

# Lecture Note 6.

# IA Assembly Programming

October 25, 2021  
Jongmoo Choi  
Dept. of Software  
Dankook University

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

(Copyright © 2020 by Jongmoo Choi, All Rights Reserved.  
Distribution requires permission)

# 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:

```
1 int accum = 0;
2
3 int sum(int x, int y)
4 {
5     int t = x + y;
6     accum += t;
7     return t;
8 }
```

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:

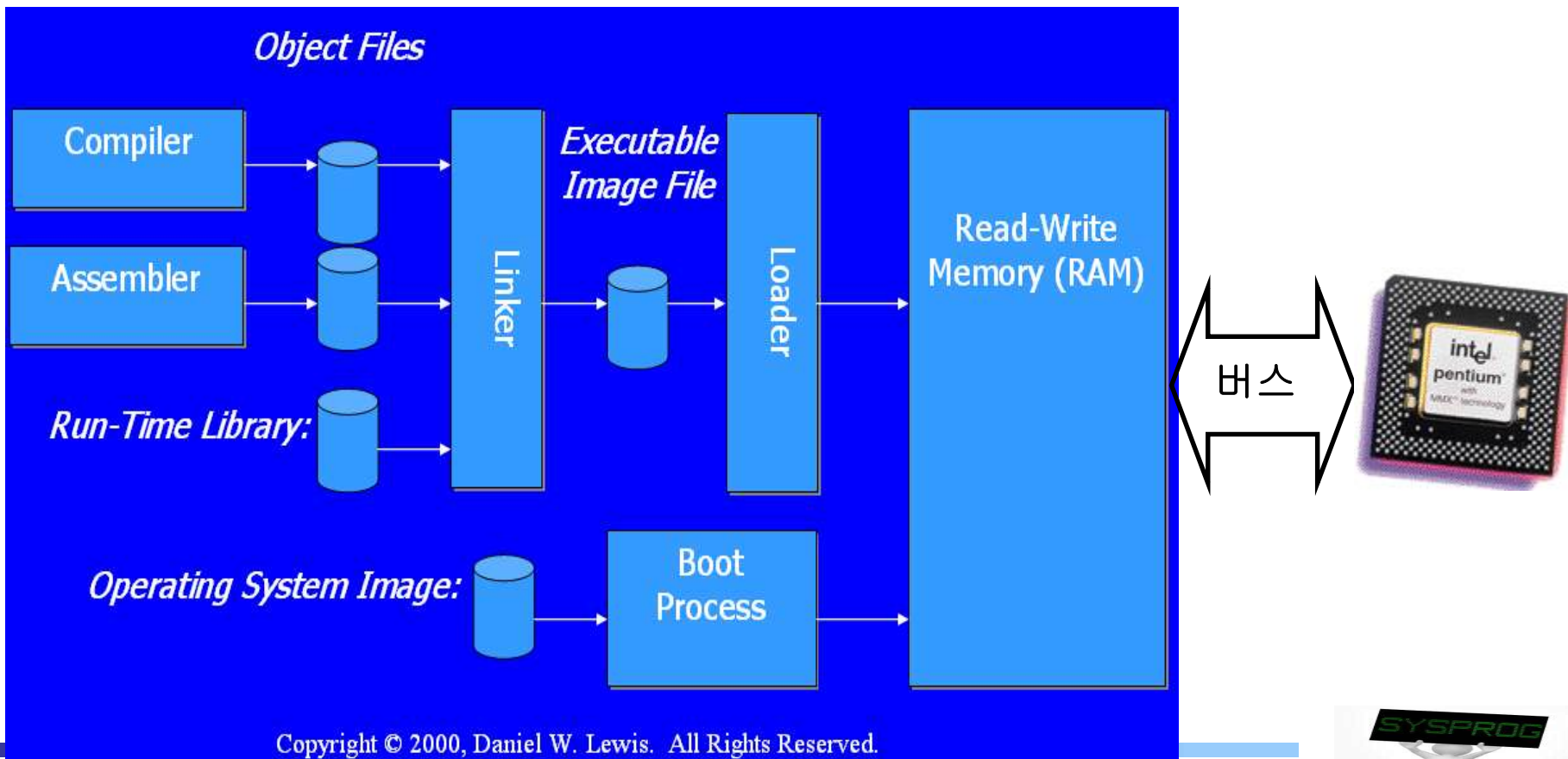
```
sum:
    pushl   %ebp
    movl   %esp, %ebp
    movl   12(%ebp), %eax
    addl   8(%ebp), %eax
    addl   %eax, %accum
    popl   %ebp
    ret
```

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



# 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**



# 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)
  - Grasp the mechanism how hardware and software are cooperated (hardware software **co-design**)



# CPU (1/5)

---

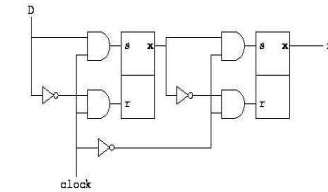
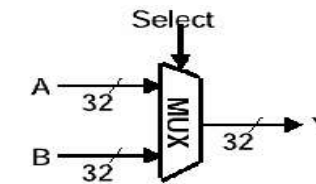
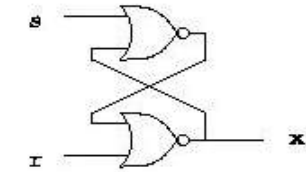
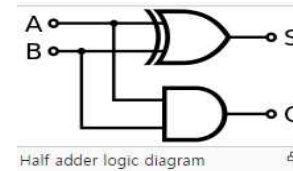
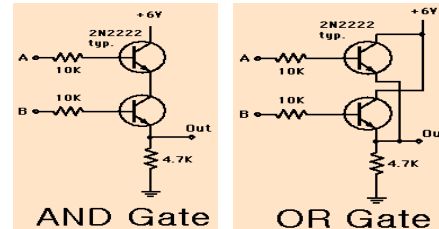
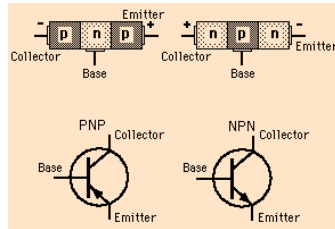
- What is a Processor?



# CPU (2/5)

## ■ Various Viewpoints of Processor

- ✓ 1. Transistor + Gate + Logic + Clock



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

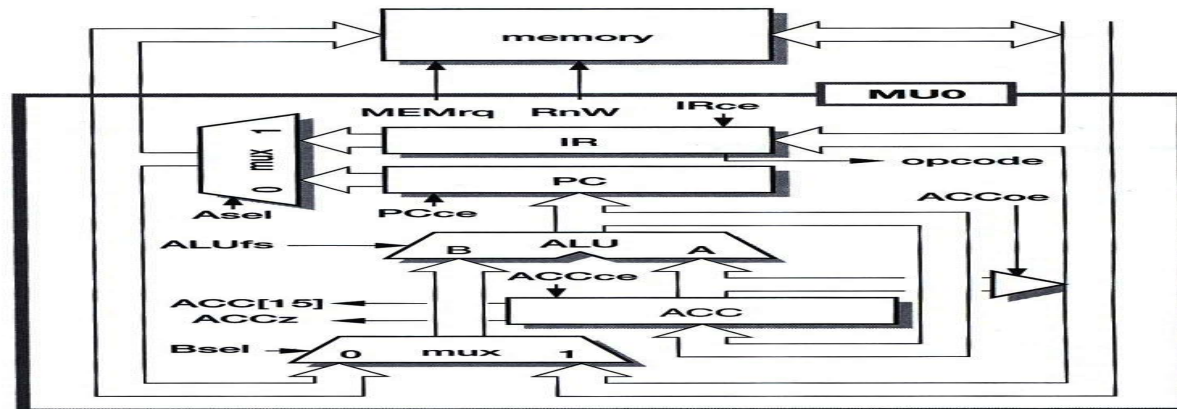


Figure 1.6 MU0 register transfer level organization.

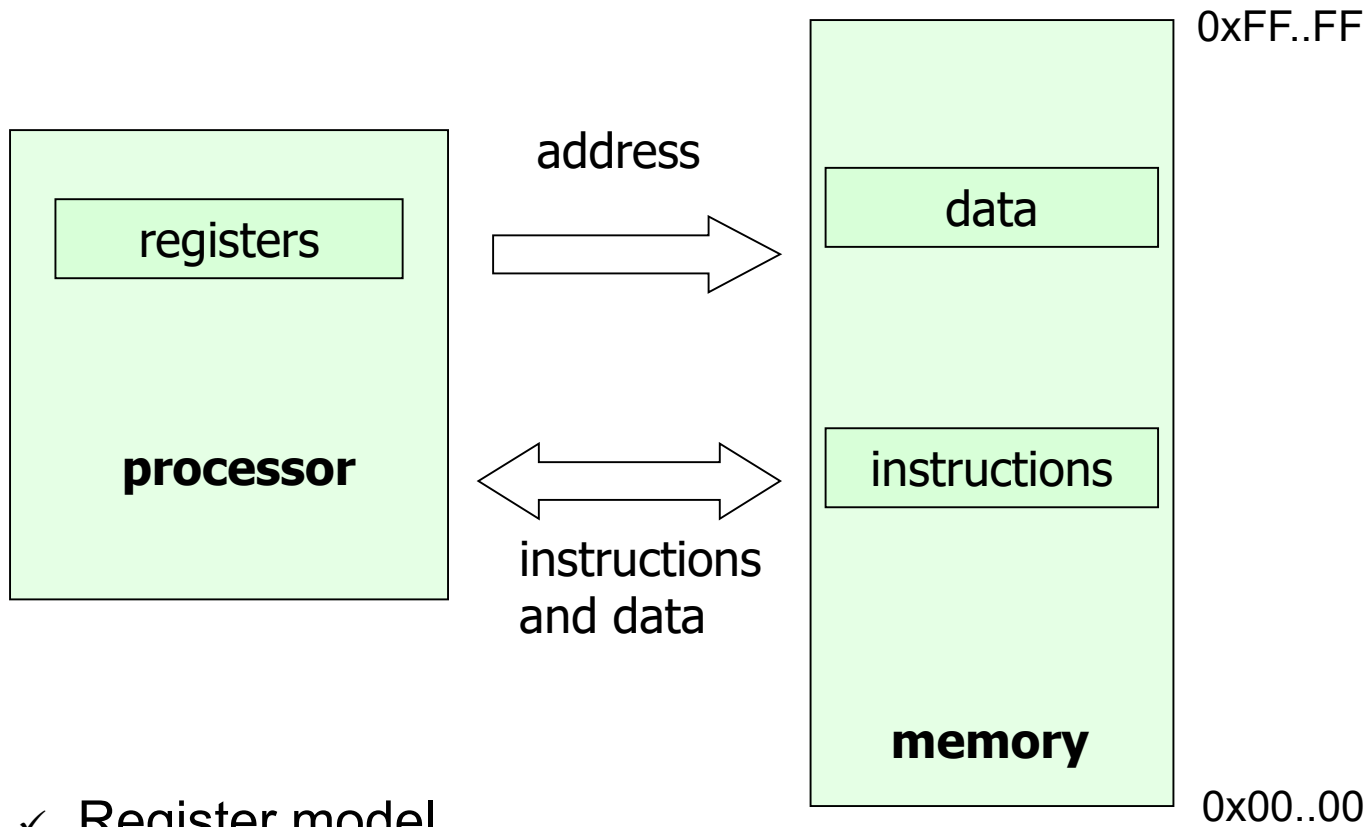
(Source: MU0 in Appendix 1)

