

# Lecture Note 7.

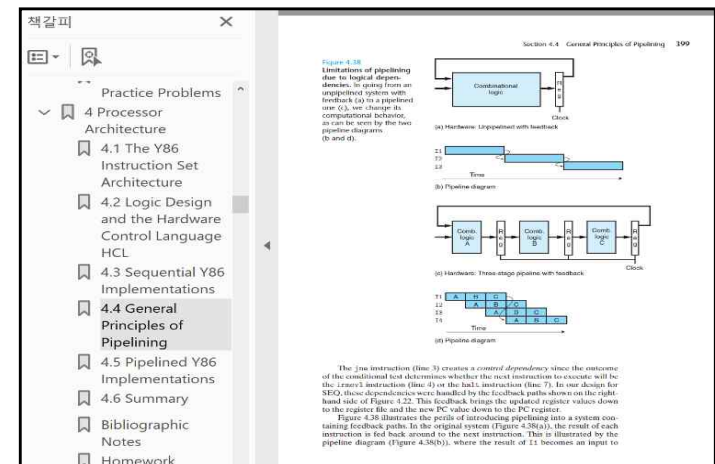
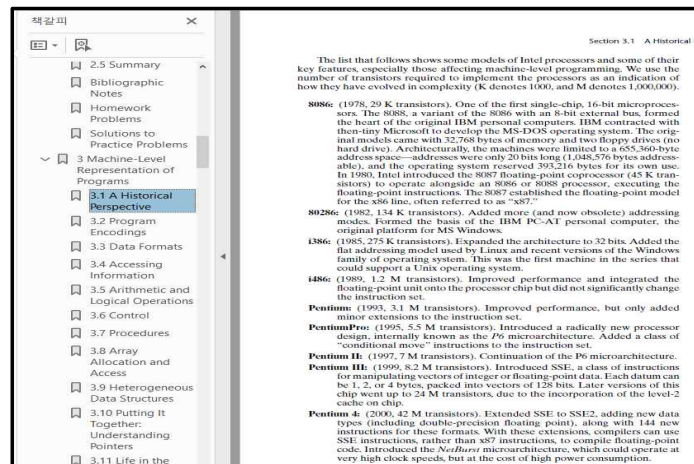
## IA: History and Features

November 11, 2023  
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, Memory overflow
- Describe the history of IA (both IA-32 and Intel 64)
- 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



# Issues on ISA (1/2)

## ■ Consideration on ISA (Instruction Set Architecture)

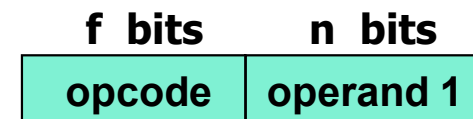
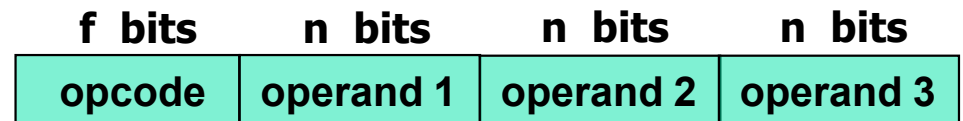
```
asm_sum:    addl    $1, %ecx
            movl   -4(%ebx, %ebp, 4), %eax
            call  func1
            leave
```

### ✓ 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?
  - Consider with  $C = A + B$ ;
- 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 x86-64
  - Register renaming
  - ...

**(Source: CSAPP Chapter 4)**

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.

# 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%--
```


```
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?**

6 46.5 모두



# 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
1 /* Based-index addressing example by choijm, Nov. 5th */
2 .data
3     size    array, 40
4 array:
5     .long   2
6     .long   3
7     .long   4
8     .long   3
9     .long   7
10    .long   6
11    .long   9
12    .long   2
13    .long   8
14    .long   9
15 P_arg:
16     .string "Sum of array = %d\n"
17     .text
18 .globl main
19     .type   main, @function
20 main:
21     pushl  %ebp
22     movl   %esp, %ebp
23     subl  $8, %esp
24
25     movl   $0, %ecx
26     movl   $0, %eax
27     movl   $array, %ebx
28 LOOP:
29     cmpl  $9, %ecx
30     jg    LOOP_OUT
31     addl  0(%ebx, %ecx, 4), %eax
32     addl  $1, %ecx
33     jmp   LOOP
34 LOOP_OUT:
35     pushl  %eax
36     pushl  $P_arg
37     call  printf
38     addl  $8, %esp
39     leave
40     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

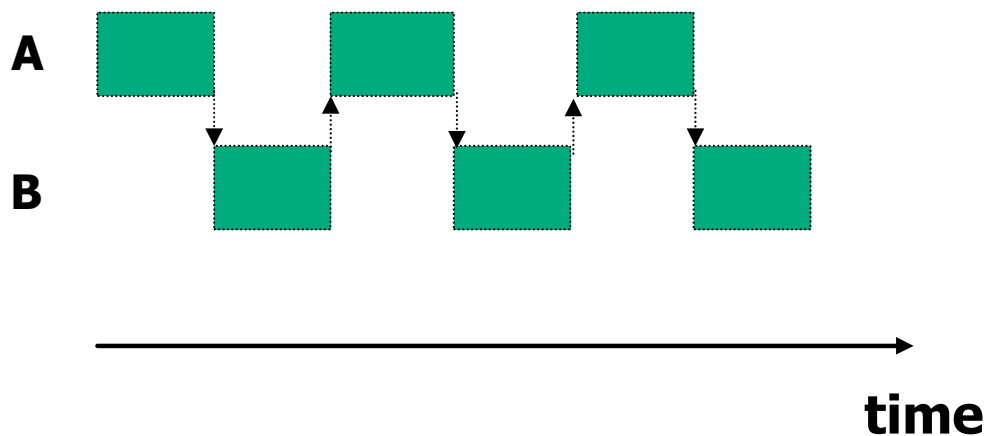
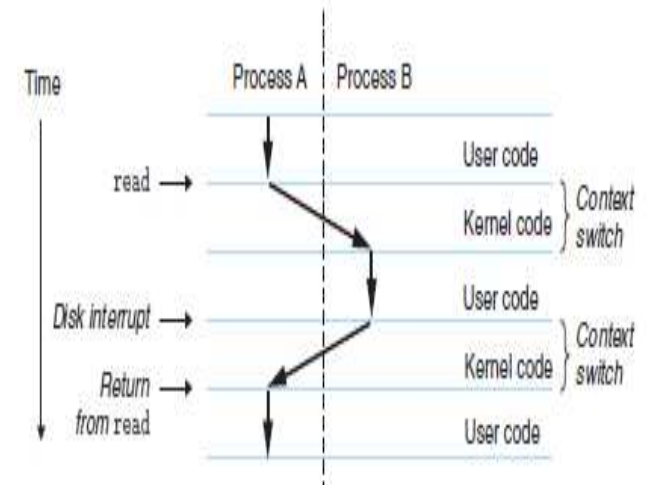
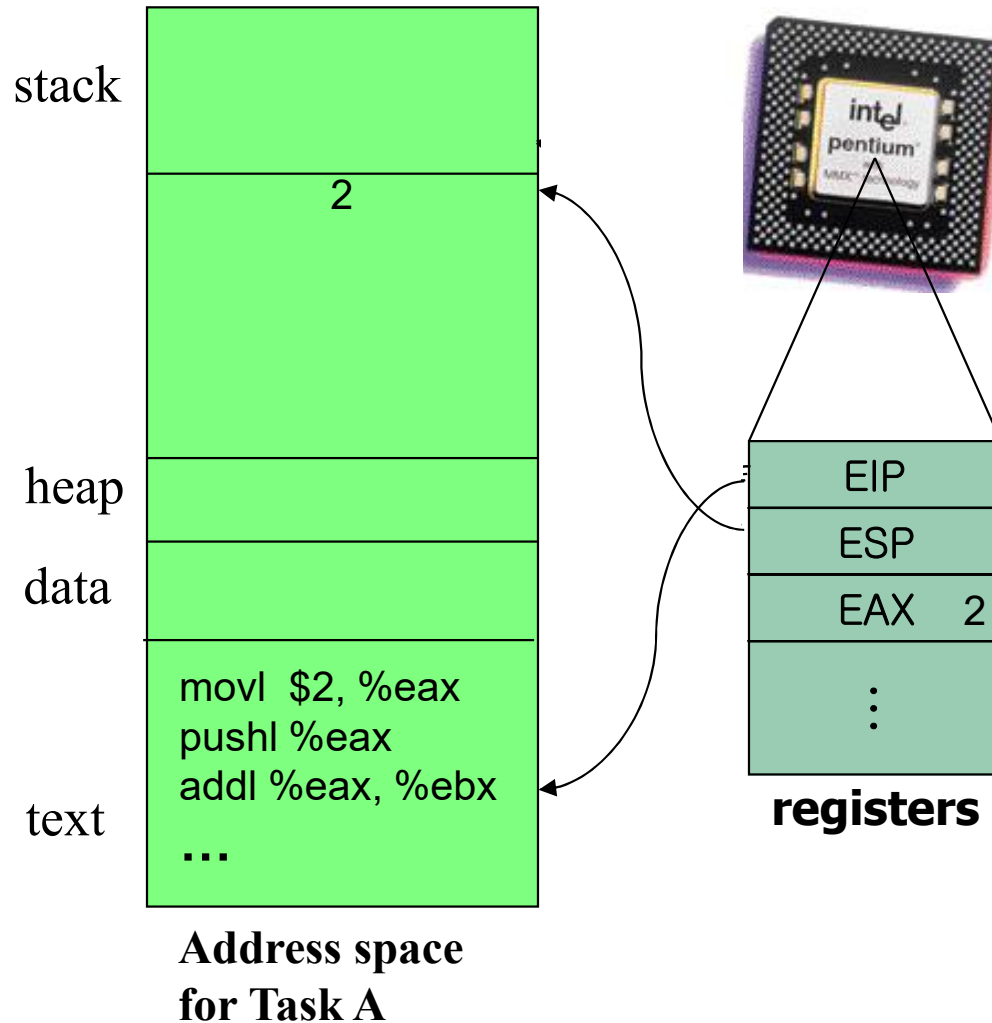


Figure 8.14  
Anatomy of a process  
context switch.



