Computer Architecture and Assembly Language

Practical Session 3

Advanced Instructions – division

DIV r/m - unsigned integer division

IDIV r/m - signed integer division

Dividend	Divisor	Quotient	Remainder
AX	*r/m8*	AL	AH
DX:AX	*r/m16*	AX	DX
EDX:EAX	*r/m32*	EAX	EDX

DIV r/m8

```
mov   ax,0083h      ; dividend
mov   bl, 2h        ; divisor
DIV   bl            ; al = 41h, ah = 01h
                    ; quotient is 41h, remainder is 1
```

DIV r/m16

```
mov   dx,0          ; clear dividend, high
mov   ax, 8003h     ; dividend, low
mov   cx, 100h      ; divisor
DIV   cx            ; ax = 0080h, dx = 0003h
                    ; quotient = 80h, remainder = 3
```

Advanced Instructions – shift

<instruction> r/m8(16,32) 1/CL/imm8

SHL, SHR – Bitwise Logical Shifts on first operand

- number of bits to shift is given by second operand
- vacated bits are filled with zero
- (last) shifted bit enters the Carry Flag

Note: shift indeed performs division / multiplication by 2

Example:

```
mov CL , 3
mov AL ,10110111b          ; AL = 10110111b
shr  AL, 1                 ; shift right 1 bit →AL = 01011011b, CF = 1
shr AL, CL                 ; shift right 3 bits →AL = 00001011b, CF = 0
```

SAL, SAR – Bitwise Arithmetic Shift on first operand

- vacated bits are filled with zero for SAL
- vacated bits are filled with copies of the **original high bit** of the source operand for **SAR**

Example:

```
mov CL , 3
mov AL ,10110111b
sar  AL, 1                 ; shift right 1 →AL = 11011011b
sar AL, CL                 ; shift right 3 →AL = 11111011b
```

Advanced Instructions - rotate

<instruction> r/m8(16,32) 1/CL/imm8

ROL, ROR – bitwise rotate (i.e. moves round) on the first operand

Example:

```
mov  CL, 3
mov BH ,10110111b        ; BH = 10110111b
rol BH, 01               ; rotate left 1 bit → BH = 01101111b
rol BH, CL               ; rotate left 3 bits → BH = 01111011b
```

RCL, RCR –bitwise rotate on first operand and Carry Flag

Example:

```
mov BH ,10110111b        ; BH = 10110111b , CF = 0
rcl BH, 01               ; rotate left 1 bit with CF → BH = 01101110b  , CF = 1
```

Advanced Instructions - loop

LOOP, LOOPE, LOOPZ, LOOPNE, LOOPNZ – loop with counter (CX or ECX)

Example:

```
mov ax, 1
mov cx, 3
my_ loop:
        add ax, ax
        loop my_ loop, cx
```

1. decrements its counter register (in this case it is CX register)

2. if the counter does not become zero as a result of this operation, it jumps to the given label

LOOPE ≡ LOOPZ: jumps if the counter ≠ 0 **and** Zero Flag = 1

LOOPNE ≡ LOOPNZ: jumps if the counter ≠ 0 **and** Zero Flag = 0

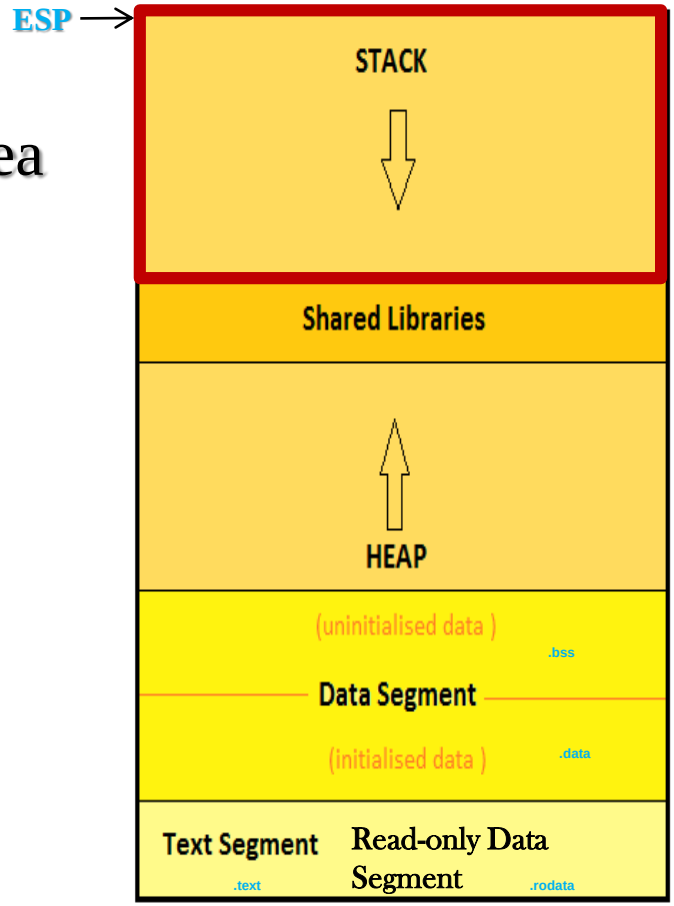
Note: LOOP instruction does not set any flags