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



# CPU (3/5)

- Instruction Set Architecture: Register + Instructions

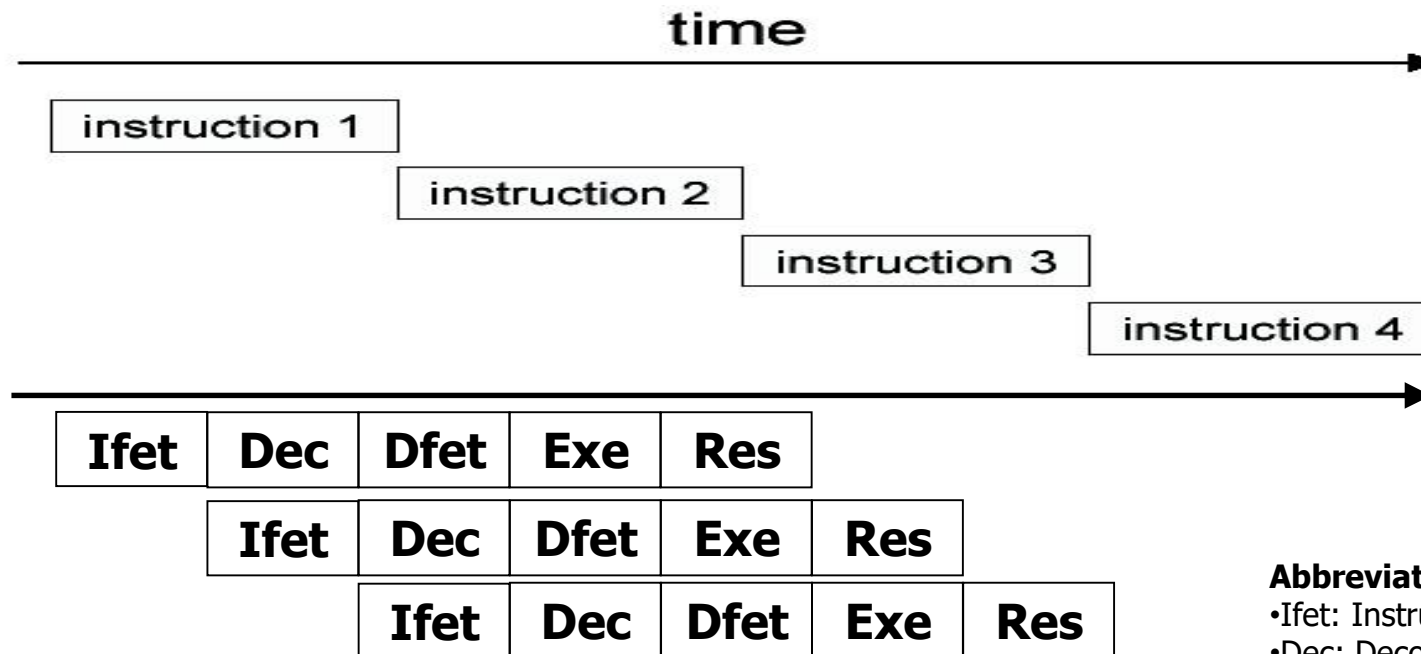


- ✓ Register model
- ✓ Memory model
- ✓ Instruction model



# CPU (4/5)

## ■ Performance Characteristics: Pipeline, Superscalar, Cache



### 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

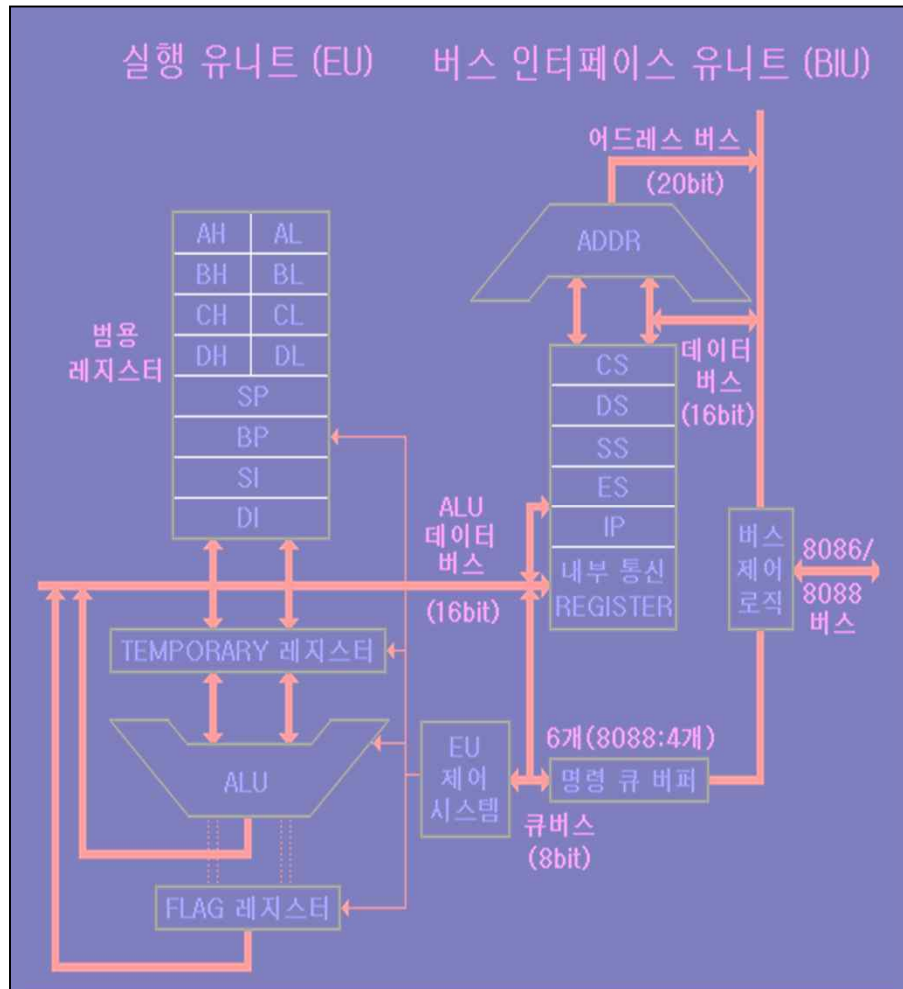




# CPU (5/5)

- Performance Characteristics: Pipeline, Superscalar, Cache

## 8086



## Pentium

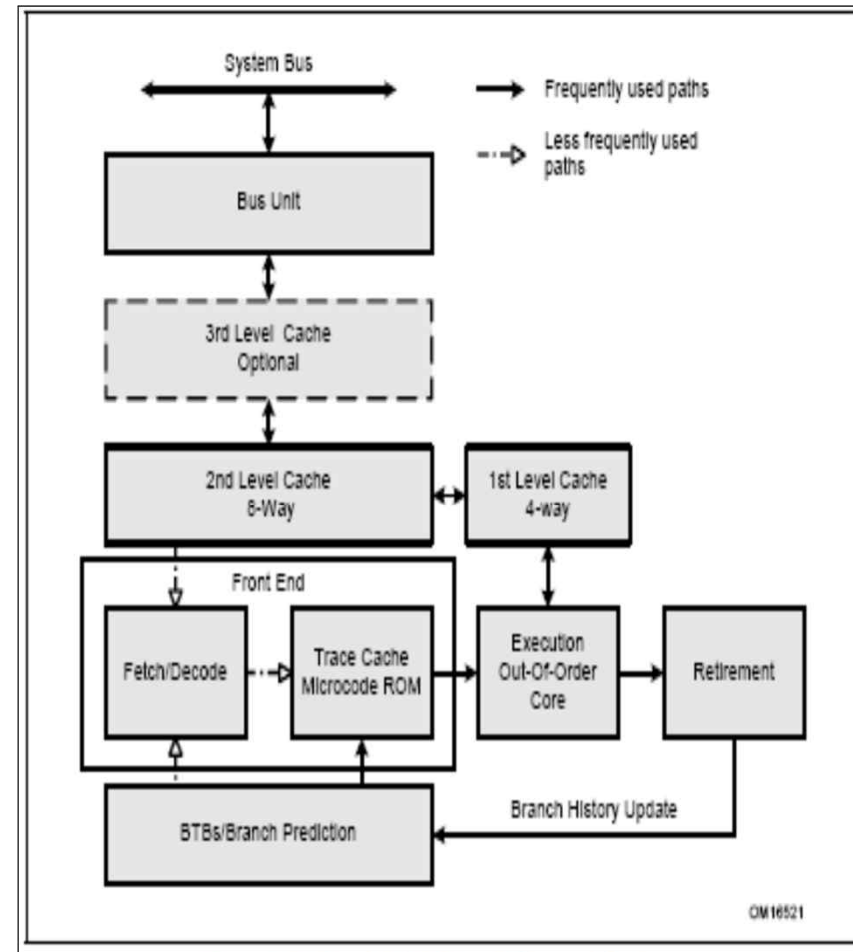


Figure 2-2. The Intel NetBurst Microarchitecture

(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

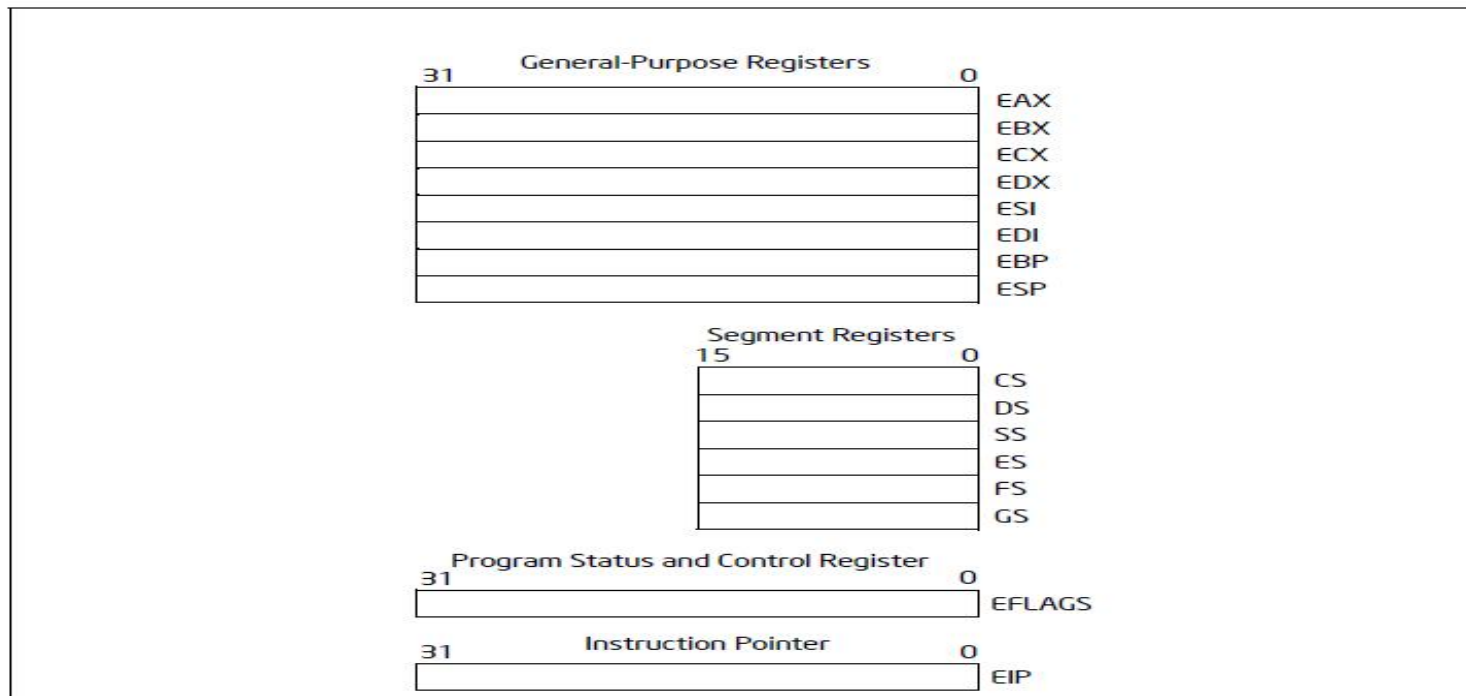


Figure 3-4. General System and Application Programming Registers

