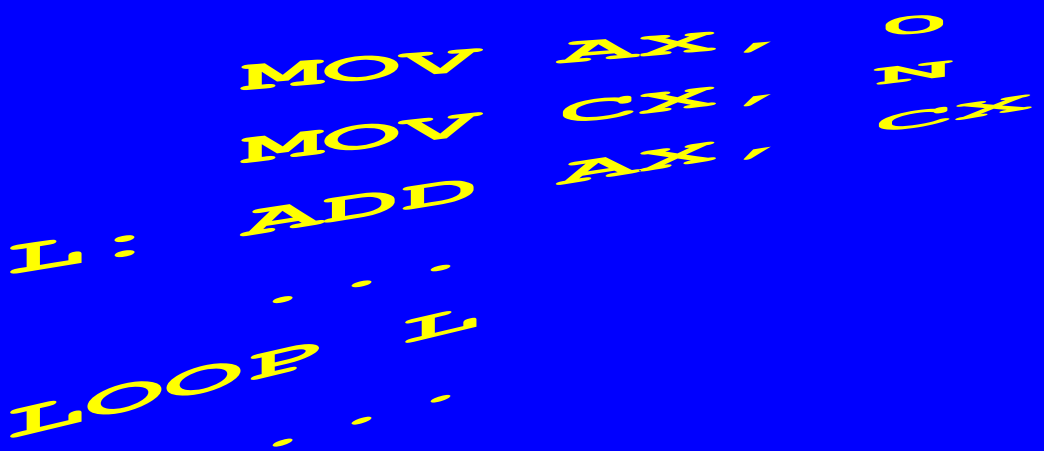
О.В. Гаркуша
Н.Ю. Добровольская

Ассемблер в примерах и задачах



Краснодар
2022

Министерство науки и высшего образования
Российской Федерации

КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

О.В. ГАРКУША
Н.Ю. ДОБРОВОЛЬСКАЯ

АССЕМБЛЕР В ПРИМЕРАХ И ЗАДАЧАХ

Учебное пособие

Краснодар
2022

УДК 004.431.4
ББК 32.973.2
Г 204

Рецензенты:
Доктор физико-математических наук, профессор
Е.Н. Калайдин
Кандидат физико-математических наук, доцент
С.Е. Рубцов

Гаркуша, О.В., Добровольская, Н.Ю.
Г 204 Ассемблер в примерах и задачах: учебное пособие / О.В. Гаркуша, Н.Ю. Добровольская; Министерство науки и высшего образования Российской Федерации, Кубанский государственный университет. – Краснодар: Кубанский гос. ун-т, 2022. – 134 с. – 500 экз.
ISBN 978-5-8209-2052-3

Изложены фундаментальные темы: организация современного компьютера, устройство процессоров семейства IA-32, синтаксис языка ассемблера, макросредства, программирование типовых управляющих структур, сложные структуры данных, оптимизация программ. Приведены многочисленные примеры, иллюстрирующие материал.

Адресуется студентам факультета компьютерных технологий и прикладной математики, изучающим основы программирования.

УДК 004.431.4
ББК 32.973.2

ISBN 978-5-8209-2052-3

ВВЕДЕНИЕ

Изучение архитектуры современных ПК, программирование на машинно-ориентированном языке — необходимая часть подготовки профессиональных программистов. Знание языка ассемблера позволяет лучше понять принципы работы ЭВМ, операционных систем и трансляторов с языков высокого уровня, разрабатывать высокоэффективные программы.

Masm32 – специализированный пакет для программирования на языке ассемблера IA-32. Являясь продуктом фирмы Microsoft, он максимально приспособлен для создания Windows-приложений на ассемблере. Кроме транслятора, компоновщика и необходимых библиотек пакет Masm32 включает сравнительно простой текстовый редактор и некоторые инструменты, предназначенные для облегчения программирования на ассемблере. Однако набор инструментов не содержит 32-разрядного отладчика и предполагает работу в командном режиме, что не очень удобно.

Для создания программ можно использовать специализированную интегрированную среду RADAsm, которая помимо других ассемблеров позволяет использовать Masm32. Точнее, используется специально настроенная среда – «сборка» RADAsm + OlleDBG, где OlleDBG – 32-разрядный отладчик.

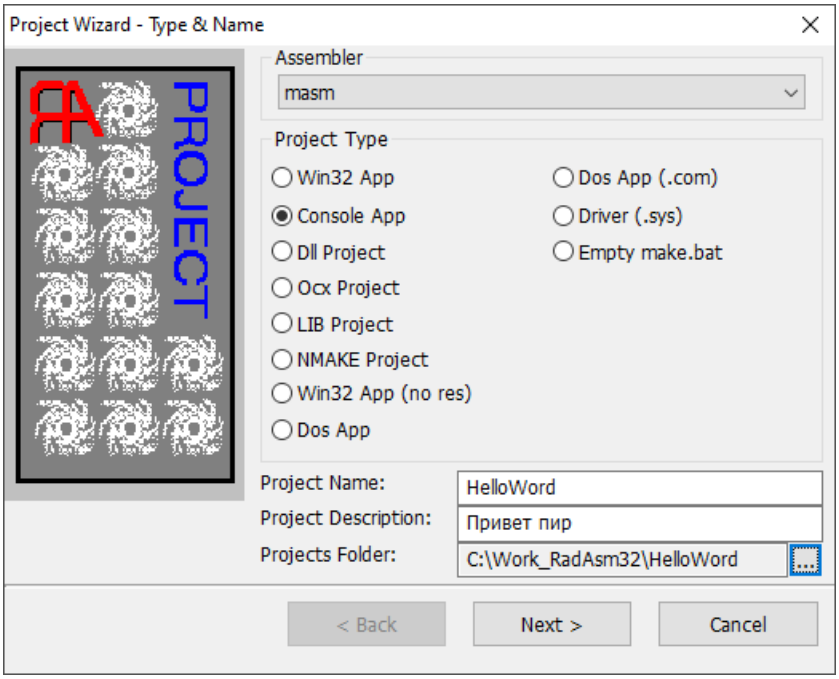
В учебном пособии рассматривается последовательность действий при разработке приложений на ассемблере в среде RADAsm, кроме того, указываются особенности архитектуры процессоров семейства IA-32.

1. НАЧАЛО РАБОТЫ СО СРЕДОЙ

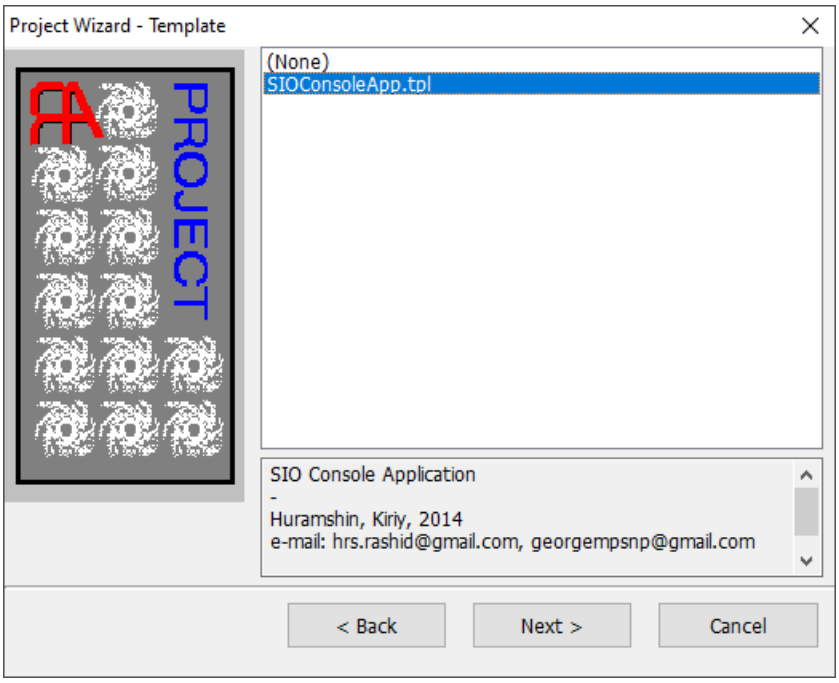
Программная среда инициируется запуском программы RadASM.exe.

Для создания нового проекта необходимо выбрать пункт меню File > New Project, после чего на экране появится первое окно Мастера создания проекта.

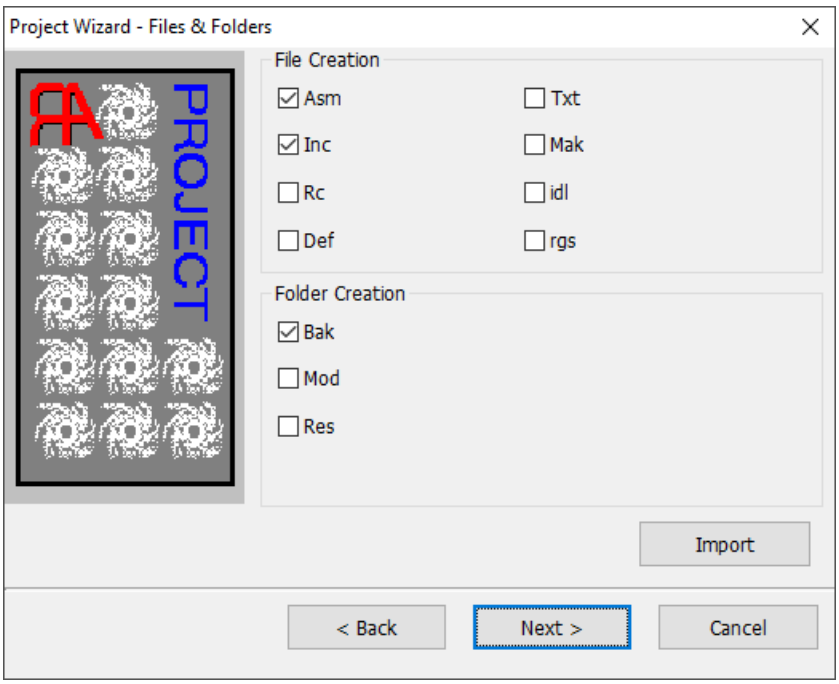
В этом окне необходимо выбрать тип проекта – в нашем случае Console App (консольное приложение), а также ввести его имя, например, HelloWord, описание, например, «Привет мир», и путь к создаваемой средой новой папке с именем проекта.



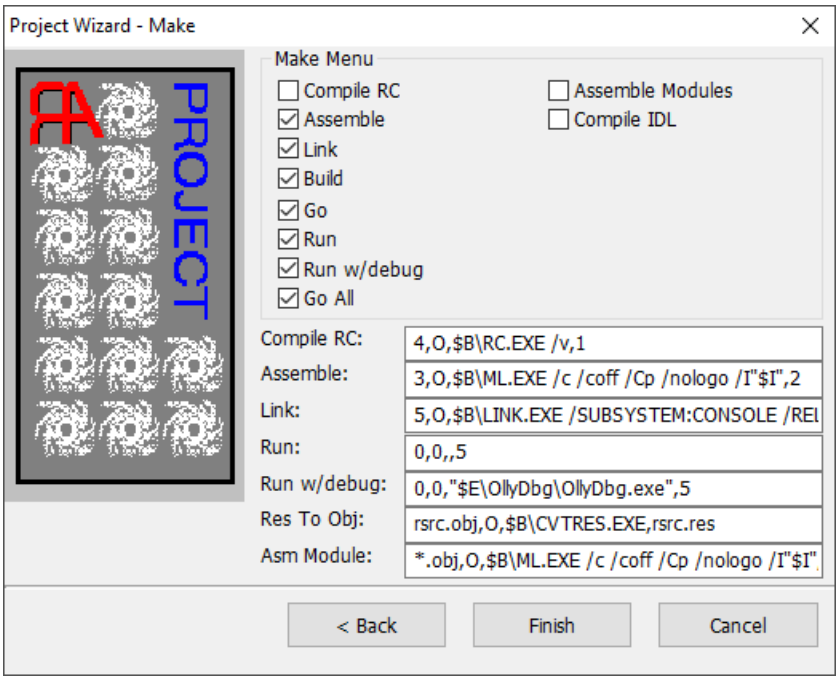
В следующем окне Мастера выбирается шаблон проекта (SIOConsoleApp.tpl), специально созданный для лабораторных работ шаблон консольного приложения.



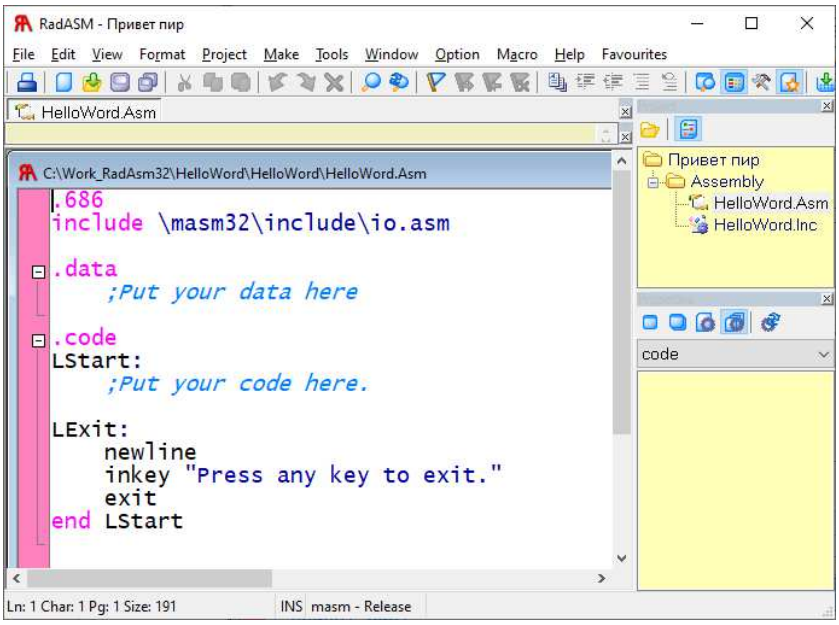
Далее предлагается выбрать типы создаваемых файлов — выбираем Asm (исходные файлы ассемблера), Inc (подключаемые библиотеки) и создаваемые папки — выбираем папку Bak для хранения предыдущих версий файлов.



По окончании создания проекта Мастер определяет доступные для работы с проектом пункты меню запуска приложения.



В этом окне рекомендуем использовать настройки по умолчанию.



Полученный шаблон консольного приложения Windows содержит:

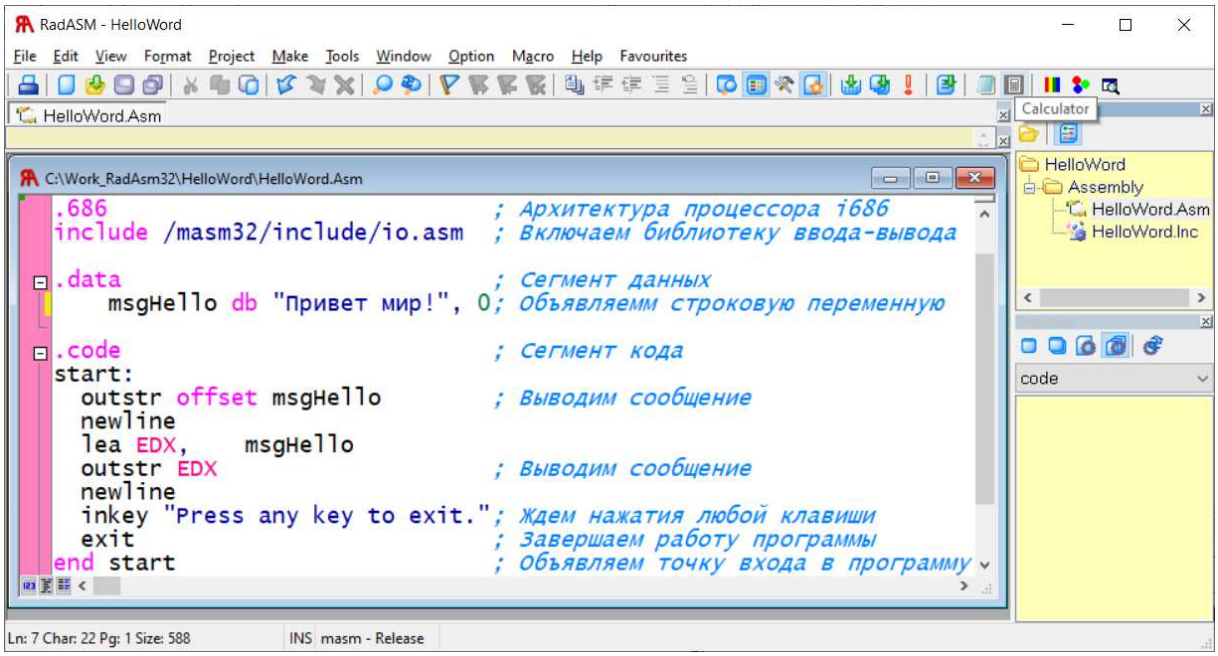
– директивы, определяющие набор команд и модель памяти;

– директивы подключения библиотек;

– разделы констант, инициализированных данных с минимально необходимыми директивами определения данных;

– раздел кода, обеспечивающий выход из программы.

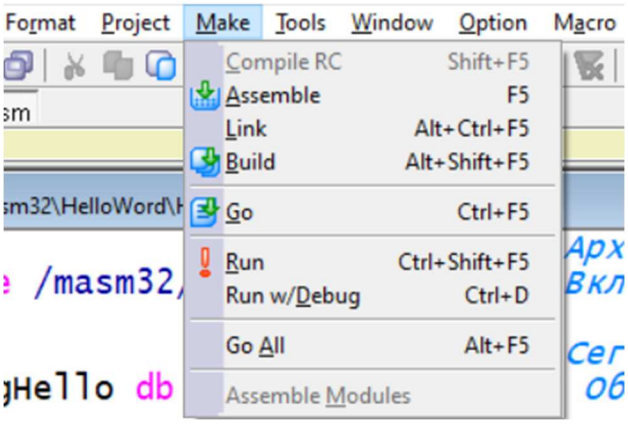
Добавим в шаблон описание строки, команду вывода этой строки на экран и команду ожидания нажатия любой клавиши клавиатуры для задержки вывода на экран результатов работы приложения. Получаем классический пример первой программы.



1.1. ФОРМИРОВАНИЕ ИСПОЛНЯЕМОГО ПРИЛОЖЕНИЯ

Для запуска шаблона необходимо выполнить:

– трансляцию Make > Assemble;
– компоновку Make > Link;
– запуск на выполнение Make > Run.



В процессе трансляции (ассемблирования) исходная программа на ассемблере преобразуется в двоичный эквивалент.

Если трансляция проходит без ошибок, то в окне Output, которое появляется под окном программы, выводится текст:

```
C:\Masm32\Bin\ML.EXE /c /coff /Cp /nologo
/I"C:\Masm32\Include" "HelloWord.asm"
 Assembling: HelloWord.asm

Make finished.
Total compile time 297 ms
```

Окно Output появляется на время ассемблирования и закрывается. Чтобы повторно увидеть результаты, необходимо курсор мыши перевести в нижнюю часть активного окна среды RadASM.

Первая строка сообщения об ассемблировании — вызов ассемблера:

`C:\Masm32\Bin\ML.EXE` — полное имя файла транслятора ассемблера masm32 (путь + имя), за которым следуют опции:

`/c` — заказывает ассемблирование без автоматической компоновки;

`/coff` — определяет формат объектного модуля Microsoft (coff);

`/Cp` — означает сохранение регистра строчных и прописных букв всех идентификаторов программы;

`/nologo` — осуществляет подавление вывода сообщений на экран в случае успешного завершения ассемблирования;

`/I"C:\Masm32\Include"` — определяет местонахождение вставляемых (.inc) файлов;

`"HelloWord.asm"` — имя обрабатываемого файла.

Остальные строки — сообщение о начале и завершении процесса ассемблирования и времени выполнения этого процесса.

Результатом нормального завершения ассемблирования является создание файла, содержащего объектный модуль программы, — файла `HelloWord.obj`.

Если при ассемблировании обнаружены ошибки, то объектный модуль не создается и после сообщения о начале ассемблирования идут сообщения об ошибках, например:

```
HelloWord.asm(14):error A2006: undefined symbol: EDY
```

В сообщении указывается:

- номер строки исходного текста (в скобках);
- номер ошибки, под которым она описана в документации;
- возможная причина.

После исправления ошибок процесс ассемблирования повторяют.

Следующий этап — компоновка программы. На этом этапе к объектному (двоичному) коду программы добавляются объектные коды используемых процедур. При этом в тех местах программы, где происходит вызов процедур, указывается их относительный адрес в модуле. Сведения о компоновке также выводятся в окно `Output`:

```
C:\Masm32\Bin\LINK.EXE /SUBSYSTEM:CONSOLE /RELEASE
/VERSION:4.0 /LIBPATH:"C:\Masm32\Lib"
/OUT:"HelloWord.exe" "HelloWord.obj"
Microsoft (R) Incremental Linker Version 5.12.8078
Copyright (C) Microsoft Corp 1992-1998. All rights
reserved.

Make finished.
Total compile time 109 ms
```

Первая строка вывода также является командной строкой вызова компоновщика:

`C:\Masm32\Bin\LINK.EXE` — полное имя компоновщика, за которым следуют опции:

`/SUBSYSTEM:CONSOLE` — подключить стандартное окно консоли;

`/RELEASE` — создать реализацию (а не отладочный вариант);

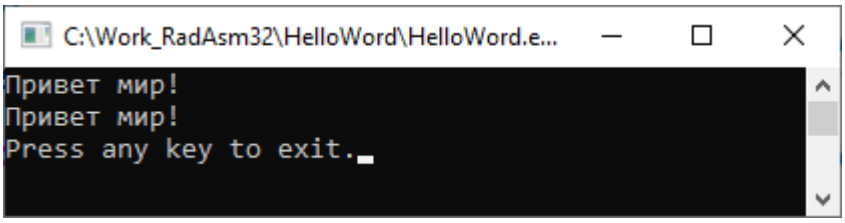
`/VERSION:4.0` — минимальная версия компоновщика;

`/LIBPATH:"C:\Masm32\Lib"` — путь к файлам библиотек;

`/OUT:"HelloWord.exe"` — имя результата компоновки — загрузочного файла и параметр `"HelloWord.obj"` — имя объектного файла.

После устранения ошибки программу необходимо перетранслировать и заново скомпоновать.

Если процессы трансляции и компоновки прошли нормально, то ее можно запустить на выполнение. При этом открывается окно консоли, в которое выводится строка запроса.



Окно закрывается при нажатии любой клавиши.

1.2. Структура программы на языке ассемблера

Программа на языке ассемблера имеет следующую структуру:

```
.686
.model flat, stdcall
option casemap: none

.data
   <инициализированные данные>

.data?
   <неинициализированные данные>

.const
   <константы>

.code
<метка>
   <код>
end <метка>
```

Директива `.686` указывает компилятору ассемблера, что необходимо использовать набор операций процессора определённого поколения.

Директива `.model` позволяет указывать используемую модель памяти и соглашение о вызовах. Как уже было сказано, на архитектуре `Win32` используется только одна модель памяти – `flat`, что и указано в приведённом примере. Соглашения о вызовах определяют порядок передачи параметров и порядок очистки стека.

Директива `option casemap: none` заставляет компилятор языка ассемблера различать большие и маленькие буквы в метках и именах процедур.

Директивы `.data, .data?, .const` и `.code` определяют то, что называется секциями. В `Win32` нет сегментов, но адресное пространство можно поделить на логические секции. Начало одной секции отмечает конец предыдущей. Есть две группы секций: данных и кода.

Секция `.data` содержит инициализированные данные программы.

Секция `.data?` содержит неинициализированные данные программы. Иногда нужно только предварительно выделить некоторое количество памяти, не инициализируя её. Эта секция для этого и предназначается. Преимущество неинициализированных данных в том, что они не занимают места в исполняемом файле. Вы всего лишь сообщаете компилятору, сколько места вам понадобится, когда программа загрузится в память.

Секция `.const` содержит объявления констант, используемых программой. Константы не могут быть изменены. Попытка изменить константу вызывает аварийное завершение программы.

Задействовать все три секции не обязательно.

Есть только одна секция для кода: `.code`. В ней содержится весь код.

Предложения `<метка>` и `end <метка>` устанавливают границы кода. Обе метки должны быть идентичны. Весь код должен располагаться между этими предложениями.

2. ЯЗЫК АССЕМБЛЕРА

Программа, написанная символическими мнемокодами, которые используются в языке ассемблер (ЯА), представляет собой исходный модуль. Для формирования исходного модуля применяют любой текстовый редактор. Затем программу подают на вход специальному транслятору, называемому ассемблером, который переводит ее на машинный язык, и далее полученную машинную программу выполняют.

При описании синтаксиса ЯА мы будем использовать формулы Бэкуса – Наура (БНФ) со следующими дополнениями:

– в квадратных скобках будем указывать конструкции, которые можно опускать; например, запись `A[B]C` означает либо текст `ABC`, либо текст `AC`;

– в фигурные скобки будем заключать конструкции, которые могут быть повторены любое число раз, в том числе и ни разу; например, запись `A{BC}` означает любой из следующих текстов: `A`, `ABC`, `ABCBC`, `ABCBCBC` и т. д.

2.1. Регистры процессоров семейства IA-32

К регистрам общего назначения (РОН) относится группа из 8 регистров, которые можно использовать в программе на языке ассемблера. Все регистры имеют размер 32 бита и могут быть разделены на 2 части или более.

Регистры данных (32 разряда)

	AH	AL	EAX
	BH	BL	EBX
	CH	CL	ECX
	DH	DL	EDX
	SI		ESI
	DI		EDI
	BP		EBP
	SP		ESP

Регистры `EAX`, `EBX`, `ECX` и `EDX` позволяют обращаться как к младшим 16 битам (по именам `AX`, `BX`, `CX` и `DX`), так и к двум младшим байтам по отдельности (по именам `AH/AL`, `BH/BL`, `CH/CL` и `DH/DL`).

Регистры `ESI`, `EDI`, `ESP` и `EBP` позволяют обращаться к младшим 16 битам по именам `SI`, `DI`, `SP` и `BP` соответственно.

