

Отчёт по лабораторной работе №8

Дисциплина: Архитектура компьютера

Долгаев Евгений Сергеевич НММбд-01-24

Содержание

1	Цель работы	5
2	Задание	6
3	Теоретическое введение	7
3.1	Организация стека	7
3.1.1	Добавление элемента в стек	8
3.1.2	Извлечение элемента из стека	8
3.2	Инструкции организации циклов	9
4	Выполнение лабораторной работы	11
4.1	Задание для самостоятельной работы	17
5	Выводы	18
	Список литературы	19

Список иллюстраций

3.1	Организация стека в процессоре	8
4.1	Создание рабочего пространства	11
4.2	Создание и работа исполняемого файла	12
4.3	Текст программы	13
4.4	Создание и работа исполняемого файла	13
4.5	Текст программы	13
4.6	Создание и работа исполняемого файла	14
4.7	Создание файла	15
4.8	Создание и работа исполняемого файла	15
4.9	Создание файла	15
4.10	Текст программы	16
4.11	Создание и работа исполняемого файла	16
4.12	Текст программы	16
4.13	Создание и работа исполняемого файла	17
4.14	Текст программы	17
4.15	Создание и работа исполняемого файла	17

Список таблиц

1 Цель работы

Приобретение навыков написания программ с использованием циклов и обработкой аргументов командной строки.

2 Задание

- 1) Выполнение лабораторной работы
 - 1) Реализация циклов в NASM
 - 2) Обработка аргументов командной строки
- 2) Задание для самостоятельной работы

3 Теоретическое введение

3.1 Организация стека

Стек — это структура данных, организованная по принципу LIFO («Last In — First Out» или «последним пришёл — первым ушёл»). Стек является частью архитектуры процессора и реализован на аппаратном уровне. Для работы со стеком в процессоре есть специальные регистры (ss, bp, sp) и команды.

Основной функцией стека является функция сохранения адресов возврата и передачи аргументов при вызове процедур. Кроме того, в нём выделяется память для локальных переменных и могут временно храниться значения регистров.

На рис. 3.1 показана схема организации стека в процессоре.

Стек имеет вершину, адрес последнего добавленного элемента, который хранится в регистре esp (указатель стека). Противоположный конец стека называется дном. Значение, помещённое в стек последним, извлекается первым. При помещении значения в стек указатель стека уменьшается, а при извлечении — увеличивается.

Для стека существует две основные операции:

- добавление элемента в вершину стека (push);
- извлечение элемента из вершины стека (pop).

3.1.1 Добавление элемента в стек

Команда `push` размещает значение в стеке, т.е. помещает значение в ячейку памяти, на которую указывает регистр `esp`, после этого значение регистра `esp` увеличивается на 4. Данная команда имеет один операнд — значение, которое необходимо поместить в стек.

Примеры:

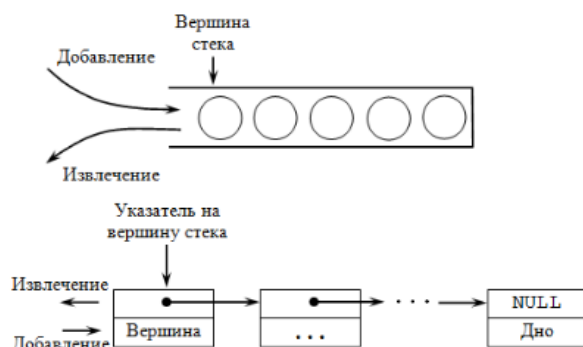


Рис. 3.1: Организация стека в процессоре

`push -10` ; Поместить -10 в стек

`push ebx` ; Поместить значение регистра `ebx` в стек

`push [buf]` ; Поместить значение переменной `buf` в стек

`push word [ax]` ; Поместить в стек слово по адресу в `ax`

Существует ещё две команды для добавления значений в стек. Это команда `pusha`, которая помещает в стек содержимое всех регистров общего назначения в следующем порядке: `ax`, `cx`, `dx`, `bx`, `sp`, `bp`, `si`, `di`. А также команда `pushf`, которая служит для перемещения в стек содержимого регистра флагов. Обе эти команды не имеют операндов.

3.1.2 Извлечение элемента из стека

Команда `pop` извлекает значение из стека, т.е. извлекает значение из ячейки памяти, на которую указывает регистр `esp`, после этого уменьшает значение

регистра `esp` на 4. У этой команды также один операнд, который может быть регистром или переменной в памяти.

Нужно помнить, что извлечённый из стека элемент не стирается из памяти и остаётся как “мусор”, который будет перезаписан при записи нового значения в стек.

