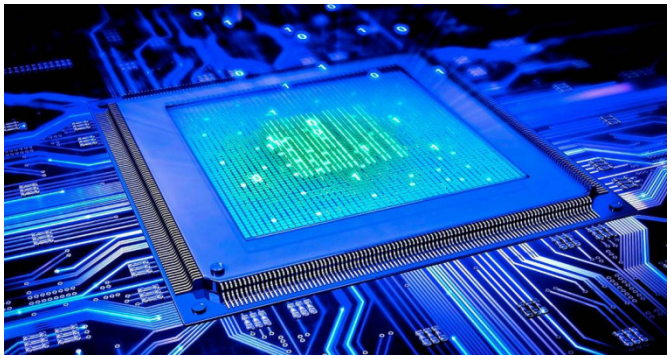


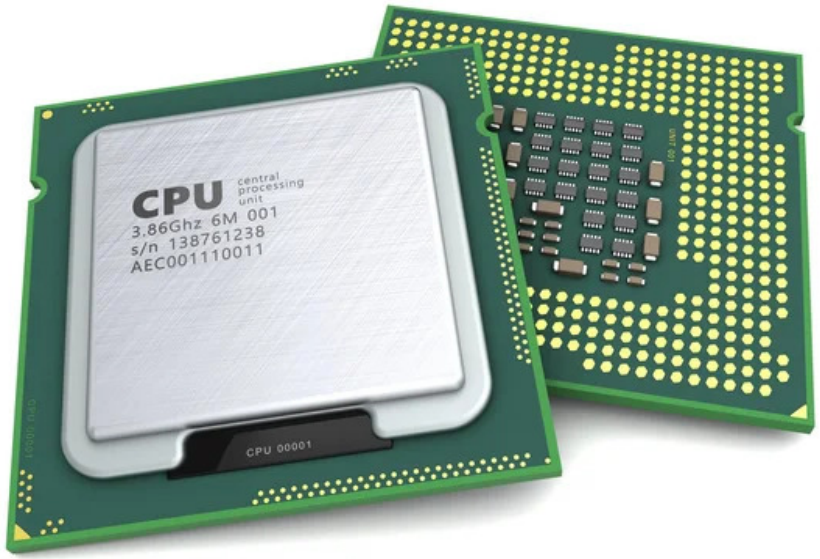
ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ ASSEMBLER

Программная модель микропроцессора, арифметические команды

Практическое пособие

для студентов специальности
1–40 01 01 «Программное обеспечение
информационных технологий»

2022



Оглавление

Тема 1. Программная модель микропроцессора. Регистры

Любая выполняемая программа получает в свое распоряжение определенный набор ресурсов микропроцессора. Эти ресурсы необходимы для выполнения и хранения в памяти команд программы, данных и информации о текущем состоянии программы и микропроцессора. Набор этих ресурсов представляет собой программную модель микропроцессора. На рисунке 2.1 представлена программная модель микропроцессора Pentium III.

Программную модель микропроцессора Intel составляют:

– пространство адресуемой памяти (для Pentium III – до 2^{36}–1 байт);

– набор регистров для хранения данных общего назначения;

– набор сегментных регистров;

– набор регистров состояния и управления;

– набор регистров устройства вычислений с плавающей точкой (сопроцессора);

– набор регистров целочисленного MMX-расширения, отображенных на регистры сопроцессора (впервые появились в архитектуре микропроцессора Pentium MMX);

– набор регистров MMX-расширения с плавающей точкой (впервые появились в архитектуре микропроцессора Pentium III);

– программный стек. Это специальная информационная структура, работа с которой предусмотрена на уровне машинных команд.

MMX-расширение микропроцессора Pentium предназначено для поддержки приложений, ориентированных на работу с большими массивами данных целого и вещественного типов, над которыми выполняются одинаковые операции. С данными такого типа обычно работают мультимедийные, графические, коммуникационные системы. Поэтому данное расширение названо MultiMedia eXtension (MMX).

MMX-расширение целочисленного устройства называется MMX-расширением.

MMX-расширение вещественного устройства называется XMM-расширением.

Наборы регистров

Регистры можно разделить на две части:

1) пользовательские регистры;

2) системные регистры.

