ВВЕДЕНИЕ

В данном пособии рассматриваются основные аспекты разработки приложений с использованием машинно-ориентированного языка Ассемблера. Содержание пособия отражает основные вопросы раздела «Принципы функционирования микропроцессоров» учебной программы курса «Программирование» для студентов факультета прикладной математики и информатики БГУ.

Для изложения выбрана версия языка Ассемблера для процессоров семейства Intel™ на платформе Windows. Выбор обоснован большой популярностью и широким распространением таких процессоров, а также тем, что их архитектура представляется достаточно типичной и ее принципы распространяются на другие платформы.

Знание языка Ассемблера необходимо профессиональным программистам по ряду причин. Во-первых, язык процессора будет необходим, пока используются сами процессоры.

Во-вторых, разработка эффективных программ невозможна без глубокого понимания процессов их выполнения на компьютере. В программах на машинно-ориентированном языке Ассемблера можно реализовать оптимизацию по различным критериям наиболее эффективно, основываясь на знаниях деталей выполнения на уровне процессора, что не всегда доступно компиляторам высокого уровня в силу их универсальности. Использование аппаратных факторов, таких как регистровая память, буферизация, кэширование, выравнивание границы адреса и других, позволяет существенно увеличить быстродействие программ.

В-третьих, общие принципы оптимизации программ, применимые к критическим по ресурсам участкам программ, наиболее эффективно реализуются на машинном языке. Они в комплексе могут использовать как аппаратную составляющую, так и теоретические подходы к оптимизации по быстродействию и по объему используемых ресурсов.

В пособии рассматриваются вопросы разработки 32-разрядных приложений на основе как встроенного Ассемблера, так и использовании отдельных ассемблерных модулей в среде Visual C++. На базе первого подхода изучаются вопросы обработки целочисленных данных, что представляется удачным введением в язык Ассемблера. Использование среды избавляет от ряда деталей Ассемблера, трудных для первоначального понимания, в частности, от процедур преобразования данных и ввода-вывода, что требует определенного уровня знаний от студента.

Разработка модулей на Ассемблере закрепляет знания о системном подходе к взаимодействию разноязыковых модулей, использованию сис-

темных соглашений и особенностям работы в конкретных операционных средах.

Вопросы, сопровождающие отдельные разделы пособия, ориентированы как на изучение изложенного материала, так и на самостоятельный поиск ответов для углубленного понимания использования языка Ассемблера в той или иной ситуации.

Для разработки программ на Ассемблере будем использовать макроассемблер MASM фирмы Microsoft, выбор которого сделан из следующих соображений:

— MASM является наиболее популярной средой разработки программ на Ассемблере;

— соглашения, принятые в MASM, поддерживаются большинством ассемблеров, например, транслятором TASM;

— стандарты и соглашения, применяемые в MASM, совместимы с принятыми в таких средах разработки, как Visual C++ и Borland Delphi. Тем самым откомпилированные макроассемблером объектные файлы можно подключать к программам, разработанным на языках высокого уровня.

Ограниченный размер пособия не позволяет достаточно подробно изложить весь учебный материал, из-за чего для более полного освоения предмета пользователю необходимо выполнить большой объем самостоятельной работы по поиску ответов как на поставленные в пособии вопросы, так и на вопросы, возникающие в процессе изучения представленного материала. Для нахождения ответов можно воспользоваться рекомендуемой литературой и другими источниками по программированию на Ассемблере. А вопросы неизбежно будут появляться по мере усложнения задач, для решения которых нужно использовать весь арсенал современных программных средств.

1 КОМАНДЫ ОБРАБОТКИ ЦЕЛЫХ ЧИСЕЛ

1.1 Общие сведения

Языки высокого уровня поддерживают возможность вставки ассемблерного кода. Последовательность команд Ассемблера в С-программе размещается в asm-блоке

```
__asm
{
        команда_1                       //комментарий_1
        команда_2                       /* комментарий_2 */
        ------------------------
        команда_n                       ; комментарий_n
}
```

Команда имеет формат:

[имя_метки:] код_операции [операнды]

Операнды обозначают объекты, над которыми производятся операции, и разделяются запятыми. Операнды обычно хранятся либо в оперативной памяти компьютера, либо в специальной регистровой памяти процессора. Некоторые команды допускают указание значений операндов непосредственно в самой команде. В качестве операндов можно использовать идентификаторы переменных, определенных в С-программе. Этот вариант определения переменных используется в данном разделе. При хранении в памяти многобайтных данных используется принцип: «по младшим адресам хранятся младшие байты». С-идентификатор интерпретируется как адрес младшего байта объекта. Для доступа к другим байтам можно использовать операторы + и -.

В команде может участвовать не более одного операнда из памяти. Команда, имеющая два операнда, адресует их одним из способов: регистр–регистр, регистр–память, память–регистр, регистр–число, память–число.

Регистровая память состоит из конечного числа регистров, имеющих фиксированные имена, которые можно использовать в командах. К регистрам относят: регистры общего назначения, сегментные регистры и регистры состояния и управления. На рисунке изображены регистры общего назначения.

К регистрам состояния и управления относят регистр флагов EFLAGS и регистр счетчика команд EIP. Биты регистра флагов называют флагами. Они используются для анализа полученного результата и управления работой программы. Флаги изменяют свое значение при определенных условиях и имеют специальные названия, например:

— CF – флаг переноса;
— OF – флаг переполнения;
— SF – флаг знака результата;
— ZF – флаг нуля.

В регистре EIP хранится адрес выполняемой команды.

1.2 Команда MOV

Это основная команда пересылки данных, которая имеет следующий формат

MOV приемник, источник

Команда обеспечивает копирование значения из операнда-источника в операнд- приемник.

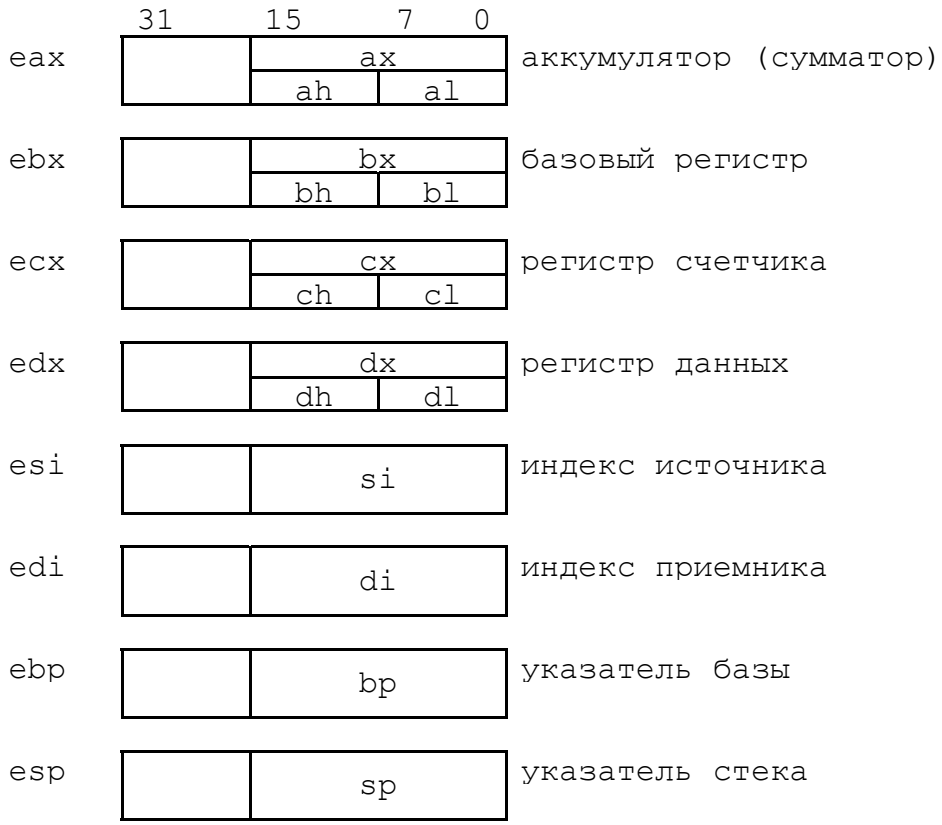

Рисунок – Регистры общего назначения

Формат данной команды типичен для большинства арифметических и логических команд процессора. Оба операнда должны быть одного типа или иметь одинаковую длину.

Примеры использования команды

```
mov    eax, ebx
mov    result, ax      ; result – 16-битная переменная
mov    ecx, count      ; count - 32-битная переменная
mov    ah, 64          ; 64 – непосредственный операнд
mov    var1, 100
```

Бывают случаи, когда не ясен размер операнда. Для такого уточнения можно использовать оператор переопределения типа **PTR**. Типы могут быть **byte**, **word**, **dword** и другие. Например, для получения младшего байта 32-битной переменной x можно использовать команду

```
mov al,byte ptr x
```

1.3 Двоичная арифметика

Операнды арифметических команд могут быть следующих типов: 8-разрядные (BYTE), 16-разрядные (WORD), 32-разрядные (int). Команды изменяют биты регистра флагов.

