



CHAPTER 7

Integer Arithmetic Part2

➤ 7.4: Multiplication and Division Instructions

✚ MUL Instruction

- The MUL (unsigned multiply) instruction comes in three versions:
 - The first version multiplies an 8-bit operand by the AL register.
 - The second version multiplies a 16-bit operand by the AX register.
 - The third version multiplies a 32-bit operand by the EAX register.
- The multiplier and multiplicand must always be the same size, and the product is twice their size.
- The three formats accept register and memory operands, but not immediate operands:
 - MUL *reg/mem8*
 - MUL *reg/mem16*
 - MUL *reg/mem32*

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

- MUL sets the Carry and Overflow flags if the upper half of the product is not equal to zero.

✚ IMUL Instruction

- ❖ The IMUL instruction is used for signed multiplication
 - ✧ Preserves the sign of the product by sign-extending it.

- ❖ One-Operand formats, as in MUL
 - **IMUL r/m8** ; **AX = AL * r/m8**
 - **IMUL r/m16** ; **DX:AX = AX * r/m16**
 - **IMUL r/m32** ; **EDX:EAX = EAX * r/m32**
- ❖ Two-Operand formats:
 - **IMUL r16, r16/m16/imm8/imm16**
 - **IMUL r32, r32/m32/imm8/imm32**
- ❖ Three-Operand formats:
 - **IMUL r16, r16/m16, imm8/imm16**
 - **IMUL r32, r32/m32, imm8/imm32**

- The Carry and Overflow flags are set if the upper half of the product is not a sign extension of the lower half.

- Example1:

```
mov al, 48
mov bl, 4
imul bl ; AX = 00C0h (decimal +192), OF = CF = 1
```

Because AH is not a sign extension of AL, the Overflow flag is set to 1.

- Example2:

```
mov ax, 48
mov bx, 4
imul bx ; DX:AX = 0000h:00C0h, OF = CF = 0
```

Here, DX is a sign extension of AX, so the Overflow flag is cleared.

- Example3:

```
mov al, -4
mov bl, 4
imul bl ; AX = FFF0h, CF = OF = 0
```

AH is a sign extension of AL, therefore the Overflow flag is cleared.

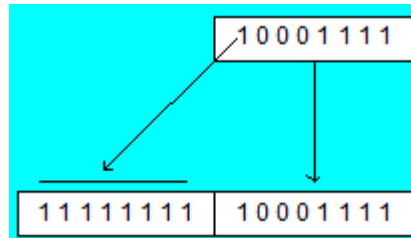
DIV Instruction

- The DIV (unsigned divide) instruction performs 8-bit, 16-bit, and 32-bit unsigned integer division.
- The single register or memory operand is the divisor.
- The formats are
 - DIV *reg/mem8*
 - DIV *reg/mem16*
 - DIV *reg/mem32*

Dividend	Divisor	Quotient	Remainder
AX	reg/mem8	AL	AH
DX:AX	reg/mem16	AX	DX
EDX:EAX	reg/mem32	EAX	EDX

✚ Signed Integer Division

- The dividend must be fully sign-extended before the division takes place.
 - Fill high byte, word, or double-word with a copy of the low byte/word/double word's sign bit.
 - Then we will apply them to the signed integer divide instruction, IDIV.
 - For Example:



✚ Sign Extension Instructions (CBW, CWD, CDQ)

- Provide important sign-extension operations before division
- **CBW**: Convert Byte to Word, sign-extends AL into AH
- **CWD**: Convert Word to Double, sign-extends AX into DX
- **CDQ**: Convert Double to Quad, sign-extends EAX into EDX

✚ IDIV Instruction

- The IDIV (signed divide) instruction performs signed integer division, using the same operands as DIV.
- Before executing 8-bit division, the dividend (AX) must be completely sign-extended.
- The remainder always has the same sign as the dividend.