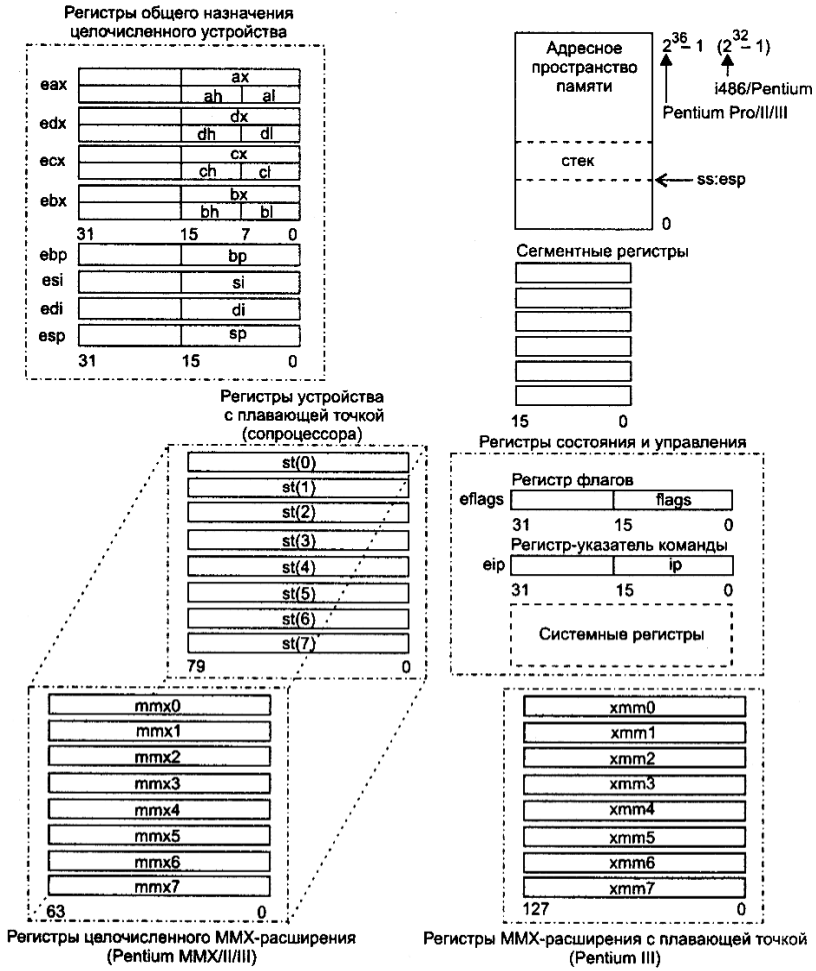


Рисунок 2.1 – Программная модель микропроцессора Intel (Pentium III)

К ***пользовательским регистрам*** относятся:
– регистры общего назначения;
– индексные регистры;
– регистры для работы со стеком;
– сегментные регистры;
– регистры сопроцессора;
– целочисленные регистры MMX-расширения;
– регистры MMX-расширения с плавающей точкой;
– регистры состояния и управления.

Регистры общего назначения. Регистры этой группы используются для хранения данных и адресов. К ним относятся:

– eax/ax/ah/al (Accumulator register) – аккумулятор. Применяется для хранения промежуточных данных;

– ebx/bx/bh/bl (Base register) – базовый регистр. Применяется для хранения базового адреса некоторого объекта в памяти;

– ecx/cx/ch/cl (Count register) – регистр-счетчик. Применяется в командах, производящих некоторые повторяющиеся действия;

– edx/dx/dh/dl (Data register) – регистр данных. Так же, как и регистр eax/ax/ah/ al, он хранит промежуточные данные.

Индексные регистры:

– esi/si (Source Index register) – индекс источника. Этот регистр в цепочечных операциях содержит текущий адрес элемента в цепочке-источнике;

– edi/di (Destination Index register) – индекс приемника (получателя). Этот регистр в цепочечных операциях содержит текущий адрес в цепочке-приемнике.

Регистры для работы со стеком:

– esp/sp (Stack Pointer register) – регистр указателя стека. Содержит указатель вершины стека в текущем сегменте стека;

– ebp/bp (Base Pointer register) – регистр указателя базы кадра стека. Предназначен для организации произвольного доступа к данным внутри стека.

Сегментные регистры. Регистры этой группы используются для хранения адресов сегментов в памяти. В программной модели микропроцессора имеется шесть сегментных регистров: cs, ss, ds, es, gs, fs. Микропроцессор аппаратно поддерживает структурную организацию программы в виде трех частей, называемых сегментами. Соответственно, такая организация памяти называется сегментной. Для того чтобы указать на сегменты, к которым программа имеет доступ в конкретный момент времени, и предназначены сегментные регистры:

– cs (code segment register) – сегментный регистр кода. Содержит команды программы;

– ds (data segment register) – сегментный регистр данных, который хранит адрес сегмента данных текущей программы;

– ss (stack segment register) – сегментный регистр стека, содержащий адрес сегмента стека;

– дополнительный сегмент данных. Неявно алгоритмы выполнения большинства машинных команд предполагают, что обрабатываемые ими данные расположены в сегменте данных, адрес которого находится в сегментном регистре ds. Если программе недостаточно одного сегмента данных, то она имеет возможность использовать еще три дополнительных сегмента данных. Но в отличие от основного сегмента данных, адрес которого содержится в сегментном регистре ds,

при использовании дополнительных сегментов данных их адреса требуется указывать явно с помощью специальных префиксов переопределения сегментов в команде. Адреса дополнительных сегментов данных должны содержаться в регистрах es, gs, fs (extension data segment registers).

Регистры сопроцессора st(0), st(1), st(2), st(3), st(4), st(5), st(6), st(7). Регистры этой группы предназначены для написания программ, использующих тип данных с плавающей точкой.

Целочисленные регистры MMX-расширения mmx0, mmx1, rnmx2, mmx3, mmx4, mmx5, mmx6, mmx7.

Регистры MMX-расширения с плавающей точкой xmm0, xmm1, xmm2, xmm3, xmm4, xmm5, xmm6, xmm7.

