Тема:

Операнди команд та способи адресації операндів

Лекція 4

Викладач: Порєв Віктор Миколайович

Операнди команд та способи адресації операндів

У системі команд архітектури IA-32 машинні інструкції сприймають нуль або більше операндів.
Якщо операнд вказує, звідки взяти дані для обробки командою – такий операнд зветься операндом-джерелом (*source operand*).
Операнд, який вказує, куди записати результати роботи команди – це операнд призначення (*destination operand*).

Дані для **операнду-джерела** можуть бути записані:

- як числове значення безпосередньо в інструкції - безпосередній операнд (*immediate operand*)
- у регістрі
- у пам'яті
- у порті вводу-виводу

Операнд **призначенн**я може вказувати на:

- регістр
- пам'ять
- порт вводу-виводу.

Безпосередні операнди

Деякі команди процесора у якості операнду-джерела можуть використовувати дані, наведені як числа безпосередньо у запису інструкції, наприклад:

mov ecx, 10

ця інструкція означає, що у регістр ECX треба записати число 10.
Або ще один приклад:

add eax, 13

тут до вмісту регістра EAX додаватиметься 13.

Усі арифметичні команди (окрім DIV та IDIV) можуть сприймати такі безпосередньо записані дані. Максимальне значення, яке може бути у якості безпосереднього операнду, залежіть від інструкції, проте ніколи (?) не може бути більшим, ніж значення цілого подвійного слова (2^{32}-1).

Відзначимо, що у літературі такий спосіб вказування операндів інколи називають *прямою адресацією* [Зубков], проте, згідно термінології Intel, це – "*immediate operands*" [Software Dev. Manual].

Регістрові операнди

Приклад використання регістрових операндів:

mov cx, ax

означає записати у регістр CX вміст регістра AX.

Наступна інструкція:

add eax, edx

означає додати до вмісту регістру EAX значення регістра EDX.

Регістрові операнди

У системі команд **архітектури IA-32** у якості операнду-джерела та операнду призначення можуть використовуватися такі регістри:

- 32-бітні регістри загального призначення (EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP)
- 16-бітні регістри загального призначення (AX, BX, CX, DX, SI, DI, SP, BP)
- 8-бітні регістри загального призначення (AH, BH, CH, DH, AL, BL, CL, DL)
- сегментні регістри (CS, DS, SS, ES, FS, GS)
- регістр EFLAGS
- регістри x87 FPU: регістри даних - від ST0 до ST7, слово статусу (*status word*), слово керування (*control word*), слово тегу (*tag word*), вказівник на операнди (*data operand pointer*), та вказівник інструкцій (*instruction pointer*)
- регістри MMX (від MM0 до MM7)
- регістри XMM (від XMM0 до XMM7) а також регістр MXCSR
- регістри керування (CR0, CR2, CR3, CR4) та регістри системних таблиць (GDTR, LDTR, IDTR, та регістр задач)
- регістри налагодження (*debug registers*) – DR0, DR1, DR2, DR3, DR6, DR7
- регістри MSR

Які з цих регістрів можливо використати – це залежіть від конкретної команди а також від інших чинників, наприклад, **рівня привілеїв**.

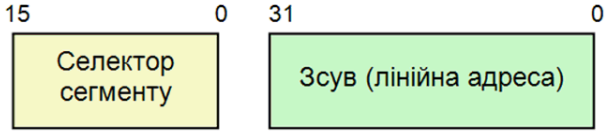
Регістрові операнди у 64-бітному режимі

- 64-бітні регістри загального призначення (RAX, RBX, RCX, RDX, RSI, RDI, RSP, RBP, R8-R15)
- 32-бітні регістри загального призначення (EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP, R8D-R15D)
- 16-бітні регістри загального призначення (AX, BX, CX, DX, SI, DI, SP, BP, R8W-R15W)
- 8-бітні регістри загального призначення: AL, BL, CL, DL, SIL, DIL, SPL, BPL. А також R8LR15L, які доступні при запису REX префіксів.
- сегментні регістри (CS, DS, SS, ES, FS, GS)
- регістр RFLAGS
- регістри x87 FPU : регістри даних – від ST0 до ST7, слово статусу (*status word*), слово керування (*control word*), слово тегу (*tag word*), вказівник на операнди (*data operand pointer*), та вказівник інструкцій (*instruction pointer*)
- регістри MMX (від MM0 до MM7)
- регістри XMM (від XMM0 до XMM7) а також регістр MXCSR
- регістри керування (CR0, CR2, CR3, CR4) та регістри системних таблиць (GDTR, LDTR, IDTR, та регістр задач)
- регістри налагодження (*debug registers*) – DR0, DR1, DR2, DR3, DR6, DR7
- регістри MSR
- регістрова пара RDX:RAX, яка репрезентує 128-бітний операнд

