

CSC 3210

Computer Organization and
Programming

CHAPTER 7: INTEGER ARITHMETIC

Outline

- **Shift and Rotate Instructions**
- Multiplication and Division Instructions
- Extended Addition and Subtraction
- ASCII and Unpacked Decimal Arithmetic
- Packed Decimal Arithmetic

Shift and Rotate Instructions

- Bit shifting means to move bits **right** and **left** inside an **operand**
- x86 processors provide a particularly **set of instructions**
- These instructions affect the **Overflow** and **Carry** flags

Table 7-1 Shift and Rotate Instructions.

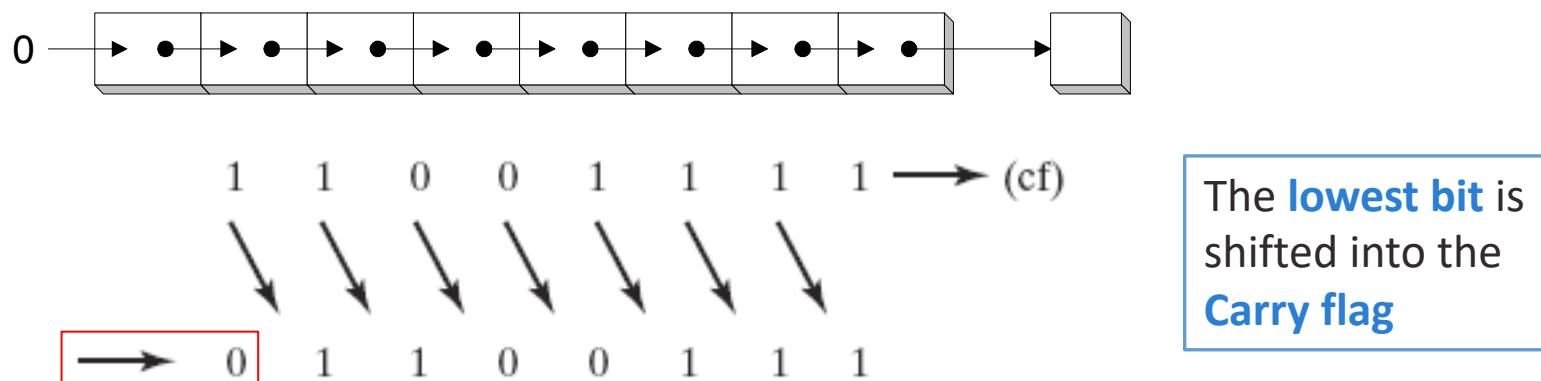
SHL	Shift left
SHR	Shift right
SAL	Shift arithmetic left
SAR	Shift arithmetic right
ROL	Rotate left
ROR	Rotate right
RCL	Rotate carry left
RCR	Rotate carry right
SHLD	Double-precision shift left
SHRD	Double-precision shift right

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

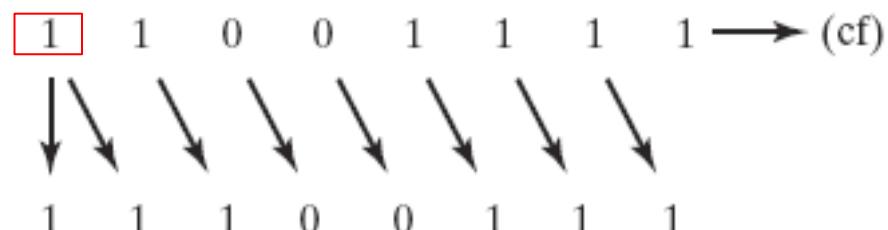
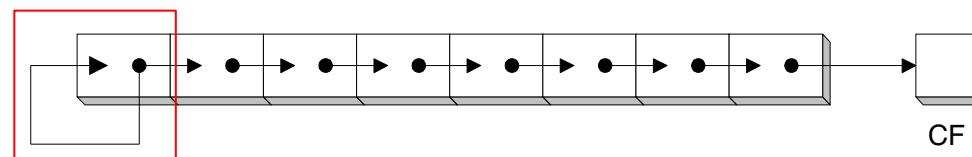
Logical Shift

- There are **two ways** to shift an operand's bits
 - **The first:** **logical shift**, fills the newly created bit position with zero
- **Example:**
 - A byte is logically shifted one position to the right
 - Each bit is moved to the next lowest bit position
 - Note that **bit 7** is assigned 0:



Arithmetic Shift

- **The Second:** arithmetic shift, the newly created bit position is filled with a copy of the original number's sign bit
- Example:
 - Binary **11001111** has a 1 in the sign bit
 - When shifted arithmetically 1 bit to the right, it becomes **11100111**:



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

SHL Instruction

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

SHL reg, imm8

SHL mem, imm8

SHL reg, CL

SHL mem, CL

Same for all shift and rotate instructions:
SHR, SAL, SAR, ROR, ROL, RCR, and RCL

- **CL** register can contain a shift count.

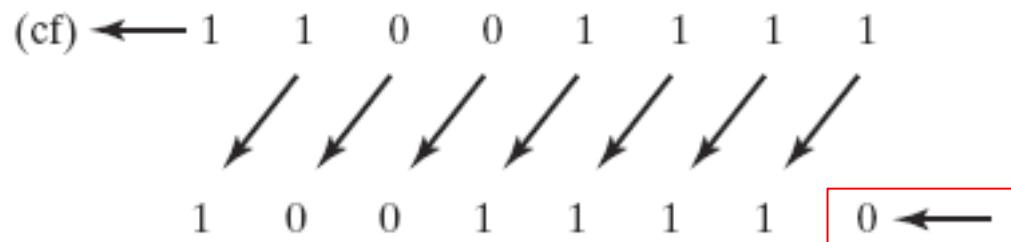
SHL Instruction

- The SHL (shift left) instruction

- Example:**

- BL is shifted once to the left.
- The highest bit is copied into the **Carry flag** and the lowest bit position is assigned zero:

```
mov bl,8Fh          ; BL = 10001111b
shl bl,1           ; CF = 1, BL = 00011110b
```



SHL reg, imm8
SHL mem, imm8
SHL reg, CL
SHL mem, CL

SHL Instruction

- When a value is **shifted leftward multiple times**,
 - The **Carry flag** contains **the last bit** to be shifted out of the most significant bit (MSB)
- **Example:**
 - **bit 7** does not end up in the **Carry flag** because it is replaced by bit 6 (a zero):

```
mov al,10000000b  
shl al,2           ; CF = 0, AL = 00000000b
```

- Similarly, when a value is **shifted rightward multiple times**,
 - The **Carry flag** contains the last bit to be shifted out of the least significant bit (LSB)

SHL Instruction

- **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**, $5 * 2^2 = 20$

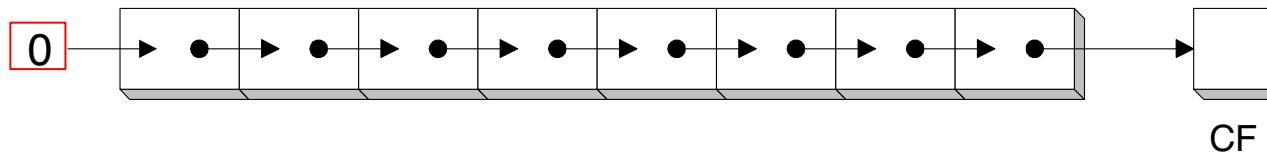
```
mov dl,5  
shl dl,2 ; DL = 20
```

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

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.



