

Министерство образования Республики Беларусь

Учреждение образования  
«Гомельский государственный университет  
имени Франциска Скорины»

**Е. А. РУЖИЦКАЯ**

# **ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ ASSEMBLER**

## **Логические команды, макросредства**

Практическое пособие

для студентов специальностей  
1–40 01 01 «Программное обеспечение  
информационных технологий»,  
1–40 04 01 «Информатика и технологии программирования»

Гомель  
ГГУ им. Ф. Скорины  
2017

УДК 004.431.4(076)  
ББК 32.973.21я73  
Р837

Рецензенты:

кандидат физико-математических наук Т. В. Тихоненко,  
кандидат технических наук С. Ф. Маслович

Рекомендовано к изданию научно-методическим советом  
учреждения образования «Гомельский государственный  
университет имени Франциска Скорины»

**Ружицкая, Е. А.**

**Р837** Программирование на языке Assembler : логические команды, макросредства : практическое пособие / Е. А. Ружицкая ; М-во образования Республики Беларусь, Гомельский гос. ун-т им. Ф. Скорины. – Гомель: ГГУ им. Ф. Скорины, 2017. – 47 с.  
ISBN 978-985-577-259-1

Практическое пособие предназначено для оказания помощи студентам в овладении машинно-ориентированным языком программирования Assembler. Излагается теоретический материал, даны практические задания и примеры их выполнения по логическим командам и макросредствам языка Assembler.

Адресовано студентам 1 курса специальностей 1–40 01 01 «Программное обеспечение информационных технологий», 1–40 04 01 «Информатика и технологии программирования».

**УДК 004.431.4(076)**  
**ББК 32.973.21я73**

**ISBN 978-985-577-259-1** © Ружицкая Е. А., 2017  
© Учреждение образования «Гомельский государственный университет имени Франциска Скорины», 2017

## Оглавление

|                                                                  |    |
|------------------------------------------------------------------|----|
| Предисловие . . . . .                                            | 4  |
| Тема 1. Логические команды . . . . .                             | 5  |
| 1.1 Логические данные . . . . .                                  | 6  |
| 1.2 Логические команды. . . . .                                  | 6  |
| 1.3 Команды сдвига. . . . .                                      | 9  |
| 1.3.1 Линейный сдвиг. . . . .                                    | 10 |
| 1.3.2 Циклический сдвиг. . . . .                                 | 12 |
| 1.4 Дополнительные команды сдвига . . . . .                      | 13 |
| 1.5 Практическое задание . . . . .                               | 14 |
| Тема 2. Макросредства языка ассемблера . . . . .                 | 23 |
| 2.1 Псевдооператоры <code>equ</code> и <code>=</code> . . . . .  | 24 |
| 2.2 Макрокоманды . . . . .                                       | 25 |
| 2.3 Макродирективы . . . . .                                     | 30 |
| 2.3.1 Директивы <code>WHILE</code> и <code>REPT</code> . . . . . | 31 |
| 2.3.2 Директива <code>IRP</code> . . . . .                       | 32 |
| 2.3.3 Директива <code>IRPC</code> . . . . .                      | 33 |
| 2.3.4 Директивы условной компиляции . . . . .                    | 33 |
| 2.3.5 Директивы генерации ошибок . . . . .                       | 40 |
| 2.3.6 Дополнительные средства управления трансляцией . . . . .   | 43 |
| 2.4 Практическое задание . . . . .                               | 43 |
| Литература . . . . .                                             | 47 |

## Предисловие

Минимально адресуемая единица данных в процессоре – байт. Логические команды позволяют манипулировать отдельными битами. Работа на битовом уровне дает возможность в отдельных случаях существенно сэкономить память, особенно при моделировании различных массивов, содержащих одноразрядные флаги или переключатели.

С помощью команд сдвига можно выполнять быстрое умножение и деление операндов на степени двойки, а также эффективно преобразовывать данные. Применение команд циклического сдвига и сдвига двойной точности позволяет реализовать максимально быстрые операции по рассогласованию, перемещению, вставке и извлечению битовых подстрок.

Преимущества языка ассемблера связаны, в частности, с макросредствами. Макросредства – это основные инструменты модификации текста программы на этапе ее трансляции. Принцип работы макросредств основан на препроцессорной обработке, которая заключается в том, что текст, поступающий на вход транслятора, перед компиляцией подвергается преобразованию и может значительно отличаться от синтаксически правильного текста, воспринимаемого компилятором. Роль препроцессора в трансляторе TASM выполняет макрогенератор.

Макрокоманда – строка в исходной программе, которой соответствует специальный блок – макроопределение. Макрокоманда может иметь аргументы, с помощью которых можно изменять текст макроопределения. Макрогенератор, встречая макрокоманду в тексте программы, корректирует текст соответствующего макроопределения, исходя из аргументов этой макрокоманды, и вставляет его в текст программы вместо данной макрокоманды. Процесс такого замещения называется макрогенерацией.

Практическое пособие предназначено для оказания помощи студентам в овладении машинно-ориентированным языком программирования *Assembler*. Излагается теоретический материал, даны практические задания и примеры их выполнения по логическим командам и макросредствам языка *Assembler*.

Язык программирования *Assembler* изучается студентами 1 курса специальностей 1–40 01 01 «Программное обеспечение информационных технологий» в рамках дисциплины «Языки программирования» и 1–40 04 01 «Информатика и технологии программирования» в курсе «Программирование».

## Тема 1. Логические команды

Под *логическими* понимаются такие преобразования данных, в основе которых лежат правила *формальной логики*. Формальная логика работает на уровне утверждений истинно и ложно. Для микропроцессора это означает 1 и 0, соответственно.

К средствам логического преобразования данных относятся *логические команды* и *логические операции*. На рисунке 1.1 показаны средства микропроцессора для организации работы с данными по правилам формальной логики. Они разбиты на две группы: команды и операции. Операнд команды ассемблера в общем случае может представлять собой выражение, которое является комбинацией операторов и операндов. Среди этих операторов могут быть и операторы, реализующие логические операции над объектами выражения.

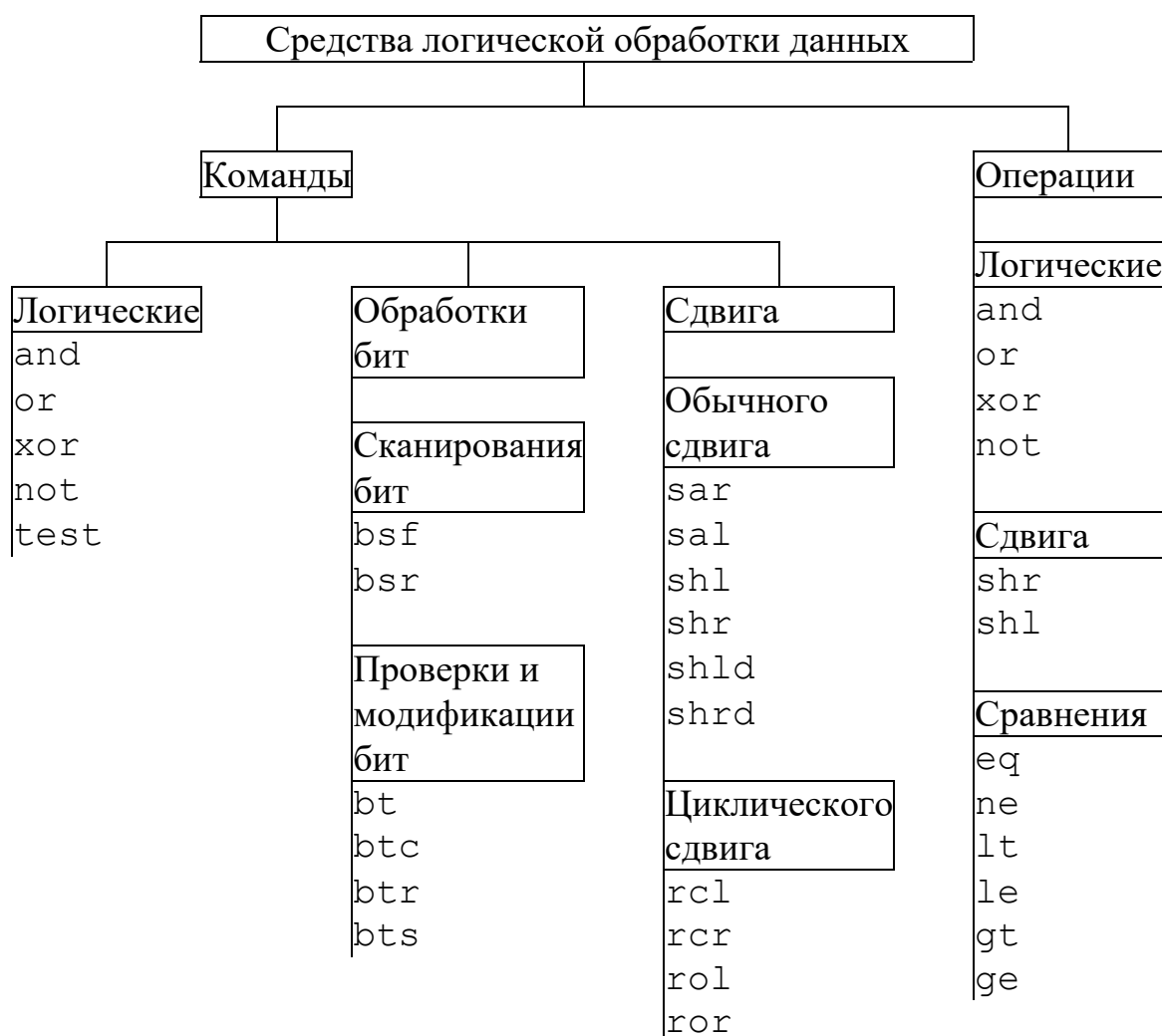


Рисунок 1.1 – Средства микропроцессора для работы с логическими данными

## 1.1 Логические данные

Теоретической базой для логической обработки данных является формальная логика. Существует несколько систем логики. Одна из наиболее известных – это исчисление высказываний. Высказывание – это любое утверждение, о котором можно сказать, что оно либо истинно, либо ложно. Исчисление высказываний представляет собой совокупность правил, используемых для определения истинности или ложности некоторой комбинации высказываний.

Над высказываниями (битами) могут выполняться следующие логические операции:

- *отрицание* (логическое НЕ) – логическая операция над одним операндом, результатом которой является величина, обратная значению исходного операнда;

- *логическое сложение* (логическое включающее ИЛИ) – логическая операция над двумя операндами, результатом которой является «истина» (1), если один или оба операнда имеют значение «истина» (1), и «ложь» (0), если оба операнда имеют значение «ложь» (0). Эта операция описывается с помощью следующей таблицы истинности:

0 или 0 = 0      0 или 1 = 1      1 или 0 = 1      1 или 1 = 1;

- *логическое умножение* (логическое И) – логическая операция над двумя операндами, результатом которой является «истина» (1) только в том случае, если оба операнда имеют значение «истина» (1). Во всех остальных случаях значение операции – «ложь» (0). Эта операция описывается с помощью следующей таблицы истинности:

0 и 0 = 0      0 и 1 = 0      1 и 0 = 0      1 и 1 = 1;

- *логическое исключающее сложение* (логическое исключающее ИЛИ) – логическая операция над двумя операндами, результатом которой является «истина» (1), если только один из двух операндов имеет значение «истина» (1), и ложь (0), если оба операнда имеют значения «ложь» (0) или «истина» (1). Эта операция описывается с помощью следующей таблицы истинности:

0 или 0 = 0      0 или 1 = 1      1 или 0 = 1      1 или 1 = 0.