(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

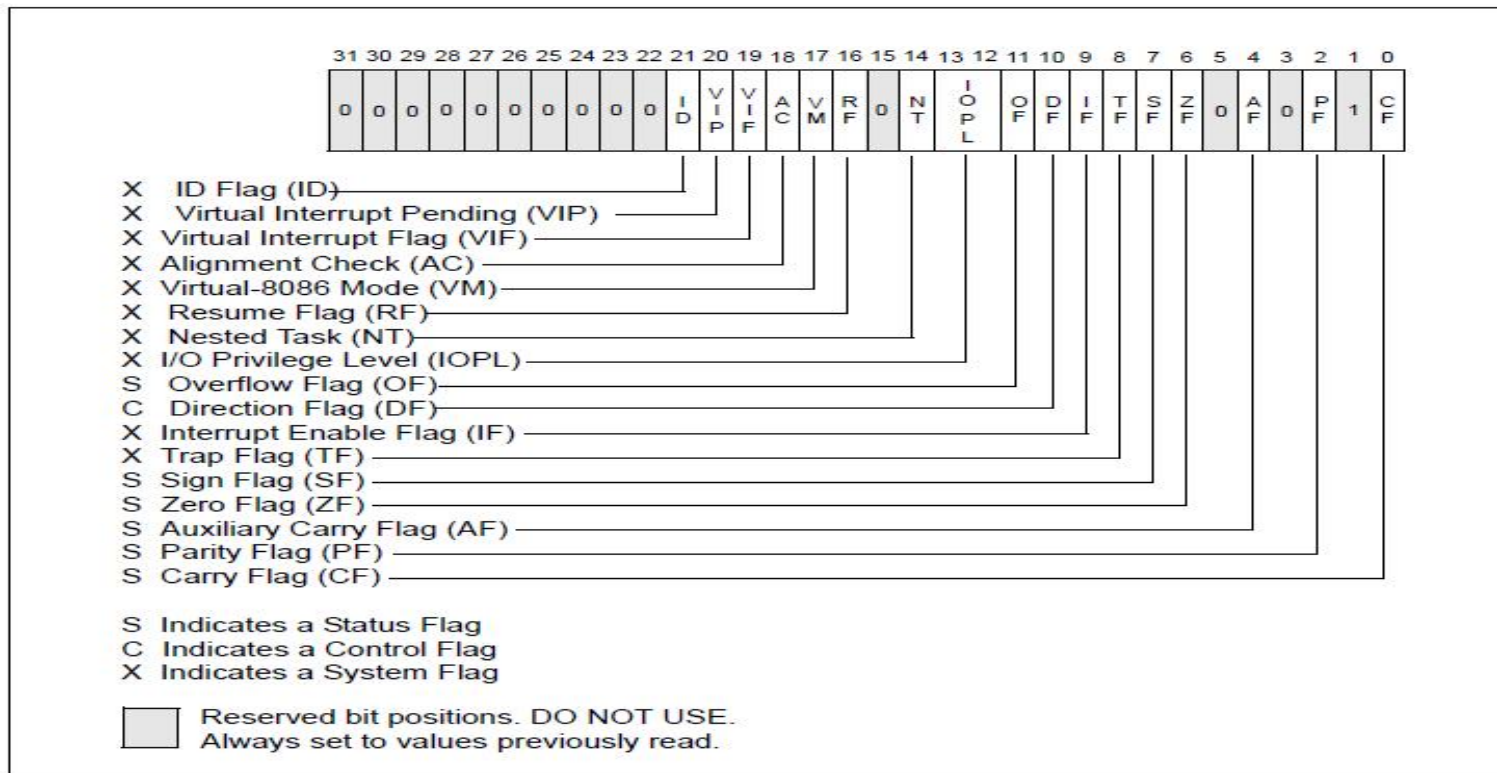


Figure 3-8. EFLAGS Register

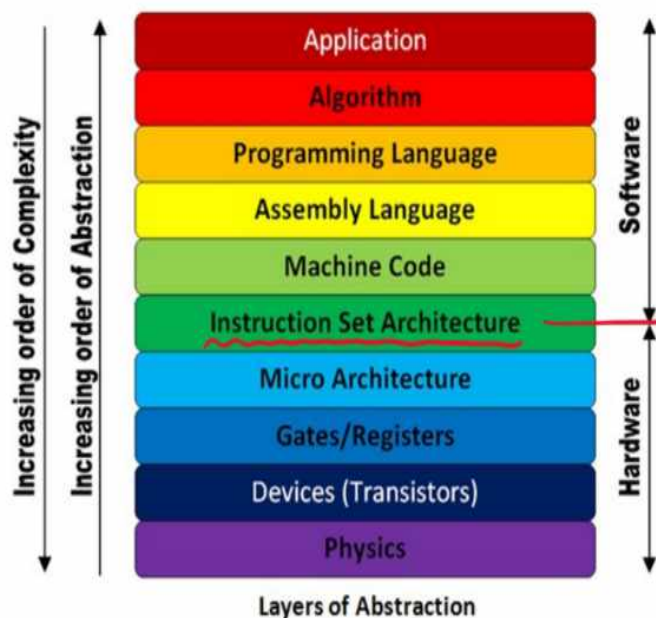
- ☞ Refer to the IA-32 Basic Architecture, Chapter 3.4.3 for the role of each bit
- ☞ Intel CPU has several additional registers such as CR0, CR2, CR3, IDTR, GDTR, debugging registers, FPU registers, and MMX registers. (see LN\_chapter 7)



# Quiz for 9<sup>th</sup>-Week 1<sup>st</sup>-Lesson

## ■ Quiz

- ✓ 1. There are various viewpoints regarding CPU. Discuss 4 viewpoints learned in this class.
- ✓ 2. Explain the differences between eax, rax and ax in the register model of IA. What is the merit of the more registers?
- ✓ Due: until 6 PM Friday of this week (5<sup>th</sup>, November)



