Assembly Language for Intel-Based Computers, 4th Edition

Kip R. Irvine

Lecture 24

Rotations, Integer Arithmetic, Extended Addition

Slides prepared by Kip R. Irvine

Revision date: 07/11/2002

Modified on April 20.2005 by Dr. Nikolay Metodiev Sirakov

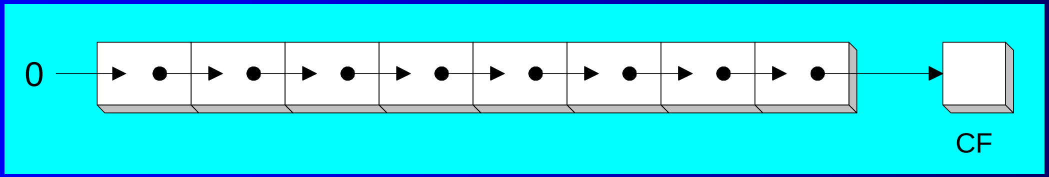
- Chapter corrections (Web)   Assembly language sources (Web)

Shift and Rotate Instructions

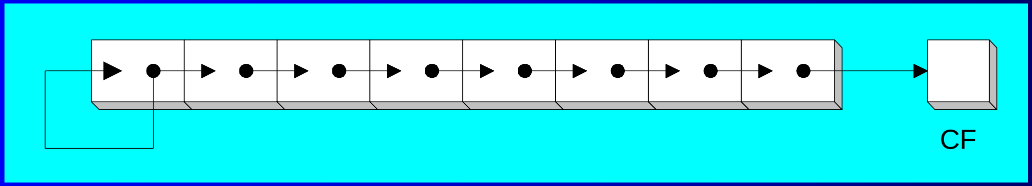
- Logical vs Arithmetic Shifts
- SHL Instruction
- SHR Instruction
- SAL and SAR Instructions
- ROL Instruction
- ROR Instruction
- RCL and RCR Instructions
- SHLD/SHRD Instructions

Logical vs Arithmetic Shifts

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

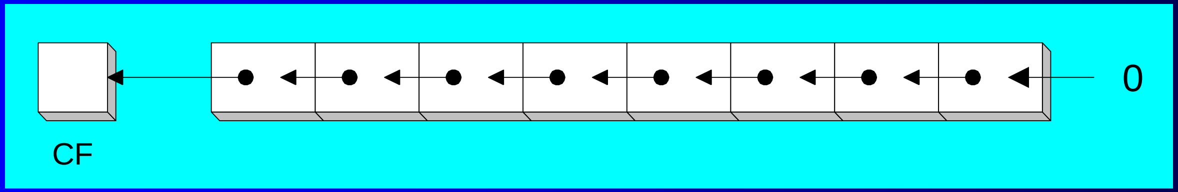


- An arithmetic shift fills the newly created bit position with a copy of the number's sign bit:



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 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

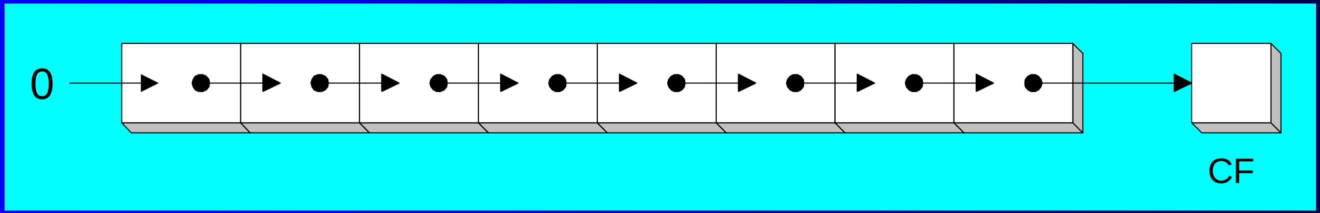
Shifting left *n* bits multiplies the operand by 2^n

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.

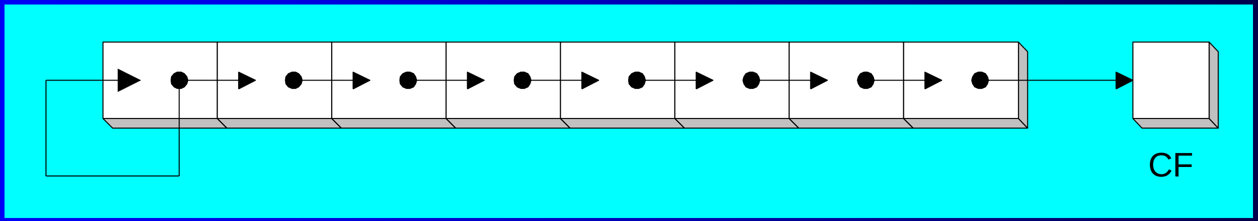


Shifting right *n* bits divides the operand by 2^n

```
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
```

