



«Министерство науки и высшего образования Российской Федерации  
Федеральное государственное бюджетное образовательное учреждение  
высшего образования  
«Московский государственный технический университет  
имени Н.Э. Баумана  
(национальный исследовательский университет)»  
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ

ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ

КАФЕДРА

КОМПЬЮТЕРНЫЕ СИСТЕМЫ И СЕТИ

## О Т Ч Е Т

по лабораторной работе № 1

Дисциплина: Машинно-зависимые языки и основы компиляции

Название: Изучение среды и отладчика ассемблера

Студент

ИУ6-42Б

(Группа)

21.02.24

(Подпись, дата)

А. П. Плютто

(И. О. Фамилия)

Преподаватель

21.02.24

(Подпись, дата)

(И. О. Фамилия)

Москва 2024 г.

# Содержание

1. Ввод чисел .....	3
1.1. Процедура <code>geti</code> .....	3
1.2. Процедура <code>stoi</code> .....	4
1.2.1. Начало .....	4
1.2.2. Процедура <code>ctoi</code> .....	6
1.2.3. Цикл .....	7
1.2.4. Окончание .....	8
2. Вывод чисел .....	10
2.1. Процедура <code>outi</code> .....	10
2.2. Процедура <code>itos</code> .....	11
2.2.1. Начало .....	11
2.2.2. Цикл .....	12
2.2.3. Окончание .....	13
2.3. Процедура <code>reverse</code> .....	14
3. Выполнение лабораторной работы .....	16
3.1. Задание .....	16
3.2. Цель работы .....	16
3.3. Выполнение .....	16
3.3.1. Работа с данными .....	16
3.3.2. Получение данных .....	17
3.3.3. Вычисление значения функции .....	17
3.3.4. Вывод и завершение .....	19
3.3.5. Компиляция .....	19
3.4. Отладка .....	20

## 1. Ввод чисел

Перед выполнением лабораторной работы, следует подумать об организации ввода-вывода. В предыдущей лабораторной уже описывалось как организовать ввод строк, поэтому задача сводится только к их обработке, но для начала напишем входную процедуру которая будет принимать ввод и отдавать его функции по обработке.

### 1.1. Процедура `geti`

```
geti:  
    push eax  
    push ebx  
    push edx  
    push ecx  
  
    mov ecx, ebx  
    mov edx, eax  
    mov eax, 4      ; Пишем приглашение на ввод  
    mov ebx, 1  
    int 80h  
  
    mov eax, 3      ; Читаем ввод  
    mov ebx, 0  
    pop edx  
    pop ecx  
    push ecx  
    push edx  
    int 80h  
  
    mov eax, 0  
    mov ebx, ecx  
    mov ecx, edx  
    mov edx, ebx  
    call stoi  
    mov dword[ebx], eax  
  
    pop ecx  
    pop edx  
    pop ebx  
    pop eax  
  
ret
```

Листинг 1 — Функция `geti`

Следует немного описать функцию. На вход мы передаем значения в регистрах. Для начала процедура организует вывод приглашения на ввод, из предыдущей лабораторной мы уже знаем как оно организовывается. Ссылка на первую букву приглашения будем просить ввести через регистр `ebx`, количество букв через `eax`. После этого будем читать введенные пользователем данные, поэтому нам нужна так же ссылка на буфер для хранения этих данных, запросим ее в регистре `edx` и длину этого буфера в регистре `ecx`. Так как буквы по определению занимают больше места чем числа в памяти дополнительной памяти для обработки нам не нужно. Поэтому возвращать процедура ничего не будет, а будет только записывать в буфер число введенное пользователем.

После ввода числа пользователь нажимает на enter (LF) и число записывается поцифально в память. Теперь нам необходимо достать каждую цифру, вычесть из нее код числа 0 (`sub eax, '0'`) и сложить эту цифру с предыдущими, умноженными на 10.

Для этого напишем новую процедуру, которая будет брать адрес строки в `edx` и длину этой строки в `ecx` и после преобразования этой строки в число класть это числовое значение в регистр `eax`.

Потом мы просто запишем число из `eax` в буфер, вернем все регистры из стека и выйдем из процедуры `geti`.

## 1.2. Процедура `stoi`

### 1.2.1. Начало

Итак, вот начало процедуры преобразования строки в число:

```

stoi:
    push esi
    push edx
    push ebx
    push ecx

    mov esi, edx
    mov eax, 0
    mov ebx, 10
    mov dx, 0

    mov dl, [esi]
    cmp dl, '-'
    je stoiminus

    push 0
    jmp stoil

```

Листинг 2 — Процедура `stoi` (начало)

В стек загружаем все использованные в процедуре регистры, чтобы их не потерять после окончания выполнения. Готовим регистры для цикла `eax`, как регистр арифметических оперций, будет содержать в себе все число, что мы получили, поэтому зануляем его. Так как мы будем производить умножение, чтобы не потерять адрес, куда в последствии необходимо сохранить число переместим его в регистр адреса `esi`.

Для умножения получившегося числа на 10 используем регистр `ebx` (например пользователь ввел `'10'` мы обработали и поместили в `eax` 1 и после обработки 0 нам необходимо сделать 2 операции, чтобы получить исходное число: умножить `eax` на 10 и прибавить 0).

Регистр `edx` будем так же использовать как временный буфер для текущей цифры. Стоит сказать, что все цифры и знак минус входят в ASCII, поэтому будут занимать лишь 1 байт.

Перед началом цикла прочтем 1 байт в `dl` (младший байт `edx`). Если 1 байт оказался с минусом, то число отрицательно, это необходимо запомнить. Создадим метку, в которой положим в стек число 1, как уведомление о том, что необходимо вернуть дополнительный код числа. После этого переходим к циклу.

```

stoiminus:
    inc esi
    push 1
    jmp stoil

```

### Листинг 3 — Если число отрицательно кладем в стек 1

Если же число положительное положим 0 и перейдем к циклу.

Но перед циклом необходимо написать обработчик одной цифры. Цифра эта будет занимать 1 байт, поэтому процедура будет брать только регистр `dl` и работать дальше с ним.

