Intel x86 Assembly Fundamentals

Computer Organization and Assembly Languages

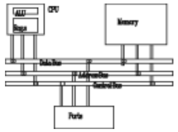
Yung-Yu Chuang

2007/12/10

with slides by Kip Irvine

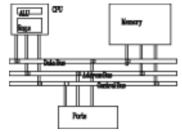
Intel microprocessor history

Early Intel microprocessors



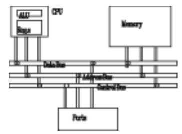
- Intel 8080 (1972)
  - 64K addressable RAM
  - 8-bit registers
  - CP/M operating system
  - 5,6,8,10 MHz
  - 29K transistros
- Intel 8086/8088 (1978) ⟵ my first computer
  - IBM-PC used 8088
  - 1 MB addressable RAM
  - 16-bit registers
  - 16-bit data bus (8-bit for 8088)
  - separate floating-point unit (8087)
  - used in low-cost microcontrollers now

The IBM-AT



- Intel 80286 (1982)
  - 16 MB addressable RAM
  - Protected memory
  - several times faster than 8086
  - introduced IDE bus architecture
  - 80287 floating point unit
  - Up to 20MHz
  - 134K transistors

Intel IA-32 Family



- Intel386 (1985)
  - 4 GB addressable RAM
  - 32-bit registers
  - paging (virtual memory)
  - Up to 33MHz
- Intel486 (1989)
  - instruction pipelining
  - Integrated FPU
  - 8K cache
- Pentium (1993)
  - Superscalar (two parallel pipelines)

Intel P6 Family

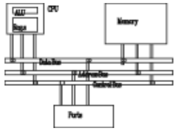
- Pentium Pro (1995)
  - advanced optimization techniques in microcode
  - More pipeline stages
  - On-board L2 cache
- Pentium II (1997)
  - MMX (multimedia) instruction set
  - Up to 450MHz
- Pentium III (1999)
  - SIMD (streaming extensions) instructions (SSE)
  - Up to 1+GHz
- Pentium 4 (2000)
  - NetBurst micro-architecture, tuned for multimedia
  - 3.8+GHz
- Pentium D (2005, Dual core)

IA-32 Architecture

IA-32 architecture

- Lots of architecture improvements, pipelining, superscalar, branch prediction, hyperthreading and multi-core.
- From programmer's point of view, IA-32 has not changed substantially except the introduction of a set of high-performance instructions

Modes of operation



- Protected mode
  - native mode (Windows, Linux), full features, separate memory

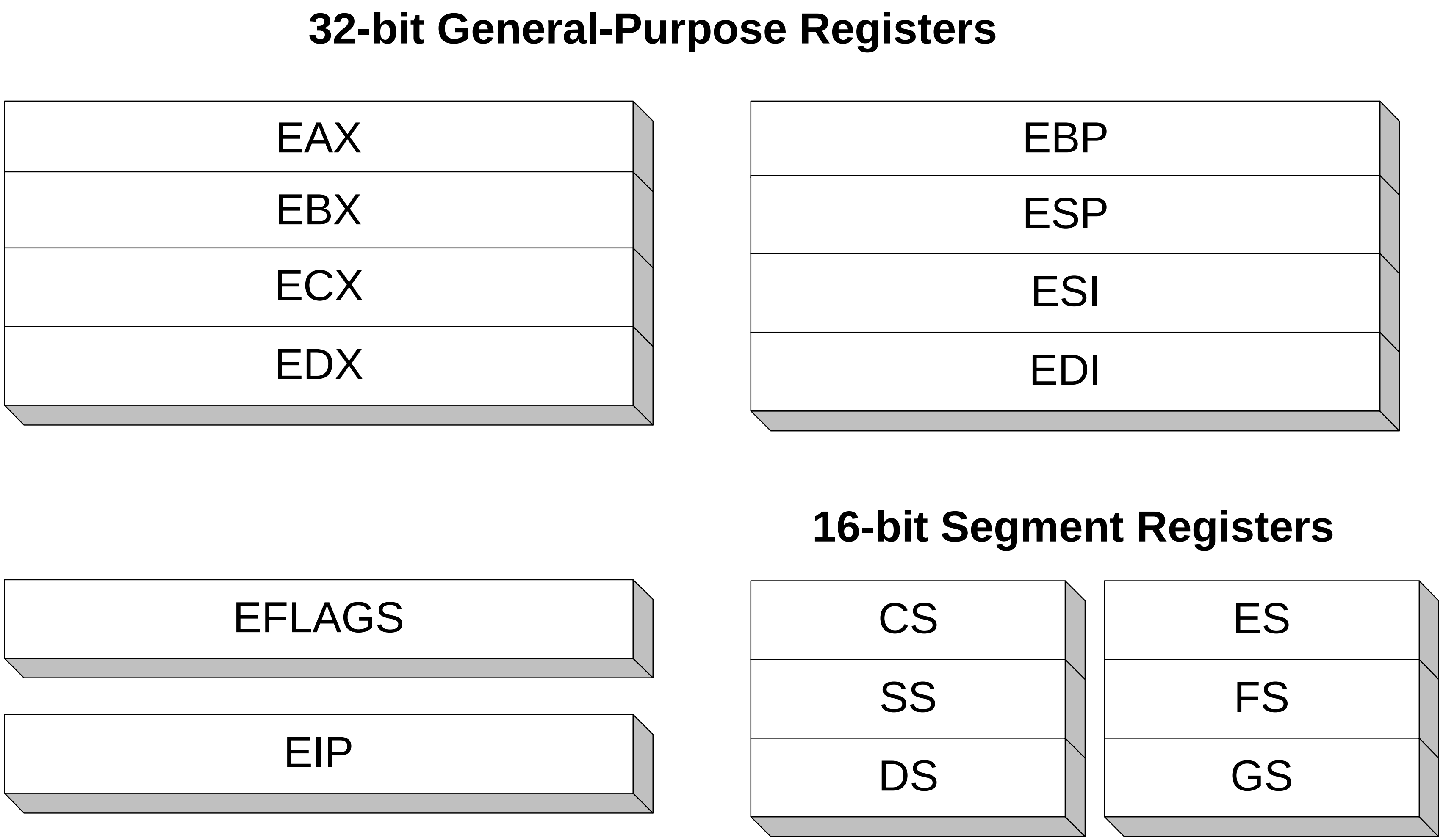
- Virtual-8086 mode
  - hybrid of Protected
  - each program has its own 8086 computer

- Real-address mode
  - native MS-DOS
- System management mode
  - power management, system security, diagnostics

Addressable memory

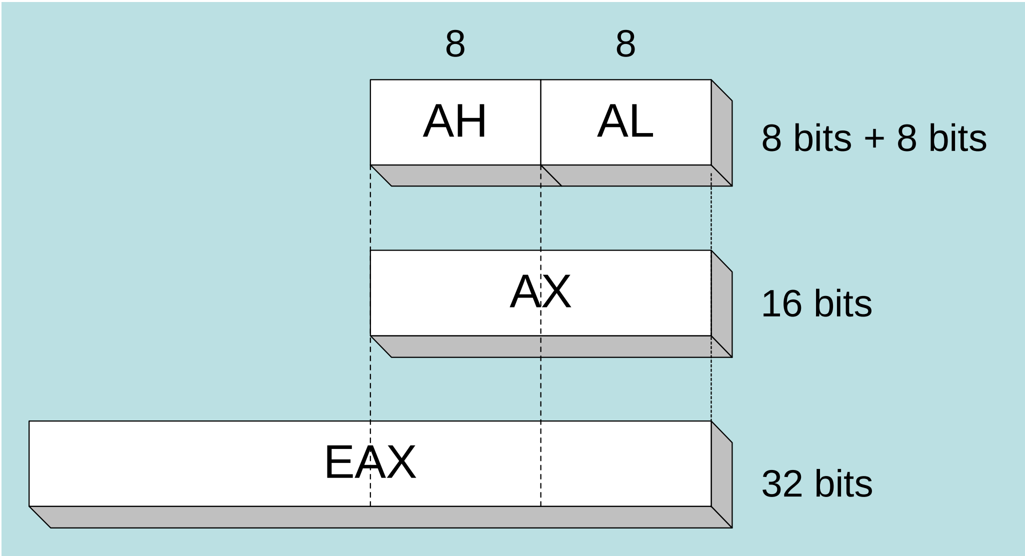
- Protected mode
  - 4 GB
  - 32-bit address
- Real-address and Virtual-8086 modes
  - 1 MB space
  - 20-bit address

General-purpose registers



Accessing parts of registers

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



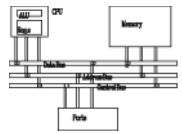
32-bit	16-bit	8-bit (high)	8-bit (low)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

Index and base registers

- Some registers have only a 16-bit name for their lower half (no 8-bit aliases). The 16-bit registers are usually used only in real-address mode.

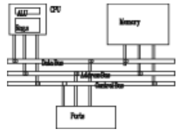
32-bit	16-bit
ESI	SI
EDI	DI
EBP	BP
ESP	SP

Some specialized register uses (1 of 2)



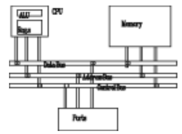
- General-Purpose
  - EAX – accumulator (automatically used by division and multiplication)
  - ECX – loop counter
  - ESP – stack pointer (should never be used for arithmetic or data transfer)
  - ESI, EDI – index registers (used for high-speed memory transfer instructions)
  - EBP – extended frame pointer (stack)

Some specialized register uses (2 of 2)



- Segment
  - CS – code segment
  - DS – data segment
  - SS – stack segment
  - ES, FS, GS - additional segments
- EIP – instruction pointer
- EFLAGS
  - status and control flags
  - each flag is a single binary bit (*set* or *clear*)
- Some other system registers such as IDTR, GDTR, LDTR etc.