Система команд микропроцессора содержит пять команд, поддерживающих данные операции. Эти команды выполняют логические операции над битами операндов. Размерность операндов должна быть одинакова.

## 1.2 Логические команды

В системе команд микропроцессора есть следующий набор команд, поддерживающих работу с логическими данными:

**and операнд\_1, операнд\_2** – операция логического умножения. Команда выполняет поразрядно логическую операцию И (конъюнкцию)

над битами операндов операнд\_1 и операнд\_2. Результат записывается на место операнд\_1.

**or операнд\_1, операнд\_2** – операция логического сложения. Команда выполняет поразрядно логическую операцию ИЛИ (дизъюнкцию) над битами операндов операнд\_1 и операнд\_2. Результат записывается на место операнд\_1.

**xor операнд\_1, операнд\_2** – операция логического исключающего сложения. Команда выполняет поразрядно логическую операцию исключающего ИЛИ над битами операндов операнд\_1 и операнд\_2. Результат записывается на место операнд\_1.

**test операнд\_1, операнд\_2** – операция «проверить» (способом логического умножения). Команда выполняет поразрядно логическую операцию И над битами операндов операнд\_1 и операнд\_2. Состояние операндов остается прежним, изменяются только флаги *zf*, *sf*, и *pf*, что дает возможность анализировать состояние отдельных битов операнда без изменения их состояния.

**not операнд** – операция логического отрицания. Команда выполняет поразрядное инвертирование (замену значения на обратное) каждого бита операнда. Результат записывается на место операнда.

С помощью логических команд возможно выделение отдельных битов в операнде с целью их установки, сброса, инвертирования или проверки на определенное значение. Для организации такой работы с битами операнд\_2 обычно играет роль маски. С помощью установленных в 1 битов этой маски определяются нужные для конкретной операции биты операнд\_1.

Для установки определенных разрядов (бит) в 1 применяется команда

**or операнд\_1, операнд\_2**

В этой команде операнд\_2, выполняющий роль маски, должен содержать единичные биты на месте тех разрядов, которые должны быть установлены в 1 в операнд\_1:

**or eax, 10b** ;установить 1-й бит в регистре *eax*

Для сброса определенных разрядов (битов) в 0 применяется команда

**and операнд\_1, операнд\_2**

В этой команде операнд\_2, выполняющий роль маски, должен содержать нулевые биты на месте тех разрядов, которые должны быть установлены в 0 в операнд\_1

**and eax, 0fffffffh** ;сбросить в 0 1-й бит в регистре *eax*

Команда

**xor операнд\_1, операнд\_2**

применяется:

– для выяснения того, какие биты в операнд\_1 и операнд\_2 различаются;

– для инвертирования состояния заданных бит в операнд\_1.

Нужные биты маски (операнд\_2) при выполнении команды `xor` должны быть единичными, остальные – нулевыми.

```
xor eax, 10b      ;инвертировать 1-й бит в регистре eax  
jz mes           ;переход, если 1-й бит в al был единичный
```

Для проверки состояния заданных бит применяется команда

```
test операнд_1, операнд_2 ;проверить операнд_1
```

Проверяемые биты операнд\_1 в маске (операнд\_2) должны иметь единичное значение. Алгоритм работы команды `test` аналогичен алгоритму команды `and`, но он не меняет значения операнд\_1. Результатом команды является установка значения флага нуля `zf`:

– если  $zf = 0$ , то в результате логического умножения получился нулевой результат, то есть один единичный бит маски не совпал с соответствующим единичным битом операнд\_1;

– если  $zf = 1$ , то в результате логического умножения получился ненулевой результат, то есть хотя бы один единичный бит маски совпал с соответствующим единичным битом операнд\_1.

```
test eax, 00000010h  
jz ml           ;переход если 4-й бит равен 1
```

Как видно из примера, для реакции на результат команды `test` целесообразно использовать команду перехода `jnz метка` (Jump if Not Zero) – переход, если флаг нуля `zf` ненулевой, или команду с обратным действием – `jz метка` (Jump if Zero) – переход, если флаг нуля  $zf = 0$ .

Следующие две команды позволяют осуществить поиск первого установленного в 1 бита операнда. Поиск можно произвести как с начала, так и от конца операнда:

**bsf операнд\_1, операнд\_2** (Bit Scanning Forward) – сканирование бит вперед. Команда просматривает (сканирует) биты операнд\_2 от младшего к старшему (от бита 0 до старшего бита) в поисках первого бита, установленного в 1. Если таковой обнаруживается, в операнд\_1 заносится номер этого бита в виде целочисленного значения. Если все биты операнд\_2 равны 0, то флаг нуля `zf` устанавливается в 1, в противном случае флаг `zf` сбрасывается в 0.

```
mov ax, 0002h  
bsf bx, ax      ;bx = 1  
jz ml           ;переход, если ax = 0000h
```

**bsr операнд\_1, операнд\_2** (Bit Scanning Reset) – сканирование битов в обратном порядке. Команда просматривает (сканирует) биты операнд\_2 от старшего к младшему (от старшего бита к биту 0) в поисках первого бита, установленного в 1. Если таковой обнаруживается, в операнд\_1 заносится номер этого бита в виде целочисленного значения.



Позиция первого единичного бита слева отсчитывается все равно относительно бита 0. Если все биты операнд\_2 равны 0, то флаг нуля *zf* устанавливается в 1, в противном случае флаг *zf* сбрасывается в 0.

В последних моделях микропроцессоров Intel в группе логических команд появилось еще несколько команд, которые позволяют осуществить доступ к одному конкретному биту операнда. Операнд может находиться как в памяти, так и в регистре общего назначения. Положение бита задается смещением бита относительно младшего бита операнда. Значение смещения может задаваться как в виде непосредственного значения, так и содержаться в регистре общего назначения. В качестве значения смещения можно использовать результаты работы команд *bsr* и *bsf*. Все команды присваивают значение выбранного бита флагу *cf*.

**bt операнд, смещение\_бита** (Bit Test) – проверка бита. Команда переносит значение бита во флаг *cf*.

```
bt ax,5           ;проверить значение бита 5
jnc m1            ;переход, если бит = 0
```

**bts операнд, смещение\_бита** (Bit Test and Set) – проверка и установка бита. Команда переносит значение бита во флаг *cf* и затем устанавливает проверяемый бит в 1.

```
mov ax,10
bts pole,ax       ;проверить и установить 10-й бит в pole
jc m1             ;переход, если проверяемый бит равен 1
```

**btr операнд, смещение\_бита** (Bit Test and Reset) – проверка и сброс бита. Команда переносит значение бита во флаг *cf* и затем устанавливает этот бит в 0.

**btc операнд, смещение\_бита** (Bit Test and Convert) – проверка и инвертирование бита. Команда переносит значение бита во флаг *cf* и затем инвертирует значение этого бита.

### 1.3 Команды сдвига

Команды этой группы также обеспечивают манипуляции над отдельными битами операндов. Все команды сдвига перемещают биты в поле операнда влево или вправо, в зависимости от кода операции, и имеют одинаковую структуру:

#### **КОП операнд, счетчик сдвигов**

Количество сдвигаемых разрядов, счетчик\_сдвигов, располагается, на месте второго операнда и может задаваться двумя способами:

- статически: предполагает задание фиксированного значения с помощью непосредственного операнда;
- динамически: занесением значения счетчика\_сдвигов в регистр *cl* перед выполнением команды сдвига.

Все команды сдвига устанавливают флаг переноса *cf*. По мере сдвига битов за пределы операнда они сначала попадают во флаг переноса, устанавливая его равным значению очередного бита, оказавшегося за пределами операнда. Куда этот бит попадет дальше, зависит от типа команды сдвига и алгоритма программы.

По принципу действия команды сдвига можно разделить на два типа:

- команды линейного сдвига,
- команды циклического сдвига.

### 1.3.1 Линейный сдвиг

К командам этого типа относятся команды, осуществляющие сдвиг по следующему алгоритму:

- очередной «выдвигаемый» бит устанавливает флаг *cf*;
- бит, вводимый в операнд с другого конца, имеет значение 0;
- при сдвиге очередного бита он переходит во флаг *cf*, при этом значение предыдущего сдвинутого бита *теряется!*

Команды линейного сдвига делятся на два подтипа:

- команды логического линейного сдвига (рисунки 1.2, 1.3);
- команды арифметического линейного сдвига (рисунки 1.4, 1.5).

К командам *логического линейного сдвига* относятся следующие:

**shl операнд, счетчик\_сдвигов** (Shift Logical Left) – логический сдвиг влево. Содержимое операнда сдвигается влево на количество битов, определяемое значением счетчик\_сдвигов. Справа (в позицию младшего бита) вписываются нули.

**shr операнд, счетчик\_сдвигов** (Shift Logical Right) – логический сдвиг вправо. Содержимое операнда сдвигается вправо на количество битов, определяемое значением счетчик\_сдвигов. Слева (в позицию старшего, знакового бита) вписываются нули.

Команды арифметического линейного сдвига отличаются от команд логического сдвига тем, что они особым образом работают со знаковым разрядом операнда:

**sal операнд, счетчик\_сдвигов** (Shift Arithmetic Left) – арифметический сдвиг влево. Содержимое операнда сдвигается влево на количество битов, определяемое значением счетчик\_сдвигов. Справа (в позицию младшего бита) вписываются нули. Команда *sal* не сохраняет знака, но устанавливает флаг *cf* в случае смены знака очередным выдвигаемым битом. В остальном команда *sal* полностью аналогична команде *shl*.

**sar операнд, счетчик\_сдвигов** (Shift Arithmetic Right) – арифметический сдвиг вправо. Содержимое операнда сдвигается вправо на количество битов, определяемое значением счетчик\_сдвигов. Слева в операнд вписываются нули. Команда *sar* сохраняет знак, восстанавливая его после сдвига каждого очередного бита.

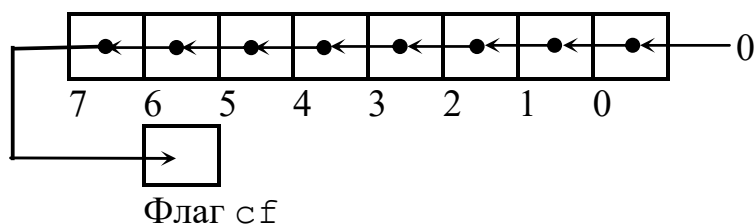


Рисунок 1.2 – Схема работы команд линейного логического сдвига влево `shl`

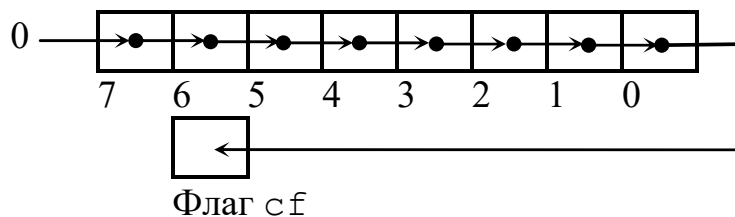


Рисунок 1.3 – Схема работы команд линейного логического сдвига вправо `shr`

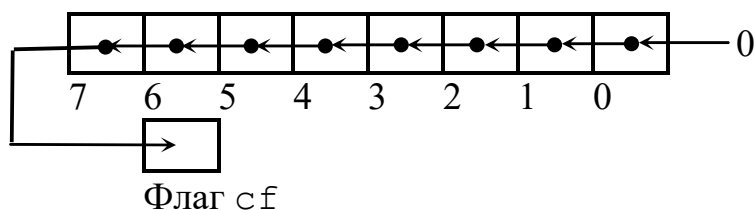


Рисунок 1.4 – Схема работы команд линейного арифметического сдвига влево `sal`

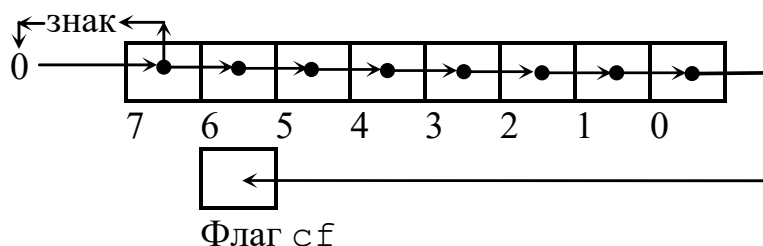


Рисунок 1.5 – Схема работы команд линейного арифметического сдвига вправо `sar`

Команды арифметического сдвига позволяют выполнить умножение и деление операнда на степени двойки. Рассмотрим двоичное представление чисел 75 и 150:  $75_{10} = 0100\ 1011_2$ ,  $150_{10} = 1001\ 0110_2$ .

