Computer Architecture and System Programming Laboratory

TA Session 2

MOV, ADD, SUB, INC, DEC, NOT, NEG,OR, AND, CMP
JUMP (unconditional, unconditional)
static memory allocation – define, reserve
addressing mode
Assignment 0

MOV - Move Instruction – copies source to target

<u>mov reg8/mem8(16,32),reg8/imm8(16,32)</u>

(copies content of register / immediate (source) to register / memory location (destination))

<u>mov reg8(16,32),reg8/mem8(16,32)</u>

(copies content of register / memory location (source) to register (destination))

mov eax, 0x2334AAFF   *mov [buffer], ax*   *mov **word** [var], 2*

reg32 imm32   mem16 reg16   mem16 imm16

Note that NASM doesn’t remember the types of variables you declare . It will deliberately remember nothing about the symbol var except where it begins, and so you must explicitly code *<u>mov **word** [var], 2.</u>*

Basic Arithmetical Instruction

<instruction> reg8/mem8(16,32),reg8/imm8(16,32)

(source - register / immediate, destination - register / memory location)

<instruction> reg8(16,32),reg8/mem8(16,32)

(source - register / immediate, destination - register / memory location)

ADD - add integers

Example:

add AX, BX ;(AX gets a value of AX+BX)

SUB - subtract integers

Example:

sub AX, BX ;(AX gets a value of AX-BX)

ADC - add integers with carry

(value of Carry Flag)

Example:

adc AX, BX ;(AX gets a value of AX+BX+CF)

SBB - subtract with borrow

(value of Carry Flag)

Example:

sbb AX, BX ;(AX gets a value of AX-BX-CF)

Basic Arithmetical Instruction

<instruction> reg8/mem8(16,32)
(register / memory location)

INC - increment integer

Example:
inc AX ;(AX gets a value of AX+1)

DEC - increment integer

Example:
dec byte [buffer] ;([buffer] gets a value of [buffer] -1)

Basic Logical Instructions

<instruction> reg8/mem8(16,32)
(register / memory location)

NOT – one's complement negation – inverts all the bits

Example:
mov al, 11111110_b
not al ;(AL gets a value of 00000001_b)
;($11111110_b + 00000001_b = 11111111_b$)

NEG – two's complement negation – inverts all the bits, and adds 1

Example:
mov al, 11111110_b
neg al ;(AL gets a value of not(11111110_b)+1=00000001_b+1=00000010_b)
;($11111110_b + 00000010_b = 100000000_b = 0$)

Basic Logical Instructions

<instruction> reg8/mem8(16,32),reg8/imm8(16,32)
(source - register / immediate, target - register / memory location)

<instruction> reg8(16,32),reg8/mem8(16,32)
(source - register / immediate, target - register / memory location)

OR – bitwise or – bit at index i of the target gets '1' if bit at index i of source or destination are '1'; otherwise '0'

Example:
mov al, 11111100 $_b$
mov bl, 00000010$_b$
or AL, BL ;(AL gets a value 11111110$_b$)

AND– bitwise and – bit at index i of the target gets '1' if bits at index i of both source and destination are '1'; otherwise '0'

Example:
or AL, BL ;(with same values of AL and BL as in previous example, AL gets a value 0)

CMP – Compare Instruction – compares integers

CMP performs a 'mental' subtraction - **affects the flags** as if the subtraction had taken place, but does not store the result of the subtraction.

cmp reg8/mem8(16,32),reg8/imm8(16,32)
(source - register / immediate, target - register / memory location)

cmp reg8(16,32),reg8/mem8(16,32)
(source - register / immediate, target - register / memory location)

Examples:

mov al, 10
mov bl, 15
cmp al, bl ;(ZF (zero flag) gets a value 0)

mov al, 10
mov bl, 10
cmp al, bl ;(ZF (zero flag) gets a value 1)

J<Condition> – conditional jump

J<Condition> execution is transferred to the target instruction only if the specified condition is satisfied. Usually, the condition being tested is the result of the last arithmetic or logic operation.

```
int x=1;
while (x < 10)
      x++;
```

<u>Example</u>:

```
mov eax, 1
inc_again:
        cmp eax, 10
        je end_of_loop     ; if eax = = 10, jump to end_of_loop
        inc eax
        jmp inc_again      ; go back to loop
end_of_loop:
```

