ЭКСПЛУАТАЦИЯ УЯЗВИМОСТЕЙ ПО
ЛЕКЦИЯ 0x01

Браницкий А.А.

Санкт-Петербургский государственный университет
телекоммуникаций им. проф. М.А. Бонч-Бруевича

Лекция 1, Санкт-Петербург, 02 сентября 2022 г.

Содержание

Ассемблер

Ассемблер — программа, переводящая мнемонические обозначения инструкций в последовательность кодов команд целевой машины (процессора).

- Для каждой машины может существовать несколько ассемблеров с различным синтаксисом
- Набор кодов машинных команд (система команд) является специфичной для каждой машины

Примеры: синтаксис Intel и AT&T

Задача: найти сумму чисел от 1 до 10 и вывести результат на консоль.

```
1 N equ 10
2 section .data
3     fmt db "1 + ... + %d = %d",10,0
4 section .text
5     extern printf
6     global _start
7 _start:
8     ; calculate sum
9     mov ecx, N
10    mov edx, 0
11 label1:
12    add edx, ecx
13    loop label1
14    ; call printf
15    push edx
16    push dword N
17    push fmt
18    call printf
19    add esp, 12
20    ; call exit
21    mov ebx, 0
22    mov eax, 1
23    int 0x80
```

```
$ nasm -f elf sum.asm -o sum.o
$ ld -m elf_i386 sum.o -o sum \
  -lc --dynamic-linker /lib/ld-linux.so.2
$ ./sum
  1 + ... + 10 = 55
```

```
1 .set N, 10
2     .global _start
3     .text
4 _start:
5     / calculate sum
6     mov $N, %ecx
7     mov $0, %edx
8 label1:
9     add %ecx, %edx
10    loop label1
11    / call printf
12    push %edx
13    push $N
14    push $fmt
15    call printf
16    add $12, %esp
17    / call exit
18    mov $0, %ebx
19    mov $1, %eax
20    int $0x80
21 fmt:
22    .ascii "1 + ... + %d = %d\n"
```

```
$ cc -m32 -c sum.s -o sum.o
$ ld -m elf_i386 sum.o -o sum \
  -lc --dynamic-linker /lib/ld-linux.so.2
$ ./sum
  1 + ... + 10 = 55
```

Машинные команды

```
$ objdump -M intel -dj .text sum

sum:     file format elf32-i386


Disassembly of section .text:

08048190 <_start>:
 8048190:       b9 0a 00 00 00          mov    ecx,0xa
 8048195:       ba 00 00 00 00          mov    edx,0x0

0804819a <label1>:
 804819a:       01 ca                   add    edx,ecx
 804819c:       e2 fc                   loop   804819a <label1>
 804819e:       52                      push   edx
 804819f:       6a 0a                   push   0xa
 80481a1:       68 6c 92 04 08          push   0x804926c
 80481a6:       e8 d5 ff ff ff          call   8048180 <printf@plt>
 80481ab:       83 c4 0c                add    esp,0xc
 80481ae:       bb 00 00 00 00          mov    ebx,0x0
 80481b3:       b8 01 00 00 00          mov    eax,0x1
 80481b8:       cd 80                   int    0x80
```

машинные команды

мнемонические команды

История Intel i386

1971 г. (15 ноября) — 1-ый 1-кристальный микропроцессор Intel 4004 (4-битный процессор, 12-битная адресная шина, 4-битная шина данных).
1972 г. (01 апреля) — 1-ый 8-битный процессор Intel 8008 (14-битная адресная шина, 8-битная шина данных).
1974 г. (апрель) — процессор Intel 8080, модернизированная версия процессора Intel 8008 (16-битная адресная шина, 8-битная шина данных).
1978 г. (08 июня) — 1-ый 16-битный процессор Intel 8086 (20-битная адресная шина, 16-битная шина данных).
1979 г. (01 июля) — 16-битный процессор Intel 8088 (20-битная адресная шина, 8-битная шина данных).
1982 г. — 16-битный процессор Intel 80186, модернизированная версия процессора Intel 8086 (20-битная адресная шина, 16-битная шина данных).
1982 г. (01 февраля) — 16-битный процессор Intel 80286 (поддержка защищенного режима работы с защитой памяти, 24-битная адресная шина, 16-битная шина данных).
1985 г. (17 октября) — 1-ый 32-битный процессор Intel 80386 (i386) (поддержка страничной организации виртуальной памяти, 32-битная адресная шина, 32-битная шина данных).
1989 г. (10 апреля) — 32-битный процессор Intel 80486 (i486) (встроенный математический сопроцессор, 32-битная адресная шина, 32-битная шина данных).