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

## ■ Virtual CPU: running A

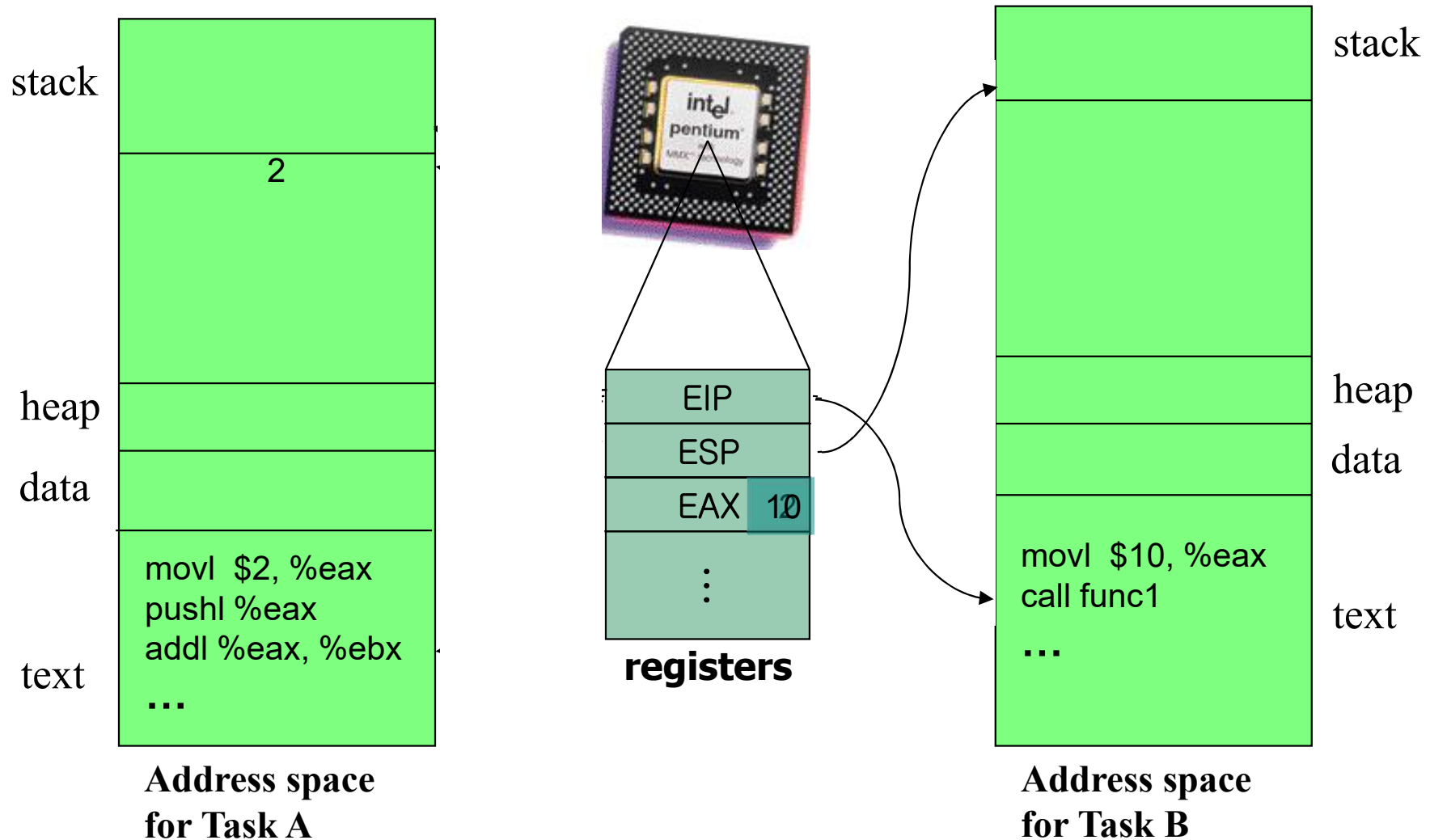


👉 Time quantum is expired, system program (scheduler) selects a Task B to run next.



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

## ■ Virtual CPU: switch to B

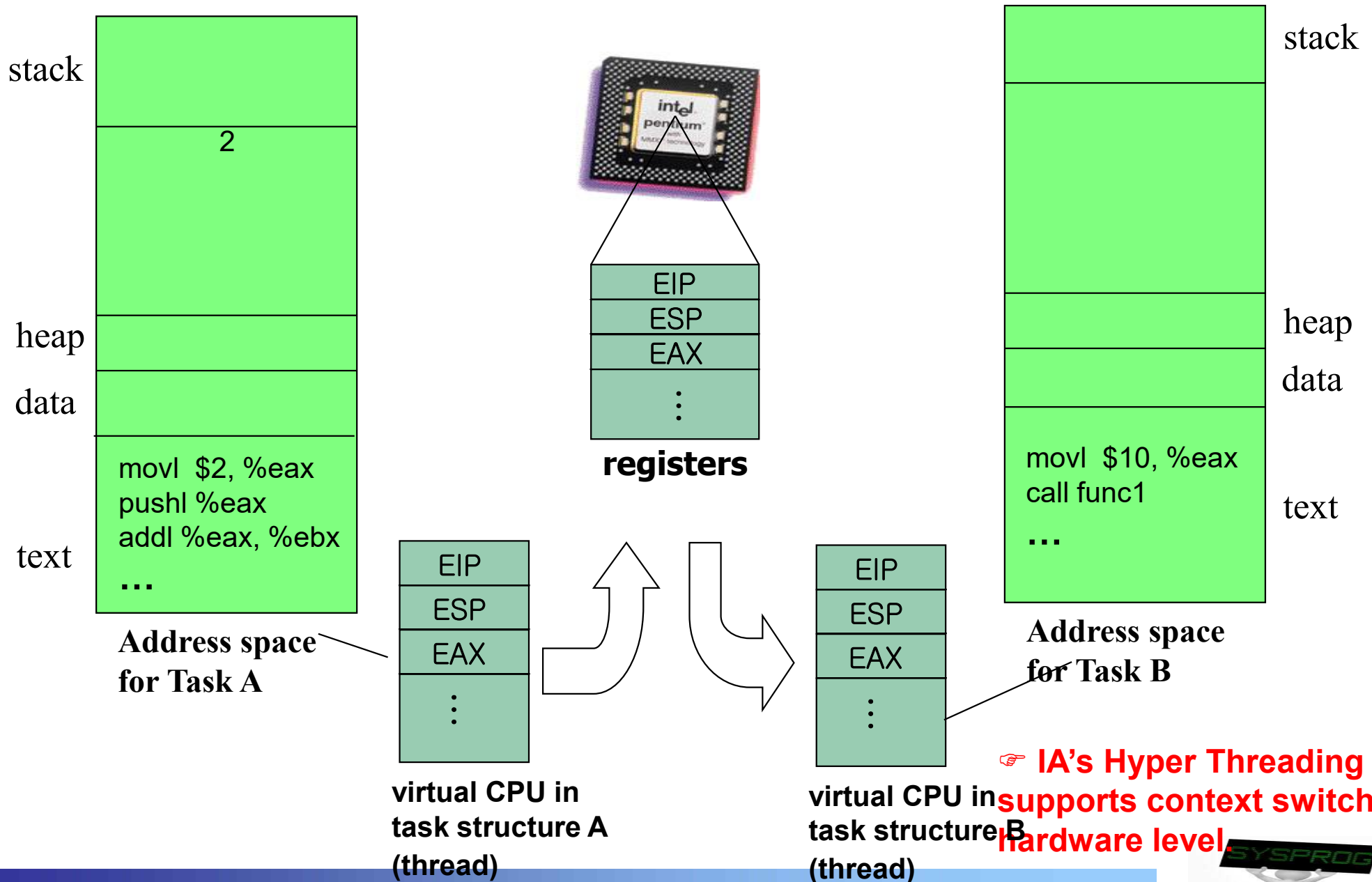


- ☞ Time quantum is expired, system program (scheduler) selects a Task B to run next.
- ☞ Time quantum is expired, again. Task A is scheduled. Then where to start?
- ☞ Context Switch → save/restore context (architectural state or thread)



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

## Virtual CPU: how to switch back to A





# Impact of ISA on system program: Memory Usage (1/6)

## ■ Little Endian vs. Big Endian

```
choijm's X desktop (embedded.wowdns.com:2)
choijm@embedded:~/public/html/sys_prog/exam_byteorder
#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@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]$ 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:~
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 ~ $
```

