Информатика

Архитектура x86. Ассемблер NASM. Основные команды. Отладка

Гирик Алексей Валерьевич

Университет ИТМО
2022

Материалы курса

- Презентации, материалы к лекциям, литература, задания на лабораторные работы
  - shorturl.at/jqRZ6

Система команд архитектуры x86-64

Типы команд

- все команды можно поделить на несколько групп:
  - основные
  - сопроцессора
  - расширений
    - MMX
    - XMM
    - SSE, SSE2, SSE3, ...
    - ...

Функциональная классификация команд

- основные команды процессора можно поделить на несколько групп:
  - пересылки данных
    - общего назначения
    - работы со стеком
  - арифметические
    - двоичной арифметики
    - десятичной арифметики
  - логические
  - передачи управления
  - цепочечные
  - управления состоянием ЦП

Команды пересылки данных

- mov <DST>, <SRC>
  - `mov rax, rbx` `; из регистра в регистр`
  - `mov rax, 0xFF` `; непосредственный операнд`
  - `mov rax, [0xFF]` `; из памяти в регистр`
  - `mov rax, [rax]` `; косвенная адресация`
  - `mov [rax], [rbx]` `; так не получится`
- xchg <OP1>, <OP2>
  - `xchg rbx, rax` `; обменять два регистра`
  - `xchg rax, [rbx]` `; обменять регистр и память`
  - `xchg [rbx], [rax]` `; это тоже не сработает`

Операнды-адреса

В общем случае, допустимы следующие варианты задания адреса в командах, где одним из операндов может быть адрес памяти:

```
[offset]                        ; смещение задано константой
[reg]                           ; смещение задано в регистре
[offset + reg]                  ; сумма смещения и регистра
[offset + reg*scale]            ; scale = 1, 2, 4 или 8
[offset + reg + reg*scale]
```

Примеры:

```
mov    rax, [0xffee]
mov    rax, [array + rbx]
```

В чем проблема?

```
...
mov [rax], 100
...
```

Маловато будет!

```
$ nasm -felf64 test.S
exp.S:16: error: operation size not specified
```

При работе с памятью небходимо задавать размер операнда

```
...
mov dword [rax], 100
...
```

Вот это мой размерчик!

либо можно задать размер непосредственного операнда:

```
mov [rax], dword 100
```

Арифметические команды

- add <OP1>, <OP2>
  - `add rax, 3 ; сложение`
- sub <OP1>, <OP2>
  - `sub rax, rbx ; вычитание`
- inc <OP>
  - `inc rax ; инкремент`
- dec <OP>
  - `dec qword [rax] ; декремент`
- neg <OP>
  - `neg rax ; изменение знака`
- cmp <OP1>, <OP2>
  - `cmp rax, rbx ; сравнение (вычитание без сохранения)`

Арифметические команды

- mul <OP>
  - `mul rdx` `; умножение беззнаковое`
- imul <OP>
  - `imul qword [rdx]` `; умножение знаковое`
- div <OP>
  - `div rbx` `; деление беззнаковое`
- idiv <OP>
  - `idiv rbx` `; деление знаковое`

Логические команды

- not <OP>
  - `not rax` `; поразрядное НЕ`
- or <OP1>, <OP2>
  - `or rax, rbx` `; поразрядное ИЛИ`
- and <OP1>, <OP2>
  - `and rax, [rbx]` `; поразрядное И`
- xor <OP1>, <OP2>
  - `xor rax, rax` `; поразрядное исключающее ИЛИ`
- test <OP1>, <OP2>
  - `test rax, rbx` `; поразрядное И без сохранения`
    `; результата (но с установкой`
    `; флагов)`

Операции сдвига

- shl <OP>, N
  - `shl rax, 10 ; сдвиг влево`
- shr <OP>, N
  - `shr dword [rax], 10 ; сдвиг вправо`
- sar <OP>, N
  - `sar rax, 1 ; арифметический сдвиг вправо`
- rol <OP>, N
  - `rol rax, 2 ; циклический сдвиг влево`
- ror <OP>, N
  - `ror rax, cl ; циклический сдвиг вправо`

Вместо N можно использовать регистр `cl` (и только этот регистр!)