Hello world!

```
 1 global _start
 2 section .data
 3     WRITE_CODE equ 4
 4     EXIT_CODE equ 1
 5     STDOUT equ 1
 6     RET_VAL equ 0
 7     msg db "Hello world!",10
 8     msg_len equ $-msg
 9 section .text
10 _start:
11     mov eax, WRITE_CODE
12     mov ebx, STDOUT
13     mov ecx, msg
14     mov edx, msg_len
15     int 0x80
16     mov eax, EXIT_CODE
17     mov ebx, RET_VAL
18     int 0x80
```

```
$ nasm -f elf hello_world.asm -o hello_world.o
$ ld -m elf_i386 hello_world.o -o hello_world
$ ./hello_world
  "Hello world!"
```

Общие понятия

- Метки, переходы и циклы
- Регистры
- Стек
- Секция кода (.text)
- Секция данных (.data)
- Секция неинициализированных данных (.bss)
- Системные вызовы и прерывания
- Арифметические операторы

Метка

Метка — заданный программистом идентификатор, ассоциирующийся с адресом в памяти и позволяющий выполнить переход на указанную позицию в коде.

`_start` — точка входа в программу.

```
$ objdump -M intel -d hello_world | grep -A 1 _start
  08048080 <_start>:
   8048080:       b8 04 00 00 00                 mov    eax,0x4
```

08048080 — адрес ячейки, которая соответствует метке `_start` и содержит машинный код первой команды программы (`mov eax,0x4`).

Переходы

Переходы предназначены для передачи управления программы в указанное меткой место.

- Условные (je, jne, jl, jle, jg, jge, ...)
- Безусловные (jmp):
  - ✓ Дальние (far) (переходы в другие сегменты)
  - ✓ Близкие (near) (переходы в произвольное место внутри сегмента)
  - ✓ Короткие (short) (переходы не более чем на 127 байт вперед или 128 байт назад)

Циклы

- Через использование условных и безусловных переходов
- Через использование команды loop (или ее производных команд loope/loopz, loopne/loopnz)

```
lab:
    ...
    dec ecx
    jnz lab
```

$\Leftrightarrow$

```
lab:
    ...
    loop lab
```

✓ занимает больше места

✓ работает медленней

✓ занимает меньше места

✓ работает быстрей

Регистры

Регистр — электронное устройство, представляющее собой набор ячеек сверхбыстрой памяти

- Сегментные регистры
- Регистры общего назначения
- Специальные регистры

Сегментные регистры

2-байтные служебные регистры, используемые операционной системой для обращения к адресам памяти сегментов кода, данных, стека, дополнительных данных.

- CS (Code Segment) — сегментный регистр кода
- DS (Data Segment) — сегментный регистр данных
- SS (Stack Segment) — сегментный регистр стека
- ES (Extra Segment) — сегментный регистр дополнительных данных
- GS — сегментный регистр дополнительных данных
- FS — сегментный регистр дополнительных данных

Регистры общего назначения

4-байтные регистры (процессор i386)

- EAX (accumulator)
  `EAX=(EAX&0xFFFF0000)|AX=(EAX&0xFFFF0000)|(AH«8)|AL`
- EBX (base)
  `EBX=(EBX&0xFFFF0000)|BX=(EBX&0xFFFF0000)|(BH«8)|BL`
- ECX (counter)
  `ECX=(ECX&0xFFFF0000)|CX=(ECX&0xFFFF0000)|(CH«8)|CL`
- EDX (data)
  `EDX=(EDX&0xFFFF0000)|DX=(EDX&0xFFFF0000)|(DH«8)|DL`
- ESI (source index)
  `ESI=(ESI&0xFFFF0000)|SI`
- EDI (destination index)
  `EDI=(EDI&0xFFFF0000)|DI`
- EBP (base pointer)
  `EBP=(EBP&0xFFFF0000)|BP`
- ESP (stack pointer)
  `ESP=(ESP&0xFFFF0000)|SP`