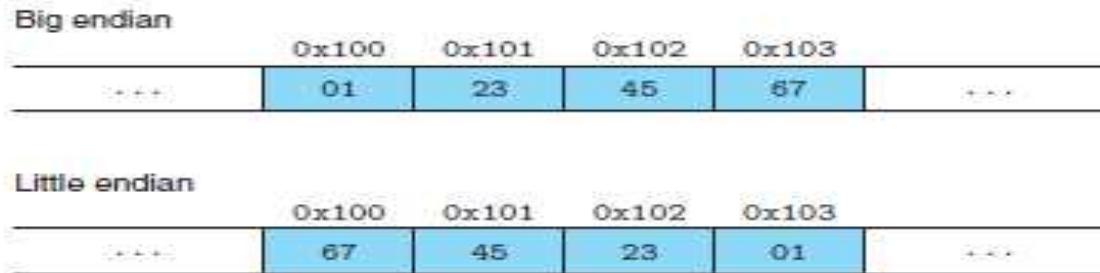




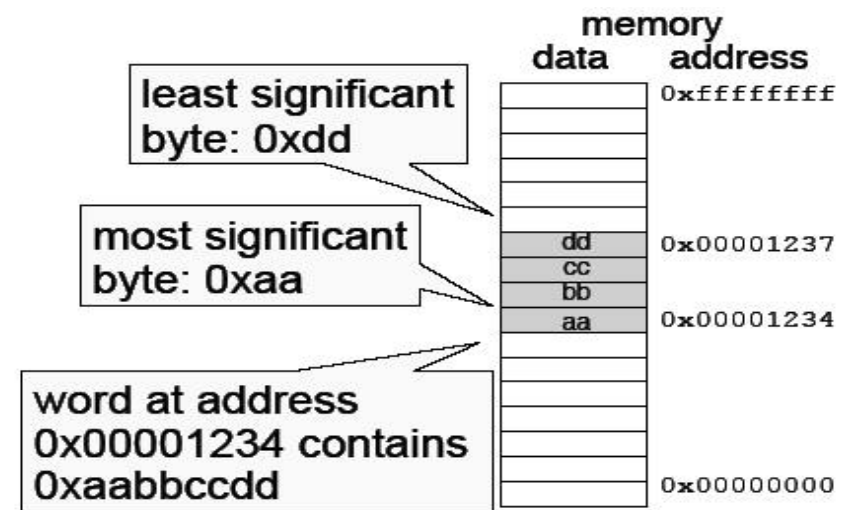
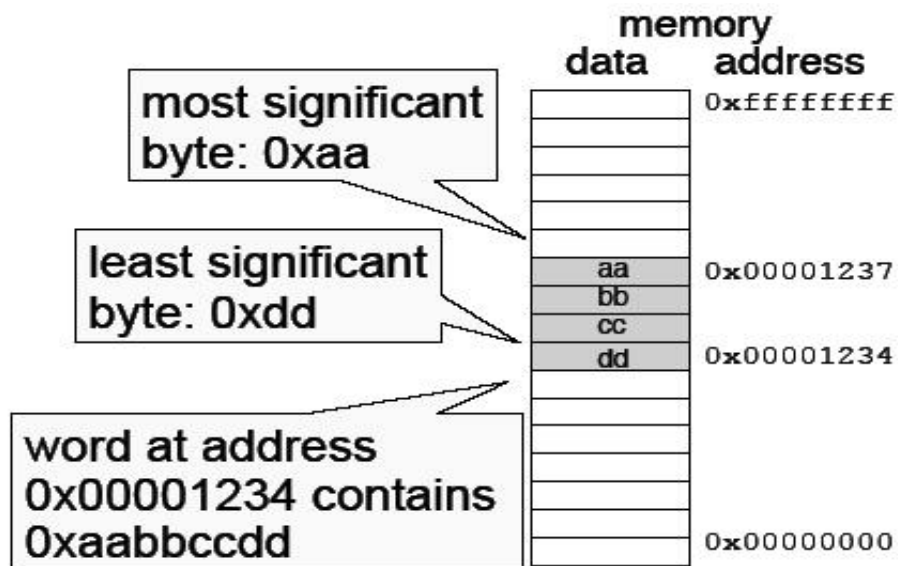
# Impact of ISA on system program: Memory Usage (2/6)

## ■ 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:



(Source: CSAPP)



# Impact of ISA on system program: Memory Usage (3/6)

## ■ 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$ gcc -c test.c
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$ gcc test.c
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$ 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$ readelf -a a.out
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 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
  [ 1] .interp                PROGBITS       0000000000000318 00000318
     000000000000001c 0000000000000000 A 0 0 1
  [ 2] .note.gnu.property    NOTE           0000000000000338 00000338
     0000000000000020 0000000000000000 A 0 0 8
  [ 3] .note.gnu.build-id   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 (4/6)

- 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
1 /* Byte alignment test bu choijm */
2 #include <stdio.h>
3
4 // #define TEST_PACKED
5
6 #ifdef TEST_PACKED
7 typedef struct {
8     int a;
9     double d1;
10    char ch;
11    double d2;
12 } __attribute__((packed)) Test;
13 #else
14 typedef struct {
15     int a;
16     double d1;
17     char ch;
18     double d2;
19 } Test;
20 #endif
21
22 int main()
23 {
24     Test test;
25
26     printf("Size of Test is %d\n", sizeof(test));
27 }
~
byte_alignment.c 27,1
"byte_alignment.c" 27L, 377C
```

☞ Depend on compiler and CPU

☞ “\_\_attribute\_\_((packed))”



# Impact of ISA on system program: Memory Usage (5/6)

## ■ Memory Alignment in stack

- ✓ Caller: need 16 bytes (for 2 local variables + 2 arguments in Fig. 3.24)
  - note that sum and diff are not allocated in stack in this example
- ✓ But, allocate 24 bytes for alignment in a frame (subl \$24, %esp)
  - 4B for saved ebp, allocated 24B, 4B for return addr. → 16B alignment

```
1 int swap_add(int *xp, int *yp)
2 {
3     int x = *xp;
4     int y = *yp;
5
6     *xp = y;
7     *yp = x;
8     return x + y;
9 }
10
11 int caller()
12 {
13     int arg1 = 534;
14     int arg2 = 1057;
15     int sum = swap_add(&arg1, &arg2);
16     int diff = arg1 - arg2;
17
18     return sum * diff;
19 }
```

Figure 3.23 Example of procedure definition and call.

(Source: CSAPP)

```
1 caller:
2     pushl %ebp           Save old %ebp
3     movl %esp, %ebp     Set %ebp as frame pointer
4     subl $24, %esp      Allocate 24 bytes on stack
5     movl $534, -4(%ebp) Set arg1 to 534
6     movl $1057, -8(%ebp) Set arg2 to 1057
7     leal -8(%ebp), %eax Compute &arg2
8     movl %eax, 4(%esp)  Store on stack
9     leal -4(%ebp), %eax Compute &arg1
10    movl %eax, (%esp)   Store on stack
11    call swap_add      Call the swap_add function
12    movl -4(%ebp), %edx
13    subl -8(%ebp), %edx
14    imull %edx, %eax
15    leave
16    ret
```

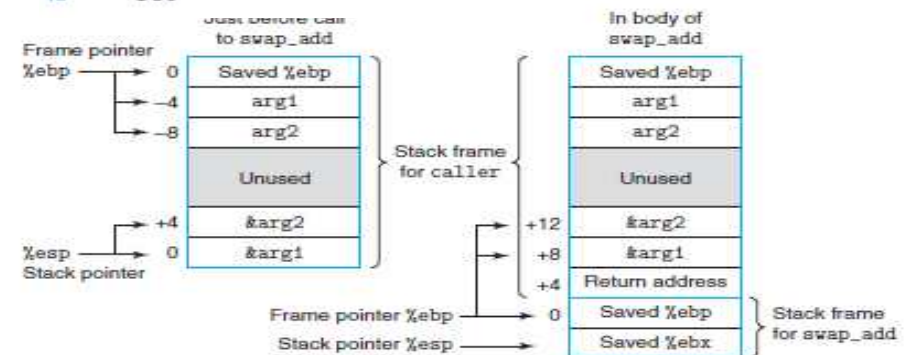


Figure 3.24 Stack frames for caller and swap\_add. Procedure swap\_add retrieves its arguments from the stack frame for caller.

# Impact of ISA on system program: Memory Usage (6/6)

## ■ Revisit the stack in LN6

- ✓ IA recommends 16 bytes alignment: `andl $-16, %esp`

```
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" [바꿈] 50 줄 --74%--
```

2's complement

# Impact of ISA on system program: Memory overflow (1/2)

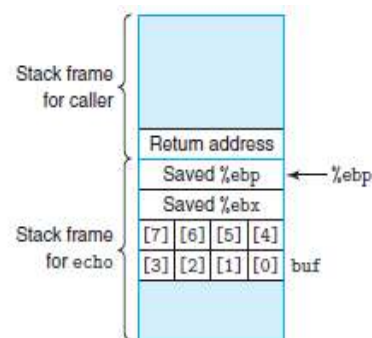
## ■ Memory overflow

- ✓ Due to no boundary check: buffer overflow, stack overflow
- ✓ How to thwart buffer overflow on stack (or Heap)
  - Stack randomization
    - One step further: ASLR (Address Space Layout Randomization) → even code, data and heap (see Appendix)
  - Stack protector (a.k.a. stack guard): e.g. Canary

```
1 /* Sample implementation of library function gets() */
2 char *gets(char *s)
3 {
4     int c;
5     char *dest = s;
6     int gotchar = 0; /* Has at least one character been read? */
7     while ((c = getchar()) != '\n' && c != EOF) {
8         *dest++ = c; /* No bounds checking! */
9         gotchar = 1;
10    }
11    *dest++ = '\0'; /* Terminate string */
12    if (c == EOF && !gotchar)
13        return NULL; /* End of file or error */
14    return s;
15 }
```

```
17 /* Read input line and write it back */
18 void echo()
19 {
20     char buf[8]; /* Way too small! */
21     gets(buf);
22     puts(buf);
23 }
```

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.



```
1 echo:
2     pushl %ebp          Save %ebp on stack
3     movl  %esp, %ebp
4     pushl %ebx          Save %ebx
5     subl  $20, %esp     Allocate 20 bytes on stack
6     leal -12(%ebp), %ebx Compute buf as %ebp-12
7     movl  %ebx, (%esp)  Store buf at top of stack
8     call  gets          Call gets
9     movl  %ebx, (%esp)  Store buf at top of stack
10    call  puts          Call puts
11    addl  $20, %esp     Deallocate stack space
12    popl  %ebx          Restore %ebx
13    popl  %ebp          Restore %ebp
14    ret                Return
```



# Impact of ISA on system program: Memory overflow (2/2)

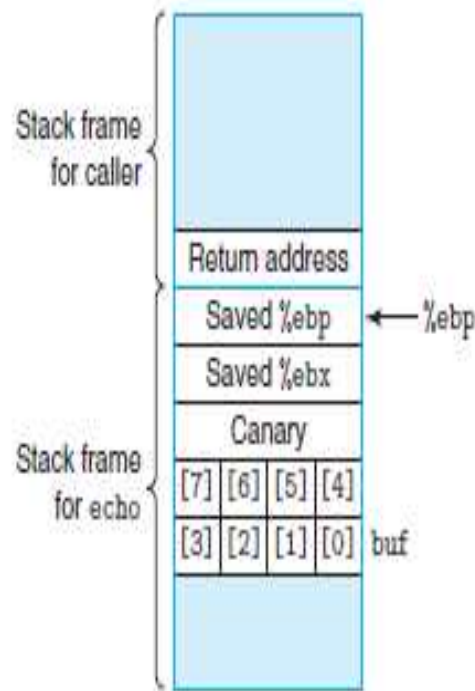
## ■ Stack protector

- ✓ 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.