#### 1.2.2. Процедура `ctoi`

```
ctoi:  
    cmp dl, '0'  
    jl ctoie  
  
    cmp dl, '9'  
    jg ctoie  
  
    sub dl, '0'  
    jmp ctoie
```

### Листинг 4 — Процедура `ctoi`

Итак, мы получили число, которое должно соответствовать коду от 0 до 9 в таблице ASCII, если это так, то вычитаем из этого числа `'0'` и получаем цифру, иначе просто вернем число без изменений.

Можно было бы написать обработчик, который при нахождении подобного числа завершал программу, или писал ошибку в stderr, но я думаю, что и такая простая проверка подойдет для выполнения данных лабораторных. В любом случае можно подобное реализовать, просто поменяв метки перехода в конец процедуры на метки реализации вывода ошибки.

И так, мы получили цифру, теперь необходимо выйти из процедуры:

```
ctoie:  
    ret
```

### Листинг 5 — Процедура `ctoi` (выход)

Вернемся к реализации цикла.

### 1.2.3. Цикл

```
stoil:  
    mov dl, [esi]  
    inc esi  
  
    cmp dl, 0x0a ; =LF  
    je stoie  
  
    call ctoi  
  
    push dx  
    mul ebx  
    mov edx, 0  
    pop dx  
    add eax, edx  
  
    sub ecx, 1  
    cmp ecx, 0  
    je stoie  
  
jmp stoil
```

Листинг 6 — Процедура `stoi` (цикл)

Для начала переместим в `dl` новую цифру. Стоит отметить, что если число положительное, то на первой итерации цикла мы просто второй раз обратимся к тому же адресу памяти, к какому обращались, когда проверяли число на отрицательность, иначе мы обратимся к следующему байту, который идет после минуса.

После этого выполним операцию инкремента (добавления единицы) в регистр с адресом, для того, чтобы в следующей итерации цикла взять новое, еще не обработанное число.

Проверим то число, которое мы только что взяли, не является ли оно завершением строки (LF). В следующих лабораторных работах, когда появится необходимость вводить числа через пробел сюда же добавим и проверку на символ пробела. Если в `dl` символ пробела, то выходим из цикла.

Теперь необходимо вызвать функцию `ctoi` и после ее выполнения в `dl` наконец будет цифра введенного числа. Добавим эту цифру в конец `eax`. Для этого, как описывалось ранее умножим `edx` на 10, не забыв, что при этой операции используется `edx` (поэтому, чтобы не потерять цифру уберем ее в стек). После умножения достаем цифру из стека и складываем с получившимся числом.

При вводе пользователь мог занять весь буфер огромным числом, поэтому символ LF мог и не поместиться в буфер, поэтому добавим в цикл проверку на конец буфера.

После этого снова переходим к метке цикла и начинается новая итерация.

#### 1.2.4. Окончание

Когда цикл завершит свою работу мы переходим к метке `stoie`.

```
stoie:  
    mov ecx, 0  
    pop ecx  
    cmp cl, 1  
    je stoiaddm  
    jmp stoiend
```

Листинг 7 — Процедура `stoi` (конец 1)

Тут нам длина буфера уже не важна, ведь он уже прочитан. Поэтому заносим в регистр `ecx` флаг минуса, который лежит наверху стека. Теперь когда у нас есть модуль числа можно и запросить его дополнительный код, если это необходимо. Прверяем `ecx` и если там лежит 1 переходим к метке для умножения `eax` на  $-1$ , иначе в конец.

```
stoiaddm:  
    mov ecx, -1  
    mul ecx  
    jmp stoiend
```

Листинг 8 — Преобразуем число в отрицательное

Регистр `edx` уже не нужен, флаг тоже, поэтому просто перемещаем  $-1$  в `ecx` и умножаем `eax` на `ecx`. После чего перемещаемся в конец.

Стоит отметить, что подобная операция эквивалентна `not eax`, `add eax, 1`.

Возвращаем все регистры, как они были, кроме `eax`, в котором ответ и выводим из процедуры `stoi`.

```
stouiend:  
    pop ecx  
    pop ebx  
    pop edx  
    pop esi  
  
    ret
```

### Листинг 9 — Процедура `stoi` (конец 2)

Как уже было сказано ранее, после выполнения этой процедуры продолжается выполнение `geti`, которая положит значение `eax` в буфер пользователя и, вернув все регистры в их начальное состояние завершится.

## 2. Вывод чисел

После выполнения арифметических операций программа должна вывести пользователю результат выполнения перед ее завершением. Для этого напишем еще несколько процедур, которые теперь будут обратно преобразовывать число в строку. Тут мы для удобства будем пользоваться все той же памятью в которой лежит число и просто впоследствии сохраним туда строку. Но для использования иного буфера необходимо поменять лишь несколько строк.

### 2.1. Процедура `outi`

```
outi:  
    push ebx  
    push edx  
    push ecx  
    push eax  
  
    mov eax, 4      ; Пишем приглашение на вывод  
    mov ebx, 1  
    int 80h  
  
    pop ebx  
    push ebx  
    call itos  
  
    mov edx, ecx  
    mov ecx, ebx  
    mov eax, 4      ; Вывод  
    mov ebx, 1  
    int 80h  
  
    pop eax  
    pop ecx  
    pop edx  
    pop ebx  
  
    ret
```

Листинг 10 — Процедура `outi`

Вот первая процедура, она все так же выдает строку, которую мы передаем в `ecx`. Длина этой строки лежит в `edx`. В `eax` лежит ссылка на буфер. Длину буфера предполагаем достаточной, что бы вывести все число целиком.

Процедура начинает свое выполнение опять с сохранения всех регистров в стек. Затем мы вызываем прерывание на вывод предложения перед числом, после этого готовим регистры и выполняем процедуру `itos`, которая преоб-

разует данное нам число в строку и сама сохраняет эту строку в память, где было число. И после выполнения процедуры преобразования мы снова организуем вывод только для числа. Затем достаем из стека значения регистров и завершаем процедуру.

## 2.2. Процедура `itos`

Теперь давайте подробнее разберем процедуру `itos`.