Сдвигая вправо операнд, мы осуществляем операцию деления на степени двойки 2, 4, 8 и т. д. Преимущество этих команд, по сравнению с командами умножения и деления, в скорости их исполнения микропроцессором.

### 1.3.2 Циклический сдвиг

К командам циклического сдвига относятся команды, сохраняющие значения сдвигаемых бит. Есть два типа команд циклического сдвига: команды простого циклического сдвига; команды циклического сдвига через флаг переноса *cf*.

К командам простого циклического сдвига относятся:

**rol** *операнд, счетчик\_сдвигов* (Rotate Left) – циклический сдвиг влево. Содержимое операнда сдвигается влево на количество бит, определяемое операндом *счетчик\_сдвигов*. Сдвигаемые влево биты записываются в тот же операнд справа (рисунок 1.6).

**ror** *операнд, счетчик\_сдвигов* – (Rotate Right) циклический сдвиг вправо. Содержимое операнда сдвигается вправо на количество бит, определяемое операндом *счетчик\_сдвигов*. Сдвигаемые вправо биты записываются в тот же операнд слева (рисунок 1.7).

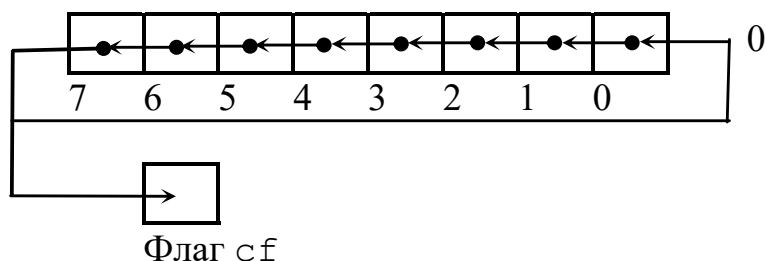


Рисунок 1.6 – Схема работы команд простого циклического сдвига влево *rol*

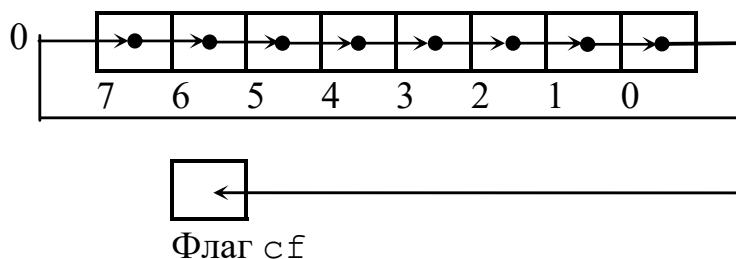


Рисунок 1.7 – Схема работы команд простого циклического сдвига вправо *ror*

Команды простого циклического сдвига имеют одну особенность: циклически сдвигаемый бит не только вдвигается в операнд с другого конца, но и одновременно его значение становится значением флага *cf*.

Команды циклического сдвига через флаг переноса *cf* отличаются от команд простого циклического сдвига тем, что сдвигаемый бит не сразу попадает в операнд с другого его конца, а записывается сначала во флаг переноса *cf*. Лишь следующее исполнение данной команды сдвига (при условии, что она выполняется в цикле) приводит к помещению выдвинутого ранее бита в другой конец операнда

(рисунок 1.8). К командам циклического сдвига через флаг переноса  $cf$  относятся следующие:

**rc1 операнд, счетчик\_сдвигов** (Rotate Through Carry Left) – циклический сдвиг влево через флаг переноса. Содержимое операнда сдвигается влево на количество бит, определяемое операндом счетчик\_сдвигов. Сдвигаемые биты поочередно становятся значением флага переноса  $cf$ .

**rcr операнд, счетчик\_сдвигов** (Rotate through Carry Right) – циклический сдвиг вправо через флаг переноса. Содержимое операнда сдвигается вправо на количество бит, определяемое операндом счетчик\_сдвигов. Сдвигаемые биты поочередно становятся значением флага переноса  $cf$ .

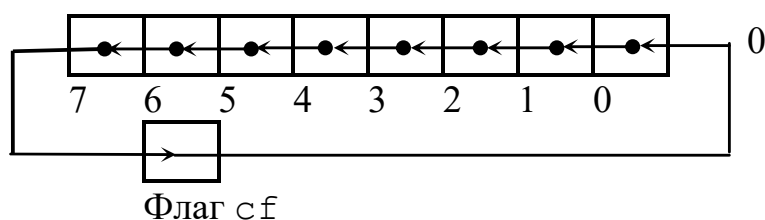


Рисунок 1.8 – Схема работы команд циклического сдвига влево **rc1** через флаг переноса  $cf$

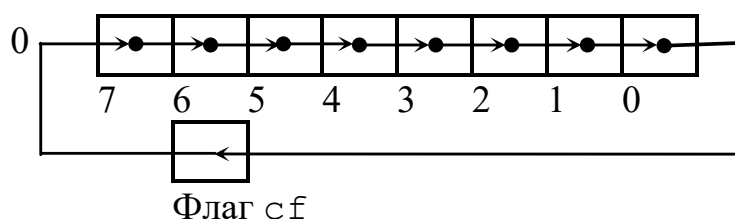


Рисунок 1.9 – Схема работы команд циклического сдвига вправо **rcr** через флаг переноса  $cf$

При сдвиге через флаг переноса появляется промежуточный элемент, с помощью которого можно производить подмену циклически сдвигаемых битов, в частности, рассогласование битовых последовательностей. Под рассогласованием битовой последовательности здесь и далее подразумевается действие, которое позволяет некоторым образом локализовать и извлечь нужные участки этой последовательности и записать их в другое место

## 1.4 Дополнительные команды сдвига

К дополнительным командам сдвига относятся команды сдвигов двойной точности:

**shld операнд\_1, операнд\_2, счетчик\_сдвигов** – сдвиг влево двойной точности. Команда shld производит замену путем сдвига битов операнда операнд\_1 влево, заполняя его биты справа значениями битов, вытесняемых из операнд\_2 (рисунок 1.10). Количество сдвигаемых бит определяется значением счетчик\_сдвигов, которое может лежать в диапазоне 0...31. Это значение может задаваться непосредственным операндом или содержаться в регистре `cl`. Значение операнд\_2 не изменяется.

**shrd операнд\_1, операнд\_2, счетчик\_сдвигов** – сдвиг вправо двойной точности. Команда производит замену путем сдвига битов операнда операнд\_1 вправо, заполняя его биты слева значениями битов, вытесняемых из операнд\_2 (рисунок 1.11). Количество сдвигаемых бит определяется значением счетчик\_сдвигов, которое может лежать в диапазоне 0...31. Это значение может задаваться непосредственным операндом или содержаться в регистре `cl`. Значение операнд\_2 не изменяется.

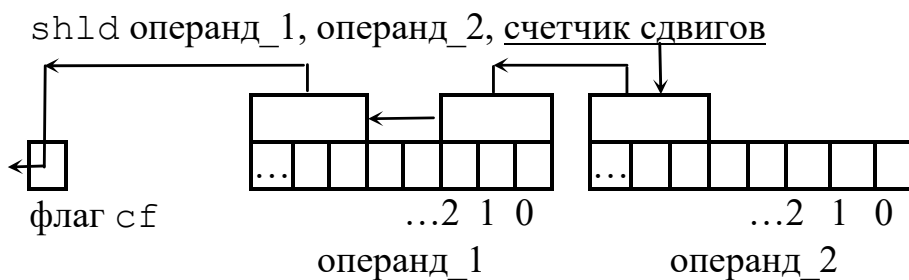


Рисунок 1.10 – Схема работы команды shld

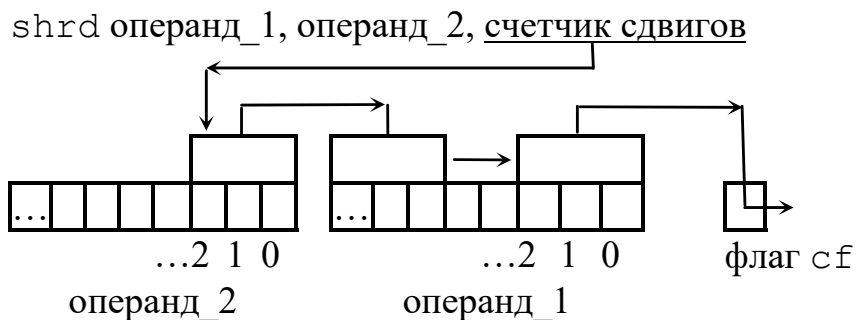


Рисунок 1.11 – Схема работы команды shrd

## 1.5 Практическое задание

Задание 1. Составить и отладить программу на языке ассемблера для выполнения логических команд над указанной переменной согласно варианта (таблица 1.1). Результаты представить в двоичной и десятичной системах счисления:

- найти номер первого (`bsf`) и последнего (`bsr`) ненулевого бита;

- проверить и установить  $n$ -й бит (bts);
- проверить и сбросить  $n + 1$ -й бит (btr);
- проверить и инвертировать  $n - 1$ -й бит (btc);
- выполнить команды линейного сдвига shl, shr, sal, sar над заданным значением на  $n$  бит;
- выполнить команды простого циклического сдвига rol и ror над заданным значением на  $n$  бит;
- реализовать операции установки  $n$ -го бита, сброса  $n + 1$  бита и инвертирования  $n - 1$ -го бита с помощью команд or, and, xor.

Таблица 1.1 – Варианты заданий

| №  | Выражение        | №  | Выражение        | №  | Выражение        |
|----|------------------|----|------------------|----|------------------|
| 1  | $a = 116, n = 6$ | 11 | $a = 126, n = 6$ | 21 | $a = 136, n = 6$ |
| 2  | $a = 117, n = 5$ | 12 | $a = 127, n = 5$ | 22 | $a = 137, n = 5$ |
| 3  | $a = 118, n = 4$ | 13 | $a = 128, n = 4$ | 23 | $a = 138, n = 3$ |
| 4  | $a = 119, n = 3$ | 14 | $a = 129, n = 3$ | 24 | $a = 139, n = 3$ |
| 5  | $a = 120, n = 2$ | 15 | $a = 130, n = 2$ | 25 | $a = 140, n = 2$ |
| 6  | $a = 121, n = 1$ | 16 | $a = 131, n = 1$ | 26 | $a = 141, n = 1$ |
| 7  | $a = 122, n = 2$ | 17 | $a = 132, n = 2$ | 27 | $a = 142, n = 2$ |
| 8  | $a = 123, n = 3$ | 18 | $a = 133, n = 3$ | 28 | $a = 143, n = 3$ |
| 9  | $a = 124, n = 4$ | 19 | $a = 134, n = 4$ | 29 | $a = 144, n = 4$ |
| 10 | $a = 125, n = 5$ | 20 | $a = 135, n = 5$ | 30 | $a = 145, n = 5$ |

*Пример выполнения задания 1.* Пусть  $a = 115, n = 3$ . Переведем число 115 в двоичную систему счисления:  $115_{10} = 0000\ 0000\ 0111\ 0011_2$ .

