Арифметика в NASM

Recap: Простейшие арифметические операции

сложение: ADD <OP1> <OP2>

вычитание: SUB <OP1> <OP2>

унарное отрицание: NEG <OP>

Recap: Простейшие арифметические операции

сложение: ADD <OP1> <OP2>

вычитание: SUB <OP1> <OP2>

унарное отрицание: NEG <OP>

Какие ограничения наложены на OP1, OP2 и почему?

Recap: Простейшие арифметические операции

сложение: ADD <OP1> <OP2>

вычитание: SUB <OP1> <OP2>

унарное отрицание: NEG <OP>

Какие ограничения наложены на OP1, OP2 и почему?

- OP1, OP2 должны быть одинакового размера.

Recap: Простейшие арифметические операции

сложение: ADD <OP1> <OP2>

вычитание: SUB <OP1> <OP2>

унарное отрицание: NEG <OP>

Какие ограничения наложены на OP1, OP2 и почему?

- OP1, OP2 должны быть одинакового размера.
- Оба операнда не могут быть памятью одновременно.

Recap: Простейшие арифметические операции

сложение: ADD <OP1> <OP2>

вычитание: SUB <OP1> <OP2>

унарное отрицание: NEG <OP>

Какие ограничения наложены на OP1, OP2 и почему?

- OP1, OP2 должны быть одинакового размера.
- Оба операнда не могут быть памятью одновременно. Но почему?

Recap: Простейшие арифметические операции

сложение: ADD <OP1> <OP2>

вычитание: SUB <OP1> <OP2>

унарное отрицание: NEG <OP>

Какие ограничения наложены на OP1, OP2 и почему?

- OP1, OP2 должны быть одинакового размера.
- Оба операнда не могут быть памятью одновременно.

- Дело в кодировании команд: **операнд с памятью занимает больше места в байткоде**, чем любой другой. Если у команды будет два таких операнда – ее **байткод будет слишком большой**

За гранью суммы

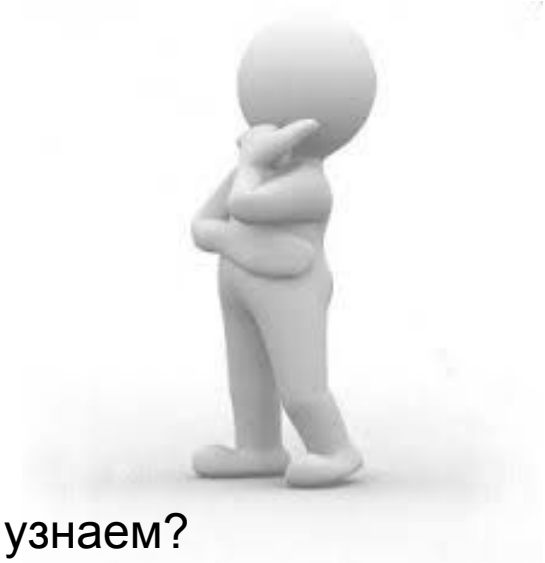
Каков результат выполнения следующей программы?

```
1
2       mov eax, 0xFFFFFFFF
3       add eax, 1
4
```

За гранью суммы

Каков результат выполнения следующей программы?

```
1
2       mov eax, 0xFFFFFFFF
3       add eax, 1
4
```



Очевидно, произойдет переполнение. Но как мы об этом узнаем?

За гранью суммы

Каков результат выполнения следующей программы?

```
1
2       mov eax, 0xFFFFFFFF
3       add eax, 1
4
```

Очевидно, произойдет переполнение. Но как мы об этом узнаем?

Спасибо создателям x86 CPU! У нас есть специальный регистр EFLAGS!

За гранью суммы

EFLAGS – это специальный регистр, который хранит в себе 4 бита-флага:

- **ZF** – Zero-flag. Выставлен в 1, если результат последней операции был 0
- **SF** – Sign-flag. Выставлен в 1, если результат последней операции отрицательный
- **CF** – Carry-flag. Выставлен в 1, если в последней операции произошло беззнаковое переполнение.
- **OF** – Overflow-flag. Выставлен в 1, если в последней операции произошло знаковое переполнение.

На флаги влияют не любые команды.

Пока мы считаем, что флаги меняются только после арифметических операций (на самом деле не только)

За гранью суммы

Как поменяются флаги?

```
mov eax, 0x7FFFFFFF
add eax, 1
```

ZF = ? CF = ?

SF = ? OF = ?

```
mov eax, 0xFFFFFFFF
add eax, 1
```

ZF = ? CF = ?

SF = ? OF = ?

