Intel x86 Instruction Set Architecture

Computer Organization and Assembly Languages
Yung-Yu Chuang
2008/12/15

with slides by Kip Irvine

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

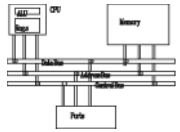
```
.data
count BYTE 100
wVal  WORD 2
.code
   mov bl,count
   mov ax,wVal
   mov count,al

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

Exercise . . .

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

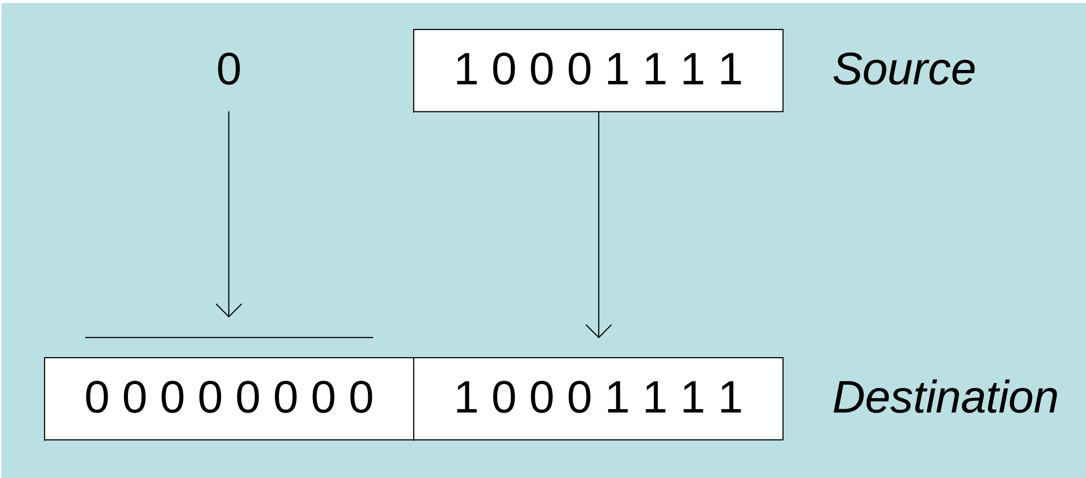
```
.data
count WORD 1
.code
mov ecx, 0
mov cx, count
```

```
.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.



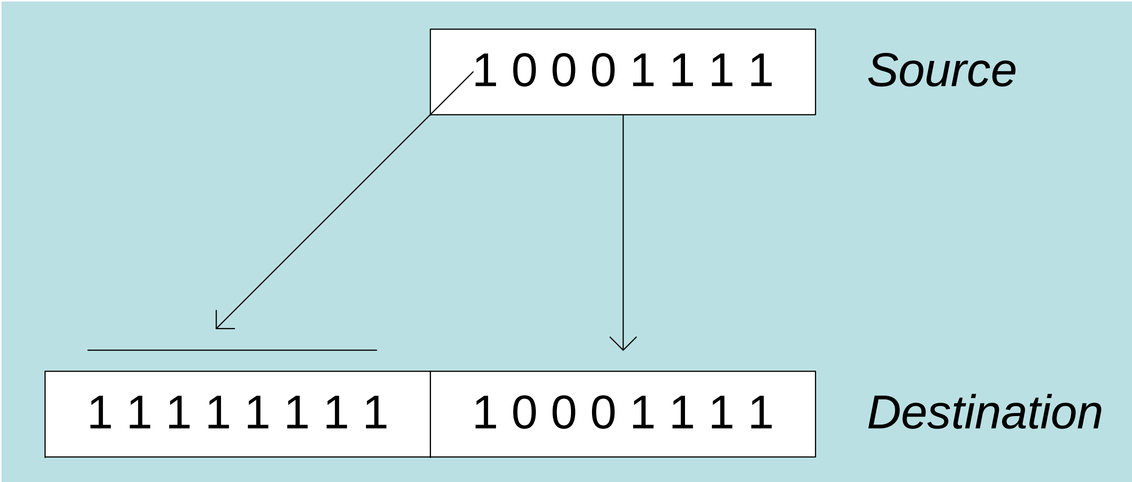

```
movzx r32,r/m8
movzx r32,r/m16
movzx r16,r/m8
```

```
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
```

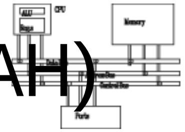
The destination must be a register.

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
```

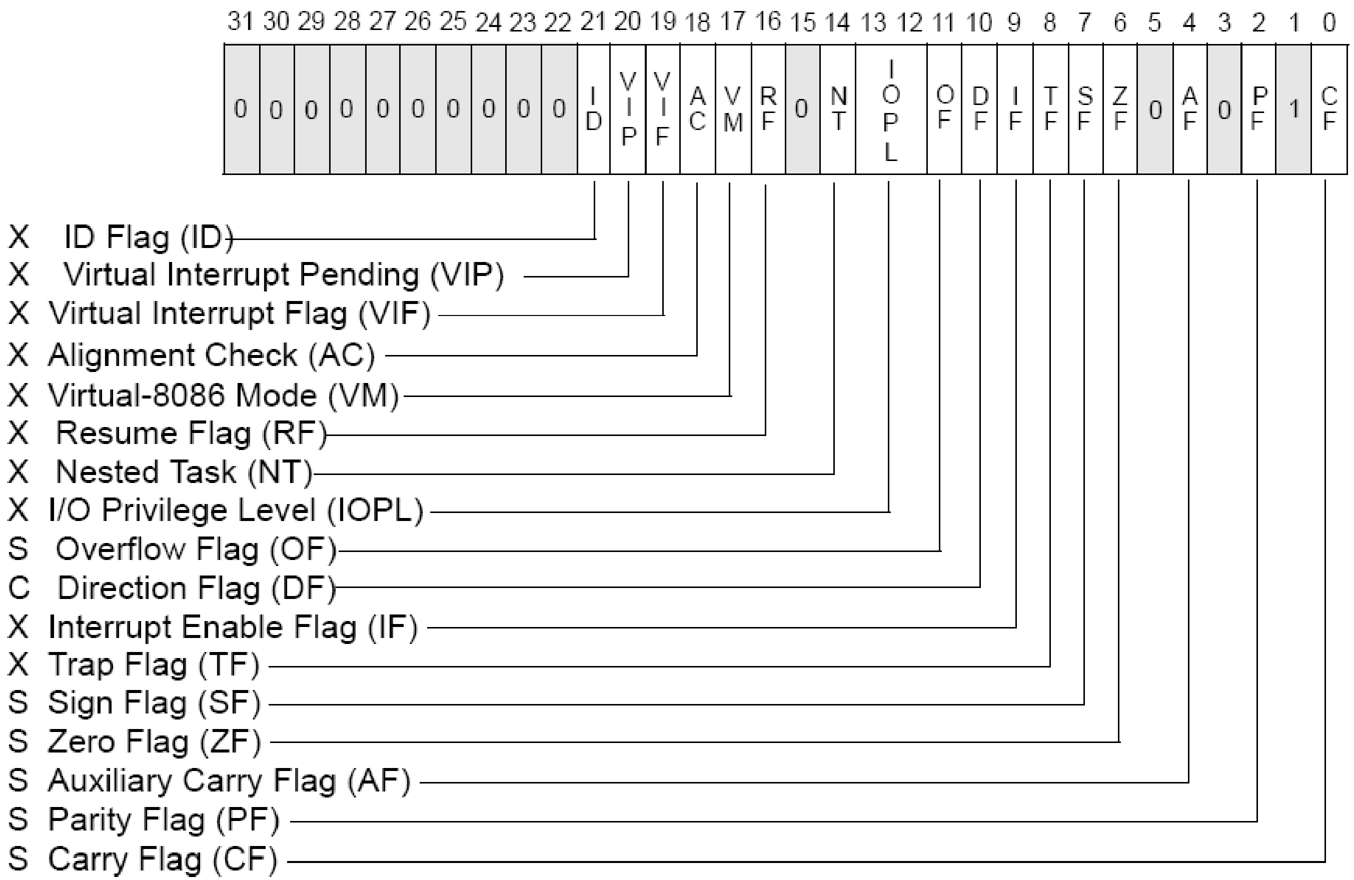
LAHF/SAHF (load/store status flag from/to AH)



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

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

EFLAGS



S Indicates a Status Flag
C Indicates a Control Flag
X Indicates a System Flag

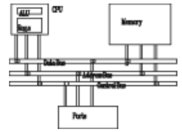
Reserved bit positions. DO NOT USE.
Always set to values previously read.

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
```