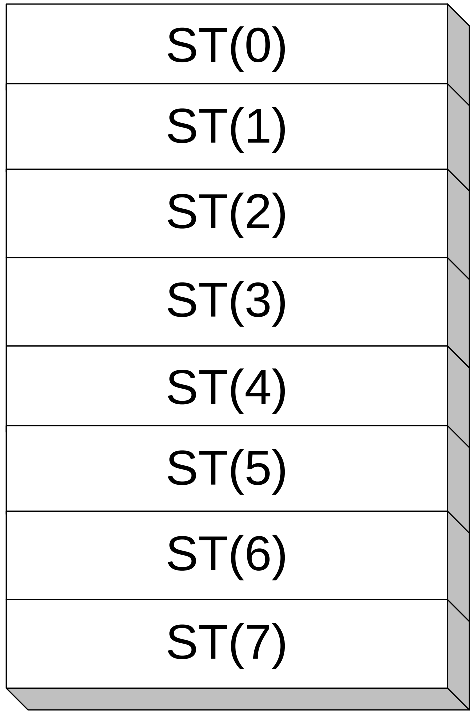
Status flags



- 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

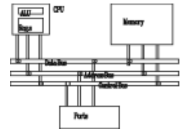
Floating-point, MMX, XMM registers

- 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



IA-32 Memory Management

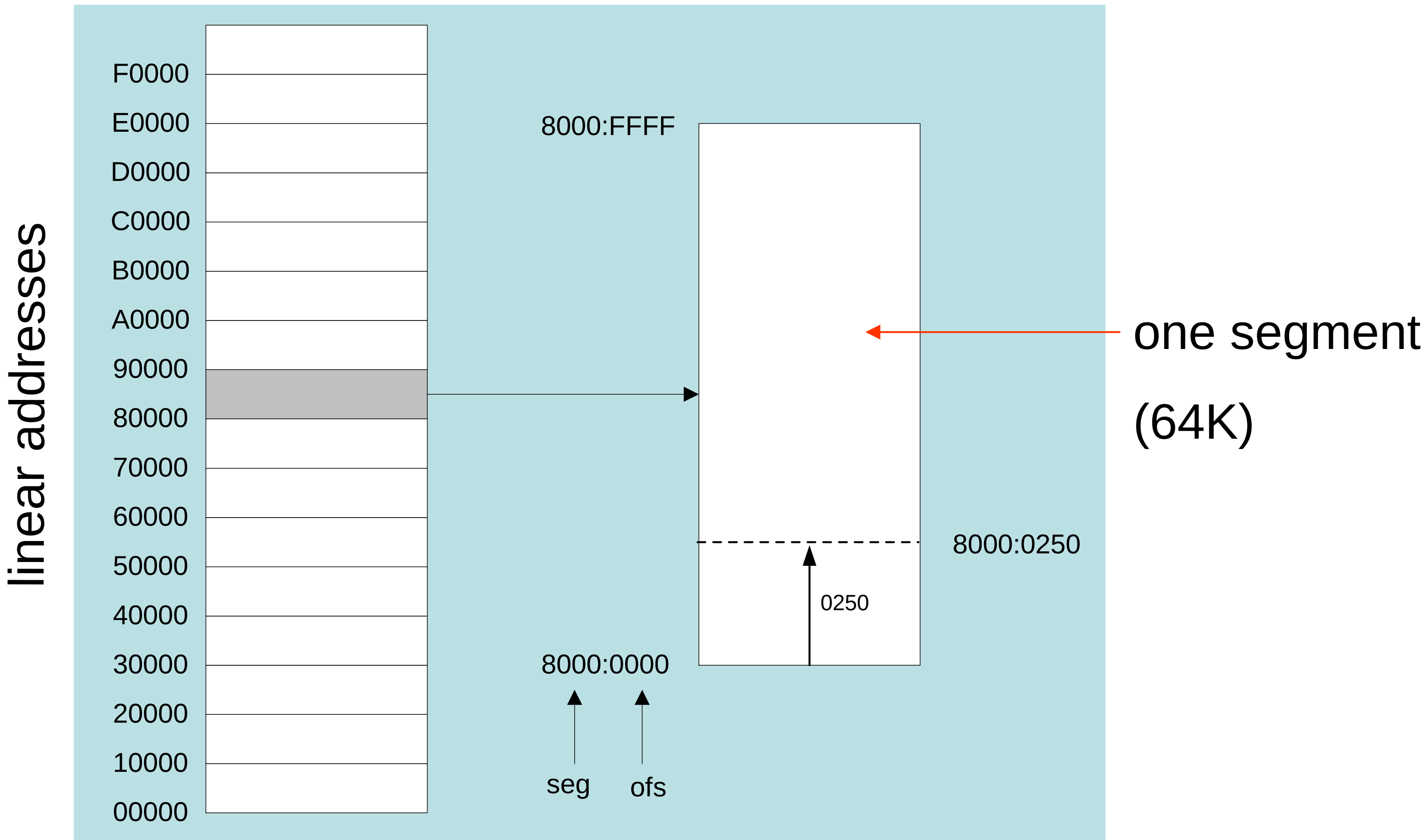
Real-address mode



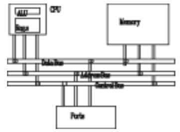
- 1 MB RAM maximum addressable (20-bit address)
- Application programs can access any area of memory
- Single tasking
- Supported by MS-DOS operating system

Segmented memory

Segmented memory addressing: absolute (linear) address is a combination of a 16-bit segment value added to a 16-bit offset



Calculating linear addresses



- Given a segment address, multiply it by 16 (add a hexadecimal zero), and add it to the offset
- Example: convert 08F1:0100 to a linear address

```
Adjusted Segment value: 0 8 F 1 0

Add the offset:           0 1 0 0

Linear address:         0 9 0 1 0
```

- A typical program has three segments: code, data and stack. Segment registers CS, DS and SS are used to store them separately.

Example

What linear address corresponds to the segment/offset address 028F:0030?

028F0 + 0030 = 02920

Always use hexadecimal notation for addresses.

Protected mode (1 of 2)

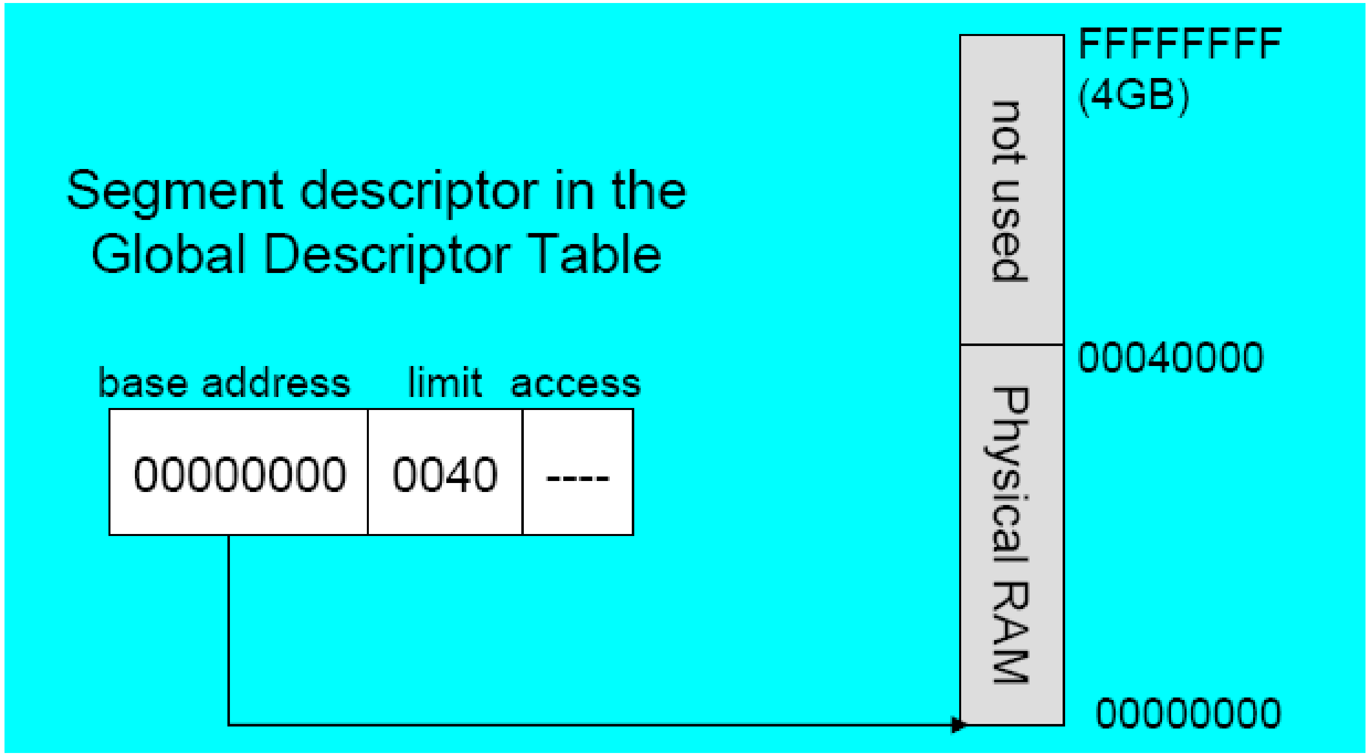
- 4 GB addressable RAM (32-bit address)
  - (00000000 to FFFFFFFFh)
- Each program assigned a memory partition which is protected from other programs
- Designed for multitasking
- Supported by Linux & MS-Windows

Protected mode (2 of 2)

- Segment descriptor tables
- Program structure
  - code, data, and stack areas
  - CS, DS, SS segment descriptors
  - global descriptor table (GDT)
- MASM Programs use the Microsoft flat memory model

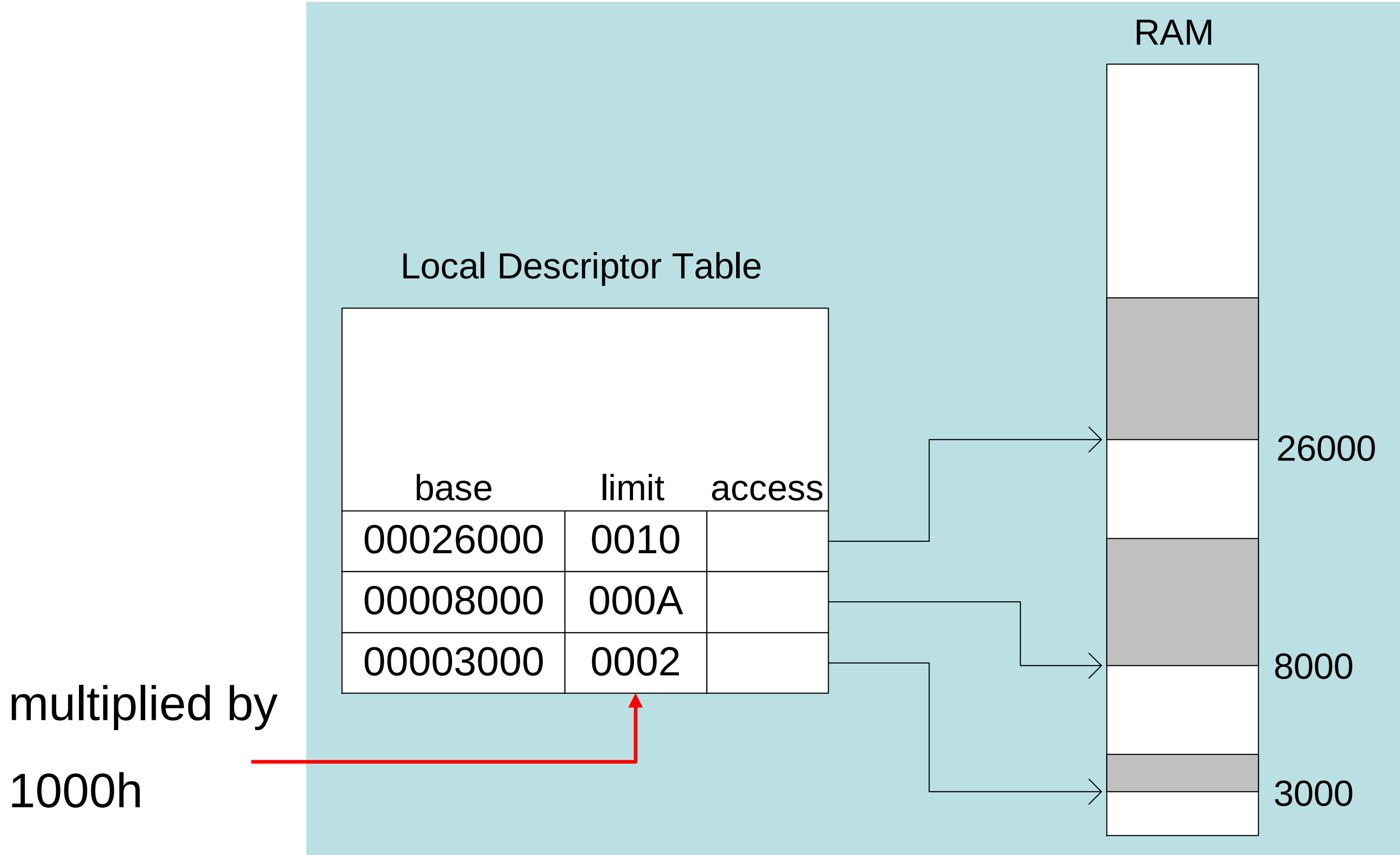
Flat segmentation model

- All segments are mapped to the entire 32-bit physical address space, at least two, one for data and one for code
- global descriptor table (GDT)

