Integer Arithmetic

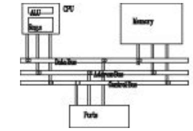
Computer Organization and Assembly Languages

with slides by Kip Irvine

Chapter 7 Integer Arithmetic Overview

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

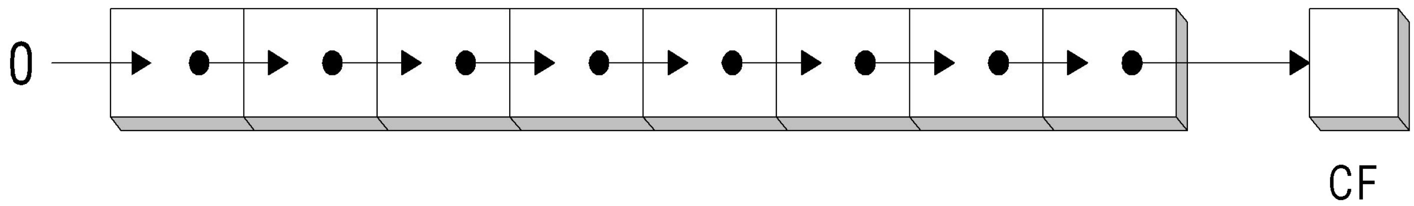
Shift and Rotate Instructions



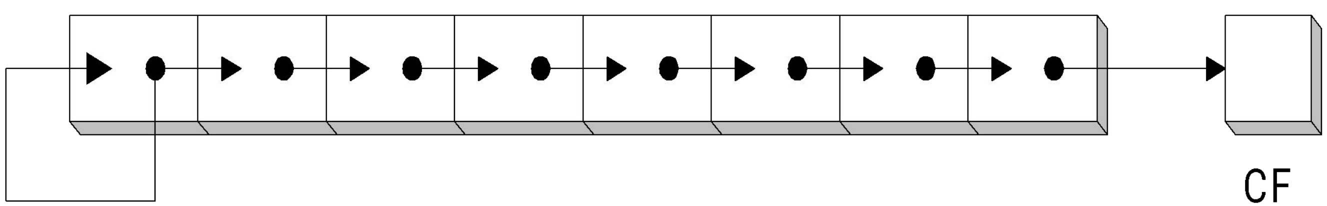
- Logical vs Arithmetic Shifts
- SHL Instruction
- SHR Instruction
- SAL and SAR Instructions
- ROL Instruction
- ROR Instruction
- RCL and RCR Instructions
- SHLD/SHRD Instructions

Logical vs arithmetic shifts

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

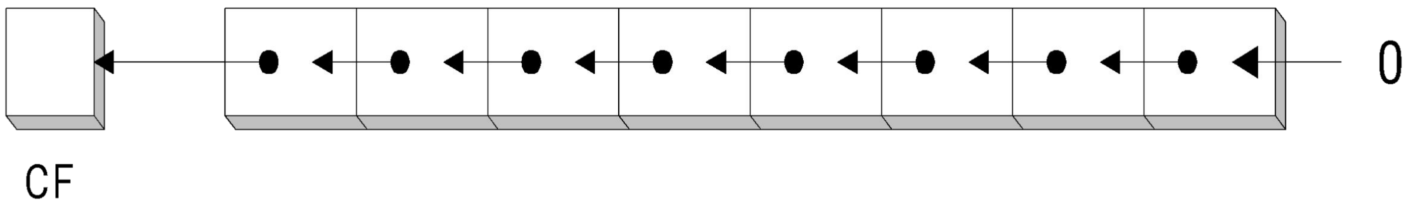


- An arithmetic shift fills the newly created bit position with a copy of the number's sign bit:



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 *destination,count***

  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

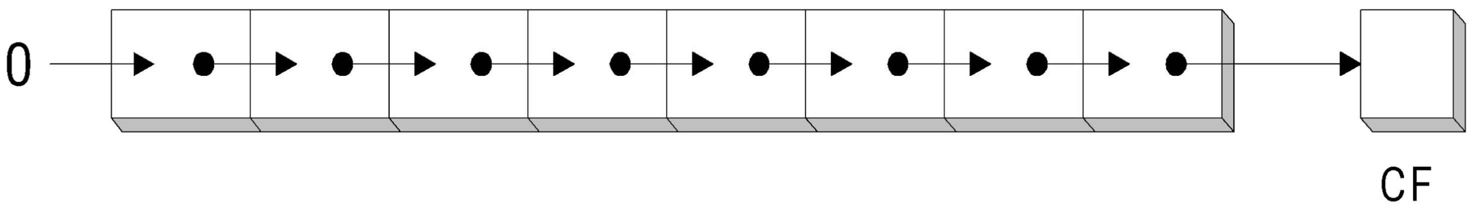
Shifting left *n* bits multiplies the operand by 2^n

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.