Регистры состояния и управления – это регистры, которые содержат информацию о состоянии микропроцессора, исполняемой программы и позволяют изменить это состояние:

– eflags/flags (flag register) – регистр флагов. Разрядность eflags/flags – 32/16 бит. Отдельные биты данного регистра имеют определенное функциональное назначение и называются флагами. Младшая часть этого регистра полностью аналогична регистру flags для i8086. На рисунке 2.2 показано содержимое регистра eflags.

Флаги регистра eflags/flags можно разделить на три группы:

1) 8 флагов состояния. Эти флаги могут изменяться после выполнения машинных команд. Флаги состояния регистра eflags отражают особенности результата исполнения арифметических или логических операций. Это дает возможность анализировать состояние вычислительного процесса и реагировать на него с помощью команд условных переходов и вызовов подпрограмм. В таблице 2.1 приведены флаги состояния и указано их назначение;

2) 1 флаг управления. Обозначается df (Directory Flag). Он находится в десятом бите регистра eflags и используется цепочечными командами. Значение флага df определяет направление поэлементной обработки в этих операциях: от начала строки к концу (df = 0) либо, наоборот, от конца строки к ее началу (df = 1). Для работы с флагом df существуют специальные команды cld (снять флаг df) и std (установить флаг df). Применение этих команд позволяет привести флаг df в соответствие с алгоритмом и обеспечить автоматическое увеличение или уменьшение счетчиков при выполнении операций со строками;

3) 8 системных, флагов, управляющих вводом/выводом, маскируемыми прерываниями, отладкой, переключением между задачами и виртуальным режимом 8086. Прикладным программам не рекомендуется модифицировать без необходимости эти флаги, так как в большинстве случаев это приведет к прерыванию работы программы.

– eip/ip (Instruction Pointer register) – указатель команд. Регистр eip/ip имеет разрядность 32/16 бит и содержит смещение следующей

подлежащей выполнению команды относительно содержимого сегментного регистра cs в текущем сегменте команд. Этот регистр непосредственно недоступен программисту, но загрузка и изменение его значения производятся различными командами управления, к которым относятся команды условных и безусловных переходов, вызова процедур и возврата из процедур. Возникновение прерываний также приводит к модификации регистра eip/ip.

Рисунок 2.2 – Содержимое регистра eflags

Таблица 2.1 – Флаги состояния

Мнемоника флага	Флаг	Номер бита в eflags	Содержание и назначение
cf	Флаг переноса (Carry Flag)	0	1 – арифметическая операция произвела перенос из старшего бита результата. Старшим является 7-й, 15-й или 31-й бит и зависимости от размерности операнда. 0 – переноса не было
pf	Флаг паритета (Parity Flag)	2	1 – 8 младших разрядов (этот флаг – только для 8 младших разрядов операнда любого размера)

Продолжение таблицы 2.1

Мнемоника флага	Флаг	Номер бита в eflags	Содержание и назначение
			результата содержат четное число единиц; 0 – 8 младших разрядов результата содержат нечетное число единиц
af	Вспомогательный флаг переноса (Auxiliary carry Flag)	4	Только для команд работающих с BCD числами. Фиксирует факт заема из младшей тетрады результата: 1 – в результате операции сложения был произведен перенос из разряда 3 в старший разряд или при вычитании был заем в разряд 3 младшей тетрады из значения в старшей тетраде; 0 – переносов и заемов в (из) 3 разряд(а) младшей тетрады результата не было
zf	Флаг нуля (Zero Flag)	6	1 – результат нулевой; 0 – результат ненулевой
sf	Флаг знака (Sign Flag)	7	Отражает состояние старшего бита результата (биты 7, 15 или 31 для 8-, 16- или 32-разрядных операндов, соответственно): 1 – старший бит результата равен 1; 0 – старший бит результата равен 0
of	Флаг переполнения (Overflow Flag)	11	Флаг of используется для фиксирования факта потери значащего бита при арифметических операциях: 1 – в результате операции происходит перенос (заем) в (из) старшего, знакового бита результата (биты 7, 15 или 31 для 8-, 16- или 32-разрядных операндов, соответственно); 0 – в результате операции не происходит переноса (заёма) в (из) старшего, знакового бита результата

Продолжение таблицы 2.1

Мнемоника флага	Флаг	Номер бита в eflags	Содержание и назначение
iopl	Уровень привилегий ввода-вывода (Input/Output Privilege Level)	12, 13	Используется в защищенном режиме работы микропроцессора для контроля доступа к командам ввода-вывода, в зависимости от привилегированности задачи
nt	Флаг вложенности задачи (Nested Task)	14	Используется в защищенном режиме работы микропроцессора для фиксации факта, что одна задача вложена в другую
tf	Флаг трассировки (Trace Flag)	8	Предназначен для организации пошаговой работы микропроцессора: 1 – микропроцессор генерирует прерывание с номером 1 после выполнения каждой машинной команды. Может использоваться при отладке программ, в частности отладчиками; 0 – обычная работа
if	Флаг прерывания (Interrupt enable Flag)	9	Предназначен для разрешения или запрещения (маскирования) аппаратных прерываний (прерываний по входу INTR): 1 – аппаратные прерывания разрешены; 0 – аппаратные прерывания запрещены
rf	Флаг возобновления (Resume Flag)	16	Используется при обработке прерываний от регистров отладки
vm	Флаг виртуального 8086 (Virtual 8086 Mode)	17	Признак работы микропроцессора в режиме виртуального 8086: 1 – процессор работает в режиме виртуального 8086; 0 – процессор работает в реальном или защищенном режиме
ac	Флаг контроля выравнивания (Alignment Check)	18	Предназначен для разрешения контроля выравнивания при обращениях к памяти.

