Bit Operations

And Sign Extension

Topics

- Bitwise Boolean Operations
- Shift and Rotate
- Writing Branch-Free Code
- Bit Operations in C/C++/Java
- Counting Bits

Sign Extension Instructions

- Four instructions with implied accumulator operands:
  - CBW - Convert Byte to word (al -> ax)
  - CWD - Convert Word to Dword (AX -> DX:AX)
  - CWDE - Convert Word to DWord Extended (ax -> eax)
  - CDQ - Convert DWord to Qword (eax -> edx:eax)

- Convert Byte to Word (CBW)
- **Syntax:**
```
CBW
```
- **Semantics:**
```
AX <- AL sign extended
Flags   ODITSZAPC unchanged
```

CWD and CWDE

- Convert Word to Double Word CWD
- **Syntax:**
```
CWD
```
- **Semantics:**
```
DX:AX <- AX sign extended
Flags: ODITSZAPC unchanged
```
- Convert Word to Double word Extended (cwde)
- **Syntax:**
```
cwde
```
- **Semantics:**
```
eax <- AX sign extended
Flags: ODITSZAPC unchanged
```

CDQ

- Convert Double word to Quad Word
- **Syntax:**
```
cdq
```
- **Semantics:**
```
edx:eax <- eax sign extended
Flags: ODITSZAPC unchanged
```

- Note that all of these instructions use the accumulator as an implied operand (including dx and/or edx)
- With the 386 processor two more general instructions were added:
```
movsx       Move with sign-extension
movzx       Move with zero-extension
```

MOVSX and MOVZX

- **Syntax:**
```
MOVSX dest, source
MOVZX dest, source
```
- **Semantics:**
```
MOVSX sign-extends source into destination
MOVZX zero-extends source into destination
Flags: ODITSZAPC unchanged
```

- **Operands:** - note that dest is ALWAYS a register
```
reg16, reg8 or mem8
reg32, reg8 or mem8
reg32, reg16 or mem16
```

Operation on the Carry Flag

- The Carry Flag (CF) is one of three flags with instructions for direct modifications
  - The other two are status flags: Direction Flag DF and Interrupt Enable flag ID
- CF is used extensively in multiword arithmetic and with instructions that work with individual bits
- CF operations are Clear Carry (CLC) , Set Carry (STC) and Complement Carry (CMC)

```
CLC Syntax:   CLC   Semantics:     CF <- 0
STC Syntax:   STC   Semantics:     CF <- 1
CMC Syntax:   CMC   Semantics:     CF <- CF-1
```

Shifts and Rotates

- Many high level languages provide shift operators, but none (to my knowledge) provide rotate operators
  - Shifting shifts bits in an operand from left to right or from right to left
  - Rotates rotate bits in an operand from left to right or from right to left
- In all of the curly brace languages (C, C++, Java, Javascript) << and >> are shift left and shift right, respectively
  - Shifts can be LOGICAL (zero-fill) or ARITHMETIC (sign-preserving)
  - C and C++ support signed and unsigned arithmetic and shift operators behave accordingly
  - Java and Javascript do not support unsigned arithmetic so all shifts are sign-preserving

Shifting and Multiplication / Division

- In general shifting bits corresponds to multiplication (left shift) or division (right shift) by a power of 2.
  - Arithmetic right shift preserves the top-order bit so that the result is mathematically correct for signed integers.
  - A logical right shift does not preserve the top-order bit, so the result is correct for unsigned integers
- Although x86 assembly has mnemonics for both logical (SHL) and arithmetic left shifts (SAL) they are in fact the same instruction
- Note that shifting left to multiply by powers of two only is correct if:
  - -for integers with 0's in high order bits, the H.O. bit remains 0
  - -for signed negative integers, the H.O. bit remains 1

Shift Instructions

- `SAL Destination, Count (shift arithmetic left)`
- `SHL Destination, Count (shift logical left)`

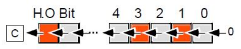
- `SAR Destination, Count (shift arithmetic right)`
- `SHR Destination, Count (shift logical right)`

