Lecture Note 7.
IA: History and Features

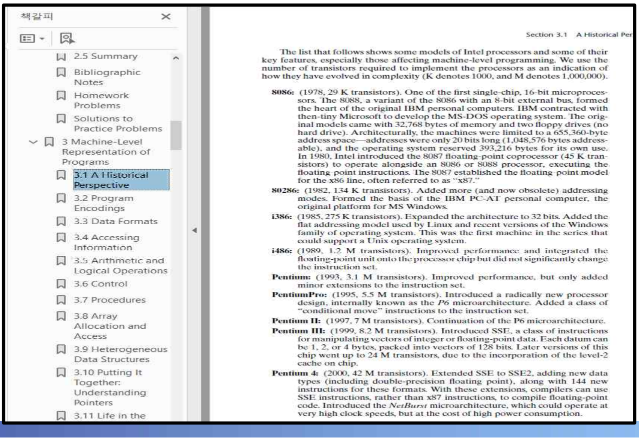
October 30, 2020

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

Objectives

- Discuss Issues on ISA (Instruction Set Architecture)
  - ✓ Opcode and operand addressing modes
- Apprehend how ISA affects system program
  - ✓ Context switch, memory alignment, stack overflow protection
- Describe the history of IA (Intel Architecture)
- Grasp the key technologies in recent IA
  - ✓ Pipeline and Moore's law

- Refer to Chapter 3, 4 in the CSAPP and Intel SW Developer Manual



책갈피

The list that follows shows some models of Intel processors and some of their key features, especially those affecting machine-level programming. We use the number of transistors required to implement the processors as an indication of how they have evolved in complexity (K denotes 1000, and M denotes 1,000,000).

8086: (1978, 29 K transistors). One of the first single-chip, 16-bit microprocessors. The 8088, a variant of the 8086 with an 8-bit external bus, formed the heart of the original IBM personal computers. IBM contracted with then-tiny Microsoft to develop the MS-DOS operating system. The original models came with 32,768 bytes of memory and two floppy drives (no hard drive). Architecturally, the machines were limited to a 655,360-byte address space—addresses were only 20 bits long (1,048,576 bytes addressable), and the operating system reserved 393,216 bytes for its own use. In 1980, Intel introduced the 8087 floating-point coprocessor (45 K transistors) to operate alongside an 8086 or 8088 processor, executing the floating-point instructions. The 8087 established the floating-point model for the x86 line, often referred to as "x87."

80286: (1982, 134 K transistors). Added more (and now obsolete) addressing modes. Formed the basis of the IBM PC-AT personal computer, the original platform for MS Windows.

i386: (1985, 275 K transistors). Expanded the architecture to 32 bits. Added the flat addressing model used by Linux and recent versions of the Windows family of operating system. This was the first machine in the series that could support a Unix operating system.

i486: (1989, 1.2 M transistors). Improved performance and integrated the floating-point unit onto the processor chip but did not significantly change the instruction set.

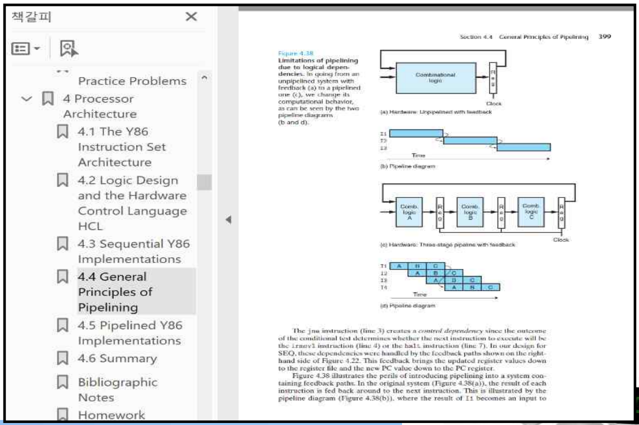
Pentium: (1993, 3.1 M transistors). Improved performance, but only added minor extensions to the instruction set.

PentiumPro: (1995, 5.5 M transistors). Introduced a radically new processor design, internally known as the *P6* microarchitecture. Added a class of "conditional move" instructions to the instruction set.

Pentium II: (1997, 7 M transistors). Continuation of the P6 microarchitecture.

Pentium III: (1999, 8.2 M transistors). Introduced SSE, a class of instructions for manipulating vectors of integer or floating-point data. Each datum can be 1, 2, or 4 bytes, packed into vectors of 128 bits. Later versions of this chip went up to 24 M transistors, due to the incorporation of the level-2 cache on chip.

Pentium 4: (2000, 42 M transistors). Extended SSE to SSE2, adding new data types (including double-precision floating point), along with 144 new instructions for these formats. With these extensions, compilers can use SSE instructions, rather than x87 instructions, to compile floating-point code. Introduced the *NetBurst* microarchitecture, which could operate at very high clock speeds, but at the cost of high power consumption.



책갈피

Figure 4.38
Limitations of pipelining due to logical dependencies. In going from an unpipelined system with feedback (a) to a pipelined one (c), we change its computational behavior, as can be seen by the two pipeline diagrams (b and d).

The jne instruction (line 3) creates a *control dependency* since the outcome of the conditional test determines whether the next instruction to execute will be the irmovl instruction (line 4) or the halt instruction (line 7). In our design for SEQ, these dependencies were handled by the feedback paths shown on the right-hand side of Figure 4.22. This feedback brings the updated register values down to the register file and the new PC value down to the PC register.

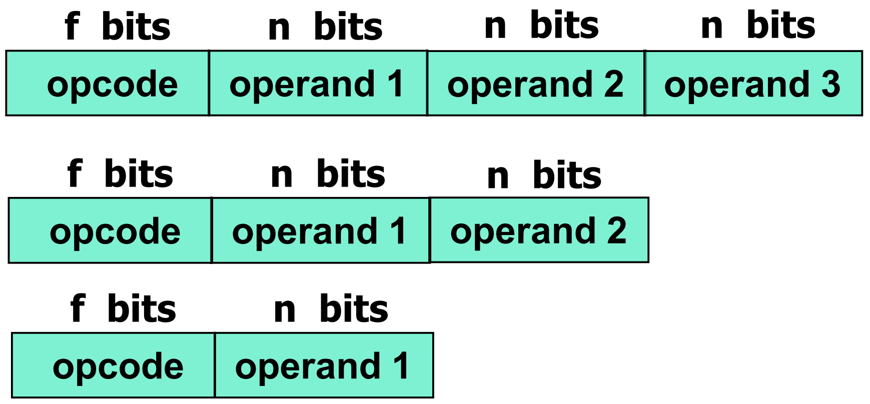
Figure 4.38 illustrates the perils of introducing pipelining into a system containing feedback paths. In the original system (Figure 4.38(a)), the result of each instruction is fed back around to the next instruction. This is illustrated by the pipeline diagram (Figure 4.38(b)), where the result of I1 becomes an input to

Issues on ISA (1/2)

- Consideration on ISA (Instruction Set Architecture)

```
asm_sum:    addl    $1, %ecx
            movl    -4(%ebx, %ebp, 4), %eax
            call    func1
            leave
```

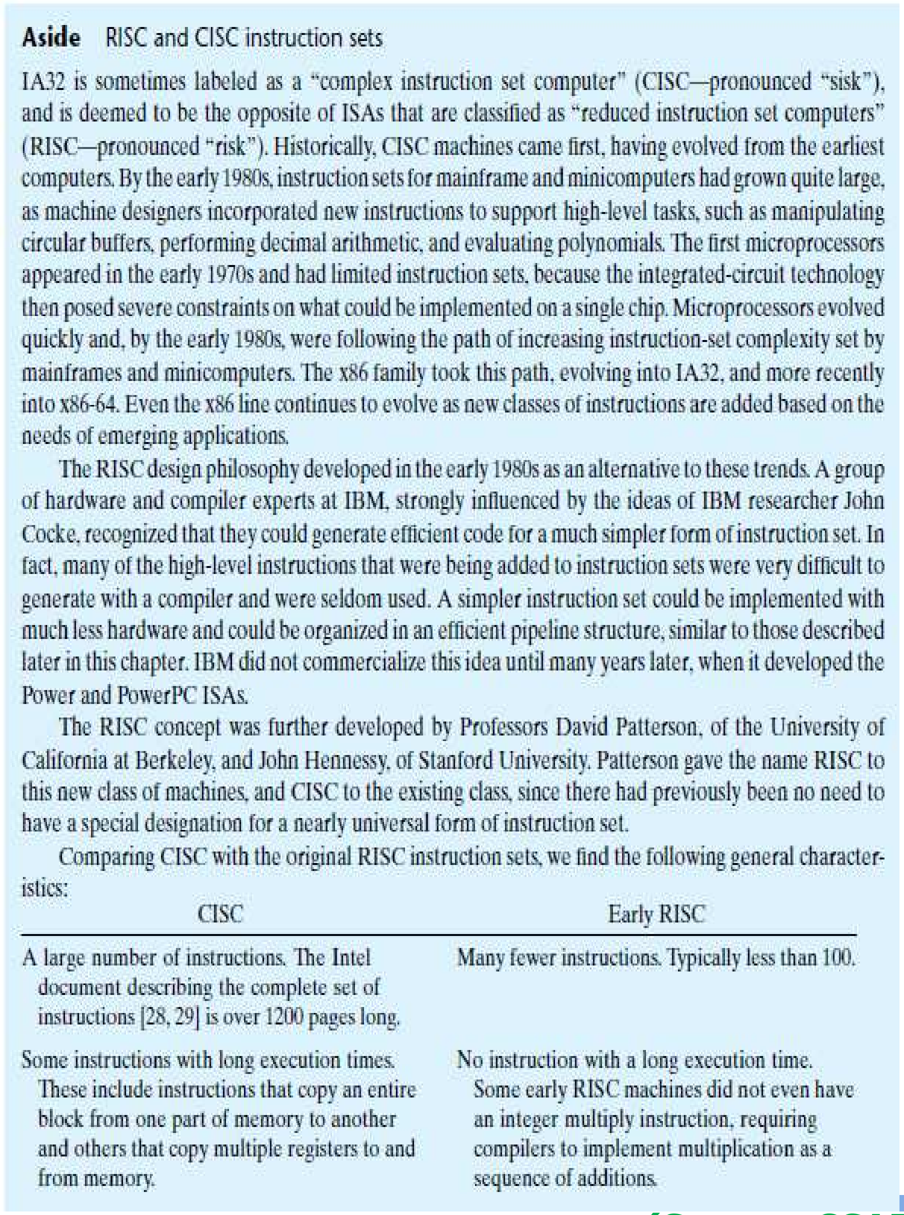
- ✓ opcode issues
  - how many? (add vs. inc ➔ RISC vs. CISC)
  - multi functions? (SISD vs. SIMD vs. MIMD …)
- ✓ operand issues
  - fixed vs. variable operands
  - fixed: how many?
  - operand addressing modes
- ✓ performance issues
  - pipeline
  - superscalar
  - multicore



Issues on ISA (2/2)

- Features of IA (Intel Architecture)
  - ✓ Basically CISC (Complex Instruction Set Computing)
    - Variable length instruction
    - Variable number of operands (0~3)
    - Diverse operand addressing modes
    - Stack based function call
    - Supporting SIMD (Single Instruction Multiple Data)
  - ✓ Try to take advantage of RISC (Reduced Instruction Set Computing)
    - Micro-operations (for instance, an instruction of "add %eax, a" is divided into three u-ops, and each u-op is executed in a pipeline manner)
    - Load-store architecture
    - Independent multi-units
    - Out-of-order execution
    - Register based function call on x64
    - Register renaming
    - …

RISC and CISC summary



Aside RISC and CISC instruction sets

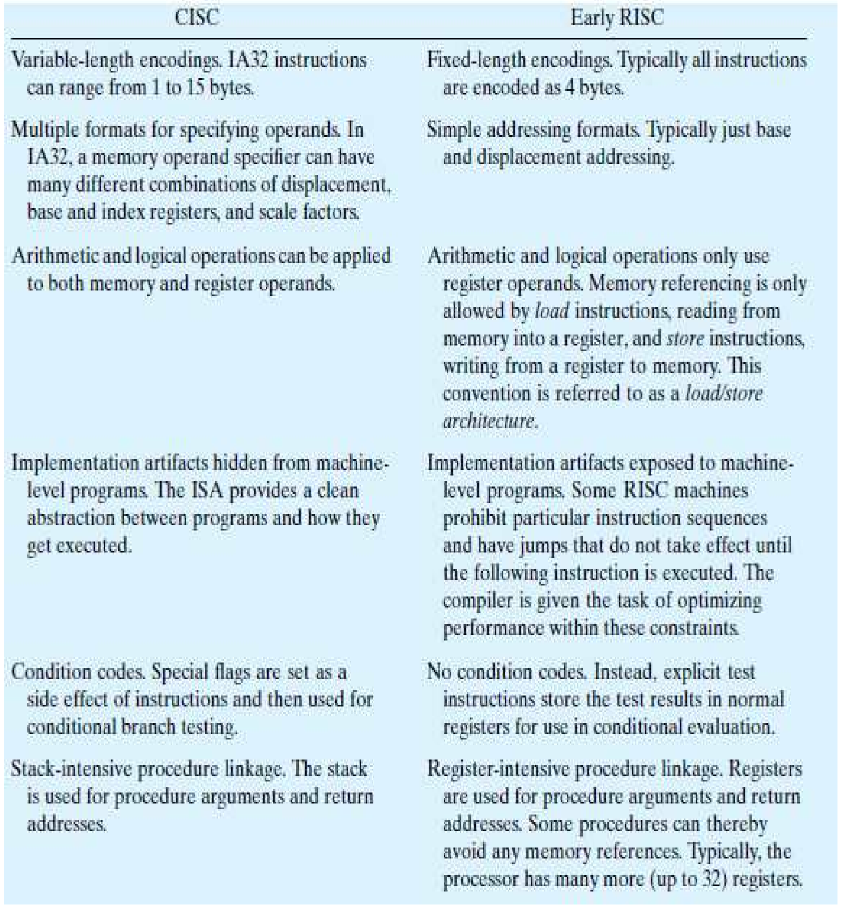
IA32 is sometimes labeled as a "complex instruction set computer" (CISC—pronounced "sisk"), and is deemed to be the opposite of ISAs that are classified as "reduced instruction set computers" (RISC—pronounced "risk"). Historically, CISC machines came first, having evolved from the earliest computers. By the early 1980s, instruction sets for mainframe and minicomputers had grown quite large, as machine designers incorporated new instructions to support high-level tasks, such as manipulating circular buffers, performing decimal arithmetic, and evaluating polynomials. The first microprocessors appeared in the early 1970s and had limited instruction sets, because the integrated-circuit technology then posed severe constraints on what could be implemented on a single chip. Microprocessors evolved quickly and, by the early 1980s, were following the path of increasing instruction-set complexity set by mainframes and minicomputers. The x86 family took this path, evolving into IA32, and more recently into x86-64. Even the x86 line continues to evolve as new classes of instructions are added based on the needs of emerging applications.

The RISC design philosophy developed in the early 1980s as an alternative to these trends. A group of hardware and compiler experts at IBM, strongly influenced by the ideas of IBM researcher John Cocke, recognized that they could generate efficient code for a much simpler form of instruction set. In fact, many of the high-level instructions that were being added to instruction sets were very difficult to generate with a compiler and were seldom used. A simpler instruction set could be implemented with much less hardware and could be organized in an efficient pipeline structure, similar to those described later in this chapter. IBM did not commercialize this idea until many years later, when it developed the Power and PowerPC ISAs.

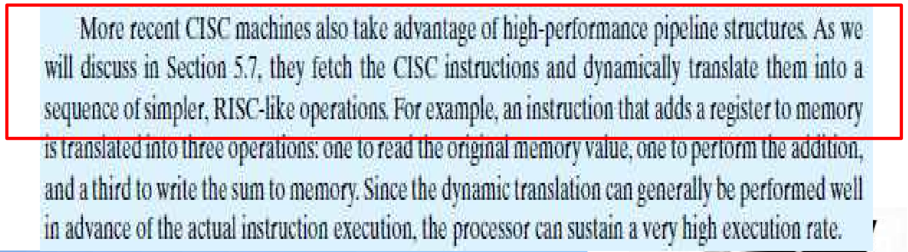
The RISC concept was further developed by Professors David Patterson, of the University of California at Berkeley, and John Hennessy, of Stanford University. Patterson gave the name RISC to this new class of machines, and CISC to the existing class, since there had previously been no need to have a special designation for a nearly universal form of instruction set.

Comparing CISC with the original RISC instruction sets, we find the following general characteristics:

CISC	Early RISC
A large number of instructions. The Intel document describing the complete set of instructions [28, 29] is over 1200 pages long.	Many fewer instructions. Typically less than 100.
Some instructions with long execution times. These include instructions that copy an entire block from one part of memory to another and others that copy multiple registers to and from memory.	No instruction with a long execution time. Some early RISC machines did not even have an integer multiply instruction, requiring compilers to implement multiplication as a sequence of additions.



CISC	Early RISC
Variable-length encodings. IA32 instructions can range from 1 to 15 bytes.	Fixed-length encodings. Typically all instructions are encoded as 4 bytes.
Multiple formats for specifying operands. In IA32, a memory operand specifier can have many different combinations of displacement, base and index registers, and scale factors.	Simple addressing formats. Typically just base and displacement addressing.
Arithmetic and logical operations can be applied to both memory and register operands.	Arithmetic and logical operations only use register operands. Memory referencing is only allowed by *load* instructions, reading from memory into a register, and *store* instructions, writing from a register to memory. This convention is referred to as a *load/store architecture*.
Implementation artifacts hidden from machine-level programs. The ISA provides a clean abstraction between programs and how they get executed.	Implementation artifacts exposed to machine-level programs. Some RISC machines prohibit particular instruction sequences and have jumps that do not take effect until the following instruction is executed. The compiler is given the task of optimizing performance within these constraints.
Condition codes. Special flags are set as a side effect of instructions and then used for conditional branch testing.	No condition codes. Instead, explicit test instructions store the test results in normal registers for use in conditional evaluation.
Stack-intensive procedure linkage. The stack is used for procedure arguments and return addresses.	Register-intensive procedure linkage. Registers are used for procedure arguments and return addresses. Some procedures can thereby avoid any memory references. Typically, the processor has many more (up to 32) registers.



More recent CISC machines also take advantage of high-performance pipeline structures. As we will discuss in Section 5.7, they fetch the CISC instructions and dynamically translate them into a sequence of simpler, RISC-like operations. For example, an instruction that adds a register to memory is translated into three operations: one to read the original memory value, one to perform the addition, and a third to write the sum to memory. Since the dynamic translation can generally be performed well in advance of the actual instruction execution, the processor can sustain a very high execution rate.

(Source: CSAPP Chapter 4)

Operand addressing modes (1/5)

- Addressing modes
  - ✓ Immediate addressing

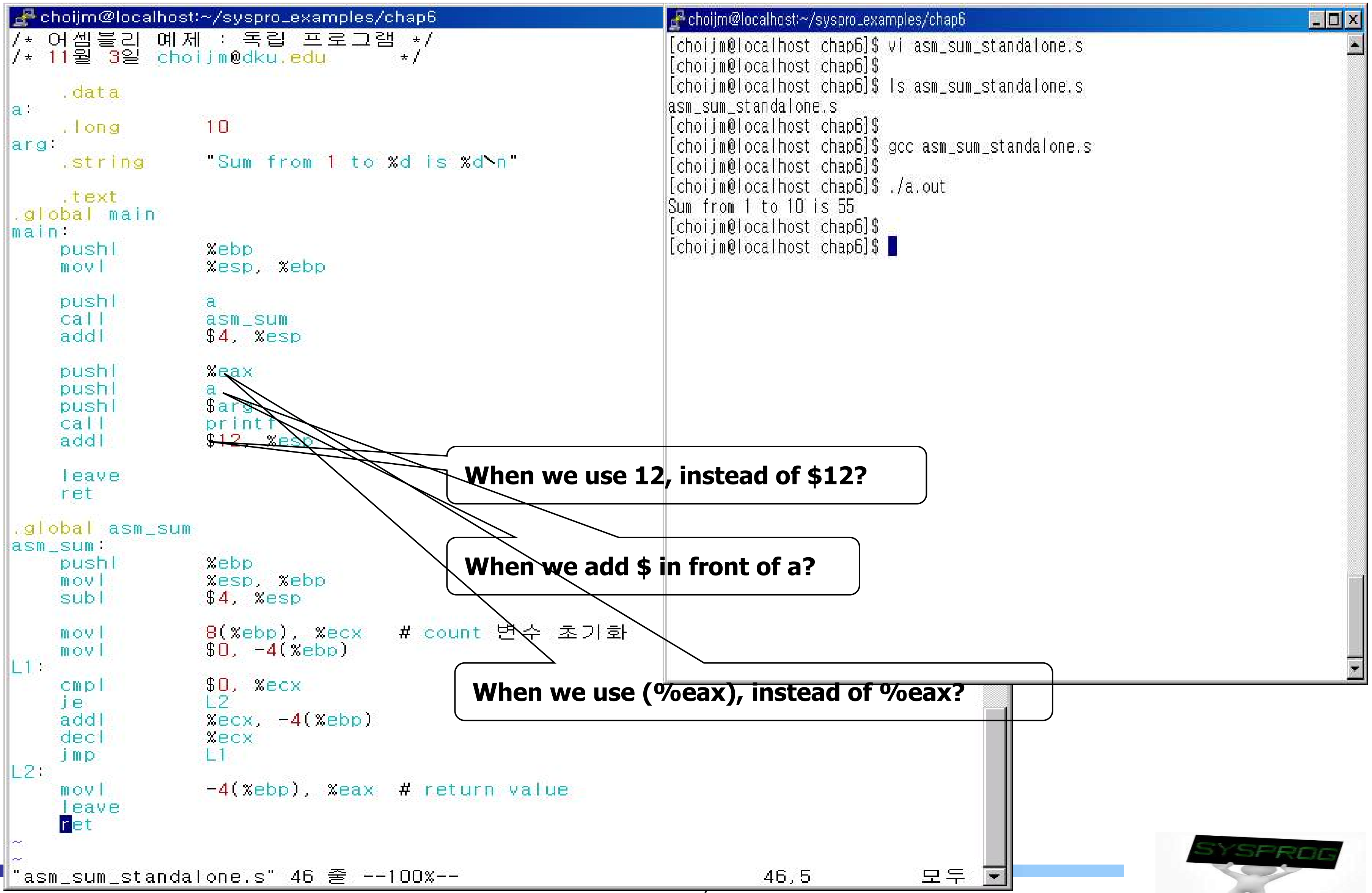
  - ✓ Register addressing
  - ✓ Register Indirect addressing

  - ✓ Direct (Absolute) addressing
  - ✓ Indirect addressing

  - ✓ Base plus Offset addressing
  - ✓ Base plus Index addressing
  - ✓ Base plus Scaled Index addressing
  - ✓ Base plus Scaled Index plus Offset addressing
  - ✓ Stack addressing

Operand addressing modes (2/5)

- Subtle differences in operand



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]$
```

When we use 12, instead of $12?

When we add $ in front of a?

When we use (%eax), instead of %eax?

Operand addressing modes (3/5)

- Operand Addressing in IA
  - ✓ immediate operand

```
addl    $0x12, %eax
```

  - ✓ register operand

```
addl    %esp, %ebp
```

  - ✓ Memory operand
    - direct addressing

```
addl    0x8049384, %eax
```

    - register indirect addressing

```
addl    (%ebp), %eax
```

    - Base plus offset addressing

```
addl    4(%ebp), %eax
```

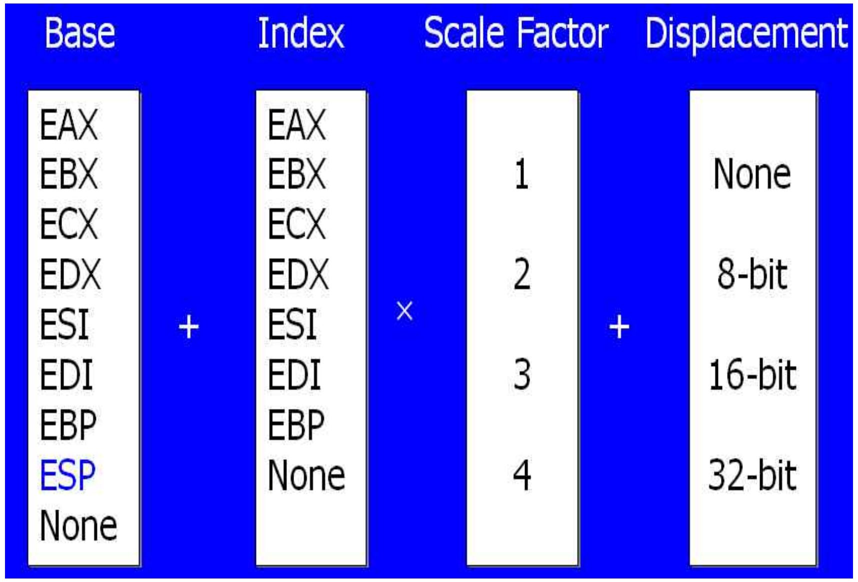
    - Base plus Scaled index plus offset addressing