Instruction	Description	Flags
JO	Jump if overflow	OF = 1
JNO	Jump if not overflow	OF = 0
JS	Jump if sign	SF = 1
JNS	Jump if not sign	SF = 0
JE **JZ**	Jump if equal Jump if zero	ZF = 1
JNE **JNZ**	Jump if not equal Jump if not zero	ZF = 0
JB **JNAE** **JC**	Jump if below Jump if not above or equal Jump if carry	CF = 1
JNB **JAE** **JNC**	Jump if not below Jump if above or equal Jump if not carry	CF = 0
JBE **JNA**	Jump if below or equal Jump if not above	CF = 1 or ZF = 1
JA **JNBE**	Jump if above Jump if not below or equal	CF = 0 and ZF = 0
JL **JNGE**	Jump if less Jump if not greater or equal	SF <> OF
JGE **JNL**	Jump if greater or equal Jump if not less	SF = OF
JLE **JNG**	Jump if less or equal Jump if not greater	ZF = 1 or SF <> OF
JG **JNLE**	Jump if greater Jump if not less or equal	ZF = 0 and SF = OF
JP **JPE**	Jump if parity Jump if parity even	PF = 1
JNP **JPO**	Jump if not parity Jump if parity odd	PF = 0
JCXZ **JECXZ**	Jump if CX register is 0 Jump if ECX register is 0	CX = 0 ECX = 0

Note that the list above is partial. The full list can be found here.

RES<size> – declare uninitialized data

Pseudo-instruction	<size> filed	<size> value
RESB	byte	1 byte
RESW	word	2 bytes
RESD	double word	4 bytes
RESQ	quadword	8 bytes
REST	tenbyte	10 bytes
RESDQ	double quadword	16 bytes
RESO	octoword	16 bytes

indeed reserved memory contain 0's

RES<size> - declare uninitialized storage space

Examples:

```
buffer:      resb 64        ; reserve 64 bytes
wordVar:     resw 1         ; reserve a word
realArray:   resq 10        ; array of ten quadwords (8 bytes)
```

Note: you can **not make any assumption** about content of a storage space cells.

d<size> – declare initialized data

Pseudo-instruction	<size> filed	<size> value
DB	byte	1 byte
DW	word	2 bytes
DD	double word	4 bytes
DQ	quadword	8 bytes
DT	tenbyte	10 bytes
DDQ	double quadword	16 bytes
DO	octoword	16 bytes

```
var: db  0x55               ; define a variable ‘var’ of size byte, initialized by 0x55
var: db  0x55,0x56,0x57     ; three bytes in succession (array)
var: dw 0x1234              ; 0x34 0x12
var: dd  0x123456           ; 0x56 0x34 0x12 0x00 – complete to dword
var: db ‘A'                 ; 0x41
var: dw  ‘AB‘               ; 0x41 0x42
var: dw  ‘ABC'              ; 0x41 0x42 0x43 0x00 – complete to word
var: db  ‘AB',10            ; 0x41 0x42 0xA (string with ‘\n’)
var: db  ‘AB’,10,0          ; 0x41 0x42 0xA 0x00 (null terminated string with ‘\n’)
```

Addressing Mode

specifies how to calculate effective memory address

Up to two registers and one 32-bit signed immediate can be added together to compute a memory address. One of registers can be optionally pre-multiplied by 2, 4, or 8.

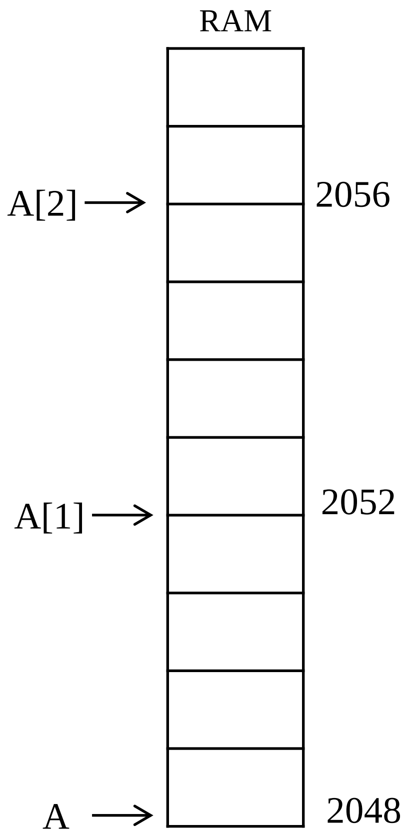
Examples of **right** usage:

```
mov eax, [ebx]        ; move 4 bytes at the address contained in EBX int EAX
mov [var], ebx        ; move the contents of EBX into 4 bytes at address "var"
mov eax, [esi-4]      ; move 4 bytes at address ESI+(-4) into EAX
mov [esi+eax], cl     ; move the contents of CL into address ESI+EAX
mov edx, [esi+4*ebx]      ; move 4 bytes at address ESI+4*EBX into EDX
mov dword [myArray + ebx*4 + eax], ecx      ; move the content of ECX
                          ; to 4 bytes at address myArray + EBX*4 + EAX
```

