Practical Foundations of Debugging

Chapter 2

Number representations and Pointers

Designed, written and illustrated by Dmitry Vostokov

Review – disassembling C program

```
int a, b;

int _tmain(int argc, _TCHAR* argv[])
{
   a = 1;                        // mov [a], 1
   b = 1;                        // mov [b], 1

   b = b + a;                    // mov eax, [a]
                                 // add [b], eax
   ++a;                          // inc eax
                                 // mov [a], eax
   b = a * b;                    // imul [b]
                                 // mov [b], eax

   // results: [a] = 2 and [b] = 4

   return 0;
}
```

WinDbg disassembly output – Debug executable

```
00411a1e c705e484420001000000 mov dword ptr [ArithmeticProjectC!a (004284e4)],0x1
00411a28 c705e084420001000000 mov dword ptr [ArithmeticProjectC!b (004284e0)],0x1
00411a32 a1e0844200           mov     eax,[ArithmeticProjectC!b (004284e0)]
00411a37 0305e4844200         add     eax,[ArithmeticProjectC!a (004284e4)]
00411a3d a3e0844200           mov     [ArithmeticProjectC!b (004284e0)],eax
00411a42 a1e4844200           mov     eax,[ArithmeticProjectC!a (004284e4)]
00411a47 83c001               add     eax,0x1
00411a4a a3e4844200           mov     [ArithmeticProjectC!a (004284e4)],eax
00411a4f a1e4844200           mov     eax,[ArithmeticProjectC!a (004284e4)]
00411a54 0faf05e0844200       imul    eax,[ArithmeticProjectC!b (004284e0)]
00411a5b a3e0844200           mov     [ArithmeticProjectC!b (004284e0)],eax
```

```
mov [a], 1     ; [a] := 1
mov [b], 1     ; [b] := 1
mov eax, [b]   ; [b] := [b] + [a]
add eax, [a]   ;
mov [b], eax   ;
add eax, 1     ; [a] := [a] + 1
mov [a], eax   ;
mov eax, [a]   ; [b] := [b] * [a]
imul eax, [b]  ;
mov [b], eax   ;
```

WinDbg disassembly output – Release executable

```
ArithmeticProjectC!main:
00401000 c705c472400002000000 mov dword ptr [ArithmeticProjectC!a (004072c4)],0x2
0040100a c705c072400004000000 mov dword ptr [ArithmeticProjectC!b (004072c0)],0x4
```

```
mov [a], 2    ; [a] := 2
mov [b], 4    ; [b] := 4
```

Numbers and their representations

- number of stones 
- a guy can only count up to 3 

The last picture is representation (notation) of the number of stones

Decimal representation (base 10)

- 12 stones

$12_{dec} = 1 * 10 + 2$ or $1*10^1 + 2*10^0$

- 123 stones

$123_{dec} = 1 * 100 + 2 * 10 + 3$ or $1 * 10^2 + 2 * 10^1 + 3*10^0$

$N_{dec} = a_n*10^n + a_{n-1}*10^{n-2} + \ldots + a_2*10^2 + a_1*10^1 + a_0*10^0$ $0 <= a_i <= 9$

$$N_{dec} = \sum_{i=0}^{n} a_i*10^i$$

Ternary representation (base 3)

- 12 stones

110 in ternary representation (notation)

$12_{dec} = 1*3^2 + 1*3^1 + 0*3^0$

$N_{dec} = a_n*3^n + a_{n-1}*3^{n-2} + \ldots + a_2*3^2 + a_1*3^1 + a_0*3^0$ $\quad a_i = 0$ or 1 or 2

$$N_{dec} = \sum_{i=0}^{n} a_i*3^i$$

Binary representation (base 2)

- 12 stones

1100 in binary representation (notation)

$12_{dec} = 1*2^3 + 1*2^2 + 0*2^1 + 0*2^0$

$N_{dec} = a_n*2^n + a_{n-1}*2^{n-2} + \ldots + a_2*2^2 + a_1*2^1 + a_0*2^0$ $\quad a_i = 0$ or 1

$$N_{dec} = \sum_{i=0}^{n} a_i*2^i$$

Hexadecimal representation (base 16)

- 12 stones

12_{dec} = C in hexadecimal representation (notation)

- 123 stones

123_{dec} = 7B in hexadecimal representation (notation)

$123_{dec} = 7*16^1 + 11*16^0$ …8, 9, A, B, C, D, E, F

$$N_{dec} = \sum_{i=0}^{n} a_i * 16^i$$

Why hexadecimal is used?

