Введение

Текст в этой книге является записью, возможно несколькими, конспектов, по мотивам лекций по архитектуре ЭВМ и языку Ассемблер читаемым на факультете ВМК в 2025 году на третьем потоке.

Ассемблеры

Давайте немного поговорим про Ассемблеры. Существует несколько видов Ассемблеров:

1. От производителей оборудования
2. Коммерческие (Например Masm от Microsoft)
3. Со свободным исходным кодом (Например nasm, yasm, gnu assembler. Отдельно отметим, что nasm разработан только под архитектуру Intel)

Виды языков программирования

Для лучшего понимая Ассемблера нам стоит поговорить про языки программирования.

Существуют языки компилируемые и интерпретируемые. К компилируемым относятся, например: Rust, Pascal, C++, которые взаимодействуют напрямую с процессором и Java, который компилируется в байт-код для виртуальной машины.

К интерпретируемым языка относятся, например, Python, Ruby, JavaScript. Разница между компилируемым и интерпретируемым языком в том, что код написанный интерпретируемым языком исполняется самим интерпретатором построчно, а код на компилируемом языку переводится в машинный код, а уже потом исполняется процессором.

Как всё работает

Файл на компилируемом языке (.c/.cpp/.pas) → компилятор → ассемблерный файл (.asm/.s (gnu assembler)) → ассемблер → объектный файл (.obj/.o) → линковка (в неё входят библиотеки (.lib/.a)) → исполняемый файл (.exe)

Также, важно узнать, что существуют динамические и статические библиотеки. Динамические типы (.so - linux/.dll - windows) подгружаются когда программа уже запущена. Эти библиотеки входят в этап 'линковки', и если тип библиотеки не указан, то приоритет будет отдан динамической библиотеке.

Команда для линковки 'link' или 'ld', а для работы ассемблера 'as'.

Работа периферийных устройств

Устройство с процессором (периферия) ↔ Процессор + оперативка (ПК) На периферии установлена программа firmware, которая нужна для управления устройством. На основном устройстве для управления устанавливается драйвер.

Также, важно знать, что существует микрокод, который 'пихается' в процессор, и который расширяет набор команд процессора, что особо важно для архитектуры Intel. Для расширения набора команд архитектуры Intel, используются CISC-системы с переменной длиной кода.

Глава 1

Представление чисел в машине

Представление чисел

Любое число хранится в ячейке фиксированной битности (зачастую степень двойки), например 8, 16, 32, 64 бита.

Существует несколько типов числовых данных: Целые без знака, целые со знаком, числа с плавающей точкой.

Целочисленный тип данных

Старший разряд							Младший разряд

Проблема переполнения

	1	1	1	1
	+			
	0	0	1	0
	=			
1	0	0	0	1

Флаги Существует несколько флагов.
CF - carry flag (флаг переноса) применяется при переполнении беззнакового типа.
OF - overflow flag (флаг переполнения) применяется при переполнении знакового типа.
SF - sign flag (знаковый флаг) выставляется при получении отрицательного ответа.
ZF - zero flag (нулевой флаг) выставляется при получении нулевого ответа.

Как хранить отрицательные числа? Ответ: Старший разряд начинает отвечать за знак (отрицательное при 1, положительное при 0). Он "превращается"в -2^m, где m это размер ячейки минус 1, остальные же разряды считаются как и раньше.

0	0	0	0

Это 0 (+0 для чисел с плавающей точкой)

1	0	0	0

Это -8 (-0 для чисел с плавающей точкой)

Как из положительного числа сделать отрицание? Ответ: Надо получить дополнительный код. Для этого нам понадобится три шага.

1. Взять число в прямом коде (беззнаковое число)
2. Инвертируем каждый бит числа (получаем обратный код)
3. Прибавляем 1 (получаем дополнительный код)

Пример:

0	0	0	0	0	0	0	1

Берём число в прямом коде.

1	1	1	1	1	1	1	0

Инвертируем каждый бит.

1	1	1	1	1	1	1	1

Прибавляем 1.

В итоге из 1 получили -1

Проблема метода:

1	0	0	0	0	0	0	0

При попытке получения доп.кода для максимального отрицательного числа мы получим его же.

Чтобы справиться с этим существует набор инструкций 'neg', который говорит компьютеру что делать при инвертации.

Изучение интервалов Возьмём 4-х битную ячейку памяти. Вычислим интервалы, которые она покрывает.
Для знакового типа ячейка принимает значения: $-2^3 \leq x \leq (2^3) - 1$
Для беззнакового типа: $0 \leq x \leq (2^4) - 1$

Когда же будут выставлены флаги?
OF: $2^3 \leq x \leq 2^4$
CF: $2^4 < x$

Пример знакового переполнения для флага OF:

0	x	x	x	x	x	x	x
+							
0	y	y	y	y	y	y	y
=							
1	z	z	z	z	z	z	z

ИЛИ

1	x	x	x	x	x	x	x
+							
1	y	y	y	y	y	y	y
=							
0	z	z	z	z	z	z	z

Способы хранения чисел

Существует три способа хранения данных:

1. Little endian
2. Big endian
3. Combined endian

Будем рассматривать все способы на примере 4-х байтной ячейки (32-х битной)

AA	BB	CC	EF

Little endian

Отметим, что лектор не согласен с данным определением, так что это little endian по версии сайта wiki.os-dev.org

EF	CC	BB	AA

Big endian

AA	BB	CC	EF

Combined endian

Актуален для чисел с плавающей точкой
Было:

Знак	Порядок	Мантисса

Стало:

Кусок мантиссы	Знак	Порядок	Кусок мантиссы

Как получить отрицательное число

Примем за отрицательное число x, а оригинальное за y. Тогда x + y = 2^n и x = 2^n - y

Случаи при переводе из знакового в беззнаковый тип и наоборот:
При переводе в беззнаковый тип:

1. x (знаковый) $\geq 0 \rightarrow$ x (знаковое)
2. x (знаковый) $< 0 \rightarrow$ x (знаковое) + 2^n

При переводе в знаковый тип:

1. x (беззнаковый) $< 2^{n-1} \rightarrow$ x (беззнаковое)
2. x (знаковый) $\geq 2^{n-1} \rightarrow$ x (знаковое) - 2^n

Операции с числами

В примере будет использована 4-х битная ячейка. Возьмём -5:

1	0	1	1

И вычтем из него -7:

1	0	0	1

Будем поразрядно вычитать (в столбик) и получим 2:

0	0	1	0

Сколько же памяти надо зарезервировать?
Ответ: 4 бита для флагов, 4 бита для частного, 4 для уменьшаемого, 4 для вычитаемого, 1 байт на случай, если уменьшаемое окажется меньше вычитаемого. Всего 17 бит.

Второй вариант предполагает замену вычитания на сложение с инвертацией вычитаемого (заменим -5 - (-7) на -5 + 7)

Возьмём -5:

1	0	1	1

Получим дополнительный код для -7:

0	1	1	1

Сложим поразрядно (в столбик):

0	0	1	0

При этом у нас за старший бит вылезла '1', выполнится CF.

Пример 2:
Сделаем 0 - 1:

Возьмём 0, в первой ячейке будет мнимый разряд:

1 hline	0	0	0	0

Возьмём 1:

0	0	0	1

Будем вычитать поразрядно, придётся занять 1 из мнимого разряда

Получим -1, при этом так как мы заняли 1 из мнимого разряда, то выставиться CF:

1	1	1	1

Числа с плавающей точкой

Мы будем рассматривать стандарт IEEE-754

Любое число в данном стандарте представлено как $-1^s * 2^p * m$

1 бит под знак (s)	p (порядок) бит	m (мантисса)

Типы с плавающей точкой

- half precision - число половинной точности, на всё отводится 16 бит, под порядок 5 бит, под мантиссу 10
- single - 32 бита всего, на порядок 8 бит, 23 под мантиссу
- double - 64 бита всего, на порядок 11, 52 под мантиссу
- quad (quadruple) - 128 бит всего, на порядок 15, 112 под мантиссу

Скорее всего пользоваться вы будете single и double.

В процессоре intel 8087 реализован дополнительный тип, сейчас инструкции этого процессора есть везде. extended - 80 бит всего, на порядок 15 бит, 64 бита под мантиссу.
Из данного числа можно превратить число в single или double.

В поле порядка (p) записывается pmodif (p modified)= P(порядок) + bias(смещение)
bias $= 2^{p-1} - 1$ где p - количество бит выделенное под ячейку порядка.

Мантисса

Мантисса бывает двух типов: нормализованная и денормализованная.

Нормализованная: бит кодирующий 1, Mmodif (M modified)
Mmodif = M без старшего бита для единички

Денормализованная: бит кодирующий 0, Mmodif Mmodif = M без старшего бита для 0

Некоторые важные кодировки

min кодируется:

0/1	00000000	0000000...

max кодируется:

0	11111110	11111111...

+inf кодируется:

0	11111111	00000...

-inf кодируется:

1	11111111	00000...

Существуют NaN, которые отвечают за некоторые ошибки (например умножили очень большое число на очень маленькое)

sNaN - громкий NaN, который вызывает прерывание в процессоре кодируется:

0/1	11111111	xxxx...0

x = 0/1

qNaN - тихий NaN кодируется:

0/1	11111111	xxxx...1

x = 0/1

При операции с NaN будет получен NaN (Заразно)

Операции с числами с плавающей точкой

Чтобы выполнить операцию сделаем выравнивание порядков (сдвинем мантиссу меньшего слагаемого вправо, чтобы оба порядка равнялись большему из них)

За чем нужно следить?
Операции производить с сопоставимыми порядками (Иначе низкая точность)
Чисел с плавающей точкой около 0 больше чем около максимальных значений

Округление
Выполняется “банковое” округление (к ближайшему чётному)

Глава 2

Принципы Фон Неймана организации вычислительных систем

Принципы Фон Неймана организации вычислительных систем

Процессор (с точки зрения Фон Неймана) - нечто, которое состоит из двух частей:

- Арифметико-логическое устройство (Их может быть больше 1)
- Устройство управления

Данные части связаны информационно, процессор с одной стороны имеет ввод и вывод, а с другой есть оперативная память. Оперативная память разбивается на ячейки, понимание этого очень пригодится при работе с Учебными Машинами.

Сами принципы:

- Принцип двоичного кодирования - Всё что хранится в ячейке оперативной памяти записано в двоичном виде.
- Принцип адресности - Все ячейки ОП имеют фиксированный размер, который называется машинным словом, также все ячейки пронумерованы, номер ячейки является её адресом.
- Принцип программного управления - Машина управляется некоторыми специальными командами (инструкциями), которые выполняются последовательно.
- Принцип последовательного исполнения - Команды исполняются последовательно, изменение порядка возможно с помощью инструкций перехода, есть инструкции перехода условные и безусловные.
- Принцип однородности - Команды и данные лежат в одной и той же памяти, они неразличимы между собой, данные интерпретируются как команда, если до неё дошёл исполнитель.

Существуют процессоры различной архитектуры, мы будем рассматривать CISC и RISC.

CISC (англ. Complex Instruction Set Computing) — концепция проектирования процессоров, которая характеризуется следующим набором свойств:

- Большим числом различных по формату и длине команд
- Введением большого числа различных режимов адресации
- Обладает сложной кодировкой инструкции

Процессору с архитектурой CISC приходится иметь дело с более сложными инструкциями неодинаковой длины. Выполнение одиночной CISC-инструкции может происходить быстрее, однако обрабатывать несколько таких инструкций параллельно сложнее.