```
1  echo:
2  pushl  %ebp
3  movl   %esp, %ebp
4  pushl  %ebx
5  subl  $20, %esp
6  movl  %gs:20, %eax  Retrieve canary
7  movl  %eax, -8(%ebp) Store on stack
8  xorl  %eax, %eax   Zero out register
9  leal  -16(%ebp), %ebx Compute buf as %ebp-16
10 movl  %ebx, (%esp) Store buf at top of stack
11 call  gets        Call gets
12 movl  %ebx, (%esp) Store buf at top of stack
13 call  puts        Call puts
14 movl  -8(%ebp), %eax Retrieve canary
15 xorl  %gs:20, %eax Compare to stored value
16 je    .L19        If =, goto ok
17 call  __stack_chk_fail Stack corrupted!
18 .L19: ok:
19 addl  $20, %esp   Normal return ...
20 popl  %ebx
21 popl  %ebp
22 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), **1<sup>st</sup> 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, **5<sup>th</sup> 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)



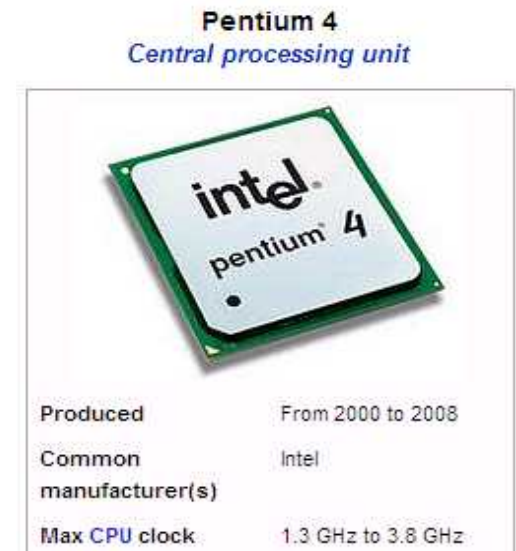
# Intel CPU History (3/9)

- P6 family (1995~1999, 6<sup>th</sup> generation )
  - ✓ **P6 Microarchitecture:** Dynamic execution
    - Out-of-order execution
    - Branch prediction
    - Speculative execution: decouple execution and commitment
    - 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



# 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)
  - ✓ **64-bit CPU** (IA-64, 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



# 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

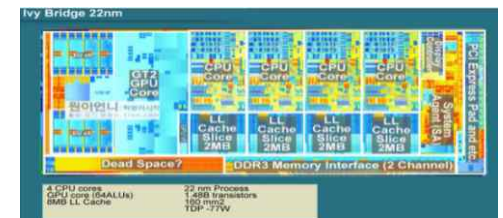
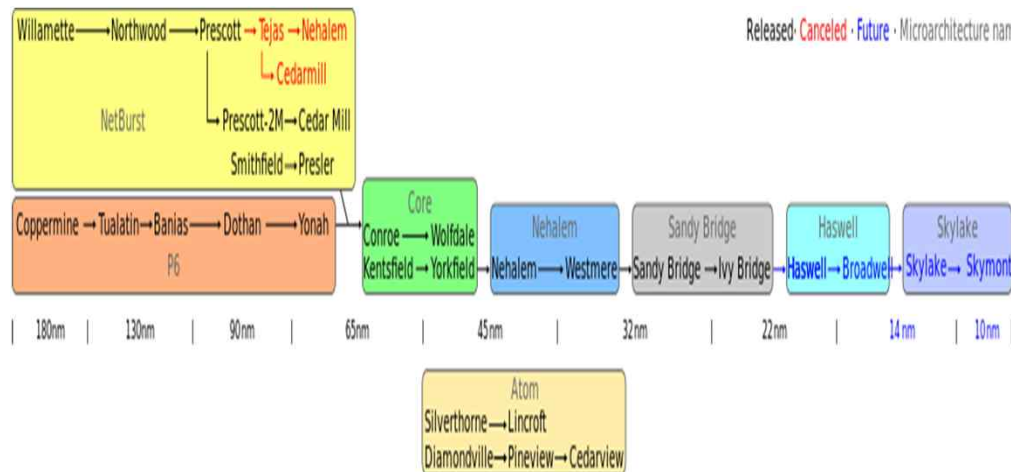
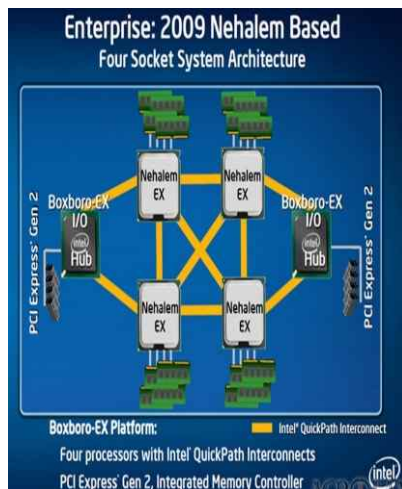


(source: <http://motoc.tistory.com/>)



# Intel CPU History (6/9)

- Intel Core i3/i5/i7 Family (2009 ~)
  - ✓ **Nehalem microarchitecture** (and its 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 1<sup>st</sup> generation**
  - ✓ **Sandy Bridge, Haswell, Skylake, Sunny Cove, Raptor Cove** u-architecture
    - Successor of Nehalem, <= 32 nm, **Tick-Tock strategy**
    - AVX (Advanced Vector extension, 256 bit SSE), **Integrated GPU**, DDR4, 10 nm
  - ✓ **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)

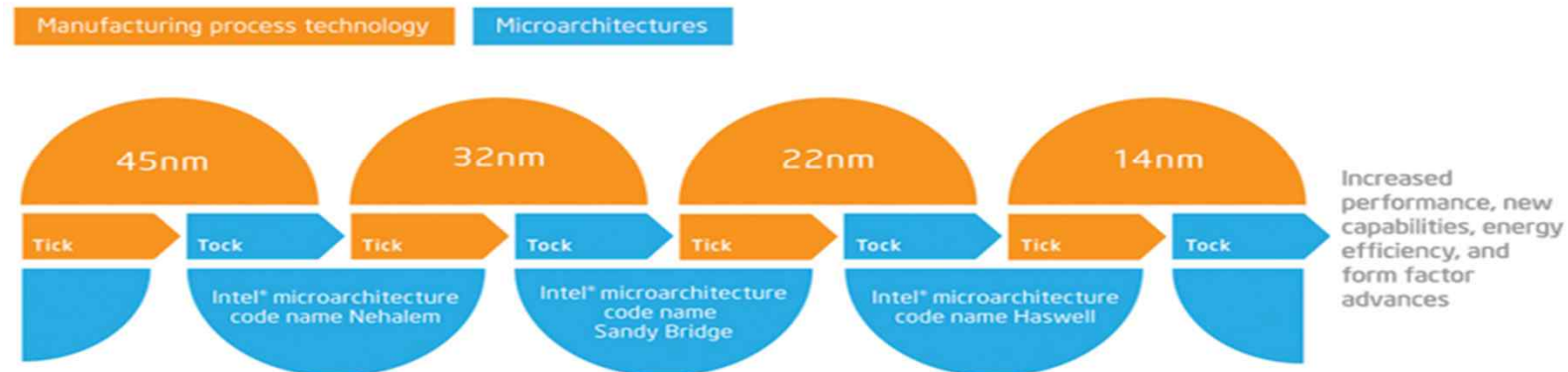




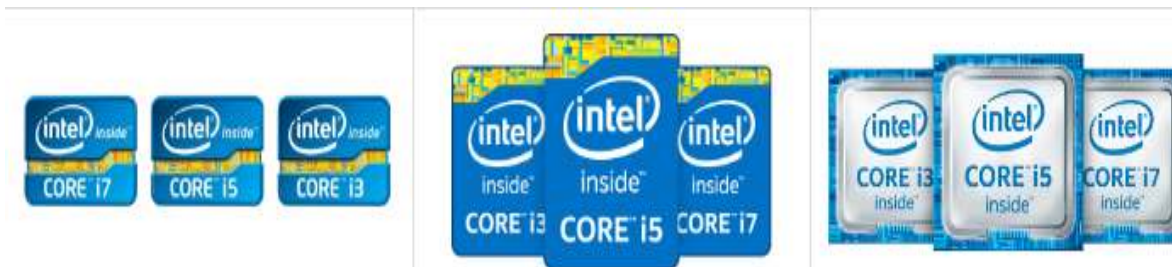
# Intel CPU History (7/9)

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

## The Tick-Tock model through the years



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



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





# Intel CPU History (8/9)

## ■ Summary of Intel CPU microarchitecture

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	
2008	Nehalem	20 unified (14 without miss prediction)	3600	45
	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	4000	

(source: [en.wikipedia.org/wiki/List\\_of\\_Intel\\_CPU\\_microarchitectures](https://en.wikipedia.org/wiki/List_of_Intel_CPU_microarchitectures))

2012	Ivy Bridge (die shrink)	fetch/retire	4100	22 nm
2013	Silvermont	14–17 (16–19 with fetch/retire)	2670	
	Haswell	14 (16 with fetch/retire)	4400	
2014	Broadwell (die shrink)	fetch/retire	3700	14 nm
2015	Airmont (die shrink)	14–17 (16–19 with fetch/retire)	2640	
	Skylake	14 (16 with fetch/retire)	5200	
2016	Goldmont	20 unified with branch prediction	2600	
2017	Goldmont Plus	20 unified with branch prediction (?)	2800	
2018	Palm Cove	14 (16 with fetch/retire)	3200	10 nm
2019	Sunny Cove	14–20 (misprediction)	4100	
	Tremont	20 unified	3300	
2020	Willow Cove	14 unified	5300	
2021	Cypress Cove	14 unified	5300	14 nm
	Golden Cove	12 unified	5500	
	Gracemont	20 unified with misprediction penalty	4300	Intel 7
2022	Raptor Cove	12 unified	6000	Intel 4
2023	Redwood Cove			
	Crestmont			

# Intel CPU History (9/9)

## ■ Summary of Intel CPU microarchitecture

- ✓ From [https://en.wikipedia.org/wiki/List\\_of\\_Intel\\_CPU\\_microarchitectures](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 → Raptor Cove (2022), Redwood Cove (2023), HW-accelerator for security and AI features