```
addl    4(%ebp, %eax, 4), %ebx
```

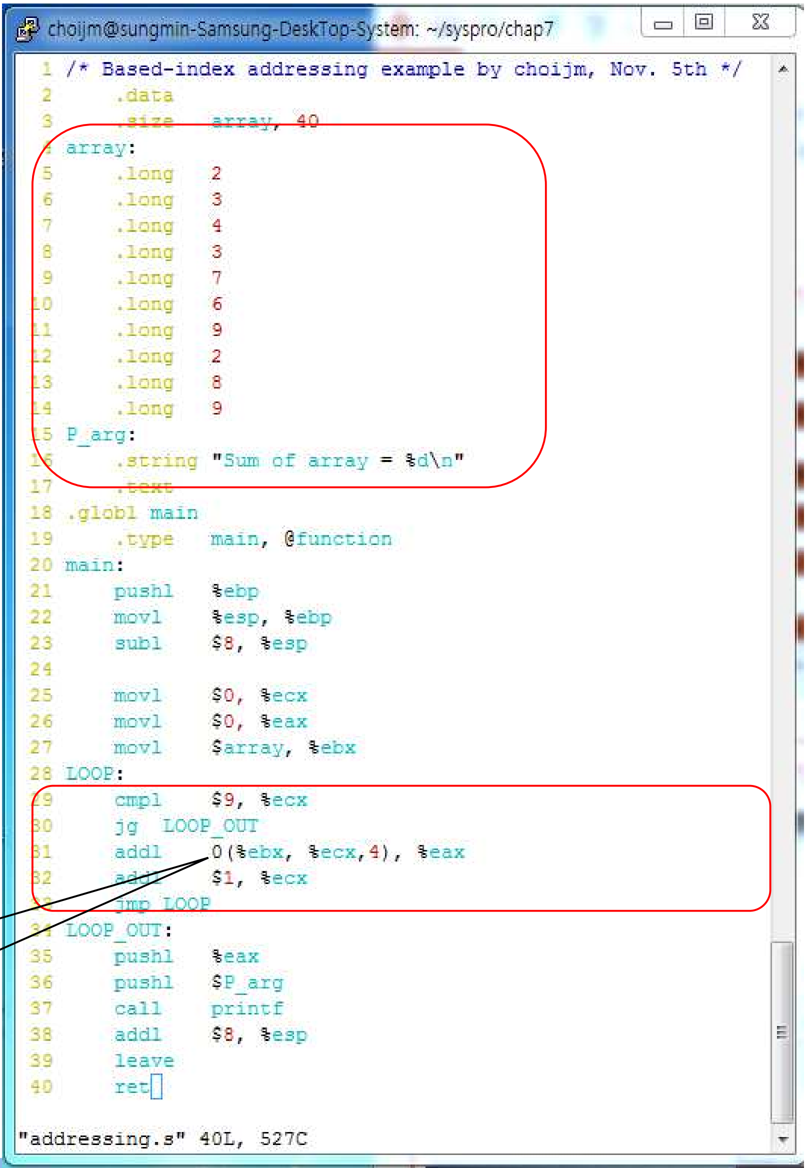
displacement(base, index, scale)

Operand addressing modes (4/5)

- Example
  - ✓ Base plus Scaled index plus offset



Base		Index		Scale Factor		Displacement
EAX		EAX				
EBX		EBX		1		None
ECX		ECX				
EDX		EDX		2		8-bit
ESI	+	ESI	×		+	
EDI		EDI		3		16-bit
EBP		EBP				
ESP		None		4		32-bit
None						



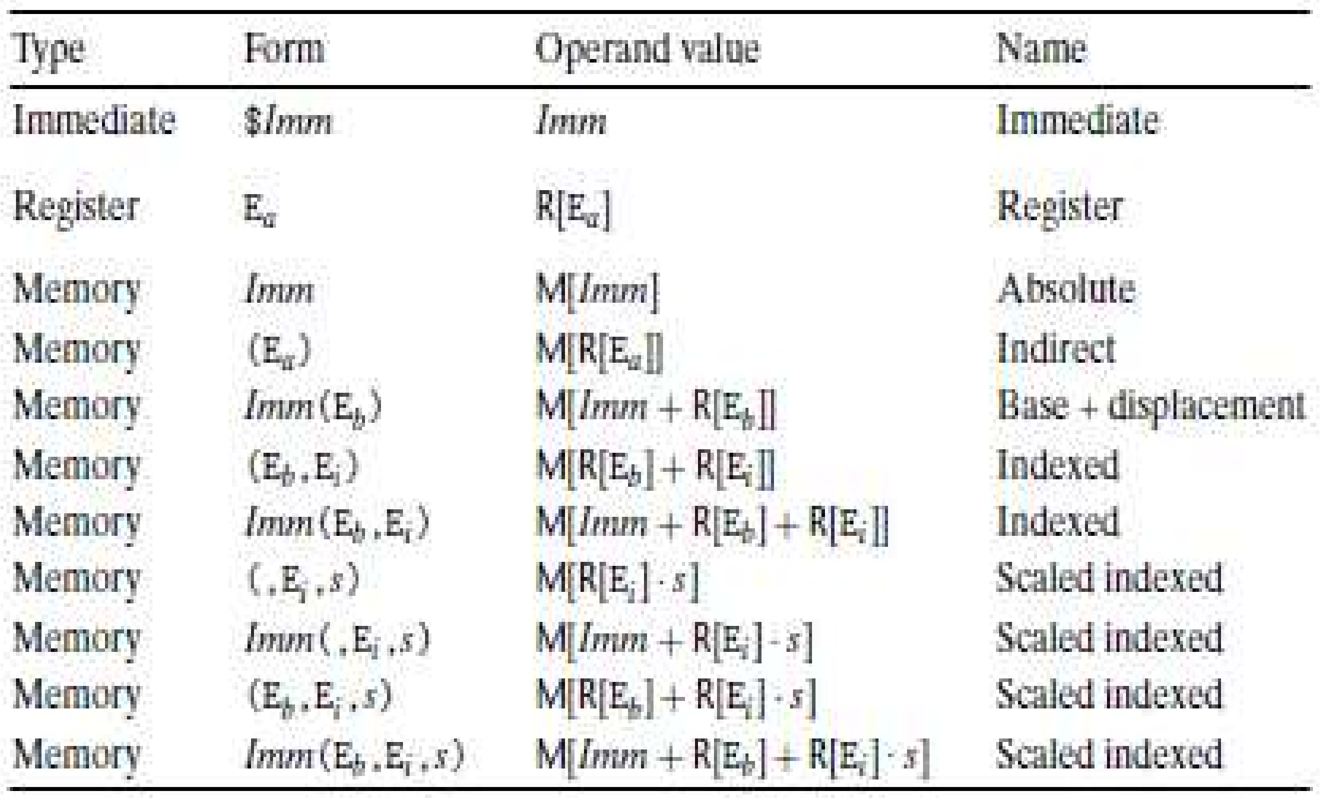
choijm@sungmin-Samsung-DeskTop-System: ~/syspro/chap7

```
/* Based-index addressing example by choijm, Nov. 5th */
    .data
    .size   array, 40
array:
    .long   2
    .long   3
    .long   4
    .long   3
    .long   7
    .long   6
    .long   9
    .long   2
    .long   8
    .long   9
P_arg:
    .string "Sum of array = %d\n"
    .text
.globl main
    .type   main, @function
main:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $8, %esp

    movl    $0, %ecx
    movl    $0, %eax
    movl    $array, %ebx
LOOP:
    cmpl    $9, %ecx
    jg  LOOP_OUT
    addl    0(%ebx, %ecx,4), %eax
    addl    $1, %ecx
    jmp LOOP
LOOP_OUT:
    pushl   %eax
    pushl   $P_arg
    call    printf
    addl    $8, %esp
    leave
    ret
"addressing.s" 40L, 527C
```

if 4(%ebx, %ecx, 4) ?

Operand addressing modes (5/5)

- Summary



Type	Form	Operand value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	E_a	$R[E_a]$	Register
Memory	Imm	$M[Imm]$	Absolute
Memory	(E_a)	$M[R[E_a]]$	Indirect
Memory	$Imm(E_b)$	$M[Imm + R[E_b]]$	Base + displacement
Memory	(E_b, E_i)	$M[R[E_b] + R[E_i]]$	Indexed
Memory	$Imm(E_b, E_i)$	$M[Imm + R[E_b] + R[E_i]]$	Indexed
Memory	$(, E_i, s)$	$M[R[E_i] \cdot s]$	Scaled indexed
Memory	$Imm(, E_i, s)$	$M[Imm + R[E_i] \cdot s]$	Scaled indexed
Memory	(E_b, E_i, s)	$M[R[E_b] + R[E_i] \cdot s]$	Scaled indexed
Memory	$Imm(E_b, E_i, s)$	$M[Imm + R[E_b] + R[E_i] \cdot s]$	Scaled indexed

Figure 3.3 **Operand forms.** Operands can denote immediate (constant) values, register values, or values from memory. The scaling factor s must be either 1, 2, 4, or 8.

(Source: CSAPP Chapter 3)

Impact of ISA on system program: Multitasking (1/5)

- Time sharing system
  - ✓ Tasks run interchangeable
  - ✓ Need to remember where to start ➔ Context
    - Context: registers, address space, opened files, IPCs, …
  - ✓ Context switch
    - When: timeout(time quantum expired), sleep, blocking I/O, …
    - How
      - Context save: CPU registers ➔ task structure (memory)
      - Context restore: task structure (memory) ➔ CPU registers

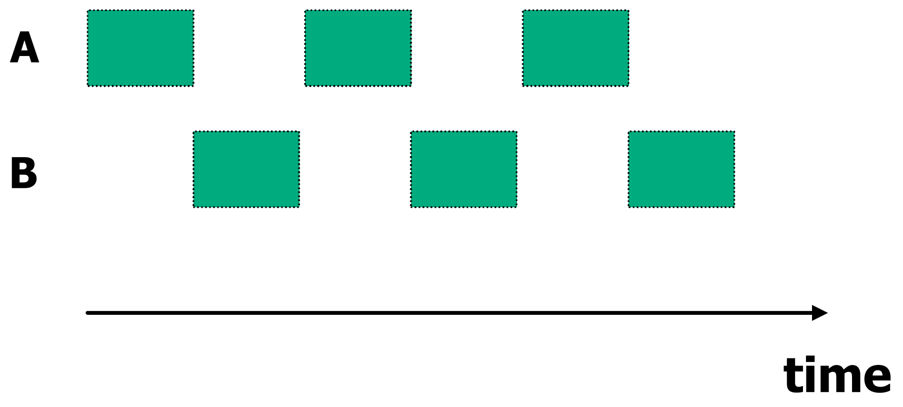
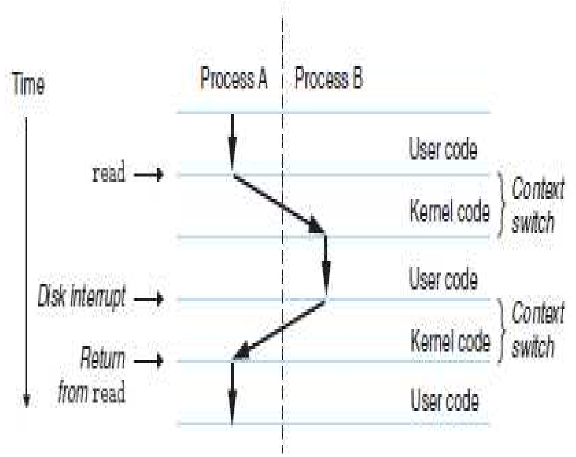
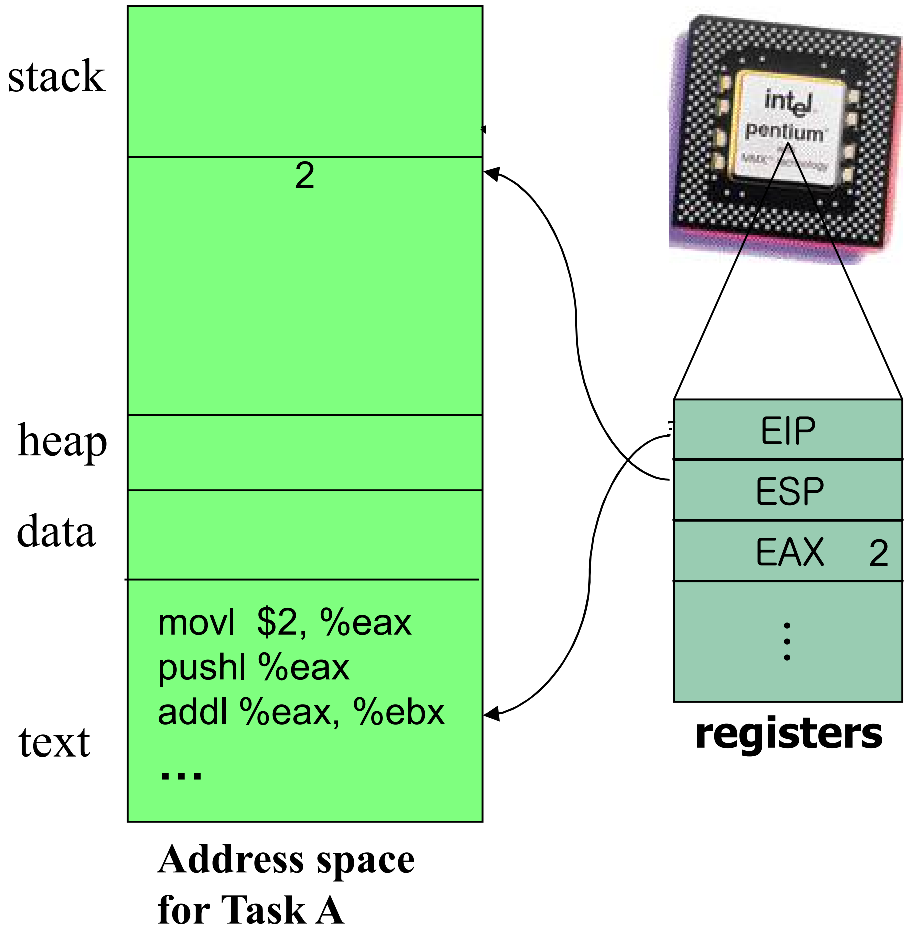


Figure 8.14
Anatomy of a process context switch.



Impact of ISA on system program: Multitasking (2/5)

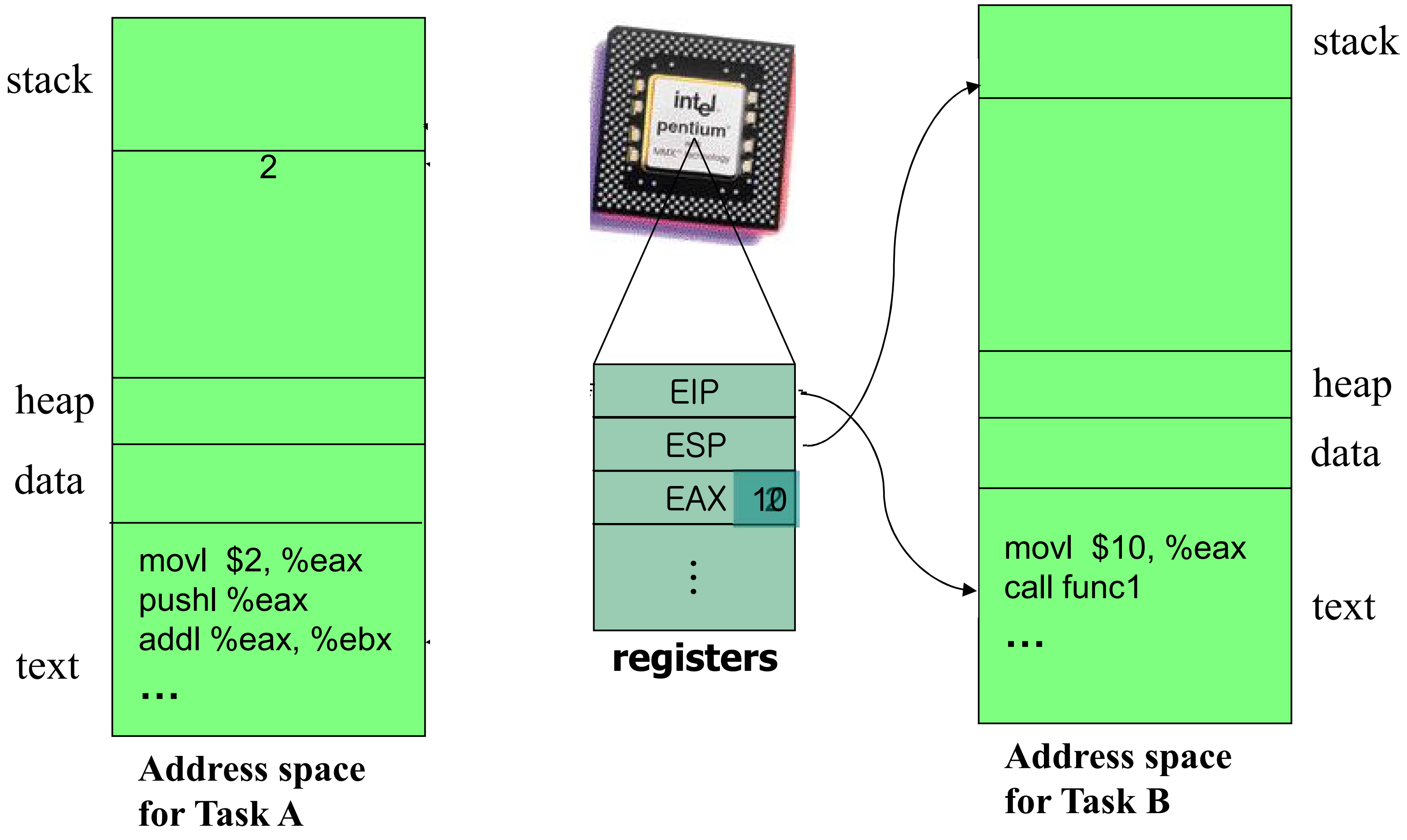
- Virtual CPU: running A



☞ **Time quantum is expired, system program (scheduler) selects a Task B to run next.**

Impact of ISA on system program: Multitasking (3/5)

- Virtual CPU: switch to B



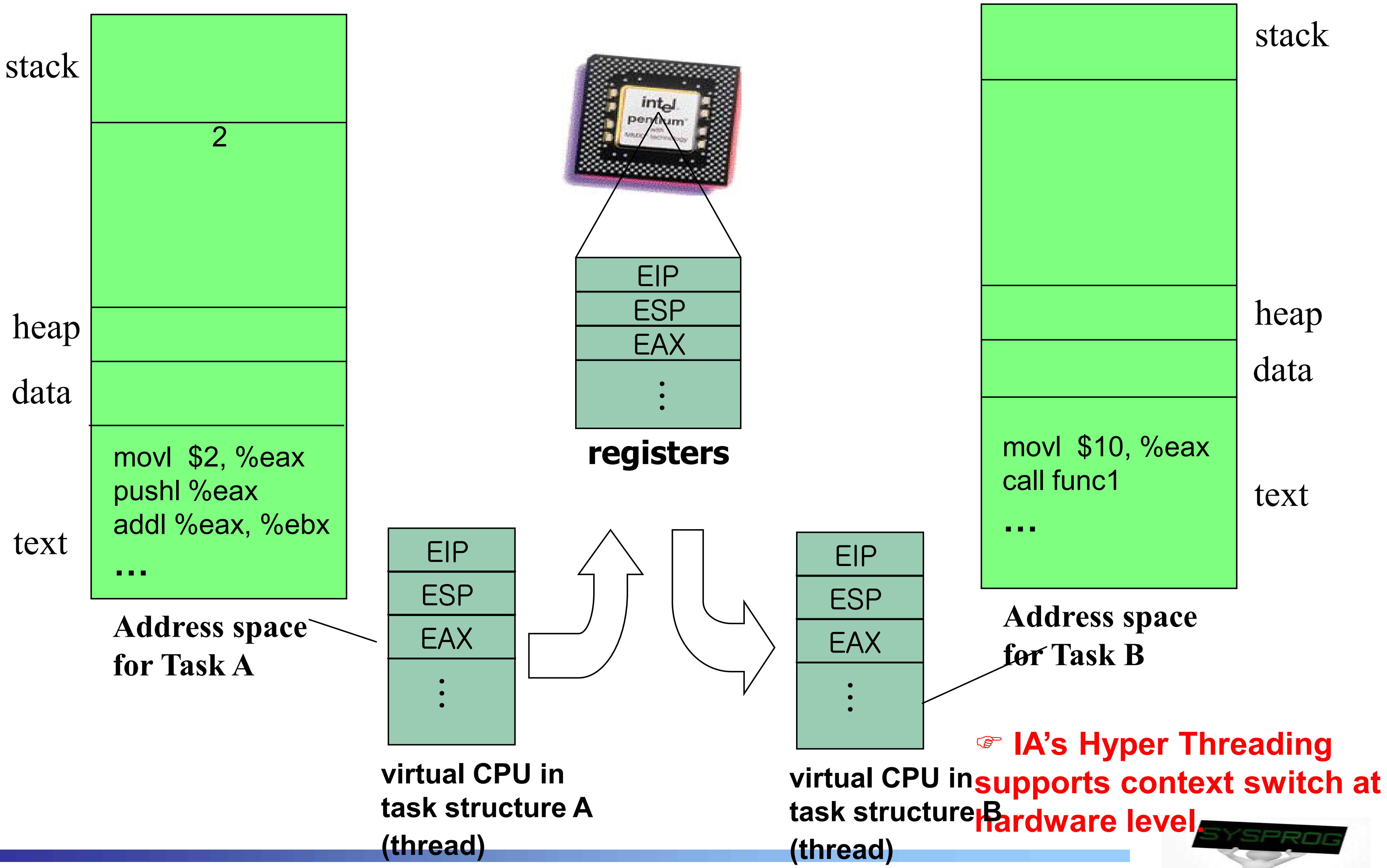
☞ Time quantum is expired, system program (scheduler) selects a Task B to run next.

☞ Time quantum is expired, again. Task A is scheduled. Then where to start?

☞ Context Switch ➔ save/restore context (architectural state or thread)

Impact of ISA on system program: Multitasking (4/5)

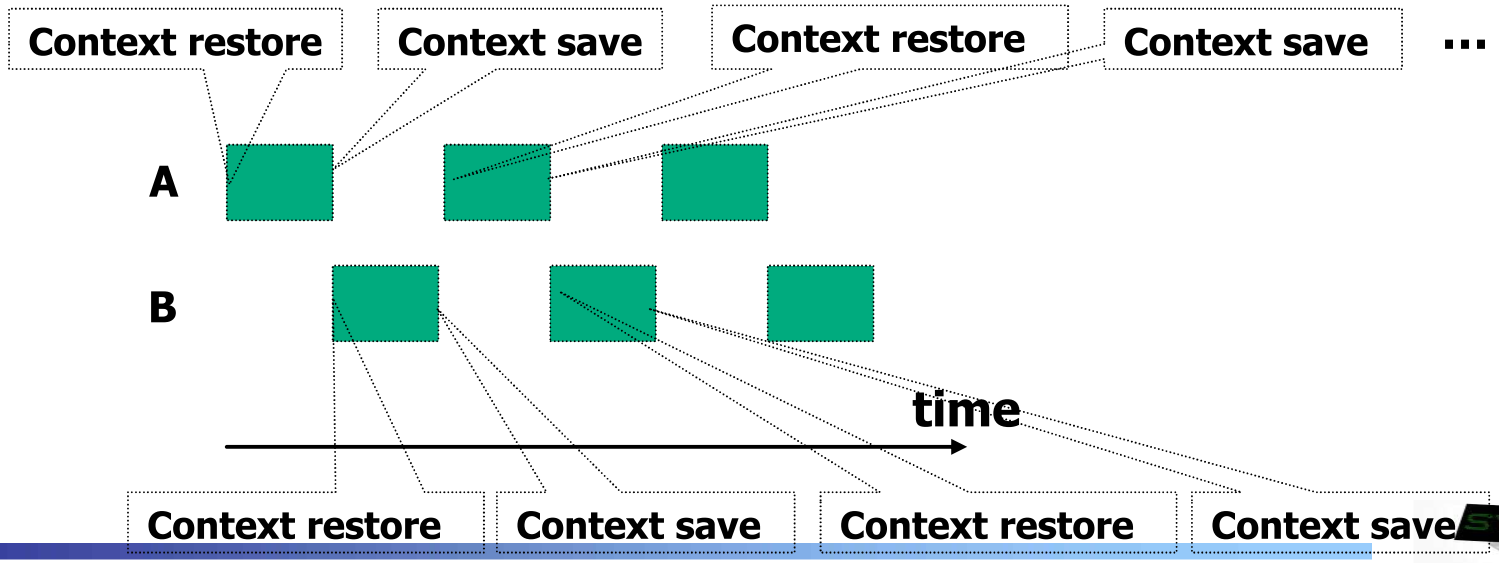
- Virtual CPU: how to switch back to A



☞ IA's Hyper Threading supports context switch at hardware level.

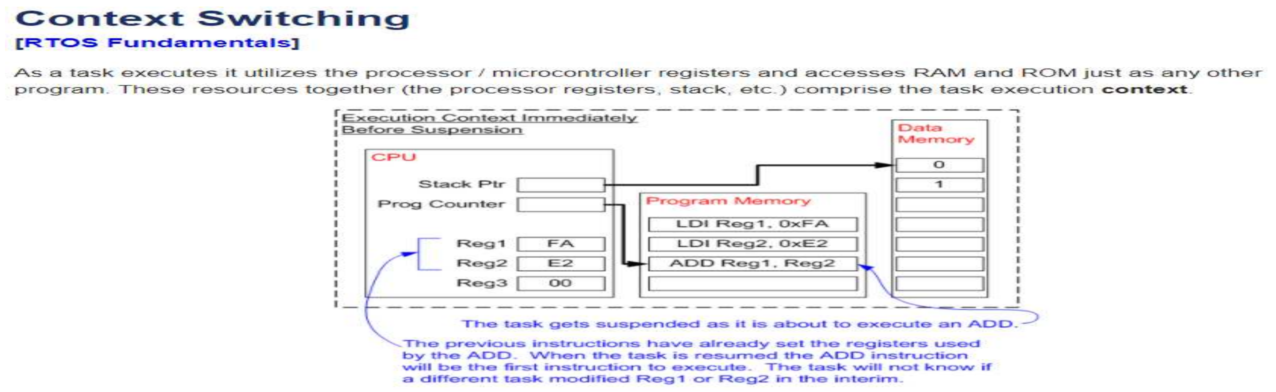
Impact of ISA on system program: Multitasking (5/5)

- Time sharing system
  - ✓ Tasks run interchangeable
  - ✓ Need to remember where to start ➔ Context
    - Context: registers, address space, opened files, IPCs, …
  - ✓ Context switch
    - When: timeout(time quantum expired), sleep, blocking I/O, …
    - How
      - Context save: CPU registers ➔ task structure (memory)
      - Context restore: task structure (memory) ➔ CPU registers



Quiz for 11th-Week 1st-Lesson

- Quiz
  - ✓ 1. Explain the differences between “movl a, %eax” and “movl $a, %eax” in operand addressing modes.
  - ✓ 2. Assume that the total execution time of a task A and B are 10 seconds (10,000ms), respectively. They run in a time-sharing manner with the time quantum as 100ms. Assume that the overhead for the context switch is 1ms. When the task A finishes if it begins at 0 second? (task B begins after task A is timed out)
  - ✓ Bonus) What if the time quantum is changed as 10ms?
  - ✓ Due: until 6 PM Friday of this week (13th, November)



Context Switching

[RTOS Fundamentals]

As a task executes it utilizes the processor / microcontroller registers and accesses RAM and ROM just as any other program. These resources together (the processor registers, stack, etc.) comprise the task execution **context**.

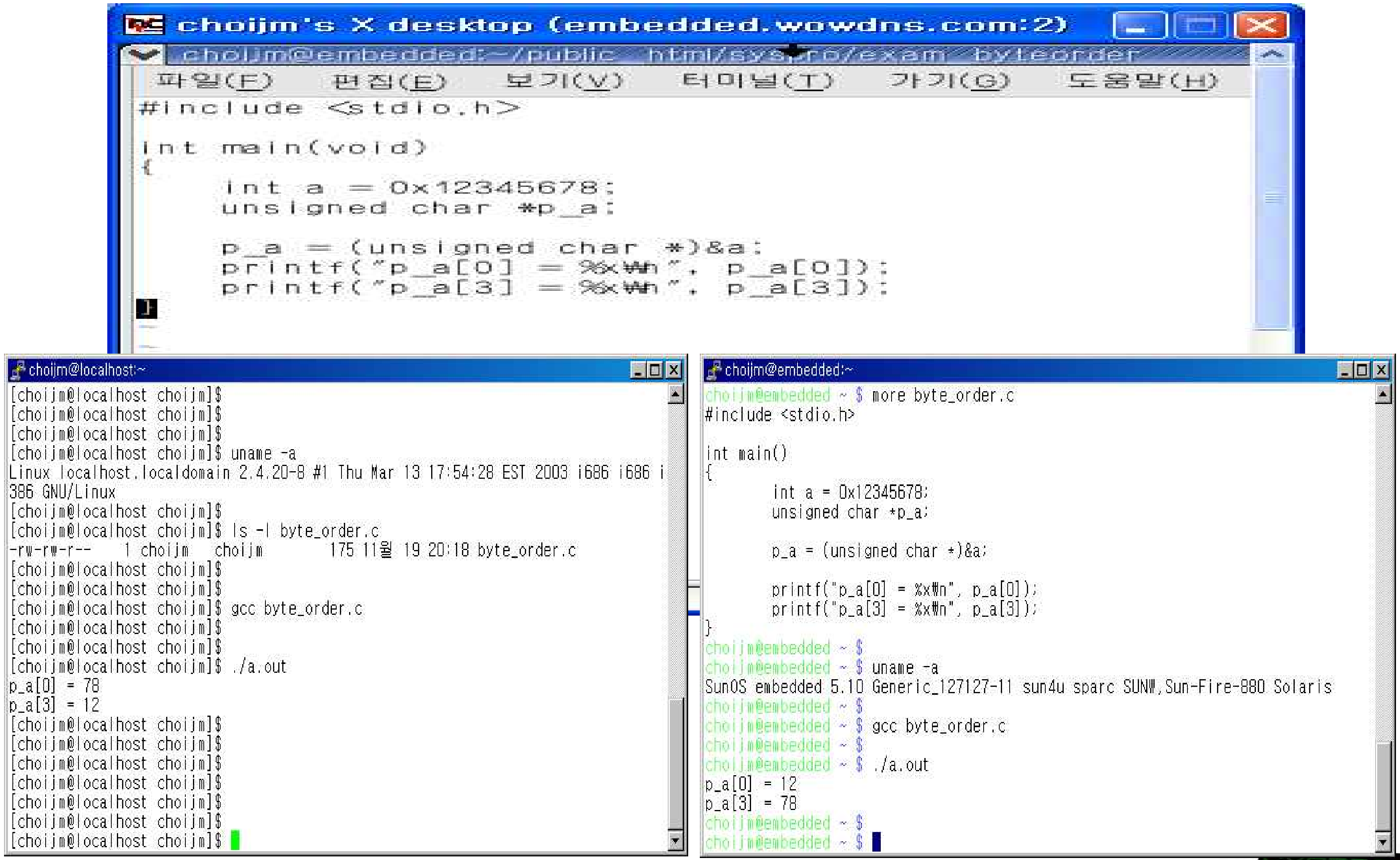
The task gets suspended as it is about to execute an ADD.

The previous instructions have already set the registers used by the ADD. When the task is resumed the ADD instruction will be the first instruction to execute. The task will not know if a different task modified Reg1 or Reg2 in the interim.

(Source: https://www.freertos.org/implementation/a00006.html)

Impact of ISA on system program: Memory Usage (1/5)

- Little Endian vs. Big Endian


choijm's X desktop (embedded.wowdns.com:2)
choijm@embedded:~/public_html/sysprog/exam/byteorder
파일(F) 편집(E) 보기(V) 터미널(T) 가기(G) 도움말(H)

