Curs ASC

# 30.09.2021

* C is a high-level programming language, but is the most low-level-high-level prog. lang.
* Only in C you can work with pointer arithmetics. Și de-aia cică îi jmk.

The x86 Microprocessor

* A register it’s the smallest and the closest unit of storage sau cv.
* Register – storage capacities – very small in terms of sizes (8,16,32 or 64 bits) but very fast as acces speed. Are used for temporary storage of the operands (data, commands codes – instructions-, ADDRESSES !!) with which a processor currently. Tre sa le tratam echitabil
* EU – EXECUTIVE UNIT - runs the instructions
* ALU – ARITHMETIC and LOGIC UNIT face toata treaba.

ARITH> Addition, subtraction, multpl, & division ONLY with INT numbers.

LOGIC> OR, AND & NOT (decisions, binary logic)

8 so-called general registers

EAX – ext. AX – Acumulator register

EBX - ext. BX – Base register

ECX - ext. CX – Counter register

EDX - ext. DX – Data register

ESP - ext. SP – Stack pointer

EBP - ext. BP – Base pointer (base of stack)

EDI - ext. DI – Destination Index

ESI - ext. SI – Source Index

* BIU – BUS INTERFACE UNIT – prepares the execution of every machine instruction

BUFFER – 15bytes

Commands(CBUS) || Data (DBUS)

*The bit* (care poate fi doar 0 sau 1) is the basic and the smallest unit of representing the information.

BYTE is the smallest ADDRESSABLE unit of information. Unitate adresabila sau accesibila.

Computer/processor on N bits

1. Software perspective – the size of the memory word = the size of the (majority of) the registers (in our case = 32 bits)
2. Engineering perspective – the size of the communication buses (channels) ABUS, CBUS, DBUS

# 07.10.2021

Data types: int (32 bits), float, char(8 bits), string,...

CPU: 0, 1

Data types: Byte: data represent on 8 bits

Word: - || - - || - 16 bits

Doubleword: - || - - || - 32 bits

Constants:

* Numbers: base 2 ex. 1001b, base 16 ex. 0ABCh, base 10: 123, -2[d] – optional
* Characters ex. ‘a’, ‘b’, ‘c’, ‘+’, ….
* Strings ex. „abc”

Variables:

* User defined variables syntax: [name] type value

Ex. a db 1 (byte) c dd 4 (double word)

b dw -2 (word)

unsinged n bits values: [0, 2n-1]

signed n bits values: [-2n-1, 2n-1 – 1]

* Predefined variables

MOV assignment instruction

Syntax: MOV dest, source

Effect: dest <- src

Ex. MOV ax,2

Every of the 8 general registers which you can use however you want has a special meaning in a special context.

AX – is usually used as one of the explicit operands.

BX – Its name comes from *base register*

Every name that you write in a prog. language it’s a title or a name/denomination of a memory itself. Every name that you use is in fact an adress used for acces that memory cell.

\*= the DEFERENCING OPERATOR in C

A[7] = \*(A + 7) in C

A – the name of an array in C is its starting adress

The Base – THE STARTING of a memory area.

DX – Data register

MULTIPLY op1(M positions) \* op2(N positions) -> M + N

B\*B 🡪 W(ord); W\*W 🡪 DW(doubleword); DW\*DW – QW (pe 64 de biti)

In 16 bits prog, the usage of 32 bits values are allowed in a limited way (result of a multpl. or expressing the dividend of a division). The same situation is in 32 bits prog. where quadword QW are allowed only in a limited way.

The RUN-TIME mechanism of any program in Computer Science FOLLOWS ALWAYS the LIFO( last-in-first-out) order of activating and running the involved programming units (subroutines = functions and procedures).

EBP and ESP will point for you as programming to the base and respectively to the top of the current executive stack frame, which is the active part of code that is running at that moment.

You don’t have acces over the full stack! Only the operating system does.

A user defined type in C is defined by TYPEDEF (which is incorrect because typedef is defining only the structure)

C, Java, VB, Pascal, Fortran – were IMPERATIVE languages, because they rely as a central element on the instructions (commands)

DATA TYPE = structure + associated OPERATIONS

(essential in this definitionis ASSOCIATED – we did not have until OOP AN ENCAPSULATION mechanism)

OOP = DATA ORIENTED PROGRAMMING (everything is built having as the central figure the notion of DATA)

For the microprocessor the notion of data type has a very primitive meaning limited only to the size of the representation of that element.