- 110001010011 (binary notation)

$3155_{dec} = 1*2^{11} + 1*2^{10} + 0*2^9 + 0*2^8 + 0*2^7 + 1*2^6 + 0*2^5 + 1*2^4 + 0*2^3 + 0*2^2 + 1*2^1 + 1*2^0$

1100*0101*0011 is **C53** in hexadecimal

$\mathbf{12}_{dec}$ $\underline{5}_{dec}$ $\mathbf{3}_{dec}$

$\mathbf{C}_{hex}$ $\underline{5}_{hex}$ $\mathbf{3}_{hex}$

- In WinDbg memory addresses are always displayed in hexadecimal notation

Binary <-> Decimal <-> Hexadecimal

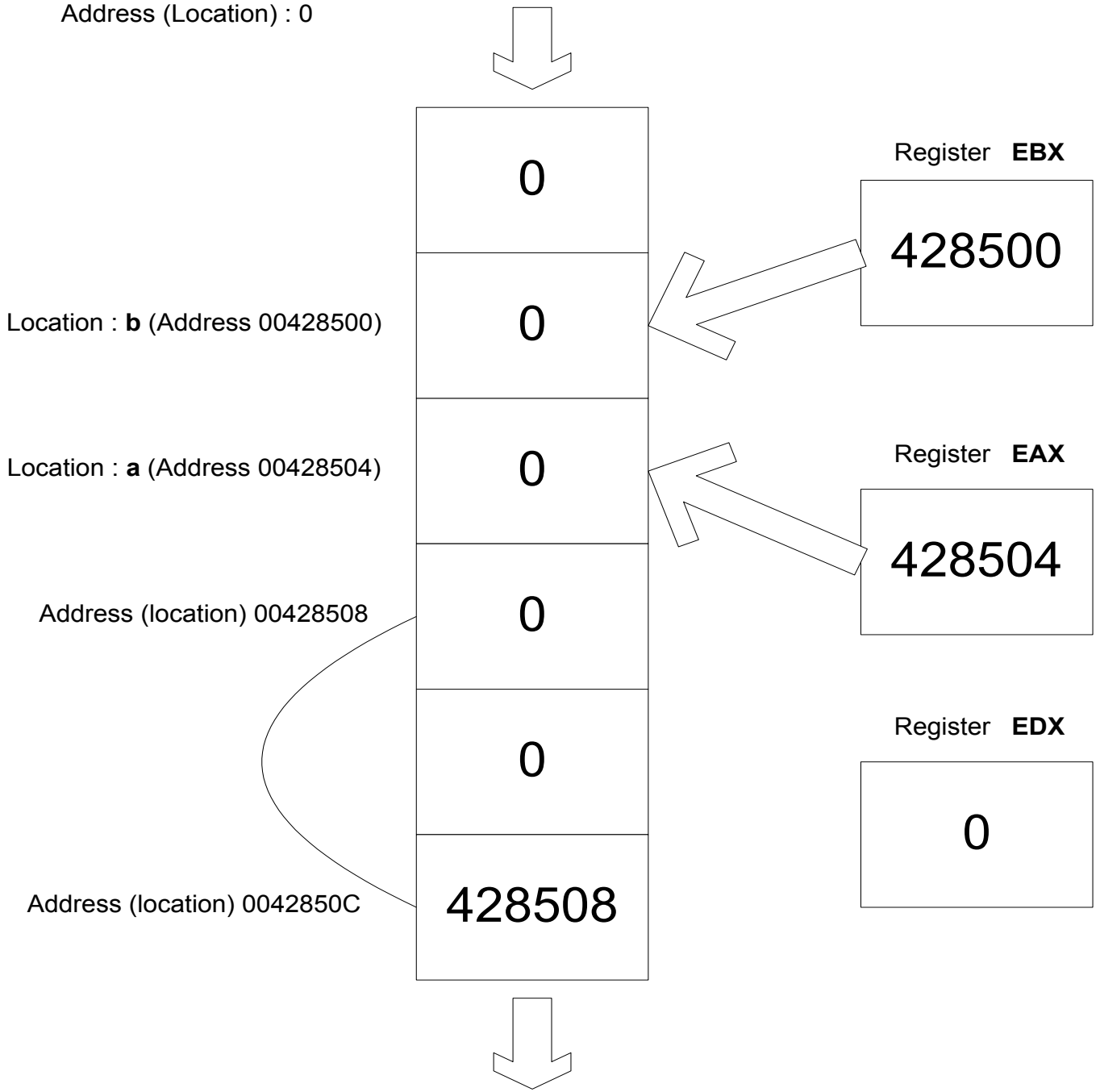
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

