Lab 10 - Integer Arithmetic Advanced

10.1 MUL Instruction

In 32-bit mode, 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.

```
MUL reg/mem8        ; AL  * reg/mem8
MUL reg/mem16       ; AX  * reg/mem16
MUL reg/mem32       ; EAX * reg/mem32
```

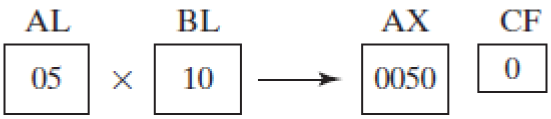
Flags: Because the destination operand is twice the size of the multiplicand and multiplier, overflow cannot occur. MUL sets the Carry and Overflow flags if the upper half of the product is not equal to zero. The Carry flag is ordinarily used for unsigned arithmetic, so we'll only focus on it while using the MUL instruction.

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

Example 1:

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:

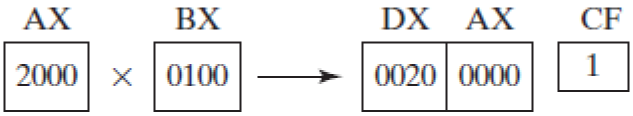
```
Include Irvine32.inc
.code
main proc
      mov AL,5h
      mov BL,10h
      mul BL              ; AX = 0050h, CF = 0
exit
main endp
end main
```



Example 2:

The following statements multiply the 16-bit value 2000h by 0100h. The Carry flag is set because the upper part of the product (located in DX) is not equal to zero:

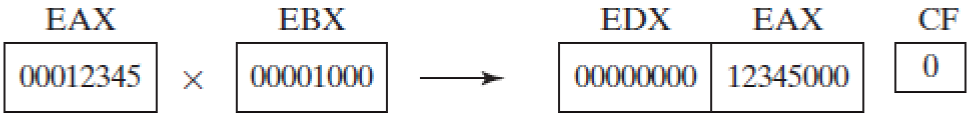
```
Include Irvine32.inc
.data
        val1 WORD 2000h
        val2 WORD 0100h
.code
main proc
        mov AX,val1          ; AX    = 2000h
        mul val2             ; DX:AX = 00200000h, CF = 1
exit
main endp
end main
```



Example 3:

The following statements multiply 12345h by 1000h, producing a 64-bit product in the combined EDX and EAX registers. The Carry flag is clear because the upper half of the product in EDX equals zero:

```
Include Irvine32.inc
.code
main proc
        mov EAX,12345h
        mov EBX,1000h
        mul EBX              ; EDX:EAX = 0000000012345000h, CF = 0
exit
main endp
end main
```



Example 4 (64-bit Mode):

A 64-bit register, or memory operand is multiplied against RDX, producing a 128-bit product in RDX:RAX.

```
ExitProcess proto
.code
main proc
        mov RAX,0FFFF0000FFFF0000h
        mov RBX,2
        mul RBX              ; RDX:RAX = 0000000000000001FFFE0001FFFE0000
call  ExitProcess
main endp
end
```

Example 5 (64-bit Mode):

```
ExitProcess proto
.data
      multiplier QWORD 10h
.code
main proc
      mov RAX,0AABBBBCCCCDDDDh
      mul multiplier     ; RDX:RAX = 00000000000000000AABBBBCCCCDDDD0h
call  ExitProcess
main endp
end
```

Example 6 – Showing the multiplication Results on Console:

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.

```
Include Irvine32.inc
.code
main PROC
;8-bit operand multiplication (Answer goes into AX)
      mov AL,150
      mov BL,2
      mul BL
      JC ax_print
      movzx EAX,AL
      call WriteInt
      call Crlf
      JMP done
ax_print:
      movzx EAX,AX
      call WriteInt
      call Crlf
done:

;16-bit operand multiplication (Answer goes into DX:AX)
      mov AX,60000
      mov BX,2
      mul BX
      JC dxax_print
      movzx EAX,AX
      call WriteInt
      call Crlf
      JMP done2
dxax_print:
      SHL EDX,16
      SHLD DX,AX,16
      mov EAX,EDX
      call WriteInt
      call Crlf
done2:
      exit
main ENDP
END main
```