1. Номер первого ненулевого бита – 0, последнего ненулевого – 6.  
 2. Проверить и установить 3-й бит. Третий бит равен 0. Результат установки третьего бита:  $0000\ 0000\ 0111\ 1011_2 = 123_{10}$ .

3. Проверить и сбросить 4-й бит. Четвертый бит равен 1. Результат сброса четвертого бита:  $0000\ 0000\ 0110\ 0011_2 = 99_{10}$ .

4. Проверить и инвертировать 2-й бит. Второй бит равен 0. Результат инвертирования 2-го бита:  $0000\ 0000\ 0111\ 0111_2 = 119_{10}$ .

5. Выполнить команды линейного сдвига на 3 бита:

– логический сдвиг влево (shl):  $0000\ 0011\ 1001\ 1000_2 = 920_{10}$ ;

– логический сдвиг вправо (shr):  $0000\ 0000\ 0000\ 1110_2 = 14_{10}$ ;

– арифметический сдвиг влево (sal):  $0000\ 0011\ 1001\ 1000_2 = 920_{10}$ ;

– арифметически сдвиг вправо (sar):  $0000\ 0000\ 0000\ 1110_2 = 14_{10}$ .

6. Выполнить команды простого циклического сдвига на 3 бита:

– циклический сдвиг влево (rol):  $0000\ 0011\ 1001\ 1000_2 = 920_{10}$ ;

– циклический сдвиг вправо (ror):  $0110\ 0000\ 0000\ 1110_2 = 24590_{10}$ .

### Код программы:

```
;сегмент стека
sseg segment para stack 'stack'
    db 256 dup (0)
sseg ends

;сегмент данных
dseg segment para public 'data'
a dw 115 ;115=0000 0000 0111 0011
mes1 db 'Number of the first non-zero bit -->$'
mes2 db 10,13,'Number of the last non-zero bit -->$'
mes3 db 10,13,'The third bit is zero$'
mes4 db 10,13,'The third bit is not zero$'
mes5 db 10,13,'The third bit is equal to one-->$'
mes6 db 10,13,'The fourth bit is zero$'
mes7 db 10,13,'The fourth bit is not zero$'
mes8 db 10,13,'The fourth bit is equal to zero-->$'
mes9 db 10,13,'The two bit is zero$'
mes10 db 10,13,'The two bit is not zero$'
mes11 db 10,13,'The two bit is convert-->$'
mes12 db 10,13,'Shift logical left-->$'
mes13 db 10,13,'Shift logical right-->$'
mes14 db 10,13,'Shift arithmetic left-->$'
mes15 db 10,13,'Shift arithmetic left-->$'
mes16 db 10,13,'Rotate left-->$'
mes17 db 10,13,'Rotate right-->$'
mes18 db 10,13,'Rotate through carry left-->$'
mes19 db 10,13,'Rotate through carry right-->$'
mes20 db 10,13,'Logical command or, and, xor:$'
dseg ends
    extrn disp:near

;сегмент кода
cseg segment para public 'code'
main proc near
    assume cs:cseg,ds:dseg,ss:sseg
    mov ax,dseg ;связываем регистр dx с сегментом
    mov ds,ax   ;данных через регистр ax
.486           ;это обязательно!
    mov ax,0002h
    int 10h
;сканирование бит вперед
    mov ax,a
    bsf bx,ax
    lea dx,mes1
    mov ax,0900h
```



```

    int 21h
    mov ax,bx
    call disp
;сканирование бит в обратном порядке
    mov ax,a
    bsr bx,ax
    lea dx,mes2
    mov ax,0900h
    int 21h
    mov ax,bx
    call disp
;проверка и установка третьего бита
    mov bx,a
    mov ax,3
    bts bx,ax
    jc m1 ;переход, если бит равен 1
    lea dx,mes3
    mov ax,0900h
    int 21h
    jmp m2
m1: lea dx,mes4
    mov ax,0900h
    int 21h
m2: lea dx,mes5
    mov ax,0900h
    int 21h
    mov ax,bx
    call disp
;проверка и сброс четвертого бита
    mov bx,a
    mov ax,4
    btr bx,ax
    jc m3 ;переход, если бит равен 1
    lea dx,mes6
    mov ax,0900h
    int 21h
    jmp m4
m3: lea dx,mes7
    mov ax,0900h
    int 21h
m4: lea dx,mes8
    mov ax,0900h
    int 21h
    mov ax,bx

```

```

    call disp
;проверка и инвертирование второго бита
    mov bx,a
    mov ax,2
    btc bx,ax
    jc m5 ;переход, если бит равен 1
    lea dx,mes9
    mov ax,0900h
    int 21h
    jmp m6
m5: lea dx,mes10
    mov ax,0900h
    int 21h
m6: lea dx,mes11
    mov ax,0900h
    int 21h
    mov ax,bx
    call disp
;ЛОГИЧЕСКИЙ СДВИГ ВЛЕВО
    mov ax,a
    shl ax,3
    mov bx,ax
    lea dx,mes12
    mov ax,0900h
    int 21h
    mov ax,bx
    call disp
;ЛОГИЧЕСКИЙ СДВИГ ВПРАВО
    mov ax,a
    shr ax,3
    mov bx,ax
    lea dx,mes13
    mov ax,0900h
    int 21h
    mov ax,bx
    call disp
;АРИФМЕТИЧЕСКИЙ СДВИГ ВЛЕВО
    mov ax,a
    sal ax,3
    mov bx,ax
    lea dx,mes14
    mov ax,0900h
    int 21h
    mov ax,bx

```

```

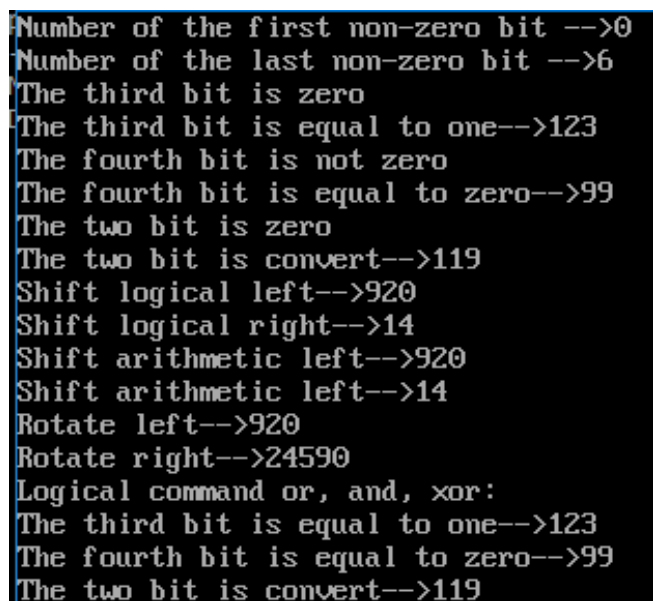
    call disp
;арифметический сдвиг вправо
    mov ax,a
    sar ax,3
    mov bx,ax
    lea dx,mes15
    mov ax,0900h
    int 21h
    mov ax,bx
    call disp
;циклический сдвиг влево
    mov ax,a
    rol ax,3
    mov bx,ax
    lea dx,mes16
    mov ax,0900h
    int 21h
    mov ax,bx
    call disp
;циклический сдвиг вправо
    mov ax,a
    ror ax,3
    mov bx,ax
    lea dx,mes17
    mov ax,0900h
    int 21h
    mov ax,bx
    call disp
;логические команды or, and, xor
    lea dx,mes20
    mov ax,0900h
    int 21h
;установка третьего бита
    mov ax,a
    or ax,1000b
    mov bx,ax
    lea dx,mes5
    mov ax,0900h
    int 21h
    mov ax,bx
    call disp
;сброс четвертого бита
    mov ax,a
    and ax,111111111101111b

```

```

mov bx,ax
lea dx,mes8
mov ax,0900h
int 21h
mov ax,bx
call disp
;инвертирование второго бита
mov ax,a
xor ax,100b
mov bx,ax
lea dx,mes11
mov ax,0900h
int 21h
mov ax,bx
call disp
;завершение программы
mov ax,4c00h
int 21h
main endp
cseg ends
end main

```



```

Number of the first non-zero bit -->0
Number of the last non-zero bit -->6
The third bit is zero
The third bit is equal to one-->123
The fourth bit is not zero
The fourth bit is equal to zero-->99
The two bit is zero
The two bit is convert-->119
Shift logical left-->920
Shift logical right-->14
Shift arithmetic left-->920
Shift arithmetic left-->14
Rotate left-->920
Rotate right-->24590
Logical command or, and, xor:
The third bit is equal to one-->123
The fourth bit is equal to zero-->99
The two bit is convert-->119

```

Рисунок 1.11 – Результат работы программы

Задание 2. Используя команды сдвига, сложения и вычитания, составить и отладить программу на языке ассемблера для вычисления заданного арифметического выражения (таблица 1.2). Команды умножения и деления не использовать.

Таблица 1.2 – Варианты заданий

| №  | Выражение                  | №  | Выражение                  | №  | Выражение                  |
|----|----------------------------|----|----------------------------|----|----------------------------|
| 1  | $y = \frac{8(a-b)+5}{4}$   | 11 | $y = \frac{8(a+b)-2b}{4}$  | 21 | $y = \frac{2a+4b-3}{8}$    |
| 2  | $y = \frac{2(a-3)+7b}{8}$  | 12 | $y = \frac{2a+16b-5}{2}$   | 22 | $y = \frac{4(a+b)-2b}{8}$  |
| 3  | $y = \frac{7a+4b-8}{8}$    | 13 | $y = \frac{2(4a-b)+6}{4}$  | 23 | $y = \frac{4(8a-b)+2a}{4}$ |
| 4  | $y = \frac{16(a-4b)+7}{4}$ | 14 | $y = \frac{2(8a+b)-3}{4}$  | 24 | $y = \frac{8(4a-b)+9}{4}$  |
| 5  | $y = \frac{8a-2b+4}{2}$    | 15 | $y = \frac{2(a+8b)-3}{4}$  | 25 | $y = \frac{4(a-16b)+4}{8}$ |
| 6  | $y = \frac{16a+8b-3}{4}$   | 16 | $y = \frac{4(a+8b)+6}{4}$  | 26 | $y = \frac{2a-8b}{4}$      |
| 7  | $y = \frac{8(a-2b)}{4}$    | 17 | $y = \frac{8a+4b-5}{2}$    | 27 | $y = \frac{8b+2a-7}{16}$   |
| 8  | $y = \frac{2(a-4b)+6}{8}$  | 18 | $y = \frac{4(a-8b)+5}{4}$  | 28 | $y = \frac{2(a-8b)+7}{4}$  |
| 9  | $y = \frac{4(a-2b)-5b}{2}$ | 19 | $y = \frac{8(a-2b)+b}{4}$  | 29 | $y = \frac{4(a+2b)-9}{8}$  |
| 10 | $y = \frac{2(8b+a)-1}{4}$  | 20 | $y = \frac{4(16a+b)-6}{8}$ | 30 | $y = \frac{2(a+8b)-b}{4}$  |

Пример выполнения задания 2. Вычислить значение выражения

$$y = \frac{2a+8b-7}{4}.$$

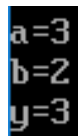
Код программы:

```
dseg segment para public 'data'
    a db 3
    b db 2
    y db ?
    mesa db 10,13,'a=$'
    mesb db 10,13,'b=$'
    mesy db 10,13,'y=$'
dseg ends
sseg segment para stack 'stack'
    db 256 dup (0)
sseg ends
```

```

extrn disp:near
cseg segment para public 'code'
main proc near
    assume cs:cseg, ds:dseg, ss:sseg
    mov ax,dseg
    mov ds,ax
    mov al,a      ;al=a
    sal al,1      ;al=2a
    mov bl,b      ;bl=b
    sal bl,3      ;bl=8b
    add al,bl     ;al=2a+8b
    sub al,7      ;al=2a+8b-7
    sar al,2      ;al=(2a+8b-7)/4
    mov y,al      ;y=al
    lea dx,mesa
    mov ax,0900H
    int 21H
    mov al,a
    cbw
    call disp
    lea dx,mesb
    mov ax,0900H
    int 21H
    mov al,b
    cbw
    call disp
    lea dx,mesy
    mov ax,0900H
    int 21H
    mov al,y
    cbw
    call disp
    mov ax,4C00H
    int 21H
main endp
cseg ends
end main

```



```

a=3
b=2
y=3

```

Рисунок 1.12 – Результат работы программы

## Тема 2. Макросредства языка ассемблера

Разработчики компиляторов ассемблера включили в язык аппарат макросредств. Транслятор ассемблера состоит из двух частей – транслятора, формирующего объектный модуль, и макроассемблера (рисунок 2.1).