Команда сложения

ADD приемник, источник

складывает источник с приемником и размещает результат в приемнике. Выполняет сложение как знаковых, так и беззнаковых (signed/unsigned на языке С) целых чисел.

Пример 1. Вычислить result =var1 + var2 + 10;

```
int var1=34, var2=-12, result;
__asm
{
        mov  eax,var1
        add  eax,var2
        add  eax,10
        mov  result,eax
}
```

Команда вычитания

SUB приемник, источник

вычитает источник из приемника и размещает результат в приемнике.

Пример 2. Вычислить result = var1 – var – 1;

```
__asm
{
        mov  eax,var1
        sub  eax,var2
        sub  eax,1          //или      dec eax
        mov  result,eax
}
```

Унарная команда **INC** увеличивает значение операнда на единицу, **DEC** – уменьшает на единицу. Унарная команда **NEG** изменяет знак операнда.

Команда умножения беззнаковых чисел

MUL множитель2

умножает содержимое регистра AL, AX, EAX (в зависимости от размера множителя2) на множитель2 и размещает результат в AX, DX:AX, EDX:EAX соответственно.

Для умножения знаковых чисел используется команда **IMUL**.

Пример 3. Вычислить result=var1*var2*10;

```
WORD var1=3, var2=-12, result;
__asm
{
  mov  ax,var1
  imul var2      //результат в паре dx:ax
```

```
  mov  bx,10
  imul bx        //результат в паре dx:ax
  mov  result,ax //если Вы уверены, что результат не
                 //превосходит 16-разрядного значения
}
```

Команда деления беззнаковых чисел

DIV делитель

делит содержимое регистра AX, DX:AX, EDX:EAX (в зависимости от размера делителя) на делитель и размещает частное в AL, AX, EAX, а остаток – в AH, DX, EDX соответственно.

Для деления знаковых чисел используется команда **IDIV**. Знак остатка совпадает со знаком делимого.

Пример 4. Разделить 20 на 3.

```
__asm
{
  mov  ax,20
  mov  cl,3
  div  cl   //частное в регистре al, остаток в ah
}
```

В процессе выполнения команд деления может возникнуть исключительная ситуация, вызванная тем, что результат не может быть размещен в отведенной ему памяти, а также в случае деления на 0. Соответствующая проверка возлагается на пользователя.

Пример 5. Вычислить значение выражения $(ax^2-bx+10)/(x-a)$

```
 int a=-2, b=4, x=11, result;
__asm
{
  mov  eax,a
  imul x
  imul x
  mov  ecx,eax;  ecx = a*x*x

  mov  eax,b
  imul x;        eax = b*x
  sub  ecx,eax
  add  ecx,10;   eax = a*x*x-b*x+10

  mov  eax,ecx
  cdq
  mov  ecx,x
  sub  ecx,a; ecx = x-a
  idiv ecx

  mov  result,eax
```

```
    }
```

Команды **ADC** и **SBB** аналогичны ADD и SUB соответственно, но при этом добавляют/вычитают флаг переноса CF.

Пример 6. Найти разность двух 64-битных целых чисел.

```
long long int
var1=8888888888888888,
var2=1234567891234567890,res;
__asm
{
  mov  eax,dword ptr var1
  sub  eax, dword ptr var2   ;вычесть младшие половинки
                             ;чисел
  mov  dword ptr res,eax     ;младшая часть результата
  mov  eax, dword ptr var1+4
  sbb  eax, dword ptr var2+4 ;вычесть старшие половинки
                             ;чисел
  mov  dword ptr res+4,eax   ;старшая часть результата
}
```

1.4 Логические команды

Логические команды выполняют побитовые действия над операндами. К логическим относят унарную команду отрицания (инвертирования) **NOT** и бинарные команды: логическое умножение (конъюнкция) **AND**, логическое сложение (дизъюнкция) **OR**, исключающее или (сложение по модулю 2) **XOR**. Они изменяют биты регистра флагов.

Формат бинарных команд совпадает с форматом команды сложения. Команда NOT имеет единственный операнд (регистр или память).

Логические операции чаще всего применяются для установки и проверки состояния отдельных битов 8-, 16-, 32-разрядных чисел.

Примеры использования команд

```
xor    ax,ax     ;очистить все биты регистра ax
and    ecx,0     ; очистить все биты регистра ecx
or     ax,1010101010101010b;установить все четные
                           ;биты регистра в 1
and    bl,11111000b   ;сбросить три младших бита в 0
not    dh        ;получить инвертированный результат
```

1.5 Команды сдвига

Команда логического сдвига вправо

SHR приемник, счетчик

сдвигает вправо биты приемника на значение счетчика (число или регистр CL) и размещает результат в приемнике. Освободившиеся биты заполняются нулями. Последний выдвинутый бит заносится во флаг CF.

Принципы работы команд **SHL** (логический сдвиг влево), **SAR** (арифметический сдвиг вправо) и **SAL** (арифметический сдвиг влево) аналогичны. Но команда SAR заполняет освободившиеся биты знаковым разрядом.

Команды сдвига можно использовать при умножении/делении на степень двойки, например `shl i,3` увеличит значение i в 8 раз.

1.6 Команды преобразования

Следующие команды обычно используют для получения эквивалентных, но больших по размеру операндов.

Команда пересылки с расширением нулями

MOVZX регистр-приемник, источник

копирует источник в приемник (16- или 32-разрядный), старшие биты при этом устанавливаются в 0.

Команда **MOVSX** работает аналогично, только старшие биты заполняются знаковым разрядом.

При подготовке делимого в операции деления можно использовать команды: **CBW** (расширение байта до слова), **CWD** (расширение слова до двойного слова) и **CDQ** (расширение двойного слова до учетверенного). Команды не имеют операндов и размещают результат так, как этого требуют команды деления.

Пример.

```
int a=-75000, b=100, result;
__asm
{
        mov  eax,a
        cdq                 ; делимое в edx:eax
        idiv b
        mov  result,eax
}
```

Вопросы для самопроверки

1. Опишите формат команды Ассемблера.
2. Где могут храниться операнды, используемые в команде?
3. Определите синтаксис asm-блока.
4. Для чего используется оператор PTR?
5. Как обработать в выражении данные разной длины?
6. Какие флаги устанавливают команды сложения/вычитания?
7. Как процессор определяет знак операнда?
8. Возможно ли переполнение при умножении?

9. Где находятся множимые и делимое/делитель?
10. Куда помещаются частное/остаток?
11. Как проверить значение отдельного бита/группы битов числа?
12. Как установить значение отдельного бита/группы битов числа?
13. Как определить количество единичных (нулевых) битов числа с помощью команд сдвига?
14. Как умножить/разделить число на степень двойки с помощью команд сдвига?
15. Когда необходимо использовать команды расширения числа?

Задачи

1. Заданы два слова. Поменять местами их значения.
2. Решить уравнение ax+b=0 при a≠0.
3. Найти значение выражения (2ac-b/x-12)/(cx+a).
4. Поменять местами байты слова.

2 ОРГАНИЗАЦИЯ ПЕРЕХОДОВ И ЦИКЛОВ

Команды переходов и циклов изменяют смещение следующей исполняемой команды (содержимое регистра EIP), нарушая тем самым линейный порядок выполнения команд. С этими командами обычно используют команды сравнения.

2.1 Команды сравнения

Они устанавливают флаги в регистре EFLAGS. Затем с помощью команд условного перехода можно организовать соответствующие действия. Не стоит забывать, что флаги могут устанавливаться и другими командами, в частности, командами сложения и вычитания.

Основной командой сравнения является команда CMP:

CMP приемник, источник

Из приемника вычитается источник и устанавливаются флаги, результат вычитания нигде не сохраняется.

Команда TEST:

TEST приемник, источник

Выполняет операцию AND над источником и приемником и формирует флаги, результат логического умножения никуда не помещается.

2.2 Команды переходов

Команды переходов делятся на команды безусловного и условных переходов. Они изменяют значение регистра EIP. Такие команды могут использовать метки.

Для безусловного перехода используется команда

JMP адрес_перехода

Команда передает управление по заданному адресу. Операндом обычно является имя метки, т.е. непосредственный адрес для перехода.

Команды условного перехода имеют вид

Jxx адрес_перехода

Эти команды осуществляют переход, если выполняется условие, определяемое состоянием одного либо нескольких флагов.

В таблице 1 приведены расшифровки мнемонических обозначений, используемых в командах условного перехода, в таблице 2 – сами команды.

