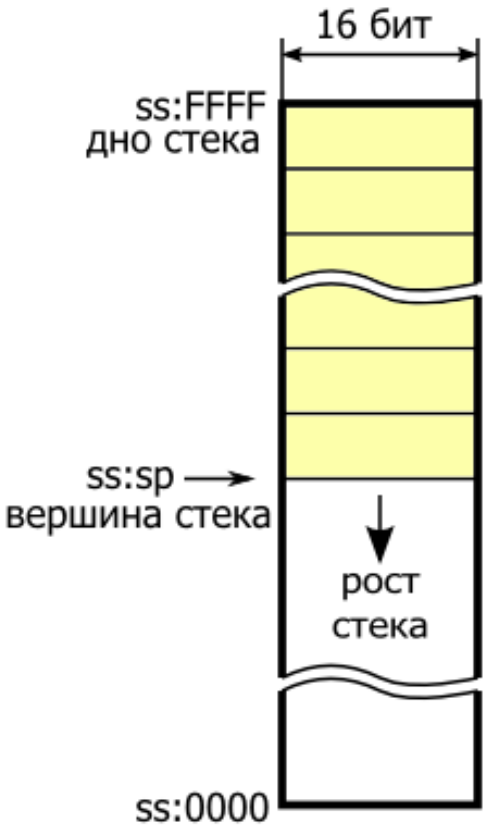
Учебный курс. Часть 20. Стек

Автор: xrnd | Рубрика: Учебный курс | 31-05-2010 | Распечатать запись

Стеком называется структура данных, организованная по принципу LIFO («Last In — First Out» или «последним пришёл — первым ушёл»). Стек является неотъемлемой частью архитектуры процессора и поддерживается на аппаратном уровне: в процессоре есть специальные регистры (SS, BP, SP) и команды для работы со стеком.

Обычно стек используется для сохранения адресов возврата и передачи аргументов при вызове процедур (о процедурах в следующей части), также в нём выделяется память для локальных переменных. Кроме того, в стеке можно временно сохранять значения регистров.

Схема организации стека в процессоре 8086 показана на рисунке:



Стек располагается в оперативной памяти в сегменте стека, и поэтому адресуется относительно сегментного регистра SS. Шириной стека называется размер элементов, которые можно помещать в него или извлекать. В нашем случае ширина стека равна двум байтам или 16 битам. Регистр SP (указатель стека) содержит адрес последнего добавленного элемента. Этот адрес также называется вершиной стека. Противоположный конец стека называется дном 🙂

Дно стека находится в верхних адресах памяти. При добавлении новых элементов в стек значение регистра SP уменьшается, то есть стек растёт в сторону младших адресов. Как вы помните, для COM-программ данные, код и стек находятся в одном и том же сегменте, поэтому если постараться, стек может разрастись и затереть часть данных и кода (надеюсь, с вами такой беды не случится :)).

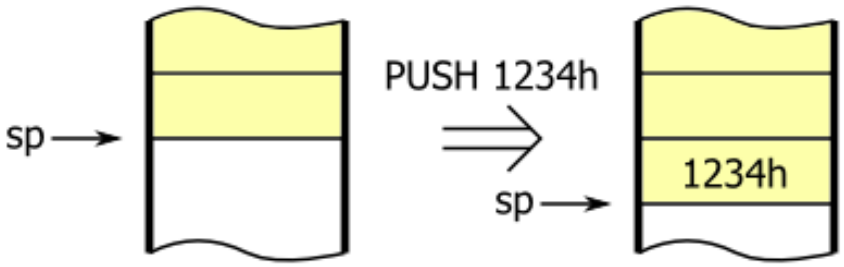
Для стека существуют всего две основные операции:

- добавление элемента на вершину стека (PUSH);
- извлечение элемента с вершины стека (POP);

Добавление элемента в стек

Выполняется командой PUSH. У этой команды один операнд, который может быть непосредственным значением, 16-битным регистром (в том числе сегментым) или 16-битной переменной в памяти. Команда работает следующим образом:

1. значение в регистре SP уменьшается на 2 (так как ширина стека — 16 бит или 2 байта);
2. операнд помещается в память по адресу в SP.