Примеры:

```
pop eax ; Поместить значение из стека в регистр eax
```

```
pop [buf] ; Поместить значение из стека в buf
```

```
pop word[si] ; Поместить значение из стека в слово по адресу в si
```

Аналогично команде записи в стек существует команда `push`, которая восстанавливает из стека все регистры общего назначения, и команда `pushf` для перемещения значений из вершины стека в регистр флагов.

3.2 Инструкции организации циклов

Для организации циклов существуют специальные инструкции. Для всех инструкций максимальное количество проходов задаётся в регистре `ecx`. Наиболее простой является инструкция `loop`. Она позволяет организовать безусловный цикл, типичная структура которого имеет следующий вид:

```
mov ecx, 100 ; Количество проходов
```

```
NextStep:
```

```
...
```

```
... ; тело цикла
```

```
...
```

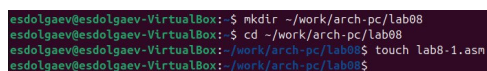
```
loop NextStep ; Повторить 'ecx' раз от метки NextStep
```

Инструкция `loop` выполняется в два этапа. Сначала из регистра `ecx` вычитается единица и его значение сравнивается с нулём. Если регистр не равен нулю,

то выполняется переход к указанной метке. Иначе переход не выполняется и управление передаётся команде, которая следует сразу после команды loop.

4 Выполнение лабораторной работы

Создим каталог для программ лабораторной работы № 8, перейдём в него и создадим файл lab8-1.asm(рис. 4.1).



```
esdolgaev@esdolgaev-VirtualBox: $ mkdir ~/work/arch-pc/lab08
esdolgaev@esdolgaev-VirtualBox: $ cd ~/work/arch-pc/lab08
esdolgaev@esdolgaev-VirtualBox: ~/work/arch-pc/lab08$ touch lab8-1.asm
esdolgaev@esdolgaev-VirtualBox: ~/work/arch-pc/lab08$
```

Рис. 4.1: Создание рабочего пространства

При реализации циклов в NASM с использованием инструкции loop необходимо помнить о том, что эта инструкция использует регистр есх в качестве счетчика и на каждом шаге уменьшает его значение на единицу. В качестве примера рассмотрим программу, которая выводит значение регистра есх.

Введём в файл lab8-1.asm текст программы из листинга 8.1. Создадим исполняемый файл и проверим его работу(рис. 4.2).

Текст программы:

```
%include 'in_out.asm'
SECTION .data
msg1 db 'Введите N: ',0h
SECTION .bss
N: resb 10
SECTION .text
global _start
_start:
; ----- Вывод сообщения 'Введите N: '
```

```

mov eax,msg1
call sprint
; ----- Ввод 'N'
mov ecx, N
mov edx, 10
call sread
; ----- Преобразование 'N' из символа в число
mov eax,N
call atoi
mov [N],eax
; ----- Организация цикла
mov ecx,[N] ; Счетчик цикла, 'ecx=N'
label:
mov [N],ecx
mov eax,[N]
call iprintLF ; Вывод значения 'N'
loop label ; 'ecx=ecx-1' и если 'ecx' не '0'
; переход на 'label'
call quit

```

Рис. 4.2: Создание и работа исполняемого файла

Данный пример показывает, что использование регистра ecx в теле цикла loop может привести к некорректной работе программы. Изменим текст программы

добавив изменение значение регистра `ecx` в цикле. Создадим исполняемый файл и проверим его работу(рис. 4.3, 4.4).

```
label:
sub ecx,1
mov [N],ecx
mov eax,[N]
call iprintLF ; Вывод значения 'N'
loop label ; 'ecx=ecx-1' и если 'ecx' не '0'
; переход на 'label'
call quit
```

Рис. 4.3: Текст программы

```
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab08$ nasm -f elf lab8-1.asm
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab08$ ld -m elf_i386 -o lab8-1 la
b8-1.o
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab08$ ./lab8-1
Введите N: 10
9
7
5
3
1
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab08$
```

Рис. 4.4: Создание и работа исполняемого файла

Регист `ecx` принимает только нечётные значения и число проходов цикла не соответствует введённому `N`.

Для использования регистра `ecx` в цикле и сохранения корректности работы программы можно использовать стек. Внесите изменения в текст программы добавив команды `push` и `pop` (добавления в стек и извлечения из стека) для сохранения значения счетчика цикла `loop`. Создадим исполняемый файл и проверим его работу(рис. 4.5, 4.6).

```
label:
push ecx
sub ecx,1
mov [N],ecx
mov eax,[N]
call iprintLF ; Вывод значения 'N'
pop ecx
loop label ; 'ecx=ecx-1' и если 'ecx' не '0'
; переход на 'label'
call quit
```

Рис. 4.5: Текст программы

```
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab08$ nasm -f elf lab8-1.asm
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab08$ ld -m elf_i386 -o lab8-1 lab8-1.o
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab08$ ./lab8-1
Введите N: 2
1
0
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab08$ ./lab8-1
Введите N: 10
9
8
7
6
5
4
3
2
1
0
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab08$
```

Рис. 4.6: Создание и работа исполняемого файла

В этом случае количество проходов цикла соответствует введённому N.