→ **Rival: AMD (x86), ARM (Atom), Nvidia (Haswell), Samsung (IMC), ..**



# Technologies of Intel CPU (1/12)

## ■ What processor do?

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

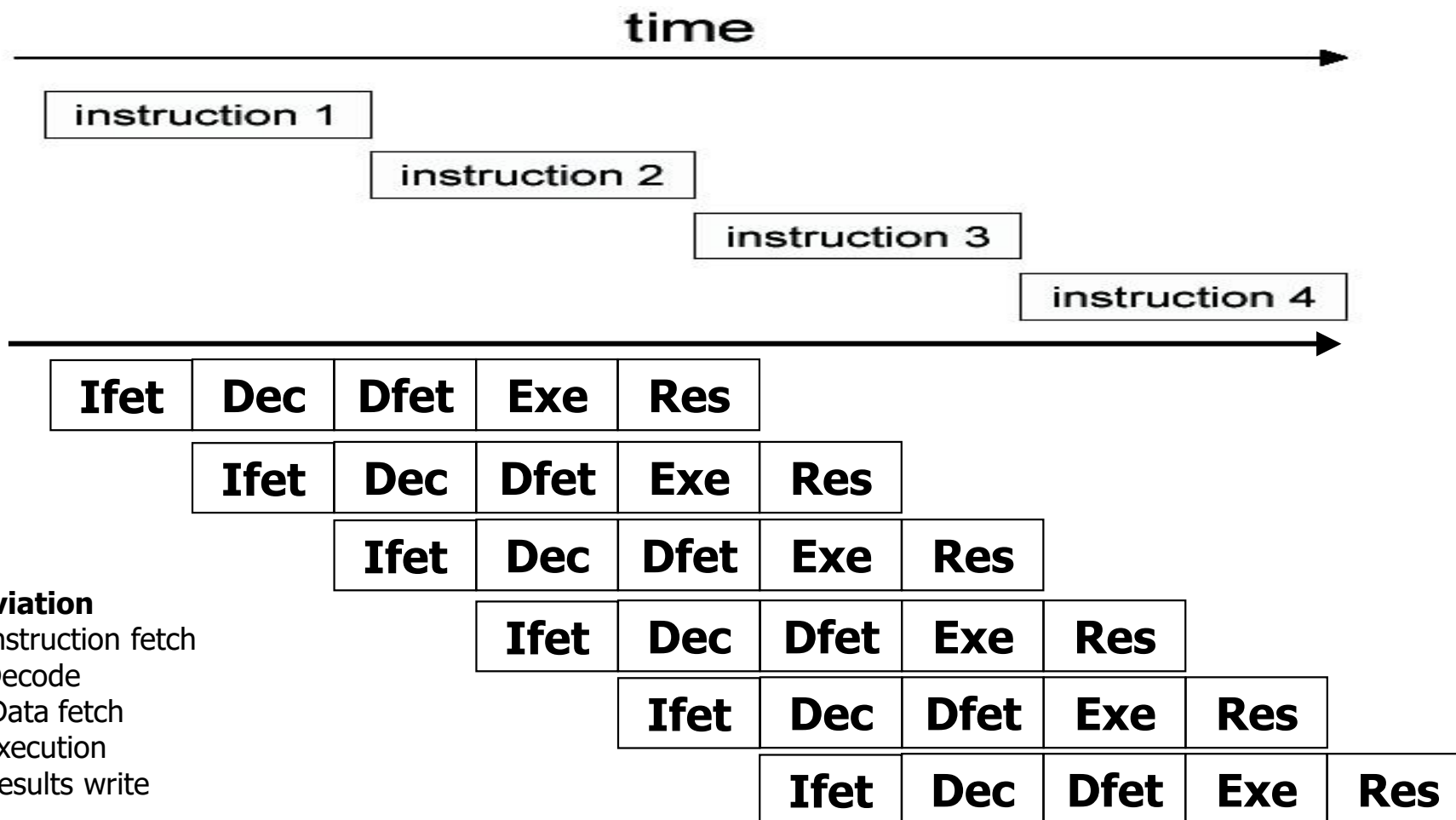
- ✓ Data movement needs to be optimized
  - ➔ CPU cache, write buffer
- ✓ Some components are idle while executing instruction
  - ➔ Pipelining
  - ➔ Superscalar



# Technologies of Intel CPU (2/12)

## ■ Pipeline

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

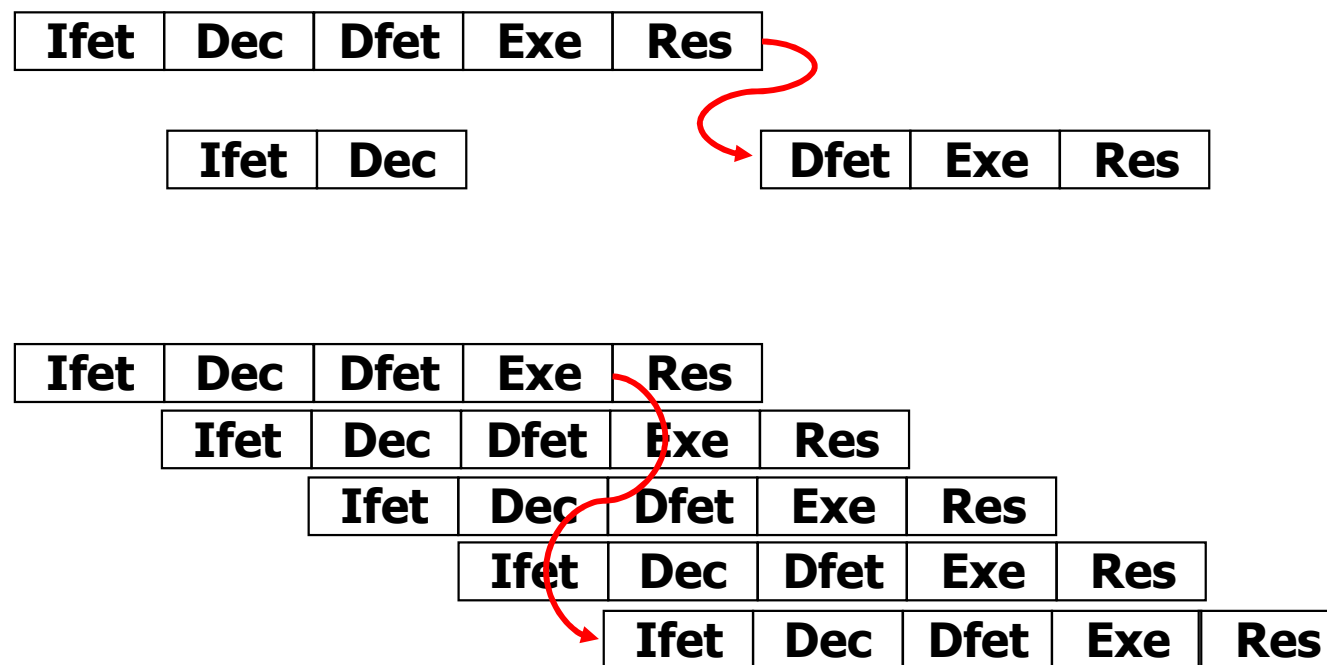


→ latency vs. throughput



# 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 (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

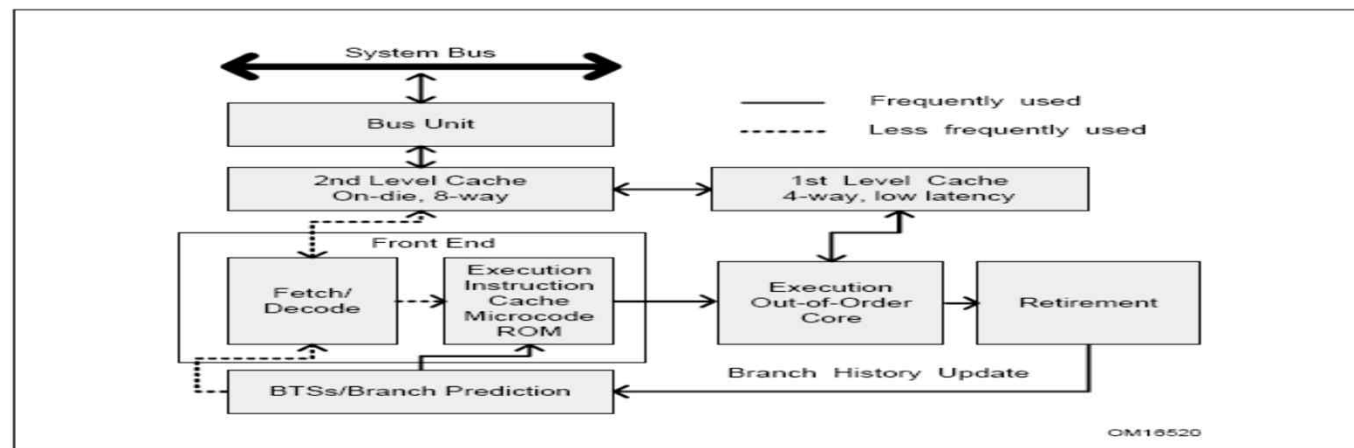


Figure 2-1. The P6 Processor Microarchitecture with Advanced Transfer Cache Enhancement



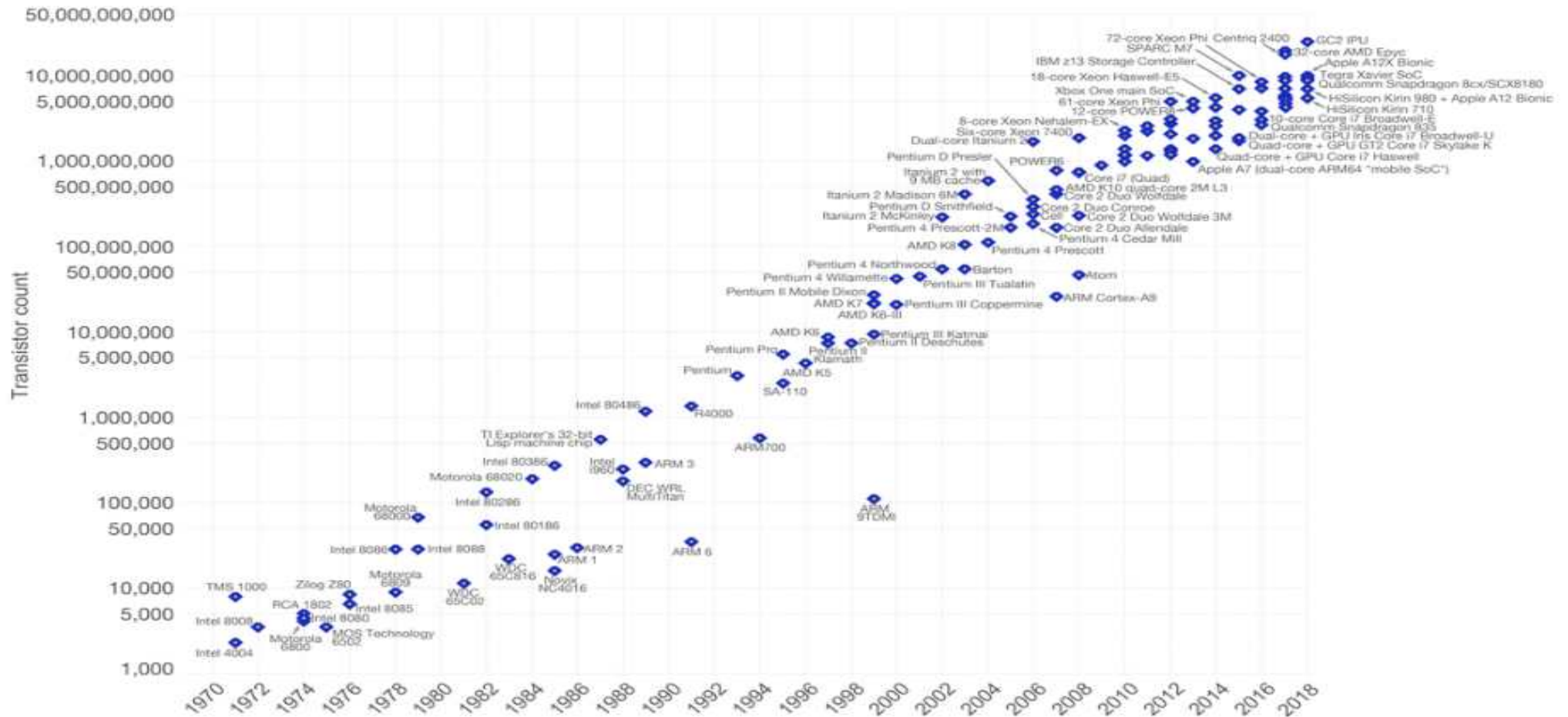


# Technologies of Intel CPU (6/12)

## ■ 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.



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

Licensed under CC-BY-SA by the author Max Roser.