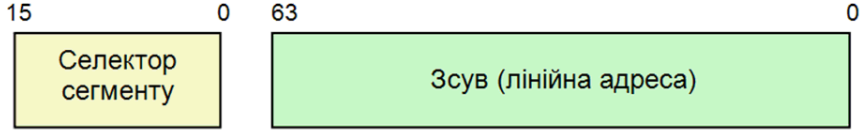
Операнд пам'ять

Операнд пам'ять вказує на потрібну комірку пам'яті, наприклад, на перший байт блока памяті, з якого потрібно прочитати, або куди записати, декілька байтів. Операндами цього типу можуть бути як операнд-джерело, так і операнд призначення.

Адреса розташування потрібної комірки описується селектором сегменту та зсуву. Селектор вказує сегмент. Зсув – відстань від початку сегменту – означає ефективну або лінійну адресу операнду. Зсув може бути 32-бітним (для нотації запису адреси **m16:32**) або 16-бітним (для нотації **m16:16**)



Операнд пам'ять у 64-бітному режимі також описується селектором сегменту та зсувом. Зсув може бути 16-бітним, 32-бітним або 64-бітним



Вказування селектора сегменту у програмах на Асемблері

Найбільш розповсюджений метод вказування селектора сегменту – **завантаження його у сегментний регістр** і далі значення цього регістру використовується у залежності від контексту виконання інструкцій

Тип посилання	Сегментний регістр	Тип сегменту	Зазвичай використовується для
Команди	CS	Сегмент коду	Для зберігання програмного коду
Стек	SS	Сегмент стеку	Для доступу до пам'яті стеку, використовуючи ESP або EBP регістр у якості регістру бази.
Локальні дані	DS	Сегмент даних	Посилання на будь-які дані, за винятком використання стеку або посилання на строку призначення.
Дані	ES	Сегмент даних, на який вказує регістр ES	Операнд призначення для команд обробки рядків

Наприклад:

```
mov eax, [esp]    ; використання регістру ESP означає SS:[ESP] – доступ до сегменту стеку SS
mov eax, [ebx]    ; тут це означає DS:[EBX] – за умовчанням доступ до даних з сегменту DS
```

Щоб використати інший сегмент, замість встановленого за умовчанням для контексту певної інструкції, можна використати оператор ":". Наприклад:

```
mov eax, es:[ebx]
```

означає запис у регістр EAX значення з пам'яті, сегмент якої вказується регістром ES, а зсув відносно цього сегменту знаходиться у регістрі EBX. Тут потрібно явно вказувати регістр ES, оскільки, якщо це не зробити, буде за умовчанням використаний сегмент, на який вказує інший регістр – регістр DS

Вказування зсуву. Пряма та опосередкована адресація

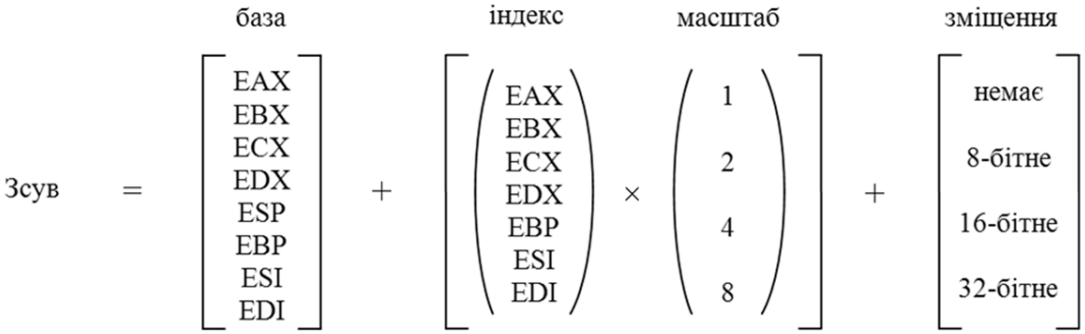
Компонента зсуву (*offset*), яка є частиною опису адреси в пам'яті, може представлятися у інструкціях по-різному. Загалом можуть використовуватися чотири компоненти опису зсуву:

- **зміщення** (*displacement*) – статичне числове значення 8-, 16-, або 32-бітне
- **база** (*base*) – значення у регістрі загального призначення
- **індекс** (*index*) – значення у регістрі загального призначення
- **масштаб** (*scale factor*) – число 2, 4 або 8, яке помножується на значення індексу

Кожна компонента може бути позитивною обо від'ємною (за винятком масштабу).