Таблица 1

Мнемоническое обозначение	Термин	Перевод	Примечание
E	Equal	Равно	
N	Not	Не	
G	Greater	Больше	для знаковых
L	Less	Меньше	для знаковых
A	Above	Выше (в смысле больше)	для беззнаковых
B	Below	Ниже (в смысле меньше)	для беззнаковых
Z	Zero	Ноль	
S	Sign	Знак	
O	Overflow	Переполнение	
P	Parity	Паритет	
C	Carry	Перенос	

Таблица 2

Команда	Условие перехода	Флаги
JA JNBE	Если выше Если не ниже и не равно	CF = 0 и ZF = 0
JAE JNB JNC	Если выше или равно Если не ниже Если нет переноса	CF = 0
JB JNAE JC	Если ниже Если не выше и не равно Если перенос	CF = 1
JBE JNA	Если ниже или равно Если не выше	CF = 1 и ZF = 1
JE JZ	Если равно Если ноль	ZF = 1

Продолжение Таблицы 2

JG JNLE	Если больше Если не меньше и не равно	ZF = 0 и SF = OF
JGE JNL	Если больше или равно Если не меньше	SF = OF
JL JNGE	Если меньше Если не больше и не равно	SF ≠ OF
JLE JNG	Если меньше или равно Если не больше	ZF= 1 или SF ≠OF
JNE JNZ	Если не равно Если не ноль	ZF = 0
JNO	Если нет переполнения	OF = 0
JO	Если есть переполнение	OF = 1
JNP JPO	Если нет паритета (количество битов результата нечетное)	PF = 0
JP JPE	Если есть паритет (количество битов результата четное)	PF = 1
JNS	Если нет знака	SF = 0
JS	Если есть знак	SF = 1

Для реализации перехода по условию CX/ECX = 0 можно использовать команды

JCXZ / JECXZ метка

2.3 Реализация ветвлений

Для обеспечения ветвления

```
if (условие)
   команды_если_истина
```

можно использовать конструкцию

```
CMP операнд1, операнд2
Jxx метка_если_ложь
   ;команды_если_истина
метка_если_ложь:
```

Для обеспечения ветвления if-else

```
if (условие)
   команды_если_истина
else
   команды_если_ложь
```

можно использовать конструкцию

```
CMP операнд1, операнд2
Jxx метка_если_ложь
   ;команды_если_истина
   JMP метка
метка_если_ложь:
   ;команды_если_ложь
метка:
```

Пример 1. Определить, является ли число четным. Для решения задачи анализируется младший бит числа.

```
int x=15;
_asm
{
     mov eax, x
     test eax, 1
     jz not_p       ; переход, если четное
     ; действия, если нечетное
     jmp end_
not_p:
     ; действия, если четное

end_:
}
```

Пример 2. Найти максимум трех чисел.

```
int a=15,b=-77,c=22,max;
_asm
{
     mov eax, a
     cmp eax, b
     jg gr1
     mov eax, b
gr1:                ; в eax максимум a и b
     cmp eax, c
     jg gr2
     mov eax, c
gr2:                ; в eax максимум трех
     mov max, eax
}
```

2.4 Реализация циклов

Для обеспечения цикла while

```
while (условие)
   команды_тела_цикла
```

в Ассемблере можно использовать конструкцию

```
метка:
    CMP операнд1, операнд2    ; проверка условия
    Jxx метка_выхода
       ;команды_тела_цикла
       JMP метка
метка_выхода:
```

Пример 1. Найти наибольший общий делитель двух чисел a и b. Алгоритм:

```
while (a!=b)
   if (a>b)
      a=a-b;
   else
      b=b-a;
```

может быть реализован:

```
int a=100,b=204,result;
_asm
{
    mov eax, a
    mov ebx, b
beg:
    cmp eax, ebx
    je end_              ; if equal
       jg gr             ; if greater
       sub ebx, eax
       jmp beg
       gr:
       sub eax, ebx
       jmp beg
end_:
    mov result,eax       ;result как в eax, так и в ebx
}
```

Для обеспечения цикла do-while

```
do
   команды_тела_цикла
while (условие)
```

в Ассемблере можно использовать конструкцию

```
метка:
   ;команды_тела_цикла
   CMP операнд1, операнд2
   Jxx метка
```

Для организации цикла со счетчиком удобно использовать команду

LOOP метка

В регистр ECX нужно поместить количество повторений цикла перед командами, составляющими тело цикла. Команда работает так: значение регистра ECX уменьшается на 1, сравнивается с нулем, и если оно не равно нулю, производится передача управления на команду, помеченную *меткой*. Иначе происходит переход к следующей за LOOP команде.

Пример 2. Найти сумму 1+2+3+…+x

```
int x=15,sum;
_asm
{
     mov eax, 0     ; сумма
     mov ecx, x     ; счетчик
beg:
        add eax, ecx
     loop beg
     mov sum,eax
}
```

Вопросы для самопроверки

1. Почему переход по результатам сравнения знаковых и беззнаковых чисел выполняется различными командами?
2. Что такое метка? Можно ли перейти на метку, определенную выше/ниже команды перехода?
3. Как далеко может отстоять метка в команде перехода?
4. Как задается количество итераций в цикле LOOP?
5. Что произойдет, если количество итераций для цикла LOOP равно нулю?
6. Можно ли выйти из цикла командой перехода?
7. Можно ли войти в тело цикла командой перехода?

Задачи

1. Дробь задается целым числителем и натуральным знаменателем. Если возможно, сократить ее.
2. Определить, является ли четным либо нечетным количество единичных разрядов в числе.
3. Определить, является ли натуральное число симметричным.
4. Найти сумму цифр натурального числа.
5. Определить количество четных цифр в натуральном числе.
6. Вычислить количество значащих разрядов в двоичной записи натурального числа.

7. Вычислить количество единичных разрядов в двоичной записи натурального числа.

8. Вычислить сумму первых N чисел Фибоначчи.

9. Приписать по единице в начало и конец записи натурального числа в десятичной системе счисления.

10. Для двух натуральных чисел n, m получить сумму m последних цифр числа n.

11. Приписать по единице в начало и конец записи натурального числа в двоичной системе счисления.

12. Поменять порядок цифр натурального числа на обратный.

13. Переставить первую и последнюю цифры натурального числа.

3 МАССИВЫ

Массив – упорядоченная последовательность элементов одного типа. Для работы с массивами необходимо уметь: определять массив и инициализировать его элементы, осуществлять доступ к элементам. Для этого используют различные способы адресации.

3.1 Способы адресации и адресная арифметика

При 32-разрядном программировании логический (виртуальный) адрес получается:

селектор_сегмента:32-разрядный_эффективный_адрес

Эффективный 32-разрядный адрес (Effective Address) равен количеству байтов от начала сегмента (смещение). В общем случае он вычисляется сложением любой комбинации следующих четырех адресных элементов:

*Эффективный_адрес = База + (Индекс * Масштаб) + Смещение,*

где

- Смещение – числовое значение;
- База – содержимое любого из регистров общего назначения; используемый здесь регистр называется базовым;
- Индекс – содержимое любого из регистров общего назначения, кроме esp; используемый здесь регистр называется индексным;
- Масштаб – константа 2, 4 или 8.

Для получения эффективного адреса переменной можно использовать оператор OFFSET

mov приемник, offset имя_переменной

или команду LEA (Load Effective Address)

LEA приемник, имя_переменной

Например,

```
lea     ebx, a          ; в ebx загружается адрес переменной  a
mov     ebx, offset a   ; аналогично предыдущей команде
```

В командах используются различные способы задания адресов хранения операндов (способы адресации):

– ***Регистровая адресация***. Операнд находится в регистре, название которого указывается в команде. Например

```
sub     eax, ebx
```

– ***Непосредственная адресация***. Операнд задается в команде.

```
add     eax, 5
mov     cl, 'D'
```

– ***Прямая адресация***. Операнд находится в памяти по адресу, который задается в команде.

```
mov     edx, ds:a ; адрес задан парой сегмент:смещение
                  ; Ассемблер заменит имя на
                  ; соответствующее смещение
mov     ebx, a    ;сегментный регистр по умолчанию ds
```

– ***Косвенно-регистровая адресация***. Операнд находится в памяти, его адрес – в регистре. Например, в команде

```
add     eax,[ebx]
```

адрес второго операнда находится в регистре ebx.

– ***Базовая адресация.*** Адрес операнда равен сумме содержимого регистра и смещения. Например:

```
mov     edx,8[ebp]      ;другие формы записи: [ebp+8]
                         ;или [ebp]+8
```

– ***Индексная адресация.*** Адрес операнда равен сумме содержимого регистра и смещения. Например, в команде

```
mov     mass[esi],al
```

адрес первого операнда равен сумме смещения, соответствующего имени mass, и содержимого регистра esi.

– ***Базово-индексная адресация.*** Адрес операнда вычисляется как сумма содержимого базового и индексного регистров. Например:

```
mul     [ebx][esi]      ; или [ebx+esi]
```

– ***Базово-индексная адресация со смещением.*** Адрес операнда вычисляется как сумма смещения и содержимого базового и индексного регистров. Например:

```
mul     4[ebx][esi];или [ebx+esi+4], [ebx][esi]+4,…
```

– ***Индексная адресация с масштабированием.*** Адрес операнда равен сумме смещения и содержимого индексного регистра, умноженного на масштаб (1, 2, 4 или 8). Например:

```
mov     eax, mas[ecx*2]
```

– ***Базово-индексная адресация с масштабированием.*** Адрес операнда равен сумме содержимого базового регистра и содержимого индексного регистра, умноженного на масштаб. Например:

```
add    eax,[ebx+esi*4]
```

– ***Базово-индексная адресация со смещением и масштабированием.*** Самая полная схема адресации, в которой для вычисления адреса используются база, индекс, масштаб и смещение.