✚ Divide Overflow

- Divide Overflow occurs when ...
 - Quotient cannot fit into the destination operand, or when
 - Dividing by Zero.
- Divide Overflow causes a CPU interrupt
 - The current program halts and an error dialog box is produced
- For Example:

```
mov ax,1000h
mov bl,10h
div bl      ; AL cannot hold 100h
```
- All status flags are undefined after executing DIV and IDIV

7.4.7 Section Review

- (7.4.7 p1) Explain why overflow cannot occur when the MUL and one-operand IMUL instructions execute.

Ans.:

The product is stored in registers that are twice the size of the multiplier and multiplicand. If you multiply 0FFh by 0FFh, for example, the product (FE01h) easily fits within 16 bits.

- (7.4.7 p2) How is the one-operand IMUL instruction different from MUL in the way it generates a multiplication product?

Ans.:

When the product fits completely within the lower register of the product, IMUL sign extends the product into the upper product register. MUL, on the other hand, zero-extends the product.

- (7.4.7 p3) What has to happen in order for the one-operand IMUL to set the Carry and Overflow flags?

Ans.:

With IMUL, the Carry and Overflow flags are set when the upper half of the product is not a sign extension of the lower half of the product.

- (7.4.7 p4) When EBX is the operand in a DIV instruction, which register holds the quotient?

Ans.:

EAX

- (7.4.7 p5) When BX is the operand in a DIV instruction, which register holds the quotient?

Ans.:

AX

- (7.4.7 p6) When BL is the operand in a MUL instruction, which registers hold the product?

Ans.:

AX.

- (7.4.7 p7) Show an example of sign extension before calling the IDIV instruction with a 16-bit operand.

Ans.:

Code example:

```

mov ax,dividendLow
cwd                ; sign-extend dividend
mov bx,divisor
idiv bx

```

- (7.4.7 p8) What will be the contents of AX and DX after the following operation?

```

mov dx,0
mov ax,222h
mov cx,100h
mul cx

```

Ans.:

DX = 0002h, AX = 2200h.

- (7.4.7 p9) What will be the contents of AX after the following operation?

```

mov ax,63h
mov bl,10h
div bl

```

Ans.:

AX = 0306h.

- (7.4.7 p10) What will be the contents of EAX and EDX after the following operation?

```

mov eax,123400h
mov edx,0
mov ebx,10h
div ebx

```

Ans.:

EDX = 0, EAX = 00012340h.

- (7.4.7 p11) What will be the contents of AX and DX after the following operation?

```

mov ax,4000h
mov dx,500h
mov bx,10h
div bx

```

Ans.:

The DIV will cause a divide overflow, so the values of AX and DX cannot be determined.

- (7.4.7 p12) Write instructions that multiply -5 by 3 and store the result in a 16-bit variable `val1`.

Ans.:

```

mov al,3
mov bl,-5
imul bl
mov val1,ax ; product

```

- (7.4.7 p13) Write instructions that divide -276 by 10 and store the result in a 16-bit variable **val1**.

Ans.:

```
mov ax,-276
cwd ; sign-extend AX into DX
mov bx,10
idiv bx
mov val1,ax ; quotient
```

- (7.4.7 p14) Implement the following C++ expression in assembly language, using 32-bit unsigned operands:

$$\text{val1} = (\text{val2} * \text{val3}) / (\text{val4} - 3)$$

Ans.:

```
mov eax,val2
mul val3
mov ebx,val4
sub ebx,3
div ebx
mov val1,eax
```

- (7.4.7 p15) Implement the following C++ expression in assembly language, using 32-bit signed operands:

$$\text{val1} = (\text{val2} / \text{val3}) * (\text{val1} + \text{val2})$$

Ans.:

```
mov eax,val2
cdq ; extend EAX into EDX
idiv val3 ; EAX = quotient
mov ebx,val1
add ebx,val2
imul ebx
mov val1,eax ; lower 32 bits of product
```

Extended Addition and Subtraction

- Extended precision addition and subtraction is adding and subtracting numbers having an almost unlimited size.