Зсув обчислюється додаванням вказаних компонент: зсув = база + (індекс × масштаб) + зміщення

Зсув ще зветься **ефективною адресою**.



Узагальнена схема можливих способів опису зсуву для адреси у визначеному сегменті

Вказування зсуву. Пряма та опосередкована адресація

База, індекс та зміщення можуть записуватися у будь-яких комбініціях, значення кожних з цих компонент може бути нульовим. Масштаб може записуватися тільки тоді, якщо присутня компонента індексу.

Таке різноманіття форм запису ефектиної адреси передбачене для зручного доступу до різних типів структур даних. При розробці програми на мові високого рівня компілятор сам знайде адекватну форму опису ефективної адреси – комбінацію компонент зміщення, бази, індексу та масштабу. При програмуванні на асемблері програміст сам вибирає форму запису ефективної адреси. На рівні машинного коду комбінація вказаних компонент кодується у інструкцію для процесора.

Можна вказати на такі обмеження адресації операндів для регістрів загального призначення:

- регістр ESP не може використовуватия у якості індексного регістру
- коли регістри ESP або EBP використовуються у якості базових, тоді сегмент стеку (SS) є сегментом за умовчанням. **В усіх інших випадках за умовчанням адресується сегмент DS.**

Далі розглянемо деякі окремі варіанти використання компонент та їхніх комбінацій для запису ефективної адреси.

Вказування зсуву. Пряма адресація. Зміщення

Зміщення записується константою. Іншими словами, зсув тут прямо визначається деяким числом. Наприклад:

mov ax, es:0001h

означатиме, що у регістр AX буде записано слово, яке знаходиться у памяті у сегменті ES зі зсувом 0001 від початку цього сегменту.

Замість числового значення може використовуватися ім'я перемінної, наприклад:

```
.data
  myVar db ?                ;одна перемінна розміром байт
  myArray dd 1,2,3,4,5,6,7,8  ;масив з 8 подвійних слів
.code
  mov ah, myVar             ;запис у регістр AH
  mov ah, [myVar]           ;синоніми
  mov ah, byte ptr[myVar]   ;синоніми

  mov eax, [myArray+4]      ;запис у регістр EAX другого елементу масиву myArray
```

Асемблер автоматично замінить у цих інструкціях імена перемінних на їхні адреси – зсув відносно початку сегменту даних DS. Ім'я **myVar** та вираження **myArray+4** у програмному коді буде замінено на адреси пам'яті.

Ця форма запису зсуву інколи зветься *прямою* або *абсолютною* або *статичною* адресацією

Вказування зсуву. Опосередкована адресація. База

Для опису зсуву може використовуватися ім'я регістру. Наприклад:

mov eax, es:[ebx]

тут значення зсуву береться з регістру EBX. Вміст регістру бази можна вільно змінювати упродовж роботи програми – а відповідно змінюється зсув відносно сегменту ES. Це можна використовувати для підтримки динамічного зберігання перемінних та інших структур даних.

Таку адресацію інколи звуть *опосередкованою* (*косвенной* рос.) адресацією.

Якщо сегмент явно не вказується, то необхідно враховувати правила визначення сегментів за умовчанням, наприклад:

mov eax, [ebx]

то буде використаний сегмент даних (DS), який є сегментом по умовчанню. Проте, якщо для бази записати регістр ESP, EBP або BP і явно не вказати сегмент, то буде використовуватися вже сегмент стеку SS, як у наступному прикладі:

mov eax, [ebp]

Вказування зсуву. База + зміщення

Поєднання бази та зміщення для обчислення зсуву (ефективної адреси) можна записати так:

mov ax, [ebx+2]

