Seminar 2

Signed and unsigned instructions. Arithmetic instructions (multiplications and divisions). Signed and unsigned conversions.

Contents

Seminar 2 1

1. Signed and unsigned instructions 1

2. (Signed and unsigned) multiplication and divisions instructions 2

3. Signed and unsigned conversions 5

1. Signed and unsigned instructions

On the IA-32 architecture, related to the unsigned and signed representation of numbers, there are 3 classes of instructions:

- instructions which do not care about signed or unsigned representation of numbers: **mov, add, sub**
- instructions which interpret the operands as unsigned numbers: **div, mul**
- instructions which interpret the operands as signed numbers: **idiv, imul, cbw, cwd, cwde**

It is important to be consistent when developing a IA-32 assembly program: either consider all numerical values in a program to be unsigned (in which case you should use only instructions from class 1 and 2) or consider all numerical values in a program to be signed (in which case you should use only instructions from class 1 and 3).

Important rules that must be obeyed by arithmetic instructions with 2 operands:

- all operands must have the same size/type (i.e. you can add byte to byte, but not byte to a word)
- at least one of the operands must be a general register or a constant and if it is a constant, this constant can not appear as a destination operand

Related to the above two rules, let's assume we have the following code:

```
a db 10
b db 11
……..
add ax, [a]
add [a], [b]
```

The instruction *add [a], [b]* would fail, meaning that there will be a compile error (i.e. assembly error) and the executable file cannot be built. This is because that instruction does not obey the second rule from above ([a] and [b] are both memory references / variables). On the other hand, although the instruction *add ax, [a]* breaks rule number one from above (since AX is a word and [a] was declared as byte), the compiler will not complain and will build the executable file! But when this instruction gets executed, it will not do what you meant: add the byte "a" to the register "AX"! Instead it will add a *word from the memory that starts where variable "a" starts* (this word is composed from the bytes 10 and 11) to the register AX. This is because although variable "a" was declared as a byte using the Define Byte (DB) directive, the NASM assembler does not link the type (data size) to a memory location in instructions. The declaration "a db 10" just declares/reserves 1 byte at the current memory address. You can then write in your code *mov ax, [a]* and the assembler will tell the CPU to move a word starting in the memory at the address of "a" into AX. Or you can write the code *mov eax, [a]* and the assembler will tell the CPU to move a doubleword starting in the memory at the address of "a" into EAX. *[a]* means just the starting point in the memory of a data – it does not say anything about the type (size) of "a". The type (size) information is inferred from the other operand of the instruction together with the first rule described above: all operands must have the same size/type.

Obs. "[ ]" is the addressing operator and its effect can be explained in the following two examples:

mov EAX, [a] ⇒ moves in EAX a doubleword starting at the address of variable "a" (i.e. it moves in EAX the value of variable "a", assuming "a" was declared as a doubleword)

mov EAX, a ⇒ moves in EAX the starting address (i.e. the offset – you will see later what this means) of variable "a" (NOT the value of variable "a" !)

2. (Signed and unsigned) multiplication and divisions instructions

MUL – unsigned multiplication instruction

Syntax: mul source

(where source is either register or variable of type byte, word or dword)

Effect: - if source is a byte ⇒ AX =AL * source

- if source is a word ⇒ DX:AX = AX * source

- if source is a dword ⇒ EDX:EAX = EAX * source

Example: The instruction *mul BX* stores in two 16-bit registers the result of the multiplication which is a 32-bit number. More specifically, the effect of this instruction is: DX:AX:= AX*BX. The result of the multiplication (a 32-bit number) is stored in the registers DX and AX instead of a proper 32-bit register like EAX for compatibility reasons with the previous Intel 8086 computing architecture. Let's assume that the result of the above multiplication would be the number 12345678h (in the hexadecimal base). The least significant (low) 16 bits of this number would be stored in AX and the most significant (high) 16 bits of this number would be stored in DX. Knowing that a hexadecimal digit is represented on 4 bits, we conclude that AX would store 5678h and DX would store 1234h (the hexadecimal number 1234h occupies 16 bits).

