Computer Organization and Assembly Language

Lecture 7 - Integer Arithmetic

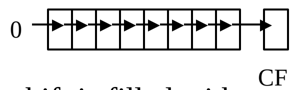
Shift and Rotate Instructions

- Shifting means to move bits right and left inside an operand.
- All of the Shift and Rotate instructions affect Overflow and Carry Flags.
- The Shift and Rotate instructions include:

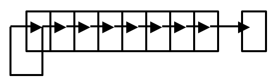
`SHL` - **Sh**ift **L**eft	`ROL` – **Ro**tate **L**eft
`SHR` - **Sh**ift **R**ight	`ROR` – **Ro**tate **R**ight
`SAL` – **S**hift **A**rithmetic **L**eft	`RCL` - **R**otate **C**arry **L**eft
`SAR` – **S**hift **A**rithmetic **R**ight	`RCR` - **R**otate **C**arry **R**ight
`SHLD` - **Sh**ift **L**eft **D**ouble	`SHRD` - **Sh**ift **R**ight **D**ouble

Logical Shifts Vs. Arithmetic Shifts

- A logical shift fills the newly created bit position with zero.If we do a single logical right shift on 11001111, it becomes 011001111.



- An arithmetic shift is filled with a copy of the original number's sign bit.If we do a single arithmetic right shift on 11001111, it becomes 11100111.



SHL Instruction

- The Shift Left instruction performs a left shift on the destinations operand, filling the lowest bit with 0. The highest bit is moved into the Carry Flag.
- The instruction format is:

  SHL *destination*, *bits_shifted*

- Instruction formats include:

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

SHL Instruction - Examples

- The following instruction sequence shifts the BL once to the left, with the highest bit copied into the Carry flag and the lowest bit cleared:
- **movbl, 8Fh ; BL = 1000111b**

```
shl      bl, 1       ; BL = 00011110b, CF = 1
```

- SHL can be used to perform a high-speed multiplication by powers of 2:

```
mov      dl, 5 ; DL = 00000101b
shl      dl, 1 ; DL = 00001010b
mov      dl, 2 ; DL = 00101000b, = 40
```

SHR Instruction

- The Shift Right instruction performs a right shift on the destinations operand, filling the lowest bit with 0. The lowest bit is moved into the Carry Flag.
- The instruction format is:

  SHR *destination*, *bits_shifted*

- Instruction formats include:

  SHR *reg*, *imm8*
  SHR *mem*, *imm8*
  SHR *reg*, *CL*
  SHR *mem*, *CL*

SHR Instruction - Examples

- The following instruction sequence shifts the AL once to the right, with the lowest bit copied into the Carry flag and the highest bit cleared:
- **moval, D0h ; AL = 11010000b**

```
shr      al, 1       ; AL = 01101000b, CF = 0
```

- SHR can be used to perform a high-speed division by 2^n:

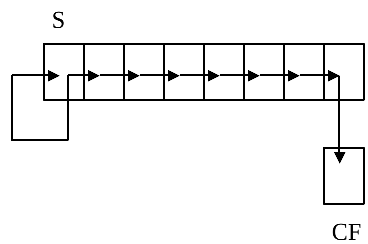
```
mov      dl, 32      ; DL = 00100000b = 32
shr      dl, 1       ; DL = 00010000b = 16
mov      al, 040h    ; AL = 01000000b = 64
shr      al, 3       ; AL = 00001000b = 8
```

SAL and SAR Instructions

- SAL (Shift Arithmetic Left) is identical to the SHL instruction.
- SAR (Shift Arithmetic Right) performs a right arithmetic shift on its operand.
- The instruction format is:

  SAR *destination*, *bits_shifted*
- Instruction formats include:

  SAR *reg*, *imm8*
  SAR *mem*, *imm8*
  SAR *reg*, *CL*
  SAR *mem*, *CL*



SAR Instruction - Examples

- The following instruction sequence shifts the AL once to the right, with the lowest bit copied into the Carry flag and the sign bit copied to the right:
- `moval, F0h ; AL = 11110000b = -16`

```
shr      al, 1       ; AL = 11111000b = -8
                     ;          CF = 0
```

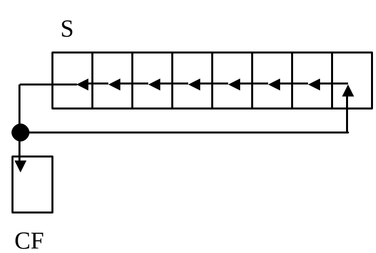
- SAR can be used to perform a high-speed signed division by 2^n:

```
mov      dl, -128    ; DL = 10000000b = -128
shr      dl, 3       ; DL = 11110000b = -16
```

