Practical Foundations of Debugging

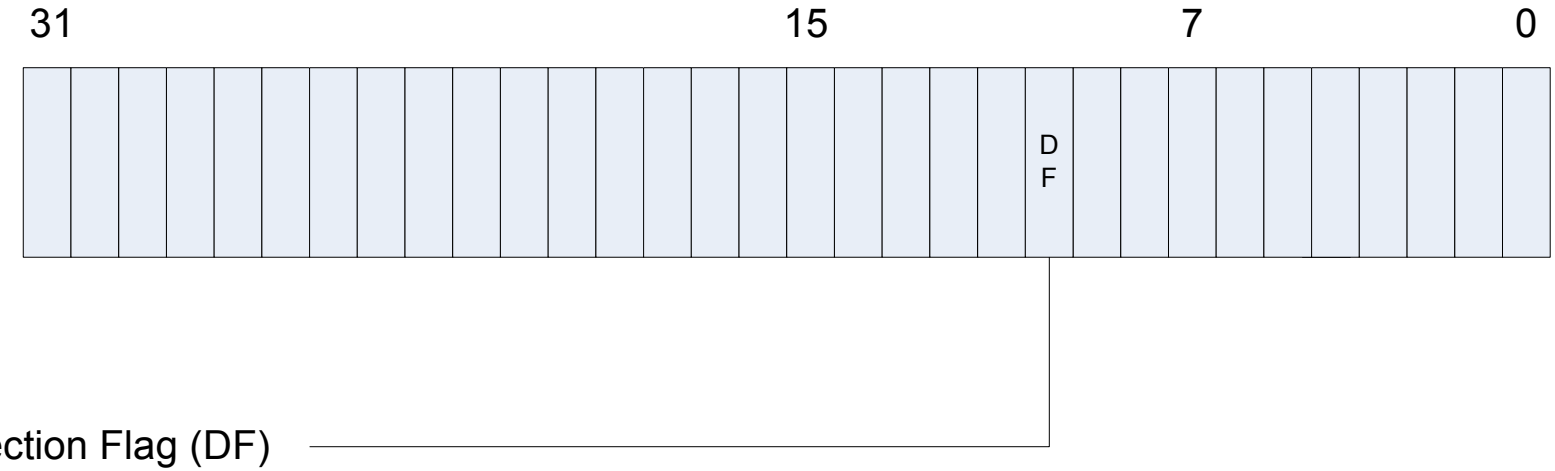
Chapter 8

Function Pointer Parameters (Part 1)

Designed, written and illustrated by Dmitry Vostokov

CPU Flags Register

EFLAGS (32-bit register)



Default value for DF bit is 0

Instructions:

CLD clears DF bit

STD sets DF bit

The fastest way to fill memory

STOSD instruction

1. Stores a dword value from EAX into memory location the address of which is in EDI (“D” - destination).
2. After the value from EAX is transferred it increments EDI by 4 (EDI now points to the next DWORD in memory) if DF flag is 0.
   If DF flag is 1 then EDI value decremented by 4 (EDI now points to the previous DWORD in memory)

REP prefix

Causes the next instruction to be repeated until the count in ECX register is decremented to 0

Example: zeroing “all memory” (will trap because of access violation)

