

WSIZ, 2023  
Computer systems architecture

# **Lecture 4. Stack. Subroutine call. Modular software design.**

Dmitry Zaitsev

<http://daze.ho.ua>

# Basics to repeat

- Segments, labels, entry point
- Registers
- Instructions: move data, arithmetic, logic, branching
- Length suffix: b, w, l, q
- Addressing modes

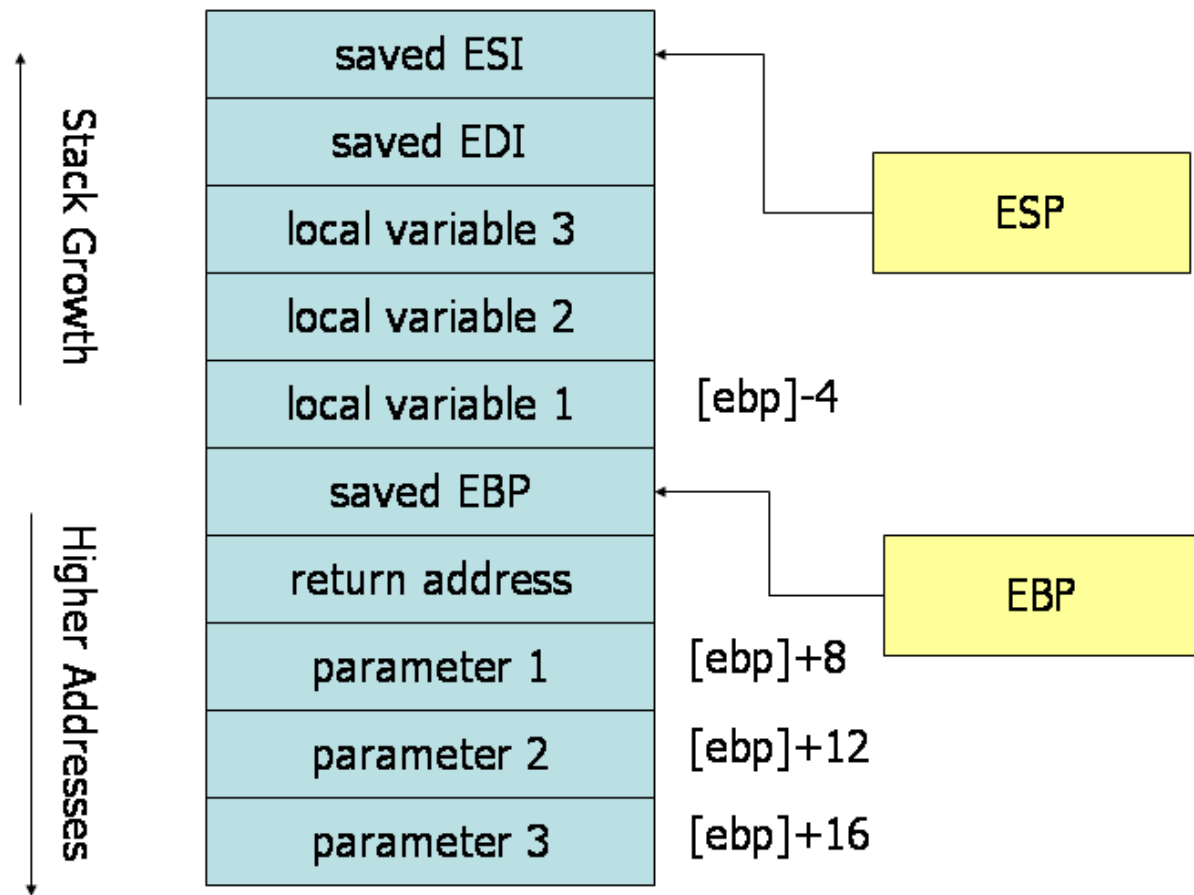
# Concept of stack

- Stack Segment - SS
- LIFO – last-in-first-out
- Stack is allocated in memory, special registers and instructions provide access
- SP register – top of stack
- BP register – base of stack
- Stack content is changed by:
  - push and pop instructions
  - call and ret instructions
  - interrupt procedure

# Function of stack

- Save registers
- Save return address at subroutine call
- Save current PSW and EIP at interrupt
- Allocate local variables of a function
- Support recursive programming style

# Using stack



# PUSH

- Put data into stack
- Decrements the stack pointer and then stores the source operand on the top of the stack.
- `push %rdx`
- $SP = SP - 8$
- $(SP) = \%rdx$

# POP

- Get data from stack
- Loads the value from the top of the stack to the destination location and then increments the stack pointer.
- `pop %rdx`
- `%rdx=(SP)`
- `SP=SP+8`

