# 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
```

# 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
```

# 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.

# IMUL Examples

Multiply 4,823,424 * –423:

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

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

# Your turn . . .

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

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

# 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 |

Irvine, Kip R. Assembly Language for Intel-Based Computers, 2003.

# DIV Examples

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

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

Same division, using 32-bit operands:

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

# 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
```

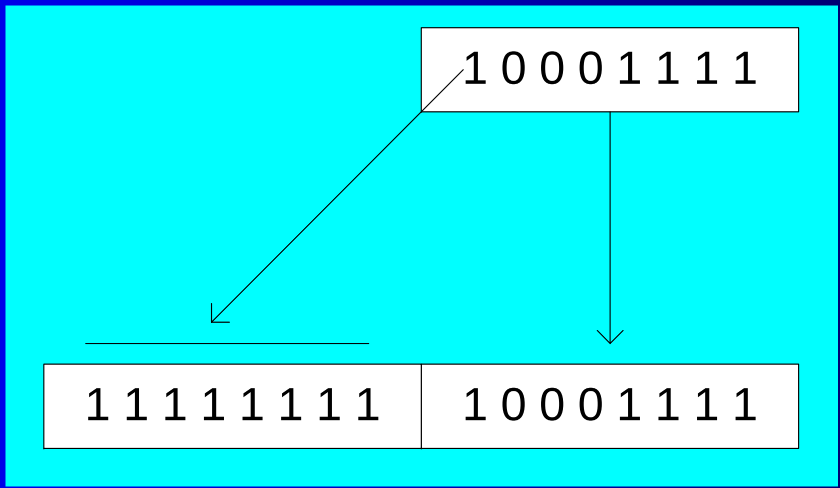
# Your turn . . .

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

# 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
```

# Implementing Arithmetic Expressions (1 of 3)

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

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

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

# Implementing Arithmetic Expressions (2 of 3)

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

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

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

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

# Implementing Arithmetic Expressions (3 of 3)

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

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

# Your turn . . .

Implement the following expression using signed 32-bit integers:

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

```
mov eax,20
imul ebx
idiv ecx
```

# Your turn . . .

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

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

```
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
imul edx                  ; left side: edx:eax
mov ecx,var3
sub ecx,ebx
idiv 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 doublewords

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

View the complete source code.

# SBB Instruction

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

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

# ASCII and Packed Decimal Arithmetic

- Unpacked BCD
- ASCII Decimal
- AAA Instruction
- AAS Instruction
- AAM Instruction
- AAD Instruction
- Packed Decimal Integers
- DAA Instruction
- DAS Instruction

# Unpacked BCD

- Binary-coded decimal (BCD) numbers use 4 binary bits to represent each decimal digit
- A number using unpacked BCD representation stores a decimal digit in the lower four bits of each byte
  - For example, 5,678 is stored as the following sequence of hexadecimal bytes:

| | | | |
|---|---|---|---|
| 05 | 06 | 07 | 08 |

# ASCII Decimal

- A number using ASCII Decimal representation stores a single ASCII digit in each byte
  - For example, 5,678 is stored as the following sequence of hexadecimal bytes:

| | | | |
|---|---|---|---|
| 35 | 36 | 37 | 38 |

# AAA Instruction

- The AAA (ASCII adjust after addition) instruction adjusts the binary result of an ADD or ADC instruction. It makes the result in AL consistent with ASCII digit representation.
  - The Carry value, if any ends up in AH
- Example: Add '8' and '2'

```
mov ah,0
mov al,'8'          ; AX = 0038h
add al,'2'          ; AX = 006Ah
aaa                 ; AX = 0100h (adjust result)
or  ax,3030h        ; AX = 3130h = '10'
```

# AAS Instruction

- The AAS (ASCII adjust after subtraction) instruction adjusts the binary result of an SUB or SBB instruction. It makes the result in AL consistent with ASCII digit representation.
  - It places the Carry value, if any, in AH
- Example: Subtract '9' from '8'

```
mov ah,0
mov al,'8'          ; AX = 0038h
sub al,'9'          ; AX = 00FFh
aas                 ; AX = FF09h (adjust result)
pushf               ; save Carry flag
or al,30h           ; AX = FF39h (AL = '9')
popf                ; restore Carry flag
```

# AAM Instruction

- The AAM (ASCII adjust after multiplication) instruction adjusts the binary result of a MUL instruction. The multiplication must have been performed on unpacked decimal numbers.

```
mov bl,05h          ; first operand
mov al,06h          ; second operand
mul bl              ; AX = 001Eh
aam                 ; AX = 0300h
```

# AAD Instruction

- The AAD (ASCII adjust before division) instruction adjusts the unpacked decimal dividend in AX before a division operation

```
.data
quotient  BYTE ?
remainder BYTE ?
.code
mov ax,0307h                    ; dividend
aad                             ; AX = 0025h
mov bl,5                        ; divisor
div bl                          ; AX = 0207h
mov quotient,al
mov remainder,ah
```

# Packed Decimal Integers

- Packed BCD stores two decimal digits per byte
  - For example, 12,345,678 can be stored as the following sequence of hexadecimal bytes:

| | | | |
|---|---|---|---|
| 12 | 34 | 56 | 78 |

# DAA Instruction

- The DAA (decimal adjust after addition) instruction converts the binary result of an ADD or ADC operation to packed decimal format.
- The value to be adjusted must be in AL
- Example: calculate BCD 35 + 48

```
mov al,35h
add al,48h                    ; AL = 7Dh
daa                           ; AL = 83h (adjusted)
```

# DAS Instruction

- The DAS (decimal adjust after subtraction) instruction converts the binary result of a SUB or SBB operation to packed decimal format.
- The value must be in AL
- Example: subtract BCD 48 from 85

```
mov al,85h
sub al,48h            ; AL = 3Dh
das                   ; AL = 37h (adjusted)
```