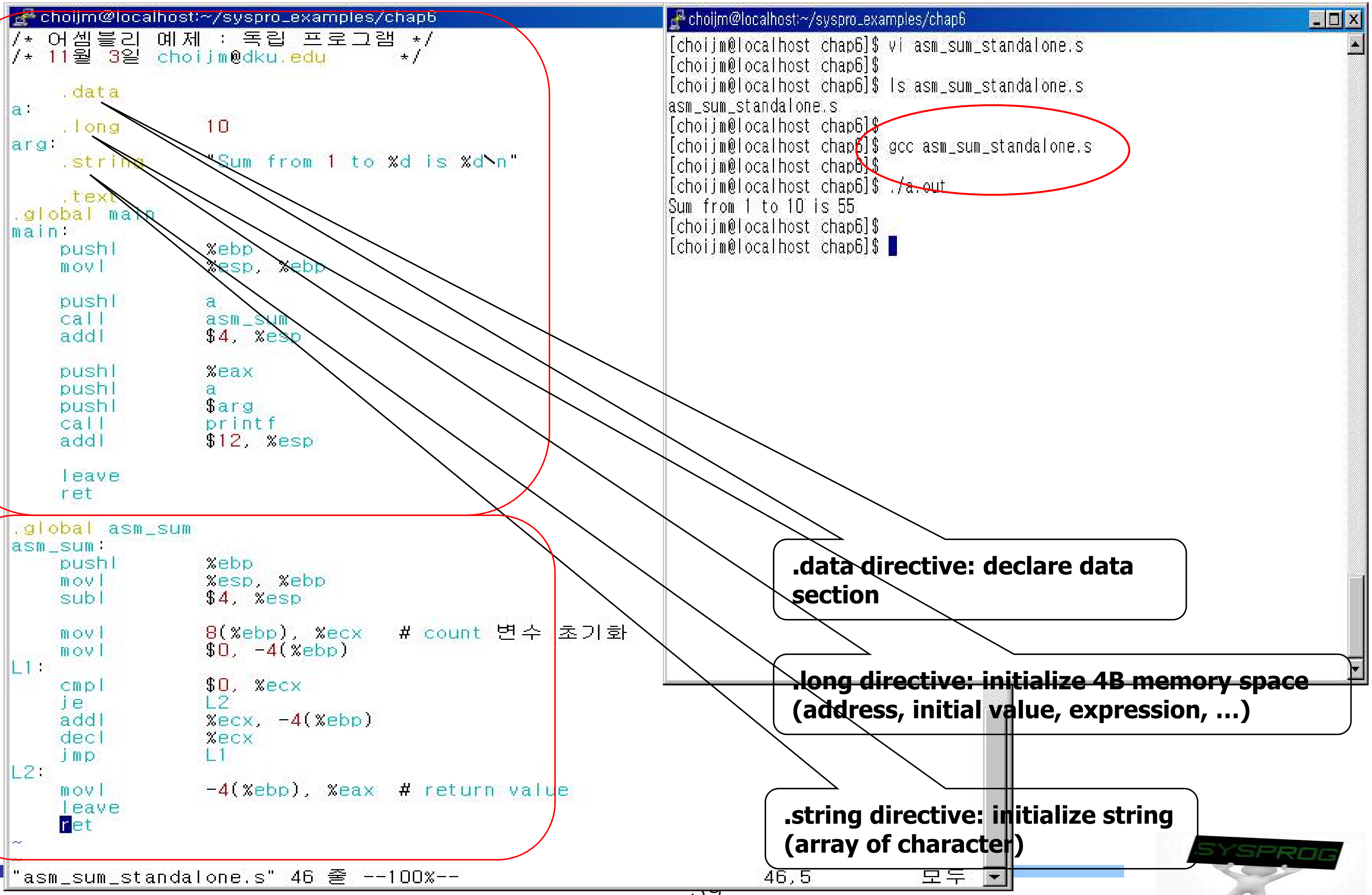


```
[choijm@localhost chap6]$
[choijm@localhost chap6]$ ls main.c
main.c
[choijm@localhost chap6]$ ls asm_sum.s
asm_sum.s
[choijm@localhost chap6]$
[choijm@localhost chap6]$ gcc main.c
/tmp/ccKEvyah.o: In function `main':
main.c:(.text+0x19): undefined reference to `asm_sum'
main.c:(.text+0x3b): undefined reference to `asm_sum'
collect2: ld returned 1 exit status
[choijm@localhost chap6]$
[choijm@localhost chap6]$
[choijm@localhost chap6]$ gcc main.c asm_sum.s
[choijm@localhost chap6]$
[choijm@localhost chap6]$ ./a.out
sum = 15
sum = 55
[choijm@localhost chap6]$
[choijm@localhost chap6]$
```