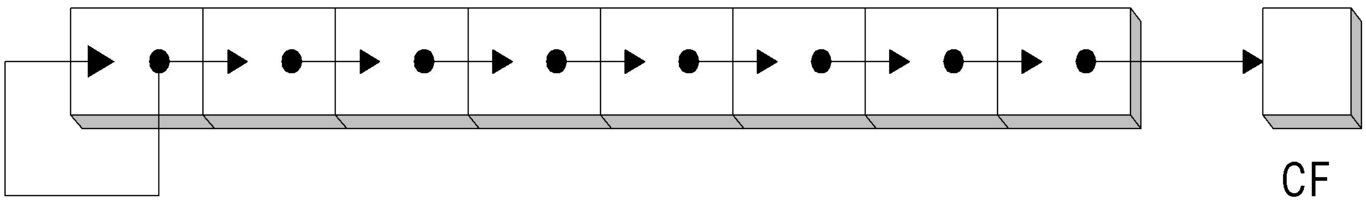


Shifting right n bits divides the operand by 2^n

```
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.

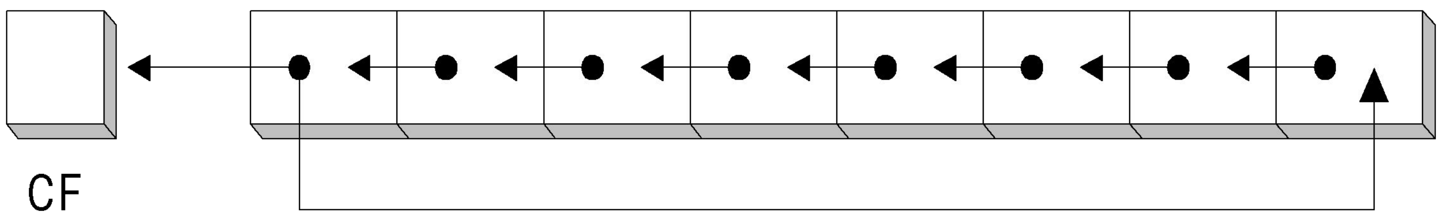
```
mov dl,-80
sar dl,1   ; DL = -40
sar dl,2   ; DL = -10
```

```
mov al,-80
 sar al,1 ; AL = -40
 sar al,2 ; AL = -10
 movsx eax,al
 call writeint
```

```
movsx eax,dl
 call writeint
```

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

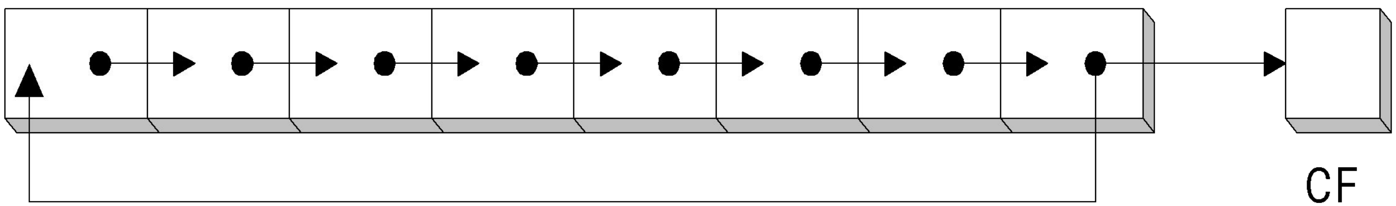