### 2.2.1. Начало

```
itos:  
    push esi  
    push eax  
    push ebx  
    push ecx  
    push edx  
  
    mov esi, ebx  
    mov eax, [esi]  
  
    mov ebx, 10  
  
    cmp eax, 0  
    jl itosminus  
  
    jmp itosl
```

Листинг 11 — Процедура `itos` (Начало)

Опять убираем все используемые регистры в стек. Перемещаем полученный адрес числа в `esi`. После чего получаем само число по этому адресу. Его записываем в регистр `eax`. В регистр `ebx` записываем основание системы счисления, так же как мы делали на вводе, но тут использоваться оно будет для деления. Если само число меньше нуля то необходимо перед циклом сразу убрать в память `' - '`. Иначе просто переходим к циклу.

```
itosminus:  
    mov byte[esi], '-'  
  
    inc esi  
  
    mov ecx, -1  
    mul ecx  
  
    jmp itosl
```

Листинг 12 — Добавляем минус в буфер

Добавляем минус в буфер, прибавляем единицу в адрес, что бы этот минус не затерся последней цифрой числа (ведь при делении на 10 мы первой получим именно последнюю цифру). После этого берем числомо число по модулю, т.е. преобразуем его дополнительный код в нормальный. Потом так же переходим в цикл.

## 2.2.2. Цикл

```
itosl:  
    mov edx, 0  
    div ebx  
    add edx, '0'  
    mov byte[esi], dl  
  
    inc esi  
  
    cmp eax, 0  
    je itose  
  
    jmp itosl
```

Листинг 13 — Процедура `itos` (Цикл)

Тут мы будем использовать беззнаковое деление (так как со знаком мы уже разобрались, само число по модулю). При выполнении операции `div` результат деления попадает `eax`, что позволяет нам не делать никаких других логических преобразований, регистр и так с каждой итерацией цикла будет уменьшаться на 10, т.е. на одну цифру. Эта цифра будет остатком от деления на 10 и будет храниться в `edx`, прибавляя к этому регистру `'0'` получим ASCII код этой цифры, которую и сохраним в памяти. Мы знаем что этот код занимает лишь байт, поэтому он весь поместился в `dl`.

После записи в текущий байт переместимся на следующий, иначе число затрется.

Когда будет последняя итерация в `eax` останется только одна цифра, после деления на 10 она перенесется в остаток, а сам регистр останется 0, поэтому когда это произойдет мы выйдем из цикла.

И перепрыгиваем снова на метку цикла, чтобы начать новую итерацию.

### 2.2.3. Окончание

```
itose:  
    pop edx  
    pop ecx  
    pop ebx  
  
    mov eax, ebx  
  
    mov ecx, esi  
    sub ecx, ebx  
  
    call reverse  
  
    pop eax  
    pop esi  
  
    ret
```

Листинг 14 — Процедура `itos` (Конец)

В конце возвращаем значения регистров на свои места. Берем начальный адрес и вычитаем его из конечного, таким образом, получая длину всей строки. Этую длину запишем в `ecx`.

А теперь немного подробнее посмотрим на проблему которую я упоминал ранее. При делении числа на 10 мы получаем последнюю его цифру, таким образом число 1234 будет записано нами в памяти как `'4321'` ( $-1234 \rightarrow '-4321'$ ).

Для решения этой проблемы напишем еще одну процедуру, которую назовем `reverse`. Вызовем ее, вернем значения в оставшиеся регистры и завершим процедуру `itos`.

## 2.3. Процедура `reverse`

```
reverse:  
    push eax  
    push ebx  
    push ecx  
    push edx  
  
    mov ebx, eax  
    add eax, ecx  
    sub eax, 1  
    mov edx, eax  
  
    mov al, [ebx]  
    cmp al, '-'  
    je reverseminus  
  
    jmp reversel
```

Листинг 15 — Процедура `reverse` (Начало)

В `eax` передадим адрес числа, а в `ecx` количество цифр в числе. Суть алгоритма заключается в том, что `ebx` указывает на начало, а `edx` на конец числа, и при каждой итерации они меняют значения друг друга на противоположные, таким образом за половину числа мы поменяем ровно все значения. (`ebx` и `edx` указывают на начало и конец, как палочка в буквах).

Оба регистра это адреса, но указывают они на байты, поэтому для сохранения самих значений будем использовать только `al`. Первым делом берем первый символ и проверяем не минус ли он.

Напишем обработчик для этого случая:

```
reverseminus:  
    inc ebx  
    jmp reversel
```

Листинг 16 — Обрабатываем минус

Просто смещаем регистр указывающий на начало на 1 и входим в цикл.

```
reversel:  
    cmp ebx, edx  
    jge reversee  
  
    mov al, [ebx]  
    mov cl, [edx]  
    mov byte[ebx], cl  
    mov byte[edx], al  
  
    inc ebx  
    sub edx, 1  
  
    jmp reversel
```

Листинг 17 — Процедура `reverse` (Цикл)

В цикле мы проверяем, чтобы `ebx` был строго меньше `edx` (если больше – число четное, если равен – число нечетное). Потом перемещаем текущие значения в свободные регистры и, меняя их местами, сразу записываем обратно. Дальше мы увеличиваем `ebx` и уменьшаем `edx`, «двигая» их друг к другу.

```
reversee:  
    pop edx  
    pop ecx  
    pop ebx  
    pop eax  
    ret
```

Листинг 18 — Процедура `reverse` (Конец)

В конце возвращаем все регистры на место и выходим из процедуры.

### 3. Выполнение лабораторной работы

#### 3.1. Задание

Вычислить целочисленное выражение:

$$f = (a - c)^2 + 2 \times a \times \frac{c^3}{k^2 + 1}$$

#### 3.2. Цель работы

#### 3.3. Выполнение

##### 3.3.1. Работа с данными

Проинициализируем все сообщения пользователю. Зарезервируем 12 байт для каждого из переменных. 12 байт потому что минимальное число типа целочисленное –2147483648 (оно же максимальное по количеству знаков, их 11), добавляя еще знак переноса получаем максимальное значение в 12 байт.

```
section .data
    ia db  "Enter a: "
    ic db  "Enter c: "
    ik db  "Enter k: "
    ot db  "Result : "
    inl equ $-ot

section .bss
    a resb 12
    c resb 12
    k resb 12
    l equ $-k
```

Листинг 19 — Выделяем и инициализируем необходимые данные

Я сразу разбил выполнение программы на блоки, первый из которых получение данных.