☞ **Use “make” utility when there are a bunch of files**

# Instruction Detail: Make a Program (3/6)

- Practice 2: Standalone assembly program



choijm@localhost:~/syspro_examples/chap6

```
/* 어셈블리 예제 : 독립 프로그램 */
/* 11월 3일 choijm@dku.edu        */

        .data
a:
        .long           10
arg:
        .string         "Sum from 1 to %d is %d\n"

        .text
.global main
main:
        pushl           %ebp
        movl            %esp, %ebp

        pushl           a
        call            asm_sum
        addl            $4, %esp

        pushl           %eax
        pushl           a
        pushl           $arg
        call            printf
        addl            $12, %esp

        leave
        ret
.global asm_sum
asm_sum:
        pushl           %ebp
        movl            %esp, %ebp
        subl            $4, %esp

        movl            8(%ebp), %ecx   # count 변수 초기화
        movl            $0, -4(%ebp)
L1:
        cmpl            $0, %ecx
        je              L2
        addl            %ecx, -4(%ebp)
        decl            %ecx
        jmp             L1
L2:
        movl            -4(%ebp), %eax  # return value
        leave
        ret
~
~
"asm_sum_standalone.s" 46 줄 --100%--                         46,5          모두
```

choijm@localhost:~/syspro_examples/chap6

```
[choijm@localhost chap6]$ vi asm_sum_standalone.s
[choijm@localhost chap6]$
[choijm@localhost chap6]$ ls asm_sum_standalone.s
asm_sum_standalone.s
[choijm@localhost chap6]$
[choijm@localhost chap6]$ gcc asm_sum_standalone.s
[choijm@localhost chap6]$
[choijm@localhost chap6]$ ./a.out
Sum from 1 to 10 is 55
[choijm@localhost chap6]$
[choijm@localhost chap6]$
```

**.data directive: declare data section**

**.long directive: initialize 4B memory space (address, initial value, expression, …)**

**.string directive: initialize string (array of character)**

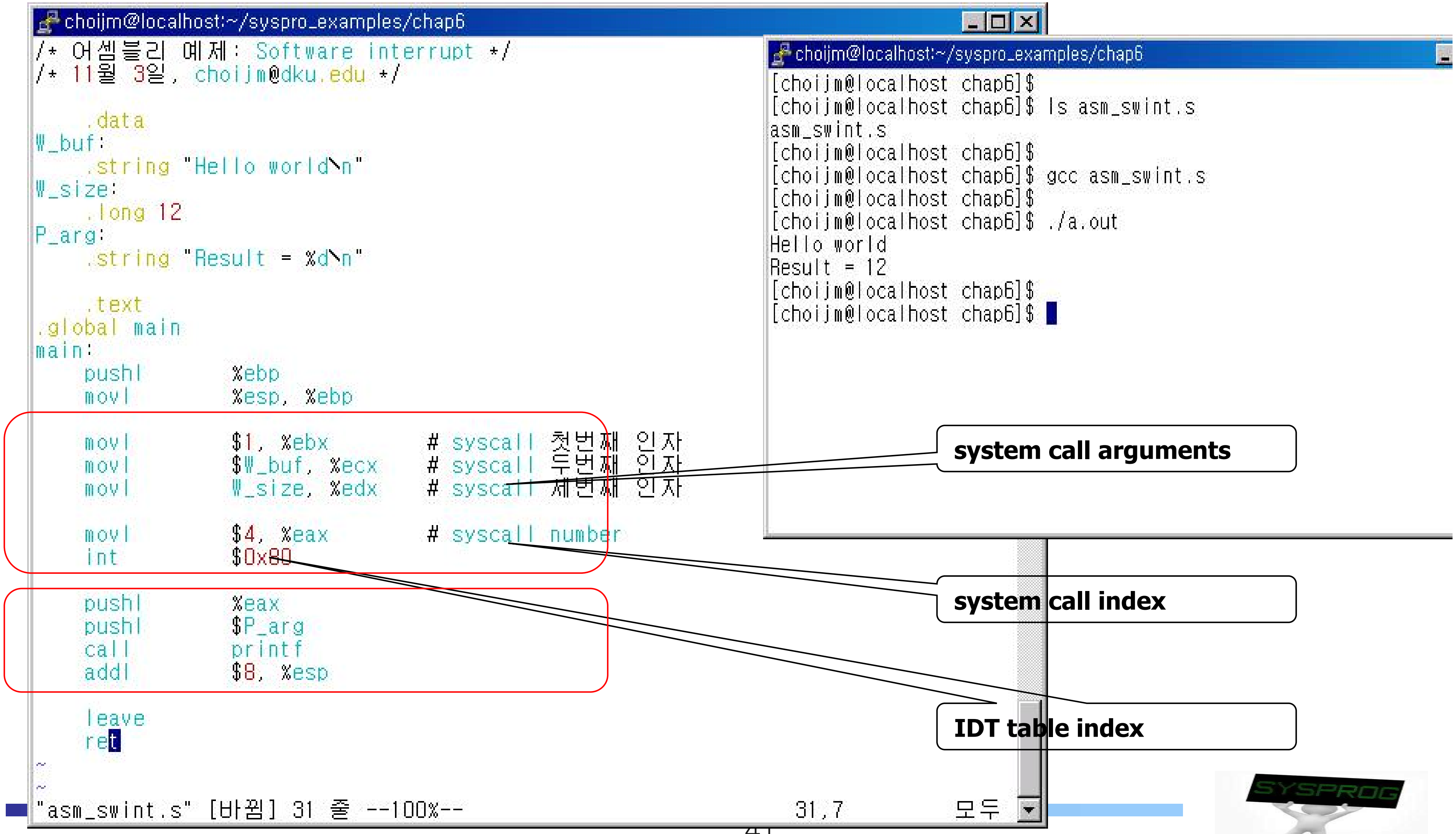
# Instruction Detail: Make a Program (4/6)