Специальные регистры

4-байтные регистры (процессор i386)

- EIP (extended instruction pointer) — регистр счетчика команд

```
EIP=(EIP&0xFFFF0000)|IP
```

- EFLAGS — регистр флагов

```
EFLAGS=(EFLAGS&0xFFFF0000)|FLAGS
```

Стек

Стек — специальная область памяти (LIFO-очередь), предназначенная для хранения локальных переменных, передачи аргументов в вызываемую функцию и возможности возврата из вызванной функции.
Некоторые команды, влияющие на адрес стека (значение регистра esp):

- push ...
- pop ...
- call ...
- ret
- enter ...
- leave
- mov esp, ...; lea esp, ...; add esp, ...; sub esp, ...
- ...

Секция кода (.text)

section .text

- Представляет собой последовательность команд
- Содержит неизменяемые в процессе выполнения программы данные

Секция данных (.data)

section .data

- Содержит инициализированные данные, хранимые в структуре исполняемого файла

Для инициализации переменных используются следующие команды:

- db (для однобайтовых ячеек)
- dw (для двухбайтовых ячеек)
- dd (для четырехбайтовых ячеек)

Секция неинициализированных данных (.bss)

section .bss
Отличия от секции .data:

- Размер секции .bss не влияет на размер исполняемого файла
- В секции .bss хранится только размер выделенной под переменные памяти (в секции .data также сохраняются и значения, присвоенные переменным)
- Секция .bss может динамически расти в процессе выполнения программы

Для резервирования памяти используются следующие команды:

- resb (для однобайтовых ячеек)
- resw (для двухбайтовых ячеек)
- resd (для четырехбайтовых ячеек)

Системные вызовы и прерывания

- Системный вызов — обращение пользовательской программы к средствам ядра операционной системы; для выполнения этого вызова используются прерывания (около 400 системных вызовов в ядре Linux, около 700 — в ядре Windows)
- Программные прерывания являются результатом исполнения команды `int 0x80`
- Внутренние прерывания являются результатом нарушения условий во время исполнения кода (например, переполнение стека или деление на ноль)
- Внешние прерывания возникают по требованию внешних устройств, полученных, например, от системного таймера или видеокарты

Арифметические операторы (ADD)

ADD SUM, NUM

Сложение двух чисел, представленных аргументами SUM и NUM. Результат помещается в SUM.

SUM:

- Область памяти
- Регистр общего назначения

NUM:

- Область памяти
- Регистр общего назначения
- Непосредственное значение, например, число

Пример: ADD EAX, 10

Арифметические операторы (SUB)

SUB DIF, NUM

Вычитание одного числа из другого, представленных аргументами DIF и NUM. Результат помещается в DIF.

DIF:

- Область памяти
- Регистр общего назначения

NUM:

- Область памяти
- Регистр общего назначения
- Непосредственное значение, например, число

Пример: SUB EAX, 10

Арифметические операторы (MUL)

MUL NUM
Умножение двух чисел, представленных аргументами (1) регистром AL (если размер NUM — 1 байт) и NUM, (2) регистром AX (если размер NUM — 2 байта) и NUM, (3) регистром EAX (если размер NUM — 4 байта) и NUM. Результат помещается в (1) регистр AX, если размер NUM — 1 байт, (2) пару регистров (DX:AX), если размер NUM — 2 байта, (3) пару регистров (EDX:EAX), если размер NUM — 4 байта.
NUM:

- Область памяти
- Регистр общего назначения

Пример: MUL BX

Арифметические операторы (DIV)