3.2 Одномерные массивы

Элементы массива располагаются в памяти компьютера последовательно. Доступ к элементу массива обычно осуществляется операцией индексирования, которую можно моделировать, зная начальный адрес массива и размер его элемента в байтах. Тогда адрес i-го (i=0,1,…) элемента одномерного массива равен

*начальный _адрес+(i*размер_элемента).*

Пример 1. Вычислить произведение элементов одномерного массива. Осуществляется последовательный доступ ко всем элементам массива, начиная с 0-го. Используется косвенно-регистровая адресация: в регистре ebx хранится адрес текущего элемента массива.

```
int a[]={1,-2,3,40,5};
__asm
{
      lea   ebx,a            ; инициализация базы
      mov   ecx,5            ; счетчик
      mov   eax,1            ; произведение
begin:
      mul   dword ptr [ebx]
      add   ebx,4            ; переход к следующему элементу
      loop  begin
}
```

Пример 2. Целочисленное сложение длинных положительных чисел. В одномерных массивах a1 и a2 представлены десятичные числа, каждый элемент массива – одна цифра, старшая цифра расположена по нулевому индексу. Вычислить сумму чисел.

Сложение многоразрядных чисел начинается с младших разрядов и выполняется с учетом переноса. Здесь удобно использовать индексную адресацию.

```
      BYTE a1[10] = {0,5,5,5,5,5,7,8,2,3};
      // тип BYTE  определен в <windows.h>
      BYTE a2[10] = {0,6,7,1,9,0,9,3,1,8};
      BYTE sum[10]; // длина результата не превосходит длины
                    //операндов
```

```
__asm
{
     mov  esi,9     ; индекс, массивы обрабатываются
                    ;справа налево
     mov  ecx,10    ; счетчик
     xor  bl,bl     ; перенос
begin:
     mov  al,bl     ; в регистре al – разряд суммы
     add  al,a1[esi]
     add  al,a2[esi]
     aaa            ; коррекция ASCII-формата для сложения
     jc   m_carry1  ; есть перенос
     xor  bl,bl
     jmp  m_carry0  ; нет переноса
m_carry1:
     mov  bl,1
m_carry0:
     mov  sum[esi],al
     dec  esi
     loop begin
}
```

При обработке массивов одинарных, двойных и учетверенных слов удобно использовать адресацию с масштабированием, в регистре будет храниться индекс, а не смещение от начала массива.

Пример 3. Вычислить целую часть среднего арифметического элементов динамического одномерного массива. Используется адресация по базе с индексированием и масштабированием.

```
int* a=new int [5];
a[0]=1,a[1]=-2,a[2]=3,a[3]=-40,a[4]=0;
__asm
{
     mov  ebx,a          ; база
     xor  esi,esi        ; индекс
     mov  ecx,5          ; счетчик
     xor  eax,eax        ; сумма
begin:
     add  eax,[ebx][esi*4]
     inc  esi
     loop begin
     mov  ecx,1
     imul ecx
     mov  ecx,5
     idiv ecx            ; в регистре eax– результат
}
```

Пример 4. Вычислить длину строки.

```
TCHAR s[100] = TEXT("Hello");
// тип TCHAR определен в <windows.h>
int  typeSize = sizeof(TCHAR);
__asm
{
     xor  esi,esi              ; индекс
     xor  ecx,ecx              ; длина
begin:
     cmp  s[esi],0             ; индексная адресация
     je   end                  ; 0 – конец строки
     inc  ecx
     add  esi,typeSize
     jmp  begin
end:
}
```

Пример 5. Вычислить сумму элементов массива:

int a[]={1,-2,3,-40,5};

```
// адресация по базе со сдвигом
__asm
{
     mov  ecx,5          ;
счетчик
     xor  eax,eax   ; сумма
     xor  ebx,ebx   ; сдвиг
begin:
     add  eax,a[ebx]
     add  ebx,4
     loop      begin
}
```

```
// адресация с масштабированием
__asm
{
     mov  ecx,5          ;
счетчик
     xor  eax,eax   ; сумма
     xor  ebx,ebx   ; индекс
begin:
     add  eax,a[ebx*4]
     inc  ebx
     loop      begin
}
```

3.3 Двухмерные массивы

Элементы двухмерного массива располагаются в памяти компьютера последовательно. В С двухмерные массивы располагаются по строкам. Тогда для массива размерности n×m элемент в строке i (0≤i<n) и столбце j (0≤j<m) имеет адрес

*начальный _адрес+(i*m*размер_элемента)+(j*размер_элемента).*

Пример 1. Вычислить позицию и значение наибольшего элемента двухмерного массива.

Осуществляется последовательный доступ к элементам двухмерного массива как к элементам одномерного. Используется косвенная адресация с масштабированием; по индексу вычисляется позиция (строка и столбец) элемента в двухмерном массиве.

```
int M[3][4] = {{100,2,33,4},{5,6,77,8},{900,10,11,120}};
int n=3, m=4, max;
__asm
{
     mov  eax,n
     mul  m
     mov  ecx,eax        ; количество элементов массива
     xor  esi,esi        ; индекс
     mov  edx,M          ; максимальный элемент
     mov  eax,0          ; индекс максимального элемента
begin:
     cmp  edx,M[esi*4]
     jnl  m_notMax       ; Jump Not Less
     mov  edx,M[esi*4]
     mov  eax,esi
m_notMax:
     inc  esi
     loop      begin
     mov  max,edx        ; сохраняем максимум
     xor  edx,edx        ; вычисляем позицию
     div  m;в eax – номер строки, в edx – номер столбца
}
```

Пример 2. В примере демонстрируется схема последовательного доступа к элементам двухмерного массива. Используется базово-индексная адресация с масштабированием.

```
int M[3][4] = {{100,2,33,4},{5,6000,77,8},{900,10,11,1200}};
int n=3, m=4;
     int mSize=m*sizeof(int);      // m*размер_элемента
     __asm
     {
          lea  ebx,M     ; адрес начала строки
          mov  ecx,n     ; счетчик внешнего цикла по строкам
     begin_n:
          push      ecx  ; сохранение счетчика внешнего цикла
          mov  ecx,m     ; счетчик внутреннего цикла по
                         ;элементам строки
          xor  esi,esi   ; индекс текущего элемента строки
     begin_m:
          // обработка текущего элемента строки
          // его адрес – [ebx][esi*4]
          // …
          inc  esi
          loop begin_m
          add  ebx,mSize ; переход к началу следующей строки
          pop  ecx  ; восстановление счетчика внешнего цикла
          loop begin_n
     }
```

Пример 3. По матрице A получить транспонированную матрицу B. Для доступа к элементам матриц используется косвенная адресация с масштабированием.

```
int A[3][4] = {{100,2,33,4},{5,600,77,8},{900,10,11,1200}};
int B[4][3];
int n=3, m=4;
__asm
{
     mov  ecx,0          ; i – номер текущей строки A
begin_n:
     mov  ebx,0          ; j – номер текущего элемента A
begin_m:
     // адрес A[i][j] равен A+(i*m+j)*размер элемента
     // вычисляем i*m+j
     mov  eax,m
     mul  ecx
     add  eax,ebx
     mov  esi,eax        ; esi ← (i*m+j)
     mov  edi,A[esi*4]   ; edi ← A[i][j]
     // адрес B[j][i] равен B+(j*n+i)*размер элемента
     // вычисляем j*n+i
     mov  eax,n
     mul  ebx
     add  eax,ecx
     mov  esi,eax        ; esi ← (j*n+i)
     mov B[esi*4],edi    ; B[j][i] ← edi
     inc  ebx            ; к следующему элементу в строке
     cmp  ebx,m
     jl begin_m          ; не все элементы обработаны
     inc  ecx            ; к следующей строке
     cmp  ecx,n
     jl begin_n          ; не все строки обработаны
}
```

Вопросы для самопроверки

1. Как располагаются в памяти элементы одномерного/двухмерного массива?

2. Как получить в цикле доступ к очередному элементу массива, используя различные способы адресации?