- **Example,**
 - The **0** from the lowest bit in AL is copied into the Carry flag,
 - And the **highest bit** in AL is filled with a zero:

mov al,0D0h	; AL = 11010000b
shr al,1	; AL = 01101000b, CF = 0

SHR Instruction

- In a multiple shift operation,
 - the **last bit to be shifted** out of position 0 (the LSB) ends up in the **Carry flag**:

```
mov al,00000010b  
shr al,2           ; AL = 00000000b, CF = 1
```

SHR Instruction

- **Fast Division**
 - **Shifting right** n bits divides the operand by 2^n

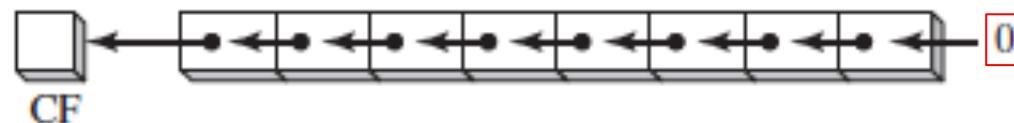
```
mov dl,80
shr dl,1          ; DL = 40  ****
shr dl,2          ; DL = 10  ****
```

Shift and Rotate Instructions

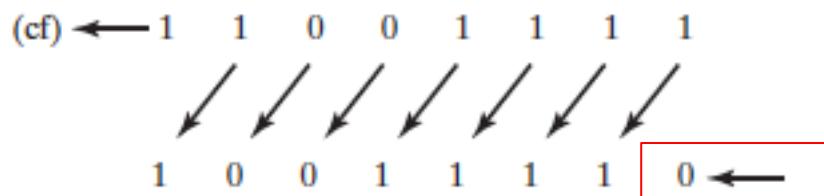
- Logical vs Arithmetic Shifts
- SHL Instruction
- SHR Instruction
- **SAL and SAR Instructions**
- ROL Instruction
- ROR Instruction
- RCL and RCR Instructions
- SHLD/SHRD Instructions

SAL Instruction

- SAL (shift arithmetic left) is identical to SHL
 - The **lowest** bit is assigned 0
 - The **highest** bit is moved to the **Carry flag**,
 - And the bit that was in the Carry flag is **discarded**:

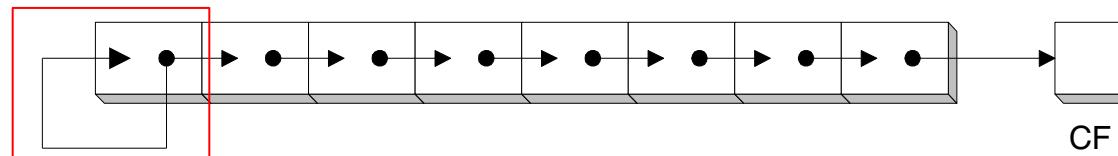


- If you shift binary 11001111 to **the left** by one bit,
 - it becomes 10011110:



SAR Instruction

- SAR is identical to SHR
- SAR (shift arithmetic right) performs a right arithmetic shift on the destination operand



- The following example shows how SAR duplicates the sign bit.
 - AL is negative before and after it is shifted to the right:

```
mov al,0F0h          ; AL = 11110000b (-16?)  
sar al,1           ; AL = 11111000b ( -8 ?)    CF = 0
```

SAR Instruction

- SAR is identical to SHR

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

Application

- Indicate the hexadecimal value of **AL** after each shift:

`mov al,6Bh`

`shr al,1`

`shl al,3`

a. 35h

b. A8h

`mov al,8Ch`

`sar al,1`

`sar al,3`

c. C6h

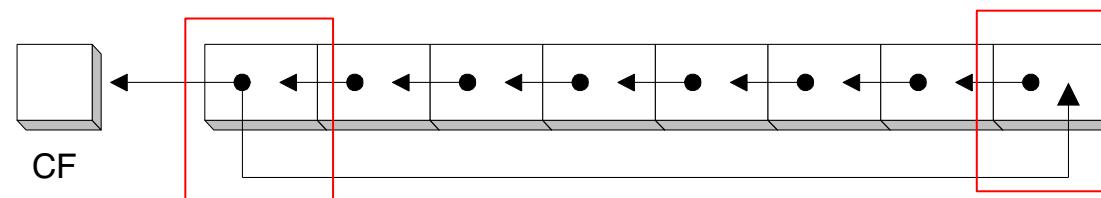
d. F8h

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

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



Example1:

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


ROL Instruction

- **ROL** (rotate) shifts each bit to the **left**

Example2:

```
mov al,40h    ; AL = 01000000b
rol al,1      ; AL = 10000000b, CF = 0
rol al,1      ; AL = 00000001b, CF = 1
rol al,1      ; AL =00000010b, CF = 0
```

ROL Instruction

- **Multiple Rotations**
 - When using a rotation count greater than 1,
 - Carry flag contains the last bit rotated out of the MSB position:

```
mov al,00100000b  
rol al,3           ; CF = 1, AL = 00000001b
```

ROL Instruction

- **Exchanging Groups of Bits (Application)**

- You can use ROL to exchange the upper (bits 4–7) and lower (bits 0–3) halves of a byte

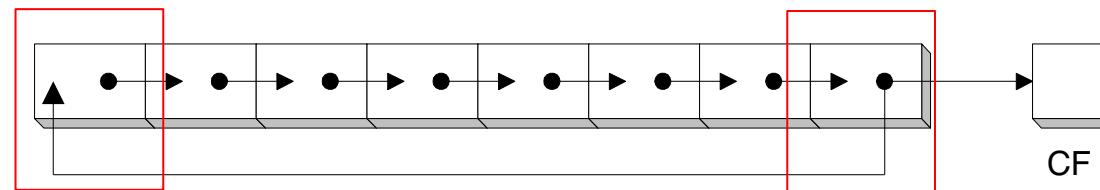
```
mov al,26h  
rol al,4      ; AL = 62h
```

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

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



Example1:

```
mov al,01h      ; AL = 00000001b
ror al,1        ; AL = 10000000b, CF = 1
ror al,1        ; AL = 01000000b, CF = 0
```

ROR Instruction

- ROR (rotate right) shifts each bit to the right

Example2:

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

ROR Instruction

- **Multiple Rotations**
 - When using a rotation count greater than 1,
 - **Carry flag contains the last bit rotated out of the LSB position:**

```
mov al,00000100b  
ror al,3           ; AL = 10000000b,      CF = 1
```

Application

- Indicate the hexadecimal value of **AL** after each rotation:

mov al,6Bh

ror al,1

rol al,3

a. **B5h**

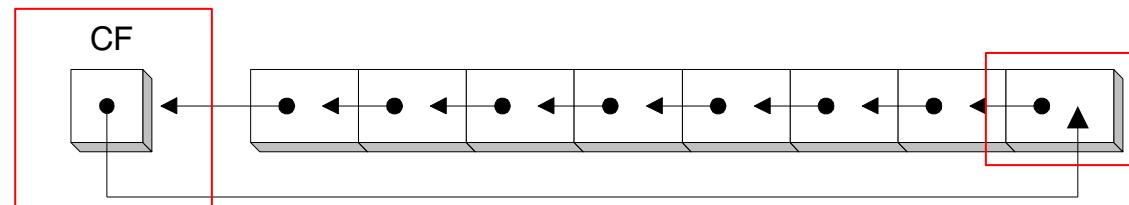
b. **ADh**

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

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

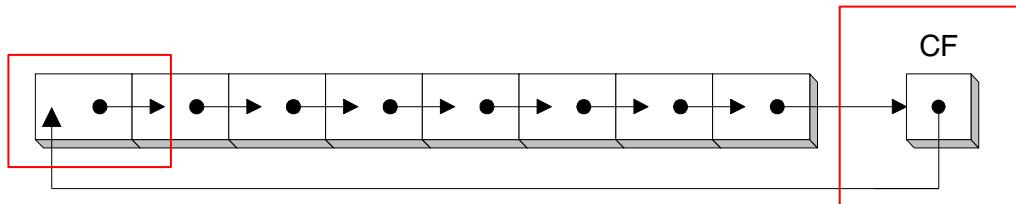


```
clc  
mov bl,88h  
rcl bl,1  
rcl bl,1
```

```
; CF = 0 , CLC instruction clears the Carry flag  
; CF = 0 , BL= 10001000b  
; CF = 1 , BL= 00010000b  
; CF = 0 , BL= 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 , STC to set the Carry flag  
mov ah,10h    ; CF = 1 , AH = 00010000b  
rcr ah,1      ; CF = 0 , AH = 10001000b
```

Application

- Indicate the hexadecimal value of AL after each rotation:

stc

mov al,6Bh

rcr al,1

rcl al,3

- a. B5h
- b. AEh

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: See the book**

Table 7-1 Shift and Rotate Instructions.