- In 32-bit processors an additional (and somewhat unusual 3-operand shift is available):

```
shrd dest, source, count
shld dest, source, count
```

SAL and SHL

- These two instructions are identical and operate on registers or memory
  - SAL stands for Shift Arithmetical Left and SHL stands for SHift logical Left--This operation corresponds to multiplication by 2
  - This works for both signed and unsigned numbers--provided the signed result is in range
  - Often used for high-speed multiplication
- Left shift follows the pattern below



Effects on Flags

- Effects on flags

```
if (count == 1)
   ....SZ.PC    modified for result
   O........    O set if CF != new MSB
   ......A..    Undefined
   .DIT.....    Unchanged
else
   ........C    modified for result
   O...SZAP.    Undefined
   .DIT.....    Unchanged
```

Syntax

- SAL and SHL have three syntactic versions
- **`SAL Mem/Reg, 1`**
  - The immediate value 1 is not actually part of the instruction
- **`SAL Mem/Reg, CL`**
  - Note the use of the CL register as shift counter
- **`SAL Mem/Reg, imm   (80186 and later)`**
  - Not the same machine language as **`SAL mem/reg 1`**

Uses

- Shifting has many uses: bit inspection (inspect CF), fast arithmetic, pack bytes into nibbles, etc.
- Packing bytes:

```
shl     ah, 4   ; move l.o. 4 bits to upper nibble
or      al, ah  ; merge into AL
```

- Fast arithmetic: what is the common English term for this operation?

```
mov eax, number    ; 2 clocks if n in cache
sal eax, 2         ; 1 clock
add eax, number    ; 1 clock
sal eax, 1         ; 1 clock
```

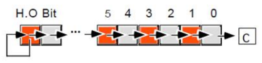
- Note that MUL is 70-150 clocks on the 8086 and still 10 clocks on the Pentium

SAR and SHR

- SAR AND SHR
  - Shift Arithmetic Right and SHift logical Right
  - Unlike SAL and SHL, these two instructions are different
- Syntax same as SAL/SHL:

```
SAR Dest, 1     SHR Dest, 1
SAR Dest, CL    SHR Dest, CL
SAR Dest, immed SHR Dest, immed
```

- SAR preserves the sign of the number by keeping the high-order bit unchanged.



SAR and SHR

•Same effect on flags as SHL/SAL

If shifting by 1 OF will reflect a sign change.

CF, ZF, PF and SF are affected by shifts of 1 bit

•Note that with a shift of 1 CF has the remainder after division by 2

Rotates

- Rotates are circular shifts, but they are not used for mathematical purposes
- Rotates can either be left or right, with or without the carry flag

```
RCL Destination, Count
ROL Destination, Count
RCR Destination, Count
ROR Destination, Count
```

- Like the shifts you can either rotate by 1 or by using the CL register

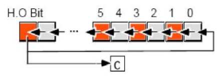
ROL and RCL

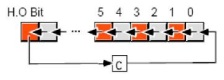
- The only flags affected are CF and OF
- The changes to CF will be seen below, while OF is 1 if the new high-order bit is different from the old.
  - The figures below illustrate these operations.
- Unlike a shift instruction bits are never lost.
- Note that ROL is an 8, 16, or 32 bit rotate and RCL is a 9, 17 or 33 bit rotate
- Using ROL to exchange nibbles:

```
rol al,4
```

- or words:

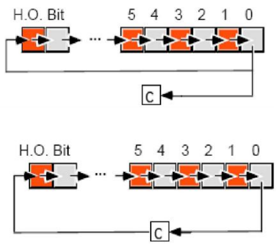
```
rol eax, 16  ; now h.o. word is in AX
```





ROR and RCR

- Like ROL and RCL except the direction is different.