```
#include <stdio.h>

int main(void)
{
        int a = 0x12345678;
        unsigned char *p_a;

        p_a = (unsigned char *)&a;
        printf("p_a[0] = %x₩n", p_a[0]);
        printf("p_a[3] = %x₩n", p_a[3]);
}
```

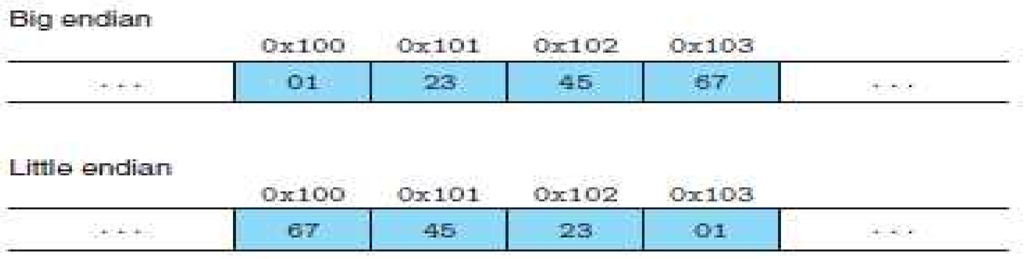
```
[choijm@localhost choijm]$
[choijm@localhost choijm]$
[choijm@localhost choijm]$
[choijm@localhost choijm]$ uname -a
Linux localhost.localdomain 2.4.20-8 #1 Thu Mar 13 17:54:28 EST 2003 i686 i686 i
386 GNU/Linux
[choijm@localhost choijm]$
[choijm@localhost choijm]$ ls -l byte_order.c
-rw-rw-r--    1 choijm   choijm        175 11월 19 20:18 byte_order.c
[choijm@localhost choijm]$
[choijm@localhost choijm]$
[choijm@localhost choijm]$ gcc byte_order.c
[choijm@localhost choijm]$
[choijm@localhost choijm]$
[choijm@localhost choijm]$ ./a.out
p_a[0] = 78
p_a[3] = 12
[choijm@localhost choijm]$
[choijm@localhost choijm]$
[choijm@localhost choijm]$
[choijm@localhost choijm]$
[choijm@localhost choijm]$
[choijm@localhost choijm]$
[choijm@localhost choijm]$
```

```
choijm@embedded ~ $ more byte_order.c
#include <stdio.h>

int main()
{
        int a = 0x12345678;
        unsigned char *p_a;

        p_a = (unsigned char *)&a;

        printf("p_a[0] = %x₩n", p_a[0]);
        printf("p_a[3] = %x₩n", p_a[3]);
}
choijm@embedded ~ $
choijm@embedded ~ $ uname -a
SunOS embedded 5.10 Generic_127127-11 sun4u sparc SUNW,Sun-Fire-880 Solaris
choijm@embedded ~ $
choijm@embedded ~ $ gcc byte_order.c
choijm@embedded ~ $
choijm@embedded ~ $ ./a.out
p_a[0] = 12
p_a[3] = 78
choijm@embedded ~ $
choijm@embedded ~ $
```

Impact of ISA on system program: Memory Usage (2/5)

- Little Endian vs. Big Endian

Continuing our earlier example, suppose the variable x of type int and at address 0x100 has a hexadecimal value of 0x01234567. The ordering of the bytes within the address range 0x100 through 0x103 depends on the type of machine:



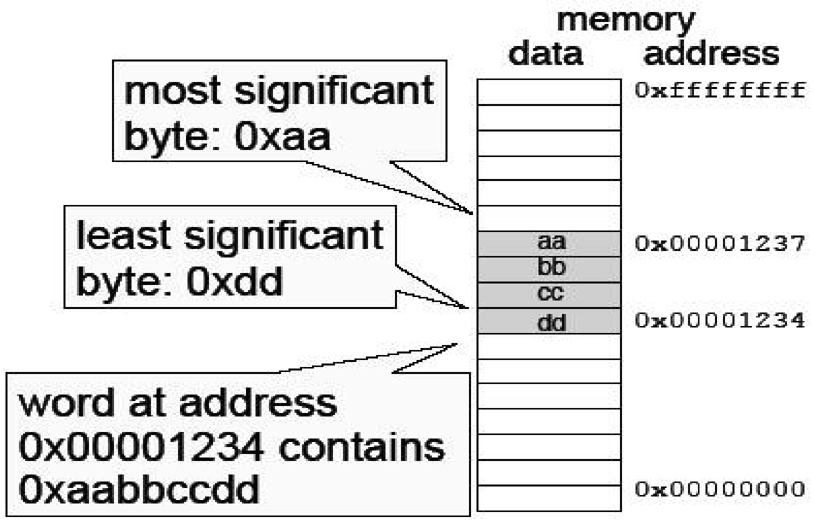
Big endian

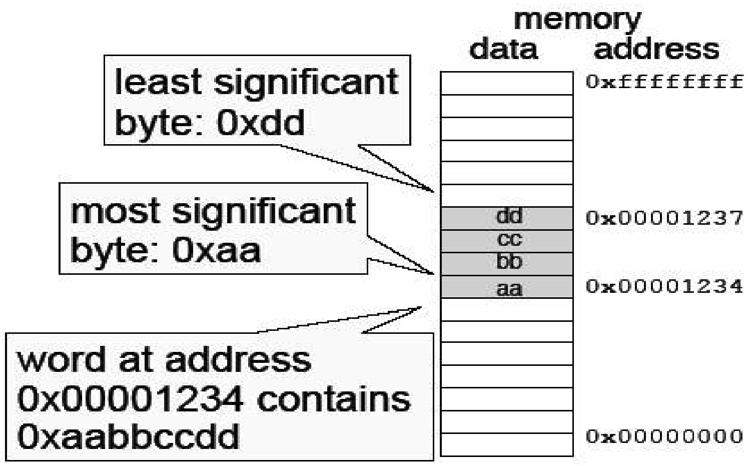
	0x100	0x101	0x102	0x103	
. . .	01	23	45	67	. . .

Little endian

	0x100	0x101	0x102	0x103	
. . .	67	45	23	01	. . .

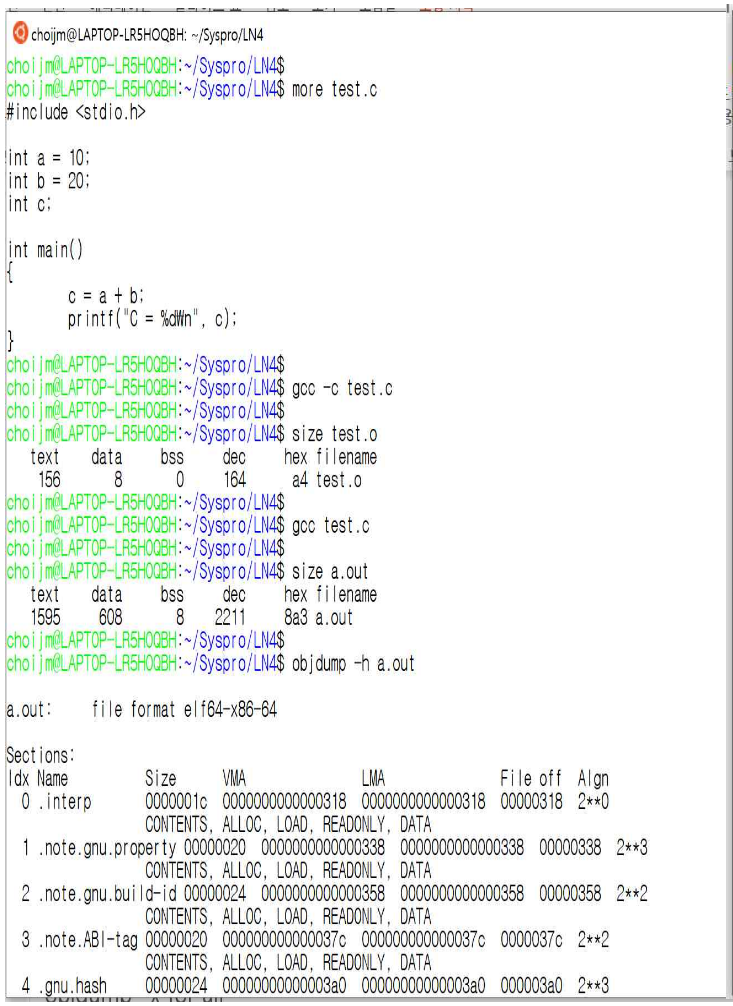
(Source: CSAPP)


memory
data
address
most significant byte: 0xaa
least significant byte: 0xdd
word at address 0x00001234 contains 0xaabbccdd
0xffffffff
aa
bb
cc
dd
0x00001237
0x00001234
0x00000000


memory
data
address
least significant byte: 0xdd
most significant byte: 0xaa
word at address 0x00001234 contains 0xaabbccdd
0xffffffff
dd
cc
bb
aa
0x00001237
0x00001234
0x00000000

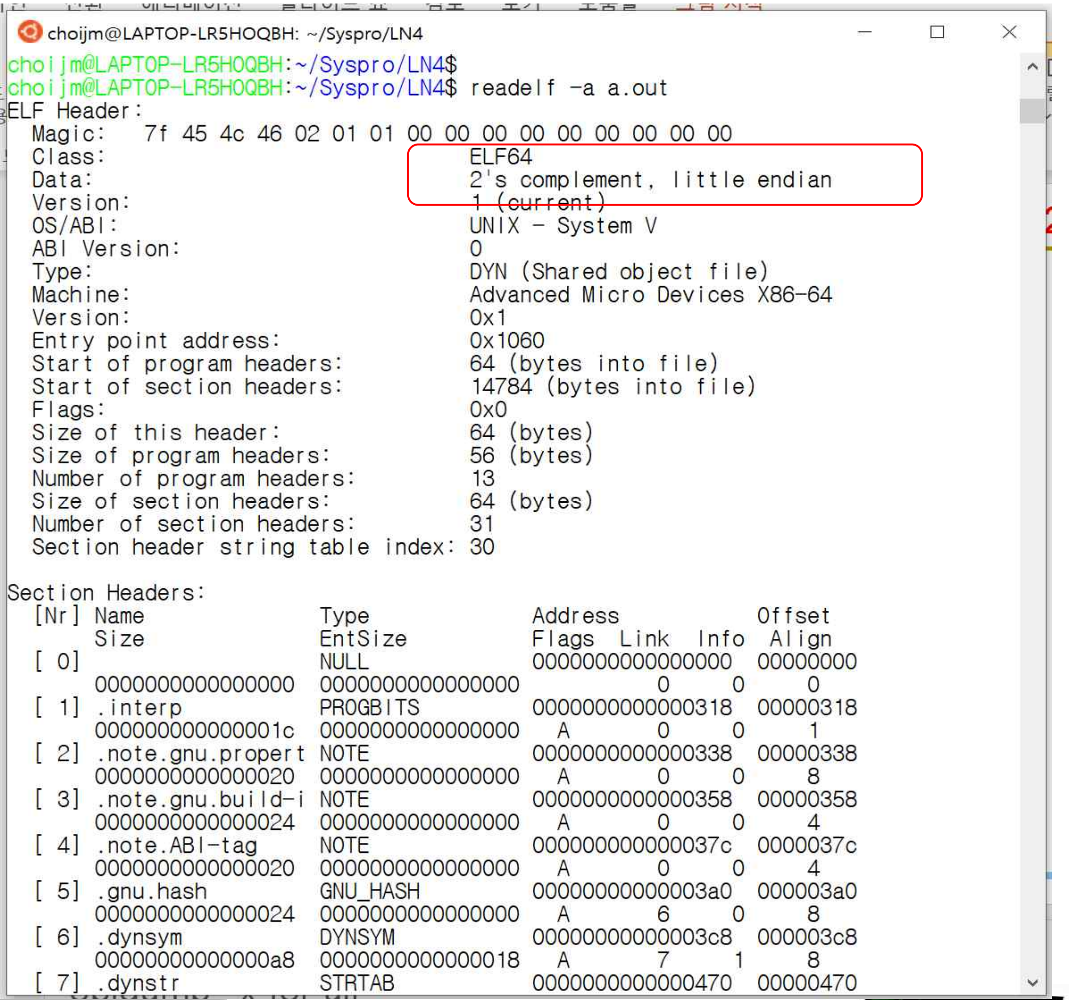
Impact of ISA on system program: Memory Usage (3/5)

- Where can we see the little endian?
  - ✓ readelf command



choijm@LAPTOP-LR5HOQBH: ~/Syspro/LN4

```
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$ more test.c
#include <stdio.h>

int a = 10;
int b = 20;
int c;

int main()
{
        c = a + b;
        printf("C = %d\n", c);
}
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$ gcc -c test.c
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$ size test.o
   text    data     bss     dec     hex filename
    156       8       0     164      a4 test.o
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$ gcc test.c
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$ size a.out
   text    data     bss     dec     hex filename
   1595     608       8    2211     8a3 a.out
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$ objdump -h a.out

a.out:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .interp       0000001c  0000000000000318  0000000000000318  00000318  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.gnu.property 00000020  0000000000000338  0000000000000338  00000338  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .note.gnu.build-id 00000024  0000000000000358  0000000000000358  00000358  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .note.ABI-tag 00000020  000000000000037c  000000000000037c  0000037c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .gnu.hash     00000024  00000000000003a0  00000000000003a0  000003a0  2**3
```

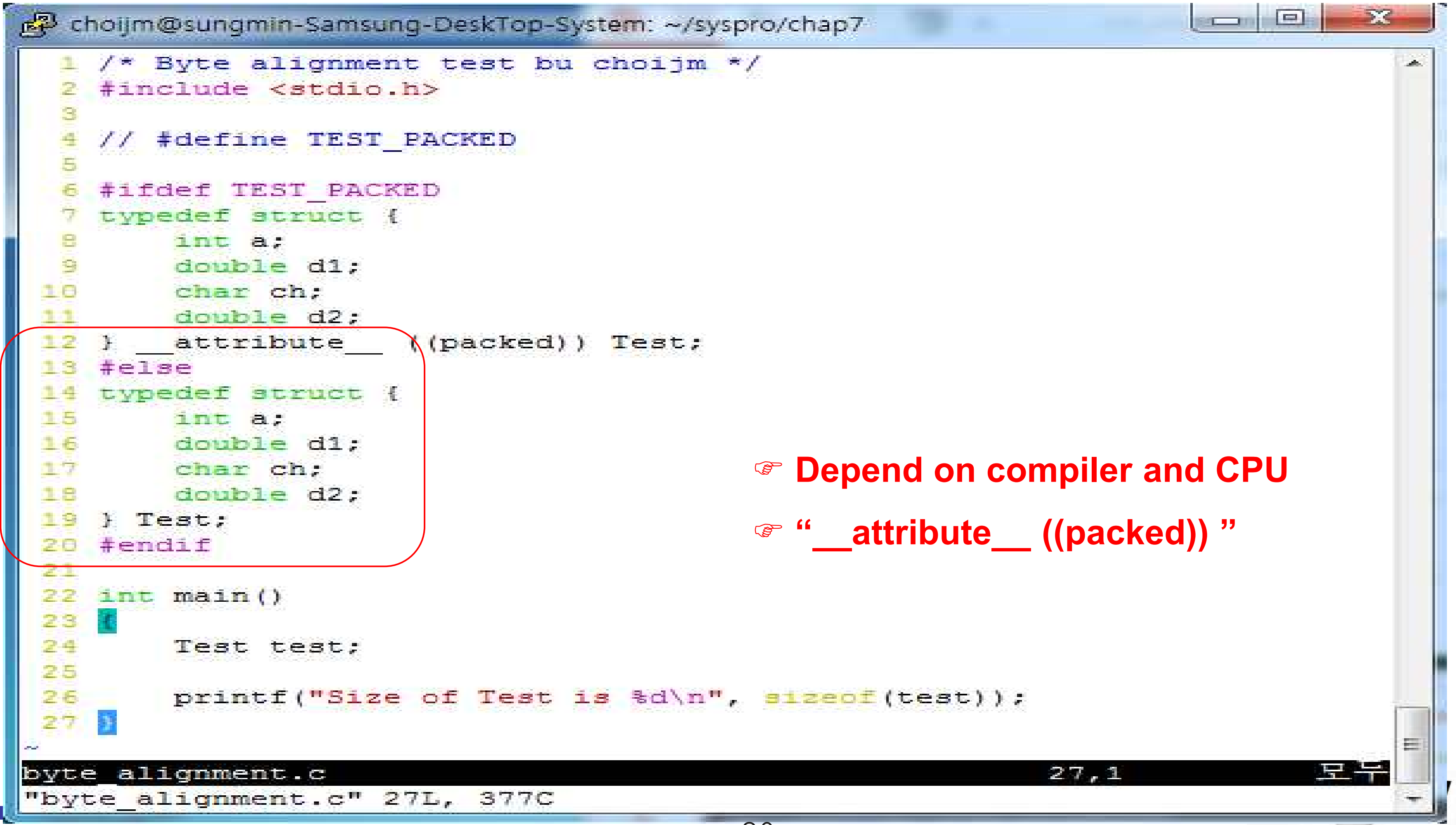


choijm@LAPTOP-LR5HOQBH: ~/Syspro/LN4

```
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$ readelf -a a.out
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x1060
  Start of program headers:          64 (bytes into file)
  Start of section headers:          14784 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         31
  Section header string table index: 30