Тут вказане зміщення = 2, що означатиме, що у регістр AX запишеться слово, яке є у сегменті, вказаному у регістрі DS, зі зсувом на два більшим, ніж число у EBX. Вказування зміщення можна використати для доступу до якогось єлемента структури даних – якщо у регістрі бази записана адреса початку цієї структури, а зміщення вказувати відповідно до розміру елементів та позиції елемента у структурі:

```
mov ax, [ebx]      ; перше слово структури (зміщення немає)
mov ax, [ebx+2]    ; друге слово
mov ax, [ebx+4]    ; третє слово
mov ax, [ebx+6]    ; четверте слово
```

Вказування зсуву. База + зміщення

Такий спосіб адресації операндів часто використовується у процедурах, яким перед викликом були передані аргументи (параметри) через стек. Щоб прочитати потрібне слово зі стеку у якості регістру бази використовується EBP та вказується відповідне зміщення, наприклад:

mov ax, [ebp+4] ; читання другого подвійного слова зі стеку

Необхідно зазначити, що варіант [база+зміщення] можна записати також по-іншому:

mov ax, [ebp]+4

або так:

mov ax, 4[ebp]

В усіх трьох прикладах обчислюється однакове значення зсуву, тобто такі форми запису є синонімами в Асемблері.

Вказування зсуву. Індекс × Масштаб + Зміщення

Цей спосіб адресації операндів є зручним для доступу до елементів статичних масивів по індексам цих елементів. Якщо зміщення зберігає адресу початку масиву, то значення масштабу 2, 4 та 8 може слугувати для вказування відповідно 2-, 4- та 8-байтових елементів масивів. Наприклад:

```
.data
  MyArray dd 10,20,30,40,50,60,70,80,90,100     ;масив з 10 подвійних слів
.code
  mov esi, 5                    ;це буде індекс шостого елементу масиву
  mov eax, dword ptr MyArray[esi*4]    ;EAX=60 – це значення елементу MyArray[5]
  mov eax, MyArray[4*esi]           ;синонім
  mov eax, [4*esi+MyArray]          ;синонім
  mov eax, [4*esi][MyArray]         ;синонім
  mov eax, [4*esi]+MyArray          ;синонім

  mov eax, [4*esi]MyArray           ;помилка, так не можна
```

У такий спосіб запрограмоване читання шостого 4-байтового елементу масиву – спочатку у регістр ESI записується індекс 5, а потім для команди MOV вказується цей індекс разом із масштабом 4.

Вказування зсуву. Індекс × Масштаб + Зміщення

Багато різновидів запису, наведених вище, є синонімами – компілюються у той самий машинний код. У цьому можна переконатися, якщо розглянути фрагмент вмісту вікна дизасемблера:

```
  mov esi, 5
00401065  mov         esi,5
  mov eax, [4*esi]+MyArray
0040106A  mov         eax,dword ptr MyArray (40401Eh)[esi*4]
  mov eax, dword ptr MyArray[esi*4]
00401071  mov         eax,dword ptr MyArray (40401Eh)[esi*4]
  mov eax, MyArray[4*esi]
00401078  mov         eax,dword ptr MyArray (40401Eh)[esi*4]
  mov eax, [4*esi+MyArray]
0040107F  mov         eax,dword ptr MyArray (40401Eh)[esi*4]
  mov eax, [4*esi][MyArray]
00401086  mov         eax,dword ptr MyArray (40401Eh)[esi*4]
  mov eax, [4*esi]+MyArray
0040108D  mov         eax,dword ptr MyArray (40401Eh)[esi*4]
```

Вказування зсуву. Індекс × Масштаб + Зміщення

У якості **індексного** регістру не може використовуватися регістр ESP.
Приклади помилок:

```
mov eax, dword ptr MyArray[esp*4]      ;помилка, так не можна
mov eax, MyArray[4*esp]                ;помилка, так не можна
mov eax, [4*esp+MyArray]               ;помилка, так не можна
mov eax, [4*esp][MyArray]              ;помилка, так не можна
```

Вказування зсуву. База + Індекс + Зміщення

Тут вже використовуються два регістри – один для бази, другий для індексу. Також можна вказувати зміщення.