3. Определить значение регистров после выполнения каждой команды

```
#include <windows.h>
void main()
{
```

```
    BYTE myBytes[] = {0x10, 0x20, 0x30, 0x40};
    WORD myWords[] = {0x8A, 0x3B, 0x72, 0x44, 0x66};
    DWORD myDoubles[] = {1, 2, 3, 4, 5};
    DWORD* myPointer = myDoubles;
    _asm
    {
         lea  esi, myBytes
         mov  al, [esi]
         mov  al, [esi+3]
         lea  esi, myWords+2
         mov  ax, [esi]
         mov  edi, 8
         mov  edx, [myDoubles+edi]
         mov  edx, myDoubles[edi]
         mov  ebx, myPointer
         mov  eax, [ebx+4]
         lea  esi, myBytes
         mov  ax,word ptr [esi]
         mov  eax, dword ptr myWords
         mov  esi, myPointer
         mov  ax, word ptr [esi+2]
         mov  ax, word ptr [esi+6]
    }
}
```

Задачи

1. Задан одномерный упорядоченный массив определенной размерности, содержащий различные элементы, и число. Используя стратегию поиска «деление отрезка пополам», определить, присутствует (тогда вывести его номер) или отсутствует такой элемент в массиве.

2. Задан одномерный массив определенной размерности. Найти длину и указать индекс начала фрагмента, содержащего наибольшее число одинаковых следующих друг за другом элементов. Учитывать, что таких фрагментов может быть несколько.

3. Задан одномерный массив определенной размерности. Получить другой массив, не содержащий повторяющихся элементов.

4. Рассматривая массив как представление некоторого множества (если значение элемента равно 1, то элемент принадлежит множеству, иначе не принадлежит), найти объединение, пересечение и разность двух множеств, заданных в виде массивов.

5. Даны координаты точек на плоскости, представленные в виде одного одномерного массива. Найти номера двух точек, расстояние между которыми наибольшее. Учитывать, что таких пар точек может быть несколько.

6. Двумя массивами задаются коэффициенты двух многочленов разных степеней. Определить их произведение и сумму.

7. Определить количество различных элементов массива и самые часто встречающиеся элементы.

8. Длинное число представлено как массив, содержащий цифры. Найти факториал заданного длинного числа.

9. Рассматривая массив как представление некоторого множества (в массиве хранятся элементы множества), найти объединение, пересечение двух множеств, заданных в виде массивов.

10. Длинное восьмеричное число представлено как массив, содержащий цифры. Выполнить сложение и вычитание двух таких «длинных» чисел.

11. Слить два упорядоченных массива.

12. Удалить из первого массива элементы, встречающиеся во втором массиве. Третий массив не использовать.

13. На плоскости задано множество точек, представленное в виде двух массивов. Определить минимальный по площади прямоугольник со сторонами, параллельными осям координат, и вмещающий в себя все точки.

4 СОЗДАНИЕ ПРИЛОЖЕНИЙ НА ЯЗЫКЕ АССЕМБЛЕРА

4.1 Процесс проектирования

Пакет MASM включает основные средства, необходимые для выполнения всех шагов разработки программ. Он поддерживает следующие этапы создания и выполнения программ:

1. Ассемблирование исходного текста программы (файл с расширением *asm*), в результате которого формируется объектный файл (с расширением *obj*). Ассемблирование выполняет утилита ***ml.exe***, вызов которой из командной строки может иметь следующий вид

ml /c /coff имя_файла.asm

Транслятор (ассемблер) ***ml.exe*** создает объектный файл, который в данном случае имеет формат ***coff***, применяемый при компоновке с приложениями, разработанными в среде Visual Studio.

2. Объединение (компоновка, сборка) полученного объектного файла с другими объектными файлами и библиотеками в исполняемый файл (с расширением *exe*). Для построения 32-разрядных приложений можно использовать:

link /subsystem:windows /opt:noref имя_файла.obj

или

link /subsystem:console /opt:noref имя_файла.obj

3. Выполнение полученного EXE-файла производится обычным способом

имя_файла.exe список параметров

4.2 Структура программы

Программы на языке Ассемблера обычно состоит из одного либо нескольких сегментов. Сегментом называют часть программы, состоящую из команд и/или данных.

Каждый сегмент имеет вполне определенное функциональное назначение, например, сегмент для хранения данных, сегмент команд программы, сегмент стека. Для процессоров Intel сегменты размещаются в оперативной памяти раздельно и независимо друг от друга. Для указания начального (базового) адреса сегмента предназначены специальные сегментные регистры CS, DS, SS, ES, FS, GS. Каждый из них используется для сегментов определенного назначения:

CS (Code Segment) предназначен для хранения базового адреса сегмента команд;

SS (Stack Segment) – для сегмента стека;

DS (Data Segment), ES (Extra Segment), FS, GS – для сегментов данных.

Программа, кроме команд процессора, содержит специальные инструкции, указывающие самому Ассемблеру на выполнение определенных действий, например, на определение переменных, сегментов.

Для определения сегмента используются директивы **SEGMENT** и **ENDS**:

```
имя_сегмента  SEGMENT [параметры]
        ...
имя_сегмента   ENDS
```

Программа на Ассемблере может выглядеть следующим образом:

```
; определение сегмента1
; ...
; определение сегментаn
END [точка входа]
```

Директива **END** завершает текст asm-файла, точка входа задается меткой первой исполняемой команды.

Директива **ASSUME** указывает, с каким сегментом или группой сегментов связан тот или иной сегментный регистр. Обычно она используется так:

ASSUME CS:имя_сегмента_кода, DS:имя_сегмента_данных, SS:имя_сегмента_стека

По умолчанию используются команды процессора 8086. Для использования команд более современных процессоров используются соответствующие директивы, например:

.386

Для определения данных в сегменте используются директивы **DB**, **DW**, **DD** и другие.

Примеры использования директив:

```
a     dw   13
b     db   'Hello',13,10,'$';  строка из восьми байтов
c     dd   ?;   неинициализированная переменная
ar1   dd   10,20,30,40,50; массив из пяти двойных слов
ar2   dw   10 dup(?); массив из десяти неинициализированных слов
ar3   db   15 dup(0); массив из 15-ти нулей
```

4.3 Модели памяти

Для упрощения процесса разработки приложений можно использовать модели памяти. Модель определяет способ организации адресного пространства оперативной памяти: количество сегментов и правила размещения. Для определения модели можно использовать директиву

.MODEL имя_модели

Различные модели памяти могут использовать один либо несколько сегментов кода и данных. Для доступа к их содержимому используются адреса типы **FAR** (селектор_сегмент:смещение) и **NEAR** (смещение).

Примеры моделей и их характеристики приведены в таблице.

Таблица – Модели памяти

Модель	**Тип адресации кода**	**Тип адресации данных**	**Назначение модели**
TINY	near	near	Код, данные и стек размещаются в одном сегменте размером до 64 Кб. Используется для создания небольших программ.
SMALL	near	near	Код занимает один сегмент, данные и стек – другой. Эту модель обычно используют для большинства программ на Ассемблере.
MEDIUM	far	near	Код занимает несколько сегментов, все данные – в одном.
COMPACT	near	far	Код в одном сегменте; данные — в нескольких

Продолжение Таблицы

LARGE	far	far	Код и данные могут занимать несколько сегментов
FLAT	near	near	Код, данные и стек размещаются в одном сегменте размером до 4 Гб. Используются для разработки 32-битных приложений.

4.4 Упрощенные директивы сегментации

Макроассемблер MASM содержит директивы, упрощающие определение сегментов программы и поддерживающие соглашения по управлению сегментами в языках высокого уровня (таблица).

Таблица – Упрощенные директивы определения сегмента

Директива	Назначение
.CODE	Начало или продолжение сегмента кода
.DATA	Начало или продолжение сегмента инициализированных данных. Также используется для определения данных типа near
.CONST	Начало или продолжение сегмента постоянных данных (констант) модуля
.DATA?	Начало или продолжение сегмента неинициализированных данных. Также используется для определения данных типа near
.STACK [размер]	Начало или продолжение сегмента стека модуля. Параметр [размер] задает размер стека
.FARDATA	Начало или продолжение сегмента инициализированных данных типа far
.FARDATA?	Начало или продолжение сегмента неинициализированных данных типа far

4.5 Модель памяти FLAT

Выберем «плоскую» модель управления памятью ***flat***, в которой все адреса являются near-адресами и определяются 32-битовыми смещениями от базового адреса соответствующего сегмента. Эта модель чаще всего используется компиляторами языков высокого уровня.

Рассмотрим простую программу на языке Ассемблера

```
.386
.model flat
.data
     sum dd 0
     a   dd 100

.code
start:
```

```
     mov eax,a
.data
     b   dword 200
.code
     add eax,b
     mov sum,eax
     ret                  ; выход
end start
```

Программа состоит из двух сегментов. Директивы .**DATA** и .**CODE** используются для определения начала или продолжения соответствующего сегмента.

Команда RET обеспечивает корректное завершение программы.