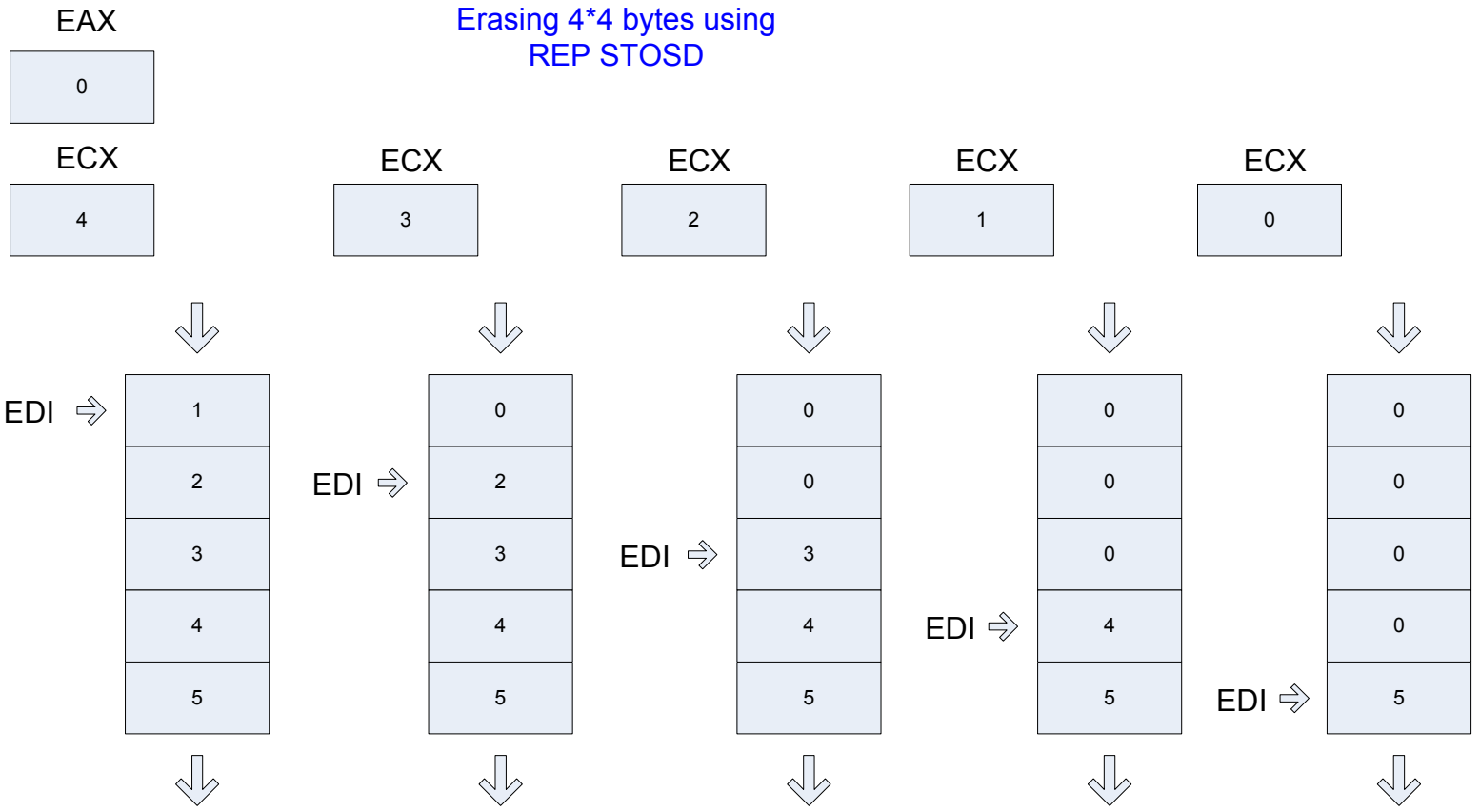
```
XOR EAX,EAX                     ; fill with 0
MOV EDI, 0                      ; starting address, XOR EDI, EDI
MOV ECX, 0xFFFFFFFF / 4         ; 0x3FFFFFFF dwords
REP STOSD
```

REP STOSD in pseudo-code (Picture 1)