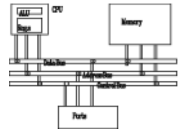
Exchange two memory locations



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

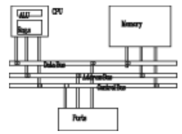
Arithmetic Instructions

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* ← *destination* + 1
- **DEC** *destination*
  - Logic: *destination* ← *destination* – 1

INC and DEC Examples

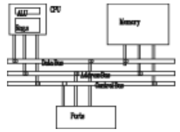
```
.data
myWord  WORD 1000h
myDword DWORD 10000000h
.code
   inc myWord              ; 1001h
   dec myWord              ; 1000h
   inc myDword             ; 10000001h

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

Exercise...

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

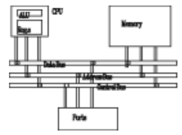


- **ADD** *destination, source*
  - Logic: *destination* $\leftarrow$ *destination* + source
- **SUB** *destination, source*
  - Logic: *destination* $\leftarrow$ *destination* – source
- Same operand rules as for the **MOV** instruction

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
```

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
```

Exercise...

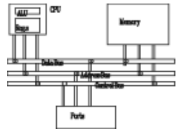
Translate the following expression into assembly language. Do not permit Xval, Yval, or Zval to be modified:

Rval = Xval - (-Yval + Zval)

Assume that all values are signed doublewords.

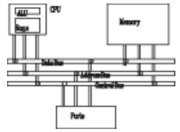
```
mov ebx,Yval
neg ebx
add ebx,Zval
mov eax,Xval
sub eax,ebx
mov Rval,eax
```

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.

Zero Flag (ZF)



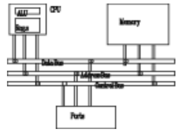
Whenever the destination operand equals Zero, the Zero flag is set.

```
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)



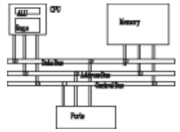
The Sign flag is set when the destination operand is negative. The flag is clear when the destination is positive.

```
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:

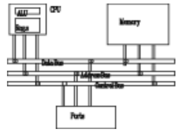
```
mov al,0
sub al,1                    ; AL=11111111b, SF=1
add al,2                    ; AL=00000001b, SF=0
```

Carry Flag (CF)



- 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

Exercise . . .



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

```
mov 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
```

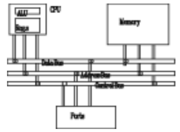
Overflow Flag (OF)

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.

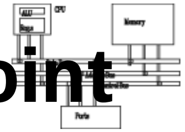
A Rule of Thumb



- 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

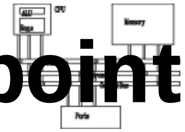
```
What will be the values of OF flag?
   mov al,80h
   add al,92h        ; OF =

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

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

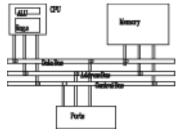
Overflow/Carry Flags: Hardware Viewpoint



- 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)

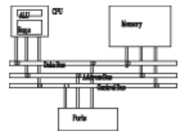
Auxiliary Carry (AC) flag



- 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

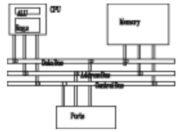


- PF is set when LSB of the destination has an even number of 1 bits.

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

Jump and Loop

JMP and LOOP Instructions



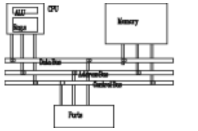
- Transfer of control or branch instructions
  - unconditional
  - conditional
- **JMP** Instruction
- **LOOP** Instruction
- **LOOP** Example
- Summing an Integer Array
- Copying a String

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:

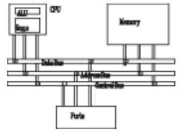
```
top:
    .
    .
    jmp top
```

LOOP Instruction



- The **LOOP** instruction creates a counting loop
- Syntax: **LOOP** *target*
- Logic:
  - ECX ← 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

Exercise . . .

If the relative offset is encoded in a single byte,

(a) what is the largest possible backward jump?

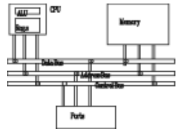
(b) what is the largest possible forward jump?

(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!

Exercise . . .



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.

