Computer Architecture

Trần Trọng Hiếu

Information Systems Department

Faculty of Information Technology

VNU - UET

hieutt@vnu.edu.vn

Intel-based Assembly

IA-32 Processor Architecture

- Modes of operation
- Basic execution environment
- Floating-point unit
- Intel Microprocessor history

Modes of Operation

- Protected mode
  - native mode (Windows, Linux)
- Real-address mode
  - native MS-DOS
- System management mode
  - power management, system security, diagnostics

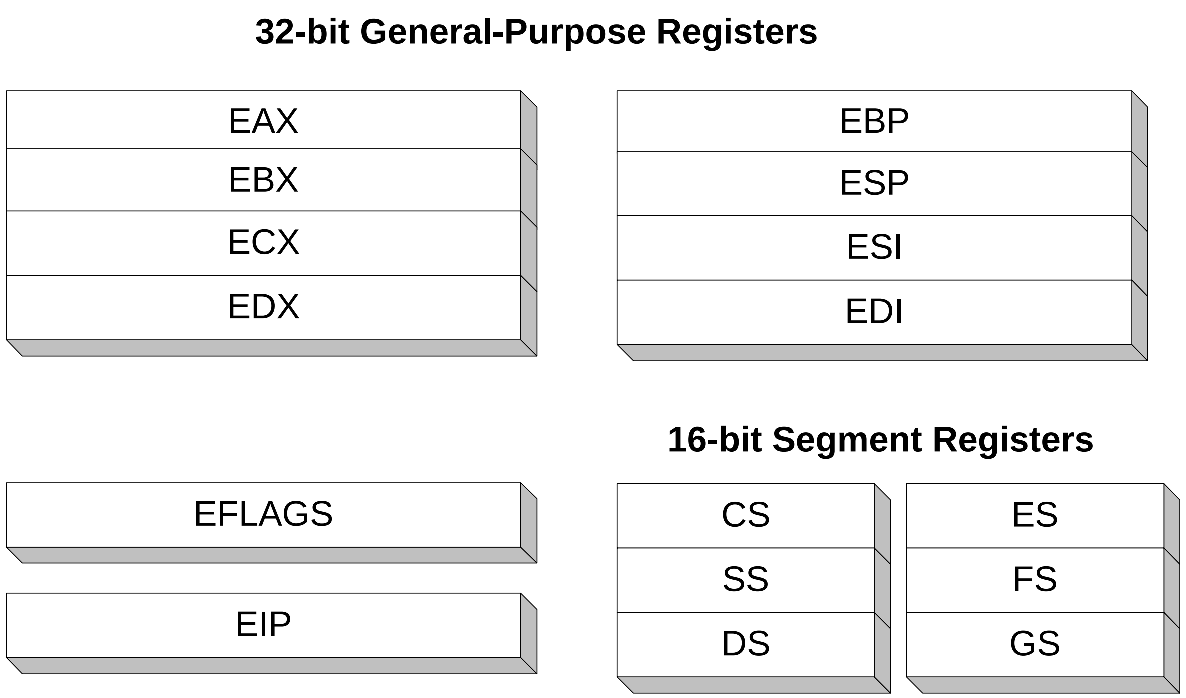
- Virtual-8086 mode
  - hybrid of Protected
  - each program has its own 8086 computer

Basic Execution Environment

- General-purpose registers
- Index and base registers
- Specialized register uses
- Status flags
- Floating-point, MMX, XMM registers

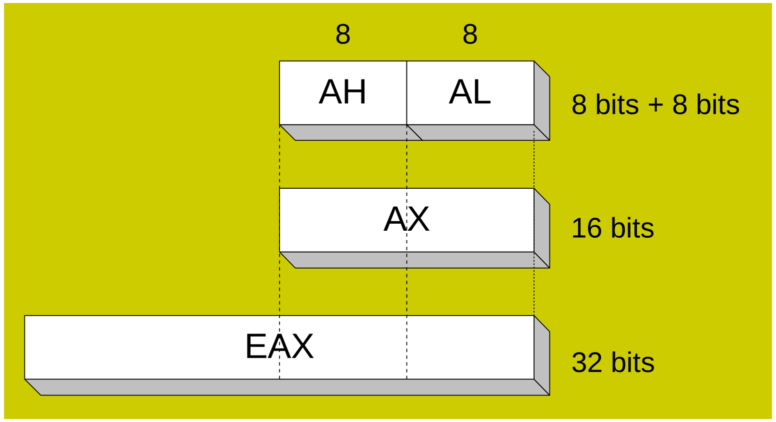
General-Purpose Registers

Named storage locations inside the CPU, optimized for speed.