# Task – check parenthesis language

- Formal grammar:  $S \rightarrow (S)S \mid (S) \mid ()S \mid ()$
- Examples:
  - correct:  $((()))$ ,  $()()()((()))()$
  - incorrect:  $((()))$ ,  $()()((()))()$
- Recognition idea:
  - on “(“ push it into stack
  - on “)” pop “(“ from stack (if absent - insuccess)
  - if stack is empty on completion - success



## # recognition of parenthesis

# language using stack:  $S \rightarrow (S) | ()S | ()$

```
.data
s:      .asciz "(()()())()()())"
t:      .asciz "correct"
f:      .asciz "incorrect"
.text
.global main
main: pushw $*'      # stack bottom symbol
      mov $s, %rdi
loop: movb (%rdi),%al
      cmpb $0,%al    # end of string
      je chk
      cmpb $'(', %al
      jne cls
      pushw %ax      # push ( into stack
      jmp nxt
cls:   popw %ax      # pop ( from stack on )
```

```
cmpb $'(', %al
      jne ff        # early stack bottom
nxt:   inc %rdi
      jmp loop
chk:   popw %ax
      cmpb $'*',%al
      jne fp
      mov $t, %rdi  # stack bottom - succes
      jmp p
fp:    popw %ax      # clear stack
      cmpb $'*',%al
      jne fp
ff:    mov $f, %rdi
p:     call puts
      xor %rax, %rax
      ret
```

# How it works

Online Assembler (GCC) Compile x +

jdoodle.com/compile-assembler-gcc-online/

JDoodle

Online Assembler - GCC Compiler IDE

```
1 # recognition of parenthesis language using stack
2 data
3 s: .asciz "(OOO)O(OO)"
4 t: .asciz "correct"
5 f: .asciz "incorrect"
6 .text
7 global main
8 main:
9     pushw $"" # stack bottom symbol
10    mov $s, %rdi
11    loop:
12        movb (%rdi), %al
13        cmpl $0, %al # end of string
14        je chk
15        cmpl $'(', %al
16        jne cis
17        pushw %ax # push ( into stack
18        jmp nxt
19    cis:
20        popw %ax # pop ( from stack when read )
21        cmpl $'(', %al
22        jne ff # early stack bottom
23    nxt: inc %rdi
24        jmp loop
25    chk:
26        popw %ax
27        cmpl $'(', %al
28        jne fp
29        mov $t, %rdi # stack bottom - succes
30        jmp p
31    fp: popw %ax # clear stack
32        cmpl $'(', %al
33        jne fp
34        mov $f, %rdi
35    p: call puts
36        xor %rax, %rax
37        ret
38
```

Execute Mode, Version, Inputs & Arguments

GCC 11.1.0

☐ Interactive

Stdin Inputs

Execute

Result

CPU Time: 0.00 sec(s), Memory: 1236 kilobyte(s)

compiled and executed in 0.777 sec(s)

correct

Activate Windows

Go to Settings to activate Windows.

Note: Please check our [documentation](#), or [Youtube channel](#) for more details

Type here to search

59°F Mostly cloudy 5:25 PM 3/21/2023

# CALL

- Saves procedure linking information on the stack and branches to the called procedure specified using the target operand.
- The target operand specifies the address of the first instruction in the called procedure.
- Types: Near Call; Far Call; Inter-privilege-level far call; Task switch.
- `call printf`

# RET

- Transfers program control to a return address located on the top of the stack.
- The address is usually placed on the stack by a CALL instruction, and the return is made to the instruction that follows the CALL instruction.
- Types: Near return; Far return; Inter-privilege-level far return.
- `ret`

# Task – compute expression

- Compute:  $ax^n + by^m$
- Use function to compute  $z^i$
- Parameters:
  - `%rbx=z`,
  - `%rcx=i`,
  - return value `%rax=zi`

```
# function pow=x^n
# %rbx=x, %ecx=n, %rax=pow
```

```
    .data
x:    .long 3
n:    .long 9
z:    .quad 0
f:    .asciz "%ld\n"
    .text
# pow(x,n)
pow:  mov $1, %rax
      inc %ecx
l:    loop next
      ret
      next: imul %rbx
      jmp l
```

```
# end pow
```

```
.global main
main:  xor %ebx, %ebx
      mov x, %ebx
      mov n, %ecx
      call pow
      mov %rax, z
# print result
      mov $f, %rdi
      movl z, %esi
      xor %rax, %rax
      call printf
      xor %rax, %rax
      ret
```