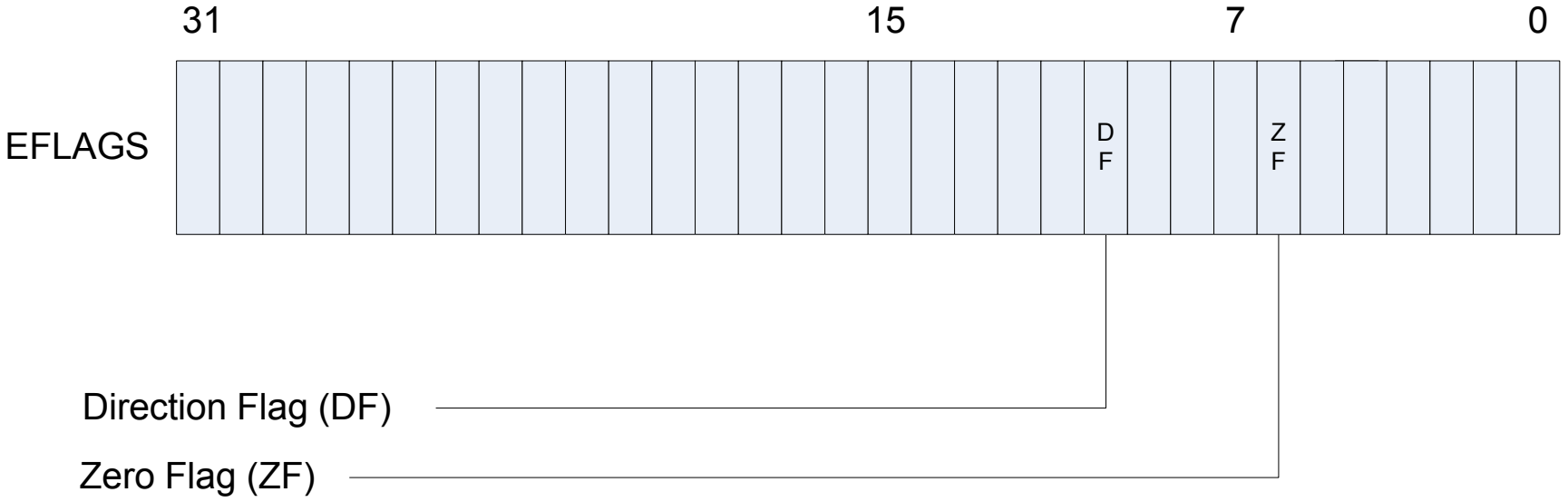
```
WHILE (ECX != 0)
{
   [EDI] := EAX

   IF DF = 0 THEN
      EDI := EDI + 4
   ELSE
      EDI := EDI - 4

   ECX := ECX - 1
}
```

Note: CPU emulation

Picture 1


EAX
0
Erasing 4*4 bytes using
REP STOSD
ECX
4
ECX
3
ECX
2
ECX
1
ECX
0
EDI
1
2
3
4
5
EDI
0
2
3
4
5
EDI
0
0
3
4
5
EDI
0
0
0
4
5
EDI
0
0
0
0
5

Testing for 0



ZF bit is set to 1 if the instruction result is 0 and cleared otherwise.

ZF bit is affected by:
arithmetic instructions (ADD, SUB, MUL, etc.)
logical compare instruction (TEST)
“arithmetical” compare instruction (CMP)

TEST - Logical Compare

- TEST reg/mem, reg/imm

Computes bit-wise logical AND between both operands and sets flags (including ZF) according to the computed result (which is discarded)

TEST EDX, EDX
TEST EDX, 1

Suppose EDX contains 4 (100_{bin})
100_{bin} AND 100_{bin} = 100_{bin} != 0 (ZF is cleared)
Suppose EDX contains 0 (0_{bin})
0_{bin} AND 1_{bin} = 0_{bin} == 0 (ZF is set)

TEST instruction in pseudo-code

Note: details not relevant to ZF are omitted

```
TEMP := OPERAND1 AND OPERAND2
IF TEMP = 0 THEN
  ZF := 1
ELSE
  ZF := 0
```

CMP – Compare Two Operands

- CMP reg/mem, reg/imm
- CMP reg, reg/mem/imm

Compares the first operand with the second and sets flags (including ZF) according to the computed result (which is discarded). Comparison is performed by subtracting the second operand from the first (like SUB instruction, **SUB EAX, 4** for example)

CMP EDI, 0
CMP EAX, 16

Suppose EDI contains 0
0 – 0 == 0 (ZF is set)
Suppose EAX contains 4_{hex}
$4_{hex} – 16_{hex} = FFFFFFEE_{hex}$!= 0 (ZF is cleared)
$4_{dec} – 22_{dec} = -18_{dec}$

CMP instruction in pseudo-code

Note: details not relevant to ZF are omitted

```
TEMP := OPERAND1 - OPERAND2
IF TEMP = 0 THEN
  ZF := 1
ELSE
  ZF := 0
```

Note: equivalent to

```
TEMP := OPERAND1
SUB TEMP, OPERAND2
```

Which comparison instruction to use: TEST or CMP?