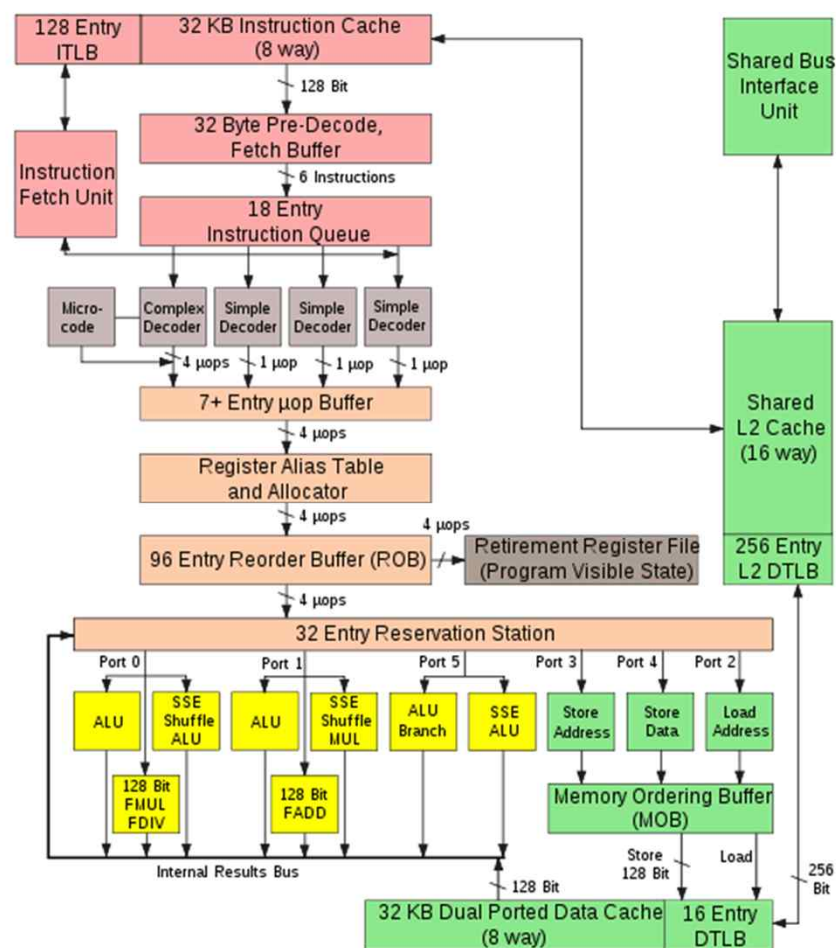
(Source: [https://en.wikipedia.org/wiki/Moore%27s\\_law](https://en.wikipedia.org/wiki/Moore%27s_law))



# 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
- ✓ Virtualization



Intel Core 2 Architecture

(From [http://en.wikipedia.org/wiki/File:Intel\\_Core2\\_arch.svg](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)

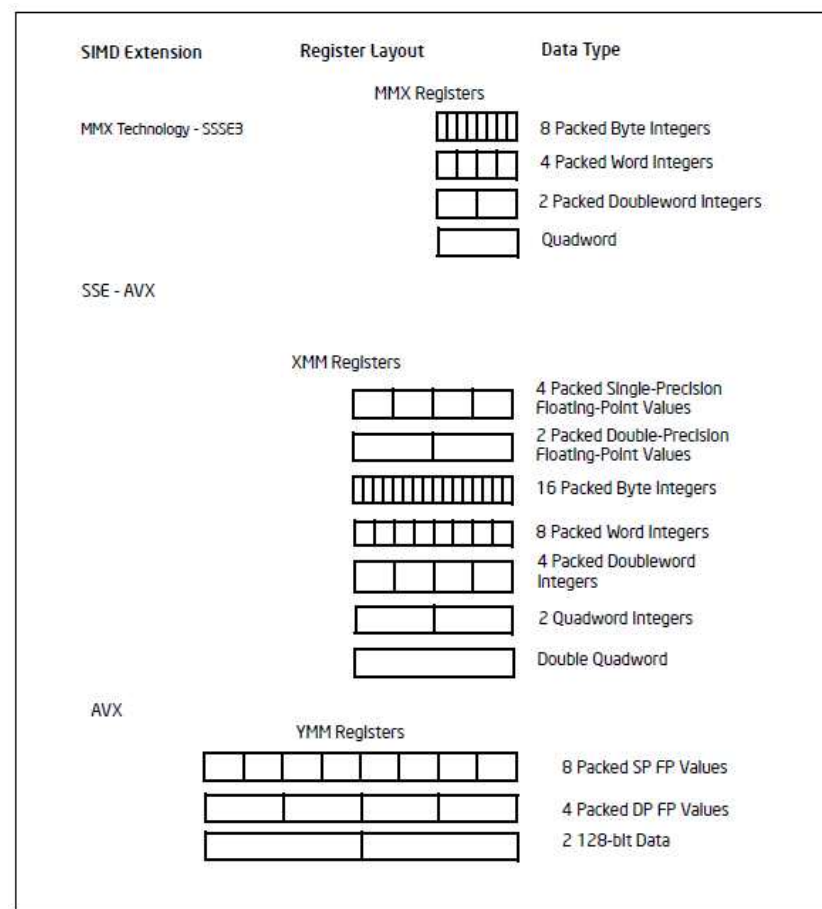


Figure 2-4. SIMD Extensions, Register Layouts, and Data Types



# Technologies of Intel CPU (9/12)

## ■ Hyper threading Technology

- ✓ Support multi-threading at CPU level
- ✓ 2 or more separated code streams using shared execution resources

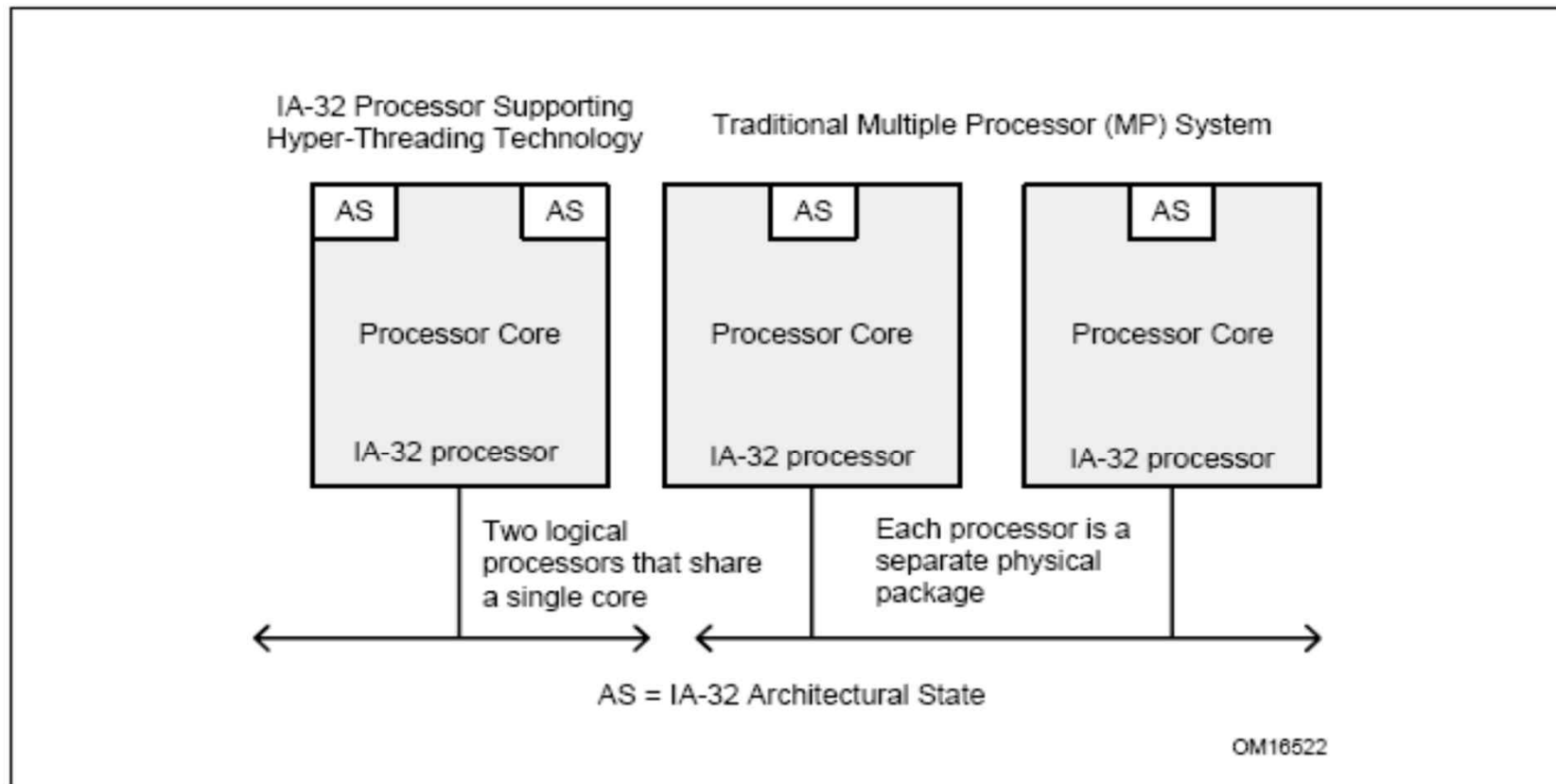


Figure 2-5. Comparison of an IA-32 Processor Supporting Hyper-Threading Technology and a Traditional Dual Processor System