SHLD and SHRD (80386+)

- **Syntax**

```
SHLD dest, source, count
SHRD dest, source, count
```

- **Semantics**

```
dest shifted L/R by count and filled from source
SHLD: bits are copied from MSBs of source
SHRD: bits are copied from LSBs of souce
source unmodified
Flags:
   S,Z modified for result
   O,A,P,C undefined
```

- **Operands**

```
reg, reg, imm      mem, reg, imm
reg, reg, CL       mem, reg, CL
```

Shift and Rotate Examples

- Multi-word shifts
- Displaying characters in binary
- Isolate bit fields

Displaying Characters in Binary

```
AsciiBinary:
  ; parameters AL     byte to convert
  ; ebx  pointer to 8-byte string for result
  ; returns all registers unmodified
  push ebx
  push ecx
  mov ecx, 8
L1:
  mov byte [ebx], '0' ; assume zero bit
  rol al, 1           ; get msb first
  adc byte [ebx],0    ; add in carry
  inc ebx
  loop L1
  pop ecx
  pop ebx
  ret
```

Example Call

```
foo db 0
asc TIMES 9 db 0 ; 8 bytes for ascii + 0
terminator
...
mov al, [foo]  ; load byte
mov ebx, asc   ; string addr in ebx
call asciibinary
mov eax, asc
call print_string
```

Multibyte Shifts

```
segment .bss
bigdata resd 4    ; 128 bit integer


…
  ; we'll shift bigdata left by 4,
 discarding h.o. bits
  ; note little-endian ordering of dwords
  mov ecx, 4   ; loop counter
shift_loop:
  shl [bigdata],1 ; msb in cf
  rcl [bigdata+4]    ; msb of dword[0] in
 lsb dword[1]
  rcl [bigdata+8]    ; msb of dword[1] in
 lsb dword[2]
```

Isolate Bit Fields

- A FAT filing system directory timestamp is a 16-bit structure:

```
y y y y y y y m m m m d d d d d
```

- Where
  - Year is 7 bits (0 = 1980, 1=1981, etc)
  - Month is 4 bits
  - Day is 5 bits
- Example: July 4, 2000 = F0D4h

```
1111000011100100
Y= 111 1000  M = 0111 D = 00100
 = 120+1980     = July   = 4
```

Bit Fields

```
year     dw ?
month    db ?
day      db ?
timestamp dw ?

; get day
mov ax, timestamp
and al, 1fh         ; mask off all but lower 5 bits
mov day, al
mov ax, timestamp
shr ax, 5           ; shift month to low order bits
and al, 0Fh         ; and mask
mov month, al
mov ax, timestamp
mov al, ah          ; copy year to low byte
shr al, 1           ; shift to correct position
sub ah, ah          ; zero out ah
add ax, 1980        ; add the beginning of the world
mov year, ax
```

ADC and SBB

- We've discussed these before and indicated their use in signed multiple precision arithmetic
- A straight binary add has a carry from bit to bit
- All we need to to generalize is to propagate the carry between bytes or words
- We can easily extended addition and subtraction to binary integers of any length

Generalized Multiword Addition

```
Multiword_Add:
  ; Parameters: ecx: operand size in dwords
  ;             esi, edi: address of source operands
  ;             ebx:address of result. ecx+1 words in length
  pusha            ; we use 5 registers, save all
  clc              ; ensure cf clear for carries
L1:
  mov eax, [edi]       ; get first operand
  adc eax, [esi]       ; add second operand with prev carry
  mov [ebx], eax       ; save result
  pushf                ; save cf
  add di, 4            ; advance all 3 pointers
  add si, 4
  add bx, 4
  popf                 ; restore cf
  loop L1
  ; may have carry left over
  mov dword [ebx], 0   ; clear msb
  adc dword [ebx], 0   ; add in carry
  popa
  ret
```

Example Call

```
segment .data
  op1 dd 12345678h, 5ABBCCDDh, 09080706h
  op2 dd 83756567h, 17173545h, 33221155h
segment .bss
  ans resd 4

  mov edi, op1
  mov esi, op2
  mov ecx, 3
  mov bx, ans
  call multiword_add
```

Counting Set Bits in a Register

- Consider this code

```
; Count the number of bits that are set in eax
sub bl, bl ; bl will contain the count of ON bits
mov ecx, 32 ; ecx is the loop counter
count_loop:
  shl eax, 1 ; shift bit into carry flag
  jnc skip_inc ; if CF == 0, goto skip_inc
  inc bl
skip_inc:
loop count_loop
```