Названия регистров происходят от их назначения:

`EAX/AX/AH/AL` (*accumulator register*) — аккумулятор;

`EBX/BX/BH/BL` (*base register*) — регистр базы;

`ECX/CX/CH/CL` (*counter register*) — счётчик;

`EDX/DX/DH/DL` (*data register*) — регистр данных;

`ESI/SI` (*source index register*) — индекс источника;

EDI/DI (*destination index register*) — индекс приёмника (получателя);

ESP/SP (*stack pointer register*) — регистр указателя стека;

EBP/BP (*base pointer register*) — регистр указателя базы стека.

Несмотря на существующую специализацию, все регистры можно использовать в любых машинных операциях. Однако надо учитывать тот факт, что некоторые команды работают только с определёнными регистрами. Например, команды умножения и деления используют регистры EAX и EDX для хранения исходных данных и результата операции. Команды управления циклом используют регистр ECX в качестве счётчика цикла.

Ещё один нюанс состоит в использовании регистров в качестве *базы*, т.е. хранилища адреса оперативной памяти. В качестве регистров базы можно использовать любые регистры, но желательно использовать регистры EBX, ESI, EDI или EBP. В этом случае размер машинной команды обычно бывает меньше.

К сожалению, количество регистров катастрофически мало, и зачастую бывает трудно подобрать способ их оптимального использования.

Любые регистры общего назначения могут использоваться для сложения и вычитания как 8-, 16-, так и 32-битовых значений.

2.1.1. Сегментные регистры CS, DS, SS и ES

Процессор имеет 6 так называемых *сегментных* регистров: CS, DS, SS, ES, FS и GS. Их существование обусловлено спецификой организации и использования оперативной памяти.

16-битные регистры могли адресовать только 64 Кб оперативной памяти, что явно недостаточно для более или менее приличной программы. Поэтому память программе выделялась в виде нескольких *сегментов*, которые имели размер 64 Кб. При этом *абсолютные* адреса были 20-битными, что позволяло адресовать уже 1 Мб оперативной памяти. Для решения задачи адресации 20-битных адресов 16-битными регистрами адрес разбивался на *базу* и *смещение*. База –адрес начала сегмента, а смещение –номер байта внутри сегмента. На адрес начала сегмента накладывалось ограничение – он должен быть кратен 16. При этом последние 4 бита были равны 0 и не хранились, а

подразумевались. Таким образом, получались две 16-битные части адреса. Для получения абсолютного адреса к базе добавлялись четыре нулевых бита, и полученное значение складывалось со смещением.

Сегментные регистры использовались для хранения адреса начала *сегмента кода* (`CS – code segment`), *сегмента данных* (`DS — data segment`) и *сегмента стека* (`SS – stack segment`). Регистры `ES`, `FS` и `GS` были добавлены позже. Существовало несколько моделей памяти, каждая из которых подразумевала выделение программе одного или нескольких сегментов кода и одного или нескольких сегментов данных: *tiny*, *small*, *medium*, *compact*, *large* и *huge*. Для команд языка ассемблера существовали определённые соглашения: адреса перехода сегментировались по регистру `CS`, обращения к данным сегментировались по регистру `DS`, а обращения к стеку – по регистру `SS`. Если программе выделялось несколько сегментов для кода или данных, то приходилось менять значения в регистрах `CS` и `DS` для обращения к другому сегменту. Существовали так называемые «ближние» и «дальние» переходы. Если команда, на которую надо совершить переход, находилась в том же сегменте, то для перехода достаточно было изменить только значение регистра `IP`. Такой переход назывался *ближним*. Если же команда, на которую надо совершить переход, находилась в другом сегменте, то для перехода необходимо было изменить как значение регистра `CS`, так и значение регистра `IP`. Такой переход назывался *дальним* и осуществлялся дольше.

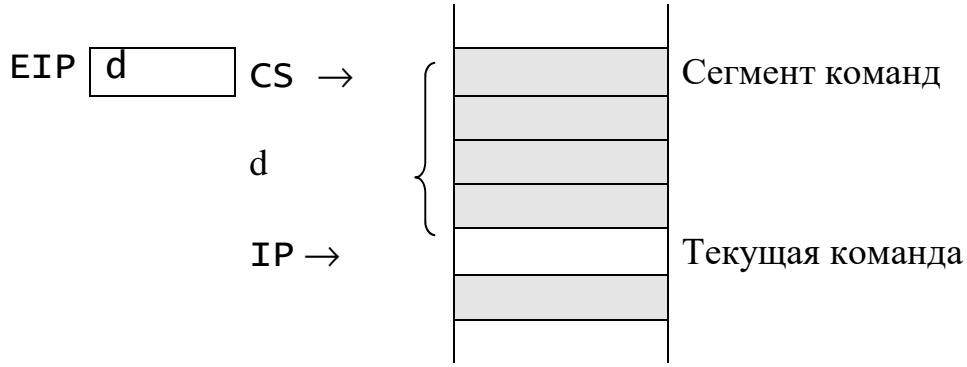
32-битные регистры позволяют адресовать 4 Гб памяти, что уже достаточно для любой программы. Каждую Win32-программу Windows запускает в отдельном виртуальном пространстве. Это означает, что каждая Win32-программа будет иметь 4-гигабайтовое адресное пространство, но вовсе не означает, что каждая программа имеет 4 Гб физической памяти, а только то, что программа может обращаться по любому адресу в этих пределах. А Windows сделает все необходимое, чтобы память, к которой программа обращается, «существовала».

Под архитектурой Win32 отпала необходимость в разделении адреса на базу и смещение и необходимость в моделях памяти. На 32-битной архитектуре существует только одна модель памяти —

flat (*сплошная* или *плоская*). Сегментные регистры остались, но используются по-другому.

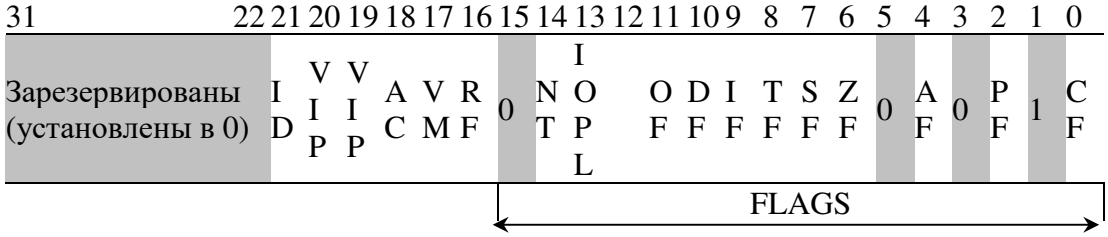
2.1.2. Регистр командного указателя EIP

Регистр указателя команд `EIP`[1] содержит смещение на команду, которая должна быть выполнена следующей.



2.1.3. Регистр флагов

Флаг — это бит, принимающий значение 1 («флаг установлен»), если выполнено некоторое условие, и значение 0 («флаг сброшен») в противном случае. Процессор имеет *регистр флагов*, содержащий набор флагов, отражающий текущее состояние процессора.



№	Флаг	Название	Описание	Тип флага
FLAGS				
0	CF	Carry Flag	Флаг переноса	Состояние
1	1		Зарезервирован	

[1] Instruction pointer – указатель команд.

№	Флаг	Название	Описание	Тип флага
2	PF	Parity Flag	Флаг чётности	Состояние
3	0		Зарезервирован	
4	AF	Auxiliary Carry Flag	Вспомогательный флаг переноса	Состояние
5	0		Зарезервирован	
6	ZF	Zero Flag	Флаг нуля	Состояние
7	SF	Sign Flag	Флаг знака	Состояние
8	TF	Trap Flag	Флаг трассировки	Системный
9	IF	Interrupt Enable Flag	Флаг разрешения прерываний	Системный
10	DF	Direction Flag	Флаг направления	Управляющий
11	OF	Overflow Flag	Флаг переполнения	Состояние
12	IOPL	I/O Privilege Level	Уровень приоритета ввода-вывода	Системный
13				
14	NT	Nested Task	Флаг вложенности задач	Системный
15	0		Зарезервирован	
EFLAGS				
16	RF	Resume Flag	Флаг возобновления	Системный
17	VM	Virtual-8086 Mode	Режим виртуального процессора 8086	Системный
18	AC	Alignment Check	Проверка выравнивания	Системный
19	VIF	Virtual Interrupt Flag	Виртуальный флаг разрешения прерываний	Системный
20	VIP	Virtual Interrupt Pending	Ожидающее виртуальное прерывание	Системный

№	Флаг	Название	Описание	Тип флага
21	ID	ID Flag	Проверка на доступность инструкции CPUID	Системный
22				
...			Зарезервированы	
31				

Значение флагов CF, DF и IF можно изменять напрямую в регистре флагов с помощью специальных инструкций (например, CLD для сброса флага направления), но нет инструкций, которые позволяют обратиться к регистру флагов как к обычному регистру. Однако можно сохранять регистр флагов в стек или регистр AH и восстанавливать регистр флагов из них с помощью инструкций LAHF, SAHF, PUSHF, PUSHFD, POPF и POPFD.

Флаги состояния (биты 0, 2, 4, 6, 7 и 11) отражают результат выполнения арифметических инструкций, таких как ADD, SUB, MUL, DIV.

– *Флаг переноса* CF устанавливается при переносе из старшего значащего бита / заёма в старший значащий бит и показывает наличие переполнения в беззнаковой целочисленной арифметике. Также используется в длинной арифметике;

– *Флаг чётности* PF устанавливается, если младший значащий байт результата содержит чётное число единичных битов. Изначально этот флаг был ориентирован на использование в коммуникационных программах: при передаче данных по линиям связи для контроля мог также передаваться бит чётности и инструкции для проверки флага чётности облегчали проверку целостности данных;

– *Вспомогательный флаг переноса* AF устанавливается при переносе из бита 3-го результата / заёма в 3-й бит результата. Этот флаг ориентирован на использование в двоично-десятичной (binary coded decimal, BCD) арифметике;

– *Флаг нуля* ZF устанавливается, если результат равен нулю;

– *Флаг знака* SF равен значению старшего значащего бита результата, который является знаковым битом в знаковой арифметике;

– *Флаг переполнения* OF устанавливается, если целочисленный результат слишком длинный для размещения в целевом операнде (регистре или ячейке памяти). Этот флаг показывает наличие переполнения в знаковой целочисленной арифметике.

Из перечисленных флагов только флаг CF можно изменять напрямую с помощью инструкций STC, CLC и CMC.

Флаги состояния позволяют одной и той же арифметической инструкции выдавать результат трёх различных типов: беззнаковое, знаковое и двоично-десятичное (BCD) целое число. Если результат считать беззнаковым числом, то флаг CF показывает условие переполнения (перенос или заём), для знакового результата перенос или заём показывает флаг OF, а для BCD-результата перенос / заём показывает флаг AF. Флаг SF отражает знак знакового результата, флаг ZF – и беззнаковый, и знаковый нулевой результат.

В длинной целочисленной арифметике флаг CF используется совместно с инструкциями сложения с переносом (ADC) и вычитания с заёмом (SBB) для распространения переноса или заёма из одного вычисляемого разряда длинного числа в другой.

Инструкции условного перехода Jcc (переход по условию cc), SETcc (установить значение байта-результата в зависимости от условия cc), LOOPcc (организация цикла) и CMOVcc (условное копирование) используют один или несколько флагов состояния для проверки условия. Например, инструкция перехода JLE (jump if less or equal – переход, если «меньше или равно») проверяет условие «ZF = 1 или SF ≠ OF».

Флаг PF был введён для совместимости с другими микропроцессорными архитектурами и по прямому назначению используется редко. Более распространено его использование совместно с остальными флагами состояния в арифметике с плавающей запятой: инструкции сравнения (FCOM, FCOMP и т. п.) в математическом сопроцессоре устанавливают в нём флаги-условия C0, C1, C2 и C3, и эти флаги можно скопировать в регистр

флагов. Для этого рекомендуется использовать инструкцию FSTSW AX для сохранения слова состояния сопроцессора в регистре AX и инструкцию SAHF для последующего копирования содержимого регистра AH в младшие 8 битов регистра флагов, при этом C0 попадает во флаг CF, C2 – в PF, а C3 – в ZF. Флаг C2 устанавливается, например, в случае несравнимых аргументов (NaN или неподдерживаемый формат) в инструкции сравнения FUCOM.

Флаг направления DF (бит 10 в регистре флагов) управляет строковыми инструкциями (MOVS, CMPS, SCAS, LODS и STOS) – установка флага заставляет уменьшать адреса (обрабатывать строки от старших адресов к младшим), обнуление заставляет увеличивать адреса. Инструкции STD и CLD соответственно устанавливают и сбрасывают флаг DF.

Системные флаги и поле IOPL управляют операционной средой и не предназначены для использования в прикладных программах.

– *Флаг разрешения прерываний* IF – обнуление этого флага запрещает отвечать на маскируемые запросы на прерывание;

– *Флаг трассировки* TF – установка этого флага разрешает пошаговый режим отладки, когда после каждой выполненной инструкции происходит прерывание программы и вызов специального обработчика прерывания;

– поле IOPL показывает уровень приоритета ввода-вывода исполняемой программы или задачи: чтобы программа или задача могла выполнять инструкции ввода-вывода или менять флаг IF, её текущий уровень приоритета (CPL) должен быть ≤ IOPL;

– *Флаг вложенности задач* NT – устанавливается, когда текущая задача «вложена» в другую, прерванную задачу, и сегмент состояния TSS текущей задачи обеспечивает обратную связь с TSS предыдущей задачи. Флаг NT проверяется инструкцией IRET для определения типа возврата – межзадачного или внутризадачного;

– *Флаг возобновления* RF используется для маскирования ошибок отладки;

– VM – установка этого флага в защищённом режиме вызывает переключение в режим виртуального 8086.

– *Флаг проверки выравнивания* AC – установка этого флага вместе с битом AM в регистре CR0 включает контроль выравнивания операндов при обращениях к памяти: обращение к невыровненному операнду вызывает исключительную ситуацию;

– VIF – виртуальная копия флага IF; используется совместно с флагом VIP;

– VIP – устанавливается для указания наличия отложенного прерывания;

– ID – возможность программно изменить этот флаг в регистре флагов указывает на поддержку инструкции CPUID.

2.2. Форматы машинных команд IA-32

2.2.1. Формат RR «регистр – регистр»

```
КОП r1, r2
```

Команды этого формата описывают действие `r1 = r1 * r2` или `r2 = r2 * r1`, где `r1` и `r2` – регистры общего назначения. Поле КОП (код операции) задает конкретную операцию (*).

2.2.2. Формат RS «регистр – память»

```
КОП reg, adr
```

Эти команды описывают операции

```
reg = reg * adr      или  adr = adr * reg,
```

где `reg` – регистр, а `adr` – адрес ячейки памяти.

2.2.3. Формат RI «регистр – непосредственный операнд»

Размер команд этого формата 3–4 байта.

Команды этого формата описывают операции

```
reg = reg * I,
```

где `reg` – регистр, а `I` – непосредственный операнд.

2.2.4. Формат SI «память – непосредственный операнд»

```
КОП adr, I
```

Команды этого формата описывают операции типа

```
adr = adr * I.
```

Смысл всех полей тот же, что и в предыдущих форматах.

Несомненно, записывать машинные команды ПК в цифровом виде с использованием перечисленных форматов команд достаточно сложно. Более высоким уровнем кодирования является уровень ассемблера, в котором программист пользуется символическими мнемокодами вместо машинных команд и описательными именами для полей данных и адресов памяти.

2.3. Идентификаторы

Понятие идентификатора в языке ассемблера ничем не отличается от понятия идентификатора в других языках. Можно использовать латинские буквы, цифры и знаки `_ . ? @ $`, причём точка может быть только первым символом идентификатора. Большие и маленькие буквы считаются эквивалентными.

2.4. Целые числа

В программе допускается использование целых чисел в десятичной, двоичной, восьмеричной и шестнадцатеричной системах счисления. Десятичные числа записываются как обычно, а при записи чисел в других системах в конце числа ставится спецификатор – буква, которая указывает, в какой системе записано это число: в конце двоичного числа ставится буква `b` (`binary`), в конце восьмеричного числа – буква `o` (`octal`) или буква `q`, в конце шестнадцатеричного числа – буква `h` (`hexadecimal`). Ради общности спецификатор, а именно букву `d` (`decimal`), разрешается указывать и в конце десятичного числа, но обычно этого не делают.

Примеры записи чисел:

10-чные	2-чные	8-ричные	16-ричные
25, +4, −386d	101b, −11000b	74q, −74q	1AFh, −1AFh

В случае использования шестнадцатеричных чисел необходимо применение следующего требования. Если шестнадцатеричное число начинается с цифры A – F, то в начале числа обязательно должен быть записан хотя бы один незначащий ноль: 0A5h — число, A5h — идентификатор. Такая запись позволяет определить, что рассматривается — число или идентификатор.

2.5. Внутреннее представление данных

Система команд ПК поддерживает работу с числами только размером байт и слово и частично размером в двойное слово.

Целые беззнаковые числа полностью используют соответствующее поле для представления значения числа в двоичной системе счисления. Для ячейки из k разрядов можно представить 2^k различных комбинаций.

Поэтому в байте можно представить целые числа 0–255 ($2^8 - 1$), в слове: 0–65535 ($2^{16} - 1$), в двойном слове: 0–4 294 967 295 ($2^{32} - 1$).

Знаковые целые числа представлены в дополнительном коде. Дополнительный код целого числа a:

$$D(a) = \begin{cases} a_2, & a \geq 0 \\ \left(2^k - |a|\right)_2, & a < 0 \end{cases}$$

Рассмотрим получение дополнительного кода числа −11 в поле байт:

$2^8 - 11 = 245_{(10)} = F5_{(16)} = 11110101_{(2)}$.

Или иначе:

Прямой код	11	0 0 0 0 1 0 1 1	0B
Обратный код		1 1 1 1 0 1 0 0	F4
	+	1	1
Дополнительный код	−11	1 1 1 1 0 1 0 1	F5

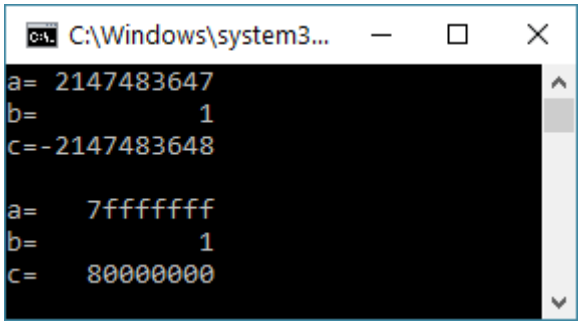
Получим сумму 11 + (− 11) = 0:

11	0 0 0 0 0 1 0 1 1	0B
-11	1 1 1 1 1 0 1 0 1	F5
1	0 0 0 0 0 0 0 0 0	0

При сложении положительных чисел можем получить отрицательное значение. Так, например, в языке C++ возможна такая ситуация:

```
#include <stdio.h>
#include <iostream>
using namespace std;
int main()
{
  int a = pow(2, 31)-1;
  int b = 1;
  int c = a + b;

  printf("a=%11d \n", a);
  printf("b=%11d \n", b);
  printf("c=%11d \n \n", c);

  printf("a=%11x \n", a);
  printf("b=%11x \n", b);
  printf("c=%11x \n", c);
  return 0;
}
```

Далее представлен результат работы программы как в 10-чном, так и в 16-ричном виде. При представлении в 16-ричном виде легко заметить, что изменяется значение знакового бита, т.е. число становится отрицательным.



Числа, размером слово или двойное слово, представляются аналогично. Однако в памяти байты полей хранятся в «обратном порядке». Так, двухбайтовое число –11 будет иметь вид FFF5, а храниться в памяти так (A — адрес слова)

A	A+1
F5	FF

При этом в регистрах числа размером слово хранится в «нормальном» порядке.

	BH	BL
BX	FF	F5

Для отображения данных в памяти или регистрах будем использовать шестнадцатеричную систему счисления.

Например, число 12345678h хранится в памяти так:

A	A+1	A+2	A+3
78	56	34	12

2.6. Символьные данные

Символы (последовательности символов) заключаются либо в одинарные, либо в двойные кавычки: 'A' или "A"; 'A+B' или "A+B".

В качестве символов можно использовать русские буквы.

В строках одноименные большие и малые буквы не отождествляются ('A+B' и 'a+B' — разные строки).

2.7. Описание данных

Все данные, используемые в программах на ассемблере, обязательно должны быть объявлены с использованием соответствующих директив, которые определяют тип данных и количество байт, необходимое для размещения этих данных в памяти:

```
[<Имя>] <Директива>[<Константа> DUP(
                    [<Список инициализаторов>])]
```

где: <Имя> — имя поля данных, которое может не присваиваться;

<Директива> — команда, объявляющая тип описываемых данных;

<Константа> DUP — используется при описании повторяющихся данных, тогда константа определяет количество повторений;

<Список инициализаторов> — последовательность инициализирующих констант через запятую или символ «?», если инициализирующее значение не определяется.

Директивы определения данных:

Директива	Описание типа данных
BYTE / SBYTE	8-разрядное целое без знака / со знаком
WORD / SWORD	16-разрядное целое без знака / со знаком
DWORD / SDWORD	32-разрядное целое без знака / со знаком или ближний указатель
FWORD	48-разрядное целое или дальний указатель
QWORD	64-разрядное целое
TBYTE	80-разрядное целое
REAL4	32-разрядное короткое вещественное
REAL8	64-разрядное длинное вещественное
REAL10	80-разрядное расширенное вещественное

В качестве директив также могут применяться:

- DB – определить байт;
- DW – определить слово;
- DD – определить двойное слово (4 байта);
- DQ – определить четыре слова (8 байт);
- DT – определить 10 байт.

Например:

```
A     DB    254   ; 0FEh
B     DB    -2    ; 0FEh (= 256 - 2 = 254)
C     DB    17h   ; 17h
```

Директива может содержать несколько констант, разделенных запятыми и ограниченных только длиной строки.

```
M     DB    2
      DB    -2
      DB    ?
      DB    '*'
```

А можно указать:

```
M     DB    2, -2, ?, '*'
```

Ассемблер определяет эти константы в виде последовательности смежных байтов и записывает в эти байты

значения операндов (для операнда ? ничего не записывает). В нашем примере ассемблер следующим образом заполнит память:

M	M+1	M+2	M+3
2	FE		2A

В массивах имя дается только первому элементу, а остальные оставляют безымянными. Если в директиве DB не указано имя, то по ней байт в памяти отводится, но он остается безымянным.

Например, для описания байтового массива R из 8 элементов с начальным значением 0 для каждого из них это можно сделать так:

```
R DB 0, 0, 0, 0, 0, 0, 0, 0
```

Или эту директиву можно записать короче:

```
R DB 8 DUP(0)
```

В общем случае эта конструкция имеет следующий вид:

```
A DB 4 DUP(1, 2) ; 1, 2, 1, 2, 1, 2, 1, 2
A DB 20 DUP(30 DUP(?))
```

Здесь резервируется 600 байт, значения которых неопределенны. Их можно рассматривать как место, отведенное для хранения элементов матрицы A размером 20×30, в которой элементы расположены в памяти следующим образом: первые 30 байтов – элементы первой строки матрицы, следующие 30 байтов – элементы второй строки и т. д.

Например:

```
B DW 1234h
C DW -2
```

При размещении числовых констант ассемблер автоматически меняет местами значения старшего и младшего байтов.

На ЯА такие числа записываются в нормальном, неперевернутом виде, а «переворачиванием» их занимается сам ассемблер, поэтому по нашим двум директивам память заполнится следующим образом:

B		C	
34	12	FE	FF

Символьный операнд в DW ограничен двумя символами, которые ассемблер представляет в «перевернутом» виде.

2.8. Директивы эквивалентности и присваивания

Рассмотрим, как в ЯА описываются константы. Это делается с помощью директивы эквивалентности.

2.8.1. Директива EQU[1]

Имеет следующий синтаксис:

```
<имя> EQU <операнд>
```

Здесь обязательно должны быть указаны имя и только один операнд.

Эта директива аналогична описанию константы в языке Паскаль:

```
Const <имя> = <операнд>;
```

С ее помощью программист информирует ассемблер о способе интерпретации некоторого имени.

Возможны три основных способа задания операндов:

Операнд — константное выражение

```
A EQU 10
```

В этом случае все последующие вхождения имени A в программу ассемблер заменяет на 10.

```
X DB A DUP(?)
Y DB A*5+1 ; резервируется 51 байт
```

Операнд — имя

```
A EQU N
```

[1] Equal – равно.

В этом случае имена A и N являются синонимами.

Операнд — произвольный текст, не являющийся константным выражением или именем. В этом случае всякое вхождение имени ассемблер заменяет на соответствующий текст.

Отметим, что директива EQU носит чисто информационный характер, по ней ассемблер ничего не записывает в машинную программу. Поэтому директиву EQU можно ставить в любое место программы — и между командами, и между описаниями переменных, и в других местах.

Примеры:

```
HELLOEQU ‘Большой привет’
LX EQU X + (N – 1)
WP EQU WORD PTR
```

В данном случае считается, что указанное имя обозначает операнд в том виде, как он записан (операнд не вычисляется). Именно на этот текст и будет заменяться каждое вхождение данного имени в программу. Например, следующие предложения эквивалентны

```
PR   DB   HELLO, ‘!’ ≡ PR EQU ‘Большой привет’,’!’
     NEG  LX         ≡        NEG X+(N-1)
     INC  WP[BX]     ≡        INC WORD PTR[BX]
```

Такой вариант директивы EQU обычно используется для того, чтобы ввести более короткие обозначения для часто встречающихся длинных текстов. Введя короткое имя, мы далее в программе можем им пользоваться, а уж ассемблер сам будет его заменять на соответствующий текст.

Отметим, что текст, указанный в правой части директивы EQU, должен быть сбалансирован по скобкам и кавычкам и не должен содержать вне скобок и кавычек символа «;». Кроме этого, поскольку текст не вычисляется, то в нем можно использовать как имена, описанные до этой директивы EQU, так и имена, описанные после нее.