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000000318  00000318
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.gnu.propert NOTE             0000000000000338  00000338
       0000000000000020  0000000000000000   A       0     0     8
  [ 3] .note.gnu.build-i NOTE             0000000000000358  00000358
       0000000000000024  0000000000000000   A       0     0     4
  [ 4] .note.ABI-tag     NOTE             000000000000037c  0000037c
       0000000000000020  0000000000000000   A       0     0     4
  [ 5] .gnu.hash         GNU_HASH         00000000000003a0  000003a0
       0000000000000024  0000000000000000   A       6     0     8
  [ 6] .dynsym           DYNSYM           00000000000003c8  000003c8
       00000000000000a8  0000000000000018   A       7     1     8
  [ 7] .dynstr           STRTAB           0000000000000470  00000470
```

Impact of ISA on system program: Memory Usage (4/5)

- Memory Alignment in data structure
  - ✓ To reduce memory fetch numbers (and atomicity)
  - ✓ To consider cache line boundary (and false sharing)


choijm@sungmin-Samsung-DeskTop-System: ~/syspro/chap7

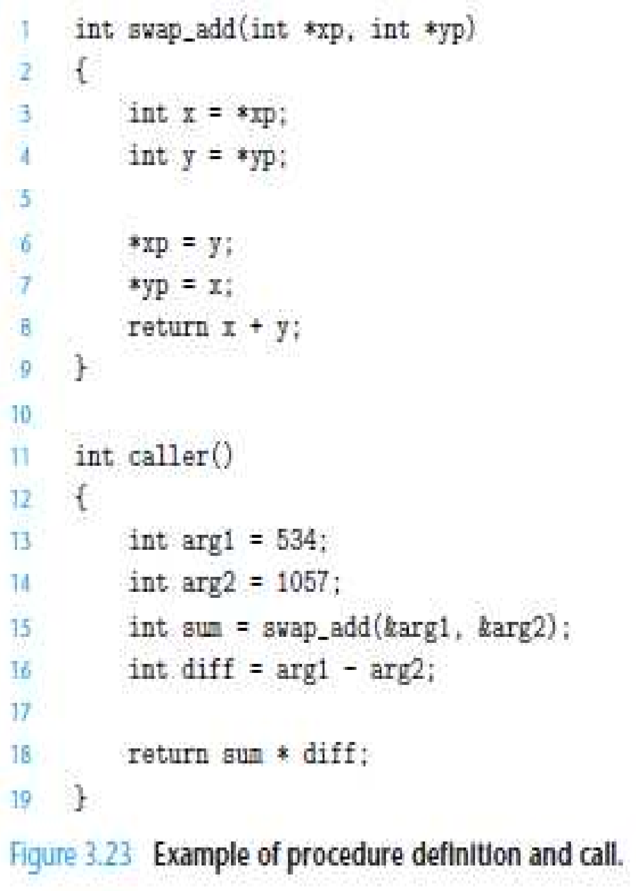
```
/* Byte alignment test bu choijm */
#include <stdio.h>

// #define TEST_PACKED

#ifdef TEST_PACKED
typedef struct {
        int a;
        double d1;
        char ch;
        double d2;
} __attribute__ ((packed)) Test;
#else
typedef struct {
        int a;
        double d1;
        char ch;
        double d2;
} Test;
#endif

int main()
{
        Test test;

        printf("Size of Test is %d\n", sizeof(test));
}
~
byte_alignment.c                                                     27,1           모두
"byte_alignment.c" 27L, 377C
```

☞ **Depend on compiler and CPU**

☞ **"__attribute__ ((packed)) "**

Impact of ISA on system program: Memory Usage (5/5)

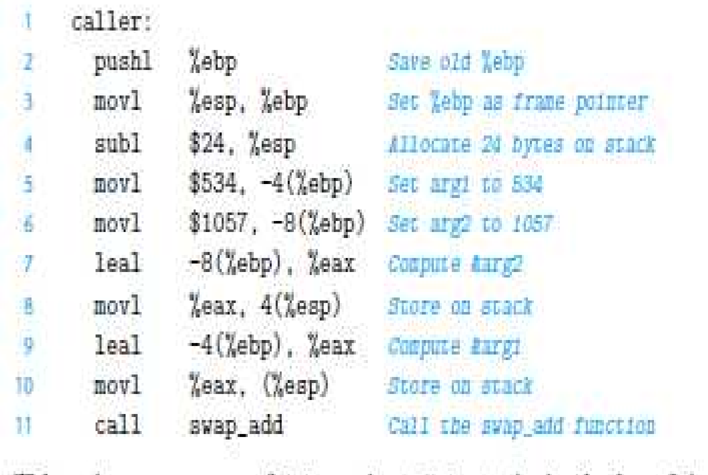
- Memory Alignment in stack
  - ✓ Need 16 bytes (8 for local variables and 8 for arguments) ➔ But allocate 24 bytes for 16 bytes alignment in a frame (recommended by IA)



```
int swap_add(int *xp, int *yp)
{
    int x = *xp;
    int y = *yp;

    *xp = y;
    *yp = x;
    return x + y;
}

int caller()
{
    int arg1 = 534;
    int arg2 = 1057;
    int sum = swap_add(&arg1, &arg2);
    int diff = arg1 - arg2;

    return sum * diff;
}
```

Figure 3.23 Example of procedure definition and call.

(Source: CSAPP)



```
caller:
  pushl  %ebp              Save old %ebp
  movl   %esp, %ebp        Set %ebp as frame pointer
  subl   $24, %esp         Allocate 24 bytes on stack
  movl   $534, -4(%ebp)    Set arg1 to 534
  movl   $1057, -8(%ebp)   Set arg2 to 1057
  leal   -8(%ebp), %eax    Compute &arg2
  movl   %eax, 4(%esp)     Store on stack
  leal   -4(%ebp), %eax    Compute &arg1
  movl   %eax, (%esp)      Store on stack
  call   swap_add          Call the swap_add function
```

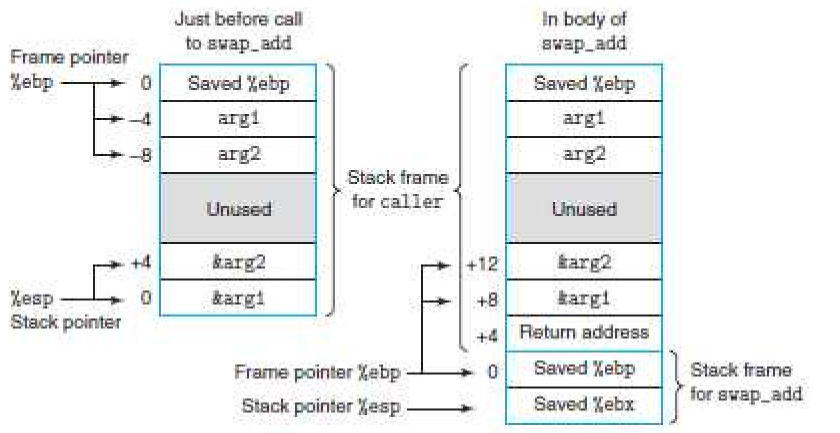
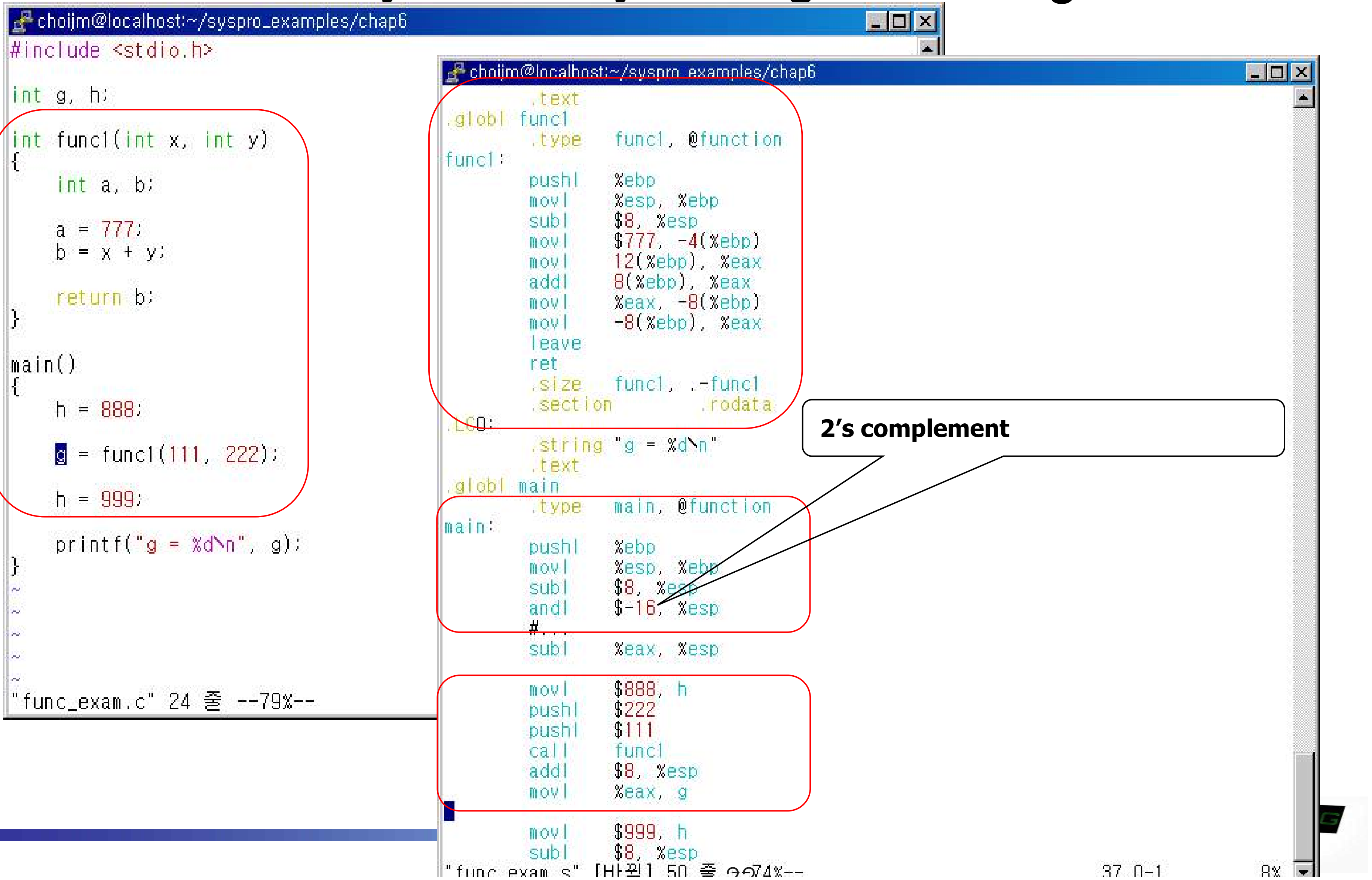


Figure 3.24 Stack frames for caller and swap_add. Procedure swap_add retrieves its arguments from the stack frame for caller.

Revisit the stack in LN 6

- Another way for 16 bytes alignment in gcc



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%--
```

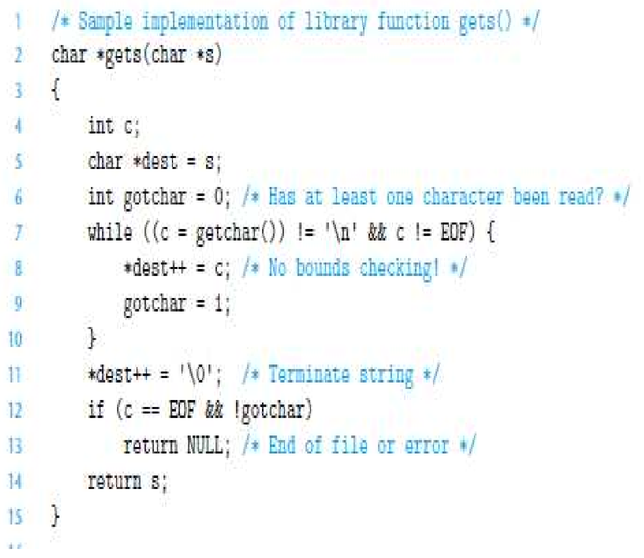
choijm@localhost:~/syspro_examples/chap6

```
        .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%
```

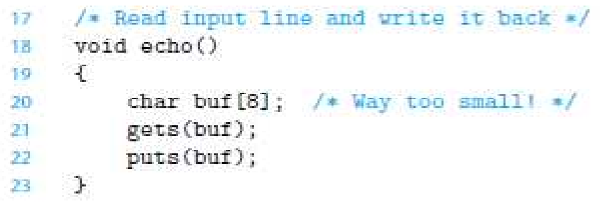
2's complement

Impact of ISA on system program: Buffer Overflow (1/3)

- Buffer overflow
  - ✓ Due to the no boundary check
  - ✓ How to thwart buffer overflow
    - Stack randomization
      - One step further: ASLR (Address Space Layout Randomization) ➔ even code, data and heap
    - Stack guard (e.g. Canary)

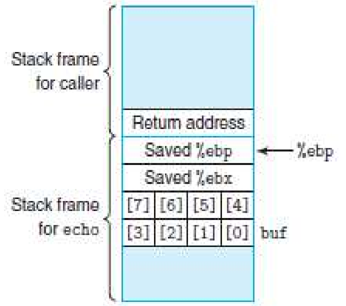


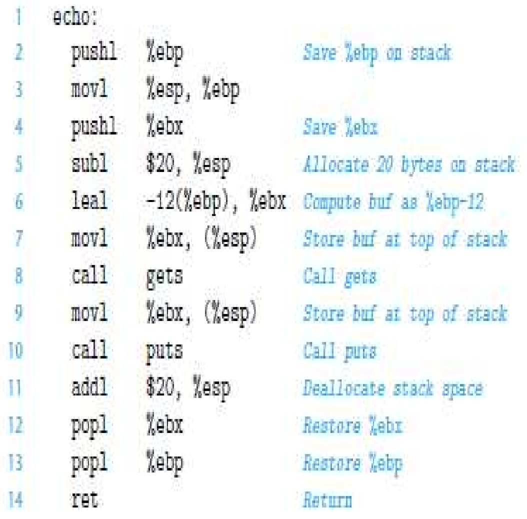
```
/* Sample implementation of library function gets() */
char *gets(char *s)
{
    int c;
    char *dest = s;
    int gotchar = 0; /* Has at least one character been read? */
    while ((c = getchar()) != '\n' && c != EOF) {
        *dest++ = c; /* No bounds checking! */
        gotchar = 1;
    }
    *dest++ = '\0';  /* Terminate string */
    if (c == EOF && !gotchar)
        return NULL; /* End of file or error */
    return s;
}
```



```
/* Read input line and write it back */
void echo()
{
    char buf[8];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

Figure 3.31
Stack organization for echo **function.** Character array buf is just below part of the saved state. An out-of-bounds write to buf can corrupt the program state.

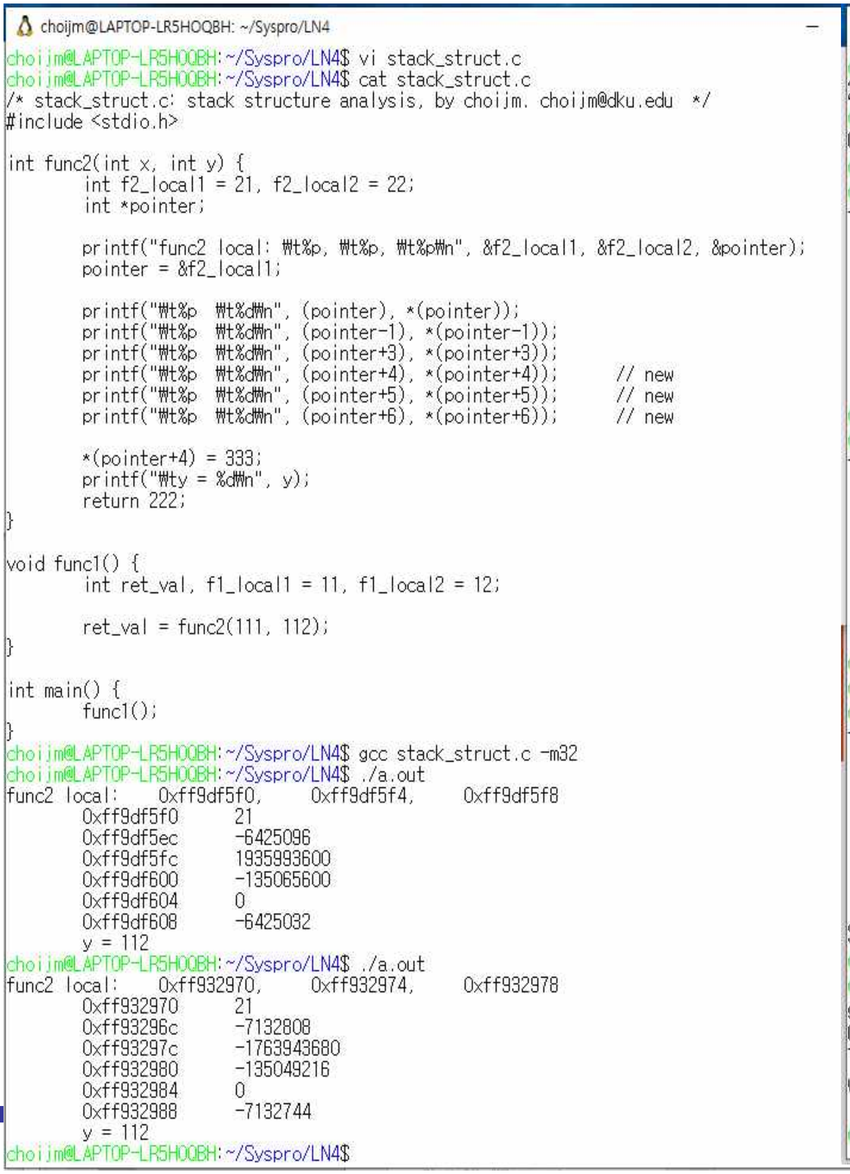




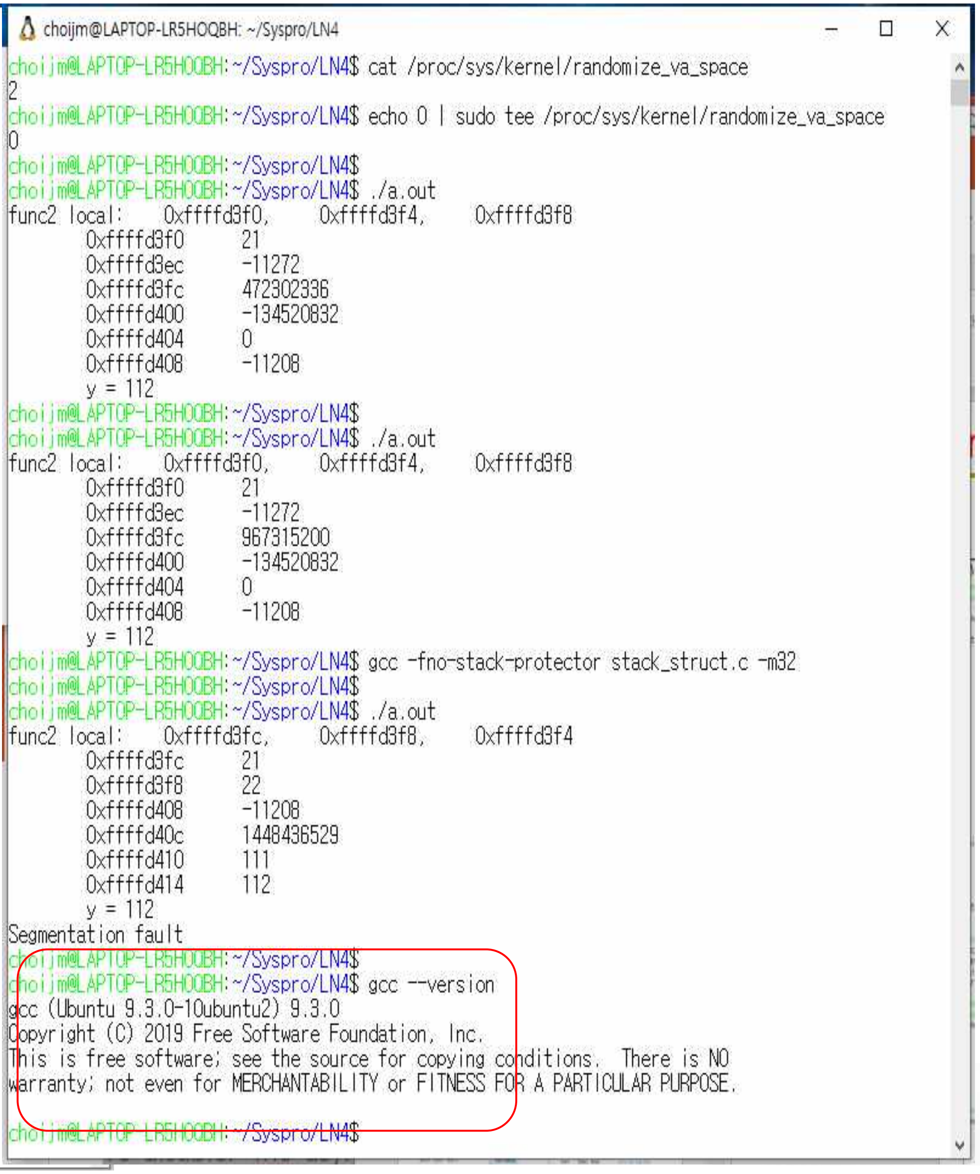
```
echo:
  pushl  %ebp             Save %ebp on stack
  movl   %esp, %ebp
  pushl  %ebx             Save %ebx
  subl   $20, %esp        Allocate 20 bytes on stack
  leal   -12(%ebp), %ebx  Compute buf as %ebp-12
  movl   %ebx, (%esp)     Store buf at top of stack
  call   gets             Call gets
  movl   %ebx, (%esp)     Store buf at top of stack
  call   puts             Call puts
  addl   $20, %esp        Deallocate stack space
  popl   %ebx             Restore %ebx
  popl   %ebp             Restore %ebp
  ret                     Return
```

Impact of ISA on system program: Buffer Overflow (2/3)

- Stack randomization



```
choijm@LAPTOP-LR5HOQBH: ~/Syspro/LN4
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$ vi stack_struct.c
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$ cat stack_struct.c
/* stack_struct.c: stack structure analysis, by choijm. choijm@dku.edu  */
#include <stdio.h>

int func2(int x, int y) {
        int f2_local1 = 21, f2_local2 = 22;
        int *pointer;

        printf("func2 local: ₩t%p, ₩t%p, ₩t%p₩n", &f2_local1, &f2_local2, &pointer);
        pointer = &f2_local1;

        printf("₩t%p  ₩t%d₩n", (pointer), *(pointer));
        printf("₩t%p  ₩t%d₩n", (pointer-1), *(pointer-1));
        printf("₩t%p  ₩t%d₩n", (pointer+3), *(pointer+3));
        printf("₩t%p  ₩t%d₩n", (pointer+4), *(pointer+4));      // new
        printf("₩t%p  ₩t%d₩n", (pointer+5), *(pointer+5));      // new
        printf("₩t%p  ₩t%d₩n", (pointer+6), *(pointer+6));      // new

        *(pointer+4) = 333;
        printf("₩ty = %d₩n", y);
        return 222;
}

void func1() {
        int ret_val, f1_local1 = 11, f1_local2 = 12;

        ret_val = func2(111, 112);
}

int main() {
        func1();
}
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$ gcc stack_struct.c -m32
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$ ./a.out
func2 local:    0xff9df5f0,     0xff9df5f4,     0xff9df5f8
        0xff9df5f0      21
        0xff9df5ec      -6425096
        0xff9df5fc      1935993600
        0xff9df600      -135065600
        0xff9df604      0
        0xff9df608      -6425032
        y = 112
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$ ./a.out
func2 local:    0xff932970,     0xff932974,     0xff932978
        0xff932970      21
        0xff93296c      -7132808
        0xff93297c      -1763943680
        0xff932980      -135049216
        0xff932984      0
        0xff932988      -7132744
        y = 112
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$
```

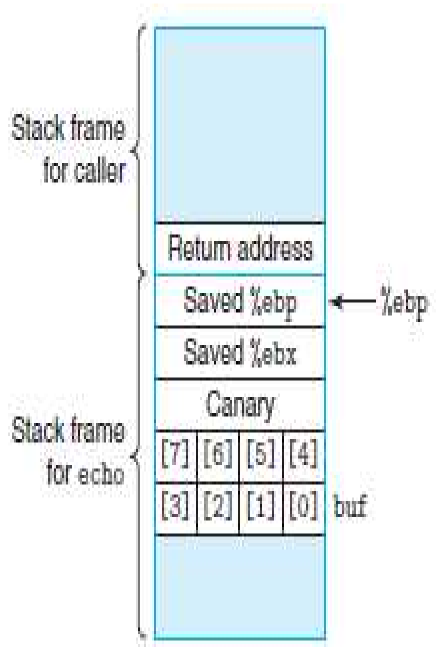


```
choijm@LAPTOP-LR5HOQBH: ~/Syspro/LN4
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$ cat /proc/sys/kernel/randomize_va_space
2
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
0
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$ ./a.out
func2 local:    0xffffd3f0,     0xffffd3f4,     0xffffd3f8
        0xffffd3f0      21
        0xffffd3ec      -11272
        0xffffd3fc      472302336
        0xffffd400      -134520832
        0xffffd404      0
        0xffffd408      -11208
        y = 112
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$ ./a.out
func2 local:    0xffffd3f0,     0xffffd3f4,     0xffffd3f8
        0xffffd3f0      21
        0xffffd3ec      -11272
        0xffffd3fc      967315200
        0xffffd400      -134520832
        0xffffd404      0
        0xffffd408      -11208
        y = 112
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$ gcc -fno-stack-protector stack_struct.c -m32
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$ ./a.out
func2 local:    0xffffd3fc,     0xffffd3f8,     0xffffd3f4
        0xffffd3fc      21
        0xffffd3f8      22
        0xffffd408      -11208
        0xffffd40c      1448436529
        0xffffd410      111
        0xffffd414      112
        y = 112
Segmentation fault
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$ gcc --version
gcc (Ubuntu 9.3.0-10ubuntu2) 9.3.0
Copyright (C) 2019 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$
```

Impact of ISA on system program: Buffer Overflow (3/3)

- Stack protector
  - ✓ Typical example: canary
  - ✓ Included as default in modern gcc

Figure 3.33
Stack organization for echo function with stack protector enabled. A special "canary" value is positioned between array buf and the saved state. The code checks the canary value to determine whether or not the stack state has been corrupted.