- directive
  - ✓ Meta-statements (pseudo-instruction)
  - ✓ Used for giving information to assembler (affect how the assembler operates. not directly executed on CPU)
  - ✓ Begin with . (period)
  - ✓ Representative directive
    - .file, .include
    - .text, .data, .comm, .section
    - .long, .byte,. string, .ascii, .float, .quad
    - .global, .align, .size
    - .set, .equal, .rept, .space
    - .macro, .endm
    - .if, .else, .endif
    - .cfi_startproc, .cfi_endproc for debugging
    - …

☞ **refer to “GNU assembler” in the lecture site or “info as” on the Linux shell**

# Instruction Detail: Make a Program (5/6)

- Software Interrupt
  - ✓ write() system call



choijm@localhost:~/syspro_examples/chap6

```
/* 어셈블리 예제: Software interrupt */
/* 11월 3일, choijm@dku.edu */

        .data
W_buf:
        .string "Hello world\n"
W_size:
        .long 12
P_arg:
        .string "Result = %d\n"

        .text
.global main
main:
        pushl           %ebp
        movl            %esp, %ebp

        movl            $1, %ebx        # syscall 첫번째 인자
        movl            $W_buf, %ecx    # syscall 두번째 인자
        movl            W_size, %edx    # syscall 세번째 인자

        movl            $4, %eax        # syscall number
        int             $0x80

        pushl           %eax
        pushl           $P_arg
        call            printf
        addl            $8, %esp

        leave
        ret
~
~
"asm_swint.s" [바뀜] 31 줄 --100%--                    31,7          모두
```

choijm@localhost:~/syspro_examples/chap6

```
[choijm@localhost chap6]$
[choijm@localhost chap6]$ ls asm_swint.s
asm_swint.s
[choijm@localhost chap6]$
[choijm@localhost chap6]$ gcc asm_swint.s
[choijm@localhost chap6]$
[choijm@localhost chap6]$ ./a.out
Hello world
Result = 12
[choijm@localhost chap6]$
[choijm@localhost chap6]$
```

