КОНСТРУИРОВАНИЕ ПРОГРАММ

Лекция № 04 Ассемблер NASM

+375 17 293 8039 (505a-5)

+375 17 320 7402 (ОИПИ НАНБ)

prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by/

Кафедра ЭВМ, 2022

2022.03.09

Оглавление

Псевдо-инструкции

Псевдо-инструкции — это некоторые директивы препроцессора и ассемблера.

Псевдо-инструкции не являются реальными инструкциями процессора, однако располагаются в исходном тексте в поле инструкций, поскольку это наиболее подходящее для них место.

Псевдо-инструкциями являются:

- директивы определения данных (выделения инициализированной памяти в .data)

  DB, DW, DD, DQ, DT, DO, DY, DZ;

- директивы резервирования памяти (выделения неинициализированной памяти в .bss)

  RESB, RESW, RESD, RESQ, REST, RESO, RESY, RESZ;

- команды

  INCBIN, EQU;

- префикс

  TIMES.

Псевдо-инструкции определения данных и резервирования памяти

Определение	Резервирование	Размер	Название
DB	RESB	1 байт (8 бит)	**B**yte
DW	RESW	2 байта (16 бит)	**W**ord
DD	RESD	4 байта (32 бита)	**D**ouble word
DQ	RESQ	8 байт (64 бита)	**Q**uad word
DT	REST	10 байт (80 бит)	Extended-precision float
DO	RESO	16 байт (128 бит)	IEEE 754r quad precision
DY	RESY	32 байта (256 бит)	Advanced Vector Extensions (AVX)
DZ	RESZ	64 байта (512 бит)	Advanced Vector Extensions (AVX-512)

DB и ее аналоги: Объявление инициализированных данных

DB, **DW**, **DD**, **DQ** … используются для объявления инициализированных данных в выходном файле. Они могут использоваться достаточно многими способами:

```
db    0x55                ; всего один байт 0x55
db    0x55,0x56,0x57      ; три байта друг за другом
db    'a',0x55            ; символьная константа
db    'hello',13,10,'$'   ; строковая константа
dw    0x1234              ; 0x34 0x12
dw    'a'                 ; 0x61 0x00 (просто число)
dw    'ab'                ; 0x61 0x62 (символьная константа)
dw    'abc'               ; 0x61 0x62 0x63 0x00 (строка)
dd    0x12345678          ; 0x78 0x56 0x34 0x12
dd    1.234567e20         ; константа с плавающей точкой
dq    0x123456789abcdef0  ; восьмибайтовая константа
dq    1.234567e20         ; константа с плавающей точкой двойной точности
dt    1.234567e20         ; константа с плавающей точкой расширенной точности
```

Листинг

```
 1                          section  .data
 2 00000000 55                       db   0x55                  ;  всего один байт 0x55
 3 00000001 555657                   db   0x55,0x56,0x57        ;  три байта друг за другом
 4 00000004 6155                     db   'a',0x55              ;  символьная константа
 5 00000006 68656C6C6F0D0A24         db   'hello',13,10,'$'     ;  строковая константа
 6 0000000E 3412                     dw   0x1234                ;  0x34 0x12
 7 00000010 6100                     dw   'a'                   ;  0x61 0x00 (просто число)
 8 00000012 6162                     dw   'ab'                  ;  0x61 0x62 (символьная константа)
 9 00000014 61626300                 dw   'abc'                 ;  0x61 0x62 0x63 0x00 (строка)
10 00000018 78563412                 dd   0x12345678            ;  0x78 0x56 0x34 0x12
11 0000001C CA29D660                 dd   1.234567e20           ;  константа с плавающей точкой
12 00000020 F0DEBC9A78563412         dq   0x123456789abcdef0    ;  восьмибайтовая константа
13 00000028 DF87313A39C51A44         dq   1.234567e20           ;  константа с плав. точкой двойной точности
14 00000030 00F83E8CD1C929D64140     dt   1.234567e20           ;  константа с плав. точкой расширенной точности
```

DT, DO, DY, DZ не допускают в качестве операндов целочисленных или строковых констант.

RESB и ее друзья: Объявление неинициализированных данных

RESB, **RESW**, **RESD**, **RESQ**, **RESO**, **RESY**, **RESZ** и **REST** разработаны для использования в BSS-секции модуля — они объявляют не инициализированное пространство для хранения данных.