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

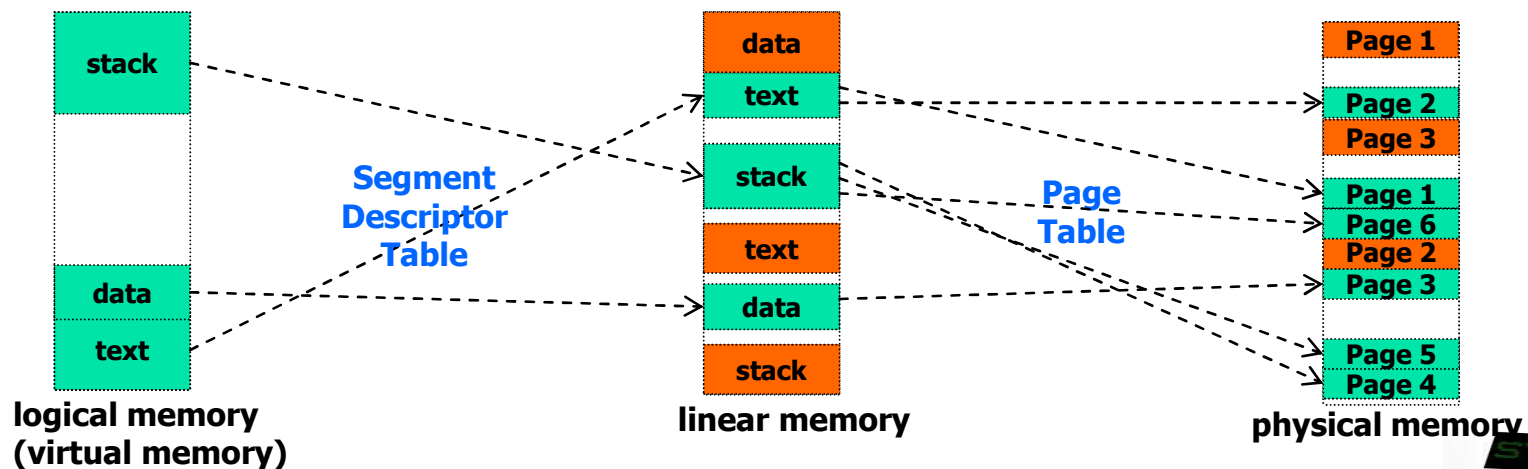
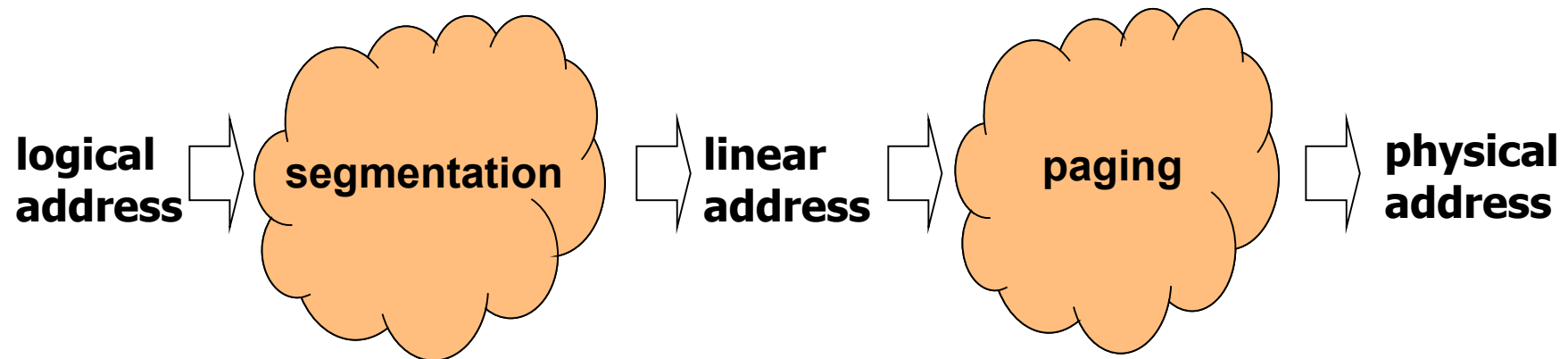
Figure 3.35 Integer registers. The existing eight registers are extended to 64-bit versions, and eight new registers are added. Each register can be accessed as either 8 bits (byte), 16 bits (word), 32 bits (double word), or 64 bits (quad word).

(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)

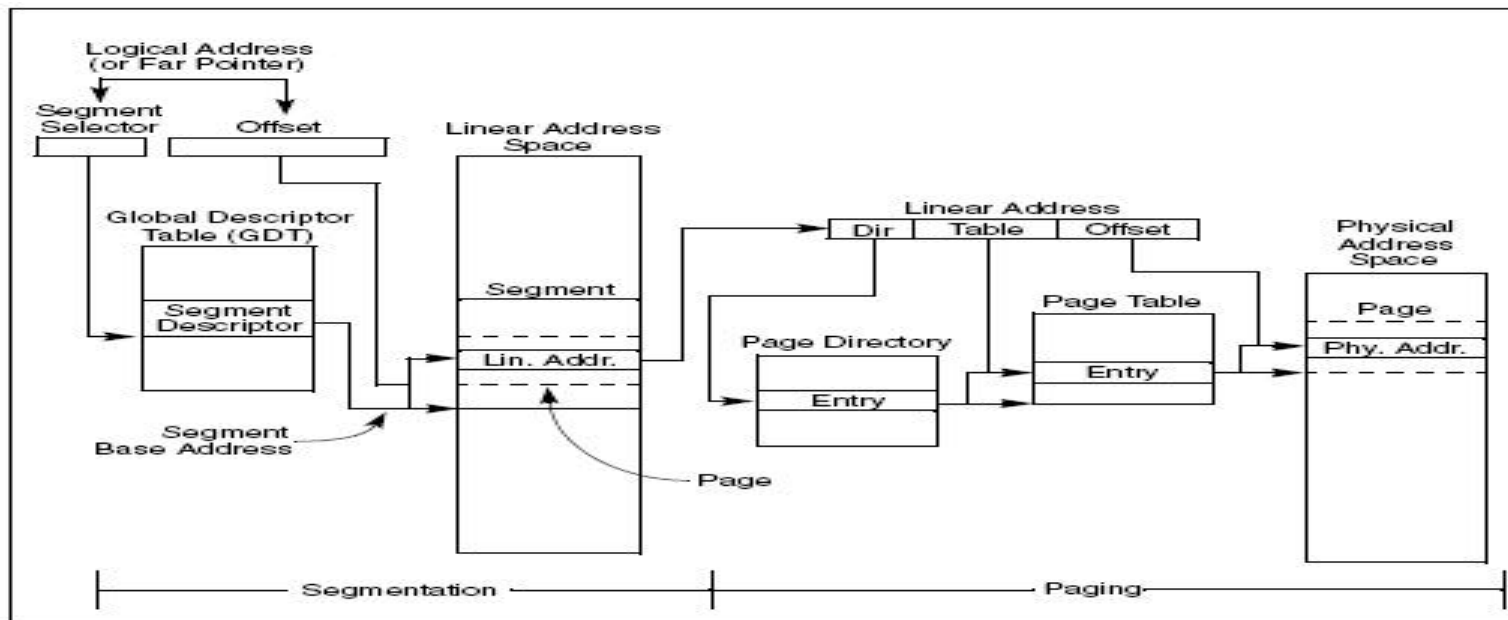


Figure 3-1. Segmentation and Paging

☞ 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

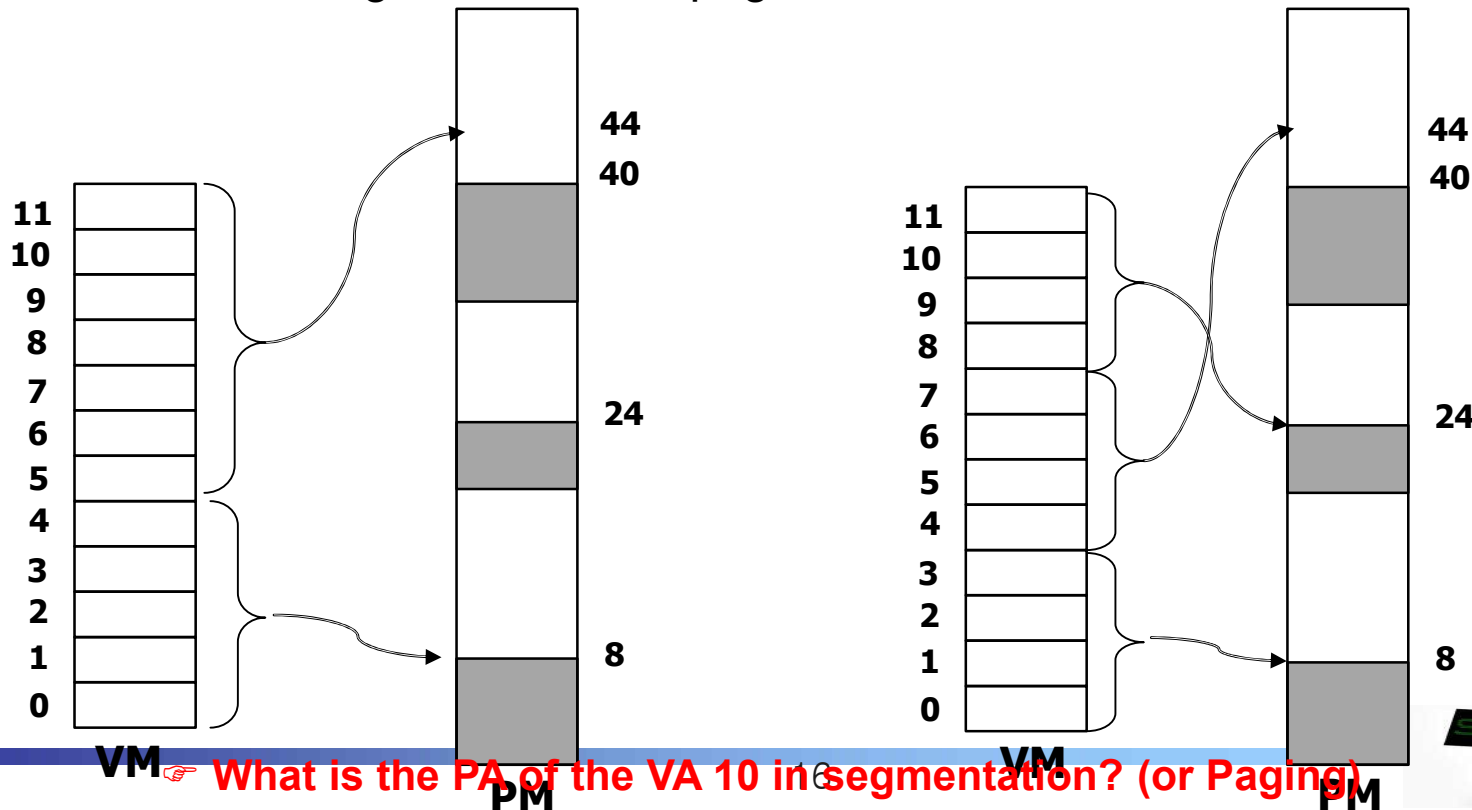
- Address translation: segment table vs. page table
- How to: seg # + offset vs. page # + offset