# How it works

Online Assembler (GCC) Compile

jdoodle.com/compile-assembler-gcc-online/

JDoodle

Sign In

## Online Assembler - GCC Compiler IDE

```
1 # compute z=x^n
2 # with fuction pow
3 # %rbx=x, %ecx=n
4 # result %rax
5 .data
6 x: .long 3
7 n: .long 9
8 z: .quad 0
9 f: .asciz "%ld\n"
10 .text
11
12 - pow:
13     mov $1, %rax
14     inc %ecx
15 - l:   loop next
16     ret
17 - next: imul %rbx
18         jmp l
19
20 .global main
21 - main:
22     xor %ebx, %ebx
23     mov x, %ebx
24     mov n, %ecx
25     call pow
26     mov %rax, z
27
28     mov $f, %rdi
29     movl z, %esi
30     xor %rax, %rax
31     call printf
32     xor %rax, %rax
33     ret
34
```

Execute Mode, Version, Inputs & Arguments

GCC 11.1.0

☐ Interactive

Stdin Inputs

Execute

...

Result

CPU Time: 0.00 sec(s). Memory: 1440 kilobyte(s)

compiled and executed in 1.263 sec(s)

19683

Activate Windows  
Go to Settings to activate Windows.

Type here to search

59°F Mostly cloudy 6:04 PM 3/21/2023

**# compute  $z=ax^n+by^m$**

```
.data
a: .long 2
x: .long 4
n: .long 3
b: .long 3
y: .long 5
m: .long 2
z: .quad 0
w: .quad 0
f: .asciz "%ld\n"

pow: ...
```

*first call pow= $ax^n$*

```
.text
.global main
main:
    xor %ebx, %ebx
    mov x, %ebx
    mov n, %ecx
    call pow
    xor %ebx, %ebx
    mov a, %ebx
    imul %rbx
    mov %rax, w
```

*second call pow= $by^m$*

*print result*

```
xor %ebx, %ebx
mov y, %ebx
mov m, %ecx
call pow
xor %ebx, %ebx
mov b, %ebx
imul %rbx
add w, %rax
mov %rax, z

mov $f, %rdi
mov z, %rsi
xor %rax, %rax
call printf
xor %rax, %rax
ret
```



# How it works

Online Assembler (GCC) Compile x +

jdoodle.com/compile-assembler-gcc-online/

```
37  xor %ebx, %ebx
38  mov b, %ebx
39  imul %rbx
40  add w, %rax
41  mov %rax, z
42
43  mov $f, %rdi
44  mov z, %rsi
45  xor %rax, %rax
46  call printf
47  mov %rax, %rax
```

Execute Mode, Version, Inputs & Arguments

GCC 11.1.0 ▾ ☐ Interactive Stdin Inputs

Result

CPU Time: 0.00 sec(s), Memory: 1356 kilobyte(s) compiled and executed in 0.845 sec(s)

203

Activate Windows  
Go to Settings to activate Windows

Type here to search

59°F Mostly cloudy 6:25 PM 3/21/2023

# Modular software design

- Top-down approach, hierarchical scheme of modules (functions)
- Bottom-up design, implementation of basic libraries
- How to pass parameters into function?
- Where to allocate local variables?