```
mov al,11110000b
rol al,1   ; AL = 11100001b

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

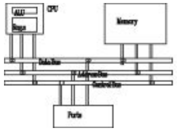
ROR instruction

- 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 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
```

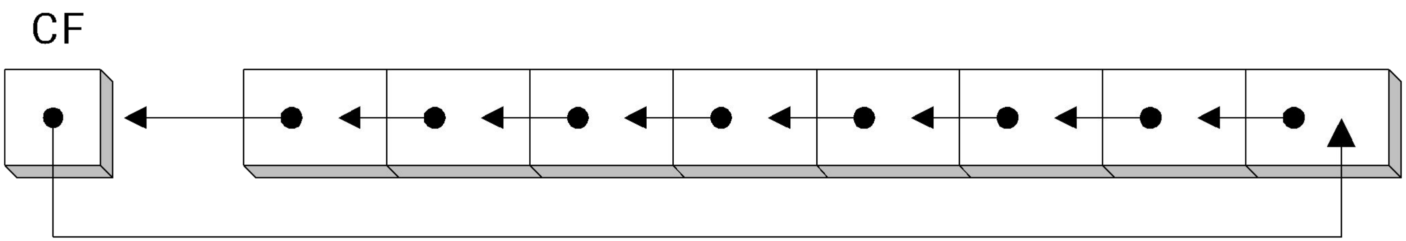
Your turn . . .

Indicate the hexadecimal value of AL after each rotation:

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

RCL instruction

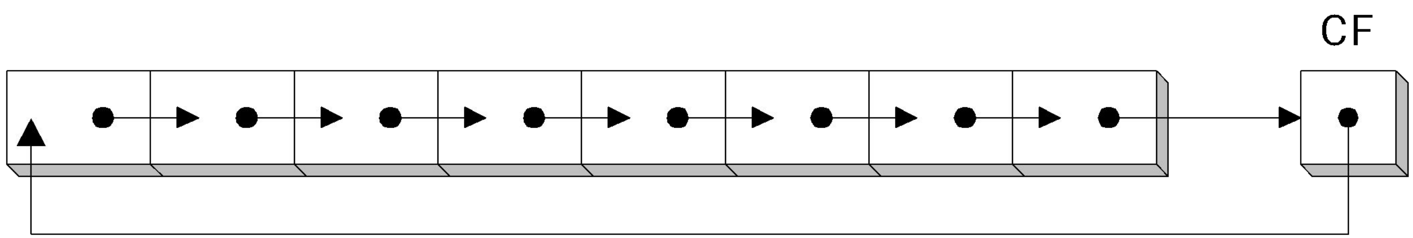
- RCL (rotate carry left) shifts each bit to the left
- Copies the Carry flag to the least significant bit
- Copies the most significant bit to the Carry flag




```
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 shift left destination

- Syntax:

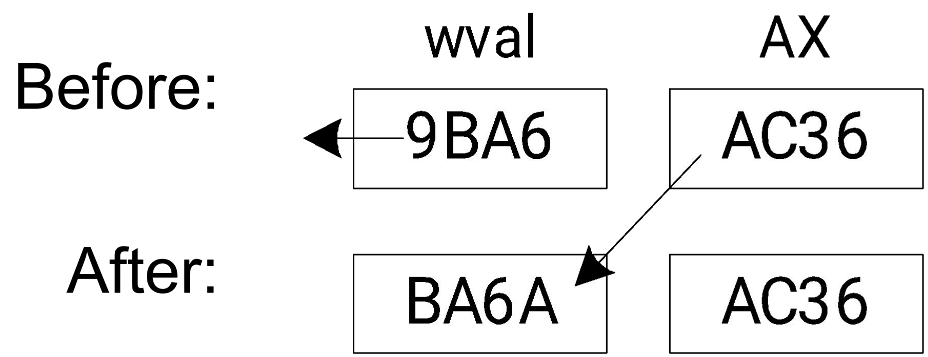
```
SHLD destination, source, count
```

- 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

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

- Syntax:

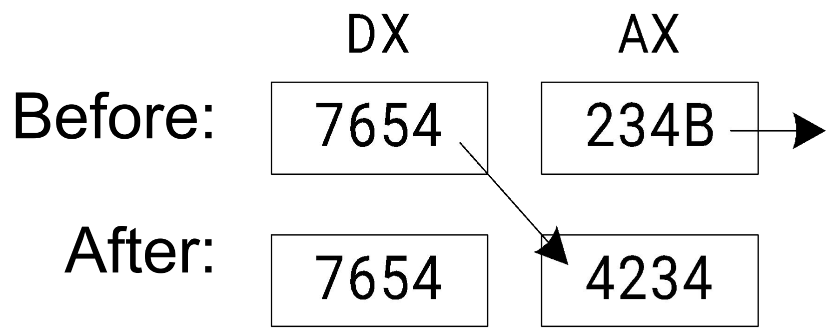
  SHRD ***destination, source, count***

- 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

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
```



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
```