# 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,

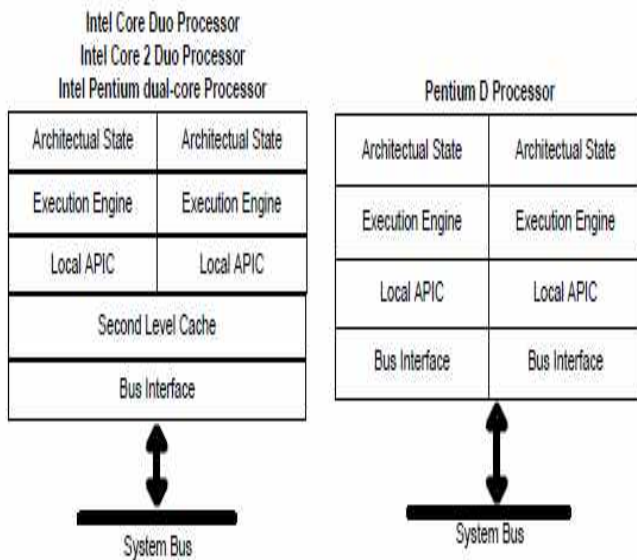


Figure 2-6. Intel 64 and IA-32 Processors that Support Dual-Core

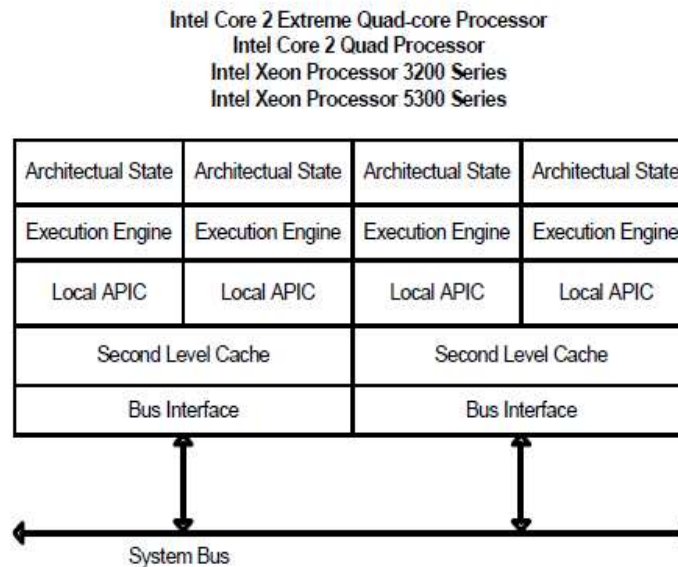


Figure 2-7. Intel 64 Processors that Support Quad-Core

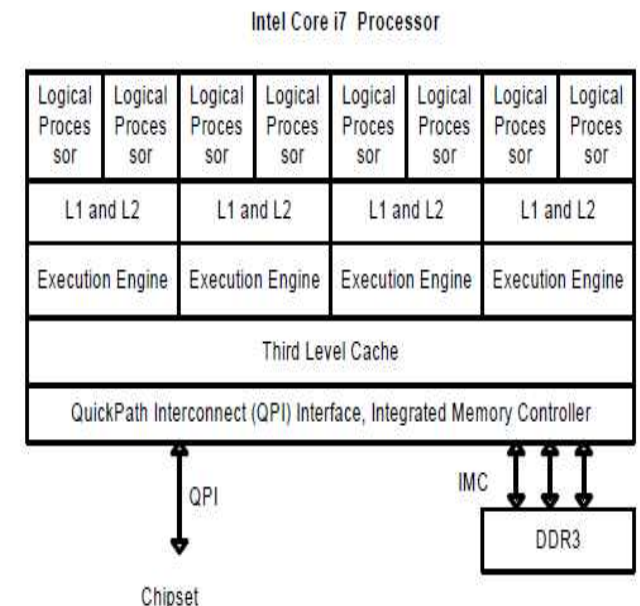


Figure 2-8. Intel Core i7 Processor



# 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

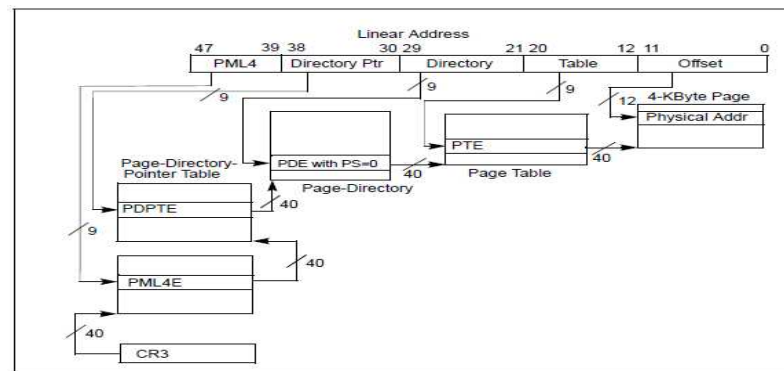
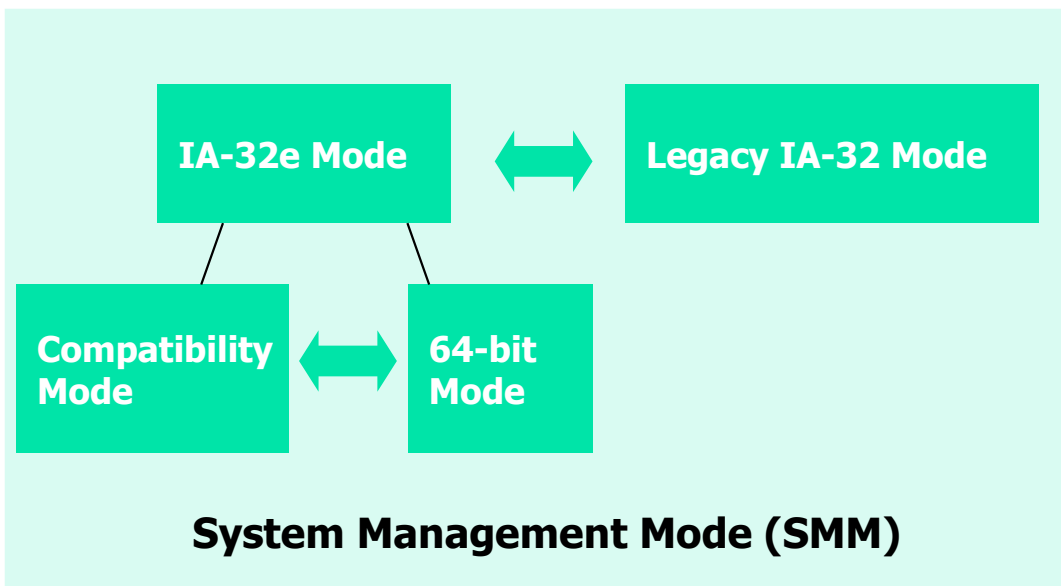


Figure 4-8. Linear-Address Translation to a 4-KByte Page using IA-32e Paging

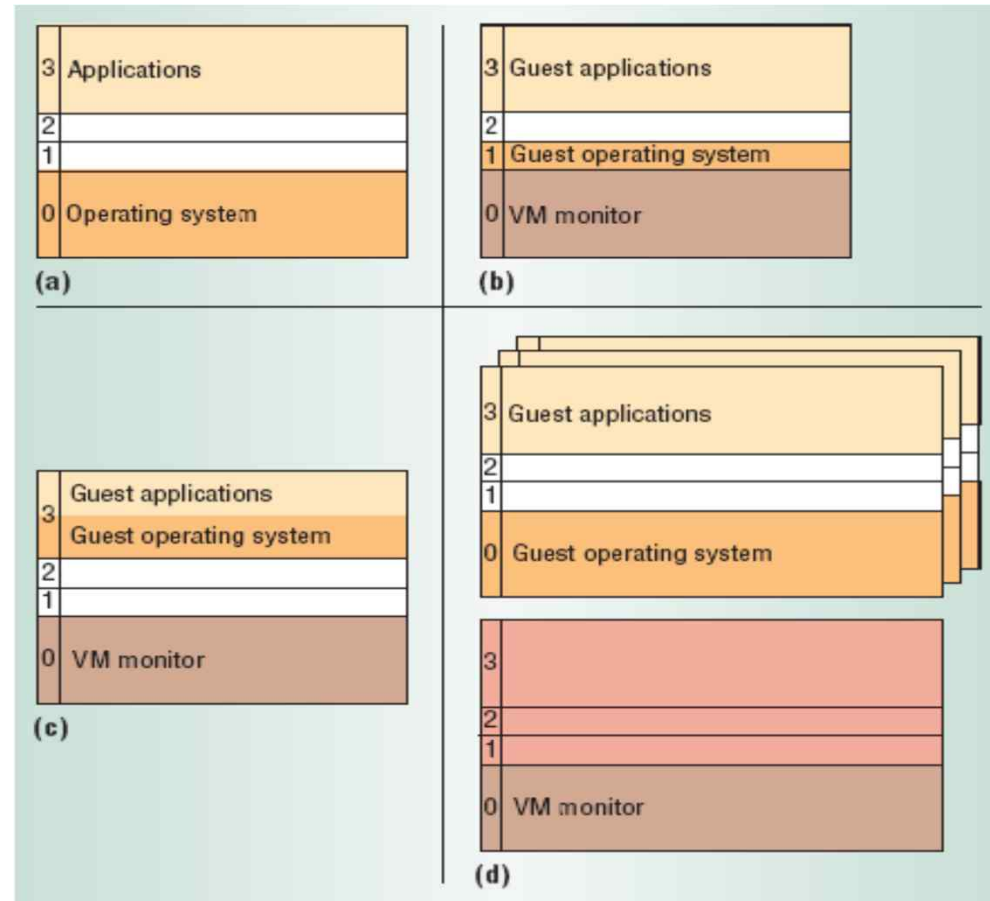
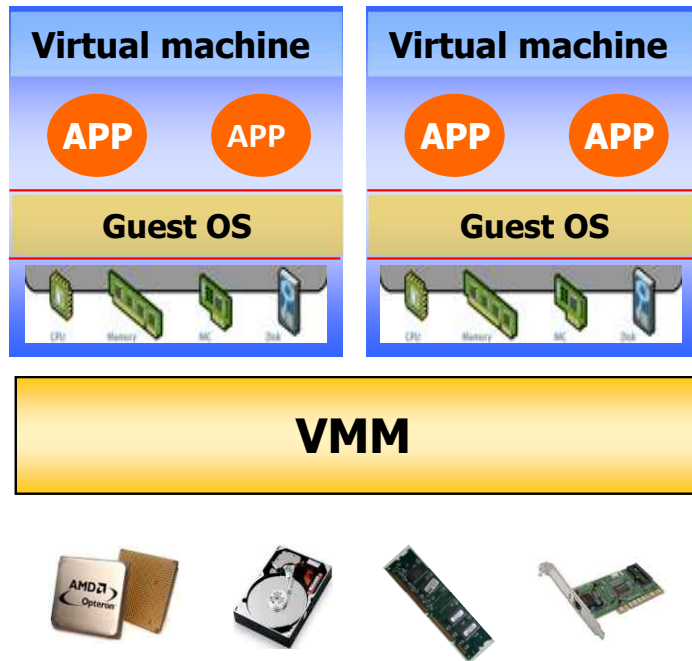