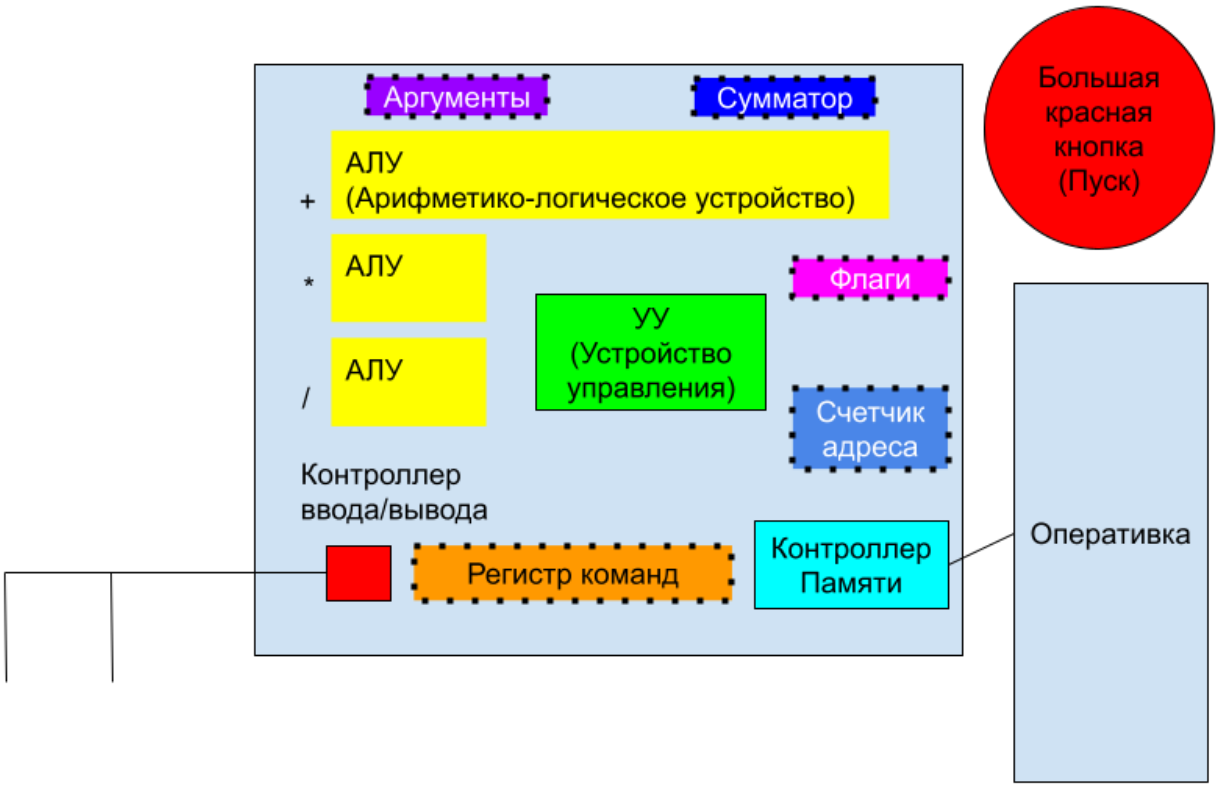
RISC (Reduced Instruction Set Computing). Процессор с сокращенным набором команд. Система команд имеет упрощенный вид. Все команды одинакового формата с простой кодировкой. Обращение к памяти происходит посредством команд загрузки и записи, остальные команды типа регистр-регистр. Команда, поступающая в CPU, уже разделена по полям и не требует дополнительной дешифрации.

Поскольку RISC-инструкции просты, для их выполнения нужно меньше логических элементов, что в конечном итоге снижает стоимость процессора. Но большая часть программного обеспечения сегодня написана и откомпилирована специально для CISC-процессоров фирмы Intel. Для использования архитектуры RISC нынешние программы должны быть перекомпилированы, а иногда и переписаны заново.

Глава 3

Учебные Машины

Вычислительные системы



Краткая сводка о частях вычислительной системы: АЛУ совершает математические операции с данными, УУ дешифрует команды и руководит их исполнением, Контроллер памяти отвечает за получение данных из оперативной памяти, части с чёрными точками по краям это регистры (что это такое вы узнаете чуть дальше) с самым различным назначением.

Историческая справка: в 50-60 годы существовало устройство ввода машинных слов в оперативную память (оно читало перфокарты, перфокарта либо просвечивалась, либо прижимала штекеры).

Регистр - устройство, которое работает с той же скоростью, что и процессор, также имеет определённую битность. Для функционирования вычислительной системы необходимо минимум шесть регистров, но быть их может намного больше.

1. Счётчик адреса
2. Регистр команды
3. Регистр операндов (2 штуки)
4. Регистр сумматор

5. Регистр флагов

Все эти регистры отмечены на рисунке.
Этапы работы вычислительной системы:

1. Предварительный - инициализируем память
2. Считывание команды из ОП по адресу на который указывает счётчик адреса (записывается в счётчик команд)
3. Декодирование команды (Надо понять какую АЛУ применить и откуда взять аргументы)
4. Считывание аргументов команды из ОП.
5. АЛУ выполняет операцию из команды. В регистр сумматор помещается результат.
6. Результат из регистра сумматора записывается в ОП.
7. Увеличение счётчика адреса на размер команды.

Учебные Машины

Существует несколько типов Учебных Машин. Далее будет использоваться сокращение УМ - Учебная Машина.

- УМ-3 (Трёхадресная)
- УМ-2 (Двухадресная)
- УМ-1 (Одноадресная)
- УМ-0 (Безадресная)

Адресность ячеек у любой Учебной Машины можно задать как отрезок [0000, FFFF]. Размер команды же совпадает с размером машинного слова.

Что идёт в в ячейку? Код команды, а потом операнды (от 3 до 0).

Размер занимаемых ячеек:

- УМ-3 = 56 бит, 8 под команду и по 16 на операнд.
- УМ-2 = 40 бит.
- УМ-1 = 24 бита.
- УМ-0 = 8 бит.

Если перейти на `http://arch32.cs.msu.su/semestr2/` то можно найти методичку 2-го потока, а также другую методичку Владимира Георгиевича, где будет другая Учебная Машина, в которой есть команды для ввода/вывода.

3.0.1 Пример написания УМ-3

Задача

$$\begin{cases} x > 2 : z := x + 1 \\ x = 2 : z := 2 \\ x < 2 : z := 2 * (x + 2) \end{cases}$$

Положим в память переменные, которые нам точно понадобятся и константы, которые позже мы переложим в начало памяти:
z = ffff
x = fffe
2 = fffd
1 = fffc

Выполним:
00 81 fffe fffd <переход к присваиванию 2>; Если x = 2, то перейдём к ячейке, где присвоим z 2, 81 - сравнение по "равно"
01 93 fffe fffd <Перейдём к случаю три>; Если x < 2, то перейдём к ячейке, где к x прибавим 2, 93 - беззнаковое сравнение по "меньше".
02 01 fffe fffc ffff; z := x + 1, 01 - сложение.
03 99 0000 0000 0000; 99 - STOP, договоримся, что если это обычный стоп, то будем вместо операндов писать все нули, иначе будем писать код ошибки.
04 00 fffd 0000 ffff; z := 2. Присваивание 2, Поэтому в инструкции 00 на месте третьего операнда напишем [0004]. 00 - передадим первый операнд в третий
05 99 0000 0000 0000; 99 - стоп
06 01 fffe fffd ffff; z := x+2. В операцию 01 на место третьего операнда напишем [0006]. 01 - сложение
07 13 ffff fffd ffff; z := (x+2)*2. 13 - беззнаковое умножение
08 99 0000 0000 0000; 99 - стоп

Полная запись будет такая:
00 81 fffe fffd 0004
01 93 fffe fffd 0006
02 01 fffe fffc ffff
03 99 0000 0000 0000
04 00 fffd 0000 ffff
05 99 0000 0000 0000
06 01 fffe fffd ffff
07 13 ffff fffd ffff
08 99 0000 0000 0000

Перенесём константы ближе к началу памяти. Получим:
09 00 0000 0000 0001 0a 00 0000 0000 0002

Придётся заменить все номера констант в таблице
00 81 fffe 000a 0004
01 93 fffe 000a 0006
02 01 fffe 0009 ffff
03 99 0000 0000 0000
04 00 000a 0000 ffff

05 99 0000 0000 0000
06 01 fffe 000a ffff
07 13 ffff 000a ffff
08 99 0000 0000 0000
09 00 0000 0000 0001
0a 00 0000 0000 0002
Конец

Глава 4

Выполнение операций на АЛУ

Разница между операциями с целочисленным типом данных и с типом данных с плавающей точкой

Разница между операциями с целочисленным типом данных и с типом данных с плавающей точкой колоссальная. Если с целочисленным типом данных всё просто (операция просто выполнится), то для чисел с плавающей точкой надо сначала выравнять порядки, потом произвести операцию с большей разрядной сеткой, затем нужно произвести нормализацию результата и наконец округлить результат.

ЗАДАЧА

Дан массив A ссылок на номера ячеек. Требуется просуммировать все ячейки по всем ссылкам в массиве в переменную Z. Массив имеет длину N не более 16 элементов.

Ячейка A - начало массива [FF00]

Решение задачи на языке Pascal:

```
A: array of integer;
for index := 0 to N-1 do
begin
Z := Z + A[index];
end;
```

Переменные:
Z - лежит в ячейке по адресу ffff
N - лежит по адресу fffe

Для удобства записи в комментариях будем обозначать:
[fffe] - доступ в память для ячейки с номером fffe.

Запись данного файла не показывает процесс сочинения программы, использование временных имён, которые мы помещаем в ячейки, в том числе для частей команд. в дальнейшем имена в ячейках превращаются в числа. Здесь представлен уже конечный результат.

По умолчанию временные переменные, и константы, которые понадобятся в ходе программирования будем помещать в конец памяти. Используя старшие адреса памяти в некотором роде в манере "стека который растёт в сторону уменьшения адресов, с той разницей, что пока из такого "стека"мы ничего удалять не будем.

В дальнейшем можно сделать дополнительный шаг преобразования программы: константы переместить ближе к началу памяти сразу же за последней инструкцией программы. Здесь этот шаг проделан небыл.

Использование памяти:
[ffff] - переменная Z
[fffe] - переменная N
[fffd] - константа 16
[fffc] - переменная index
[fffb] - константа 2^{32} – 00 0001 0000 0000 – (Используется для сдвигов)

[fffa] - константа 1

Для удобстваперед командой слева будем записывать номер ячейки в которой она находится. В интересных случаях перед командой будет оставлен большой комментарий.

Собственно сама программа:
00 95 fffe fffd 0002 ; сравниваем N с 16 и по <= перепрыгиваем останов программы с ошибкой.
01 99 0000 0000 0001 ; останов с ошибкой 1 - код ошибки (на доске было в старшем операнде инструкции).
02 02 fffc fffc fffc ; обнуляем index для цикла. [fffc] := [fffc]-[fffc]
03 94 fffc fffe 000c ; проверяем условие цикла. Если index >= N "выпрыгиваем"из цикла на ячейку c.

Здесь вы вычисляем индекс ячейки массива A+index. Результат поместим в ячейку 0. Она к текущему моменту уже не нужна, а для дальнейшей арифметики, с целью сформировать в памяти нужную команду пересылки так удобнее.

Стоит отметить, что в этом месте мы переписываем собственный код программы, что нормально для архитектуры Фон Неймана, но в общем случае будет недоступно в современных процессорах, как павило такое допускается для строго определённых небольших областей памяти. Например таблицы plt `https://lvee.org/be/abstracts/280`. Большие области памяти переписывает ядро операционной системы, по запросу программы через механизм системных вызовов к ядру операционной системы (темы 1-го семестра 2-го курса).

[0000] := A + index

04 01 ff00 fffc 0000

Здесь мы формируем команду пересылки для доступа к элементу массива. Для этого значение A+index, которое лежит в ячейке 0 в текущий момент умножаем на $2^3 2$. смысл этого действия вместо a) получить b) a) 00 0000 0000 A+index b) 00 A+index 0000 0000

Вычисленное значение мы должны положить по адресу следующей команды, которая будет исполнена. Это адрес 6.

Здесь мы опять пользуемся свойством архитектуры Фон Неймана, а именно меняем свой код. Формируем команду, которую будем исполнять далее.

05 13 0000 fffb 0006

Значение в этой ячейке вообще не важно, так как оно будет переписано предыдущей инструкцией. Для визуального выделения данного места просто заполним всё единичными битами. Можно наверно было заполнить нулями, но это не так "бросается в глаза".

06 ff ffff ffff ffff

Повторяем приём для формирования команды. Он необходим, так как A[index] на самом

деле является ссылкой на элемент.

Поэтому теперь в результате исполнения сформированной команды в ячейке 0 значение по ссылке (указателю). [0000] := A[index$\hat{]}$

Замечание: Операция двойного разименования указателя на самом деле в программах происходит достаточно часто при передаче адресов в функции по var.

07 13 0000 fffb 0008

08 ff ffff ffff ffff ; Ячейка будет переписана для реализации [0000] := A[index$\hat{]}$

Прибавляем к Z содержимое ячейки до которой добрались с помощью предыдущей инструкции.

Z := Z + [0000]

09 01 ffff 0000 ffff

0A 01 fffc fffa fffc ; увеличиваем счётчик цикла на единицу index := index + 1
0B 80 0000 0000 0003 ; Переходим на проверку условия цикла. (проверка в ячейке 3)
0C 99 0000 0000 0000 ; успешное завершение программы с кодом ошибки 0.

