Volatile vs. non-volatile resources

- **volatile** resources are represented by those registers that **the calling convention is defining them as belonging to the called subroutine**, thus, the caller being responsible **as part of the call code** to save their values (if the called subroutine is using them) and after that, at the end of the call to restore the initial (old) values. So: who is saving the volatile resources ? **The caller** (as part of the call code) . Who is restoring in the end those values ? Also **the caller** <u>but NOT as part of a certain call/entry or exit code</u>. Just restore them after the call in the regular code as a mandatory responsibility.

- **non-volatile** resources are any memory addresses or registers which do not belong explicitly to the called subroutine, but if this one needs to modify those resources, it is necessary that the called subroutine to save them at the entry as part of the entry code and restore them back at exit, as part of the exit code. So: who is saving the non-volatile resources ? **The callee** (apelatul = the **called** subroutine, as part of the entry code) . Who is restoring in the end these values ? Also **the callee** (as part of the exit code).

Call code, entry code, exit code
- what they represent
- what they are necessary
- steps

definition
↳ what's given
↳ implication
↳ conclusion

* One of the theory subject for written exam

Call code (THE CALLER):

a). Saving the volatile resources (EAX, ECX, EDX, EFLAGS)
b). Passing parameters
c). Saving the returning address and performing the call

Entry code (THE CALLEE – called subroutine):

a). Building the new stackframe PUSH EBP, MOV EBP, ESP > for exam, theory

b). Allocating space for local variables SUB ESP, nr_bytes
c). Saving non-volatile resources exposed to be modified

08' exit code

* explain the inter-relation between call code – entry code – exit code

Exit code (THE CALLEE):

* present steps for start – end (do smt → unroll it)

* gotta convince him that I understood

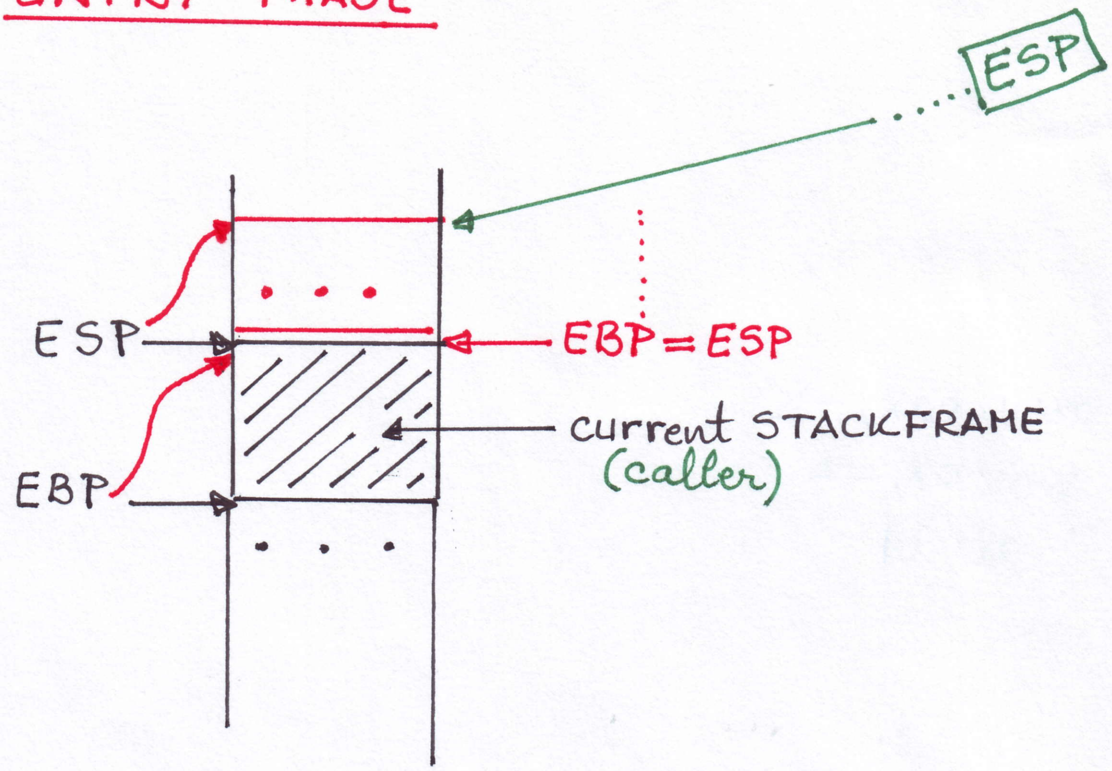
a). Restoring non-volatile resources
b). Freeing the space allocated for local variables [ADD ESP, nr_bytes_locals] – mentioned here just as a reverse for the above b) from the entry code, but not really necessary because deallocating the stackframe (mov esp, ebp) includes this action anyway from a practically point of view.

c). Deallocating the stackframe MOV ESP, EBP (if we know exactly the size of the stackframe , ADD ESP, sizeof(stackframe) solves similarly…)
and restoring the base of the caller stackframe (old EBP) POP EBP
(a, b c – the reverse of the entry code)

d). Returning from the subroutine (RET) and deallocating passed parameters (if we have a STDCALL function) - (reverse of b + c from the call code)

It is still to be done the reverse of a) from call code. It is the task of the CALLER to do it together with a possible parameters take out from the stack (if it is a CDECL function).