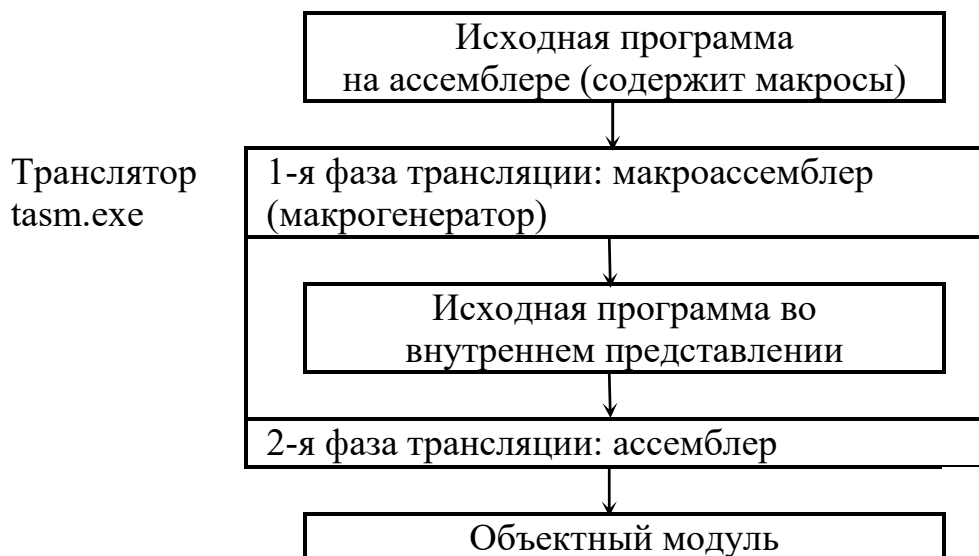


Рисунок 2.1 – Макроассемблер в общей схеме трансляции программы на tasm

Он является аналогом механизма, заложенного в работу макроассемблера. Основная идея – **использование подстановок**, которые замещают определенным образом организованную символьную последовательность другой символьной последовательностью. Создаваемая таким образом последовательность может быть как последовательностью, описывающей данные, так и последовательностью программных кодов. На входе макроассемблера может быть текст программы, не похожий на программы на языке ассемблера, а на выходе будет текст на чистом ассемблере, содержащем символические аналоги команд системы машинных команд микропроцессора. Обработка программы на ассемблере с использованием макросредств неявно осуществляется транслятором в две фазы (рисунок 2.1). На первой фазе работает часть компилятора, называемая макроассемблером. На второй фазе трансляции участвует непосредственно ассемблер, который формирует объектный код, содержащий текст исходной программы в машинном виде.

Пакет макроассемблера `masm` позволяет задавать макроопределения (или макросы), представляющие собой именованные группы команд, которые 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`.

## 2.1 Псевдооператоры `equ` и `=`

К простейшим макросредствам языка ассемблера относятся псевдооператоры `equ` и «`=`» (равно), которые предназначены для присвоения некоторому выражению символического имени или идентификатора. Когда в ходе трансляции этот идентификатор встретится в программе, макроассемблер подставит вместо него соответствующее выражение. В качестве выражения могут быть использованы константы, имена меток, символические имена и строки в апострофах. После присвоения этим конструкциям символического имени его можно использовать везде, где требуется размещение данной конструкции.

Синтаксис псевдооператора `equ`:

имя\_идентификатора `equ` строка\_или\_числовое\_выражение

Синтаксис псевдооператора `=`:

имя\_идентификатора `=` числовое\_выражение

Псевдооператоры `equ` и `=` отличаются следующим:

- с помощью `equ` идентификатору можно ставить в соответствие как числовые выражения, так и текстовые строки, а псевдооператор `=` может использоваться только с числовыми выражениями;
- идентификаторы, определенные с помощью `=`, можно переопределять в исходном тексте программы, а определенные с использованием `equ` – нельзя.

Ассемблер всегда пытается вычислить значение строки, воспринимая ее как выражение. Для того чтобы строка воспринималась именно как текстовая, необходимо заключить ее в угловые скобки: `<строка>`. Угловые скобки являются оператором ассемблера, с помощью которого транслятору сообщается, что заключенная в них строка должна трактоваться как текст, даже если в нее входят служебные слова ассемблера или операторы.

Псевдооператор `equ` удобно использовать для настройки программы на конкретные условия выполнения, замены сложных в обозначении объектов, многократно используемых в программе, более простыми именами и т. п. Пример:

```
count equ ax ;переименовать регистр
```

Псевдооператор `=` удобно использовать для определения простых абсолютных математических выражений. Пример:

```
len = 43  
len = len+1
```

Компилятор `tasm` содержит директивы, значительно расширяющие его возможности по работе с текстовыми макросами. Эти директивы аналогичны некоторым функциям обработки строк в языках высокого уровня. Под строками здесь понимается текст, описанный с помощью псевдооператора `equ`.



1. Директива слияния строк `catstr`:

идентификатор `catstr` строка\_1, строка\_2,...

Значением этого макроса будет новая строка, состоящая из сцепленной слева направо последовательности строк строка\_1, строка\_2,.... В качестве сцепляемых строк могут быть указаны имена ранее определенных макросов. Пример:

```
pre equ <Привет,>
name equ <Юля>
privet catstr pre,name;privet = «Привет, Юля»
```

2. Директива выделения подстроки в строке `substr`:

идентификатор `substr` строка, номер\_позиции, размер

Значением данного макроса будет часть заданной строки, начинающаяся с позиции с номером номер\_позиции и длиной, указанной в размер. Если требуется только остаток строки, начиная с некоторой позиции, то достаточно указать только номер позиции без указания размера. Пример:

```
privet catstr pre,name;privet = «Привет, Юля»
name substr privet,7,3;name = «Юля»
```

3. Директива определения вхождения одной строки в другую `instr`:

идентификатор `instr` номер\_нач\_позиции, строка\_1, строка\_2

После обработки данного макроса транслятором идентификатору будет присвоено числовое значение, соответствующее номеру (первой) позиции, с которой совпадают строка\_1 и строка\_2. Если такого совпадения нет, то идентификатор получит значение 0;

4. Директива определения длины строки в текстовом макросе

`sizestr`:

идентификатор `sizestr` строка

В результате обработки данного макроса значение идентификатор устанавливается равным длине строки. Пример:

```
len sizestr privet;len = 10
```

## 2.2 Макрокоманды

Макрокоманда представляет собой механизм замены текста. Повторяющиеся участки кода программы можно оформить в виде макрокоманд и использовать эти повторяющиеся фрагменты в различных программах.

*Макрокоманда* представляет собой строку, содержащую некоторое символическое имя – имя макрокоманды, и предназначенную для того, чтобы быть замещенной одной или несколькими другими строками. Имя макрокоманды может сопровождаться параметрами.

Задание шаблона-описания макрокоманды называют *макроопределением*. Синтаксис макроопределения:

```
имя_макрокоманды macro список_формальных_аргументов
тело макроопределения
endm
```

Есть три варианта размещения макроопределения:

1. В начале исходного текста программы до сегмента кода и данных с тем, чтобы не ухудшать читабельность программы. Этот вариант следует применять в случаях, если определяемые макрокоманды нужны только к пределам одной программы.

2. В отдельном файле. Этот вариант подходит при работе над несколькими программами. Чтобы сделать доступными эти макроопределения в конкретной программе, необходимо в начале исходного текста этой программы записать директиву

```
include имя_файла
```

Пример:

```
masm
model small
include show.inc
```

;в это место будет вставлен текст файла show.inc

3. В макробιβлиотеке. Если есть универсальные макрокоманды, которые используются практически во всех программах, то их целесообразно записать в так называемую макробιβлиотеку. Подключается бιβлиотека с помощью директивы `include`. Недостаток этого и предыдущего способов в том, что в исходный текст программы включаются абсолютно все макроопределения. Директива `purge`, в качестве операндов которой через запятую перечисляются имена макрокоманд, позволяет не включать в текст программы заданные макросы. Пример:

```
include imac.inc
purge outstr,exit
```

В данном случае в исходный текст программы перед началом компиляции `tasm` вместо строки `include imac.inc` вставит строки из файла `imac.inc`. Но вставленный текст будет отличаться от оригинала тем, что в нем будут отсутствовать макроопределения `outstr` и `exit`.

Пример программы с макроопределениями:

```
init_ds macro ;макрос настройки ds на сегмент данных
    mov ax,data
    mov ds,ax
endm

out_str macro str ;макрос вывода строки на экран
    push ax
    mov ah,09h
    mov dx,offset str
    int 21h
    pop ax
endm
```

```

clear_r macro rg ;макрос очистки регистра rg
    xor rg,rg
endm

get_char macro ;макрос ввода символа, введенный символ в al
    mov ah,1h
    int 21h
endm

conv_16_2 macro ;макрос преобразования символа
;шестнадцатеричной цифры в ее двоичный эквивалент в al
    sub al,30h
    cmp al,9h
    jle $+4
    sub al,7h
endm

exit macro ;макрос конца программы
    mov ax,4c00h
    int 21h
endm

data segment para public 'data'
message db 'Введите две шестнадцатеричные цифры
(буквы A,B,C,D,E,F - прописные):$'
data ends

stk segment stack
    db 256 dup(0)
stk ends

code segment para public 'code'
    assume cs:code,ds:data,ss:stk
main proc
    init_ds
    out_str message
    clear_r ax
    get_char
    conv_16_2
    mov cl,4h
    shl al,cl
    mov dl,al
    get_char
    conv_16_2
    add dl,al
    xchg dl,al ;результат в al
    exit
main endp
code ends
end main

```

2-ой вариант в отдельном файле:

```
masm
model small          ;модель памяти
stack 256            ;размер стека
include mac1.inc     ;подключение файла с макросами
.data
...
```

Функционально макроопределения похожи на процедуры. Сходство их в том, что и те, и другие достаточно один раз где-то описать, а затем вызывать их специальным образом. Различия между процедурами и макроопределениями:

- в отличие от процедуры, текст которой неизменен, макроопределение в процессе макрогенерации может меняться в соответствии с набором фактических параметров. При этом коррекции могут подвергаться как операнды команд, так и сами команды. Процедуры в этом отношении менее гибки;

- при каждом вызове макрокоманды ее текст в виде макрорасширения вставляется в программу. При вызове процедуры микропроцессор осуществляет передачу управления на начало процедуры, находящейся в некоторой области памяти в одном экземпляре. Код в этом случае получается более компактным хотя быстродействие несколько снижается за счет необходимости осуществления переходов.