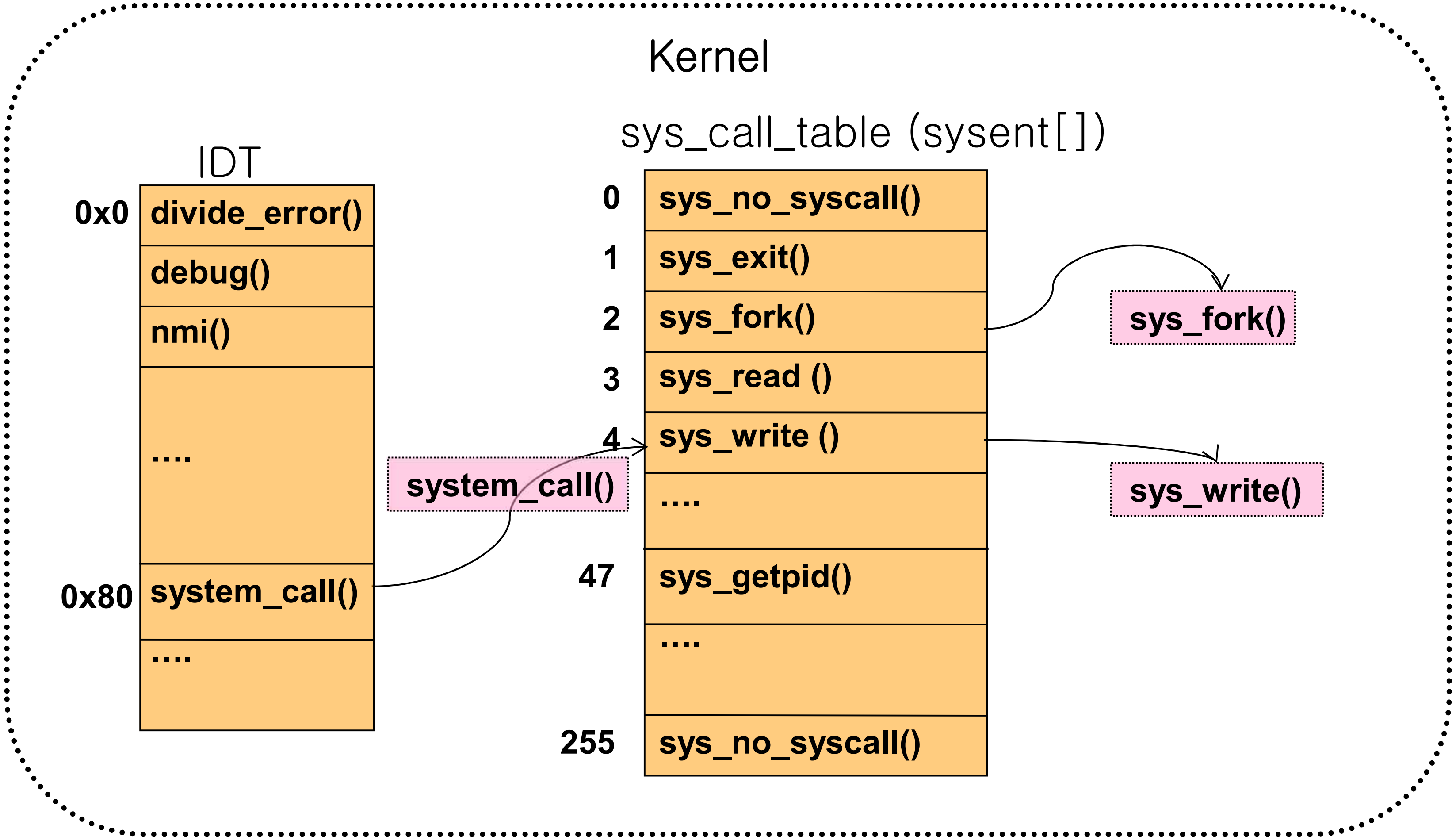
**system call arguments**

**system call index**

**IDT table index**

# Instruction Detail: Make a Program (6/6)

- Software Interrupt (cont')
  - ✓ Interrupt and system call handling



☞ **64bit CPU: use "sysenter (syscall on AMD)" instead of "int"**

# Summary

- Understand ISA
- Know about IA register, memory, and instruction model
- Learn the format of IA instruction
  - ✓ label, opcode, operands, comments
- Learn the types of IA opcode
  - ✓ mov, add, cmp, jmp, push, call, ret, int, …

**☞ Homework 5: Make an assembly program**

**✓ Requirements**

**- print out the prime number from 1 to 100 (using loop → 28 page)**

**- using a function (36 page)**

**- shows student's ID and date (using whoami and date)**

**- Make a report that includes a snapshot and discussion.**

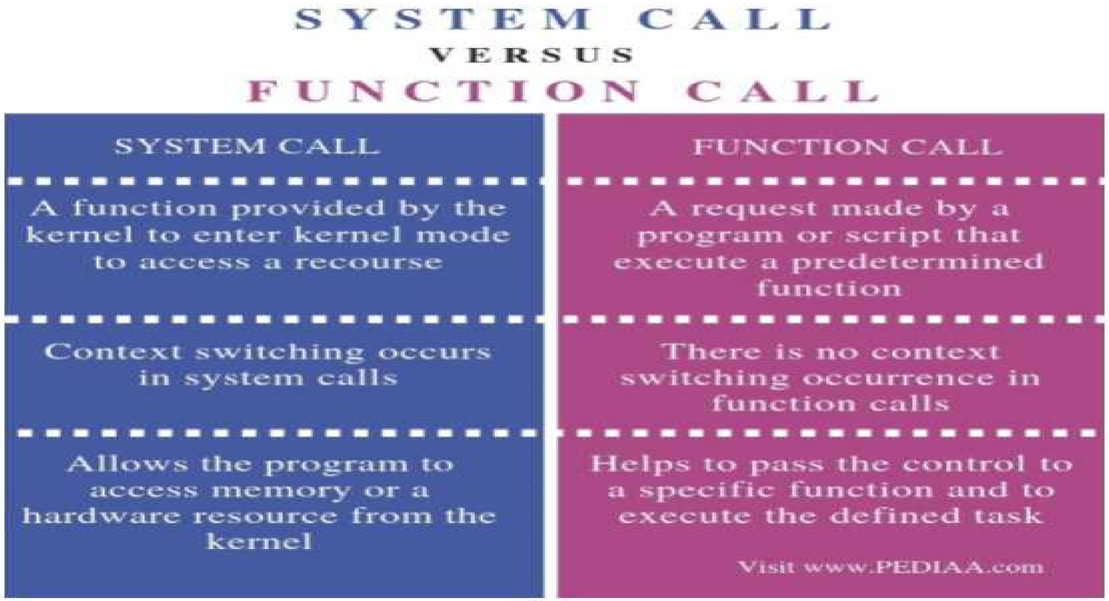
**1) Upload the report to the e-Campus (pdf format!!, 30th October)**