Accessing Parts of Registers

- Use 8-bit name, 16-bit name, or 32-bit name
- Applies to EAX, EBX, ECX, and EDX



32-bit	16-bit	8-bit (high)	8-bit (low)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

Index and Base Registers

- Some registers have only a 16-bit name for their lower half:

32-bit	16-bit
ESI	SI
EDI	DI
EBP	BP
ESP	SP

Some Specialized Register Uses (1 of 2)

- General-Purpose
  - EAX – accumulator
  - ECX – loop counter
  - ESP – stack pointer
  - ESI, EDI – index registers
  - EBP – extended frame pointer (stack)
- Segment
  - CS – code segment
  - DS – data segment
  - SS – stack segment
  - ES, FS, GS - additional segments

Some Specialized Register Uses (2 of 2)

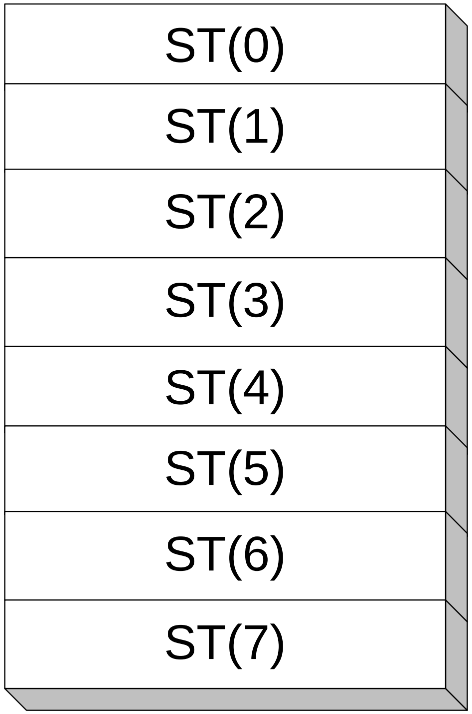
- EIP – instruction pointer
- EFLAGS
  - status and control flags
  - each flag is a single binary bit

Status Flags

- Carry
  - unsigned arithmetic out of range
- Overflow
  - signed arithmetic out of range
- Sign
  - result is negative
- Zero
  - result is zero
- Auxiliary Carry
  - carry from bit 3 to bit 4
- Parity
  - sum of 1 bits is an even number

Floating-Point, MMX, XMM Registers

- Eight 80-bit floating-point data registers
  - ST(0), ST(1), . . . , ST(7)
  - arranged in a stack
  - used for all floating-point arithmetic
- Eight 64-bit MMX registers
- Eight 128-bit XMM registers for single-instruction multiple-data (SIMD) operations



Intel Microprocessor History

- Intel 8086, 80286
- IA-32 processor family
- P6 processor family

Early Intel Microprocessors

- Intel 8080
  - 64K addressable RAM
  - 8-bit registers
  - CP/M operating system
  - S-100 BUS architecture
  - 8-inch floppy disks!
- Intel 8086/8088
  - IBM-PC Used 8088
  - 1 MB addressable RAM
  - 16-bit registers
  - 16-bit data bus (8-bit for 8088)
  - separate floating-point unit (8087)

Intel IA-32 Family

- Intel386
  - 4 GB addressable RAM, 32-bit registers, paging (virtual memory)
- Intel486
  - instruction pipelining
- Pentium
  - superscalar, 32-bit address bus, 64-bit internal data path