За гранью суммы

Как поменяются флаги?

```
mov eax, 0x7FFFFFFF
add eax, 1
```

ZF = 0 CF = 0

SF = 1 OF = 1

```
mov eax, 0xFFFFFFFF
add eax, 1
```

ZF = 1 CF = 1

SF = 0 OF = 0

За гранью суммы

Произошло знаковое переполнение. Число было положительным, а стало отрицательным. Значит выставим флаг отриц. результата SF, флаг знакового переполнения OF

```
mov eax, 0x7FFFFFFF
add eax, 1
```

ZF = 0 CF = 0

SF = 1 OF = 1

Произошло беззнаковое переполнение. Результат стал 0. Значит выставим флаг нуля ZF, флаг переполнения CF

```
mov eax, 0xFFFFFFFF
add eax, 1
```

ZF = 1 CF = 1

SF = 0 OF = 0

За гранью сум

Произошло знаковое пе ало отрицательным.
Значит выставим флаг с ения OF

```
mov eax, 0x7
add eax, 1
```

CF = 0

OF = 1

Произошло беззнаково выставим флаг нуля
ZF, флаг переполнени

```
mov eax, 0xF
add eax, 1
```

CF = 1

OF = 0

За гранью вычитания

Как поменяются флаги?

```
mov eax, 0x80000000
sub eax, 1
```

ZF = ? CF = ?

SF = ? OF = ?

```
mov eax, 0
sub eax, 1
```

ZF = ? CF = ?

SF = ? OF = ?

За гранью вычитания

Произошло знаковое переполнение. Результат был отрицательным, стал положительным.

```
mov eax, 0x80000000
sub eax, 1
```

ZF = 0 CF = 0

SF = 0 OF = 1

Произошло беззнаковое переполнение. Мы хотим “занять” 1, чтоб вычесть, поэтому выставляем CF (carry flag)

```
mov eax, 0
sub eax, 1
```

ZF = 0 CF = 1

SF = 1 OF = 0

Работа с флагами

Группа команд **set* <OP>** выставляет операнд <OP> в 1, если выставлен соответствующий флаг

Много других команд ориентируются на флаги, например JZ, но о них мы поговорим позже

```
setz byte [zf_flag]
sets byte [sf_flag]
setc byte [cf_flag]
seto byte [of_flag]
```

Функции и метки

Научившись складывать 2 + 2, перейдем к вещам для взрослых.

Функция – это просто метка, заканчивающаяся **ret**

Вызов функции – через **call function**

Как CPU понимает, куда возвращаться при ret – **магия!**


```
save_flags:
    setz byte [zf_flag]
    sets byte [sf_flag]
    setc byte [cf_flag]
    seto byte [of_flag]

    ret
```

объявление

```
print_flags:
    call save_flags
```

вызов из другой функции

Функции и метки

Научившись складывать 2 + 2, перейдем к вещам для взрослых.

Функция – это просто метка, заканчивающаяся **ret**

Вызов функции – через **call function**

Но где тут аргументы?


```
save_flags:
    setz byte [zf_flag]
    sets byte [sf_flag]
    setc byte [cf_flag]
    seto byte [of_flag]

    ret
```

объявление

```
print_flags:
    call save_flags
```

вызов из другой функции

Функции и метки

Аргументы передаются через регистры.

Как именно – договоренность.

Поэтому **важно писать док-стринги** – краткие описания функций, в которых говорится о том, что функция ждет и в каких регистрах и какие регистры после нее замусорены

```
save_flags:
    setz byte [zf_flag]
    sets byte [sf_flag]
    setc byte [cf_flag]
    seto byte [of_flag]

    ret
```

объявление

```
print_flags:
    call save_flags
```

вызов из другой функции

Функции и метки

Докстринги – **обязательное требование к коду в ДЗ**.

Формат докстрингов:

1. Название функции
2. Описание функции
3. Что и в каких регистрах функция ожидает
4. Какие регистры функция портит
5. Что и в каких регистрах функция возвращает

```
; ------------------------------------------------
; print_flags                                    1. Название функции
;
; prints flags ZF, SF, CF, OF                    2. Описание функции
;
; EXPECTS:  EAX -- header message                3. Что и в каких регистрах функция ожидает
;
; DESTROYS: EAX  4. Какие регистры функция портит
;
;
; RETURNS:  None 5. Что и в каких регистрах функция возвращает
; ------------------------------------------------
print_flags:
    call save_flags

    call io_print_string
    call io_newline

    print_one_flag zf_msg, [zf_flag]
    print_one_flag sf_msg, [sf_flag]
    print_one_flag cf_msg, [cf_flag]
    print_one_flag of_msg, [of_flag]

    call io_newline
    ret
```

Серьезные вещи

- Мы вспомнили сложение
- Узнали про вычитани
- Мы разобрались с флагами
- Научились писать функции и документировать их