- We can simplify (and preserve eax)

```
sub bl, bl ; bl will contain the count of ON bits
mov ecx, 32 ; ecx is the loop counter
count_loop:
  rol eax, 1 ; shift bit into carry flag
  adc bl,0
Loop count_loop
```

AND

- **Syntax:**

```
AND dest, source
```

- **Semantics:**

```
dest <- dest AND source
flags:  ....SZ.P   modified for result
        O......C   cleared
        ......A.   undefined
        .DIT....   unchanged
```

- **Operands:**

```
reg,reg      mem,reg      reg,mem
reg,immed    mem,immed
```

AND

- Used for clearing individual bits in an operand (masking)
- Put a 0 in each bit position that you want to clear
- Example: Convert ASCII digit in AL to binary number:
  - AND AL, CFh ;1100 1111b
- Note the effects on the flags

```
flags:  ....SZ.P   modified for result
        O......C   cleared
        ......A.   undefined
        .DIT....   unchanged
```

- S, Z, P carry useful information
- O C are cleared always, so carry no information

TEST

- **Syntax:**

```
TEST dest, source
```

- **Semantics:**

```
compute (dest AND source) and modify flags
flags:  ....SZ.P   modified for result
        O......C   cleared
        ......A.   undefined
        .DIT....   unchanged
```

- **Operands:**

```
reg,reg      mem,reg      reg,mem
reg,immed    mem,immed
```

- TEST is an AND operation that modifies flags only and does not affect the destination
- TEST / AND is like CMP / SUB
- Typically used with conditional jumps

OR

- **Syntax:**

```
OR dest,source
```

- **Semantics:**

```
dest <- dest OR source
flags:  ....SZ.P   modified for result
        O......C   cleared
        ......A.   undefined
        .DIT....   unchanged
```

- **Operands:**

```
reg,reg      mem,reg      reg,mem
reg,immed    mem,immed
```

- Notes:
  - Used for setting individual bits in an operand (masking)
  - Put a 1 in each bit position that you want to set
  - With AND we mask with 0's; with OR we mask with 1's
  - This is the conventional inclusive OR

XOR

- XOR is eXclusive OR
- Compare truth tables:

OR	T	F
T	T	T
F	T	F

XOR	T	F
T	F	T
F	T	F

- **Syntax:**

```
XOR dest, source
```

- **Semantics:**

```
dest <- dest XOR source
flags:  ....SZ.P   modified for result
        O......C   cleared
        ......A.   undefined
        .DIT....   unchanged
```

- **Operands:**

```
reg,reg      mem,reg      reg,mem
reg,immed    mem,immed
```

XOR

- Used to complement bits
- Create a mask that has a 1 in each position to be complemented and a 0 in each position to be left unchanged

```
         1011 1100
    XOR  0110 0110
         1101 1010
```

- Can be used to clear a register: XOR AX,AX
  - Same effect and speed as subtraction: SUB AX,AX
  - But either is faster than MOV AX,0
- XOR over a word computes an even parity bit

```
1011 1100
```

- This word has 5 1-bits, so the XOR is 1

XOR and Messages

- XOR has some interesting properties that make it very useful with cryptography
- (A xor B) xor A = B

```
        1011 1100   A
XOR     0110 0110   B
=       1101 1010
XOR     1011 1100   A
=       0110 0110   B
```

- So if A is a message, and B is a secret key, A xor B is an encrypted message and (A xor B) xor A is the decrypted message

Steganography

- Steganography is the hiding of a message inside another object in such a way that the presence of the message is imperceptible
- XOR can be used to encode messages into innocuous digital objects such as bitmaps or sound files
  - A bitmap has 3 color channels plus an alpha channel (used for opacity)
  - Changes to the low order bit of color channels are not readily detectable by the human eye
  - A message can be hidden by xoring the message with the low order bits of color bytes
- Off-topic: but interesting: many color printer manufacturers now encode steganographic printer identification data in tiny yellow dots that are invisible to the human eye
  - See http://w2.eff.org/Privacy/printers/docucolor/

