**Seminar II. Converting numbers between numbering bases 2, 10, 16. Representation of integer numbers in the computer’s memory. Signed and unsigned instructions. Arithmetic instructions. Signed and unsigned conversions.**

**I.1. Converting numbers between numbering bases 2, 10, 16**

A number is converted from a *source base* to a *destination base*. There are two algorithms that are mostly used for converting a natural number between numbering bases: one is based upon successive division operations and the other is based on successive multiplication operations.

The conversion algorithm that uses successive divisions

* This algorithm is useful when converting a number from base 10 to another numbering base (because computations are done in the source base; i.e. base 10)
* The initial number is continuously divided to the destination base (i.e. the initial number is divided to the destination base, then the obtained quotient (romanian: catul) is divided to the destination base and so on..) until we get a zero quotient. The remainders (Romanian: resturile) obtained taken in the reverse order form the representation of the initial number in the destination base

Ex.1: The representation of the number 23 (currently written in base 10) in the new base 2 is: 10111.

Ex.2: The representation of the number 28 (currently written in base 10) in the new base 16 is: 1C (because the digit representing 12 in base 16 is **C**).

The conversion algorithm that uses successive multiplications

* This algorithm is useful when converting a number from base different than 10 to base 10 (because computations are done in the destination base; i.e. base 10)
* Considering that the representation of the number in base *s* is: *anan-1… a1a0*, the representation of this number in the destination base *d* will be computed like this:

*an \*s^n + an-1\*s^(n-1) +… +a1 \*s^1 + a0\*s^0*

where computations are done in base *d*.

Ex.1: The representation of the number 10111 (currently written in base 2) in the new base 10 is: 1\*2^4 + 0\*2^3 + 1\*2^2 + 1\*2^1 + 1\*2^0 = 23

Ex.2: The representation of the number 1C (currently written in base 16) in the new base 10 is: 1\*16^1 + 12\*16^0 = 28

Table useful for performing fast conversions between bases 2, 10 and 16

The following table will be useful for performing fast conversions of a semi-byte (a 4 bits/binary digits number) from base 2 to 10 and 16 and reverse.

|  |  |  |
| --- | --- | --- |
| **Base 2** | **Base 10** | **Base 16** |
| 0000 | 0 | 0 |
| 0001 | 1 | 1 |
| 0010 | 2 | 2 |
| 0011 | 3 | 3 |
| 0100 | 4 | 4 |
| 0101 | 5 | 5 |
| 0110 | 6 | 6 |
| 0111 | 7 | 7 |
| 1000 | 8 | 8 |
| 1001 | 9 | 9 |
| 1010 | 10 | A |
| 1011 | 11 | B |
| 1100 | 12 | C |
| 1101 | 13 | D |
| 1110 | 14 | E |
| 1111 | 15 | F |

**Binary digit** = **Bit** (a software concept that represents the smallest quantity of information)

**I.2. Representation of integer numbers in the computer’s memory**

Consider the following instruction:

mov ax, 7

The above instruction instructs the CPU (i.e. microprocessor) to set the value of the ax register (which is a memory zone on the CPU) to 7. A natural question that arises is: how does the CPU *represent integer numbers in the memory (and also on the CPU registers)* ?

The CPU represents an integer number on 1, 2, 4 or 8 bytes on the IA-32 architecture (1 byte = 8 consecutive bits). In fact, there are two kinds of representation of integer numbers in the computer’s memory: signed representation and unsigned representation. *The CPU choses one of these two representations depending on the specific instruction it executes*.

Unsigned representation of numbers

* in unsigned representation we can only represent positive natural numbers
* the unsigned representation of a positive number is equal to the representation of that number in base 2
* ex.1: the unsigned representation of 17 on 8 bits is : 0001 0001
* ex.2: the unsigned representation of 39 on 8 bits is : 0010 0111

Signed representation of numbers

* in the signed representation we can represent positive and negative integer numbers
* the signed representation of a positive number is equal to the unsigned representation of that number (i.e. it is equal to the representation of that number in base 2)
* the signed representation of a negative number is equal to the representation in *2’s complementary code* (Romanian: codul complementar fata de 2) of that number; in order to obtain the *2’s complementary code* of a negative number, we substract the absolute value of the number (Romanian: modulul numarului) from 1 followed by as many zeroes as needed in order to represent the absolute value of the number.
* in the signed representation, the most significant bit (i.e. binary digit) of the representation is the sign bit(1=negative number; 0=positive number).
* ex.1: the signed representation of 17 on 8 bits is : 0001 0001 (the most significant bit is 0, so the number is positive)
* ex.2: the signed representation of -17 on 8 bits is : 1110 1111 (note that the sign bit is 1 in this case, so the number is negative)

1 0000 0000 –

1 0001

1110 1111

* ex.3: the signed representation of -39 on 16 bits is : 1111 1111 1101 1001 (note that the sign bit is 1 in this case, so the number is negative)

1 0000 0000 0000 0000–

10 0111

1111 1111 1101 1001

Now, we can consider the reverse problem of “representation”, that is “interpretation”. Let’s assume that the binary content of the AL register is 1110 1111 and the next instruction to be executed is:

mul bl

The mul instruction just multiplies the value from the AL register with the value from the BL register and stores the result in AX (more details about the mul instruction will be given below). When the CPU executes the above instruction it needs to ask itself the question (the human programmer also asks himself the same question): *what integer number does the sequence of bits from AL (i.e. 1110 1111) represents in our conventional numbering system (i.e. base 10)*? The CPU must *interpret* the sequence of bits from AL into a number in order to perform the mathematical operation (multiplication).

Just like we have two types of “representations”, we also have two corresponding types of “interpretations”: signed interpretation and unsigned interpretation.

In our example where we have in the AL register the sequence of 8 bits: 1110 1111, this value can be interpreted:

* unsigned: in this case, we know that in the unsigned representation, only positive numbers are represented, so our sequence of bits represents a positive number and it is the representation in base 2 of that number; so the number in base 10 is:

1\*2^7 + 1\*2^6 + 1\*2^5 + 0\*2^4 + 1\*2^3 + 1\* 2^2 + 1\*2^1 + 1\*2^0=

128 + 64 + 32 + 0 + 8 + 4 + 2 + 1 = 239

* signed: in this case, we know that in the signed representation the most significant bit of the representation is the sign bit; our sign bit is 1 which means that this is the signed representation of a negative number; in other words this is the complementary code representation of the negative number; in order to obtain the *direct code* (the representation in base 2 of the absolute value of the number) we use the following rule: *take all the bits of the complementary code representation, from right to left, keep all the bits until the first 1, including this one, and reverse the remaining bits (1 becomes 0, 0 becomes 1)*. So, for our example of 1110 1111, the direct code is: 0001 0001 = 17. So the sequence of 8 bits 1110 1111 from the memory is interpreted signed into the number -17.

**I.3. 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 unsigned (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 can not 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.

**I.4. (Signed and unsigned) Arithmetic 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 numbers. 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**  and **IDIV** – does the same thing as MUL and 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

**I.5. Signed and unsigned conversions**

**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 1110**b. The number 30

; represented unsigned on 16 bits looks like this: 0000 0000 **0001 1110**b. 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 0100**b

; -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** – (singned) convert word to dword extended

*Syntax*: cwde

*Effect*: converts signed AX to EAX

**CDQ** – (singned) 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