Software Visible Register	64-Bit Mode			Legacy and Compatibility Modes		
	Name	Number	Size (bits)	Name	Number	Size (bits)
General Purpose Registers	RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, R8-15	16	64	EAX, EBX, ECX, EDX, EBP, ESI, EDI, ESP	8	32
Instruction Pointer	RIP	1	64	EIP	1	32
Flags	EFLAGS	1	32	EFLAGS	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 IA-32 to 64-bit CPU (1/4)

- Brief history from IA-32 (x86) to Intel 64 (x86-64)
  - ✓ Intel traditional ISA: called as IA-32
    - 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 IA-64
    - Totally new ISAs called EPIC (Explicitly Parallel Instruction Computing)  
→ MIMD
    - Market name: Itanium (2001)
  - ✓ AMD ISA for 64-bit CPU
    - Compatible with IA-32 → win at the market
    - Intel follows: Intel 64 (This is why SW developer manual is named as Intel 64 and IA-32 ...)
    - AMD renames AMD 64 (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 IA-32 to 64-bit CPU (2/4)

## ■ Features of x86-64

- ✓ New size for some data types
  - E.g. 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.

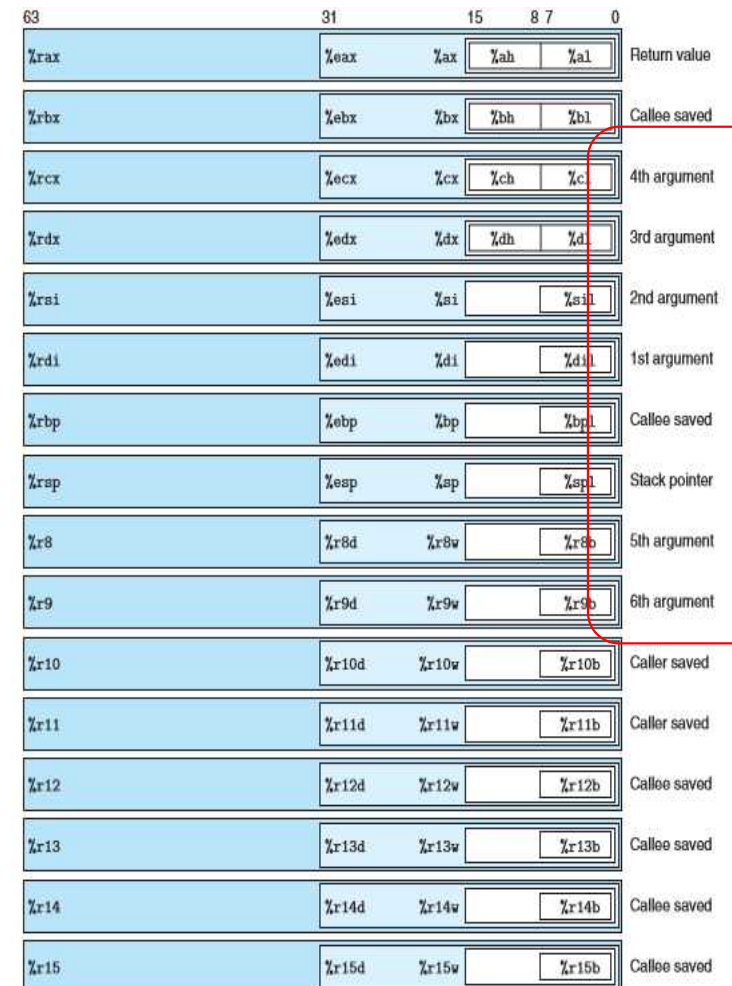


Figure 3.35 Integer registers. The existing eight registers are extended to 64-bit versions, and eight new registers are added. Each register can be accessed as either 8 bits (byte), 16 bits (word), 32 bits (double word), or 64 bits (quad word).



# x86-64: extending IA-32 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) use PIC (Position Independent Code) based on rip (rip-relative address), ...
  - 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*

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

*x86-64 version of function simple\_l.*

*xp in %rdi, y in %rsi*

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





# x86-64: extending IA-32 to 64-bit CPU (4/4)

## ■ Assembly code example2

- ✓ Recent gcc using PIC
- ✓ 1) using GOT (Global Offset Table), 2) using **rip-relative** addressing

```
choijm@LAPTOP-LR5HQQBH: ~/Syspro/LN4
choijm@LAPTOP-LR5HQQBH:~/Syspro/LN4$
choijm@LAPTOP-LR5HQQBH:~/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-LR5HQQBH:~/Syspro/LN4$
choijm@LAPTOP-LR5HQQBH:~/Syspro/LN4$ gcc -S -o test64.s test.c -m64
choijm@LAPTOP-LR5HQQBH:~/Syspro/LN4$
choijm@LAPTOP-LR5HQQBH:~/Syspro/LN4$ gcc -S -o test32.s test.c -m32
choijm@LAPTOP-LR5HQQBH:~/Syspro/LN4$
choijm@LAPTOP-LR5HQQBH:~/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 9.3.0-10ubuntu2' --with-bugurl=file:///usr/share/doc/gcc-9/README.Bugs --enable-languages=c,ada,c++,go,brig,d,fortran,objc,obj-c++,gm2 --prefix=/usr --with-gcc-major-version-only --program-suffix=-9 --program-prefix=x86_64-linux-gnu- --enable-shared --enable-linker-build-id --libexecdir=/usr/lib --without-included-gettext --enable-threads=posix --libdir=/usr/lib --enable-objc-gc --enable-cloCALE=glibc --enable-libstdc++-debug --enable-libstdc++-time=yes --with-default-libstdc++-abi=new --enable-gnu-unique-object --disable-vtable-verify --enable-plugin --enable-default-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-tune=generic --enable-offload-targets=nvptx-none,hsa --without-cuda-driver --enable-checking=release --build=x86_64-linux-gnu --host=x86_64-linux-gnu --target=x86_64-linux-gnu
Thread model: posix
gcc version 9.3.0 (Ubuntu 9.3.0-10ubuntu2)
choijm@LAPTOP-LR5HQQBH:~/Syspro/LN4$
```

```
choijm@LAPTOP-LR5HQQBH: ~/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-LR5HQQBH: ~/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
.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 context switch works, memory alignment, ...
- Apprehend the technologies of IA
  - ✓ Pipelining
  - ✓ Dynamic execution
  - ✓ Cache (L1, L2, L3)
  - ✓ Superscalar
  - ✓ MMX
  - ✓ Hyper-threading
  - ✓ Multi core
  - ✓ Intel 64
  - ✓ Virtualization Technology

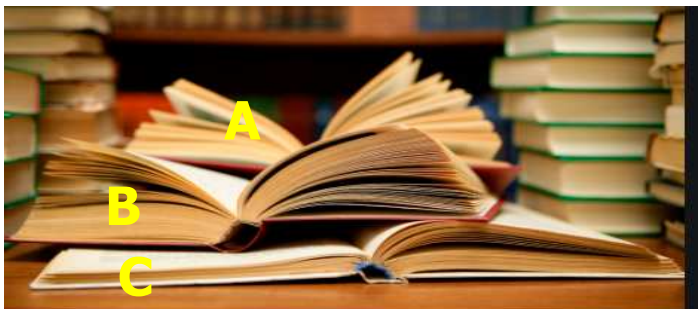




# Quiz for this Lecture

## ■ Quiz

- ✓ 1. Explain the differences between “movl \$array, %ebx” and “movl array, %ebx” in operand addressing modes.
- ✓ 2. Assume that a student reads three books (called A, B, C) in a library. Also assume that he/she reads a book for 10 minutes and turns to a next book. Explain the **context save** and **context restore** in this scenario.
- ✓ 3. Explain the key techniques of the dynamic execution in the Intel P6 **microarchitecture** (5 techniques)
- ✓ 4. Discuss what pipeline hazard can be occurred in the center below figure (from LN6) and how to overcome that hazard.
- ✓ 5. What are the Spectre vulnerabilities (or Meltdown) ? Explain it using the Intel technologies learned in this LN.
- ✓ 6. Discuss the differences between x86 (32-bit) and x86-64 (64-bit) in an assembly code (at least 3).



```

# Compile at gcc@ubuntu:~/syspro-examples/chap6
#include <stdio.h>

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

main()
{
    c = a + b;
    d = b + d;
    printf("c = %d, d = %d, e = %d\n", c, d, e);
}

# Compile at gcc@ubuntu:~/syspro-examples/chap6
pushl %ebx
movl $0, %ebx
subl $0, %ebx
andl $0, %esp
movl $0, %eax
addl $15, %eax
addl $4, %eax
shll $4, %eax
orl $0, %eax
subl %eax, %eax

movl %d, %eax
movl %eax, %eax
movl %eax, %eax

movl %d, %eax
movl $4, %eax
movl %eax, %d
movl %eax, %e

movl %d, %eax
orl $0, %eax
orl %eax, %d

pushl 0
popl %d

arith.exe: 61L, 030C 저장했습니다
  
```



(Source: [www.analyticsvidhya.com/blog/2019/01/27-amazing-data-science-books-every-data-scientist-should-read/](http://www.analyticsvidhya.com/blog/2019/01/27-amazing-data-science-books-every-data-scientist-should-read/))



# Appendix

## ■ Stack randomization

- ✓ To disable ASLR: `echo 0 > /proc/sys/kernel/randomize_va_space`
- ✓ To disable stack protector: `-fno-stack-protector`

```
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
0xff93297c -7132808
0xff932980 -1763943680
0xff932984 -135049216
0xff932988 0
0xff93298c -7132744
y = 112
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$ ./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$ gcc -fno-stack-protector stack_struct.c -m32
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$ ./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$ 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$
```