Каждая принимает один операнд, являющийся числом резервируемых байт, слов, двойных слов и т.д.

```
buffer:         resb    64              ; зарезервировать 64 байта
wordvar:        resw    1               ; зарезервировать слово
realarray       resq    10              ; массив из десяти вещественных (float)
ymmval:         resy    1               ; один YMM регистр
zmmvals:        resz    32              ; 32 ZMM регистров
```

Листинг

```
1                           section  .bss
2 00000000 <res 00000040>  buffer:   resb 64 ; зарезервировать 64 байта
3 00000040 <res 00000002>  wordvar:  resw  1 ; зарезервировать слово
4 00000042 <res 00000050>  realarray resq 10 ; массив из десяти вещественных (float)
5 00000092 <res 00000020>  ymmval:   resy  1 ; один YMM регистр
6 000000B2 <res 00000800>  zmmvals:  resz 32 ; 32 ZMM регистров
```

Dx: Дополнение к объявлению данных

NASM версии ниже 2.15 не поддерживает синтаксис резервирования неинициализированного пространства, реализованный в MASM/TASM, где можно делать **DW ?** или подобное.

```
dw      ?
```

Начиная с версии 2.15 это стало возможным. Также была реализована возможность использовать расширенный синтаксис **DUP**.

Версия NASM 2.15 имеет следующую спецификацию синтаксиса — заглавные полужирные буквы обозначают буквальные ключевые слова:

```
dx      := DB | DW | DD | DQ | DT | DO | DY | DZ
type    := BYTE | WORD | DWORD | QWORD | TWORD | OWORD | YWORD | ZWORD
atom    := expression | string | float | '?'
parlist := '(' value [, value ...] ')'
duplist := expression DUP [type] ['%'] parlist
list    := duplist | '%' parlist | type ['%'] parlist
value   := atom | type value | list

stmt    := dx value [, value...]
```

Следует обратить внимание, что список должен начинаться со знака **%**, если он не имеет префикса **DUP** или типа.

Все следующие выражения допустимы:

```
      db 33
      db (44)                 ; Целочисленное выражение
    ; db (44,55)              ; Недействительно -- ошибка
      db %(44,55)
      db %('XX','YY')
      db ('AA')               ; Целочисленное выражение – выводится один байт
      db %('BB')              ; список, содержащий строку
      db ?
      db 6 dup (33)
      db 6 dup (33, 34)
      db 6 dup (33, 34), 35
      db 7 dup dword (?, word ?, ?)
      dw byte (?,44)
      dw 3 dup (0xcc, 4 dup byte ('PQR'), ?), 0xabcd
      dd 16 dup (0xaaaa, ?, 0xbbbbbb)
      dd 64 dup (?)
```

Использование **$** (текущий адрес) в псевдоинструкциях **Dx** в текущей версии NASM не определено, за исключением следующих случаев:

- Для первого выражения в инструкции, либо в DUP или элементе данных.

- Выражение вида «**expr - $**», которое преобразуется в смещение относительно текущего выражения.

В будущих версиях NASM, вероятно, будет по-другому или в этом случае возникнет ошибка.

Такого рода ограничений на использование символов **$$** или символов, определенных смещением внутри секций нет.

```
buffer:         resb    64              ; зарезервировать 64 байта
wordvar:        resw    1               ; зарезервировать слово
realarray       resq    10              ; массив из десяти вещественных (float)
ymmval:         resy    1               ; один YMM регистр
zmmvals:        resz    32              ; 32 ZMM регистров
```

Поскольку, начиная с NASM 2.15, поддерживается MASM-синтаксис использования `?` и **`DUP`** в директивах **`Dx`**, приведенный выше пример можно также записать как:

```
buffer:         db      64 dup (?)      ; зарезервировать 64 байта
wordvar:        dw      ?               ; зарезервировать слово
realarray       dq      10 dup (?)      ; массив из десяти вещественных (float)
ymmval:         dy      ?               ; один YMM регистр
zmmvals:        dz      32 dup (?)      ; 32 ZMM регистров
```

INCBIN: Включение внешних бинарных файлов

INCBIN включает бинарный файл в выходной файл, оставляя его (бинарный файл) неизменным.

Это может быть полезно (например) для включения картинок и музыки непосредственно в исполняемый файл.

Эта псевдо-инструкция может быть вызвана тремя разными способами:

```
    incbin "file.dat"            ; включение файла целиком
    incbin "file.dat", 1024      ; пропуск первых 1024 байт
    incbin "file.dat", 1024, 512 ; пропуск первых 1024 и включение следующих 512 байт
```

INCBIN — это и директива, и стандартный макрос; стандартная версия макроса ищет файл в пути поиска включаемых файлов и добавляет файл в списки зависимостей. При желании этот макрос можно переопределить.

EQU: Определение констант

EQU вводит символ для указанного константного значения.

Если используется **EQU**, в этой строке кода должна присутствовать метка.