```
echo:
  pushl   %ebp
  movl    %esp, %ebp
  pushl   %ebx
  subl    $20, %esp
  movl    %gs:20, %eax          Retrieve canary
  movl    %eax, -8(%ebp)        Store on stack
  xorl    %eax, %eax            Zero out register
  leal    -16(%ebp), %ebx       Compute buf as %ebp-16
  movl    %ebx, (%esp)          Store buf at top of stack
  call    gets                  Call gets
  movl    %ebx, (%esp)          Store buf at top of stack
  call    puts                  Call puts
  movl    -8(%ebp), %eax        Retrieve canary
  xorl    %gs:20, %eax          Compare to stored value
  je      .L19                  If =, goto ok
  call    __stack_chk_fail      Stack corrupted!
.L19:                         ok:
  addl    $20, %esp             Normal return ...
  popl    %ebx
  popl    %ebp
  ret
```

Intel CPU History (1/9)

- 8080 (1974)
  - ✓ 8bit register, 8bit bus, 64KB memory support
- 8086 (1978)
  - ✓ 16bit register, 16bit data bus, 20bit address bus (8088: 8bit data bus for backward compatibility, others are same as 8086), 1st generation of x86 ISA
  - ✓ Segmentation (real addressing mode, 1MB memory support)
- 80286 (1982)
  - ✓ 16bit, 24bit address bus
  - ✓ Segmentation (use segment descriptors, 16MB memory support)
  - ✓ 4 privilege level
- 80386 (1985)
  - ✓ 32bit register and bus (80386 SX: 16bit bus for backward compatibility)
  - ✓ First 32bit addressing (4GB memory support)
  - ✓ Paging with a fixed 4-KBytes page size

Intel CPU History (2/9)

- 80486 (1989)
  - ✓ Pipelining support (3 stages of execution, introduce u-op)
  - ✓ Use L1 cache (keep recently used instruction, 8KB)
  - ✓ An integrated x87 FPU (no FPU in 486SX)
  - ✓ power saving support, system management mode for notebook (486SL)

- Pentium (1993, 5th generation)
  - ✓ 5-stage pipeline, Superscalar support (two pipelines (u and v), which allows to execute at most two u-ops at a cycle in parallel)
  - ✓ L1 cache is divided into D-Cache, I-Cache, Use L2 cache, write back protocol (MESI protocol)
  - ✓ Introduce Branch Prediction
  - ✓ APIC for multiple processor

☞ **Why not the 80586?**

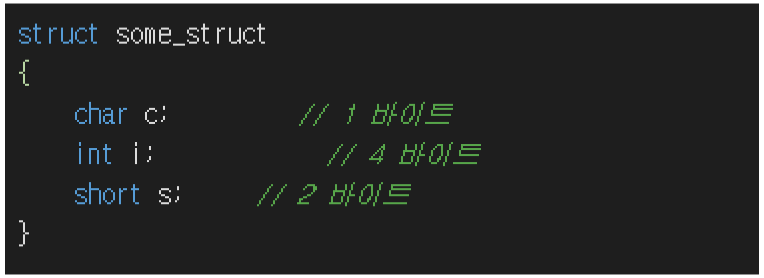
  - ✓ Pentium with MMX Technology
    - Equip Multimedia Accelerator.
    - SIMD(Single Instruction Multiple Data): High performance for Matrix processing (one of the big changes in x86 ISA, CISC flavor)

Intel CPU History (3/9)

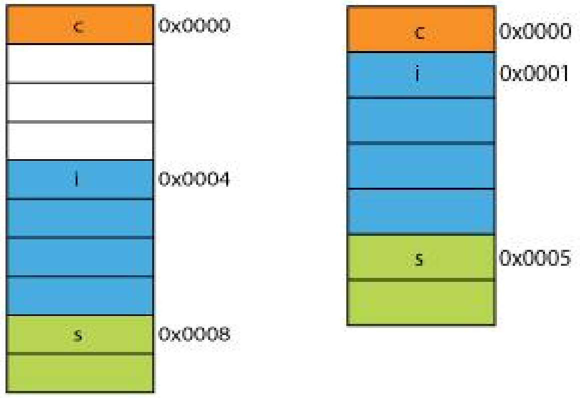
- P6 family (1995~1999, 6th generation)
  - ✓ P6 Microarchitecture: Dynamic execution
    - Out-of-order execution
    - Branch prediction
    - Speculative execution: decouple execution and commitment (retirement unit)
    - Data flow analysis: detect independent instructions on real time
    - Register renaming
  - ✓ Pentium Pro
    - Three instructions per clock cycle (3-way superscalar), 256KB L2 cache
    - Even though its name is similar to Pentium, its internal is quite novel (eg. employ diverse RICS features such as first out-of-order execution)
  - ✓ Pentium II
    - MMX enhancement, 16KB L1 cache, 1MB L2 cache
    - Multiple low power state (Autohalt, Stop-grant, sleep, deep sleep)
    - Pentium II Xeon: Premium Pentium II (for server, large cache and scalability)
    - Pentium II Cerelon: For lower system cost (for cost-optimization, no L2 or small)
  - ✓ Pentium III
    - SSE (Streaming SIMD Extension): 128bit register(XMM), FPU support , Multimedia specialized instruction (around 70), Coopermine, Tualatin, …
    - Pentium III Xeon: Premium Pentium III

Quiz for 11th-Week 2nd-Lesson

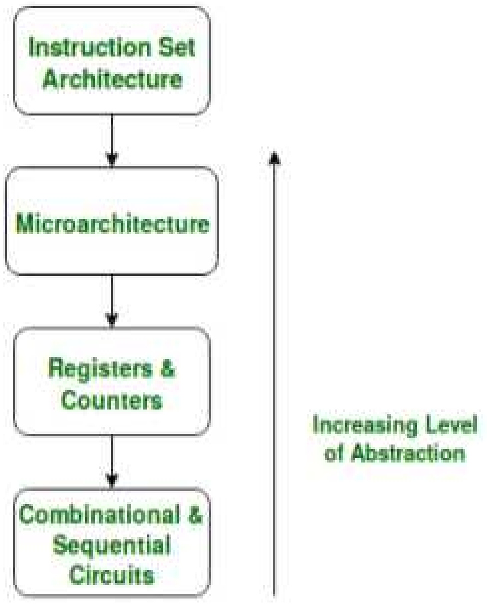
- Quiz
  - ✓ 1. Discuss the benefits of memory alignment (at least 2)
  - ✓ 2. Explain the key techniques of the dynamic execution in the Intel P6 microarchitecture (5 techniques)
  - ✓ Due: until 6 PM Friday of this week (13th, November)



```
struct some_struct
{
    char c;          // 1 바이트
    int i;             // 4 바이트
    short s;      // 2 바이트
}
```



(source: minusi.tistory.com)



Instruction Set Architecture
Microarchitecture
Registers & Counters
Combinational & Sequential Circuits
Increasing Level of Abstraction

(source: https://www.geeksforgeeks.org/microarchitecture-and-instruction-set-architecture/)

Intel CPU History (4/9)

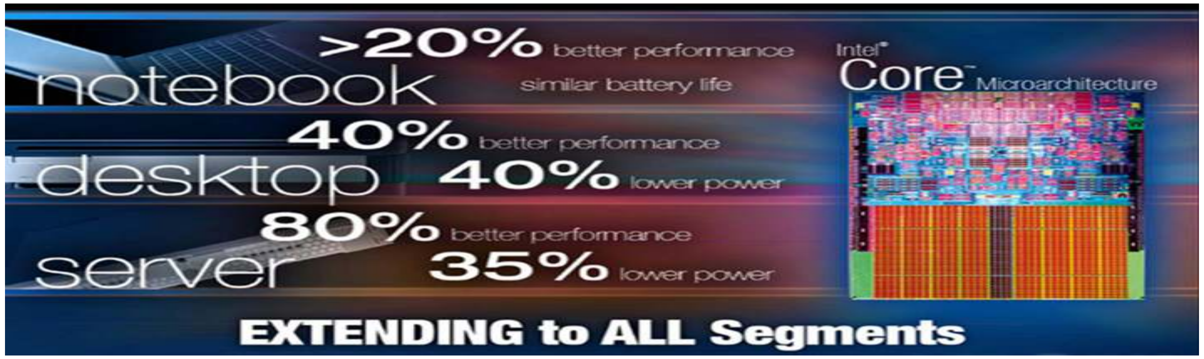
- Pentium 4 Processor Family (2000~2006, also release Itanium)
  - ✓ NetBurst microarchitecture
    - Deep pipelining (Hyper Pipelining: 20~31 stages u-op, expected up to 10GHz)
    - Wider design: Rapid Execution (ALU 2X), System Bus (4X)
    - Advanced Dynamic Execution
      - Deep, out-of-order execution engine, Enhanced branch prediction
    - New cache system (Advanced Trace Cache for decoded instructions)
  - ✓ Hyper-Threading: support Multithread at the CPU level (AS)
  - ✓ Pentium 4 with SSE2, SSE3
  - ✓ Pentium D (Smithfield, beginning of the dual core era)
  - ✓ Intel 64 (IA64, x86-64)
  - ✓ Virtualization technology
  - ✓ Market Name
    - Pentium 4
      - Northwood, Prescott, Cedermill, Smithfield, Willamette, …
    - Pentium M: low power, high performance mobile CPU
    - Intel Xeon Processor: Premium Pentium 4
      - 64-bit Xeon MP: 3.3GHz, 16KB L1, 1MB L2, 8MB L3
    - Intel Pentium Processor Extreme Edition (Gallatin)
      - For High performance PC

Pentium 4
Central processing unit

Produced	From 2000 to 2008
Common manufacturer(s)	Intel
Max CPU clock	1.3 GHz to 3.8 GHz

Intel CPU History (5/9)

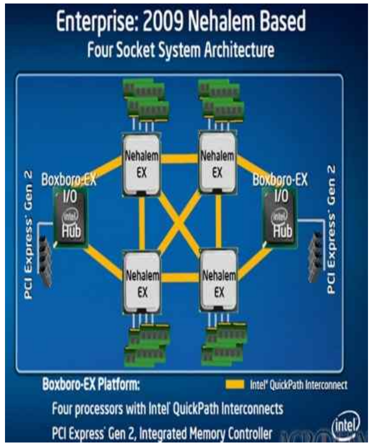
- Intel Core Processor Family (2006 ~)
  - ✓ Intel Core microarchitecture
    - NetBurst problem: high power consumption, pipeline inefficiency
    - Reengineering based on P6 Microarchitecture (14 stage of pipeline)
    - Increased L2 cache (6MB), 4 way superscalar, combine u-ops
    - Native Dualcore: not just packaging two cores, but integrating as the design stage (eg. Advanced Smart Cache (L2 sharing), Enhanced prefetcher)
  - ✓ Marketing name: use Core, not Pentium
    - Core Solo/Duo (32 bit)
      - Yonah (laptop), actually based on P6 microarchitecture
    - Core 2 Solo/Duo/Quad (64 bit)
      - Merom, Penryn (laptop), Conroe, Kentsfield, Yorkfield (desktop), Woodcrest, Clovertown(Server)
      - Develop rapidly to multiple cores

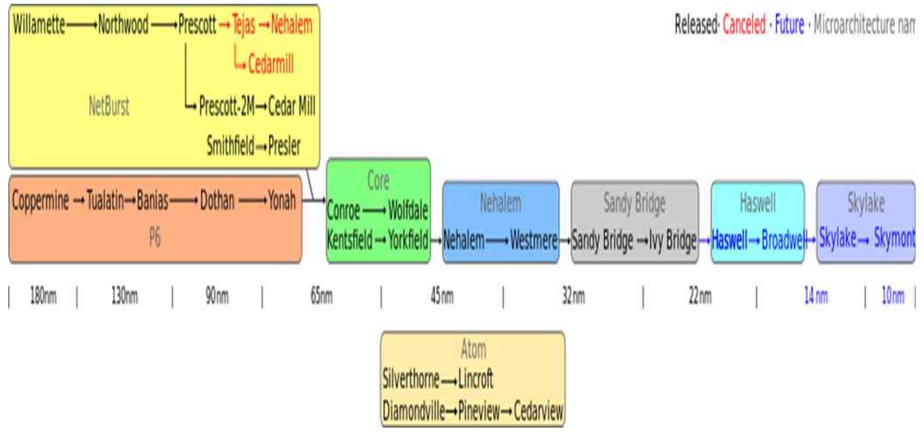


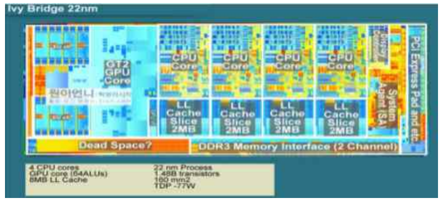
(source: http://motoc.tistory.com/)

Intel CPU History (6/9)

- Intel Core i3/i5/i7 Family (2009 ~)
  - ✓ Nehalem microarchitecture (and it's tick version Westmere)
    - Quickpath interconnect(for competing AMD's hyper-transport, supporting NUMA), IMC (Integrated Memory Controller), SMT, 45nm
    - Turbo mode, 256KB L2 cache/core, 12MB L3 cache, Intel Core 1st generation
  - ✓ Sandy Bridge, Haswell, Sky lake, Sunny Cove microarchitecture
    - Successor of Nehalem, <= 32 nm, Integrated GPU, AVX (Advanced Vector extensions, 256 bit SSE), HW-supported video transcoding/encryption,
    - Tick-Tock strategy
  - ✓ Marketing name: Core i3, i5, i7 (From mid-range (i3) to high-end (i7))
    - Lynnfield, Sandy bridge(Laptop), Gulftown, Sandy bridge-E(P) (Server), Arrandale, Sandy bridge-M (Mobile)

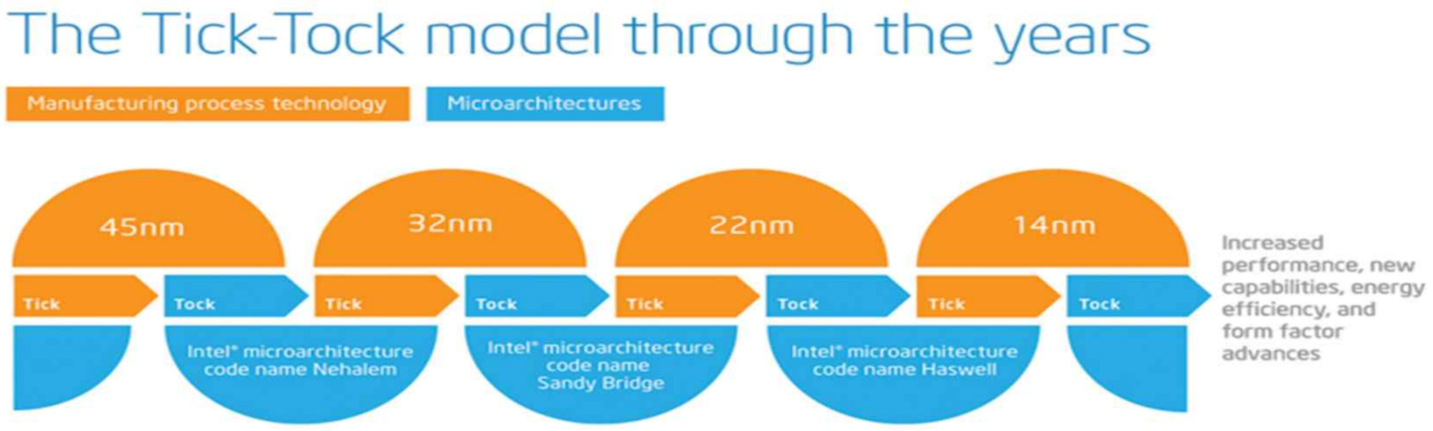