On 32 bits these can be – byte, word, doubleword and it can be also a qword (these are the ASSEMBLY language DATA TYPES). You can define variables/operands in the RAM memory by using DATA DEFINITION DIRECTIVES – db, dw, dd, dq .

RAM (Random Acces Memory) – WHO is random?

* The acces time at any given location from the RAM is THE SAME independently of the position (randomely far from the beginning of the memory...)
* In contrast from ROM (read only memories) a RAM supports/allows any number of R/W and in any ORDER (Randomely... reads and writes in a randomely order ... The order in which R/W appear is RANDOM)

# 14.10.2021

* EFLAGS – register to 32 bits
* There are 9 flags
* A flag is a indicator represented on 1 bit
* A config of the flags register shows a synthetic overview of the execution of the each instruction

CF – Carry flag DF – Direction flag

SF – Sign flag OF – Overflow flag

ZF – Zero flag

CF – is the transport flag. It will be set to 1 if in the last performed operations there was a transport digit outside the representation domain of the obtained result and set to 0 otherwise

PF – (parity flag) are val 1 daca rezultatul ultimei op executate are un numar par de cifre de 1 si val 0 daca nu.

ZF – are val 1 daca rezultatul ultimei op executate va fi 0 si valoarea 0 daca nu

SF - are val 1 daca rezultatul ultimei op executate este un numar fix negativ si valoarea 0 daca nu

TF ( „Flagu Capcana” – Vancea 2021) – is a debugging flag. If it’s set to 1, then

DF – for operating string instructions. If set to 0, then string parsing will be perfomed in an ascending order (from the beginning to the end) and in descending order if set to 1.

OF (OnlyFans, glm) – flags the signed overflow. If the resut of the last performed op. didn’t fit the reserved space(admissible representation interval), then the OF will be set to 1 or 0 otherwise.

Both OF and CF are reffering to addition and subtraction.

CF signals overflow for the case of UNSIGNED interpretation. OF flags the sign overflow.

Overflow – main topic for the next time

# 21.10.2021 - Overflow

An overflow it’s a mathematical condition wich expresses the fact that the result of the LPO didn’t fit the reserved space for it OOOOR

147 + 179 = 326 but the carry flag is set to 1 expressing the fact that the result can not fit a byte.

-109 = 147’s complement -109 +

-77 = 179’s compl - 77

-------

* 186 !!

326 and -186 are the correct results in base 10 for the coresponding interpretations

BUT, in ASSEMBLY language we have ADD b + b 🡪 b.

CF and OF express overflow situations ONLY for ADDITION and SUBTRACTION. If set to 1 both of them are expressing INCORRECT MATHEMATICAL OPERATIONS (That’s why in fact they are setted to 1 !)

What about multiplications and divisions?

Op 1 – m positions, Op 2 – n positions, than the result op1\*op2 will be represented on m+n positions. Fortunately, the assembly language IS ALWAYS providing multiplications with enough space for dza rizalt.

Multpl in assembly lang. WILL NEVER issue an REAL overflow.

Anyway, CF and OF will have a role in case of multpl.

B\*b = word; b\*b = b(ex. 2\*3 = 6) 🡪 OF = CF = 0 (no ”overflow”)

b\*b = w(ex. 255 \* 3 =...) 🡪 OF = CF = 1(”overflow”)

In the case of the DIVISION, WE DO have an overflow and IT IS THE WORST CASE OF ALL!!!!

w/b = byte ; 1002/3 = 334, buut... 334 is a byte? No it isn’t and IT IS FATAL!! 🡪 RUN TIME ERROR (programu crapă). The operating system will stop the running of the program and will issue one of the 3 semantic equivalent messages: ”Divide overflow”, ”Division by zero” or ”Zero Divide”

DZA 2 FLAGS CF and OF are set exactly for letting come up with a correction if you want..

Number / 0 = Zero divide – this is a forbidden operation

Number / 0 in assembly language will issue a DIVIDE OVERFLOW (ZERO DIVISION) because inifinit doesn’t fit a byte, a word or a double word.

If I intend to make the div ; 1002/3 =..., the assembly language will issue a DIVIDE OVERFLOW because 334 doesn’t fit a byte. But from above, the proccesor already considered that 3 messages equivalent...so, it still will issue the same things... Why? Because from the microprocessor point of view it’s the same shit.