10.2 IMUL Instruction

The IMUL (signed multiply) instruction performs signed integer multiplication. Unlike the MUL instruction, IMUL preserves the sign of the product. It does this by sign extending the highest bit of the lower half of the product into the upper bits of the product. The x86 instruction set supports three formats for the IMUL instruction:

10.2.1 Single-Operand Formats

The one-operand formats store the product in AX, DX:AX, or EDX:EAX same as the MUL instruction:

```
IMUL reg/mem8      ;      AX = AL * reg/mem8
IMUL reg/mem16     ;   DX:AX = AX * reg/mem16
IMUL reg/mem32     ; EDX:EAX = EAX * reg/mem32
```

Flags: The Carry and Overflow flags are set same as the MUL instruction does. Also, the Carry and Overflow flags are set if the upper half of the product is not a sign extension of the lower half. You can use this information to decide whether to ignore the upper half of the product.

```
Include Irvine32.inc
.code
main proc
      mov AL,80h          ; -128
      mov BL,2
      imul BL             ; AX = FF00 (-256), CF = 1, OF = 1, SF = 1

      mov AL,0FBh         ; -5
      mov BL,2
      imul BL             ; AX = FFF6 (-10), CF = 0, OF = 0, SF = 1

      mov AX,0E890h       ; -6000
      mov BX,2
      imul BX             ; DX:AX = FFFFD120 (-12000), CF = 0, OF = 0, SF = 1

      mov AX,0E890h       ; -6000
      mov BX,0FFFEh       ; -1
      imul BX             ; DX:AX = 00001770 (+6000), CF = 0, OF = 0, SF = 0

      mov AX,0E890h       ; -6000
      mov BX,0E890h       ; -6000
      imul BX             ; DX:AX = 02255100 (+36000000), CF = 1, OF = 1, SF = 0
exit
main endp
end main
```

10.2.2 Two-Operand Formats

It 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 formats:

```
IMUL reg8,reg/mem8        ;8-bit destination NOT ALLOWED!!!
```

So, we can only use 16-bit or 32-bit register as the destination operand. 16-bit formats are as follows:

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

32-bit formats are as follows:

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

Note: 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.

```
Include Irvine32.inc
.code
main proc
      mov AX,0FFB0h       ; -80
      imul AX,2           ; AX = FF60 (-160), CF = 0, OF = 0, SF = 1

      mov AX,0E890h       ; -6000
      imul AX,2           ; AX = D120 (-12000), CF = 0, OF = 0, SF = 1

      mov AX,0E890h       ; -6000
      imul AX,200         ; AX = B080 (-20352 wrong), CF = 1, OF = 1, SF = 1
exit
main endp
end main
```

10.2.3 Three-Operand Formats

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, an 8-bit or 16-bit immediate value:

```
IMUL destination, operand2, operand3
```

16-bit formats are as follows:

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

32-bit formats are as follows:

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

Again 8-bit register as the destination is not allowed:

```
IMUL reg8,reg/mem8,imm8         ;8-bit destination NOT ALLOWED!!!
```

Note: The three-operand formats also truncate the product to the length of the destination. If significant digits are lost when IMUL executes, the Overflow and Carry flags are set.

```
Include Irvine32.inc
.code
main proc
      mov BX,0FFB0h       ; -80
      imul AX,BX,2        ; AX = FF60 (-160), CF = 0, OF = 0, SF = 1

      mov BX,0E890h       ; -6000
      imul AX,BX,2        ; AX = D120 (-12000), CF = 0, OF = 0, SF = 1

      mov BX,0E890h       ; -6000
      imul AX,BX,200      ; AX = B080 (-20352 wrong), CF = 1, OF = 1, SF = 1
exit
main endp
end main
```