Адреса (зсув) розташування операнда у памяті обчислюється як сума значення регістру бази плюс значення регістру індексу плюс зміщення (якщо воно є). Усі наведені нижче інструкції означають однакові дії:

```
mov ax, [bx+si+2]                ;синонім
mov ax, [bx][si]+2               ;синонім
mov ax, [bx+2][si]               ;синонім
mov ax, [bx][si+2]               ;синонім
mov ax, 2[bx][si]                ;синонім
```

У регістр AX записується слово із двох байтів, розташоване у пам'яті зі зсувом, який обчислюється як **сума значень регістрів BX, SI та числа 2.**

Вказування зсуву. База + Індекс + Зміщення

Такий спосіб обчислення ефективної адреси можна використати, наприклад, до організації доступу до статичного двовимірного масиву. Наприклад, читання однобайтового елементу двовимірного масиву **MyArray[ ][ ]**, який загалом містить 12 байтів, можна запрограмувати як

```
.data
  MyArray db 0,1,2,3,4,5,6,7,8,9,10,11

.code

  mov al, MyArray[ebx][esi]          ;EBX=? ESI=?
```

Асемблер транслює інструкцію копіювання у наступнй код:

```
mov al, MyArray[ebx][esi]
00401071  mov         al,byte ptr MyArray (40401Eh)[esi+ebx]
```

Тут вже наочно видно, що адреса елементу буде обчислюватися як **сума** ESI + EBX.

Можна поставити питання: оскільки масив MyArray записується у памяті як одномірний вектор байтів, то які значення повинні бути записані у регістрах EBX та ESI щоб прочитати елемент з індексами [2][1] масиву розмірами [3][4] ? Вирішить це питання самостійно у якості вправи.

Вказування зсуву. База + Індекс × Масштаб + Зміщення

У цьому варіанті задіяні усі чотири компоненти адресації операнду.
Масштаб може бути 2, 4 або 8 (якщо масштаб 1 – тоді він взагалі не вказується).
Приклад:

mov eax, [ebx+4*esi+MyData]

Така схема надає можливості зручно робити доступ до 2-, 4- та 8-байтових елементів різноманітних структур даних.

Стек

Стек

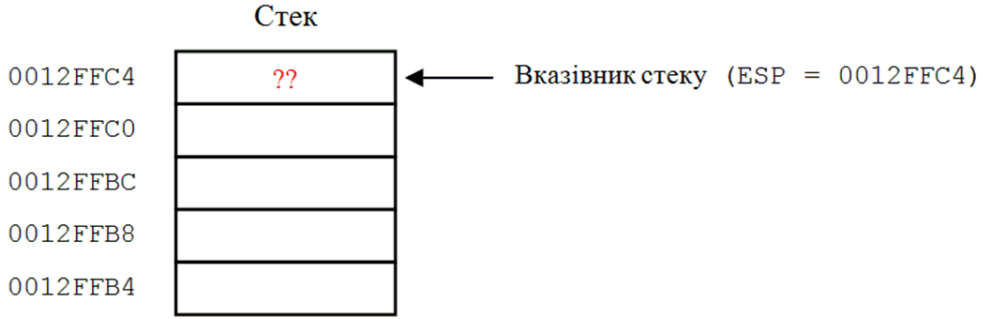
Стек – це зарезервований блок пам'яті, доступ до якого організований спеціальним чином. Стек розташовується у сегменті, селектор якого записується у регістрі SS. Для *flat*-моделі пам'яті стек може розташовуватися будь-де у лінійному адресному просторі, доступному програмі. Теоретично розмір стеку може бути до 4ГБ – це максимум розміру сегменту. На практиці для стеку резервується значно менший обсяг пам’яті.

Стек може зберігати 16-, 32- та 64-бітові елементи відповідно режиму роботи процесора (64-бітовий стек тільки для процесорів Intel 64).

Може бути декілька стеків. Наприклад, у багатозадачній операційній системі кожна задача отримує власний стек. Кількість стеків обмежується максимально можливою кількістю сегментів та розміром доступної фізичної пам'яті. Коли стеків декілька, у кожний момент доступним є тільки один стек, селектор якого зараз міститься у регістрі SS.

Стек

Стек має підтримку з боку процесора – для роботи зі стеком передбачено спеціальні команди. Запис у стек робиться командою PUSH, читання зі стеку – командою POP. Вміст регістру ESP – це вказівник (адреса) на останній записаний елемент у стек. Проілюструємо це на прикладі.