### 3.3.2. Получение данных

```
getack:  
    mov  eax, inl  
    mov  ecx, l  
    mov  ebx, ia  
    mov  edx, a  
    call geti  
  
    mov  eax, inl  
    mov  ecx, l  
    mov  ebx, ic  
    mov  edx, c  
    call geti  
  
    mov  eax, inl  
    mov  ecx, l  
    mov  ebx, ik  
    mov  edx, k  
    call geti  
  
    mov  eax, [a]  
    mov  ebx, [c]  
    mov  ecx, [k]  
  
    jmp calc
```

Листинг 20 — Запрашиваем данные на ввод

Для этого просто перемещаем приглашение пользователю, длину приглашения, ссылку на буфер для данных и длину буфера в нужные регистры и вызываем `geti`. Проделываем эту процедуру 3 раза и получаем в памяти 3 числа, готовых к обработке.

Все выполнения арифметических операций будет в `calc` переходим в нее, когда значения успешно введены.

### 3.3.3. Вычисление значения функции

Перепишем задание и разобьем его на шаги.

$$f = (a - c)^2 + 2 \times a \times \frac{c^3}{k^2 + 1}$$

1.  $a - c$
2.  $(a - c)^2$
3.  $2 \times a$
4.  $c^3$

5.  $k^2$
6.  $k^2 + 1$
7.  $2 \times a \times c^3$
8.  $2 \times a \times c^3 / (k^2 + 1)$
9.  $(a - c)^2 + 2 \times a \times c^3 / (k^2 + 1)$

```
calc:
    push eax
    sub eax, ebx
    mul eax
    mov edx, eax
    pop eax
    push edx

    mov edx, 2
    mul edx
    push eax

    mov eax, ebx
    mul ebx
    mul ebx
    push eax

    mov eax, ecx
    mul ecx
    add eax, 1
    mov ecx, eax

    pop eax
    pop ebx
    mul ebx
    cdq
    div ecx

    pop ecx
    add eax, ecx

    jmp outandex
```

Листинг 21 — Вычисляем значение функции

В начале сохраним в стеке значение `a`, оно нам понадобится для шага 3. Выполним шаг 1 и запишем значение в `eax`. После чего умножим этот регистр сам на себя. Итак, в `eax` лежит значение шага 2, оно нам не нужно до 10 шага, поэтому уберем его в стек, предварительно достав от туда `a`.

Умножим `eax` на 2 и поместим значение в стек, оно нам не понадобится до 7 шага. Переместим значение `c` в `eax` и умножим `eax` на `c` два раза. Куб `c` нам так же не понадобится до 7 шага, тоже уберем в стек. Умножим `k` саму на себя и прибавим к ней единицу.

Таким образом мы сделали все простейшие операции, осталось только вынуть из стека все значения, которые у нас получились.  $c^3 \rightarrow eax$ ;  $2 \times a \rightarrow ebx$ . Умножаем `eax` на `ebx` и получаем ответ на 7 шаг.

Для 8 шага необходимо подготовить регистр `edx`. При делении используется расширенная версия регистра `eax`: `edx:eax`, поэтому если в `edx` у нас нет расширения для `eax`, то необходимо заполнить `edx` нулями, если число положительно и единицами, если число отрицательно. Для этого перед делением вызовем `cdq`.

После этого просто делим со знаком весь результат 7 шага на  $k^2 + 1$ . В стеке осталось лежать только  $(a - c)^2$ , достаем и это значение, суммируем с результатом деления и получаем целочисленный ответ. Осталось только вывести ответ и завершить программу.

### 3.3.4. Вывод и завершение

```
outandex:  
    mov dword[a], eax  
  
    mov eax, a  
    mov ecx, ot  
    mov edx, inl  
    call outi  
  
    mov eax, 1  
    int 80h
```

Листинг 22 — Вывод и выход

Перемещаем получившееся значение в память, а ссылку указываем в регистр. После этого указываем сообщение перед выводом и вызываем вывод.

После того, как вывод отработает остается только вызвать завершение программы.

### 3.3.5. Компиляция

Ввод, вывод и код лабораторной я занес в отдельные файлы, которые и назвал `input.asm`, `output.asm` и `lab2.asm`.

Поэтому компилировать их будем тоже отдельно, а после этого соберем.

Вот, что необходимо ввести в терминал, для компиляции и сборки всех 3 файлов:

```
mod=lab2/lab2 # Название ассемблерного файла без расширения
nasm -f elf -o $mod.o $mod.asm
nasm -f elf -o lab2/input.o lab2/input.asm
nasm -f elf -o lab2/output.o lab2/output.asm
ld -m elf_i386 -o $mod $mod.o lab2/input.o lab2/output.o
```

Листинг 23 — Команда в терминале

### 3.4. Отладка

Отладим все выполнение, заодно посмотрим как работают операции ввода-вывода.

Первой процедурой посмотрим выполнение `stoi`, так как простейший ввод и вывод был показан в предыдущей лабораторной работе.

Введем при запросе `a` большое отрицательное число: `-123477790`.

```
(gdb) br _start
Breakpoint 1 at 0x8049000
(gdb) r
Starting program: /home/debianuser/labs/lab2/lab2
Breakpoint 1, 0x08049000 in _start ()
(gdb) s
Single stepping until exit from function _start,
which has no line number information.
0x080490b0 in geti ()
(gdb)
Single stepping until exit from function geti,
which has no line number information.
Enter a: -123477790
0x080490fb in stoi ()
(gdb)
```

Рисунок 1 — Запуск

Register group: general		
eax	0x0	0
ecx	0xc	12
edx	0x804a024	134520868
ebx	0x804a024	134520868
esp	0xffffd378	0xffffd378
ebp	0x0	0x0
esi	0x0	0
edi	0x0	0
eip	0x80490ff	0x80490ff <stoi+4>

```
0x80490fe <stoi>    push    esi
0x80490fc <stoi+1>  push    edx
0x80490fd <stoi+2>  push    ebx
0x80490fe <stoi+3>  push    ecx
>0x80490ff <stoi+4> mov     esi,edx
0x8049100 <stoi+5>  mov     eax,0x0
0x8049106 <stoi+11>  mov     eax,0xa
0x804910b <stoi+16>  mov     dx,0x0
0x804910f <stoi+20>  mov     dl,BYTE PTR [esi]
0x8049111 <stoi+22>  cmp    dl,0x2d
native process 1872375 In: stoi
(gdb) lay next
(gdb) si
0x080490fe in stoi ()
0x080490fd in stoi ()
0x080490fc in stoi ()
0x080490fe in stoi ()
0x0804910b in stoi ()
0x0804910f in stoi ()
0x08049111 in stoi ()
0x08049111 in stoi ()
0x08049111 in stoi ()
(gdb)
```