Окончание таблицы 2.1

Мнемоника флага	Флаг	Номер бита в eflags	Содержание и назначение
vif	Флаг виртуального прерывания (Virtual Interrupt Flag)	19	При определенных условиях (одно из которых – работа микропроцессора в V-режиме) является аналогом флага if. Флаг vif используется совместно с флагом vip.
vip	Флаг отложенного виртуального прерывания (Virtual Interrupt Pending flag)	20	Устанавливается в 1 для индикации отложенного прерывания. Используется при работе в V-режиме совместно с флагом vif.
id	Флаг идентификации (IDentification flag)	21	Используется для того, чтобы показать факт поддержки микропроцессором инструкции cpuid.

Системные регистры – это регистры для поддержания различных режимов работы, сервисных функций, а также регистры, специфичные для определенной модели микропроцессора.

На схеме рисунка регистры этой группы не показаны по двум причинам: во-первых, их достаточно много, и, во-вторых, состав их может отличаться для различных моделей микропроцессора.

Многие из этих регистров приведены с наклонной разделительной чертой. Это не разные регистры – это части одного большого 32-разрядного регистра. Их можно использовать в программе как отдельные объекты.

Организация памяти

Бит является наименьшей единицей измерения памяти в компьютере. 8 бит образуют байт. Каждому байту поставлен в соответствие уникальный адрес в диапазоне от 0000 до FFFF. Любые два смежных байта образуют слово. Слово состоит из 16 бит. Байт с большим номером содержит старшие биты слова, а байт с меньшим номером – младшие.

Вся память разбивается на блоки (сегменты), каждый из которых содержит не более 2^{16} (64 КБ). Сегменты начинаются по адресам, кратным 16, то есть имеющим четыре нулевых младших бита. В любой момент времени программе доступны 4 сегмента: сегмент кода, сегмент стека, сегмент данных, дополнительный сегмент. Каждый сегмент

определяется путем размещения старших 16 бит адреса первого байта сегмента в одном из четырех сегментных регистров. Таким образом, сегментный регистр содержит адрес начала сегмента.

Обращение к байтам или словам внутри сегмента осуществляется с помощью 16-битного внутрисегментного смещения. Микропроцессор образует 20-битовый адрес байта (слова), суммируя 16-битное смещение с содержимым 16-битного сегментного регистра, к которому добавляются четыре нуля:

***Физический адрес = смещение* + 16·(*сегментный регистр*)**

Умножение на 16 соответствует добавлению к содержимому сегмента 4 младших нулевых битов.

Микропроцессор аппаратно поддерживает две модели использования оперативной памяти:

1) *сегментированную модель*. В этой модели программе выделяются непрерывные области памяти (сегменты), а сама программа может обращаться только к данным, которые находятся в этих сегментах;

2) *страничную модель* можно рассматривать как надстройку над сегментированной моделью. В случае использования этой модели оперативная память рассматривается как совокупность блоков фиксированного размера (4 Кбайт и более). Основное применение этой модели связано с организацией виртуальной памяти, что позволяет операционной системе использовать для работы программ пространство памяти большее, чем объем физической памяти. Для процессоров i486 и Pentium размер возможной виртуальной памяти может достигать 4 Тбайт.

Сегментация – механизм адресации, обеспечивающий существование нескольких независимых адресных пространств как в пределах одной задачи, так и в системе в целом для защиты задач от взаимного влияния. В основе механизма сегментации лежит понятие *сегмента*, который представляет собой независимый поддерживаемый на аппаратном уровне блок памяти.

Тема 2. Нотация языка Assembler. Программирование линейных вычислительных процессов

Любая команда языка ассемблер состоит из полей метки, мнемокода, операнда и комментария.

Формат команды:

```
[метка:] код [операнд] [;комментарий]
```

Обязательным в записи команды ассемблера является только поле мнемокода. Поля команды должны быть разделены хотя бы одним пробелом.

Метка может содержать до 31 символа и должна заканчиваться двоеточием. В имя метки могут входить:

– буквы от A до Z или от a до z;

– цифры от 0 до 9;

– специальные знаки: ? . @ _ $

Метку можно начать любым символом, кроме цифры. Если используется точка, то она должна быть только первым символом метки.