Вопросы для самопроверки

1. Определите основные этапы разработки программы.
2. Какие утилиты можно использовать для выполнения отдельных этапов?
3. Укажите основные параметры Ассемблера MASM для генерации объектного файла.
4. Укажите основные параметры компоновщика для генерации исполняемого файла.
5. Как получить файл листинга?
6. Какие файлы могут быть сгенерированы Ассемблером?
7. Какой формат объектного файла используется, если средой разработки является Visual C++?
8. На какие логические единицы разбивается исходная программа?
9. Какие директивы нужно использовать для разбиения на сегменты?
10. Приведите основные значения параметров директив сегментации.
11. Зачем на этапе ассемблирования нужны директивы сегментации?
12. Какие действия необходимо выполнить для корректного определения сегментных адресов на этапе выполнения?
13. Какие умолчания используются в упрощенных директивах сегментации?
14. Как определить имена сегментов, присвоенные им по умолчанию упрощенными директивами сегментации?
15. Как выделить память под переменные?
16. Как инициализировать переменные в директивах определения данных?
17. Как определить массив в директивах определения данных?
18. Как инициализировать переменную адресом другой переменной?

19. Чем отличаются модели памяти друг от друга?

20. В чем специфика модели flat?

21. Какую модель памяти использует компилятор Visual C++ на платформе Windows?

22. Почему для модели памяти flat нет необходимости использовать директиву assume?

5 ПРОЦЕДУРЫ

5.1 Стек

Часто программе требуется временно запомнить информацию. Для этого в программе используется специальный сегмент – сегмент стека, называемый стеком. При помещении элементов в стек происходит уменьшение указателя вершины стека, а при извлечении – увеличение. Т.е. стек всегда «растет» в сторону меньших адресов памяти.

Для работы со стеком используются регистры SS, ESP и EBP. Содержимое SS является базой стека. В ESP хранится смещение вершины стека. Первоначально ESP инициализируется наибольшим смещением, которого может достигать стек, изменяется операциями включения и извлечения. Регистр EBP обычно используется для обращений к элементам стека.

Ниже рассмотрены некоторые команды работы со стеком.

Команда сохранения данных в стеке

PUSH источник

Источником может быть регистр, сегментный регистр или переменная. ESP уменьшается на размер источника в байтах (2 или 4) и содержимое источника помещается в память по адресу SS:ESP.

Команда извлечения данных из стека

POP приемник

помещает в приемник слово или двойное слово, находящееся в вершине стека, увеличивая ESP на 2 или 4 соответственно. Приемником может быть регистр общего назначения, сегментный регистр, кроме CS или переменная. Если в роли приемника выступает операнд, использующий ESP для косвенной адресации, команда POP вычисляет адрес операнда уже после того, как она увеличивает ESP.

Команда PUSH зачастую используется в паре с POP.

Копирование одного сегментного регистра в другой (что нельзя выполнить одной командой MOV), можно реализовать так:

```
push   ds
pop    es
```

Для временного хранения данных можно поступить так:

```
push eax
; команды изменения EAX
pop eax
```

Команды **PUSHA** и **PUSHAD** помещают в стек все регистры общего назначения. PUSHA располагает в стеке регистры в следующем порядке: AX, CX, DX, BX, SP, BP, SI и DI. PUSHAD помещает в стек EAX, ECX, EDX, EBX, ESP, EBP, ESI и EDI. В случае с SP и ESP используется значение, которое находилось в регистре до начала работы команды. Команды используются в паре с командами **POPA** и **POPAD**, считывающими эти же регистры из стека в обратном порядке, это позволяет писать процедуры, которые не должны изменять значения регистров по окончании своей работы. В начале такой подпрограммы вызывают команду PUSHA, а в конце – POPA.

PUSHF/POPF

Сохранить/извлечь в/из стека регистр флагов

5.2 Определение и вызов процедуры

Традиционно сегмент кода состоит из процедур, хотя это и необязательно. Для оформления процедуры используется конструкция

```
Имя_процедуры PROC [NEAR|FAR]
    ;тело процедуры
    RET
Имя_процедуры ENDP
```

Можно указать тип процедуры: NEAR или FAR, по умолчанию используется NEAR. В зависимости от типа при вызове процедуры поразному запоминается в стеке адрес возврата. В случае ближней процедуры в стек помещается смещение, а при дальней – полный логический адрес (база и смещение). Команда RET извлекает из стека адрес возврата в регистр EIP .

Для вызова процедуры используется команда CALL:

CALL Имя_процедуры

5.3 Передача параметров

Передача параметров (по значению и по адресу) производится через регистры и через стек.

При передаче через регистры параметры заносятся в заранее оговоренные регистры, и вызываемая процедура оперирует с ними.

Пример 1. Процедура сложения двух целых. Использовать передачу параметров через регистры.

```
; слагаемые передаются через регистры eax и ebx
; сумма возвращается в eax
SUM     proc
        add     eax,ebx
        ret
SUM     endp

; вызов процедуры
            mov     eax,A
            mov     ebx,B
            call    SUM
            mov     S,eax
```

При передаче параметров через стек эти параметры перед вызовом процедуры помещаются командой PUSH в стек и командой CALL вызывается процедура. Но в процедуре к параметрам командой POP обратиться не получится, т.к. при вызове процедуры в стек был помещен адрес возврата. Поэтому в этой процедуре придется использовать регистр EBP для работы со стеком, предварительно сохранив его.

Рассмотрим вызов NEAR-процедуры с двумя параметрами:

```
push eax ; второй параметр
push edx ; первый параметр
call myproc
```

Содержимое стека после `call myproc`:

адрес возврата	←	[esp]
первый параметр	←	[esp+4]
второй параметр	←	[esp+8]

Фрагмент процедуры:

```
myproc proc
    push ebp
    mov ebp, esp
    mov eax, dword ptr [esp+8] ; первый параметр
    mov ebx, dword ptr [esp+12]; второй параметр
    …
    pop ebp
    …
myproc endp
```

В процедуре после команды `push ebp` стек выглядит так:

старый ebp	←	[esp]
адрес возврата	←	[esp+4]
первый параметр	←	[esp+8]
второй параметр	←	[esp+12]

Важно, кто очищает параметры, переданные через стек. Возможно два варианта:

1. очистку производит вызываемая процедура;
2. очистку производит вызывающая процедура.

В первом случае процедура должна заканчиваться командой

RET количество_байтов

где **количество_байтов** – суммарный размер переданных параметров. В примере процедура myproc должна использовать RET 8.

Во втором случае процедура заканчивается, как обычно, командой RET, но после команды CALL из стека извлекаются все параметры:

```
        mov eax, x
        mov edx, y
        call myproc
        pop edx
        pop eax
```

Пример 2. Процедура вычитания двух 32-битных целых. Использовать передачу параметров через стек. Стек очищает вызываемая процедура.

```
; результат возвращается в eax
DIF     proc
        push    ebp
        mov     ebp,esp

        mov     eax,dword ptr [ebp+8]
        mov     ebx,dword ptr [ebp+12]
        sub     eax,ebx
        pop     ebp
        ret     8
DIF     endp

; вызов процедуры
            push    B
            push    A
            call    DIF
            mov     result,eax     ; result = A-B
```

Пример 3. Процедура вычитания двух 8-битных целых. Использовать передачу параметров через стек. Стек очищает вызывающая процедура.

```
DIFb    proc
        push    ebp
        mov     ebp,esp

        mov     ax,word ptr [ebp+8]
        mov     bx,word ptr [ebp+10]
        sub     ax,bx
        pop     ebp
        ret
DIFb     endp

; вызов процедуры
            mov     al,bB
            cbw
            push    ax
            mov     al,bA
            cbw
            push    ax
            call    DIFb
            mov     result,al
            pop     ax
            pop     ax
```

Вопросы для самопроверки

1. Определите понятия ближний/дальний вызов.
2. Определите понятия ближняя/дальняя процедура.
3. Какие команды могут учитывать тип вызова?
4. Как оформляется процедура?
5. Какая команда используется для вызова процедуры?
6. Определите синтаксис команды выхода из процедуры. Как определяется значение необязательного параметра команды?
7. Назовите основные способы передачи параметров в процедуры?
8. В чем заключаются ограничения передачи параметров через общие регистры?
9. В чем различие передачи параметров по значению/по адресу?
10. Сколько параметров может передать процедура через стек?
11. Как передать процедуре массив?
12. Какой размер поля используется при работе со стеком?
13. Что делать, если размер переменной не совпадает с размером поля стека?

14. Как вернуть результат из процедуры?
15. Как вернуть несколько значений из процедуры?
16. Что находится в стеке после вызова процедуры с параметрами?
17. В каком порядке находятся параметры процедуры в стеке?
18. Кто должен очистить стек после завершения процедуры?
19. Какая последовательность команд может заменить команду CALL?
20. Какая последовательность команд может заменить команду RET?

6 ОБРАБОТКА СТРОК

6.1 Общие сведения

Особенности команд обработки строк:

— процессор может выполнять строковые операции над байтами, словами или двойными словами;

— для адресации строки-источника используется регистровая пара DS:ESI, а для строки-приемника – ES:EDI;

— флаг направления в регистре состояния управляет направлением обработки строки: слева направо или справа налево. Команды **CLD** и **STD** используются соответственно для установки в 0 и 1 флага направления;

— строковые команды увеличивают или уменьшают адреса операндов после выполнения операции. Если флаг направления равен 0, то адрес увеличивается, а если флаг равен 1, то уменьшается;

— команды обработки строк над байтами изменяют адрес на 1 после каждой операции, над словами – на 2, над двойными словами – на 4;

— строковые команды можно использовать с префиксами повторения **REPx**:

REP – повторять команду, пока CX≠0;
REPE/REPZ – повторять команду, пока CX≠0 или ZF≠0;
REPNE/REPNZ – повторять команду, пока CX≠0 или ZF=0.

