«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	21.02.24	А. П. Плютто
	(Группа)	(Подпись, дата)	(И. О. Фамилия)
Преподаватель		21.02.24	
		(Подпись, дата)	(И. О. Фамилия)

Москва 2024 г.

Содержание

1. Ввод чисел

Перед выполением 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 регистр `edx` и длина 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 буфкр число введенное пользователем.

После ввода числа пользователь нажимает на enter (LF) и число записывается пофиерно 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 числов 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 арифмитических операций, 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 ACSII, 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 напистъ 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 `esx` 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 выходим из процедуры `stoi`.

```
stoiend:
    pop ecx
    pop ebx
    pop edx
    pop esi

    ret
```

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

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

2. Вывод чисел

После выполнения арифмитических 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 прервывание на вывод предложения перед числом, после этого готовим регистры и выполняем процкдуру `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 убераем 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 чисамо 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 с 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. Эту 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 → `'-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 позльзователю, 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 арифмитических операций будет в `calc` переходим в нее, когда значения успешно введены.

3.3.3. Вычисление значения функции

Перепишем задание и разобъем 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.