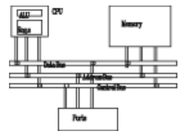


Multi-segment model

- Each program has a local descriptor table (LDT)
  - holds descriptor for each segment used by the program



Paging



- Virtual memory uses disk as part of the memory, thus allowing sum of all programs can be larger than physical memory
- Divides each segment into 4096-byte blocks called pages
- Page fault (supported directly by the CPU) – issued by CPU when a page must be loaded from disk
- Virtual memory manager (VMM) – OS utility that manages the loading and unloading of pages

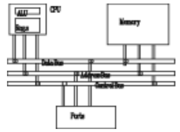
x86 Assembly Language Fundamentals

Instructions

- Assembled into machine code by assembler
- Executed at runtime by the CPU
- Member of the Intel IA-32 instruction set
- Four parts
  - Label (optional)
  - Mnemonic (required)
  - Operand (usually required)
  - Comment (optional)

Label:	Mnemonic	Operand(s)	;Comment

Labels

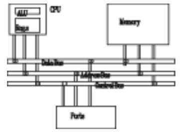


- Act as place markers
  - marks the address (offset) of code and data
- Easier to memorize and more flexible

  `mov ax, [0020]` → **`mov ax, val`**
- Follow identifier rules
- Data label
  - must be unique
  - example: **`myArray  BYTE  10`**
- Code label (ends with a colon)
  - target of jump and loop instructions
  - example:

```
L1: mov ax, bx
    ...
    jmp L1
```

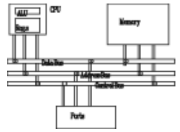
Reserved words and identifiers



- Reserved words (Appendix D) cannot be used as identifiers
  - Instruction mnemonics, directives, type attributes, operators, predefined symbols
- Identifiers
  - 1-247 characters, including digits
  - case insensitive (by default)
  - first character must be a letter, **`_`**, **`@`**, or **`$`**
  - examples:

```
var1       Count     $first
_main      MAX       open_file
@@myfile   xVal      _12345
```

Mnemonics and operands



- Instruction mnemonics
  - "reminder"
  - examples: **`MOV, ADD, SUB, MUL, INC, DEC`**
- Operands
  - constant (immediate value), **`96`**
  - constant expression, **`2+4`**
  - Register, **`eax`**
  - memory (data label), **`count`**
- Number of operands: 0 to 3
  - **`stc                ; set Carry flag`**
  - **`inc ax             ; add 1 to ax`**
  - **`mov count, bx      ; move BX to count`**

Directives

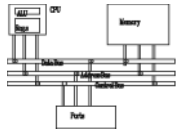
- Commands that are recognized and acted upon by the assembler
  - Part of assembler's syntax but not part of the Intel instruction set
  - Used to declare code, data areas, select memory model, declare procedures, etc.
  - case insensitive
- Different assemblers have different directives
  - NASM != MASM, for example
- Examples: **`.data`** **`.code`** **`PROC`**

Comments

- Comments are good!
  - explain the program's purpose
  - tricky coding techniques
  - application-specific explanations
- Single-line comments
  - begin with semicolon (;)
- block comments
  - begin with COMMENT directive and a programmer-chosen character and end with the same programmer-chosen character

```
COMMENT !
    This is a comment
    and this line is also a comment
!
```

Example: adding/subtracting integers



directive marking a comment

```
TITLE Add and Subtract              (AddSub.asm)
```

comment

```
; This program adds and subtracts 32-bit integers.

INCLUDE Irvine32.inc
.code
main PROC
    mov eax,10000h          ; EAX = 10000h
    add eax,40000h          ; EAX = 50000h
    sub eax,20000h          ; EAX = 30000h
    call DumpRegs           ; display registers
    exit
main ENDP
END main
```

copy definitions from Irvine32.inc

code segment. 3 segments: code, data, stack

beginning of a procedure

source

destination

defined in Irvine32.inc to end a program

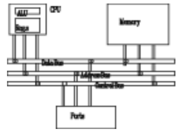
marks the last line and define the startup procedure

Example output

Program output, showing registers and flags:

```
EAX=00030000  EBX=7FFDF000  ECX=00000101  EDX=FFFFFFFF
ESI=00000000  EDI=00000000  EBP=0012FFF0  ESP=0012FFC4
EIP=00401024  EFL=00000206  CF=0  SF=0  ZF=0  OF=0
```

Alternative version of AddSub



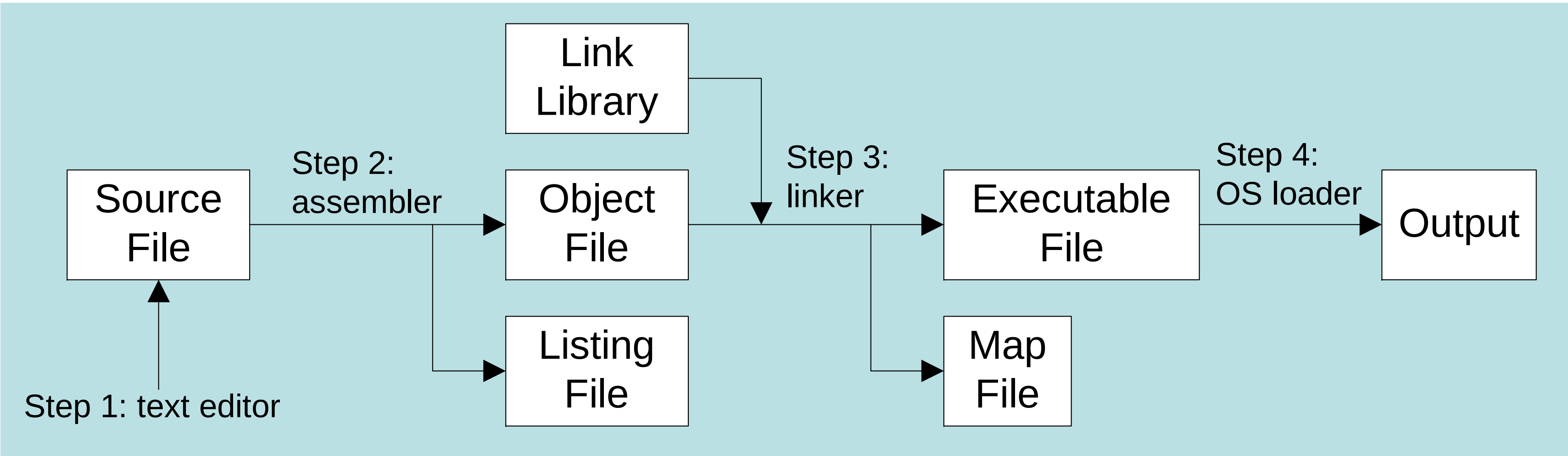
```
TITLE Add and Subtract                    (AddSubAlt.asm)

; This program adds and subtracts 32-bit integers.
.386
.MODEL flat,stdcall
.STACK 4096

ExitProcess PROTO, dwExitCode:DWORD
DumpRegs PROTO

.code
main PROC
   mov eax,10000h                 ; EAX = 10000h
   add eax,40000h                 ; EAX = 50000h
   sub eax,20000h                 ; EAX = 30000h
   call DumpRegs
   INVOKE ExitProcess,0
main ENDP
END main
```

Program template

```
TITLE Program Template            (Template.asm)

; Program Description:
; Author:
; Creation Date:
; Revisions:
; Date:              Modified by:

INCLUDE Irvine32.inc
.data
    ; (insert variables here)
.code
main PROC
    ; (insert executable instructions here)
    exit
main ENDP
    ; (insert additional procedures here)
END main
```

Assemble-link execute cycle

- The following diagram describes the steps from creating a source program through executing the compiled program.
- If the source code is modified, Steps 2 through 4 must be repeated.



Defining data

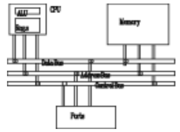
Integer constants

- [{+|-}] *digits* [*radix*]
- Optional leading + or – sign
- binary, decimal, hexadecimal, or octal digits
- Common radix characters:
  - **h** – hexadecimal
  - **d** – decimal (default)
  - **b** – binary
  - **r** – encoded real
  - **o** – octal

Examples: **`30d, 6Ah, 42, 42o, 1101b`**

Hexadecimal beginning with letter: **`0A5h`**

Integer expressions



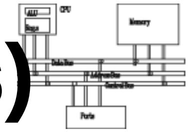
- Operators and precedence levels:

Operator	Name	Precedence Level
( )	parentheses	1
+, -	unary plus, minus	2
*, /	multiply, divide	3
MOD	modulus	3
+, -	add, subtract	4

- Examples:

Expression	Value
16 / 5	3
-(3 + 4) * (6 - 1)	-35
-3 + 4 * 6 - 1	20
25 mod 3	1

Real number constants (encoded reals)



- Fixed point v.s. floating point

1	8	23
S	E	M

$$\pm 1.bbbb \times 2^{(E-127)}$$

- Example **3F800000r=+1.0,37.75=42170000r**

- double

1	11	52
S	E	M

Real number constants (decimal reals)

- [*sign*]*integer*.[*integer*][*exponent*]

  sign → {+|-}

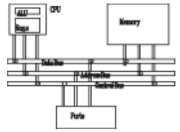
  exponent → E[{+|-}]integer
- Examples:

```
2.
+3.0
-44.2E+05
26.E5
```

Character and string constants

- Enclose character in single or double quotes
  - **`'A', "x"`**
  - ASCII character = 1 byte
- Enclose strings in single or double quotes
  - **`"ABC"`**
  - **`'xyz'`**
  - Each character occupies a single byte
- Embedded quotes:
  - **`‘Say "Goodnight," Gracie’`**
  - **`"This isn't a test"`**

Intrinsic data types (1 of 2)

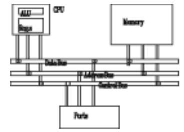


- **BYTE**, **SBYTE**
  - 8-bit unsigned integer; 8-bit signed integer
- **WORD**, **SWORD**
  - 16-bit unsigned & signed integer
- **DWORD**, **SDWORD**
  - 32-bit unsigned & signed integer
- **QWORD**
  - 64-bit integer
- **TBYTE**
  - 80-bit integer

Intrinsic data types (2 of 2)

- **REAL4**
  - 4-byte IEEE short real
- **REAL8**
  - 8-byte IEEE long real
- **REAL10**
  - 10-byte IEEE extended real

Data definition statement



- A data definition statement sets aside storage in memory for a variable.
- May optionally assign a name (label) to the data.
- Only size matters, other attributes such as signed are just reminders for programmers.
- Syntax:

  [*name*] *directive initializer* [,*initializer*] . . .

  At least one initializer is required, can be **?**
- All initializers become binary data in memory

Defining BYTE and SBYTE Data

Each of the following defines a single byte of storage:

```
value1 BYTE 'A‘    ; character constant
value2 BYTE 0      ; smallest unsigned byte
value3 BYTE 255    ; largest unsigned byte
value4 SBYTE -128  ; smallest signed byte
value5 SBYTE +127  ; largest signed byte
value6 BYTE ?      ; uninitialized byte
```

A variable name is a data label that implies an offset (an address).