2.8.2. Директива присваивания

```
<имя> = <константное выражение>
```

Директива определяет константу с именем, указанным в левой части, и с числовым значением, равным значению выражения справа. Но в отличие от констант, определенных по директиве EQU, данная константа может менять свое значение, принимая в тексте программы различные значения. Например:

```
K = 1
N EQU K ; N и K - синонимы
A DB N  ; A = 1
K = 2
A DB N  ; A = 2
```

Директива присваивания, в отличие от директивы эквивалентности, определяет только числовую константу. Кроме того, если имя указано в левой части директивы EQU, то оно не может появляться в левой части других директив (его нельзя переопределять). А имя, появившееся в левой части директивы присваивания, может снова появиться в начале другой такой директивы (но только такой!). Поэтому ошибочными являются три следующих фрагмента программы:

```
K EQU 1       K EQU 1         K = 1
K EQU 2       K = 2           K EQU 2
```

Появление в языке констант, которые могут менять свои значения, вносит некоторую неопределенность. Рассмотрим, к примеру, фрагмент слева:

```
  K = 1                   K = 1
N EQU K                 N EQU K + 10
A DW N  ; A = 1         C DW N       ; C = 11
  K = 2                   K = 2
B DW N  ; B = 2         D DW N       ; D = 11
```

Таким образом, необходимо внести следующее уточнение в действие EQU:

– если в правой части директивы указано имя константы, то имя слева надо понимать не как имя константы (не как имя числа), а как синоним имени справа;

– если же в правой части указано любое другое константное выражение, тогда имя слева действительно становится именем константы (обозначением числа).

Что же касается директивы присваивания, то ее правая часть всегда вычисляется сразу и полученное число тут же становится новым значением константы.

2.9. Структура программы

Программа на ЯА – это последовательность инструкций, каждая из которых записывается в отдельной строке.

Инструкции ЯА делятся на три группы:

– комментарии;
– директивы;
– команды.

2.9.1. Комментарии

Использование комментариев в программе улучшает ее ясность, поясняет смысл набора команд.

Комментарий начинается на любой строке программы со знака «;» (точка с запятой) и ассемблер полагает в этом случае, что все символы, находящиеся справа от «;», являются комментарием. Комментарий может содержать любые символы, включая пробел.

Комментарий может занимать всю строку или следовать за командой на той же строке. Например:

```
; Эта строка является комментарием
ADD AX, BX ; Комментарий вместе с командой
```

Комментарии появляются только в исходном коде программы и не приводят к генерации машинных кодов, поэтому можно включать любое количество комментариев, не влияя на эффективность выполнения программы.

В ЯА допустим и многострочный комментарий. Он должен начинаться со строки

```
COMMENT <маркер> <текст>
```

В качестве маркера берется первый за словом COMMENT символ, отличный от пробела; этот символ начинает комментарий. Концом такого комментария считается конец первой из последующих строк программы, в которой (в любой позиции) снова встретился этот же маркер. Например:

```
comment * все
это является комментарием * и это тоже
```

Такой вид комментария позволяет временно исключить из программы некоторый ее фрагмент (например, при отладке).

2.9.2. Директивы

Ассемблер имеет ряд операторов, которые позволяют управлять процессом ассемблирования. Эти операторы называются директивами. Они действуют только в процессе ассемблирования программы и не генерируют машинных кодов.

Синтаксис директив следующий:

```
[<имя>] <директивы> [<операнды>] [;<комментарий>]
```

Пример:

```
A DW 200 ; число A
```

Названия директив, как и мнемокоды, – служебные слова.

2.9.3. Команды

Основной формат кодирования команд языка ассемблер имеет следующий вид:

```
[<метка>:] <мнемокод> [<операнды>][;<комментарий>]
```

Метка (если имеется), команда и операнд (если имеется) разделяются по крайней мере одним пробелом или символом табуляции.

Примеры:

```
L1: MOV AX, 2  ; изменение значения
```

В качестве операндов команд языка ассемблера могут использоваться:

– регистры, обращение к которым осуществляется по именам;

– непосредственные операнды – константы, записываемые непосредственно в команде;

– ячейки памяти – в команде записывается адрес нужной ячейки.

Для задания адреса существуют следующие возможности:

– *Имя переменной* по сути является адресом этой переменной. Встретив имя переменной в операндах команды, компилятор понимает, что нужно обратиться к оперативной памяти по определённому адресу. Обычно адрес в команде указывается в квадратных скобках, но имя переменной является исключением и может быть указано как в квадратных скобках, так и без них. Например, для обращения к переменной `x` в команде можно указать `x` или `[x]`;

– если переменная была объявлена как массив, то к элементу массива можно обратиться, указав *имя* и *смещение*. Для этого существует ряд синтаксических форм, например:
`<имя>[<смещение>]` и `[<имя> + <смещение>]`.
Однако следует понимать, что смещение – это вовсе не индекс элемента массива. Индекс элемента массива – это его номер, и этот номер не зависит от размера самого элемента. Смещение же задаётся в байтах, и при задании смещения программист сам должен учитывать размер элемента массива;

– адрес ячейки памяти может храниться в регистре. Для обращения к памяти по адресу, хранящемуся в регистре, в команде указывается имя регистра в квадратных скобках, например, `[EBX]`. Как уже говорилось, в качестве регистров базы рекомендуется использовать регистры `EBX`, `ESI`, `EDI` и `EBP`;

– адрес может быть вычислен по определённой формуле. Для этого в квадратных скобках можно указывать достаточно сложные выражения, например, `[EBX+ECX]` или `[EBX+4*ECX]`.

2.10. Обозначения операндов команд

При описании команд будем пользоваться стандартными сокращениями.

i8, i16, i32	Непосредственные операнды (т.е. задаваемые в самой команде) длиной соответственно 8, 16 или 32 бита.
r8, r16, r32	Регистры общего назначения. r8 – байтовые регистры (AH, AL, BH и т.п.). r16 – регистры размером в слово (AX, BX, SI и т.п.). r32 – регистры размером в двойное слово (EAX, EBX, ESI и т.п.).
sr	Сегментные регистры (CS, DS, SS, ES).
m8, m16, m32	Адрес памяти соответствующей длины.

В описаниях команд языка ассемблера для обозначения возможных операндов будем использовать эти сокращения. Например:

```
ADD r8/r16/r32, r8/r16/r32     ; Регистр + Регистр
ADD r8/r16/r32, m8/m16/m32     ; Регистр + Память
ADD r8/r16/r32, i8/i16/i32     ; Регистр + неп.операнд
ADD m8/m16/m32, r8/r16/r32     ; Память + Регистр
ADD m8/m16/m32, i8/i16/i32     ; Память + Неп.операнд
```

Команды языка ассемблера обычно имеют 1 или 2 операнда или не имеют операндов вообще. Во многих, хотя не во всех, случаях операнды (если их два) должны иметь одинаковый размер. Команды языка ассемблера обычно не работают с двумя ячейками памяти.

Размер операндов при этом определяется:

– объемом регистра, хранящего число – если хотя бы один операнд находится в регистре;

– размером числа, заданным директивой определения данных;

– специальными описателями, например, BYTE PTR (байт), WORD PTR (слово) и DWORD PTR (двойное слово), если ни один операнд не находится в регистре и размер операнда отличен от размера, определенного директивой определения данных.

3. ВВОД И ВЫВОД

Для организации ввода-вывода удобно использовать библиотеку ввода вывода, которая была модифицирована из библиотеки io.asm для 16-битной версии для TAsm студентами 2-го курса ФКТиПМ Кирий Георгием и Хурамшиным Рашидом (`io.asm` — 2138 байтов). Эти ребята настолько увлеклись данной работой, что одному из них пришлось уйти в академический отпуск, а другому перевестись на другой факультет. Расширение возможностей и исправление неизбежных неточностей сделал Зырянов Максим (`io.asm` — 6760 байтов).

Репозиторий библиотеки и инсталляторов среды указан в [3].

Для использования процедур ввода-вывода следует подключить файл `io.asm`, который содержит необходимые макросы.

Обращаем внимание, что все идентификаторы **регистрозависимые**.

3.1. ВВОД. МАКРОСЫ inint*

Макрос	Описание
`inint8 op`	Ввод целого. `op – r8 | m8`
`inint16 op`	Ввод целого. `op – r16 | m16`
`inint32 op`	Ввод целого. `op – r32 | m32`
`inch op`	Ввод символа. `op – r8 | m8`
`inint op`	Ввод целого. `op – r32, m32`. Автоматически определяется размер `op`. Для вывода массива рекомендуется использовать `inint8, inint16, inint32.`

3.2. ВЫВОД. МАКРОСЫ outint*

Макрос	Описание
`outint8 op [,n]`	Вывод целого. `op – r8, m8` или `i8`.
`outint16 op [,n]`	Вывод целого. `op – r16, m16` или `i16`.
`outint32 op [,n]`	Вывод целого. `op – r32, m32` или `i32`.
Параметр `n` — (необязательный) количество позиций, отводимых для числа (если длина числа меньше, то дополняется пробелами).	

Макрос	Описание
`outint op [,n]`	Вывод целого. `op` – `r32, m32`. Автоматически определяется размер `op`. Для вывода массива рекомендуется использовать `outint8, outint16, outint32.`
`outch c`	Вывод символа. `c` – `r8, m8` или `i8`. Например, `outch 'A'`
`print [arg]`	Выводит строку `arg`. Если `arg` отсутствует, то макрос ждёт нажатия любой клавиши. Особенности: нет необходимости использовать `offset`. Нельзя выводить пустую строку. Необходимо использовать только двойные кавычки.
`println [arg]`	Аналогично `print`, но с переходом на новую строку после вывода.
`Read arg`	Если `arg` регистр или переменная размером в байт, то макрос считывает в неё код нажатой клавиши. Особенности: рекомендуется использовать в качестве аргумента регистр, так как тогда обеспечивается ввод по горизонтали. Нельзя использовать регистры или переменные размером больше, чем байт.
`readln arg`	Аналогично `read`, но с переходом на новую строку после ввода.

3.3. ДОПОЛНИТЕЛЬНЫЕ ВОЗМОЖНОСТИ

Макрос	Описание
`newline`	Перевод строки.
`outstr adr_str`	Вывод строки, которая расположена по адресу `adr_str` (т.е. `offset` строки). Строка обязательно должна оканчиваться 0 байтом. Например `Text DB "Text",0.`
`outstrln adr_str`	То же самое, с переходом на следующую строку.
`inkey [текст]`	Выводим `[текст]` и ждем нажатия любой клавиши.

4. КОМАНДЫ ЦЕЛОЧИСЛЕННОЙ АРИФМЕТИКИ

4.1. Оператор указания типа PTR[1]

Могут возникнуть ситуации, когда ассемблер не может однозначно определить размер пересылаемой величины.

```
MOV [SI], 0  ; В память по адресу, который
             ; содержится в SI, помещается 0
```

По этой команде ассемблер не может однозначно определить, что необходимо сформировать по адресу, содержащемуся в регистре SI: нулевой байт или нулевое слово. В таких случаях программист должен явно сообщить ассемблеру такую информацию с помощью оператора указания типа, который записывается следующим образом:

```
<тип> PTR <выражение>
```

где <тип> – это BYTE, WODR или DWORD, а выражение может быть константным или адресным.

```
MOVBYTE PTR [SI], 0    ;    нулевой байт
MOVWORD PTR [SI], 0    ;    нулевое слово
```

Можно записать

```
MOV [SI], BYTE PTR 0   ;    нулевой байт
```

Оператор PTR полезен в ситуации, когда требуется не уточнить, а изменить тип непосредственного операнда.

Пусть есть переменная

```
Z DW 1234h      ; Z: 34h, Z+1: 12h
```

и требуется записать 0 в первый байт этого слова (там, где хранится 34h). По команде MOV Z, 0 ноль запишется в оба байта.

Команда

```
MOV BYTE PTR Z, 0 ; Z: 00h, Z+1: 12h
```

позволяет рассматривать Z как байт только в данной конкретной команде.

[1] Pointer – указатель.

4.2. Команды пересылки MOV

Команда пересылки данных – пересылает число размером 1, 2 или 4 байта из op2 в op1:

```
MOV Op1, Op2
```

Примеры

```
MOV AX, BX
MOV ESI, 1000
MOV 0[DI], AL
MOV AX, code
MOV DS, AX
```

Команда перемещения и дополнения нулями **–** при перемещении значение op2 помещается в младшие разряды, а в старшие заносятся нули:

```
MOVZX op1, op2
```

Допустимые варианты

```
MOVZX r16/r32, r/m8
MOVZX r32, r/m16
```

Примеры

```
MOVZX EAX, BX
MOVZX SI, AH
```

Команда перемещения и дополнения знаковым разрядом – команда выполняется аналогично, но в старшие разряды заносятся знаковые биты:

```
MOVSX Op1, Op2
```

4.3. Команда обмена данных XCHG[1]

Меняет местами содержимое операндов (они должны быть одного размера).

```
XCHG Op1, Op2
```

Допустимые варианты

```
XCHG reg, reg
```

[1] Exchange – перестановка.

```
XCHG mem, reg
XCHG reg, mem
```

Например:

```
MOV AX, 10   ; AX=10
MOV SI, 15   ; SI=15
XCHG AX, SI  ; AX=15, SI=10
```

4.4. Команды работы со стеком PUSH и POP

Команды записи слова или двойного слова в стек и извлечения из стека.

```
PUSH i16 / i32 / r16 / r32 / m16 / m32
POP r16 / r32 / m16 / m32
```

Если в стек помещается 16-разрядное значение, то значение `ESP = ESP – 2`, если помещается 32-разрядное значение, то `ESP = ESP – 4`.

Если из стека извлекается 16-разрядное значение, то значение `ESP = ESP + 2`, если помещается 32-разрядное значение, то `ESP = ESP + 4`.

Примеры

```
PUSH SI
POP word ptr [EBX]
```

4.5. Команды сложения и вычитания

4.5.1. ADD[1], SUB[2]

```
ADD op1, op2 ; op1 = op1 + op2
SUB op1, op2 ; op1 = op1 – op2
```

Сложение и вычитание для чисел со знаком и чисел без знака выполняются по одному и тому же алгоритму. Поэтому программист сам должен учитывать, над каким типом целых чисел выполняются операции.

При выполнении операции результат может не помещаться в отведенное ему место. Тогда для контроля корректности

[1] Addition – сложение.

[2] Subtract – вычитание.

полученного результата следует анализировать содержимое битов флагового регистра CF (перенос) и OF (переполнение).

Если работа происходит с беззнаковыми числами, то следует анализировать флаг CF. Если же работаем со знаковыми числами, то анализируется флаг OF.

Если результат операции с беззнаковыми числами не помещается в соответствующий байт или слово, то CF = 1 (это означает, что результат сформирован по модулю 2^{16}).

Аналогичная ситуация для знаковых чисел приводит к установке OF = 1.

Термины «установить флаг» и «сбросить флаг» трактуются так:
«установить флаг» – флаг = 1;
«сбросить флаг» – флаг = 0.

Кроме указанных флагов, команды сложения или вычитания меняют флаги ZF и SF.

Если результат равен 0, то устанавливается флаг ZF, иначе ZF сбрасывается.

Если результат меньше 0, то устанавливается флаг SF, иначе SF сбрасывается.

4.5.2. INC[1], DEC[2]

```
INC op  ; op = op + 1
DEC op  ; op = op – 1
```

Иначе эти команды можно записать:

```
ADD op, 1    ; op = op + 1
SUB op, 1    ; op = op – 1
```

4.5.3. NEG[3]

```
NEG op  ; op = -op
```

[1] Increment – увеличение на 1.
[2] Decrement – уменьшение на 1.
[3] Negative – изменение знака.

Команда NEG рассматривает свой операнд как число со знаком и меняет его на противоположный.

Есть особый случай: если op байт и op=−128(80h), то операнд не меняется, так как нет знакового числа +128 (байт: −128..127). Аналогично если op слово и op=−32768 (8000h).

В этом особом случае флаг OF получает значение 1 (при других операндах 0). При нулевом операнде флаг CF = 0, при других – 1.

При этом обычным образом устанавливаются флаги SF и ZF.

4.5.4. ADC[1], SBB[2]

Следующие команды предназначены для моделирования сложения и вычитания длинных целых чисел, т.е. целых чисел, занимающих двойное слово.

Сложение с переносом

```
ADC op1, op2 ; op1 = op1 + op2 + CF
```

Вычитание с учетом заёма

```
SBB op1, op2 ; op1 = op1 - op2 - CF
```

При выполнении ADC к результату прибавляется содержимое флага CF, а при SBB из результата вычитается содержимое флага CF.

```
  xxx…x 1xx…x
+ xxx…x 1xx…x      CF=1
  xxx…x 0xx…x
```

Пример 1. Сложение «длинных» чисел.

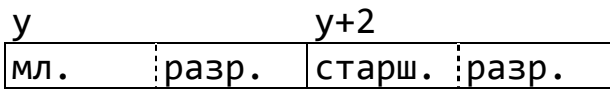
```
X DD ?  ; x, y – некоторые числа
Y DD ?
Z DD ?  ; z = x + y
```

Напомним, что значения ассемблер записывает в память в «перевернутом» виде.

x		x+2	
мл.	разр.	старш.	разр.

[1] Add with carry – сложение с переносом.

[2] Subtract with borrow – вычитание с учетом заёма.




```
MOV AX, WORD PTR x
ADD AX, WORD PTR y      ; CF = 0 | 1

MOV BX, WORD PTR x+2
ADC BX, WORD PTR y+2

MOV WORD PTR z, AX
MOV WORD PTR z+2, BX
```

Аналогично можно реализовать вычитание «длинных» беззнаковых целых чисел.

4.6. Изменение размера регистров

В операциях деления размер делимого в два раза больше, чем размер делителя. Поэтому нельзя просто загрузить данные в регистр EAX и поделить его на какое-либо значение, так как в операции деления будет задействован также и регистр EDX. Поэтому прежде чем выполнять деление, надо установить корректное значение в регистр EDX, иначе результат будет неправильным. Значение регистра EDX должно зависеть от значения регистра EAX. Тут возможны два варианта — для знаковых и беззнаковых чисел.

Если мы используем беззнаковые числа, то в любом случае в регистр EDX необходимо записать значение 0:

```
AAAAAAAAh → 00000000AAAAAAAAh.
```

Если же мы используем знаковые числа, то значение регистра EDX будет зависеть от знака числа:

```
55555555h → 0000000055555555h,
AAAAAAAAh → FFFFFFFFAAAAAAAAh.
```

Записать значение 0 не сложно, а вот для знакового расширения необходимо анализировать знак числа. Однако нет необходимости делать это вручную, так как язык ассемблера имеет

ряд команд, позволяющих расширять байт до слова, слово до двойного слова и двойное слово до учетверённого слова.

```
CBW ; байт в слово AL → AX
CWD ; слово в двойное слово AX → DX:AX
CWDE ; слово в двойное слово AX → EAX
CDQ ; двойное слово в учетверенное EAX → EDX:EAX
```

Таким образом, если делитель имеет размер 2 или 4 байта, то нужно устанавливать значение не только регистра AX/EAX, но и регистра DX/EDX. Если же делитель имеет размер 1 байт, то можно просто записать делимое в регистр AX.

```
x DD ?
  MOV EAX, x    ; EAX = x, которое заранее неизвестно
  CDQ           ; Знаковое расширение EAX в EDX:EAX
```

В языке ассемблера существуют также команды, позволяющие занести в регистр значение другого регистра или ячейки памяти со знаковым или беззнаковым расширением.

```
MOVSX op1, op2; Заполнение знаковым битом
MOVZX op1, op2; Старшие биты заполняются нулём
```

op1, op2 могут иметь любой размер. Понятно, что op1 должен быть больше, чем op2. В случае равенства размера операндов следует использовать обычную команду пересылки MOV, которая выполняется быстрее.

4.7. Команды умножения и деления

В отличие от сложения и вычитания, умножение и деление для беззнаковых и знаковых чисел выполняются по разным алгоритмам.

4.7.1. Команды умножения

Операции умножения для беззнаковых данных выполняются командой MUL[1], а для знаковых – IMUL[2].

[1] Multiply – умножение.

[2] Integer multiply – целочисленное умножение.

Эти команды реализованы для процессора 80186+.

```
MUL op  ; умножение целых без знака
IMUL op ; умножение целых со знаком
```

Если размерность операнда op 8 бит, то команды MUL и IMUL умножают содержимого регистра AL на значение операнда и помещают результат в регистр AX.

Если операнд – 16-битное слово, команды MUL и IMUL умножают содержимого регистра AX на значение операнда и помещают результат в пару регистров DX:AX.

Если операнд – двойное слово, команды MUL и IMUL умножают содержимого регистра EAX на значение операнда и помещают результат в пару регистров EDX:EAX.

Команда знакового умножения имеет еще две формы.

```
IMUL op1, op2;  op1 = op1 * op2
  op1: (r16, r32)
  op2: (r16, m16, r32, m32, i8, i16, i32)
```

Команда выполняет умножение первого операнда на второй и помещает результат в первый операнд.

Разрядность операндов должна совпадать. Исключением является использование в качестве второго операнда непосредственного 8-битного значения.

```
IMUL op1, op2, op3;  op1 = op2 * op3
  op1: (r16, r32)
  op2: (r16, m16, r32, m32)
  op3: (i8, i16, i32)
```

Команда выполняет умножение второго операнда на третий операнд и помещает результат в первый операнд.

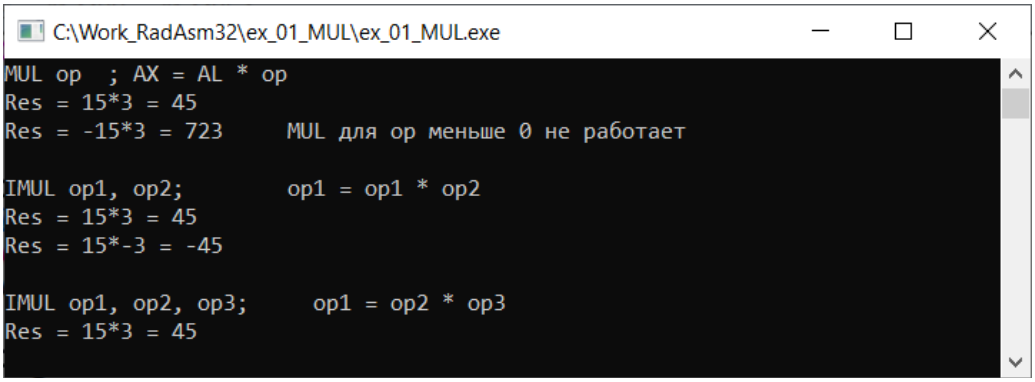
Разрядность операндов должна совпадать. Исключением является использование в качестве второго операнда непосредственного 8-битного значения.

Ответственность за контроль над форматом обрабатываемых чисел и за выбор подходящей команды умножения лежит на программисте.

```
; Пример использования MUL, IMUL
```

```
.686
include /masm32/include/io.asm


.data
A_DB DB ?
A_DW DW ?

.code
start:
; MUL op = = = = = = = = = = = = = = = = = = = = = = = =
  println "MUL op  ; AX = AL * op"
  ; op1 (r8, r16, r32)
  MOV AL,   15
  MOV A_DB, 3
  print "Res = "
  outint8 AL
  print "*"
  outint8 A_DB
  print " = "

  MUL A_DB    ; AX = AL * A_DB
  outint16 AX
  newline

  MOV AL, -15
  MOV A_DB, 3

  print "Res = "
  outint8 AL
  print "*"
  outint8 A_DB
  print " = "

  MUL A_DB    ; AX = AL * A_DB
  outint16 AX
  println "   !!!  MUL для op меньше 0 не работает"
  newline

; IMUL op1, op2 = = = = = = = = = = = = = = = = = = = =
  println "IMUL op1, op2;      op1 = op1 * op2"
  ; op1: (r16)
  ; op2: (r16)
  MOV AX, 15
  MOV BX, 3

  print "Res = "
  outint16 AX
  print "*"
  outint16 BX
```

```
  print " = "

  IMUL AX, BX     ; AX = AX * BX
  outint16 AX
  newline

  MOV EAX, 15
  MOV EBX, -3

  print "Res = "
  outint32 EAX
  print "*"
  outint32 EBX
  print " = "
  IMUL EAX, EBX     ; EAX = EAX * EBX
  outint32 EAX
  newline

; IMUL op1, op2, op3;     op1 = op2 * op3 = = = = = =
  println "IMUL op1, op2, op3;op1 = op2 * op3"
  ; op1: (r8, r16, r32)
  ; op2: (i8, i16, i32)
  ; op3: (r8, r16, r32), (m8,m16,m32), (i8,i16,i32)
  MOV AX, 15
  MOV A_DW, 3

  print "Res = "
  outint16 AX
  print "*"
  outint16 A_DW
  print " = "

  IMUL BX, AX, 3   ; BX = AX * 3
  outint16 BX
  newline

  inkey          ; ожидание нажатия клавиши
  exit
end start
```



Команда MUL при AL = -15 и op=3 дает результат 723, так как отрицательное значение -15 представлено в дополнительном коде:

15 Прямой код $00001111_{(2)}$
Инверсия $11110000_{(2)}$
+1
-15 Дополнительный код $11110001_{(2)}$ = F1h

В десятичной системе счисления: $F1h = 15*16+1 = 241_{(10)}$, тогда умножение 241*3=723.

4.7.2. Команды деления

Как и умножение, деление чисел без знака и со знаком также реализуется двумя командами:

```
DIV[1] op  ; деление целых без знака
IDIV[2] op ; деление целых со знаком
```

Деление слова на байт (op DB ?):

```
AH = AX % op,          AL = AX / op
```

Деление двойного слова на слово (op DW ?):

```
DX = (DX,AX) % op,     AX = (DX,AX) / op
```

Деление четверного слова на двойное слово (op DD ?):

```
EDX=(EDX:EAX) % op,    EAX=(EDX:EAX) / op
```

При делении слова на байт делимое находится в регистре AX, а делитель – в байте памяти или в однобайтовом регистре. После деления остаток получается в регистре AH, а частное – в AL. Так как однобайтовое частное очень мало – максимально 255 для беззнакового деления и 127 для знакового, то данная операция имеет ограниченное использование.

При делении двойного слова на слово делимое находится в регистровой паре DX:AX, а делитель – в слове памяти или регистре. После деления остаток получается в регистре DX, а частное – в

[1] Divide – деление.
[2] Integer divide – целочисленное деление.