Примеры меток:

```
Metka
.cikl
Get_cikl
```

Поле операнда может содержать источник и приемник, причем ***приемник указывается первым***. Источник от приемника отделяются запятой:

```
Приемник, Источник
```

В ассемблере используются следующие константы:

1) двоичная – число, записанное в двоичной системе счисления, заканчивающееся буквой B, например: `1001010B`;

2) десятичная – число, записанное в десятичной системе счисления, заканчивающееся буквой D, например: `1234D` или `1234`;

3) шестнадцатеричная – число, записанное в шестнадцатеричной системе счисления, заканчивающееся буквой H, например: `67ADH`;

4) литерал – строка букв, цифр и других символов, заключенная в апострофы или кавычки, например: `'Ассемблер'`.

В качестве констант можно использовать и отрицательные числа. В случае десятичного числа перед ним достаточно поставить знак минус. Если число двоичное или шестнадцатеричное, то его необходимо записать в дополнительном коде.

Комментарии используются для пояснения текста программы. Для задания многострочных комментариев нужно в начале каждой строки поставить точку с запятой.

Псевдооператоры

Псевдооператоры управляют работой ассемблера, а не микропроцессора. В отличие от команд ассемблера большинство псевдооператоров не генерируют объектного кода и выполняются на шаге трансляции программы.

Формат задания псевдооператора:

```
[идентификатор] псевдооператор [операнд]
[;комменарий]
```

Обязательным является только поле псевдооператора.

Псевдооператоры определения имен

Эти псевдооператоры позволяют программисту определить символические имена для часто используемых выражений. Такие имена создаются с помощью псевдооператоров EQU и =. После присваивания выражению имени можно использовать это имя всюду, где требуется указать выражение. Определенные псевдооператором = имена можно переопределить, а определенные псевдооператором EQU нельзя. Псевдооператор EQU можно использовать как с числовыми, так и с текстовыми выражениями, а псевдооператор = только с числовыми.

Пример:

```
count equ cx
const = 512
const = 2*const+1
```

Псевдооператоры определения переменных

Поименованные элементы данных, содержимое которых может быть изменено, называются переменными. Любая переменная имеет три атрибута: сегмент, смещение и тип.

Сегмент (SEG) определяет сегмент, содержащий переменную.

Смещение (OFFSET) представляет собой расстояние или дистанцию в байтах от начала сегмента.

Тип (TYPE) определяет единицу памяти, выделяемую для хранения перемнной.

Псевдооператоры определения перенных:

```
[имя] db выражение [,…]
[имя] dw выражение [,…]
[имя] dd выражение [,…]
```

Псевдооператор db резервирует байты, dw – слова, dd – двойные слова. Выражение может быть числовым или адресным.

Пример:

```
a db 4
b dw 6
c db 3*6-2     ;числовое выражение
```

```
d db a          ;адресное выражение
```

При определении переменной без присваивания ей начального значения необходимо указать в поле выражения вопросительный знак

```
c   db  ?
```

Если имя переменной не указывается, то просто резервируется память:

```
db  ?
```

Простые типы данных:

`db` – резервирование памяти размером в 1 байт;

`dw` – резервирование памяти размером в 2 байт;

`dd` – резервирование памяти размером в 4 байт;

`df` – резервирование памяти размером в 6 байт;

`dp` – резервирование памяти размером в 6 байт;

`dq` – резервирование памяти размером в 8 байт;

`dt` – резервирование памяти размером в 10 байт.

Псевдооператоры определения сегментов

В исходной программе должны быть определены сегменты, из которых состоит программа. Может быть определено до четырех сегментов: сегмент кода, сегмент данных, сегмент стека и дополнительный сегмент.

Формат описания сегмента:

```
Имя segment [тип подгонки] [тип связи] ['класс']
…
Имя ends
```

Существуют два типа подгонки или выравнивания сегмента:

`para` – сегмент будет расположен начиная с ячейки памяти, адрес которой кратен 16. Это значит, что первый байт сегмента будет иметь нулевое смещение относительно сегментного регистра;

`byte` – располагает сегмент начиная с любого адреса.

Тип связи определяет способ, которым сегмент объединяется с другими сегментами:

`public` – вызывает объединение всех сегментов с одним и тем же именем в виде одного большого сегмента;

`at` – располагает сегмент по заданному абсолютному адресу.

Класс определяет категорию сегмента:

`code` – сегмент кода;

`data` – сегмент данных;

`extra` – дополнительный сегмент;

`stack` – сегмент стека.

Псевдооператор `segment` не сообщает ассемблеру, какого рода сегмент должен быть определен. Для этой цели служит псевдооператор `assume`.

Формат оператора:

```
assume сегментный_регистр:имя_сегмента[,…]
```

Оператор `assume` связывает каждый сегментный регистр с сегментом, который сегментный регистр адресует в данный момент.

Командой `assume` нельзя загрузить адрес начала сегмента в соответствующий сегментный регистр. Это осуществляется через регистр общего назначения.

Оператор `assume` помещается сразу же за оператором `segment`, определяющим сегмент кода:

```
cseg segment para public 'code'
    assume ds:dseg, cs:cseg, ss:sseg
    mov ax,dseg
    mov ds,ax
    ...
cseg ends
```

Псевдооператоры определения процедур

Формат описания процедуры:

```
имя_процедуры proc атрибут_дистанции
    ...
    ret
имя_процедуры endp
```

Каждая процедура должна начинаться оператором `proc` и заканчиваться оператором `endp`. Процедура должна содержать команду возврата из процедуры `ret`.