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

**✓ Warn: DO NOT utilize "gcc –S option" (easily detected)**

# Quiz for 10th-Week 2nd-Lesson

- Quiz
  - ✓ 1. What is the make utility in Linux? What is the role of a Makefile?
  - ✓ 2. Discuss the differences between function call and system call at an assembly language viewpoint (at least three).
  - ✓ Due: until 6 PM Friday of this week (6th, November)



SYSTEM CALL VERSUS FUNCTION CALL

| SYSTEM CALL | FUNCTION CALL |
|---|---|
| A function provided by the kernel to enter kernel mode to access a recourse | A request made by a program or script that execute a predetermined function |
| Context switching occurs in system calls | There is no context switching occurrence in function calls |
| Allows the program to access memory or a hardware resource from the kernel | Helps to pass the control to a specific function and to execute the defined task |

Visit www.PEDIAA.com

**(System call and function call at the abstract viewpoint**
**Source: https://pediaa.com/what-is-the-difference-between-system-call-and-function-call/)**

# Appendix1: MU0, A Simple CPU

- Simple CPU from Manchester University
- Architecture
  - Register set
    - PC : program counter
    - ACC : accumulator
    - IR : Instruction Register
  - ALU : Arithmetic-Logic Unit
  - CU : Control Unit (instruction decode and control logic)
  - Memory