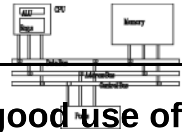
```
.data
count DWORD ?
.code
   mov ecx,100        ; set outer loop count
L1:
   mov count,ecx      ; save outer loop count
   mov ecx,20         ; set inner loop count
L2:...
   loop L2            ; repeat the inner loop
   mov ecx,count      ; restore outer loop count
   loop L1            ; repeat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



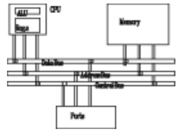
good use of SIZEOF

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
```

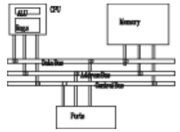
Conditional Processing

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:

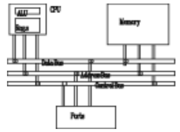
```
mov al, 00111011b
and al, 00001111b
```

```
              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 ∧ 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
```

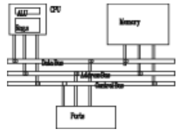
```
            0 0 1 1 1 0 1 1
       XOR  0 0 0 0 1 1 1 1
            ---------------
unchanged — 0 0 1 1|0 1 0 0 — inverted
```

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

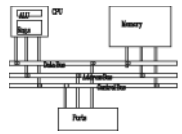
Applications (1 of 4)



- 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
```

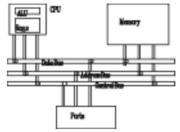
Applications (4 of 4)

- 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



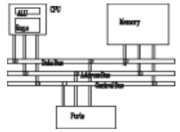
- Performs a nondestructive **AND** operation between each pair of matching bits in two operands
- No operands are modified, but the flags are affected.
- Example: jump to a label if either bit 0 or bit 1 in AL is set.

```
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

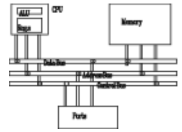
```
mov al,-1
cmp al,5         ; Sign flag != Overflow flag
```

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

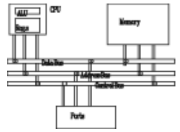
Setting and clearing individual flags



```
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

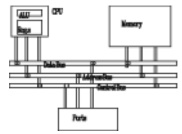
Conditional structures



- 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



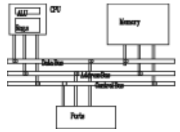
- A conditional jump instruction branches to a label when specific register or flag conditions are met

  `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

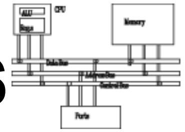


Mnemonic	Description	Flags
JZ	Jump if zero	ZF = 1
JNZ	Jump if not zero	ZF = 0
JC	Jump if carry	CF = 1
JNC	Jump if not carry	CF = 0
JO	Jump if overflow	OF = 1
JNO	Jump if not overflow	OF = 0
JS	Jump if signed	SF = 1
JNS	Jump if not signed	SF = 0
JP	Jump if parity (even)	PF = 1
JNP	Jump if not parity (odd)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

Jumps based on equality

Mnemonic	Description
JE	Jump if equal ($leftOp = rightOp$)
JNE	Jump if not equal ($leftOp \neq rightOp$)
JCXZ	Jump if CX = 0
JECXZ	Jump if ECX = 0

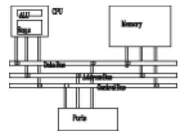
Jumps based on unsigned comparisons



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

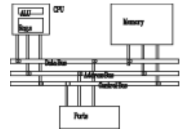
- Compare unsigned AX to BX, and copy the larger of the two into a variable named Large

```
    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:
```

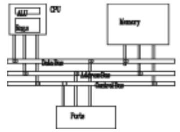
Examples



- 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 ← 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:

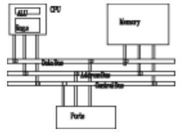
```
.data
array SWORD -3,-6,-1,-10,10,30,40,4
sentinel SWORD 0
.code
   mov esi,OFFSET array
   mov ecx,LENGTHOF array
next:
   test WORD PTR [esi],8000h   ; test sign bit
   pushfd                      ; push flags on stack
   add esi,TYPE array
   popfd                       ; pop flags from stack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

Exercise ...

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



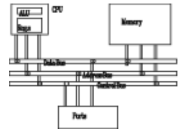
```
.data
array SWORD 50 DUP(?)
sentinel SWORD 0FFFFh
.code
   mov esi,OFFSET array
   mov ecx,LENGTHOF array
L1:cmp WORD PTR [esi],0  ; check for zero
   pushfd                ; push flags on stack
   add esi,TYPE array
   popfd                  ; pop flags from stack
   loope L1              ; continue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

Conditional structures

If statements

```
if [C] then [T] else [E]

         [  C  ]
         JNE else
         [  T  ]
         JMP endif
else:    [  E  ]

endif:
```

Block-structured IF statements

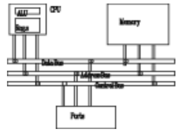


Assembly language programmers can easily translate logical statements written in C++/Java into assembly language. For example:

```
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;
}
```

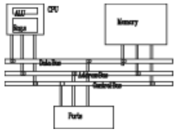
```
    mov eax,var1
    cmp eax,var2
    jle L1
    mov var3,6
    mov var4,7
    jmp L2
L1: mov var3,10
L2:
```

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:
```

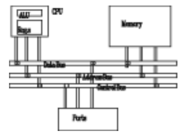
Compound expression with AND

```
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:
```

Exercise . . .



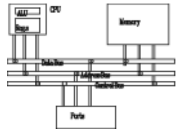
Implement the following pseudocode in assembly language. All values are unsigned:

```
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.)

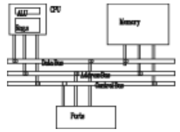
Compound Expression with OR



- 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

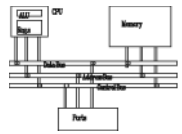


```
if (al > bl) OR (bl > cl)
  X = 1;
```

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:
```

Exercise . . .