Вызов макроопределения:

имя\_макрокоманды список\_фактических\_аргументов

Результатом применения данной синтаксической конструкции в исходном тексте программы будет ее замещение строками из конструкции тела макроопределения. В результате применения макрокоманды в программе формальные аргументы в макроопределении замещаются соответствующими фактическими аргументами. Процесс такого замещения называется макрогенерацией, а результатом этого процесса является макрорасширение.

Каждый фактический аргумент представляет собой строку символов, для формирования которой применяются следующие правила:

1. Строка может состоять:

- из последовательности символов без пробелов, точек, запятых, точек с запятой;

- из последовательности любых символов, заключенных в угловые скобки: <...>. В этой последовательности можно указывать как пробелы, так и точки, запятые, точки с запятыми.

2. Для того чтобы указать, что некоторый символ внутри строки, представляющей фактический параметр, является собственно символом, применяется специальный оператор «!». Этот оператор ставится непосредственно перед описанным выше символом, и его действие эквивалентно заключению данного символа в угловые скобки.

3. Если требуется вычисление в строке некоторого константного выражения, то в начале этого выражения нужно поставить знак %:

%константное\_выражение – значение константное\_выражение вычисляется и подставляется в текстовом виде в соответствии с текущей системой счисления.

Транслятор распознает формальные аргументы в теле макроопределения по их именам в заголовке макроопределения. В процессе генерации макрорасширения компилятор ассемблера ищет в тексте тела макроопределения последовательности символов, совпадающие с теми последовательностями символов, из которых состоят формальные параметры. После обнаружения такого совпадения формальный параметр из тела макроопределения замещается соответствующим фактическим параметром из макрокоманды. Этот процесс называется *подстановкой аргументов*. Полный синтаксис списка\_формальных\_аргументов содержит не только перечисление формальных элементов через запятую, но и некоторую дополнительную информацию:

имя\_формального\_аргумента [:тип]

где тип может принимать значения:

REQ, которое говорит о том, что требуется обязательное явное задание фактического аргумента при вызове макрокоманды;

=<любая\_строка> – если аргумент при вызове макрокоманды не задан, то в соответствующие места в макрорасширении будет вставлено значение по умолчанию соответствующее значению любая\_строка. Символы, входящие в любая\_строка, должны быть заключены в угловые скобки.

Если формальный аргумент является частью некоторого идентификатора, то в этом случае последовательность символов формального аргумента отделяют от остального контекста с помощью специального символа &. Этот прием часто используется для задания модифицируемых идентификаторов и кодов операций.

Пример. Определим макрос, который предназначен для генерации в программе некоторой таблицы, причем параметры этой таблицы можно задавать с помощью аргументов макрокоманды:

```
def_table macro type=b, len=REQ
    tabl_&type d&type len dup (0)
endm
.data
    def_table b,10
    def_table w,5
```

После трансляции программы получим следующие макрорасширения:

```
tabl_b db 10 dup (0)
tabl_w dw 10 dup (0)
```

Символ & можно применять и для распознавания формального аргумента в строке, заключенной в кавычки ". Например:

```
num_char macro message
    elem db "Строка &message содержит"
endm
```

Если в программе некоторая макрокоманда вызывается несколько раз, то в процессе макрогенерации один и тот же идентификатор будет определен несколько раз, что будет распознано транслятором как ошибка. В этом случае применяется директива `local`, которая имеет следующий синтаксис:

```
local список_идентификаторов
```

Данную директиву необходимо задавать непосредственно за заголовком макроопределения. Результатом работы этой директивы будет генерация в каждом экземпляре макрорасширения уникальных имен для всех идентификаторов, перечисленных в списке идентификаторов. Эти уникальные имена имеют вид `??xxxx`, где `xxxx` – шестнадцатеричное число. Для первого идентификатора в первом экземпляре макрорасширения `xxxx = 0000`, для второго – `xxxx = 0001` и т. д. Контроль за правильностью размещения и использования этих уникальных имен берет на себя ассемблер.

В теле макроопределения можно размещать комментарии. Если для обозначения комментария используются две подряд идущие точки с запятой, то при генерации макрорасширения этот комментарий будет исключен. Если необходимо присутствие комментария в макрорасширении, то его нужно задавать с помощью одинарной точки с запятой:

```
mes macro message
;этот комментарий будет включен в текст листинга
;;этот комментарий не будет включен в текст листинга
endm
```

## 2.3 Макродирективы

С помощью макросредств ассемблера можно не только частично изменять входящие в макроопределение строки, но и модифицировать сам набор этих строк и даже порядок их следования. Сделать это можно с помощью *набора макродиректив* (далее – просто директив). Их можно разделить на две группы.

1. Директивы *повторения* `WHILE`, `REPT`, `IRP` и `IRPC` предназначены для создания макросов, содержащих несколько идущих подряд одинаковых последовательностей строк. При этом возможна частичная модификация этих строк.

2. Директивы *управления процессом генерации макрорасширений* `EXITM` и `GOTO` предназначены для управления процессом формирования макрорасширения из набора строк соответствующего макроопределения. С помощью этих директив можно как исключать отдельные

строки из макрорасширения, так и прекращать процесс генерации. Директивы EXITM и GOTO обычно используются вместе с условными директивами компиляции.

### 2.3.1 Директивы WHILE и REPT

Директивы WHILE и REPT применяются для повторения определенное количество раз некоторой последовательности строк. Эти директивы имеют следующий синтаксис:

```
WHILE константное_выражение
    последовательность строк
ENDM
REPT константное_выражение
    последовательность строк
ENDM
```

Последовательность повторяемых строк в обеих директивах ограничена директивой ENDM.

При использовании директивы WHILE макрогенератор будет повторять последовательность строк до тех пор, пока значение константное\_выражение не станет равным нулю. Это значение вычисляется каждый раз перед очередной итерацией цикла повторения (значение константное\_выражение в процессе макрогенерации должно подвергаться изменению внутри последовательности строк).

Директива REPT, подобно директиве WHILE, повторяет последовательность строк столько раз, сколько это определено значением константное\_выражение. Отличие этой директивы от WHILE состоит в том, что она автоматически уменьшает на единицу значение константное\_выражение после каждой итерации.

Пример использования директив WHILE и REPT для резервирования области памяти в сегменте данных. Имя идентификатора и длина области задаются в качестве параметров соответствующих макросов def\_sto\_1 и def\_sto\_2. Счетчик повторений в директиве REPT уменьшается автоматически после каждой итерации цикла.

```
def_sto_1 macro id_table,ln:=<5>
;макрос резервирования памяти длиной len, используя WHILE
    id_table label byte
    len=ln
    while len
        db 0
        len=len-1
    endm
endm
def_sto_2 macro id_table,len
;макрос резервирования памяти длиной len, используя REPT
```

```

    id_table label byte
    rept len
        db 0
    endm
endm
data segment para public 'data'
    def_sto_1 tab_1,10
    def_sto_2 tab_2,10
data ends
init_ds macro ;макрос настройки ds на сегмент данных
    mov ax,data
    mov ds,ax
endm
exit macro ;макрос конца программы
    mov ax, 4c00h
    int 21h
endm
code segment para public 'code'
    assume cs:code,ds:data
    main proc
        init_ds
        exit
    main endp
code ends
end main

```

Таким образом, директивы REPT и WHILE удобно применять для «размножения» в тексте программы последовательности одинаковых строк без внесения в эти строки каких-либо изменений на этапе трансляции.

### 2.3.2 Директива IRP

Директива IRP имеет следующий синтаксис:

```

IRP формальный_аргумент, <строка_символов_1,...
                               строка_символов_n>
последовательность строк
ENDM

```

Действие данной директивы заключается в том, что она повторяет последовательность строк *n* раз, то есть столько раз, сколько строк символов заключено в угловые скобки во втором операнде директивы IRP. Повторение последовательности строк сопровождается заменой в этих строках формального аргумента очередной строкой символов из второго операнда. При первой генерации последовательности строк формальный аргумент в них заменяется первой строкой символов (то есть аргументом строка\_символов\_1). Если есть вторая строка символов (строка\_символов\_2), это приводит к генерации второй



копии последовательности строк, в которой формальный аргумент заменяется второй строкой символов. Эти действия продолжаются до последней строки символов (строка\_символов\_n) включительно.

Пример.

```
irp ini, <1,2,3,4>
    db ini
endm
```

Макрогенератором будет сгенерировано следующее макрорасширение:

```
db 1
db 2
db 3
db 4
```

### 2.3.3 Директива **IRPC**

Директива **IRPC** имеет следующий синтаксис:

```
IRPC формальный_аргумент, строка_символов
    последовательность строк
ENDM
```

Действие данной директивы подобно действию директивы **IRP**, но отличается тем, что она на каждой очередной итерации заменяет формальный аргумент очередным символом из строка\_символов. Количество повторений последовательности строк будет определяться количеством символов в строке\_символов. Пример:

```
irpc rg,abcd
    push rg&x
endm
```

В процессе макрогенерации эта директива развернется в следующую последовательность строк:

```
push ax
push bx
push cx
push dx
```

Если строка символов, задаваемая в директиве **IRP**, содержит спецсимволы, такие как точки и запятые, то она должна быть заключена в угловые скобки: <ab, , cd>.

### 2.3.4 Директивы условной компиляции

Существует два вида этих директив *условной компиляции*:

1. *Директивы компиляции по условию* позволяют проанализировать определенные условия в ходе генерации макрорасширения и при необходимости изменить этот процесс.

2. *Директивы генерации ошибок по условию* контролируют ход генерации макрорасширения с целью генерации или обнаружения ситуаций, которые могут интерпретироваться как ошибочные.

С этими директивами применяются директивы управления процессом генерации *макрорасширений* EXITM и GOTO.

Директива EXITM не имеет операндов, она немедленно прекращает процесс генерации макрорасширения, как только встречается в макроопределении.

Директива GOTO имя\_метки переводит процесс генерации макроопределения в другое место, прекращая тем самым последовательное разворачивание строк макроопределения. Метка, на которую передается управление, имеет специальный формат:

:имя\_метки

*Директивы компиляции по условию* предназначены для выборочной трансляции фрагментов программного кода. Это означает, что в макрорасширение включаются не все строки макроопределения, а только те, которые удовлетворяют определенным условиям. Какие конкретно условия должны быть проверены, определяется типом условной директивы. Всего имеются 10 типов условных директив компиляции. Их попарно объединяют в четыре группы:

1. IF и IFE – условная трансляция по результату вычисления логического выражения;
2. IFDEF и IFNDEF – условная трансляция по факту определения символического имени;
3. IFB и IFNB – условная трансляция по факту определения фактического аргумента при вызове макрокоманды;
4. IFIDN, IFIDNI, IFDIF и IFDIFI – условная трансляция по результату сравнения строк символов.

Условные директивы компиляции имеют общий синтаксис и применяются в составе следующей синтаксической конструкции:

```
IFxxx логическое_выражение_или_аргументы  
фрагмент_программы_1  
ELSE  
фрагмент_программы_2  
ENDIF
```

Заключение некоторых фрагментов текста программы (фрагмент\_программы\_1 и фрагмент\_программы\_2) между директивами IFxxx, ELSE и ENDIF приводит к их выборочному включению в объектный модуль. Какой именно из этих фрагментов будет включен в объектный модуль, зависит от конкретного типа условной директивы, задаваемого значением xxx, и значения условия, определяемого операндом (операндами) условной директивы логическое\_выражение\_или\_аргумент(ы).