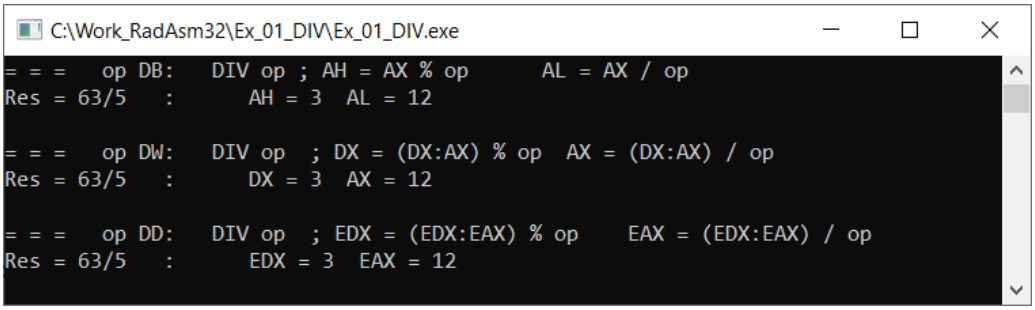
регистре AX. Частное в одном слове допускает максимальное значение 65535 для беззнакового деления и 32767 для знакового.

В единственном операнде команд DIV и IDIV указывается делитель.

Пример 2. Использование DIV

```
.686
include /masm32/include/io.asm
.data
A_DB DB ?
A_DW DW ?
A_DD DD ?
.code
start:
; op DB = = = = = = = = = = = = = = = = = = = = = = = = = =
  println "op DB: DIV op ; AH=AX%op   AL=AX/op"

  MOV AL, 63 ; AL = 3Fh = 63
  MOV A_DB, 5
  MOV AH, 0 ; !!!

  print "Res = "
  outint16 AX
  print "/"
  outint8 A_DB
  print "   :   "

  DIV A_DB  ; AH = AX%A_DB        AL=AX/A_DB
  print "   AH = "
  outint8 AH
  print "  AL = "
  outint8 AL
  newline

; op DW = = = = = = = = = = = = = = = = = = = = = = = = = =
  println "op DW: DIV op; DX=(DX:AX)%op
AX=(DX:AX)/op"

  MOV AX, 63  ; AX = 003Fh = 63
  MOV A_DW, 5
  MOV DX, 0  ; !!!

  print "Res = "
  outint16 AX
  print "/"
  outint16 A_DW
  print "   :   "

  DIV A_DW  ; DX=(DX:AX)%A_DW  AX=(DX:AX)/A_DW
```

```
  print "   DX = "
  outint16 DX
  print "  AX = "
  outint16 AX
  newline
; op DD = = = = = = = = = = = = = = = = = = = = = = = = = =
  println "op DD: DIV op; EDX=(EDX:EAX)%op
EAX=(EDX:EAX)/op"

  MOV EAX, 63
  MOV A_DD, 5
  MOV EDX, 0  ; !!!

  print "Res = "
  outint16 AX
  print "/"
  outint32 A_DD
  print "   :   "

  DIV A_DD    ; DX = (EDX:EAX) % A_DD
              ; AX = (EDX:EAX) / A_DD

  print "   EDX = "
  outint32 EDX
  print "  EAX = "
  outint32 EAX

  newline
  inkey         ; Ожидание нажатия клавиши
  exit
end start
```



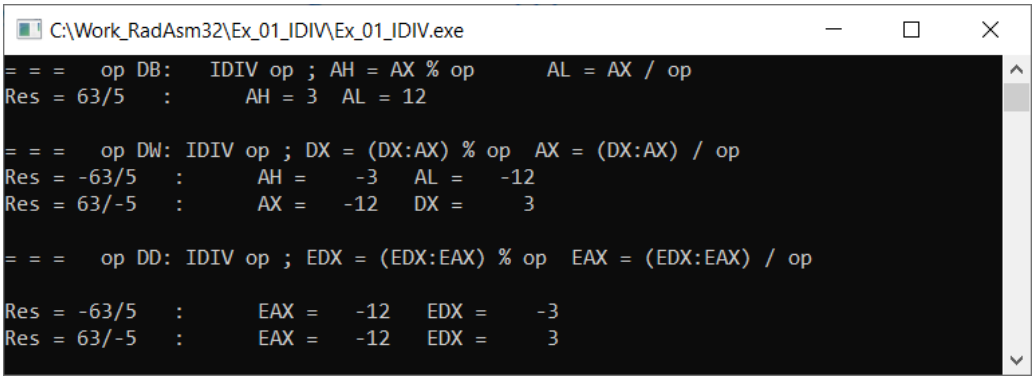
Пример 3. Использование IDIV

```
.686
include /masm32/include/io.asm
; .data - Сегмент данных не описываем,
; т.к. используются регистры
.code

start:
```

```
; op DB = = = = = = = = = = = = = = = = = = = = = = = = = = =
  println "= = =   op DB: IDIV op; AH = AX % op
                                  AL = AX / op"
  MOV AL, 63 ; AL = 3Fh = 63
  MOV BL, 5
  CBW  ; Знаковое расширение AL до AX:  AX=003Fh = 63
       ; !!! В данном случае не обязательно !!!
  print "Res = "
  outint16 AX
  print "/"
  outint8 BL
  print "   :   "

  ; AX = 003Fh = 63
  IDIV   BL     ; AH = 03h = 3,  AL = 0Ch = 12
  print "   AH = "
  outint8 AH
  print "  AL = "
  outint8 AL
  newline

; op DW = = = = = = = = = = = = = = = = = = = = = = = = = = =
  println "= = = op DW: IDIV op; DX = (DX:AX) % op
                                  AX = (DX:AX) / op"
  MOV AL, -63 ; AL = C1h = -63
  MOV BL, 5
  CBW ; Знаковое расширение AL до AX: AX=FFC1h=-63
  print "Res = "
  outint8 AL
  print "/"
  outint8 BL
  print "   :   "

  ; AX = FFC1h = -63
  IDIV   BL     ; AH = 03h = 3,  AL = F4h = -12
  print "   AH = "
  outint8 AH, 5
  print "   AL = "
  outint8 AL, 5
  newline

; op DD = = = = = = = = = = = = = = = = = = = = = = = = = = =
  MOV AX, 63  ; AX = 003Fh = 63
  MOV BX, -5
  CWD
     ; Знаковое расширение AX до (DX:AX)=0000003Fh=63
     ; !!! В данном случае не обязательно !!!
  print "Res = "
  outint16 AX
  print "/"
```

```
  outint16 BX
  print "   :   "

  IDIV   BX    ; DX = 0003h = 3,  AX = FFF4h = -12
  print "   AX = "
  outint16 AX, 5
  print "   DX = "
  outint16 DX, 5
  newline

; op DD = = = = = = = = = = = = = = = = = = = = = = = =
  println "= = op DD: IDIV op; EDX = (EDX:EAX) % op
                              EAX = (EDX:EAX) / op"
  newline

  MOV EAX, -63 ; EAX = FFFF FFC1h = -63
  MOV EBX, 5
  CDQ ; Знаковое расширение EAX до (EDX:EAX)
      ; (EDX:EAX)  = FFFF FFFF FFFF FFC1h = -63
  print "Res = "
  outint32 EAX
  print "/"
  outint32 EBX
  print "   :   "

  IDIV   EBX ; EDX=FFFF FFFDh=-3, EAX=FFFF FFF4h=-12

  print "   EAX = "
  outint32 EAX, 5
  print "   EDX = "
  outint32 EDX, 5
  newline

; op DD = = = = = = = = = = = = = = = = = = = = = = = =
  MOV EAX, 63 ; EAX = 0000 003Fh = 63
  MOV EBX, -5
  CDQ ; Знаковое расширение EAX до (EDX:EAX)
      ; (EDX:EAX) = 0000 0000 0000 003Fh = 63
      ; !!! В данном случае не обязательно !!!
  print "Res = "
  outint32 EAX
  print "/"
  outint32 EBX
  print "   :   "

  IDIV   EBX ; EDX = 0000 0003h=3, EAX=FFFF FFF4h=-12
  print "   EAX = "
  outint32 EAX, 5
  print "   EDX = "
  outint32 EDX, 5
```

```
  newline

  inkey          ; ожидание нажатия клавиши
  exit
end start
```



Отметим, что при выполнении деления возможно появление ошибки «деление на 0 или переполнение». Она возникает в двух случаях:

– делитель равен 0 (`op = 0`);

– неполное частное не вмещается в отведенное ему место (регистр `AL` или `AX`); это, например, произойдет при делении 600 на 2:

```
  MOV AX, 600
  MOV BH, 2
  DIV BH; 600 div 2=300, но 300 не вмещается в AL
```

При такой ошибке ПК прекращает выполнение программы.

Пример 4. Дано целое трехзначное беззнаковое число. Найти сумму цифр этого числа.

```
X   DB ?
Sum DB 0
Ten DB 10
  MOV AL, X     ; Исходное число
  SUB AH, AH
  DIV Ten       ; Деление на 10
  ADD Sum, AH   ; AH – последняя цифра
  SUB AH, AH    ; Очистка AH
  DIV Ten       ; Деление на 10
  ADD Sum, AH   ; AH – средняя цифра
  ADD Sum, AL   ; AL – первая цифра числа
                ; Вывод Sum – Сумму цифр числа
```

5. ПЕРЕХОДЫ И ЦИКЛЫ

Процессор выполняет команды машинной программы в том порядке, как они записаны в памяти.

Выполнение любой команды начинается с того, что содержимое регистра `EIP/IP` увеличивается на длину текущей команды и, таким образом, в регистре адресов команд оказывается адрес следующей команды.

Если команда во время своего выполнения меняет содержимое `EIP/IP`, то в результате за данной командой будет выполняться не обязательно следующая команда.

Такие команды называются командами перехода, или командами передачи управления.

Отметим, что команды перехода не изменяют флаги: какое значение флаги имели до команды перехода, такое же значение они будут иметь и после нее.

5.1. Безусловный переход

Команда безусловного перехода имеет следующий синтаксис:

```
JMP[1] op     ; безусловный переход
```

Операнд op указывает адрес перехода. Существует два способа указания этого адреса, соответственно различают *прямой* и *косвенный* переходы.

5.1.1. Прямой переход

Если в команде перехода указывается метка команды, на которую надо перейти, то переход называется ***прямым***.

```
    JMP   L
    ...
L:  MOV   EAX, x
```

[1] Jump – прыжок.

Вообще, любой переход заключается в изменении адреса следующей исполняемой команды, т.е. в изменении значения регистра EIP/IP.

Запись в команде перехода не абсолютного, а относительного адреса перехода позволяет уменьшить размер команды перехода. Абсолютный адрес должен быть 32-битным, а относительный может быть и 8-битным, и 16-битным.

5.1.2. Косвенный переход

При ***косвенном*** переходе в команде указывается не адрес перехода, а регистр или ячейка памяти, где этот адрес находится. Содержимое как абсолютный адрес перехода. Косвенные переходы используются в тех случаях, когда адрес перехода становится известен только во время работы программы.

```
JMP    EBX
```

5.2. Команды сравнения и условного перехода

Команды условного перехода осуществляют переход, который выполняется только в случае истинности некоторого условия. Истинность условия проверяется по значениям флагов. Поэтому обычно непосредственно перед командой условного перехода ставится команда ***сравнения***, которая формирует значения флагов:

```
CMP[1] op1, op2
```

Команда сравнения эквивалентна команде SUB за исключением того, что вычисленная разность никуда не заносится. Назначение команды CMP – установка и сброс флагов.

Все команды условного перехода записываются единообразно:

```
Jxx <метка>
```

Все команды условного перехода можно разделить на три группы.

[1] Compare – сравнивать.

В первую группу входят команды, которые обычно ставятся после команды сравнения. В их мнемокодах указывается тот результат сравнения, при котором надо делать переход.

Мнемокод	CMP op1, op2	Примечание
JE	op1 = op2	Для всех чисел
JNE	op1 ≠ op2	
JL/JNGE	op1 < op2	Для чисел со знаком
JLE/JNG	op1 ≤ op2	
JG/JNLE	op1 > op2	
JGE/JNL	op1 ≥ op2	
JB/JNAE	op1 < op2	Для чисел без знака
JBE/JNA	op1 ≤ op2	
JA/JNBE	op1 > op2	
JAE/JNB	op1 ≥ op2	

Рассмотрим пример: даны две переменные X и Y, в переменную Z нужно записать максимальное из чисел X и Y.

```
   MOV  EAX, X
   CMP  EAX, Y
   JGE/JAE L    ; JGE – для знаковых чисел и
                ; JAE – для беззнаковых
   MOV  EAX, Y
L: MOV  Z, EAX
```

Во вторую группу команд условного перехода входят те, которые обычно ставятся после команд, отличных от команды сравнения, и которые реагируют на значение определенного флага.

Мнемокод	Условие перехода	Мнемокод	Условие перехода
JZ	ZF = 1	JNZ	ZF = 0
JS	SF = 1	JNS	SF = 0
JC	CF = 1	JNC	CF = 0
JO	OF = 1	JNO	OF = 0
JP	PF = 1	JNP	PF = 0

Рассмотрим пример: пусть A, B и C – беззнаковые переменные размером 1 байт, требуется вычислить C = A * A + B, но если результат превосходит размер байта, передать управление на метку ERROR.

```
  MOV   AL, A
  MUL   AL
  JC    ERROR
  ADD   AL, B
  JC    ERROR
  MOV   C, AL
```

В третью группу входят две команды условного перехода, проверяющие не флаги, а значение регистра ECX или CX:

```
JCXZ <метка>    ; Переход, если CX = 0
JECXZ <метка>   ; Переход, если ECX = 0
```

Отметим общую особенность команд условного перехода: все они осуществляют только короткий переход, т.е. с их помощью можно передать управление не далее, чем на 128 байтов вперед или назад. Это примерно 30–40 команд (в среднем одна команда ПК занимает 3–4 байта).

Для реализации длинных условных переходов надо привлекать команду длинного безусловного перехода. Например, при «далекой» метке M оператор

```
if AX=BX then  goto M
```

следует реализовывать так:

```
if  AX<>BX  then  goto  L ; Короткий  переход
goto M                    ; Длинный переход
L:   …
     …
M:   …
```

На ЯА это записывается следующим образом:

```
  CMP AX, BX
  JNE L         ; Короткий переход
  JMP M         ; Длинный переход
L:   …
     …
M:   …
```

Отметим, что использовать в командах условного перехода оператор SHORT не надо, так как все эти переходы и так короткие.

Пример 5. Дана последовательность чисел. Признак завершения ввода — 0. Найти количество чисел, кратных 3.

```
A    DW ? ; Переменная для ввода числа
K    DW 0 ; Результат – количество чисел
TRI  DB 3 ; Делитель

L1:  ; Ввод числа a
  CMP A,  0     ; Проверка на окончание ввода
  JE EX
  MOV AX, A
  DIV TRI       ; Проверка свойства кратности 3
  CMP AH,0
  JNE L2
  INC K         ; Подсчет количества
L2:  JMP L1
EX:  ; Вывод числа k
```

Рассмотренный пример некорректно работает при вводе отрицательных чисел. Модифицируем пример так, чтобы отрицательное число перед проверкой на кратность изменяло знак.

Пример 6.

```
A    DW ? ; переменная для ввода числа
K    DW 0 ; результат – количество чисел
TRI  DB 3 ; делитель

L1:  ; Ввод числа A
  CMP A,  0
  JE EXIT
  JGE L3
  NEG A   ; Если A < 0, то меняем знак
L3:
  MOV AX, A
  DIV TRI
  CMP AH, 0
  JNE L2
  INC K
L2:
  JMP L1
EXIT: ; Вывод числа k
```

Пример 7. Найти сумму цифр заданного натурального числа.

```
.686
include /masm32/include/io.asm

.data
MsgInput db "Введите положительное число > ",0
MsgOutput db "Сумма цифр равна ",0
ten dw 10

.code
start:
  print "Введите положительное число > "
  inint EAX          ; Ввод числа

  MOV EBX, 0         ; EBX = 0
L:
  MOV EDX, 0         ; Делим EDX:EAX
  DIV ten            ; на 10
  ADD EBX, EDX       ; Прибавляем остаток (последнюю
цифру)
  CMP EAX, 0         ; Если число не равно нулю, то
  jne L              ; возврат к метке L

  print "Сумма цифр равна "
  outint EBX          ; вывод результата
  exit
end start
```

Пример 8. Дано натуральное число N. Дана последовательность, состоящая из N чисел. Найти количество чисел, кратных пяти.

```
.686
include /masm32/include/io.asm

.data
MsgInput db "Введите количество элементов
последовательности > ",0
MsgInput2 db "Введите элементы > ",0
MsgOutput db "Количество : ",0
five dw 5
a dd ?
N dd ?
count dd ?

.code
start:
  print "Количество элементов последовательности >"
  inint N      ; Ввод N
```

```
  print "Введите элементы > "
  MOV ECX, N    ; Число повторений цикла
  MOV count, 0 ; Обнуляем счетчик
L:
  inint a       ; Ввод элемента
  MOV EDX, 0
  MOV EAX, a    ; Деление элемента
  DIV five      ; на 5
  CMP EDX, 0    ; Если не делится нацело,
  JNE LL        ; то переход в конец цикла,
  INC count     ; иначе увеличиваем количество

  LL: LOOP L    ; следующая итерация цикла
                ; (возврат к метке L)

  print "Количество : "
  outint count ; вывод результата
  exit
end start
```

5.3. Вычисление логических выражений

При программировании на ЯА сложных булевских выражений можно обойтись без логических команд, достаточно лишь команд сравнения и условных переходов. Например, условный оператор

```
  if (AX>0) or (DX=1) then goto L
```

можно запрограммировать так:

```
  CMP AX, 0
  JG  L         ; AX>0 → L
  CMP DX, 1
  JE  L         ; DX=1 → L
```

Для оператора

```
  if (AX > 0) && (DX = 0) {
    goto L
  }
```

допустима такая последовательность команд:

```
  CMP AX, 0
  JLE M    ; AX<=0 → M
  CMP DX, 0
  JNE M    ; DX<>0 → M
  JMP L    ; Если ни один из переходов не состоялся
```

```
M: …
L: …
```

5.4. Моделирование циклов

В большинстве языков программирования существует один или несколько операторов цикла. С помощью команд перехода можно реализовать подобные конструкции. Например, следующие операторы языка C++ реализуются схемами (где `S`, `S1` и `S2` — операторы, а `X` — знаковая переменная).

`if (X>0)`
`{ S1 }`
`else { S2 }`
	`while (X>0)`
	`{ S }`
		`do`
		`{ S }`
		`while (X<=0)`

`  CMP X, 0`
`  JLE L2`
`  S1`
`  JMP Fin`
`L2: S2`
`Fin: …`
	`Beg:`
	`  CMP X, 0`
	`  JLEFin`
	`  S`
	`  JMP Beg`
	`Fin: …`
		`Beg:`
		`  S`
		`  CMP X, 0`
		`  JLE Beg`
		`  …`

С помощью команд перехода можно реализовать любые разветвления и циклы.

`; if (x > 0) S`
 	
`  CMP x, 0`
`  JLE L`
`  ...  ; S`
`L:`
	`; if (A > 0 || B > 0) S`
	
	`  CMP A, 0`
	`  JG  L1`
	`  CMP B, 0`
	`  JLE L2`
	`L1: ...  ; S`
	`L2:`

`; if (X = 0) S1 else S2`
 	
`  CMP X, 0`
`  JE  L1`
`  ...  ; S1`
`  JMP L2`
`L1:  ; S2`
`L2:`
	`; if (A > 0 && B > 0) S`
	
	`  CMP A, 0`
	`  JLE L`
	`  CMP B, 0`
	`  JLE L`
	`  ...  ; S`
	`L:`

```
; while (X > 0) do S

L1: CMP X, 0
    JLE L2
    ...              ; S
    JMP L1
L2:
```

```
; do S while (X > 0)

L:  ...          ; S
    CMP X, 0
    JG L
```

5.5. Команды управления циклом

5.5.1. Команда LOOP

Наиболее часто используемым является цикл с заранее известным числом повторений тела цикла.

В качестве счетчика цикла обязательно использовать регистр CX/ECX. Начальное значение для CX/ECX должно быть присвоено до цикла.

Описать работу этой команды можно так:

```
  CX = N
L:   [ … ] {тело цикла}
  CX = CX - 1
  if CX <> 0 then goto L
```

Тогда повторение N раз (N > 0) некоторой группы команд (тело цикла) можно реализовать так:

```
  MOV CX/ECX, N
L:   [ … ] {тело цикла}
  LOOP L
```

Команда LOOP требует, чтобы в качестве счётчика цикла использовался регистр CX/ECX. Собственно, команда LOOP вычитает единицу именно из этого регистра, сравнивает полученное значение с нулём и осуществляет переход на указанную метку, если значение в регистре ECX больше 0. Метка определяет смещение перехода, которое не может превышать 128 байт.

При использовании команды LOOP следует также учитывать, что с её помощью реализуется цикл с постусловием,

следовательно, тело цикла выполняется хотя бы один раз. Если до начала цикла записать в регистр CX/ECX значение ≤ 0, то при вычитании единицы, которое выполняется до сравнения с нулём, в регистре CX/ECX окажется ненулевое значение, и цикл будет выполняться 2^{32} раз.

Команда LOOP не относится к самым быстрым командам. В большинстве случаев её можно заменить последовательностью других команд.

Поскольку команда LOOP ставится в конце цикла, то тело цикла хотя бы раз обязательно выполнится. Поэтому для случая ECX = 0 наша схема цикла не подходит. Если возможен вариант, когда число повторений может быть и нулевым, то при ECX = 0 надо сделать обход цикла:

```
  MOV     ECX, N
  JECXZ   L1          ; ECX=0 -> L1;
L:    …   ; тело цикла
  LOOP    L
L1:  …
```

Для осуществления таких обходов в ПК была введена команда условного перехода JECXZ. В иных ситуациях она используется редко.

Рассмотрим пример использования команды LOOP.

Пример 9. Пусть X и Y – байтовые переменные со значением от 0 до 6 и надо в регистр AX записать степень Y числа X: AX = XY (отметим, что $6^6 = 46656 < 2^{16}$).

Для решения этой задачи надо вначале положить AX = 1, а затем Y раз выполнить умножение AX = AX*X. При этом следует учитывать, что при Y = 0 цикл не должен выполняться.

```
  MOV AX, 1     ; AX=1
  MOV CL, Y
  MOV CH, 0     ; CX=Y как слово (счетчик цикла)
  JCXZ L1       ; При N=0 обойти цикл
  MOV SI, X
L: MOV DX, 0
  MUL SI        ; (DX,AX)=AX*X (DX=0)
  LOOP L
```

```
L1:  …
```

Другой пример использования оператора цикла в программе — обработка последовательности чисел.

Пример 10. Дана последовательность из N чисел. Найти сумму положительных чисел последовательности.

```
A  DW ?
S  DW 0
N  DW ?
  MOV CX, N     ; Ввод количества чисел N
L1: ; Ввод числа a
    CMP A, 0          ;если <0, то на метку L1
    JL L2
    MOV AX, A
    ADD S, AX   ;вычисление суммы чисел
L2: LOOP L1
    ; Вывод числа S
```

5.5.2. Команды LOOPE/LOOPZ и LOOPNE/LOOPNZ

Эти команды похожи на команду LOOP, но позволяют также организовать и досрочный выход из цикла.

```
LOOPE <метка>          ; Команды являются синонимами
LOOPZ <метка>
```

Действие этой команды можно описать следующим образом:

```
ECX = ECX - 1; if (ECX != 0 && ZF == 1) goto <метка>;
```

До начала цикла в регистр ECX необходимо записать число повторений цикла. Команда LOOPE/LOOPZ, как и команда LOOP, ставится в конце цикла, а перед ней помещается команда, которая меняет флаг ZF (обычно это команда сравнения CMP). Команда LOOPE/LOOPZ заставляет цикл повторяться ECX раз, но только если предыдущая команда фиксирует равенство сравниваемых величин (вырабатывает нулевой результат, т.е. ZF = 1).

По какой именно причине произошёл выход из цикла, надо проверять после цикла. Причём надо проверять флаг ZF, а не

регистр ECX, так как условие ZF = 0 может появиться как раз на последнем шаге цикла, когда и регистр ECX стал нулевым.

Команда LOOPNE/LOOPNZ аналогична команде LOOPE/LOOPZ, но досрочный выход из цикла осуществляется, если ZF = 1.

Пример 11. Записать в регистр BL ноль, если число N является простым (N – байтовая переменная).

Для этого нужно последовательно делить N на числа 2, 3, ..., N–1 и сравнивать остатки от деления с 0 – до тех пор, пока не найдется нулевой остаток либо не будут исчерпаны все числа отрезка.

```
  MOV DL, N
  MOV DH, 0     ; DX = N как слово
  MOV CL, N
  MOV CH, 0
  SUB CX, 2     ; CX = N-2 (счетчик цикла)
  MOV BL, 1
DV:
  INC BL        ; Очередное число из [2,N-1]
  MOV AX, DX
  DIV BL        ; AH = N % BL
  CMP AH, 0     ; AH = 0?
  LOOPNE DV     ; Цикл выполняется CX раз и пока AH<>0
  JE DV1        ; Если AH = 0, то переход на DV1
  MOV BL, 0     ; нет делителей исходного числа
DV1:            ; непростое число – найден делитель
```

Рассмотрим другое решение определения простоты числа. Для поиска потенциального делителя используется регистр CX.

Пример 12. Пусть дано двухбайтовое беззнаковое число. Определить, является ли число простым.

```
A    DW   ?     ; Тестируемое число
F    DB   1     ; Флаг простоты: 1 – простое, 0 - нет

  MOV CX, A     ; CX - потенциальный делитель
  DEC CX        ; Исключаем делитель, равный CX
L1:
  MOV AX, A
  SUB DX, DX
  DIV CX
  CMP DX, 0     ; Если CX делитель, то
```

```
  JNE L2
  MOV F,  0     ; устанавливаем флаг в ноль
L2:
  CMP CX, 2     ; Исключаем делитель, равный 1
  JE L3
  LOOP L1
L3:  ; Анализируем флаг
```

5.5.3. Программирование вложенных циклов

Очевидно, что вложенные циклы типа `While` или `Do While` программируются на ассемблере без принципиальных сложностей в соответствии с рассмотренными стандартными схемами.