Note: if a counter is not specified explicitly, the BITS setting dictates which is used.

The BITS directive specifies whether NASM should generate code designed to run on a processor operating in 16-bit mode, or code designed to run on a processor operating in 32-bit mode. The syntax is BITS 16 or BITS 32.

Stack

- **STACK** is temporary storage memory area
- every cell in stack is of size 2 / 4 bytes
- **ESP** register points to the top of stack
- stack addresses go from **high to low**



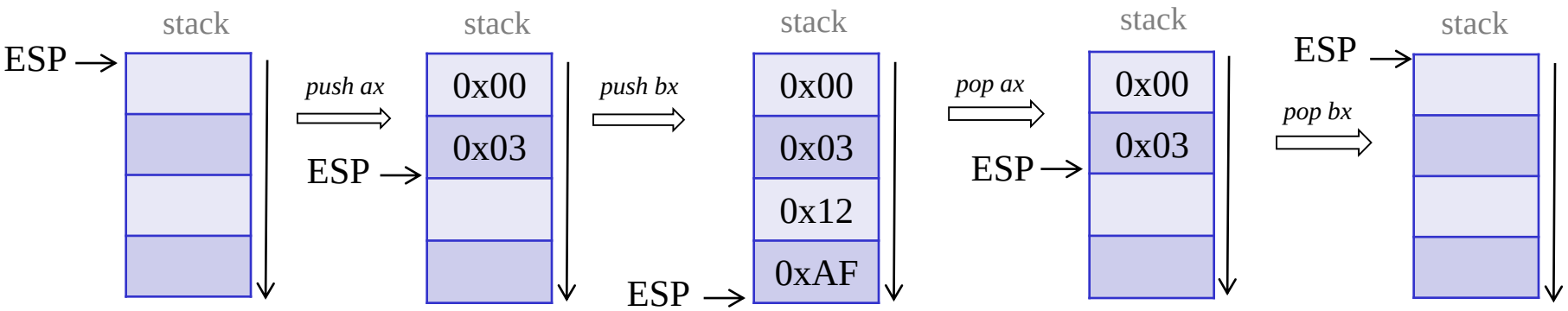
Stack Operations

- **PUSH -** push data on stack
  - decrements ESP by 2 / 4 bytes (according to the operand size)
  - stores the operand value at ESP address on stack (in Little Endian manner)

- **POP** - load a value from the stack
  - reads the operand value at ESP address on stack (in Little Endian manner)
  - increment ESP by 2 / 4 bytes (according to operand size)

Example:

```
mov ax, 3
mov bx, 0x12AF
push ax
push bx
pop ax          ; ax = 0x12AF
pop bx          ; bx = 3
```