Enterprise: 2009 Nehalem Based
Four Socket System Architecture
Nehalem EX
Boxboro-EX I/O Hub
PCI Express Gen 2
Boxboro-EX Platform:
Intel QuickPath Interconnect
Four processors with Intel QuickPath Interconnects
PCI Express Gen 2, Integrated Memory Controller



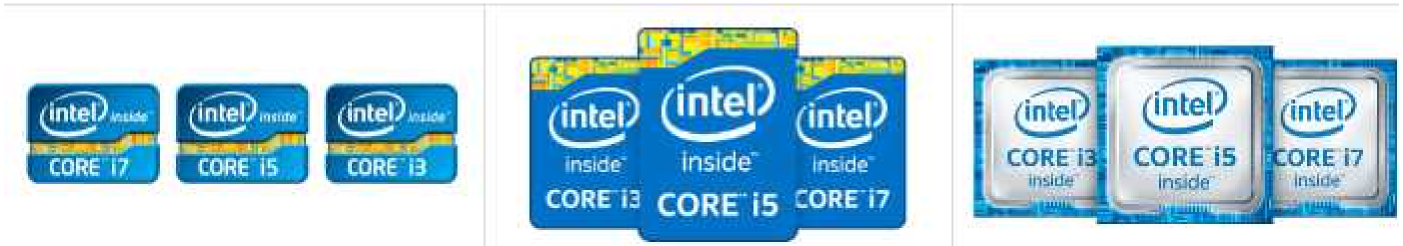


Intel CPU History (7/9)

- Intel tick-tock model
  - ✓ Tick: innovations in manufacturing process technology
  - ✓ Tock: innovations in processor microarchitecture


The Tick-Tock model through the years
Manufacturing process technology
Microarchitectures
45nm
32nm
22nm
14nm
Tick
Tock
Intel® microarchitecture code name Nehalem
Intel® microarchitecture code name Sandy Bridge
Intel® microarchitecture code name Haswell
Increased performance, new capabilities, energy efficiency, and form factor advances

(Source: http://www.intel.com/content/www/us/en/ silicon-innovations/intel-tick-tock-model-general.html)



(Intel Logo for Sandy Bridge, Haswell, and Sky lake. Source: http://namu.wiki)

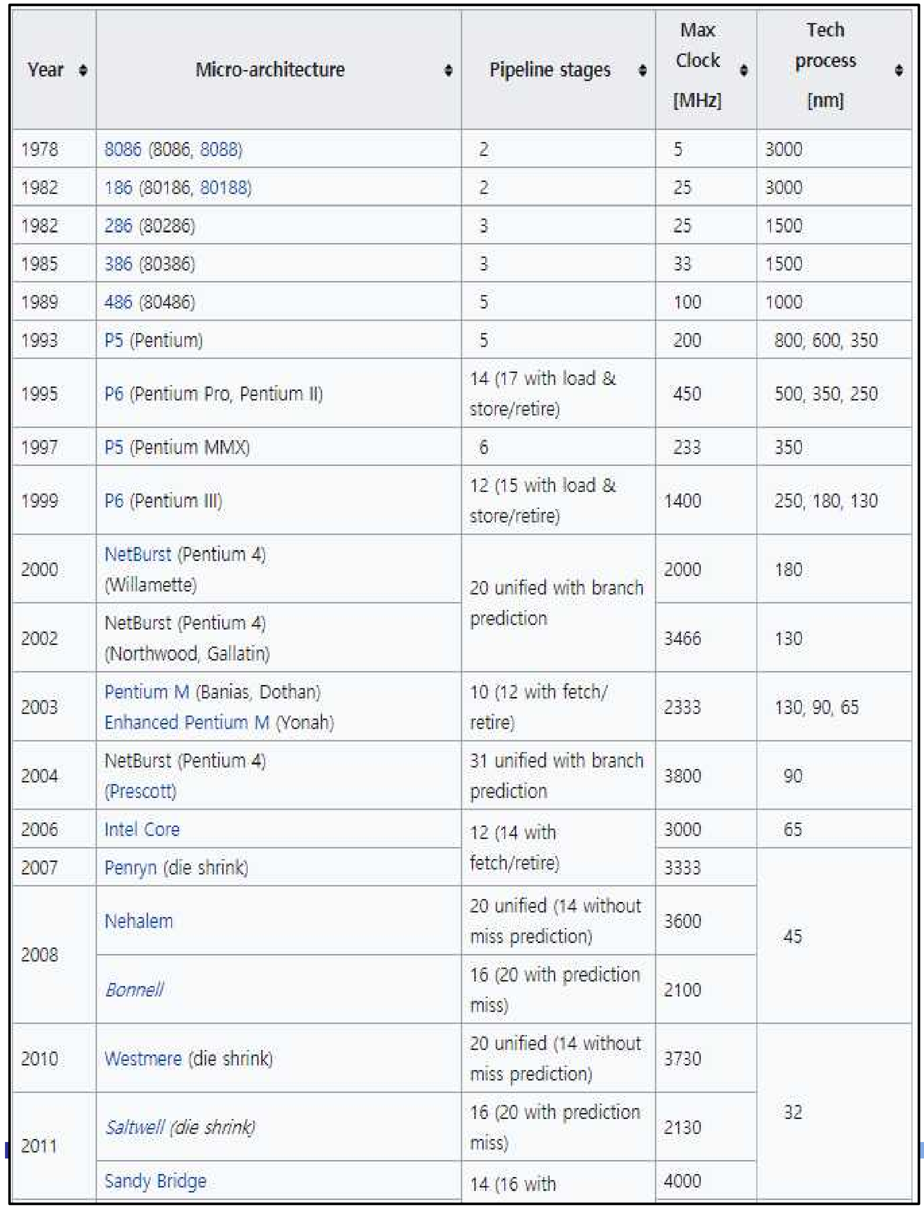
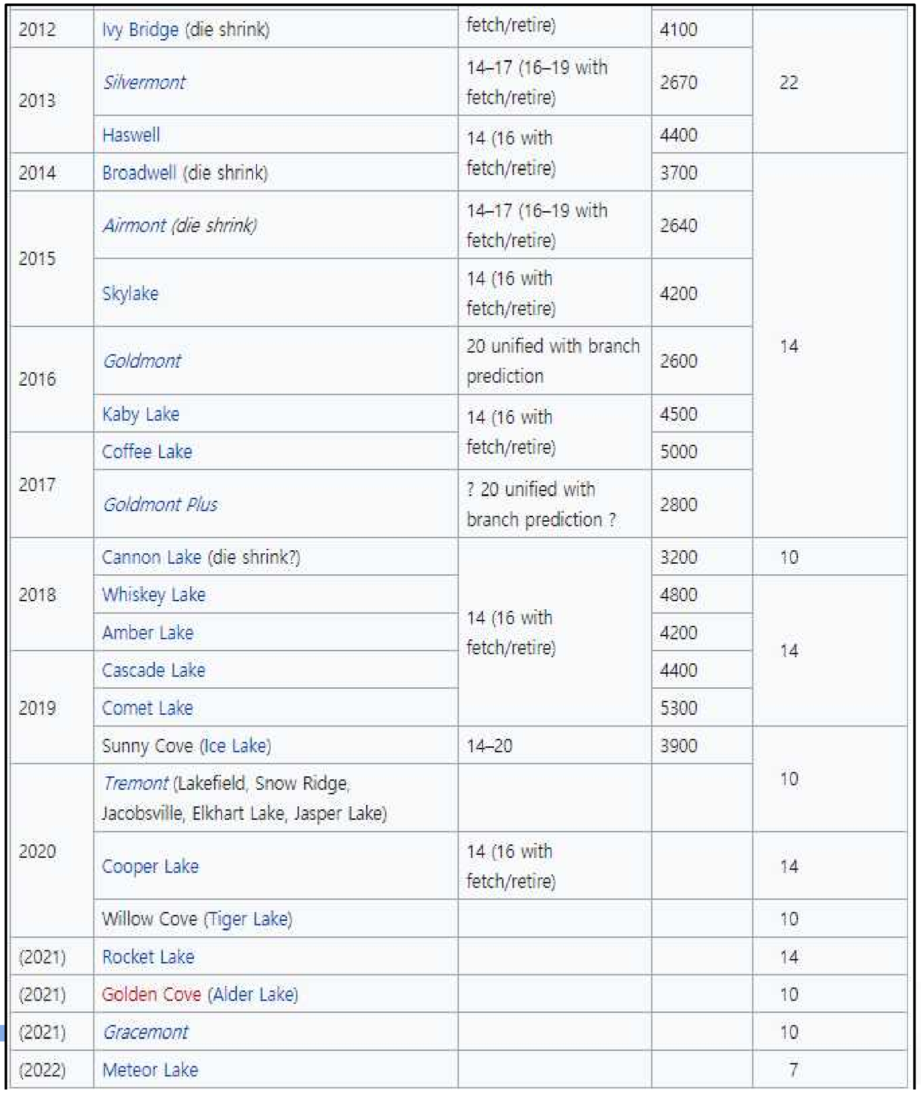
Intel CPU History (8/9)

- Intel CPU microarchitecture
  - ✓ From https://en.wikipedia.org/wiki/List_of_Intel_CPU_microarchitectures
  - ✓ Pre-P5: 1) 8086: first x86 processor, 2) 286: protected mode, 3) 386: 32-bit CPU, paging, 4) 486: FPU, pipeline, L1 cache
  - ✓ P5: Advanced pipeline, Superscalar, MMX
  - ✓ P6 (Pentium Pro, II, III): O3, SSE (Quite novel)
  - ✓ Netburst (Pentium 4, Xeon): Deep pipeline

  - ✓ Core (Core, Xeon): Mar. 2006, reengineered P6-based microarchitecture, 65nm, Multicore, (Tock ➔ Penryn: 45nm)
  - ✓ Nehalem (i3, i5, i7): 2008, 45nm, Integrated Memory Controller, QPI, (Tick ➔ Westmere: 32nm)
  - ✓ Sandy Bridge: 2011, 32nm, AVX, HW-support for video encoding and decoding, Encryption instruction set.(Tick ➔ Ivy Bridge: 22nm)
  - ✓ Haswell: 2013, 22nm, Integrated GPU, advanced power-saving (Tick ➔ Broadwell: 14nm)
  - ✓ Skylake: 2015, 14nm, DDR4 (64GB), PCI-e 3.0 (20 lane) (Optimization➔ kaby lake, Tick ➔ Cannon lake, 2018)
  - ✓ Sunny Cove (Ice lake): 2019, 10nm (Optimization ➔ Willow Cove (Tiger Lake), HW-accelerator such as SHA hash, security and AI featu

Intel CPU History (9/9)

- Intel CPU microarchitecture: summary

Year	Micro-architecture	Pipeline stages	Max Clock [MHz]	Tech process [nm]
1978	8086 (8086, 8088)	2	5	3000
1982	186 (80186, 80188)	2	25	3000
1982	286 (80286)	3	25	1500
1985	386 (80386)	3	33	1500
1989	486 (80486)	5	100	1000
1993	P5 (Pentium)	5	200	800, 600, 350
1995	P6 (Pentium Pro, Pentium II)	14 (17 with load & store/retire)	450	500, 350, 250
1997	P5 (Pentium MMX)	6	233	350
1999	P6 (Pentium III)	12 (15 with load & store/retire)	1400	250, 180, 130
2000	NetBurst (Pentium 4) (Willamette)	20 unified with branch prediction	2000	180
2002	NetBurst (Pentium 4) (Northwood, Gallatin)		3466	130
2003	Pentium M (Banias, Dothan) Enhanced Pentium M (Yonah)	10 (12 with fetch/ retire)	2333	130, 90, 65
2004	NetBurst (Pentium 4) (Prescott)	31 unified with branch prediction	3800	90
2006	Intel Core	12 (14 with fetch/retire)	3000	65
2007	Penryn (die shrink)		3333	45
2008	Nehalem	20 unified (14 without miss prediction)	3600	
	Bonnell	16 (20 with prediction miss)	2100	
2010	Westmere (die shrink)	20 unified (14 without miss prediction)	3730	32
2011	*Saltwell (die shrink)*	16 (20 with prediction miss)	2130	
	Sandy Bridge	14 (16 with fetch/retire)	4000	
2012	Ivy Bridge (die shrink)		4100	22
2013	*Silvermont*	14–17 (16–19 with fetch/retire)	2670	
	Haswell	14 (16 with fetch/retire)	4400	
2014	Broadwell (die shrink)		3700	14
2015	*Airmont (die shrink)*	14–17 (16–19 with fetch/retire)	2640	
	Skylake	14 (16 with fetch/retire)	4200	
2016	*Goldmont*	20 unified with branch prediction	2600	
	Kaby Lake	14 (16 with fetch/retire)	4500	
2017	Coffee Lake		5000	
	Goldmont Plus	? 20 unified with branch prediction ?	2800	
2018	Cannon Lake (die shrink?)	14 (16 with fetch/retire)	3200	10
	Whiskey Lake		4800	14
	Amber Lake		4200	
2019	Cascade Lake		4400	
	Comet Lake		5300	
	Sunny Cove (Ice Lake)	14–20	3900	10
2020	*Tremont* (Lakefield, Snow Ridge, Jacobsville, Elkhart Lake, Jasper Lake)			
	Cooper Lake	14 (16 with fetch/retire)		14
	Willow Cove (Tiger Lake)			10
(2021)	Rocket Lake			14
(2021)	Golden Cove (Alder Lake)			10
(2021)	*Gracemont*			10
(2022)	Meteor Lake			7

Technologies of Intel CPU (1/12)

- What processor do?

Instruction type	**Dynamic usage**
Data movement	43%
Control flow	23%
Arithmetic operations	15%
Comparisons	13%
Logic operations	5%
Other	1%

✓ Data movement needs to be optimized

➔ CPU cache, write buffer

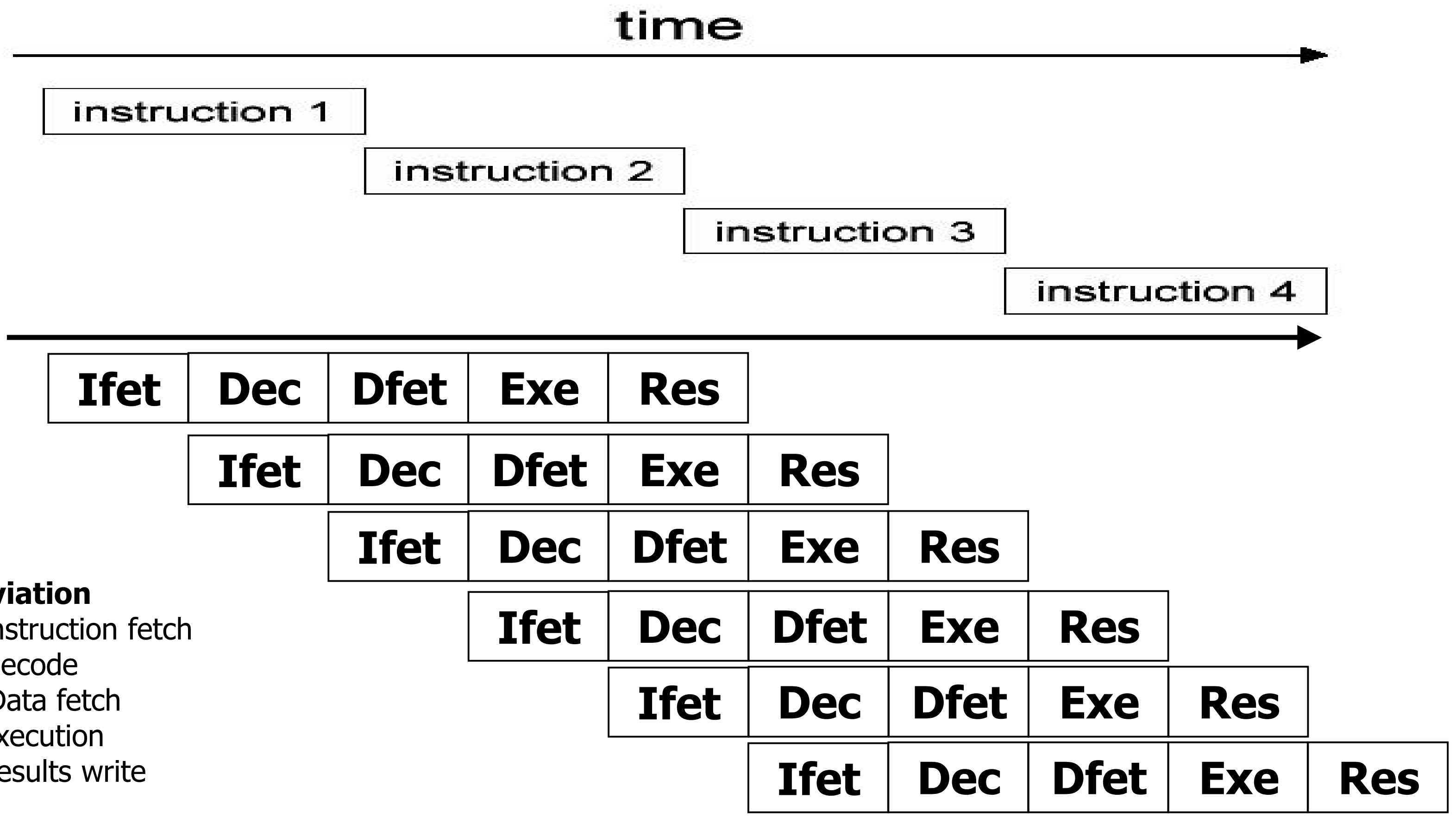
✓ Some components are idle while executing instruction

➔ Pipelining

➔ Superscalar

Technologies of Intel CPU (2/12)

- Pipeline
  - ✓ Execution of an instruction is divided into multiple stages
  - ✓ Overlapping execution of multiple instructions

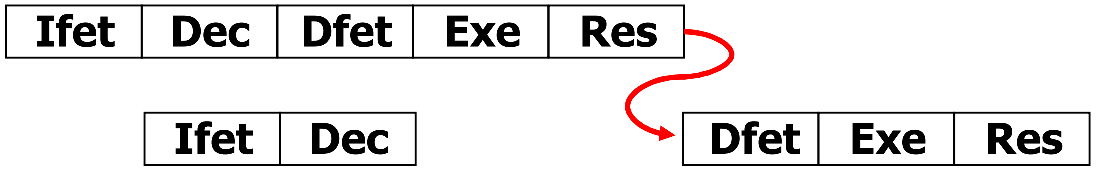


Abbreviation
- •Ifet: Instruction fetch
- •Dec: Decode
- •Dfet: Data fetch
- •Exe: Execution
- •Res: Results write

➔ latency vs. throughput

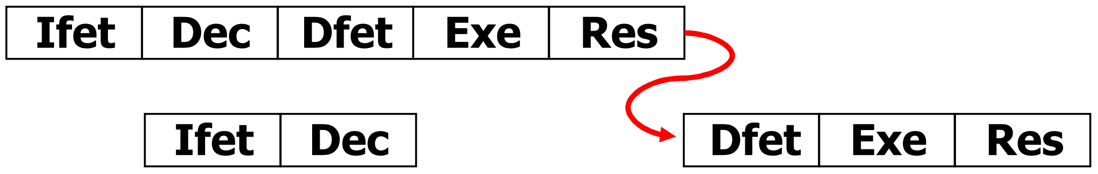
Technologies of Intel CPU (3/12)

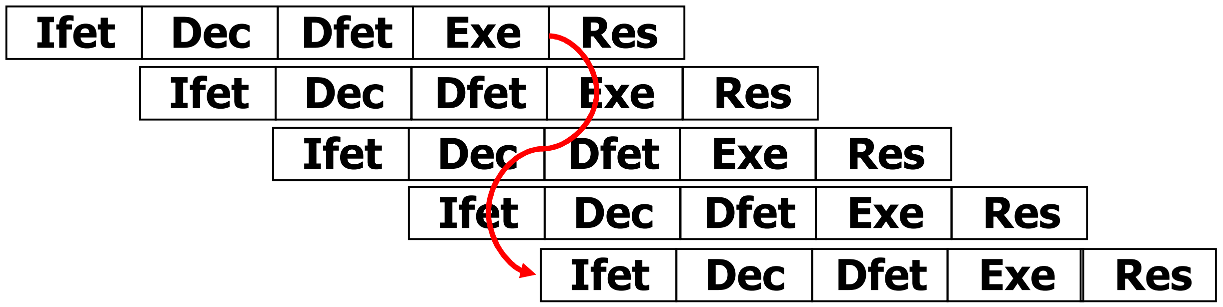
- For the efficiency of Pipelining (no free lunch)
  - ✓ All instructions should have similar execution time (simple format)
    - RISC (addl a, b   vs. movl a, %eax; addl b, %eax; movl %eax, b)
  - ✓ CPU components are independent each other ➔ I/D cache
  - ✓ No resource conflict (sharing at the same time) ➔ dual component
  - ✓ Overcome pipeline hazard (data, control)



Technologies of Intel CPU (3/12)

- For the efficiency of Pipelining (no free lunch)
  - ✓ All instructions should have similar execution time (simple format)
    - RISC (addl a, b   vs. movl a, %eax; addl b, %eax; movl %eax, b)
  - ✓ CPU components are independent each other ➔ I/D cache
  - ✓ No resource conflict (sharing at the same time) ➔ dual component
  - ✓ Overcome pipeline hazard (data, control)





Technologies of Intel CPU (4/12)

- Techniques for overcome pipeline hazard
  - ✓ Compiler optimization
    - Instruction reordering
    - Loop unrolling
  - ✓ Branch prediction
    - Static prediction
    - Dynamic prediction
  - ✓ Out of order execution
    - Dynamic reordering with data flow analysis
  - ✓ Speculative execution and retirement
  - ✓ Register renaming

Technologies of Intel CPU (5/12)

- P6 microarchitecture revisit
  - ✓ Dynamic execution
    - Out-of-order execution
    - Branch prediction
    - Speculative execution: decouple execution and commitment (retirement unit)
    - Data flow analysis: detect independent instructions on real time
    - Register renaming
  - ✓ Pipelined (12 stage) architecture, 3-way superscalar
  - ✓ L1 cache and L2 cache

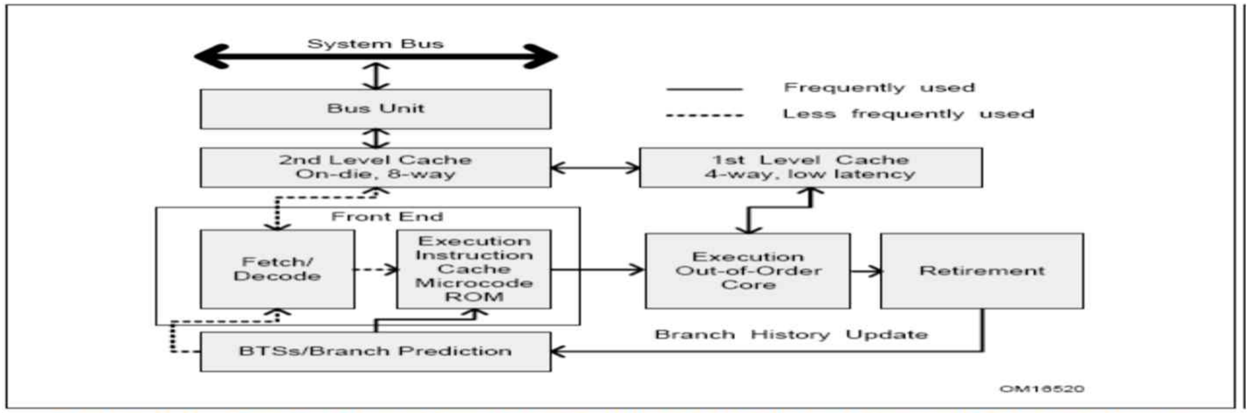


Figure 2-1. The P6 Processor Microarchitecture with Advanced Transfer Cache Enhancement

Quiz for 12th-Week 1st-Lesson

- Quiz
  - ✓ 1. What are the data hazard and control hazard?
  - ✓ 2. What are the Meltdown and Spectre vulnerabilities?
  - ✓ Due: until 6 PM Friday of this week (20th, November)