Создайте файл lab8-2.asm в каталоге ~/work/arch-pc/lab08 и введите в него текст программы(рис. 4.7).

Текст программы:

```
%include 'in_out.asm'
SECTION .text
global _start
_start:
pop ecx ; Извлекаем из стека в 'ecx' количество
; аргументов (первое значение в стеке)
pop edx ; Извлекаем из стека в 'edx' имя программы
; (второе значение в стеке)
sub ecx, 1 ; Уменьшаем 'ecx' на 1 (количество
; аргументов без названия программы)
next:
cmp ecx, 0 ; проверяем, есть ли еще аргументы
jz _end ; если аргументов нет выходим из цикла
; (переход на метку '_end')
pop eax ; иначе извлекаем аргумент из стека
call sprintLF ; вызываем функцию печати
loop next ; переход к обработке следующего
; аргумента (переход на метку 'next')
```

`_end:`

`call quit`

```
esdolgaev@esdolgaev-VirtualBox: ~/work/arch-pc/lab08$ touch lab8-2.asm
esdolgaev@esdolgaev-VirtualBox: ~/work/arch-pc/lab08$ ls
in_out.asm lab8-1 lab8-1.asm lab8-1.o lab8-2.asm
esdolgaev@esdolgaev-VirtualBox: ~/work/arch-pc/lab08$
```

Рис. 4.7: Создание файла

Создадим исполняемый файл и проверим его работу(рис. 4.8).

```
esdolgaev@esdolgaev-VirtualBox: ~/work/arch-pc/lab08$ nasm -f elf lab8-2.asm
esdolgaev@esdolgaev-VirtualBox: ~/work/arch-pc/lab08$ ld -n elf_i386 -o lab8-2 lab8-2.o
esdolgaev@esdolgaev-VirtualBox: ~/work/arch-pc/lab08$ ./lab8-1 1 2 '1'
Введите N: 2
1
0
esdolgaev@esdolgaev-VirtualBox: ~/work/arch-pc/lab08$ ./lab8-2 1 2 '1'
1
2
1
esdolgaev@esdolgaev-VirtualBox: ~/work/arch-pc/lab08$ ./lab8-2 4 3 '5'
4
3
5
esdolgaev@esdolgaev-VirtualBox: ~/work/arch-pc/lab08$
```

Рис. 4.8: Создание и работа исполняемого файла

Программа обработала 3 аргумента.

Рассмотрим еще один пример программы которая выводит сумму чисел, которые передаются в программу как аргументы. Создадим файл lab8-3.asm в каталоге ~/work/arch-pc/lab08 и введём в него текст программы.

```
esdolgaev@esdolgaev-VirtualBox: ~/work/arch-pc/lab08$ touch lab8-3.asm
esdolgaev@esdolgaev-VirtualBox: ~/work/arch-pc/lab08$ ls
in_out.asm lab8-1 lab8-1.asm lab8-1.o lab8-2.asm lab8-2.o lab8-3.asm
esdolgaev@esdolgaev-VirtualBox: ~/work/arch-pc/lab08$
```

Рис. 4.9: Создание файла

```

/home/esdolgaev/work/arch-pc/lab08/lab8-3.asm [----] 32 L: [ 1+28 29/ 29] *(1428/1428b) <EOF> [*][X]
%include "in_out.asm"
SECTION .data
msg db "Результат: ",0
SECTION .text
global _start
_start:
pop ecx ; Извлекаем из стека в 'ecx' количество
; аргументов (первое значение в стеке)
pop edx ; Извлекаем из стека в 'edx' имя программы
; (второе значение в стеке)
sub ecx,1 ; Уменьшаем 'ecx' на 1 (количество
; аргументов без названия программы)
mov esi, 0 ; Используем 'esi' для хранения
; промежуточных сумм
next:
cmp ecx,0h ; проверяем, есть ли еще аргументы
jz _end ; если аргументов нет выходим из цикла
; (переход на метку '_end')
pop eax ; иначе извлекаем следующий аргумент из стека
call atoi ; преобразуем символ в число
add esi,eax ; добавляем к промежуточной сумме
; след. аргумент 'esi=esi+eax'
loop next ; переход к обработке следующего аргумента
_end:
mov eax, msg ; вывод сообщения "Результат: "
call sprint
mov eax, esi ; записываем сумму в регистр 'eax'
call fprintf ; печать результата
call quit ; завершение программы