DIV NUM

Частное от двух чисел, представленных аргументами (1) регистром AX (если размер NUM — 1 байт) и NUM, (2) парой регистров (DX:AX) (если размер NUM — 2 байта) и NUM, (3) парой регистров (EDX:EAX) (если размер NUM — 4 байта) и NUM. Результат целочисленного деления помещается в (1) регистр AL, если размер NUM — 1 байт, (2) регистр AX, если размер NUM — 2 байта, (3) регистр EAX, если размер NUM — 2 байта. Результат остатка от деления помещается в (1) регистр AH, если размер NUM — 1 байт, (2) регистр DX, если размер NUM — 2 байта, (3) регистр EDX, если размер NUM — 4 байта. NUM:

- Область памяти
- Регистр общего назначения

Пример: DIV BX

Арифметические операторы (IMUL)

Формы IMUL:

- IMUL NUM. Работает аналогично MUL NUM.
- IMUL PROD, NUM. PROD := PROD * NUM. Ограничения: не работает с однобайтовыми операндами.
- IMUL PROD, NUM1, NUM2. PROD := NUM1 * NUM2. Ограничения: не работает с однобайтовыми операндами, NUM2 — непосредственное значение.

Пример: IMUL DX, BX, 10

Числа Фибоначчи

Числа Фибоначчи — числовая последовательность, в которой каждый следующий элемент равен сумме двух предыдущих.
Рекурсивное определение:

$$Fib_n = \begin{cases} 0, \text{ если } n = 0 \\ 1, \text{ если } n = 1 \\ Fib_{n-1} + Fib_{n-2}, \text{ если } n \geqslant 2. \end{cases}$$

Итеративное определение:

```
def fib(n) do
  a, b <- 0, 1
  while (n-- > 0) do
    a, b <- b, a + b
  od
  return a
od
```

$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987,$
$1597, 2584, 4181, 6765$

Рекурсивное вычисление чисел Фибоначчи

```
; rec_fib.asm
%include "print.inc"
global _start
section .data
    NUMBER equ 10
    EXIT_CODE equ 1
    RET_VAL equ 0
section .text
; input argument: in eax
; return value: in eax
fib:
    cmp eax, 0
    je fib_lab1
    cmp eax, 1
    je fib_lab1
    dec eax
    push eax
    call fib
    mov ebx, eax
    pop eax
    dec eax
    push ebx
    call fib
    pop ebx
    add eax, ebx
fib_lab1:
    ret
_start:
    mov eax, NUMBER
    cmp eax, 0
    jl lab1
    call fib
    call print_number
    mov eax, 10
    call print_symbol
lab1:
    mov eax, EXIT_CODE
    mov ebx, RET_VAL
    int 0x80
```

```
$ nasm -f elf rec_fib.asm -o rec_fib.o
$ ld -m elf_i386 rec_fib.o -o rec_fib
$ ./rec_fib
  55
```

```
$ for i in `seq 0 20`; do \
  perl -pi -e "s/NUMBER equ ([0-9]+)$/NUMBER equ ${i}/" rec_fib.asm; \
  nasm -f elf rec_fib.asm -o rec_fib.o; \
  ld -m elf_i386 rec_fib.o -o rec_fib; \
  ./rec_fib; \
  done
  0
  1
  ...
  6765
```

Итеративное вычисление чисел Фибоначчи

```
; iter_fib.asm
%include "print.inc"
global _start
section .data
    NUMBER equ 17
    EXIT_CODE equ 1
    RET_VAL equ 0
section .text
; input argument: in eax
; return value: in eax
fib:
    cmp eax, 0
    je fib_lab1
    cmp eax, 1
    je fib_lab1
    mov ecx, eax
    dec ecx
    xor eax, eax
    mov ebx, 1
fib_lab2:
    mov edx, eax
    add edx, ebx
    mov eax, ebx
    mov ebx, edx
    loop fib_lab2
    mov eax, ebx
fib_lab1:
    ret
_start:
    mov eax, NUMBER
    cmp eax, 0
    jl lab1
    call fib
    call print_number
    mov eax, 10
    call print_symbol
lab1:
    mov eax, EXIT_CODE
    mov ebx, RET_VAL
    int 0x80
```

```
$ nasm -f elf iter_fib.asm -o iter_fib.o
$ ld -m elf_i386 iter_fib.o -o iter_fib
$ ./iter_fib
  55
```

```
$ for i in `seq 0 20`; do \
  perl -pi -e "s/NUMBER equ ([0-9]+)$/NUMBER equ ${i}/" iter_fib.asm; \
  nasm -f elf iter_fib.asm -o iter_fib.o; \
  ld -m elf_i386 iter_fib.o -o iter_fib; \
  ./iter_fib; \
  done
  0
  1
  ...
  6765
```