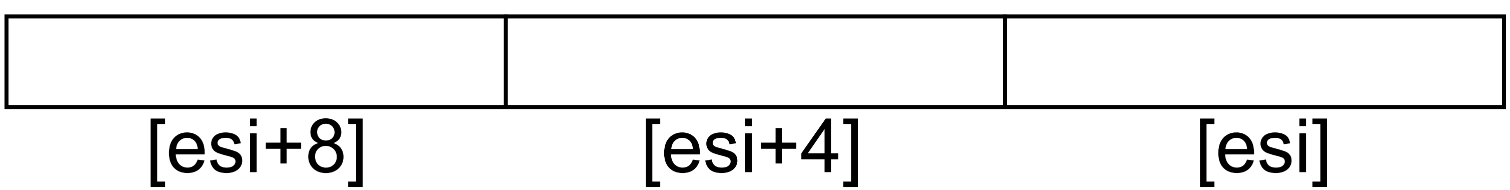
Shift and rotate applications

- Shifting Multiple Doublewords
- Binary Multiplication
- Displaying Binary Bits
- Isolating a Bit String

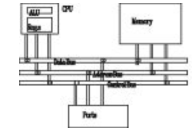
Shifting multiple doublewords

- 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:

```
mov esi,0
shr array[esi + 8],1     ; high dword
rcr array[esi + 4],1     ; middle dword,
rcr array[esi],1   ; low dword,
```



Binary multiplication



- We already know that SHL performs unsigned multiplication efficiently when the multiplier is a power of 2.
- 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
shl ebx,2
add eax,ebx
```

Your turn . . .

Multiply AX by 26, using shifting and addition instructions. *Hint*: 26 = 16 + 8 + 2.

```
mov ax,2  ; test value

mov dx,ax
shl dx,4  ; AX * 16
push dx ; save for later
mov dx,ax
shl dx,3  ; AX * 8
shl ax,1  ; AX * 2
add ax,dx ; AX * 10
pop dx  ; recall AX * 16
add ax,dx ; AX * 26
```

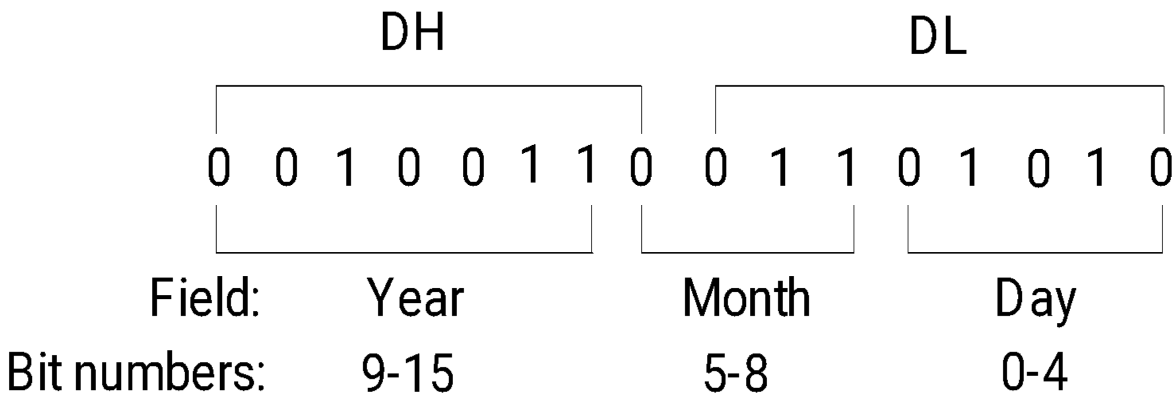
Displaying binary bits

Algorithm: Shift MSB into the Carry flag; If CF = 1, append a "1" character to a string; otherwise, append a "0" character. Repeat in a loop, 32 times.

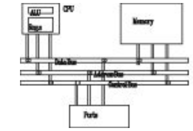
```
     mov ecx,32
     mov esi,offset buffer
L1:  shl eax,1
     mov BYTE PTR [esi],'0'
     jnc L2
     mov BYTE PTR [esi],'1'
L2:  inc esi
     loop L1
```

Isolating a bit string

- The MS-DOS file date field packs the year (relative to 1980), month, and day into 16 bits:



Isolating a bit string

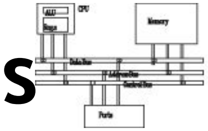