Атрибуты дистанции:

`far` – дальняя процедура,

`near` – близкая процедура.

Процедура с атрибутом `near` может быть вызвана только из того сегмента команд, в котором она определена.

Формат вызова процедуры:

```
call имя_процедуры
```

Псевдооператоры управления трансляцией

Псевдооператор `end` отмечает конец исходной программы и указывает ассемблеру, где завершить трансляцию. Псевдооператор `end` должен присутствовать в каждой программе.

Формат оператора:

```
end [метка точки входа]
```

где метка определяет исходную программу. Например, оператор

```
end proc
```

отмечает конец программы `proc`.

Команды пересылки

Формат команды:

```
mov приемник, источник    ;И→П
```

Команда позволяет пересылать байт или слово между регистром и ячейкой памяти или между двумя регистрами:

```
mov регистр1, регистр2
mov регистр, память
```

В команде `mov` запрещается пересылка из одной ячейки памяти в другую.

```
mov память1, память2 ;НЕЛЬЗЯ!
```

Нельзя загрузить непосредственно адресуемый операнд в регистр сегмента. Это можно сделать только через регистр общего назначения

```
mov ds,dseg ;НЕЛЬЗЯ!

mov ax,dseg ;НУЖНО!
mov ds,ax
```

Нельзя пересылать значение одного регистра сегмента в другой регистр сегмента. Подобная пересылка делается через регистр общего назначения.

```
mov ds,es ;НЕЛЬЗЯ!

mov ax,es ;НУЖНО!
mov ds,ax
```

Нельзя использовать регистр CS в качестве приемника в команде пересылки.

Для работы со стеком используются следующие команды:

```
push источник ;поместить слово в стек
pop приемник  ;извлечь слово из стека
```

Арифметические команды

Команды сложения

Формат команды:

```
add приемник, источник ;П+И→П
```

Команда `add` используется для сложения приемника и источника, результат помещается в приемник.

Оба операнда не могут быть именами переменных. Источником может быть число.

Можно складывать:

ПАМЯТЬ+РЕГИСТР
РЕГИСТР+ПАМЯТЬ
РЕГИСТР +РЕГИСТР

ПАМЯТЬ+ЧИСЛО
РЕГИСТР+ЧИСЛО

Нельзя складывать ПАМЯТЬ + ПАМЯТЬ

```
inc приемник ;П+1→П
```

Команда `inc` используется для увеличения содержимого приемника на единицу. Приемник – это регистр или ячейка памяти.

Команды вычитания

Формат команды:

```
sub приемник, источник ;П–И→П
```

Команда `sub` используется для вычитания содержимого источника из содержимого приемника, результат помещается в приемник.

Оба операнда не могут быть именами переменных. Источником может быть число.

Можно вычитать:

ПАМЯТЬ – РЕГИСТР
РЕГИСТР – ПАМЯТЬ
РЕГИСТР – РЕГИСТР
ПАМЯТЬ – ЧИСЛО
РЕГИСТР – ЧИСЛО

Нельзя вычитать ПАМЯТЬ – ПАМЯТЬ

```
dec приемник ;П–1→П
```

Команда `dec` используется для уменьшения содержимого приемника на единицу. Приемник – это регистр или ячейка памяти.

Команды умножения

Формат команды умножения чисел без знака:

```
mul источник  ;al*И→ax  при работе с байтами
              ;ax*И→ax-dx  при работе со словами
```

Формат команды умножения чисел со знаком:

```
imul источник ;al*И→ax     при работе с байтами
              ;ax*И→ax-dx  при работе со словами
```

Команды `mul` и `imul` умножают содержимое регистра `al` или `ax` (в зависимости от размерности операндов) на содержимое источника, указанного в команде умножения. Источник – это или регистр или ячейка памяти. В результате работы команд умножения с данными длиной байт регистр `al` расширяется до `ax`, а с данными длиной слово – регистр `ax` расширяется до `dx`.

***НЕЛЬЗЯ использовать в команде умножения в качества источника непосредственное значение*!**

Команды деления

Формат команды деления чисел без знака:

```
div источник  ;al/И→al-ah   при работе с байтами
              ;ax/И→ax-dx   при работе со словами
              ;al – частное, ah – остаток
              ;ax – частное, dx – остаток
```

Формат команды деления чисел со знаком:

```
idiv источник ;al/И→al-ah   при работе с байтами
              ;ax/И→ax-dx   при работе со словами
              ;al – частное, ah – остаток
              ;ax – частное, dx – остаток
```

Команды `div` и `idiv` делят содержимое регистра `al` или `ax` (в зависимости от размерности операндов) на содержимое источника, указанного в команде умножения. Источник – это или регистр или ячейка памяти. В результате работы команд умножения с данными длиной байт в регистр `al` помещается частное, в регистр `ah` – остаток, а с данными длиной слово – в регистр `ax` помещается частное, в регистр `dx` – остаток.