ENTRY PHASE



- involves the creation of a NEW STACKFRAME for the CALLED subroutine:

```
push EBP ; for restoring the base of the
           CURRENT STACKFRAME when returning

MOV EBP,ESP ; This is the BIRTH of the
              NEW STACKFRAME (initial sizeof = 0)
```

ESP will start to "grow" by currently performed PUSHES

Call code → generated by caller

a) volatile
b) params
c) call

restoring

Entry code → generated by the callee

a) new stackframe
b) local variables
c) new stackframe

Exit code

* perfect stack order, the exact same!

a) restoring non-volatile
b) free local varia
c) delete stack frame
d) ret + [freeing params]

↳ maybe, if it is a std call since is our respons.

* if it is a cdecl call,

*exam, understand

Caller	Callee	Function / proced. 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	NOTHING MANDATORY	ret [n]

write it manually

* in ASM you don't have the concept of a function, it is just a label

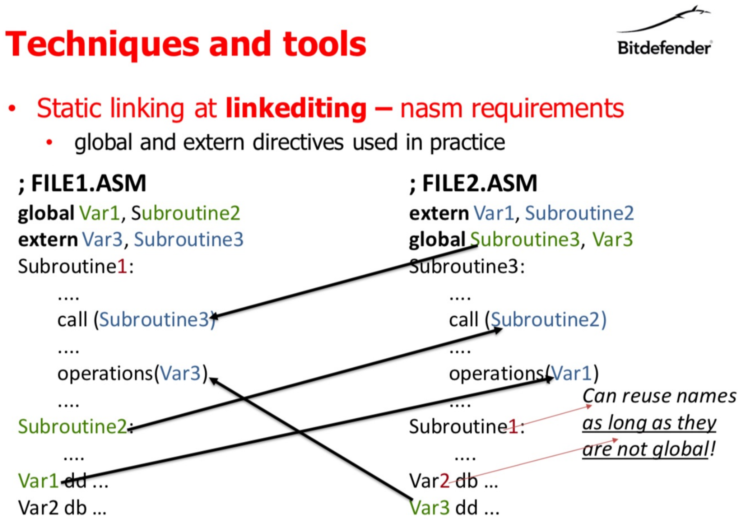
* we don't have volatile resources or params

* explain CDECL, STD CALL

↳ what they mean, how they are used, difference between them (ppt bitdefender)

* 38' example

multi-module program



Techniques and tools

- Static linking at **linkediting –** nasm requirements
  - global and extern directives used in practice

```
; FILE1.ASM                          ; FILE2.ASM
global Var1, Subroutine2             extern Var1, Subroutine2
extern Var3, Subroutine3             global Subroutine3, Var3
Subroutine1:                         Subroutine3:
      ....                                 ....
      call (Subroutine3)                   call (Subroutine2)
      ....                                 ....
      operations(Var3)                     operations(Var1)
      ....                                 ....
Subroutine2:                         Subroutine1:
       ....                                 ....
Var1 dd ...                          Var2 db ...
Var2 db ...                          Var3 dd ...
```

Can reuse names as long as they are not global!

Prefixes

a) Instruction prefixes

REP movsb

b) Segment prefixes

ES xlat

→ explicit prefixes given by the programmer

(segment overwrite prefix changes the default segment of that instruction)

c) Operand - size prefix

d) Address - size prefix

implicit generated by the assembler

* Trebuie stiute pe de rost

F3h - REP, REPE

F2h - REPNE

(REP "sting instr."
F3h instr.-code

2Ah - CS

36 h - SS

3E h - DS

26 h - ES

mov eax, [CS: ebx] → 2E: 8B03

Es lodsb → 26: AC

66 h ⟶ opperand

67 h ⟶ address

```
bits 32
mov eax, [bx]    ; 67:8B07
```

since DS:[BX] is 16 bits addressing

```
bits 16
mov bx, [eax]    ; 67:8B18
```

since DS:[eax] is 32 bits addressing

```
bits 32
cbw       ; 66:98
cwde      ; 98

push ax   ; 66: 50
push eax  ; 50

mov ax, a ; 66 : B8 0010
```

not conformant with 32, since it is not a dw

```
bits 16
cbw    ; 98
cwd    ; 99
cwde   ; 66:98
```