Смысл **EQU** — связать имя метки со значением ее (только) операнда.

Данное определение абсолютно и не может быть позднее изменено. Например,

```
message   db 'Привет, Вася!'
msglen    equ $-message
```

определяет **msglen** как константу **13**.

Константа, определенная с помощью EQU, не может быть переопределена.

Т.е. **msglen** переопределена быть не может.

Это не определение препроцессора — значение **msglen** обрабатывается здесь только один раз при помощи значения **$**.

TIMES: Повторение инструкций или данных

Префикс **TIMES** заставляет инструкцию ассемблироваться несколько раз.

Можно написать, например

```
     zerobuf: times 64 db 0
```

или что-то подобное.

Аргумент **TIMES** — не просто числовая константа, а числовое выражение, поэтому можно писать следующее:

```
buffer:               db 'Привет!'
    times 64-$+buffer db ' '
```

При этом будет резервироваться строго определенное пространство, начиная от метки **buffer** и длиной 64 байта.

TIMES может использоваться в обычных инструкциях — это позволяет писать тривиальные развернутые циклы:

```
times 100 movsb
```

Нет никакой принципиальной разницы между **times 100 resb 1** и **resb 100** за исключением того, что последняя инструкция будет обрабатываться примерно в 100 раз быстрее из-за внутренней структуры ассемблера.

TIMES не может использоваться в макросах.

Эффективные адреса (EA)

Эффективный адрес — это любой операнд инструкции, которая ссылается на память.

Эффективные адреса в NASM имеют очень простой синтаксис — они состоят из выражения, вычисляющего желаемый адрес, заключенного в квадратные скобки. Например:

```
section .data
wordvar dw      123                     ; объявление инициализированных данных
        dw      456
section .text
        mov     ax, [wordvar]
        mov     ax, [wordvar + 2]
        mov     ax, [es:wordvar + bx]
```

Любая другая ссылка, не соответствующая данному синтаксису, недействительна:

```
mov ax, es:wordvar[bx]
```

Более сложные эффективные адреса, когда в вычисление EA вовлечено более одного регистра, имеют аналогичный синтаксис:

```
        mov eax, [ebx * 2 + ecx + offset]
        mov ax,  [bp + di + 8]
```

NASM воспринимает алгебру таких выражений и правильно транслирует выражения, выглядящие на первый взгляд недопустимыми с точки зрения Intel ISA:

```
        mov eax, [ebx * 5]              ; ассемблируется как [ebx * 4 + ebx]
        mov eax, [label1 * 2 – label2]  ; то есть [label1 + (label1 - label2)]
```

Некоторые варианты эффективных адресов имеют более одной ассемблерной формы и в большинстве таких ситуаций NASM будет генерировать самую короткую из них.

Например, у нас имеются простые ассемблерные инструкции

```
mov eax, [eax * 2 + 0]
mov eax, [eax + eax]
```

NASM будет генерировать последнюю из них, т.к. первый вариант требует дополнительно 4 байта для хранения нулевого смещения.

NASM имеет механизм, позволяющий создавать из **[eax + ebx]** и **[ebx + eax]** разные инструкции, например

[esi + ebp] (ES/DS)

и

[ebp + esi] (SS)

будут использовать разные сегментные регистры по умолчанию.

Тем не менее, можно заставить NASM генерировать требуемые формы эффективных адресов при помощи ключевых слов

BYTE, WORD, DWORD и NOSPLIT

Если требуется, чтобы **[eax + 3]** ассемблировалась со смещением размером в двойное слово, вместо одного байта по умолчанию, следует написать **[dword eax + 3]**.

Аналогичным образом используя форму **[byte eax + offset]** можно заставить NASM использовать смещение размером в один байт для небольшого значения, которое он не увидел на первом проходе.

```
        mov eax, [ebx + offset]       ; на offset будет отведено 4 байта
offset  equ 10                        ; на самом деле offset занимает 1 байт
```

Проблема решается либо явным указанием размера:

```
        mov eax, [byte ebx + offset]  ; на offset будет отведен 1 байт
```

либо объявлением смещения **offset** перед его использованием:

```
offset  equ 10
        mov eax, [ebx + offset]       ; на offset будет отведен 1 байт
```

В особых случаях, **[byte eax]** будет кодироваться как **[eax + 0]** с нулевым байтовым смещением, а **[dword eax]** будет кодироваться с нулевым смещением в двойное слово.