Примеры:

```
    push -5             ;Поместить -5 в стек
    push ax             ;Поместить AX в стек
    push ds             ;Поместить DS в стек
    push [x]            ;Поместить x в стек (x объявлен как слово)
    push word [bx]      ;Поместить в стек слово по адресу в BX
```

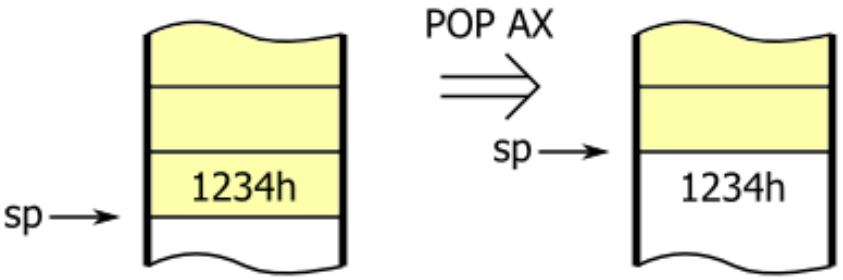
Существуют ещё 2 команды для добавления в стек. Команда PUSHF помещает в стек содержимое регистра флагов. Команда PUSHA помещает в стек содержимое всех регистров общего назначения в следующем порядке: AX, CX, DX, BX, SP, BP, SI, DI (значение DI будет на вершине стека). Значение SP помещается то, которое было до выполнения команды. Обе эти команды не имеют операндов.

Извлечение элемента из стека

Выполняется командой POP. У этой команды также один операнд, который может быть 16-битным регистром (в том числе сегментым, но кроме CS) или 16-битной переменной в памяти. Команда работает следующим образом:

1. операнд читается из памяти по адресу в SP;
2. значение в регистре SP увеличивается на 2.

Обратите внимание, что извлеченный из стека элемент не обнуляется и не затирается в памяти, а просто остаётся как мусор. Он будет перезаписан при помещении нового значения в стек.



Примеры:

```
    pop cx              ;Поместить значение из стека в CX
    pop es              ;Поместить значение из стека в ES
    pop [x]             ;Поместить значение из стека в переменную x
    pop word [di]       ;Поместить значение из стека в слово по адресу в DI
```

Соответственно, есть ещё 2 команды. POPF помещает значение с вершины стека в регистр флагов. POPA восстанавливает из стека все регистры общего назначения (но при этом значение для SP игнорируется).

Пример программы

Имеется двумерный массив — таблица 16-битных значений со знаком размером *n* строк на *m* столбцов. Программа вычисляет сумму элементов каждой строки и сохраняет результат в массиве *sum*. Первый элемент массива будет содержать сумму элементов первой строки, второй элемент — сумму элементов второй строки и так далее.