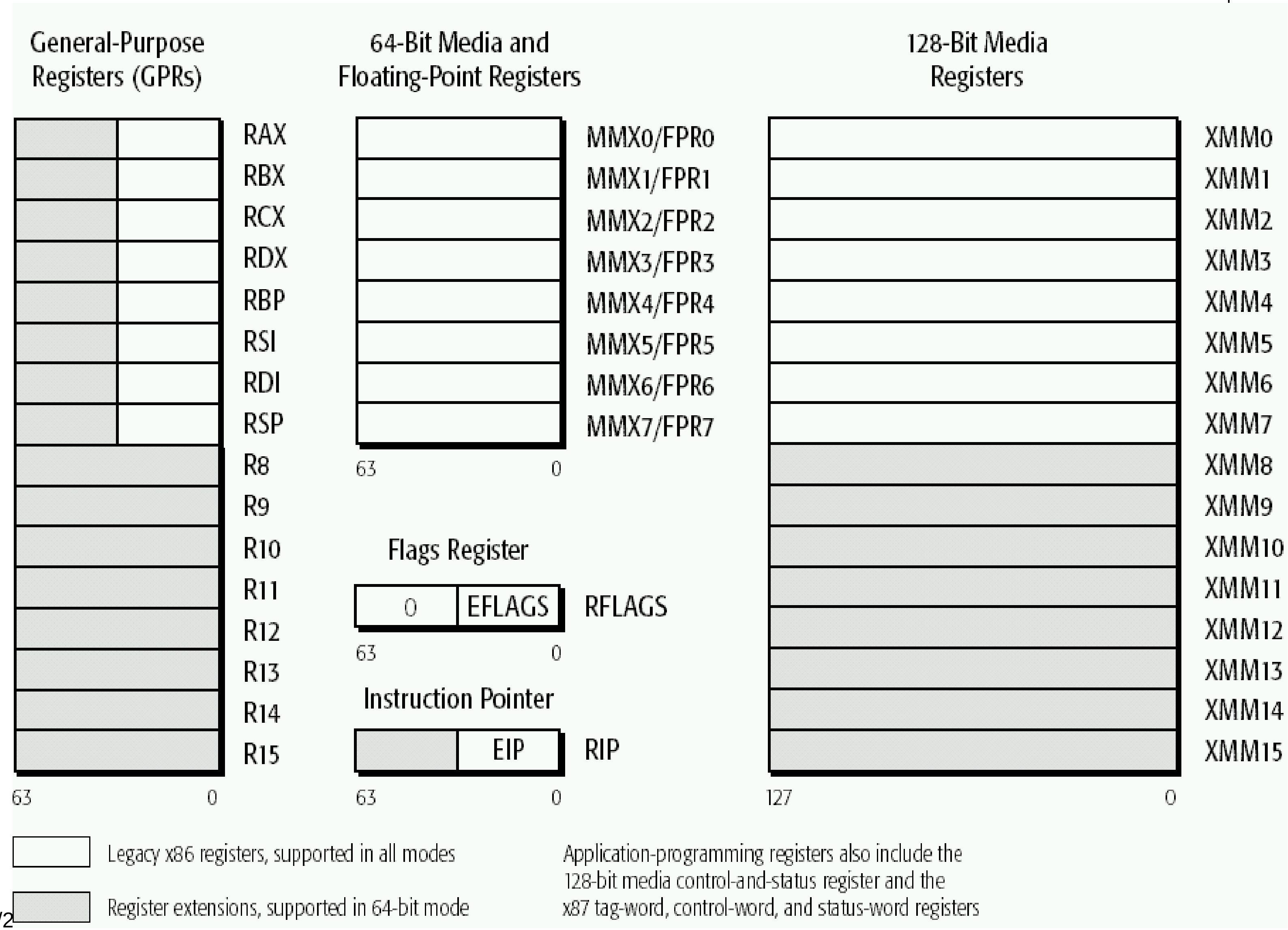
Intel P6 Family

- Pentium Pro
  - advanced optimization techniques in microcode
- Pentium II
  - MMX (multimedia) instruction set
- Pentium III
  - SIMD (streaming extensions) instructions
- Pentium 4
  - NetBurst micro-architecture, tuned for multimedia

