Chapter 8

MACRO FUNCTION

Macro

- Definition : macro is a predefined set of instructions that can easily be inserted wherever needed
- After defined, macro can be used as many times as necessary
- Macro must be defined before of using
- Macro can be used in text section
- There are 2 types of macros : single-line macro and multi-line macro
-

Single – line macro

- Single-line macros are defined using the %define directive.
- Example : %define mulby4(x) shl x, 2
- Use the macro by entering : mulby4 (rax)
- Explain : in the source, which will multiply the contents to the **rax** register by 4 (via shifting two bits).

Multi-Line Macros

- Multi-line macros can include a varying number of lines (including one). The multi-line macros are more useful and the following sections will focus primarily on multi-line macros.
- **Macro Definition** : before using
- Syntax :
  - %macro <name> <number of arguments>
    ; [body of macro]
    %endmacro
  - The arguments can be referenced within the macro by **%<number>**, with **%1** being the first argument, and **%2** the second argument, and so forth.

- In order to use labels, the labels within the macro must be prefixing the label name with a **%%**.
- This will ensure that calling the same macro multiple times will use a different label each time.
- For example, a macro definition for the absolute value function would be as follows:
  - ```
    %macro abs 1
            cmp %1, 0
            jge %%done
            neg %1
    %%done:
    %endmacro
    ```

Using a Macro

- Example : given declaration as follows
  - qVar dq 4
  - Invoke (call) abs macro (twice)
  - mov eax, -3
  - abs eax
  - abs qword [qVar]
- The list file will display the code as follows (for the first invocation):
    ```

- 27 00000000 B8FDFFFFFF mov eax, -3
  28 abs eax
  29 00000005 3D00000000 <1> cmp %1, 0
  30 0000000A 7D02 <1> jge %%done
  31 0000000C F7D8 <1> neg %1
  32 <1> %%done:
  The macro will be copied from the definition into the code, with the appropriate arguments replaced in the body of the macro, *each* time it is used. The **<1>** indicates code copied from a macro definition. In both cases, the **%1** argument was replaced with the given argument; **eax** in this example.

Macro Example

- ; Example Program to demonstrate a simple macro
- ;***
  ; Define the macro
- ; called with three arguments:
- ; aver <lst>, <len>, <ave>

```
%macro aver 3
mov eax, 0
mov ecx, dword [%2] ; length
mov r12, 0
lea rbx, [%1]
```

```
%%sumLoop:
        add eax, dword [rbx+r12*4] ; get list[n]
        inc r12
        loop %%sumLoop
        cdq
        idiv dword [%2]
        mov dword [%3], eax
%endmacro
```

- ;***************************************;
  Data declarations

```
section .data
        ; -----
        ; Define constants
EXIT_SUCCESS equ 0      ; success code
SYS_exit equ 60         ; code for terminate
; Define Data.
section .data
        list1 dd 4, 5, 2, -3, 1
        len1 dd 5
        ave1 dd 0
```