При программировании вложенных циклов типа `For` с использованием команды `LOOP` перед входом во вложенный цикл необходимо позаботиться о сохранении содержимого регистра `CX`, в котором хранится текущее значение параметра внешнего цикла, а после выхода из вложенного цикла требуется восстановить `CX`.

Пример 13. Рассмотрим двузначные числа от 10 до 99. Поместить в поле `P` сумму четных двузначных чисел.

Используя вложенные циклы, задачу можно решить с использованием следующего фрагмента программы.

```
BL=0;
For (DH=1; DH<=9; DH++)
  For (DL=0; DL<=9; DL++)
    If (DL mod 2 == 0)
      BL=BL+DH*10+DL;
```

Соответствующая программа на ассемблере будет такой.

```
Comment @ P – сумма искомых чисел
          DH – старшая цифра числа
          DL – младшая цифра числа
          AH – для вычисления суммы цифр
          BX – для сохранения содержимого CX
     @
  DVA DB  2
  TEN DB  10

  MOV P,  0
  MOV CX, 9
  MOV DH, 1
```

```
L1:
  MOV DL, 0
  MOV BX, CX    ; Сохранение текущего значения
  MOV CX, 10    ; счетчика внешнего цикла

L2:
  MOV AH, 0
  ADD AL, DL
  DIV DVA       ; Проверка младшей цифры на четность
  CMP AH, 0
  JNE Count
  MOV AL, DH
  MOV AH, 0
  MUL TEN       ; Получение AX = DH * 10
  ADD P,  AL
  ADD P,  DL

Count:
  INC DL
  LOOP L2

  MOV CX, BX    ; Восстановление текущего значения
  INC DH        ; счетчика внешнего цикла
  LOOP L1
```

Команда `LOOP L1` при переходе «вперед» на метку `L1` реализует короткие прыжки (от –128 до +127 байт). Поэтому часто при большом количестве операторов в теле цикла возможна ошибка `jmp destination too far`.

Эту проблему можно решить либо заменой организации цикла без команды `LOOP`, либо вставить «мостик»:

```
  MOV ECX, N    ; ECX - число повторений цикла
L01:
    <Операторы>
    ; "Мостик"
    JMP M00
L001:  JMP L01
M00:
    <Операторы>
  LOOP L001
```

6. ASM-ВСТАВКИ В ЯЗЫКАХ ВЫСОКОГО УРОВНЯ

Для создания эффективных программ как по размеру, так и по скорости используется ассемблер. При этом часто следует оптимизировать только некоторую часть программы или

реализовать «тонкие» операции. Поэтому ввод-вывод можно организовать на языке высокого уровня, а ассемблерные вставки использовать для организации эффективных вычислений.

Существуют следующие формы комбинирования программ на языках высокого уровня с ассемблером.

Это использование ассемблерных вставок (встроенный ассемблер, режим `inline`). Ассемблерные коды в виде команд ассемблера вставляются в текст программы на языке высокого уровня. Компилятор языка распознает их как команды ассемблера и без изменений включает в формируемый им объектный код. Эта форма удобна, если надо вставить небольшой фрагмент.

А также использование внешних процедур и функций. Это более универсальная форма комбинирования. У нее есть ряд преимуществ:

– написание и отладку программ можно производить независимо;

– написанные подпрограммы можно использовать в других проектах;

– облегчаются модификация и сопровождение подпрограмм.

6.1. ASM-вставки в Pascal

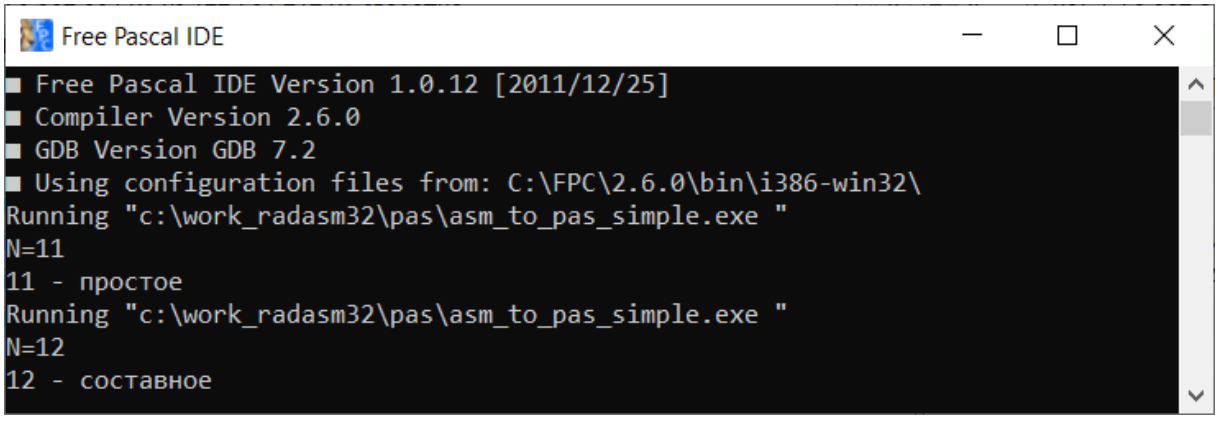
Формат вставки:

```
asm
     <Операторы ASM>
end;
```

Приведем рассмотренный ранее пример для определения «простоты» заданного `N`. Обращаем внимание на соответствие размеров переменных и регистров. В PascalABC.Net нет возможности делать ассемблерные вставки, поэтому приведем пример для Free Pascal.

```
Program Asm_to_Pas;
{$ASMMODE INTEL}
Var N: word;
    F: byte; // Флаг простоты: 1 – простое, 0 - нет
Label L1, L2, L3;
```

```
begin
  write('N='); readln(N);

  asm
    MOV F, 1          // Устанавливаем флаг в 1
    MOV CX, N         // CX - потенциальный делитель
    DEC CX            // Исключаем делитель, равный CX
L1: MOV AX, N
    SUB DX, DX
    DIV CX
    CMP DX, 0         // Если CX делитель, то
    JNE L2
    MOV F, 0          // устанавливаем флаг в 0
L2: CMP CX, 2         // Исключаем делитель, равный 1
    JE L3             // Если да, то выход
    LOOP L1
L3:                   // Выход
  end;

  writeln;
  if F=1 then writeln(N, ' - простое')
         else writeln(N, ' - составное');
  readln; // искусственная задержка
end.
```



6.2. ASM-вставки в C++

Формат вставки:

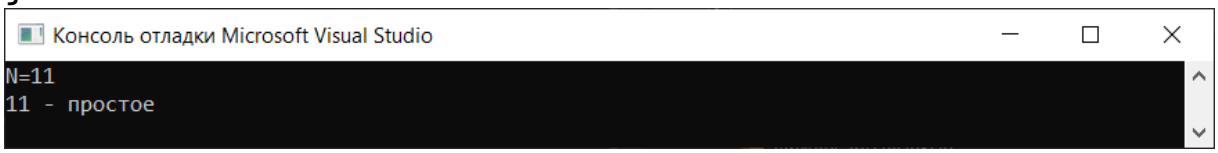
```
_asm {
     <Операторы ASM>
}
```

Далее приведен улучшенный вариант проверки числа на простоту. Он основан на том, что простое число не может быть четным. Поэтому, если число нечетное, то есть смысл проверять только нечетные делители в диапазоне от 3 до N/2.

Обращаем внимание на то, что в C++ тип int имеет размер 4 байта, поэтому используются 32-разрядные регистры EAX, EBX, ECX, EDX.

```
#include <iostream>
using namespace std;
int main()
{
  setlocale(LC_ALL, "rus");
  int N, N2;  // Тестируемое число
  int F;      // f- флаг: 1 – простое, 0 - нет
  const int DVA = 2;
  cout << "N=";      cin >> N;
  N2 = N;
  _asm {
    XOR EDX, EDX     ; Очищаем EDX
    MOV EAX, N       ; Готовимся к делению
    DIV DVA          ; Проверяем N на четность
    MOV N2, EAX      ; N2 = N / 2
    MOV F, EDX       ; F - остаток от деления  N / 2
    CMP EDX, 0
    JE L02           ; Если N четное = > N - составное

    MOV EBX, 3       ; Первый нечетный делитель
  L01:
    CMP EBX, N2      ; EBX > N / 2
    JG L03
    XOR EDX, EDX
    MOV EAX, N
    DIV EBX
    CMP EDX, 0       ; Если делитель = > N - составное
    JE L02
    ADD EBX, 2       ; Проверяем только нечетные
    JMP L01
```

```
  L02:
    MOV F, 0          ; Устанавливаем флаг в 0
    L03:
  }
  cout << N;
  if (F == 1) {
    cout << " - простое" << endl;
  }
  else {
    cout << " - составное" << endl;
  }
}
```

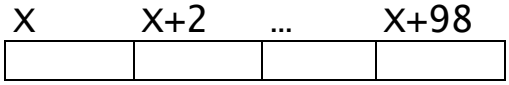


7. МАССИВЫ

7.1. Модификация адресов

Массивы в языке ассемблера описываются директивами определения данных с использованием конструкции повторения. Для того чтобы обратиться к элементу массива, необходимо так или иначе указать ***адрес*** начала массива и ***смещение*** элемента в массиве. Смещение первого элемента массива всегда равно 0. Смещения остальных элементов массива зависят от размера элементов.

Пусть 100 элементов одномерного массива размером `DW` располагаются последовательно, начиная с адреса `X`. Для доступа к значению элемента массива необходимо формировать его адрес как сумму некоторой постоянной части (адрес `X`) и переменной части (смещения относительно начала) – индекса. В данном случае смещение 2, так как размер элемента `DW`.



Конструкция

```
MOV AX, X[BX]
```

формирует исполнительный адрес
$A_{исп}$ = X + [BX]
и помещает значение элемента в регистр AX.

Тогда адрес элемента массива можно вычислить по следующей формуле:

```
адрес(X[i]) = X + (type X) * i,
```

где i – номер элемента массива, начинающийся с 0.

Имя переменной эквивалентно её адресу (для массива – адресу начала массива), а операция type определяет размер переменной (для массива определяется размер элемента массива в соответствии с использованной директивой).

7.2. Обработка одномерных массивов

Рассмотрим пример с программной переадресацией.

Пример 14. Найти сумму N элементов массива X, считая, что все промежуточные и окончательные результаты помещаются в слово.

```
; Сумма элементов массива размерности N
.686                ; Архитектура процессора i686
include /masm32/include/io.asm

.data               ; Сегмент данных
X   DW 100 DUP(?)   ; X[0..99]
N   DW ?            ; Количество элементов массива
SUM DW ?            ; Сумма
I   DB 0            ; Номер вводимого элемента X
.code               ; Сегмент кода
start:
  print "N="
  inint16 N
  MOV ECX, 0
  MOV CX,  N        ; Подготовка счетчика повторений

  MOV EBX, 0        ; Подготовка EBX
L0:
  print "X["
```

```
  INC I             ; Номер вводимого элемента X
  outint8 I         ; Выводим номер вводимого элемента
  print "]="
  inint16 X[EBX]    ; Ввод очередного элемента
  ADD EBX, 2        ; Следующий индекс
  LOOP L0

  MOV AX, 0         ; Подготовка для суммирования
  MOV CX, N         ; Подготовка счетчика повторений
  MOV EBX, 0        ; Подготовка EBX
L1: ADD AX, X[EBX] ; Суммирование элементов
  ADD EBX, 2        ; Следующий индекс
  LOOP  L1

  MOV SUM, AX
  print "Sum="
  outint16 SUM
  newline
  inkey "Press any key..."
  exit
end start
```

В приведенном примере N – количество элементов размером DW. Цикл LOOP использует регистр ECX, поэтому сначала необходимо очистить весь регистр ECX, а потом поместить в CX значение N.

Так как размер элементов массива – слово, поэтому шаг изменения адреса 2. Если суммировать элементы, стоящие на нечетных местах, то адрес должен изменяться на 4.

При работе с элементами массива – байтами, т.е. при

```
X  DB 100 DUP(?)
```

следовало бы записывать

```
  ADD EBX, 1    ; Переход к следующему элементу
```

или

```
  INC EBX
```

Пример 15. Дан одномерный массив размерности N. Найти сумму отрицательных элементов, расположенных на четных позициях.

Ввод N и элементов массива можно взять из предыдущего примера, с учетом того, что в этом примере размер элемента DD, т.е. в цикле ввода следует указать

```
  ADD EBX, 4    ; Следующий индекс
```

Далее приведен текст программы:

```
.data
…
two  DW 2
A    DD 100 dup (?)
N    DD ?
p    DD ?
count  DD ?
.code
start:
comment @
@
  MOV ECX, N
  MOV EDI, 0    ; Адрес первого элемента
  MOV p, 1      ; Позиция первого элемента
  MOV EBX, 0    ; Начальное значение суммы
L2:
  CMP A[EDI], 0; Проверка элемента на отрицательность
  JG LL2        ; Если > 0, то переход в конец цикла
  MOV EAX, p    ; Проверка позиции на четность
  MOV EDX, 0
  DIV two
  CMP EDX, 0    ; Если нечетная позиция, то
  JNE LL2       ; переход в конец цикла

  ADD EBX, A[EDI]    ; Если выполнены условия, то
                     ; Вычисляем сумму
LL2: ADD EDI, 4      ; Переход на следующий элемент
  INC p
  LOOP L2

  print "Sum="
  outint32 EBX
newline

inkey           ; Ожидание нажатия клавиши
exit
end start
```

Сделаем замечание. В данном случае можно было не вводить счетчик позиций элементов `p`, а за счет изменения смещения сразу перебирать только те элементы, которые стоят на четных позициях. То есть обеспечить изменение `EDI` начиная с 4, с шагом 8, так как элементы массива имеют размер `DD`.

Пример 16. Дан одномерный массив размерности N. Проверить, является ли массив упорядоченным по возрастанию.

```
…
.data
X DD 100 dup (?)
N DD ?
flag DD ?

.code
start:
  print "Введите размерность массива > "
  inint N
  print "Введите элементы массива > "

  MOV ECX, N
  MOV EDI, 0
L: inint X[EDI]      ;Ввод массива
  ADD EDI, 4
  LOOP L

  MOV ECX, N         ; Помещаем в ECX число N-1, чтобы
  DEC ECX            ; не выйти за границы массива
  MOV EDI, 0
  MOV flag,1         ; flag=1 (массив упорядочен)

L2: MOV EAX, X[EDI]  ; Сравниваем два соседних
  CMP EAX, X[EDI+4]  ; элемента
  JLE LL2            ; Если первый меньше или
                     ; равен, то переход в конец цикла
  MOV flag,0         ; иначе flag=0
                     ; (массив не упорядочен)
LL2: ADD EDI, 4
  LOOP L2

  Print "Результат: "
  outint flag
newline
inkey                ; Ожидание нажатия клавиши
exit
end start
```

7.3. Команда LEA

Команда LEA[1] осуществляет загрузку в регистр так называемого ***эффективного адреса***:

```
LEA <регистр>, <ячейка памяти>
```

Команда не меняет флаги. В простейшем случае с помощью команды LEA можно загрузить в регистр адрес переменной или начала массива:

```
x DD 100 dup (0)
LEA   EBX, x
```

Однако поскольку адрес может быть вычислен с использованием операций сложения и умножения, команда LEA имеет также ряд других применений.

7.4. Обработка двумерных массивов (матриц)

При обработке двумерных массивов в ЯА можно использовать модификацию адреса по двум регистрам. Причем один из них обязательно должен быть регистром EBX или EBP, а другой – регистром ESI или EDI (модифицировать по парам EBX и EBP или ESI и EDI нельзя). Возможный пример:

```
MOV AX, A[EBX][ESI]
```

В данном случае исполнительный адрес вычисляется по формуле

$$A_{исп} = A + [EBX] + [ESI]$$

Компиляторы большинства языков программирования размещают двумерные массивы по строкам.

Матрица $A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}$ в оперативной памяти размещается так:

[1] Load Effective Address – загрузка эффективного адреса.

a_{11}	a_{12}	a_{13}	a_{21}	a_{22}	a_{23}

Однако способ размещения элементов двумерного массива (по строкам или по столбцам) и соответствующая их обработка определяются программистом.

Пусть матрица произвольной размерности N×M расположена в ОП по строкам.

$$A = \begin{bmatrix} a_{11} & \dots & a_{1M} \\ a_{21} & \dots & a_{2M} \\ \dots & & \dots \\ a_{N1} & \dots & a_{NM} \end{bmatrix}$$

1	2		M	M+1	M+2		2M		K		(N-1)M	(N-1)M+1		(N-1)M+M
a_{11}	a_{12}	…	a_{1M}	a_{21}	a_{22}	…	a_{2M}	…	a_{ij}	…	a_{N1}	a_{N2}	…	a_{NM}

Тогда порядковый номер K элемента a_{ij} можно определить так:

$$K = (i - 1) \times M + j. \qquad (1)$$

Обратную задачу (по номеру K определить индексы элемента (i, j) рекомендуем выполнить самостоятельно.

Пример 17. Найти сумму элементов матрицы A размерности 5×7, расположенной в ОП по строкам. Каждый элемент размером байт.

```
A DB 5 DUP(7 DUP(?))
SUM DW ?
  MOV AX, 0          ; Для суммирования
  MOV ECX, 35        ; Подготовка счетчика
  MOV EBX, 0         ; Подготовка BX для модификации

CKL: ADD AL, A[EBX]  ; Суммирование элементов массива
  INC EBX            ; Следующий индекс
  LOOP CKL
  MOV SUM, AX
```

Пример 18. Найти сумму диагональных элементов той же матрицы A.

```
A DB 5 DUP(5 DUP(?))
SUM DW ?
  MOV EAX, 0          ; Для суммирования
  MOV ECX, 5          ; Подготовка счетчика
  MOV EBX, 0          ; Подготовка BX для модификации

L1: ADD AL, A[EBX]    ; Суммирование элементов массива
  ADD EBX, 6          ; Следующий индекс
  LOOP L1
  MOV SUM, AX
```

Легко заметить, что решение этих задач практически не отличается от решения задачи по обработке одномерного массива, т.е. переадресация с постоянным шагом ведется с использованием одного регистра-модификатора.

Пример 19. Приведем вариант кода программы ввода и вывода элементов матрицы, который будем использовать в дальнейшем.

```
.686
include /masm32/include/io.asm

.data
A DD 10 dup (10 dup (?))
type_A DD type A ; Размер элемента матрицы
Step DD ?        ; Длина (в байтах) строки матрицы:
                 ; Step = type_A * M
N DD ?           ; Число строк
M DD ?           ; Число столбцов

.code
start:
; Ввод размерности матрицы A размерности N * M
  print "N="
  inint32 N
  print "M="
  inint32 M

; Длина (в байтах) строки матрицы:
; Size_str = type_A * M для перехода по строкам
  MOV EAX, type_A
  MUL M
```

```
  MOV Size_str,EAX  ; Size_str = type_A * M
  Outstr offset msg_Arr

; Ввод элементов матрицы
; ECX - число повторений внешнего цикла
  MOV ECX, N

; Адрес первого элемента очередной строки
  MOV EBX, 0

L01:
  ; Сохраняем ECX в EDX,
  ; т.к. он изменится во внутреннем цикле.
  ; Можно сохранить ECX в стеке: PUSH ECX
  MOV EDX, ECX

  ; ECX - число повторений внутреннего цикла
  MOV ECX, M

  ; Адрес первого элемента в очередной строке
  MOV EDI, 0
  L02:
    inint32 A[EBX][EDI]
    ; Переход на элемент в очередной строке
    ADD EDI, type_A
    LOOP L02

  MOV ECX,EDX  ; Восстанавливаем ECX из EDX
               ; Или извлечь ECX из стека: POP ECX
  ADD EBX, Step; переход на следующую строку
  LOOP L01

; Вывод матрицы
  ; ECX - число повторений внешнего цикла
  MOV ECX, N
  ; Адрес первого элемента очередной строки
  MOV EBX, 0

L11:
  MOV EDX, ECX ; сохраняем ECX в EDX
  ; ECX - число повторений внутреннего цикла
  MOV ECX, M
  ; Адрес первого элемента в очередной строке
  MOV EDI, 0

  L12:
    ; Вывод элемента в 6 позициях
    outint32 A[EBX][EDI], 6
    ; Переход на элемент в очередной строке
    ADD EDI, type_A
```

```
    LOOP L12
  MOV ECX,EDX  ; Восстанавливаем ECX из EDX
  ADD EBX, Step; Переход на следующую строку
  newline
  LOOP L11
  newline
  inkey        ; Ожидание нажатия клавиши
  exit
end start
```

Пример 20. Найти количество строк той же матрицы A, в которых первый элемент строки встречается в ней еще один раз.

При расположении элементов матрицы в памяти по строкам (первые 5 байтов – начальная строка матрицы, следующие 5 байтов – вторая строка и т. д.) адрес элемента A[i, j] равен

```
A + 5*(i – 1) + j.
```

Для хранения величины 5*i отведем регистр BX, а для хранения j – регистр SI. Тогда A[BX] – это начальный адрес i-й строки матрицы, а A[BX][SI] – адрес j-го элемента этой строки.

```
.686
include /masm32/include/io.asm

.data
A  DB 10 dup (10 dup (?))
type_A DD type A    ; Размер элемента матрицы
Step DD ?           ; Длина (в байтах) строки матрицы:
                    ; Step = type_A * M

R  DD ?   ; Для сохранения ECX
msg_KDB "K=", 0
K  DB ?

.code
start:
; Ввод размерности матрицы N, M
…

; Длина (в байтах) строки матрицы: Step = type_A * M.
  MOV EAX, type_A
  MUL M
  MOV Step, EAX  ; Step = type_A * M

; Ввод элементов матрицы
…
```

```
; Вывод матрицы
…
Comment @  AL - счетчик количества строк
           AH - 1-й элемент очередной строки
           EBX - индекс строки i
           EDI - индекс столбца j
           ECX - счетчик цикла
           R - для сохранения содержимого CX
@

  MOVEAX, 0
  MOV EBX, 0 ; Адрес первого элемента строки
  MOV ECX, N ; ECX - число повторений внешнего цикла

  ; Внешний цикл (по строкам i)
L0:
  MOV AH, A[EBX]     ; 1-й элемент очередной строки
  MOV R, ECX         ; Сохранение CX внешнего цикла

  ; Внутренний цикл (по столбцам j)
  MOV ECX, M    ; Счетчик внутреннего цикла
  MOV EDI, 0    ; Индекс элемента внутри строки j
L1: ADD EDI, type_A       ; j = j + type_A
    CMP A[EBX][EDI], AH   ; A[i,j]=AH?
    LOOPNEL1              ; Цикл, пока A[i,j]<>AH,
                          ; но не более M раз
  JNE L2                  ; AH не повторился -> L2
  INC AL                  ; Учет строки
  ; Конец внутреннего цикла

L2:
  ; Восстанавливаем CX для внешнего цикла
  MOV ECX, R
  ADD EBX, Step      ; На начало следующей строки
  LOOP L0            ; Цикл N раз
  MOV K, AL

  outstr offset msg_K
  outint8 K
  newline

  inkey         ; Ожидание нажатия клавиши
  exit
end start
```

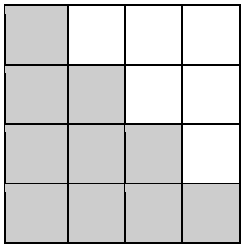
В различных задачах для номеров строк и столбцов иногда удобно вводить переменные, явно отвечающие за эти значения. Рассмотрим следующий пример.

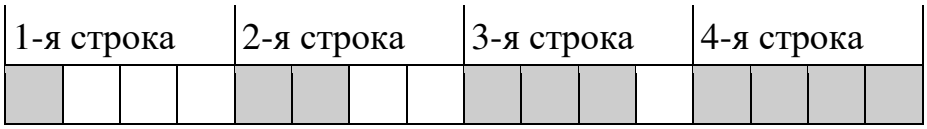
Пример 21. Дана квадратная матрица. Требуется обнулить элементы, лежащие ниже и на главной диагонали.

При вычислении порядкового номера K элемента a_{ij} по его индексам (1) есть возможность другой организации доступа к элементам матрицы, схему которой можно описать так:

```
L = 1; { Размер элемента }
For (i = 1; i <= N; i++)
  For (j = 1; j <= i; j++) {
    K = (i - 1) * N + j - 1;
    MOV A[(K - 1) * L], 0;
  }
```

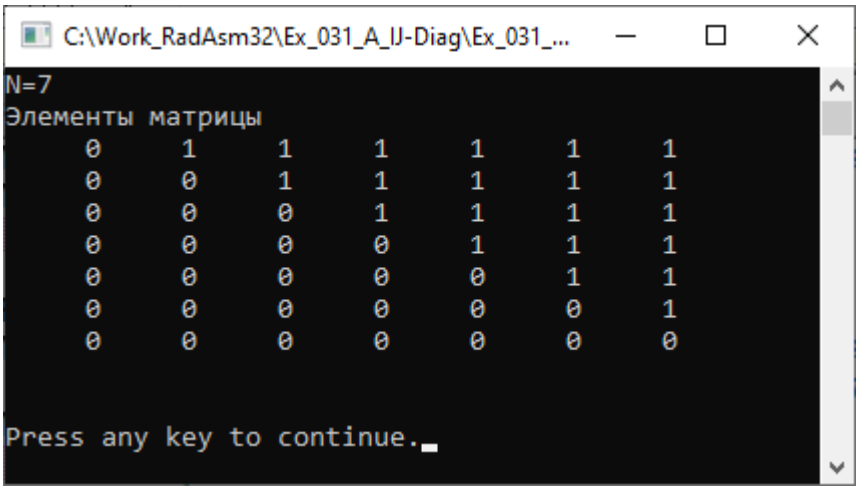
Рассмотрим элементы, которые необходимо обработать на примере матрицы 4×4, т.е. в первой строке нужно получить доступ к одному элементу, во второй — к двум и т.д.






```
.686
include /masm32/include/io.asm
.data
; Для простоты все элементы матрицы заполним 1
A DW 100 DUP(1)

type_A DD type A ; Размер элемента в байтах (=4)
Step DD ?        ; Длина (в байтах) строки матрицы:
                 ; Step = type_A * N
N DD ?     ; Размерность матрицы
I DW ?     ; Номер строки текущего элемента
J DW ?     ; Номер столбца текущего элемента

.code
start:
  print "N="
  inint32 N
```

```
  ; Длина (в байтах) строки матрицы: Step=type_A*N.
  ; Для перехода по строкам
  MOV EAX, type_A
  MUL N
  MOV Step, EAX  ; Step=type_A*N
  MOV ECX, N   ; Число итераций внешнего цикла
  MOV I, 1

L01:
  ; Сохранение CX внешнего цикла в стеке
  PUSH ECX
  MOV ECX, 0   ; Число итераций внутреннего цикла
  ; Число итераций внутреннего цикла совпадает
  ; с номером строки
  MOV CX, I

  MOV J, 1
L02:
    ; Вычисляем смещение A[I][J] по индексам элемента
    ; по формуле (1)
    ; K = (I - 1) * N + J
    ; Порядковый № элемента A[I][J], EBX = type_A * K
    MOV EAX, 0
    MOV AX, I
    DEC AX
    MUL N
    ADD AX, J        ; AX = K
    DEC AX
    MUL type_A       ; EAX = type_A * K
    MOV EBX, EAX     ; EBX = EAX

    MOV A[EBX], 0    ; Заполняем 0

    INC J
    LOOP L02

  INC I
  POP ECX; Восстанавливаем CX из стека

  LOOP L01

  ; Вывод матрицы
  …

inkey           ; ожидание нажатия клавиши
exit
end start
```



Пример 22. Транспонировать заданную квадратную матрицу размерности N × N.