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

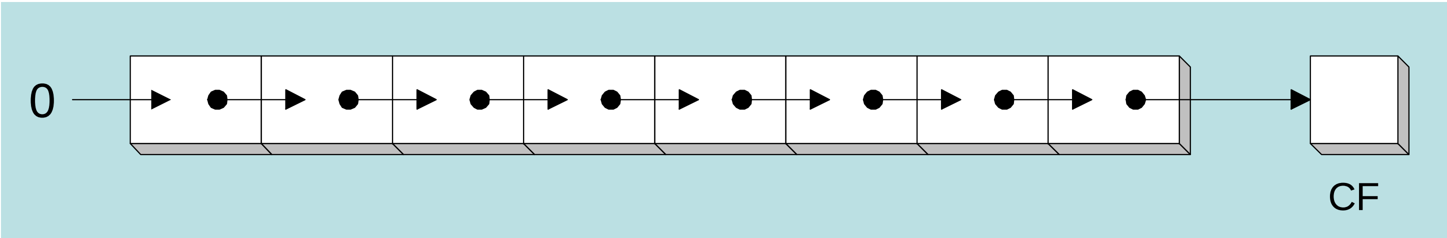
Shift and rotate

Shift and Rotate Instructions

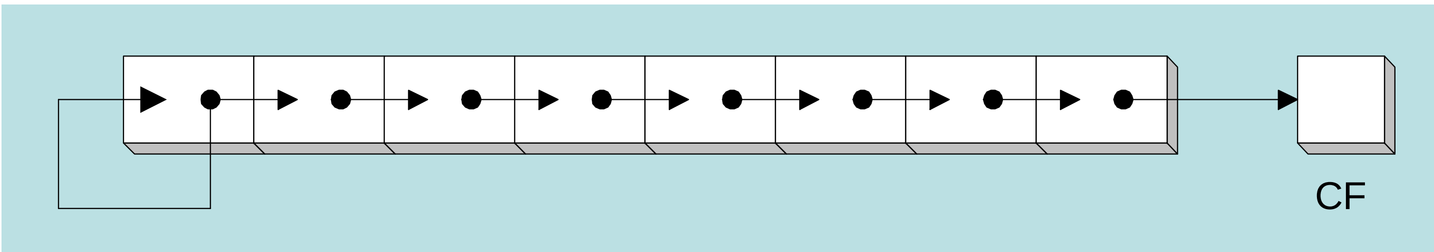
- Logical vs Arithmetic Shifts
- SHL Instruction
- SHR Instruction
- SAL and SAR Instructions
- ROL Instruction
- ROR Instruction
- RCL and RCR Instructions
- SHLD/SHRD Instructions

Logical vs arithmetic shifts

- A logical shift fills the newly created bit position with zero:

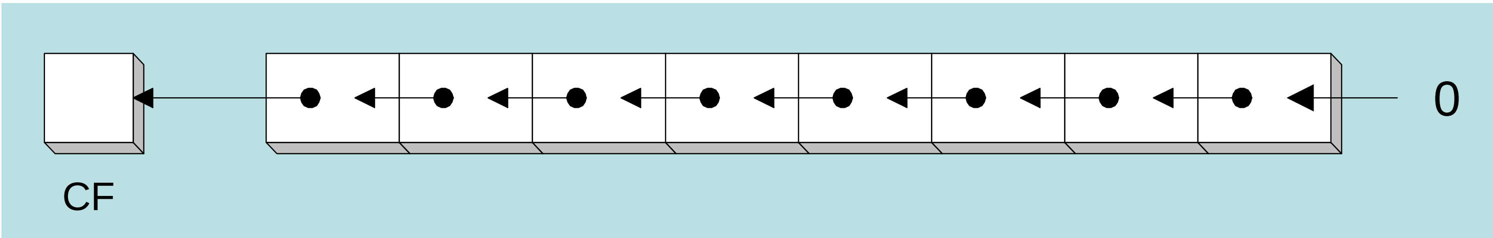


- An arithmetic shift fills the newly created bit position with a copy of the number’s sign bit:



SHL instruction

- The SHL (shift left) instruction performs a logical left shift on the destination operand, filling the lowest bit with 0.



- Operand types: **SHL *destination,count***

SHL *reg,imm8*

SHL *mem,imm8*

SHL *reg*,CL

SHL *mem*,CL

Fast multiplication

Shifting left 1 bit multiplies a number by 2

```
mov dl,5
shl dl,1
```

Before: 0 0 0 0 0 1 0 1 = 5

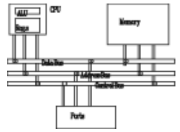
After: 0 0 0 0 1 0 1 0 = 10

Shifting left *n* bits multiplies the operand by 2^n

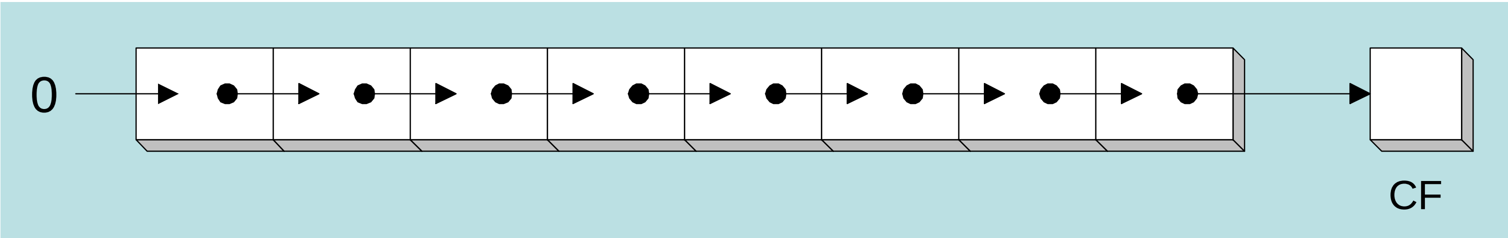
For example, $5 * 2^2 = 20$

```
mov dl,5
shl dl,2                    ; DL = 20
```

SHR instruction



- The SHR (shift right) instruction performs a logical right shift on the destination operand. The highest bit position is filled with a zero.

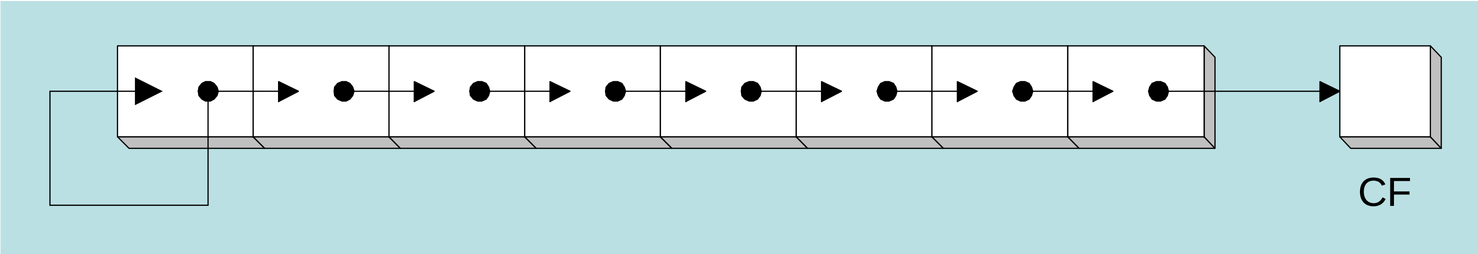


Shifting right n bits divides the operand by 2^n

```
mov dl,80
shr dl,1                ; DL = 40
shr dl,2                ; DL = 10
```

SAL and SAR instructions

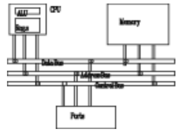
- SAL (shift arithmetic left) is identical to SHL.
- SAR (shift arithmetic right) performs a right arithmetic shift on the destination operand.