SHL	Shift left
SHR	Shift right
SAL	Shift arithmetic left
SAR	Shift arithmetic right
ROL	Rotate left
ROR	Rotate right
RCL	Rotate carry left
RCR	Rotate carry right
SHLD	Double-precision shift left
SHRD	Double-precision shift right

Outline

- Shift and Rotate Instructions
- **Multiplication and Division Instructions**
- Extended Addition and Subtraction
- ASCII and Unpacked Decimal Arithmetic
- Packed Decimal Arithmetic

Multiplication and Division Instructions

- MUL Instruction
- IMUL Instruction
- DIV Instruction
- Signed Integer Division
- CBW, CWD, CDQ Instructions
- IDIV Instruction
- Implementing Arithmetic Expressions

MUL Instruction

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

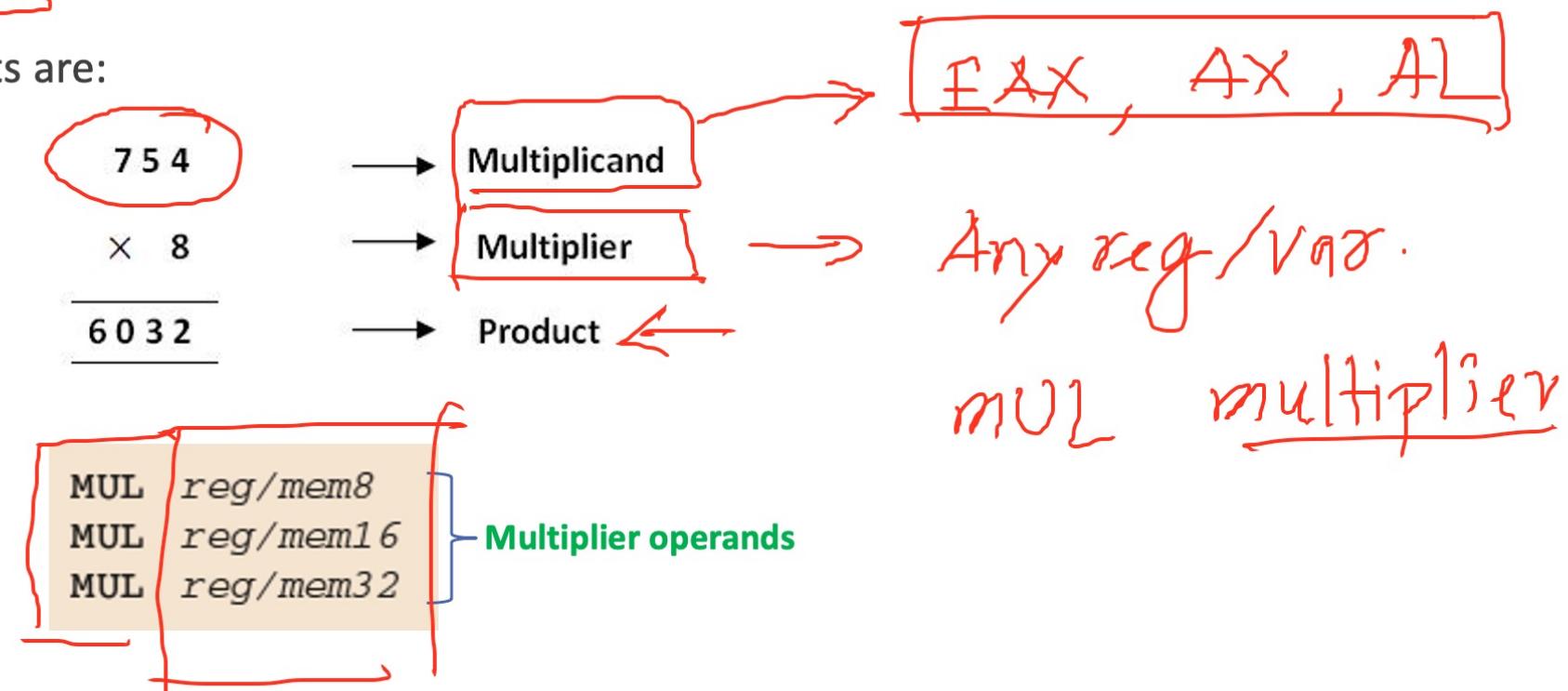
$$\begin{array}{r} 7 \ 5 \ 4 \\ \times \ 8 \\ \hline 6 \ 0 \ 3 \ 2 \end{array} \quad \begin{array}{l} \longrightarrow \text{Multiplicand} \\ \longrightarrow \text{Multiplier} \\ \longrightarrow \text{Product} \end{array}$$

MUL reg/mem8
MUL reg/mem16
MUL reg/mem32

Multiplier operands

MUL Instruction

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



MUL Instruction

- The instruction formats are:

MUL reg/mem8
MUL reg/mem16
MUL reg/mem32 }

Multiplier
operands

$$\begin{array}{r} 754 \\ \times 8 \\ \hline 6032 \end{array}$$

→ Multiplicand
→ Multiplier
→ Product

TABLE 7-2 MUL Operands.

Multiplicand	Multiplier	Product
AL	reg/mem8	AX
AX	reg/mem16	DX:AX
EAX	reg/mem32	EDX:EAX

Because the destination operand is twice the size of the multiplicand and multiplier, overflow **cannot occur**.

The colon (:) means **concatenation**. This means that **DX** are the bits 16-31 and **AX** are bits 0-15 of the input number

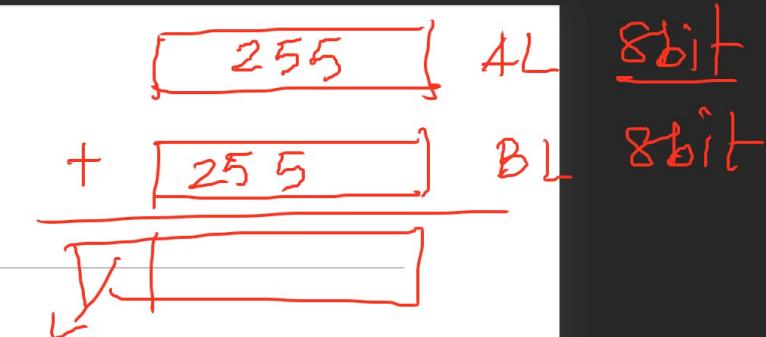
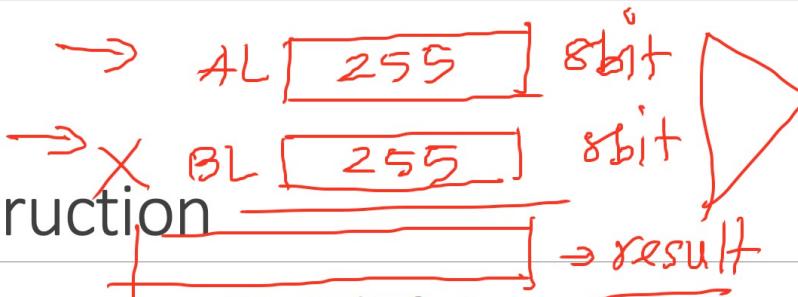
over sized data

MUL Instruction

- The instruction formats are:

MUL reg/mem8
 MUL reg/mem16
 MUL reg/mem32

Multiplier
operands



$$\begin{array}{r}
 754 \\
 \times 8 \\
 \hline
 6032
 \end{array}
 \begin{array}{l}
 \text{→ Multiplicand} \\
 \text{→ Multiplier} \\
 \text{→ Product}
 \end{array}$$

Table 7-2 MUL Operands.