Глава 5

Учебные Машины с одним и двумя операндами

Принцип работы машин с одним и двумя операндами

Давайте посмотрим на список команд для УМ-3 (Учебная машина трёхоперандная)

Система команд:

Название	КОП	Операция	Примечание
останов	99	стоп	A1, A2, A3 - любые
пересылка	00	A3:= A1	A2 - любой
		арифметические	
сложение	01	A3:= A1+A2	
вычитание	02	A3:= A1-A2	
умножение			
со знаком	03	A3:= A1*A2	
без знака	13		
деление			
со знаком	04	A3:= A1 div A2, A3+1:= A1 mod A2	
без знака	14		
		переходы	
безусловный	80	перейти к A3	A1, A2 – любые
по =	81	при A1 = A2 перейти к A3	
по ≠	82	при A1 ≠ A2 перейти к A3	
по <	с/зн - 83 б/зн - 93	при A1 < A2 перейти к A3	
по ≥	с/зн - 84 б/зн - 94	при A1 ≥ A2 перейти к A3	
по ≤	с/зн - 85 б/зн - 95	при A1 ≤ A2 перейти к A3	
по >	с/зн - 86 б/зн - 96	при A1 > A2 перейти к A3	

Данная таблица используется и для УМ-2 и для УМ-1, но для обоих машин добавится несколько команд и изменится количество операндов.

Например, так выглядит система для двухоперандной учебной машины.

Код операции	Операнд 1	Операнд 2

Важно помнить, что в двухоперандной учебной машине результат записывается не в третий операнд, а в первый.
Новые команды: 05. Она принимает два операнда, а в следующей строке надо написать по какому принципу эти операнды будут сравниваться, после чего на месте второго операнда написать куда перейти.

Пример:
05 000A 000B

Система для однооперандной учебной машины:

Код операции	Операнд 2

Результат записывается в регистр сумматор, который всегда и выступает как первый операнд.
Новые команды: 20, 10 и 00. 00 передаёт значение регистра сумматора в операнд, а 10, напротив, передаёт значение регистра сумматора в операнд, про 20 поговорим далее.

Также, отметим, что результат операции деления занимает две ячейки памяти. Div передаётся в указанную ячейку, а mod в следующую. (В УМ-1 для этого появляется отдельный регистр, будем называть регистр сумматор S, а дополнительный - S1, команда 20 меняет местами значение S и S1)

Решим задачу.

Если переменная A делится на 5 и на 3 с остатком, то в переменной Z просуммируем остатки. Иначе запишем -1 в Z.

Решение на языке Pascal:

```
if a mod 5 and a mod 3 then
z := a mod 5 + a mod 3
else
z := -1;
```

Запишем переменные:
z = ffff
a = fffe

00 00 fffe; Передадим в S значение переменной a.
01 14 <5> (0011); Поделим S на 5, в скобках написано будущий адрес переменной, В процессе написания команды он нам не известен.
02 20 0000; Поменяем местами значения S и S1.
Теперь в S остаток от деления на 5.
03 10 ffff; Передадим остаток в Z
04 05 <0> (0013); Будем сравнивать S и 0
05 81 <else> (000e); Если остаток от деления a на 5 = 0, то уйдём на ветку else.
06 00 fffe; Передадим в S значение переменной a.
07 14 <3> (0012); Поделим S на 3.
08 20 0000; Поменяем местами S и S1.
09 05 <0> (0013); Аналогично с 04.
0a 81 <else> (000e); Аналогично с 05.
0b 01 ffff; Сложим остатки.
0c 10 ffff; Передадим в z результат сложения.
0d 99 00; Конец.
0e 00 <-1> (0014); Ветка else. Передадим -1 в S.
0f 10 ffff; Передадим -1 в Z.
10 99 0000; Конец.
11 00 0005; Константа 5.
12 00 0003; Константа 3.
13 00 0000; Константа 0.
14 ff ffff; Константа -1.

КОНЕЦ.

Глава 6

Assembler

MASM

Мы будем пользоваться MASM 6.14 (довольно древним).

Важно знать, что существуют исполняемые и объектные файлы.

Исполняемый файл - файл, содержащий программу в виде инструкций, которые могут быть исполнены компьютером. Исполняемым файлам противопоставляются объектные файлы - файлы, которые читаются и парсятся определённой программой, а сами по себе не могут быть исполнены.

На MASM пишутся именно объектные файлы.

Секции объектного файла:
1) Заголовок (может включать в себя переменные, на случай взаимодействия с другим исполняемыми файлами)
2) .data (инициализированные переменные)
3) .data? (Неинициализированные переменные)
4) .code
5) .stack (известен только размер, сюда складываются локальные переменные от функций)

Секция .data может быть разделена другими секциями, например:

```
.data
...
.code
...
.data
```

Также, в секции .data могут быть как инициализированные переменные, так и не инициализированные. А в .data? только неинициализированные.

Тип "переменной"в объектном файле:
1) Константа
2) Функция
3) Переменная
4) Локальная переменная
5) Локальная функция

Отличие локальной "переменной"от не локальной в том, что при "склейке"объектных файлов локальные "переменные"могут быть не уникальны, а обычные ТОЛЬКО уникальны.

В Assembler есть очень важное правило:
Одна инструкция - одна строчка.

Виды записи данных:
1) Десятиричная запись - 99
2) Шестнадцатиричная запись - 0aah, где h указывает на шестнадцатиричность, а перед буквами должен стоять 0
3) Восьмиричная запись - 44q, где q указывает на восьмиричность
4) Двоичная запись - 011101b, где b указывает на двоичность

Резервация памяти

Для резервации памяти стоит указать резервируемый размер. Вот некоторые типы:

db - 1 bait
dw - 2 bait
dd - 4 bait
df - 6 bait
dq - 8 bait
dy - 10 bait
owrd - 16 bait

real4 - 4 байта под число с плавающей точкой
real8 - 8 байт под число с плавающей точкой
real10 - 10 байт под число с плавающей точкой

Несколько интересных моментов:
mstr db "MAMA 0 - кодировка русских символов по умолчанию cp1251, тогда размер выделенной памяти будет 4 байта (1 байт под каждый символ), но все используют utf8, тогда проблематично узнать размер строки.
Помимо этого, при резервации памяти под строку в конце надо писать ноль, запомните это.

Конструкция "name db/dd/... x dup(y)"позволяет выделить x раз память под y с типом db, dd и пр.

Пример:
var3 dd 1024 dup(-8) - Число -8 повторить 1024 раза

Конструкции MASM:
Ключевые слова type and size

Для примера: var2 dd 10
type var2 вернет dd, size var2 вернет 4.
type 12 вернет const
size 12 вернет ошибку
type var3 вернет dd (см. var3 выше)
size var3 вернет 1024*4

varinteger db oAAh, 0BBh, 0CCh, 0DDh. (Более старший разряд лежит на большем адресе, начинается чтение разрядов справа налево, т.е. DDCCBBAA)

Именовать можно:
Переменные
Функции
Метки
Макропеременные

В качестве имён нельзя использовать названия секций. Имя может начинаться с точки и иметь в составе @, ?, $

Собаку и знак доллара надо использовать аккуратно, из-за того что существуют анонимные переменные. А все метки начинающиеся с собаки, являются локальными.

6.0.1 Инструкции

Внутри секции .code можно задать метку start:, но рано или поздно она должна кончится, как end start.

Существуют разные списки имён
local: список имён, объявление локальных переменных
public: список имён
extern: список имён с модификаторами (сюда передаётся тип Директивы type (byte, word, dword, qword, near, for))

В самом начале файла указываем для какого процессора мы используем инструкции (например .686p,xmn)
.model - директива указываем модель памяти (актуально для 32-х битной архитектуры, мы используем flat - все адреса 32-х битные)

6.0.2 Описание команд

Архитектура интел является архитектурой с переменной адресностью.
Что есть у команды:
Префикс, нас будет интересовать префикс lock, который выставляет блокировку на память, если процессор в многопроцессорном/многоядерном режиме, то нельзя будет менять ячейки памяти.
Код операции (имя инструкции), например mov, add
Типы операндов, инструкции могут накладывать ограничения
Операнды
Типы операндов - константа(imm), регисты(r), адрес памяти (m), косвенная адресация (в masm выглядит как обращение к элементу массива)

Какие операции запрещены:
Операции память - память Запись в константу, например mov 12, eax
mov eax, offset, varinteger - с помощью offset из имени делаем константу

6.0.3 Регистры i386 архитектуры

Регистры общего назначения - 8шт. 32-х битные
eax - аккумулятор, собирает результаты арифмитических операций
ebx - база, организует точки отсчёта
ecx - счётчик, счётчик в циклах
edx - расширение аккумулятора
esi - указатель источник (откуда)
edi - указатель пункта назначения(куда)

НЕ РЕКОМЕНДУЕТСЯ МЕНЯТЬ
esp - указатель на вершину стека
ebp - указатель на начало стекового кадра функции

Дополнительные:
eflags - регистр флагов
eip - счётчик адреса (не виден программисту)

К регистрам общего назначения серии e*x можно обращаться по частям (16 и 8 бит). Пример на eax - с 16 по 8 бит AH, с 8 по 0 бит AL, с 16 по 0 AX

К регистрам серии e*i, e*p можно обращаться только пополам, тогда пропадёт буква e из серии.

Служебные регистры (НЕ МЕНЯТЬ)- 16 бит:
cs, ss, ds, es, gs, fs

Существуют также контрольные и дебаг регистры, но нам про них знать не обязательно. Также отметим, что за локальные сегменты отвечает регистр ldtr, а за глобальные gdtr.

Минимальный набор инструкций для пользования masm - add - сложение, sub - вычитание, int (позволяет увеличить значение регистра на 1), dec (уменьшить значение регистра на 1), cmp (выставляет флаги), mov (присваивание), jmp (безусловный переход), jz (прыжок если флаг z выставлен), jnz (прыжок если z флаг не выставлен), je, jne (тоже самое, но используется для обозначения эквивалентности), ja (больше без знака), jae(больше или равно без знака), jb (меньше без знака), jbe (меньше или равно без знака), jg (больше знаковое), jge (больше или равно знаковое), jl (меньше знаковое), jle (меньше или равно знаковое), jo, jc, js (Прыжок если OF, CF, SF флаги выставлены), jno, jnc, jns (Прыжок если OF, CF, SF флаги не выставлены).

6.0.4 Дополнительно Что можно записать в .data

1) Константы
N = 5
M EQU 5
Также, можно не задавать имя переменной, а переходить к ней по метке предыдущей константы со смещением.

2) Переменная
a размер (db, dw, dd, dq) значение

3) Массив
X dw N dup (0)
Y dw M dup (1, 2, 3)

4) Строки
str db 'Hello', 0

Примеры:
A db 'A', 'B', 'C' → ABC
B dd 'ABC' → CBA0
C dd ABCh → BCh 0Ah 00h 00h

Глава 7

Простейшие программы

Адресация

Переходы

Существует множество видов переходов между блоками инструкций, но можно разделить их на две части.
jmp - безусловный переход и jcc (тут c - некая буква) - условный переход. Вот некоторые типы:

Переходы по флагам
js/jns - переход когда SF выставлен/нет
jo/jno - переход когда OF выставлен/нет
jc/jnc - переход когда CF выставлен/нет
jz/jnz - переход когда ZF выставлен/нет

jb/jl - переход по условию "меньше"(беззнак/знак)
jbe/jle - переход по условию "меньше или равно"(беззнак/знак)
ja/jg - переход по условию "больше"(беззнак/знак)
jae/jge - переход по условию "больше или равно"(беззнак/знак)

ВНИМАНИЕ! СЛЕДУЮЩИЕ КОМАНДЫ ПОЗВОЛЯЮТ ПЕРЕЙТИ НА МЕТКУ, ОТСТАЮЩУЮ НЕ БОЛЕЕ ЧЕМ НА 128 БАЙТОВ ОТ КОМАНДЫ ПЕРЕХОДА!

jecxz - переход, если ecx = 0
jcxz - переход, если cx = 0
loop - переход на метку, пока ecx $\neq$ 0, при этом ecx = ecx-1.

Абсолютный и относительный адрес

Существует абсолютный адрес и относительный.

Для фиксации адреса существует специальный регист - eip.

Относительный переход можно описать так: eip = eip+offset (offset - разница между двумя точками в коде (offset = метка - текущий адрес))

Некоторые полезные команды и макросы

INC - прибавляет 1 к операнду, лучше add тем, что не трогает CF, выполняется быстрее и является одноопернадной.

Макрос exit - завершает выполнение программы, иначе она продолжит выполнять команды процессора.

Макрос outstr - выводит строку, пока не встретит 0. Вызывать этот макрос несколько раз с одной и той же строкой плохо, потому что память будет выделяться для одного и того же несколько раз.

После печати строки пишем 0, иначе будет погром памяти (когда строку мы задавали заранее, а не передали как параметр макроса)

Макрос inchar - чтение одного символа. Операнд op1 может иметь формат r8 или m8. Код (номер в алфавите) введённого символа записывается в место памяти, определяемое операндом op1.

Макрос ReadKey - Выдаёт код (номер в алфавите) введённого символа. Код символа возвращается в регистре AL.

Макрос Inint - ввод целого числа. Выставляет CF=0 при правильном вводе или CF=1 при неправильном. Для правильного числа также устанавливается флаг ZF:=1, если лексема введённого числа начиналась со знака "– иначе ZF:=0.

Макрос new - Макрокоманда порождает динамическую переменную размером size (это беззнаковый операнд формата i32) и возвращает адрес этой переменной в регистре EAX.