X86_64

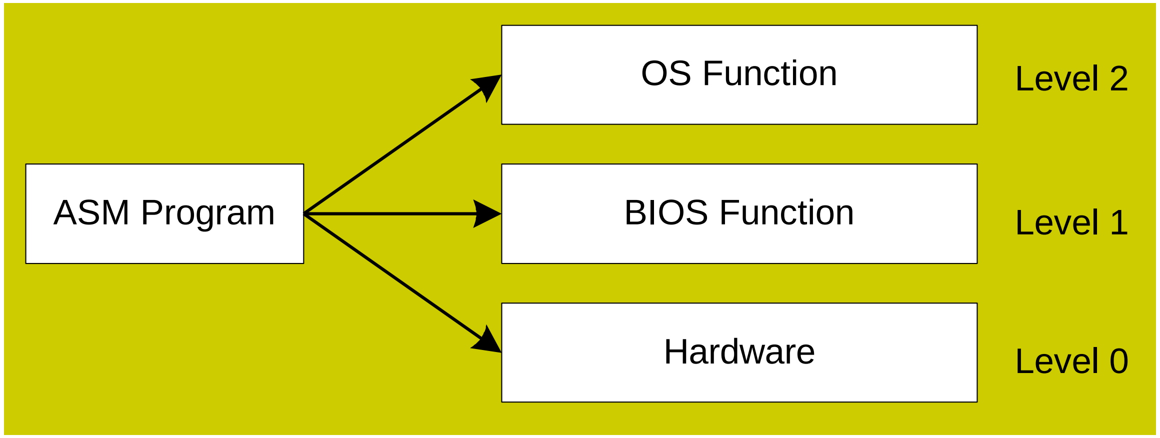
- AMD architecture
  - http://developer.amd.com/documentation/guides/Pages/default.aspx
- Expand the registers into 64bits, rax, rbx, rcx, rdx, …

X86_64 registers



ASM Programming levels

ASM programs can perform input-output at each of the following levels:



Program structure

```
.section .data
output: .asciz “The processor Vendor ID is ‘%s’\n”
.section .text
.globl _start
_start:
    program_body
```

Data Definition Statement

- A data definition statement sets aside storage in memory for a variable.
- Syntax:

  [*name:*] *directive initializer* [,*initializer*] . . .
- All initializers become binary data in memory

```
value1: .BYTE 'A'            # character constant
value2: .BYTE 0              # smallest unsigned byte
str: .asciz “Hello World”  # string
```

Operand Types

- Three basic types of operands:
  - Immediate – a constant integer (8, 16, or 32 bits)
    - value is encoded within the instruction
  - Register – the name of a register
    - register name is converted to a number and encoded within the instruction
  - Memory – reference to a location in memory
    - memory address is encoded within the instruction, or a register holds the address of a memory location

Instruction Operand Notation

Operand	Description
r8	8-bit general-purpose register: AH, AL, BH, BL, CH, CL, DH, DL
r16	16-bit general-purpose register: AX, BX, CX, DX, SI, DI, SP, BP
r32	32-bit general-purpose register: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP
reg	any general-purpose register
sreg	16-bit segment register: CS, DS, SS, ES, FS, GS
imm	8-, 16-, or 32-bit immediate value
imm8	8-bit immediate byte value
imm16	16-bit immediate word value
imm32	32-bit immediate doubleword value
r/m8	8-bit operand which can be an 8-bit general register or memory byte
r/m16	16-bit operand which can be a 16-bit general register or memory word
r/m32	32-bit operand which can be a 32-bit general register or memory doubleword
mem	an 8-, 16-, or 32-bit memory operand

MOV Instruction

- Move from source to destination. Syntax:

  MOV *source, destination*

  - Both operands must be the same size
  - No more than one memory operand permitted

```
.section .data
Output:  .asciz “The result is: ”
Val: .int 10
.section text
…
 movl $4, %eax
 movl $1, %ebx
 movl $output, %ecx
 movl $12, %edx
…
 movl val, %eax
…
 movl  %eax,val
```

Direct-Offset Operands

An offset is added to a data label to produce an effective address (EA).

```
arr: .int 34,3,12,4,3,5
…
 xor %edx,%edx
 movl arr(,%edx, 4),%ebx        # ebx = ?
 inc %edx
…
 movb %eax,arrB(, %ebx, 1)
 movb %al,[arrB+1]              # alternative notation
```

Addition and Subtraction

- INC and DEC Instructions
- ADD and SUB Instructions
- NEG Instruction
- Implementing Arithmetic Expressions
- Flags Affected by Arithmetic
  - Zero
  - Sign
  - Carry
  - Overflow

INC and DEC Instructions

- Add 1, subtract 1 from destination operand
  - operand may be register or memory
- INC *destination*
  - Logic: *destination* $\leftarrow$ *destination* + 1
- DEC *destination*
  - Logic: *destination* $\leftarrow$ *destination* – 1