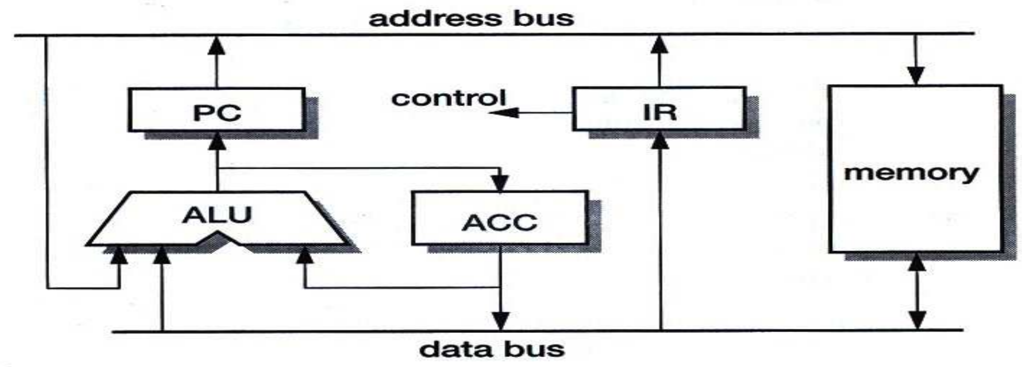


**Figure 1.5** MU0 datapath example.

**(Source: ARM System-on-Chip Architecture, by S. Furber)**

# Appendix1: MU0, A Simple CPU

- Data Transfer

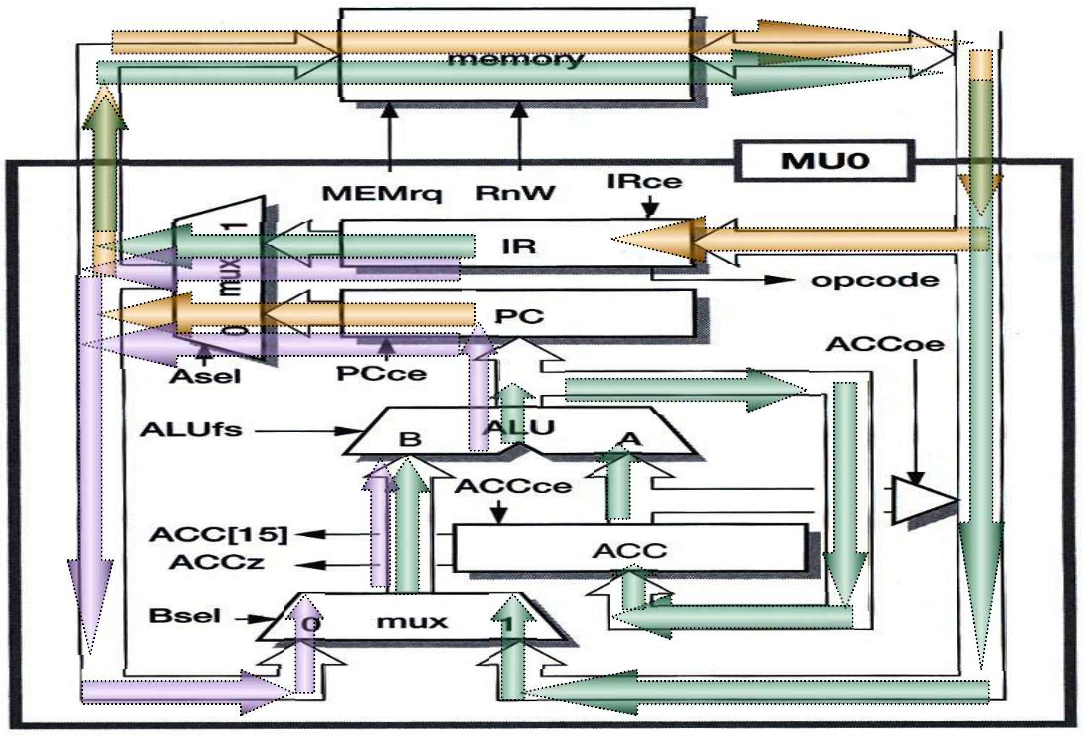


**Figure 1.6** MU0 register transfer level organization.

✓ fetch and execution

# Appendix1: MU0, A Simple CPU

- MU0 instruction set
  - ✓ 16-bit machine with 12-bit address space
  - ✓ 8 instructions (4-bit opcode)
  - ✓ 12-bit operand (4096 address space)