0 → 8, size = 5
5 → 44, size = 7

Segment table

0 → 8
4 → 44
8 → 24

Page table





# Memory Model (4/6)

## ■ Revisit

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

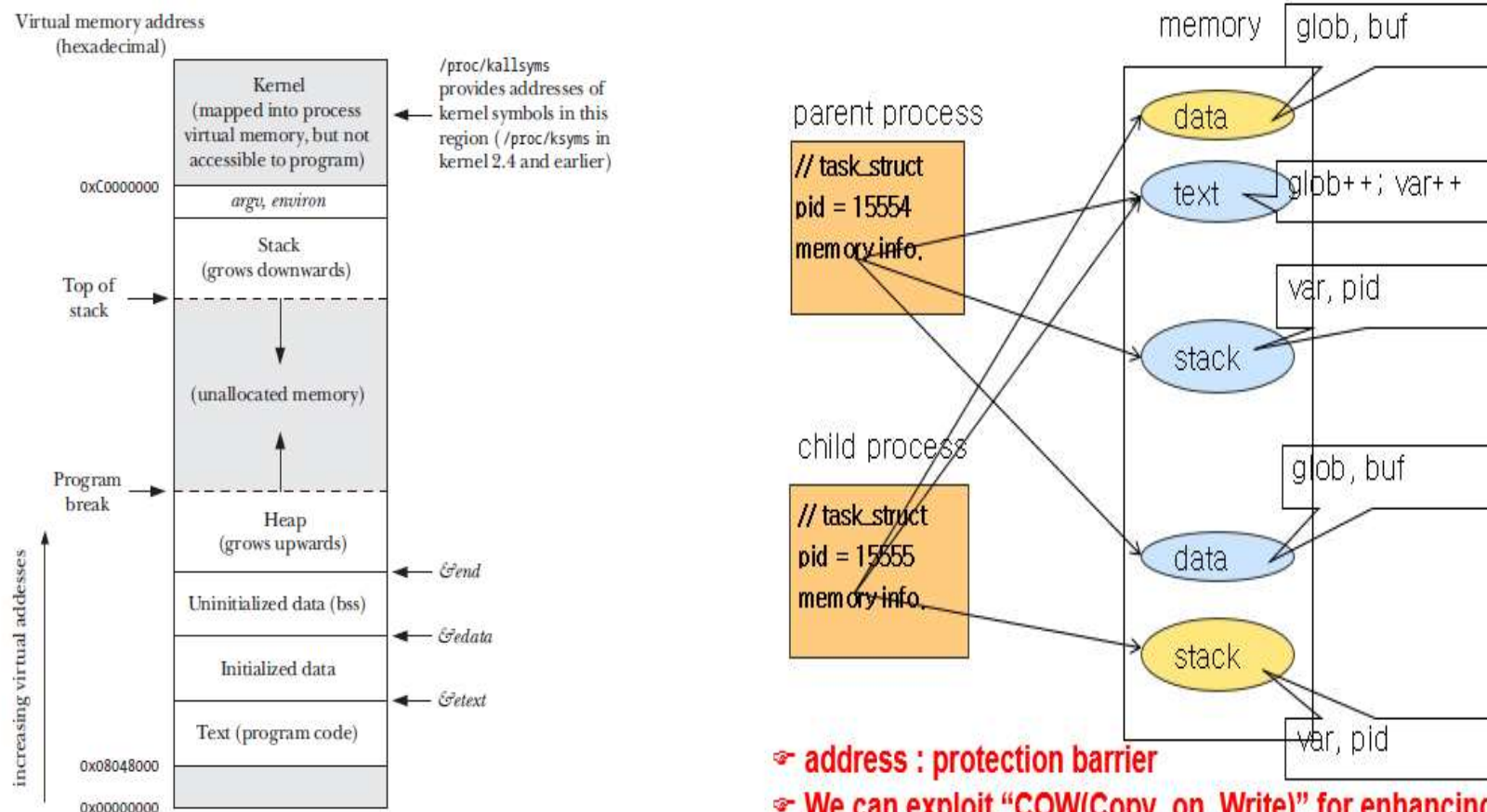


Figure 6-1: Typical memory layout of a process on Linux/x86-32

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

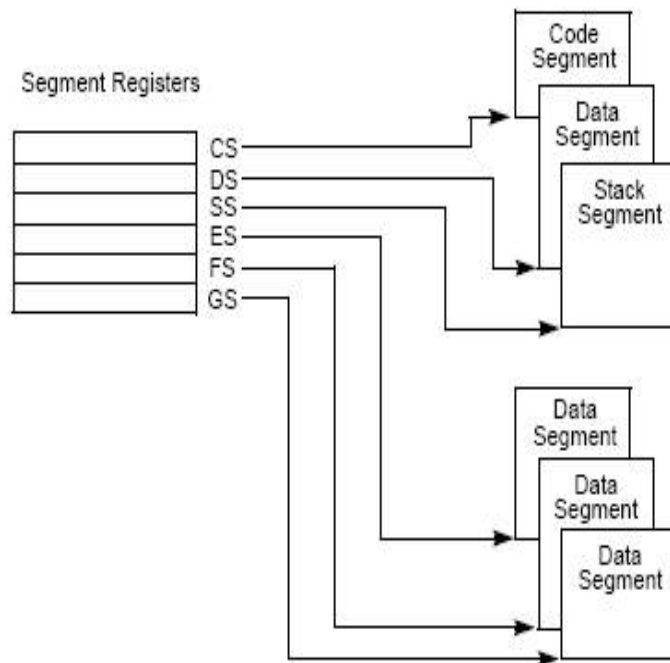


# Memory Model (Optional) (5/6)

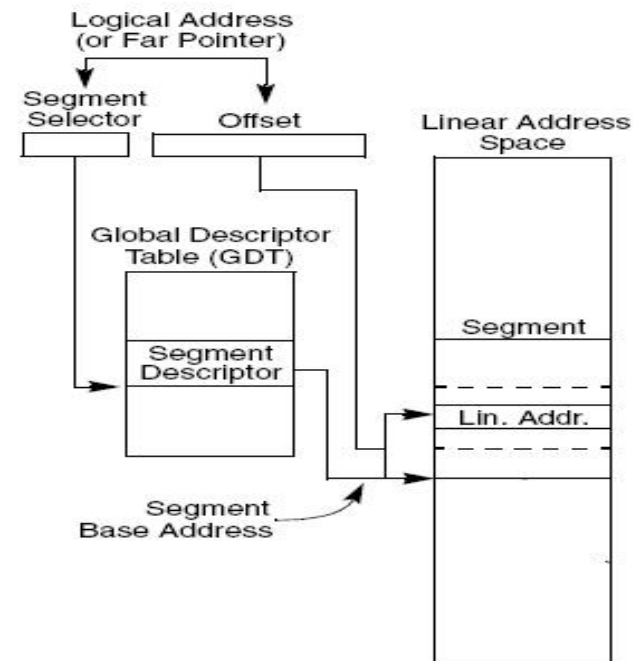
## ■ Segmentation on IA

- ✓ Real Address Model: 8086 compatible, support 1MB ( $\text{seg.} \ll 4 + \text{offset}$ )
- ✓ 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

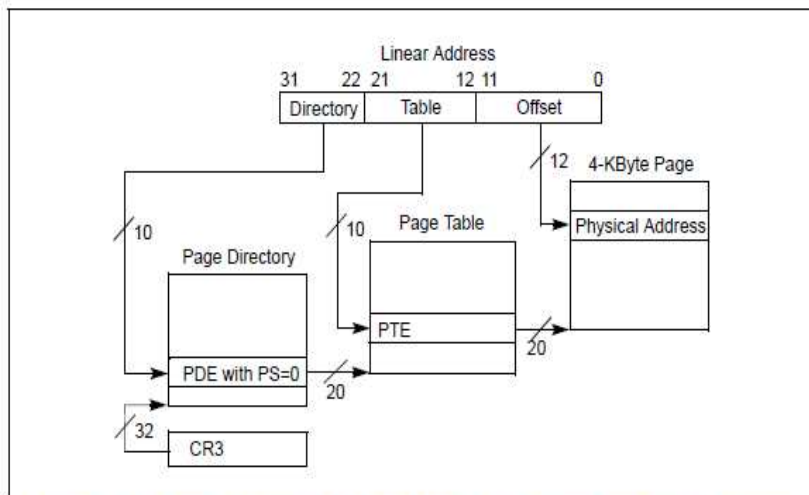


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

### 64 bit CPU

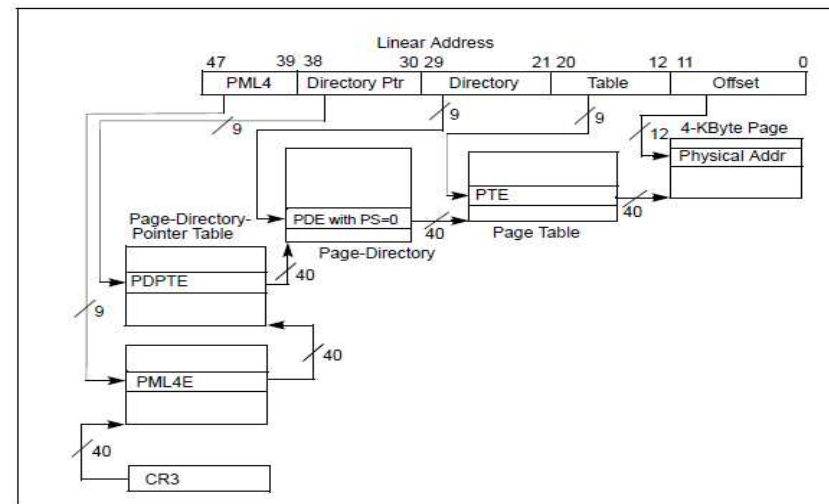


Figure 4-8. Linear-Address Translation to a 4-KByte Page using IA-32e Paging

(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$ 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
.global 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, %eax      # a = 2
    movl   %eax, %eax    # b = a
    movl   %eax, %eax
    addl   %eax, %eax
    movl   %eax, %eax    # c = a + b
    subl   $8, %esp
    pushl   c
    pushl   $.LC0
    call   printf
    addl   $16, %esp
    leave
    .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%--
```

**operand : reg, mem, literal**

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

movl \$2, %eax # a = 2  
movl %eax, %eax # b = a  
movl %eax, %eax  
addl %eax, %eax  
movl %eax, %eax # c = a + b

**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, %eax
    movl   %eax, %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  $.LCD
    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.

31	15	8	7	0
%eax	%ax	%ah	%al	
%ecx	%cx	%ch	%cl	
%edx	%dx	%dh	%dl	
%ebx	%bx	%bh	%bl	
%esi	%si			
%edi	%di			
%esp	%sp			
%ebp	%bp			

Stack pointer  
Frame pointer

# 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$
```