➤ ADC Instruction

- ADC (add with carry) instruction adds both a source operand and the contents of the Carry flag to a destination operand.
- The instruction formats are the same as for the ADD instruction, and the operands must be the same size:
 - ✓ ADC reg,reg
 - ✓ ADC mem,reg
 - ✓ ADC reg,mem
 - ✓ ADC mem,imm
 - ✓ ADC reg,imm

- Example: Add two 32-bit integers (FFFFFFFFh + FFFFFFFFh), producing a 64-bit sum in EDX:EAX:

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

- **Task:** Add 1 to EDX:EAX
 - Starting value of EDX:EAX: 00000000FFFFFFFFh
 - Add the lower 32 bits first, setting the Carry flag.
 - Add the upper 32 bits, and include the Carry flag.

```
mov edx,0      ; set upper half
mov eax,FFFFFFFFh ; set lower half
add eax,1      ; add lower half
adc edx,0      ; add upper half
               ;EDX:EAX = 00000001 00000000
```

➤ SBB Instruction

- The SBB (subtract with borrow) instruction subtracts both a source operand and the value of the Carry flag from a destination operand.
- Operand syntax:
 - Same as for the ADC instruction.
- Task: Subtract 1 from EDX:EAX
 - Starting value of EDX:EAX: 00000000100000000h
 - Subtract the lower 32 bits first, setting the Carry flag.
 - Subtract the upper 32 bits, and include the Carry flag.

```
mov edx,1      ; set upper half
mov eax,0      ; set lower half
sub eax,1      ; subtract lower half
sbb edx,0      ; subtract upper half
               ;EDX:EAX = 00000000 FFFFFFFF
```

🚦 7.5.4 Section Review

- (7.5.4 p1) Describe the ADC instruction.

Ans.:

The ADC instruction adds both a source operand and the Carry flag to a destination operand.

- (7.5.4 p2) Describe the SBB instruction.

Ans.:

The SBB instruction subtracts both a source operand and the Carry flag from a destination operand.

- (7.5.4 p3) What will be the values of EDX:EAX after the following instructions execute?

```
mov edx,10h
mov eax,0A0000000h
add eax,20000000h
adc edx,0
```

Ans.:

EAX = C0000000h, EDX = 00000010h.

- (7.5.4 p4) What will be the values of EDX:EAX after the following instructions execute?

```
mov edx,100h
mov eax,80000000h
sub eax,90000000h
sbb edx,0
```

Ans.:

EAX = F0000000h, EDX = 000000FFh.

- (7.5.4 p5) What will be the contents of DX after the following instructions execute (STC sets the Carry flag)?

```
mov dx,5
stc          ; set Carry flag
mov ax,10h
adc dx,ax
```

Ans.:

DX = 0016h.

- (7.5.4 p6) The following program is supposed to subtract **val2** from **val1**. Find and correct all logic errors (CLC clears the Carry flag):

```
.data
    val1 QWORD 20403004362047A1h
    val2 QWORD 055210304A2630B2h
    result QWORD 0
.code
    mov cx,8          ; loop counter
    mov esi,val1      ; set index to start
    mov edi,val2
    clc               ; clear Carry flag
top:
    mov al,BYTE PTR[esi] ; get first number
```



```
sbb al,BYTE PTR[edi] ; subtract second
mov BYTE PTR[esi],al ; store the result
dec esi
dec edi
loop top
```

Ans.:

In correcting this example, it is easiest to reduce the number of instructions. You can use a single register (ESI) to index into all three variables. ESI should be set to zero before the loop because the integers are stored in little endian order with their low-order bytes occurring first:

```
mov ecx,8 ; loop counter
mov esi,0 ; use the same index reg
clc ; clear Carry flag
top:
mov al,byte ptr val1[esi] ; get first number
sbb al,byte ptr val2[esi] ; subtract second
mov byte ptr result[esi],al ; store the result
inc esi ; move to next pair
loop top
```

Of course, you could easily reduce the number of loop iterations by adding doublewords rather than bytes.

Quiz Next Week