10.2.4 Unsigned Multiplication using IMUL

We cannot perform unsigned multiplication using IMUL instruction because it will generate wrong answers:

```
Include Irvine32.inc
.code
main proc
      mov AL,96h    ; 150
      mov BL,2
      mul BL        ; 300 (AX = 012Ch, CF = 1) Correct Answer

      mov AL,96h    ; 150
      mov BL,2
      imul BL       ; -212 (AX = FF2Ch, CF = 1) Wrong Answer
exit
main endp
end main
```

The two-operand and three-operand IMUL formats can be used for unsigned multiplication in some cases where the lower half of the product is the same for signed and unsigned numbers (e.g., when the answer does not exceed the input operand's size). There is a small disadvantage to doing so: The Carry and Overflow flags will not indicate whether the upper half of the product equals zero.

10.2.5 Examples of IMUL:

The following instructions multiply 48 by 4, producing +192 in AX. Although the product is correct, AH is not a sign extension of AL, so the Overflow flag is set:

```
mov AL,48
mov BL,4
imul BL             ; AX = 00C0h, OF = 1
```

The following instructions multiply -4 by 4, producing -16 in AX. AH is a sign extension of AL, so the Overflow flag is clear:

```
mov AL,-4
mov BL,4
imul BL             ; AX = FFF0h, OF = 0
```

The following instructions multiply 48 by 4, producing +192 in DX:AX. DX is a sign extension of AX, so the Overflow flag is clear:

```
       mov AX,48
       mov BX,4
       imul BX             ; DX:AX = 000000C0h, OF = 0
```

The following instructions perform 32-bit signed multiplication (4,823,424 * -423), producing -2,040,308,352 in EDX:EAX. The Overflow flag is clear because EDX is a sign extension of EAX:

```
       mov EAX,+4823424
       mov EBX,-423
       imul EBX            ; EDX:EAX = FFFFFFFF86635D80h, OF = 0
```

10.3 Comparing MUL and IMUL to Bit Shifting Execution Times

```
Include Irvine32.inc

LOOP_COUNT = 0FFFFFFFFh
.data
       intval DWORD 5
       startTime DWORD ?
.code
main PROC
       ; Time used by Bit-Shift Mothod:
       call GetMseconds          ; get start time
       mov startTime,eax
       mov eax,intval            ; multiply now
       call mult_by_shifting
       call GetMseconds          ; get stop time
       sub eax,startTime
       call WriteDec             ; display elapsed time
       call Crlf

       ; Time used by MUL instruction:
       call GetMseconds          ; get start time
       mov startTime,eax
       mov eax,intval
       call mult_by_MUL
       call GetMseconds          ; get stop time
       sub eax,startTime
       call WriteDec             ; display elapsed time
       call Crlf
       exit
main ENDP

;---------------------------------
mult_by_shifting PROC
;
; Multiplies EAX by 36 using SHL
;    LOOP_COUNT times.
; Receives: EAX
;---------------------------------
       mov    ecx,LOOP_COUNT
L1:    push   eax                ; save original EAX
       mov    ebx,eax
       shl    eax,5
       shl    ebx,2
       add    eax,ebx
```

```
        pop     eax                 ; restore EAX
        loop    L1
        ret
mult_by_shifting ENDP

;---------------------------------
mult_by_MUL PROC
;
; Multiplies EAX by 36 using MUL
;    LOOP_COUNT times.
; Receives: EAX
;---------------------------------
        mov     ecx,LOOP_COUNT
L1:     push    eax                 ; save original EAX
        mov     ebx,36
        mul     ebx
        pop     eax                 ; restore EAX
        loop    L1
        ret
mult_by_MUL ENDP
END main
```

10.4 DIV Instruction

In 32-bit mode, 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 syntax is:

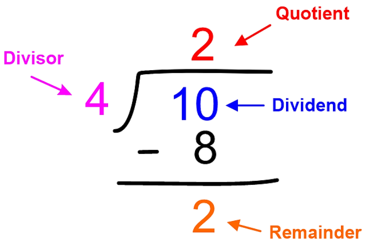
```
        DIV divisor
```

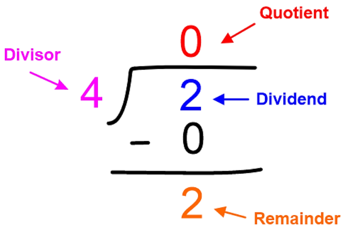
The formats are

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

The following table shows the relationship between the dividend, divisor, quotient, and remainder:

Dividend	Divisor	Quotient	Remainder
AX	reg/mem8	AL	AH
DX:AX	reg/mem16	AX	DX
EDX:EAX	reg/mem32	EAX	EDX
RDX:RAX	reg/mem64	RAX	RDX

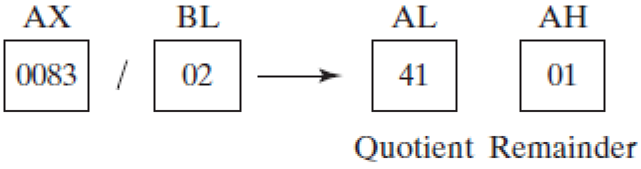




10.4.1 Examples of DIV:

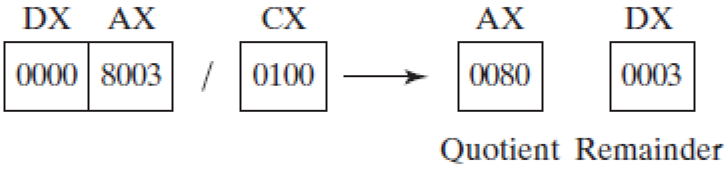
The following instructions perform 8-bit unsigned division (83h/2), producing a quotient of 41h and a remainder of 1:

```
Include Irvine32.inc
.code
main PROC
      mov AX,0083h        ; dividend
      mov BL,2            ; divisor
      div BL              ; AL = 41h, AH = 01h
exit
main ENDP
END main
```



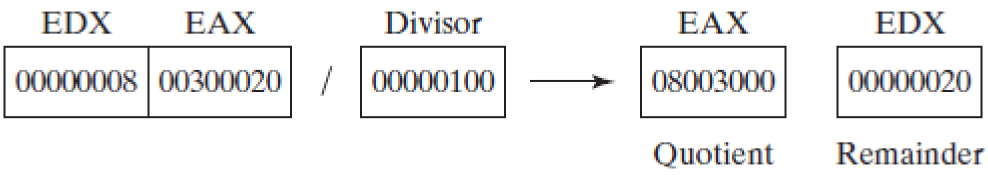
The following instructions perform 16-bit unsigned division (8003h/100h). DX contains the high part of the dividend, so it must be cleared before the DIV instruction executes:

```
Include Irvine32.inc
.code
main PROC
      mov DX,0            ; clear dividend, high
      mov AX,8003h        ; dividend, low
      mov CX,100h         ; divisor
      div CX              ; AX = 0080h, DX = 0003h
exit
main ENDP
END main
```



The following instructions perform 32-bit unsigned division using a memory operand as the divisor:

```
Include Irvine32.inc
.data
      dividend QWORD 0000000800300020h
      divisor DWORD 00000100h
.code
main PROC
      mov edx,DWORD PTR dividend + 4         ; high doubleword (EDX=00000008)
      mov eax,DWORD PTR dividend             ; low doubleword (EAX=00300020)
      div divisor                            ; EAX = 08003000h, EDX = 00000020h
exit
main ENDP
END main
```



The following 64-bit division produces the quotient (0108000000003330h) in RAX and the remainder (0000000000000020h) in RDX:

```
ExitProcess proto
.data
      dividend_hi QWORD 0000000000000108h
      dividend_lo QWORD 0000000033300020h
      divisor QWORD 0000000000010000h
.code
main proc
      mov rdx, dividend_hi
      mov rax, dividend_lo
      div divisor ; RAX = 0108000000003330, RDX = 0000000000000020
call  ExitProcess
main endp
end
```

Notice how each hexadecimal digit in the dividend was shifted 4 positions to the right, because it was divided by 64. (Division by 16 would have moved each digit only one position to the right.)

10.5 Signed Integer Division

Signed integer division is nearly identical to unsigned division, with one important difference: The dividend must be sign-extended before the division takes place.