Multiplicand	Multiplier	Product
AL	reg/mem8	AX
AX	reg/mem16	DX:AX
EAX	reg/mem32	EDX:EAX

Because the destination operand is twice the size of the multiplicand and multiplier, overflow cannot occur.

The colon (:) means concatenation. This means that DX are the bits 16-31 and AX are bits 0-15 of the input number

over sized data

8-bit
16-bit
32-bit

16-bit 16-bit
↓ ↓
↓ ↓
DX:AX
↓
Upper part of result
↓
Lower part of result
EDX : EAX
Upper 32 Lower 32

pter / Integer Arithmetic with annotations

windows10_fall21 [Running]

File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help Search (Ctrl+Q) Project4 MN

Process: [0x1D38] Project4.exe Lifecycle Events Thread: [0x2104] Main Thread Stack Frame: main

Source.asm X Project4.lst

```
4 ExitProcess Proto, dwExitCode:DWORD
5 .DATA
6 var1 DWORD 110000h
7 var2 DWORD 1000h
8
9 .CODE
10 main PROC
11     mov eax, var1
12     mul var2
13     INVOKE ExitProcess, 0
14 main ENDP
15 END main
```

var1 + var2

00000000 11000000

⇒ EAX * var2 → [EDX][EAX]

OV = 1 UP = 0 EI = 1 PL = 0 ZR = 0 AC = 0 PE = 1
CY = 1

Upper 32bit 32bit

EDX 00000001 10000000

EAX

Watch 1

Name	Value	Type
var2	0x00001000	unsigned long

No issues found

Error List

Entire Solution 0 Errors 0 Warnings 0 Messages Build + IntelliSense

Call Stack Error List

Add item to watch

Ready Add to Source

MUL Instruction

MUL Examples

The following statements multiply AL by BL, storing the product in AX. The Carry flag is clear (CF = 0) because AH (the upper half of the product) equals zero:

```
mov al,5h  
mov bl,10h  
mul bl
```

; AX = 0050h, CF = 0

The following diagram illustrates the movement between registers:

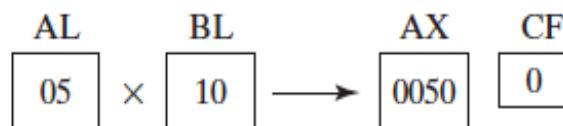


Table 7-2 MUL Operands.

Multiplicand	Multiplier	Product
AL	reg/mem8	AX
AX	reg/mem16	DX:AX
EAX	reg/mem32	EDX:EAX

The Carry flag indicates whether or not the upper half of the product contains **significant digits (?)**

Because the **destination operand** is **twice the size** of the multiplicand and multiplier, **overflow cannot occur**.

MUL: Example1

- Example1: $100h * 2000h$, unsigned 16-bit operands:

```
.data  
val1 WORD 2000h  
val2 WORD 0100h  
.code  
mov ax, val1  
mul val2
```

Table 7-2 MUL Operands.

Multiplicand	Multiplier	Product
AL	reg/mem8	AX
AX	reg/mem16	DX:AX
EAX	reg/mem32	EDX:EAX

; DX:AX = 00200000h, CF=1

Result = 200000h



Because the destination operand is twice the size of the multiplicand and multiplier, overflow cannot occur.

The Carry flag indicates whether or not the upper half of the product contains significant digits (?)

MUL: Example2

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

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

Result = 12345000h



Table 7-2 MUL Operands.

Multiplicand	Multiplier	Product
AL	reg/mem8	AX
AX	reg/mem16	DX:AX
EAX	reg/mem32	EDX:EAX

The Carry flag indicates whether or not the upper half of the product contains **significant digits (?)**

MUL: Example3

- 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
```

Result = 123400h

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

Table 7-2 MUL Operands.

Multiplicand	Multiplier	Product
AL	reg/mem8	AX
AX	reg/mem16	DX:AX
EAX	reg/mem32	EDX:EAX

You are screen sharing

File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help

Process: [0x624] Project4.exe Lifecycle Events Thread: [0x1884] Main Thread Stack Frame: main

Source.asm Project4.lst

```
4 ExitProcess Proto, dwExitCode:DWORD
5 .
6 .
7 var1 DWORD 110000h
8 var2 DWORD 1000h
9 .
10 .CODE
11 main PROC
12     mov ax, 1234h
13     mov bx, 100h
14     mul bx
15     mov cx, dx
16     rol ecx, 16
17     mov cx, ax
18
19     INVOKE ExitProcess, 0 ≤ 1ms elapsed
20 main ENDP
21 END main
```

Registers Memory 1

EAX = 00CF3400	EBX = 00B10100	ECX = 00123400
EDX = 00020012	ESI = 00021005	EDI = 00021005
EIP = 00021024	ESP = 00CFFCB4	EBP = 00CFFCC0
EFL = 00000A06		

OV = 1 UP = 0 EI = 1 PL = 0 ZR = 0 AC = 0 PE = 1
CY = 0

Watch 1

Name	Value	Type
var2	0x00001000	unsigned long

Add item to watch

Error List

Entire Solution	0 Errors	0 Warnings	0 Messages	Build + IntelliSense		
Search Error List	Code	Description	Project	File	Line	Suppression State

Call Stack Error List

Java, C, C++
(Easy way out)

MUL: Example4

- 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
```

Result = 1287650000h

Table 7-2 MUL Operands.

Multiplicand	Multiplier	Product
AL	reg/mem8	AX
AX	reg/mem16	DX:AX
EAX	reg/mem32	EDX:EAX

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

Multiplication and Division Instructions

- MUL Instruction
- **IMUL Instruction**
- DIV Instruction
- Signed Integer Division
- CBW, CWD, CDQ Instructions
- IDIV Instruction
- Implementing Arithmetic Expressions

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
- x86 instruction set supports **three formats for the IMUL instruction:**
 - One operand, two operands, and three operands.

IMUL Instruction

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

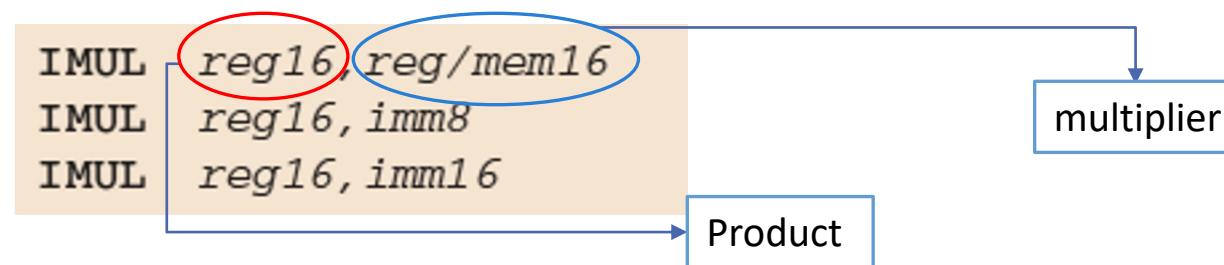
1. The **one-operand format**, the multiplier and multiplicand are the same size and the product is twice their size