Пример

Напишем программу получающую на вход число в двоичном виде, а выводящую в десятичном.

```
include console.inc

.data
number dd 0

.code
mesg db 'Please input number in binary base system',0 (сообщение приглашающее к вводу)

begin:
outstrln offset mesg (вывод сообщения)

loopread:
inchar bl (ввод символа в младшую часть регистра ebx)

cmp bl, '0'
jne checkone
mov eax, number
add number, eax ; number := number*2
jmp loopread

checkone:
cmp bl, '1'
jne printresult
mov eax, number
inc eax
```

```
add number, eax ; number := number*2+1

jmp loopread

printresult:

outstr "Collected number is: "
outwordln number
exit 0

end begin
```

Глава 8

Режимы адресации в аргументах команд

Режимы адресации в аргументах команд

Существует несколько типов адресации команд:

1. imm константы - При таком способе адресации поле адреса команды содержит, как говорят, непосредственный (immediate) операнд. Разумеется, такие непосредственные операнды могут быть только (неотрицательными) целыми числами, по величине не превышающими максимального значения, которое можно записать в поле адреса.
2. Регистр (r) - Подставляется значение регистра, к которому у нашей программы есть доступ.
3. Адрес в памяти (m) - При этом способе адресации число на месте операнда задаёт адрес ячейки основной памяти, в котором и содержится необходимый в команде операнд.
4. Косвенная адресация по одному регистру - сначала будет получено значение регистра, а потом будет осуществлён переход по полученному значению.
   [базовый регистр+множитель*регистр-счётчик +const] - конструкция для косвенной адресации, const может быть отрицательной, множитель нет. Регистр счётчик - любой кроме esp.
   Множитель может быть - пусто (1), 1, 2, 4, 8.
   Косвенная адресация работает для близких адресов (near, 32-х битные), архитектура intel позволяет пользоваться и более длинными, задействуя сегментные регистры.

   Сегментные регистры - 16-битные, их 6 штук: cs - code, ds - data, ss - stek, gs - не используется, fs - не используется, es - дополнительный регистр, который может хранить адрес любого сегмента.

   Есть таблицы ldt и gdt, в gdt заводить записи может только ядро операционной системы, в ldt может и программа. Таблицы задаются регистрами ldtr и gdtr (в них хранятся адреса сегментов в ОПЕРАТИВНОЙ, а не виртуальной памяти, также в них хранится размер записи).
5. Косвенная адресация по двум регистрам - похоже на косвенную адресацию с одним регистром.

Регистр eflags

Существует 32-х битный регистр eflags, в котором хранятся значения различных флагов.

Для взаимодействия с ним существует несколько команд:

1. lahf - последние 8-бит eflags записывает в регистр AH
2. sahf - из AH делает младшую часть eflags

Флаги, которые достаёт lahf:

1. CF - Флаг переноса фиксирует значение переноса (заема), возникающего при сложении (вычитании). Иногда используется и в других ситуациях.

2. PF - Флаг четности фиксирует наличие четного числа единиц в младшем байте результата операции, может быть использован, например, для контроля правильности передачи данных.

3. AF - Флаг вспомогательного переноса фиксирует перенос (заем) из младшей тетрады, т.е. из бита 3 в старшую тетраду при сложении (вычитании). Используется только для двоично-десятичной арифметики, которая оперирует исключительно младшими байтами.

4. ZF - Флаг нуля сигнализирует о получении нулевого (ZF = 1) или ненулевого (ZF = 0) результата операции.

5. SF - Флаг знака дублирует значение старшего бита результата, который при использовании дополнительного кода соответствует знаку числа (0 – положительное число, 1 – отрицательное).

6. TF - При установке флага трассировки TF = 1, микропроцессор переходит в пошаговый режим работы, применяемый при отладке программ, когда автоматически генерируется особая ситуация отладки после выполнения каждой команды.

Ещё несколько важных флагов:

1. DF - флаг направления, 0 - перемещение в сторону увеличения адресов, 1 - в сторону уменьшения адресов.

2. ID - выясняет поддерживает ли процессор CPUID.

3. IF - выясняет обрабатывается ли сейчас прерывание.

4. IOPL - задаёт уровень привилегий с которыми работает процессор.

5. CPL - текущий уровень привилегий.

Некоторые интересные команды:

1. clc/cld/cli - clear CF/DF/IF (CF = 0)

2. stc/std/sti - set CF/DF/IF (CF = 1)

3. cmc - инвертировать CF

4. movsx - знаковое расширение op2 в op1

5. movzx - беззнаковое расширение op2 в op1

6. cbw - расширяет al -> ax (знаковый)

7. cwde - расширяет ax -> eax (знаковый)

8. cwd (convert word double) - расширяет ax -> dx:ax (знаковое для деления)

9. cdq eax -> edx:eax (знаковое для деления)

Сегментация памяти

Сегмент вырезает какую-то часть в виртуальной памяти. Этот сегмент рано или поздно должен быть отображен в оперативку. В intel архитектуре за это отвечает механизм страниц. Сегмент вырезается за счёт сегментных регистров (cs, ds, ss, es, fs, gs). Из программы нельзя менять регистр cs.

В операциях, где одним из операндов является адрес в памяти, можно указывать каким сегментным регистром мы хотим для этого воспользоваться. например es:[eax].

В сегментные регистры можно передавать только ax и eax.

far - тоже что jmp, но происходит не по одному, а по паре регистров. Используется, чтобы переключиться с одной программы на другую)

Глава 9

Умножение и деление

Умножение и деление

9.0.1 Умножение

Для умножения есть две инструкции: mul (беззнаковое) и imul (знаковое)

imul бывает нескольких типов:

1. с одним аргументом (одним из операндов идёт eax/ax/al)
2. с двумя аргументами (На курсе не используется)
3. с тремя аргументами (На курсе не используется)

Почему же мы не пользуемся умножением с тремя аргументами?
Потому что в архитектуре i186 нельзя было сделать
lea edx, [ecx +8*eax + 5], поэтому пользовались умножением с тремя аргументами.

Что можно указывать в качестве операнда?

1. Регистр
2. Операнд-адрес в памяти

Константу нельзя использовать, потому что процессор не знает какой размер у константы.

Умножение бывает

1. 8-битное (пришло из 8086). В ax (в два раза больше после умножения, было 8, стало 16) кладём al * op1 (обязан быть 8 битным).
2. 16-битное. В dx:ax (в два раза больше после умножения, было 16, стало 32) клад ём ax * op1 (обязан быть 16 битным)
3. 32-битное. В edx:eax (Было 32, станет 64) кладём eax * op1 (обязан быть 32)

9.0.2 Деление

Для деления как и для умножения есть две инструкции: div (беззнаковое) и idiv (знаковое)

Также бывает три версии:

1. 8-битное. AX делим на op1, в AL получаем частное, а в AH остаток от деления.
2. 16-битное. Пара регистров DX:AX делится на op1, в AX - помещается частное, в DX остаток от деления.
3. 32-битное. Пару регистров EDX:EAX делится на op1, в EAX - помещается частное, в EDX остаток от деления.

Важно помнить, что при делении на 0 сработает прерывание.
Также важно помнить, что если результат не влезает в новый регистр, то также сработает прерывание.

Примеры:
Мы хотим поделить edi на bx, как нам это сделать?
Так как показано дальше сделать не выйдет, потому что при делении на bx задействуются регистры DX:AX, а не EAX.
mov eax, edi
idiv bx (если edi было знаковым числом)

А теперь посмотрим как правильно.
mov eax, edi
cdq (Знаково расширяет EAX до пары EDX:EAX)
movsx ebx, bx (Знаково расширяет bx до ebx)
idiv ebx

9.0.3 Арфимитическое представление умножения умножения 64-х битного числа на 32-х битное

Пусть мы хотим умножить A (64 бита) на B (32 бита) и поместить результат в C (64 бита).
Тогда разобьём A на старшую и младшую часть
Тогда наше уравнение можно представить как:
C (64) = (A(32, младшая часть) + A(32, старшая часть) * 2^{32}) * B (32)
Раскроем скобки
C (64) = A (32, младшая часть) * B (32) + A (32, старшая часть) * B (32) * 2^{32}
Введём D (64) = A (32) * B (32)
Тогда C(64) = D (64 из младшей части) + D (64 из старшей части) * 2^{32}
Разобьём D на старшую и младшую часть
Тогда C (64) = D (32, Младшая из младшей) + D (32, Старшая из младшей) * 2^{32} + D (32, Младшая из старшей) * 2^{32} + D (32, Старшая из старшей) * 2^{64}
Заметим, что последнее слагаемое не влезет в 64 бита, поэтому выкинем его.
В итоге получим C (64) = D (32, Младшая из младшей) + D (32, Старшая из младшей) * 2^{32} + D (32, Младшая из старшей) * 2^{32}

Рассмотрим тоже самое действие, но на языке Assembler

```
include console.inc

    .data
        a dq 80000000h
        b dd ?
        c dq ?

    .code
        mesg db 'Please input number',0

    begin:
        outstrln  offset mesg
```

```
        inint b

        mov eax, dword ptr a; записываем младшую часть a
        mul dword ptr b

        mov ebx, eax ; младшая часть произведения
        mov ecx, edx ; старшая часть произведения

        mov eax, dword ptr a+4 ; записываем старшую часть a
        mul dword ptr b

        mov edi, eax ; младшая часть произведения
        mov esi, edx ; старшая часть произведения  (d^h)

        mov eax, 0
        mov edx, 0

        add eax, ebx
        add edx, ecx
        add edx, edi

        test esi, -1
        jnz not_set_cf
        stc ; выставляет CF

not_set_cf:

        mov dword ptr c,   eax
        mov dword ptr c+4, edx


        outwordln a
        outwordln b
        outwordln c

        exit 0

end begin
```

Глава 10

Побитовые операции и сдвиги

Побитовые операции

Из основных операций стоит знать:

1. OR - два операнда, результат записывается в первый. Логическое "или".
2. AND - два операнда, результат записывается в первый. Логическое "и".
3. XOR - два операнда, результат записывается в первый. Побитовое сложение по модулю 2.
4. NOT - один операнд, производит отрицание каждого бита.
5. TEST - один операнд, аналог CMP, но вместо SUB делает AND.

Побитовые операции действуют на флаги ZF и CF.

Операции сдвигов

Существуют циклические и не циклические сдвиги.
Существую два типа не циклических сдвигов: арифмитические (sal, sar) и беззнаковые (shl, shr). У данных сдвигов первым операндом может быть регистр или память, а вторым константа или регистр cl.

1. Арифметический сдвиг влево (sal) сдвигает все биты влево на значение второго операнда, заполняя освободившиеся биты нулями.
2. Арифметический сдвиг вправо (sar) сдвигает все биты вправо на значение второго операнда, заполняя освободившиеся биты значением старшего бита.
3. Беззнаковый сдвиг влево (shl) сдвигает все биты влево на значение второго операнда, заполняя освободившиеся биты нулями. При этом байт который был вытолкнут последним попадает в флаг CF.
4. Беззнаковый сдвиг вправо (shr) сдвигает все биты вправо на значение второго операнда, заполняя освободившиеся биты нулями. При этом байт который был вытолкнут последним попадает в флаг CF.

Примеры:

1	0	0	0	1	1	0	1

AL до сдвига

Используем shl AL, 3

0	1	1	0	1	0	0	0

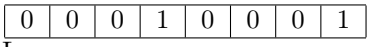
AL после сдвига, при этом в регистре CF окажется 0.

Другой пример:

1	0	0	0	1	1	0	1

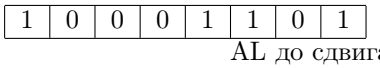
AL до сдвига

Используем shr AL, 3



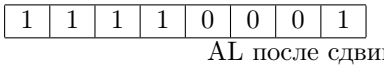
AL после сдвига, при этом в регистре CF окажется 1.

Третий пример:



AL до сдвига

SAR AL, 3



AL после сдвига.

Существую два типа циклических сдвигов: "локальные"(rol, ror) и "флаговые"(rcl, rcr). У данных сдвигов также первым операндом может быть регистр или память, а вторым константа или регистр cl.

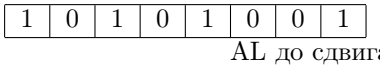
1. Локальный сдвиг влево (rol) сдвигает все биты влево на значение второго операнда, только что вытолкнутый бит попадает на свободное место. Также вытолкнутый бит дублируется в CF.

2. Локальный сдвиг вправо (ror) сдвигает все биты вправо на значение второго операнда, только что вытолкнутый бит попадает на свободное место. Также вытолкнутый бит дублируется в CF.

3. Флаговый сдвиг влево (rcl) сдвигает все биты влево на значение второго операнда, освободившееся место заполняется значением CF, а CF заполняется вытолкнутым битом.