DIV – unsigned division instruction

Syntax: div source

(where source is either register or variable of type byte, word or dword)

Effect: - if source is a byte ⇒ AL =AX / source (quotient/catul) and

AH:=AX % source (remainder/restul)

- if source is a word ⇒ AX =DX:AX / source (quotient) and

DX:=DX:AX % source (remainder)

- if source is a dword ⇒ EAX = EDX:EAX / source (quotient) and

EDX = EDX:EAX % source (remainder)

IMUL – does the same thing as MUL but considers the operands as signed numbers

IDIV – does the same thing as DIV but considers the operands as signed numbers

Examples

Ex1. Compute the value of the expression x = ((a + b) * c) / d where all numbers are unsigned numbers and a, b, c, d are all bytes.

```
;
; BEGIN 32 bits PROGRAM
;
bits 32
; declare the EntryPoint (a label defining the very first instruction of the program)
global start

; declare external functions needed by our program
extern exit                 ; tell nasm that exit exists even if we won't be defining it
import exit msvcrt.dll      ; exit is a function that ends the calling process. It is defined in msvcrt.dll
```

```
; our data is declared here (the variables needed by our program)
segment data use32 class=data
        a db 3
        b db 4
        c db 2
        d db 3
        x db 0

; our code starts here
segment code use32 class=code
start:
        mov al, [a]                     ; AL:=a = 3
        add al, [b]                     ; AL:=AL+b = 3+4 = 7
        mul byte [c]                    ; AX:=AL*c = 7*2 = 14
                                        ; for this mul instruction we had to specify the type of operand [c]
                                        ; (i.e. byte), so that mul knows what to do. Remember, "c" is just
                                        ; a memory reference, it does not have a type associated to it!
        div byte [d]                    ; AL:=AX / d = 14 / 3 = 4      AH:=AX % d = 14 % 3 = 2
                                        ; similar to the above mul, we had to explicitly specify the type of
                                        ; [d] (i.e. byte)
        mov [x], AL                     ; x:=AL = 4

        ; exit(0)
        push    dword 0         ; push the parameter for exit onto the stack
        call    [exit]              ; call exit to terminate the program
```

3. Signed and unsigned conversions

Unsigned conversion: place zeroes in the high part ("manually")

How do we convert AL to AX?

- MOV AH, 0

How do we convert BX to CX:BX?

- MOV CX, 0

How do we convert AX to EAX?

```
MOV BX, AX
MOV EAX, 0   ;because we cannot directly access the high word from EAX!
MOV AX, BX
```

Ex2. Compute the value of the expression x = (a – b * c) / d where all numbers are unsigned numbers and a, b, c, d are all bytes.

```
bits 32
; declare the EntryPoint (a label defining the very first instruction of the program)
global start
; declare external functions needed by our program
extern exit                ; tell nasm that exit exists even if we won't be defining it
import exit msvcrt.dll     ; exit is a function that ends the calling process. It is defined in msvcrt.dll

; our data is declared here (the variables needed by our program)
segment data use32 class=data
        a db 30
        b db 4
        c db 2
        d db 3
        x db 0

; our code starts here
segment code use32 class=code
start:
```

```
mov al, [b]                    ; AL:=b = 4
mul byte [c]                   ; AX:=AL*c = 4*2 = 8
mov bl, [a]                    ; BL:=a=30
; Now we need to substract AX from BL, but we can not do that directly due to the rule that both
; operands of sub must have the same type/size. In such situations we always convert the smallest
; type to the larger one (i.e. we convert BL from byte to word). BL:=30=0001 1110b. The number 30
; represented unsigned on 16 bits looks like this: 0000 0000 0001 1110b. So we see that the only
; difference in the unsigned representation between the representation of 30 on 8 bits and the
; representation of 30 on 16 bits, is the fact that 30 on 16 bits has an additional 8 zero bits in the
; front. This is why the unsigned conversion of a byte/word/dword is realized by adding non
; significant zeroes in front of the number.
mov bh, 0                      ; BX:=0000 0000 0001 1110b
                               ; we converted unsigned BL to BX
sub bx, ax                     ; BX:= BX-AX = 30 – 8 = 22
mov ax, bx                     ; AX:=BX=22
div byte [d]                   ; AL:=AX / d =22 / 3 = 7 (quotient)    AH:=AX % d =22 % 3 = 1 (remainder)
mov [x], AL                    ; x:=AL=7

; exit(0)
push    dword 0          ; push the parameter for exit onto the stack
call    [exit]             ; call exit to terminate the program
```