История создания C

1958 г. — разработан ALGOL 58 (ALGorithmic Language) (Джон Бэкус, Джозеф Уэгстен, Джон Маккарти, Петер Наур, Эдсгер Дейкстра)
1963 г. — разработан язык программирования CPL совместными усилиями сотрудников кембриджского и лондонского университетов (Кристофер Стрейчи, Дэвид Бэррон и др.)
1966 г. — Мартин Ричардс разработал язык программирования BCPL (Basic Combined Programming Language)
1969 г. — Кен Томпсон и Деннис Ритчи реализовали язык программирования B (бестипный), прообраз C, предназначенный для первой ОС UNIX на ЭВМ PDP-7
1972 г. — Кен Томпсон и Деннис Ритчи реализовали язык программирования C, использующийся при разработке ядра Linux и многих прикладных программ

Hello world!

```c
// hello_world.c
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("Hello world!\n");
    return 0;
}
```

```
$ gcc -c hello_world.c -o hello_world.o
$ gcc hello_world.o -o hello_world
$ ./hello_world
  Hello world!
```

Оператор ветвления if-else

```c
// even_odd.c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    srand(time(NULL));
    const int r = rand();
    if (r % 2 == 0)
        printf("%d is even number\n", r);
    else
        printf("%d is odd number\n", r);
    return 0;
}
```

```
$ gcc -c even_odd.c -o even_odd.o && gcc even_odd.o -o even_odd
$ for i in `seq 0 2`; do
> ./even_odd
> sleep 1
> done
  1266093312 is even number
  953012229 is odd number
  652176710 is even number
```

Оператор выбора switch-case

```c
// number.c
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    if (argc < 2) return 1;
    const int n = atoi(argv[1]);
    switch (n)
    {
    case 1:
        printf("one\n");
        break;
    case 2:
        printf("two\n");
        break;
    case 3:
        printf("three\n");
        break;
    case 4:
    case 5:
    case 6:
    case 7:
    case 8:
    case 9:
        printf("I forgot\n");
        break;
    default:
        printf("I don't know\n");
        break;
    }
    return 0;
}
```

```
$ gcc -c number.c -o number.o
$ gcc number.o -o number
$ ./number
$ ./number 1
  one
$ ./number 2
  two
$ ./number 3
  three
$ ./number 4
  I forgot
$ ./number 5
  I forgot
$ ./number 6
  I forgot
$ ./number 7
  I forgot
$ ./number 8
  I forgot
$ ./number 9
  I forgot
$ ./number 0
  I don't know
$ ./number 10
  I don't know
$ ./number -1
  I don't know
```

Цикл со счетчиком

```
 1// sum_for.c
 2#include <stdio.h>
 3
 4int main(int argc, char *argv[])
 5{
 6    printf("Enter a non-negative number: ");
 7    int n = 0;
 8    scanf("%d", &n);
 9    if (n < 0) return 1;
10    int i = 1, s = 0;
11    for (; i <= n; ++i)
12        s += i;
13    printf("0 + ... + %d = %d\n", n, s);
14    return 0;
15}
```

```
$ gcc -c sum_for.c -o sum_for.o
$ gcc sum_for.o -o sum_for
$ ./sum_for
  Enter a non-negative number: 7
  0 + ... + 7 = 28
```