Обычная форма, **[eax]**, будет оставлена без смещения.

```
15 00000010 8B00              mov     eax, [ebx]
16 00000012 8B4000            mov     eax, [byte ebx]
17 00000015 8B8000000000      mov     eax, [dword ebx]
```

Также форма **`[тип регистр]`**, описанная выше, полезна, если необходимо получить доступ к данным в 32-битном сегменте из 16-битного кода (адресация смешанного размера). Это случается, если необходимо получить доступ к данным с известным смещением, которое больше, чем уместится в 16-битном значении. Если не указать, что это смещение размером в двойное слово, nasm старшее слово смещения потеряет.

NASM также разделит **`[eax * 2]`** на **`[eax + eax]`**, потому что это позволяет исключить поле смещения из инструкции и сэкономить место.

Точно так же он также разделит **`[eax * 2 + offset]`** на **`[eax + eax + offset]`**.

NOSPLIT

Можно воспрепятствовать такому поведению, используя ключевое слово **NOSPLIT**, которое заставляет генерировать код эффективного адреса строго в той форме, которая указана.

[nosplit eax * 2] сгенерирует буквально **[eax * 2 + 0]**.

[nosplit eax * 1] приведет точно к такому же эффекту.

```
19 0000001B 8B0400                  mov     eax, [eax * 2]
20 0000001E 8B044500000000          mov     eax, [nosplit eax * 2]
21 00000025 8B040500000000          mov     eax, [nosplit eax * 1]
```

Другой способ состоит в том, чтобы использовать расщепление в форме **[0, eax * 2]**.

Однако **NOSPLIT** в **[nosplit eax + eax]** будет проигнорирован, потому что здесь усматривается замысел пользователя, как **[eax + eax]**.

В 64-битном режиме NASM по умолчанию генерирует абсолютные адреса. Ключевое слово **REL** заставляет его создавать адреса относительно **RIP**.

Так как это обычно желаемое поведение, существует директива **DEFAULT**, с помощью которой можно устанавить желаемый режим адресации. Ключевое слово **ABS** имеет приоритет над **REL**.

Также поддерживается новая форма синтаксиса разделения эффективных адресов. Она, в основном, предназначено для mib-операндов[1], которые используются в инструкциях **MPX**[2], но может использоваться для любой ссылки на память.

Основная идея этой формы — разделение базы и индекса.

```
mov eax, [ebx + 8, ecx * 4] ; ebx = base, ecx = index, 4 = scale, 8 = disp
```

Для mib-операндов существует несколько способов записи эффективного адреса в зависимости от используемого инструментария. NASM поддерживает все возможные на данный момент способы синтаксиса для mib-операндов:

```
; bndstx -- Store Extended Bounds Using Address Translation
; next 5 lines are parsed same
; base = rax, index = rbx, scale = 1, displacement = 3
bndstx [rax + 0x3, rbx], bnd0      ; NASM - split EA
bndstx [rbx * 1 + rax + 0x3], bnd0 ; GAS - '*1' indecates an index reg
bndstx [rax + rbx + 3], bnd0       ; GAS - without hints
bndstx [rax + 0x3], bnd0, rbx      ; ICC-1
bndstx [rax + 0x3], rbx, bnd0      ; ICC-2
```

Когда используется широковещательный декоратор, ключевое слово **opsize** должно соответствовать размеру каждого элемента.

```
VDIVPS zmm4, zmm5, dword [rbx]{1to16}   ; single-precision float
VDIVPS zmm4, zmm5, zword [rbx]          ; packed 512 bit memory
```

1 mib — Memory-Index-Base (SIB)

2 MPX (Memory Protection Extensions) — расширения защиты памяти Intel. Нынче не используется, поскольку никаких преимуществ, в том числе и в производительности, перед программными проверками не дает. Не поддерживается gcc с версии 9.0.

Константы

NASM знает четыре различных типа констант:

- числовые;
- символьные;
- строковые;
- с плавающей точкой.

Числовые константы

Числовая константа — это просто число.

Числа могут определяться в различных системах счисления и различным образом:

- можно использовать суффиксы **H**, **Q** и **B** для шестнадцатеричных, восьмеричных и двоичных чисел соответственно;
- можно использовать для шестнадцатеричных чисел префикс 0x в стиле C;
- можно использовать префикс $ в стиле Borland Pascal (!!! возможен конфликт с метками !!!).
- можно использовать (_) для более лучшего восприятия длинных строк цифр

Некоторые примеры числовых констант:

```
mov  ax, 100            ; десятичная
mov  ax, 0a2h           ; шестнадцатеричная
mov  ax, $0a2           ; снова hex: нужен 0
mov  ax, 0xa2           ; опять hex
mov  ax, 777q           ; восьмеричная
mov  ax, 10010011b      ; двоичная
mov  ax, 1001_0011b     ; двоичная, то же самое, что и выше
```

Символьные константы

Символьная константа содержит от одного до четырех символов, заключенных в одиночные или двойные кавычки.