Other uses of Boolean Ops

- Sometimes Boolean operators are used for purposes other than modifying or inspecting bits
- With a value in a register
  `and reg, reg    OR or reg, reg OR test reg, reg`
- Will set PSZ flags that can tested with a conditional jump. Operand is not modified
- XOR can be used in place of NOT
  `xor ax, 0ffffh`
- This can be useful because xor modifies PSZ flags but NOT does not
- `XOR reg, reg` sets reg to 0, same as `SUB reg, reg`

NOT

- Used to logically negate or complement an operand
- Same effect can be achieved with XOR but if the entire operand has to be complemented then NOT is faster
- **Syntax:**
  `NOT dest`
- **Semantics:**
  `dest <- NOT dest`
  `Flags:  ODITSZAPC unchanged`
- **Operands:**
  reg   mem
- Notes:
  1. NOT is a unary operation
  2. NOT is logical negation (1's complement); every bit is inverted; NEG is the two's complement negation
  3. Unlike other boolean operations NOT has no effect on the flags

Conditional SETcc Instructions

- These instructions were added to the 80386 instruction set for a number of reasons but they can be quite useful in avoiding branch instructions and therefore pipeline stalls
- The conditional SETs take a single byte destination operand and write a 1 to the operand if condition is true, 0 if false.
- SETcc Conditions are same as conditional jump
- Note that
  ```
  SETcc AL
  DEC AL
  ```
  Leaves AL with all 1's or all 0's

SETcc

- (SETcc) instructions:
  seta, setae, setb, setbe, setc, sete, setg, setge, setl, setle,setna, setnae, setnb, setnbe, setnc, setne, setng, setnge, setnl, setnle, setno, setnp, setns, setnz, seto, setp, setpe, setpo, sets, setz
- Syntax:
  `SETcc dest`
- Semantics:
  `dest <- 1 if condition true, 0 otherwise (See Jcc)`
- Flags:
  `ODITSZAPC unchanged`
- Operands:
  `reg8 mem8`

Find MAX Without Branches

```
%include "asm_io.inc"
segment .data
message1 db "Enter a number: ",0
message2 db "Enter another number: ", 0
message3 db "The larger number is: ", 0
segment .bss
input1  resd    1        ; first number entered

segment .text
        global  _asm_main
_asm_main:
        enter   0,0           ; setup routine
        pusha
        mov     eax, message1 ; print out first message
        call    print_string
        call    read_int      ; input first number
        mov     [input1], eax

        mov     eax, message2 ; print out second message
        call    print_string
        call    read_int       ; second number (in eax)
```

max.asm:2

```
xor  ebx, ebx    ; ebx = 0
cmp eax,[input1] ; compare second and first number
setg bl          ; ebx= (input2 > input1) ?  1 : 0
neg ebx          ; ebx= (input2 > input1) ? 0xFFFFFFFF : 0
mov ecx, ebx     ; ecx= (input2 > input1) ? 0xFFFFFFFF : 0
and ecx, eax     ; ecx= (input2 > input1) ? input2 : 0
not ebx          ; ebx= (input2 > input1) ?  0 : 0xFFFFFFFF
and ebx,[input1] ; ebx= (input2 > input1) ?  0 : input1
or   ecx, ebx    ; ecx= (input2 > input1) ?  input2 : input1
```

Bit Test Instructions

- A set of four instructions that test a single, specified bit in register or memory and copy it to CF
- BT (Bit Test)
  - Test a specified bit by copying it to CF
- BTS (Bit Test and Set)
  - BTS will also set the bit after it is copied
- BTR (Bit Test and Reset)
  - BTR will clear (reset) the bit after it is copied
- BTC (Bit Test and Complement)
  - BTC will complement the bit after it is copied

BT, BTS, BTR, BTC

Syntax:

```
BT dest, source   BTR dest, source
BTS dest, source  BTC dest, source
```

Semantics:

```
CF <- bit in dest specified by source;
dest modified by BTS, BTR, BTC as specified above
```

Flags:

```
.........C gets copy of bit from dest
ODITSZAP. unchanged
```

Operands:

```
reg, reg mem, reg
reg, imm8 reg, imm8
```