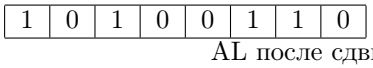
4. Флаговый сдвиг вправо (rcr) сдвигает все биты вправо на значение второго операнда, освободившееся место заполняется значением CF, а CF заполняется вытолкнутым битом.

Пример:



AL до сдвига

Сделаем ROL AL, 2



AL после сдвига

Рассмотрим несколько примеров:

```
comment *
    Программа предназначена для тестирования операций сдвигов.
    В начале программы считывается число со стандартного потока ввода,
    затем с данным числом производится побитовая операция сдвига,
    затем оно побитово распечатывается начиная со значения флага CF.

    C -  CF выставлен
    # -  CF не выставлен.
*

include console.inc

NEW_LINE equ 10

    .data?
        number dd ?

    .code
        mesg db 'Please enter 32-bit signed number',0

    Start:
        OUTSTRLN  offset mesg

        ININT number

        ror number, 3

        mov edx, 'C'
        mov eax, '#'
        cmovc eax, edx ; mov если CF = 1
        OUTCHAR al


        mov ebx, number

        mov edx, '1'
        mov ecx, 32
    loop_bits:
        test ecx, 8-1 ; low 3 bits
        jnz check_bit
        OUTCHAR ' '

    check_bit:
        shl ebx, 1
        mov eax, '0'
        cmovc eax, edx
        OUTCHAR al
        dec ecx
        jne loop_bits
```

```
        ;loop loop_bits

        mov al, NEW_LINE
        OUTCHAR al

        EXIT 0
    end Start
```

Второй пример:

```
comment *
    Со стандартного потока ввода вводится x32- битное число.
    Далее это число распечатывается в своём битвом представлении,
    каждый байт отделяется пробелом.
*

include console.inc

;
; Так как стандартная esc последовательность '\n'
; не работает приходится заводить синоним для константы
; 10, которая означает перевод строки.
;
NEW_LINE equ 10

    .data?
        number dd ?

    .code
        mesg db 'Please␣enter␣32-bit␣signed␣number',0

Start:
        OUTSTRLN  offset mesg

        ININT number

        mov ebx, number

        mov edx, '1'

        ;
        ; организуем цикл смещений, по числу бит в машинном
        ; представлении числа.
        ;
        mov ecx, 32
    loop_bits:
        ;
        ; проверяем, при помощи test (and с выкидыванием результата)
        ; что последние 3 бита равны 0. Это означает, что число делится на 8.
        ;
```

```
        ; маска будет выглядеть старшие биты 0, последние 3 бита единички.
        ;
        test ecx, 8-1 ; low 3 bits
        jnz check_bit
        OUTCHAR '_'

    check_bit:
        shl ebx, 1
        mov eax, '0'
        ;
        ; присваивание сработает если флаг CF выставлен. если нет,
        ; то эффект как будто- данную инструкцию просто пропустили.
        ; аналог jc - только не в смысле перехода, а в смысле присваивания.
        ;
        cmovc eax, edx
        OUTCHAR al
        dec ecx
        jne loop_bits
        ;
        ; здесь уже инструкций столько, что
        ; команда процессору loop не" дотягивается"
        ; до метки loop_bits
        ;
        ;loop loop_bits

        mov al, NEW_LINE
        OUTCHAR al

        EXIT 0
end Start
```

Третий пример:

```
;
; печатаем побитово x64- битное число.
;

include console.inc

NEW_LINE equ 10

    ;.data?
    ;    number dq ?

    .data
         number dq -64

    .code
        mesg db 'Please_enter_32-bit_signed_number',0
```

```
Start:
    ;OUTSTRLN  offset mesg

    ;ININT qword ptr number

    ;
    ; формируем старшую и младшую части числа в регистрах
    ; edx b eax соттветственно.
    ;
    mov eax, dword ptr number
    mov edx, dword ptr number+4


    mov esi, '1'
    mov ecx, 64
loop_bits:
    test ecx, 8-1 ; low 3 bits
    jnz check_bit
    OUTCHAR ' '

check_bit:
    shl eax, 1 ; младшую часть сдвинули старший бит уехал в CF
    rcl edx, 1 ; старшую часть сдвинули
               ; и прицепили старший бит из младшей половины числа,
               ; который находится во флаге CF.
               ; При этом разряд из старшей части ещё не сдвинутого числа
               ; после циклического сдвига окажется во флаге CF.

    mov ebx, '0'
    cmovc ebx, esi
    OUTCHAR bl
    dec ecx
    jne loop_bits
    ;loop loop_bits

    mov al, NEW_LINE
    OUTCHAR al

    EXIT 0
end Start
```

Глава 11

Стек

Представление стека

Стек - “стакан”, который находится в виртуальной памяти.
Виртуальная память пронумерована от 0 до $2^{32} - 1$
В начало памяти доступ запрещён, если воспользоваться перемещением на адрес 0 или 0 с маленьким смещением, то можно попасть сюда.
Внизу стека лежат данные операционной системы.
Регистр ESP указывает на вершину стека. Там может лежать число аргументов, аргументы, переменные окружения и данные.
Если уменьшить значение ESP, то это тоже самое что выделить место в стеке (Нельзя пользоваться отрицательными значениями относительно смещения esp)
Если же увеличить значение ESP, то это тоже что освободить место в стеке.

ВАЖНОЕ ЗАМЕЧАНИЕ: НЕЛЬЗЯ ДЕЛАТЬ add [esp-8], 4 - ОШИБКА, потому что над esp может быть прерывание и оно затрется.

Стек используется как механизм передачи параметров в функции или чтобы временно что-то сохранить.
Для этого используются операции push и pop.
push имеет один операнд, который может быть памятью или регистром, 2 и 4 байтов. Если подаётся константа, то она принимается как 4 байта. push Также существуют команды pusha/pushad (16/32 бита) и popa/popad (16/32 бита), которые позволяют положить и достать все регистры общего пользования в стек.
Упомянем похожие команды pushf/pushfd и popf/popfd, которые кладут на стек или достают значения регистра флагов (flags/eflags)

Немного забежим вперёд. Операция call op1 - берет значение из op1 и интерпретирует как начало функции, которую нужно исполнить. При этом значение EIP кладётся на вершину стека
Это позволит нам изучить единственный способ узнать значение регистра EIP. Нужно сделать:
CALL func

func:
mov eax, [esp]
ret (что это вы узнаете чуть дальше)

Распишем call в трех инструкциях:
sub esp, 4
mov [esp], eip (на самом деле [ss:esp], а eip указывает на инструкцию следующую за call)
jmp метка

ret арг (но можно без аргумента, тогда будем считать его нулём) в трёх инструкциях:
move тень (теневой регистр), [esp] (узнаем откуда должна продолжиться программа)
add esp, 4+арг (зачистим стек)
jmp тень (Продолжим исполнение программы)

Что такое стековый кадр? Скажем, что это блок для работы отдельно взятой функции.
Стековый кадр выглядит так:

Локальные переменные
EBP
EIP
Параметры функции

Вы можете задать вопрос, откуда взялся регистр EBP (остальное понятно, локальные переменные передаются во время выполнения функции, EIP кладётся во время call, а параметры функции передаются до вызова call)
EBP мы будем передавать сами, чтобы удобно взаимодействовать со стеком. Как мы будем это делать:
push EBP
mov EBP, ESP (EBP и ESP указывают в одну точку)
Как сохранить на стек 64-битную переменную?
mov [ebp-8], eax
mov [ebp-8+4], edx
Так мы сохранили 64-х битную переменную.

Соглашение о передаче параметров

Важное о соглашениях:

1. Зависит от языка программирования
2. Описывает способ передачи параметров
3. Описывают способ возвращения результата
4. Описывает кто чистит стек от переменных
5. Описывает какие параметры не меняются функцией

Соглашения о передаче параметров:

1. SystemV ABI - Unix-ы
2. Cdecl - Windows C
3. Stdcall - winAPI, FreePascal
4. Syscall - системные вызовы windows
5. Fastcall - linux-версия и windows-версия
6. Pascal - turbopascal
7. Thiscall - C++

Способы передачи параметров:

1. Через регистры

2. Через стек в прямом порядке
3. Через стек в обратном порядке - cdecl и stdcall

Кто чистит стек?
cdecl, systemV ABI - стек чистит тот, кто вызывал функцию
stdcall, pascal - стек чистит функция

Какие регистры не меняются функцией?
Меняются eax, ecx, edx, eflags, остальное сохраняется

Глава 12

Процедуры

Процедура

Чтобы написать функцию надо использовать конструкцию:
Имя-функции proc (тут может быть написано ещё что-то, например public, чтобы функция стала как модуль)
Тело-функции
Имя-функции endp

Напишем пример функции.
Функция на языке Pascal:
function sum-numbs (a, b: integer (16 bit)): long integer (32 bit);
sum-numbs := a+b;

А теперь рассмотрим эту же функцию на языке ассемблер:

```
sum–numbs proc
push ebp
mov ebp, esp ; Стандартная конструкция из предыдущей главы

mov ax, [ebp +8]; Добрались до a, до этого ктото– делал push a
mov dx, [ebp+12]; Добрались до b
movsx eax, ax
movsx edx, dx
add eax, edx
pop ebp
ret 8 общий( размер a и b 8 байт)
sum–numbs endp
```

Рассмотрим стековый кадр для данной функции:

EBP (0-3 байта)
EIP (4-7 байт)
a 8-9
trash 10-11, появился он из-за того как мы пушили
b 12-13
trash 14-15

Как мы вызываем функцию?

```
.data
a dw 3
b dw 4

.code...

mov ax, a
mov bx, b
push ebx ; отсюда и появится trash
push eax
```

```
call sum–numbs
```

По соглашению мы пушили сначала b, потом a, а также в функции можно портить регистры eax, ecx, edx.

Рассмотрим ещё один пример, в нём надо будет положить результат суммы по адресу и вывести был ли выставлен OF при сумме.
Как она задаётся: function sum-abc(a, b: integer, var c: longint): boolean

```
sum–abc proc
push ebp
mov ebp, esp
mov ax, [ebp+8]
mov dx, [ebp+12]
add ax, dx
mov ecx, 0 ; Передаём флаг OF
jno @F ; Переход к следующей анонимной метке
mov ecx, 1
@@ – анонимная метка ассемблер( её превратит в ?? 0001):
mov edx, [ebp+16] ; Передали адрес C
cwde
mov [edx], eax ; Положили значение суммы по адресу C
mov eax, ecx
pop ebp
ret 12
sum–abc endp
```

Рассмотрим стековый кадр для данной функции:

EBP (0-3 байт)
EIP (4-7 байт)
a 8-9
trash 10-11, появился он из-за того как мы пушили
b 12-13
trash 14-15
Адрес c 16-19

Как мы вызываем функцию?

```
.data
a dw 5
b dw 7
c dd ?

.code
push offset c
mov ax, b
push eax
mov ax, a
```

```
push eax
call sum_abc
```