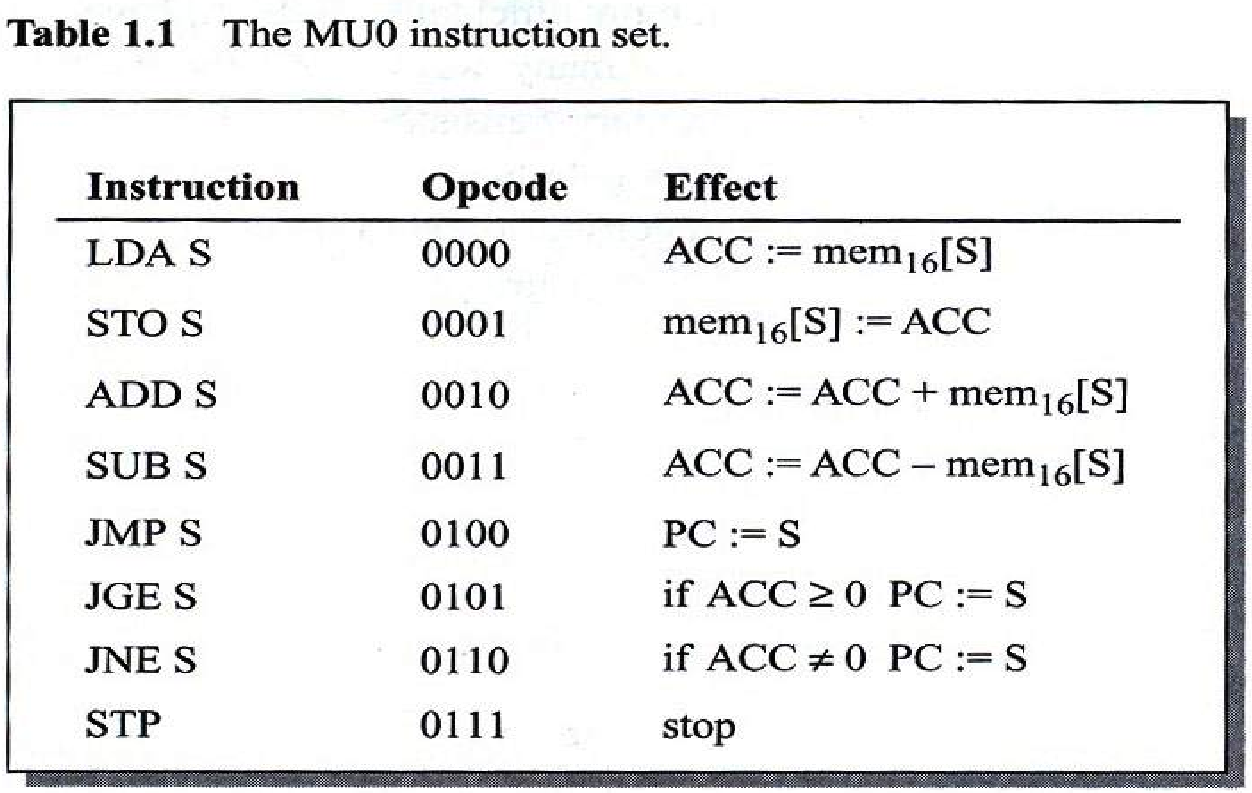


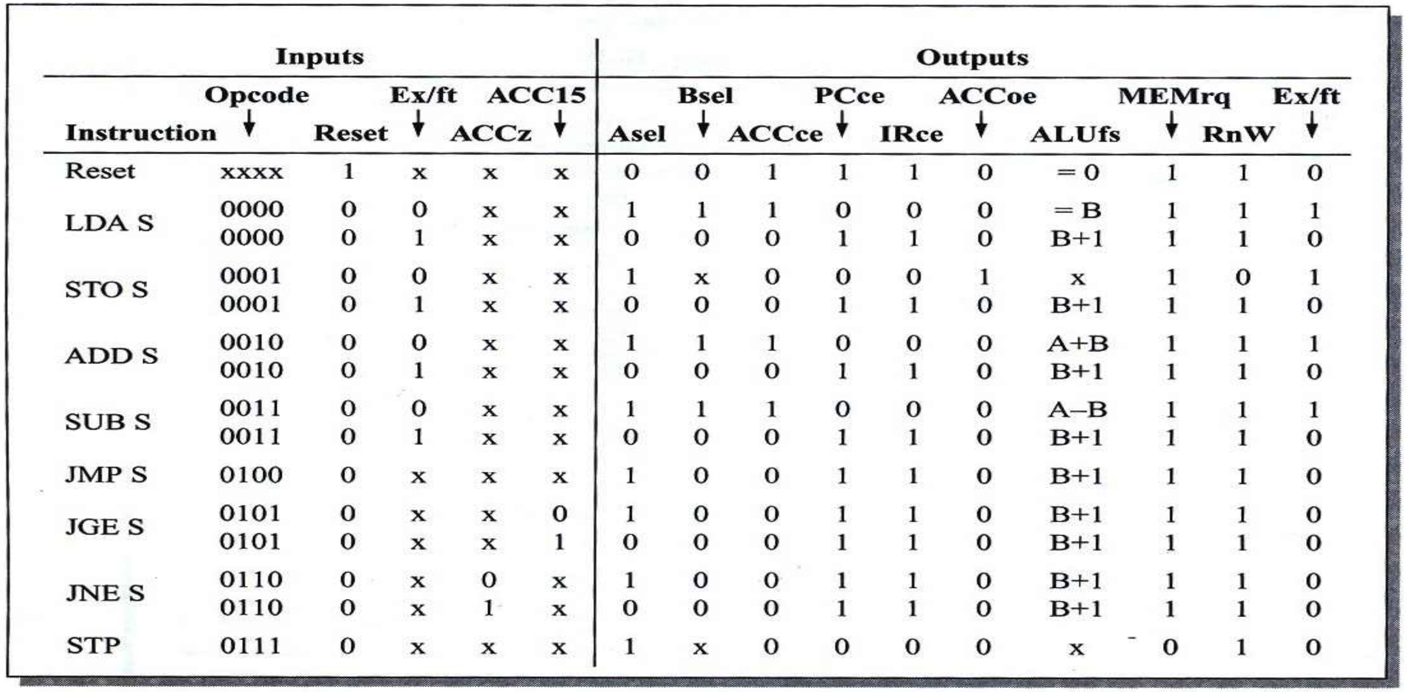
**Table 1.1** The MU0 instruction set.

| Instruction | Opcode | Effect |
|---|---|---|
| LDA S | 0000 | ACC := $\text{mem}_{16}$[S] |
| STO S | 0001 | $\text{mem}_{16}$[S] := ACC |
| ADD S | 0010 | ACC := ACC + $\text{mem}_{16}$[S] |
| SUB S | 0011 | ACC := ACC – $\text{mem}_{16}$[S] |
| JMP S | 0100 | PC := S |
| JGE S | 0101 | if ACC $\geq$ 0 PC := S |
| JNE S | 0110 | if ACC $\neq$ 0 PC := S |
| STP | 0111 | stop |

# Appendix1: MU0, A Simple CPU

- Control Logic

Table 1.2 MU0 control logic.