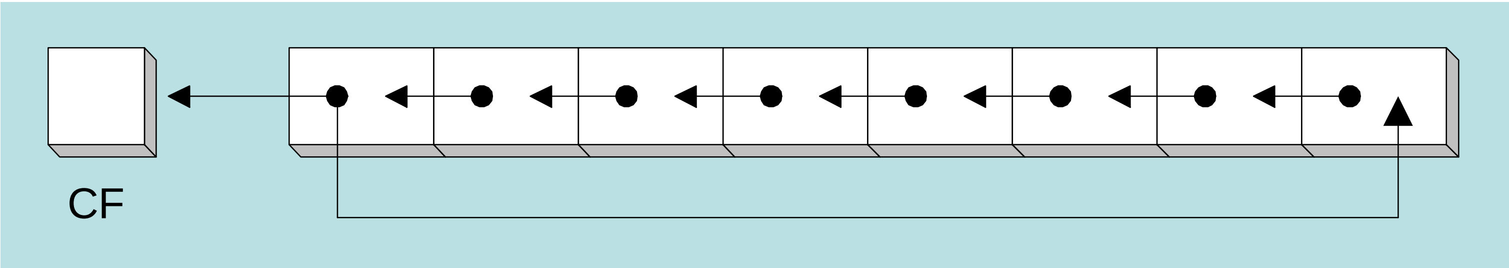
An arithmetic shift preserves the number's sign.

```
mov dl,-80
sar dl,1                    ; DL = -40
sar dl,2                    ; DL = -10
```

ROL instruction



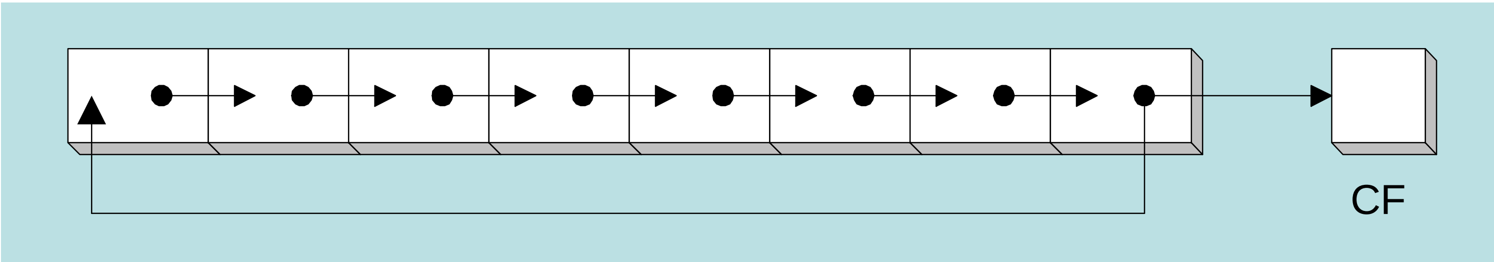
- ROL (rotate) shifts each bit to the left
- The highest bit is copied into both the Carry flag and into the lowest bit
- No bits are lost




```
mov al,11110000b
rol al,1                ; AL = 11100001b

mov dl,3Fh
rol dl,4                ; DL = F3h
```

ROR instruction

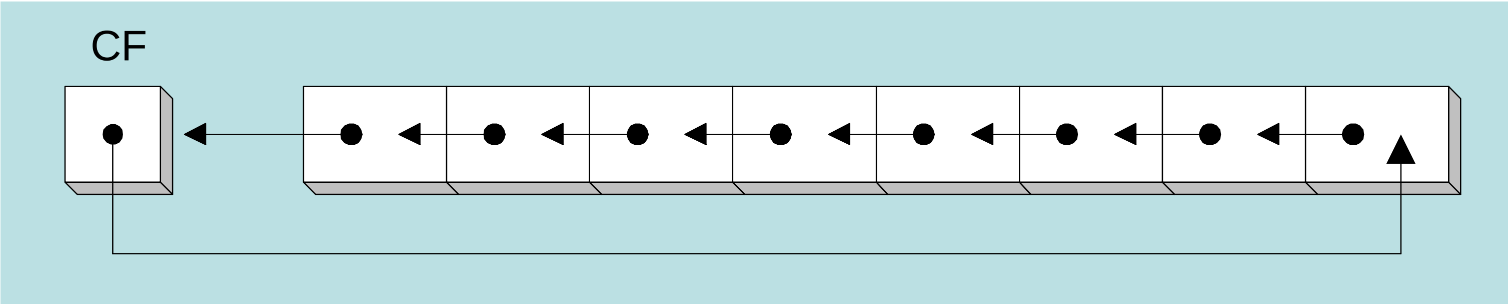
- ROR (rotate right) shifts each bit to the right
- The lowest bit is copied into both the Carry flag and into the highest bit
- No bits are lost




```
mov al,11110000b
ror al,1                  ; AL = 01111000b

mov dl,3Fh
ror dl,4                  ; DL = F3h
```

RCL instruction

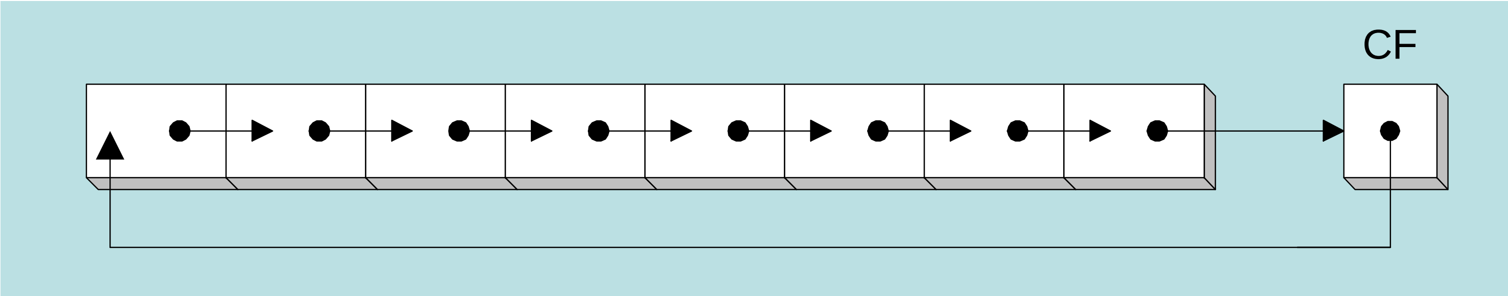
- RCL (rotate carry left) shifts each bit to the left
- Copies the Carry flag to the least significant bit
- Copies the most significant bit to the Carry flag




```
clc                     ; CF = 0
mov bl,88h              ; CF,BL = 0 10001000b
rcl bl,1                ; CF,BL = 1 00010000b
rcl bl,1                ; CF,BL = 0 00100001b
```