Pointers (Picture 1)

- Pointer is a memory cell or a register that contains the address of another memory cell. Has its own address (as any memory cell)
- Another name: Indirect address (vs. direct address, the address of memory cell). Another level of indirection
- Levels of indirection: pointer to a pointer

  (Memory cell or register that contains the address of another memory cell that contains the address of another memory cell)

Picture 1

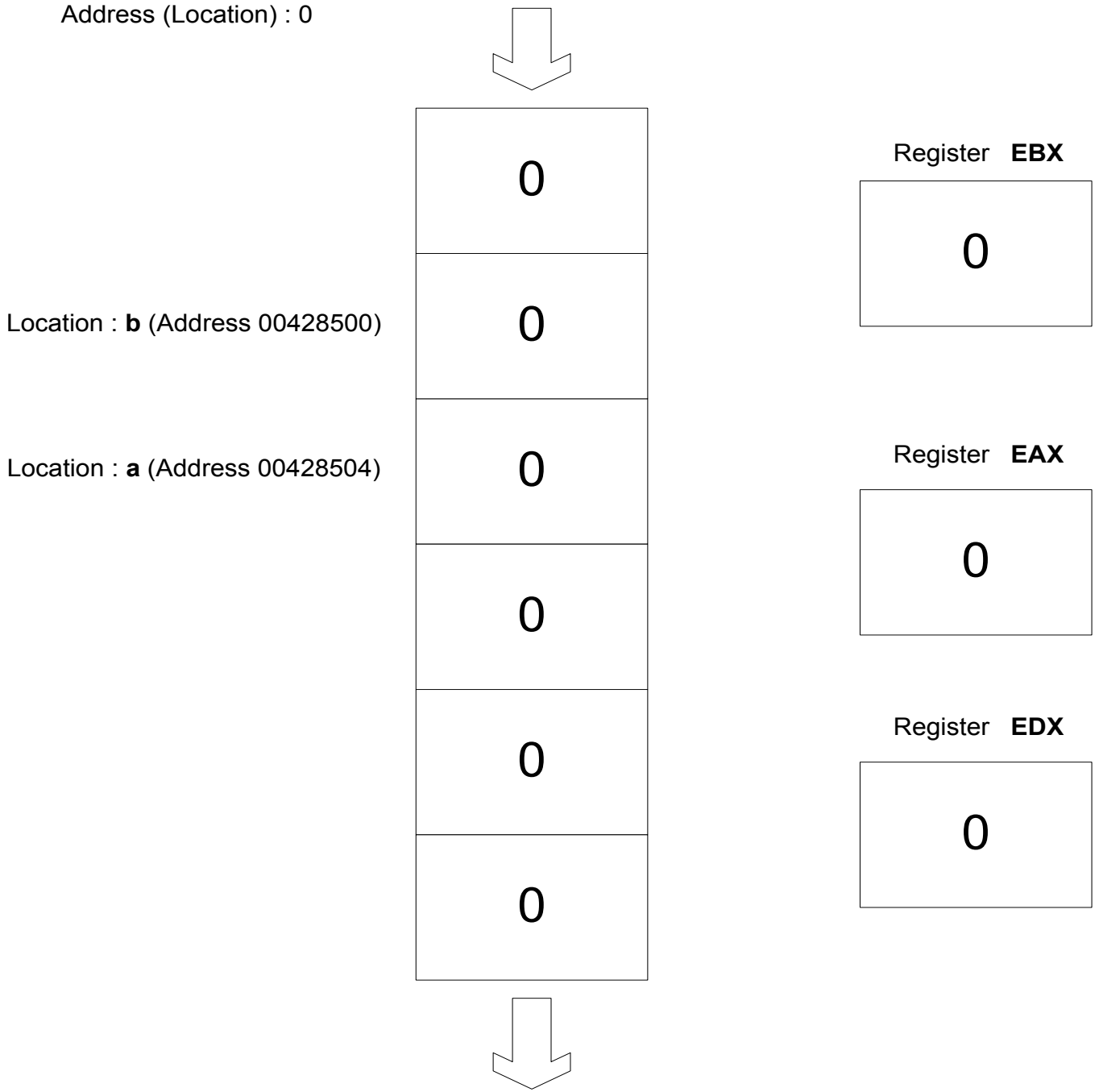

Address (Location) : 0
0
0
0
0
0
428508
Location : **b** (Address 00428500)
Location : **a** (Address 00428504)
Address (location) 00428508
Address (location) 0042850C
Register **EBX**
428500
Register **EAX**
428504
Register **EDX**
0