Команды передачи управления

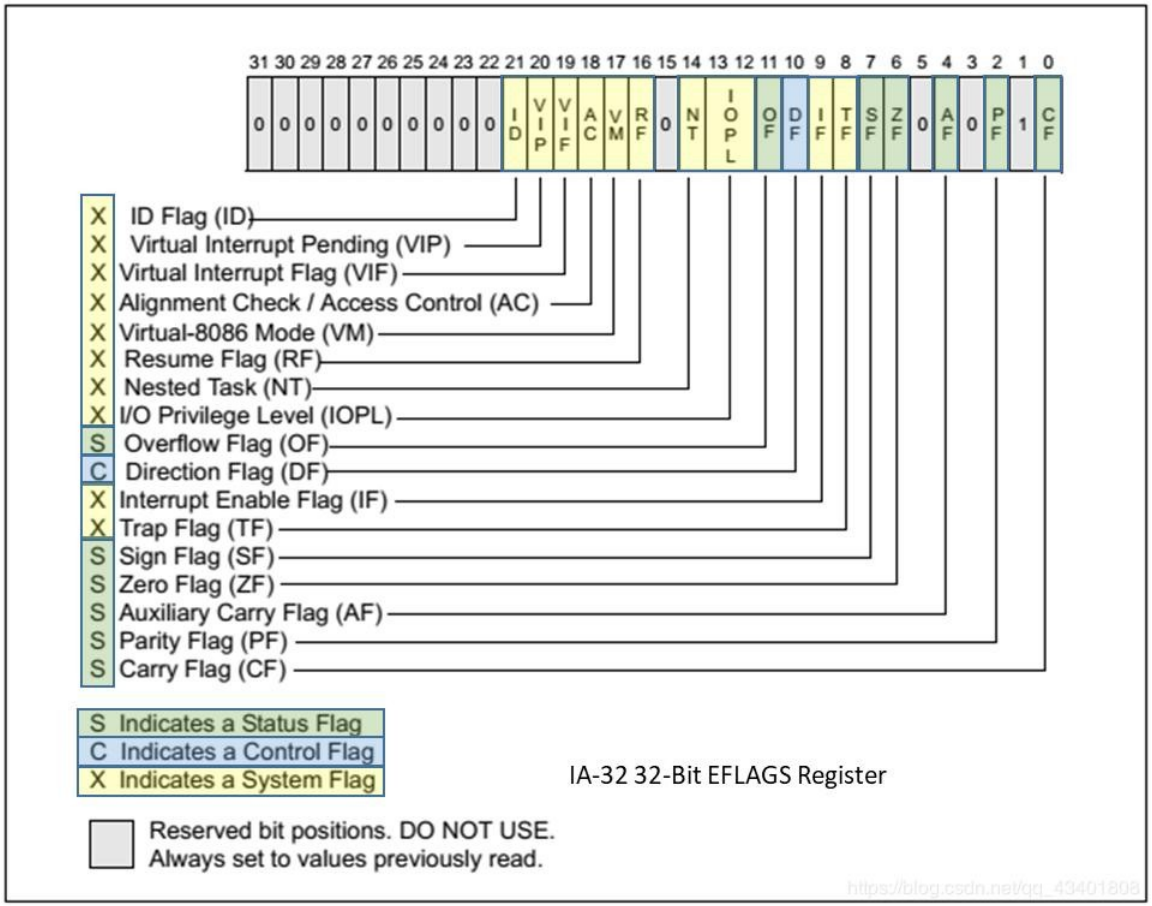
- jmp <ADDR>
  - `jmp loop_beg` ; безусловный переход
  - `jmp [rax]` ; переход на адрес, содержащийся ; в регистре rax

```
loop_beg:              ; начало цикла
   ...
   ...
   jmp loop_beg        ; переход в начало цикла
```