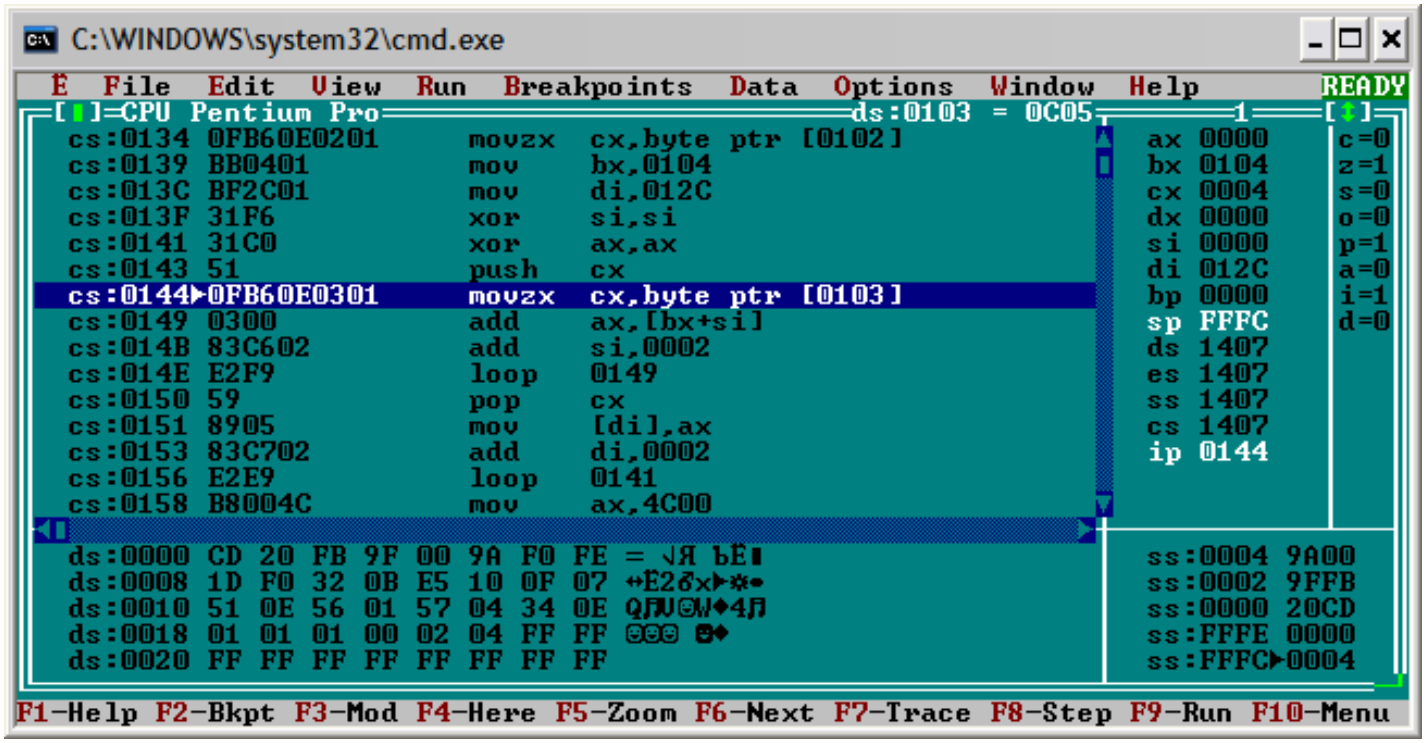
```
 1 use16                ;Генерировать 16-битный код
 2 org 100h             ;Программа начинается с адреса 100h
 3     jmp start        ;Переход к метке start
 4 ;---------------------------------------------------------------------

 5 ; Данные
 6 n   db 4             ;Количество строк
 7 m   db 5             ;Количество столбцов
 8 ;Двумерный массив - таблица с данными
 9 table:
10     dw 12,45, 0,82,34
11     dw 46,-5,87,11,56
12     dw 35,21,77,90,-9
13     dw 44,13,-1,99,32
14 sum rw 4             ;Массив для сумм каждой строки
15 ;---------------------------------------------------------------------
16 start:
17     movzx cx,[n]     ;Счётчик строк
18     mov bx,table     ;BX = адрес таблицы
19     mov di,sum       ;DI = адрес массива для сумм
20     xor si,si        ;SI = смещение элемента от начала таблицы
21
22 rows:
23     xor ax,ax        ;Обнуление AX. В AX будет считаться сумма
24     push cx          ;Сохранение значения CX
25
26     movzx cx,[m]     ;Инициализация CX для цикла по строке
27 calc_sum:
28     add ax,[bx+si]   ;Прибавление элемента строки
29     add si,2         ;SI = смещение следующего элемента
30     loop calc_sum    ;Цикл суммирования строки
```

```
31
32      pop cx              ;Восстановление значения CX
33      mov [di],ax         ;Сохранение суммы строки
34      add di,2            ;DI = адрес следующей ячейки для суммы строки
35      loop rows           ;Цикл по всем строкам таблицы
36
37      mov ax,4C00h        ;\
38      int 21h             ;/ Завершение программы
```

Как видите, в программе два вложенных цикла: внешний и внутренний. Внешний цикл — это цикл по строкам таблицы. Внутренний цикл вычисляет сумму элементов строки. Стек здесь используется для временного хранения счётчика внешнего цикла. Перед началом внутреннего цикла CX сохраняется в стеке, а после завершения восстанавливается. Такой приём можно использовать для программирования и большего количества вложенных циклов.

Turbo Debugger

В отладчике Turbo Debugger стек отображается в нижней правой области окна CPU. Левый столбец чисел — адреса, правый — данные. Треугольник указывает на вершину стека, то есть на тот адрес, который содержится в регистре SP. Если запустить программу в отладчике, то можно увидеть, как работают команды «push cx» и «pop cx».