From a tehnical point of view an INT 0 will be issued in ALL situations of this type!

When I make an addition or a subtraction in b2 in assembly language what happends is in fact that we have 2 different simultaneous operations in b10.

1 addition in b2 = 2 diff additions in base 10!!

1 subtraction in b2 = 2 diff subtractions in base 10!!

That is why CF and OF are NECESSARY to be TOGETHER present and associated with this operations.

Add and sub in b2 works exactly in the same way independently of the interpretation in b10. I mean that they function IDENTICALLY for the BOTH intepretations simultaneously.

What values will be associated to the CF and OF in case of division?

If we have div overfl and program crashes NOBODY CARES about the values of the CF and OF and they can’t be checked.

In case of division, the CF and OF are undefined.

--------

WHY do I NEED IMUL and IDIV???

BECAUSE in contrast from addition and subtraction which functions IDENTICALLY for the BOTH intepretations simultaneously, multpl and div function differently need to be told before the operation how it should be the interpretation of the operands

# 21.10.2021 – Flags

We have 2 flags categories:

1. Reporting the status of the last perfomed op (having a so called previous effect) : CF, PF, AF, ZF, SF
2. Flags to be set by the programmer having a future effect on instructions that follows. CF, TF, IF, DF

How? By using SPECIAL instructions – 7 instructions

Now it should be clear that the flags are present at the level of mP for

offering the programmer the possibility of verifying some conditions. In that aspect, conditionals jump instructions were introduced in the assembly lang. for the programmers to be able to express the situations to be verified at the level of the program.

The instructions that are meant to set the flags values DO NOT have explicit operands because each of them is working on a separate and distinct flag.

STC – CF = 1

CLC – CF = 0 ; 3 instr for CF

CMC – complements the value of CF (if CF == 1: CF = 1; if CF == 0: CF = 1)

CLD – sets DF = 0

STD – sets DF = 1 ; 2 instr for DF

CLI – sets IF = 0

STI – sets IF = 1 ; 2 instr for IF – they can be used only on 16bits programming; on 32 bits, the OS restricts

Adress registers and adress computation

*Adress of a memory location –* nr. of consecutive bytes from the beginning of the RAM memory and the beginning of that memory location.

An uninrerrupted sequence of memory locations, used for similar purposes during a program execution, represents *a segment*. So, a segment represents a logical section of a program’s memory, featured by its *basic adress* (beginning), by its *limit (size)* and by its type. Both basic adress and segment’s size have 32 bits value representations.

Limit (size) – in loc de limit mai bine DIMENSION

An adress specification is the only form that a programmer can use in his programs for expressing adresses.

a7a6a5a4a3a2a1a0 := b7b6b5b4b3b2b1b0 + o7o6o5o4o3o2o1o0

The x86 arhitecture allows 4 types of segments:

1. *Code segment*, which contains instructions
2. *Data segment*, containing data which instructions work on
3. *Stack segment*
4. *Extra segment;* (supplementary data segm)

The only necessary segments for a program to be runned are the code segm and stack segm.

For us, as programmers the only mandatory segm necessary to be defined is the code segment.

# 04.11.2021 – 2 AM in the morning formula

A segment: *physical segment* – a block of memory of discrete size.

*Logical segment* – a variable-sized block of memory, occupied by a program’s code or data.

We will call offset the address of a location relative to the beginning of a segment, or, in other words, the number of bytes between the beginning and the destination(idk)

Handling the segment part is the job of the OS. Handling the offset is the job of the programmer.

We can have in our programs multi-modules.

In 16 bits programming, the segment reg. CS, DS, SS and ES contain the starting addresses of the curently active segments. On 32 bits prog, these segment registers are containing the segment selectors of the curent active segments.

Register EIP contains the offset of the current instruction inside the current code segment, the register being managed exclusively by

The main task of the assembler is to generate the coresp btyes.

At any given moment ONLY ONE segment of every type may be ACTIVE.

An offset is always relative to a certain segment.(you can not work independent offsets).