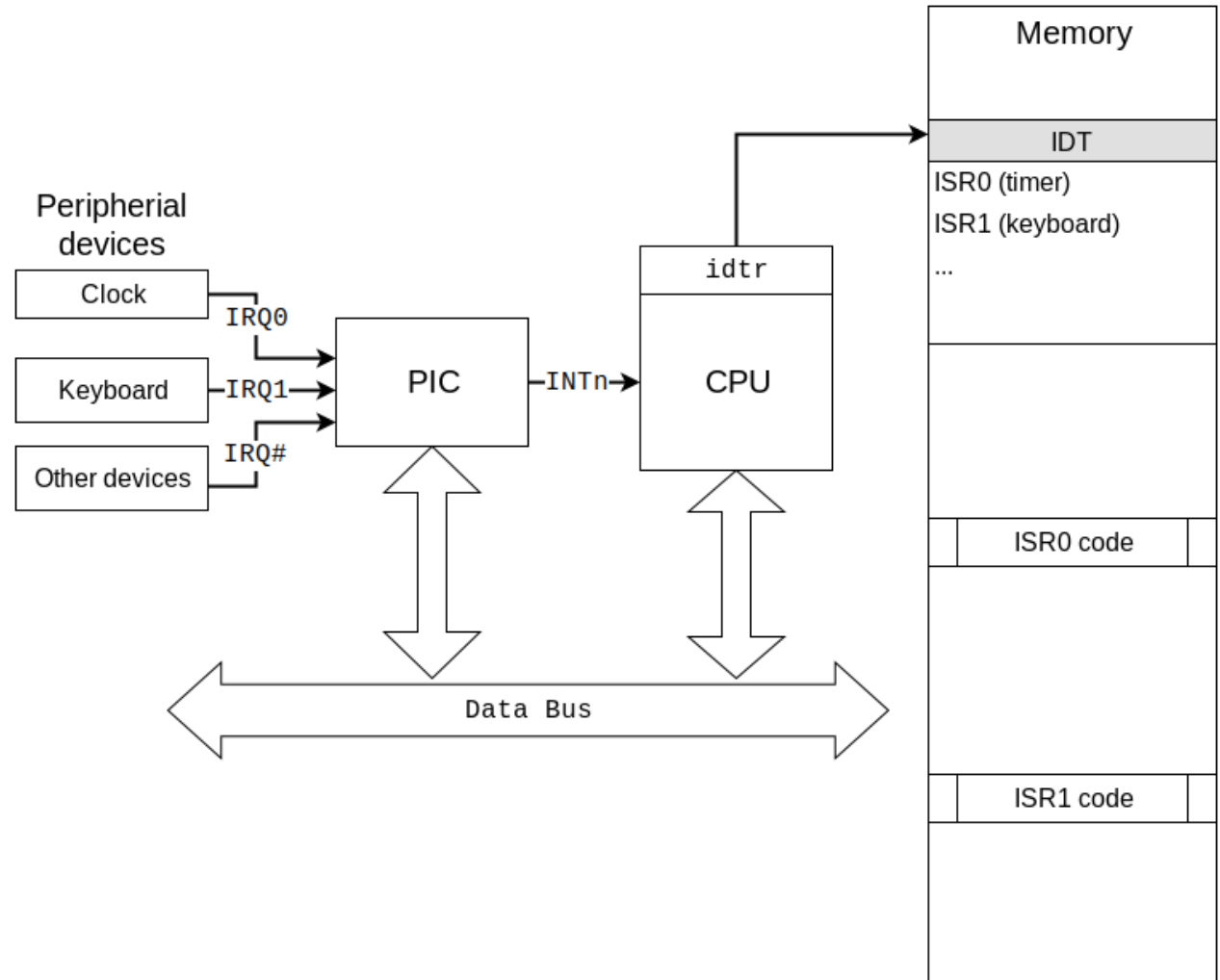
# Interrupt procedure

- Store current processor state and the next instruction address in stack
- Load new values from interrupt vector
- After handling an interrupt – return: restore from stack processor state and instruction pointer

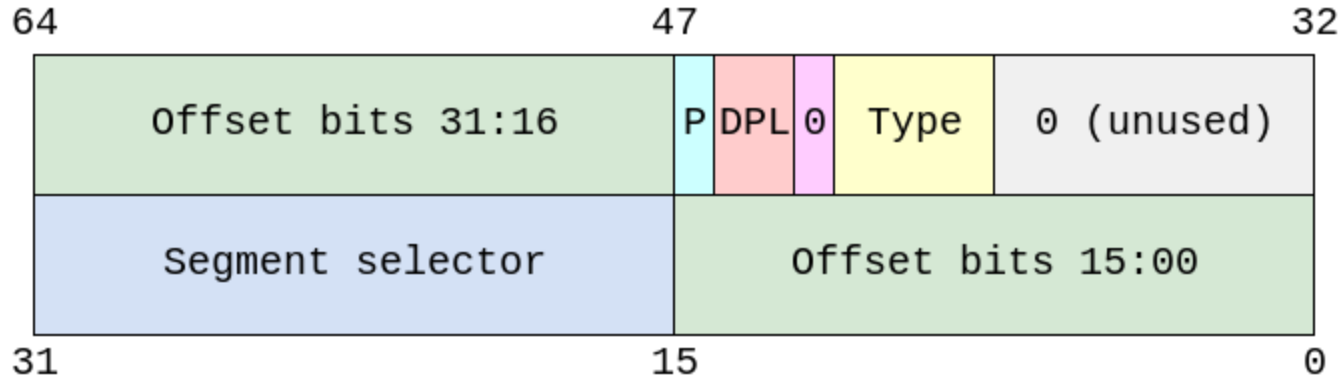
# Interrupt handling components

- Interrupt types: hardware, software (int), exceptions
- Interrupt descriptor table (IDT) indexed by interrupt number
- Programmable interrupt controller (PIC)
- Interrupt Service Routines (ISRs) - OS

# Interrupt processing configuration



# Interrupt descriptor table



- P - Segment present
- DPL - Descriptor privilege level

# Drivers design

- Asynchronous work of devices
- Structure of hardware registers – input/output ports
- Hardware interrupts
- Basic entry points of a driver: function call from OS, interrupt handler