“Pointers” Project – Memory Layout and Registers (Picture 2)

- Two memory addresses (locations):
  “a” and “b”. We can think about “a” and “b” as names of addresses (locations)
- Notation [a] means contents at the memory address (location) “a”
- Registers *EAX* and *EBX;* pointers to “a” and “b”; contain addresses of “a” and “b”
- Notation [EAX] means contents of the memory cell whose address is in EAX
- In C we declare pointers to “a” and “b” as:

int *a, *b;

Picture 2


Address (Location) : 0
0
0
Location : **b** (Address 00428500)
0
Location : **a** (Address 00428504)
0
0
0
Register **EBX**
0
Register **EAX**
0
Register **EDX**
0

“Pointers” Project - Calculations

```
eax := address a
[eax] := 1    ; [a] = 1
ebx := address b
[ebx] := 1    ; [b] = 1
[ebx] := [ebx] + [eax]
                  ; [b] = 2
[ebx] := [ebx] * 2
                  ; [b] = 4
```

Using pointers to assign numbers to memory cells

- eax := address a
- [eax] := 1

means using the contents of eax as address of a memory cell and assign a value to the contents of this memory cell

- In C language it is called a “pointer” and we write:

```
int *a;      // definition of a pointer
*a = 1;      // get memory cell (dereferencing a pointer) and assign a value to it
```

- In Assembler we write:

```
lea   eax, a        ; load the address a into eax
mov [eax], 1        ; use eax as a pointer
```

- In WinDbg disassembly output we see:

```
00411a2e 8d0504854200    lea     eax,[PointersProject!a (00428504)]
00411a34 c60001          mov     byte ptr [eax],0x1
```

“Arithmetical” Project – Calculations (Picture 3)

eax := address a

[eax] := 1 *; [a] = 1*

ebx := address b

[ebx] := 1 *; [b] = 1*

[ebx] := [ebx] + [eax]

