CURS 12

Hello good evening how are you today tired yeah ok the last week so uhh let’s say that you must still uhhh … 2 hours ma mai suportati vreo 2 ore si scapati de mine hahahahahahahah [evil disgusted laugh]

..internal format for an instruction yeah and …

What the assembler generates for an instruction with some prefixes

* Which instruction prefixes are possible. For any instruction we have 0 up to 4 prefixes. Even 3 of them are reasonable to think that they appear simultaneously, maybe even 4.
* For segment override and instruction prefix their values are determined by what we write explicitely
  + Ex: string manipulation instruction for instruction prefix:
    - REP, REPE
      * If you put these prefixes in an instruction (like REP MOVSB) in the internal format of the instruction F3h will be generated as a code. This is the code of the instruction prefix
    - REPNE
      * If you put these prefixes in an instruction (like REPNE MOVSB) in the internal format of the instruction F2h will be generated as a code. This is the code of the instruction prefix
  + Segment Override
    - If you want to override an implicit rule with a certain register
    - If you write mov ax, [ebx], you would have an implicit rule that states that the segment address would be represented by ds. If I don’t want ds to be put in front of ebx, we can specify in an explicit way another register (segment override) (mov ax, [CS:ebx]. Every … has a diff code (ex: CS has 2Fh)
    - XLAT instruction
      * If you specify just xlat, the starting address would be ds:ebx. If we want to change that you could write ES xlat. The translation table will not start at ds:ebx , but at ES:ebx. For this instruction you will obtain 26h
* The last 2 prefixes (address-size prefix and operant-size prefix) can not be put explicitely by the coder
  + Example:
    - Operand size prefixA
      * Bits 32
      * If you write bits 16, you can force the code to be generated and analyzed under 16 bits
      * Cbw – instruction that has no explicit operands. If you analyze what it generates, you will see that this instruction has only 1 byte: 98h. What you will see in column 2 will. CBW returns 16 bits addresses, so while The implicit size should be 32, the assembler will signal you that you don’t have a 32 bits address returned, so you will also have a 66h
      * Recapitulare
        + 66h – operand size prefix
        + 67h – address size prefix
        + 98h – address cbw
      * CWD – the corresponding code is 99h, but you will also have returned 66h
      * CWDE – We won’t see an operand size prefix, only the 98h (it has the same code as cbw, but since it doesn’t have 66 in front since it returns a 32 bits value it won’t generate the same thing)
      * Push ax (is not in 32 bits, so it will generate 66 50 h, 50 belongs to ax)
      * Mov ax, a (in ax will be transferred the offset of a which will be truncated to 16 bits 66 B8 0010)
      * Push eax (it will generate only one byte with 50h)
    - Adress size prefix – 67 h
      * Bits 32
      * Mov eax, [bx] (it is correct because on 16 bits we have a different offset specification formula: [BX/BP] + [SI/DI] + [const] (base = bx/bp, index = si/di). The segment register will be DS implicitely, but this is not an address on 32 bits. What will be generated here will be 67:8B07)
      * Bits 16
      * Mov bx, [eax] (it is ok from the point of view from the formula, but what will be generated will be 67:8B18)
      * When you put bits 16 it’s not like it won’t understand 32 bits, but it will be implied that all the code is generated on 16 bits
      * Push dword [ebx] (a 67 will be generated along with FF33. Not only do we use a 32 bits address, but we also use push dword, so the operand is a 32 bits, on 16 bits, so what will be generated will be 66 67 FF33)

TECHNIQUES AND TOOLS

* A call phase has an associated code called generated automatically by the compiler. So does an entry phase have an associated entry code. Only the subject of theory can include multi module programming (it can come in practical only if it’s not asm + asm)
* We discussed about these phases because as far as we stay inside the same language the compiler will do it for us. When we combine 2 languages, one language has to include the other one. Try to think that cdecl and stdcall conventions put the parameters in reverse order (c rule), but if we combine them with another programming language in …
* Example of multimodule programming asm + c (the first example from the last meeting in which we had one module written in c (afisare.c) and another module written in assembly). We have a function which won’t be used in c, but in assembly (afisare). The only thing that the c module start is calling the start from assembly (call asmstart) that asm start procedure will also call afisare. Who is generating the code for asm start? The c compiler, which is generating it automatically regardless of the fact that this is asm. So, where the hell is the entry code in asm start? Because we are in assembler, the assembler does not automatically generated the call code for a c function, we as programmers must do it explicitely. What does add esp, 8 represent? It’s the responsibility of the caller to free the stack as part of the cdecl convention, a convention applied to the “afisare” function. We discussed the call code for asm start and afisare, so where is the entry code of afisare? It is generated automatically by the c compiler, just like the exit code. SO WHERE IS THE ENTRY CODE IN ASM START?? There is no entry code because we don’t need it. It is to the latitude of the programmer if he wants to do it, it is not mandatory. You can avoid these steps, you can pass them. What about the exit code? Ret. Because it’s cdecl function, it’s only ret. If it’s stdcall function, it’s the responsibility of the callee to free the stack, therefore ret 4
* All these examples are in order to complete the following table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Caller | Callee | (function call)  Call code | { …  Entry code | … }  Exit code |
| C | C | C Compiler | C Compiler | C Compiler |
| C | Asm | C Compiler | ASM Programmer | ASM Programmer |
| asm | C | ASM Programmer | C Compiler | C Compiler |
| asm | asm | Call instruction (saving the returning address) | NOTHING MANDATORY | Ret (grabs the returning address and jumps to that address) |