Цикл с предусловием

```c
// sum_while.c
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("Enter a non-negative number: ");
    int n = 0;
    scanf("%d", &n);
    if (n < 0) return 1;
    int i = 0, s = 0;
    while (++i <= n)
        s += i;
    printf("0 + ... + %d = %d\n", n, s);
    return 0;
}
```

```
$ gcc -c sum_while.c -o sum_while.o
$ gcc sum_while.o -o sum_while
$ ./sum_while
  Enter a non-negative number: 7
  0 + ... + 7 = 28
```

Цикл с постусловием

```c
// sum_do_while.c
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("Enter a non-negative number: ");
    int n = 0;
    scanf("%d", &n);
    if (n < 0) return 1;
    int i = 0, s = 0;
    do
        s += i;
    while (++i <= n);
    printf("0 + ... + %d = %d\n", n, s);
    return 0;
}
```

```
$ gcc -c sum_do_while.c -o sum_do_while.o
$ gcc sum_do_while.o -o sum_do_while
$ ./sum_do_while
  Enter a non-negative number: 7
  0 + ... + 7 = 28
```

Цикл через оператор goto

```c
// sum_goto.c
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("Enter a non-negative number: ");
    int n = 0;
    scanf("%d", &n);
    if (n < 0) return 1;
    int i = 0, s = 0;
    lab:
        s += i;
    if (++i <= n)
        goto lab;
    printf("0 + ... + %d = %d\n", n, s);
    return 0;
}
```

```
$ gcc -c sum_goto.c -o sum_goto.o
$ gcc sum_goto.o -o sum_goto
$ ./sum_goto
  Enter a non-negative number: 7
  0 + ... + 7 = 28
```

Рекурсивное вычисление чисел Фибоначчи

```c
// rec_fib.c
#include <stdio.h>

int fib(int n)
{
    if (n == 0 || n == 1)
        return n;
    return fib(n - 1) + fib(n - 2);
}

int main(int argc, char *argv[])
{
    printf("Enter a non-negative number: ");
    int n = 0;
    scanf("%d", &n);
    if (n < 0) return 1;
    printf("fib(%d) = %d\n", n, fib(n));
    return 0;
}
```

```
$ gcc -c rec_fib.c -o rec_fib.o
$ gcc rec_fib.o -o rec_fib
$ for i in `seq 0 20`; do
  echo $i | ./rec_fib |
  sed -e 's/^.\+: //';
  done
  fib(0) = 0
  fib(1) = 1
  fib(2) = 1
  fib(3) = 2
  fib(4) = 3
  fib(5) = 5
  fib(6) = 8
  fib(7) = 13
  fib(8) = 21
  fib(9) = 34
  fib(10) = 55
  fib(11) = 89
  fib(12) = 144
  fib(13) = 233
  fib(14) = 377
  fib(15) = 610
  fib(16) = 987
  fib(17) = 1597
  fib(18) = 2584
  fib(19) = 4181
  fib(20) = 6765
```

Итеративное вычисление чисел Фибоначчи

```c
// iter_fib.c
#include <stdio.h>

int fib(int n)
{
    int a = 0, b = 1, c;
    while (n-- > 0)
    {
        c = a + b;
        a = b;
        b = c;
    }
    return a;
}

int main(int argc, char *argv[])
{
    printf("Enter a non-negative number: ");
    int n = 0;
    scanf("%d", &n);
    if (n < 0) return 1;
    printf("fib(%d) = %d\n", n, fib(n));
    return 0;
}
```

```
$ gcc -c iter_fib.c -o iter_fib.o
$ gcc iter_fib.o -o iter_fib
$ for i in `seq 0 20`; do
  echo $i | ./iter_fib |
  sed -e 's/^.\+: //';
  done
  fib(0) = 0
  fib(1) = 1
  fib(2) = 1
  fib(3) = 2
  fib(4) = 3
  fib(5) = 5
  fib(6) = 8
  fib(7) = 13
  fib(8) = 21
  fib(9) = 34
  fib(10) = 55
  fib(11) = 89
  fib(12) = 144
  fib(13) = 233
  fib(14) = 377
  fib(15) = 610
  fib(16) = 987
  fib(17) = 1597
  fib(18) = 2584
  fib(19) = 4181
  fib(20) = 6765
```