Defining multiple bytes

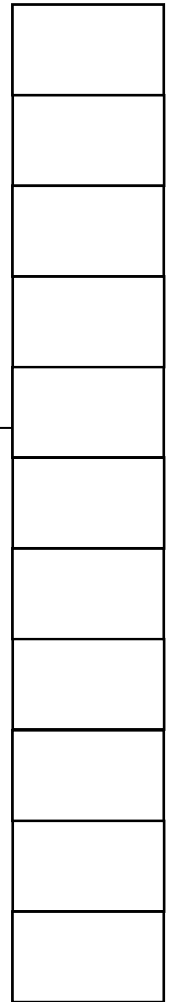
Examples that use multiple initializers:

```
list1 BYTE 10,20,30,40
list2 BYTE 10,20,30,40
      BYTE 50,60,70,80
      BYTE 81,82,83,84
list3 BYTE ?,32,41h,00100010b
list4 BYTE 0Ah,20h,'A',22h
```

Defining strings (1 of 2)

- A string is implemented as an array of characters
  - For convenience, it is usually enclosed in quotation marks
  - It usually has a null byte at the end
- Examples:

```
str1 BYTE "Enter your name",0
str2 BYTE 'Error: halting program',0
str3 BYTE 'A','E','I','O','U'
greeting1 BYTE "Welcome to the Encryption Demo program "
          BYTE "created by Kip Irvine.",0
greeting2 \
   BYTE "Welcome to the Encryption Demo program "
   BYTE "created by Kip Irvine.",0
```



Defining strings (2 of 2)

- End-of-line character sequence:
  - 0Dh = carriage return
  - 0Ah = line feed

```
str1 BYTE "Enter your name:    ",0Dh,0Ah
     BYTE "Enter your address: ",0

newLine BYTE 0Dh,0Ah,0
```

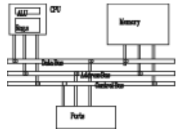
Idea: Define all strings used by your program in the same area of the data segment.

Using the DUP operator

- Use **DUP** to allocate (create space for) an array or string.
- Counter and argument must be constants or constant expressions

```
var1 BYTE 20 DUP(0) ; 20 bytes, all zero
var2 BYTE 20 DUP(?) ; 20 bytes,
                    ; uninitialized
var3 BYTE 4 DUP("STACK") ; 20 bytes:
                     ;"STACKSTACKSTACKSTACK"
var4 BYTE 10,3 DUP(0),20
```

Defining WORD and SWORD data



- Define storage for 16-bit integers
  - or double characters
  - single value or multiple values

```
word1 WORD   65535  ; largest unsigned
word2 SWORD –32768  ; smallest signed
word3 WORD   ?      ; uninitialized,
                    ; unsigned
word4 WORD "AB"     ; double characters
myList WORD 1,2,3,4,5  ; array of words
array WORD 5 DUP(?) ; uninitialized array
```

Defining DWORD and SDWORD data

Storage definitions for signed and unsigned 32-bit integers:

```
val1 DWORD 12345678h      ; unsigned
val2 SDWORD –2147483648  ; signed
val3 DWORD 20 DUP(?)      ; unsigned array
val4 SDWORD –3,–2,–1,0,1 ; signed array
```

Defining QWORD, TBYTE, Real Data

Storage definitions for quadwords, tenbyte values, and real numbers:

```
quad1 QWORD  1234567812345678h
val1  TBYTE  1000000000123456789Ah
rVal1 REAL4  -2.1
rVal2 REAL8  3.2E-260
rVal3 REAL10 4.6E+4096
ShortArray REAL4 20 DUP(0.0)
```

Little Endian order

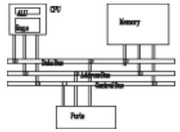
- All data types larger than a byte store their individual bytes in reverse order. The least significant byte occurs at the first (lowest) memory address.

- Example:

```
val1 DWORD 12345678h
```

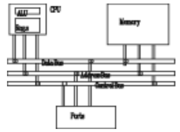
0000:	78
0001:	56
0002:	34
0003:	12

Adding variables to AddSub



```
TITLE Add and Subtract, (AddSub2.asm)
INCLUDE Irvine32.inc
.data
val1 DWORD 10000h
val2 DWORD 40000h
val3 DWORD 20000h
finalVal DWORD ?
.code
main PROC
   mov eax,val1            ; start with 10000h
   add eax,val2            ; add 40000h
   sub eax,val3            ; subtract 20000h
   mov finalVal,eax        ; store the result (30000h)
   call DumpRegs           ; display the registers
   exit
main ENDP
END main
```

Declaring unitialized data



- Use the **`.data?`** directive to declare an unintialized data segment:

  `.data?`

- Within the segment, declare variables with "?" initializers: (will not be assembled into .exe)

> Advantage: the program's EXE file size is reduced.

```
.data
smallArray DWORD 10 DUP(0)
.data?
bigArray   DWORD 5000 DUP(?)
```

Mixing code and data

```
.code
mov eax, ebx
.data
temp DWORD ?
.code
mov temp, eax
```

Symbolic constants

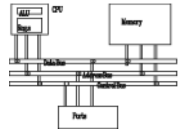
Equal-sign directive

- *name* = *expression*
  - expression is a 32-bit integer (expression or constant)
  - may be redefined
  - *name* is called a symbolic constant
- good programming style to use symbols
  - Easier to modify
  - Easier to understand, **ESC_key**

```
Array DWORD COUNT DUP(0)
COUNT=5
mov al, COUNT
COUNT=10
mov al, COUNT
```

```
COUNT = 500
.
mov al,COUNT
```

Calculating the size of a byte array



- current location counter: $
  - subtract address of list
  - difference is the number of bytes

```
list BYTE 10,20,30,40
ListSize = 4
```

```
list BYTE 10,20,30,40
ListSize = ($ - list)
```

```
list BYTE 10,20,30,40
Var2 BYTE 20 DUP(?)
ListSize = ($ - list)
```

```
myString BYTE “This is a long string.”
myString_len = ($ - myString)
```

Calculating the size of a word array

- current location counter: $
  - subtract address of list
  - difference is the number of bytes
  - divide by 2 (the size of a word)

```
list WORD 1000h,2000h,3000h,4000h
ListSize = ($ - list) / 2
```

```
list DWORD 1,2,3,4
ListSize = ($ - list) / 4
```

EQU directive

- name EQU expression

  name EQU symbol

  name EQU <text>
- Define a symbol as either an integer or text expression.
- Can be useful for non-integer constant
- Cannot be redefined

EQU directive

```
PI EQU <3.1416>
pressKey EQU <"Press any key to continue...",0>
.data
prompt BYTE pressKey
```

```
Matrix1 EQU 10*10
matrix1 EQU <10*10>
.data
M1 WORD matrix1        ; M1 WORD 100
M2 WORD matrix2        ; M2 WORD 10*10
```

Addressing

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

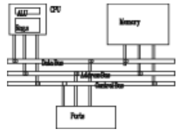
Direct memory operands

- A direct memory operand is a named reference to storage in memory
- The named reference (label) is automatically dereferenced by the assembler

```
.data
var1 BYTE 10h,
.code
mov al,var1              ; AL = 10h
mov al,[var1]            ; AL = 10h
```

alternate format; I prefer this one.

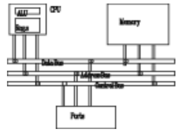
Direct-offset operands



A constant offset is added to a data label to produce an effective address (EA). The address is dereferenced to get the value inside its memory location. (no range checking)

```
.data
arrayB BYTE 10h,20h,30h,40h
.code
mov al,arrayB+1       ; AL = 20h
mov al,[arrayB+1]     ; alternative notation
mov al,arrayB+3       ; AL = 40h
```

Direct-offset operands (cont)



A constant offset is added to a data label to produce an effective address (EA). The address is dereferenced to get the value inside its memory location.

```
.data
arrayW  WORD 1000h,2000h,3000h
arrayD  DWORD 1,2,3,4
.code
mov ax,[arrayW+2]      ; AX = 2000h
mov ax,[arrayW+4]      ; AX = 3000h
mov eax,[arrayD+4]     ; EAX = 00000002h
```

```
; will the following assemble and run?
mov ax,[arrayW-2]         ; ??
mov eax,[arrayD+16]       ; ??
```

Your turn. . .

Write a program that rearranges the values of three doubleword values in the following array as: 3, 1, 2.

```
.data
arrayD DWORD 1,2,3
```

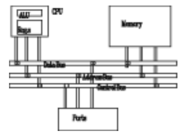
- Step1: copy the first value into EAX and exchange it with the value in the second position.

```
mov eax,arrayD
xchg eax,[arrayD+4]
```

- Step 2: Exchange EAX with the third array value and copy the value in EAX to the first array position.

```
xchg eax,[arrayD+8]
mov  arrayD,eax
```

Evaluate this . . .



- We want to write a program that adds the following three bytes:

```
.data
myBytes BYTE 80h,66h,0A5h
```

- What is your evaluation of the following code?

```
mov al,myBytes
add al,[myBytes+1]
add al,[myBytes+2]
```

- What is your evaluation of the following code?

```
mov ax,myBytes
add ax,[myBytes+1]
add ax,[myBytes+2]
```

Evaluate this . . . (cont)

```
.data
myBytes BYTE 80h,66h,0A5h
```

- How about the following code. Is anything missing?

```
movzx ax,myBytes
mov   bl,[myBytes+1]
add   ax,bx
mov   bl,[myBytes+2]
add   ax,bx                  ; AX = sum
```

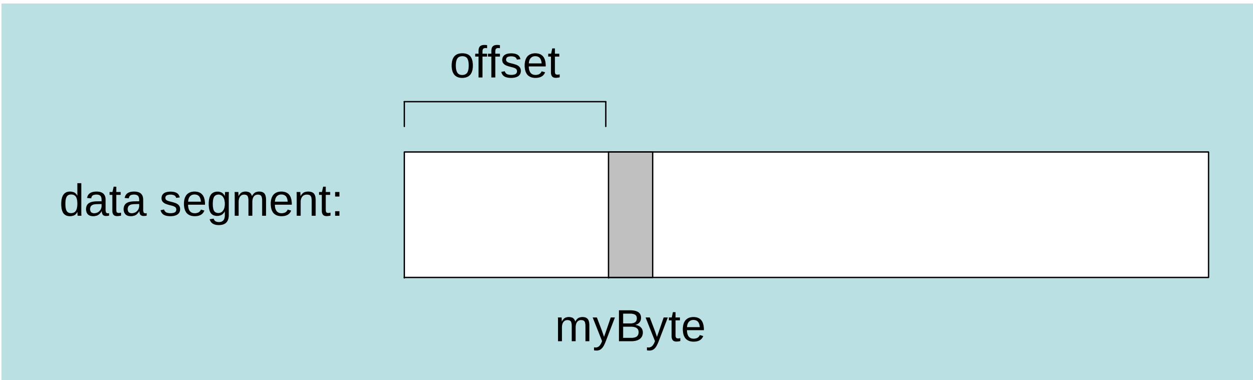
Yes: Move zero to BX before the MOVZX instruction.