IMUL reg/mem8	}	Multiplier operands	; AX = AL * reg/mem8
IMUL reg/mem16			; DX:AX = AX * reg/mem16
IMUL reg/mem32			; EDX:EAX = EAX * reg/mem32

IMUL Instruction

2. The two-operand version of the IMUL instruction in 32-bit mode

- Stores the product in the first operand, which must be a **register**
- The **second operand** (the multiplier) can be a **register**, a **memory operand**, or an **immediate value**
- Following are the **16-bit formats**:



IMUL Instruction

2. The two-operand version of the IMUL instruction in 32-bit mode

- The 32-bit operand types showing that the multiplier can be
 - a 32-bit register, a 32-bit memory operand, or an immediate value (8 or 32 bits):

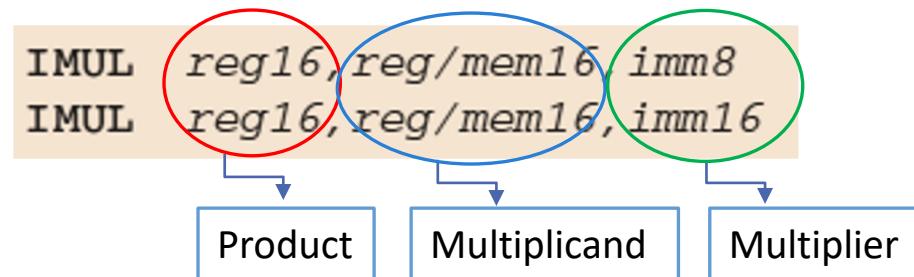
```
IMUL    reg32, reg/mem32  
IMUL    reg32, imm8  
IMUL    reg32, imm32
```

- The two-operand formats **truncate** the **product** to the length of the destination
- If significant digits are lost,
 - the **Overflow** and **Carry** flags are set
 - Be sure to check one of these flags after performing an IMUL operation with two operands

IMUL Instruction

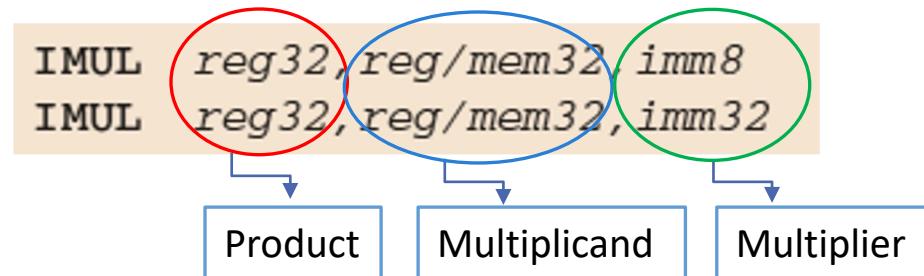
3. The three-operand formats in 32-bit mode

- Store the **product** in the first operand
- The **second operand** can be
 - a 16-bit register or memory operand, which is multiplied by the third operand,
 - the third operand can be an 8- or 16-bit immediate value:



IMUL Instruction

- A 32-bit register or memory operand can be **multiplied** by
 - an 8- or 32-bit immediate value:



- **Truncate** the **product** to the length of the destination, If significant digits are lost,
 - the **Overflow** and **Carry** flags are set
 - Be sure to check one of these flags after performing an **IMUL operation** with two operands

IMUL: Example1

- The following instructions demonstrate **one-operand formats**:
- multiply $48 * 4$, using **8-bit** operands:

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

IMUL *reg/mem8*
IMUL *reg/mem16*
IMUL *reg/mem32*

Multiplier operands

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

Preserves the sign of the
product by sign-extending it
into the upper half of the
destination register

IMUL: Example2

- The following instructions demonstrate **one-operand formats**:
- Multiply 4,823,424 * -423:

```
mov eax,4823424  
mov ebx,-423  
imul ebx          ; EDX:EAX = FFFFFFFF86635D80h, OF=0
```

IMUL *reg/mem8*
IMUL *reg/mem16*
IMUL *reg/mem32*

Multiplier operands

OF=0 because EDX is a sign extension of EAX.

Preserves the sign of the
product by sign-extending it
into the upper half of the
destination register

IMUL: Example3

- The following instructions demonstrate **one-operand formats**:
- 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
```

IMUL *reg/mem8*
IMUL *reg/mem16*
IMUL *reg/mem32*

Multiplier operands

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

DX is not a sign extension

Product = 876000h

Preserves the sign of the **product** by sign-extending it into the upper half of the destination register

IMUL: Example4

- The following instructions demonstrate **two-operand formats**:

```
.data  
word1 SWORD 4  
dword1 SDWORD 4  
.code  
mov ax,-16      ; AX = -16  
mov bx,2        ; BX = 2  
imul bx,ax    ; BX = -32  
imul bx,2      ; BX = -64  
imul bx,word1 ; BX = -256
```

```
mov eax,-16      ; EAX = -16  
mov ebx,2        ; EBX = 2  
imul ebx,eax  ; EBX = -32  
imul ebx,2      ; EBX = -64  
imul ebx,dword1 ; EBX = -256
```

IMUL *reg16 reg/mem16*
IMUL *reg16, imm8*
IMUL *reg16, imm16*

Product

IMUL *reg32, reg/mem32*
IMUL *reg32, imm8*
IMUL *reg32, imm32*

IMUL: Example5

- The following instructions demonstrate **two-operand formats**:

```
mov ax,-32000  
imul ax,2      ; OF = 1
```

IMUL *reg16, reg/mem16*
IMUL *reg16, imm8*
IMUL *reg16, imm16*

- signed overflow** because **64,000 cannot fit within the 16-bit destination operand**

$2^{16} = 65536$ (UNSIGNED) : 0 to 65535

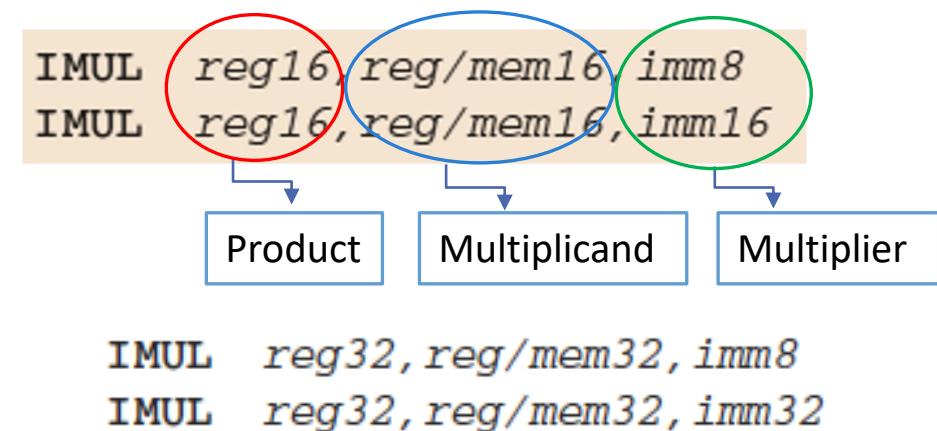
$2^{15} = 32768$ (SIGNED) : -1 to -32768 and 0 to 32767

- The **two-operand** and **three-operand IMUL instructions** use a destination operand that is **the same size as the multiplier**.
- Therefore, it is possible for **signed overflow to occur**.

IMUL: Example6

- The following instructions demonstrate **three-operand formats**:

```
.data  
word1 SWORD 4  
dword1 SDWORD 4  
.code  
imul bx,word1,-16      ; BX = word1 * -16  
imul ebx,dword1,-16    ; EBX = dword1 * -16  
imul ebx,dword1,-2000000000 ; signed overflow!
```



- The **two-operand** and **three-operand** **IMUL** instructions use a destination operand that is the same size as the **multiplier**.
- Therefore, it is possible for **signed overflow to occur**.

Multiplication and Division Instructions

- MUL Instruction
- IMUL Instruction
- **DIV Instruction**
- Signed Integer Division
- CBW, CWD, CDQ Instructions
- IDIV Instruction
- Implementing Arithmetic Expressions

DIV Instruction

- The DIV (**unsigned** divide) instruction performs 8-bit, 16-bit, and 32-bit division on **unsigned** integers

$$\begin{array}{r} & \begin{array}{c} 7 \leftarrow \text{quotient} \\ \hline \end{array} \\ \text{divisor} \rightarrow 5 & \begin{array}{c} | \\ 37 \leftarrow \text{dividend} \\ - \\ \hline \end{array} \\ & \begin{array}{c} 35 \\ - \\ \hline \end{array} \\ & \begin{array}{c} 2 \leftarrow \text{remainder} \\ \hline \end{array} \end{array}$$

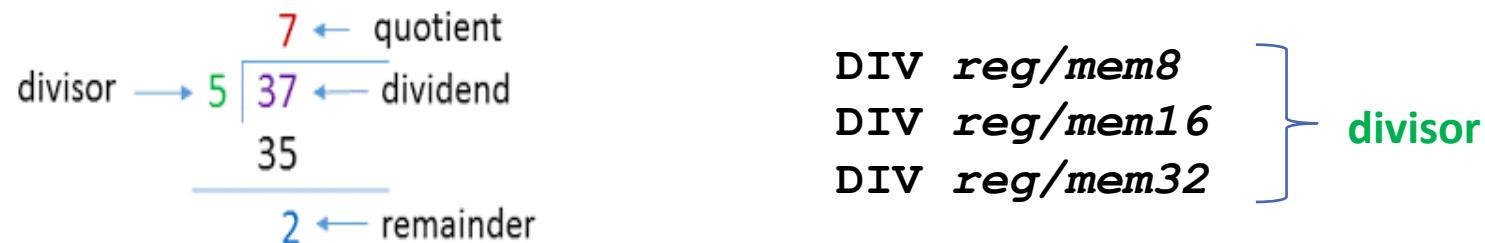
- A **single operand** is supplied (register or memory operand), which is assumed to be the **divisor**
- Instruction formats:

DIV reg/mem8
DIV reg/mem16
DIV reg/mem32

} divisor

DIV Instruction

- The DIV (unsigned divide) instruction performs 8-bit, 16-bit, and 32-bit division on unsigned integers

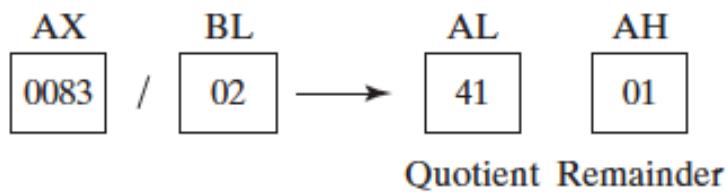


Dividend	Divisor	Quotient	Remainder
AX	r/m8	AL	AH
DX:r/AX	r/m16	AX	DX
EDX:EAX	r/m32	EAX	EDX

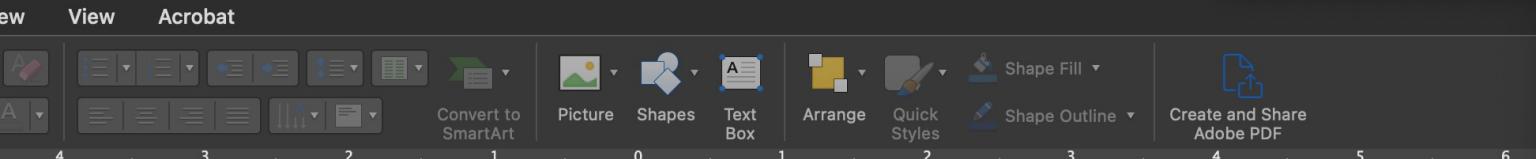
DIV: Example1

- The following instructions perform 8-bit unsigned division ($83h/2$),
 - producing a **quotient** of $41h$
 - and a **remainder** of 1

```
mov ax,0083h      ; dividend  
mov bl,2          ; divisor  
div bl           ; AL = 41h, AH = 01h
```



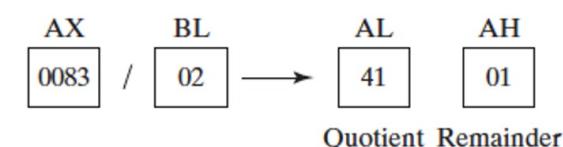
Dividend	Divisor	Quotient	Remainder
AX	<i>r/m8</i>	AL	AH
DX:AX	<i>r/m16</i>	AX	DX
EDX:EAX	<i>r/m32</i>	EAX	EDX



Example1

following instructions perform 8-bit unsigned division ($83h/2$),
producing a **quotient** of $41h$
and a **remainder** of 1

```
mov ax,0083h      ; dividend
mov bl,2          ; divisor
div bl           ; AL = 41h, AH = 01h
```



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

File Edit View Git Project Build Debug Test Analyze Tools

Process: [0x1D5C] Project4.exe Lifecycle Events Thread: [0xEF0] Ma

Source.asm* Project4.lst

```

4 ExitProcess Proto, dwExitCode:DWORD
5 .DATA
6
7 var1 DWORD 110000h
8 var2 DWORD 1000h
9
10 .CODE
11 main PROC
12     mov ax, 83h
13     mov dx,0
14     mov bx, 2
15     div bx
16
17     INVOKE ExitProcess, 0    ≤ 1ms elapsed
18 main ENDP
19 END main
20
21

```

120% No issues found

Watch 1

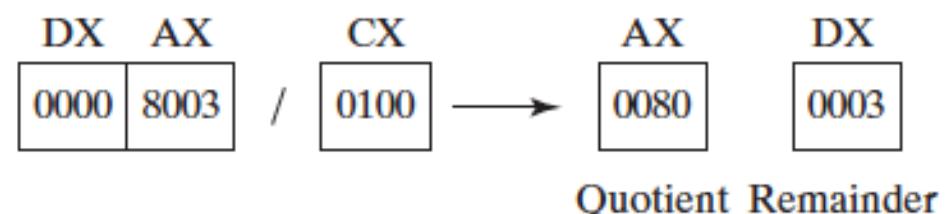
Search (Ctrl+E) Search Depth: 3

Name	Value
var2	0x00001000
Add item to watch	

DIV: Example2

- 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
```



Dividend	Divisor	Quotient	Reminder
AX	<i>r/m8</i>	AL	AH
DX:AX	<i>r/m16</i>	AX	DX
EDX:EAX	<i>r/m32</i>	EAX	EDX

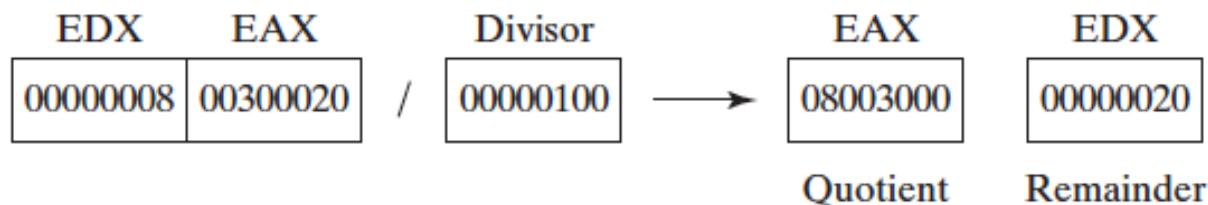
DIV: Example3

- Same division, using 32-bit operands:

000800300020h

```
mov edx,0008h           ; dividend, high  
mov eax,00300020h       ; dividend, low  
mov ecx,100h            ; divisor  
div ecx                 ; EAX = , DX =
```

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



DIV: Example4

- 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:

00876000h

```
mov dx,0087h  
mov ax,6000h  
mov bx,100h  
div bx           Quotient = 8760, Rem = 0  
  
DX = , AX =
```

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

Use Visual Studio

DIV: Example5

- 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, 6002h  
mov bx, 10h  
div bx
```

Answer: Divide Overflow

If a division operand produces a quotient that will not fit into the destination operand, a divide overflow condition results

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

- All arithmetic status flag values are undefined after executing DIV and IDIV

DIV: Division by Zero

- To prevent division by zero, **test the divisor** before dividing:

```
mov ax,dividend  
mov bl,divisor  
cmp bl,0          ; check the divisor  
je NoDivideZero ; zero? display error  
div bl           ; not zero: continue  
  
. .  
NoDivideZero:    ;(display "Attempt to divide by zero")
```

Design Tra

File Edit View Git Project Build Debug Test Analyze Tools Extensions Window Help Search (Ctrl+Q) Project4

Layout Reset Section

Server Explorer Toolbox

Source.asm Project4.lst

Debug x86 Local Windows Debugger

Output

Build started...
----- Build started: Project: Project4, Configuration: I
Assembling Source.asm...
Project4.vcxproj -> C:\Users\Zulkar\Source\Repos\Project4\Project4\Source.asm
===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0

4 ExitProcess Proto, dwExitCode:DWORD

5 .DATA

6

7 var1 DWORD 110000h

8 var2 DWORD 1000h

9

10 .CODE

11 main PROC

12 mov ax, 83h

13 mov dx, 0

14 mov bx, 0

15 cmp bx, 0

16 JE Next

17

18 div bx

19

20 Next:

21

22 INVOKE ExitProcess, 0

23 main ENDP

24 END main

25

26

Java^o
toy

if(bx ≠ 0)
 ↓
 {
 ↑
 }

catch,

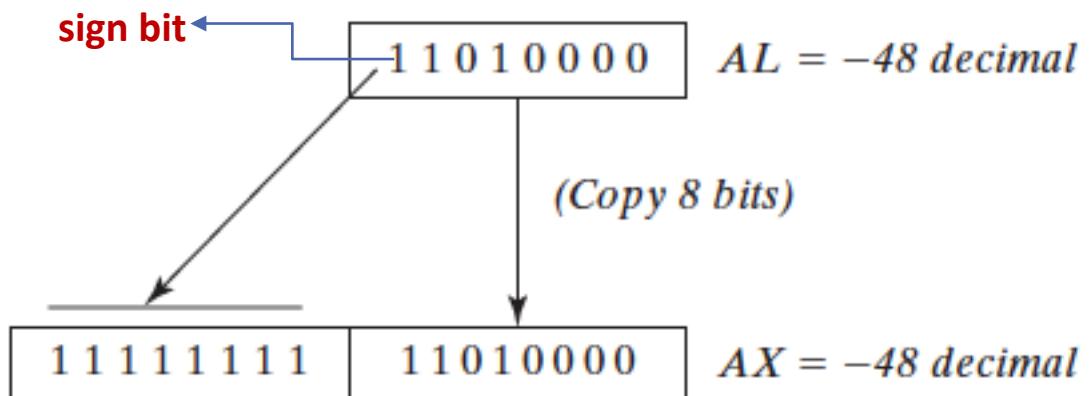
120 % No issues found Ln: 20 Ch: 3 Col: 6 TABS CRLF

Multiplication and Division Instructions

- MUL Instruction
- IMUL Instruction
- DIV Instruction
- **Signed Integer Division**
- CBW, CWD, CDQ Instructions
- IDIV Instruction
- Implementing Arithmetic Expressions

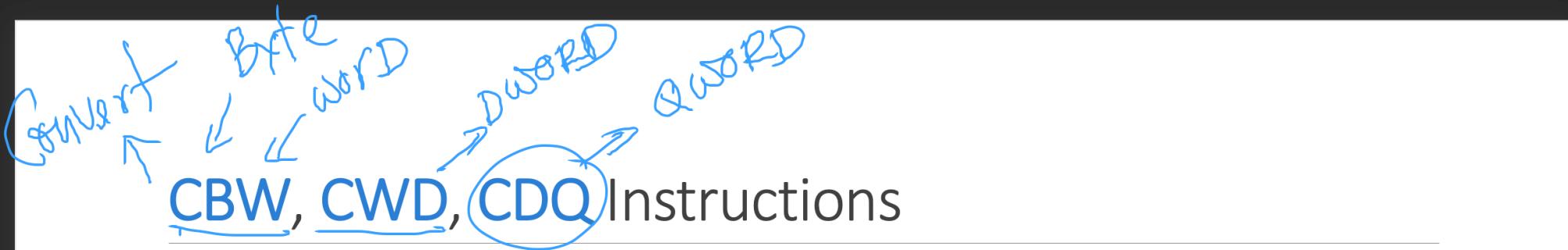
Signed Integer Division (IDIV)

- 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
- Example: The high byte contains a copy of the sign bit from the low byte:



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

How to do sign-extension 🤔?



- 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

- Example:**

```
.data
dwordVal SDWORD -101      ; FFFFFFF9Bh
.code
mov eax,dwordVal
cdq
```

Annotations: "dwordVal" is highlighted in red; "SDWORD" is highlighted in blue; "r/m8" is highlighted in blue; "r/m16" is highlighted in blue; "r/m32" is highlighted in blue; "EDX:EAX" is highlighted in blue; "FFFFFFFFFF9Bh" is highlighted in red; "EDX:EAX = FFFFFFFF9Bh" is highlighted in red.

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



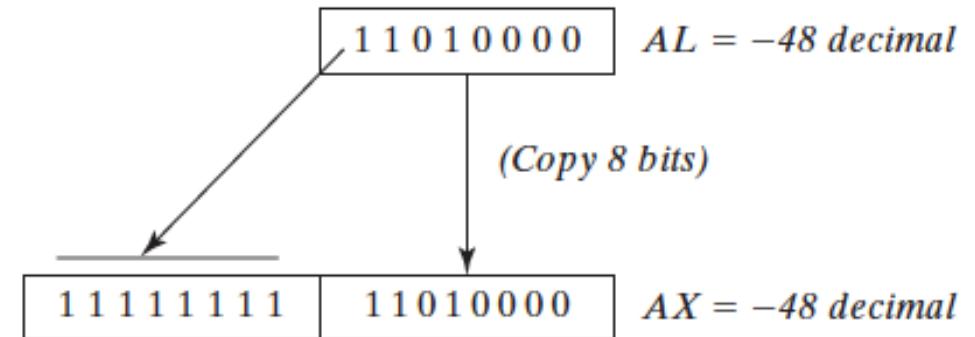
← 32bit

IDIV Instruction

- **IDIV** (signed divide) performs signed integer division
- Same syntax and operands as **DIV** instruction
- **Example1:** 8-bit division of **-48** by 5

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

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



IDIV Examples

- **Example2:** 16-bit division of -48 by 5

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

- **Example3:** 32-bit division of -48 by 5

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

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

IDIV Examples

- **Example4:** 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)

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

Unsigned Arithmetic Expressions

- Some good reasons to learn how to implement integer expressions:
 - Learn how do compilers do it
 - Test your understanding of **MUL**, **IMUL**, **DIV**, **IDIV**
 - Check for overflow (**Carry** and **Overflow** flags)

Unsigned Arithmetic Expressions

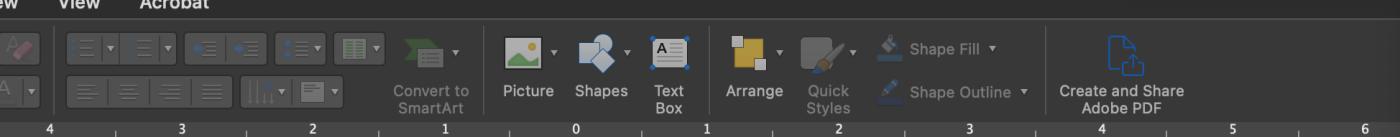
- Example1: `var4 = (var1 + var2) * var3`

; Assume **unsigned** operands

Table 7-2 MUL Operands.

Multiplicand	Multiplier	Product
AL	reg/mem8	AX
AX	reg/mem16	DX:AX
EAX	reg/mem32	EDX:EAX

A good reason for checking the Carry flag after executing MUL is to know whether the upper half of the product can safely be ignored.



Signed Arithmetic Expressions

example1: $\text{var4} = (\text{var1} + \text{var2}) * \text{var3}$

; Assume **unsigned** operands

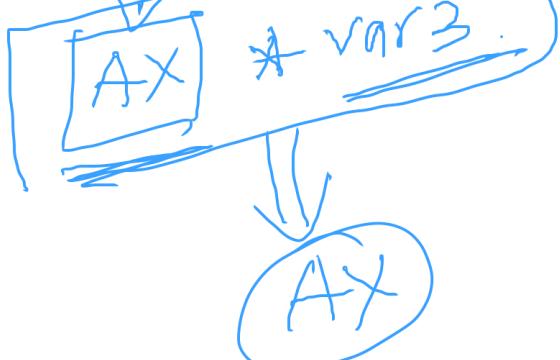


Table 7-2 MUL Operands.

Multiplicand	Multiplier	Product
AL	reg/mem8	AX
AX	reg/mem16	DX:AX
EAX	reg/mem32	EDX:EAX

A good reason **for checking the Carry flag after executing MUL** is to know whether the upper half of the product can safely be ignored.

The screenshot shows Microsoft Visual Studio with an assembly file named 'Source.asm'. The code is as follows:

```

4 ExitProcess Proto, dwExitCode:DWORD
5 .
6 .
7 var1 WORD 1100h
8 var2 WORD 1000h
9 var3 WORD 2000h
10 var4 WORD 3000h
11 .
12 main PROC
13     mov ax, var1
14     add ax, var2
15     mul var3
16     mov var4, ax
17 .
18     INVOKE ExitProcess, 0
19 main ENDP
20 END main
21 .
22 .
23

```

To the right of the assembly code, there is handwritten text: 'AX * var3' with an arrow pointing down to 'DX:AX' and a box around 'AX'.

Unsigned Arithmetic Expressions

- Example1: `var4 = (var1 + var2) * var3`

; Assume **unsigned** operands

```
mov eax,var1  
add eax,var2      ; EAX = var1 + var2  
mul var3         ; EAX = EAX * var3  
jc TooBig        ; check for carry  
mov var4,eax     ; save product
```

Table 7-2 MUL Operands.

Multiplicand	Multiplier	Product
AL	reg/mem8	AX
AX	reg/mem16	DX:AX
EAX	reg/mem32	EDX:EAX

A good reason for checking the Carry flag after executing MUL is to know whether the upper half of the product can safely be ignored.

Unsigned Arithmetic Expressions

- Example2: `eax = (-var1 * var2) + var3`

; Assume **signed** operands

Table 7-2 MUL Operands.

Multiplicand	Multiplier	Product
AL	reg/mem8	AX
AX	reg/mem16	DX:AX
EAX	reg/mem32	EDX:EAX

Unsigned Arithmetic Expressions

- Example2: `eax = (-var1 * var2) + var3`

; Assume **signed** operands

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

Table 7-2 MUL Operands.

Multiplicand	Multiplier	Product
AL	reg/mem8	AX
AX	reg/mem16	DX:AX
EAX	reg/mem32	EDX:EAX

Unsigned Arithmetic Expressions

- Example3: `var4 = (var1 * 5) / (var2 - 3)`

Do not modify any variables other than var4:

; Assume **unsigned** operands

Table 7-2 MUL Operands.

Multiplicand	Multiplier	Product
AL	reg/mem8	AX
AX	reg/mem16	DX:AX
EAX	reg/mem32	EDX:EAX

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

Unsigned Arithmetic Expressions

- Example3: $\text{var4} = (\text{var1} * 5) / (\text{var2} - 3)$

Do not modify any variables other than var4:

; Assume **unsigned** operands

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

Table 7-2 MUL Operands.

Multiplicand	Multiplier	Product
AL	reg/mem8	AX
AX	reg/mem16	DX:AX
EAX	reg/mem32	EDX:EAX

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

Var1=-2
Var2=20
Var3=2

Unsigned Arithmetic Expressions

- Example4: `var4 = (var1 * -5) / (-var2 / var3);`
; assume signed 32-bit integers

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

Table 7-2 MUL Operands.

Multiplicand	Multiplier	Product
AL	reg/mem8	AX
AX	reg/mem16	DX:AX
EAX	reg/mem32	EDX:EAX

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

Var1=-2
Var2=20
Var3=2

Unsigned Arithmetic Expressions

- Example4: `var4 = (var1 * -5) / (-var2 / var3);`

; assume signed 32-bit integers

```
mov eax,var2
neg eax
cdq
idiv var3
mov ebx,eax
mov eax,-5
imul var1
idiv ebx
mov var4,eax
```

; begin right side

; sign-extend dividend
; EDX = remainder
; EBX = right side
; begin left side
; EDX:EAX = left side
; final division
; quotient

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

Table 7-2 MUL Operands.

Multiplicand	Multiplier	Product
AL	reg/mem8	AX
AX	reg/mem16	DX:AX
EAX	reg/mem32	EDX:EAX

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

Unsigned Arithmetic Expressions

- **Example5:** Implement the following expression using **signed 32-bit integers**.
- Do not modify any variables other than var3:

$$\text{var3} = (\text{var1} * -\text{var2}) / (\text{var3} - \text{ebx})$$

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

Table 7-2 MUL Operands.

Multiplicand	Multiplier	Product
AL	reg/mem8	AX
AX	reg/mem16	DX:AX
EAX	reg/mem32	EDX:EAX

Unsigned Arithmetic Expressions

- **Example5:** Implement the following expression using **signed 32-bit integers**.
- Do not modify any variables other than var3:

$$\text{var3} = (\text{var1} * -\text{var2}) / (\text{var3} - \text{ebx})$$

```
mov eax,var1  
mov edx,var2  
neg edx  
imul edx ; left side: EDX:EAX  
mov ecx,var3  
sub ecx,ebx  
idiv ecx ; EAX = quotient  
mov var3,eax
```

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

Table 7-2 MUL Operands.

Multiplicand	Multiplier	Product
AL	reg/mem8	AX
AX	reg/mem16	DX:AX
EAX	reg/mem32	EDX:EAX

Outline

- Shift and Rotate Instructions
- Shift and Rotate Applications
- Multiplication and Division Instructions
- Extended Addition and Subtraction **See the book**
- ASCII and Unpacked Decimal Arithmetic **See the book**
- Packed Decimal Arithmetic **See the book**

Extended Addition and Subtraction

- Extended precision **addition and subtraction** is the technique of
 - adding and subtracting numbers having **an almost unlimited size**

```
mov al,0FFh  
add al,0FFh      ; AL = FEh  😳
```

- In assembly language, the **ADC** instruction is well suited to this type of operation
- How about subtraction?

```
mov eax,1  
sub eax,2      ;EAX= ?
```

- the **SBB** instruction is well suited to this type of operation