??????????

Арифметика и логика

Mov копирование значений.

```
mov destination, source
```

destination: регистр или память. Source: регистр, память, число. Одновременно не может из памяти в память. Должны совпадать по размерам. Для расширения нулями меньших регистров (только регистр - регистр):

```
movsxd dest, source     ; если dest - 64-разрядный операнд и source - 32-разрядный
movsx dest, source      ; для всех остальных комбинаций операндов
```

Однако с знаком будут проблемы. Для беззнакового расширения нулями movzx:

```
mov al, 5
movzx rdi, cx
```

Есть относительная адресация, то есть

```
mov eax, [ebx + 8]      ; EAX = значение по адресу EBX + 8 Надо разобраться
mov [esi + ecx*4], edx  ; Записать EDX по адресу ESI + ECX*4
```

Однако разность [esi - 4] может не работать.

Значение регистра AL помещается в самый младший байт регистра RDI. Остальные байты (7 байт) регистра RDI заполняются нулями. Если не сделать заполнение нулями, то в старший из 8 байт попадет байт аргумента, дальше - некое непотребство.

```
movzx rsi, byte [cura] ; если cura это байт
```

lea загрузка/вычисление адреса. Есть адресная арифметика.

Add / sub сложение и вычитание

```
add operand1, operand2  ; operand1 = operand1 + operand2
inc rdi    ; rdi = rdi + 1
```

```
sub rdi, rsi    ; rdi = rdi - rsi
dec rdi    ; rdi = rdi - 1
```

mul и imul умножает два целых числа. imul умножает числа со знаком, а mul - беззнаковые числа. Обе инструкции принимают один операнд - регистр или адрес в памяти, который умножается на значение в регистре RAX. Результат помещается в регистры RAX/RDX

```
mul operand8   ; если операнд 8-разрядный, результат в AX
mul operand16  ; если операнд 16-разрядный, результат в DX:AX
mul operand32  ; если операнд 32-разрядный, результат в EDX:EAX
mul operand64  ; если операнд 64-разрядный, результат в RDX:RAX
```

В AX/EAX/RAX помещается младшая часть результата, а в DX/EDX/RDX - старшая.

```
global _start

section .text
_start:
    mov rdi, 2
    mov rax, 4
    mul rdi          ; RAX = RAX * RDI
    mov rdi, rax     ; RDI = RAX = 8
    mov rax, 60
    syscall
```

div и idiv. idiv делит два числа со знаком, а div - беззнаковые числа.

- Если операнд 8-разрядный, то div делит регистр AX на операнд, помещая частное в AL, а остаток (по модулю) в AH.
- Если операнд 16-разрядный, то инструкция div делит 32-разрядное число в DX:AX на операнд, помещая частное в AX, а остаток в DX.
- Если операнд 32-разрядный, div делит 64-битное число в EDX:EAX на операнд, помещая частное в EAX, а остаток в EDX.
- И если операнд 64-разрядный, div делит 128-битное число в RDX:RAX на операнд, помещая частное в RAX, а остаток в RDX.

```
global _start

section .text
_start:
    mov rax, 0  ; обнуляем регистр
    mov ax, 22  ; 16-разрядный регистр
```

```
    mov bl, 5   ; 8-разрядный регистр
    div bl      ; AX/BL = AL =4 (результат), AH = 2 (остаток)
    movzx rdi, al   ; RDI = 4
    mov rax, 60
    syscall
```

При этом в x86-64 нельзя разделить два числа одинаковой разрядности, например, одно 8-разрядное на другое 8-разрядное. Если знаменатель представляет собой 8-битное значение, числитель должен быть 16-битным значением. Если же нужно разделить одно 8-битное значение без знака на другое, то необходимо дополнить числитель нулями до 16 бит, загрузив числитель в регистр AL, а затем переместив 0 в регистр AH. Отсутствие расширения AL до нуля перед выполнением div может привести к тому, что x86-64 выдаст некорректный результат.