Data-Related Operators and Directives

- OFFSET Operator
- PTR Operator
- TYPE Operator
- LENGTHOF Operator
- SIZEOF Operator
- LABEL Directive

OFFSET Operator

- OFFSET returns the distance in bytes, of a label from the beginning of its enclosing segment
  - Protected mode: 32 bits
  - Real mode: 16 bits



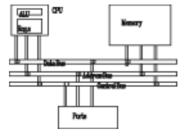
The Protected-mode programs we write only have a single segment (we use the flat memory model).

OFFSET Examples

Let's assume that **`bVal`** is located at 00404000h:

```
.data
bVal BYTE ?
wVal WORD ?
dVal DWORD ?
dVal2 DWORD ?

.code
mov esi,OFFSET bVal ; ESI = 00404000
mov esi,OFFSET wVal ; ESI = 00404001
mov esi,OFFSET dVal ; ESI = 00404003
mov esi,OFFSET dVal2; ESI = 00404007
```

Relating to C/C++

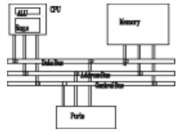


The value returned by **OFFSET** is a pointer. Compare the following code written for both C++ and assembly language:

```
; C++ version:
char array[1000];
char * p = &array;
```

```
.data
array BYTE 1000 DUP(?)
.code
mov esi,OFFSET array   ; ESI is p
```

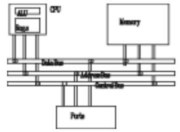
ALIGN Directive



- **`ALIGN`** *bound* aligns a variable on a byte, word, doubleword, or paragraph boundary for efficiency. (*bound* can be 1, 2, 4, or 16.)

```
bVal   BYTE ?    ; 00404000
ALIGN 2
wVal   WORD ?    ; 00404002
bVal2  BYTE ?    ; 00404004
ALIGN 4
dVal   DWORD ?   ; 00404008
dVal2  DWORD ?   ; 0040400C
```

PTR Operator



Overrides the default type of a label (variable).
Provides the flexibility to access part of a variable.

```
.data
myDouble DWORD 12345678h
.code
mov ax,myDouble                ; error – why?

mov ax,WORD PTR myDouble       ; loads 5678h

mov WORD PTR myDouble,4321h    ; saves 4321h
```

To understand how this works, we need to know about little endian ordering of data in memory.

Little Endian Order

- Little endian order refers to the way Intel stores integers in memory.
- Multi-byte integers are stored in reverse order, with the least significant byte stored at the lowest address
- For example, the doubleword 12345678h would be stored as:

byte	offset
78	0000
56	0001
34	0002
12	0003

When integers are loaded from memory into registers, the bytes are automatically re-reversed into their correct positions.

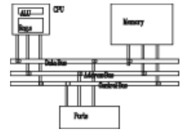
PTR Operator Examples

```
.data
myDouble DWORD 12345678h
```

doubleword	word	byte	offset	
12345678	5678	78	0000	myDouble
		56	0001	myDouble + 1
	1234	34	0002	myDouble + 2
		12	0003	myDouble + 3

```
mov al,BYTE PTR myDouble        ; AL = 78h
mov al,BYTE PTR [myDouble+1]    ; AL = 56h
mov al,BYTE PTR [myDouble+2]    ; AL = 34h
mov ax,WORD PTR [myDouble]      ; AX = 5678h
mov ax,WORD PTR [myDouble+2]    ; AX = 1234h
```

PTR Operator (cont)



PTR can also be used to combine elements of a smaller data type and move them into a larger operand. The CPU will automatically reverse the bytes.

```
.data
myBytes BYTE 12h,34h,56h,78h

.code
mov ax,WORD PTR [myBytes]       ; AX = 3412h
mov ax,WORD PTR [myBytes+2]     ; AX = 5634h
mov eax,DWORD PTR myBytes       ; EAX
                                ; =78563412h
```

Your turn . . .

Write down the value of each destination operand:

```
.data
varB BYTE 65h,31h,02h,05h
varW WORD 6543h,1202h
varD DWORD 12345678h

.code
mov ax,WORD PTR [varB+2] ; a. 0502h
mov bl,BYTE PTR varD     ; b. 78h
mov bl,BYTE PTR [varW+2] ; c. 02h
mov ax,WORD PTR [varD+2] ; d. 1234h
mov eax,DWORD PTR varW   ; e. 12026543h
```

TYPE Operator

The TYPE operator returns the size, in bytes, of a single element of a data declaration.

```
.data
var1 BYTE ?
var2 WORD ?
var3 DWORD ?
var4 QWORD ?

.code
mov eax,TYPE var1    ; 1
mov eax,TYPE var2    ; 2
mov eax,TYPE var3    ; 4
mov eax,TYPE var4    ; 8
```

LENGTHOF Operator

The LENGTHOF operator counts the number of elements in a single data declaration.

```
.data                                 LENGTHOF
byte1  BYTE 10,20,30                  ; 3
array1 WORD 30 DUP(?),0,0             ; 32
array2 WORD 5 DUP(3 DUP(?))           ; 15
array3 DWORD 1,2,3,4                  ; 4
digitStr BYTE "12345678",0            ; 9

.code
mov ecx,LENGTHOF array1               ; 32
```

SIZEOF Operator

The SIZEOF operator returns a value that is equivalent to multiplying LENGTHOF by TYPE.

```
.data                                SIZEOF
byte1  BYTE 10,20,30                 ; 3
array1 WORD 30 DUP(?),0,0            ; 64
array2 WORD 5 DUP(3 DUP(?))          ; 30
array3 DWORD 1,2,3,4                 ; 16
digitStr BYTE "12345678",0           ; 9

.code
mov ecx,SIZEOF array1                ; 64
```

Spanning Multiple Lines (1 of 2)

A data declaration spans multiple lines if each line (except the last) ends with a comma. The LENGTHOF and SIZEOF operators include all lines belonging to the declaration:

```
.data
array WORD 10,20,
   30,40,
   50,60

.code
mov eax,LENGTHOF array   ; 6
mov ebx,SIZEOF array     ; 12
```

Spanning Multiple Lines (2 of 2)

In the following example, array identifies only the first WORD declaration. Compare the values returned by LENGTHOF and SIZEOF here to those in the previous slide:

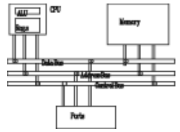
```
.data
arrayWORD 10,20
     WORD 30,40
     WORD 50,60

.code
mov eax,LENGTHOF array   ; 2
mov ebx,SIZEOF array     ; 4
```

LABEL Directive

- Assigns an alternate label name and type to an existing storage location
- LABEL does not allocate any storage of its own; it is just an alias.
- Removes the need for the PTR operator

```
.data
dwList   LABEL DWORD
wordList LABEL WORD
intList  BYTE  00h,10h,00h,20h
.code
mov eax,dwList      ; 20001000h
mov cx,wordList     ; 1000h
mov dl,intList      ; 00h
```

Indirect Operands (1 of 2)



An indirect operand holds the address of a variable, usually an array or string. It can be dereferenced (just like a pointer). **[reg]** uses reg as pointer to access memory

```
.data
val1 BYTE 10h,20h,30h
.code
mov esi,OFFSET val1
mov al,[esi]  ; dereference ESI (AL = 10h)

inc esi
mov al,[esi]  ; AL = 20h

inc esi
mov al,[esi]  ; AL = 30h
```

Indirect Operands (2 of 2)

Use PTR when the size of a memory operand is ambiguous.

```
.data
myCount WORD 0

.code
mov esi,OFFSET myCount
inc [esi]              ; error: ambiguous
inc WORD PTR [esi]     ; ok
```

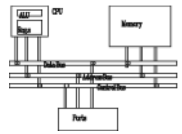
unable to determine the size from the context

Array Sum Example

Indirect operands are ideal for traversing an array. Note that the register in brackets must be incremented by a value that matches the array type.

```
.data
arrayW WORD 1000h,2000h,3000h
.code
  mov esi,OFFSET arrayW
  mov ax,[esi]
  add esi,2        ; or: add esi,TYPE arrayW
  add ax,[esi]
  add esi,2        ; increment ESI by 2
  add ax,[esi]     ; AX = sum of the array
```

Indexed Operands



An indexed operand adds a constant to a register to generate an effective address. There are two notational forms: **[*label* + *reg*]** **_label_[_reg_]**

```
.data
arrayW WORD 1000h,2000h,3000h
.code
   mov esi,0
   mov ax,[arrayW + esi]  ; AX = 1000h
   mov ax,arrayW[esi]  ; alternate format
   add esi,2
   add ax,[arrayW + esi]
   etc.
```

Index Scaling

You can scale an indirect or indexed operand to the offset of an array element. This is done by multiplying the index by the array's TYPE:

```
.data
arrayB BYTE  0,1,2,3,4,5
arrayW WORD  0,1,2,3,4,5
arrayD DWORD 0,1,2,3,4,5
.code
mov esi,4
mov al,arrayB[esi*TYPE arrayB]    ; 04
mov bx,arrayW[esi*TYPE arrayW]    ; 0004
mov edx,arrayD[esi*TYPE arrayD]  ; 00000004
```

Pointers

You can declare a pointer variable that contains the offset of another variable.

```
.data
arrayW WORD 1000h,2000h,3000h
ptrW DWORD arrayW
.code
   mov esi,ptrW
   mov ax,[esi]        ; AX = 1000h
```

Data Transfers Instructions

MOV instruction

- Move from source to destination. Syntax:

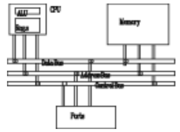
  MOV *destination, source*

- Source and destination have the same size
- No more than one memory operand permitted
- CS, EIP, and IP cannot be the destination
- No immediate to segment moves

MOV instruction

```
.data
count BYTE 100
wVal  WORD 2
.code
   mov bl,count
   mov ax,wVal
   mov count,al

   mov al,wVal          ; error
   mov ax,count         ; error
   mov eax,count        ; error
```

Your turn . . .



Explain why each of the following **MOV** statements are invalid:

```
.data
bVal  BYTE   100
bVal2 BYTE   ?
wVal  WORD   2
dVal  DWORD  5
.code
   mov ds,45              ; a.
   mov esi,wVal           ; b.
   mov eip,dVal           ; c.
   mov 25,bVal            ; d.
   mov bVal2,bVal         ; e.
```

Memory to memory

```
.data
var1 WORD ?
var2 WORD ?
.code
mov ax, var1
mov var2, ax
```

Copy smaller to larger

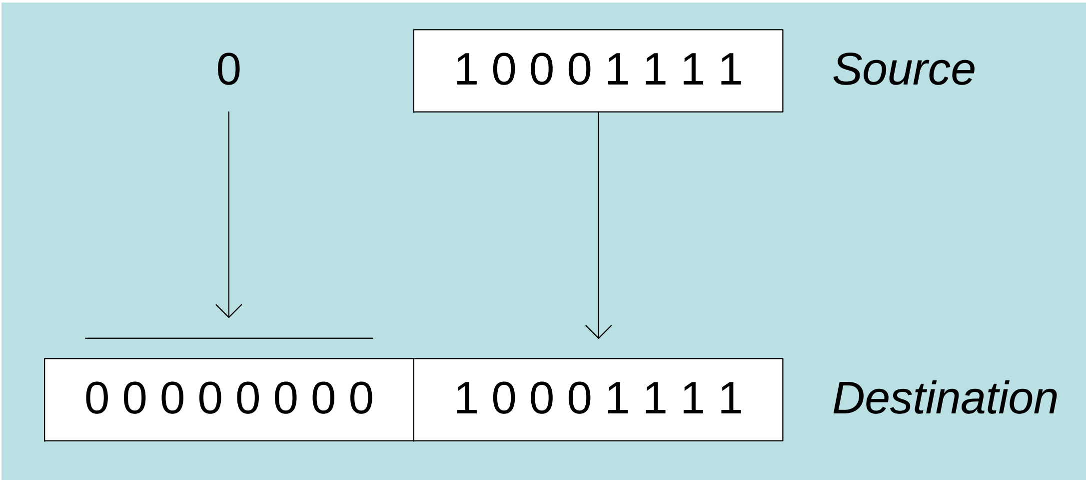
```
.data
count WORD 1
.code
mov ecx, 0
mov cx, count
```

```
.data
signedVal SWORD -16 ; FFF0h
.code
mov ecx, 0           ; mov ecx, 0FFFFFFFFh
mov cx, signedVal
```

`MOVZX` and **`MOVSX`** instructions take care of extension for both sign and unsigned integers.