**Директивы IF и IFE.** Синтаксис директив:

```
IF (E) логическое_выражение  
фрагмент_программы_1
```

```
ELSE
    фрагмент_программы_2
ENDIF
```

Обработка этих директив макроассемблером заключается в вычислении логического выражения и включении в объектный модуль первого (фрагмент\_программы\_1) или второго (фрагмент\_программы\_2) фрагмента программы в зависимости от того, в какой директиве (IF или IFE) это выражение встретилось.

Если в директиве IF логическое выражение истинно, то в объектный модуль помещается первый фрагмент программы. Если логическое выражение ложно, то при наличии директивы ELSE в объектный код помещается второй фрагмент программы. Если же директивы ELSE нет, то вся часть программы между директивами IF и ENDIF игнорируется, и в объектный модуль ничего не включается. Ложным оно будет считаться, если его значение равно нулю, а истинным – при любом значении, отличном от нуля.

Директива IFE аналогично директиве IF анализирует значение логического выражения. Но теперь для включения первого фрагмента программы в объектный модуль требуется, чтобы логическое выражение было ложным.

Директивы IF и IFE очень удобно использовать для изменения текста программы в зависимости от некоторых условий.

Пример. Составим макрос для определения в программе области памяти длиной не более 50 и не менее 10 байт.

```
masm
model small
stack 256
def_tab_50 macro len
    if len GE 50
        GOTO exit
    endif
    if len LT 10
        :exit
    EXITM
    endif
    rept len
        db 0
    endm
endm
.data
    def_tab_50 15
    def_tab_50 5
.code
main:
```

```

    mov ax, @data
    mov ds, ax
exit:
    mov ax, 4c00h
    int 21h
end main

```

Условные директивы действуют только на шаге трансляции, и поэтому результат их работы можно увидеть лишь после макрогенерации, то есть в листинге программы.

Другой вариант применения директив IF и IFE — отладочная печать. В процессе отладки программы почти всегда возникает необходимость динамически отслеживать состояние определенных программно-аппаратных объектов, в качестве которых могут выступать переменные, регистры процессора и т. п. После этапа отладки отпадает необходимость в таких диагностических сообщениях. Для их устранения приходится корректировать исходный текст программы, после чего подвергать ее повторной трансляции. Можно определить в программе некоторую переменную, к примеру debug, и использовать ее совместно с условными директивами IF или IFE:

```

debug equ i
.code
...
if debug
;любые команды и директивы ассемблера
endif

```

На время отладки и тестирования программы можно заключить отдельные участки кода в своеобразные операторные скобки в виде директив IF и ENDIF, реагирующие на значение логической переменной debug. При значении debug = 0 транслятор полностью проигнорирует текст внутри этих условных операторных скобок; при debug = 1, наоборот, будут выполнены все действия, описанные внутри них.

**Директивы IFDEF и IFNDEF.** Синтаксис директив:

```

IF (N) DEF символическое_имя
    фрагмент_программы_1
ELSE
    фрагмент_программы_2
ENDIF

```

Данные директивы позволяют управлять трансляцией фрагментов программы в зависимости от того, определено или нет в программе некоторое символическое имя.

Директива IFDEF проверяет, описано или нет в программе символическое имя, и если имя описано, то в объектный модуль помещается первый фрагмент программы (фрагмент\_программы\_1). В противном

случае при наличии директивы ELSE в объектный код помещается второй фрагмент программы (фрагмент\_программы\_2). Если же директивы ELSE нет (и символическое имя в программе не описано), то вся часть программы между директивами IF и ENDIF игнорируется и в объектный модуль не включается.

Действие IFNDEF обратно действию IFDEF. Если символического имени в программе нет, то транслируется первый фрагмент программы. Если оно присутствует, то при наличии ELSE транслируется второй фрагмент программы. Если ELSE отсутствует, а символическое имя в программе определено, то часть программы, заключенная между IFNDEF и ENDIF, игнорируется.

В качестве примера рассмотрим ситуацию, когда в объектный модуль программы должен быть включен один из трех фрагментов кода в зависимости от значения некоторого идентификатора switch: если switch = 0, то сгенерировать фрагмент для вычисления выражения  $y = x \cdot 2^n$ ; если switch = 1, то сгенерировать фрагмент для вычисления выражения  $y = x / 2^n$ ; если идентификатор switch не определен, то ничего не генерировать.

```
ifndef sw ;если sw не определено, то выйти из макроса
    EXITM
else ;иначе на вычисление
    mov cl,n
    ife sw
        sal x,cl ;умножение на степень 2 сдвигом влево
    else
        sar x,cl ;деление на степень 2 сдвигом вправо
    endif
endif
```

Эти директивы логически связаны с директивами IF и IFE, то есть их можно применять в тех же самых случаях, что и последние.

**Директивы IFB и IFNB.** Синтаксис директив:

```
IF (N) B аргумент
    фрагмент_программы_1
ELSE
    фрагмент_программы_2
ENDIF
```

Данные директивы используются для проверки фактических параметров, передаваемых в макрос. При вызове макрокоманды они анализируют значение аргумента и в зависимости от того, равно оно пробелу или нет, транслируется либо первый фрагмент программы (фрагмент\_программы\_1), либо второй (фрагмент\_программы\_2). Какой именно фрагмент будет выбран, зависит от кода директивы.

Директива `IFB` проверяет равенство аргумента пробелу. В качестве аргумента могут выступать имя или число. Если его значение равно пробелу (то есть фактический аргумент при вызове макрокоманды не был задан), то транслируется и помещается в объектный модуль первый фрагмент программы. В противном случае при наличии директивы `ELSE` в объектный код помещается второй фрагмент программы. Если же директивы `ELSE` нет, то при равенстве аргумента пробелу вся часть программы между директивами `IFB` и `ENDIF` игнорируется и в объектный модуль не включается.

Действие `IFNB` обратно действию `IFB`. Если значение аргумента в программе не равно пробелу, то транслируется первый фрагмент программы. В противном случае при наличии директивы `ELSE` в объектный код помещается второй фрагмент программы. Если же директивы `ELSE` нет, то вся часть программы (при неравенстве аргумента пробелу) между директивами `IFNB` и `ENDIF` игнорируется и в объектный модуль не включается.

Примером применения этих директив являются строки в макроопределении, проверяющие, указывается ли фактический аргумент при вызове соответствующей макрокоманды:

```
show macro reg
  ifb <reg>
    display "не задан регистр"
  exitm
endif
endm
```

Если в сегменте кода вызвать макрос `show` без аргументов, то будет выведено сообщение о том, что не задан регистр, и генерация макрорасширения прекратится директивой `EXITM`.

**Директивы `IFIDN`, `IFIDNI`, `IFDIF` и `IFDIFI`** позволяют не просто проверить наличие или значения аргументов макрокоманды, но и выполнить идентификацию аргументов как строк символов. Синтаксис этих директив:

```
IFIDN(I) аргумент_1, аргумент_2
  фрагмент_программы_1
ELSE
  фрагмент_программы_2
ENDIF
```

```
IFDIF(I) аргумент_1, аргумент_2
  фрагмент_программы_1
ELSE
  фрагмент_программы_2
ENDIF
```

В этих директивах проверяются аргумент\_1 и аргумент\_2 как строки символов. Какой именно код (фрагмент\_программы\_1 или фрагмент\_программы\_2) будет транслироваться по результатам сравнения, зависит от кода директивы. Наличие двух пар этих директив объясняется тем, что они позволяют учитывать либо не учитывать различие строчных и прописных букв. Директивы IFIDNI и IFDIFI игнорируют это различие, а IFIDN и IFDIF – учитывают.

Директива IFIDN(I) сравнивает символьные значения аргумент\_1 и аргумент\_2. Если результат сравнения положительный, то транслируется и помещается в объектный модуль первый фрагмент программы. В противном случае при наличии директивы ELSE, в объектный код помещается второй фрагмент программы. Если же директивы ELSE нет, то вся часть программы между директивами IFIDN(I) и ENDIF игнорируется и в объектный модуль не включается.

Действие IFDIF(I) обратно действию IFIDN(I). Если результат сравнения отрицательный (строки не совпадают), транслируется первый фрагмент программы. В противном случае все происходит аналогично рассмотренным ранее директивам.

Эти директивы удобно применять для проверки фактических аргументов макрокоманд. Например, проверим, какой из регистров – al или ah – передан в макрос в качестве параметра (проверка проводится без учета различия строчных и прописных букв):

```
show macro rg
    ifdifi <al>,<rg>
        GOTO m_al
    else
        ifdifi <ah>,<rg>
            goto m_ah
        else
            exitm
        endif
    endif
:m_al
...
:m_ah
...
endm
```

Транслятор tasm допускает вложенность условных директив компиляции и предоставляет набор дополнительных директив формата ELSEIFxxx, которые заменяют последовательность идущих подряд директив ELSE и IFxxx в структуре

```
IFxxx
ELSE
```

```

IFxxx
ENDIF
ENDIF

```

Эту последовательность условных директив можно заменить эквивалентной последовательностью дополнительных директив

```

IFxxx
ELSEIFxxx
ENDIF

```

Символы xxx в ELSExxx говорят о том, что каждая из директив – IF, IFB, IFIDN и т. д. – имеет соответствующую директиву ELSEIF, ELSEIFB, ELSEIFIDN и т. д. В последнем примере последовательности ELSE и IFDIFI можно записать так:

```

show macro rg
    ifdifi <al>, <rg>
        goto M_al
    elseifdifi <ah>, <rg>
        goto M_ah
    else
        exitm
    endif
:M_al
...
:M_ah
...
endm

```

### 2.3.5 Директивы генерации ошибок

В языке Assembler есть ряд директив, называемых *директивами генерации пользовательской ошибки*. Они предназначены для обнаружения различных ошибок в программе, таких как неопределенные метки или пропуск параметров макроса. Директивы генерации пользовательской ошибки по принципу работы можно разделить на два типа: безусловные директивы генерируют ошибку трансляции без проверки каких-либо условий; условные директивы генерируют ошибку трансляции после проверки определенных условий. Большинство директив генерации ошибок имеет два обозначения, хотя принцип их работы одинаков. Второе название отражает их сходство с директивами условной компиляции. Парные директивы будут приводиться в скобках.

**Безусловная генерация пользовательской ошибки.** К директивам безусловной генерации пользовательской ошибки относится одна директива ERR (.ERR). Она безусловно приводит к генерации ошибки на этапе трансляции и удалению объектного модуля. Директива используется с директивами условной компиляции для отладки программы.



**Условная генерация пользовательской ошибки.** Набор условий, на которые реагируют директивы условной генерации пользовательской ошибки, такой же, как и у директив условной компиляции. Использовать большинство директив условной генерации пользовательской ошибки, как и директив условной компиляции, можно и в макроопределениях, и в любых местах программы.

**Директивы .ERRB (ERRIFB) и .ERRNB (ERRIFNB).** Синтаксис директив:

`.ERRB (ERRIFB) <имя_формального_аргумента>`

`.ERRNB (ERRIFNB) <имя_формального_аргумента>`

Директива `.ERRB (ERRIFB)` вызывает генерацию пользовательской ошибки, если формальный аргумент с именем `<имя_формального_аргумента>` пропущен.

Директива `.ERRNB (ERRIFNB)` вызывает генерацию пользовательской ошибки, если формальный аргумент с именем `<имя_формального_аргумента>` присутствует.