Examples of **wrong** usage:

```
mov eax, [ebx-ecx]        ; can only add register values
mov [eax+esi+edi], ebx    ; at most 2 registers in address computation
```

```
int A[5];
for (int i=0;i<5; i++)
   A[i]++;
```



Addressing Mode

specifies how to calculate effective memory address

Up to two registers and one 32-bit signed immediate can be added together to compute a memory address. One of registers can be optionally pre-multiplied by 2, 4, or 8.

Examples of **right** usage:

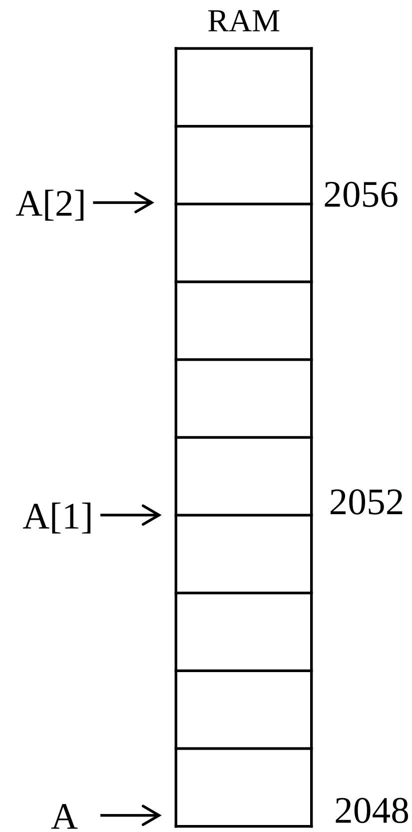
```
mov eax, [ebx]          ; move 4 bytes at the address contained in EBX int EAX
mov [var], ebx          ; move the contents of EBX into 4 bytes at address "var"
mov eax, [esi-4]        ; move 4 bytes at address ESI+(-4) into EAX
mov [esi+eax], cl       ; move the contents of CL into address ESI+EAX
mov edx, [esi+4*ebx]        ; move 4 bytes at address ESI+4*EBX into EDX
mov dword [myArray + ebx*4 + eax], ecx        ; move the content of ECX
                            ; to 4 bytes at address myA
```

Examples of **wrong** usage:

```
mov eax, [ebx-ecx]          ; can only add register valu
mov [eax+esi+edi], ebx      ; at most 2 registers in add
```

```
int A[5];
for (int i=0;i<5; i++)
   A[i]++;
```

```
A: resd 5
mov ebx, 0
startLoop:
    cmp ebx, 5
    je endLoop
    inc dword [A+4*ebx]
    inc ebx
    jmp startLoop
endLoop:
...
```



mainAssignment0.c

```c
#include <stdio.h>
#define MAX_LEN 100                           /* maximal input string size */

extern int do_Str(char*);

int main(int argc, char** argv) {

   char str_buf[MAX_LEN];
   int counter = 0;

   fgets(str_buf, MAX_LEN, stdin);            /* get user input string */
   counter = do_Str(str_buf);                 /* call your assembly function */
   printf("%s%d\n", str_buf, counter);        /* print result string and counter */

   return 0;
}
```

Assignment 0

You get a simple program that receives a string from user.
Then, it calls to a function (that you'll implement in assembly) that receives this string as an argument and should do the following:

- convert 'TAB' (\t) and 'SPACE' (' ') into '_'
- count (and return) the number of converted characters

The characters conversion should be <u>in-place</u>.