Воспользуемся рассуждениями из предыдущего примера. Нам нужно получать доступ к элементам a_{ij} ниже главной диагонали (не включая саму диагональ) и менять местами с элементом a_{ji}.

```
L = 1; { Размер элемента }
For (i = 2; i <= N; i++)
  For (j = 1; j <= I - 1; j++) {
    K1 = ((i - 1) * N + j - 1) * L;
    K2 = ((j - 1) * N + i - 1) * L;
    // Меняем местами элементы A[K1] и A[K2]
  }
```

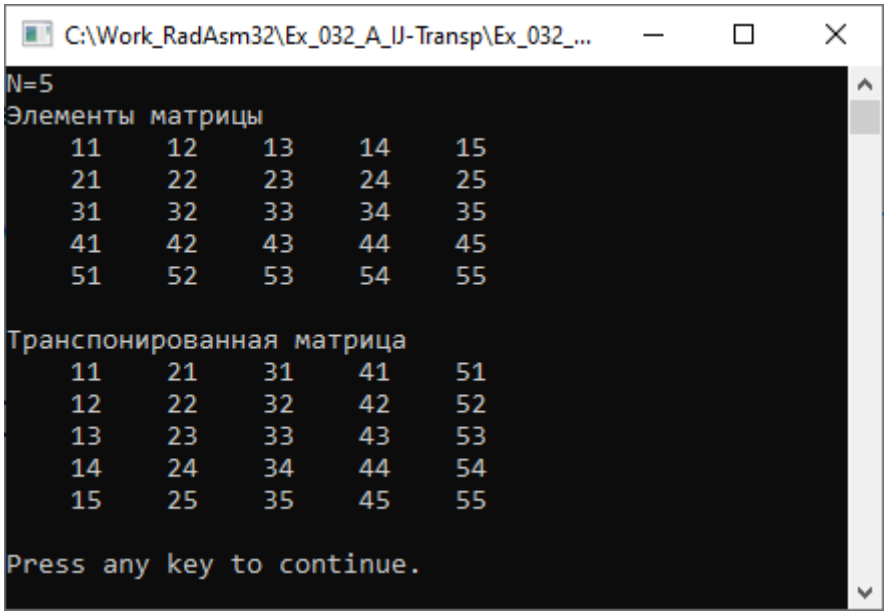
Далее приведен код программы.

```
…
.data
A DW 100 DUP(1)

type_A DD type A ; Размер элемента в байтах (=4)
Step DD ?        ; Длина (в байтах) строки матрицы:
                 ; Step = type_A * N
N DD ?           ; Размерность матрицы
I DW ?           ; Номер строки текущего элемента
J DW ?           ; Номер столбца текущего элемента
TEN DW 10

.code
start:
  print "N="
  inint16 N
```

```
; Размер (в байтах) строки матрицы
  MOV EAX, type_A
  MUL N
  MOV Step, EAX  ; Step = type_A * N

  MOV ECX, 0
  MOVCX,  N    ; Число итераций внешнего цикла
  MOVI,   1

L01:
  PUSH ECX
  MOV ECX, 0   ; Число итераций внутреннего цикла
  ; Число итераций внутреннего цикла совпадает
  ; с номером строки
  MOV CX, N
  MOV J, 1
L02:
    ; Вычисляем смещение по формуле (1)
    MOV EAX, 0
    MOV AX, I
    DEC AX
    MUL N
    ADD AX, J        ; AX = K
    DEC AX
    MUL type_A       ; EAX = type_A * K
    MOV EBX, EAX     ; EBX = EAX
    ; Заполним элементы матрицы значениями I * 10 + j
    MOV AX, I
    MUL TEN
    ADD AX, J
    MOV A[EBX], AX

    INC J
    LOOP L02
    INC I
    POP ECX
    ; LOOP L01
    ; Т.к. нужен "далекий" переход,
    ; вынуждены вместо LOOP использовать конструкцию
    DEC ECX
    CMP CX, 0
    JLE L03
    JMP L01
L03:

; Вывод матрицы
…
; Транспонирование
  MOV CX, N    ; Число итераций внешнего цикла
  DEC ECX
```

```
  MOV I, 2
L21:
  PUSH ECX
  MOV ECX, 0   ; Число итераций внутреннего цикла
  ; Номер итерации внутреннего цикла совпадает
  ; с номером строки
  MOV CX, I
  DEC CX

  MOV J, 1
L22:
    ; K1 = (I - 1) * N + J -1 ; № элемента A[I][J]
    ; EBX = type_A * K1
    MOV EAX, 0
    MOV AX, I
    DEC AX
    MUL N
    ADD AX, J
    DEC AX
    MUL type_A       ; EAX = type_A * K1
    MOV EBX, EAX     ; EBX = EAX

    ; K2 = (J - 1) * N + I -1 ; № элемента A[J][I]
    ; EDX = type_A * K2
    MOV EAX, 0
    MOV AX, J
    DEC AX
    MUL N
    ADD AX, I
    DEC AX
    MUL type_A       ; EAX = type_A * K2
    MOV EDX, EAX     ; EDX = EAX

    ; Меняем местами  A[K1] и A[K2]
    PUSH A[EBX]
    PUSH A[EDX]
    POP A[EBX]
    POP A[EDX]
    INC J
    LOOP L22

  INC I
  POP ECX
  ; Вынуждены вместо LOOP использовать такую
  ; конструкцию, т.к. нужен "далекий" переход
  DEC ECX
  CMP CX, 0
  JLE L23
  JMP L21
L23:
```

```
; Вывод матрицы
     print "Транспонированная матрица \n"
…
inkey
exit
end start
```



Пример 23. Дана неквадратная матрица размерности `N × M`. Требуется найти номера строки и столбца, на пересечении которых находится наибольший элемент матрицы.

Введем две переменные `I` и `J`, хранящие координаты текущего элемента матрицы. При поиске наибольшего элемента будет обновляться не только его значение, но и в переменных `IMAX` и `JMAX` будут сохраняться координаты этого элемента.

```
A DW 100 DUP(?)
type_A DD type A     ; Размер элемента в байтах (=4)
Step DD ?            ; Размер строки матрицы: Step=type_A*M
N DD ?               ; Количество строк матрицы
M DD ?               ; Количество столбцов матрицы
I DW ?               ; Номер строки текущего элемента
J DW ?               ; Номер столбца текущего элемента
IMAX DW ?            ; Номер строки с наибольшим элементом
JMAX DW ?            ; Номер столбца с наибольшим элементом
MAX  DW ?            ; Значение наибольшего элемента
```

```
.code
start:
; Ввод размерности матрицы A размерности N*M
…
; Ввод матрицы
…
  MOV DX, A[0] ; Начальное значение максимума
  MOV MAX, DX

  MOV IMAX, 1  ; Начальный № строки максимума
  MOV JMAX, 1  ; Начальный № столбца максимума
  MOV ECX, N   ; Число итераций внешнего цикла
  MOV I, 1

L4: PUSH ECX
  MOV ECX, M   ; Число итераций внутреннего цикла
  MOV J, 1
L3:
; = = = = = = = =
; Вычисляем смещение A[I][J] по индексам элемента (1)
  ; K = (I - 1)*N + J – № элемента A[I][J]
  ; EBX = type_A * K – EBX=смещение A[I][J]
  MOV EAX, 0
  MOV AX, I
  DEC AX
  MUL N
  ADD AX, J          ; AX = K
  MUL type_A   ; EAX = type_A * K
  MOV EBX, EAX ; EBX = EAX
; = = = = = = = =
  MOV DX, MAX
; Сравниваем очередной элемент с текущим MAX
  CMP A[EBX], DX
  JLE L5       ; Если <=, то переход к следующему
  MOV DX, A[EBX][EDI]     ; Обновление значения
                          ; максимального элемента
  MOV MAX, DX
  MOV DX, I
  MOV IMAX, DX       ; Сохранение номера строки
  MOV DX, J
  MOV JMAX, DX       ; Сохранение номера столбца

L5: INC J
  LOOP L3
  INC I
  POP ECX
  LOOP L4

  ; Вывод значений IMAX и JMAX
  print "I max="
```

```
  outint16 IMAX
  newline
  print "J max="
  outint16 JMAX
  inkey   ; ожидание нажатия клавиши
  exit
end start
```

Пример 24. В одномерном массиве X[N] каждый отрицательный элемент заменить его квадратом. Формирование элементов и вывод результата реализуем в C++, а обработку элементов массива сделаем на ассемблере. Обращаем внимание на то, что тип int в C++ имеет размер в 4 байта.

Элементы массива заполним случайными числами из диапазона [A, B].

```
#include <iostream>
#include <ctime>
using namespace std;
int main()
{
  setlocale(LC_ALL, "rus");
  int i, N;        // Количество элементов массива
  const int A = -20, B = 30; // Диапазон случайных чисел
  int X[50];       // Массив, каждый элемент размера DD
  const int type_X = 4; // Размер элемента (DD=4)
  cout << "N=";    cin >> N;

  srand(time(0)); // Инициализация датчика случайных чисел
  for (i = 0; i < N; i++) {
    X[i] = rand() % (B - A + 1) + A;
    // Заполнение элементов массива случайными числами
    // из диапазона [A, B]
}

  cout << "= = = = = = = = = = =" << endl;
  for (i = 0; i < N; i++) // Вывод элементов массива
    cout << X[i] << "\t";
  cout << endl;

  _asm {
    MOV ECX, N;
```

```
    LEA EBX, X
    L01:
      MOV EAX, [EBX]    // Очередной элемент массива
      CMP EAX, 0
      JGE L02           // Если >=0, переход к следующему
        MUL EAX         // AX=AX*AX
        MOV [EBX], EAX
      L02:
      ADD EBX, type_X
    LOOP L01
}

  cout << "= = = = = = = = = = =" << endl;
  for (i = 0; i < N; i++) // Вывод элементов массива
    cout << X[i] << "\t";
  cout << endl;
  system("pause");
  return 0;
}
```

8. БИТОВЫЕ ОПЕРАЦИИ

8.1. Логические команды

Логические команды выполняют булевские операции – отрицание, конъюнкцию, дизъюнкцию и сложение по модулю 2.

Они реализуют поразрядные операции, т.е. каждый бит результата зависит от соответствующих битов операндов.

Во всех этих командах бит 1 трактуется как «истина», а бит 0 — как «ложь».

Эти команды меняют все флаги условий, но обычно используется только флаг нуля `ZF` (1, если результат = `0` и `0`, если в результате есть хотя бы одна `1`). Другие флаги, предназначенные для работы с числами и в логических операциях, мало информативны.

Первый может быть регистром или ячейкой памяти, а второй – регистром, ячейкой памяти или непосредственным операндом. Операнды должны иметь одинаковый размер. Результат помещается на место первого операнда. Операции меняют флаги `CF, OF, PF, SF` и `ZF`.

Конъюнкция (логическое умножение)

```
AND op1, op2 ; op1=op1 and op2
```

Как обычно могут использоваться все сочетания операндов, допускаемые форматами машинных команд.

Команду AND часто используют для выделения некоторых битов байта или слова памяти.

Пусть, например:

```
MOV AL, 01101101b
AND AL, 00001111b ; [AL]=00001101b
```

Дизъюнкция (логическое сложение)

```
OR op1, op2 ; op1=op1 or op2
```

Исключающее ИЛИ (eXclusive OR)

```
XOR op1, op2 ; op1=op1 xor op2
```

Очевидно, что

```
XOR EAX, EAX
```

обнуляет содержимое EAX. По сравнению с другими приемами, например,

```
MOV EAX, 0      или   SUB EAX, EAX
```

обнуление выполняется быстрее.

Операцию XOR можно также использовать для обмена значений двух переменных.

```
XOR EAX, EBX ; EAX = EAX XOR EBX
XOR EBX, EAX
; Теперь EBX содержит исходное значение EAX
XOR EAX, EBX
; Теперь EAX содержит исходное значение EBX
```

Отрицание NOT

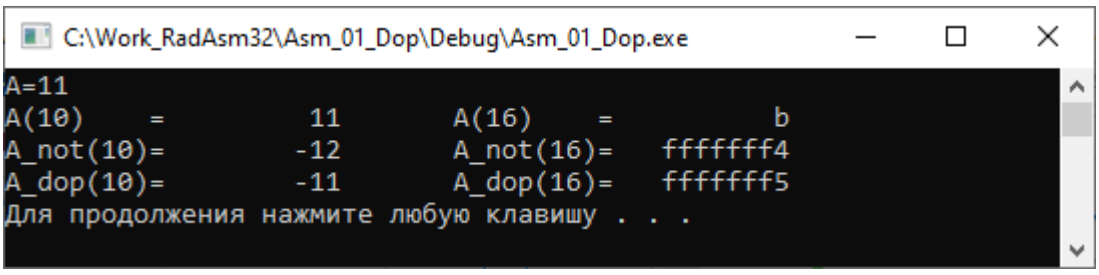
```
NOT op
```

Меняет значение каждого бита операнда на противоположное.

Пример 25. Получить дополнительный код заданного числа. (см. 2.5).

Воспользуемся ассемблерной вставкой в C++, т.к. в C++ есть возможность вывода в 16-ричном виде.

```
#include <iostream>
using namespace std;
int main()
{
  setlocale(LC_ALL, "rus");
  int A, A_not, A_dop; // Размер 4 байта
  cout << "A=";  cin >> A;

  _asm {
    MOV EAX, A
    NOT EAX
    MOV A_not, EAX
    ADD EAX, 1 // Формирует дополнительный код
    MOV A_dop, EAX
  }

  printf("A(10)    =%11d  A(16)    =%11x \n", A, A);
  printf("A_not(10)=%11d  A_not(16)=%11x \n",
          A_not, A_not);
  printf("A_dop(10)=%11d  A_dop(16)=%11x \n",
          A_dop, A_dop);
}
```



Проверка

`TEST op1, op2`

Выполняется так же, как и команда `AND`, но при этом результат никуда не записывается. Команда `TEST` формирует значение флага нуля `ZF`. Он равен 1, если в результате получился нулевой ответ, и равен 0, если есть хотя бы одна двоичная 1.

Пример 26. Осуществить переход на метку `L`, если третий бит содержимого регистра `AX` равен 1.

```
TEST AX, 00000000 00000100b
JNZ L
```

8.2. Команды сдвига

Операции сдвига вправо и сдвига влево сдвигают биты в переменной на заданное количество позиций. Каждая команда сдвига имеет две разновидности:

```
<МКО> op1, op2
<МКО> op1, CL
```

где МКО – мнемонический код операции;

op1 – регистр или поле памяти;

op2 – количество позиций сдвига.

Первый операнд op1 должен быть регистром или ячейкой памяти. Именно в нём осуществляется сдвиг. Второй операнд определяет количество позиций для сдвига, которое задаётся непосредственным операндом или хранится в регистре CL (и только CL).

Команды сдвига меняют флаги CF, OF, PF, SF и ZF.

Существует несколько разновидностей сдвигов, которые отличаются тем, как заполняются «освобождающиеся» биты.

8.2.1. Логические сдвиги

В этих командах в сдвиге участвуют все биты первого операнда. При этом бит, «уходящий» за пределы ячейки, заносится в флаг CF, а с другого конца в операнд добавляется 0.

Логический сдвиг влево

```
SHL op, <количество>    ; Shift Left
```

Логический сдвиг вправо

```
SHR op, <количество>    ; Shift Right
```

Условно действия этих команд можно изобразить так:

Примеры:

```
MOV BH, 01000111b
SHL BH, 1          ; CF=0, BH=10001110b
MOV AH, 01000111b
SHR AH, 1          ; CF=1, AH=00100011b
MOV DH, 00111000b
MOV CL, 3
SHL DH, CL         ; CF=1, DH=11000000b
```

8.2.2. Арифметические сдвиги

Арифметический сдвиг влево эквивалентен логическому сдвигу влево (это одна и та же команда) – «освобождающие» биты заполняются нулями. При *арифметическом сдвиге вправо* «освобождающиеся» биты заполняются знаковым битом. Последний ушедший бит сохраняется во флаге CF.

Арифметический сдвиг влево

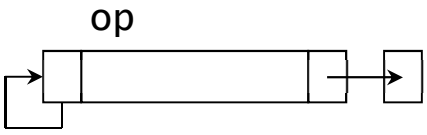
```
SAL op, <количество>    ; Shift Arithmetic Left
```

Арифметический сдвиг вправо

```
SAR op, <количество>    ; Shift Arithmetic Right
```

Как и в команде логического сдвига вправо, здесь также сдвигаются вправо все биты первого операнда, причем «уходящий» бит заносится в флаг CF, однако затем знаковый (самый левый) бит операнда восстанавливает свое исходное значение.

Условно действие этой команды можно изобразить так:



Примеры:

```
MOV AL, 10001110b
SAR AL, 1          ; AL=11000111b, CF=0
MOV AH, 00001110b
SAR AH, 1          ; AH=00000111b, CF=0
```

8.2.3. Циклические сдвиги

Особенность циклических сдвигов заключается в том, что «уходящий» бит не теряется, а возвращается в операнд, но с другой стороны.

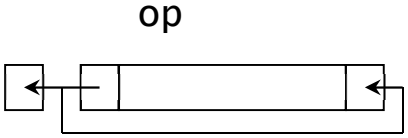
Циклический сдвиг влево

```
ROL op, <количество>    ; Rotate Left
```

Циклический сдвиг вправо

```
ROR op, <количество>    ; Rotate Right
```

В команде ROL все биты сдвигаются влево, причем самый левый бит возвращается в операнд с правого конца и одновременно заносится в флаг CF.



Команды циклического сдвига обычно используются для перестановки частей содержимого ячейки или регистра. Например, поменять местами правую и левую половины регистра AL можно циклическим сдвигом этого байта на 4 разряда влево (или вправо):

```
MOV AL, 8Eh   ; AL=10001110b
MOV CL, 4
ROL AL, CL    ; AL=11101000b = 0E8h
```

8.2.4. Расширенные сдвиги

Расширенные сдвиги немного отличаются от остальных сдвигов. В расширенных сдвигах участвуют два регистра или ячейка памяти и регистр, которые как бы объединяются в единое целое и «освобождающиеся» биты одного операнда заполняются битами из другого операнда.

Расширенный сдвиг влево

```
SHLD op1, op2, <количество>
```

Расширенный сдвиг вправо

```
SHRD op1, op2, <количество>
```

Команда SHLD сдвигает влево биты op1 на указанное количество позиций. Младшие («освободившиеся») биты op1 заполняются старшими битами op2. Сам op2 не меняется.

Команда SHRD сдвигает вправо биты op1 на указанное количество позиций. Старшие («освободившиеся») биты op1 заполняются младшими битами op2. Сам op2 не меняется.

Количество, как и в других операциях сдвига, задаётся непосредственным операндом или хранится в регистре CL. Но используются только последние 5 бит операнда, определяющего *количество*, т.е. максимальное количество позиций сдвига равно 32.

Команды расширенного сдвига обычно используют для создания упакованных данных.

8.3. Умножение и деление с помощью поразрядных операций

Для любой системы счисления сдвиг числа влево или вправо соответствует умножению или делению на основание системы счисления в некоторой степени. Двоичная система счисления, используемая в компьютере, не является исключением. Причём команды сдвига работают на порядок быстрее обычных операций умножения и деления.

8.4. Быстрое умножение и деление на степени 2

Команды логического сдвига часто используются для реализации быстрого умножения и деления целых чисел на 2^k.

Сдвиг десятичного числа 123 на 2 цифры влево – это приписывание двух нулей справа, т.е. умножение на 10^2: 12300. Аналогично сдвиг двоичного числа на k разрядов влево – это приписывание справа k двоичных нулей, т.е. умножение на 2^k. Например, при сдвиге числа 5 на 3 разряда влево получаем:

```
9 = 1001b          1001000b = 72 = 9*23
```

Пусть K (байт) – некоторое поле, содержащее константу. Тогда последовательность команд

```
MOV CL, K          ; Умножаем содержимое
SHL AX, CL         ; AX на 2^k
```

умножает содержимое AX на 2^k.

Заметим, что умножение с помощью сдвига выполняется значительно быстрее и корректно как для положительных, так и для отрицательных целых чисел при условии, что результат помещается в отведенное для него место.

Деление на 2^k с помощью SHR корректно выполняется только для положительных целых чисел. Для работы с отрицательными величинами следует использовать команды арифметического сдвига.

8.4.1. Умножение

Для умножения используется сдвиг влево. Несмотря на наличие двух команда, по сути сдвиг влево один. Он используется для умножения как знаковых, так и беззнаковых чисел. Однако результат будет правильным только в том случае, если он умещается в регистр или ячейку памяти.

```
MOV AX, 250     ; AX = 00fah = 250
SAL AX, 4       ; Умножение на 2^4 = 16, AX=0fa0h=4000

MOV AX, 1       ; AX = 1
SAL AX, 10      ; Умножение на 2^10, AX = 0400h = 1024

MOV AX, -48     ; AX = ffd0h = -48 (в доп. коде)
SAL AX, 2       ; AX = ff40h = -192 (в доп. коде)

MOV AX, 26812   ; AX = 68bch = 26812
SAL AX, 1       ; AX = d178h = -11912
                ; Знаковое >0 перешло в отрицательное

MOV AX, 32943   ; AX = 80afh = 32943
SAL AX, 2       ; AX = 02bch = 700
; Большое беззнаковое число стало гораздо меньше
```

Сочетая сдвиги со сложением и вычитанием, можно выполнить умножение на любое положительное число. Для умножения на отрицательное число следует добавить команду NEG.

```
MOV EBX, X
MOV EAX, EBX
SAL EAX, 2
ADD EAX, EBX        ; EAX = X * 5

MOV EBX, X
MOV EAX, EBX
SAL EAX, 3
SUB EAX, EBX        ; EAX = X * 7

MOV EBX, X
MOV EAX, EBX
SAL EAX, 2
ADD EAX, EBX
SAL EAX, 1          ; EAX = X * 10
```

Такой набор операций выполняется в 1,5–2 раза быстрее, чем обычное умножение. Но если оба сомножителя заранее неизвестны, то лучше использовать умножение.

8.4.2. Деление

Для деления используется сдвиг вправо. При делении нет проблем с переполнением, но для знаковых и беззнаковых чисел надо использовать разные механизмы.

Для деления беззнаковых чисел следует использовать логический сдвиг вправо.

```
MOV AX, 43013       ; AX = a805h = 43013
SHR AX, 1           ; AX = 5402h = 21506
```

Со знаковыми числами дело обстоит несколько сложнее. В принципе, для деления знаковых чисел следует использовать арифметический сдвиг вправо. Однако для отрицательных чисел получается не совсем корректный результат: `1 / 2 = 0`, `3 / 2 = 1`, но `-1 / 2 = -1`, `-3 / 2 = -2`, т.е. результат отличается от правильного на единицу. Для того чтобы получить правильный

результат, необходимо прибавить к делимому делитель, уменьшенный на 1. Однако это необходимо только для отрицательных чисел, поэтому для того чтобы не делать проверок, используют следующий алгоритм.

```
; Деление на 2
MOV EAX, x
CDQ ; Расширяем двойное слово до учетверённого.
    ; Если в EAX находится положительное число,
    ; то регистр EDX будет содержать 0,
    ; а если в EAX находится число <0,
    ; то регистр EDX будет содержать -1 (ffffffffh)
SUB EAX, EDX ; Если регистр EDX содержит 0,
             ; то регистр EAX не меняется.
             ; Если же EDX содержит -1 (при EAX<0),
             ; то к EAX будет прибавлена единица
SAR EAX, 1

; Деление на 2^n (в данном примере n = 3)
MOV EAX, x
CDQ        ; Расширяем двойное слово до учетверённого
AND EDX, 111b  ; Если EAX < 0, то EDX содержит
               ; делитель, уменьшенный на 1
ADD EAX, EDX   ; Если EAX<0,
               : прибавляем полученное значение
SAR EAX, 3     ; Если EAX>0, то EDX=0,
               ; и ничего не изменяется
```

Если число беззнаковое или положительное, можно просто использовать сдвиг вправо, который выполняется примерно в 10 раз быстрее, чем деление. Если же для знакового числа неизвестно, положительное оно или отрицательное, то придётся использовать вышеприведённую последовательность команд, которая, однако, также выполняется примерно в 5–7 раз быстрее, чем деление.

8.4.3. Получение остатка от деления

Для беззнаковых и положительных чисел остаток от деления на 2^n – это последние *n* бит числа. Поэтому для получения остатка от деления на 2^n нужно выделить эти последние n бит с помощью операции AND.

```
MOV EAX, x
AND EAX, 111b      ; EAX = EAX % 23
```

Для отрицательного делимого x и положительного делителя n (x%n)=-(-x%n).

```
MOV EAX, x
NEG EAX
AND EAX, 1111B     ; EAX = EAX % 24
NEG EAX
```

9. КОМАНДЫ РАБОТЫ СО СТЕКОМ

Стек используется для передачи параметров и для хранения локальных данных процедур. В принципе, для работы со стеком существуют всего две операции: PUSH и POP. Для каждой операции (положить данные и взять данные) существует несколько команд, которые отличаются тем, с какими данными они работают.