Флаги в регистре rflags

Флаги:

- CF
- ZF
- SF
- OF



Команды передачи управления

- j??? <ADDR>

```
                                ; условный переход, если...
  jz, js, jc, jo                ; установлен флаг
  jnz, jns, jnc, jno            ; НЕ установлен флаг
  je                            ; операнды равны
  jne                           ; операнды НЕ равны
  jl, jnge                      ; меньше
  jle, jng                      ; меньше или равно
  jg, jnle                      ; больше
  jge, jnl                      ; больше или равно
  jb, jnae                      ; меньше
  jbe, jna                      ; меньше или равно
  ja, jnbe                      ; больше
  jae, jnb                      ; больше или равно
```

для чисел со знаком

для чисел без знака

Справка по системе команд x86/x86-64

- официальная документация Intel
  - https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html
- неофициальная документация
  - https://www.felixcloutier.com/x86/
- ...

Пример: вывод звездочек

- Задача: написать программу на ассемблере, которая выводит на консоль фиксированное количество звездочек

```
$ ./stars
*
**
***
****
*****
******
*******
********
```

Как получить случайное число?

- можно использовать пару библиотечных функций srand() и rand()

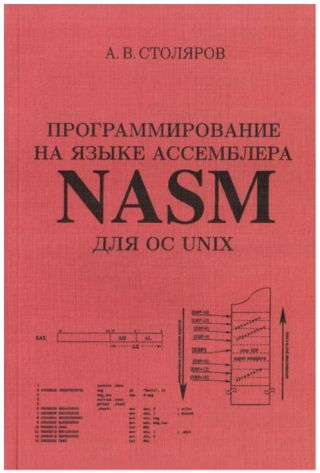
```
srand(time(NULL));
int n = rand() % 16;
```

- на x86-64 можно использовать инструкцию rdrand

```
rdrand   rax
and      rax, 0xf
mov      [n], rax
```

Цепочечные (строковые) инструкции

- используют регистры rsi, rdi, rcx, rax и флаг DF
  - `stosb/stosw/stosd/stosq`
  - `lodsb/lodsw/lodsd/lodsq`
  - `movsb/movsw/movsd/movsq`
  - `cmpsb/cmpsw/cmpsd/cmpsq`
  - `scasb/scasw/scasd/scasq`
- могут использоваться либо в цикле, либо с префиксами повторения
- направление работы определяется флагом DF
  - `std ; установка DF = адреса уменьшаются`
  - `cld ; сброс DF = адреса увеличиваются`



Подробнее –§2.7

Применение цепочечных инструкций

- заполнить массив звездочками

```
mov     al, '*'         ; байт для записи
mov     rdi, buf        ; куда записываем, buf – адрес строки
mov     rcx, 8          ; сколько раз повторять
cld                     ; увеличивать адрес
rep stosb               ; заполнить строку
```

- цепочечные инструкции работают быстро! В этом основной смысл их использования

Отладка программ

Что такое отладчик

- Отладчик (debugger) – программа, которая позволяет выполнять другую программу в пошаговом режиме и наблюдать за состоянием памяти отлаживаемой программы
- Отладчик может быть интегрирован в среду разработки (Visual Studio, XCode) или поставляться отдельно (gdb, lldb)
- Отладку можно выполнять на уровне исходных кодов или машинных инструкций

Что можно узнать с помощью отладчика

- Значения переменных во время выполнения
- Стек вызовов функций
- Значения аргументов функций
- Состояние памяти
- Состояние регистров процессора