Тип кавычек для NASM несущественен, поэтому если используются одинарные кавычки, двойные могут выступать в роли символа и, соответственно, наоборот.

Символьная константа, содержащая более одного символа, будет загружаться в обратном порядке следования байт — в строке

```
mov eax,'abcd'
```

загружаться будет не **0x61626364**, а **0x64636261**.

Если сохранить эту константу в память, а затем прочитать, получится снова **abcd**, но никак не **dcba**.

Строковые константы

Строковые константы допустимы только в некоторых псевдо-инструкциях, а именно в семействе **DB**, и инструкции **INCBIN**. Строковые константы похожи на символьные, только длиннее.

Строковые константы обрабатываются как сцепленные друг с другом символьные константы.

Так, например, следующие строки кода эквивалентны:

```
    db 'hello'                  ; строковая константа
    db 'h', 'e', 'l', 'l', 'o'  ; эквивалент из символьных констант
```

Следующие строки также эквивалентны:

```
    dd 'ninechars'              ; строковая константа в двойное слово
    dd 'nine','char', 's'       ; три двойных слова
    db 'ninechars', 0, 0, 0     ; и действительно похоже
```

Следует учитывать, что когда используется **db**, константа типа **'ab'** обрабатывается как строковая, хотя и достаточно коротка, чтобы быть символьной, потому что иначе **db 'ab'** имело бы тот же смысл, какой и **db 'a'**, т.е. байт **'b'** пропал бы.

Соответственно, трех- или четырехсимвольные константы, являющиеся операндами инструкции **dw**, обрабатываются также как строки.

Строки Unicode

Специальные операторы позволяют определять строки Unicode:

```
__utf16__, __utf16le__, __utf16be__,
__utf32__, __utf32le__, __utf32be__
```

Они принимают строки в формате UTF-8 и преобразуют их в форматы UTF-16 или UTF-32, соответственно, в формах **littleendian (le)** или **bigendian (be)**.

Если форма не указана явно, результат будет **littleendian**.

Например:

```
%define u(x) __utf16__(x)
%define w(x) __utf32__(x)
```

```
      dw u('C:\WINDOWS'), 0        ; Pathname in UTF-16
      dd w(`A + B = \u206a`), 0    ; String in UTF-32
```

Константы с плавающей точкой

Константы с плавающей точкой допустимы только в качестве аргументов псевдоинструкций объявления данных **DD**, **DQ** и **DT** размером в 4, 8 и 10 байт, соответственно.

Выражаются они традиционно — цифры, затем точка, затем возможно цифры после точки, и наконец, необязательная E с последующей степенью.

Точка обязательна, поскольку **dd 1** будет воспринято, как объявление целой константы, в то время как **dd 1.0** будет воспринято правильно.

Несколько примеров:

```
    dd 1.2                          ; просто число
    dq 1.e10                        ; 10,000,000,000
    dq 1.e+10                       ; синоним 1.e10
    dq 1.e-10                       ; 0.000 000 000 1
    dt 3.141592653589793238462      ; число pi (80 бит)
    dt 3.141_592_653_589_793_238_462 ; число pi (80 бит) — удобство восприятия
```

В процессе компиляции NASM не может проводить вычисления над константами с плавающей точкой (это сделано с целью переносимости).

```
9 00000056 9A99993F                dd  1.2                          ; просто число
10 0000005A 000000205FA00242       dq  1.e10                        ; 10,000,000,000
11 00000062 000000205FA00242       dq  1.e+10                       ; синоним 1.e10
12 0000006A BBBDD7D9DF7CDB3D       dq  1.e-10                       ; 0.000 000 000 1
13 00000072 35C26821A2DA0FC900-    dt  3.141592653589793238462      ; число pi (80 бит)
13 0000007B 40
14 0000007C 35C26821A2DA0FC900-    dt  3.141_592_653_589_793_238_462 ; число pi (80
14 00000085 40
```

Упакованные BCD константы

Упакованные BCD-константы в стиле x87 могут использоваться в тех же контекстах, что и 80-битные числа с плавающей запятой. Они имеют суффикс **p** или префикс **0p** и могут включать до 18 десятичных цифр.

Как и в других числовых константах, для разделения цифр можно использовать подчеркивание. Например:

```
      dt    12_345_678_901_245_678p
      dt    -12_345_678_901_245_678p
      dt    +0p33
      dt    33p
```

```
16 00000086 7856240189674523 01-      dt   12_345_678_901_245_678p
16 0000008F 00
17 00000090 7856240189674523 01-      dt   -12_345_678_901_245_678p
17 00000099 80
18 0000009A 3300000000000000 00-      dt   +0p33
18 000000A3 00
19 000000A4 3300000000000000 00-      dt   33p
19 000000AD 00
```

Выражения

Синтаксис выражений NASM подобен синтаксису выражений языка C.