Stack Operations

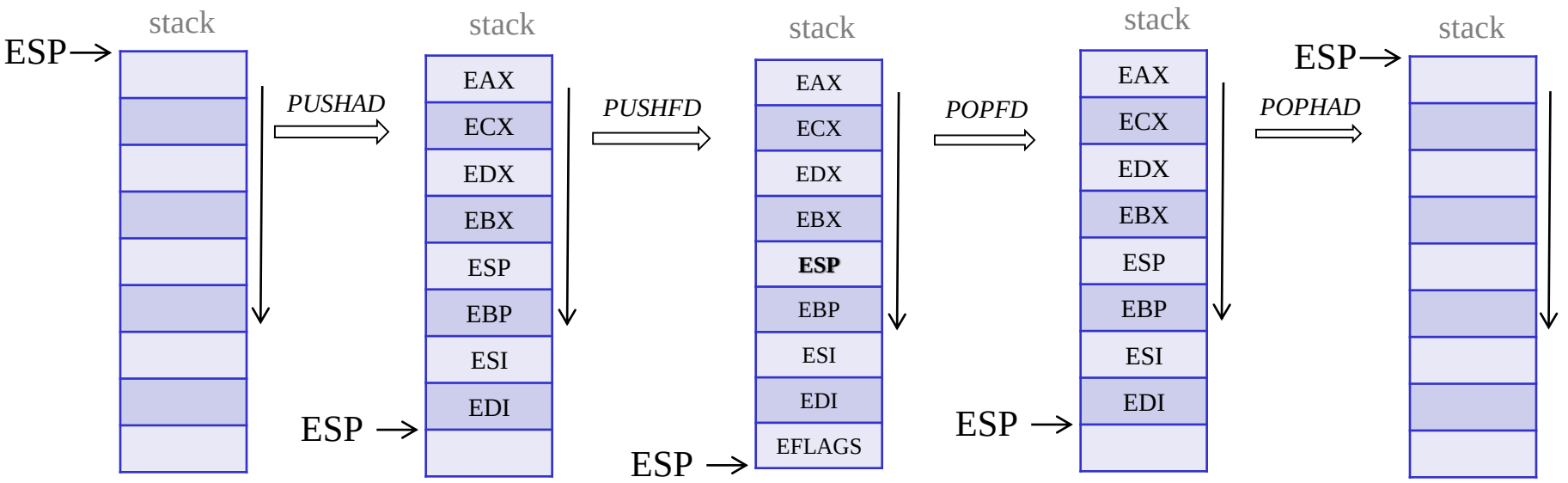
- **PUSHAD** (push all double) - pushes EAX, ECX, EDX, EBX, **ESP**, EBP, ESI, and EDI onto the stack

  original ESP value before PUSHAD

- **PUSHFD** (push flags double) - push flags register onto the stack

- **POPAD** (pop all double) - pop a dword from the stack into each one of (successively) EDI, ESI, EBP, nothing (placeholder for ESP), EBX, EDX, ECX, and EAX

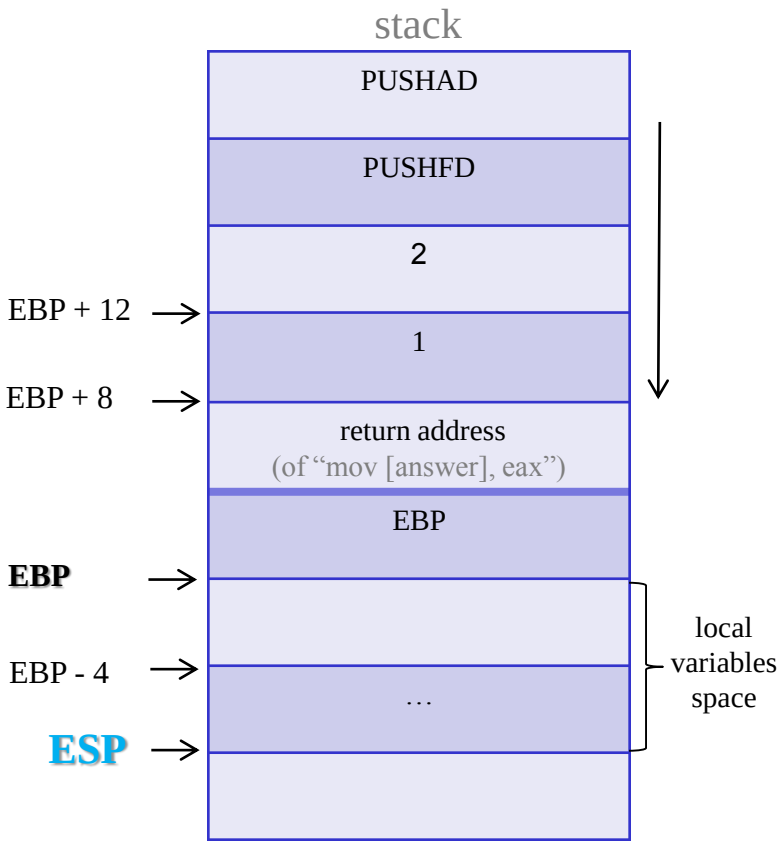
- **POPFD** (pop flags double) - pop a dword and stores it in EFLAGS