Рисунок 2 — Сохраняем регистры

Register group: general		
eax	0x0	0
ecx	0xc	12
edx	0x804002d	134479917
ebx	0xa	10
esp	0xffffd378	0xffffd378
ebp	0x0	0x0
esi	0x804a024	134520868
edi	0x0	0
eip	0x8049111	0x8049111 <stoi+22>

```
0x80490fe <stoi+3>  push    ecx
0x80490ff <stoi+4>  mov     esi,edx
0x8049101 <stoi+6>  mov     eax,0x0
0x8049106 <stoi+11>  mov     ebx,0xa
0x804910b <stoi+16>  mov     dx,0x0
0x804910f <stoi+20>  mov     dl,BYTE PTR [esi]
0x8049111 <stoi+22>  cmp    dl,0x2d
0x8049114 <stoi+25>  je     0x804911a <stoiminus>
0x8049116 <stoi+27>  push    0x0
0x8049118 <stoi+29>  jmp    0x804911f <stoi>
native process 1872375 In: stoi
(gdb) lay next
(gdb) si
0x080490fc in stoi ()
0x080490fd in stoi ()
0x080490fe in stoi ()
0x080490fe in stoi ()
0x0804910b in stoi ()
0x0804910f in stoi ()
0x08049111 in stoi ()
0x08049111 in stoi ()
0x08049111 in stoi ()
(gdb)
```

Register group: general		
eflags	0x246	[ PF ZF IF ]
eax	0x0	0
ecx	0xc	12
edx	0x804002d	134479917
ebx	0xa	10
esp	0xffffd378	0xffffd378
ebp	0x0	0x0
esi	0x804a024	134520868
edi	0x0	0
eip	0x804911a	0x804911a <stoiminus>

```
>0x804911a <stoiminus> inc    esi
0x804911b <stoiminus+1> push   0x1
0x804911d <stoiminus+3> jmp    0x804911f <stoi>
0x804911f <stoi>      mov    dl,BYTE PTR [esi]
0x804911f <stoi+2>  inc    esi
0x804911f <stoi+3>  cmp    dl,0xa
0x8049125 <stoi+6>  je     0x8049145 <stoi>
0x8049127 <stoi+8>  call   0x80490eb <ctoi>
0x804912c <stoi+13> push   dx
0x804912e <stoi+15> mul    ebx
native process 1872375 In: stoiminus
(gdb) lay next
(gdb) si
0x080490fc in stoi ()
0x080490fd in stoi ()
0x080490fe in stoi ()
0x080490fe in stoi ()
0x0804910b in stoi ()
0x0804910f in stoi ()
0x08049111 in stoi ()
0x08049111 in stoi ()
0x08049111 in stoi ()
(gdb)
```

Рисунок 3 — Переносим 1-ый символ

```

eflags 0x202 [ IF ]
eax 0x0 0
ecx 0xc 12
edx 0x804002d 134479917
ebx 0xa 10
esp 0xfffffd374 0xfffffd374
ebp 0x0 0x0
esi 0x804a025 134520869
edi 0x0 0
eip 0x804911f 0x804911f <stoil>

0x804911a <stoinminus> inc esi
0x804911b <stoinminus+1> push 0x1
0x804911d <stoinminus+3> jmp 0x804911f <stoil>
>0x804911f <stoil> mov dl,BYTE PTR [esi]
0x8049121 <stoil+2> inc esi
0x8049122 <stoil+3> cmp dl,0xa
0x8049125 <stoil+6> je 0x8049145 <stoie>
0x8049127 <stoil+8> call 0x80490eb <ctoi>
0x804912c <stoil+13> push dx
0x804912e <stoil+15> mul ebx

```

native process 1872375 In: stoil L?? PC: 0x804911f  
0x80490ff in stoi ()  
0x8049101 in stoi ()  
0x8049106 in stoi ()  
0x804910b in stoi ()  
0x804910f in stoi ()  
0x8049111 in stoi ()  
0x8049114 in stoi ()  
0x804911a in stoinminus ()  
0x804911b in stoinminus ()  
0x804911c in stoinminus ()  
0x804911f in stoil ()  
(gdb) ■

Рисунок 5 — Заходим в цикл

```

eflags 0x202 [ IF ]
eax 0x0 0
ecx 0xc 12
edx 0x8040001 134479873
ebx 0xa 10
esp 0xfffffd370 0xfffffd370
ebp 0x0 0x0
esi 0x804a026 134520870
edi 0x0 0
eip 0x80490f8 0x80490f8 <ctoi+13>

0x80490f0 <ctoi> cmp dl,0x30
0x80490f4 <ctoi+3> jl 0x80490fa <ctoie>
0x80490f6 <ctoi+5> cmp dl,0x39
0x80490f9 <ctoi+8> jg 0x80490fa <ctoie>
0x80490f5 <ctoi+10> sub dl,0x30
0x80490f8 <ctoi+13> jmp 0x80490fa <ctoie>
0x80490f0 <ctoie> ret
0x80490f2 <stoil> push esi
0x80490f4 <stoil+1> push edx
0x80490f6 <stoil+2> push ebx

```

native process 1872375 In: ctoi L?? PC: 0x80490f8  
0x804911f in stoil ()  
0x8049121 in stoil ()  
0x8049122 in stoil ()  
0x8049125 in stoil ()  
0x8049127 in stoil ()  
0x80490eb in ctoi ()  
0x80490f0 in ctoi ()  
0x80490f3 in ctoi ()  
0x80490f5 in ctoi ()  
0x80490f8 in ctoi ()  
(gdb) ■