Упражнение

Объявите в программе строку «$!olleH». Напишите код для переворачивания строки с использованием стека (в цикле поместите каждый символ в стек, а затем извлеките в обратном порядке). Выведите полученную строку на экран. Свои результаты пишите в комментариях 🙂

Следующая часть »

Комментарии:

RoverWWWorm
02-06-2010 09:26

```
;Привет, xrnd!
use16
org 100h

jmp start

;—————————————
string db «$!olleH»
n db 7
;—————————————

start:
xor si,si
movzx cx,[n]
lp1:
add si,1
movzx dx,byte[string+si]
push dx
loop lp1

movzx cx,[n]
mov ah,02h
lp2:
pop dx
int 21h
loop lp2

mov ah,08h
int 21h

mov ax,4C00h
int 21h
```

[Ответить]

xrnd
03-06-2010 00:29

Всё правильно. Только вместо «add si,1» обычно пишут «inc si» 🙂 И я предполагал, что перевёрнутую строку можно записать на место исходной в память и вывести функцией 09h 🙂

[Ответить]

fufel
29-06-2010 21:18

```
use16
org 100h

jmp start
;—————————————
```

```
string db '$!olleH'
;————————————

start:
mov cx,7
xor di,di

lp:
mov al,[string+di]
push ax
inc di
loop lp
xor di,di
mov cx,7
lp_2:
pop ax
mov [string+di],al
inc di
loop lp_2

mov dx,string
mov ah,09h
int 21h
mov ah,08h
int 21h

mov ax,4c00h
int 21h
```

[Ответить]

xrnd
03-07-2010 02:07

Ага, всё правильно 🙂

[Ответить]

IgorKing
27-09-2010 16:13

А как сохранить в стек ah или другой байт, можно, конечно так:

```
push dx
…
movzx dx,al
push dx
…
pop dx
ret
```

Но нельзя ли попроще?

[Ответить]

xrnd
27-09-2010 17:23

Специальных команд для 8-битных регистров нет, так как в стек всегда сохраняется минимум 2 байта.

Чтобы было проще, надо стараться сохранять 16-битные регистры.
Либо можно не делать movzx dx,al , а сохранять AX целиком, вместе со старшей частью.

[Ответить]

Гость
18-01-2011 20:33

```
use16
org 100h
jmp start
;—————————————————————————
vk db '$!olleH'
m du 7
;—————————————————————————
start:
mov cx,[m]
mov si,0
xor dx,dx
s4t4ik1:
mov dl,byte[vk+si]
push dx
inc si
loop s4t4ik1
Print:
mov ah,02h
mov cx,[m]
s4t4ik3:
pop dx
int 21h
loop s4t4ik3
mov ah,08h
int 21h
mov ax,4C00h
int 21h
```

[Ответить]

xrnd
18-01-2011 22:07

Вроде правильно.
Но такое чувство, что некоторые команды лишние 🙂
Например, что-то считается в регистре SI, но дальше никак не используется.
Команда «XOR DX,DX» тоже не нужна.

Еще хотелось бы строку перевернуть и записать на то же место, а затем вывести функцией DOS 09h.

По одному символу можно просто вывести её в цикле, начиная с конца, даже стек не нужен.

[Ответить]

Гость
19-01-2011 00:48

Да переборщил с командами пожалуй.

У меня вопрос возник по поводу PUSHF,POPA,PUSHA,POPF
Они работают также как и PUSH ,
например
PUSHA
push -5
POPA
В таком варианте значения с летают , и в регистрах не те значения которые были
и за нехватке pop перед POPA.
На этом вся разнетца и заканчивается , если отбросить ,что одна команда сохраняет все общие регистры а другая 1регистр .

[Ответить]

xrnd
19-01-2011 18:38

В общем, да. Команды PUSHA/POPA можно заменить на 8 PUSH/POP (однако, небольшое отличие будет с регистром SP).

В твоем варианте значения и должны слетать, потому что в регистры восстановится не то, что нужно.

[Ответить]

kotya
16-11-2015 13:48

>>Команда «XOR DX,DX» тоже не нужна.
она нужна, так как в первом цикле он пушит в стек dx, загоняя в dl значение символов из массива. в dh без xor останется мусор.

другое дело: вайпнуть dh вместо dx, или же вместо mov dl,byte[vk+si] (не знаю, зачем он явно обращается к байту, значение массивов и приёмника и так типа байт) написать movzx dx, [vk+si].

[Ответить]

Knight212
23-02-2011 16:07

My code:
use16
org 100h
jmp start

```
gnirts db '$!olleH'
string db ?
len dw 7

start:
xor bx, bx
mov cx, [len]

lp1:
movzx ax, [gnirts+bx]
inc bx
push ax
loop lp1

xor bx, bx
mov cx, [len]
lp2:
pop word [string+bx]
inc bx
loop lp2

mov ah, 09h
mov dx, string
int 21h

mov ah, 08h
int 21h

mov ax, 4C00h
int 21h
```

[Ответить]

xrnd
25-02-2011 22:22

Очень хорошо.
Только для строки string в памяти нужно 7 байт, а не 1. В твоей программе «хвост» строки перезапишет переменную len и первые команды кода.