NASM не гарантирует размер целых чисел, используемых для вычисления выражений при компиляции, поэтому не следует полагать, что выражения вычисляются регистрах определенного размера.

NASM гарантирует только то, что и ANSI C — выражения вычисляются, как минимум, с 32-битными регистрами.

В выражениях NASM поддерживает два специальных символа **`$`** и **`$$`**, позволяющих при вычислениях выражений получать текущую позицию (смещение) ассемблирования:

Символ **`$`** вычисляет позицию начала строки, содержащей выражение, т.е. можно сделать бесконечный цикл при помощи команды **`JMP $`**;

Символ **`$$`** определяет начало текущей секции (сегмента), поэтому чтобы узнать, как далеко мы находимся от начала секции, можно использовать выражение (**`$-$$`**);

Арифметические операторы, предоставляемые NASM, перечислены далее в порядке возрастания приоритета.

Логическое значение истинно, если оно не равно нулю, и ложно, если оно равно нулю. Операторы, возвращающие логическое значение, всегда возвращают 1 истинного и 0 для ложного.

Логические операторы

`?...:...` — условный оператор;
Синтаксис этого оператора аналогичен синтаксису условного оператора C:

boolean `?` *trueval* `:` *falseval*

Этот оператор возвращает значение **`trueval`**, если логическое значение *boolean* равно **`true`**, в противном случае **`falseval`**.

Следует помнить, что NASM позволяет использовать символ `?' в именах символов. Поэтому настоятельно рекомендуется всегда оставлять пробелы вокруг символов `?' и `:'.

`||` — логический OR;
`^^` — логический XOR;
`&&` — логический AND.

Операторы сравнения

= или **==** — сравнение на равенство
!= или **<>** — сравнение на неравенство
< — сравнение на меньше
<= — сравнение на меньше или равно (не больше)
> — сравнение на больше
>= — сравнение на больше или равно (не меньше)

Эти операторы выдают значение 0 для ложных или 1 для истинных выражений.

<=> — выполняет сравнение со знаком и принимает значение **–1**, если меньше чем, **0** — равно и **1** — больше.

Бинарные операторы

`|` — побитовый оператор ИЛИ;
`^` — побитовый оператор ИСКЛЮЧАЮЩЕЕ ИЛИ;
`&` — побитовый оператор И;
`<<` и `>>` — операторы беззнакового сдвига бит. Работают, как в C (5<<3 — как умножение на 8);
`<<<` и `>>>` — операторы знакового сдвига бит.;
`+` и `–` операторы сложения и вычитания;
`*` — оператор умножения;
`/` и `//` — беззнаковое и знаковое деление;
`%` и `%%` — беззнаковое и знаковое получение остатка от деления (взятие по модулю).

Поскольку символ '%' часто используется макропроцессором, следует быть особо внимательным при применении знакового и беззнакового операторов взятия по модулю — они должны отделяться от других символов строки по крайней мере одним пробелом.

NASM, как и ANSI C, не дает никаких гарантий относительно разумной работы знакового оператора получения остатка от деления (взятия по модулю).

В большинстве систем он будет соответствовать знаковому оператору деления, например:

```
b * (a // b) + (a %% b) = a              (b! = 0)
```

Унарные операторы

- — оператор "минус"изменяет знак своего операнда:

+ — оператор "плюс" ничего не делает (введен для симметричности с минусом);

~ — оператор "тильда" вычисляет побитовую инверсию операнда;

SEG — извлекает сегментный адрес операнда (16-битный код).

STRICT: Подавление оптимизации

При ассемблировании с оптимизацией, установленной на уровень 2 или выше, NASM будет использовать спецификаторы размера (**BYTE**, **WORD**, **DWORD**, **QWORD**, **TWORD**, **OWORD**, **YWORD** или **ZWORD**), но предоставит им минимально возможный размер.

Чтобы запретить оптимизацию и принудительно выдать определенный операнд в указанном размере, можно использовать ключевое слово **STRICT**. Например, при включенном оптимизаторе в режиме **BITS 16**

```
        push dword 33
```

будет кодироваться тремя байтами **66 6A 21**, в то время как

```
        push strict dword 33
```

будет кодироваться шестью байтами **66 68 21 00 00 00** с непосредственным операндом размером 4 байта. При выключенной оптимизации генерируется один и тот же код (шесть байтов) независимо от того, использовалось ключевое слово **STRICT** или нет.

