

Computer Architecture and Assembly Language

Practical Session 4

• valid characters in labels are: letters, numbers, _, \$, #, @, ~, ., and ?

• first character can be: letter, _, ?, and .

Local Labels Definition

A label beginning with a single period (.) is treated as a **local label**, which means that it is associated with the previous non-local label.

Example:

label1:

mov eax, 3

.loop:

dec eax

jne .loop

ret

(this is indeed **label1.loop**)

label2:

mov eax, 5

.loop:

dec eax

jne .loop

ret

(this is indeed **label2.loop**)

Each JNE instruction jumps to the closest .loop, because the two definitions of .loop are kept separate.

Assembly program with no .c file usage – sample.s

```
section .data
    numeric:    DD 0x12345678
    string:     DB 'abc'
    answer:     DD 0

section .text
    global _start                ;entry point (main)

_start:

    pushad                    ; backup registers
    push dword 2               ; push argument #2
    push dword 1               ; push argument #1
    CALL myFunc                 ; call the function myFunc
returnAddress:
    mov [answer], eax          ; retrieve return value from EAX
    add esp, 8                 ; "delete" function arguments
    popad

    mov ebx,0                   ; exit program
    mov eax,1
    int 0x80

myFunc:
    push ebp                   ; save previous value of ebp
    mov ebp, esp               ; set ebp to point to myFunc frame
    mov eax, dword [ebp+8]      ; get function argument #1
    mov ebx, dword [ebp+12]     ; get function argument #2

myFunc_code:
    add eax, ebx                ; eax = 3

returnFrom_myFunc:
    mov esp, ebp               ; "delete" local variables of myFunc
    pop ebp                    ; restore previous value of ebp
    RET                        ; return to the caller
```

GNU Linker

ld links together compiled assembly without using .c main file

```
> nasm -f elf sample.s -o sample.o
```

```
> ld -m elf_i386 sample.o -o sample
```

```
> sample
```

or with gdb debugger

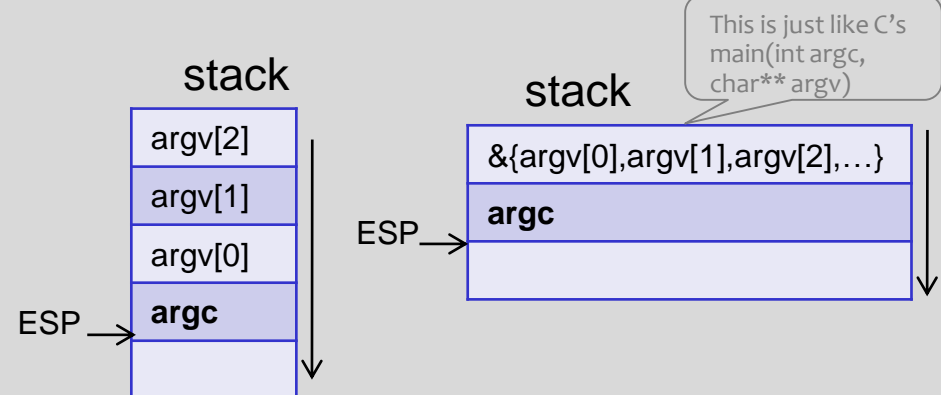
```
> gdb sample
```

Command-line arguments

ld(_start)

vs.

gcc (main)



Producing a listing file: > nasm -f elf sample.s -l sample.lst



. The **first column** (from the left) is the **line number** in the listing file

.The **second column** is the **relative address** of where the code will be placed in memory
. each section starts at relative address 0

.The **third column** is the **compiled code**

.The **forth column** is the **original code**

. Labels do not create code; they are a way to tell assembler that those locations have symbolic names.