ADD and SUB Instructions

- ADD source, destination
  - Logic: *destination* $\leftarrow$ *destination* + source
- SUB source, destination
  - Logic: *destination* $\leftarrow$ *destination* – source
- Same operand rules as for the MOV instruction

ADD and SUB Examples

```
var1: .int 10000h
var2: .int 20000h
…
    movl var1, %eax          # 00010000h
    movl var2, %ebx
    add %ebx,%eax,           # 00030000h
    add $0xFFFF,%ax          # 0003FFFFh
    add $1,%eax
     sub  $1,%ax
```

NEG (negate) Instruction

Reverses the sign of an operand. Operand can be a register or memory operand.

```
valB: .BYTE -1
valW .int +32767
…
    movb valB,%al                  # AL = -1
    neg %al                        # AL = +1
    neg valW                       # valW = -32767
```

Flags Affected by Arithmetic

- The ALU has a number of status flags that reflect the outcome of arithmetic (and bitwise) operations
  - based on the contents of the destination operand
- Essential flags:
  - Zero flag – set when destination equals zero
  - Sign flag – set when destination is negative
  - Carry flag – set when unsigned value is out of range
  - Overflow flag – set when signed value is out of range
- The MOV instruction never affects the flags.

Zero Flag (ZF)

The Zero flag is set when the result of an operation produces zero in the destination operand.

```
movw $1,%cx
sub $1,%cx              # CX = 0, ZF = 1
movw $0xFFFF,%ax
inc %ax                 # AX = 0, ZF = 1
inc %ax                 # AX = 1, ZF = 0
```

Remember...

- A flag is set when it equals 1.
- A flag is clear when it equals 0.

JMP Instruction

- JMP is an unconditional jump to a label that is usually within the same procedure.
- Syntax: JMP *target*
- Logic: EIP $\leftarrow$ *target*
- Example:

```
top:
    .
    .
    jmp top
```

A jump outside the current procedure must be to a special type of label called a global label (see Section 5.5.2.3 for details).

LOOP Instruction

- The LOOP instruction creates a counting loop
- Syntax: LOOP *target*
- Logic:
  - ECX ← ECX – 1
  - if ECX != 0, jump to *target*
- Implementation:
  - The assembler calculates the distance, in bytes, between the offset of the following instruction and the offset of the target label. It is called the relative offset.
  - The relative offset is added to EIP.

Status Flags - Review

- The Zero flag is set when the result of an operation equals zero.
- The Carry flag is set when an instruction generates a result that is too large (or too small) for the destination operand.
- The Sign flag is set if the destination operand is negative, and it is clear if the destination operand is positive.
- The Overflow flag is set when an instruction generates an invalid signed result (bit 7 carry is XORed with bit 6 Carry).
- The Parity flag is set when an instruction generates an even number of 1 bits in the low byte of the destination operand.
- The Auxiliary Carry flag is set when an operation produces a carry out from bit 3 to bit 4

TEST Instruction

- Performs a nondestructive AND operation between each pair of matching bits in two operands
- No operands are modified, but the Zero flag is affected.
- Example: jump to a label if either bit 0 or bit 1 in AL is set.

```
test $11,%al
jnz  ValueFound
```

CMP Instruction (1 of 3)

- Compares the destination operand to the source operand
  - Nondestructive subtraction of source from destination (destination operand is not changed)
- Syntax: CMP *source, destination*

```
mov $5,%al
cmp %al,%bl                    # Zero flag set
```

- Example: destination == source?

J*cond* Instruction

- A conditional jump instruction branches to a label when specific register or flag conditions are met
- Examples:
  - JB, JC jump to a label if the Carry flag is set
  - JE, JZ jump to a label if the Zero flag is set
  - JS jumps to a label if the Sign flag is set
  - JNE, JNZ jump to a label if the Zero flag is clear
  - JECXZ jumps to a label if ECX equals 0

Jumps Based on Specific Flags

Mnemonic	Description	Flags
JZ	Jump if zero	ZF = 1
JNZ	Jump if not zero	ZF = 0
JC	Jump if carry	CF = 1
JNC	Jump if not carry	CF = 0
JO	Jump if overflow	OF = 1
JNO	Jump if not overflow	OF = 0
JS	Jump if signed	SF = 1
JNS	Jump if not signed	SF = 0
JP	Jump if parity (even)	PF = 1
JNP	Jump if not parity (odd)	PF = 0