ROL Instruction

- The **`ROL`** instruction shifts each bit to the left, with the highest bit copied in the Carry flag and ***<u>into the lowest bit</u>***.
- The instruction format is:

`ROL` *destination*, *bits_shifted*



- Instruction formats include:

ROL	*reg*, *imm8*
ROL	*mem*, *imm8*
ROL	*reg*, *CL*
ROL	*mem*, *CL*

ROL Instruction - Examples

- The following instruction sequence shifts the AL three times (once each) to the left, with the highest bit copied into the Carry flag ***and*** into the lowest bit:

```
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
```

- You can use ROL to exchange the upper and lower halves of a byte:

```
mov     al, 26h
rol     al, r ; AL = 01100010b = 62h
```

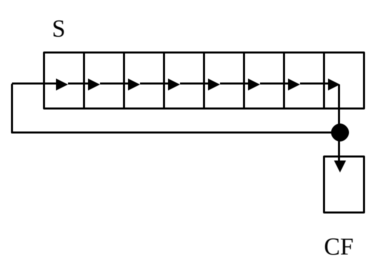
ROR Instruction

- The **ROR** instruction shifts each bit to the right, with the lowest bit copied in the Carry flag and ***into the highest bit***.
- The instruction format is:

ROR *destination*, *bits_shifted*

- Instruction formats include:

ROR *reg*, *imm8*
ROR *mem*, *imm8*
ROR *reg*, *CL*
ROR *mem*, *CL*



ROR Instruction - Examples

- The following instruction sequence shifts the AL three times (once each) to the right, with the lowest bit copied into the Carry flag ***and*** into the highest bit:

```
mov     al, 01h     ; AL = 00000001b
ror     al, 1       ; AL = 10000000b, CF = 1
ror     al, 1       ; AL = 01000000b, CF = 0
ror     al, 1       ; AL = 00100000b, CF = 0
```

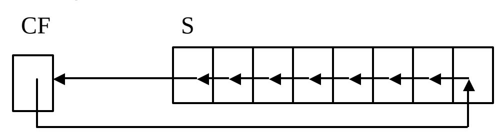
- You can use ROL to exchange the upper and lower halves of a byte:

```
mov     al, 26h
ror     al, r ; AL = 01100010b = 62h
```

RCL Instruction

- The **RCL** (**R**otate and **C**arry **L**eft) instruction shifts each bit to the left, copies the Carry flag to the least significant bit and copies the most significant bit into the Carry flag.
- In this examples, the lowest bit is copied into the Carry flag and into the highest bit of the result:

```
clc                 ; CF = 0
mov     bl, 88h     ; CF = 0 BL = 10001000b
rcl     bl, 1       ; CF = 1 AL = 00010000b
rcl     bl, 1       ; CF = 0 AL = 00100001b
```



Example – Recovery a Carry Flag Bit

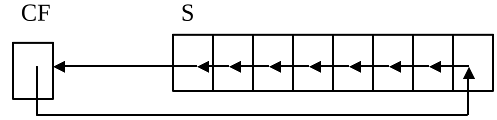
- RCL can recover a bit that has previously been shifted into the Carry flag:

```
.data
testval   BYTE  01101010b
.code
          shr   testval, 1   ; shift LSB into CF
          jc    quit         ; exit if Carry
                             ; Flag set
          rcl   testval, 1   ; else restore the
                             ; number
```

RCR Instruction

- The **RCR** (**R**otate and **C**arry **R**ight) instruction shifts each bit to the right, copies the Carry flag to the most significant bit and copies the least significant bit into the Carry flag.
- In this examples, the lowest bit is copied into the Carry flag and into the highest bit of the result:

```
stc                 ; CF = 1
mov      ah, 10h    ; CF = 1 AH = 00010000b
rcr      ah, 1      ; CF = 0 AL = 00001000b
rcr      ah, 1      ; CF = 0 AL = 00000100b
```



SHLD/SHRD Instructions

- The SHLD and SHLR instructions (**Sh**ift **L**eft/**R**ight **D**oubleword) require at least a 386 processor.
- When the SHLD (SHRD) is called, the bit positions opened by the shift in the first operand are filled by the the most (least) significant bits of the second operand.
- The second operand is unaffected but the Sign, Zero, Auxiliary Parity and Carry Flags are affected.