- Если нужно разделить два беззнаковых 16-разрядных числа, то надо расширить регистр AX (который содержит числитель) до регистра DX. Для этого достаточно загрузить 0 в регистр DX.
- Если нужно разделить одно беззнаковое 32-битное значение на другое, перед делением надо расширить регистр EAX нулями до EDX (загрузив 0 в EDX).
- И чтобы разделить одно 64-битное число на другое, перед делением нужно расширить RAX нулями до RDX (поместив 0 в RDX).

Логические операции.

and, or, xor, not, neg,

and reg1 reg2 reg1 = reg1 and reg2

Есть сдвиг и вращение

Переходы.

Безусловный переход Регистр rip указывает на адрес памяти, по которому будет выполняться следующая инструкция. Во время выполнения каждой инструкции процессор увеличивает rip, чтобы указывал на следующую.

JMP - безусловный переход.

```
jmp метка
jmp регистр
jmp адрес_в_памяти
```

Переход по метке:

```
global _start
section .text
_start:
    mov rdi, 11         ; RDI = 11
    jmp exit            ; переходим к метке exit
    mov rdi, 22         ; не выполняется
exit:                   ; метка exit
    mov rax, 60         ; 60 - номер системного вызова exit
    syscall
```

Переход по адресу в регистре:

```
global _start
section .text
_start:
    mov rbx, exit       ; в регистр RBX помещаем адрес метки exit
    mov rdi, 22         ; RDI = 22
    jmp rbx             ; переходим к адресу из регистра RBX
    mov rdi, 33         ; не выполняется
exit:                   ; метка exit
    mov rax, 60         ; 60 - номер системного вызова exit
    syscall
```

Переход к адресу в памяти. Переменная должна быть qword, четверичное слово, которое занимает 64 бит.

```
global _start
section .text
_start:
    mov rdi, 23         ; RDI = 23
    jmp [exitPtr]       ; переходим к адресу из exitPtr
    mov rdi, 33         ; не выполняется
exit:                   ; метка exit
    mov rax, 60         ; 60 - номер системного вызова exit
    syscall             ; выполняем системный вызов exit
exitPtr: dq exit      ; переменная exitPtr хранит адрес метки exit
```

Условный переход

В регистре eflags 4 бита используются для проверки состояния исполнения предыдущей команды и перехода. Инструкции, выполняющие математические или логические операции

(add, sub, and, or, xor и not) влияют на установку флагов, а инструкции загрузки данных типа mov или lea не влияют.

Флаг	**Описание**	**Команда**	**Описание**
CF	Флаг переноса. Беззнаковое переполнение (сумма с переносом или вычитании с заимствованием).	jc	переход к метке, если флаг переноса установлен
		jnc	переход, если флаг переноса НЕ установлен
		clc	сброс флага переноса
		setc	установка флага переноса
OF	Флаг переполнения. Переполнение со знаком	jo	переход к метке, если флаг переполнения установлен
		jno	переход к метке, если флаг переполнения не установлен
SF	Флаг знака. Если старший бит результата установлен. То есть флаг знака отражает состояние старшего бита результата.	js	переход к метке, если флаг знака установлен
		jns	переход к метке, если флаг знака не установлен
ZF	Флаг нуля. Если результат вычисления дает 0	jz	переход к метке, если флаг нуля установлен
		jnz	переход к метке, если флаг нуля не установлен
	Сохранение/восстановление состояния		

Флаг	Описание	Команда	Описание
	Порядок битов для обоих операций: 0. Флаг переноса (CF) 1. Всегда равен 1 2. Флаг паритетности (PF) 3. Всегда равен 0 4. Дополнительный флаг переноса (AF) 5. Всегда равен 0 6. Флаг нуля (ZF) 7. Флаг знака (SF) Биты 1, 3, и 5 не используются.	lahf	копирует флаги состояния из регистра eflags в регистр ah
		sahf	сохраняет флаги состояния из регистра ah в регистр eflags

Пример перехода

```
global _start
section .text
_start:
    mov al, 255
    add al, 3       ; AL = AL + 3
    jc carry_set    ; если флаг переноса установлен, переход к метке carry_set
    mov rdi, 2      ; если флаг переноса не установлен, RDI = 2
    jmp exit
carry_set:          ; если флаг переноса установлен
    mov rdi, 4      ; RDI = 4
exit:               ; метка exit
    mov rax, 60
    syscall
```

Сравнение cmp (от compare) сравнивает значения и устанавливает флаги. Результат сравнения используется для условного перехода.

```
cmp left_operand, right_operand
```

Могут участвовать регистры, переменные, непосредственные операнды. Если сравнивается непосредственный операнд, то он указывается вторым. Оба операнда целые числа, числа с плавающей точкой НЕ сравниваются.