<u>Example</u>:

```
> Assignment0.out
42 aB	cDafg!
42_aB_cDafg!
2
```

asmAssignment0.s

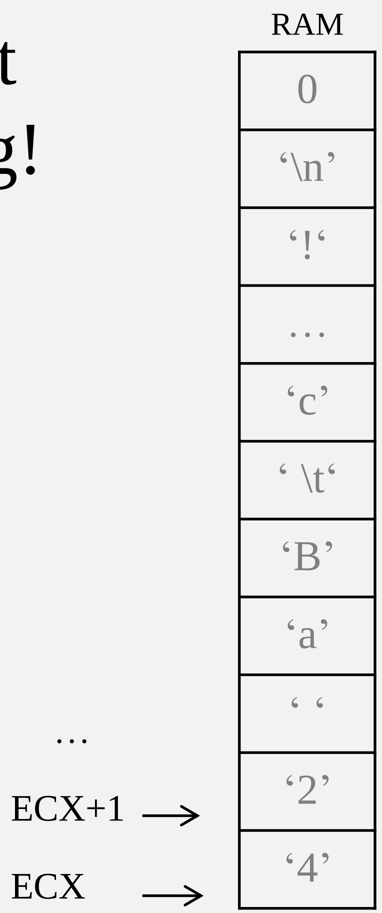
```
section .data                 ; we define (global) initialized variables in .data section
      an: dd 0                ; an is a local variable of size double-word, we use it to count the string characters

section .text                 ; we write code in .text section
      global do_Str           ; 'global' directive causes the function do_Str(...) to appear in global scope

do_Str:                       ; do_Str function definition - functions are defined as labels
   push ebp                   ; save Base Pointer (bp) original value
   mov ebp, esp               ; use Base Pointer to access stack contents (do_Str(...) activation frame)
   pushad                     ; push all significant registers onto stack (backup registers values)
   mov ecx, dword [ebp+8]     ; get function argument on stack
                              ; now ecx register points to the input string
   yourCode:                  ; use label to build a loop for treating the input string characters

        ; your code goes here...

        inc ecx               ; increment ecx value; now ecx points to the next character of the string
        cmp byte [ecx], 0     ; check if next character(character = byte) is zero (i.e. null string termination)
        jnz yourCode          ; if not, keep looping until meet null termination character

   popad                      ; restore all previously used registers
   mov eax,[an]               ; return an (returned values are in eax)
   mov esp, ebp               ; free function activation frame
   pop ebp                    ; restore Base Pointer previous value (to return to the activation frame of main(...))
   ret                        ; returns from do_Str(...) function
```

asmAssignment0.s

```
section .data                        ; we define (global) initialized variables in .data section
      an: dd 0                       ; an is a local variable of size double-word, we use it to count the string characters

section .text                        ; we write code in .text section
      global do_Str                  ; 'global'

do_Str:                              ; do_Str
  push ebp                           ; save Ba
  mov ebp, esp                       ; use Ba
  pushad                             ; push al
  mov ecx, dword [ebp+8]             ; get fun
                                     ; now ec
yourCode:                            ; use lab

        ; your code goes here...

        inc ecx                      ; increme
        cmp byte [ecx], 0            ; check i
        jnz yourCode                 ; if not, k

  popad                              ; restore
  mov eax,[an]                       ; return a
  mov esp, ebp                       ; free fu
  pop ebp                            ; restore
  ret                                ; returns
```

> Assignment0.out
42 aB   cDafg!
42_aB_cDafg!
2

what you need to do:
- manage current character (its address is in ECX)
- if needed, increment 'an' variable which counts whitespaces



Compiling with NASM

To assemble a file, you issue a command of the form

> nasm –f <format> <filename> [-o <output>] [-l listing]

Example:

> nasm -f elf32 asmAssignment0.s -o asmAssignment0.o

NASM creates asmAssignment0.o file that has elf (executable and linkable) format.
To compile asmAssignment0.c with our assembly file we should execute the following command:

> gcc –m32 asmAssignment0.c asmAssignment0.o -o asmAssignment0.out

-m32 option is being used to comply with 32-bit environment.
gcc creates executable file asmAssignment0.out.
In order to run it you should write its name on the command line:

> ./asmAssignment0.out