```
        list2 dd 2, 6, 3, -2, 1, 8, 19
        len2 dd 7
        ave2 dd 0
;***************************************section
 .text
global _start
_start:
; Use the macro in the program
        aver list1, len1, ave1 ; 1st, data set 1
        aver list2, len2, ave2
last:
        mov rax, SYS_exit ; exit
        mov rdi, EXIT_SUCCESS ; success
        syscall
```

Functions

- Functions and procedures (i.e., void functions), help break-up a program into smaller parts making it easier to code, debug, and maintain.
- Function calls involve two main actions:
  - Linkage : Since the function can be called from multiple different places in the code, the function must be able to return to the correct place in which it was originally called.
  - Argument Transmission : The function must be able to access parameters to operate on or to return results (i.e., access call-by-reference parameters).

Function Declaration

- A function must be written before it can be used. Functions are located in the code segment. The general format is:
- global <procName>
  <procName>:
  ; function body
  ret
- A function may be defined only once.
- Functions cannot be
- A function definition should be started and ended before the next function's definition can be started.

Linkage

- The linkage is about getting to and returning from a function call correctly. There are two instructions that handle the linkage, call <funcName> and ret instructions.
- The call transfers control to the named function, and **ret** returns control back to the calling routine.
- The call works
  - Push RIP
  - Jump to ***label***
- Ret instruction
  - POP RIP
  - Jump to address

- The function calling or linkage instruction is summarized as follows:

Instruction	Explanation
`call <funcName>`	Calls a function. Push the 64-bit **rip** register and jump to the <***funcName***>.
Examples:	`call printString`
`ret`	Return from a function. Pop the stack into the **rip** register, effecting a jump to the line after the call.
Examples:	`ret`

Argument Transmission

- Argument transmission refers to sending information (variables, etc.) to a function and obtaining a result as appropriate for the specific function.
- Transmitting values to a function is referred to as *call-byvalue.*
- Transmitting addresses to a function is referred to as *call-by-reference.*
- There are various ways to pass arguments to and/or from a function
- Placing values in register
  - Easiest, but has limitations (i.e., the number of registers).
  - Used for first six integer arguments.
  - Used for system calls.

- Globally defined variables
  - Generally poor practice, potentially confusing, and will not work in many cases.
  - Occasionally useful in limited circumstances.
- Putting values and/or addresses on stack
  - No specific limit to count of arguments that can be passed.
  - Incurs higher run-time overhead.
- In general, the calling routine is referred to as the ***caller*** and the routine being called is referred to as the ***callee***.

Parameter Passing

- As noted, a combination of registers and the stack is used to pass parameters to and/or from a function.The first six integer arguments are passed in registers as follows:

Argument Number	Argument Size			
	64-bits	32-bits	16-bits	8-bits
1	rdi	edi	di	dil
2	rsi	esi	si	sil
3	rdx	edx	dx	dl
4	rcx	ecx	cx	cl
5	r8	r8d	r8w	r8b
6	r9	r9d	r9w	r9b

- The seventh and any additional arguments are passed on the stack.

- when the function is completed, the calling routine is responsible for clearing the arguments from the stack
- Instead of doing a series of pop instructions, the stack pointer, **rsp**, is adjusted as necessary to clear the arguments off the stack.
- Since each argument is 8 bytes, the adjustment would be adding [(number of arguments) * 8] to the **rsp**
- For value returning functions, the result is placed in the **A** register based on the size of the value being returned. Specifically, the values are returned as follows:

- The **rax** register may be used in the function as needed as long as the return value is set appropriately before returning.

Return Value Size	Location
byte	**al**
word	**ax**
double-word	**eax**
quadword	**rax**
floating-point	**xmm0**

Register Usage

- some registers are expected to be preserved across a function call. That means that if a value is placed in a *preserved register* or *saved register* and the function must use that register, the original value must be preserved by placing it on the stack, altered as needed, and then restored to its original value before returning to the

Register	Usage
rax	Return Value
rbx	Callee Saved
rcx	4th Argument
rdx	3rd Argument
rsi	2nd Argument
rdi	1st Argument
rbp	Callee Saved
rsp	Stack Pointer
r8	5th Argument
r9	6th Argument
r10	Temporary
r11	Temporary
r12	Callee Saved
r13	Callee Saved

- The temporary registers (**r10** and **r11**) and the argument registers (**rdi**, **rsi**, **rdx**, **rcx**, **r8**, and **r9**) are not preserved across a function call This means that any of these registers may be used in the function without the need to preserve the original value.
- None of the floating-point registers are preserved across a function call

Call Frame

- The items on the stack as part of a function call are referred to as a *call frame* (also referred to as an *activation record* or *stack frame*).
- The possible items in the call frame include:
  - Return address (required).
  - Preserved registers (if any).
  - Passed arguments (if any).
  - Stack dynamic local variables (if any).

- For example, assuming a function call has eight (8) arguments and assuming the function uses **rbx**, **r12**, and **r13** registers (and thus must be pushed), the call frame would be as follows:

...	
<8th Argument>	← rbp + 24
<7th Argument>	← rbp + 16
rip	(return address)
rbp	← rbp
rbx	
r12	
r13	← rsp
...	

Illustration 24: Stack Frame Layout

Red Zone

- In the Linux standard calling convention, the first 128-bytes after the stack pointer, **rsp**, are reserved. For example, extending the previous example, the call frame would be as follows:

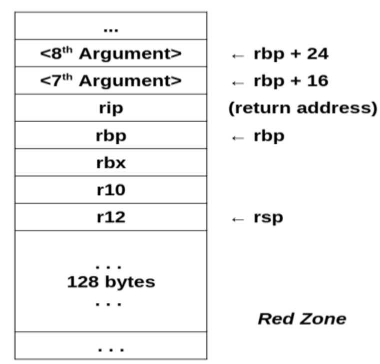

Illustration 25: Stack Frame Layout with Red Zone

Example, Statistical Function 1 (leaf)

- Example will demonstrate calling a simple void function to find the sum and average of an array of numbers
- The High-Level Language (HLL) call for C/C++ is as follows:
  stats1(arr, len, sum, ave);
- The array, ***arr***, is call-by-reference and the length, ***len***, is call-by-value. The arguments for ***sum*** and ***ave*** are both call-by-reference (since there are no values as yet)

Caller

- There are 4 arguments, and all arguments are passed in registers in accordance with the standard calling convention. The assembly language code in the calling routine for the call to the stats function would be as follows:
- ; stats1(arr, len, sum, ave);
  - mov rcx, ave ; 4th arg, addr of ave
  - mov rdx, sum ; 3rd arg, addr of sum
  - mov esi, dword [len] ; 2nd arg, value of len
  - mov rdi, arr ; 1st arg, addr of arr
  - call stats1

Callee

- The function being called, the callee, must perform the prologue and epilogue operations (as specified by the standard calling convention) before and after the code to perform the function goal
- For this example, the function must perform the summation of values in the array, compute the integer average, return the sum and average values
-