- Если происходит исключение, то
  - тип исключения
  - место возникновения

Отладчик gdb

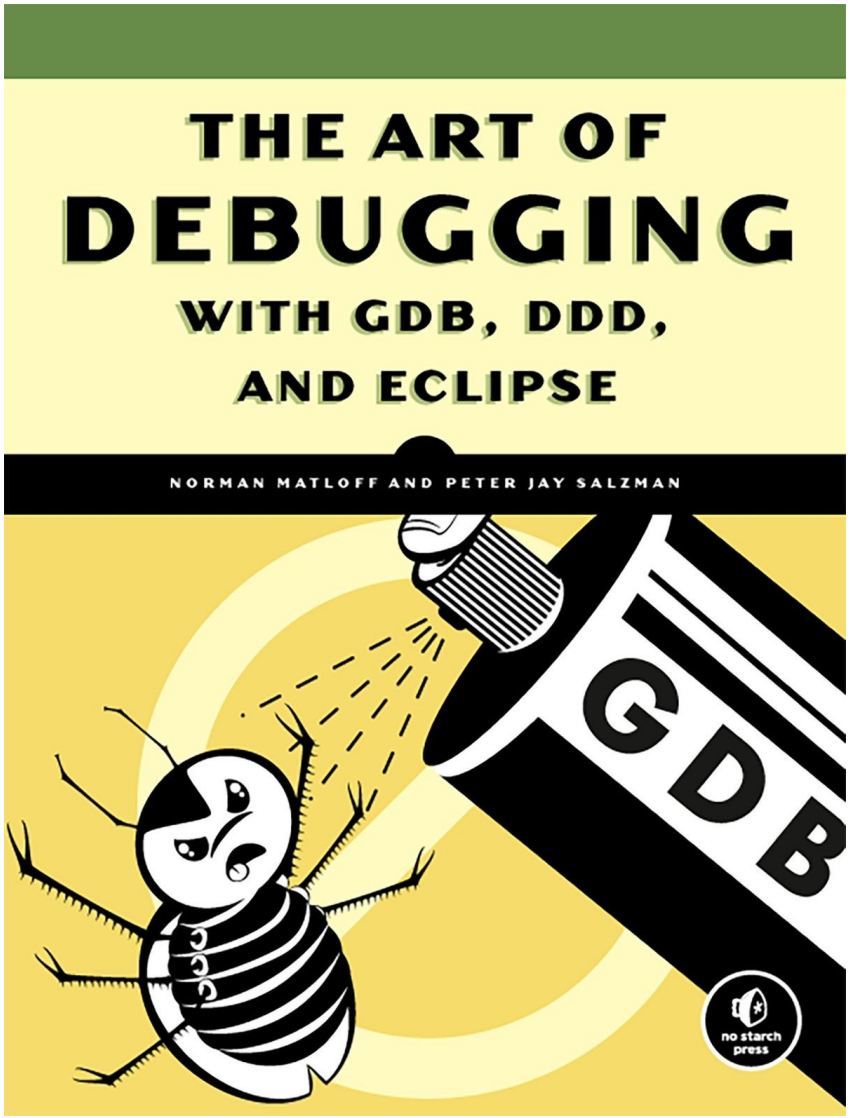
- Основные команды:
  - help <cmd> – показать справку по cmd
  - break <func> – точка останова в func
  - run – запустить программу
  - next, step – выполнить следующую строку исходного кода
  - ni, si – выполнить следующую инструкцию
  - list – показать исходный код
  - disas /s – показать код+инструкции
  - frame – показать текущее положение
  - kill – остановить выполнение программы
  - quit – выход

Отладчик gdb

- Информация о текущем состоянии:
  - info reg – состояние регистров
  - info local – локальные переменные
  - info arg – значения аргументов функции
  - p <переменная> – вывести значение переменной
  - x/<формат> <объект> – вывести значение объекта (переменной, адреса,...) в заданном формате
  - bt – показать состояние стека вызовов

Книжка по gdb

- *N. Matloff, P. Salzman*
  The Art of Debugging with gdb, ddd and Eclipse



gdb dashboard

https://github.com/cyrus-and/gdb-dashboard

Задание к следующей лекции

- Столяров. Программирование на языке ассемблера NASM для ОС UNIX
  - гл. 2 §2.6 –§2.9
  - гл. 3