***НЕЛЬЗЯ использовать в команде деления в качества источника непосредственное значение*!**

Команды расширения знака

Форматы команд:

```
cbw ;расширить байт до слова
cwd ;расширить слово до двойного слова
```

Команда `cbw` воспроизводит 7 бит регистра AL во всех битах регистра AH.

Команда `cwd` воспроизводит 15 бит регистра AX во всех битах регистра DX.

Примеры использования арифметических команд

Вычислить значение выражения $y = \frac{7(a+7b)-3}{4}$.

```
;Программа работы с байтами
dseg segment para public 'data'
     a db 3
     b db 2
     y db ?
mes db 'конец программы$'
dseg ends
```



```
sseg segment para stack 'stack'
     db 30 dup (0)
sseg ends
cseg segment para public 'code'
   osn proc near
   assume cs:cseg,ds:dseg,ss:sseg
   mov ax,dseg
   mov ds,ax
   mov bl,4       ;bl=4
   mov al,7       ;al=7
   imul b         ;al=7*b
   add al,a       ;al=a+7*b
   mov cl,7       ;cl=7
   imul cl        ;al=7*(a+7*b)
   sub al,3       ;al=7*(a+7*b)-3
   cbw
   idiv bl        ;al=(7*(a+7*b)-3)/4 остаток в ah
   mov y,al       ;y=al
   lea dx,mes ;вывод сообщения 'конец программы'
   mov ax,0900H
   int 21H
   mov ax,4C00H ;завершение программы
   int 21H
   osn endp
cseg ends
end osn
```

```
;Программа работы со словами
dseg segment para public 'data'
     a dw 3
     b dw 2
     y dw ?
mes db 'конец программы$'
dseg ends
sseg segment para stack 'stack'
     db 30 dup (0)
sseg ends
cseg segment para public 'code'
   osn proc near
   assume cs:cseg,ds:dseg,ss:sseg
   mov ax,dseg
   mov ds,ax
   mov bx,4       ;bx=4
   mov ax,7       ;ax=7
   imul b         ;ax=7*b
```

```
   add ax,a        ;ax=a+7*b
   mov cx,7        ;cx=7
   imul cx         ;ax=7*(a+7*b)
   sub ax,3        ;ax=7*(a+7*b)-3
   cwd
   idiv bx         ;ax=(7*(a+7*b)-3)/4 остаток в dx
   mov y,ax        ;y=ax
   lea dx,mes    ;вывод сообщения 'конец программы'
   mov ax,0900H
   int 21H
   mov ax,4C00H   ;завершение программы
   int 21H
   osn endp
cseg ends
end osn
```

Порядок работы с программой на ассемблере

Все действия с программой на ассемблере могут быть представлены в виде схемы, предсталенной на рисунке 3.1.

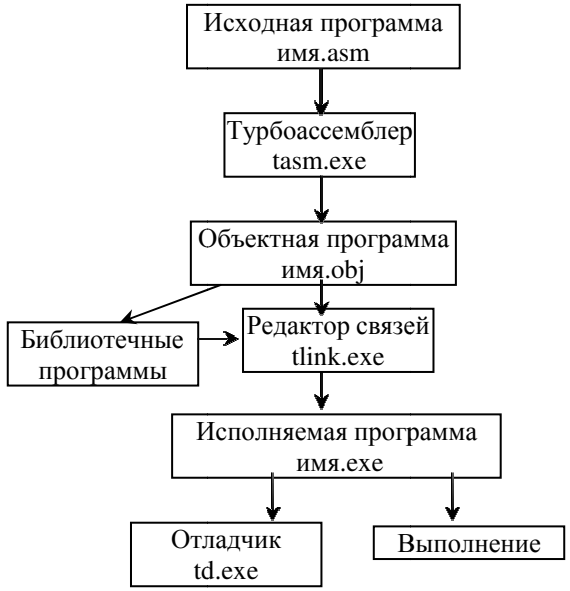


Рисунок 3.1 – Порядок работы с программой на ассемблере

Порядок работы с программой:

1. Исходный текст программы можно набрать, используя любой тектовый редактор.

2. Трансляция программы осуществляется с помощью транслятора ассемблера tasm.exe

`tasm.exe имя.asm >>1.txt`

Результатом трансляции является файл объектной программы имя.obj.

3. Создание исполняемого файла осуществляется с помощью редактора связей (загрузчика)

`tlink.exe имя.obj`

Редактор связей – это программа, осуществляющая связь объектных программ, оттранслированных с помощью ассемблера. Редактор связей объединяет все программы в один загрузочный модуль, готовый к выполнению. Результатом загрузки является файл загрузочного модуля имя.exe.

4. Исполнение программы возможно в среде MS DOS или с помощью отладчика:

`td.exe имя.exe`

Отладчик позволяет проверить работу программы.

Сокращенные директивы сегментации

Изложенные ранее директивы сегментации называются стандартными директивами сегментации.