. '**CALL myFunc**' is compiled to **opcode E8** followed by a **4-byte target address**, relative to the next instruction after the call.
→ address of myFunc label = 0x1F
→ address of the next instruction after the call (i.e. 'mov [answer], eax') is 0xA
→ $0x1F - 0xA = 0x15$, and we get exactly the binary code written here '**E815000000**'

0x15 is how many bytes EIP should jump forward

```
1
2          section .data
3 00000000 78563412      numeric:  DD 0x12345678
4 00000004 616263      string:   DB 'abc'
5 00000007 00000000      answer:  DD 0
6
7          section .text
8          global _start
9
10         _start:
11
12 00000000 60          pushad
13 00000001 6A02      push dword 2
14 00000003 6A01      push dword 1
15 00000005 E815000000 CALL myFunc
16
17         returnAddress:
18 0000000A A307000000 mov [answer], eax
19 0000000F 83C408    add esp, 8
20 00000012 61          popad
21 00000013 BB00000000 mov ebx, 0
22 00000018 B801000000 mov eax, 1
23 0000001D CD80      int 0x80
24
25         myFunc:
26 0000001F 55          push ebp
27 00000020 89E5      mov ebp, esp
28 00000022 8B4508    mov eax, dword [ebp+8]
29 00000025 8B5D0C    mov ebx, dword [ebp+12]
30
31         myFunc_code:
32 00000028 01D8      add eax, ebx
33
34         returnFrom_myFunc:
35 0000002A 89EC      mov esp, ebp
36 0000002C 5D          pop dword ebp
37 0000002D C3          ret
```

```
Breakpoint 1, 0x08048080 in _start () executable
(gdb) x /1xg $eip
0x8048080 <_start>: 0x0015e8016a026a60
(gdb)
0x8048088 <_start+8>: 0x83080490b7a30000
(gdb)
0x8048090 <returnAddress+6>: 0x00000000bb6108c4
(gdb)
0x8048098 <returnAddress+14>: 0x5580cd00000001b8
(gdb)
0x80480a0 <myFunc+1>: 0x0c5d8b08458be589
(gdb)
0x80480a8 <myFunc_code>: 0x0000c35dec89d801
```

Debugging with GDB guide

- examining memory
- examining data

```
section .data
    numeric:    DD 0x12345678
    string:     DB 'abc'
    answer:     DD 0

section .text
    global _start

_start:

    pushad
    push dword 2
    push dword 1
    CALL myFunc
returnAddress:
    mov [answer], eax
    add esp, 8
    popad

    mov ebx,0
    mov eax,1
    int 0x80

myFunc:
    push ebp
    mov ebp, esp
    mov eax, dword [ebp+8]
    mov ebx, dword [ebp+12]

myFunc_code:
    add eax, ebx

returnFrom_myFunc:
    mov esp, ebp
    pop ebp
    ret
```

```
(gdb) break _start
Breakpoint 1 at 0x8048080
(gdb) break myFunc
Breakpoint 2 at 0x80480a2
(gdb) break myFunc_code
Breakpoint 3 at 0x80480a8
(gdb) run
Starting program: /users/studs/msc/sadetsky/PhD/WO...

Breakpoint 1, 0x08048080 in _start ()
(gdb) p /x numeric
$1 = 0x12345678
(gdb) p/x (char[4])numeric
$2 = {0x78, 0x56, 0x34, 0x12}
(gdb) p/x string
$3 = 0x636261
(gdb) p/x (char[4])string
$4 = {0x61, 0x62, 0x63, 0x0}
(gdb) p $esp
$5 = (void *) 0xffffd640
(gdb) si
0x08048081 in _start ()
(gdb) p $esp
$6 = (void *) 0xffffd620
(gdb) si
0x08048083 in _start ()
(gdb) x $esp
0xffffd61c: 0x00000002
(gdb) si
0x08048085 in _start ()
(gdb) x $esp
0xffffd618: 0x00000001
(gdb) si
0x0804809f in myFunc ()
(gdb) x $esp
0xffffd614: 0x0804808a
(gdb) x returnAddress
0x804808a <returnAddress>: 0x0490b7a3
```

print 'numeric' global variable

numeric into memory – little endian

print 'string' global variable

string into memory – little endian

pushad

$0xffffd640 - 0xffffd620 = 0x20 = 32$
bytes = 8 registers * 4 bytes

push function's arguments
into stack

CALL myFunc

return address