Function Calls - Stack Frame

```
main:          ; caller code
pushad                      ; backup registers
pushfd                      ; backup flags register
push dword 2                ; push argument #2
push dword 1                ; push argument #1
CALL myFunc                 ; call the function → the address of the next instruction of caller
                            ; (i.e. return address) is pushed automatically onto the stack
mov [answer], eax           ; retrieve return value from EAX
add esp, <argumentsSize>    ; "delete" function arguments (in our case argumentsSize = 2 * dword = 8 bytes)
popfd                       ; restore flags register
popad                       ; restore registers
```

```
myFunc:        ; callee code
push ebp                    ; backup EBP
mov ebp, esp                ; reset EBP to the current ESP
sub esp, <localsSize>       ; allocate space for locals
mov ebx, [ebp+12]           ; get second argument
mov ecx, [ebp+8]            ; get first argument
mov [ebp-4], ebx            ; initialize a local
...                         ; function code
mov eax, <return value>     ; put return value into EAX
mov esp, ebp                ; move EBP to ESP
pop ebp                     ; restore old EBP
RET                         ; return from the function
```



Gdb-GNU Debugger

- Run Gdb from the console by typing
  gdb executableFileName
- Adding breaking points by typing:
  break label
- Start debugging by typing:
  run parameters (argv)

- **si** – one step forward
- **c** – continue to run the code until the next break point.
- **q** – quit gdb
- **p $eax** – prints the value in eax
- **x $esp+4** – prints the address in esp + 4 hexadecimal and the value (dword) that stores in this address. It is possible to use label instead of esp.
  Type **x** again will print the next dword in memory.

Assignment 1

Task 1

Converting "Number in base 4" to "Hexadecimal".

You can get up to 32 binary digits.

Examples:

Input: 213022013 ➔ Output: 27287

Input: 221001➔ Output: A41

Task 2

Practice parameters passing from assembly to C and vice versa.

Write a C program that:

- read long long (64 bits) unsigned number x (in hexadecimal) from user
- read one (32 bits) integer numOfRounds (in decimal) from user
- call ‘calc_func (long long *x, int numOfRounds)’ written in ASSEMBLY

Write an assembly function ‘calc_func' that runs the following loop:

- calculates a "descriptive number" y
- call 'compare (long long * x, long long * y)' C function to compare x and y
- if x == y, print x and break the from the loop
- if x != y
  - if the loop was executed less than numOfRounds times
    - set x <– y and continue
  - else
    - print y and return

Example:

>1AF2345FF23B0001 // x - input
>1 // 1 round
>3222110000110003 // y - output

Task 2

- A single round execution explanation: x = 1AF2345FF23B0001
- digit - counter of appearances of the digit in x:

  0 - 3
  1 - 2
  2 - 2
  3 - 2
  4 - 1
  5 - 1
  6 - 0
  7 - 0 ⇨ We get y = 3222110000110003
  8 - 0
  9 - 0
  A - 1
  B - 1
  C - 0
  D - 0
  E - 0
  F - 3

main1.c file for task 1

```
#include <stdio.h>
#define  BUFFER_SIZE      (128)
extern int my_func(char* buf);

int main(int argc, char** argv)
{
  char buf[BUFFER_SIZE];

  fflush(stdout);
  fgets(buf, BUFFER_SIZE, stdin);

  my_func(buf);

  return 0;
}
```

The skeleton of assembly programs in both tasks

```
section   .rodata
LC0:
          DB        "%s", 10, 0         ; Format string

section .bss
LC1:
          RESB      32

section .text
          align 16
          global my_func
          extern printf

my_func:
          push      ebp
          mov       ebp, esp            ; Entry code - set up ebp and esp
          pusha                         ; Save registers

          mov ecx, dword [ebp+8]        ; Get argument (pointer to string)

;       Your code should be here...

          push      LC1                 ; Call printf with 2 arguments: pointer to str
          push      LC0                 ; and pointer to format string.
          call      printf
          add       esp, 8              ; Clean up stack after call

          popa                          ; Restore registers
          mov       esp, ebp            ; Function exit code
          pop       ebp
          ret
```