| | Inputs | | | | | Outputs | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Instruction | Opcode | Reset | Ex/ft | ACCz | ACC15 | Asel | Bsel | ACCce | PCce | IRce | ACCoe | ALUfs | MEMrq | RnW | Ex/ft |
| Reset | xxxx | 1 | x | x | x | 0 | 0 | 1 | 1 | 1 | 0 | = 0 | 1 | 1 | 0 |
| LDA S | 0000 | 0 | 0 | x | x | 1 | 1 | 1 | 0 | 0 | 0 | = B | 1 | 1 | 1 |
| | 0000 | 0 | 1 | x | x | 0 | 0 | 0 | 1 | 1 | 0 | B+1 | 1 | 1 | 0 |
| STO S | 0001 | 0 | 0 | x | x | 1 | x | 0 | 0 | 0 | 1 | x | 1 | 0 | 1 |
| | 0001 | 0 | 1 | x | x | 0 | 0 | 0 | 1 | 1 | 0 | B+1 | 1 | 1 | 0 |
| ADD S | 0010 | 0 | 0 | x | x | 1 | 1 | 1 | 0 | 0 | 0 | A+B | 1 | 1 | 1 |
| | 0010 | 0 | 1 | x | x | 0 | 0 | 0 | 1 | 1 | 0 | B+1 | 1 | 1 | 0 |
| SUB S | 0011 | 0 | 0 | x | x | 1 | 1 | 1 | 0 | 0 | 0 | A–B | 1 | 1 | 1 |
| | 0011 | 0 | 1 | x | x | 0 | 0 | 0 | 1 | 1 | 0 | B+1 | 1 | 1 | 0 |
| JMP S | 0100 | 0 | x | x | x | 1 | 0 | 0 | 1 | 1 | 0 | B+1 | 1 | 1 | 0 |
| JGE S | 0101 | 0 | x | x | 0 | 1 | 0 | 0 | 1 | 1 | 0 | B+1 | 1 | 1 | 0 |
| | 0101 | 0 | x | x | 1 | 0 | 0 | 0 | 1 | 1 | 0 | B+1 | 1 | 1 | 0 |
| JNE S | 0110 | 0 | x | 0 | x | 1 | 0 | 0 | 1 | 1 | 0 | B+1 | 1 | 1 | 0 |
| | 0110 | 0 | x | 1 | x | 0 | 0 | 0 | 1 | 1 | 0 | B+1 | 1 | 1 | 0 |
| STP | 0111 | 0 | x | x | x | 1 | x | 0 | 0 | 0 | 0 | x | 0 | 1 | 0 |

- ✓ FSM(Finite State Machine): Execute, Fetch state
  - Initialization: reset (known state) makes the ALU output as zero
  - Register change: when XXce is '1'
  - Multiplexer: Asel, Bsel

# Appendix1: MU0, A Simple CPU

- ALU logic for one bit
  - ✓ ALU functions required
    - A+B: normal adder
    - A-B: complement and adding
    - B: force A and carry-in to zero
    - B+1: force A to zero and carry-in to 1
    - 0: reset

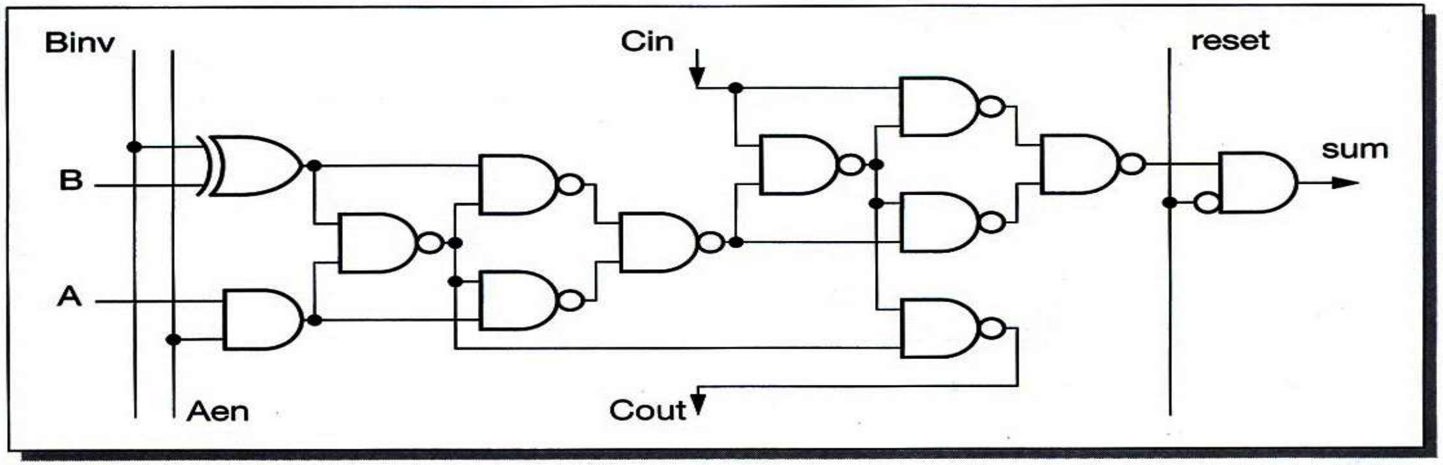


**Figure 1.7** MU0 ALU logic for one bit.

# Appendix1: MU0, A Simple CPU

- MU0 extensions
  - ✓ Extending the address space
  - ✓ Adding more addressing modes
  - ✓ Allowing the PC to be saved in order to support a subroutine mechanism
  - ✓ Adding more registers
  - ✓ Support interrupts
  - ✓ ...