Пример 3, программа и функция распечатки числа в 32-х битном знаковом представлении.
Алгоритм следующий: На стеке функцией будем формировать строчку, для дальнейшей печати. Начнём с нулевого байта который положим сперва на стек, затем будем наращивать цифрами числа от младших разрядов к старшим. Wифра будет вычисляться как остаток от деления на 10. В конце запишем на вершину стека символ минус, если число было отрицательное.

```
include console.inc

    .data
        number dd 1025 ; знаковый

    .code

print_dec proc
        push ebp
        mov ebp, esp

        ;
        ; резервирование памяти под локальные переменные.
        ;
        ; В этой процедуре они не нужны, поэтому здесь пусто.
        ; Иначе было бы:
        ;   sub esp, суммарныйразмерлокальныхпеременных___
        ;
        ; при вычислении размера надо учитывать ещё выравнивание и
        ; дырки"" в памяти которые могут из за этого появляться.
        ;

        ;
        ; Сохранение регистров, которые могут портиться функцией, но которые
        ; по соглашению stdcall должны сохранить своё состояния до входа в
        ; функцию после после выхода из функции.
        ;
        push ebx
        push edi

        ;
        ; Сохраняем в eax переданный в функцию параметр -
        ; число, которое надо печатать.
        ;
        mov eax, [ebp+8]

        ;
```

```
    ; Константа 80000000h
    ; соответствует максимальному по модулю
    ; отрицательному числу. С таким числом будут проблемы при переводе в дополнительный код
    ;
    ; Поэтому для него нужно напечатать значение отдельно.
    ;
    cmp eax, 80000000h
    jne normal_print_dec
    OUTSTRLN "-2147483648"
    jmp print_dec_end ; как напечатали переходим на конец функции

normal_print_dec:

    ;
    ; Сохраняем в edi текущее состояние стека.
    ; нужно, чтобы вернуть состояние обратно, после того,
    ; как сформируем строку с цифрами числи и напечатаем
    ; эту строку.
    ;
    mov edi, esp

    ;
    ; Кладём 0 на дно, чтобы макрос печати не напечатал лишнего
    ; и не убежал за границы доступной программе памяти.
    ;
    sub esp, 1
    mov byte ptr [esp], 0

    ;
    ; ecx будет содержать 0 если число неотрицательное
    ; и ffffffff если число отрицательное.
    ;
    mov ecx, 0

    ;
    ; проверка числа на отрицательность, выставление ecx
    ; если отрицательное.
    ;
    ; используются анонимные метки (@@)
    ; @F – переход на ближайшую анонимную метку ниже по коду
    ; @B – переход на ближайшую анонимную метку выше по коду
    ;
    test eax, eax
    jns @F
    neg eax
    not ecx
@@:

    mov ebx, 10
```

```
print_digits_loop:
    mov edx, 0
    div ebx

    ;
    ; формируем символ цифры из остатка от деления и кода символа '0'.
    ;
    add edx, '0'

    ;
    ; Записываем символ цифры на вершину стека.
    ;
    sub esp, 1
    mov byte ptr [esp], dl

    ;
    ; проверяем не обнулился ли регистр eax,
    ; если обнулился формирование строки с цифрами завершено.
    ;
    test eax, eax
    jnz print_digits_loop

    ;
    ; Разбираемся нужно ли на вершину стека доложить
    ; символ '-'.
    ;
    test ecx, ecx
    jz not_print_minus
    sub esp, 1
    mov byte ptr [esp], '-'
not_print_minus:
    ;
    ; Печать строки с цифрами числа и знаком.
    ; На всякий случай, передадим макросу ecx, а не esp,
    ; так как у макроса в этом случае могут быть проблебемы
    ; с корректной работой со стеком.
    ;
    mov ecx, esp
    OUTSTRLN ecx

    ;
    ; после печати восстанавливаем стек к состоянию,
    ; когда мы не начинали формировать строку.
    ;
    mov esp, edi
print_dec_end:

    ;
    ; Восстанавливаем сохранённые регистры.
```

```
        ; перед выходом из функции.
        ;
        pop edi
        pop ebx

        ;
        ; Как правило здесь стираются локальные переменные.
        ; простейший способ это сделать:
        ; mov esp, ebp
        ;

        ;
        ; тандартныйC выход из функции.
        ; Восстанавливаем значение регистра ebp
        ; и ещё стираем стек от переданных в него параметров
        ; требование( соглашения stdcall).
        ; Из за очистки стека функцией требуется ret 4.
        ; В результате после выхода из функции значение esp увеличится ещё на 4
        ; суммарный( объём переданных параметров в функцию,
        ; тк.. один параметр, и он <=4 байт, то 4).
        ;
        pop ebp
        ret 4
print-dec endp


    start:
        OUTSTRLN "Please enter signed 32-bit number"
        ININT number

        ;
        ; передача параметра и вызов функции.
        ;
        push number
        call print-dec

        EXIT 0

end start
```

И последний пример: Сортировка массива record x,y,z :integer; end; по возрастанию радиус вектора.

```
include console.inc

Vec3D_X equ 0
Vec3D_Y equ 2
Vec3D_Z equ 4
```

```
    .data
        array dw 2, 3,  4
              dw 1, 1, -1
              dw 0, 1,  0
              dw 1, 1,  1
              dw 8, 9, 10
              dw 0, 1,  1
              dw 1, 0,  0


    .code


cmp_elms proc
    push ebp
    mov ebp, esp

    push esi
    push edi
    push ebx

    mov esi, [ebp +  8] ; адрес левого
    mov edi, [ebp + 12] ; адрес правого

    movsx eax, word ptr Vec3D_X[esi]
    imul eax
    mov ecx, eax

    movsx eax, word ptr  Vec3D_Y[esi]
    imul eax
    add ecx, eax

    movsx eax, word ptr  Vec3D_Z[esi]
    imul eax
    add ecx, eax

    movsx eax, word ptr Vec3D_X[edi]
    imul eax
    mov ebx, eax

    movsx eax, word ptr  Vec3D_Y[edi]
    imul eax
    add ebx, eax

    movsx eax, word ptr  Vec3D_Z[edi]
    imul eax
```

```
    add ebx, eax


    sub ecx, ebx
    mov eax, ecx

    pop ebx
    pop edi
    pop esi

    pop ebp
    ret 8
cmp_elms endp

swap_elms proc
    push ebp
    mov ebp, esp

    push esi
    push edi

    mov esi, [ebp +  8] ; адрес левого
    mov edi, [ebp + 12] ; адрес правого

    mov   ax, word ptr Vec3D_X[esi]
    xchg  ax, word ptr Vec3D_X[edi]
    xchg  ax, word ptr Vec3D_X[esi]

    mov   cx, word ptr Vec3D_Y[esi]
    xchg  cx, word ptr Vec3D_Y[edi]
    xchg  cx, word ptr Vec3D_Y[esi]

    mov   dx, word ptr Vec3D_Z[esi]
    xchg  dx, word ptr Vec3D_Z[edi]
    xchg  dx, word ptr Vec3D_Z[esi]


    pop edi
    pop esi

    pop ebp
    ret 8
swap_elms endp



arr_pointer equ    8
arr_length  equ    8+4
flg_has_chg equ   -4
```

```
sort_array proc
        push ebp
        mov  ebp, esp

        sub esp, 4 ; place on stack in local variables  for flag

        push esi
        push ebx

        mov esi, arr_pointer[ebp]
        mov ebx, arr_length[ebp]

        mov dword ptr flg_has_chg[ebp], 1

    loop_by_flag:
        cmp dword ptr flg_has_chg[ebp], 0
        je after_loop_by_flag
        mov  dword ptr flg_has_chg[ebp], 0

        mov ecx, 1
    loop_bubble:
        cmp ecx, ebx
        jae after_loop_bubble

        ;
        ; вычислить адрес ecxитого– элемента массива
        ;
        mov eax, 6
        mul ecx


        push ecx

        ;
        ; передача параметров в функцию сравнения.
        ;
        add eax, esi
        push eax
        sub eax, 6 ; адрес предыдущего
        push eax
        call cmp_elms

        pop ecx

        test eax, eax
        ;js not_do_swap проблема с совпадающими элементами
        jle not_do_swap

        ;
```

```
        ; вычислить адрес ecxитого– элемента массива
        ;
        mov eax, 6
        mul ecx

        push ecx
        ;
        ; передача мараметров в функцию сравнения.
        ;
        add eax, esi
        push eax
        sub eax, 6 ; адрес предыдущего
        push eax
        call swap_elms
        mov dword ptr flg_has_chg[ebp], 1

        pop ecx
    not_do_swap:

        inc ecx
        jmp loop_bubble
    after_loop_bubble:


        jmp loop_by_flag
    after_loop_by_flag:

        pop ebx
        pop esi

        mov esp, ebp ; стираем локальные переменные
        pop ebp
        ret 4+4
sort_array endp


; для массива нарушен принцип выравнивания на конце,
; поэтому размер элемента массива 6.
arr_pointer equ    8
arr_length  equ    8+4
print_array proc
        push ebp
        mov  ebp, esp

        push esi
        push ebx
        ;push edi

        mov esi, arr_pointer[ebp]
```

```
        mov ebx, arr_length[ebp]

        mov ecx, 0
    loop_print:
        cmp ecx, ebx
        jae after_loop_print

        ;
        ; вычислить адрес ecxитого- элемента массива
        ;
        mov eax, 6
        mul ecx

        OUTSTR "arr ["
        OUTWORD ecx
        OUTSTR "] = (x: "
        OUTINT word ptr Vec3D_X[eax+esi]
        OUTSTR ", y: "
        OUTINT word ptr Vec3D_Y[eax+esi]
        OUTSTR ", z: "
        OUTINT word ptr Vec3D_Z[eax+esi]
        OUTSTRLN ")"

        inc ecx
        jmp loop_print
    after_loop_print:

        ;pop edi
        pop ebx
        pop esi

        ;mov esp, ebp
        pop ebp
        ret 4+4
print_array endp

after_sort_msg db "Sorted array",0

    start:
        push 7 ; array size
        push offset array
        call print_array

        push 7 ; array size
        push offset array
        call sort_array


        OUTSTRLN offset after_sort_msg
```

```
        push 7 ; array size
        push offset array
        call print_array


        EXIT 0
end start
```

Глава 13

Числа с плавающей точкой

FPU

Существует FPU (сопроцессор 8087) - Floating point unit. Это самая древняя часть интеловской архитектуры для работы с числами с плавающей точкой, работает медленно, но задаёт хорошую точность. Позволяет вычислять значения в 10-байтном формате.
Здесь есть регистры, которые именуются достаточно странным образом, а именно st0, st1, . . . , st7.
Почему же st?
Потому что FPU это стыковая машина. St0 - вершина стека. St регистры по 80 бит (10 байт).
Будем пользоваться специальными командами, чтобы взаимодействовать со стеком.

Интересующие нас команды:
1) fld (операндом является только память) - принимает память, интерпретирует как число с плавающей точкой и кладёт на st0.
Пример работы:
Fld dword ptr mem; single precision/float
Fld qword ptr mem; double
Fld tword ptr mem; extended/long double
2) fstp (операнд - память) - забирает число из st0 и помещает в память.

Как преобразовать число из одинарной точности в двойную?
Примерно так:
Fld dword ptr . . .
Fstp qword ptr . . .

Расширение MMX

Изначально существовало расширение 3DNOW, потом появилось MMX, после оно трансформировалось в SSE, SSE2 и SSE3. Все эти расширения существовали во времена Pentium Pro
Между SSE3 и SSE4 появилась архитектура AMD64.
После этого SSE3 превратился в SSE4, а после в AVX. Начиная с AVX двухоперандные инструкции превратились в трёхоперандные.
AVX также превращались в AVX2 и наконец в AVX512.

Теперь у нас есть регистры помимо st, это серия xmm0...xmm7 - 128-ми битные. Они делятся на 4 части по 32 бита, но по ним нельзя обращаться. Также он делится на 2 части по 64 бита.
У этих регистров два способа жизни - скалярное обращение (задействуется только младшая часть регистра (32 или 64 бита), scalar) и векторное обращение (весь регистр целиком, packed), буковки s и p задействуются в инструкциях.
В AMD64 регистров стало в два раза больше, то есть 16. В AVX их уже 32, и регистры сменили название на ymm, который еще и стал 256 битным. В AVX512 регистры снова изменились на zmm - 512 бит.

Скалярные инструкции

Все команды взаимодействуют с 32/64 битами

1. Movss (операнд1 либо память, либо xmm), (операнд2 аналогично) - аналог mov первая s показывает, что инструкция скалярная, а вторая s показывает, что число одинарной точности (32 бита).
2. Movsd - всё аналогично, но теперь число double
3. Addss - аналог add одинарной точности, op1 - только xmm
4. Addsd - всё аналогично, но теперь число double
5. Subss - аналог sub одинарной точности, op1 - только xmm
6. Subsd - всё аналогично, но теперь число double
7. Mulss - Перемножить младшее из упакованных значений op1 и младшее из op2, результат положить в op1. op1 - xmm регистр, op2 - xmm регистр или 4-х байтная ячейка памяти.
8. Mulsd - всё аналогично, но теперь число double
9. Divss - Поделить младшее из упакованных значений op1 и младшее из op2, результат положить в op1. op1 - xmm регистр, op2 - xmm регистр или 4-х байтная ячейка памяти.
10. Divsd - всё аналогично, но теперь число double
11. Comiss - Сравнивает младшие короткие вещественные значения, op1 - только xmm. Выставляет флаги ZF и CF.
12. Comisd - Сравнивает вещественные значения, op1 - только xmm. Выставляет флаги ZF и CF.

Векторные инструкции

1. Movups - аналог mov, op1 - xmm или память 128 байт
2. Movupd - аналог mov, op1 - xmm или память 128 байт
3. Movaps - аналог mov для выровненных по 16 адресов, op1 - xmm или память 128 байт
4. Movapd - аналог mov для выровненных по 16 адресов, op1 - xmm или память 128 байт
5. Addps - аналог add одинарной точности, op1 - только xmm
6. Subps - аналог add одинарной точности, op1 - только xmm
7. Mulps - Перемножить op1 и op2, результат положить в op1. op1 - xmm регистр, op2 - xmm регистр или 16-х байтная ячейка памяти.
8. Divps - Поделить op1 и op2, результат положить в op1. op1 - xmm регистр, op2 - xmm регистр или 16-х байтная ячейка памяти.