История создания Python

1986 г. — разработан язык ABC, прообраз языка Python, предназначенный для пользователей, которые ранее не программировали
1991 г. — выпущена версия языка Python 0.9.0 (Гвидо ван Россум)
1994 г. — выпущена первая версия языка Python
1995 г. — создана некоммерческая организация Python Software Foundation, отвечающая за защиту и развитие языка Python
2000 г. — выпущена вторая версия языка Python
2008 г. — выпущена третья версия языка Python

Hello world!

```
1// hello_world.py
2#!/usr/bin/python
3
4print("Hello World")
```

```
$ chmod +x hello_world.py
$ ./hello_world.py
  Hello world!
```

Оператор ветвления if-else

```
if Condition_1:
    Set_of_instructions_1
elif Condition_2:
    Set_of_instructions_2
...
else:
    Set_of_instructions_N
```

```
// even_odd.py
#!/usr/bin/python
r = int(input())
if r % 2 == 0:
    print("{} is even number".format(r));
else:
    print("{} is odd number".format(r));
```

```
$ chmod +x even_odd.py
$ ./even_odd.py
  12
  12 is even number
$ ./even_odd.py
  13
  13 is even number
```

Цикл for

```
1// sum_for.py
2#!/usr/bin/python
3r = int(input())
4s = 0
5for i in range(r + 1):
6    s += i
7print("1 + ... + {} = {}".format(r, s))
```

```
$ chmod +x sum_for.py
$ ./sum_for.py
  5
  1 + ... + 5 = 15
```

Цикл while

```
1// sum_while.py
2#!/usr/bin/python
3#!/usr/bin/python
4r = int(input())
5i, s = 1, 0
6while i <= r:
7     s += i
8     i += 1
9print("1 + ... + {} = {}".format(r, s))
```

```
$ chmod +x sum_while.py
$ ./sum_while.py
  5
  1 + ... + 5 = 15
```

Рекурсивное вычисление чисел Фибоначчи

```
1// rec_fib.py
2#!/usr/bin/python
3def fib(n):
4    if n == 0 or n == 1:
5        return n
6    return fib(n - 1) + fib(n - 2)
7
8r = int(input())
9print("fib({}) = {}".format(r, fib(r)))
```

```
$ chmod +x rec_fib.py
$ ./rec_fib.py
  0
  fib(0) = 0
$ ./rec_fib.py
  1
  fib(1) = 1
$ ./rec_fib.py
  2
  fib(2) = 1
$ ./rec_fib.py
  3
  fib(3) = 2
$ ./rec_fib.py
  4
  fib(4) = 3
$ ./rec_fib.py
  5
  fib(5) = 5
$ ./rec_fib.py
  6
  fib(6) = 8
```

Итеративное вычисление чисел Фибоначчи

```
1 // iter_fib.py
2 #!/usr/bin/python
3 def fib(n):
4     a, b = 0, 1
5     while n > 0:
6         a, b = b, a + b
7         n -= 1
8     return a
9
10 r = int(input())
11 print("fib({}) = {}".format(r, fib(r)))
```

```
$ chmod +x iter_fib.py
$ ./iter_fib.py
  0
  fib(0) = 0
$ ./iter_fib.py
  1
  fib(1) = 1
$ ./iter_fib.py
  2
  fib(2) = 1
$ ./iter_fib.py
  3
  fib(3) = 2
$ ./iter_fib.py
  4
  fib(4) = 3
$ ./iter_fib.py
  5
  fib(5) = 5
$ ./iter_fib.py
  6
  fib(6) = 8
```

Упражнения

1. Реализовать программу рекурсивного вычисления факториала заданного числа на C
2. Реализовать программу итеративного вычисления факториала заданного числа на C
3. Реализовать программу рекурсивного вычисления факториала заданного числа на python
4. Реализовать программу итеративного вычисления факториала заданного числа на python
5. Реализовать программу вычисления суммы ряда $\frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \cdots + \frac{1}{n^2}$ (на assembler, C или python)