RCR instruction

- RCR (rotate carry right) shifts each bit to the right
- Copies the Carry flag to the most significant bit
- Copies the least significant bit to the Carry flag




```
stc                     ; CF = 1
mov ah,10h              ; CF,AH = 00010000 1
rcr ah,1                ; CF,AH = 10001000 0
```

SHLD instruction

- Syntax: (shift left double)

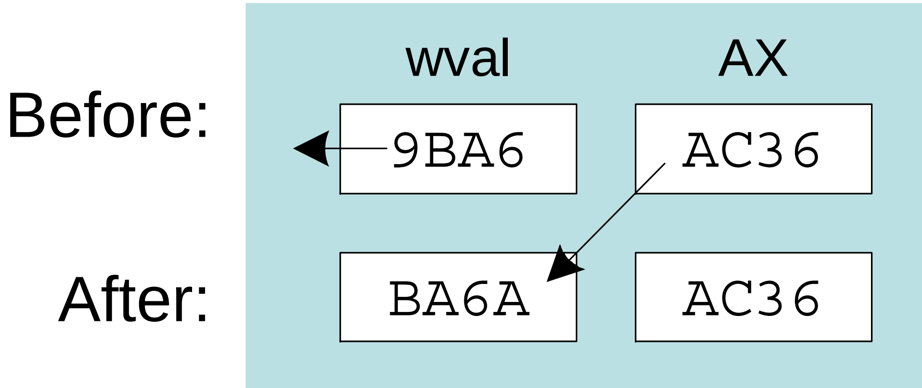
  SHLD ***destination, source, count***

- Shifts a destination operand a given number of bits to the left
- The bit positions opened up by the shift are filled by the most significant bits of the source operand
- The source operand is not affected

SHLD example

Shift **wval** 4 bits to the left and replace its lowest 4 bits with the high 4 bits of **AX**:

```
.data
wval WORD 9BA6h
.code
mov  ax,0AC36h
shld wval,ax,4
```



SHRD instruction

- Syntax:

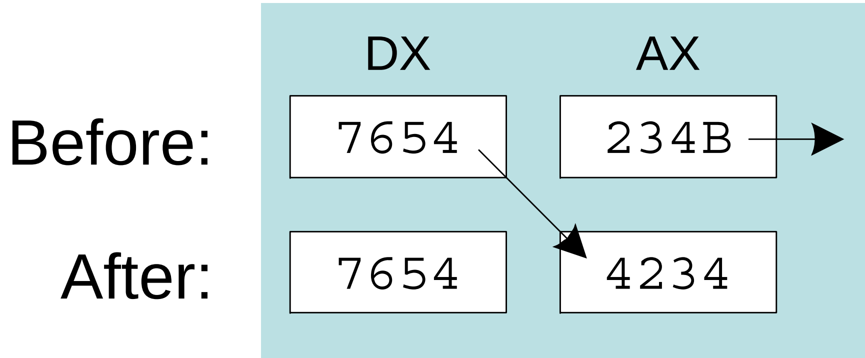
  SHRD *destination, source, count*

- Shifts a destination operand a given number of bits to the right
- The bit positions opened up by the shift are filled by the least significant bits of the source operand
- The source operand is not affected

SHRD example

Shift AX 4 bits to the right and replace its highest 4 bits with the low 4 bits of DX:

```
mov  ax,234Bh
mov  dx,7654h
shrd ax,dx,4
```



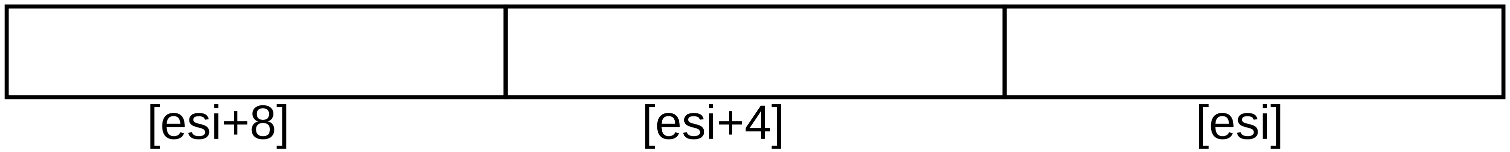
Shift and rotate applications

- Shifting Multiple Doublewords
- Binary Multiplication
- Displaying Binary Bits
- Isolating a Bit String

Shifting multiple doublewords

- Programs sometimes need to shift all bits within an array, as one might when moving a bitmapped graphic image from one screen location to another.
- The following shifts an array of 3 doublewords 1 bit to the right:

```
shr array[esi + 8],1 ; high dword
rcr array[esi + 4],1 ; middle dword,
rcr array[esi],1     ; low dword,
```



Binary multiplication

- We already know that SHL performs unsigned multiplication efficiently when the multiplier is a power of 2.
- Factor any binary number into powers of 2.
  - For example, to multiply EAX * 36, factor 36 into 32 + 4 and use the distributive property of multiplication to carry out the operation:

```
EAX * 36
= EAX * (32 + 4)
= (EAX * 32)+(EAX * 4)
```

```
mov eax,123
mov ebx,eax
shl eax,5
shl ebx,2
add eax,ebx
```