Для того чтобы положить данные в стек, используется команда PUSH:

```
PUSH op
```

Операнд op может быть регистром, ячейкой памяти или непосредственным операндом. Размер операнда должен быть 2 или 4 байта. Операнд кладётся на вершину стека, а значение регистра ESP уменьшается на размер операнда.

Выполнение этой команды можно описать так:

```
ESP = [ESP] – (TYPE op); TYPE op = 2 или 4 байта
[ESP] = op
```

Это означает, что сначала значение регистра ESP уменьшается на 2 или 4 (вычитание происходит по модулю 2^{16}), т.е. ESP сдвигается вверх и указывает на свободную ячейку области стека, а затем в нее записывается операнд.

Флаги команда не меняет.

Для записи в стек числа его предварительно придется поместить в регистр, например:

```
MOV  AX, 5
PUSH AX       ; 5 -> стек
```

Для того чтобы взять данные из стека, используется команда POP:

```
POP op
```

Операнд op может быть регистром или ячейкой памяти. Размер операнда должен быть 2 или 4 байта. В соответствии с размером операнда из вершины стека берутся 2 или 4 байта и помещаются в указанный регистр или ячейку памяти. Значение регистра ESP увеличивается на размер операнда.

Выполнение команды:

```
Op = [ESP]
ESP = [ESP] + (TYPE op);     TYPE op = 2 или 4 байта
```

Кроме этих основных команд существуют ещё команды, которые позволяют сохранять в стеке и восстанавливать из стека содержимое всех регистров общего назначения, и команды, которые позволяют сохранять в стеке и восстанавливать из стека содержимое регистра флагов.

```
PUSHA
PUSHAD
```

Команда PUSHA сохраняет в стеке содержимое регистров AX, CX, DX, BX, SP, BP, SI, DI. Команда PUSHAD сохраняет в стеке содержимое регистров EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI. Для регистра (E)SP сохраняется значение, которое было до того, как мы положили регистры в стек. После этого значение регистра (E)SP изменяется как обычно.

```
POPA
POPAD
```

Эти команды противоположны предыдущим — они восстанавливают из стека значения регистров (E)DI, (E)SI, (E)BP, (E)SP, (E)BX, (E)DX, (E)CX, (E)AX. Содержимое регистра (E)SP не восстанавливается из стека, а изменяется как обычно.

```
PUSHF
PUSHFD
```

Команда PUSHF сохраняет в стеке младшие 16 бит регистра флагов. Команда PUSHFD сохраняет в стеке все 32 бита регистра флагов.

```
POPF
POPFD
```

Команда POPF восстанавливает из стека младшие 16 бит регистра флагов. Команда POPFD восстанавливает из стека все 32 бита регистра флагов.

9.1. Некоторые приемы работы со стеком

Сохранение значений регистров

Стек часто используется для временного хранения значений регистров. При организации вложенных циклов с использованием команды LOOP требуется сохранять значение регистра CX. Это значение можно сохранять в другом регистре или ячейке памяти. Однако с использованием стека это можно сделать так:

```
  PUSH CX       ; CX в стек
  MOV CX, N
L: …
  LOOP L
  POP CX        ; Из стека в CX
```

Пересылка данных через стек

Присваивание X = Y, где X и Y – переменные, можно реализовать так:

```
MOV EAX, Y          |       PUSH Y
MOV X, EAX          |       POP X
```

Проверка на выход за пределы стека

Команды PUSH и POP не осуществляют проверку на выход за пределы стека. Например, если стек пуст, и мы применяем команду чтения из стека, то ошибки не будет (считывается слово, следующее за сегментом стека). Аналогично не будет зафиксирована ошибка, если мы записываем в стек, когда он уже полон. Такие проверки, если требуется, нужно делать самостоятельно. Делаются они так:

```
ESP = 0? – стек пуст?
ESP = N? – стек полон?{N – размер стека в байтах)
```

При пустом стеке в регистре ESP находится число, равное размеру области стека в байтах.

Очистка и восстановление стека

Очистка стека от N слов осуществляется просто увеличением значения регистра ESP на 2*N:

```
ADD ESP, 2*N ; очистка стека от N слов
```

Еще один вариант очистки стека заключается в том, что вначале следует запомнить значение указателя стека SP, до которого нужно будет делать очистку. Затем использовать его по своему усмотрению, но в конце надо просто восстановить в SP это значение:

```
MOV EAX, ESP
…  ; записи в стек
MOV ESP, EAX
```

10. ПРОЦЕДУРЫ

10.1. Синтаксис процедур

Описание процедуры на языке ассемблера выглядит следующим образом:

```
<имя процедуры> PROC
    <тело процедуры>
<имя процедуры> ENDP
```

Несмотря на то что после имени процедуры не ставится двоеточие, это имя является меткой, обозначающей первую команду процедуры.

В языке ассемблера имена и метки, описанные в процедуре, не локализуются внутри неё, поэтому они должны быть уникальны.

Размещать процедуру в программе на языке ассемблера следует таким образом, чтобы команды процедуры выполнялись не сами по себе, а только тогда, когда происходит обращение к процедуре. Обычно процедуры размещают либо в конце секции кода после вызова функции `ExitProcess`, либо в самом начале секции кода, сразу после директивы `.code`.

10.2. Вызов процедуры и возврат из процедуры

Вызов процедуры — это, по сути, передача управления на первую команду процедуры. Для передачи управления можно использовать команду безусловного перехода на метку, являющуюся именем процедуры. Можно даже не использовать директивы `proc` и `endp`, а написать обычную метку.

В связи с тем, что обращаться к процедуре можно из разных мест основной программы, то и возврат из процедуры должен осуществляться в разные места. Поэтому при обращении к процедуре основная программа должна сообщить ей адрес возврата, т.е. адрес той команды, на которую процедура должна сделать переход по окончании своей работы. Поскольку при разных обращениях к процедуре будут указываться разные адреса возврата, то и возврат управления будет осуществляться в разные места программы. Это можно описать следующей схемой, когда адрес возврата передается через регистр:

```
.code
 …
P: …
  …  ; Процедура
  JMP [EBX]     ; Переход по адресу в EBX
START: …
  LEA EBX, L1   ; [EBX] содержит адрес метки L1
  JMP P
L1: …
  LEA EBX, L2   ; [EBX] содержит адрес метки L1
  JMP P
L2: …
END START
```

Адрес возврата можно передавать и через стек.

```
.code
 …
P: …
  …  ; Процедура
  POP EBX       ; Извлекаем адрес возврата
  JMP [EBX]
START: …
  PUSH L1       ; Адрес метки L1 помещаем в стек
  JMP P
L1: …
  PUSH L2       ; Адрес метки L2 помещаем в стек
  JMP P
L2: …
END START
```

Система команд языка ассемблера включает специальные команды для вызова процедуры и возврата из процедуры.

```
CALL <имя проц.> ; Вызов процедуры
                 ; (переход с возвратом)
RET              ; Возврат из процедуры (return)
```

Команда CALL записывает адрес следующей за ней команды в стек и осуществляет переход на первую команду указанной процедуры. Команда RET считывает из вершины стека адрес и выполняет переход по нему.

10.3. Передача параметров процедуры

Существуют несколько способов передачи параметров в процедуру.

10.3.1. Передача через регистры

Если процедура получает небольшое число параметров, идеальным местом для их передачи оказываются регистры. Существуют соглашения о вызовах, предполагающие передачу параметров через регистры ECX и EDX. Этот метод самый быстрый, но он удобен только для процедур с небольшим количеством параметров.

Рассмотрим типичный пример. Пусть надо вычислить R = MAX (A, B) – MAX (B + 1, 5), где все числа знаковые и

размером в слово. Опишем процедуру, которая находит наибольшее из двух значений, находящихся в регистрах AX и BX, а результат помещает в регистр AX.

Тогда программа может быть такой:

```
; R = MAX(A, B) + MAX(B+1, 5)
.686
include /masm32/include/io.asm
.data
A DW ?
B DW ?
R DW ?

.code
MAX PROC
  CMP AX, BX
  JGE RT
  MOV AX, BX
RT: RET
MAX ENDP

start:
  print "A="    ; Ввод данных
  inint A
  print "B="    ; Ввод данных
 inint B

  MOV AX, A
  MOV BX, B
  CALL MAX
  MOV R, AX
  print "R="    ; Вывод результата
  outint16 R
  newline
  println "= = = = = = = ="

  MOV AX, B
  ADD AX, 1     ; [AX]=B+1
  MOV BX, 5
  CALL  MAX
  ADD R, AX

  print "R="    ; Вывод результата
  outint16 R
  inkey         ; ожидание нажатия клавиши
  exit
end start
```

В этом примере параметры передаются по значению: перед обращением к процедуре основная процедура вычисляет значения фактических параметров и именно эти значения записывает в регистры.

```
.686
include \masm32\include\io.asm

.data
  X DD 10 DUP (1)
  Y DD 10 DUP (2)
  L_XY = type X
  NX DD ?
  NY DD ?

.code
OUT_ARR1 PROC
L_PRN:
  outint32 [EBX], 6
  ADD EBX, L_XY
  LOOP L_PRN
  newline
  ret
OUT_ARR1 endp

LStart:
  print "NX="
  inint NX
  MOV ECX, NX
  LEA EBX, X
  CALL OUT_ARR1
  print "= = = = = = = = = = = ="
  newline

  print "NY="
  inint NY
  MOV ECX, NY
  LEA EBX, Y
  CALL OUT_ARR1

LExit:
  newline
  inkey "Press any key to exit."
  exit
end LStart
```

10.3.2. Передача в глобальных переменных

Параметры процедуры можно записать в глобальные переменные, к которым затем будет обращаться процедура. Однако этот метод является неэффективным, и его использование может привести к тому, что рекурсия и повторная входимость станут невозможными.

Приведем пример передачи параметров через глобальные переменные. В данном случае входные данные находятся в глобальных переменных AA и BB, результат помещается в поле R.

```
; R = MAX(AA, BB) + MAX(BB+1, 5)
…
.data
…
A  DW ?
B  DW ?
AA DW ?    ; Параметр
BB DW ?    ; Параметр
R  DW ?

.code
MAX PROC
  MOV AX, AA
  CMP AX, BB
  JGE RT
  MOV AX, BB
RT: MOV R, AX
  RET
MAX ENDP

start:
; Ввод данных
…
  MOV AX, A
  MOV AA, AX    ; AA=A
  MOV AX, B
  MOV BB, AX    ; BB=B
  CALL MAX
  MOV BX, R

  MOV AX, B
  ADD AX, 1
  MOV AA, AX    ; AA=B+1
  MOV BB, 5          ; BB=5
```

```
  CALL MAX
  ADD R, BX

  ; вывод результата
…
exit
end start
```

Пусть имеется процедура на языке C++:

```
void UMN8(int *N) {
  *N = *N * 8;
  return;
}
```

Очевидно, что процедура, работающая с адресом переменной (*N), должна получать из вызывающей программы адрес своего фактического параметра для того, чтобы при необходимости изменить значение этого параметра.

Запишем процедуру UMN8, предполагая, что адрес параметра находится в регистре EBX. При этом будем считать, что фактические параметры размещаются в сегменте данных (адрес параметра может находиться в любом регистре-модификаторе, т.е. EBX, EBP, ESI или EDI).

```
.686
include /masm32/include/io.asm
.data
N DD ?
.code
UMN8 PROC
  SHL DWORD PTR [EBX], 3 ; X = X * 8
 RET
UMN8 ENDP

start:
  print "N="    ; Ввод данных
  inint N

  LEA EBX, N    ; BX=адрес N
  CALL UMN8     ; UMN8(N)
  print "N*8=" ; Вывод результата
  outint32 N
  newline
  inkey         ; Ожидание нажатия клавиши
  exit
end start
```

Процедуру UMN8 можно переписать на ассемблере (UMN8_ASM) для обеспечения эффективной работы программы.

```
void UMN8(int *N) {
  *N = *N * 8;
  return;
}

void UMN8_ASM(int *N) {
  _asm {
     MOV EBX, N
     SHL DWORD PTR[EBX], 3  ; N: = N * 8
  }
  return;
}

int main() {
  int N, NN; // Тестируемое число
  cout << "N=";  cin >> N;
  NN = N;
  UMN8(&N);
  cout << "N=" << N << endl;
  UMN8_ASM(&NN);
  cout << "NN=" << NN << endl;
…
}
```

10.3.3. Передача в блоке параметров

Блок параметров — это участок памяти, содержащий параметры и располагающийся обычно в сегменте данных. Процедура получает адрес начала этого блока при помощи любого метода передачи параметров (в регистре, переменной, стеке, коде или даже в другом блоке параметров).

10.3.4. Передача через стек

Передача параметров через стек — наиболее распространённый способ. Именно его используют языки

высокого уровня, такие как C++ и Паскаль. Параметры помещаются в стек непосредственно перед вызовом процедуры.

В стек кладутся несколько параметров и затем вызывается процедура. Команда CALL также кладёт в стек адрес возврата, таким образом, адрес возврата оказывается в стеке поверх параметров. Однако поскольку в рамках своего участка стека процедура может обращаться без ограничений к любой ячейки памяти, нет необходимости перекладывать куда-то адрес возврата, а потом возвращать его обратно в стек. Для обращения к первому параметру используют адрес [ESP+4] (прибавляем 4, так как на архитектуре Win32 адрес имеет размер 32 бита), для обращения ко второму параметру — адрес [ESP+8] и т.д.

В приведенном далее примере переменные A=1, B=2, R=3 помещаются в стек, процедура изменяет их значения в соответствии с адресами:

	Адрес возврата	PUSH R	PUSH B	PUSH A	
…	EBP=ESP	ESP+4	ESP+8	ESP+12	…

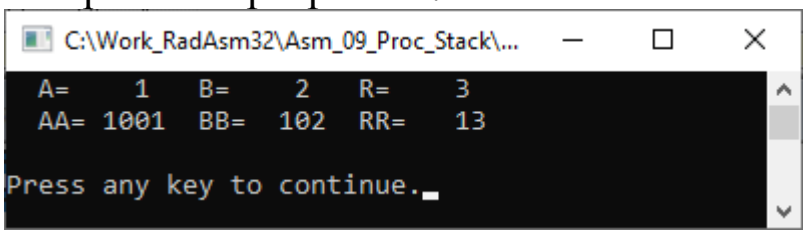
После обращения к процедуре извлекаем результаты работы в переменные RR, BB, AA.

Пример 27. Передача параметров через стек.

```
.686
include /masm32/include/io.asm
.data
  A DD 1        ; Инициализируем входные параметры
  B DD 2
  R DD 3

  AA DD ?       ; Выходные параметры
  BB DD ?
  RR DD ?
  Col_Pos = 5   ; Количество позиций при выводе

.code
EX_PROC_STACK PROC
  MOV EAX, [ESP+4]
  ADD EAX, 10         ; Изменяем R
  MOV [ESP+4], EAX

  MOV EAX, [ESP+8]
  ADD EAX, 100        ; Изменяем B
```

```
  MOV [ESP+8], EAX

  MOV EAX, [ESP+12]
  ADD EAX, 1000      ; Изменяем A
  MOV [ESP+12], EAX
  RET
EX_PROC_STACK ENDP

start:
; Вывод данных
  print "  A="
  outint32 A, Col_Pos
  print "   B="
  outint32 B, Col_Pos
  print "   R="
  outint32 R, Col_Pos
  newline

  PUSH A        ; Помещаем в стек входные параметры
  PUSH B
  PUSH R

  CALL EX_PROC_STACK

  POP RR  ; Извлекаем из стека выходные параметры
  POP BB
  POP AA
  ; Вывод результата
  print "  AA="
  outint32 AA, Col_Pos
  print "  BB="
  outint32 BB, Col_Pos
  print "  RR="
  outint32 RR, Col_Pos
  newline
  ADD ESP, 12  ; Освобождаем 12 байтов стека

  inkey         ; Ожидание нажатия клавиши
  exit
end start

END
```

Результат работы программы:



10.4. Возврат результата процедуры

Для передачи результата процедуры обычно используется регистр EAX. Этот способ используется не только в программах на языке ассемблера, но и в программах на языке C++.

Иногда удобно передать в качестве параметра адрес ячейки памяти, куда будет записан результат.

```
; Передача параметров через стек,
; возврат результата по адресу
…
.data
A DD 11          ; Инициализируем входные параметра
B DD 22
R DD ?
Col_Pos = 5      ; Количество позиций при выводе

.code
EX_PROC_A_ADD_B PROC
  MOV EAX, [ESP+4]   ; EAX = A
  MOV EBX, [ESP+8]   ; EBX = B
  ADD EAX, EBX       ; EAX = A + B
  MOV EDX, [ESP+12]  ; EDX - адрес R
  MOV [EDX], EAX     ; R = EAX
  RET
EX_PROC_A_ADD_B ENDP

start:
…
  PUSH offset R ; В стеке адрес переменной R
  PUSH B        ; Помещаем в стек входные параметры
  PUSH A
  CALL EX_PROC_A_ADD_B
  print "  R="
  outint32 R, Col_Pos
  ADD ESP, 12   ; Освобождаем 12 байт стека
  …
END start
END
```

10.5. Сохранение регистров в процедуре

Практически любые действия в языке ассемблера требуют использования регистров. Однако регистров очень мало и даже в небольшой программе невозможно будет разделить регистры между частями программы.

Поэтому для сохранения регистров обычно используется стек. Можно сохранить используемые регистры по одному с помощью команды PUSH или все сразу с помощью команды PUSHAD. В первом случае в конце процедуры нужно будет восстановить значения сохранённых регистров с помощью команды POP в обратном порядке. Во втором случае для восстановления значений регистров используется команда POPAD.

При сохранении регистров указатель стека изменится на некоторое значение, зависящее от количества сохранённых регистров. Это нужно учитывать при вычислении адресов параметров процедуры, передаваемых через стек.

```
; Процедура получает два параметра по 4 байта
ANY_PROC PROC
  PUSHAD                ; Сохраняем все регистры
  MOV EAX, [ESP+4+32]   ; Извлекаем параметры
                        ; из стека.
                        ; Адрес вычисляется
  MOV EBX, [ESP+8+32]   ; с учётом 32 байт,
                        ; использованных
                        ; при сохранении регистров
  …
  POPAD               ; Извлекаем сохранённые регистры
  RET
ANY_PROC ENDP
```

10.6. Рекурсивные процедуры

Рекурсия — ресурсоёмкий способ реализации алгоритмов. Она требует много места для хранения локальных данных на каждом шаге рекурсии, кроме того, рекурсивные процедуры обычно выполняются не очень быстро. Поэтому языку ассемблера, предназначенному для написания быстрых программ, рекурсия, в общем, не свойственна. Но при желании и на ассемблере можно написать рекурсивную процедуру. Принципы реализации

рекурсивной процедуры на языке ассемблера такие же, как и на других языках. В процедуре должна быть терминальная ветвь, в которой нет рекурсивного вызова, и рабочая ветвь.

При реализации рекурсивных процедур становится особенно важным использование стека для передачи параметров и адреса возврата, что позволяет хранить данные, относящиеся к разным уровням рекурсивных вызовов, в разных областях памяти.

Для примера рассмотрим рекурсивную процедуру вычисления факториала целого беззнакового числа. Процедура получает параметр через стек и возвращает результат через регистр EAX.

```
FACTORIAL PROC
  MOV EAX, [ESP+4]   ; Заносим в EAX параметр
процедуры
  TEST EAX, EAX      ; Проверяем значение в регистре
EAX
  JZ END             ; Если EAX = 0, то выход
  DEC EAX            ; EAX = EAX-1
  PUSH EAX           ; Кладём в стек параметр для
                     ; следующего рекурсивного вызова
  CALL FACTORIAL     ; Вызываем процедуру
  ADD ESP, 4         ; Очищаем стек, т.к. процедура
                     ; использует RET без параметров
  MUL DWORD PTR [ESP+4] ;Умножаем EAX, результат
                     ; предыдущего вызова, на параметр
                     ; текущего вызова процедуры
  RET      ; Возврат из процедуры (без параметров)
END:
  INC EAX ; Если EAX был равен 0, записываем в EAX 1
  RET      ; Возврат из процедуры (без параметров)
FACTORIAL ENDP
```

10.7. Использование внешних процедур

Для связи посредством внешних процедур создается многофайловая программа. При этом в общем случае возможны два варианта вызова:

– программа на языке высокого уровня вызывает процедуру на языке ассемблера;

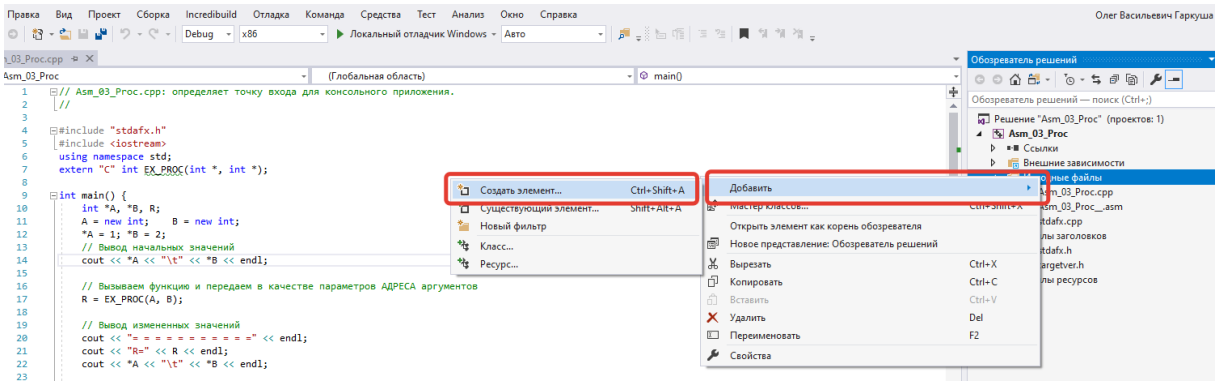
– программа на языке ассемблера вызывает процедуру на языке высокого уровня.

Рассмотрим более подробно первый вариант. Получение и передача параметров в языке ассемблера производится явно, без помощи транслятора. При связи процедуры, написанной на языке ассемблера, с языком высокого уровня, необходимо учитывать соглашение по передаче параметров.

Для совмещения файлов, написанных на разных языках программирования, необходимо к существующему проекту на C++ добавить файлы .asm. Для этого выполняем следующие действия.

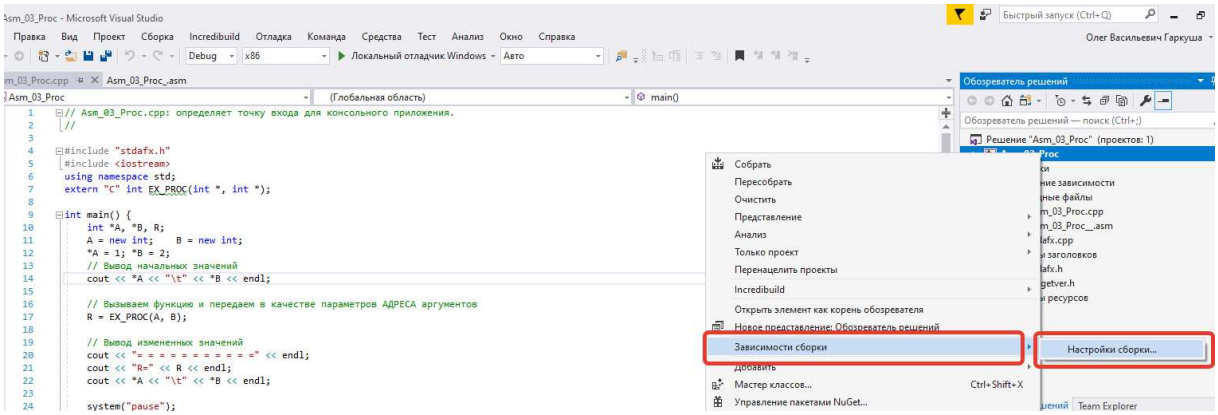
Во вкладке Обозреватель решений > Исходные файлы (правая кнопка мыши) > Добавить > Создать элемент.

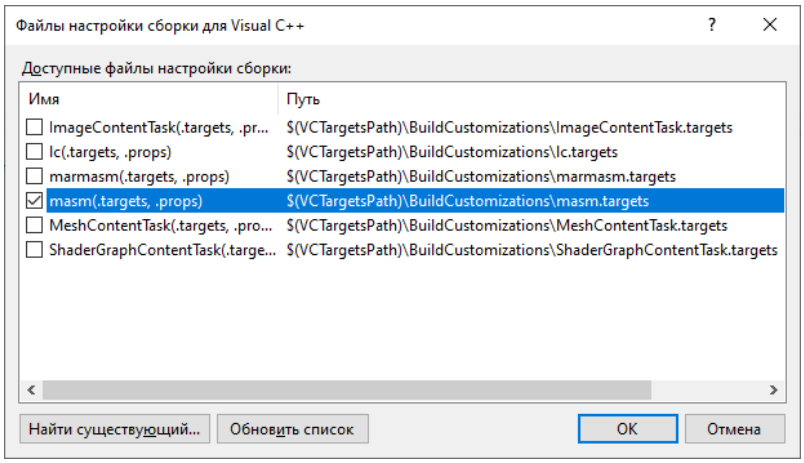
Добавляемый элемент должен иметь расширение *.asm, которое необходимо указать явно. Имя файла должно отличаться от имени файла .cpp, так как после компиляции оба будут иметь расширение .obj.