Sign extension is the term used for copying the highest bit of a number into all of the upper bits of its enclosing variable or register. To show why this is necessary, let's try leaving it out. The following code uses MOV to assign -101 to AL, which is the lower half of AX:

```
Include Irvine32.inc
.data
      var1 SBYTE -101     ; 9Bh
.code
main proc
      mov EAX,0           ; EAX = 00000000h
      mov AL,var1         ; EAX = 0000009Bh (+155)
      mov BL,2            ; divisor
      idiv BL             ; divide AX by BL (signed operation) Wrong Answer
exit
main endp
end main
```

Unfortunately, the 009Bh in AX is not equal to -101. It is equal to +155, so the quotient produced by the division is +77, which is not what we wanted.

The correct way to set up the problem is to use the **CBW** instruction (convert byte to word), which sign-extends AL into AX before performing the division:

```
Include Irvine32.inc
.data
        var1 SBYTE -101     ; 9Bh
.code
main proc
        mov EAX,0           ; EAX = 00000000h
        mov AL,var1         ; AX = 009Bh (+155) wrong
        cbw                 ; AX = FF9Bh (-101) correct
        mov BL,2            ; divisor
        idiv BL             ; divide AX by BL (signed operation) Correct Answer
exit
main endp
end main
```

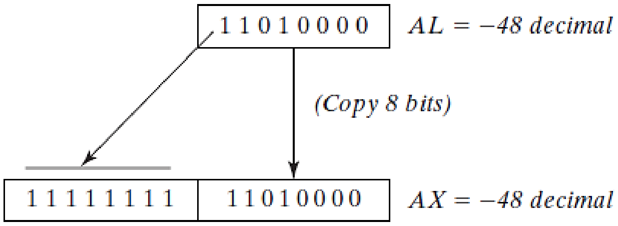
Note: We studied the concept of sign extension in previous Labs along with the MOVSX instruction as well as using shift instructions.

10.5.1 Sign Extension Instructions (CBW, CWD, CDQ)

The **CBW** instruction (convert byte to word) extends the sign bit of AL into AH, preserving the number's sign. In the next example, D0h (in AL) and FFD0h (in AX) both equal −48 decimal:

```
.data
        byteVal SBYTE -48   ; D0h
.code
        mov AL,byteVal      ; AL = D0h
        cbw                 ; AX = FFD0h
```

The following illustration shows how AL is sign-extended into AX by the CBW instruction:



The **CWD** (convert word to doubleword) instruction extends the sign bit of AX into DX:

```
.data
        wordVal SWORD -101  ; FF9Bh
.code
        mov AX,wordVal      ; AX = FF9Bh
        cwd                 ; DX:AX = FFFFFF9Bh
```

The **CDQ** (convert doubleword to quadword) instruction extends the sign bit of EAX into EDX:

```
.data
        dwordVal SDWORD -101        ; FFFFFF9Bh
.code
        mov EAX,dwordVal
        cdq                         ; EDX:EAX = FFFFFFFFFFFFFF9Bh
```

10.5.2 IDIV Instruction

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

Example 1:

```
Include Irvine32.inc
.data
      byteVal SBYTE -48   ; D0 hexadecimal
.code
main PROC
      mov AL,byteVal      ; lower half of dividend
      cbw                 ; extend AL into AH
      mov BL,+5           ; divisor
      idiv BL             ; AL = -9, AH = -3
exit
main ENDP
END main
```

Example 2:

```
Include Irvine32.inc
.data
      wordVal SWORD -5000
.code
main PROC
      mov AX,wordVal      ; dividend, low
      cwd                 ; extend AX into DX
      mov BX,+256         ; divisor
      idiv BX             ; quotient AX = -19, rem DX = -136
exit
main ENDP
END main
```

Example 3:

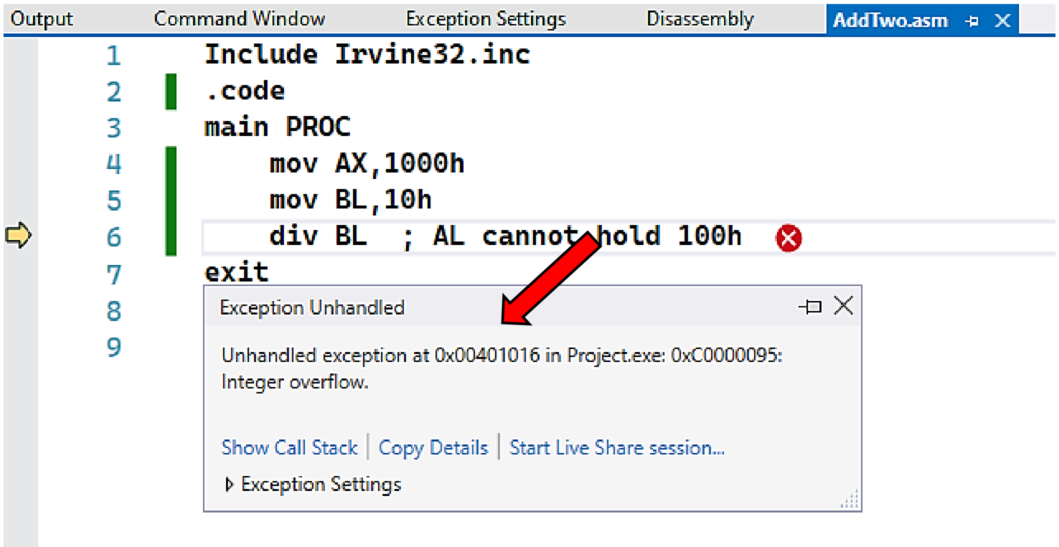
```
Include Irvine32.inc
.data
      dwordVal SDWORD +50000
.code
main PROC
      mov EAX,dwordVal    ; dividend, low
      cdq                 ; extend EAX into EDX
      mov EBX,-256        ; divisor
      idiv EBX            ; quotient EAX = -195, rem EDX = +80
exit
main ENDP
END main
```

Note: All arithmetic status flag values are undefined after executing DIV and IDIV.

10.5.3 Divide Overflow

If a division operand produces a quotient that will not fit into the destination operand, a divide overflow condition occurs. This causes a processor exception and halts the current program.

```
Include Irvine32.inc
.code
main PROC
       mov AX,1000h
       mov BL,10h
       div BL       ; AL cannot hold 100h
exit
main ENDP
END main
```



```
Include Irvine32.inc
.code
main PROC
    mov AX,1000h
    mov BL,10h
    div BL   ; AL cannot hold 100h
exit
```

Exception Unhandled

Unhandled exception at 0x00401016 in Project.exe: 0xC0000095: Integer overflow.

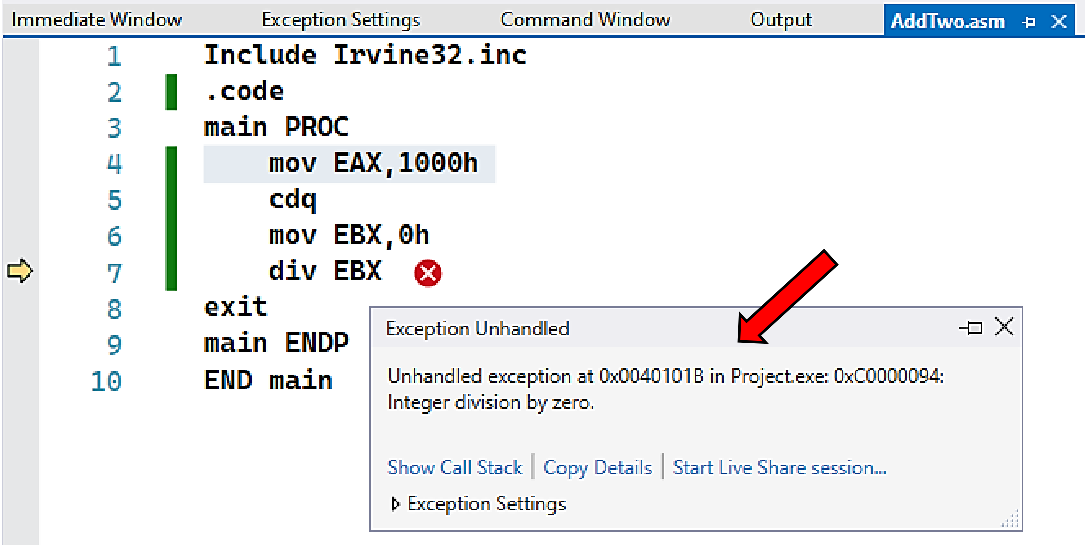
The solution is to use a 32-bit divisor and 64-bit dividend to reduce the probability of a divide overflow condition. In the following code, the divisor is EBX, and the dividend is placed in the 64-bit combined EDX and EAX registers:

```
Include Irvine32.inc
.code
main PROC
       mov EAX,1000h
       cdq
       mov EBX,10h
       div EBX      ; EAX = 00000100h
exit
main ENDP
END main
```