6.2 Команды обработки строк

Команды обработки строк над байтами заканчиваются символом *B* над словами – символом *W*, над двойными словами – символом *D*.

Команды копирования имеют вид:

MOVSx приемник, источник

Содержимое источника копируется в приемник

Команды сравнения:

CMPSx приемник, источник

сравнивают приемник и источник, устанавливают флаги, в частности, ZF.

Команды сканирования:

SCASx приемник

выполняют поиск значения, содержащегося в регистре AL/AX/EAX.

Команды загрузки:

LODSx источник

выполняют загрузку элемента из строки в регистр AL/AX/EAX.

Команды сохранения:

STOSx приемник

выполняют сохранение содержимого регистра AL/AX/EAX в строку.

6.3 Использование строковых команд

Пример 1. Переслать 20 байт из первой строки во вторую. Регистры ds и es инициализированы адресом сегмента данных.

```
str1   db   20 dup('*')
str2   db   20 dup(?)
     cld                  ;сброс флага df
     mov    ecx,20        ; кол-во пересылаемых байт
     lea    edi,str2      ;адрес области "куда"
     lea    esi,str1      ;адрес области "откуда"
     rep    movsb         ;пересылка данных
```

Префикс повторения выполняет строковую команду в цикле. Команды, эквивалентные использованной выше цепочечной команде:

```
     jecxz     m_end          ;переход, если cx=0
m_loop:
     mov  al,[esi]
     mov  [edi],al
     inc  esi
     inc  edi
     loop m_loop
m_end:
```

Пример 2. Дублирование образца с использованием команды stosw. Сформировать строку вида 1010…(длина=20байтов).

```
str1   db   20 dup(?)
     mov    ax,"01"   ;обратная последовательность байтов
     cld
     mov    ecx,10
     lea    edi,str1
     rep stosw
```

Пример 3. Сравнение двух строк одинаковой длины: побайтно слева направо, операция прекращается, когда обнаружено «не равно».

```
     cld
     mov  ecx,10    ;длина строк
     lea  esi,str1
     lea  edi,str2
     repe cmpsb
     jne  not_equal
     ; обработка, если равны
     ;в ecx –количество необработанных байтов
     jmp  m_end
not_equal:
     ; обработка, если не равны
m_end:
```

Команды cmpsw и cmpsd не используются для сравнения строк символов, так как при сравнении они переставляют байты.

Пример 4. Сканирование и замена. В строке заменить первый найденный символ ‘+’ на символ ‘-‘.

```
   str1   db   '+Iv+an+Iva+nov+'
     cld
     mov  ecx,15    ;длина строки
     mov  al,'+'
     lea  edi,str1
     repne scasb    ;сканируем строку, пока не найдем
                    ;или не закончится строка
     jecxz m_not    ;символ не найден
     ; символ найден, edi указывает на следующий символ
     mov  byte ptr [edi-1],'-'
        ...
```

Пример 5. В строке заменить все символы ‘+’ на символы ‘-‘.

```
     char s1[] = "+Iv+an+Iva+nov+";
     _asm
     {
          xor ebx,ebx
          cld
          mov  ecx,15    ;количество элементов
          mov  al,'+'
          lea  edi,s1
m_beg:
          or   al,0      ;сброс zf
          repne scasb
          jz   m_replace ;zf=1 - найдено вхождение
          jecxz m_end    ;ecx=0 - вся строка просмотрена
```

```
          jmp  m_beg
m_replace:
          mov  byte ptr[edi-1],'-'
          jmp  m_beg
m_end:
     }
```

Пример 6. Вычислить количество нулевых элементов массива.

```
     int A[10] = {0, 1, 0, -2, 34, 0, 0, 7, -1, 0};
     int  result;
     _asm
     {
          cld
          xor  ebx,ebx   ;количество нулевых элементов
          mov  ecx,10    ;количество элементов
          mov  eax,0
          lea  edi,A
m_beg:
          repne scasd
          jz   m_inc     ;zf=1 - найдено вхождение
          jecxz m_end    ;ecx=0 - весь массив просмотрен
          jmp  m_beg
m_inc:
          inc  ebx
          jmp  m_beg
m_end:
          mov  result,ebx
     }
```

Пример 7. Реверсирование строки. Обработка исходной строки выполняется справа налево. Регистр esi, используемый в команде lodsb, устанавливается на последний байт исходной строки.

```
     char s1[10] = "Hello";
     char s2[10] = {0};
     _asm
     {
          mov  ecx,5     ;длина строки
          lea  esi,s1
          add  esi,ecx
          dec  esi
          lea  edi,s2
          std
m_beg:
          lodsb
          mov  byte ptr[edi],al
          inc  edi
          loop m_beg
```

```
        cld
    }
```

Пример 8. Поиск текстовой подстроки s2 в строке символов s1, длина s1 превышает длину s2. Алгоритм: последовательно просматриваются символы строки s1, сравнивается очередной символ строки s1 с первым символом строки s2, после совпадения последовательно сравниваются соответствующие элементы s1 и s2.

```
    char s1[10] = "124412375";
    char s2[10] = "4124";
    int len1 = strlen(s1), len2 = strlen(s2);
    int  result;
    _asm
    {
    cld
    mov  ecx,len1
    mov  al,s2
    lea  edi,s1
m_beg:
    repne scasb         ;поиск вхождения первого символа s2 в s1
    jecxz     m_not     ;подстрока не найдена
    push edi
    push ecx
    mov  ecx,len2
    dec  ecx
    lea  esi,s2+1
    ;сравнение s2, начиная со второго символа, с частью s1
    repe cmpsb
    jz m_found          ;подстрока найдена
m_notfound:
    pop  ecx
    pop  edi
    jmp  m_beg
m_not:
    mov  eax,0
    jmp  m_end
m_found:
    pop  ecx
    pop  edi
    mov  eax,1
m_end:
    mov  result,eax
    }
```

Вопросы для самопроверки

1. Как символы строк размещаются в оперативной памяти?

2. Как определить, что строка обрабатывается по 1/2/4 байта?

3. Как определить направление обработки строк (слева направо или наоборот)?

4. Где находятся операнды команд обработки строк?

5. Для чего используются префиксы rep/repe/repne? Как определяется длина обрабатываемой строки?

6. Как после команды сканирования определить, что искомое найдено?

Задачи

1. Сравнить две строки и вывести результат (равны или индекс первого символа, в котором они различаются).

2. В массиве целочисленных элементов заменить все элементы 0 на 1.

3. Определить, является ли строка палиндромом.

4. Строка представляет собой слова, разделенные пробелами. Вычислить количество слов.

5. Определить индекс последнего вхождения подстроки в строку.

6. Удалить из строки все символы, равные заданному.

7. Строка представляет собой слова, разделенные пробелами. Удалить лишние пробелы.

8. Из заданной строки исключить символы, расположенные между круглыми скобками.

9. Из заданного целочисленного массива удалить из каждой группы идущих подряд равных элементов все, кроме одного.

10. Найти в строке наибольшее количество цифр, идущих подряд.

11. Строка представляет собой слова, разделенные пробелами. Найти длину самого короткого слова.

12. Отсортировать слова в строке.

13. Выровнять строку по правому краю

14. Выровнять строку по центру

7 МНОГОМОДУЛЬНЫЕ ПРИЛОЖЕНИЯ

7.1 Связь C/C++–Ассемблер

Рассмотрим разработку многомодульных приложений C-Ассемблер. По-прежнему, ввод-вывод осуществляется в C-программе, из которой вызываются процедуры на Ассемблере.

Создаем консольное приложение в среде Visual C++ 2005. При добавлении к проекту (Add-Existing Item…) файла с расширением asm появляется окно, изображенное на рисунке, в котором следует выбрать исполь-

зуемый Ассемблер MASM. Если это окно не появляется, попробуйте перезапустить среду.