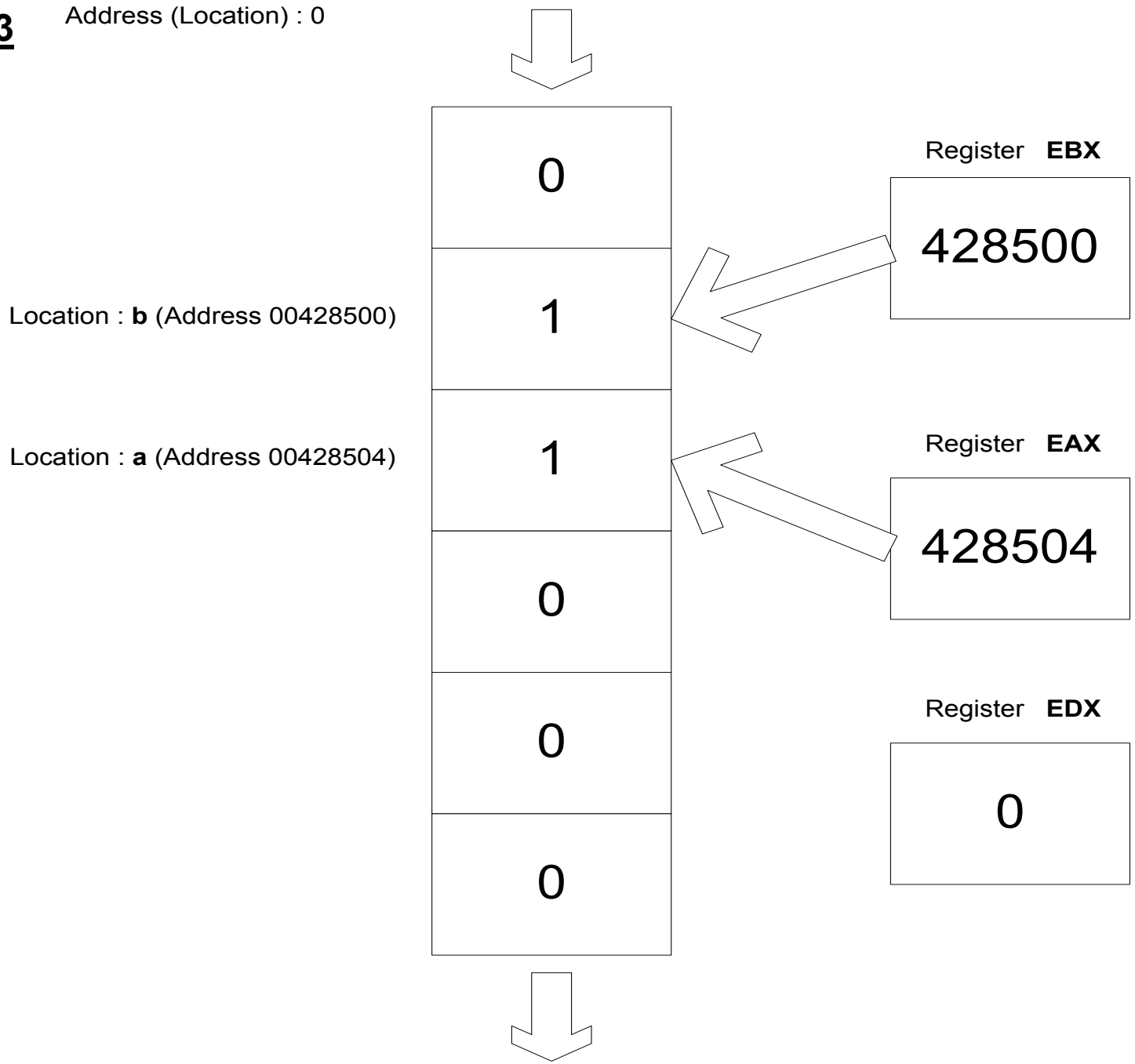
; [b] = 2

[ebx] := [ebx] * 2

; [b] = 4

```
lea eax, a
mov [eax], 1
lea ebx, b
mov [ebx], 1
```

Picture 3


Address (Location) : 0
0
1
1
0
0
0
Location : **b** (Address 00428500)
Location : **a** (Address 00428504)
Register **EBX**
428500
Register **EAX**
428504
Register **EDX**
0

Adding numbers using pointers

- [ebx] := [ebx] + [eax]

[eax] and [ebx] mean contents of memory cells whose addresses (locations) are stored in eax and ebx

- In C language we write:

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

- In Assembler we use instruction **add**

We cannot use both memory addresses in one step (instruction):

add [ebx], [eax]

We can only use **add [ebx], register**

register := [eax]

[ebx] := [ebx] + register

- In Assembler we write:

mov eax, [eax]

add [ebx], eax

- In WinDbg disassembly output we see:

00411a40 8b00 mov eax,[eax]

00411a42 0103 add [ebx],eax

“Arithmetical” Project – Calculations (Picture 4)

eax := address a

[eax] := 1 *; [a] = 1*

ebx := address b

[ebx] := 1 *; [b] = 1*

[ebx] := [ebx] + [eax]