Zero extension

When you copy a smaller value into a larger destination, the **MOVZX** instruction fills (extends) the upper half of the destination with zeros.



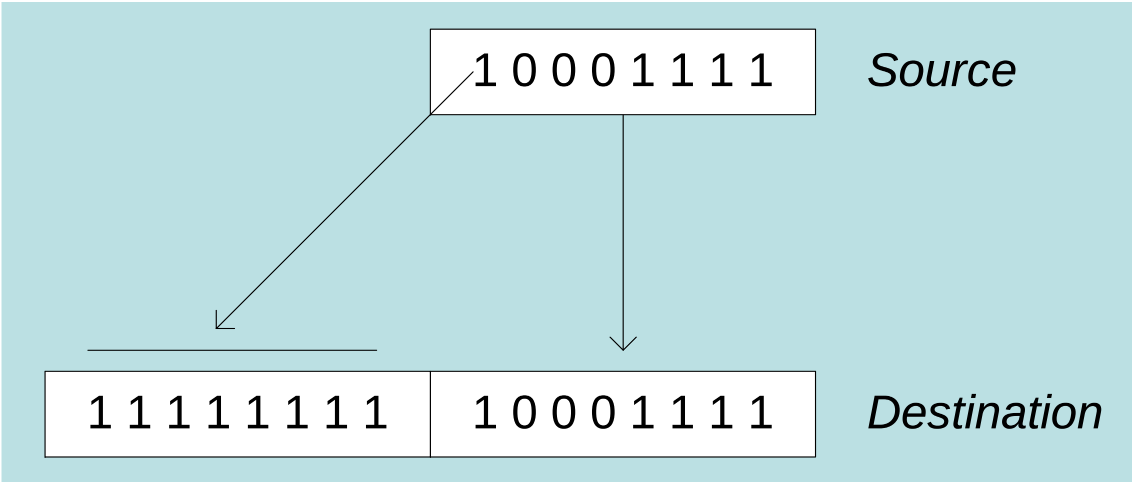

```
movzx r32,r/m8
movzx r32,r/m16
movzx r16,r/m8
```

```
mov bl,10001111b
movzx ax,bl              ; zero-extension
```

The destination must be a register.

Sign extension

The **MOVSX** instruction fills the upper half of the destination with a copy of the source operand's sign bit.




```
mov bl,10001111b
movsx ax,bl            ; sign extension
```

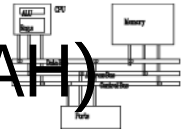
The destination must be a register.

MOVZX MOVSX

From a smaller location to a larger one

```
mov   bx,  0A69Bh
movzx eax, bx         ; EAX=0000A69Bh
movzx edx, bl         ; EDX=0000009Bh
movzx cx,  bl         ; EAX=009Bh

mov   bx,  0A69Bh
movsx eax, bx         ; EAX=FFFFA69Bh
movsx edx, bl         ; EDX=FFFFFF9Bh
movsx cx,  bl         ; EAX=FF9Bh
```

LAHF/SAHF (load/store status flag from/to AH)



```
.data
saveflags BYTE ?
.code
lahf
mov  saveflags, ah
...
mov  ah, saveflags
sahf
```

S,Z,A,P,C flags are copied.

XCHG Instruction

XCHG exchanges the values of two operands. At least one operand must be a register. No immediate operands are permitted.

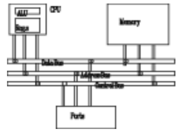
```
.data
var1 WORD 1000h
var2 WORD 2000h
.code
xchg ax,bx          ; exchange 16-bit regs
xchg ah,al          ; exchange 8-bit regs
xchg var1,bx        ; exchange mem, reg
xchg eax,ebx        ; exchange 32-bit regs

xchg var1,var2      ; error 2 memory operands
```

Exchange two memory locations

```
.data
var1 WORD 1000h
var2 WORD 2000h
.code
mov  ax, val1
xchg ax, val2
mov  val1, ax
```

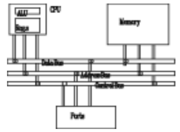
Arithmetic Instructions

Addition and Subtraction



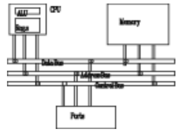
- **`INC`** and **`DEC`** Instructions
- **`ADD`** and **`SUB`** Instructions
- **`NEG`** Instruction
- Implementing Arithmetic Expressions
- Flags Affected by Arithmetic
  - Zero
  - Sign
  - Carry
  - Overflow

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

INC and DEC Examples



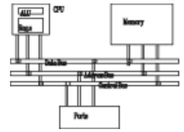
```
.data
myWord  WORD 1000h
myDword DWORD 10000000h
.code
   inc myWord              ; 1001h
   dec myWord              ; 1000h
   inc myDword             ; 10000001h

   mov ax,00FFh
   inc ax                  ; AX = 0100h
   mov ax,00FFh
   inc al                  ; AX = 0000h
```

Your turn. . .

Show the value of the destination operand after each of the following instructions executes:

```
.data
myByte BYTE 0FFh, 0
.code
   mov al,myByte         ; AL = FFh
   mov ah,[myByte+1] ; AH = 00h
   dec ah                    ; AH = FFh
   inc al                    ; AL = 00h
   dec ax                    ; AX = FEFF
```

ADD and SUB Instructions

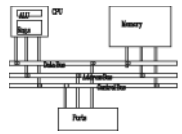


- **ADD** *destination, source*
  - Logic: *destination* ← *destination* + source
- **SUB** *destination, source*
  - Logic: *destination* ← *destination* – source
- Same operand rules as for the **MOV** instruction

ADD and SUB Examples

```
.data
var1 DWORD 10000h
var2 DWORD 20000h
.code                    ; ---EAX---
   mov eax,var1          ; 00010000h
   add eax,var2          ; 00030000h
   add ax,0FFFFh         ; 0003FFFFh
   add eax,1             ; 00040000h
   sub ax,1              ; 0004FFFFh
```

NEG (negate) Instruction



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

```
.data
valB BYTE -1
valW WORD +32767
.code
   mov al,valB         ; AL = -1
   neg al              ; AL = +1
   neg valW            ; valW = -32767
```

Suppose AX contains –32,768 and we apply NEG to it. Will the result be valid?

Implementing Arithmetic Expressions

HLL compilers translate mathematical expressions into assembly language. You can do it also. For example:

Rval = -Xval + (Yval – Zval)

```
Rval DWORD ?
Xval DWORD 26
Yval DWORD 30
Zval DWORD 40
.code
   mov eax,Xval
   neg eax              ; EAX = -26
   mov ebx,Yval
   sub ebx,Zval         ; EBX = -10
   add eax,ebx
   mov Rval,eax         ; -36
```

Your turn. . .

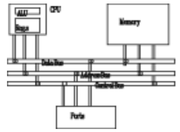
Translate the following expression into assembly language. Do not permit Xval, Yval, or Zval to be modified:

Rval = Xval - (-Yval + Zval)

Assume that all values are signed doublewords.

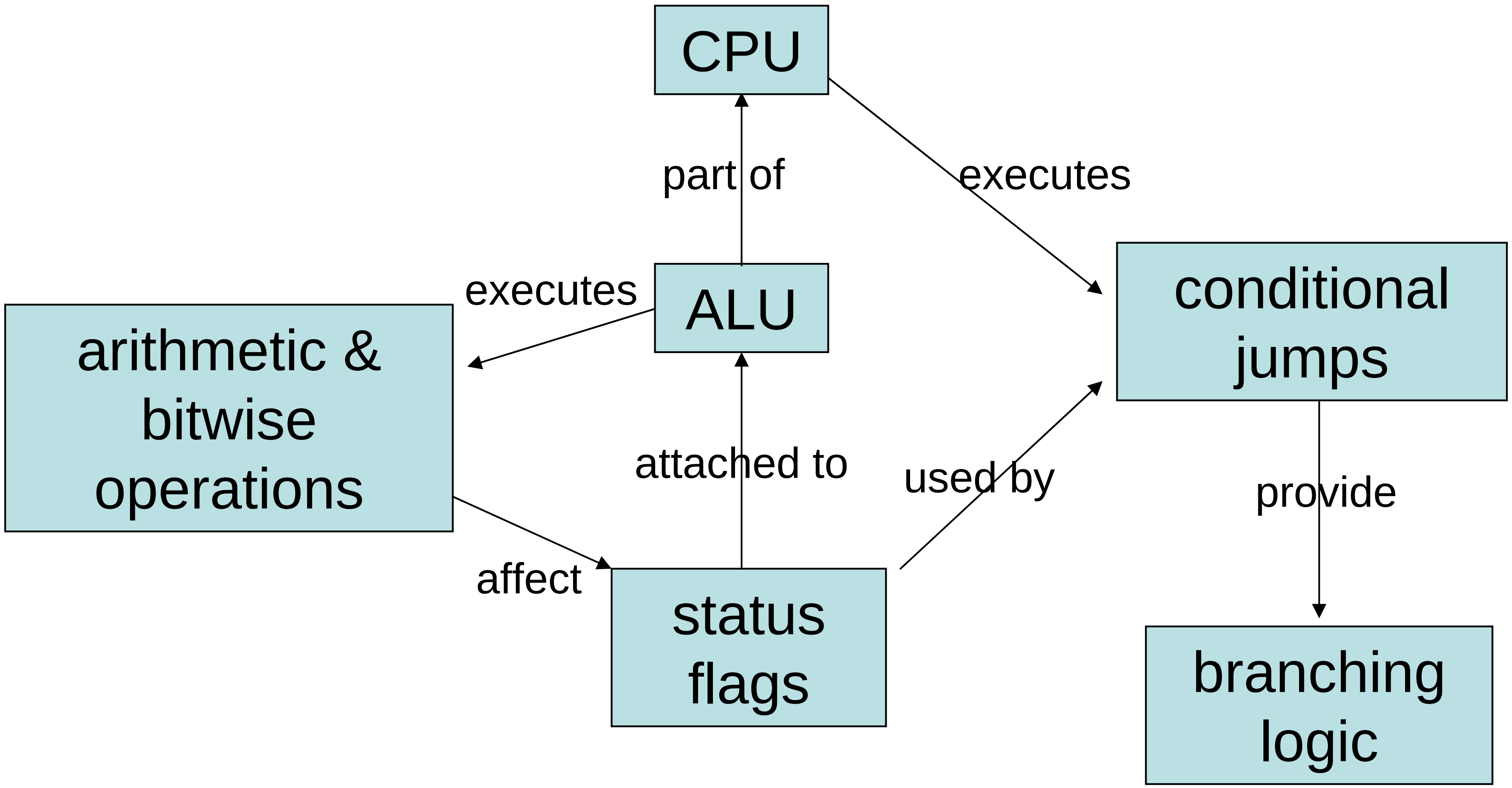
```
mov ebx,Yval
neg ebx
add ebx,Zval
mov eax,Xval
sub ebx
mov Rval,eax
```

Flags Affected by Arithmetic

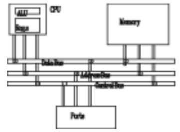


- The ALU has a number of status flags that reflect the outcome of arithmetic (and bitwise) operations
  - based on the contents of the destination operand
- Essential flags:
  - Zero flag – destination equals zero
  - Sign flag – destination is negative
  - Carry flag – unsigned value out of range
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

Concept Map


CPU
part of
executes
ALU
executes
arithmetic & bitwise operations
conditional jumps
attached to
used by
provide
affect
status flags
branching logic

Zero Flag (ZF)



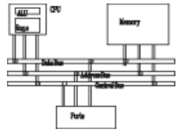
Whenever the destination operand equals Zero, the Zero flag is set.

```
mov cx,1
sub cx,1              ; CX = 0, ZF = 1
mov ax,0FFFFh
inc ax                ; AX = 0, ZF = 1
inc ax                ; AX = 1, ZF = 0
```

A flag is set when it equals 1.