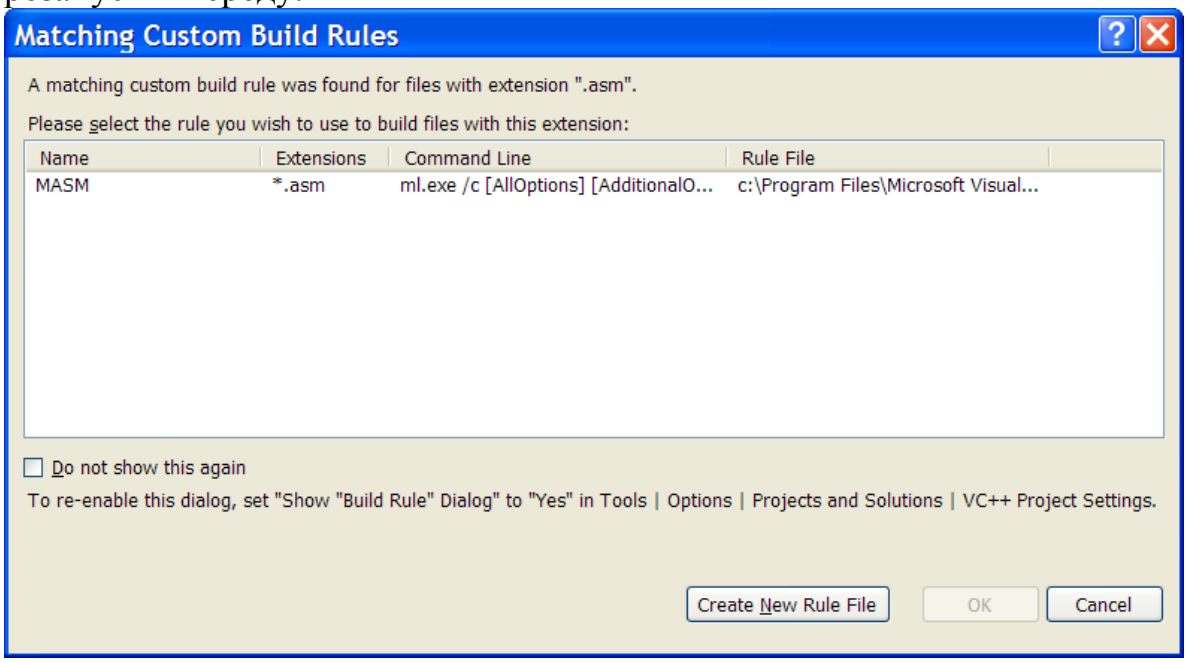

Рисунок – Окно выбора Ассемблера

При стыковке модулей C/C++ и Ассемблера следует учитывать:

– компиляторы языков C и C++ искажают имена функций;

– соглашение вызова (calling convention) определяет, как передаются параметры в подпрограмму, осуществляется возврат из подпрограммы и возвращается результат. Наиболее распространенные соглашения приведены в таблице 1.

Таблица 1 – Соглашения вызова

Название	Передача параметров	Очистка стека	Использование регистров
fastcall	слева направо	вызывающая программа	ecx, edx
stdcall	справа налево	процедура	нет
cdecl	справа налево	вызывающая программа	нет

Требования к модулю на языке C/C++ относятся к объявлению функции, реализованной на Ассемблере, и предполагают использование:

– описателя **extern "C"**;

– ключевого слова, соответствующего соглашению о вызовах, например, **__stdcall** или **__cdecl**.

Они определяют правила декорирования (искажения) имени функции и представлены в таблице 2. Используемое *Число* – количество байтов, занимаемое параметрами.

Таблица 2 – Правила декорирования

Описатель	Соглашение о вызовах	Декорирования имя
extern "C"	__fastcall	@Имя@Число
extern "C"	__stdcall	_Имя@Число
extern "C"	__cdecl	_Имя

Прототип функции, реализованной на Ассемблере, должен иметь вид:

extern "C" тип_возврата тип_соглашения имя(параметры);

По умолчанию тип соглашения __cdecl.

Требования к модулю на Ассемблере:

– имя процедуры должно быть доступно из других модулей, поэтому объявляется с директивой **PUBLIC**;

– параметры являются 32-разрядными;

– при доступе к параметрам учитывать их порядок;

– при возврате управления при необходимости очищать стек от параметров.

7.2 Примеры многомодульных приложений

Пример 1. Вычислить остаток от деления двух целых. Используется самый быстрый способ передачи параметров – регистровый, соглашение вызова – fastcall

```
#include <iostream>

extern "C"  int __fastcall Remainder(int,int);

void main()
{
     std::cout <<"remainder="<<Remainder(-12,5) << std::endl;
}
```

```
.386
PUBLIC @Remainder@8
.model flat
.code
@Remainder@8   proc
          mov  eax,ecx   ;первый параметр
          mov  ecx,edx   ;второй параметр
          cdq
          idiv ecx
          mov  eax,edx
          ret
@Remainder@8   endp
end
```

Пример 2. Процедура уменьшает в 2 раза свой аргумент. Используется cdecl по умолчанию.

```
#include <iostream>

//extern "C" int __cdecl DivideByTwo(int);
extern "C" int DivideByTwo(int);

void main()
{
     std::cout<<"DivideByTwo="<<DivideByTwo(-12)<< std::endl;
}
.386
PUBLIC _DivideByTwo
.model flat
.code
_DivideByTwo   proc
     push ebp
     mov  ebp,esp
     mov  eax,[ebp+8]
     sar  eax,1          ;арифметический сдвиг вправо
     pop  ebp
     ret
_DivideByTwo   endp
end
```

Пример 3. Процедура получает строчную латинскую букву, возвращает прописную.

```
#include <iostream>

extern "C" char __cdecl CapitalLetter(char);

void main()
{
  std::cout<<"CapitalLetter="<<CapitalLetter('w')<<std::endl;
}
.386
PUBLIC _CapitalLetter
.model flat
.code
_CapitalLetter proc
     push ebp
     mov  ebp,esp
     mov  eax,[ebp+8]
     add  eax,'A'
     sub  eax,'a'
     pop  ebp
     ret
```

```
_CapitalLetter endp
end
```

Пример 4. Процедура вычисляет индекс вхождения символа в строку. Используется соглашение вызова stdcall.

```
#include <iostream>

extern "C" int __stdcall StrIndex(char*,int,char);

void main()
{
     std::cout << "index=" <<
          StrIndex("!Hello, World!",10,'!') << std::endl;
}
```

```
.386
PUBLIC _StrIndex@12
.model flat
.code
_StrIndex@12   proc
     push ebp
     mov  ebp,esp
     mov  edi,[ebp+8]    ;адрес строки
     mov  ecx,[ebp+12]   ;длина строка
     mov  eax,[ebp+16]   ;искомый символ
     repne scasb
     jz   m_found
     mov  eax,-1
     jmp  m_end
m_found:
     mov  eax,[ebp+12]
     sub  eax,ecx
     dec  eax
m_end:
     pop  ebp
     ret  12
_StrIndex@12   endp
end
```

Пример 5. Процедура меняет местами значения своих аргументов.

```
#include <iostream>

extern "C" void __stdcall IntSwap(int&,int&);

void main()
{
     int  a=12, b=-7;
```

```
     IntSwap(a,b);
     std::cout << "a=" << a << "; b=" << b << std::endl;
}
.386
PUBLIC _IntSwap@8
.model flat
.code
_IntSwap@8     proc
     push ebp
     mov  ebp,esp
     mov  esi,[ebp+8]    ;адрес первого числа
     mov  edi,[ebp+12]   ;адрес второго числа
     mov  eax,[esi]      ;первое число
     xchg eax,[edi]
     mov  [esi],eax
     pop  ebp
     ret  8
_IntSwap@8     endp
End
```

Вопросы для самопроверки

1. В каком порядке размещаются параметры в стеке при вызове ассемблерной процедуры из C++?
2. Какое назначение описателя extern “C” при объявлении внешней процедуры?
3. Определите внешнюю процедуру. Как объявить, что процедура внешняя.
4. Что такое декорирование имен при вызове внешних процедур из С-программ?
5. Объясните назначение соглашений вызова stdcall, pascal, cdecl. Чем они различаются?
6. Как удаляются аргументы процедуры из стека по различным соглашениям?
7. Изобразите структуру стека, если для доступа к параметрам используется регистр EBP сразу после его сохранения в стеке.
8. Какие регистры можно использовать для доступа к параметрам процедуры в стеке?

Задачи

Разработать двухмодульное приложение: модуль на С осуществляет ввод/вывод, модуль на Ассемблере – вычисления.

1. Возвести целое число в натуральную степень.
2. Выполнить конкатенацию двух строк.

3. Определить начало и длину самой длинной подпоследовательности одинаковых символов.

4. Проверить, является ли строка числом в 16 с/с, если да, то получить его представление в 10 с/с.

5. Для заданного числа в римской записи получить его представление в 10 с/с.

Рекомендуемая литература

1. Assembler : учебник / В. Юров [и др.]. – СПб.: Питер, 2000. – 623 с.

2. Голубь, Н. Г. Искусство программирования на Ассемблере: лекции и упражнения/ Н. Г. Голубь – СПб.: ООО «ДиаСофтЮП», 2002. – 656 с.

3. Зубков, С. В. Assembler для DOS, Windows и UNIX для программистов/ С. В. Зубков – СПб.: Питер, 2004. – 608 с.

4. Магда, Ю. С. Ассемблер для процессоров Intel Pentium/ Ю. С. Магда – СПб.: Питер, 2006. – 410 с.

СОДЕРЖАНИЕ