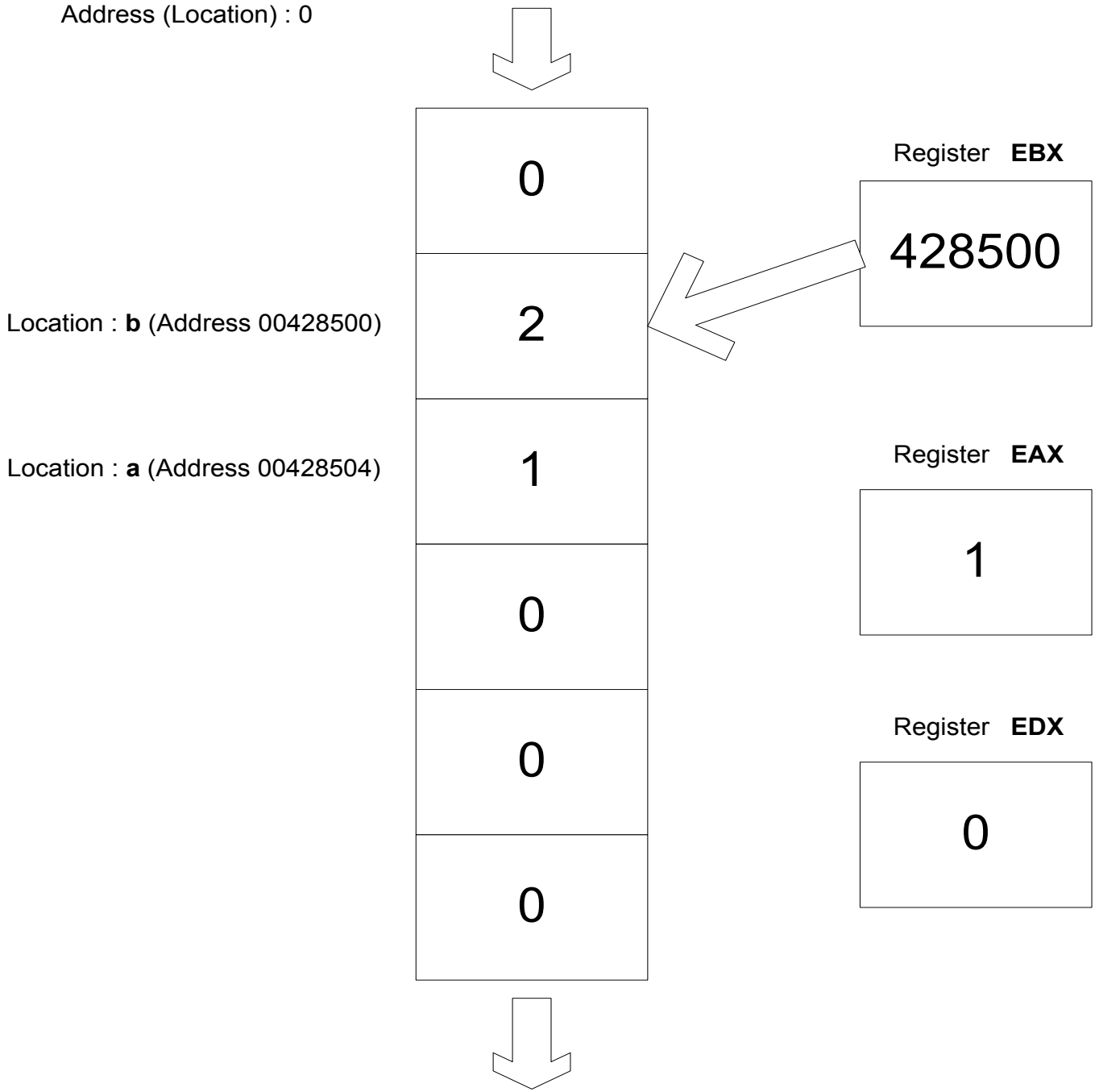
; [b] = 2

[ebx] := [ebx] * 2

; [b] = 4

```
lea eax, a
mov [eax], 1
lea ebx, b
mov [ebx], 1
mov eax, [eax]
add [ebx], eax
```

Picture 4


Address (Location) : 0
0
2
1
0
0
0
Location : **b** (Address 00428500)
Location : **a** (Address 00428504)
Register **EBX**
428500
Register **EAX**
1
Register **EDX**
0

Multiplying numbers using pointers

- [ebx] := [ebx] * 2

Means multiply contents of the memory whose address is stored in ebx by 2

- In C language we write:

***b = *b * 2;** or ***b *= 2;**

- In Assembler we use instruction **imul** (integer multiply)

imul [ebx]

Means [ebx] := [ebx] * eax, so we have to put 2 into eax, but we already have 1 in **eax** so we use **inc eax** before **imul** to increment by 1

Result of multiplication is put into registers eax only! This is because the compiler recognized that we multiply small numbers.

- In WinDbg disassembly output we see:

```
00411a44 40                  inc     eax
00411a45 f62b                imul    byte ptr [ebx]
00411a47 8903                mov     [ebx],eax
```

“Arithmetical” Project – Calculations (Picture 5)

eax := address a

[eax] := 1 *; [a] = 1*

ebx := address b

[ebx] := 1 *; [b] = 1*

[ebx] := [ebx] + [eax]