A NEAR address represents an offset. Is always inside of one of the 4 active segments

A FAR address can be specified in 3 ways:

- s3s2s1s0 *: offset specification* where s3s2s1s0

- segment register: offset specif, where segment registers are CS, DS, SS,ES, FS, GS

- FAR [variable], where variable is of type QWORD and contains 6 bytes representing the FAR address (pointer variable)

Usually, we will NOT use FAR addresses specifically

OFFSET = [base] + [index \* scale] + [constant]

2 AM in the morning formula (address computing formula)

# 11.11.2021 –

Pointer arithmetic represents the set of arithmetic operations allowed to be perfomed with pointers. This meaning using arithmetic expresions which have addresses as operands.

Operation with pointers that makes sense and are practical (THERE ARE ONLY 3, tine minte pana la PENSION):

* Adding a constant value to a pointer (A[7] = \*(A+7) ) – useful for going into memory forth and back relative to a starting address ( = A POINTER)
* Subtracting ..... A[-4] ( = A POINTER)
* Subtracting 2 pointers !! (SCALAR VALUE) this can be very useful to det the length of a memory area

The name of variable is associated in assembly language with its offset

relative to the segment in which its definition appears.

Assembly languages in C are value-oriented languages meaning that everything is finally or in the end reduced to a numeric value, This is a low-level feature. The value associated to a variable it’s not its content but it is its address.

The offsets of the variables defined in a program are always constant values determinable at assembly time (compiling time).

In a high level programming lang, the programmer can acces the memory only by using the names of variables defined and declared in the program. In contrast, in assembly language, the memory is accesed by ussing the offset computation formula (formula de la 2 noaptea) OR by using pointer arithmetic (which is also true in C – using pnt. arth.)

How do I make the difference betw the address of a variable and its content?

var – invoked like that its an adress(offset) ; [var] – is its content

[ ] = the deferencing operator !!! (like \*p in C)

Assignment: i := i + 1

LHS = is always and address (left-value / left-hand-side)

RHS = is a content (is part of an expression)

Symbol := expression

Adress\_Expression := Value\_Expression

The result of an address computation doesn’t have an associated data type. Because of that, in the case of our movs instructions (pe care nu le ai aici) the destination operand is the one that decides the data type of transfer and the transfer will be made accordingly to the little endian representation.

There can be performed operations only with operands that can be read as constants at assembly time. The single execption for this rule is the offset specification formula.

# 16.11.2021 – Assembly Language Basics

*Machine Language* of a Computing System (CS) – the set of machine instructions to which the processor directly reacts. These are represented as bit strings with predefined semantics.

*Assembly Language* – a prog lang in which the basic instrunctions set corresponds with the machine operations and which data structures are the machine primary structures. This is a symbolic language. Symbols – Mnemonics + labels.

There is no only one location counter for an entire program, but instead every separate memory segment does have its own location counter.

Location counter – an integer number managed by the assemble for every separate memory segment.

The general form of a source line format applies also to data segment syntax, not only to code segment syntax.

[label] + [mnemonic] + [operands] + [comments]

Code labels are the target of jumps. They can also appear in data segment. We can define a code label in a data segment and we can perform in the segment. We can have a jump in a code segment to have as destination in a data segment. This is a feature allowed only in NASM. TASM and MASM does not allow such things.

Lea eax, [v] is similar in some way with mov eax, v

The offset of a define variable is always a constant determined at assembly time.

Instructions operands may be spec in 3 diff ways, called specification modes.

The 3 operand types are: immediate operands, register operands and memory operands. Their values are computed at assembly time for the immediate operands and for the direct addressed operands(the offset part only!), at loading time for memory operands in direct addressing mode (as a complete FAR address – segment address is determinable here so the whole FAR address is known now! ) – this step involves a so called ADDRESS RELOCATION (adjusting an address by fixing its segment part), and at run time for the registers operands and for indirectly accesed memory operands.

The offsets of data labels and code labels are values computable at assembly time and they remain constant during the whole program’s run time.

There are 2 types of memory operands: direct addressing operands and indirect add op.

The value of the location counter is an address.

The offset of a direct addressing operand is computed at the assembly time. The address of every relative to the executable program’s structure (establishing the segments to which the computed offsets are relative to) is computed at linking time. The actual physical address is computed at loading time.

As a pointer type we can have offsets only on 16, 32 and 64 bits. We can not have offsets on 8 bits, Attempting to do that it will issue an syntax error.

January 5th

A conversion is a prog lang mechanism that allows you to acces data under a different form that the initial one. Destructive conversions can not be made with loss of information.