Cmp вычитает второй из первого и устанавливает флаги кода условия на основе результата вычитания. Cmp не сохраняет результат вычитания. Аналогичные команды для получения результатов сравнения:

Команда	**Описание**
je / jz	проверяет ZF == 1 выполняет переход, если оба операнда равны.
jne / jnz	проверяет ZF == 0 выполняет переход, если оба операнда не равны.
ja / jnbe	проверяет одновременно CF == 0 и ZF == 0. Переход, если первый операнд больше второго. Оба операнда беззнаковые.
jae / jnb	проверяет CF == 0 Переход, если первый операнд больше или равен второму. Оба операнда беззнаковые. Аналогичен инструкции jnc
jb / jnae	проверяет условие CF == 1 и выполняет переход, если первый операнд меньше второго. Оба операнда беззнаковые. Аналогичен инструкции jc.
jbe / jna	проверяет одновременно два условия CF == 1 и ZF == 1 (достаточно, чтобы выполнялось хотя бы одно из этих условий). Выполняет переход, если первый операнд меньше или равен второму. Оба операнда беззнаковые.
jg / jnle	проверяет одновременно два условия SF == OF и ZF == 0 (оба условия должны быть истинными). Выполняет переход, если первый операнд больше второго. Оба операнда со знаком.
jge / jnl	проверяет условие SF == OF и выполняет переход, если первый операнд больше или равен второму. Оба операнда со знаком.
jl / jnge	проверяет условие SF != OF (флаги SF и OF не должны быть равны) и выполняет переход, если первый операнд меньше второго. Оба операнда со знаком.

Команда	Описание
jle / jng	проверяет одновременно два условия SF != OF и ZF == 1 (достаточно, чтобы выполнялось хотя бы одно из этих условий). Выполняет переход, если первый операнд меньше или равен второму. Оба операнда со знаком.

Через / - одинаковые операторы, и машинный код одинаковый.

Условное копирование. В зависимости от сравнения загрузить в регистр некоторое значение.

Команда	Описание
cmovc / cmovb / cmovnae	копирует значение, если флаг переноса CF = 1
cmovnc / cmovnb / cmovae	копирует значение, если флаг переноса CF = 0
cmovz / cmove	копирует значение, если флаг нуля ZF = 1
cmovnz / cmovne	копирует значение, если флаг нуля ZF = 0
cmovs	копирует значение, если флаг знака SF = 1
cmovns	копирует значение, если флаг знака SF = 0
cmovo	копирует значение, если флаг переполнения OF = 1
cmovno	копирует значение, если флаг переполнения OF = 0

Инструкции для сравнения с копированием. Здесь есть инструкции для сравнения беззнаковых чисел:

Команда	Описание
cmova	копирует значение, если первый операнд больше второго (CF=0, ZF=0)
cmovnbe	копирует значение, если первый операнд не меньше и не равен второму (CF=0, ZF=0)
cmovae / cmovnc / cmovnb	копирует значение, если первый операнд больше или равен второму (CF=0)
cmovnb / cmovnc / cmovae	копирует значение, если первый операнд не меньше второго (CF=0)
cmovb / cmovc / cmovnae	копирует значение, если первый операнд меньше второго (CF=1)
cmovnae / cmovc / cmovb	копирует значение, если первый операнд не больше и не равен второму (CF=1)

Команда	Описание
cmovbe	копирует значение, если первый операнд меньше или равен второму (CF=1 или ZF=1)
cmovna	копирует значение, если первый операнд не больше второго (CF=1 или ZF=1)

Инструкции сравнения чисел со знаком:

Команда	Описание
cmovg	копирует значение, если первый операнд больше второго (SF=OF или ZF=0)
cmovnle	копирует значение, если первый операнд не меньше и не равен второму (SF=OF или ZF=0)
cmovge	копирует значение, если первый операнд больше или равен второму (SF=OF)
cmovnl	копирует значение, если первый операнд не меньше второго (SF=OF)
cmovl	копирует значение, если первый операнд меньше второго (SF != OF)
cmovnge	копирует значение, если первый операнд не больше и не равен второму (SF != OF)
cmovle	копирует значение, если первый операнд меньше или равен второму (SF != OF или ZF=1)
cmovng	копирует значение, если первый операнд не больше второго (SF != OF или ZF=1)

И две общие инструкции как для чисел со знаком, так и для беззнаковых чисел:

cmove: копирует значение, если первый операнд равен второму (ZF=1). Аналогичен инструкции cmovz

cmovne: копирует значение, если первый операнд не равен второму (ZF=0). Аналогичен инструкции cmovnz

Первый параметр этих инструкций (куда копируем) представляет либо регистр, либо переменную (16, 32 или 64-битные). Второй параметр (что копируем) - регистр общего назначения(также 16, 32 или 64-битные).

```
global _start


section .text
_start:
```

```
    mov al, 255
    mov bl, 3
    add al, bl          ; складываем AL и BL

    mov rcx, 2          ; вариант, если флаг переноса сброшен (CF = 0)
    mov rdx, 4          ; вариант, если флаг переноса установлен (CF = 1)

    cmovnc rdi, rcx     ; Если CF = 0
    cmovc rdi, rdx      ; Если CF = 1
    mov rax, 60
    syscall
```

Стек

Команда	**Описание**
push	Кладёт значение в стек push eax
pop	Извлекает значение из стека pop ebx
enter	Создаёт стековый фрейм enter 16, 0
leave	Удаляет стековый фрейм

Циклы

Простой цикл:

```
global _start

section .text
_start:
    mov rcx, 5
    mov rdi, 0
loop:
    add rdi, 2      ; RDI = RDI + 2
    dec rcx         ; RCX = RCX - 1
    jnz loop        ; если флаг нуля НЕ установлен, переход обратно к метке loop
    mov rax, 60
    syscall
```

Встроенный цикл:

Команда	Описание
loop	уменьшает на 1 число в регистре RCX и переходит к определенной метке, если RCX не равен нулю.
loope	продолжает цикл, если установлен флаг нуля
loopne	повторяет цикл, если флаг нуля не установлен
jrcxz	проверяет значение RCX, и если оно рано 0, то переходит к определенной метке.

Пример использования на Linux:

```
global _start

section .text
_start:
    mov rcx, 5      ; регистр-счетчик
    mov rdi, 0
mainloop:           ; цикл
    add rdi, 2      ; некоторые действия цикла
    loop mainloop   ; уменьшаем rcx на 1, переходим к mainloop, если rcx не содержит 0

    mov rax, 60
    syscall
```

Пример для jrcxz

```
global _start

section .text
_start:
    mov rcx, 5      ; регистр-счетчик
    mov rdi, 1
mainloop:           ; цикл
    jrcxz exit      ; если rcx = 0, то переход к метке exit
    add rdi, 2      ; некоторые действия цикла
    loop mainloop  ; уменьшаем значение в rcx на 1, переходим к метке mainloop, если rcx не
содержит 0

exit:
```

```
    mov rax, 60
    syscall
```

Вложенные циклы

Для вложенных циклов нужно сохранять значение внешнего счётчика (например, в стеке).
`Пример: Таблица умножения (5x5)`

```
section .text
    global _start

_start:
    mov ebx, 1       ; Внешний счётчик (строки)

outer_loop:
    mov ecx, 1       ; Внутренний счётчик (столбцы)

inner_loop:
    ; Вычисляем произведение EBX * ECX
    mov eax, ebx
    mul ecx          ; EAX = EBX * ECX

    ; Здесь можно вывести EAX (пропущено для краткости)

    ; Увеличиваем внутренний счётчик
    inc ecx
    cmp ecx, 5
    jle inner_loop

    ; Увеличиваем внешний счётчик
    inc ebx
    cmp ebx, 5
    jle outer_loop

    ; sys_exit(0)
    mov eax, 1
    xor ebx, ebx
    int 0x80
```

Оптимизация циклов в ассемблере

1. Разворот цикла (Loop Unrolling) Уменьшение числа итераций за счёт повторения тела цикла внутри одной итерации.