Jumps Based on Equality

Mnemonic	Description
JE	Jump if equal (*leftOp* = *rightOp*)
JNE	Jump if not equal (*leftOp* ≠ *rightOp*)
JCXZ	Jump if CX = 0
JECXZ	Jump if ECX = 0

Jumps Based on Unsigned Comparisons

Mnemonic	Description	
JA	Jump if above (if *leftOp* > *rightOp*)	CF=0 and ZF=0
JNBE	Jump if not below or equal (same as JA)	CF=0 and ZF=0
JAE	Jump if above or equal (if *leftOp* >= *rightOp*)	CF=0
JNB	Jump if not below (same as JAE)	CF=0
JB	Jump if below (if *leftOp* < *rightOp*)	CF=1
JNAE	Jump if not above or equal (same as JB)	CF=1
JBE	Jump if below or equal (if *leftOp* <= *rightOp*)	CF=1 or ZF=1
JNA	Jump if not above (same as JBE)	CF=1 or ZF=1

Jumps Based on Signed Comparisons

Mnemonic	Description	
JG	Jump if greater (if *leftOp* > *rightOp*)	SF=OF and ZF=0
JNLE	Jump if not less than or equal (same as JG)	
JGE	Jump if greater than or equal (if *leftOp* >= *rightOp*)	SF=OF SF=OF
JNL	Jump if not less (same as JGE)	SF=OF
JL	Jump if less (if *leftOp* < *rightOp*)	SF≠OF SF≠OF
JNGE	Jump if not greater than or equal (same as JL)	SF≠OF
JLE	Jump if less than or equal (if *leftOp* <= *rightOp*)	SF≠OF or ZF=1
JNG	Jump if not greater (same as JLE)	

Applications

- Task: Jump to a label if unsigned EAX is greater than EBX
- Solution: Use CMP, followed by JA

```
cmp %ebx, %eax
ja  Larger
```

- Task: Jump to a label if signed EAX is greater than EBX
- Solution: Use CMP, followed by JG

```
cmp %ebx,%eax
jg  Greater
```

Conditional Structures

- Block-Structured IF Statements
- Compound Expressions with AND
- Compound Expressions with OR
- WHILE Loops
- Table-Driven Selection

Block-Structured IF Statements

Assembly language programmers can easily translate logical statements written in C++/Java into assembly language. For example:

```
if( op1 == op2 )
  X = 1#
else
  X = 2#
```

```
    mov op1,%eax
    mov op2,%ebx
    cmp %eax,%ebx
    jne L1
    movl $1,X
    jmp L2
L1: movl $2,X
L2:
```

Your turn . . .

Implement the following pseudocode in assembly language. All values are unsigned:

```
if( ebx <= ecx )
{
  eax = 5#
  edx = 6#
}
```

```
    cmp %ebx,%ecx
    ja  next
    mov $5, %eax
    mov $6,%edx
next:
```

(There are multiple correct solutions to this problem.)

Compound Expression with AND (2 of 3)

```
if (al > bl) AND (bl > cl)
  X = 1#
```

This is one possible implementation . . .

```
    cmp %al,%bl             # first expression...
    ja  L1
    jmp next
L1:
    cmp %bl,%cl             # second expression...
    ja  L2
    jmp next
L2:                         # both are true
    mov $1,X                # set X to 1
next:
```

Compound Expression with AND (3 of 3)

```
if (al > bl) AND (bl > cl)
  X = 1#
```

But the following implementation uses 29% less code by reversing the first relational operator. We allow the program to "fall through" to the second expression:

```
    cmp %al,%bl                 # first expression...
    jbe next                    # quit if false
    cmp %bl,%cl                 # second expression...
    jbe next                    # quit if false
    mov $1,X                    # both are true
next:
```

Your turn . . .

Implement the following pseudocode in assembly language. All values are unsigned:

```
if( ebx <= ecx
    && ecx > edx )
{
  eax = 5#
  edx = 6#
}
```

```
     cmp %ebx,%ecx
     ja  next
     cmp %ecx,%edx
     jbe next
     mov $5,%eax
     mov $6,%edx
next:
```

(There are multiple correct solutions to this problem.)