9. shufps - Копировать в op1 (Только XMM) слова из op2. Команда SHUFPS осуществляет пересылку любых двух из четырех коротких вещественных значений, упакованных в операнде-источнике команды (SIMD-регистр или операнд в памяти) в младшие позиции в операнде-назначении (SIMD-регистр), а также любых двух из четырех коротких вещественных значений, упакованных в операнде-назначении команды (SIMD-регистр) в старшие позиции этого же операнда. Для каждой позиции в операнде-назначении третий операнд команды (imm8) задает номер копируемого в нее слова из операнда-источника (для двух младших полей) или из операнда-назначения (для двух старших полей).
   Биты 0,1 параметра imm8 задают номер копируемого слова для младшего 16-битного поля операнда-назначения, биты 2, 3 — для следующего, 4,5 — для третьего и 6, 7 — для самого старшего. Эти биты кодируются обычными двоичными кодами, например код 11b в битах 2, 3 означает, что в соответствующую позицию будет скопировано самое старшее слово из операнда-источника.
10. Cvt - (convert) 'Что (p/s s/d/i)' 2 'Куда (p/s s/d/i)' (op1, op2). Op1- что, op2 - куда. Пример конвертами одинарной точности в двойную: Cvt ss 2 sd xmm0, xmm1

Пример

```
include console.inc

xmm0 equ XMM0
xmm1 equ XMM1
xmm2 equ XMM2
xmm3 equ XMM3
xmm4 equ XMM4
xmm5 equ XMM5
xmm6 equ XMM6
xmm7 equ XMM7

.listmacro

    .code
        L_fmt_str_macros_for_float db "%f", 0
        L_fmt_str_macros_for_tenbyte_float db "%lf", 0

outfloat equ OUTFLOAT
OUTFLOAT macro numb_to_print
    if type numb_to_print EQ 4
        pushad
        pushfd

        mov eax, numb_to_print
        sub esp, 8
        mov [esp], eax
        fld dword ptr [esp] ; load to FPU with conversion to 80-bit float
        fstp qword ptr [esp] ; get from fpu with conversion to 64-bit
        push offset L_fmt_str_macros_for_float
```

```
        call crt_printf
        add esp, 4+8

        popfd
        popad
    endif
    if type numb_to_print EQ 8 ; support only memory argument

        pushad
        pushfd

        lea eax, numb_to_print
        invoke crt_printf, offset L_fmt_str_macros_for_float, dword ptr [eax], dword ptr

        popfd
        popad
    endif
    if type numb_to_print EQ 10 ; support only memory argument

        pushad
        pushfd

        lea eax, numb_to_print
        sub esp, 10
        mov ecx, [eax]
        mov [esp], ecx
        mov edx, [eax+4]
        mov [esp+4], edx
        mov bx, [eax+8]
        mov [esp+8], bx
        push offset L_fmt_str_macros_for_float
        call crt_printf
        add esp, 4+10

        popfd
        popad
    endif

endm



Vec3D_X equ 0
Vec3D_Y equ 4
Vec3D_Z equ 8
size_Vec3D equ 12


    .data
```

```
        array real4 2.1, 3.3,  4.4
              real4 1.0, 1.0, -1.0
              real4 0.0, 1.1,  0.0
              real4 1.5, 1.5,  1.5
              real4 8.8, 9.9, 10.99
              real4 0.0, 1.0,  1.0
              real4 1.0, 0.0,  0.0


    .code


cmp_elms proc
    push ebp
    mov ebp, esp

    push esi
    push edi
    push ebx

    mov esi, [ebp +  8] ; адрес левого
    mov edi, [ebp + 12] ; адрес правого

    xorps XMM0, XMM0
    xorps xmm1, xmm1

    movss XMM2, dword ptr Vec3D_X[esi]
    movss XMM3, dword ptr Vec3D_Y[esi]
    movss XMM4, dword ptr Vec3D_Z[esi]

    movss XMM5, dword ptr Vec3D_X[edi]
    movss XMM6, dword ptr Vec3D_Y[edi]
    movss XMM7, dword ptr Vec3D_Z[edi]

    mulss XMM2, XMM2
    mulss XMM3, XMM3
    mulss XMM4, XMM4

    addss XMM0, XMM2
    addss XMM0, XMM3
    addss XMM0, XMM4

    mulss XMM5, XMM5
    mulss XMM6, XMM6
    mulss XMM7, XMM7

    addss XMM1, XMM5
    addss XMM1, XMM6
    addss XMM1, XMM7
```

```
    comiss XMM0, XMM1
    mov eax,  0
    mov ecx,  1
    mov edx, -1
    cmovb eax, edx
    cmova eax, ecx

    pop ebx
    pop edi
    pop esi

    pop ebp
    ret 8
cmp_elms endp

swap_elms proc
    push ebp
    mov ebp, esp

    push esi
    push edi

    mov esi, [ebp +  8] ; адрес левого
    mov edi, [ebp + 12] ; адрес правого

    mov  eax, dword ptr Vec3D_X[esi]
    xchg eax, dword ptr Vec3D_X[edi]
    xchg eax, dword ptr Vec3D_X[esi]

    mov  ecx, dword ptr Vec3D_Y[esi]
    xchg ecx, dword ptr Vec3D_Y[edi]
    xchg ecx, dword ptr Vec3D_Y[esi]

    mov  edx, dword ptr Vec3D_Z[esi]
    xchg edx, dword ptr Vec3D_Z[edi]
    xchg edx, dword ptr Vec3D_Z[esi]


    pop edi
    pop esi

    pop ebp
    ret 8
swap_elms endp



arr_pointer equ    8
```

```
arr_length  equ    8+4
flg_has_chg equ   -4
sort_array proc
        push ebp
        mov  ebp, esp

        sub esp, 4 ; place on stack in local variables  for flag

        push esi
        push ebx

        mov esi, arr_pointer[ebp]
        mov ebx, arr_length[ebp]

        mov dword ptr flg_has_chg[ebp], 1

    loop_by_flag:
        cmp dword ptr flg_has_chg[ebp], 0
        je after_loop_by_flag
        mov  dword ptr flg_has_chg[ebp], 0

        mov ecx, 1
    loop_bubble:
        cmp ecx, ebx
        jae after_loop_bubble

        ;
        ; вычислить адрес ecxитого– элемента массива
        ;
        mov eax, size_Vec3D
        mul ecx


        push ecx

        ;
        ; передача параметров в функцию сравнения.
        ;
        add eax, esi
        push eax
        sub eax, size_Vec3D ; адрес предыдущего
        push eax
        call cmp_elms

        pop ecx

        test eax, eax
        ;js not_do_swap проблема с совпадающими элементами
        jle not_do_swap
```

```
        ;
        ; вычислить адрес ecxитого- элемента массива
        ;
        mov eax, size_Vec3D
        mul ecx

        push ecx
        ;
        ; передача мараметров в функцию сравнения.
        ;
        add eax, esi
        push eax
        sub eax, size_Vec3D ; адрес предыдущего
        push eax
        call swap_elms
        mov dword ptr flg_has_chg[ebp], 1

        pop ecx
    not_do_swap:

        inc ecx
        jmp loop_bubble
    after_loop_bubble:


        jmp loop_by_flag
    after_loop_by_flag:

        pop ebx
        pop esi

        mov esp, ebp ; стираем локальные переменные
        pop ebp
        ret 4+4
sort_array endp


; для массива нарушен принцип выравнивания на конце,
; поэтому размер элемента массива size_Vec3D.
arr_pointer equ    8
arr_length  equ    8+4
print_array proc
        push ebp
        mov  ebp, esp

        push esi
        push ebx
        ;push edi
```

```
        mov esi, arr_pointer[ebp]
        mov ebx, arr_length[ebp]

        mov ecx, 0
    loop_print:
        cmp ecx, ebx
        jae after_loop_print

        ;
        ; вычислить адрес ecxитого- элемента массива
        ;
        mov eax, size_Vec3D
        mul ecx

        OUTSTR "arr ["
        OUTWORD ecx
        OUTSTR "] = (x: "
        OUTFLOAT dword ptr Vec3D_X[eax+esi]
        OUTSTR ", y: "
        OUTFLOAT dword ptr Vec3D_Y[eax+esi]
        OUTSTR ", z: "
        OUTFLOAT dword ptr Vec3D_Z[eax+esi]
        OUTSTRLN ")"

        inc ecx
        jmp loop_print
    after_loop_print:

        ;pop edi
        pop ebx
        pop esi

        ;mov esp, ebp
        pop ebp
        ret 4+4
print_array endp

after_sort_msg db "Sorted array",0

    start:
        push 7 ; array size in elements
        push offset array
        call print_array

        push 7 ; array size in elements
        push offset array
        call sort_array
```

```
        OUTSTRLN offset after_sort_msg

        push 7 ; array size in elements
        push offset array
        call print_array


        EXIT 0
end start
```

Глава 14

Структуры

Пример без структур

Рассмотрим программу, которая позволяет нам строить "дерево"и выводить значения его "листьев".

```
include console.inc

;
; tree-node = record
;               num: integer;
;               count: longword;
;               left, right: ^tree-node;
;             end;
;
; tree = ^tree-node;
;
tree-node-num   equ  0
tree-node-count equ  4
tree-node-left  equ  8
tree-node-right equ 12
tree-node-SIZE  equ 16


    .code

;
; function add-to-tree(root-pointer: ^tree, elm: longint): bool
; Функция принимает 2 параметра:
;
; 1 - указатель на на то место где хранится корень дерева,
;     чтобы его можно было поменять.
;
; 2 - число, которое будем добавлять в дерево.
;
add-to-tree proc public
        push ebp
        mov ebp, esp

        push edi
        push esi
        push ebx

        mov edi, [ebp+8] ; edi  указатель на конень дерева
        mov esi, [edi]   ; esi текущее значение, которое лежит в корне.

        mov bx,  [ebp+12] ; число, которое добавляем.

        test esi, esi     ; Проверка а не nil ли корень дерева.
        jne check-subtrees
```

```
    NEW tree-node-SIZE
    test eax, eax ; проверяем на то, что память правда выделилась.
    je exit-add-to-tree ; В eax значение 0 - false
                        ; вернёмся как раз с false из функции


    ;
    ; Инициализируем память нового
    ; элемента дерева.
    ;
    mov [eax+tree-node-num], bx
    mov dword ptr [eax+tree-node-count], 1
    mov dword ptr [eax+tree-node-left], 0
    mov dword ptr [eax+tree-node-right], 0
    mov [edi], eax ; сохраняем новый корень в память того места
                   ; где он должен лежать фактически.

    ;
    ; формируем возвращаемое значение функции
    ; как true.
    ;
    mov eax, 1
    jmp exit-add-to-tree
check-subtrees:

    cmp [esi+tree-node-num], bx
    jle check-upper-or-equal

    ;
    ; num < root.num
    ;

    lea eax, [esi+tree-node-left]
    push ebx
    push eax
    call add-to-tree
    jmp exit-add-to-tree
check-upper-or-equal:
    je process-equal

    ;
    ; num > root.num
    ;
    lea eax, [esi+tree-node-right]
    push ebx
    push eax
    call add-to-tree
    jmp exit-add-to-tree
```

```
    process-equal:
        inc dword ptr [esi+tree-node-count]
        mov eax, 1

    exit-add-to-tree:

        pop ebx
        pop esi
        pop edi

        pop ebp
        ret 8
add-to-tree endp

print-tree proc public
        push ebp
        mov ebp, esp

        push esi

        mov esi, [ebp+8]
        test esi, esi
        je exit-print-tree

        push [esi+tree-node-left]
        call print-tree

        OUTCHAR '('
        mov ax, [esi+tree-node-num]
        OUTINT ax
        OUTCHAR ','
        mov eax, [esi+tree-node-count]
        OUTWORD eax
        OUTCHARLN ')'

        push [esi+tree-node-right]
        call print-tree

    exit-print-tree:

        pop esi

        pop ebp
        ret 4
print-tree endp

    .data
        tree-root dd 0
```

```
.code

    welcome-message db "Please enter 2-byte signed numbers."
                    db " zero value - stop."
                    db 0

    by-by-message db "finally",0
start:
    OUTSTRLN  offset welcome-message

reading-loop:
    ININT ax
    test ax, ax
    je finish-reading

    push eax
    push offset tree-root
    call add-to-tree

    test  eax, eax
    je exit-with-error

     push tree-root
     call print-tree

    jmp reading-loop
finish-reading:

    OUTSTRLN offset by-by-message
    push tree-root
    call print-tree

    EXIT 0
exit-with-error:
    OUTSTRLN "Can't add to tree number"
    EXIT 1
end start
```

Структуры

Синтаксис самой конструкции выглядит так:
tree-node struct
Описание полей
tree-node ends

Чтобы обратиться к полю структуры нужно использовать:
tree-node.имя-поля

Также узнаем про size, size of, length, length of на примере
A dd 10 dup (42), -1, -2
size A = 4*10 = 40
size of A = 4*12 = 48
length A = 10
length of A = 12