SHLD/SHRD Instructions (continued)

- The syntax is:

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

  `SHLR` *`destination, source, count`*

- The instruction formats for both are:

`SHLD` *`reg16, reg16,`* **`CL`**/*`imm8`*

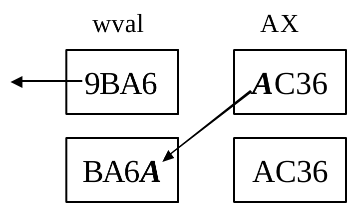
`SHLD` *`mem16, reg16,`* **`CL`**/*`imm8`*

`SHLD` *`reg32, reg32,`* **`CL`**/*`imm8`*

`SHLD` *`mem32, reg32,`* **`CL`**/*`imm8`*

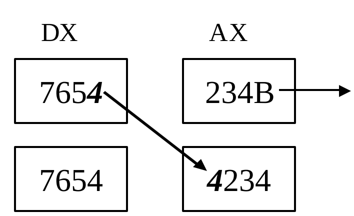
SHLD – An Example

```
.data
wval WORD 9BA6H
.code
     mov  ax, AC36H
     shld wval, ax, 4    ; wval = BA6Ah
```



SHRD – An Example

```
mov  ax, 234Bh
mov  dx, 7654h
shrd ax, dx, 4 ; wval = 4234h
```



Shift and Rotate Applications

- Shift and Rotate instructions are included because they are helpful in certain applications.
- These applications includes:
  - Shifting Multiple Doublewords (for bit-mapped graphics images)
  - Binary multiplication
  - Display Binary Bits
  - Isolating a Bit String

Shifting Multiple Doublewords

- Some programs need to manipulate all the bits within an array, such as in a bit-mapped graphic image one location location on a screen to another.
- `.data`

```
ArraySize = 3
array DWORD ArraySize DUP(99999999H);1001  etc.
.code
      mov   esi, 0
      shr   array[esi+8], 1   ; high dword
      rcr   array[esi+4], 1   ; middle dword & CF
      rcr   array[esi], 1     ; low dword & CF
Before 1001 1001 1001 1001 1001 1001 1001 1001 …
After  0100 1100 1100 1100 1100 1100 1100 1100 …
```

Binary Multiplication

- We can save time multiplying if we can use shifting to replace multiplying by 2, even if we need to add afterwards:

  EAX * 36 = EAX * (32 + 4)

  = EAX*32 + EAX * 4

```
.code
  mov     eax, 123
  mov     ebx, eax
  shl     eax, 5      ; multiply by 2^5
  shl     ebx, 2      ; mulitply by 2^2
  add     eax, ebx    ; add the products
```

Displaying Binary Bits

```
TITLE Displaying Binary Bits
; Display a 32-bit integer in binary

INCLUDE Irvine32.inc

.data
binValue    DWORD 1234ABCDh   ; sample bin. value
buffer      BYTE  32 dup(0), 0
.code
main  PROC
  mov       eax, binValue ; number to display
  mov       ecx, 32       ; number of bits in EAX
  mov       esi, offset buffer
```

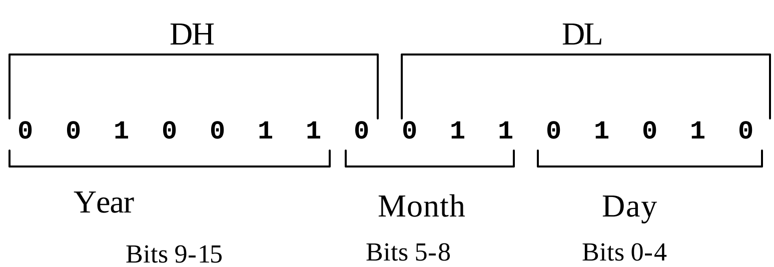
```
L1:   shl   eax, 1      ; shift high bit into CF
      mov   BYTE ptr [esi], '0‘
                        ; choose 0 as default
                        ; digit
      jnc   L2
      mov   BYTE ptr[esi], '1‘
                        ; else move to buffer

L2:   inc   esi         ; next buffer position
      loop  L1          ; shift a bit to left

      mov   edx, OFFSET buffer
      call  WriteString
      call  CrLf
      exit
main  ENDP
      END   main
```

Isolating A Bit String

- Often a byte or word contains more than one field, making it necessary to extract short sequences of bit called ***bit strings***.
- MS-DOS function **`57h`** returns a file date stamp.