- They are equivalent if you want to test for zero, but CMP instruction affects more flags

TEST EAX, EAX

CMP EAX, 0

- CMP instruction is used to compare for inequality

CMP EAX, 0 ; > 0 or < 0 ?

- TEST instruction is used to see if individual bit is set

TEST EAX, 2 ; 2 == 0010_{bin} or in C language: **if (var & 0x2)**

Warning: TEST instruction cannot be used to compare for equality or inequality:

Suppose EAX has 2

TEST EAX, 4 ; 0010_{bin} AND 0100_{bin} = 0000_{bin} (ZF is set)

TEST EAX, 6 ; 0010_{bin} AND 0110_{bin} = 0010_{bin} (ZF is cleared)

Conditional jumps

```
if (a == 0)
{
   ++a;
}
else
{
   --a;
}
```

```
if (a != 0)
{
   ++a;
}
else
{
   --a;
}
```

CPU fetches instructions sequentially, so we must tell CPU that we want to skip some instructions if some condition is (not) met, for example if a != 0

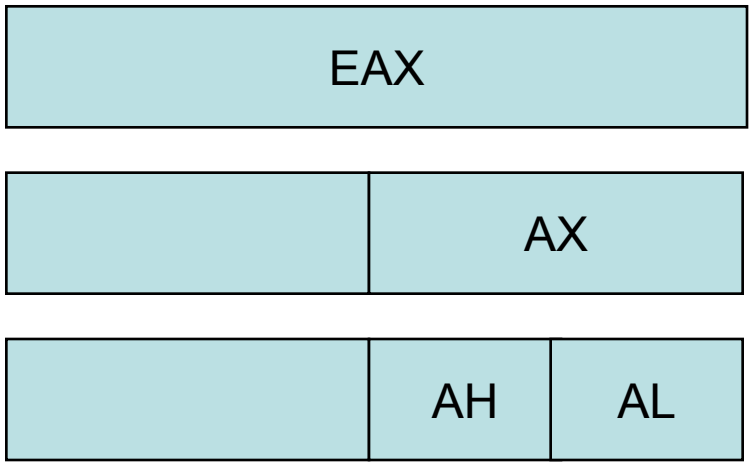
JNZ (jump if not zero) and JZ (jump if zero) test ZF flag and if not set (set) changes EIP

```
        CMP     [A], 0
        JNZ     label1
        INC     [A]
        JMP     label2
label1: DEC     [A]
label2: …
```

```
        MOV  EAX, [A]
        TEST EAX, EAX
        JZ   label1
        INC  EAX
        JMP  label2
label1: DEC  EAX
label2: …
```

The structure of registers (EAX, EBX, ECX and EDX)

- EAX 32-bit register
- AX 16-bit part of EAX (lower half)
- AL 8-bit part of AX (lower half)
- AH 8-bit part of AX (higher half)



Warning!

- Never read a book about Intel assembly language that uses AX, BX, etc. in examples throughout the text (prehisto'ric text).
- Unfortunately almost all Intel assembly language textbooks are prehistoric.
- I remember 5 years ago I met a guy who was interviewed for a job in Intel. They asked him about assembly language. The guy started talking about AX, BX, etc. No wonder he wasn't hired. (Probably similar to being enthusiastic about Java and Solaris during Microsoft interview).

Returning value from functions (Visual C++)

- int func();

return value is in EAX

- bool func();

return value is in AL

bool values occupy one byte in memory

Using byte registers with dword registers

Task:
Suppose we have a value in AL (a byte value).
We don't know what other parts of EAX contain.
We want to add this value to ECX.