We see that the Intel and AT&T 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.

```
19  shr  %eax, 4
20  sal  %eax, 4
21  sub  %esp, %eax
22  mov  DWORD PTR a1, 64
23  mov  %eax, DWORD PTR a1
24  add  %eax, 1
25  mov  DWORD PTR a2, %eax
26  mov  %eax, DWORD PTR a1
27  mov  WORD PTR b, %ax
28  mov  %eax, DWORD PTR a2
29  mov  BYTE PTR c, %al
30  mov  DWORD PTR d+8, 7
31  movsx %eax, BYTE PTR c
32  mov  DWORD PTR [%esp+4], %eax
33  mov  DWORD PTR [%esp], OFFSET FLAT
34  call printf

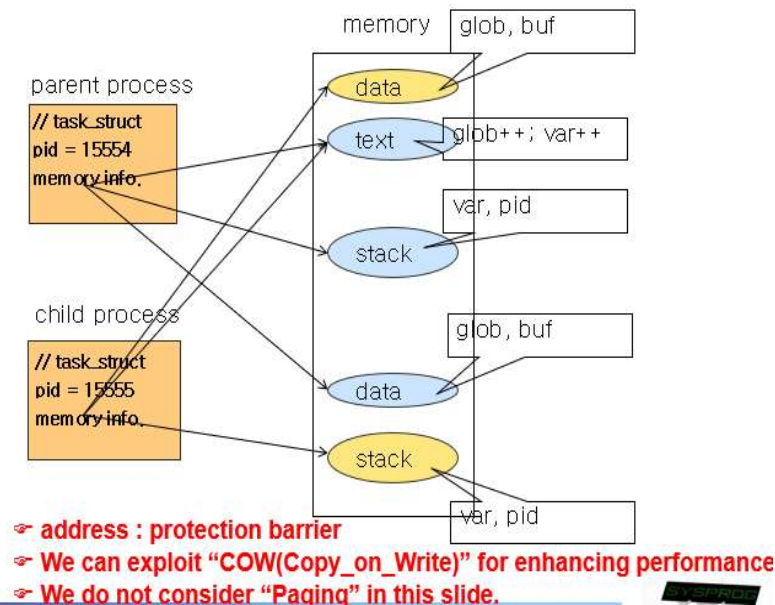
addl  $15, %eax
shrl  $4, %eax
sall  $4, %eax
subl  %eax, %esp
movl  $64, a1
movl  a1, %eax
incl  %eax
movl  %eax, a2
movl  a1, %eax
movw  %ax, b
movl  a2, %eax
movb  %al, c
movl  $7, d+8
subl  $8, %esp
movsbl c, %eax
pushl %eax
pushl $.LC0
call  printf
addl  $16, %esp
```



# Quiz for 9<sup>th</sup>-Week 2<sup>nd</sup>-Lesson

## ■ Quiz

- ✓ 1. There are three memory abstractions in the memory model of IA, namely virtual, linear and physical memory. What does the memory shown in the below left figure correspond to the three memory abstractions? Explain why?
- ✓ 2. Explain the three components of an IA instruction format. What are the differences between “movl \$2, a” and “movl 2, a”?
- ✓ Due: until 6 PM Friday of this week (5<sup>th</sup>, November)



```
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 +
    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
```



# Instruction Detail: Component (5/11)

## Control Transfer Instruction: if

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

```
#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);
}
```

The assembly output on the right is as follows:

```
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   $.LCD
```

Annotations:

- 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%--

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

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

.L3:

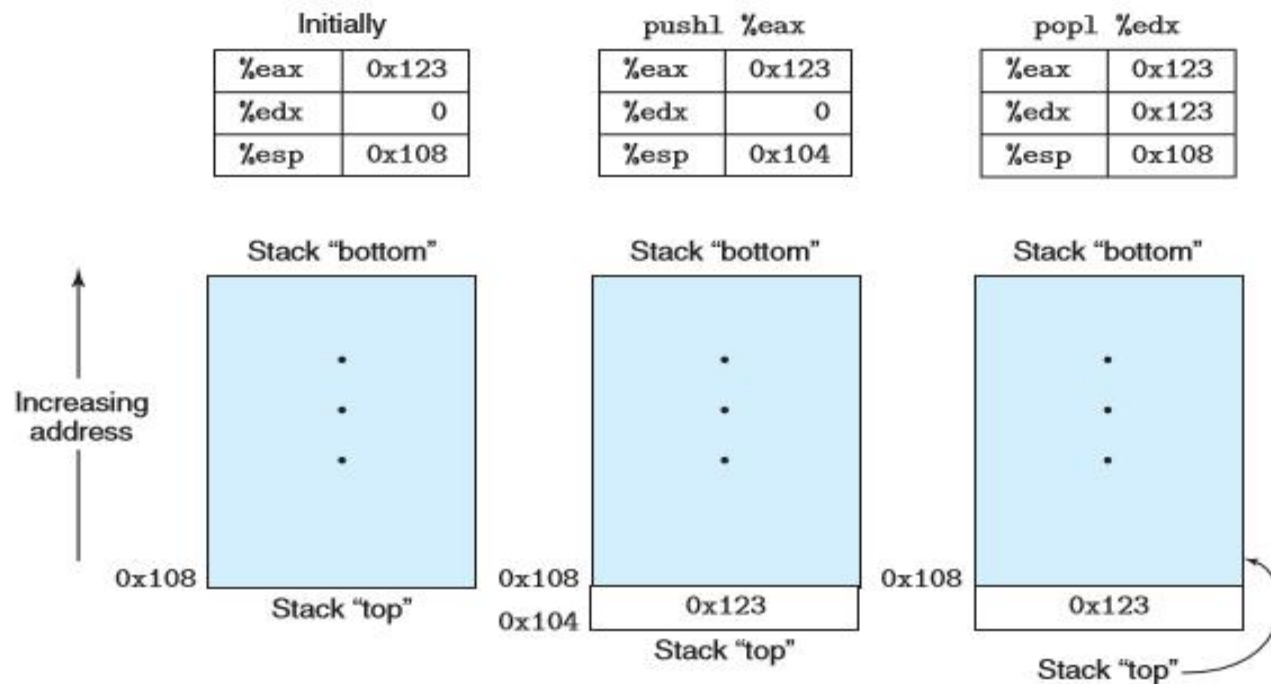
subl    $8, %esp
pushl   a
pushl   $.LC0
call   printf
addl   $16, %esp
leave
ret
.size   main, .-main
.comm   i,4,4
"for_exam.s" [바뀜] 41 줄 --24%--
```

**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)



**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

The image shows two windows from a debugger. The left window displays the C source code for `func1` and `main`. The right window displays the assembly code for `func1` and `main`. A stack frame for `main` is shown on the right, containing the values 222 and 111, and the return address. Two callouts explain the assembly instructions: `movl $222, 4(%esp)` and `call func1`.

```
#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%--
```

```
.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
.LRO:
.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%--
```

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`).**

"func\_exam.s" [바뀜] 50 줄 --74%-- 37,0-1 8%

# Instruction Detail: Component (9/11)

## ■ Function-related Instruction: in function

The image shows a side-by-side comparison of C source code and its assembly representation, along with a stack frame diagram.

**Left Panel (C Source Code):**

```
#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%--
```

**Middle Panel (Assembly Code):**

```
.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
.LRO:
.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%--
```

**Right Panel (Stack Frame Diagram):**

stack frame for main	
222	
111	
ret. address	← EBP
saved ebp	
a	
b	← ESP

**Annotations:**

- Decrease ESP. Put operand on the stack. (cf. movl \$222, 4(%esp))** - points to the `pushl $222` instruction.
- Push EIP. Jump to the operand (EIP = func1).** - points to the `call func1` instruction.

**Bottom Annotation:**

☞ Use relative address based on ebp instead of variable name

# Instruction Detail: Component (10/11)

## Function-related Instruction: after function

The image shows a side-by-side comparison of C source code and its assembly representation, with a stack frame diagram and explanatory callouts.

**Source Code (Left):**

```
#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%--
```

**Assembly Code (Middle):**