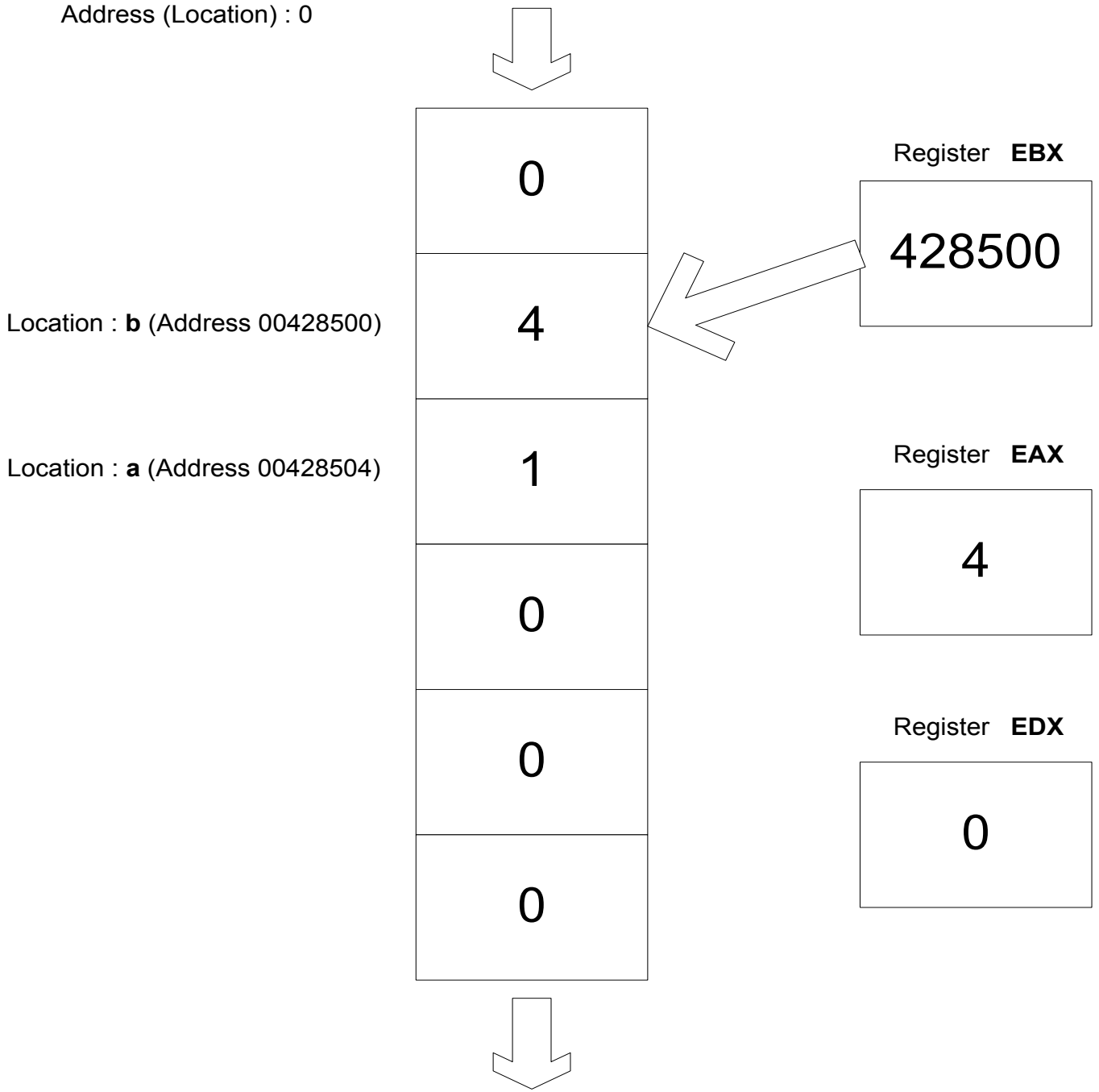
; [b] = 2

[ebx] := [ebx] * 2

; [b] = 4

```
lea eax, a
mov [eax], 1
lea ebx, b
mov [ebx], 1
mov eax, [eax]
add [ebx], eax
inc eax
imul [ebx]
mov [ebx], eax
```

Picture 5


Address (Location) : 0
0
4
1
0
0
0
Location : **b** (Address 00428500)
Location : **a** (Address 00428504)
Register **EBX**
428500
Register **EAX**
4
Register **EDX**
0

What’s next?

- More practice with binary and hexadecimal notations.
- Bits, bytes, words and double words.
- Pointers to bytes and double words.
- Pointers as variables. Null pointer.
- We will rewrite our “arithmetical” project using pointers as variables.