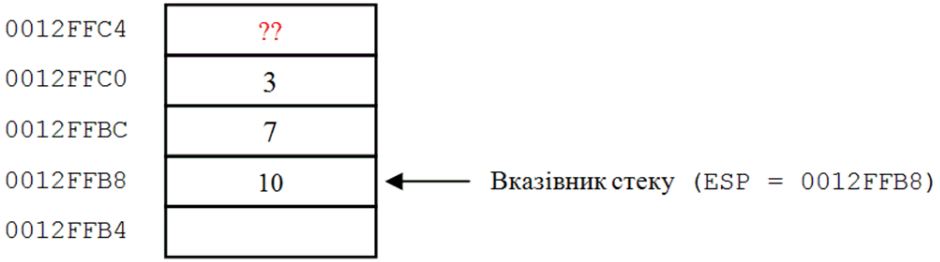


Стек

Після виконання наступних команд PUSH

```
push 3              ; запис числа 3 у стек
push 7              ; запис числа 7 у стек
push 10             ; запис числа 10 у стек
```

стан стеку буде такий:



Для читання зі стеку призначена команда POP:

```
pop eax             ; у регістр EAX запишеться 10
pop ebx             ; у регістр EBX запишеться 7
pop ecx             ; у регістр ECX запишеться 3
```

Стек

Таким чином, якщо для запису у стек виконувалося декілька команд PUSH, то необхідно виконати стільки ж команд POP, тоді вказівник стеку – регістр ESP прийме первісне значення і це означатиме, що стек очистився, звільнився (краще казати: повернувся у попередній стан, хоча і це не зовсім точно). По аналогії "скільки дужок відкрили, стільки повинні закрити".

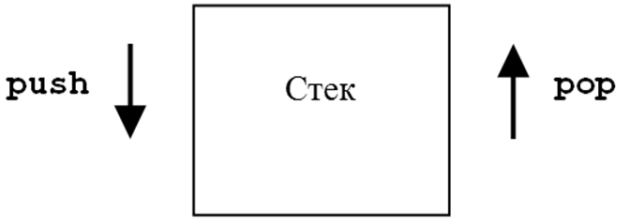
Якщо для доступу до стеку використовувати **тільки** команди PUSH та POP, то для користувача стеку це означає роботу зі сховищем даних, яке працює за принципом "**першим записаний – останім прочитаний**".

Стек

Для кращого розуміння, те, що виконують команди PUSH і POP, можна запрограмувати власноруч наступним чином:

```
; -- push EAX ---
sub esp, 4
mov ss:[esp], eax

; -- pop EAX ----
mov eax, ss:[esp]
add esp, 4
```



Стек

Оскільки відомо, що селектор сегменту стеку міститься у регістрі SS, а зсув відносно початку сегменту (вказівник стеку) – зберігається у регістрі ESP, то у програмах на асемблері часто можна зустріти читання елементів стеку і безпосередньо – без використання команд POP. Для ілюстрації цього розглянемо наступний приклад.

```
; --- запис у стек трьох значень ---
push 3
push 7
push 10

; --- читання зі стеку ----
mov EAX, SS:[ESP]     ; читання останнього записаного - у регістрі EAX буде 10
mov EBX, SS:[ESP+4]   ; у регістр EBX запишеться 7
mov ECX, SS:[ESP+8]   ; у регістр ECX запишеться 3
add ESP, 12           ; немов би очищення стеку – відновлення вказівника ESP
```

У наведеному прикладі очищення стеку запрограмовано як додавання до вказівника 12. Чому саме 12 – тому що у стек були записані 3 чотирибайтові значення. Додавання 12 відновлює попереднє значення вказівника, яке було до виконання трьох команд PUSH.

Стек

Таким чином, для архітектур IA-32 та Intel 64 запис у стек розпочинається з верхніх адрес – з максимального зсуву, який був призначений як первісне значення для вказівника стеку при ініціалізації стеку. По мірі заповнення стеку його елементи посідають місця по нижнім адресам сегменту стеку. Межа заповнення стеку – коли вказівник стеку прийме нульове значення.

Щодо такої реалізації стека, прийнятої для процесорів x86-подібної архітектури, у літературі можна зустріти епітет на кшалт "пам'ять догори дригом" [Зубков].

У деяких середовищах розробки програм, зокрема Microsoft Visual Studio, прийнято за умовчанням, щоб програма створювалася так, щоб їй виділявся стек розміром 1 МБ. Проте, це можливо змінювати.

Стек використовується для зберігання невеличких об'ємів локальних, тимчасових даних, а також для передачі параметрів при виклику процедур.