Compound Expression with OR (1 of 2)

- When implementing the logical OR operator, consider that HLLs use short-circuit evaluation
- In the following example, if the first expression is true, the second expression is skipped:

```
if (al > bl) OR (bl > cl)
  X = 1#
```

Compound Expression with OR (1 of 2)

```
if (al > bl) OR (bl > cl)
  X = 1#
```

We can use "fall-through" logic to keep the code as short as possible:

```
    cmp %al,%bl                 # is AL > BL?
    ja  L1                      # yes
    cmp %bl,%cl                 # no: is BL > CL?
    jbe next                    # no: skip next statement
L1: mov $1, X                   # set X to 1
next:
```

WHILE Loops

A WHILE loop is really an IF statement followed by the body of the loop, followed by an unconditional jump to the top of the loop. Consider the following example:

```
while( eax < ebx)
    eax = eax + 1#
```

This is a possible implementation:

```
top: cmp %eax,%ebx              # check loop condition
     jae next                   # false? exit loop
     inc %eax                   # body of loop
     jmp top                    # repeat the loop
next:
```

Your turn . . .

Implement the following loop, using unsigned 32-bit integers:

```
while( ebx <= val1)
{
    ebx = ebx + 5#
    val1 = val1 - 1
}
```

```
top: cmp ebx,val1              # check loop condition
     ja  next                  # false? exit loop
     add ebx,5                 # body of loop
     dec val1
     jmp top                   # repeat the loop
next:
```

Procedure

- Define

```
convert:
mov $10,%ebx
xor %ecx, %ecx
…
ret
```

- Call

```
call convert
```

System call

- Each system call has different arguments
- Assign parameters to appropriate registers
- Use int 0x80
- Example

Exit from the program

```
movl $0, %ebx
movl $1, %eax
int $0x80
```

Print a string

```
msg: .asciz "Hello World"
movl $4, %eax
movl $1, %ebx
movl $msg, %ecx
movl $10, %edx
int $0x80
```

Call C library function

- Differ from 32bit to 64bit OS

64bit architecture

Use registers and stack to pass arguments

63	31	15	8	7 0	
%rax	%eax	%ax	%ah	%al	Return value
%rbx	%ebx	%ax	%bh	%bl	Callee saved
%rcx	%ecx	%cx	%ch	%cl	4th argument
%rdx	%edx	%dx	%dh	%dl	3rd argument
%rsi	%esi	%si		%sil	2nd argument
%rdi	%edi	%di		%dil	1st argument
%rbp	%ebp	%bp		%bpl	Callee saved
%rsp	%esp	%sp		%spl	Stack pointer
%r8	%r8d	%r8w		%r8b	5th argument
%r9	%r9d	%r9w		%r9b	6th argument

Call C library function

- 64bit architecture: use registers to pass arguments

```
.section .data
format_string:    .asciz "Vendor ID: %d\n"
vendor_id: .int 12
.section .text
.globl _start
_start:
#Arguments to C functions:
movl $format_string, %edi
movl vendor_id, %esi
movl $0, %eax
call printf
call exit
```

printf(“Vendor ID is: %d\n”,id);

Call C library function

- 32bit architecture
  - Use stack to pass arguments

```
.section .data
output: .asciz "The Vendor ID is '%d'\n"
buffer: .byte 12
.section .text
.globl _start
_start:
push $12
push $output
call printf
addl $8, %esp
push $0
call exit
```

Development tools

- compiler: as, linker: ld, debugger: gdb

```
.section .data
output: .asciz "The Vendor ID is '%d'\n“
vendor_id : .byte 12
.section .text
.globl _start
_start:
movl $format_string, %edi
movl vendor_id, %esi
movl $0, %eax
call printf
call exit
```

Compile, link and run the program

$ as –o print.o printf.s

$ ld –dynamic-linker /lib64/ld-linux-x86-64.so.2 –lc –o print print.o

$ ./print

Exercises

- Write a procedure to print a number (in %eax)
- Write a program to print the value of factorial N (N!)
- Write a program to print the value of factorial N (N!) in a recursive procedure
- Write a program to print the product of two integer numbers (a*b) by an addition procedure
- Write a program to print the dividend of two integer numbers (a%b) by a recursive subtraction procedure
- Write a program to calculate the sum of an array
- Write a program to calculate the sum of the first n natural numbers (1+2+3+…+n)