10.5.4 Divide by Zero

CPU generates exception and halts the current program if we divide the operand with zero.

```
Include Irvine32.inc
.code
main PROC
       mov EAX,1000h
       cdq
       mov EBX,0h
       div EBX      ; Divide by Zero, CPU Exception!!!
exit
main ENDP
END main
```



```
Include Irvine32.inc
.code
main PROC
    mov EAX,1000h
    cdq
    mov EBX,0h
    div EBX
exit
main ENDP
END main
```

Exception Unhandled

Unhandled exception at 0x0040101B in Project.exe: 0xC0000094: Integer division by zero.

Show Call Stack | Copy Details | Start Live Share session...

▷ Exception Settings

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

```
Include Irvine32.inc
.data
       error_message BYTE "Divide by Zero Error!!!",0
.code
main PROC
       mov EAX,1000h
       cdq
       mov EBX,0h
       cmp EBX,0                  ; check the divisor
       je NoDivideZero            ; zero? display error
       div EBX
       exit
NoDivideZero:
       mov EDX, OFFSET error_message
       call WriteString
       call Crlf
       exit
main ENDP
END main
```

10.6 Implementing Arithmetic Expressions

Example 1:

Implement the following C++ statement in assembly language, using unsigned 32-bit integers:

$$var4 = (var1 + var2) * var3;$$

```
Include Irvine32.inc
.data
       var1 DWORD 5
       var2 DWORD 15
       var3 DWORD 2000000000
       var4 DWORD ?
       error_message BYTE "Overflow!!!",0
.code
main proc
       mov eax,var1
       add eax,var2
       mul var3            ; EAX = EAX * var3
       jc tooBig           ; unsigned overflow?
       mov var4,eax
       call WriteInt
       call Crlf
       jmp done
tooBig:                          ; display error message
       mov EDX, OFFSET error_message
       call WriteString
       call Crlf
done:
       exit
main endp
end main
```

If the MUL instruction generates a product larger than 32 bits, the JC instruction jumps to a label that handles the error.

Example 2:

Implement the following C++ statement, using unsigned 32-bit integers:

$$var4 = (var1 * 5) / (var2 - 3);$$

```
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
```

Example 3:

Implement the following C++ statement, using signed 32-bit integers:

$$var4 = (var1 * -5) / (-var2 \% var3);$$

We can begin with the expression on the right side and store its value in EBX. Because the operands are signed, it is important to sign extend the dividend into EDX and use the IDIV instruction:

```
mov EAX,var2        ; begin right side
neg EAX
cdq                 ; sign-extend dividend
idiv var3           ; EDX = remainder
mov EBX,EDX         ; EBX = right side
```

Next, we calculate the expression on the left side, storing the product in EDX:EAX:

```
mov EAX,-5          ; begin left side
imul var1           ; EDX:EAX = left side
```

Finally, the left side (EDX:EAX) is divided by the right side (EBX):

```
idiv EBX            ; final division
mov var4,EAX        ; quotient
```