A flag is clear when it equals 0.

Sign Flag (SF)



The Sign flag is set when the destination operand is negative. The flag is clear when the destination is positive.

```
mov cx,0
sub cx,1                    ; CX = -1, SF = 1
add cx,2                    ; CX = 1, SF = 0
```

The sign flag is a copy of the destination's highest bit:

```
mov al,0
sub al,1                    ; AL=11111111b, SF=1
add al,2                    ; AL=00000001b, SF=0
```

Carry Flag (CF)

The Carry flag is set when the result of an operation generates an unsigned value that is out of range (too big or too small for the destination operand).

```
mov al,0FFh
add al,1                ; CF = 1, AL = 00

; Try to go below zero:
mov al,0
sub al,1                ; CF = 1, AL = FF
```

In the second example, we tried to generate a negative value. Unsigned values cannot be negative, so the Carry flag signaled an error condition.

Carry Flag (CF)

- Addition and CF: copy carry out of MSB to CF
- Subtraction and CF: copy inverted carry out of MSB to CF
- **`INC`/`DEC`** do not affect CF
- Applying **`NEG`** to a nonzero operand sets CF

Your turn . . .

For each of the following marked entries, show the values of the destination operand and the Sign, Zero, and Carry flags:

```
mov add ax,00FFh
add ax,1        ; AX= 0100h  SF= 0 ZF= 0 CF= 0
sub ax,1        ; AX= 00FFh  SF= 0 ZF= 0 CF= 0
add al,1        ; AL= 00h    SF= 0 ZF= 1 CF= 1
mov bh,6Ch
add bh,95h      ; BH= 01h    SF= 0 ZF= 0 CF= 1

mov al,2
sub al,3        ; AL= FFh    SF= 1 ZF= 0 CF= 1
```

Overflow Flag (OF)

The Overflow flag is set when the signed result of an operation is invalid or out of range.

```
; Example 1
mov al,+127
add al,1              ; OF = 1,   AL = ??

; Example 2
mov al,7Fh            ; OF = 1,   AL = 80h
add al,1
```

The two examples are identical at the binary level because 7Fh equals +127. To determine the value of the destination operand, it is often easier to calculate in hexadecimal.

A Rule of Thumb

- When adding two integers, remember that the Overflow flag is only set when . . .
  - Two positive operands are added and their sum is negative
  - Two negative operands are added and their sum is positive

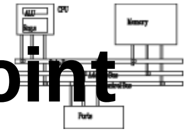
```
What will be the values of OF flag?
   mov al,80h
   add al,92h          ; OF =

   mov al,-2
   add al,+127         ; OF =
```

Your turn . . .

What will be the values of the Carry and Overflow flags after each operation?

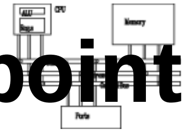
```
mov al,-128
neg al              ; CF = 0   OF =  1

mov ax,8000h
add ax,2            ; CF = 0   OF =  0

mov ax,0
sub ax,2            ; CF = 1   OF =  0

mov al,-5
sub al,+125         ; CF = 0   OF =  1
```

Signed/Unsigned Integers: Hardware Viewpoint



- All CPU instructions operate exactly the same on signed and unsigned integers
- The CPU cannot distinguish between signed and unsigned integers
- YOU, the programmer, are solely responsible for using the correct data type with each instruction

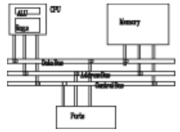
Overflow/Carry Flags: Hardware Viewpoint



- How the **ADD** instruction modifies OF and CF:
  - CF = (carry out of the MSB)
  - OF = (carry out of the MSB) XOR (carry into the MSB)

- How the **SUB** instruction modifies OF and CF:
  - NEG the source and ADD it to the destination
  - CF = INVERT (carry out of the MSB)
  - OF = (carry out of the MSB) XOR (carry into the MSB)

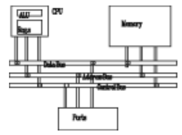
Auxiliary Carry (AC) flag



- AC indicates a carry or borrow of bit 3 in the destination operand.
- It is primarily used in binary coded decimal (BCD) arithmetic.

```
mov al, oFh
add al, 1          ; AC = 1
```

Parity (PF) flag



- PF is set when LSB of the destination has an even number of 1 bits.

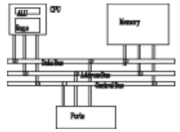
```
mov al, 10001100b
add al, 00000010b; AL=10001110, PF=1
sub al, 10000000b; AL=00001110, PF=0
```

Jump and Loop

`JMP` and **`LOOP`** Instructions

- Transfer of control or branch instructions
  - unconditional
  - conditional
- **`JMP`** Instruction
- **`LOOP`** Instruction
- **`LOOP`** Example
- Summing an Integer Array
- Copying a String

JMP Instruction



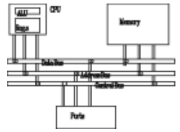
- **`JMP`** is an unconditional jump to a label that is usually within the same procedure.
- Syntax: **`JMP`** *target*
- Logic: EIP $\leftarrow$ *target*
- Example:

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

LOOP Instruction

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

LOOP Example



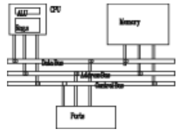
The following loop calculates the sum of the integers 5 + 4 + 3 +2 + 1:

```
offset     machine code     source code
00000000  66 B8 0000          mov   ax,0
00000004  B9 00000005         mov   ecx,5

00000009  66 03 C1         L1:add   ax,cx
0000000C  E2 FB               loop L1
0000000E
```

When **LOOP** is assembled, the current location = 0000000E. Looking at the **LOOP** machine code, we see that –5 (FBh) is added to the current location, causing a jump to location 00000009:

00000009 ← 0000000E + FB

Your turn . . .



If the relative offset is encoded in a single byte,

(a) what is the largest possible backward jump?

(b) what is the largest possible forward jump?

(a) –128

(b) +127

Average sizes of machine instructions are about 3 bytes, so a loop might contain, on average, a maximum of 42 instructions!

Your turn . . .

What will be the final value of AX?

10

```
    mov ax,6
    mov ecx,4
L1:
    inc ax
    loop L1
```

How many times will the loop execute?

4,294,967,296

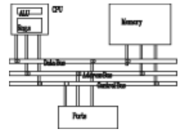
```
    mov ecx,0
X2:
    inc ax
    loop X2
```

Nested Loop

If you need to code a loop within a loop, you must save the outer loop counter's ECX value. In the following example, the outer loop executes 100 times, and the inner loop 20 times.

```
.data
count DWORD ?
.code
   mov ecx,100        ; set outer loop count
L1:
   mov count,ecx      ; save outer loop count
   mov ecx,20         ; set inner loop count
L2:...
   loop L2            ; repeat the inner loop
   mov ecx,count      ; restore outer loop count
   loop L1            ; repeat the outer loop
```

Summing an Integer Array



The following code calculates the sum of an array of 16-bit integers.

```
.data
intarray WORD 100h,200h,300h,400h
.code
   mov edi,OFFSET intarray     ; address
   mov ecx,LENGTHOF intarray   ; loop counter
   mov ax,0                    ; zero the sum
L1:
   add ax,[edi]                ; add an integer
   add edi,TYPE intarray       ; point to next
   loop L1                ; repeat until ECX = 0
```

Copying a String

good use of SIZEOF

The following code copies a string from source to target.

```
.data
source  BYTE  "This is the source string",0
target  BYTE  SIZEOF source DUP(0),0

.code
   mov  esi,0                ; index register
   mov  ecx,SIZEOF source    ; loop counter
L1:
   mov  al,source[esi]       ; get char from source
   mov  target[esi],al       ; store in the target
   inc  esi                  ; move to next char
   loop L1                   ; repeat for entire string
```

Conditional Processing

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.
- Less important:
  – 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

NOT instruction

- Performs a bitwise Boolean NOT operation on a single destination operand
- Syntax: (no flag affected)

  NOT ***destination***

- Example:

```
mov al, 11110000b
not al
```

```
NOT   0 0 1 1 1 0 1 1
      ---------------
      1 1 0 0 0 1 0 0 ——— inverted
```

NOT

X	¬X
F	T
T	F

AND instruction

- Performs a bitwise Boolean AND operation between each pair of matching bits in two operands
- Syntax: (O=0,C=0,SZP)

  AND *destination, source*

- Example:

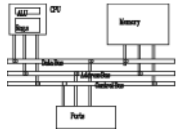
```
mov al, 00111011b
and al, 00001111b
```

```
              0 0 1 1 1 0 1 1
        AND   0 0 0 0 1 1 1 1
              ---------------
cleared ----  0 0 0 0 1 0 1 1  ---- unchanged
                bit extraction
```

AND

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

OR instruction



- Performs a bitwise Boolean OR operation between each pair of matching bits in two operands
- Syntax: (O=0,C=0,SZP)

  `OR` ***`destination, source`***

- Example:

```
mov dl, 00111011b
or  dl, 00001111b
```

```
              0 0 1 1 1 0 1 1
          OR  0 0 0 0 1 1 1 1
              ---------------
unchanged --- 0 0 1 1 1 1 1 1 --- set
```

OR

x	y	x ∨ y
0	0	0
0	1	1
1	0	1
1	1	1

XOR instruction

- Performs a bitwise Boolean exclusive-OR operation between each pair of matching bits in two operands
- Syntax: (O=0,C=0,SZP)

  XOR *destination, source*

- Example:

```
mov dl, 00111011b
xor dl, 00001111b
```

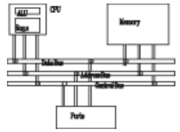
```
          0 0 1 1 1 0 1 1
    XOR   0 0 0 0 1 1 1 1
          ---------------
unchanged 0 0 1 1|0 1 0 0 inverted
```

XOR

x	y	x ⊕ y
0	0	0
0	1	1
1	0	1
1	1	0

XOR is a useful way to invert the bits in an operand and data encryption

Applications (1 of 4)



- Task: Convert the character in AL to upper case.
- Solution: Use the AND instruction to clear bit 5.

```
mov al,'a‘          ; AL = 01100001b
and al,11011111b    ; AL = 01000001b
```

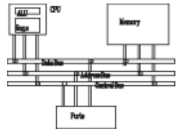
Applications (2 of 4)

- Task: Convert a binary decimal byte into its equivalent ASCII decimal digit.
- Solution: Use the OR instruction to set bits 4 and 5.

```
mov al,6              ; AL = 00000110b
or  al,00110000b      ; AL = 00110110b
```

The ASCII digit '6' = 00110110b

Applications (3 of 4)



- Task: Jump to a label if an integer is even.
- Solution: AND the lowest bit with a 1. If the result is Zero, the number was even.

```
mov ax,wordVal
and ax,1            ; low bit set?
jz  EvenValue       ; jump if Zero flag set
```

Applications (4 of 4)

- Task: Jump to a label if the value in AL is not zero.
- Solution: OR the byte with itself, then use the JNZ (jump if not zero) instruction.

```
or  al,al
jnz IsNotZero     ; jump if not zero
```

ORing any number with itself does not change its value.