In the above example we have seen that in order to convert unsigned BL to BX, we just added 8 non significant zeros to BH (by moving zero to BH). For the signed representation however, there is a different story. If the 8-bit number is positive, signed converting this number to 16 bits is the same as for unsigned representations: just put 8 non significant zeros in front of it (in the high part). Example:

```
mov  al, 12
mov ah, 0
```

Now AX=12= 0000 0000 0000 1100b (AH=0000 0000b and AL=0000 1100b).

But if the number is negative, we have to put 8 digits of 1 in front of it. See below:

```
mov al, -12                ; AL:=1111 0100b
```

; -12 represented on 16 bits is: 1111 1111 **1111 0100**b

Because for the signed representation, we have two cases (when the number is positive and when the number is negative), we have specialized instructions that perform the signed conversion (i.e. these instructions either add 8 (or 16 or 32) zero bits in front of the number if the number is positive or they add 8 (or 16 or 32) one bit in front of the number if the number is negative); in other words, these instructions add the sign bit of the number 8 (or 16 or 32) times in front of the number. These instructions are presented below.

CBW – (signed) convert byte to word

Syntax: cbw

Effect: converts signed AL to AX

CWD – (signed) convert word to dword (doubleword)

Syntax: cwd

Effect: converts signed AX to DX:AX

CWDE – (signed) convert word to dword extended

Syntax: cwde

Effect: converts signed AX to EAX

CDQ – (signed) convert doubleword to quadword

Syntax: cdq

Effect: converts signed EAX to EDX:EAX

To summarize the unsigned conversions (we do not have specialized instructions for this):

- unsigned convert AL to AX: mov ah, 0
- unsigned convert AX to DX:AX : mov dx, 0
- unsigned convert EAX to EDX:EAX: mov edx, 0

Ex. 3 Compute the value of the expression (a * b) / d – c where all numbers are signed and *a, c, d* are bytes and *b* is a word.

```
bits 32
global start
extern exit
import exit msvcrt.dll

; our data is declared here (the variables needed by our program)
segment data use32 class=data
        a db -3
        b dw 4
        c db 2
        d db 3
        x dw 0

; our code starts here
segment code use32 class=code
start:
        mov al, [a]     ; AL:=a = -3
        cbw             ; AX:=-3    (convert AL to AX signed)
        imul word [b]   ; DX:AX:= AX * B = -3 * 4 = -12
        ; we need to convert “d” from byte to word
        mov bx, ax      ; DX:BX:=-12
        mov al, [d]     ; AL:=d = 3
        cbw             ; AX:=AL=3
        mov cx, ax      ; CX:=3
        mov ax, bx      ; move BX back to AX so that DX:AX=-12
        idiv cx         ; AX:=DX:AX / CX = -12 / 3 = -4       DX:=DX:AX % CX = -12 % 4 = 0
        ; we must convert “c” from byte to word; first clear the AX register
        mov bx, ax      ; BX:=AX=-4
        mov al, [c]     ; AL:=c=2
```

```
 cbw                ; AX:=AL=2
 sub BX, AX         ;BX:=BX-AX = -4 -2 = -6
 mov [x], BX        ; x:=-6


; exit(0)
push    dword 0             ; push the parameter for exit onto the stack
call    [exit]                ; call exit to terminate the program
```