```
.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
.LRO:
.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, %ebp
    ret
.size main, .-main
"func_exam.s" [바뀜] 50 줄 --74%--
```

**Stack Frame for main (Right):**

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

**Annotations:**

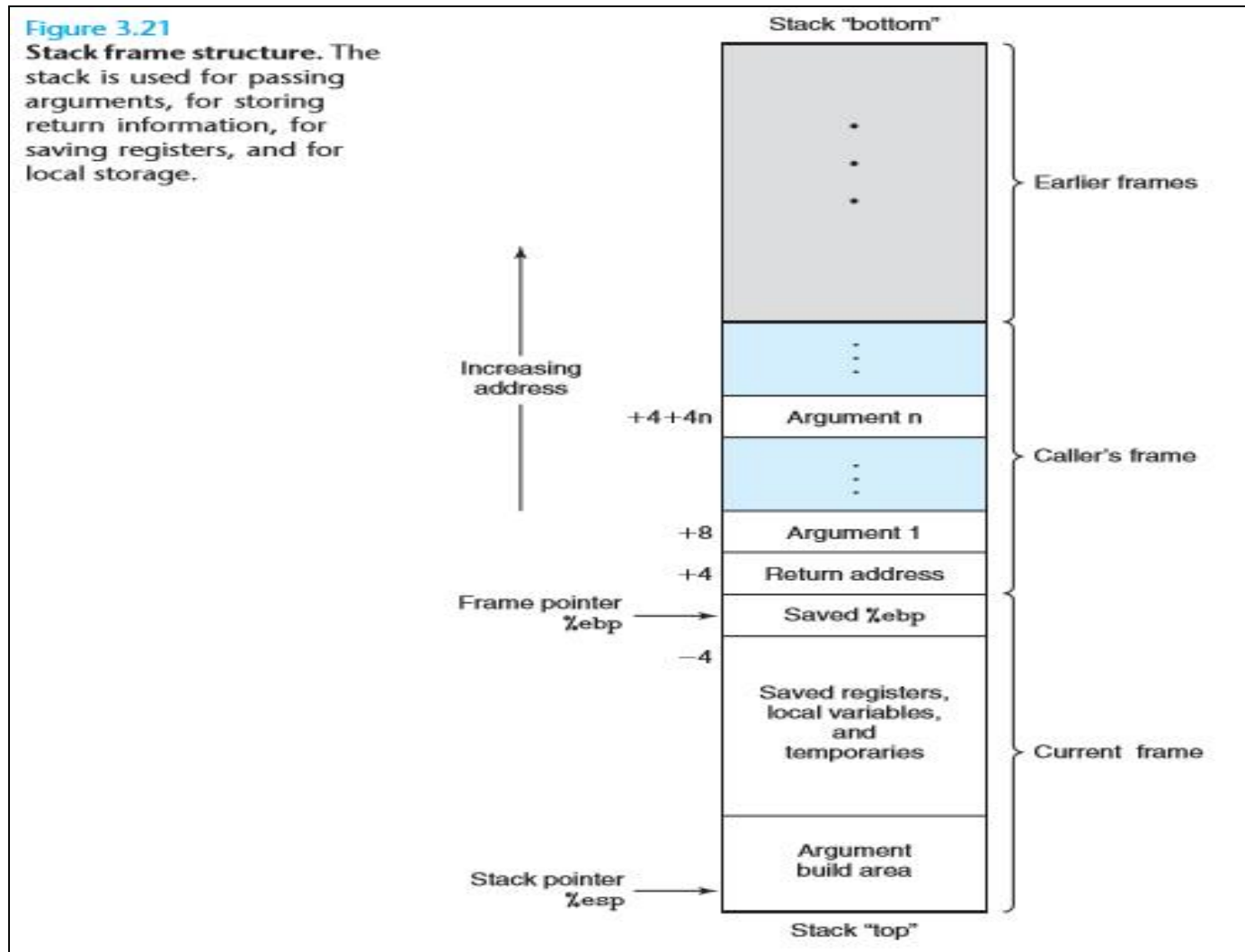
- EBP:** Points to the 'saved ebp' entry in the stack frame.
- ESP:** Points to the 'b' entry in the stack frame.
- ESP = EBP. Then pop. (Eventually pop local variables and saved ebp from the stack)**: Points to the 'leave' instruction in the assembly.
- pop and set it into EIP (EIP = return address)**: Points to the 'ret' instruction in the assembly.
- Decrease ESP. Put operand on the stack. (cf. movl \$222, 4(%esp))**: Points to the 'pushl \$222' instruction in the assembly.
- Push EIP. Jump to the operand (EIP = func1).**: Points to the 'call func1' instruction in the assembly.
- Pop arguments from the stack.**: Points to the 'addl \$8, %esp' instruction in the assembly.
- Return value is in eax**: Points to the 'movl %eax, g' instruction in the assembly.

**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



# 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");
}

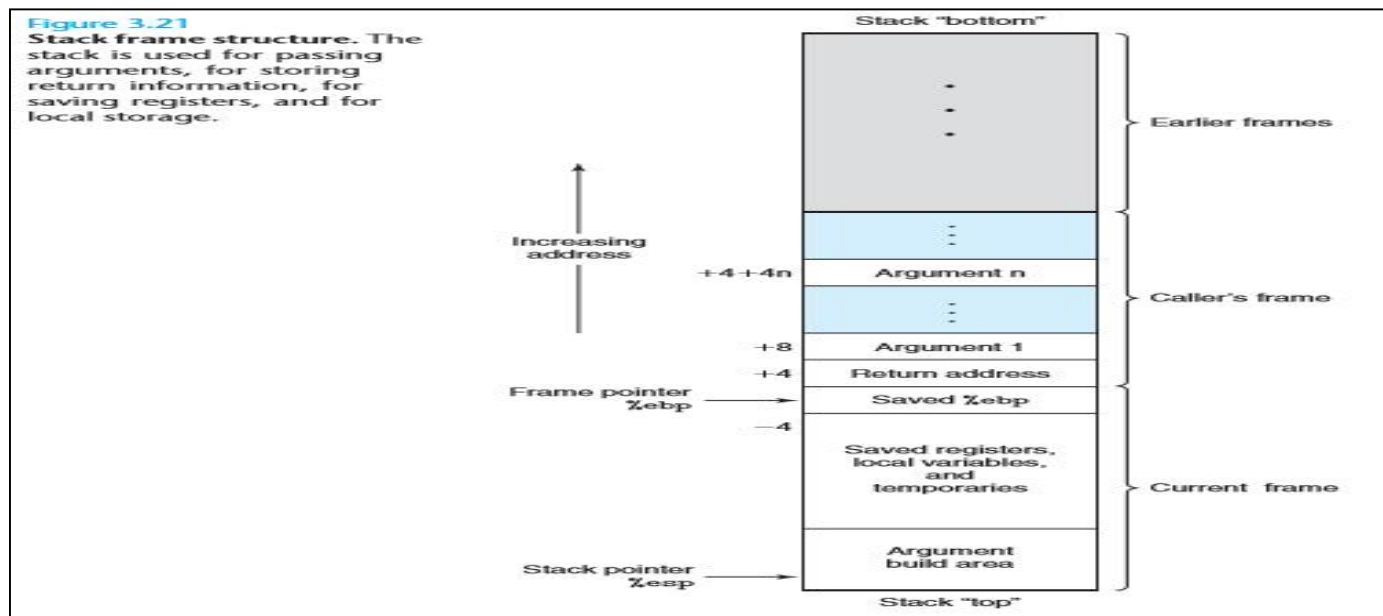
main() {
    f3();
}
```



# Quiz for 10<sup>th</sup>-Week 1<sup>st</sup>-Lesson

## ■ Quiz

- ✓ 1. Explain two ways how the C statement “ $d = b * 7$ ” is translated into assembly language.
- ✓ 2. Describe how the first arguments and the first local variables are accessed in CPU.
- ✓ Due: until 6 PM Friday of this week (12<sup>th</sup>, November)



# 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:

```
1 int accum = 0;
2
3 int sum(int x, int y)
4 {
5     int t = x + y;
6     accum += t;
7     return t;
8 }
```

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
<code>jmp Label</code>		1	Direct jump
<code>jmp *Operand</code>		1	Indirect jump
<code>je Label</code>	<code>jz</code>	ZF	Equal / zero
<code>jne Label</code>	<code>jnz</code>	$\neg$ ZF	Not equal / not zero
<code>js Label</code>		SF	Negative
<code>jns Label</code>		$\neg$ SF	Nonnegative
<code>jg Label</code>	<code>jnl</code>	$\neg$ (SF ^ OF) & $\neg$ ZF	Greater (signed >)
<code>jge Label</code>	<code>jnl</code>	$\neg$ (SF ^ OF)	Greater or equal (signed >=)
<code>jl Label</code>	<code>jnge</code>	SF ^ OF	Less (signed <)
<code>jle Label</code>	<code>jng</code>	(SF ^ OF)   ZF	Less or equal (signed <=)
<code>ja Label</code>	<code>jnb</code>	$\neg$ CF & $\neg$ ZF	Above (unsigned >)
<code>jae Label</code>	<code>jnb</code>	$\neg$ CF	Above or equal (unsigned >=)
<code>jb Label</code>	<code>jnae</code>	CF	Below (unsigned <)
<code>jbe Label</code>	<code>jna</code>	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:

```
1 int dw_loop(int x, int y, int n) {
2     do {
3         x += n;
4         y **= n;
5         n--;
6     } while ((n > 0) && (y < n));
7     return x;
8 }
```

`gcc` generates the following assembly code:

```
    x at %ebp+8, y at %ebp+12, n at %ebp+16
1   movl   8(%ebp), %eax
2   movl   12(%ebp), %ecx
3   movl   16(%ebp), %edx
4   .L2:
5   addl   %edx, %eax
6   imull %edx, %ecx
7   subl   $1, %edx
8   testl %edx, %edx
9   jle   .L5
10  cmpl  %edx, %ecx
11  jl   .L2
12  .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

The image shows two windows from a code editor. The left window displays a C program named `main.c` with the following code:

```
/* 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);
}
```

The right window displays the assembly code for `asm_sum.s`:

```
.text
.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
```

Callouts from the assembly code explain the following:

- .text directive: declare text section (the following instructions are resided in the text section)** - points to the `.text` directive.
- .global directive: declare "asm\_sum" visible to the linker** - points to the `.global asm_sum` directive.
- Memory addressing: displacement(base) or displacement(base, index, scale)** - points to the `8(%ebp), %ecx` instruction.

At the bottom of the windows, the status bars show: `"main.c" 15 줄 --100%--` and `"asm_sum.s" 22 줄 --81%--`. The bottom right corner features a logo for `SYSPROG`.

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

## ■ Execution results of Practice 1

```
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  --100%--

choijm@localhost:~/syspro_examples/chap6
[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]$ 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

The image displays two terminal windows side-by-side. The left window shows the assembly source code for a program named 'asm\_sum\_standalone.s'. The code is annotated with red boxes and arrows pointing to explanatory text boxes. The right window shows the execution of the program, including compilation with 'gcc' and the resulting output.

```
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
~
```

Annotations:

- .data directive: declare data section** (points to the .data section)
- .long directive: initialize 4B memory space (address, initial value, expression, ...)** (points to the .long directive)
- .string directive: initialize string (array of character)** (points to the .string directive)

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



# 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

The image shows two windows from a terminal or IDE. The left window displays the assembly code for a program named `asm_swint.s`. The code includes data sections for a buffer and a printf argument, and a `main` function. The `main` function sets up registers for a `write` system call and then prints the result.

```
/* 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%--
```

The right window shows the execution of the program. It lists the files, compiles the assembly, and runs the resulting executable, showing the output: `Hello world` and `Result = 12`.

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

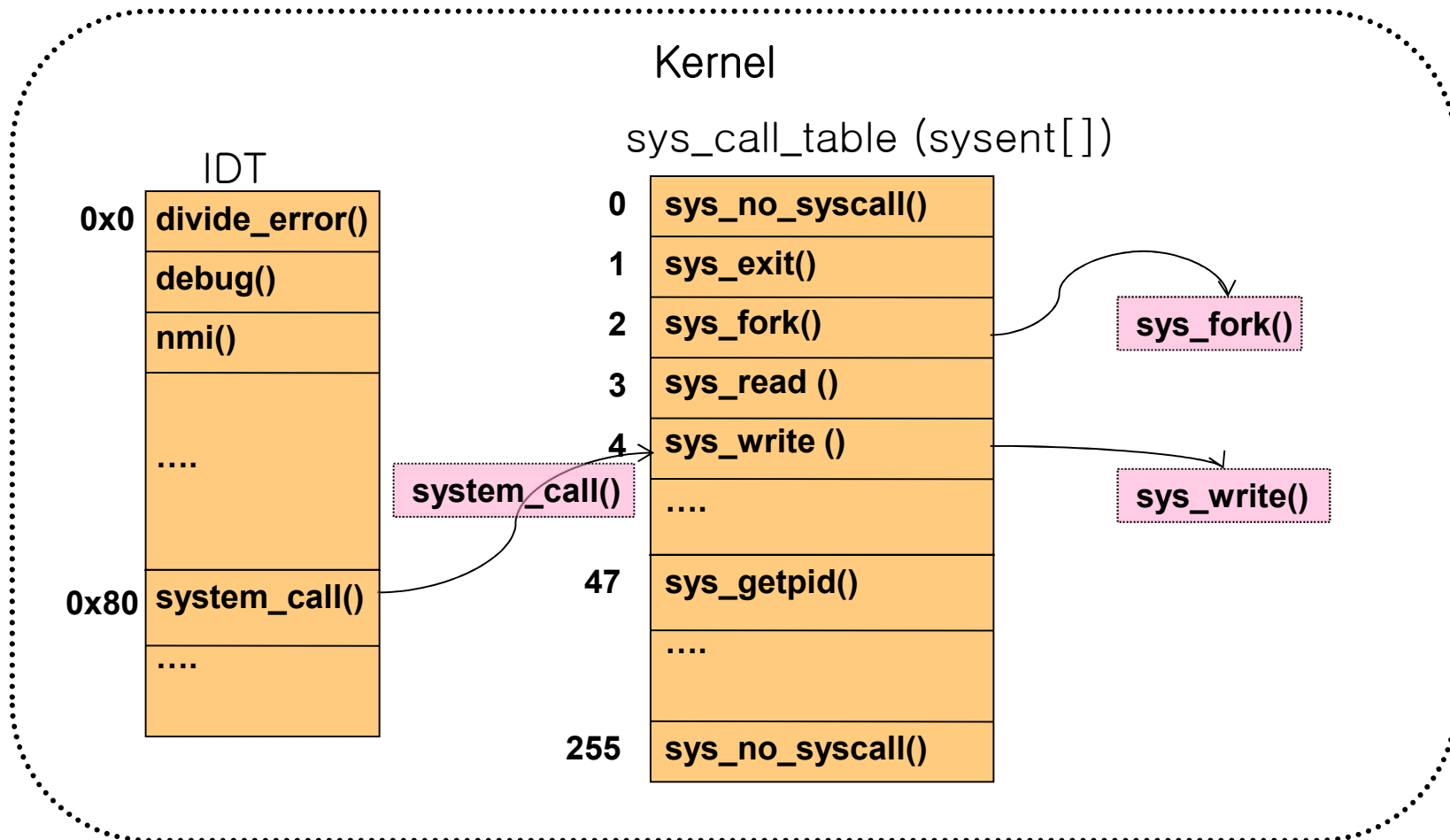
Annotations with callouts point to specific parts of the assembly code:

- system call arguments**: Points to the `movl $1, %ebx`, `movl $W_buf, %ecx`, and `movl W_size, %edx` instructions.
- system call index**: Points to the `movl $4, %eax` instruction.
- IDT table index**: Points to the `int $0x80` instruction.



# 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 50 (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!!, 19<sup>th</sup> November)
  - 2) Send the report and source code to TA (이제연: 2reenact@naver.com)
- ✓ Warn: DO NOT utilize “gcc -S option” (easily detected)



# Quiz for 10<sup>th</sup>-Week 2<sup>nd</sup>-Lesson

## ■ Quiz

- ✓ 1. Discuss the differences between function call and system call (e.g. printf() vs. write()), at least three).
- ✓ 2. There are various optimization options in gcc such as “O0, O1, O2, O3 and Os”. What if we create an assembly program using O3 when we create the move\_exam.s?
- ✓ 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 (12<sup>th</sup>, November)

gcc	O0	O1	O2	O3
Basic	1.535250	0.740309	0.750498	0.744492
Branch	4.331550	0.745291	0.738352	0.745730
Infinite for	1.756510	0.743746	0.752588	0.741769
Infinite while	1.743800	0.740280	0.751422	0.739939
Nested for	1.595010	0.623453	0.440711	0.437040
Induction variable	5.004260	0.203298	0.203790	0.192637
Loop inversion	6.033830	0.182433	0.187445	0.187577

```
choijm@embedded: ~/Syspro/chap6
choijm@embedded:~/Syspro/chap6$ gcc -v
Reading specs from /usr/lib/gcc/1486-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-libs --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:
    .type main, @function
    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
    movl   a, %eax
    movl   %eax, b
    movl   b, %eax
    addl   a, %eax
    movl   %eax, c
    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
```

(Source: [http://www.cs.columbia.edu/~ecj2122/research/x86\\_loop\\_optimizations/x86\\_loop\\_optimizations.pdf](http://www.cs.columbia.edu/~ecj2122/research/x86_loop_optimizations/x86_loop_optimizations.pdf))



# Appendix 1: 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

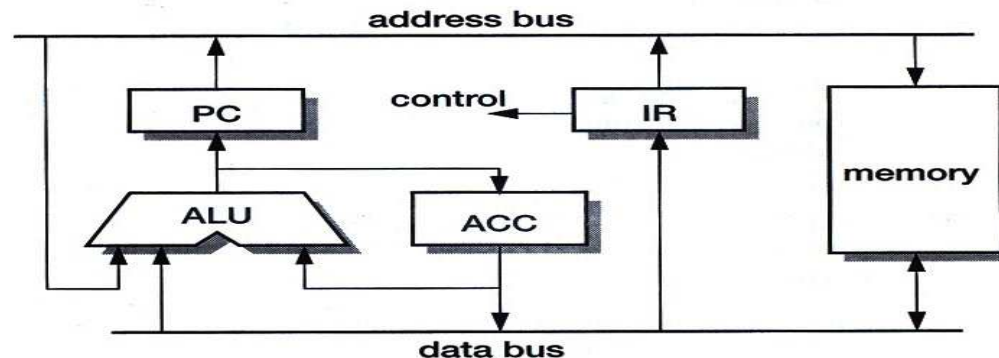


Figure 1.5 MU0 datapath example.

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



# Appendix 1: MU0, A Simple CPU

## ■ Data Transfer

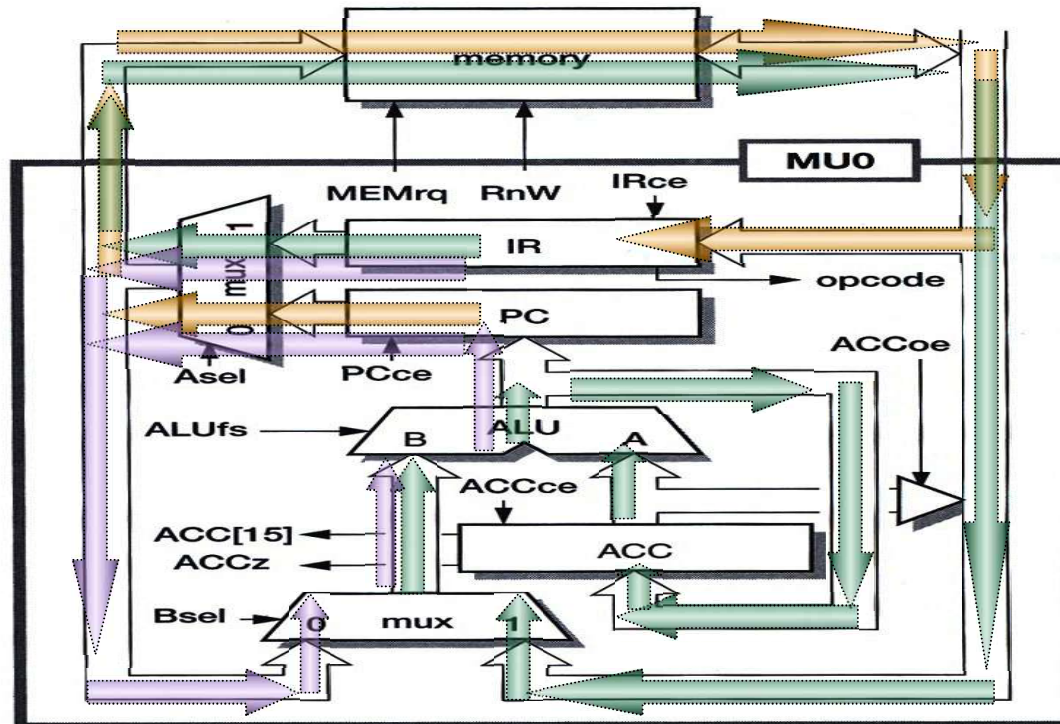


Figure 1.6 MU0 register transfer level organization.

- ✓ 1) fetch, 2) execution, 3) flow control



# Appendix 1: 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.

<b>Instruction</b>	<b>Opcode</b>	<b>Effect</b>
LDA S	0000	ACC := mem <sub>16</sub> [S]
STO S	0001	mem <sub>16</sub> [S] := ACC
ADD S	0010	ACC := ACC + mem <sub>16</sub> [S]
SUB S	0011	ACC := ACC - mem <sub>16</sub> [S]
JMP S	0100	PC := S
JGE S	0101	if ACC ≥ 0 PC := S
JNE S	0110	if ACC ≠ 0 PC := S
STP	0111	stop



# Appendix 1: MU0, A Simple CPU

## ■ Control Logic

Table 1.2 MU0 control logic.

Instruction	Inputs					Outputs									
	Opcode	Ex/ft	ACC15			Bsel		PCce		ACCoe		MEMrq		Ex/ft	
	↓	↓	↓	↓	↓	Asel	↓	ACCce	↓	IRce	↓	ALUfs	↓	RnW	↓
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





# Appendix 1: 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

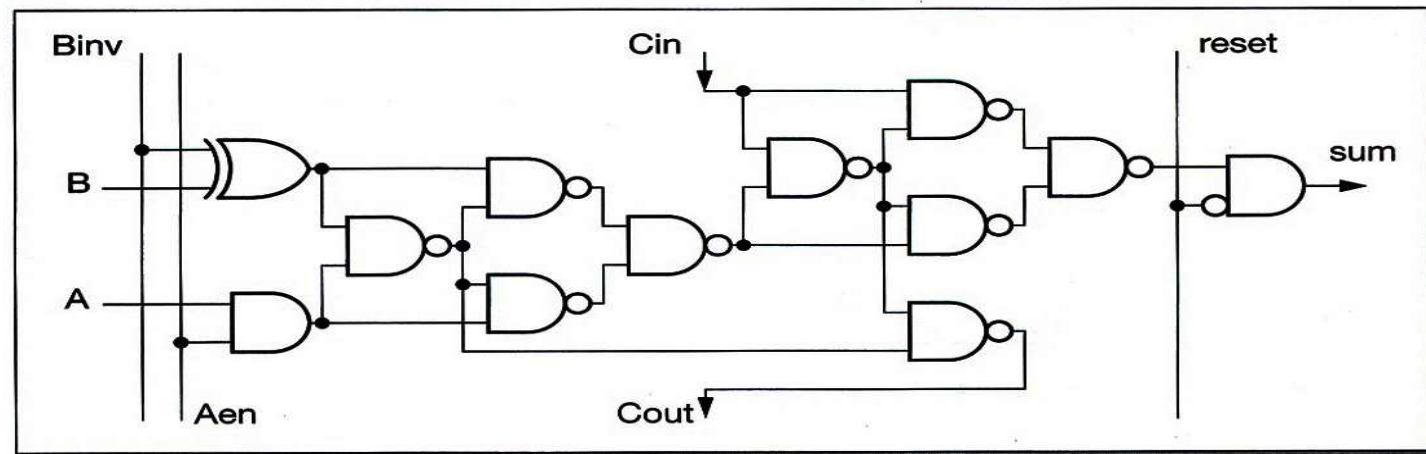


Figure 1.7 MU0 ALU logic for one bit.



# 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
- ✓ ...