Пример со структурами

```
include console.inc

;
; tree-node = record
;                 num: integer;
;                 count: longword;
;                 left, right: ^tree-node;
;               end;
;
; tree = ^tree-node;
;

;tree-node-num    equ  0
;tree-node-count  equ  4
;tree-node-left   equ  8
;tree-node-right  equ 12
;tree-node-SIZE   equ 16


Tree-node struct
    num dw ?
        dw ?
    count dd ?
    left  dd ?
    right dd ?
Tree-node ends


    .code

;
; function add-to-reee(root-pointer: ^tree, elm: integer): bool
; Функция принимает 2 параметра:
;
; 1 - указатель на на то место где хранится корень дерева,
;     чтобы его можно было поменять.
;
; 2 - число, которое будем добавлять в дерево.
```

```
;
add-to-tree proc public
        push ebp
        mov ebp, esp

        push edi
        push esi
        push ebx

        mov edi, [ebp+8] ; edi  указатель на конень дерева
        mov esi, [edi]   ; esi текущее значение, которое лежит в корне.

        mov bx,  [ebp+12] ; число, которое добавляем.

        test esi, esi     ; Проверка а не nil ли корень дерева.
        jne check-subtrees

        NEW sizeof Tree-node
        test eax, eax ; проверяем на то, что память правда выделилась.
        je exit-add-to-tree ; В eax значение 0 - false
                            ; вернёмся как раз с false из функции


        ;
        ; Инициализируем память нового
        ; элемента дерева.
        ;
        mov Tree-node.num[eax], bx
        mov Tree-node.count[eax], 1
        mov Tree-node.left[eax], 0
        mov Tree-node.right[eax], 0
        mov [edi], eax ; сохраняем новый корень в память того места
                       ; где он должен лежать фактически.

        ;
        ; формируем фозвращаемое значение функции
        ; как true.
        ;
        mov eax, 1
        jmp exit-add-to-tree
    check-subtrees:

        cmp Tree-node.num[esi], bx
        jle check-upper-or-equal

        ;
        ; num < root.num
        ;
```

```
        lea eax, Tree-node.left[esi]
        push ebx
        push eax
        call add-to-tree
        jmp exit-add-to-tree
    check-upper-or-equal:
        je process-equal

        ;
        ; num > root.num
        ;
        lea eax, Tree-node.right[esi]
        push ebx
        push eax
        call add-to-tree
        jmp exit-add-to-tree

    process-equal:
        inc dword ptr Tree-node.count[esi]
        mov eax, 1

    exit-add-to-tree:

        pop ebx
        pop esi
        pop edi

        pop ebp
        ret 8
add-to-tree endp

print-tree proc public
        push ebp
        mov ebp, esp

        push esi

        mov esi, [ebp+8]
        test esi, esi
        je exit-print-tree

        push Tree-node.left[esi]
        call print-tree

        OUTCHAR '('
        mov ax, Tree-node.num[esi]
        OUTINT ax
        OUTCHAR ','
        mov eax, [esi+Tree-node.count]
```

```
        OUTWORD eax
        OUTCHARLN ')'

        push [esi+Tree-node.right]
        call print-tree

    exit-print-tree:

        pop esi

        pop ebp
        ret 4
print-tree endp

    .data
        tree-root dd 0
    .code

        welcome-message db "Please␣enter␣2-byte␣signed␣numbers."
                        db "␣zero␣value␣-␣stop."
                        db 0

        by-by-message db "finally",0
    start:
        OUTSTRLN  offset welcome-message

    reading-loop:
        ININT ax
        test ax, ax
        je finish-reading

        push eax
        push offset tree-root
        call add-to-tree

        test  eax, eax
        je exit-with-error

         push tree-root
         call print-tree

        jmp reading-loop
    finish-reading:

        OUTSTRLN offset by-by-message
        push tree-root
        call print-tree

        EXIT 0
```

```
    exit-with-error:
        OUTSTRLN "Can't add to tree number"
        EXIT 1
end start
```

Глава 15

Строковые команды

Строковые команды

Изучение строковых команд начнём с флага DF - флаг напраления движения по памяти. Когда DF = 0, то мы просматриваем строку вперёд, а когда DF = 1, то назад.
Управлять флагом DF можно с помощью команд cld (DF = 0) и std (DF = 1)

Строковый mov
movsb, movsw, movsd - копирует текущий элемент из массива esi в текущий элемент edi. (Для byte, word и dword)
Чтобы передать всю строку нужно использовать префикс повторения rep (эквивалентно)
Можно переписать тоже самое без строковых команд на трёх регистрах.
metka:
mov [edi], [esi]
Dec ecx
Если DF = 1:
Dec ESI
Dec EDI
Иначе:
Inc ESI
Inc ESI
cmp ecx, 0
jnz metka

Строковые сравнения
cmpsb, cmpsw, cmpsd - сравнивает текущие элементы в массиве по адресу ESI, и массиве по адресу EDI. (Для byte, word, dword)
Чтобы сравнить обе строки используются префиксы повторения repE - повторение пока строки равны и repNE - повторять пока строки не равны.

Сканирование строки
scasb (сравнивает строку с AL), scasw (сравнивает строку с регистром AX), scasd (сравнивает строку с регистром EAX). Строка которая сравнивается лежит в регистре EDI
Можно использовать префикс repE, тогда строка будет сканироваться до того, пока не будет найден первый элемент отличный от образца. Если использовать repNE - строка сканируется до первого совпадения.

Загрузка и выгрузка элементов из строки.
lodsb, lodsw, lodsd - загружает содержимое [ESI] в AL/AX/EAX.
stosb, stosw, stosd - выгружает содержимое AL/AX/EAX в [EDI]

Пример

```
include console.inc

    .data
        arr-1 dd 1, 2, 1, 4,  5, -3, 8, -2
        arr-2 dd 5, 6, 1, 4, -5,  9, 3,  2
```

```
    .code

print-array proc
        push ebp
        mov ebp, esp

        mov ecx, 0
        mov edx, [ebp+12] ; размер массива
        mov eax, [ebp+8] ; указатель на начало
    loop-print-arr:
        cmp ecx, edx
        jae after-loop-print-arr

        OUTCHAR 9   ; символ табуляции
        OUTWORD ecx
        OUTSTR ": "
        OUTINTLN dword ptr [eax+ ecx*4]

        inc ecx
        jmp loop-print-arr
    after-loop-print-arr:
        pop ebp
        ret 4+4
print-array endp
    start:

        OUTSTRLN "arr 1: before"
        push 8
        push offset arr-1
        call print-array

        OUTSTRLN "arr 2: before"
        push 8
        push offset arr-2
        call print-array

comment $
        mov ecx, 8
        mov esi, offset arr-1
        mov edi, offset arr-2
        cld
        repe cmpsd

        mov eax, 7
        sub eax, ecx
$

comment $
        mov ecx, 8
```

```
        mov esi, offset arr-1
        mov edi, offset arr-2
        cld
        rep movsd

$


comment $
        mov ecx, 8
        mov esi, offset arr-1
        mov edi, offset arr-2
        mov eax, 5
        cld
        repne scasd

        mov eax, 7
        sub eax, ecx
$

        mov ecx, 8
        mov edi, offset arr-1 + sizeof arr-2
        mov eax, 1
        std
        repne scasd

        mov eax, ecx


        OUTSTR "We stop at "
        OUTWORDLN eax

        OUTSTRLN "arr 1: after"
        push 8
        push offset arr-1
        call print-array

        OUTSTRLN "arr 2: after"
        push 8
        push offset arr-2
        call print-array


        exit 0
end start
```

Глава 16

Макросы

MASM

Рассмотрим этапы компиляции кода:

1. Макроподстановка
2. Компиляция
3. Получение объектного файла

Во время выполнение make в ассемблере, в какой-то момент мы дойдём до подстановки макросов. Один макрос может быть вложен в другой. Закончится отдельная подстановка в тот момент, когда мы подставим текст вместо всех вложенных макросов.
Текст подстановки нельзя получить в привычном нам виде во время подстановки. В языке C для этого есть ключ -E, который остановит компиляцию и позволит нам увидеть подставленный текст.

В директиве .listmacro начинается перечисление инструкций расширения макросов, которые создают код или данные.

Если в макросе написать имя equ "выражение"или имя = "Выражение то во время макроподстановки будет подставлено “выражение” целиком, без вычисления. Если перед именем (макропеременной) выставлен процент, то будет подставлено уже вычисленное выражение.

Как выглядит макрос?
Имя macro параметр 1, параметр 2, ...
Тело макроса
endm
Существует инструкция exitm, которая перемещает программу на endm.
Параметром может быть:

1. Параметр (имя)
2. Обязательный параметр (имя:req)
3. Параметр со значением по умолчанию (имя:=текстовое выражение)

Если у нас есть имя переменной "a a мы хотим передать "a"как символ, то надо будет добавить восклицательный знак вот так: a!
Пример: BIAKA a!, , 12 (первый параметр - символ a, второго нет, а третий 12)
Если же написать BIAKA a,<что-то> , b, то будем считать, что в <> устойчивая конструкция

При выводе текста в макросе можно напрямую передавать в него значение переменных.
Например:
YEAR = 2025
“Сегодня &YEAR& год” - будет выведено Сегодня 2025 год
Также, отметим, что чтобы добавить комментарий в макрос, то надо написать две ; иначе во время макроподстановки обычный комментарий тоже будет подставлен.

Чтобы пользоваться метками придётся задействовать инструкцию local, без неё при макроподстановке может быть подставлена одна и та же метка, из-за чего будет невозможно

совершить прыжок.
Пример:
local mark-err
...
mark-err (при листинге станет ??000017 или другие цифры)

Также отметим конструкцию .err и echo
Первая выглядит как .err текст
Когда компилятор доходит до .err, то компиляция закончится и выведется текст (Остановка с ошибкой)
Но, это не остановит макроподстановку и надо написать exitm
Вторая как echo текст, она просто выведет текст в стандартный поток вывода.

Пример

Данный пример описывает макрос, который складывает два 64-х битных числа.

```
include console.inc
.listmacro

SUM64 macro l-arg:req, r-arg:req
    push esi
    push edi

    echo l-arg
    echo r-arg
    ;; достали младшую часть левого аргумента
    lea esi, l-arg
    lea edi, r-arg

    mov eax, dword ptr [esi]
    ;; достали старшую часть
    mov edx, dword ptr [esi+4]
    add eax, dword ptr [edi]
    adc edx, dword ptr [edi+4]

    pop edi
    pop esi
endm

    .data
        a dq -1
        b dq 42

    .code
    start:

        SUM64 a, b
```

```
        OUTINTLN edx
        OUTINTLN eax

        exit 0
end start
```

Циклы и условия

Надо рассмотреть несколько инструкций, а именно серию ifxxxx.

1. ifdiff <text1>, <text2> (ставим угловые скобки, чтобы запятая из текста не вызывала проверку второго операнда). Если текст не совпадает, то true.
2. ifdiffi <text1>, <text2>. Если текст не совпадает без учёта регистров, то true.
3. ifidn <text1>, <text2> - если тексты совпадают, то true.
4. ifidni <text1>, <text2> - если тексты совпадают без учёта регистров, то true. Например biaka = BiAkA.
5. ifb <text> - true, если текст пустой.
6. ifnb <text> - true, если текст не пустой.
7. if - логическое выражение с большой вариативностью.

После if может идти else или даже elseif.
Для if доступны следующие конструкции:

1. a EQ b - a = b
2. a NE b - a $\neq$ b
3. a GE b - a $\geqslant$ b
4. a GT b - a $>$ b
5. a LE b - a $\leq$ b
6. a LT b - a $<$ b
7. and
8. or
9. xor
10. not

Рассмотрим циклические инструкции
цикл while - цикл с логическим выражением. Если условие после while является неверным, MASM пропускает текст между выражениями while и endm (аналогично оператору if). Если выражение является истинным, MASM обработает операторы между выражениями while и endm, а затем повторяет этот процесс, пока истинно условие.

цикл for имеет идентификатор и список аргументов в $< >$, которые идут через запятую. Данный идентификатор проходится по каждому элементу и использует его в макроподстановке на месте идентификатора.
цикл forc имеет идентификатор и какое-то слово. Данный идентификатор проходится по каждому символу слова и использует его в макроподстановке на месте идентификатора.
Пример для forc
forc reg-p, abcd
push e&reg-p&x
endm

Чтобы проверить, что именно представляет аргумент макроса, применяется оператор opattr.
opattr выражение
В качестве операнда он принимает некоторое выражение - это может быть и аргумент макроса, и какое-то более сложное по структуре выражение. opattr возвращает целочисленное значение, которые представляет битовую маску (то есть цифра в пояснении это номер бита).

1. 0 - Выражение представляет метку
2. 1 - Выражение представляет перемещаемое значение
3. 2 - Выражение представляет константу
4. 3 - Выражение представляет прямую адресацию
5. 4 - Выражение представляет регистр
6. 5 - устаревшее (выражение содержит неопределенный идентификатор)
7. 6 - Выражение представляет обращение к памяти стека
8. 7 - Выражение ссылается на внешний идентификатор