Рисунок 7 — Выходим из `ctoi`

```

eflags 0x202 [ IF ]
eax 0x1 1
ecx 0xc 12
edx 0x1 1
ebx 0xa 10
esp 0xfffffd374 0xfffffd374
ebp 0x0 0x0
esi 0x804a026 134520870
edi 0x0 0
eip 0x8049139 0x8049139 <stoil+26>

0x8049125 <stoil+6> je 0x8049145 <stoie>
0x8049127 <stoil+8> call 0x80490eb <ctoi>
0x804912c <stoil+13> push dx
0x804912e <stoil+15> mul ebx
0x8049130 <stoil+17> mov edx,0x0
0x8049135 <stoil+22> pop dx
0x8049137 <stoil+24> add eax,edx
>0x8049139 <stoil+26> jmp 0x804911f <stoil>
0x804913b <stoil+28> sub cx,0x1
0x804913f <stoil+32> cmp cx,0x0

```

native process 1872375 In: stoil L?? PC: 0x8049139  
0x80490f0 in ctoi ()  
0x80490f3 in ctoi ()  
0x80490f5 in ctoi ()  
0x80490f8 in ctoi ()  
0x80490fa in ctoie ()  
0x804912c in stoil ()  
0x804912e in stoil ()  
0x8049130 in stoil ()  
0x8049135 in stoil ()  
0x8049137 in stoil ()  
0x8049139 in stoil ()  
(gdb) ■

Рисунок 9 — Добавляем к `eax`

Рисунок 4 — Обрататываем минус

```

eflags 0x216 [ PF AF IF ]
eax 0x0 0
ecx 0xc 12
edx 0x8040031 134479921
ebx 0xa 10
esp 0xfffffd370 0xfffffd370
ebp 0x0 0x0
esi 0x804a026 134520870
edi 0x0 0
eip 0x80490eb 0x80490eb <ctoi>

0x80490eb <ctoi> cmp dl,0x30
0x80490f0 <ctoi+3> jl 0x80490fa <ctoie>
0x80490f2 <ctoi+5> cmp dl,0x39
0x80490f4 <ctoi+8> jg 0x80490fa <ctoie>
0x80490f6 <ctoi+10> sub dl,0x30
0x80490f8 <ctoi+13> jmp 0x80490fa <ctoie>
0x80490f0 <ctoie> ret
0x80490f2 <stoil> push esi
0x80490f4 <stoil+1> push edx
0x80490f6 <stoil+2> push ebx

```

native process 1872375 In: ctoi L?? PC: 0x80490eb  
0x8049111 in stoil ()  
0x8049114 in stoil ()  
0x8049118 in stoinminus ()  
0x804911b in stoinminus ()  
0x804911d in stoinminus ()  
0x804911f in stoil ()  
0x8049123 in stoil ()  
0x8049122 in stoil ()  
0x8049129 in stoil ()  
0x8049127 in stoil ()  
0x80490eb in ctoi ()  
(gdb) ■

Рисунок 6 — Заходим в `ctoi`

```

eflags 0x202 [ IF ]
eax 0x0 0
ecx 0xc 12
edx 0x8040001 134479873
ebx 0xa 10
esp 0xfffffd370 0xfffffd374
ebp 0x0 0x0
esi 0x804a026 134520870
edi 0x0 0
eip 0x804912c 0x804912c <stoil+13>

0x804911f <stoil> mov dl,BYTE PTR [esi]
0x8049121 <stoil+2> inc esi
0x8049123 <stoil+3> cmp dl,0xa
0x8049125 <stoil+6> je 0x8049115 <stoie>
0x8049127 <stoil+8> call 0x80490eb <ctoi>
>0x804912c <ctoi+13> push dx
0x804912e <ctoi+15> mul ebx
0x8049130 <ctoi+17> mov edx,0x0
0x8049135 <ctoil+22> pop dx
0x8049137 <ctoil+24> add eax,edx

```

native process 1872375 In: stoil L?? PC: 0x804912c  
0x8049122 in stoil ()  
0x8049125 in stoil ()  
0x8049127 in stoil ()  
0x80490eb in ctoi ()  
0x80490f0 in ctoi ()  
0x80490f3 in ctoi ()  
0x80490f5 in ctoi ()  
0x80490f8 in ctoi ()  
(gdb) ■

Рисунок 8 — Готовим цифру

```

eflags 0x212 [ AF IF ]
eax 0xc 12
ecx 0xc 12
edx 0x23 51
ebx 0xa 10
esp 0xfffffd374 0xfffffd374
ebp 0x0 0x0
esi 0x804a028 134520872
edi 0x0 0
eip 0x8049127 0x8049127 <stoil+8>

0x804911f <stoil> mov dl,BYTE PTR [esi]
0x8049121 <stoil+2> inc esi
0x8049123 <stoil+3> cmp dl,0xa
0x8049125 <stoil+6> je 0x8049145 <stoie>
>0x8049127 <stoil+8> call 0x80490eb <ctoi>
0x804912e <ctoi+13> push dx
0x804912c <ctoi+15> mul ebx
0x8049130 <ctoil+17> mov edx,0x0
0x8049135 <ctoil+22> pop dx
0x8049137 <ctoil+24> add eax,edx

```

native process 1872375 In: stoil L?? PC: 0x8049127  
0x8049122 in stoil ()  
0x8049125 in stoil ()  
0x8049127 in stoil ()  
0x8049130 in stoil ()  
0x8049135 in stoil ()  
0x8049137 in stoil ()  
0x8049139 in stoil ()  
0x804911f in stoil ()  
0x8049121 in stoil ()  
0x8049122 in stoil ()  
0x8049125 in stoil ()  
0x8049127 in stoil ()  
(gdb) ■

Рисунок 10 — Добавляем 2-ую цифру

```

eflags 0x206 [ PF IF ]
eax 0xb 123
ecx 0xc 12
edx 0x3 3
ebx 0xa 10
esp 0xfffffd374 0xfffffd374
ebp 0x0 0x0
esi 0x804a028 134520872
edi 0x0 0
eip 0x8049139 0x8049139 <stoil+26>

0x8049125 <stoil+6> je 0x8049145 <stole>
0x8049127 <stoil+8> call 0x80490eb <ctoi>
0x804912c <stoil+13> push dx
0x804912e <stoil+15> mul ebx
0x8049130 <stoil+17> mov edx,0x0
0x8049135 <stoil+22> pop dx
0x8049137 <stoil+24> add eax,edx
0x8049139 <stoil+26> jmp 0x804911f <stoil>
0x804913b <stoil+28> sub cx,0x1
0x804913f <stoil+32> cmp cx,0x0

native process 1872375 In: stoil
L?? PC: 0x8049139
0x80490f0 in ctoi()
0x80490f3 in ctoi()
0x80490f5 in ctoi()
0x80490f8 in ctoi()
0x80490fa in ctoie()
0x804912c in stoil()
0x804912e in stoil()
0x8049130 in stoil()
0x8049135 in stoil()
0x8049137 in stoil()
0x8049139 in stoil()

(gdb)