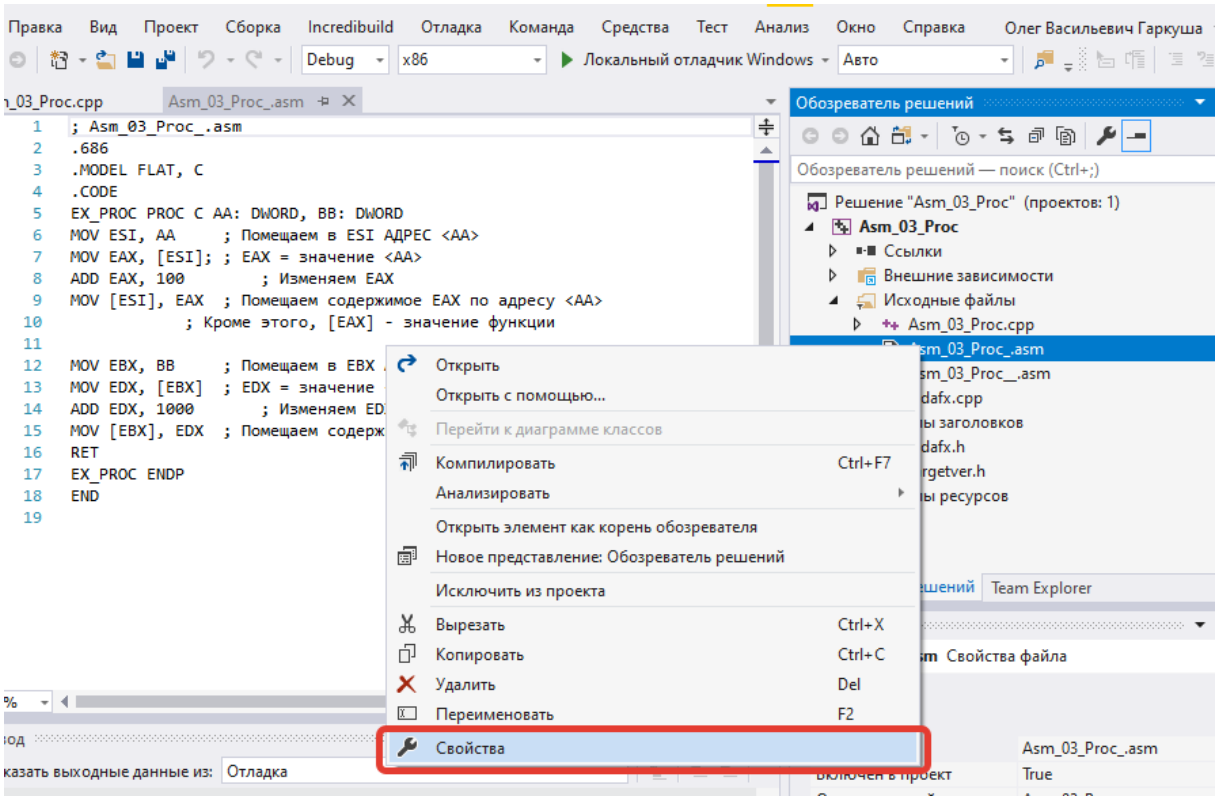
Следующий шаг: подключение инструмента Microsoft Macro Assembler.

Во вкладке Обозреватель решений (правая кнопка мыши) > Зависимости сборки > Настройки сборки. В появившемся окне выбираем masm.

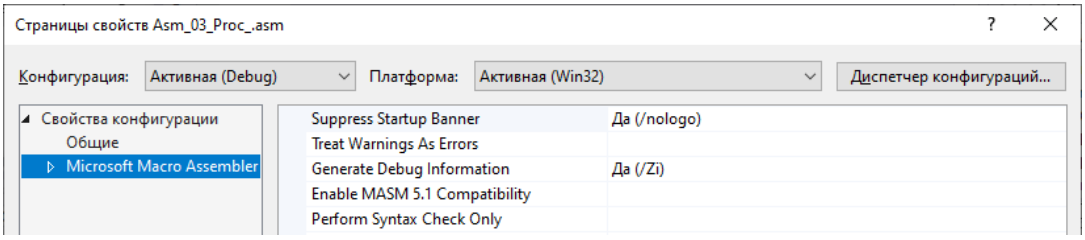




Для файла на языке ассемблера (правая кнопка мыши) > Свойства.



В появившемся окне выбираем для этого файла инструмент Microsoft Macro Assembler.



Далее, как обычно, выполняем построение проекта.

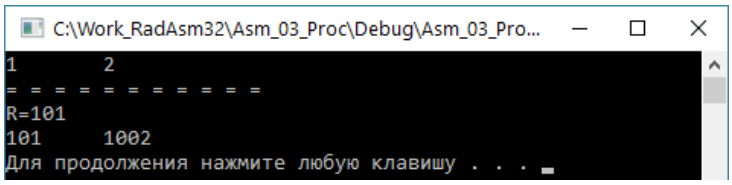
Приведем пример проекта, содержащего внешнюю процедуру. В программе создаются ссылки на переменные A и B. Адреса этих переменных передаются в процедуру на языке ассемблера.

Процедура EX_PROC изменяет значения этих переменных и передает их в вызывающую программу. Обращаем внимание, что вызов происходит как к функции. Результат выполнения в процедуре помещается в регистр EAX и присваивается переменной R.

```
// Asm_03_Proc.cpp
#include <iostream>
using namespace std;
extern "C" int EX_PROC(int *, int *);

int main() {
  int *A, *B, R;
  A = new int;B = new int;
  *A = 1;*B = 2;
  // Вывод начальных значений
  cout << *A << "\t" << *B << endl;

  // Вызываем функцию и передаем в качестве параметров
  // АДРЕСА аргументов
  R = EX_PROC(A, B);
  // Вывод измененных значений
  cout << "= = = = = = = = = = =" << endl;
  cout << "R=" << R << endl;
  cout << *A << "\t" << *B << endl;

  return 0;
}

// Asm_03_Proc_.asm
.686
.model FLAT, C
.code
  EX_PROC PROC C AA: DWORD, BB: DWORD
  MOV ESI, AA     ; Помещаем в ESI АДРЕС <AA>
  MOV EAX, [ESI];; EAX = значение <AA>
  ADD EAX, 100    ; Изменяем EAX
```

```
  MOV [ESI], EAX ; Помещаем EAX по адресу <AA>
                 ; Кроме этого, [EAX] - значение функции
  MOV EBX, BB    ; Помещаем в EBX АДРЕС <BB>
  MOV EDX, [EBX] ; EDX = значение <BB>
  ADD EDX, 1000  ; Изменяем EDX
  MOV [EBX], EDX ; Помещаем EDX по адресу <BB>
  RET
  EX_PROC ENDP
END
```

Результат работы программы:

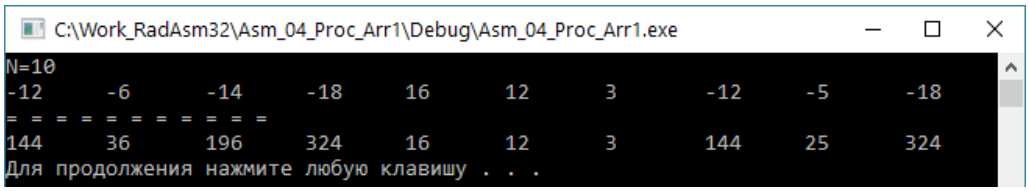


Вернемся к примеру о замене в одномерном массиве каждого отрицательного элемента его квадратом. При этом ассемблерную вставку оформим в виде внешней процедуры. В процедуру `EX_PROC_ARR1(X, N)` передаем адрес массива `X` и количество элементов `N`.

```
// Asm_04_Proc_Arr1.cpp
#include …
extern "C" int EX_PROC_ARR1(int *, int *);
int main()
{
…
  int *X;     // Ссылка на массив. Элементы размером DD
  int i, *N;  // Количество элементов массива
  N = new int;
  cout << "N=";    cin >> *N;
  X = new int[*N]; // Ссылка на массив

  // Заполнение массива случайными числами из [A, B]
  // Вывод элементов массива
  // Вызываем функцию и передаем АДРЕСА аргументов
  EX_PROC_ARR1(X, N)
  //Вывод элементов массива
…
}
```

```
// Asm_04_Proc_Arr1_.asm
.686
.model FLAT, C
.code
EX_PROC_ARR1 PROC C XX: DWORD, NN: DWORD
type_X = TYPE XX; Размер элемента массива =4

  MOV EBX, XX      ; В EBX АДРЕС первого элемента XX;
  MOV ECX, NN
  MOV ECX, [ECX]   ; В ECX = NN - количество элементов
L01:
  MOV EAX, [EBX]   ; Очередной элемент массива
  CMP EAX, 0       ; Сравнение элемента массива с 0
  JGE L02          ; Если >=0, то переход к следующему
    MUL EAX        ; EDX:EAX = EAX * EAX
    MOV [EBX], EAX
  L02 :
  ADD EBX, type_X
  LOOP L01
  RET
  EX_PROC_ARR1 ENDP
END
```

Результат работы программы:



11. МАКРОСРЕДСТВА

11.1. Макросы

В рассмотренной схеме организации подпрограмм независимо от количества обращений к подпрограмме в памяти всегда находится один ее экземпляр. Такие подпрограммы называются закрытыми, или замкнутыми.

Возможна и другая схема, при которой текст подпрограммы подставляется вместо каждого обращения к ней. Такие

подпрограммы называются открытыми, или макросами. Соответствующие средства языка называются макросредствами.

Будем использовать следующую терминологию:

– описание макроса называется макроопределением;

– ссылка на макрос – макрокомандой;

– процесс замены макрокоманды на макрос – макроподстановкой;

– результат такой подстановки – макрорасширением.

Совокупность языковых средств, позволяющих создавать программы на ЯА с использованием макросредств, называется макроассемблером.

11.2. Макроопределения и макрокоманды

Макроопределение в ассемблере имеет следующий вид:

```
<имя макроопределения> MACRO [<параметры>]
     <тело макроопределения>
ENDM
```

Тогда вызов макроса (макрокоманда) указывается в виде

```
<имя макроса> <параметры>
```

Как обычно в описании макроса параметры называются формальными, а при вызове – фактическими. Тело макроопределения – последовательность любых предложений ЯА.

Вызов макроса заменяется в процессе ассемблирования соответствующим телом, в котором каждый формальный параметр заменяется соответствующим фактическим.

Рассмотрим пример:

```
ADDM MACRO X, Y
  MOV AX, X
  ADD AX, Y
ENDM
```

Вызов вида `ADDM A, BX` будет заменен макроассемблером на последовательности команд:

```
  MOV AX, A
  ADD AX, BX
```

Макроопределения могут быть размещены в любом месте текста программы, но обязательно до первой ссылки на этот макрос.

Число фактических параметров, указываемых в макрокоманде, должно равняться числу формальных параметров макроса, причем i-й фактический параметр соответствует i-му формальному параметру. Однако если фактических указано больше, то лишние фактические параметры игнорируются, а если меньше, то считается, что в качестве недостающих фактических параметров заданы пустые тексты.

Пример 28. Определить макрос, моделирующий конструкцию

```
If X=Y Then goto L;
```

Будем считать, что параметрами этой конструкции являются величины X и Y, а также метка L. Соответствующий макрос имеет вид

```
IFEQ MACRO X, Y, L
  MOV AX, X
  CMP AX, Y
  JE L
ENDM
```

В последующем всякий раз вместо последовательности команд сравнения и передачи управления можно использовать конструкцию IFEQ.

Используя этот макрос, решим задачу равенства цифр двузначного числа: дано двузначное число A, если его цифры равны, то в F положить 1, иначе 0:

```
      MOV AX, A
      DIV DES
      IFEQ AX, DX, M1
      MOV F, 0
      JMP M2
M1:   MOV F, 1
M2:
```

```
      MOV AX, A
      DIV DES
      MOV AX, AX
      CMP AX, DX
      JE M1
      MOV F, 0
      JMP M2
M1:   MOV F, 1
M2:
```

Слева указан текст программы, а справа — те команды, которые будут реально выполняться.

Замечание: если макрокоманда помечена меткой, то в макрорасширении эта метка размещается в отдельной строке, а тело макроса начинается со следующей строки, так как в общем случае первая команда тела макроса может быть помечена своей меткой.

Из примера видно, что использование макросов сокращает размеры исходного текста программы и позволяет составлять программу в терминах более крупных операций. Программист может в виде макросов описать наиболее часто используемые операции и пользоваться такой библиотекой для составления программ.

Очень часто макроопределения используют для организации стандартной последовательности команд вызова некоторой подпрограммы.

Пусть MAX – некоторая процедура, вычисляющая наибольший элемент среди величин, находящихся в регистрах AX и BX, результат помещается в AX. Требуется вычислить MAX (A, B) + MAX (C, D). Обычная последовательность команд:

```
  MOV AX, A
  MOV BX, B
  CALL MAX
  MOV DX, AX
  MOV AX, C
  MOV BX, D
  CALL MAX
  ADD AX, DX
```

Используя макросредства, этот фрагмент программы можно записать в виде

```
CALLMAX MACRO P1, P2
  MOV AX, P1
  MOV BX, P2
  CALL    MAX
ENDM
  CALLMAX A, B
  MOV DX, AX
  CALLMAX C, D
  ADD AX,DX
```

11.3. Макросы и процедуры

Рассмотрим различия и общие черты между макросами и процедурами.

И макрос, и процедура описываются в программе только один раз. В обоих случаях в них указываются короткие ссылки на описание макроса или процедуры. Таким образом, с точки зрения процесса написания текста программы, различия между макросами и процедурами нет.

После трансляции программы процедура остается в единственном экземпляре, а при использовании макроса его тело подставляется во все места, где указано обращение к нему. Очевидно, что размеры программы при подстановке макрорасширения увеличиваются. Таким образом, применение процедур более эффективно по объему используемой памяти.

Однако недостатком использования процедур является увеличение времени работы программы, так как при обращении к процедуре выполняются команды пересылки ее параметров в те или иные регистры (либо в стек) и команда вызова процедуры. Кроме этого по окончании работы процедуры необходимо выполнить команду возврата из нее. При использовании макроса все эти команды выполнять не надо.

При подстановке макрорасширения на место макрокоманды также затрачивается время, но это происходит на этапе трансляции, до выполнения программы.

Итак, использование в программе процедур дает выигрыш по памяти, а использование макросов – выигрыш по времени. Выбор того или иного средства определяется отдельно для конкретной задачи.

Общая рекомендация такова: большие фрагменты кода рекомендуется описывать как процедуры, а маленькие – как макросы.

11.4. Директива LOCAL

Рассмотрим пример построения макроопределения, которое по значениям переменных A и B находит их максимум и помещает его в AX.

```
MAX MACRO A, B
      MOV AX, A
      CMP AX, B
      JGE L
      MOV AX, B
L:    NOP  ; нет операции
ENDM
```

Используя этот макрос, можно вычислить величину y = MAX (x, y) + MAX (u, v):

```
   MAX X, Y
   MOV DX, AX
   MAX U, V
   ADD DX, AX
   MOV Y, DX
   …
```

```
   MOV AX, X
   CMP AX, Y
   JGE L
   MOV AX, Y
L: NOP
   MOV DX, AX
   MOV AX, U
   CMP AX, V
   JGE L
   MOV AX, V
L: NOP
   ADD DX, AX
   MOV Y, DX
```

Слева указан текст программы, а справа — те команды, которые будут реально выполняться.

Однако в окончательном тексте программы появились две команды, помеченные одной и той же меткой L, а это ошибка.

Простейшим решением этой проблемы является помещение метки макроса в список его параметров. В дальнейшем при вызове макроса необходимо указывать различные значения меток как фактических параметров. Однако это неудачное решение, так как метка относится только к макросу и носит локальный характер.

В ЯА есть более предпочтительное решение данной проблемы. После заголовка макроса (директивы MACRO) нужно указать директиву макроязыка:

```
LOCAL L1, L2, …, Ln
```

где Li – имена меток, локализованных в макроопределении. Тогда при макроподстановке макрогенератор будет эти имена заменять на специальные имена вида:

```
  ??xxxx
```

где xxxx – четырехзначное шестнадцатеричное число, т.е. на имена ??0000, ??0001 и так далее до ??FFFF. Таким образом, возможны 164 различные метки.

Изменим макрос MAX следующим образом:

```
MAX MACRO A, B
  LOCAL L
  MOV AX, A
     CMP AX, B
     JGE L
     MOV AX, B
L:
  NOP      ; нет операции
ENDM
```

При таком макроопределении MAX команды, которые будут реально выполняться, имеют вид:

```
  MOV AX, X
  CMP AX, Y
  JGE ??0000
  MOV AX, Y
??0000:
  MOV DX, AX
  MOV AX, U
  CMP AX, V
  JGE ??0001
  MOV AX, V
??0001:
  ADD DX, AX
  MOV Y, DX
```

Пример 29. Дан массив целых чисел. Требуется найти количество совершенных элементов массива. Опишем макрос с параметром, который проверяет, является ли число совершенным.

Приведем полный текст программы, готовой к выполнению.

```
.686
include /masm32/include/io.asm
.data
N DD ?          ; Количество элементов массива
```

```
A DD 100 DUP(?); Массив
type_A = TYPE A;
SUM DD ?        ; Сумма делителей тестируемого числа
F DB ?          ; флаг = 1, если число совершенное,
                ;      = 0 - иначе
K DW ?          ; Количество совершенных чисел

.code
; Описание макроса для определения совершенного числа
SOV MACRO N
  LOCAL L1, L2, L3  ; Описание локальн. меток
  PUSHAD            ; Сохранение регистров в стек
  MOV F, 0          ; Если F=0 - число не совершенное
  MOV SUM, 0        ; Сумма делителей числа X
  MOV ECX, N
  SHR ECX, 1        ; CX = CX / 2
L2:
    MOV EAX, N
    MOV EDX, 0
    DIV ECX         ; Потенциальный делитель
    CMP EDX, 0
    JNE L1
    ADD SUM, ECX    ; Сумма делителей числа N
L1: LOOP L2

  MOV EAX, N
  CMP EAX, SUM      ; Сравнение SUM = N ?
  JNE L3
  MOV F, 1          ; F = 1 - число совершенное
L3:
  POPAD             ; Восстановление регистров
ENDM

start:
  print "N="
  inint32 N
  println "Введите элементы массива"

  MOV ECX, N   ; Количество элементов массива
  MOV EBX, 0
L11:
  inint32 A[EBX]
  ADD EBX, type_A
  LOOP L11

  MOV K, 0     ; Количество совершенных чисел
  MOV ECX, N   ; Количество элементов массива
  MOV EBX, 0
L22:
  SOV A[EBX]   ; Вызов макроса с параметром -
```

```
                ; элемент массива
  CMP F, 1      ; F = 1 - число совершенное,
                ;   = 0 - нет
  JNE L33
  INC K         ; Количество совершенных элементов
L33:
  ADD EBX, type_A
  LOOP L22
  print "Количество совершенных элементов K="
  outint16 K    ; Вывод K
  newline
  inkey         ; ожидание нажатия клавиши
  exit
end start
END
```

12. ОПТИМИЗАЦИЯ ПРОГРАММ

Популярным применением ассемблера считается оптимизация программ, т.е. уменьшение времени выполнения программ по сравнению с языками высокого уровня. Но если просто переписать текст, например, с языка С на ассемблер, переводя каждую команду наиболее очевидным способом, часто оказывается, что процедура на языке С выполняется быстрее.

Проблему оптимизации принято делить на три основных уровня:

1) выбор наиболее оптимального алгоритма — высокоуровневая оптимизация;

2) наиболее оптимальная реализация алгоритма — оптимизация среднего уровня;

3) подсчёт тактов, тратящихся на выполнение каждой команды, и оптимизация их порядка для конкретного процессора — низкоуровневая оптимизация.

12.1. Высокоуровневая оптимизация

Выбор оптимального алгоритма для решения задачи всегда приводит к лучшим результатам, чем любой другой вид оптимизации. Поиск лучшего алгоритма – универсальная стадия, и она относится не только к ассемблеру, но и к любому языку программирования.

12.2. Оптимизация среднего уровня

Реализация алгоритма на данном конкретном языке программирования – именно здесь можно получить выигрыш в скорости в десятки раз. Есть общие приёмы оптимизации, например, хранение переменных, с которыми выполняется активная работа, в регистрах, использование таблиц переходов вместо длинных последовательностей проверок и условных переходов и т.п. Можно утверждать, что все проблемы оптимизации на среднем уровне так или иначе связаны с циклами.

12.2.1. Вычисление констант вне цикла

Важным правилом при создании цикла на любом языке программирования является вынос за его пределы всех переменных, которые не изменяются на протяжении цикла. В случае ассемблера имеет смысл также по возможности разместить все переменные, которые будут использоваться внутри цикла, в регистры, а старые значения нужных после цикла регистров сохранить в стеке.

12.2.2. Перенос проверки условия в конец цикла

Циклы типа `while` или `for`, которые так часто применяются в языках высокого уровня, оказываются менее эффективными по сравнению с циклами типа `until` из-за того, что в них требуется лишняя команда перехода.

```
; for (i = start_i; i < n; i++) <тело цикла>
  MOV EDI, start_i ; Начальное значение счётчика
  MOV ESI, n       ; Конечное значение счётчика
L1:
  CMP edi, esi     ; Пока EDI < ESI – выполнять
  JE L2
    <тело цикла>
  INC EDI
  JMP L1
L2:
```

```
  i = start_i;
  do {
  <тело цикла> }
  while (i < n);

  MOV EDI, start_i
  MOV ESI, n
L1:
  <тело цикла>
  INC EDI
  CMP EDI, ESI
  JB L1       ; Пока EDI < ESI – выполнять
```

Цикл с постусловием всегда выполняется хотя бы один раз, и во многих случаях перед циклом приходится добавлять ещё одну проверку, но в любом случае даже небольшое уменьшение тела цикла всегда оказывается необходимой операцией.

12.2.3. Выполнение цикла в обратном порядке

Циклы, в которых значение счётчика растёт, можно реализовать вообще без операции сравнения, выполняя цикл в обратном направлении. Флаги меняются не только командой сравнения, но и многими другими. В частности, команда `DEC` меняет флаги `AF, OF, PF, SF` и `ZF`. Команда сравнения кроме этих флагов меняет также флаг `CF`, но для сравнения с нулём можно обойтись флагами `SF` и `ZF`.

```
; Цикл от 10 до 1
  MOV   EDX, 10
L1:
   <тело цикла>
   DEC EDX    ; Уменьшаем EDX на 1. Если EDX = 0, то ZF =
1
   JNZ L1     ; Переход если ZF = 0.
              ; Когда EDX=0, ZF=1, выходим из цикла

; Цикл от 10 до 0
   MOV EDX, 10
L1:
   <тело цикла>
```

```
   DEC EDX     ; Уменьшаем EDX на 1. Если EDX=-1, то SF = 1
   JNS L1      ; Переход если SF = 0.
```

Не все циклы можно реализовать в обратном направлении. Но в целом всегда следует стремиться к циклам, выполняющимся задом наперёд.

12.2.4. «Разворачивание» циклов

Для небольших циклов время выполнения проверки условия и перехода на начало цикла может оказаться значительным по сравнению с временем выполнения самого тела цикла. В таких случаях можно вообще не создавать цикл, а просто повторить его тело нужное число раз. Для очень коротких циклов можно, например, удваивать или утраивать тело цикла, если, конечно, число повторений кратно двум или трём. Кроме этого бывает удобно часть работы сделать в цикле, а часть развернуть.

```
; Цикл от 10 до -1
   MOV edx, 10
L1:
   <тело цикла>
   DEC edx
   JNS L1          ; Выходим из цикла, когда EDX станет =-1
   <тело цикла>    ; Но повторяем тело цикла ещё раз
```

Умение оптимизировать программы нельзя сформулировать в виде набора простых алгоритмов — слишком много существует различных ситуаций, в которых всякий алгоритм оказывается неоптимальным.

Именно потому, что оптимизация всегда занимает очень много времени, рекомендуется приступать к ней только после того, как программа окончательно написана.

12.3. Низкоуровневая оптимизация

Переходить к этому уровню оптимизации можно только после того, как текст программы окончательно написан и максимально оптимизирован на среднем уровне.

Перечислим основные рекомендации:

– по возможности использовать регистр EAX;

– если к переменной в памяти, адресуемой со смещением, выполняется несколько обращений – загрузить её в регистр;

– не использовать сложных команд — ENTER, LEAVE, LOOP, строковых команд, если аналогичное действие можно выполнить небольшой последовательностью простых команд;

– следует программировать условия и переходы так, чтобы переход выполнялся по менее вероятному событию;

– организовать программу последовательным образом. В результате очередь команд будет почти всегда заполнена, а программу будет легче читать, сопровождать и отлаживать. Процедуры, особенно небольшие, нужно не вызывать, а встраивать. Это увеличивает размер программы, но даёт существенный выигрыш во времени её исполнения;

– использовать короткую форму команды JMP, где возможно (JMP short <метка>);

– команда LEA быстро выполняется и имеет много неожиданных применений;

– стараться выравнивать данные и метки по адресам, кратным 2/4/8/16;

– если команда обращается к 32-битному регистру, например, EAX, сразу после команды, выполнявшей запись в соответствующий частичный регистр (AX, AL, AH), может происходить пауза в один или несколько тактов.

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Кольцов Ю.В., Гаркуша О.В., Добровольская Н.Ю. Программирование на языке ассемблера: учеб. пособие. Краснодар: Кубанский гос. ун-т, 2011.
2. Кольцов Ю.В., Гаркуша О.В., Добровольская Н.Ю., Харченко А.В. Программирование на языке ассемблера IA-32 в среде RadASM: учеб. пособие. Краснодар: Кубанский гос. ун-т, 2014.
3. Репозиторий библиотеки и инсталляторов среды. URL: https://github.com/KubSU/SIOMASM
4. Ирвин К. Язык ассемблера для процессоров Intel. М.: «Вильямс», 2005.
5. Пильщиков В.Н. Assembler. Программирование на языке ассемблера IBM PC. М.:Диалог-МИФИ, 2005.
6. Юров В.И. Assembler. Практикум. 2-е изд. СПб.: Питер, 2006.
7. Программирование на языке ассемблера. URL: http://natalia.appmat.ru/c%26c%2B%2B/assembler.html
8. Связь ассемблера с языками высокого уровня. URL: https://prog-cpp.ru/asm-c/

ОГЛАВЛЕНИЕ

Учебное издание

Г а р к у ш а Олег Васильевич
Д о б р о в о л ь с к а я Наталья Юрьевна

АССЕМБЛЕР В ПРИМЕРАХ И ЗАДАЧАХ

Учебное пособие

Подписано в печать 17.03.2022. Выход в свет 25.03.2022.
Печать цифровая. Формат 60×84 $^{1}/_{16}$. Уч.-изд. л. 8,4.
Тираж 500 экз. Заказ № 4819.

Кубанский государственный университет
350040, г. Краснодар, ул. Ставропольская, 149.

Издательско-полиграфический центр
Кубанского государственного университета
350040, г. Краснодар, ул. Ставропольская, 149.