Only because we have those 3 things do we have to learn programming in c according to asm.3

Conversions classification in asm

* Usually in computer science, conversion means a technique that allows you to access some data under another form than that of the initial definition. (change of interpretation)
* Even in high level programming languages you have implicit conversions (float -> integer; ex; e = a + b + c , where e - float and a, b, c – integer)e = a + b + c – implicit conversion
* 3 criteria of conversions in asm
  + Destructive / nondestructive
    - Modifying its’ size by enlarging it
    - Destructive: cbw, cwd, cwde etc
    - Non-destructive: type operators
  + Signed / Unsigned
    - Signed: cbw, cwd, cwde (destructive instructions that take into account the signed value, therefore SIGNED conversions)
    - Unsigned: movzx, mov ah, 0, mov dx, 0
  + By enlargement (all the destructive ones) / by narrowing
    - There are conversions by narrowing, but they are temporary and non-destructive. You can have a sequence of dwords and apply the byte of it (TEMPORARY CONVERSION)\\

EXAMPLE OF SUBJECT AT EXAM

* From the mechanism (I will not tell you instruction, directives, or operators) that you know in asm language, PLEeeeeASE tell me which are the mechanism that use signed conversions and unsigned conversions
* Care sunt instructiunile limbajului de asamblare care tin cont de semn ?
* The following sequence is given.
  + V dw 23456
  + …
  + Add ebx, v
  + Sub ebx, 6
  + Mov eax, ebx
* Write one single instruction to have the same effect on eax register
  + Lea eax, [ebx + v – 6]
  + We speculate the power of the offset specification formula which allows the addition of one base register with the addition with the index register adding also a direct address variable and using some constants (puteam pune si 4) to do all of them in a single instruction
* The following sequence is given
  + V dw 23456
  + …
  + Add ebx, v
  + Sub ebx, 6
  + Mov eax, [ebx]
  + ANSWER: mov eax, [ebx + v – 6]
* The following sequence is given
  + V dw 23456
  + …
  + Add ebx, [v]
  + Sub ebx, 6
  + Mov eax, ebx
  + ANSWER: we cannot do that because we can no longer speculate the offset specification formula. YOU CANNOT KNOW the contents of the address of v. Because you only use contents of variables
* The following sequence is given
  + Xor edx, edx
  + Mov dl, 0fh
  + Complete this instruction sequence with ONE INSTRUCTION OR MORE TO OBTAIN the multiplication with 4 of the value represented in edx:eax
  + ANSWER:
    - Cu semn sau fara?
    - TRE SA ITI DAI SEAMA
    - Avand in vedere ca pune 0 prin xor edx, edx e evident daca e cu semn sau nu (nu este)
    - We are talking about a quadword, so we are not sure that the output result is a quadword. We can in this case, because this puts 0 in all edx
    - We want to “shl” all this structure, basically “shl edx:eax, 2” we cannot write sth like that
    - The challenge is that the first 2 bits from eax must be the last 2 bits from edx
    - We can use an intermediate element for this, the carry frlag
  + Shl eax, 1
  + Rcl edx, 1
    - These 2 instructions obtained the multiplication with 2
    - By writing these 2 instructions again, we obtain the answer
  + Shl eax, 1
  + Rcl edx, 1
  + ‘nu e asa de greu’ – elev foarte naiv din spatele meu

ABOUT THE EXAM

SUBJECT 1 – THEORY

* POATE SA FIE ORICE DIN CE AM DISCUTAT SLIDE-URI

SUBJECT 4 – PROBLEM

* CE AM FACUT LA SEMINAR (nu am facut)(same)

SUBJECT 2, 3 – COMBINATION BETWEEN PRACTICE AND THEORY

* SUBJECT 3 COULD/PROBABLY WILL BE a data segment and generate it in little Indian form. Please provide the memory layout (it might be best to generate for every line what it does. If some lines are wrong, explain why and ignore them further in generating the data segment)
* SUBJECT 3 COULD/PRBABLY WILL BE THE MEMORY LAYOUT (shorter) and have us generate the code (slabe sanse ca lui vancea ii e sila ca noua)
* SUBJECT 3 COULD/PROBABLY WILL BE SEQUENCES OF CODE, LIKE WHAT WE DID ABOVE. If you find an error, specify where are the errors and move on
* TARGETS: LITTLE ENDIAN REPRESENTATION, SIGNED VS UNSIGNED, OVERFLOW