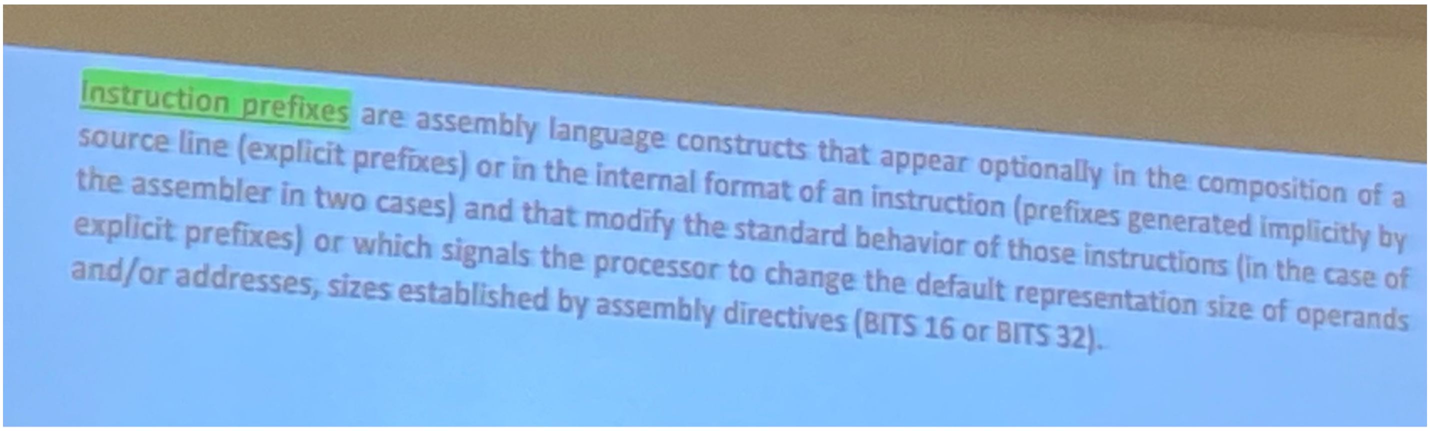
```
bits 16
push dword [ebx] ;  66:67: FF 33

push dword [CS: ebx] ;  2E : 66 : 67 : FF33

rep push dword [cs: ebx] ;
```

* what are prefixes?

language constructs that appear optionally in the composition of a source line that modify the standard behaviour of those instructions



Instruction prefixes are assembly language constructs that appear optionally in the composition of a source line (explicit prefixes) or in the internal format of an instruction (prefixes generated implicitly by the assembler in two cases) and that modify the standard behavior of those instructions (in the case of explicit prefixes) or which signals the processor to change the default representation size of operands and/or addresses, sizes established by assembly directives (BITS 16 or BITS 32).

* possible exam subject

Conversions classification

a) distructive

- * instructions : cbw cwd cwde cdq
- * they overwritie the original register, that's why they are distructive

non-destructive : type operators : byte, word, dword, qword

b) signed : cbw, cwd, cwde, cdq, movsx

unsigned : movzx, mov ah, 0; mov dx, 0;

c) by enlargement → all the destructive ones ! + word, dword, qword

by narrowing → byte, word, dword

* destructive conversions by narrowing are not possible in high lvl prog.

d) implicit vs explicit conversion

EXAM

09'

I structure of microprocessor
registers
segment registers

address computations (8100h sm?) + examples illustrating the theory + explanation
offset specification formula
NEAR and FAR addresses
offset caracteristics
basics of asm:
- instr.
- directives
- label
- location counter

2's complement
representation
why do we need 2's complement
working with negative numbers
overflow (both tehnigal and mathematical wiew)
↳ for +, -, :, * , examples, how the flags work

why do we have imul and idiv but no iadd and isub

xlat / lea

IV conversions
bits inversions
strings
arithmetic
sistem functions <u>only</u> printf & scanf

II/III memory layout
data segment
code segment

5 source code sequences ⇒ explain what they are doing & the effect on registers

```
v   resw   2
---------------
  add  ebx , v
  sub  ebx , 6
  mov  eax , ebx
```

" mov eax, ebx + v - 6 "

```
lea eax, [ebx+v -6]
```

write one instruction that has the same effect on eax as the sequence

```
v   resw   2
---------------
  add  ebx , v
  sub  ebx , 6
  mov  eax , [ebx]
```

```
mov eax, [ebx +v-6]
```

probably memory violation error

```
v   resw   2
---------------
  add  ebx , [v]
  sub  ebx , 6
  mov  eax , ebx
```

* no solution

you cannot put the contents of a memory area into the offset specification formula