```
mov al,dl           ; make a copy of DL
and al,00011111b ; clear bits 5-7
mov day,al          ; save in day variable
```

```
mov ax,dx           ; make a copy of DX
shr ax,5            ; shift right 5 bits
and al,00001111b ; clear bits 4-7
mov month,al        ; save in month variable
```

```
mov al,dh           ; make a copy of DX
shr al,1            ; shift right 1 bit
mov ah,0            ; clear AH to 0
add ax,1980         ; year is relative to 1980
mov year,ax         ; save in year
```

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 (E)DX, (E)AX, and the Carry flag after the following instructions execute?

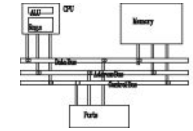
```
mov ax,1234h
mov bx,100h
mul bx
```

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

```
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.

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 . . .

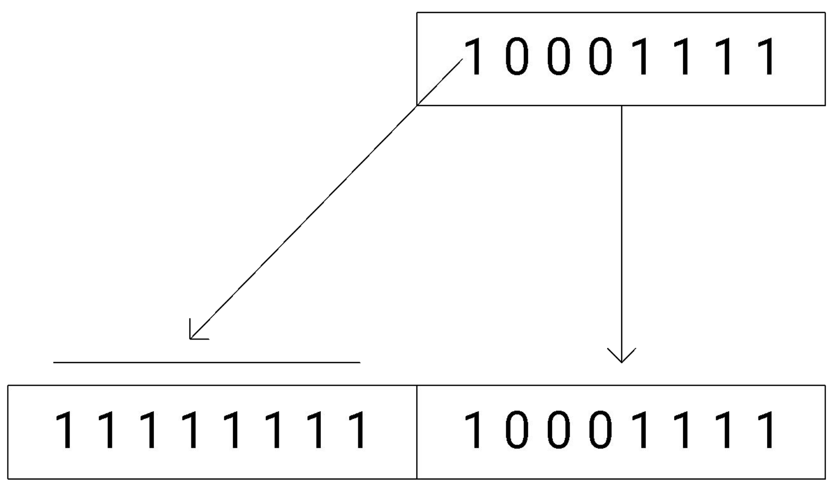
What will be the hexadecimal values of DX and AX after the following instructions execute?

```
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:

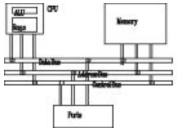


CBW, CWD, CDQ instructions

- 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; -101 (32 bits)
cdq; EDX:EAX = FFFFFFFFFFFFFF9Bh
    ; -101 (64 bits)
```

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?

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

DX = FFFFh (−1), AX = FFFEh (−2)

Divide overflow

- *Divide overflow* happens when the quotient is too large to fit into the destination.

```
mov ax, 1000h
mov bl, 10h
div bl
```

It causes a CPU interrupt and halts the program. (divided by zero cause similar results)

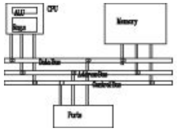
Implementing arithmetic expressions

- Some good reasons to learn how to implement expressions:
  - Learn how compilers do it
  - Test your understanding of MUL, IMUL, DIV, and IDIV
  - Check for 32-bit 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



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  cdq ?
div ebx      ; final division
mov var4,eax
```

Implementing arithmetic expressions

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
```

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

Your turn . . .

Implement the following expression using signed 32-bit integers:

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

```
mov eax,20
mul ebx
div ecx
```

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 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 leftover carry
```

Extended addition example

```
.data
op1 QWORD 0A2B2A40674981234h
op2 QWORD 08010870000234502h
sum DWORD 3 dup(?)
    ; = 0000000122C32B0674BB5736
.code
...
mov  esi,OFFSET op1 ; first operand
mov  edi,OFFSET op2 ; second operand
mov  ebx,OFFSET sum ; sum operand
mov  ecx,2          ; number of doublewords
call Extended_Add
...
```

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
```