```
# prog1
0x000: irmovl $10,%edx
0x006: irmovl  $3,%eax
0x00c: nop
0x00d: nop
0x00e: nop
0x00f: addl %edx,%eax
0x011: halt
```

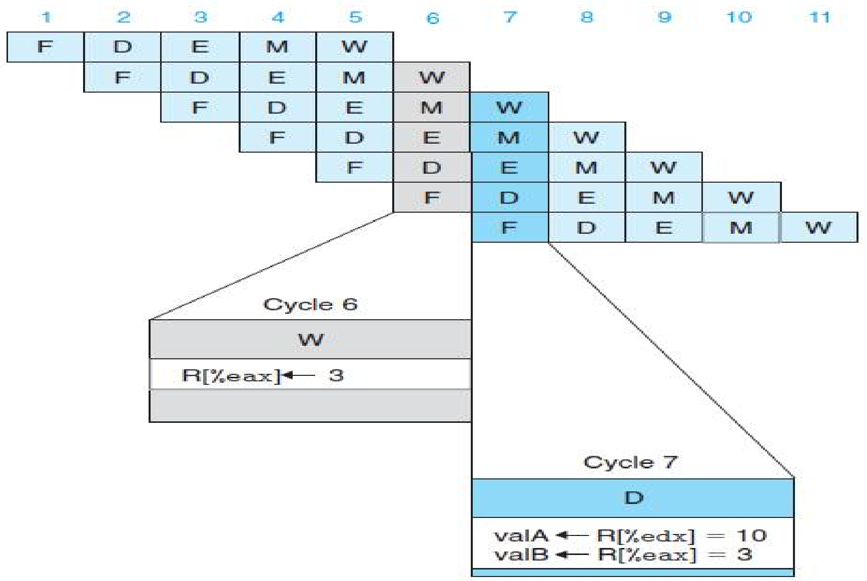


Figure 4.43 **Pipelined execution of prog1 without special pipeline control.** In cycle 6, the second irmovl writes its result to program register %eax. The addl instruction reads its source operands in cycle 7, so it gets correct values for both %edx and %eax.

with data hazards. Control hazards will be discussed as part of the overall pipeline control (Section 4.5.11).

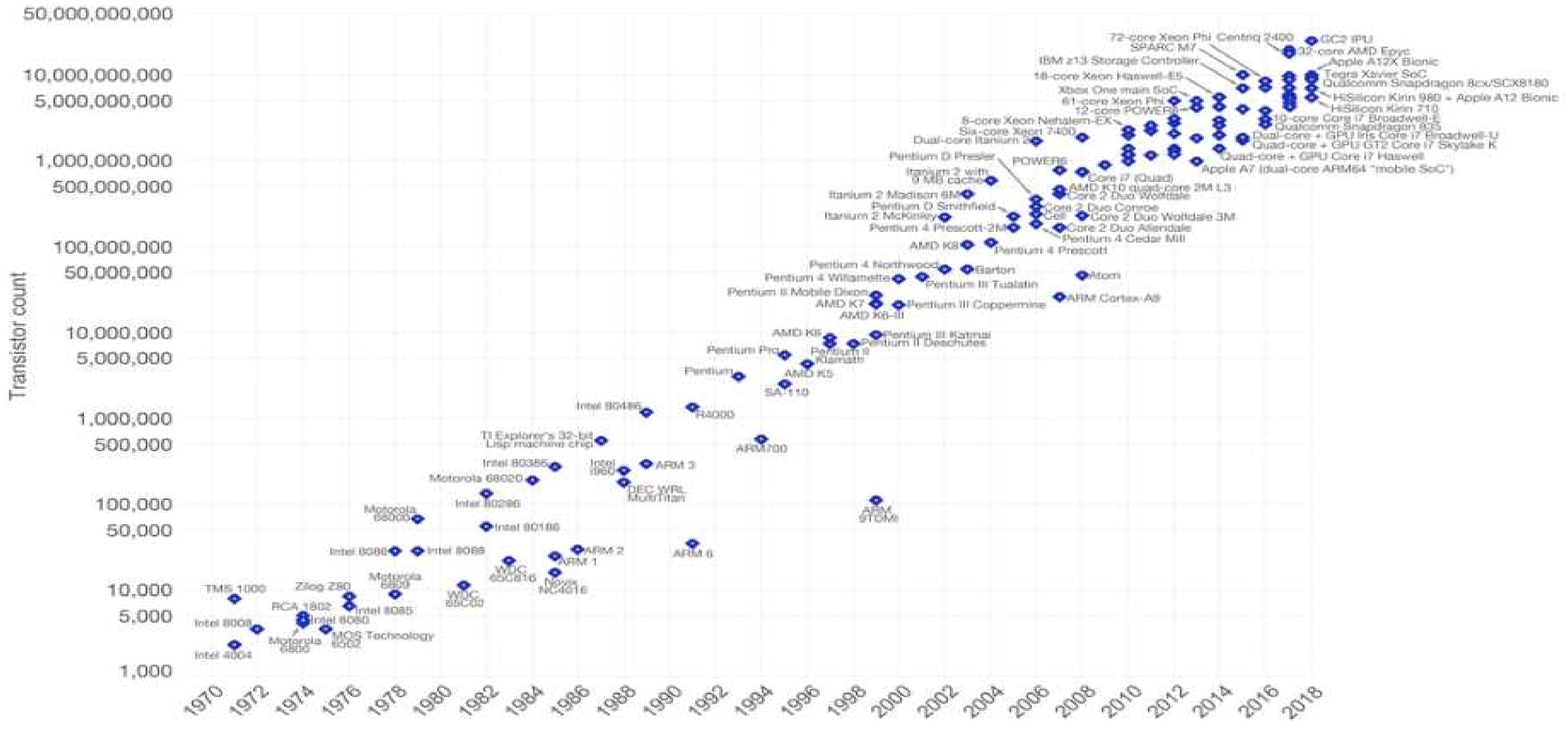
Technologies of Intel CPU (6/12)

- Moore’s law

Moore’s Law – The number of transistors on integrated circuit chips (1971-2018)

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are linked to Moore's law.

Our World in Data

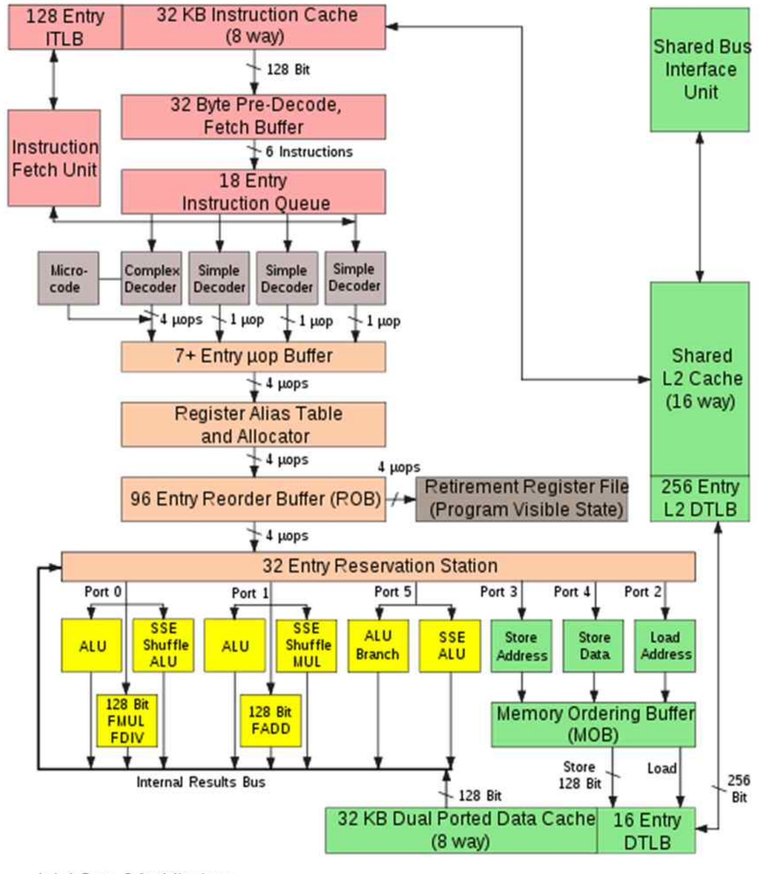


Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

(Source: https://en.wikipedia.org/wiki/Moore%27s_law)

Technologies of Intel CPU (7/12)

- Trend
  - ✓ Increasing available transistors: multi components, multi channels
  - ✓ Superscalar
  - ✓ Multimedia support: SIMD
    - MMX technology
    - SSE
    - SSE2/3, AVX
  - ✓ Hyper threading
  - ✓ 64-bit Supporting
    - IA64 (EPIC)
    - Intel 64
  - ✓ Multicore



Intel Core 2 Architecture

(From http://en.wikipedia.org/wiki/File:Intel_Core2_arch.svg)

Technologies of Intel CPU (8/12)

- SIMD instructions
  - ✓ A group of instructions can be performed in parallel
  - ✓ Using MMX (64), XMM(128), YMM(256) registers
  - ✓ MMX
    - integer
  - ✓ SSE (Pentium 3)
    - Streaming SIMD Extension
    - Single precision floating point
  - ✓ SSE2 (Pentium 4)
    - Double precision floating point
  - ✓ SSE3 (Pentium 4)
    - HT support
    - 13 new SIMD instructions
  - ✓ AVX (Sandy Bridge)
    - Advanced Vector Extension
    - From Sandy Bridge, 256 bit (YMM)

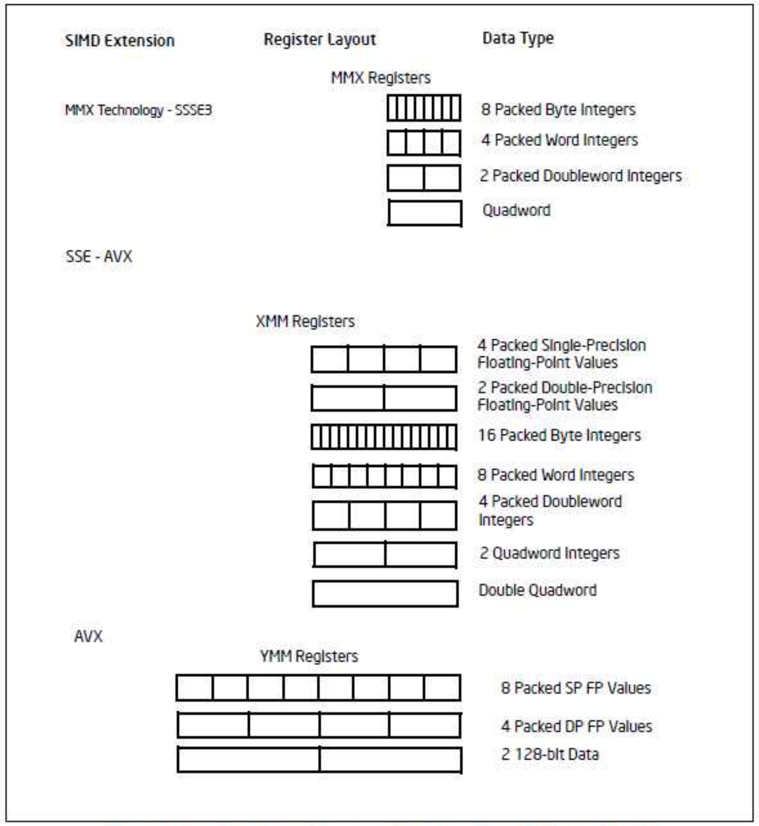


Figure 2-4. SIMD Extensions, Register Layouts, and Data Types

Technologies of Intel CPU (9/12)

- Hyper threading Technology
  - ✓ Support multi-threading operating system
  - ✓ 2 or more separated code streams using shared execution resources

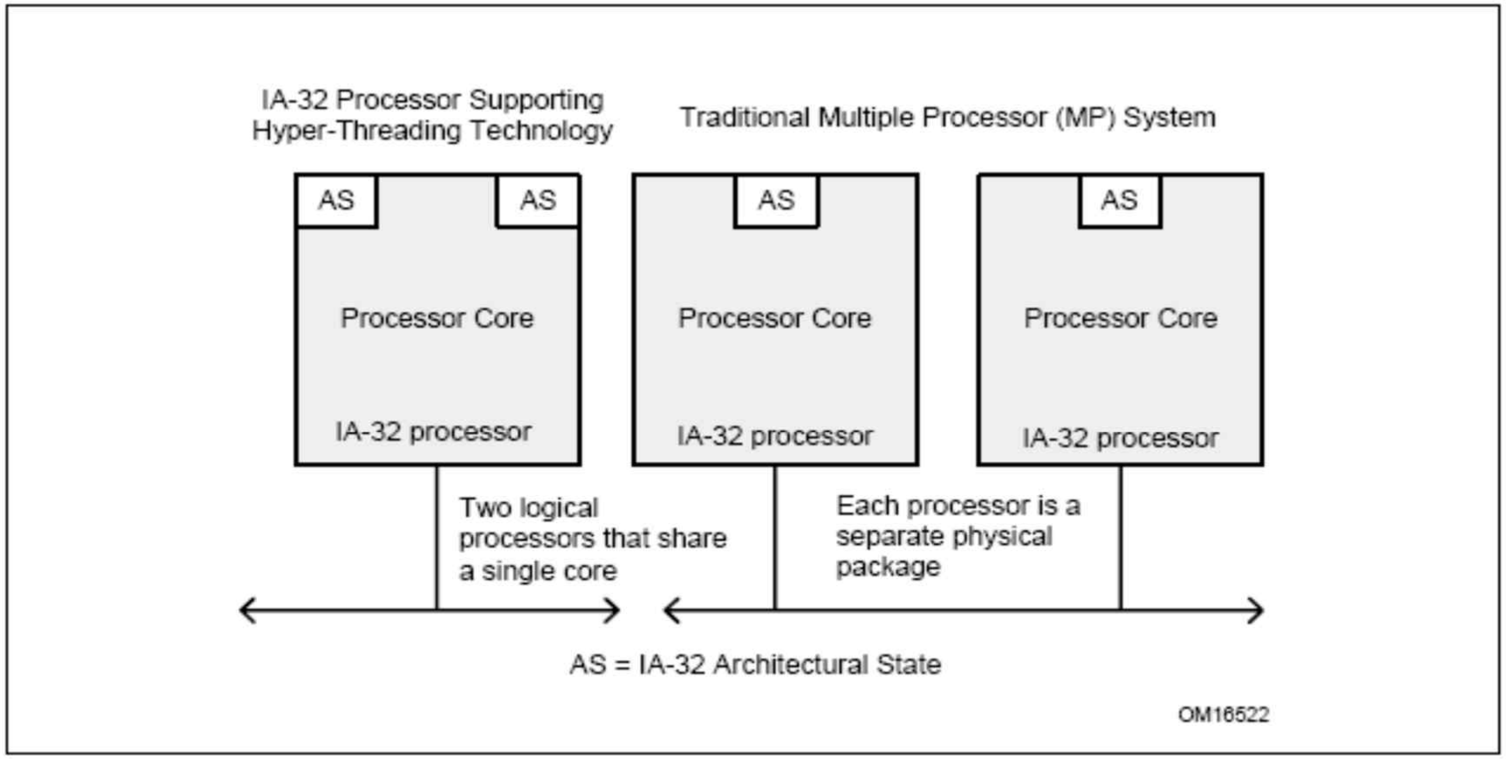

IA-32 Processor Supporting Hyper-Threading Technology
Traditional Multiple Processor (MP) System
AS
AS
AS
AS
Processor Core
Processor Core
Processor Core
IA-32 processor
IA-32 processor
IA-32 processor
Two logical processors that share a single core
Each processor is a separate physical package
AS = IA-32 Architectural State
OM16522

Figure 2-5. Comparison of an IA-32 Processor Supporting Hyper-Threading Technology and a Traditional Dual Processor System

Technologies of Intel CPU (10/12)

- Multi core Technology
  - ✓ Intel Pentium D: dual core based on two Pentium 4 (without HT)
  - ✓ Intel Core Duo, Core 2 Duo: dual core with shared bus interface (dual core performance with low cost)
  - ✓ Intel Core 2 Quad Processor: Duplicated Core Duo, Core 2 Duo
    - Extreme edition: multi-core with multi architectural states (with HT)
  - ✓ Intel Core i7: Quick Path Interconnect, L3, IMC,

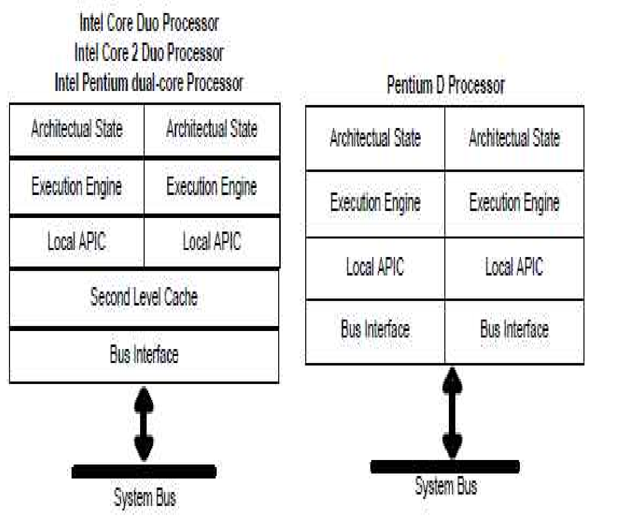


Figure 2-6. Intel 64 and IA-32 Processors that Support Dual-Core

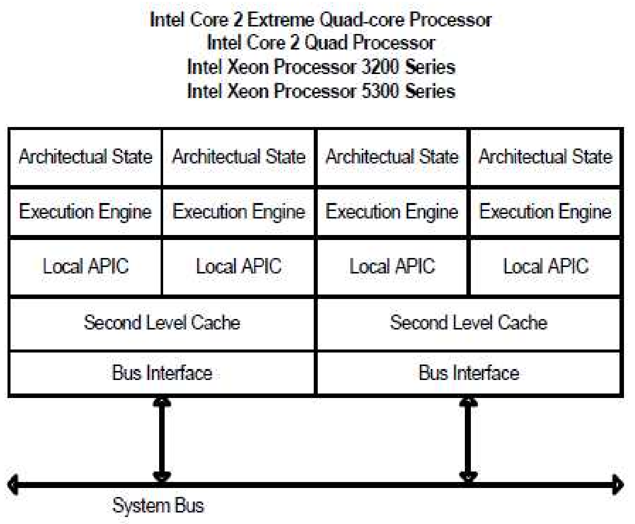


Figure 2-7. Intel 64 Processors that Support Quad-Core

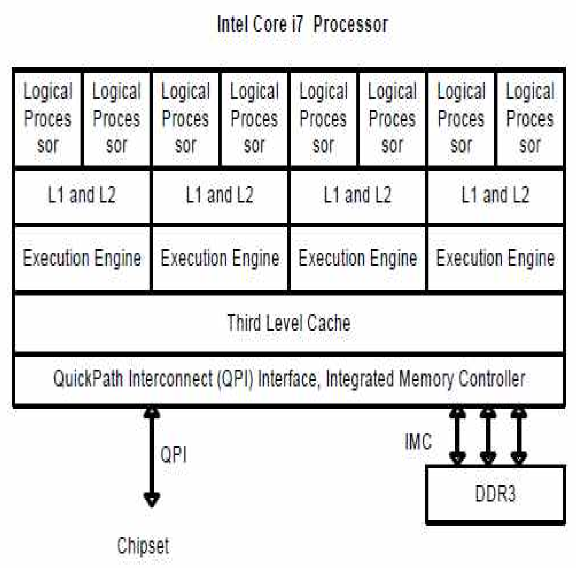


Figure 2-8. Intel Core i7 Processor

Technologies of Intel CPU (11/12)

- Intel 64
  - ✓ Support 64bit address extension: EM64T (Extended Memory 64 Technology), x86-64, IA-32e
  - ✓ new operation modes
  - ✓ new/enhanced register sets
  - ✓ new/enhanced instruction sets
  - ✓ 64bit address translation

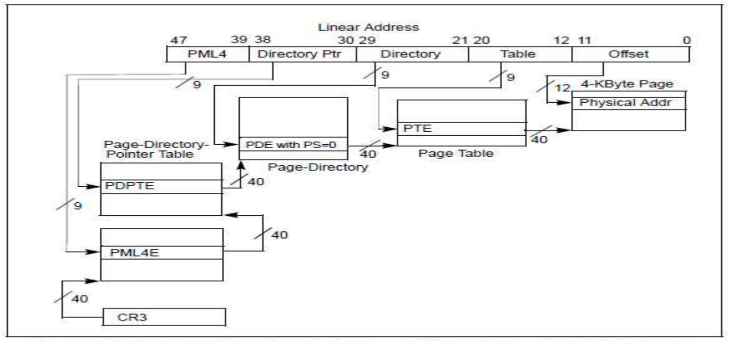
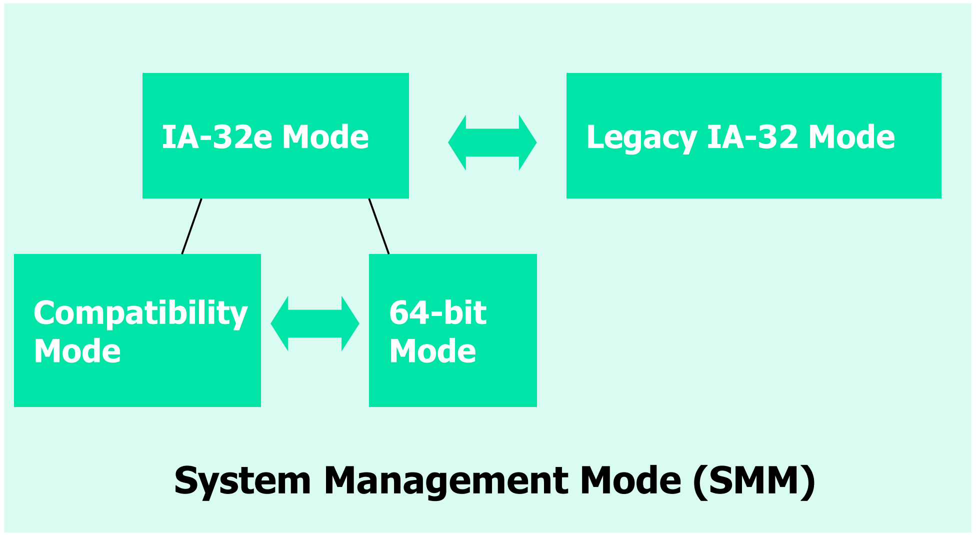
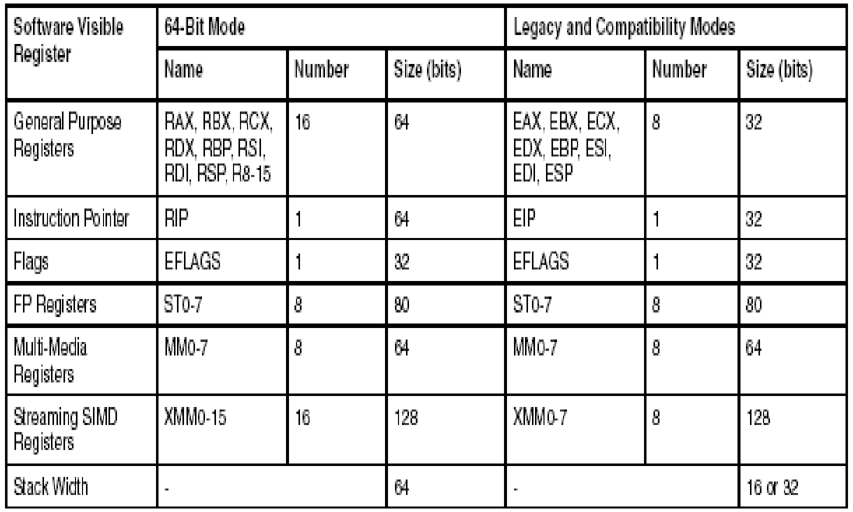


Figure 4-8. Linear-Address Translation to a 4-KByte Page using IA-32e Paging

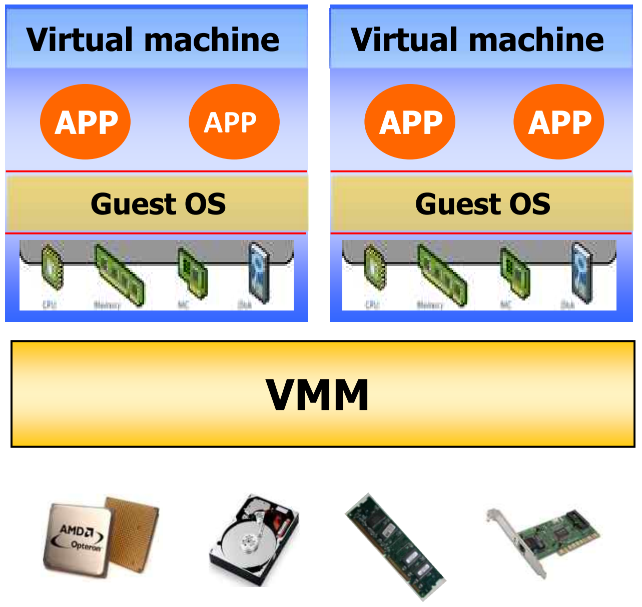


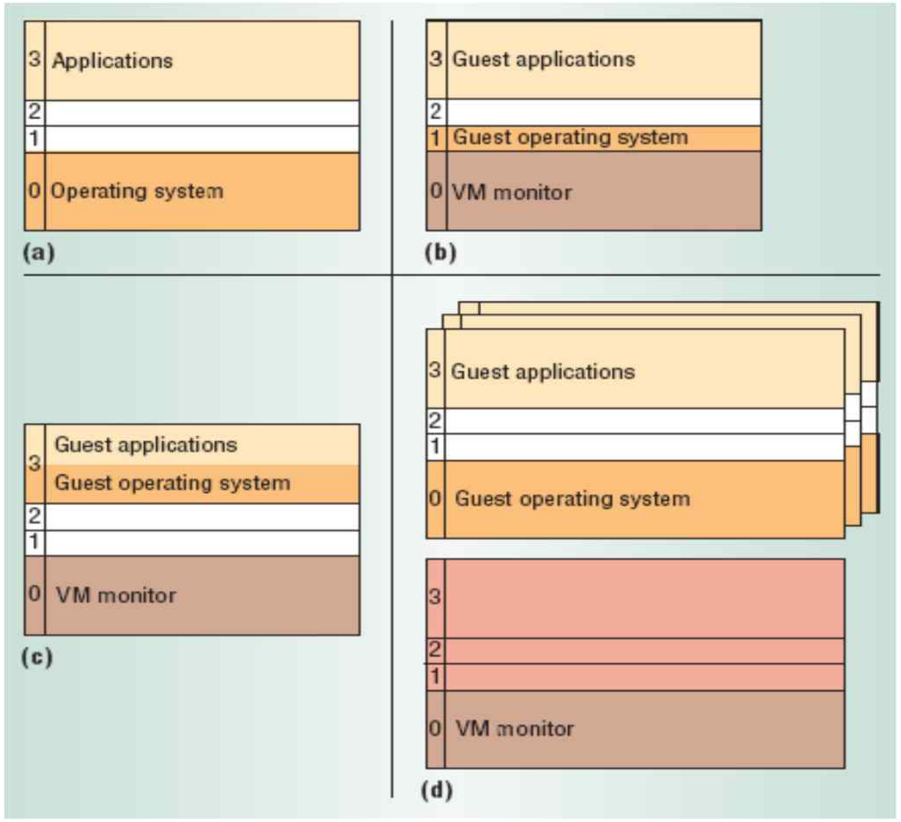


Software Visible Register	64-Bit Mode			Legacy and Compatibility Modes		
	Name	Number	Size (bits)	Name	Number	Size (bits)
General Purpose Registers	RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, R8-15	16	64	EAX, EBX, ECX, EDX, EBP, ESI, EDI, ESP	8	32
Instruction Pointer	RIP	1	64	EIP	1	32
Flags	EFLAGS	1	32	EFLAGS	1	32
FP Registers	ST0-7	8	80	ST0-7	8	80
Multi-Media Registers	MM0-7	8	64	MM0-7	8	64
Streaming SIMD Registers	XMM0-15	16	128	XMM0-7	8	128
Stack Width	-		64	-		16 or 32

Technologies of Intel CPU (12/12)

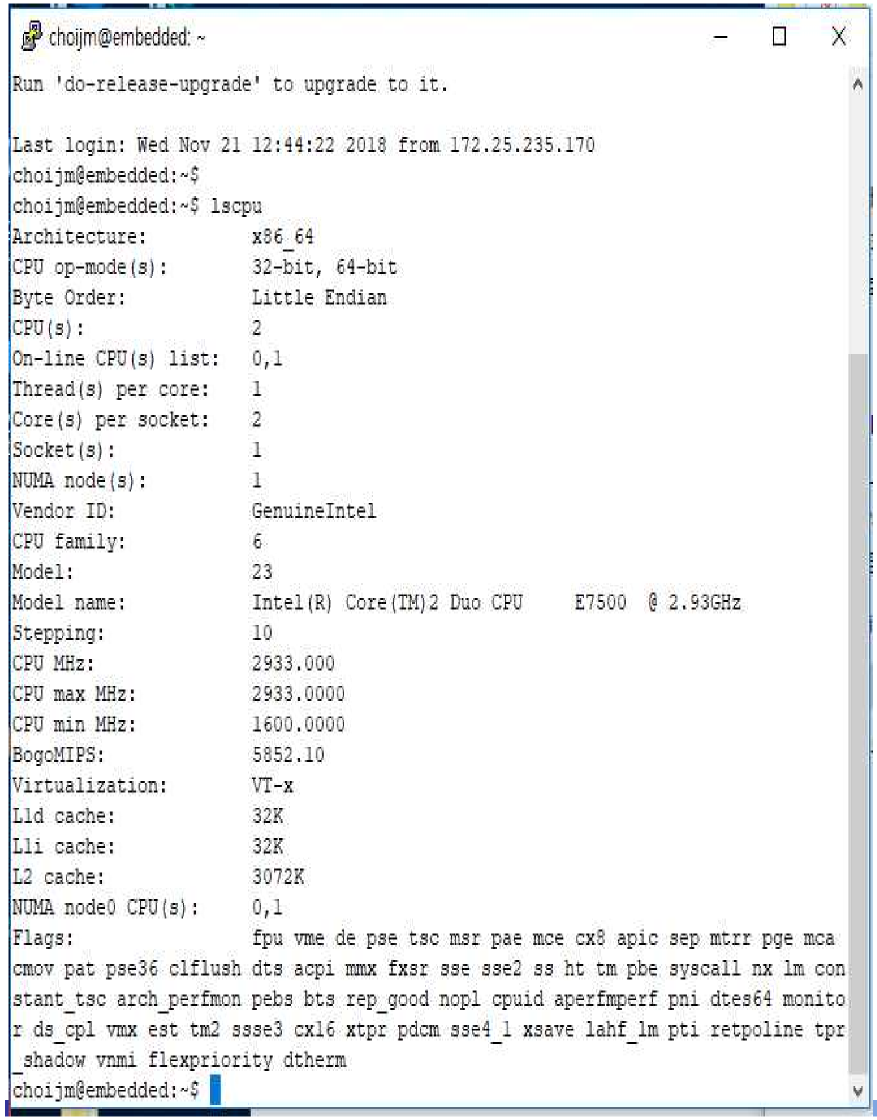
- VT (Virtualization Technology)
  - ✓ VMX (Virtual Machine Extension)
    - Direct execution
    - New privilege level





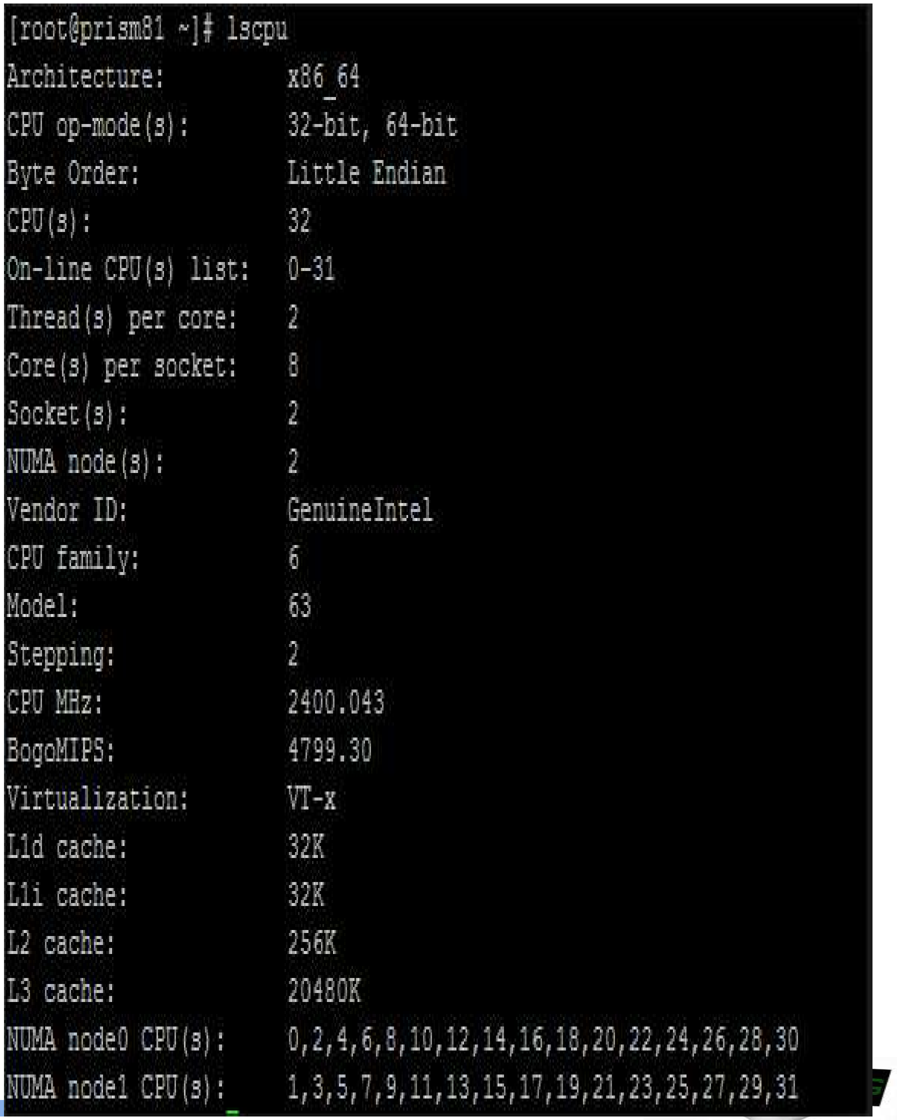
CPU information in Linux

- lscpu



choijm@embedded: ~