```

Рис. 4.10: Текст программы

Создадим исполняемый файл и проверим его работу(рис. 4.11).

```

esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab08$ nasm -f elf lab8-3.asm
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab08$ ld -m elf_i386 -o lab8-3 lab8-3.o
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab08$ ./lab8-3 12 35 46 12
Результат: 105
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab08$ ./lab8-3 34 234 12 43
Результат: 323
esdolgaev@esdolgaev-VirtualBox:~/work/arch-pc/lab08$

```

Рис. 4.11: Создание и работа исполняемого файла

Изменим текст программы для вычисления произведения аргументов командной строки. Создадим исполняемый файл и проверим его работу(рис. 4.12, 4.13).

```

/home/esdolgaev/work/arch-pc/lab08/lab8-3.asm [----] 4 L: [ 1+20 21/ 29] *(968 /1429b) 0032 0x020 [*][X]
%include "in_out.asm"
SECTION .data
msg db "Результат: ",0
SECTION .text
global _start
_start:
pop ecx ; Извлекаем из стека в 'ecx' количество
; аргументов (первое значение в стеке)
pop edx ; Извлекаем из стека в 'edx' имя программы
; (второе значение в стеке)
sub ecx,1 ; Уменьшаем 'ecx' на 1 (количество
; аргументов без названия программы)
mov esi, 1 ; Используем 'esi' для хранения
; промежуточных сумм
next:
cmp ecx,0h ; проверяем, есть ли еще аргументы
jz _end ; если аргументов нет выходим из цикла
; (переход на метку '_end')
pop eax ; иначе извлекаем следующий аргумент из стека
call atoi ; преобразуем символ в число
imul esi,eax ; добавляем к промежуточной сумме
; след. аргумент 'esi=esi+eax'
loop next ; переход к обработке следующего аргумента
_end:
mov eax, msg ; вывод сообщения "Результат: "
call sprint
mov eax, esi ; записываем сумму в регистр 'eax'
call fprintf ; печать результата
call quit ; завершение программы

```

Рис. 4.12: Текст программы


```

esdolgaev@esdolgaev-VirtualBox: /work/arch-pc/lab08$ nasm -f elf lab8-3.asm
esdolgaev@esdolgaev-VirtualBox: /work/arch-pc/lab08$ ld -n elf_i386 -o lab8-3 lab8-3.o
esdolgaev@esdolgaev-VirtualBox: /work/arch-pc/lab08$ ./lab8-3 1 2 3 4
Результат: 24
esdolgaev@esdolgaev-VirtualBox: /work/arch-pc/lab08$ ./lab8-3 5 6
Результат: 30

```

Рис. 4.13: Создание и работа исполняемого файла

4.1 Задание для самостоятельной работы

Введём текст программы в созданный файл, создадим исполняемый файл и проверим его работу(рис. 4.14, 4.15).

```

#include "in_out.asm"
SECTION .data
msgf db "f(x)=7 + 2x",0
msg db "Результат: ",0
SECTION .text
global _start
_start:
mov eax,msgf
call sprintf
pop ecx ; Извлекаем из стека в 'ecx' количество
; аргументов (первое значение в стеке)
pop edx ; Извлекаем из стека в 'edx' имя программы
; (второе значение в стеке)
sub ecx,1 ; Уменьшаем 'ecx' на 1 (количество
; аргументов без названия программы)
mov esi,0 ; Используем 'esi' для хранения
; промежуточных сумм
next:
cmp ecx,0h ; проверяем, есть ли еще аргументы
jz _end ; если аргументов нет выходим из цикла
; (переход на метку _end)
pop eax ; иначе извлекаем следующий аргумент из стека
call atoi ; преобразуем символ в число
imul eax,2
add eax,7
add esi,eax ; добавляем к промежуточной сумме
; след. аргумент 'esi=esi+eax'
loop next ; переход к обработке следующего аргумента
_end:
mov eax, msg ; вывод сообщения "Результат: "

```

Рис. 4.14: Текст программы

```

esdolgaev@esdolgaev-VirtualBox: /work/arch-pc/lab08$ nasm -f elf variant8.asm
esdolgaev@esdolgaev-VirtualBox: /work/arch-pc/lab08$ ld -n elf_i386 -o variant8 variant8.o
esdolgaev@esdolgaev-VirtualBox: /work/arch-pc/lab08$ ./variant8 1 2 3
f(x)=7 + 2x
Результат: 33
esdolgaev@esdolgaev-VirtualBox: /work/arch-pc/lab08$ ./variant8 4 5 6
f(x)=7 + 2x
Результат: 51
esdolgaev@esdolgaev-VirtualBox: /work/arch-pc/lab08$ mc

```

Рис. 4.15: Создание и работа исполняемого файла

5 Выводы

В ходе выполнения лабораторной работы я приобрёл навыки написания программ с использованием циклов и обработкой аргументов командной строки.

Список литературы