TEST instruction

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

```
test al,00000011b
jnz  ValueFound
```

- Example: jump to a label if neither bit 0 nor bit 1 in AL is set.

```
test al,00000011b
jz   ValueNotFound
```

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: (OSZCAP)

  CMP *destination, source*

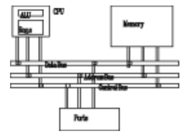
- Example: destination == source

```
mov al,5
cmp al,5              ; Zero flag set
```

- Example: destination < source

```
mov al,4
cmp al,5              ; Carry flag set
```

CMP instruction (2 of 3)



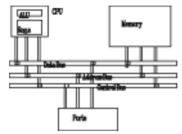
- Example: destination > source

```
mov al,6
cmp al,5              ; ZF = 0, CF = 0
```

(both the Zero and Carry flags are clear)

The comparisons shown so far were unsigned.

CMP instruction (3 of 3)



The comparisons shown here are performed with signed integers.

- Example: destination > source

```
mov al,5
cmp al,-2        ; Sign flag == Overflow flag
```

- Example: destination < source

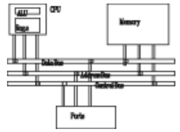
```
mov al,-1
cmp al,5         ; Sign flag != Overflow flag
```

Conditions

unsigned	ZF	CF
destination<source	0	1
destination>source	0	0
destination=source	1	0

signed	flags
destination<source	SF != OF
destination>source	SF == OF
destination=source	ZF=1

Setting and clearing individual flags



```
and al, 0           ; set Zero
or  al, 1           ; clear Zero
or  al, 80h         ; set Sign
and al, 7Fh         ; clear Sign
stc                 ; set Carry
clc                 ; clear Carry

mov al, 7Fh
inc al              ; set Overflow

or eax, 0           ; clear Overflow
```

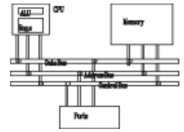
Conditional jumps

Conditional structures

- There are no high-level logic structures such as if-then-else, in the IA-32 instruction set. But, you can use combinations of comparisons and jumps to implement any logic structure.
- First, an operation such as **CMP**, **AND** or **SUB** is executed to modified the CPU flags. Second, a conditional jump instruction tests the flags and changes the execution flow accordingly.

```
      CMP AL, 0
      JZ  L1
      :
L1:
```

J*cond* instruction



- A conditional jump instruction branches to a label when specific register or flag conditions are met

  `Jcond` ***`destination`***

- Four groups: (some are the same)

1. based on specific flag values
2. based on equality between operands
3. based on comparisons of unsigned operands
4. based on comparisons of signed operands

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

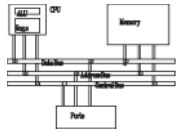
Mnemonic	Description
JE	Jump if equal (*leftOp* = *rightOp*)
JNE	Jump if not equal (*leftOp* ≠ *rightOp*)
JCXZ	Jump if CX = 0
JECXZ	Jump if ECX = 0

Jumps based on unsigned comparisons

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

＞ ≧ ＜ ≦

Jumps based on signed comparisons



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

Examples

- Compare unsigned AX to BX, and copy the larger of the two into a variable named Large

```
    mov Large,bx
    cmp ax,bx
    jna Next
    mov Large,ax
Next:
```

- Compare signed AX to BX, and copy the smaller of the two into a variable named Small

```
    mov Small,ax
    cmp bx,ax
    jnl Next
    mov Small,bx
Next:
```

Examples

- Find the first even number in an array of unsigned integers

```
.date
intArray DWORD 7,9,3,4,6,1
.code
...
        mov  ebx, OFFSET intArray
        mov  ecx, LENGTHOF intArray
L1:     test DWORD PTR [ebx], 1
        jz   found
        add  ebx, 4
        loop L1
...
```

BT (Bit Test) instruction

- Copies bit *n* from an operand into the Carry flag
- Syntax: **BT *bitBase, n***
  - bitBase may be *r/m16* or *r/m32*
  - n may be *r16, r32*, or *imm8*
- Example: jump to label L1 if bit 9 is set in the AX register:

```
bt AX,9             ; CF = bit 9
jc L1               ; jump if Carry
```

- **BTC *bitBase, n:*** bit test and complement
- **BTR *bitBase, n:*** bit test and reset (clear)
- **BTS *bitBase, n:*** bit test and set

Conditional loops

LOOPZ and LOOPE

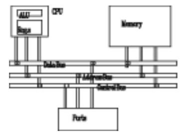
- Syntax:

  LOOPE ***destination***

  LOOPZ ***destination***

- Logic:
  - ECX $\leftarrow$ ECX – 1
  - if ECX != 0 and ZF=1, jump to *destination*
- The destination label must be between -128 and +127 bytes from the location of the following instruction
- Useful when scanning an array for the first element that meets some condition.

LOOPNZ and LOOPNE



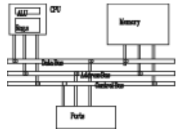
- Syntax:

  LOOPNZ *destination*

  LOOPNE *destination*

- Logic:
  - ECX ← ECX – 1;
  - if ECX != 0 and ZF=0, jump to *destination*

LOOPNZ example



The following code finds the first positive value in an array:

```
.data
array SWORD -3,-6,-1,-10,10,30,40,4
sentinel SWORD 0
.code
   mov esi,OFFSET array
   mov ecx,LENGTHOF array
next:
   test WORD PTR [esi],8000h   ; test sign bit
   pushfd                      ; push flags on stack
   add esi,TYPE array
   popfd                       ; pop flags from stack
   loopnz next                 ; continue loop
   jnz quit                    ; none found
   sub esi,TYPE array          ; ESI points to value
quit:
```

Your turn

Locate the first nonzero value in the array. If none is found, let ESI point to the sentinel value:

```
.data
array SWORD 50 DUP(?)
sentinel SWORD 0FFFFh
.code
    mov esi,OFFSET array
    mov ecx,LENGTHOF array
L1: cmp WORD PTR [esi],0    ; check for zero




quit:
```

Solution

```
.data
array SWORD 50 DUP(?)
sentinel SWORD 0FFFFh
.code
   mov esi,OFFSET array
   mov ecx,LENGTHOF array
L1:cmp WORD PTR [esi],0  ; check for zero
   pushfd                ; push flags on stack
   add esi,TYPE array
   Popfd                  ; pop flags from stack
   loope next            ; continue loop
   jz quit               ; none found
   sub esi,TYPE array    ; ESI points to value
quit:
```

Conditional structures

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 eax,op1
    cmp eax,op2
    jne L1
    mov X,1
    jmp L2
L1: mov X,2
L2:
```

Example

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

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

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

Example

Implement the following pseudocode in assembly language. All values are 32-bit signed integers:

```
if( var1 <= var2 )
  var3 = 10;
else
{
  var3 = 6;
  var4 = 7;
}
```

```
    mov eax,var1
    cmp eax,var2
    jle L1
    mov var3,6
    mov var4,7
    jmp L2
L1: mov var3,10
L2:
```

Compound expression with AND

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

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

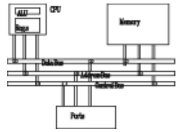
Compound expression with AND

```
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 X,1             ; set X to 1
next:
```

Compound expression with AND



```
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 X,1             ; 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 eax,5
        mov edx,6
next:
```

(There are multiple correct solutions to this problem.)

Compound Expression with OR

- In the following example, if the first expression is true, the second expression is skipped:

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

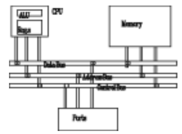
Compound Expression with OR

```
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 X,1          ; set X to 1
next:
```

WHILE Loops

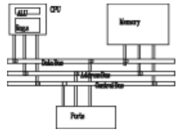


A WHILE loop is really an IF statement followed by the body of the loop, followed by an unconditional jump to the top of the loop. Consider the following example:

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

```
_while:
    cmp eax,ebx      ; check loop condition
    jae _endwhile    ; false? exit loop
    inc eax          ; body of loop
    jmp _while       ; repeat the loop
_endwhile:
```

Your turn . . .



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

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

```
_while:
    cmp ebx,val1    ; check loop condition
    ja  _endwhile   ; false? exit loop
    add ebx,5       ; body of loop
    dec val1
    jmp while       ; repeat the loop
_endwhile:
```

Example: IF statement nested in a loop

```
while(eax < ebx)
{
  eax++;
  if (ebx==ecx)
    X=2;
  else
    X=3;
}
```

```
_while:   cmp  eax, ebx
          jae  _endwhile
          inc  eax
          cmp  ebx, ecx
          jne  _else
          mov  X, 2
          jmp  _while
_else:    mov  X, 3
          jmp  _while
_endwhile:
```

Table-driven selection

- Table-driven selection uses a table lookup to replace a multiway selection structure (switch-case statements in C)
- Create a table containing lookup values and the offsets of labels or procedures
- Use a loop to search the table
- Suited to a large number of comparisons

Table-driven selection

Step 1: create a table containing lookup values and procedure offsets:

```
.data
CaseTable BYTE 'A'            ; lookup value
   DWORD Process_A            ; address of procedure
   EntrySize = ($ - CaseTable)
   BYTE 'B'
   DWORD Process_B
   BYTE 'C'
   DWORD Process_C
   BYTE 'D'
   DWORD Process_D

NumberOfEntries = ($ - CaseTable) / EntrySize
```

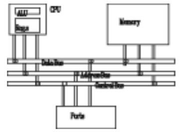
Table-driven selection

Step 2: Use a loop to search the table. When a match is found, we call the procedure offset stored in the current table entry:

```
   mov ebx,OFFSET CaseTable ; point EBX to the table
   mov ecx,NumberOfEntries ; loop counter

L1:cmp al,[ebx]             ; match found?
   jne L2                   ; no: continue
   call NEAR PTR [ebx + 1]  ; yes: call the procedure
   jmp L3                   ; and exit the loop
L2:add ebx,EntrySize        ; point to next entry
   loop L1                  ; repeat until ECX = 0

L3:
```

required for procedure pointers

Assignment #4 CRC32 checksum



```
unsigned int crc32(const char* data,
                   size_t length)
{
  // standard polynomial in CRC32
  const unsigned int POLY = 0xEDB88320;
  // standard initial value in CRC32
  unsigned int reminder = 0xFFFFFFFF;
  for(size_t i = 0; i < length; i++){
    // must be zero extended
    reminder ^= (unsigned char)data[i];
    for(size_t bit = 0; bit < 8; bit++)
      if(reminder & 0x01)
        reminder = (reminder >> 1) ^ POLY;
      else
        reminder >>= 1;
  }
  return reminder ^ 0xFFFFFFFF;
}
```