```
Run 'do-release-upgrade' to upgrade to it.

Last login: Wed Nov 21 12:44:22 2018 from 172.25.235.170
choijm@embedded:~$
choijm@embedded:~$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                2
On-line CPU(s) list:   0,1
Thread(s) per core:    1
Core(s) per socket:    2
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 23
Model name:            Intel(R) Core(TM)2 Duo CPU     E7500  @ 2.93GHz
Stepping:              10
CPU MHz:               2933.000
CPU max MHz:           2933.0000
CPU min MHz:           1600.0000
BogoMIPS:              5852.10
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              3072K
NUMA node0 CPU(s):     0,1
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx lm con
stant_tsc arch_perfmon pebs bts rep_good nopl cpuid aperfmperf pni dtes64 monito
r ds_cpl vmx est tm2 ssse3 cx16 xtpr pdcm sse4_1 xsave lahf_lm pti retpoline tpr
_shadow vnmi flexpriority dtherm
choijm@embedded:~$
```



```
[root@prism81 ~]# lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                32
On-line CPU(s) list:   0-31
Thread(s) per core:    2
Core(s) per socket:    8
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 63
Stepping:              2
CPU MHz:               2400.043
BogoMIPS:              4799.30
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              20480K
NUMA node0 CPU(s):     0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30
NUMA node1 CPU(s):     1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31
```

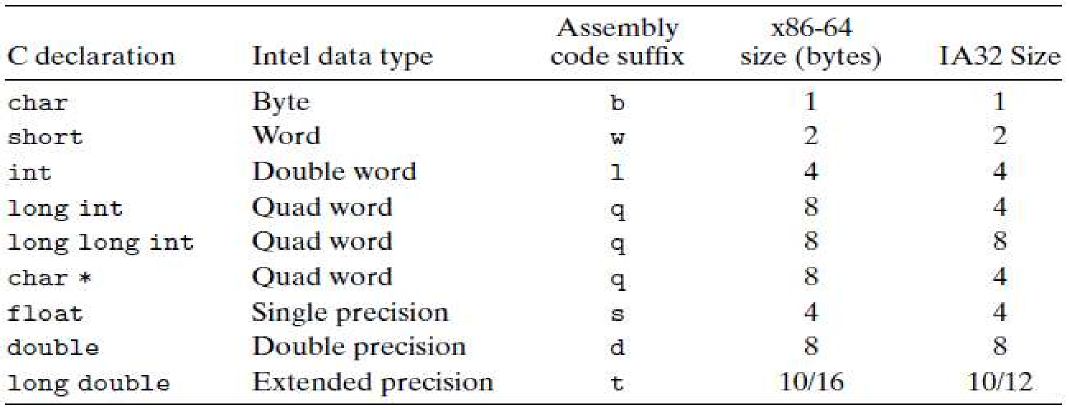
x86-64: extending IA32 to 64-bit CPU (1/4)

- From IA32 to Intel64 (a.k.a. x86 and x86-64, respectively)
  - ✓ Intel traditional ISA: called as IA32
    - Start at 1985 (80386)
    - Evolution: add new instructions (e.g. conditional move), also keep backward compatibility
  - ✓ New Intel ISA for 64-bit CPU: called as IA64
    - Totally new ISAs called EPIC (Explicitly Parallel Instruction Computing) ➔ MIMD
    - Market name: Itanium (2001)
  - ✓ AMD ISA for 64-bit CPU: called x86-64
    - Compatible with IA32 ➔ win at the market
    - Intel follows: Intel64
    - AMD renames AMD64 (but x86-64 "persists as a favored name")

(Source: https://www.extremetech.com/extreme/167168-the-chip-that-changed-the-world-amds-64-bit-fx-51-ten-years-later/2)

x86-64: extending IA32 to 64-bit CPU (2/4)

- Features of x86-64
  - ✓ New data type
    - Pointer becomes 8 bytes
  - ✓ Make use of RISC techniques
    - 8 GPR ➔ 16 GPR
    - Register based arguments passing
  - ✓ 2^{64} address space (2^{48} in practical)
  - ✓ Backward compatible
    - Can run existing SW in compatible mode



C declaration	Intel data type	Assembly code suffix	x86-64 size (bytes)	IA32 Size
`char`	Byte	b	1	1
`short`	Word	w	2	2
`int`	Double word	l	4	4
`long int`	Quad word	q	8	4
`long long int`	Quad word	q	8	8
`char *`	Quad word	q	8	4
`float`	Single precision	s	4	4
`double`	Double precision	d	8	8
`long double`	Extended precision	t	10/16	10/12

Figure 3.34 Sizes of standard data types with x86-64. These are compared to the sizes for IA32. Both long integers and pointers require 8 bytes, as compared to 4 for IA32.

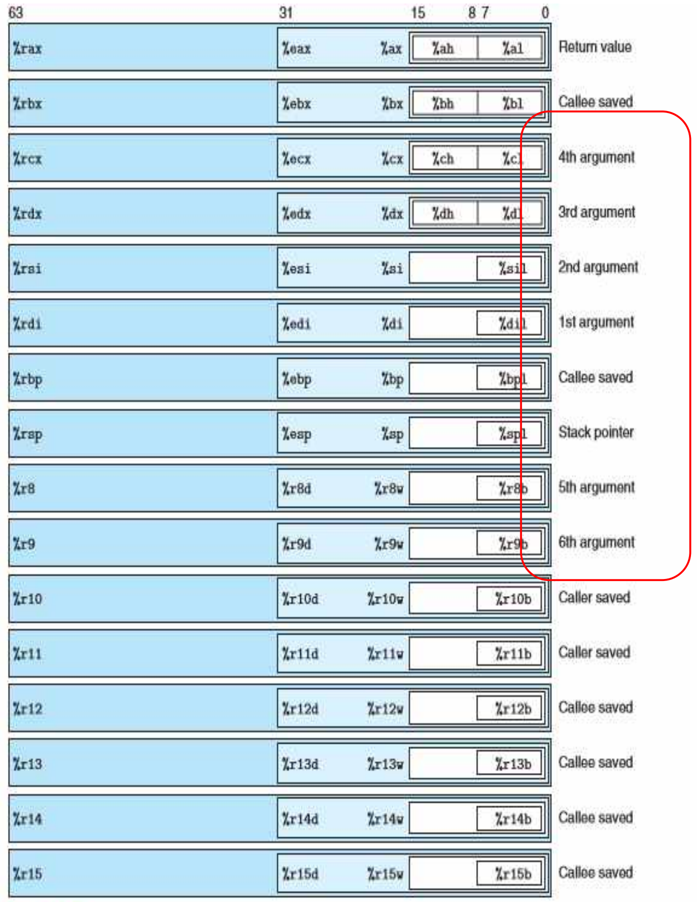


Figure 3.35 **Integer registers.** The existing eight registers are extended to 64-bit versions, and eight new registers are added. Each register can be accessed as either 8 bits (byte), 16 bits (word), 32 bits (double word), or 64 bits (quad word).

x86-64: extending IA32 to 64-bit CPU (3/4)

- Assembly code example1
  - ✓ Syntax: 1) rax instead of eax, 2) movq instead of movl, 3) argument passing using registers, 4) No stack frame if possible, 5) make use of PIC (Position Independent Code), …
    - Register passing ➔7 memory references vs. 3 memory references

```
long int simple_l(long int *xp, long int y)
{
    long int t = *xp + y;
    *xp = t;
    return t;
}
```

IA32 implementation of function simple_l.
xp at %ebp+8, y at %ebp+12

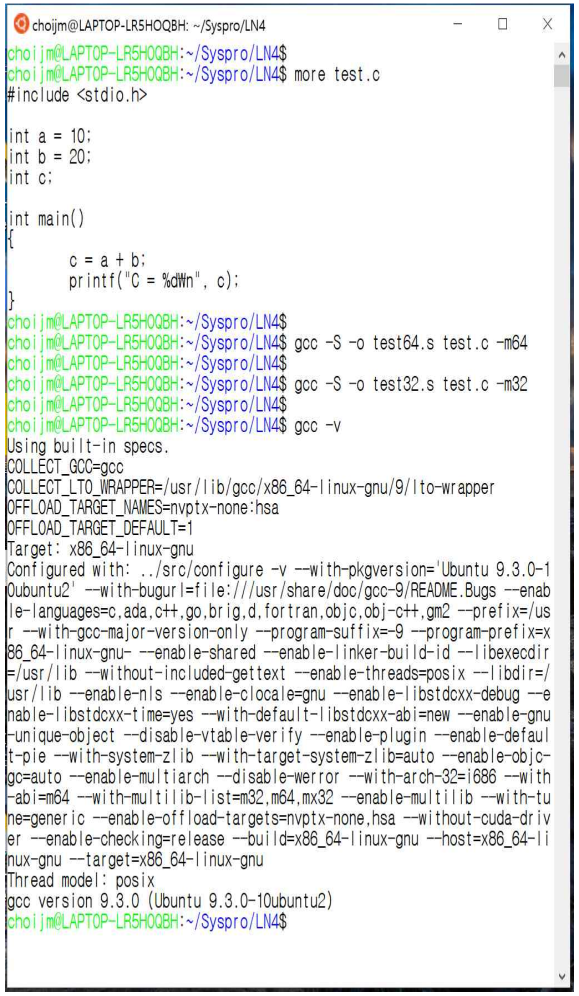
```
simple_l:
  pushl   %ebp             Save frame pointer        (W)
  movl    %esp, %ebp       Create new frame pointer
  movl    8(%ebp), %edx    Retrieve xp               (R)
  movl    12(%ebp), %eax   Retrieve yp               (R)
  addl    (%edx), %eax     Add *xp to get t          (R)
  movl    %eax, (%edx)     Store t at xp             (W)
  popl    %ebp             Restore frame pointer     (R)
  ret                      Return                    (R)
```

x86-64 version of function simple_l.
xp in %rdi, y in %rsi

```
simple_l:
  movq    %rsi, %rax      Copy y
  addq    (%rdi), %rax    Add *xp to get t   (R)
  movq    %rax, (%rdi)    Store t at xp      (W)
  ret                     Return             (R)
```

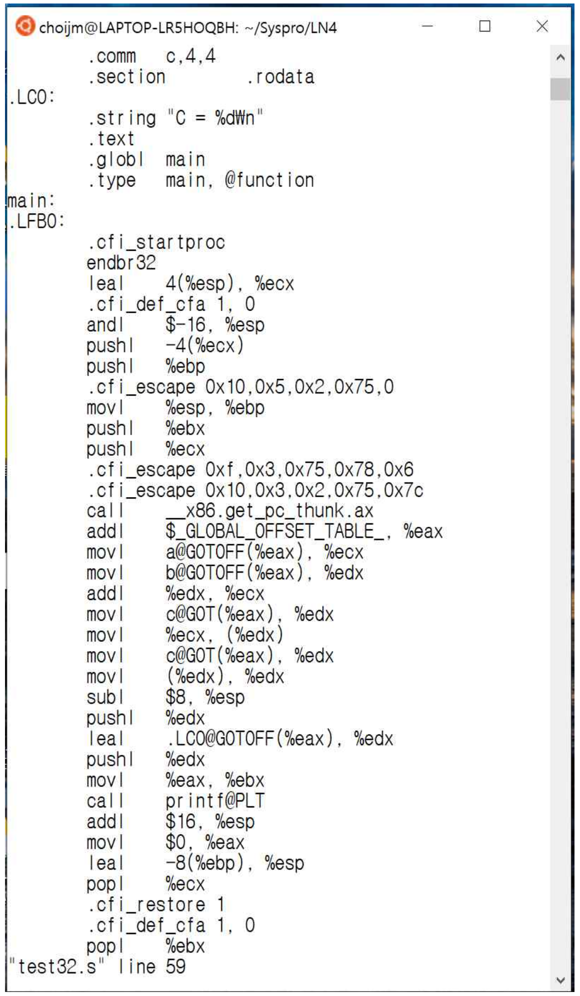
x86-64: extending IA32 to 64-bit CPU (4/4)

- Assembly code example2



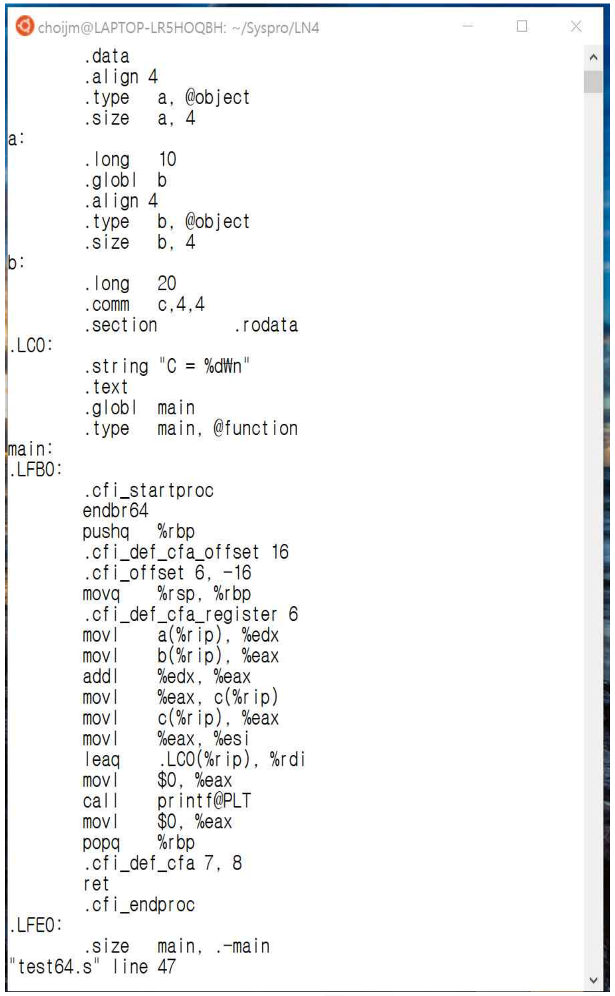
choijm@LAPTOP-LR5HOQBH: ~/Syspro/LN4

```
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$ more test.c
#include <stdio.h>

int a = 10;
int b = 20;
int c;

int main()
{
        c = a + b;
        printf("C = %d₩n", c);
}
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$ gcc -S -o test64.s test.c -m64
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$ gcc -S -o test32.s test.c -m32
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$ gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/9/lto-wrapper
OFFLOAD_TARGET_NAMES=nvptx-none:hsa
OFFLOAD_TARGET_DEFAULT=1
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 9.3.0-1
0ubuntu2' --with-bugurl=file:///usr/share/doc/gcc-9/README.Bugs --enab
le-languages=c,ada,c++,go,brig,d,fortran,objc,obj-c++,gm2 --prefix=/us
r --with-gcc-major-version-only --program-suffix=-9 --program-prefix=x
86_64-linux-gnu- --enable-shared --enable-linker-build-id --libexecdir
=/usr/lib --without-included-gettext --enable-threads=posix --libdir=/
usr/lib --enable-nls --enable-clocale=gnu --enable-libstdcxx-debug --e
nable-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-gnu
-unique-object --disable-vtable-verify --enable-plugin --enable-defaul
t-pie --with-system-zlib --with-target-system-zlib=auto --enable-objc-
gc=auto --enable-multiarch --disable-werror --with-arch-32=i686 --with
-abi=m64 --with-multilib-list=m32,m64,mx32 --enable-multilib --with-tu
ne=generic --enable-offload-targets=nvptx-none,hsa --without-cuda-driv
er --enable-checking=release --build=x86_64-linux-gnu --host=x86_64-li
nux-gnu --target=x86_64-linux-gnu
Thread model: posix
gcc version 9.3.0 (Ubuntu 9.3.0-10ubuntu2)
choijm@LAPTOP-LR5HOQBH:~/Syspro/LN4$
```



choijm@LAPTOP-LR5HOQBH: ~/Syspro/LN4

```
        .comm   c,4,4
        .section        .rodata
.LC0:
        .string "C = %d₩n"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        endbr32
        leal    4(%esp), %ecx
        .cfi_def_cfa 1, 0
        andl    $-16, %esp
        pushl   -4(%ecx)
        pushl   %ebp
        .cfi_escape 0x10,0x5,0x2,0x75,0
        movl    %esp, %ebp
        pushl   %ebx
        pushl   %ecx
        .cfi_escape 0xf,0x3,0x75,0x78,0x6
        .cfi_escape 0x10,0x3,0x2,0x75,0x7c
        call    __x86.get_pc_thunk.ax
        addl    $_GLOBAL_OFFSET_TABLE_, %eax
        movl    a@GOTOFF(%eax), %ecx
        movl    b@GOTOFF(%eax), %edx
        addl    %edx, %ecx
        movl    c@GOT(%eax), %edx
        movl    %ecx, (%edx)
        movl    c@GOT(%eax), %edx
        movl    (%edx), %edx
        subl    $8, %esp
        pushl   %edx
        leal    .LC0@GOTOFF(%eax), %edx
        pushl   %edx
        movl    %eax, %ebx
        call    printf@PLT
        addl    $16, %esp
        movl    $0, %eax
        leal    -8(%ebp), %esp
        popl    %ecx
        .cfi_restore 1
        .cfi_def_cfa 1, 0
        popl    %ebx
"test32.s" line 59
```



choijm@LAPTOP-LR5HOQBH: ~/Syspro/LN4

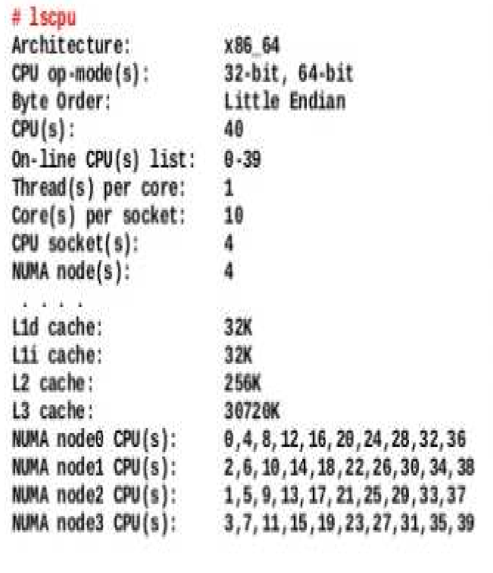
```
        .data
        .align 4
        .type   a, @object
        .size   a, 4
a:
        .long   10
        .globl  b
        .align 4
        .type   b, @object
        .size   b, 4
b:
        .long   20
        .comm   c,4,4
        .section        .rodata
.LC0:
        .string "C = %d₩n"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    a(%rip), %edx
        movl    b(%rip), %eax
        addl    %edx, %eax
        movl    %eax, c(%rip)
        movl    c(%rip), %eax
        movl    %eax, %esi
        leaq    .LC0(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
"test64.s" line 47
```

Summary

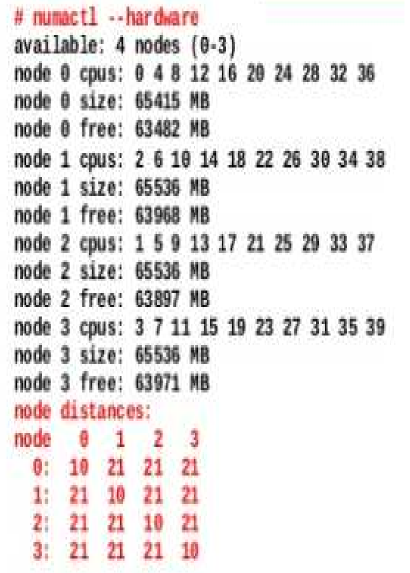
- Discuss the issues of ISA
- Grasp several operand addressing modes
- Understand how the context switch works
- Apprehend the technologies of IA
  - ✓ Pipelining
  - ✓ Dynamic execution
  - ✓ Cache (L1, L2, L3)
  - ✓ Superscalar
  - ✓ MMX
  - ✓ Hyper-threading
  - ✓ Multi core
  - ✓ Intel 64
  - ✓ Virtualization Technology

Quiz for 12th-Week 2nd-Lesson

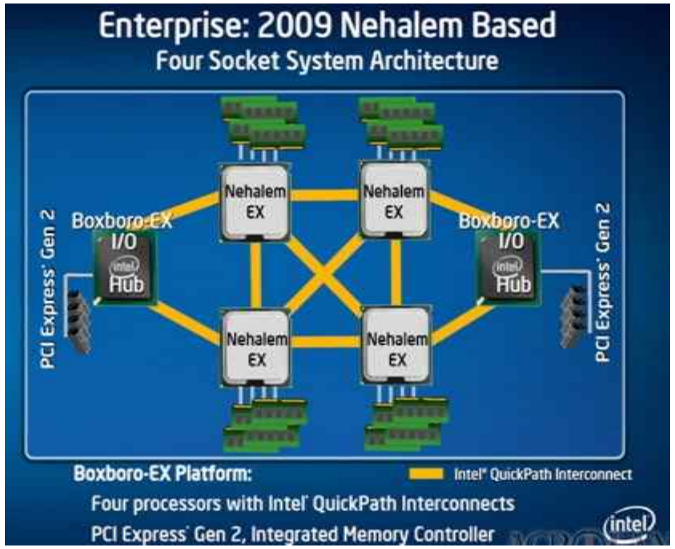
- Quiz
  - ✓ 1. Discuss the differences between x86 and x86-64 in an assembly code (at least 3)
  - ✓ 2. What is the NUMA that you can see the “lscpu” command?
  - ✓ Bonus). Discuss the differences between CPU and GPU in the viewpoint of ALU (Arithmetic and Logic Unit).
  - ✓ Due: until 6 PM Friday of this week (20th, November)



```
# lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                40
On-line CPU(s) list:   0-39
Thread(s) per core:    1
Core(s) per socket:    10
CPU socket(s):         4
NUMA node(s):          4
 . . . .
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              30720K
NUMA node0 CPU(s):     0,4,8,12,16,20,24,28,32,36
NUMA node1 CPU(s):     2,6,10,14,18,22,26,30,34,38
NUMA node2 CPU(s):     1,5,9,13,17,21,25,29,33,37
NUMA node3 CPU(s):     3,7,11,15,19,23,27,31,35,39
```



```
# numactl --hardware
available: 4 nodes (0-3)
node 0 cpus: 0 4 8 12 16 20 24 28 32 36
node 0 size: 65415 MB
node 0 free: 63482 MB
node 1 cpus: 2 6 10 14 18 22 26 30 34 38
node 1 size: 65536 MB
node 1 free: 63968 MB
node 2 cpus: 1 5 9 13 17 21 25 29 33 37
node 2 size: 65536 MB
node 2 free: 63897 MB
node 3 cpus: 3 7 11 15 19 23 27 31 35 39
node 3 size: 65536 MB
node 3 free: 63971 MB
node distances:
node   0   1   2   3
  0:  10  21  21  21
  1:  21  10  21  21
  2:  21  21  10  21
  3:  21  21  21  10
```


Enterprise: 2009 Nehalem Based
Four Socket System Architecture
Nehalem EX
Nehalem EX
Nehalem EX
Nehalem EX
Boxboro-EX I/O Hub
Boxboro-EX I/O Hub
PCI Express* Gen 2
PCI Express* Gen 2
Boxboro-EX Platform:
Intel® QuickPath Interconnect
Four processors with Intel® QuickPath Interconnects
PCI Express* Gen 2, Integrated Memory Controller
intel

(source: https://www.slideshare.net/tommylee98229/shak-larryjederperfandtuningsummit14part1final)