[Ответить]

plan4ik
05-04-2011 16:14

```
org 100h
use16
jmp Start

;=================================[DATA]===
test_str db «$76543210»
test_str_len db $-test_str
;=========================================
Start:
```

```
movzx cx, byte [test_str_len]
xor si, si
Reverse:
movzx dx, byte [test_str+si]
push dx
inc si
loop Reverse

movzx cx, byte [test_str_len]
xor si, si
Putin:
pop dx
mov byte [test_str+si], dl
inc si
loop Putin

mov ah, 9
mov dx, test_str
int 21h
mov ah, 8
int 21h
mov ax, 4c00h
int 21h
```

[Ответить]

xrnd
08-04-2011 20:15

Всё правильно.

З.Ы. Увидел в коде метку Путин и не мог понять, причем он тут 🙂

[Ответить]

plan4ik
09-04-2011 18:32

put in test_str

[Ответить]

Shov
08-04-2011 11:13

```
use16
org 100h

xor ax,ax
lp:
mov al,byte[hstring+di]
cmp al,25h ; признак начала строки
je output
inc di
```

```
push ax
jmp lp

output:

pop dx
cmp dl,24h ; признак конца строки
je exit
mov ah,02h
int 21h
jmp output

exit:

mov ah,08h
int 21h
mov ax,4C00h
int 21h

;—
hstring db '$!olleH%'
```

[Ответить]

xrnd
08-04-2011 20:45

Хорошая программа. Такого варианта ещё не было. Метка начала строки позволяет не считать количество символов.

[Ответить]

NimRoen
19-05-2011 15:12

```
;кажется наиболее быстрый вариант применимый к данному случаю 😉
use16
org 100h
   jmp main

;=================================;
strReverse db '$!olleH'
;=================================;

;=================================;
main:
;сохраняем строку в стек
   mov bx,strReverse
   mov ax,[bx]
   xchg al,ah
   push ax
   mov ax,[bx+2]
   xchg al,ah
   push ax
```

```
    mov ax,[bx+4]
    xchg al,ah
    push ax
    mov ax,[bx+6]
    push ax
;вытаскиваем из стека уже перевернутую
    pop ax
    mov [bx],al
    pop ax
    mov word[bx+1],ax
    pop ax
    mov word[bx+3],ax
    pop ax
    mov word[bx+5],ax
;отображаем строку
    mov dx,bx
    mov ah,09h
    int 21h

    mov ax,4c00h
    int 21h
;=================================;
```

[Ответить]

xrnd
23-06-2011 14:57

Согласен. Скорее всего такой вариант кода быстрее, так как цикл преобразован в линейный алгоритм. Однако, работать будет только со строкой из 7 символов.

Ещё можно вот так сделать:

```
    pop word[bx+1]
    pop word[bx+3]
    pop word[bx+5]
```

[Ответить]

алекс
19-03-2012 12:09

```
use16
org 100h

jmp start
press db 13,10,'Press any key…$'
string db '$!olleH'
len db 7

start:

movzx cx,[len]
xor si,si
```

```
cyk1:
movzx ax,[string+si]
push ax
inc si
loop cyk1

movzx cx,[len]
xor si,si

cyk2:
pop ax
mov [string+si],al
inc si
loop cyk2

mov ah,09h
mov dx,string
int 21h
mov ah,09h
mov dx,press
int 21h
mov ah,08h
int 21h
mov ax,4c00h
int 21h
```

[Ответить]

andrew.NET
10-06-2012 13:27

```
USE16
ORG 0x100
JMP START
;_____
STR1 DB '$!olleH'
STR1_LENGTH DW 7
;_____
START:

MOV CX, [STR1_LENGTH]
XOR DI, DI

LOOP1:
MOV AL, [STR1+DI]
PUSH AX
INC DI
LOOP LOOP1

MOV CX, [STR1_LENGTH]
XOR DI, DI

LOOP2:
POP AX
```

```
MOV [STR1+DI], AL
INC DI
LOOP LOOP2

MOV AH, 0x09
MOV DX, STR1
INT 0x21

MOV AH, 0x08
INT 0x21

MOV AX, 0x4C00
INT 0x21
```

[Ответить]

Ваш комментарий

Имя *

Почта (скрыта) *

Сайт

Добавить

Уведомить меня о новых комментариях по email.

Уведомлять меня о новых записях почтой.