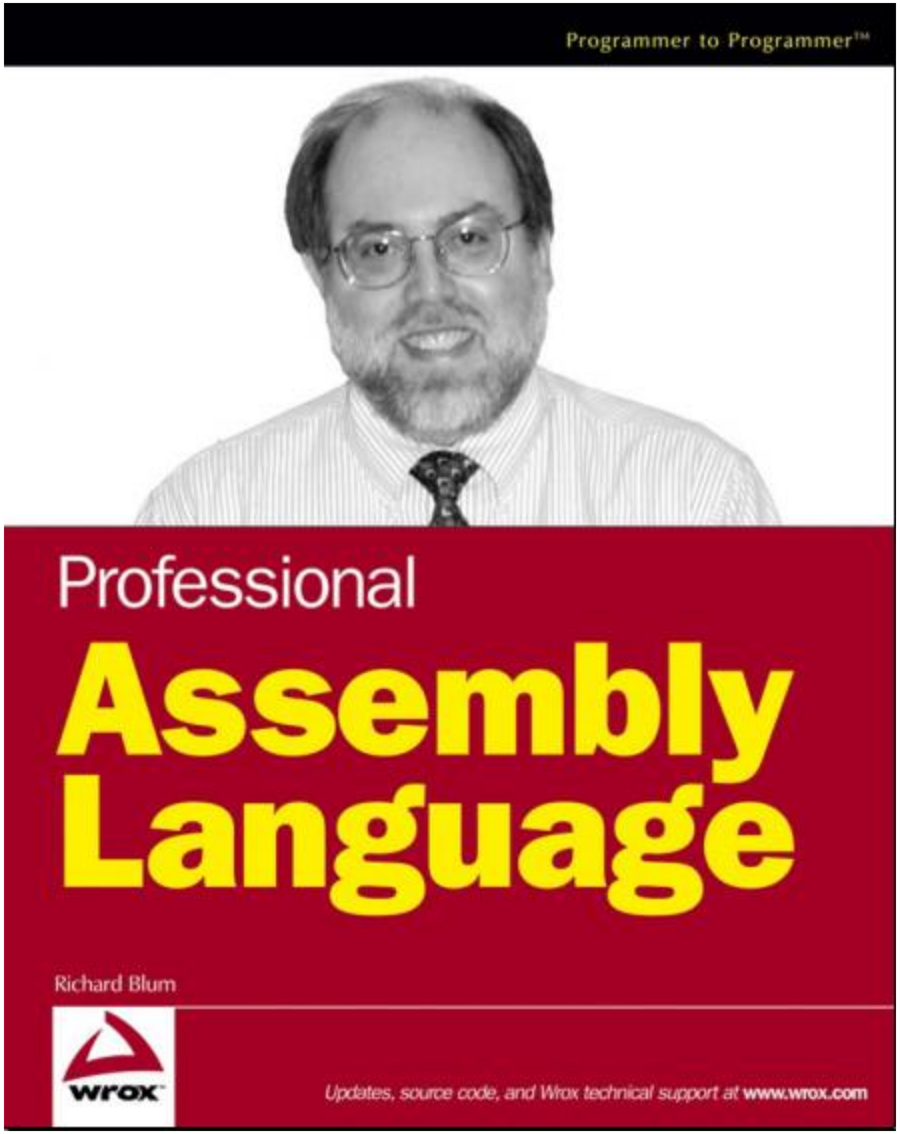
Exercises cont'd

- Write a program to print the first *n* fibonaci numbers
- Write a program to print the first of *n* numbers of a *geometric sequence* with a given value of *a* and *r*
- Write a program to print the first of n number in an *arithmetic sequence* with a given value of *d* and *u*
- Write a program to find out the greatest common divisor of the two numbers *a* and *b*
- Write a program to find out the *lowest common multiple* of the two numbers *a* and *b*
- Write a program to sort an array

Reference

- Professional Assembly Language,

  Richard Blum, 2005



Reference

- Assembly Language for Intel-Based Computers,

  Kip R.Irvine, 2003