```

Рисунок 11 — Добавляем 3-ую цифру

```

eflags 0x216 [ PF AF IF ]
eax 0x4d2 1234
ecx 0xc 12
edx 0x4 4
ebx 0xa 10
esp 0xfffffd374 0xfffffd374
ebp 0x0 0x0
esi 0x804a029 134520873
edi 0x0 0
eip 0x8049139 0x8049139 <stoil+26>

0x8049125 <stoil+6> je 0x8049145 <stole>
0x8049127 <stoil+8> call 0x80490eb <ctoi>
0x804912c <stoil+13> push dx
0x804912e <stoil+15> mul ebx
0x8049130 <stoil+17> mov edx,0x0
0x8049135 <stoil+22> pop dx
0x8049137 <stoil+24> add eax,edx
0x8049139 <stoil+26> jmp 0x804911f <stoil>
0x804913b <stoil+28> sub cx,0x1
0x804913f <stoil+32> cmp cx,0x0

native process 1872375 In: stoil
L?? PC: 0x8049139
0x80490f0 in ctoi()
0x80490f3 in ctoi()
0x80490f5 in ctoi()
0x80490f8 in ctoi()
0x80490fa in ctoie()
0x804912c in stoil()
0x804912e in stoil()
0x8049130 in stoil()
0x8049135 in stoil()
0x8049137 in stoil()
0x8049139 in stoil()

(gdb)

```

Рисунок 12 — Добавляем 4-ую цифру

```

eflags 0x202 [ IF ]
eax 0x303b 12347
ecx 0xc 12
edx 0x7 7
ebx 0xa 10
esp 0xfffffd370 0xfffffd370
ebp 0x0 0x0
esi 0x804a02b 134520875
edi 0x0 0
eip 0x80490fa 0x80490fa <ctoie>

0x80490ee <ctoie> cmp dl,0x30
0x80490ee <ctoie+3> jl 0x80490fa <ctoie>
0x80490f0 <ctoie+5> cmp dl,0x39
0x80490f3 <ctoie+8> jg 0x80490fa <ctoie>
0x80490f5 <ctoie+10> sub dl,0x30
0x80490f8 <ctoie+13> jmp 0x80490fa <ctoie>
0x80490fc <ctoie+16> ret
0x80490fb <stoil> push esi
0x80490fc <stoil+1> push edx
0x80490fd <stoil+2> push ebx

native process 1872375 In: ctoie
L?? PC: 0x80490fa
0x8049121 in stoil()
0x8049123 in stoil()
0x8049125 in stoil()
0x8049127 in stoil()
0x80490e8 in ctoi()
0x80490f0 in ctoi()
0x80490f3 in ctoi()
0x80490f5 in ctoi()
0x80490f8 in ctoi()
0x80490fa in ctoie()

(gdb)

```

Рисунок 13 — Добавляем 5-ую цифру

```

eflags 0x216 [ PF AF IF ]
eax 0x1e255 123477
ecx 0xc 12
edx 0x7 7
ebx 0xa 10
esp 0xfffffd374 0xfffffd374
ebp 0x0 0x0
esi 0x804a02b 134520875
edi 0x0 0
eip 0x8049139 0x8049139 <stoil+26>

0x8049125 <stoil+6> je 0x8049145 <stole>
0x8049127 <stoil+8> call 0x80490eb <ctoi>
0x804912c <stoil+13> push dx
0x804912e <stoil+15> mul ebx
0x8049130 <stoil+17> mov edx,0x0
0x8049135 <stoil+22> pop dx
0x8049137 <stoil+24> add eax,edx
0x8049139 <stoil+26> jmp 0x804911f <stoil>
0x804913b <stoil+28> sub cx,0x1
0x804913f <stoil+32> cmp cx,0x0

native process 1872375 In: stoil
L?? PC: 0x8049139
0x80490f0 in ctoi()
0x80490f3 in ctoi()
0x80490f5 in ctoi()
0x80490f8 in ctoi()
0x80490fa in ctoie()
0x804912c in stoil()
0x804912e in stoil()
0x8049130 in stoil()
0x8049135 in stoil()
0x8049137 in stoil()
0x8049139 in stoil()

(gdb)

```

Рисунок 14 — Добавляем 6-ую цифру

```

eflags 0x206 [ PF IF ]
eax 0x12d759 1234777
ecx 0xc 12
edx 0x7 7
ebx 0xa 10
esp 0xfffffd374 0xfffffd374
ebp 0x0 0x0
esi 0x804a02c 134520876
edi 0x0 0
eip 0x804911f 0x804911f <stoil>

0x804911f <stoil> mov dl,BYTE PTR [esi]
0x8049121 <stoil+2> inc esi
0x8049122 <stoil+3> cmp dl,0xa
0x8049123 <stoil+6> jle 0x8049145 <stole>
0x8049125 <stoil+8> call 0x80490eb <ctoi>
0x8049127 <stoil+13> push dx
0x804912e <stoil+15> mul ebx
0x8049130 <stoil+17> mov edx,0x0
0x8049135 <stoil+22> pop dx
0x8049137 <stoil+24> add eax,edx
0x8049139 <stoil+26> jmp 0x804911f <stoil>

native process 1872375 In: stoil
L?? PC: 0x804911f
0x80490f3 in ctoi()
0x80490f5 in ctoi()
0x80490f8 in ctoi()
0x80490fa in ctoie()
0x804912c in stoil()
0x804912e in stoil()
0x8049130 in stoil()
0x8049135 in stoil()
0x8049137 in stoil()
0x8049139 in stoil()

(gdb)