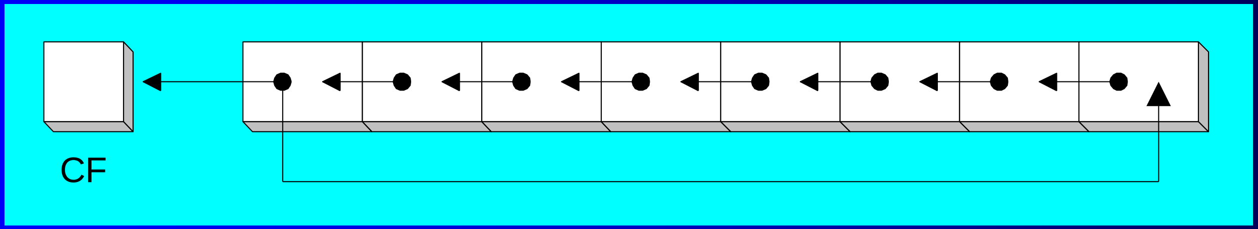
Your turn . . .

Indicate the hexadecimal value of AL after each shift:

`mov al,6Bh`	
`shr al,1`	a. **35h**
`shl al,3`	b. **A8h**
`mov al,8Ch`	
`sar al,1`	c. **C6h**
`sar al,3`	d. **F8h**

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

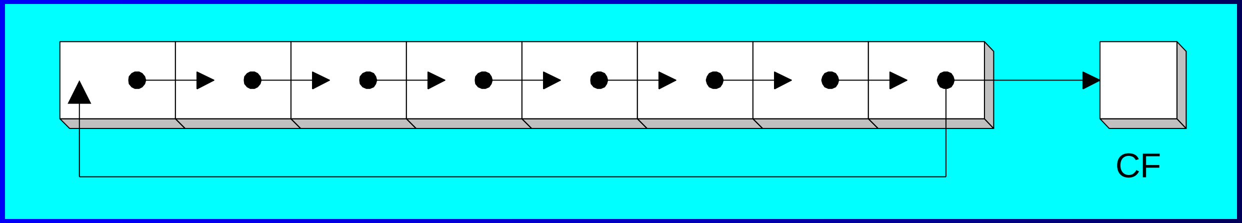



```
mov al,11110000b
rol al,1                         ; AL = 11100001b

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
```

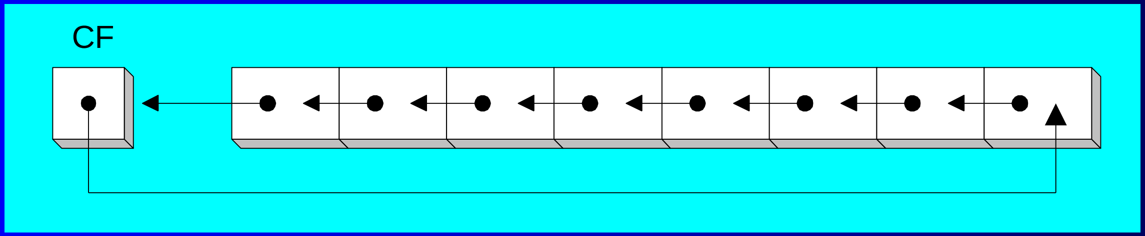
Your turn . . .

Indicate the hexadecimal value of AL after each rotation:

```
mov al,6Bh
ror al,1                              a.  B5h
rol al,3                              b.  ADh
```

RCL Instruction

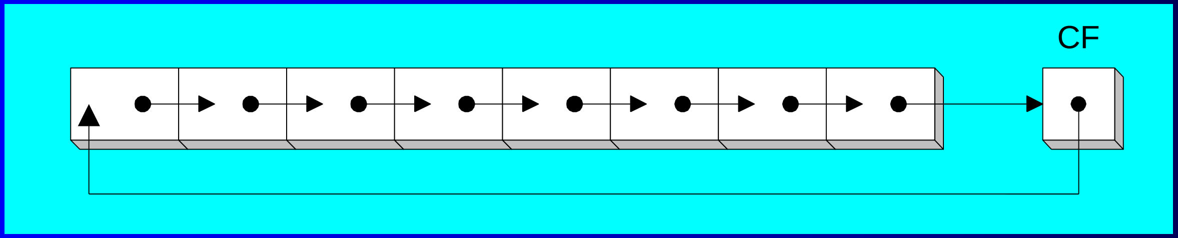
- RCL (rotate carry left) shifts each bit to the left
- Copies the Carry flag to the least significant bit
- Copies the most significant bit to the Carry flag




```
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
```

Your turn . . .