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

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

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

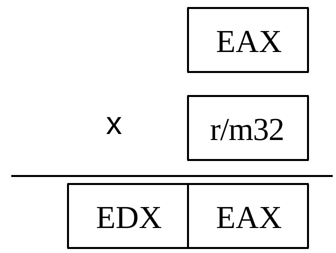
Multiplication And Division Instructions

- Unlike addition and subtraction, multiplication and division operations are different for signed and unsigned operands.
- The Intel architecture allows you multiply and divide 8- 16- and 32-bit integers.
- The operators are:
  - **`MUL`** and **`DIV`** for unsigned multiplication and division.
  - **`IMUL`** and **`IDIV`** for unsigned multiplication and division.

`MUL` Instruction

- The **`MUL`** instruction multiplies an 8-, 16, or 32-bit ***unsigned*** operand by either the AL, AX or EAX register (depending on the operand's size).
- The instruction formats are:

```
MUL    r/m8
MUL    r/m16
MUL    r/m32
```



`MUL` Instruction (continued)

Multiplicand	**Multiplier**	**Product**
AL	r/m8	AX
AX	r/m16	DX:AX
EAX	r/m32	EDX:EAX

- The **`MUL`** instruction sets the Carry and Overflow flags if the upper half of the product is not equal to zero.
  - E.g., if AX is multiplied by a 16-bit multiplier, the product is stored in DX:AX. IF the DX is not zero, the Carry and Overflow flags are set.

MUL Instruction - Examples

- 8-bit unsigned multiplication (5 * 10H)

```
mov       al, 5h
mov       bl, 10h
mul       bl    ; CF = 0
```

- 16-bit unsigned multiplication (0100h*2000h)

```
.data
val1  WORD  2000h
val2  WORD  0100h
.code
      mov   ax, val1
      mul   val2  ; CF = 1
```

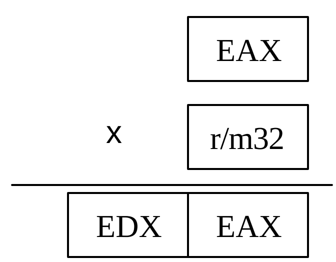
- 32-bit unsigned multiplication (12345h*1000h)

```
      mov   eax, 12345h
      mov   ebx, 1000h
      mul   ebx         ; CF = 1
```

IMUL Instruction

- The **IMUL** instruction multiplies an 8-, 16, or 32-bit ***signed*** operand by either the AL, AX or EAX register (depending on the operand's size).
- The instruction formats are:

```
IMUL   r/m8
IMUL   r/m16
IMUL   r/m32
```



`IMUL` Instruction (continued)

- The **`IMUL`** instruction sets the Carry and Overflow flags if the upper half of the product is not a sign extension of the low-order product.equal to zero.
- E.g., if AX is multiplied by a 16-bit multiplier, the product is stored in DX:AX. IF the AX contains a negative value and the DX is not all 1s, the Carry and Overflow flags are set.

`IMUL` Instruction - Examples

- 8-bit signed multiplication (48 * 4)

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

- 16-bit signed multiplication (-4 * 4)

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

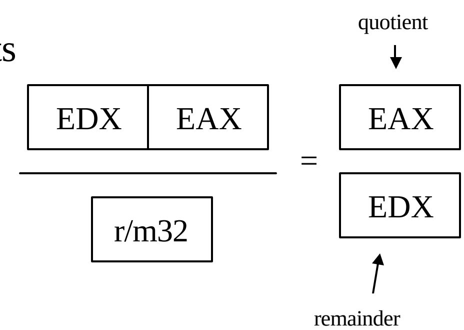
- 32-bit signed multiplication (12345h*1000h)

```
    mov   eax, +4823424
    mov   ebx, -423
    imul  ebx   ; EDX:EAX =
                ; FFFFFFFF86636D80h, OF = 0
```

`DIV` Instruction

- The **`DIV`** instruction divides an 8-, 16, or 32-bit ***unsigned*** divisor into either the AL, AX or EAX register (depending on the operand's size).
- The instruction formats are:

```
DIV     r/m8
DIV     r/m16
DIV     r/m32
```



`DIV` Instruction (continued)

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

DIV Instruction - Examples

- 8-bit unsigned division (83h/2)

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

- 16-bit unsigned division (8003h/100h)

```
    mov   dx, 0
    mov   ax, 8003h
    mov   cx, 100h
    div   cx    ; AX = 0080h, DX = 0003h
```

- 32-bit unsigned division (800300020h/100h

```
.data
dividend    QWORD 0000000800300020h
divisor     DWORD 00000100h
.code
      mov   edx, DWORD ptr dividend+4
      mov   eax, DWORD ptr dividend
      div   divisor
```