Для простых программ, содержащих по одному сегменту кода, данных и стека, можно упростить их написание. Трансляторы `masm` и `tasm` ввели возможность использования упрощенных директив сегментации. Чтобы управлять в этом случае размещением сегментов, необходимо указывать директиву `model`, которая стала управлять размещением сегментов и выполнять функции директивы `assume` (поэтому при использовании упрощенных директив сегментации директиву `assume` можно не использовать). Эта директива связывает сегменты, которые, в случае использования упрощенных директив сегментации, имеют предопределенные имена с сегментными регистрами.

Структура программы с использованием упрощенных директив сегментации:

```
masm
model small    ;модель памяти
;описание сегмента данных
.data
   a db 3
```

```
   b db 2
   y db ?
mes db 'конец программы$'
;описание сегмента стека
.stack
   db 256 dup (0)
;описание сегмента кода
.code
   osn proc near  ; начало основной процедуры
   mov ax,@data   ; заносим адрес сегмента данных
                  ; в регистр ax
   mov ds,ax      ; ax в ds
;тело программы
   mov bl,4       ;bl=4
   mov al,7       ;al=7
   imul b         ;al=7*b
   add al,a       ;al=a+7*b
   mov cl,7       ;cl=7
   imul cl        ;al=7*(a+7*b)
   sub al,3       ;al=7*(a+7*b)-3
   cbw
   idiv bl        ;al=(7*(a+7*b)-3)/4 остаток в ah
   mov y,al       ;y=al
   lea dx,mes ;вывод сообщения 'конец программы'
   mov ax,0900H
   int 21H
;завершение программы
   mov ax,4C00H
   int 21H
   osn endp
end osn
```

Пакет макроассемблера masm позволяет задавать макроопределения (или макросы), представляющие собой именованные группы команд. Они обладают тем свойством, что их можно вставлять в программу в любом месте, указав только имя группы в месте вставки. Режим работы masm поддерживает все основные возможности макроассемблера.

Совместно с упрощенными директивами сегментации используется директива указания модели памяти model, которая управляет размещением сегментов и выполняет функции директивы assume.

Формат model:

model модификатор Модель памяти

Обязательным параметром директивы model является модель памяти. Этот параметр определяет модель сегментации памяти для программного модуля, т. е. набор сегментов программы, размер сегментов данных и кода, способ связывания сегментов и сегментных регистров.

Таблица 3.1 – Модели памяти

Модель	Тип кода	Тип данных	Назначение модели
tiny	near	near	Код и данные объединены в одну группу с именем DGROUP. Используется для создания программ формата .com
small	near	near	Код занимает один сегмент, данные объединены в одну группу с именем DGROUP.
medium	far	near	Код занимает несколько сегментов, по одному на каждый объединяемый программный модуль. Все ссылки на передачу управления типа far. Данные объединены в одной группе, все ссылки на них – типа near
compact	near	far	Код в одном сегменте, ссылки на данные – типа far
large	far	far	Код в нескольких сегментах, по одному на каждый объединяемый программный модуль
flat	near	near	Код и данные в одном 32-битном сегменте (плоская модель памяти)

Упрощенные директивы определения сегментов режима masm:

.code[размер] – начало или продолжение сегмента кода;

.data – начало или продолжение сегмента инициализированных данных;

.const – начало или продолжение сегмента постоянных данных (констант);

.data? – начало или продолжение сегмента неинициализированных данных;

.stack [размер] – начало или продолжение сегмента стека;

.fardata [имя] – начало или продолжение сегмента инициализированных данных типа far;

.fardata? [имя] – начало или продолжение сегмента неинициализированных данных типа far.

Модификаторы:

`use16` – сегменты используются как 16-битные;

`use32` – сегменты используются как 32-битные;

`dos` – программа будет работать в MS-DOS.

При использовании директивы `model` транслятор делает доступными несколько идентификаторов, к которым можно обращаться во время работы программы:

`@code` – физический адрес сегмента кода;

`@data` – физический адрес сегмента данных типа `near`;

`@fardata` – физический адрес сегмента данных типа `far`;

`@fardata?` – физический адрес сегмента неинициализированных данных типа `far`;

`@curseg` – физический адрес сегмента неинициализированных данных типа `far`;

`@stack` – физический адрес сегмента стека.

Литература

1 Гук, М. Процессоры Pentium III, Athlon и другие / М. Гук,В. Юров. – СПб. : Питер, 2000. – 379 с.

2 Зубков, С. В. Ассемблер для DOS, Windows и UNIX / С. В. Зубков. – М. : ДМК Пресс, 2000. – 534 с.

3 Программирование на языке ассемблера для персональных ЭВМ : учебное пособие / А. Ф. Каморников [и др.]. – Гомель : ГГУ, 1995. – 95 с.4

Пустоваров, В. И. Ассемблер : программирование и анализ корректности машинных программ / В. И. Пустоваров. – К. : Издательская группа BHV, 2000. – 480 с.

5 Сван, Т. Освоение Turbo Assembler / Т. Сван. – Киев : Диалектика, 1996. – 540 с.

6 Юров, В. Assembler / В. Юров. – СПб. : Питер, 2001. – 624 с.

7 Юров, В. Assembler : практикум / В. Юров. – СПб. : Питер, 2002. – 400 с.

8 Юров, В. Assembler : специальный справочник / В. Юров. – СПб. : Питер, 2000. – 496 с.