Критические выражения

NASM является двухпроходным ассемблером и всегда делает только два прохода.

Из-за этого он не может обрабатывать сложные исходные файлы, требующими три и более проходов.

Первый проход помимо прочего используется для назначения адресов всем символам, а также для определения размера всех ассемблируемых инструкций и данных;

Второй проход используется для генерации кода.

На втором проходе известны адреса всех символов, на которые имеются ссылки. Поэтому NASM не сможет обработать код, в котором размер зависит от значения символа, объявленного позднее, например:

```
times (label-$) db 0
label:          db 'Какой-то текст'
```

Аргумент префикса **`TIMES`** (**`label-$`**) должен точно рассчитываться для всех меток, поэтому NASM воспримет этот исходный код ошибочным, поскольку узнать размер строки в данном случае невозможно.

NASM отклоняет такой код при помощи концепции т.н. **критического выражения**.

Критическое выражение — выражение, значение которого должно быть рассчитано на первом проходе и которое, следовательно, должно зависеть только от символов, описанных перед ним.

Аргумент префикса **`TIMES`** является критическим выражением.

Аргументы псевдо-инструкций семейства **`RESB`** также являются критическими выражениями.

Проблема, связанная с опережающими ссылками:

```
          mov eax, [ebx + offset]
offset    equ 10
```

Длину инструкции **`mov eax,[ebx+offset]`** NASM должен вычислить на первом проходе, еще до того, как станет известно значение **`offset`**. Это значение представляет собой малую величину, которая вписывается в однобайтное поле смещения.

Поскольку на первом проходе еще не известно, что такое **`offset`** (это может быть символ в сегменте кода), поэтому NASM использует полную четырехбайтовую форму инструкции и ее размер рассчитывается исходя из четырехбайтовой адресной части.

Сделав это предположение, на втором проходе NASM вынужден оставлять длину инструкции как есть, генерируя при этом не совсем оптимальный код.

Данная проблема может быть разрешена путем объявления **`offset`** перед ее первым использованием:

```
offset    equ 10
          mov eax, [ebx + offset]
```

или явным указанием на байтовый размер смещения:

```
          mov eax, [byte ebx + offset]
offset    equ 10
```

Локальные метки

Метка, начинающаяся с точки, обрабатывается как локальная. Это означает, что она неразрывно связана с предыдущей нелокальной меткой. Например:

```
label1
     ...         ; некоторый код
.loop
     ...         ; еще какой-то  код
     jne .loop
     ret
label2
     ...         ; некоторый код
.loop
     ...         ; еще какой-то  код
     jne .loop
     ret
```

В приведенном фрагменте каждая инструкция **JNE** переходит на метку **.loop**, расположенную непосредственно перед ней, т.к. два определения **.loop** разделены нелокальной меткой **label2**.

Первая локальная метка **.loop** связана с нелокальной меткой **label1**, а вторая локальная метка **.loop** — с нелокальной меткой **label2**.

Вышеприведенный способ обработки локальных меток является классическим, однако NASM позволяет обращаться к локальным меткам из другой части кода.

Это достигается путем описания локальной метки на основе предыдущей нелокальной.

Описания **.loop** в приведенном выше примере в действительности описывают два разных символа — **label1.loop** и **label2.loop**, поэтому можно написать даже так:

```
label1
        ...
.loop
        ...         ; некоторый код
label3              ; "не виртова" точка входа в цикл
        ...         ; некоторый код
        jmp label1.loop
```

Иногда бывает полезно, например, в макросах, определить метку, на которую можно ссылаться отовсюду, но которая не пересекается с обычным механизмом локальных меток.

Такая метка не может быть нелокальной, поскольку она будет мешать последующим определениям и ссылкам на локальные метки.

Такая метка также не может быть и локальной, потому что макрос, который ее определил, не будет знать полного имени метки.

Поэтому в NASM вводится третий тип меток, который, скорее всего, полезен только в определениях макросов — если метка начинается со специального префикса `..@`, то он ничего не делает с механизмом локальных меток.

Таким образом, можно написать:

```
label1:             ; нелокальная метка
.local:             ; это label1.local
..@foo:             ; это специальный символ
label2:             ; другая нелокальная метка
.local:             ; это label2.local
        jmp ..@foo ; переход на три строки вверх
```

NASM имеет возможность определять другие специальные символы, начинающиеся с двух точек: например, **`..start`** используется для указания точки входа в объектном формате obj.