Пример: Сумма элементов массива (4 элемента за итерацию)

```
section .data
    arr dd 1, 2, 3, 4, 5, 6, 7, 8
    len equ ($ - arr) / 4  ; 8 элементов

section .text
    global _start

_start:
    mov esi, arr        ; Указатель на массив
    mov ecx, len / 4    ; Количество итераций (8 / 4 = 2)
    xor eax, eax        ; Сумма = 0

sum_loop:
    add eax, [esi]      ; Элемент 1
    add eax, [esi + 4]  ; Элемент 2
    add eax, [esi + 8]  ; Элемент 3
    add eax, [esi + 12] ; Элемент 4
    add esi, 16         ; Сдвиг на 4 элемента (4 * 4 байта)
    loop sum_loop

    ; Проверка остатка (если len не кратен 4)
    mov ecx, len % 4
    jz done

remainder_loop:
    add eax, [esi]
    add esi, 4
    loop remainder_loop

done:
    ; EAX = сумма
```

Преимущества	Недостатки
• Уменьшение накладных расходов на проверку условия. • Лучшее использование конвейера процессора.	• Увеличение размера кода. • Сложность обработки остатков.

2. Замена loop на dec + jnz

Инструкция loop медленнее, чем связка dec + jnz т к процессоры лучше оптимизируют dec + jnz. Пример:

```
mov ecx, 100

; Медленнее:
; loop_label:
;     ...
;     loop loop_label

; Быстрее:
loop_label:
    ...
    dec ecx
    jnz loop_label
```

3. Вынос инвариантов из цикла. Вычисление константных выражений до цикла. Пример:

```
; Плохо:
mov ecx, 100
loop_start:
    mov eax, [esi]
    add eax, 10       ; 10 — инвариант
    mov [edi], eax
    add esi, 4
    add edi, 4
    loop loop_start

; Лучше:
```

```
mov ecx, 100
mov ebx, 10           ; Вынесли инвариант
loop_start:
    mov eax, [esi]
    add eax, ebx
    mov [edi], eax
    add esi, 4
    add edi, 4
    loop loop_start
```

4. Использование регистров вместо памяти. Минимизация обращений к памяти внутри цикла. Пример:

```
; Плохо:
mov ecx, 100
loop_start:
    mov eax, [esi]
    add eax, [edi]    ; Чтение из памяти
    mov [esi], eax
    add esi, 4
    add edi, 4
    loop loop_start

; Лучше:
mov ecx, 100
loop_start:
    mov eax, [esi]
    mov ebx, [edi]    ; Загрузили в регистр
    add eax, ebx
    mov [esi], eax
    add esi, 4
    add edi, 4
    loop loop_start
```

5. Устранение зависимостей данных. Параллельное выполнение независимых операций. Пример:

```
; Плохо (зависимость по EAX):
mov ecx, 100
loop_start:
    add eax, [esi]
    add eax, [edi]    ; Ждёт завершения предыдущего ADD
    mov [esi], eax
    add esi, 4
    add edi, 4
    loop loop_start

; Лучше:
mov ecx, 100
loop_start:
    mov ebx, [esi]
    add ebx, [edi]    ; Независимая операция
    mov [esi], ebx
    add esi, 4
    add edi, 4
    loop loop_start
```

6. Инструкции SIMD (SSE/AVX) Обработка нескольких данных одной командой. Пример

```
section .data
    arr1 dd 1.0, 2.0, 3.0, 4.0
    arr2 dd 5.0, 6.0, 7.0, 8.0

section .text
    global _start

_start:
    mov ecx, 4
    mov esi, arr1
    mov edi, arr2

loop_start:
    movaps xmm0, [esi]  ; Загрузка 4 float
    movaps xmm1, [edi]
    addps xmm0, xmm1    ; Параллельное сложение
```

```
    movaps [esi], xmm0
    add esi, 16
    add edi, 16
    sub ecx, 1
    jnz loop_start
```

Revision #11
Created 7 November 2025 13:23:50 by Admin
Updated 11 November 2025 02:12:51 by Admin