Solution:

```
EBX := AL                    or        EAX := AL
ECX := ECX + EBX                       ECX := ECX + EAX
```

We can use:

```
MOV  BL, AL
MOV  byte ptr [b], AL        ; in C language: static bool b = func()
```

We cannot use:

```
MOV EBX, AL                  ; operand size conflict
```

Moving bytes to double words

MOVZX reg, reg/mem – move with zero extend

Replaces the contents of the first operand with the contents of the second filling the rest of bits with zeros.

Solution for the previous task:

```
MOVZX   EBX, AL
ADD     ECX, EBX
```

or if we want to reuse EAX register

```
MOVZX   EAX, AL
ADD     ECX, EAX
```

“Arithmetical” project

```
// FunctionParameters.cpp
#include <stdio.h>
#include "Arithmetic.h"

int main(int argc, char* argv[])
{
    int a, b;

    printf("Enter a and b: ");
    scanf("%d %d", &a, &b);

    if (arithmetic (a, &b))
    {
         printf("Result = %d", b);
    }

    return 0;
}
```

```
// Arithmetic.h
#ifndef __ARITHMETIC_H__
#define __ARITHMETIC_H__

bool arithmetic (int a, int *b);

#endif
```

```
// Arithmetic.cpp
#include "Arithmetic.h"

bool arithmetic (int a, int *b)
{
        if (!b)
        {
                return false;
        }

        *b = *b + a;
        ++a;
        *b = a * *b;

        return true;
}
```

```
FunctionParameters!main:
push    ebp                         ; establishing stack frame
mov     ebp,esp                     ;
sub     esp,0xd8                    ; creating stack frame for locals
push    ebx                         ; saving registers that might be used
push    esi                         ;   outside
push    edi                         ;
lea     edi,[ebp-0xd8]              ; getting lowest address of stack frame
mov     ecx,0x36                    ; filling stack frame with 0xCC
mov     eax,0xcccccccc              ;
rep     stosd                       ;
push    0x427034                    ; address of "Enter a and b: " string
call    FunctionParameters!ILT+1285(_printf) (0041150a)
add     esp,0x4                     ; adjust stack pointer (1 parameter)
lea     eax,[ebp-0x14]              ; address of b
push    eax                         ;
lea     ecx,[ebp-0x8]               ; address of a
push    ecx                         ;
push    0x42702c                    ; address of "%d %d" string
call    FunctionParameters!ILT+990(_scanf) (004113e3)
add     esp,0xc                     ; adjust stack pointer (3 parameters,
…                                   ;   3*4 = 12 bytes, 0xc in hexadecimal)
```

```
lea     eax,[ebp-0x14]                  ; address of b
push    eax                             ;
mov     ecx,[ebp-0x8]                   ; value of a
push    ecx                             ;
call    FunctionParameters!ILT+535(?arithmeticYA_NHPAHZ) (0041121c)
add     esp,0x8                         ; adjust stack pointer (2 parameters)
movzx   edx,al                          ; bool result from arithmetic
test    edx,edx                         ; testing for zero
jz      FunctionParameters!main+0x68 (00411bf8)
mov     eax,[ebp-0x14]                  ; value of b
push    eax                             ;
push    0x42701c                        ; address of "Result = %d" string
call    FunctionParameters!ILT+1285(_printf) (0041150a)
add     esp,0x8                         ; adjust stack pointer (2 variables)
00411bf8:
xor     eax,eax                         ; return result 0
push    edx                             ; saving register ?
mov     ecx,ebp                         ; pasing parameter via ecx
push    eax                             ; saving register ?
lea     edx,[FunctionParameters!main+0x8f (00411c1f)] ; probably address of info about
                                        ;  stack frame
call    FunctionParameters!ILT+455(_RTC_CheckStackVars (004111cc)
pop     eax                             ; restoring register
pop     edx                             ;
pop     edi                             ;
pop     esi                             ;
pop     ebx                             ;
add     esp,0xd8                        ; adjusting stack pointer
cmp     ebp,esp                         ; ESP == EBP ?
call    FunctionParameters!ILT+1035(__RTC_CheckEsp) (00411410)
mov     esp,ebp                         ; restoring previous stack pointer
pop     ebp                             ; restoring previous stack frame
ret                                     ; return
```

```
FunctionParameters!arithmetic:
push    ebp
mov     ebp,esp
sub     esp,0xc0
push    ebx
push    esi
push    edi
lea     edi,[ebp-0xc0]
mov     ecx,0x30
mov     eax,0xcccccccc
rep     stosd
cmp     dword ptr [ebp+0xc],0x0         ; &b == 0 ?
jnz     FunctionParameters!arithmetic+0x28 (00411b48)
xor     al,al                           ; return bool value false (0)
jmp     FunctionParameters!arithmetic+0x4e (00411b6e)
00411b48:
mov     eax,[ebp+0xc]                   ; eax := address of b
mov     ecx,[eax]
add     ecx,[ebp+0x8]                   ; ecx := ecx + [a]  (in C: t = *b + a)
mov     edx,[ebp+0xc]                   ; edx := address of b
mov     [edx],ecx                       ;                   (in C: *b := t)
mov     eax,[ebp+0x8]                   ; eax := [a]        (in C: ++a)
add     eax,0x1
mov     [ebp+0x8],eax                   ; [a] := eax
mov     eax,[ebp+0xc]                   ; eax := address of b
mov     ecx,[ebp+0x8]                   ; ecx := [a]
imul    ecx,[eax]                       ; ecx := ecx * [b]  (in C: t = a * *b)
mov     edx,[ebp+0xc]                   ; edx := address of b
mov     [edx],ecx                       ;                   (in C: *b = t)
mov     al,0x1                          ; return bool value true (0)
00411b6e:
pop     edi
pop     esi
pop     ebx
mov     esp,ebp
pop     ebp
ret
```