Операции NASM в порядке возрастания приоритета

| — побитовый оператор ИЛИ

^ — побитовый оператор ИСКЛЮЧАЮЩЕЕ ИЛИ

& — побитовый оператор И

<<, <<< и >>, >>> — операторы сдвига бит

+ и — Операторы сложения и вычитания

*, /, //, % и %%: Умножение и деление

Унарные операторы: +, -, ~ и оператор SEG

Наивысший приоритет в грамматике выражений NASM имеют операторы, применяемые к одному аргументу:

Инструкции сопроцессора

Для инструкций сопроцессора NASM допускает различные формы синтаксиса.
- двух-операндная форма, поддерживается MASM'ом;
- одно-операндная форма NASM'а.
Например, можно написать:

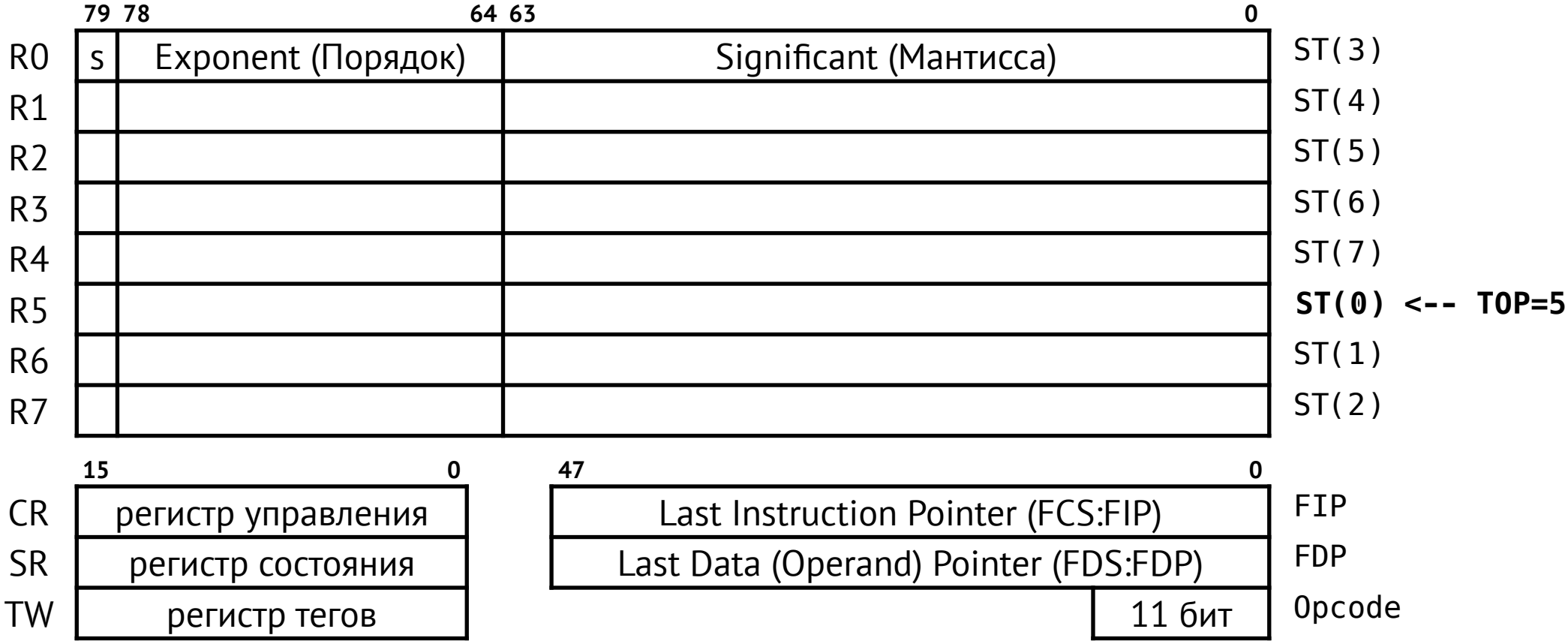
```
        fadd st1                ; это значит st0 := st0 + st1
        fadd st0, st1           ; это то же самое

        fadd st1, st0           ; это значит st1 := st1 + st0
        fadd to st1             ; это то же самое
```

Почти любая инструкция сопроцессора, ссылающаяся на содержимое памяти, должна использовать один из префиксов **DWORD**, **QWORD** или **TWORD** для указания на то, операнд какого размера должен участвовать в команде.

Регистры FPU (справка)

FPU содержит восемь регистров для хранения данных (bin, bcd) и пять (+1) вспомогательных.



Регистры данных (R0 — R7) — не адресуются по именам, а рассматриваются в качестве регистрового стека, вершина которого называется ST(0), более глубокие элементы — ST(1)..ST(7). Регистры организованы в виде кольца.