Теперь мы готовы к по-настоящему серьезным вещам

Серьезные вещи. Умножение

Серьезные вещи. Умножение

MUL <REG / MEM> – умножает EAX на свой операнд и записывает результат в EDX:EAX

То есть имеем:
- один операнд явный
- второй – всегда регистр EAX / AX / AL

Результат ложится в два регистра разом:
- Нижняя половина в EAX / AX
- Верхняя половина в EDX / DX

MUL – беззнаковое умножение
IMUL – знаковое умножение

Серьезные вещи. Умножение

Размер	**Явный множитель,**	**Неявный множитель**	**Результат**
8 бит	reg / 8-bit mem	AL	**AX** = AL * op8
16 бит	reg / 16-bit mem	AX	**DX:AX** = AX * op16
32 бит	reg / 32-bit mem	EAX	**EDX:EAX** = EAX * op32

Серьезные вещи. Умножение

Умножаем 0xFFFFFFFF * 2 = 0x1FFFFFFFE

0x1FFFFFFFE – не влезет в один регистр, поэтому нужно два:
EAX = 0xFFFFFFFE (нижние 4 байта)
EDX = 0x1 (верхние 4 байта)
EDX:EAX = 0x1FFFFFFFE

```
; ----------------------------------------
; 32-bit MUL example
; ----------------------------------------
mov eax, 0xFFFFFFFF     ; multiplier-1
mov ebx, 2              ; multiplier-2

mul ebx                 ; unsigned multiply: EDX:EAX = EAX * EBX
```

Серьезные вещи. Деление

Серьезные вещи. Деление

DIV <REG / MEM> – делит EDX:EAX на свой операнд. В EAX кладет частное, в EDX – остаток

То есть имеем:

- делимое – EDX:EAX
- делитель – всегда регистр EAX / AX / AL

Результат ложится в два регистра разом:

- частное в EAX
- остаток в EDX

DIV – беззнаковое деление
IDIV – знаковое

Серьезные вещи. Деление

Размер	**Делимое**	**Делитель**	**Результат:** частное остаток
8 бит	AX	reg / 8-bit mem	**AL** = AX / op8 **AH** = AX % op8
16 бит	DX:AX	reg / 16-bit mem	**AX** = DX:AX / op16 **DX** = DX:AX % op16
32 бит	EDX:EAX	reg / 32-bit mem	**EAX** = EDX:EAX / op32 **EDX =** EDX:EAX % op32

Серьезные вещи. Деление

Делим -20 / 3 = -3, остаток -1

IDIV ожидает делимое в EDX:EAX, надо знаково расширить EAX до EDX:EAX – для этого есть **CDQ** – Convert Dword to Qword

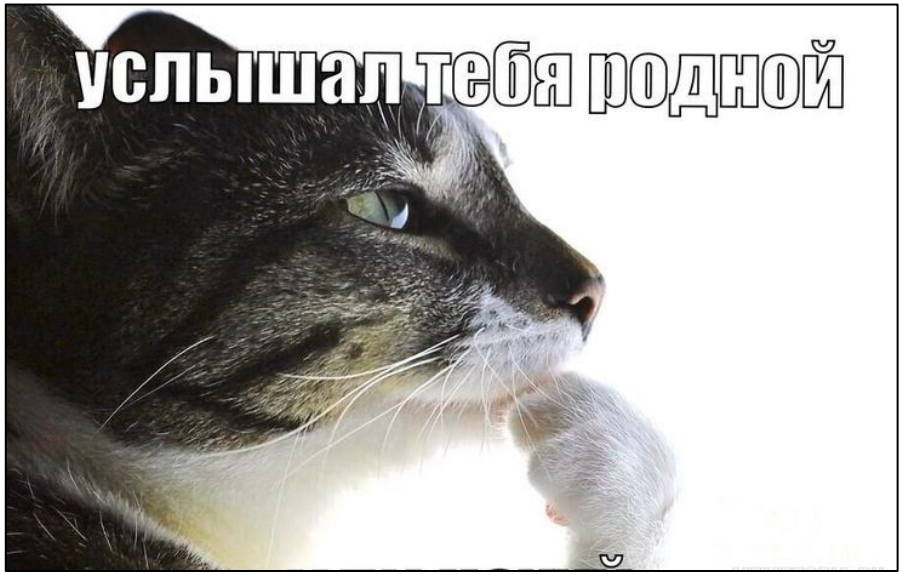
Других приколов нету

```
; ================================
; IDIV (signed)
; ================================
mov eax, msg_idiv
call io_print_string
call io_newline

mov eax, -20            ; 0xFFFFFFF6
cdq                     ; convert DW to QW: sign-extend EAX into EDX:EAX
mov ebx, 3

idiv ebx                ; EAX = -3, EDX = -1
```

Рекап

Сегодня:

- вспомнили сложение
- узнали про вычитание
- испугались умножения
- деление??



- **услышали** что я **не приму** код **без докстрингов** у функций
- выучили все флаги арифметических операций

Вопросы?