With FPO (dynamic addressing of local variables)

```
FunctionParameters!main:
sub     esp,0x8           ; allocating room for local variables
push    0x408110          ; address of "Enter a and b: "
call    FunctionParameters!printf (00401085)
lea     eax,[esp+0x4]     ; address of b   ([ESP + 0 + 4])
push    eax
lea     ecx,[esp+0xc]     ; address of a   ([ESP + 4 + 8])
push    ecx
push    0x408108          ; address of "%d %d"
call    FunctionParameters!scanf (0040106e)
mov     eax,[esp+0x14]    ; value of a     ([ESP + 4 + 10])
lea     edx,[esp+0x10]    ; address of b   ([ESP + 0 + 10])
push    edx
push    eax
call    FunctionParameters!arithmetic (00401000)
add     esp,0x18          ; adjusting stack after all pushes
test    al,al             ; al == 0 ?
jz      FunctionParameters!main+0x48 (00401068)
mov     ecx,[esp]         ; address of b   ([ESP + 0])
push    ecx
push    0x4080fc          ; address of "Result = %d"
call    FunctionParameters!printf (00401085)
add     esp,0x8           ; adjust stack pointer (2 parameters)
00401068:
xor     eax,eax           ; return value 0
add     esp,0x8           ; adjust stack pointer (local variables)
ret
```

With FPO

```
FunctionParameters!arithmetic:
mov     eax,[esp+0x8]       ; address of b
test    eax,eax             ; &b == 0 ?
jnz     FunctionParameters!arithmetic+0xb (0040100b)
xor     al,al               ; return value false (0)
ret
0040100b:
mov     edx,[eax]           ; edx := [b]   (in C: *b)
mov     ecx,[esp+0x4]       ; ecx := [a]
add     edx,ecx
inc     ecx
imul    edx,ecx
mov     [eax],edx           ; [b] := edx
mov     al,0x1              ; return value true (1)
ret
```

What's next?

- What to do if you can't find exact symbol files (discovering function boundaries)
- Real-life stack examples
- Dumps with stack overflow