CBW, CWD and CDQ Instructions

- **CBW** intends the sign bit of AL into the AH register.
- **CWD** intends the sign bit of AX into the DX register.
- **CDQ** intends the sign bit of EAX into the EDX register.

```
.data
byteVal     SBYTE       -65   ; 9Bh
wordVal     SWORD       -65   ; FF9Bh
dwordVal    SDWORD      -65   ;FFFFFF9Bh
.code
  mov       al, byteVal ; AL = 9Bh
  cbw                   ; AX = FF9Bh
  mov       ax, wordVal ; AX = FF9Bh
  cwd                   ; DX:AX = FFFFFF9Bh
  mov       eax, dwordVal; EAX = FFFFFF9Bh
  cdq             ; EDX:EAX = FFFFFFFFFFFFFF9Bh
```

IDIV Instruction

- The **`IDIV`** instruction divides an 8-, 16, or 32-bit ***signed*** divisor into either the AL, AX or EAX register (depending on the operand's size).
- Signed division requires that the sign bit be extend into the AH, DX or EDX (depending on the operand's size) using **`CBW`**, **`CWD`** or **`CDQ`**.

IDIV Instruction – 8-bit Example

```
.data
byteVal     SBYTE -48
.code
            mov   al, byteVal ; dividend
            cbw               ; extend Al into AH
            mov   bl, 5       ; divisor
            idiv  bl          ; AL = -9, AH = -3
```

IDIV Instruction – 16-bit Example

```
.data
wordVal     SWORD -5000
.code
            mov   ax, wordVal ; dividend, low
            cwd               ; extend AX into DX
            mov   bx, 256     ; divisor
            idiv  bx          ; quotient AX = -19
                              ; rem. DX = -136
```

IDIV Instruction – 32-bit Example

```
.data
wordVal     SWORD -50000
.code
      mov   eax, dwordVal     ; dividend, low
      cdq               ; extend EAX into EDX
      mov   ebx, 256    ; divisor
      idiv  bx          ; quotient EAX = -195
                        ; remainder EDX = -80
```

Divide Overflow

- Divide Overflow occurs when a quotient is too large to fit into the destination operand.

```
mov       ax, 1000h
mov       bl, 10h
div       bl    ; AL can't hold 100h
```

- We are not yet equipped to handle it; the safest thing is to try avoiding it by using a 32-bit divisor.

```
mov       eax, 1000h
mov       ebx, 10h
div       ebx   ; EAX = 00000100h
```

Dividing By 0

- It is fairly easy to handle division by zero:

```
  mov     ax, dividend
  mov     bl, divisor
  cmp     bl, 0
  je      NoDivideZero
  div     bl
  … …
NoDivideZero: … … ; Display error message
```

Implementing Arithmetic Expressions

- Implement var4 = (var1 + var2) * var3

```
  mov     eax, var1
  add     eax, var2
  mul     var3       ; EAX = EAX * var3
  jc      tooBig     ; unsigned overflow?
  mov     var4, eax
  jmp     next
tooBig:         ; display error message
```

Implementing Arithmetic Expressions

- Implement 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
```

Extended Addition and Subtraction Instructions

- Extended addition and subtraction involves adding or subtracting number of almost unlimited size.
- We use the **`ADC`** and **`SBB`** instruction to add with carry or subtract with borrow, extending the operation beyond a single byte, word or doubleword.

ADC Instruction

- **`ADC`** (***Ad***d With ***C***arry) adds the source operand and the carry flag to the destination operand.
- Its formats are the same as the **`mov`** instruction:

```
ADC     reg, reg
ADC     mem, reg
ADC     reg, mem
ADC     mem, imm
ADC     reg, imm
```

Extended Addition Example

```
Extended_Add      PROC
; Calculates the sum of two extended integers
; that are stored as an array of doublewords.
; Receives ESI and EDI point to the two integers.
; EBX points to the a variable that will hold the
; sum.
; ECX indicates the number of doublewords to be
; added.
  pushad
  clc             ; Clear the carry flag
```

```
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
      loop  L1

  adc word ptr [ebx], 0
  popad
  ret
Extended_Add      ENDP
```

SBB Instruction

- **SBB** (***S***u***b***tract With ***B***orrow) subtracts the source operand ***and*** the carry flag from the destination operand.
- Its formats are the same as the **mov** instruction:

```
SBB     reg, reg
SBB     mem, reg
SBB     reg, mem
SBB     mem, imm
SBB     reg, imm
```

SBB Instruction - Example

- Subtracting two 32-bit integers (100000000h - 1)

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