Indicate the hexadecimal value of AL after each rotation:

```
stc
mov al,6Bh
rcr al,1                          a. B5h
rcl al,3                          b. AEh
```

SHLD Instruction

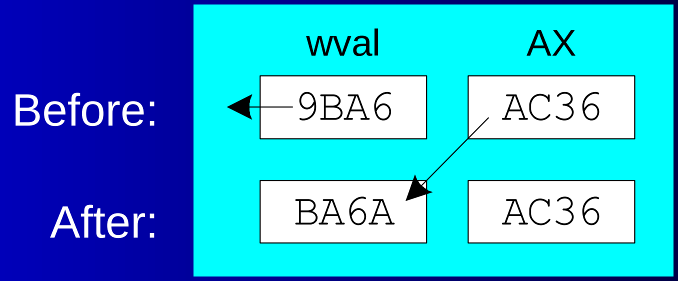
- Shifts a destination operand a given number of bits to the left
- The bit positions opened up by the shift are filled by the most significant bits of the source operand
- The source operand is not affected
- Syntax:

  SHLD *destination, source, count*

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

- 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
- Syntax:

  SHRD *destination, source, count*

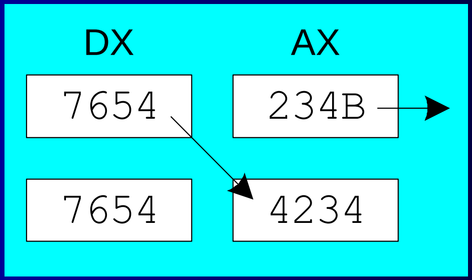
SHRD Example

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
```

Before:

After:



Your turn . . .

Indicate the hexadecimal values of each destination operand:

```
mov   ax,7C36h
mov   dx,9FA6h
shld dx,ax,4                ; DX = FA67h
shrd dx,ax,8                ; DX = 36FAh
```

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 (view complete source code):

```
.data
ArraySize = 3
array DWORD ArraySize DUP(99999999h)      ; 1001 1001...
.code
mov esi,0
shr array[esi + 8],1          ; high dword
rcr array[esi + 4],1          ; middle dword, include Carry
rcr array[esi],1              ; low dword, include Carry
```

Binary Multiplication

- We already know that SHL performs unsigned multiplication efficiently when the multiplier is a power of 2.
- You can 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          ; mult by 2^5
shl ebx,2          ; mult by 2^2
add eax,ebx
```

Multiplication and Division Instructions

- MUL Instruction
- IMUL Instruction
- DIV Instruction
- Signed Integer Division
- Implementing Arithmetic Expressions

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
```

Your turn . . .

What will be the hexadecimal values of DX, AX, and the Carry flag after the following instructions execute?

```
mov ax,1234h
mov bx,100h
mul bx
```

DX = 0012h, AX = 3400h, CF = 1

Your turn . . .

What will be the hexadecimal values of EDX, EAX, and the Carry flag after the following instructions execute?

```
mov eax,00128765h
mov ecx,10000h
mul ecx
```

EDX = 00000012h, EAX = 87650000h, CF = 1

IMUL Instruction

- IMUL (signed integer multiply) multiplies an 8-, 16-, or 32-bit signed operand by either AL, AX, or EAX
- 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.

Your turn . . .

What will be the hexadecimal values of DX, AX, and the Carry flag after the following instructions execute?

```
mov ax,8760h
mov bx,100h
imul bx
```

DX = FF87h, AX = 6000h, OF = 1

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, DX = 3
```

Your turn . . .

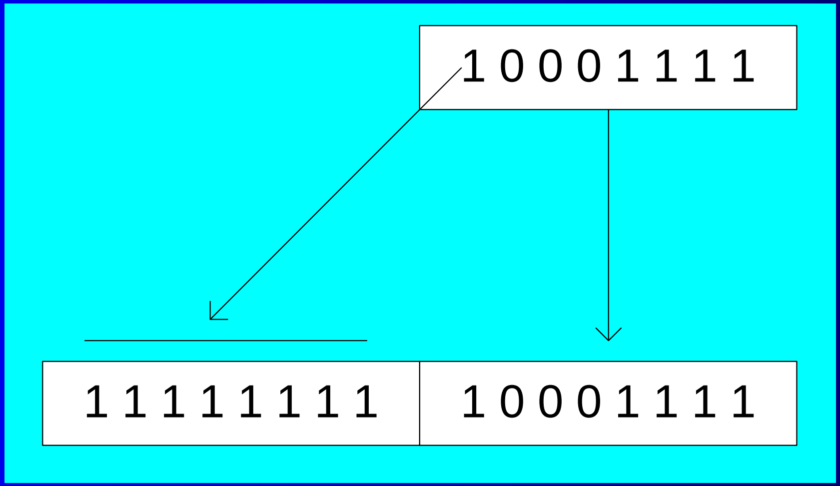
What will be the hexadecimal values of DX and AX after the following instructions execute? Or, if divide overflow occurs, you can indicate that as your answer:

```
mov dx,0087h
mov ax,6000h
mov bx,100h
div bx
```

DX = 0000h, AX = 8760h

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
cdq            ; EDX:EAX = FFFFFFFFFFFFFF9Bh
```

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
```

Your turn . . .

What will be the hexadecimal values of DX and AX after the following instructions execute? Or, if divide overflow occurs, you can indicate that as your answer:

```
mov ax,0FDFFh                    ; -513
cwd
mov bx,100h
idiv bx
```

DX = FFFFh (–1), AX = FFFEh (–2)

Your turn . . .

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
```

Your turn . . .

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 ASCII Addition and Subtraction

- ADC Instruction
- Extended Addition Example
- SBB Instruction

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 and sum
- ECX indicates the number of words

```
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 any leftover carry
```

View the complete source code.