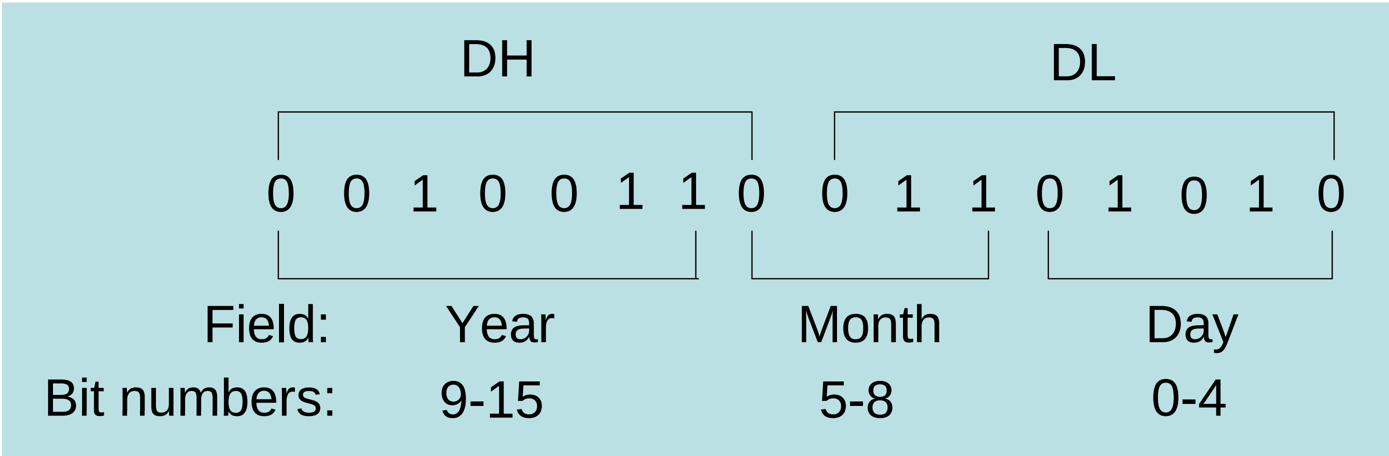
Displaying binary bits

Algorithm: Shift MSB into the Carry flag; If CF = 1, append a "1" character to a string; otherwise, append a "0" character. Repeat in a loop, 32 times.

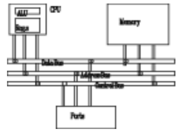
```
    mov ecx,32
    mov esi,offset buffer
L1: shl eax,1
    mov BYTE PTR [esi],'0'
    jnc L2
    mov BYTE PTR [esi],'1'
L2: inc esi
    loop L1
```

Isolating a bit string

- The MS-DOS file date field packs the year (relative to 1980), month, and day into 16 bits:



Isolating a bit string



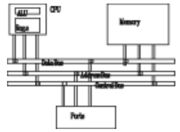
```
mov al,dl          ; make a copy of DL
and al,00011111b   ; clear bits 5-7
mov day,al         ; save in day variable
```

```
mov ax,dx          ; make a copy of DX
shr ax,5           ; shift right 5 bits
and al,00001111b   ; clear bits 4-7
mov month,al       ; save in month variable
```

```
mov al,dh          ; make a copy of DX
shr al,1           ; shift right 1 bit
mov ah,0           ; clear AH to 0
add ax,1980        ; year is relative to 1980
mov year,ax        ; save in year
```

Multiplication and division

MUL instruction



- The MUL (unsigned multiply) instruction multiplies an 8-, 16-, or 32-bit operand by either AL, AX, or EAX.
- The instruction formats are:

MUL r/m8

MUL r/m16

MUL r/m32

Implied operands:

Multiplicand	Multiplier	Product
AL	*r/m8*	AX
AX	*r/m16*	DX:AX
EAX	*r/m32*	EDX:EAX

MUL examples

100h * 2000h, using 16-bit operands:

```
.data
val1 WORD 2000h
val2 WORD 100h
.code
mov ax,val1
mul val2  ; DX:AX=00200000h, CF=1
```

The Carry flag indicates whether or not the upper half of the product contains significant digits.

12345h * 1000h, using 32-bit operands:

```
mov eax,12345h
mov ebx,1000h
mul ebx   ; EDX:EAX=0000000012345000h, CF=0
```

IMUL instruction

- IMUL (signed integer multiply) multiplies an 8-, 16-, or 32-bit signed operand by either AL, AX, or EAX (there are one/two/three operand format)
- Preserves the sign of the product by sign-extending it into the upper half of the destination register

Example: multiply 48 * 4, using 8-bit operands:

```
mov  al,48
mov  bl,4
imul bl       ; AX = 00C0h, OF=1
```

OF=1 because AH is not a sign extension of AL.

DIV instruction

- The DIV (unsigned divide) instruction performs 8-bit, 16-bit, and 32-bit division on unsigned integers
- A single operand is supplied (register or memory operand), which is assumed to be the divisor
- Instruction formats:

DIV *r/m8*

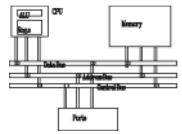
DIV *r/m16*

DIV *r/m32*

Default Operands:

Dividend	Divisor	Quotient	Remainder
AX	*r/m8*	AL	AH
DX:AX	*r/m16*	AX	DX
EDX:EAX	*r/m32*	EAX	EDX

DIV examples



Divide 8003h by 100h, using 16-bit operands:

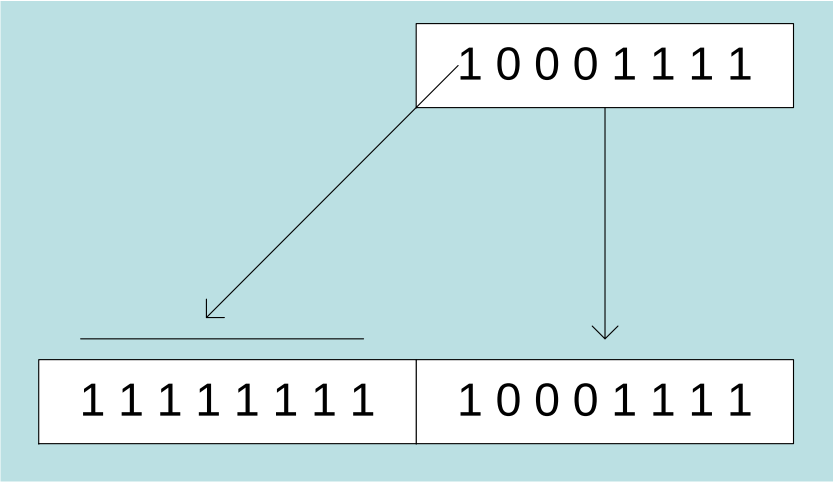
```
mov dx,0            ; clear dividend, high
mov ax,8003h        ; dividend, low
mov cx,100h         ; divisor
div cx              ; AX = 0080h, DX = 3
```

Same division, using 32-bit operands:

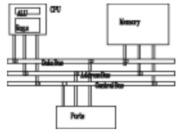
```
mov edx,0           ; clear dividend, high
mov eax,8003h       ; dividend, low
mov ecx,100h        ; divisor
div ecx             ; EAX=00000080h,EDX=3
```

Signed integer division

- Signed integers must be sign-extended before division takes place
  - fill high byte/word/doubleword with a copy of the low byte/word/doubleword's sign bit
- For example, the high byte contains a copy of the sign bit from the low byte:



CBW, CWD, CDQ instructions



- The CBW, CWD, and CDQ instructions provide important sign-extension operations:
  - CBW (convert byte to word) extends AL into AH
  - CWD (convert word to doubleword) extends AX into DX
  - CDQ (convert doubleword to quadword) extends EAX into EDX
- For example:

```
mov eax,0FFFFFF9Bh      ; -101 (32 bits)
cdq             ; EDX:EAX = FFFFFFFFFFFFFF9Bh
                ; -101 (64 bits)
```

IDIV instruction

- IDIV (signed divide) performs signed integer division
- Uses same operands as DIV

Example: 8-bit division of –48 by 5

```
mov al,-48
cbw             ; extend AL into AH
mov bl,5
idiv bl         ; AL = -9,  AH = -3
```

IDIV examples

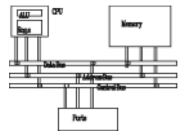
Example: 16-bit division of –48 by 5

```
mov  ax,-48
cwd                  ; extend AX into DX
mov  bx,5
idiv bx         ; AX = -9,  DX = -3
```

Example: 32-bit division of –48 by 5

```
mov  eax,-48
cdq             ; extend EAX into EDX
mov  ebx,5
idiv ebx        ; EAX = -9,  EDX = -3
```

Divide overflow



- *Divide overflow* happens when the quotient is too large to fit into the destination.

```
mov ax, 1000h
mov bl, 10h
div bl
```

It causes a CPU interrupt and halts the program. (divided by zero cause similar results)

Arithmetic expressions

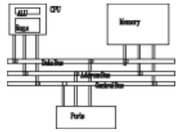
Implementing arithmetic expressions

- Some good reasons to learn how to implement expressions:
  - Learn how compilers do it
  - Test your understanding of MUL, IMUL, DIV, and IDIV
  - Check for 32-bit overflow

Example: **`var4 = (var1 + var2) * var3`**

```
mov eax,var1
add eax,var2
mul var3
jo  TooBig      ; check for overflow
mov var4,eax    ; save product
```

Implementing arithmetic expressions



Example: **eax = (-var1 * var2) + var3**

```
mov eax,var1
neg eax
mul var2
jo  TooBig      ; check for overflow
add eax,var3
```

Example: **var4 = (var1 * 5) / (var2 – 3)**

```
mov eax,var1      ; left side
mov ebx,5
mul ebx           ; EDX:EAX = product
mov ebx,var2      ; right side
sub ebx,3
div ebx           ; final division
mov var4,eax
```

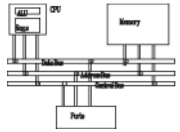
Implementing arithmetic expressions

Example: **`var4 = (var1 * -5) / (-var2 % var3);`**

```
mov  eax,var2      ; begin right side
neg  eax
cdq                ; sign-extend dividend
idiv var3          ; EDX = remainder
mov  ebx,edx       ; EBX = right side
mov  eax,-5        ; begin left side
imul var1          ; EDX:EAX = left side
idiv ebx           ; final division
mov  var4,eax      ; quotient
```

Sometimes it's easiest to calculate the right-hand term of an expression first.

Exercise . . .

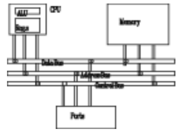


Implement the following expression using signed 32-bit integers:

```
eax = (ebx * 20) / ecx
```

```
mov eax,20
mul ebx
div ecx
```

Exercise . . .



Implement the following expression using signed 32-bit integers. Save and restore ECX and EDX:

```
eax = (ecx * edx) / eax
```

```
push ecx
push edx
push eax                ; EAX needed later
mov  eax,ecx
mul  edx                ; left side: EDX:EAX
pop  ecx                ; saved value of EAX
div  ecx                ; EAX = quotient
pop  edx                ; restore EDX, ECX
pop  ecx
```

Exercise . . .

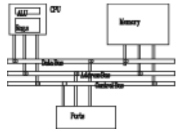
Implement the following expression using signed 32-bit integers. Do not modify any variables other than var3:

var3 = (var1 * -var2) / (var3 – ebx)

```
mov eax,var1
mov edx,var2
neg edx
mul edx          ; left side: edx:eax
mov ecx,var3
sub ecx,ebx
div ecx          ; eax = quotient
mov var3,eax
```

Extended addition and subtraction

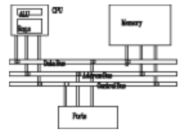
ADC instruction



- ADC (add with carry) instruction adds both a source operand and the contents of the Carry flag to a destination operand.
- Example: Add two 32-bit integers (FFFFFFFFh + FFFFFFFFh), producing a 64-bit sum:

```
mov edx,0
mov eax,0FFFFFFFFh
add eax,0FFFFFFFFh
adc edx,0 ;EDX:EAX = 00000001FFFFFFFEh
```

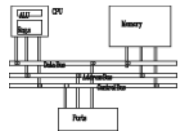
Extended addition example



- Add two integers of any size
- Pass pointers to the addends (ESI, EDI) and sum (EBX), ECX indicates the number of doublewords

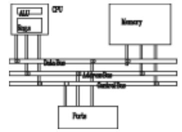
```
L1:
  mov eax,[esi] ; get the first integer
  adc eax,[edi] ; add the second integer
  pushfd        ; save the Carry flag
  mov [ebx],eax ; store partial sum
  add esi,4     ; advance all 3 pointers
  add edi,4
  add ebx,4
  popfd         ; restore the Carry flag
  loop L1       ; repeat the loop
  adc word ptr [ebx],0 ; add leftover carry
```

Extended addition example



```
.data
op1 QWORD 0A2B2A40674981234h
op2 QWORD 08010870000234502h
sum DWORD 3 dup(?)
    ; = 0000000122C32B0674BB5736
.code
...
mov  esi,OFFSET op1 ; first operand
mov  edi,OFFSET op2 ; second operand
mov  ebx,OFFSET sum ; sum operand
mov  ecx,2          ; number of doublewords
call Extended_Add
...
```

SBB instruction



- The SBB (subtract with borrow) instruction subtracts both a source operand and the value of the Carry flag from a destination operand.
- The following example code performs 64-bit subtraction. It sets EDX:EAX to 0000000100000000h and subtracts 1 from this value. The lower 32 bits are subtracted first, setting the Carry flag. Then the upper 32 bits are subtracted, including the Carry flag:

```
mov edx,1          ; upper half
mov eax,0          ; lower half
sub eax,1          ; subtract 1
sbb edx,0          ; subtract upper half
```

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;
}
```