```

Рисунок 15 — Добавляем 7-ую цифру

```

eflags 0x212 [ AF IF ]
eax 0xbc6983 12347779
ecx 0xc 12
edx 0x9 9
ebx 0xa 10
esp 0xfffffd374 0xfffffd374
ebp 0x0 0x0
esi 0x804a02d 134520877
edi 0x0 0
eip 0x8049139 0x8049139 <stoil+26>

0x8049125 <stoil+6> je 0x8049145 <stole>
0x8049127 <stoil+8> call 0x80490eb <ctoi>
0x804912c <stoil+13> push dx
0x804912e <stoil+15> mul ebx
0x8049130 <stoil+17> mov edx,0x0
0x8049135 <stoil+22> pop dx
0x8049137 <stoil+24> add eax,edx
0x8049139 <stoil+26> jmp 0x804911f <stoil>
0x804913b <stoil+28> sub cx,0x1
0x804913f <stoil+32> cmp cx,0x0

native process 1872375 In: stoil
L?? PC: 0x8049139
0x80490f0 in ctoi()
0x80490f3 in ctoi()
0x80490f5 in ctoi()
0x80490f8 in ctoi()
0x80490fa in ctoie()
0x804912c in stoil()
0x804912e in stoil()
0x8049130 in stoil()
0x8049135 in stoil()
0x8049137 in stoil()
0x8049139 in stoil()

(gdb)

```

Рисунок 16 — Добавляем 8-ую цифру

```

eflags      0x206      [ PF IF ]
eax         0x75c1fie   123477790
ecx         0xc        12
edx         0x0        0
ebx         0xa        10
esp         0xfffffd374  0xfffffd374
ebp         0x0        0x0
esi         0x804a02e   134520878
edi         0x0        0
eip         0x804911f   0x804911f <stoi>

0x804911f <stoi>    mov dl,BYTE PTR [esi]
0x8049121 <stoi+2>  inc esi
0x8049122 <stoi+3>  cmp dl,0xa
0x8049125 <stoi+6>  je 0x8049165 <stoi>
0x8049127 <stoi+8>  call 0x80490eb <ctoi>
0x804912c <stoi+13> push dx
0x804912e <stoi+15> mul ebx
0x8049130 <stoi+17> mov edx,0x0
0x8049135 <stoi+22> pop dx
0x8049137 <stoi+24> add eax,edx

native process 1872375 In: stoi
0x080490f3 in ctoi ()
0x080490f5 in ctoi ()
0x080490f8 in ctoi ()
0x080490f9 in ctoi ()
0x0804912c in stoi ()
0x0804912d in stoi ()
0x08049130 in stoi ()
0x08049135 in stoi ()
0x08049137 in stoi ()
0x08049139 in stoi ()
0x0804913f in stoi ()

(gdb) 

```

Рисунок 17 — Добавляем 9-ую цифру

```

eflags      0xa87      [ CF PF SF IF OF ]
eax         0xf8a3e0e2  -123477790
ecx         0xfffffff
edx         0x75c1f1d   123477789
ebx         0xa        10
esp         0xfffffd378  0xfffffd378
ebp         0x0        0x0
esi         0x804a02f   134520879
edi         0x0        0
eip         0x8049159   0x8049159 <stoiend>

0x8049145 <stoi>    mov ecx,0x0
0x804914a <stoi+5>  pop ecx
0x804914b <stoi+6>  cmp cl,0x1
0x804914d <stoi+9>  je 0x8049152 <stoiaddm>
0x8049150 <stoi+11> jmp 0x8049159 <stoiend>
0x8049152 <stoiaddm> mov cx,0xffffffff
0x8049157 <stoiaddm+5> mul ecx
0x8049159 <stoiend>  pop ecx
0x804915a <stoiend+1> pop ebx
0x804915b <stoiend+2> pop edx

native process 1872375 In: stoiend
0x0804911f in stoi () 
0x08049121 in stoi () 
0x08049122 in stoi () 
0x08049125 in stoi () 
0x08049145 in stoi () 
0x0804914a in stoi () 
0x0804914b in stoi () 
0x0804914c in stoiaddm () 
0x08049152 in stoiaddm () 
0x08049157 in stoiend () 
0x08049159 in stoiend () 

(gdb) 

```

Рисунок 18 — Инвертируем и выходим

Более подробно о работе этой функции я расписал в части 1.2, поэтому тут приведу только скриншоты с подписями. В конечном итоге мы получили ответ в регистре `eax` теперь переносим его в память и смотрим как он отображается в памяти:

```

eflags      0xa87      [ CF PF SF IF OF ]
eax         0xf8a3e0e2  -123477790
ecx         0xc        12
edx         0x804a024   134520868
ebx         0x804a024   134520868
esp         0xfffffd38c  0xfffffd38c
ebp         0x0        0x0
esi         0x0        0
edi         0x0        0
eip         0x80490e6   0x80490e6 <geti+54>

0x80490d4 <geti+36>  mov eax,0x0
0x80490d9 <geti+41>  mov ebx,ecx
0x80490db <geti+43>  mov ecx,edx
0x80490dd <geti+45>  mov edx,ebx
0x80490df <geti+47>  call 0x80490fb <stoi>
0x80490e4 <geti+52>  mov DWORD PTR [ebx],eax
0x80490e6 <geti+54>  pop ecx
0x80490e7 <geti+55>  pop edx
0x80490e8 <geti+56>  pop ebx
0x80490e9 <geti+57>  pop eax

native process 1872375 In: geti
0x08049152 in stoiaddm ()
0x08049157 in stoiaddm ()
0x08049159 in stoiend ()
0x0804915a in stoiend ()
0x0804915b in stoiend ()
0x0804915c in stoiend ()
0x0804915d in stoiend ()
0x080490e4 in geti ()
0x080490e6 in geti ()
(gdb) x/x 0x804a024
0x804a024: 0xf8a3e0e2
(gdb) 

```

Рисунок 19 — Число в памяти

Для проверки `calc` необходимы числа поменьше, поэтому перезапустим программу и с помощью команды `break calc` устроновим точку останова уже непосредственно перед арифметическими операциями.

В качестве переменных будем использовать  $a = 1; c = -13; k = 5$ .

$$\begin{aligned}f &= (1 - (-13))^2 + 2 \times 1 \times \frac{(-13)^3}{5^2 + 1} = 14^2 + 2 \times -\frac{2197}{26} = \\&= 196 + \frac{-4394}{26} = 196 - 169 = 27\end{aligned}$$

При выполнении деления перед умножением в результате мы получили бы  $-84,5$ , остаток отбросился и результат был бы неверен.