Данные директивы применяются для генерации ошибки трансляции в зависимости от того, задан или нет при вызове макрокоманды фактический аргумент, соответствующий формальному аргументу в заголовке макроопределения с именем `<имя_формального_аргумента>`. По принципу действия эти директивы полностью аналогичны соответствующим директивам условной компиляции `IFB` и `IFNB`. Их обычно используют для проверки задания параметров при вызове макроса. Строка, являющаяся именем формального аргумента, должна быть заключена в угловые скобки.

Пример. Определим обязательность задания фактического аргумента, соответствующего формальному аргументу `rg`, в макросе `show`:

```
show macro rg
```

```
;если rg в макрокоманде не будет задан, то завершить компиляцию
```

```
.errb <rg>
```

```
;текст макроопределения
```

```
endm
```

**Директивы .ERRDEF (ERRIFDEF) и .ERRNDEF (ERRIFNDEF).** Директивы генерируют ошибку трансляции в зависимости от того, определено или нет символическое имя в программе. Синтаксис директив:

`.ERRDEF (ERRIFDEF) символическое_имя`

`.ERRNDEF (ERRIFNDEF) символическое_имя`

Директива `.ERRDEF (ERRIFDEF)` генерирует пользовательскую ошибку, если указанное символическое имя определено до выдачи этой директивы в программе.

Директива `.ERRNDEF (ERRIFNDEF)` генерирует пользовательскую ошибку, если указанное символическое имя не определено до момента обработки транслятором этой директивы.

## Директивы **.ERRDIF (ERRIFDIF)** и **.ERRIDN (ERRIFIDN)**.

Синтаксис директив:

**.ERRDIF (ERRIFDIF)** <строка\_1>, <строка\_2>

**.ERRIDN (ERRIFIDN)** <строка\_1>, <строка\_2>

Директива **.ERRDIF (ERRIFDIF)** генерирует пользовательскую ошибку, если две строки посимвольно не совпадают. Строки могут быть символическими именами, числами или выражениями и должны быть заключены в угловые скобки. Аналогично директиве условной компиляции **IFDIF**, при сравнении учитывается различие прописных и строчных букв.

Директива **.ERRIDN (ERRIFIDN)** генерирует пользовательскую ошибку, если строки посимвольно идентичны. Строчные и прописные буквы воспринимаются как разные.

Для того чтобы игнорировать различия строчных и прописных букв, существуют аналогичные директивы:

**ERRIFDIFI** <строка\_1>, <строка\_2>

**ERRIFIDNI** <строка\_1>, <строка\_2>

Директива **ERRIFDIFI** аналогична директиве **ERRIFDIF**, но игнорируется различие строчных и прописных букв при сравнении строк <строка\_1> и <строка\_2>.

Директива **ERRIFIDNI** аналогична директиве **ERRIFIDN**, но игнорируется различие строчных и прописных букв при сравнении строк <строка\_1> и <строка\_2>.

Данные директивы удобно применять для проверки передаваемых в макрос фактических параметров.

Директивы **.ERRE (ERRIFE)** и **.ERRNZ (ERRIF)**. Синтаксис директив:

**.ERRE (ERRIFE)** константное\_выражение

**.ERRNZ (ERRIF)** константное\_выражение

Директива **.ERRE (ERRIFE)** вызывает пользовательскую ошибку, если константное выражение ложно (равно нулю).

Директива **.ERRNZ (ERRIF)** вызывает пользовательскую ошибку, если константное выражение истинно (не равно нулю).

Вычисление константного выражения должно приводить к абсолютному значению и не может содержать ссылок вперед. Результат вычисления этого выражения обязательно должен быть константой, хотя его компонентами могут быть и символические параметры, но их сочетание в выражении должно давать абсолютный результат. Константное выражение не должно содержать компоненты, которые транслятор еще не обработал к тому месту программы, где находится условная директива. Логические результаты «истина» и «ложь» являются условными в том смысле, что ноль соответствует логическому результату «ложь», а любое ненулевое значение – «истине». Однако в

языке ассемблера существуют операторы, которые позволяют сформировать и «чисто логический» результат – это операторы отношений. В контексте условных директив вместе с операторами отношений (таблица 2.1) можно рассматривать и логические операторы (таблица 2.2). Результатом работы тех и других может быть одно из двух значений: истина – число, которое содержит двоичные единицы во всех разрядах; ложь – число, которое содержит двоичные нули во всех разрядах.

Таблица 2.1 – Операторы отношения

| Оператор отношения                       | Синтаксис оператора        |
|------------------------------------------|----------------------------|
| EQ (equal) – равно                       | выражение_1 EQ выражение_2 |
| NE (not equal) – не равно                | выражение_1 NE выражение_2 |
| LT (less than) – меньше                  | выражение_1 LT выражение_2 |
| LE (less or equal) – меньше или равно    | выражение_1 LE выражение_2 |
| GT (greater than) – больше               | выражение_1 GT выражение_2 |
| GE (greater or equal) – больше или равно | выражение_1 GE выражение_2 |

Таблица 2.2 – Логические операторы

| Логический оператор        | Синтаксис оператора         |
|----------------------------|-----------------------------|
| NOT – логическое отрицание | NOT выражение               |
| AND – логическое И         | выражение_1 AND выражение_2 |
| OR – логическое ИЛИ        | выражение_1 OR выражение_2  |
| XOR – исключающее ИЛИ      | выражение_1 XOR выражение_2 |

### 2.3.6 Дополнительные средства управления трансляцией

Assembler предоставляет средства для вывода текстового сообщения во время трансляции программы – директивы DISPLAY и %OUT, с которых можно следить за ходом трансляции. Например,

```
display 'недопустимые аргументы макрокоманды'
%out 'недопустимое имя регистра'
```

В результате обработки этих директив на экран будут выведены тексты сообщений. Если эти директивы использовать совместно с директивами условной компиляции, то можно отслеживать путь, по которому осуществляется трансляция исходного текста программы.

## 2.4 Практическое задание

Используя макросредства языка ассемблера, задачу обработки вектора оформить с использованием макроопределений.

Пример. Задание 1 по логическим командам оформим с использованием макроопределений настройки сегмента данных, вывода строки

на экран, завершения программы. Это позволяет сократить код программы в 2 раза и улучшить читаемость кода.

init\_ds macro ;макрос настройки ds на сегмент данных

mov ax,dseg

mov ds,ax

endm

out\_str\_bx macro str,rg

;макрос вывода строки и целого числа на экран

push ax

mov ah,09h

mov dx,offset str

int 21h

mov ax,rg

call disp

pop ax

endm

out\_str macro str ;макрос вывода строки на экран

push ax

mov ah,09h

mov dx,offset str

int 21h

pop ax

endm

exit macro ;макрос конца программы

mov ax,4c00h

int 21h

endm

sseg segment para stack 'stack'

db 256 dup (0)

sseg ends

dseg segment para public 'data'

a dw 115 ;115=0000 0000 0111 0011

mes1 db 'Number of the first non-zero bit -->\$'

mes2 db 10,13,'Number of the last non-zero bit -->\$'

mes3 db 10,13,'The third bit is zero\$'

mes4 db 10,13,'The third bit is not zero\$'

mes5 db 10,13,'The third bit is equal to one-->\$'

mes6 db 10,13,'The fourth bit is zero\$'

mes7 db 10,13,'The fourth bit is not zero\$'

mes8 db 10,13,'The fourth bit is equal to zero-->\$'

mes9 db 10,13,'The two bit is zero\$'

mes10 db 10,13,'The two bit is not zero\$'

mes11 db 10,13,'The two bit is convert-->\$'

mes12 db 10,13,'Shift logical left-->\$'

```

mes13 db 10,13,'Shift logical right-->$'
mes14 db 10,13,'Shift arithmetic left-->$'
mes15 db 10,13,'Shift arithmetic left-->$'
mes16 db 10,13,'Rotate left-->$'
mes17 db 10,13,'Rotate right-->$'
mes18 db 10,13,'Rotate through carry left-->$'
mes19 db 10,13,'Rotate through carry right-->$'
mes20 db 10,13,'Logical command or, and, xor:$'
dseg ends
    extrn disp:near
cseg segment para public 'code'
    main proc near
        assume cs:cseg,ds:dseg,ss:sseg
        init_ds
.486                ;это обязательно
        mov ax,0002h
        int 10h
        mov ax,a ;сканирование бит вперед
        bsf bx,ax
        out_str_bx mes1,bx
        mov ax,a ;сканирование бит в обратном порядке
        bsr bx,ax
        out_str_bx mes2,bx
        mov bx,a ;проверка и установка третьего бита
        mov ax,3
        bts bx,ax
        jc m1      ;переход, если бит равен 1
        out_str mes3
        jmp m2
m1: out_str mes4
m2: out_str_bx mes5,bx
        mov bx,a ;проверка и сброс четвертого бита
        mov ax,4
        btr bx,ax
        jc m3 ;переход, если бит равен 1
        out_str mes6
        jmp m4
m3: out_str mes7
m4: out_str_bx mes8,bx
        mov bx,a ;проверка и инвертирование второго бита
        mov ax,2
        btc bx,ax
        jc m5 ;переход, если бит равен 1
        out_str mes9

```

```

    jmp m6
m5: out_str mes10
m6: out_str_bx mes11,bx
    mov ax,a ;ЛОГИЧЕСКИЙ СДВИГ ВЛЕВО
    shl ax,3
    mov bx,ax
    out_str_bx mes12,bx
    mov ax,a ;ЛОГИЧЕСКИЙ СДВИГ ВПРАВО
    shr ax,3
    mov bx,ax
    out_str_bx mes13,bx
    mov ax,a ;АРИФМЕТИЧЕСКИЙ СДВИГ ВЛЕВО
    sal ax,3
    mov bx,ax
    out_str_bx mes14,bx
    mov ax,a ;АРИФМЕТИЧЕСКИЙ СДВИГ ВПРАВО
    sar ax,3
    mov bx,ax
    out_str_bx mes15,bx
    mov ax,a ;ЦИКЛИЧЕСКИЙ СДВИГ ВЛЕВО
    rol ax,3
    mov bx,ax
    out_str_bx mes16,bx
    mov ax,a ;ЦИКЛИЧЕСКИЙ СДВИГ ВПРАВО
    ror ax,3
    mov bx,ax
    out_str_bx mes17,bx
    out_str mes20 ;ЛОГИЧЕСКИЕ КОМАНДЫ or,and,xor
    mov ax,a ;УСТАНОВКА ТРЕТЬЕГО БИТА
    or ax,1000b
    mov bx,ax
    out_str_bx mes5,bx
    mov ax,a ;СБРОС ЧЕТВЕРТОГО БИТА
    and ax,1111111111101111b
    mov bx,ax
    out_str_bx mes8,bx
    mov ax,a ;ИНВЕРТИРОВАНИЕ ВТОРОГО БИТА
    xor ax,100b
    mov bx,ax
    out_str_bx mes11,bx
    exit
main endp
cseg ends
end main

```

## 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.

Производственно-практическое издание

**Ружицкая** Елена Адольфовна

## **ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ ASSEMBLER**

### **Логические команды, макросредства**

Практическое пособие

Редактор *В. И. Шкредова*

Корректор *В. В. Калугина*

Подписано в печать 22.02.2017. Формат 60x84 1/16.

Бумага офсетная. Ризография. Усл. печ. л. 2,8.

Уч-изд. л. 3,1. Тираж 25 экз. Заказ 143.

Издатель и полиграфическое исполнение:

учреждение образования

«Гомельский государственный университет

имени Франциска Скорины».

Свидетельство о государственной регистрации издателя, изготовителя,  
распространителя печатных изданий № 1/87 от 18.11.2013.

Специальное разрешение (лицензия) № 02330 / 450 от 18.12.2013.

Ул. Советская, 104, 246019, Гомель.