```
;  Arguments:
;   arr, address – rdi
;   len, dword value – esi
;   sum, address – rdx
;   ave, address - rcx

global stats1
stats1:
    push   r12                      ; prologue

    mov    r12, 0                   ; counter/index
    mov    rax, 0                   ; running sum
sumLoop:
    add    eax, dword [rdi+r12*4]   ; sum += arr[i]
    inc    r12
    cmp    r12, rsi
    jl     sumLoop

    mov dword [rdx], eax            ; return sum

    cdq
    idiv   esi                      ; compute average
    mov    dword [rcx], eax         ; return ave
```

Example, Statistical Function2 (non-leaf)

- This extended example will demonstrate calling a simple void function to find the minimum, median, maximum, sum and average of an array of numbers.
- The HighLevel Language (HLL) call for C/C++ is as follows:
  stats2(arr, len, min, med1, med2, max, sum, ave);
- For this example, it is assumed that the array is sorted in ascending order
- the median will be the middle value. For an even length list, there are two middle values, ***med1*** and ***med2***, both of which are returned

Caller

- There are 8 arguments and only the first six can be passed in registers. The last two arguments are passed on the stack
- The assembly language code in the calling routine for the call to the stats function would be as follows:

```
; stats2(arr, len, min, med1, med2, max, sum, ave);
push    ave                         ; 8th arg, add of ave
push    sum                         ; 7th arg, add of sum
mov     r9, max                     ; 6th arg, add of max
mov     r8, med2                    ; 5th arg, add of med2
mov     rcx, med1                   ; 4th arg, add of med1
mov     rdx, min                    ; 3rd arg, addr of min
mov     esi, dword [len]            ; 2nd arg, value of len
mov     rdi, arr                    ; 1st arg, addr of arr
call    stats2
add     rsp, 16                     ; clear passed arguments
```

Callee

- The function must perform the summation of values in the array, find the minimum, medians, and maximum, compute the average, return all the values.
- When call-by-reference arguments are passed on the stack, two steps are required to return the value.
  - Get the address from the stack.
  - Use that address to return the value.

```
;   arr, address – rdi
;   len, dword value – esi
;   min, address – rdx
;   med1, address - rcx
;   med2, address - r8
;   max, address - r9
;   sum, address – stack (rbp+16)
;   ave, address – stack (rbp+24)

global stats2
stats2:
    push    rbp                             ; prologue
    mov     rbp, rsp
    push    r12

; -----
;  Get min and max.

    mov     eax, dword [rdi]                ; get min
    mov     dword [rdx], eax                ; return min

    mov     r12, rsi                        ; get len
    dec     r12                             ; set len-1
    mov     eax, dword [rdi+r12*4]          ; get max
    mov     dword [r9], eax                 ; return max
```

```
; -----
;  Get medians

    mov    rax, rsi
    mov    rdx, 0
    mov    r12, 2
    div    r12                          ; rax = length/2

    cmp    rdx, 0                       ; even/odd length?
    je     evenLength

    mov    r12d, dword [rdi+rax*4]      ; get arr[len/2]
    mov    dword [rcx], r12d            ; return med1
    mov    dword [r8], r12d             ; return med2
    jmp    medDone
```

```
evenLength:
    mov     r12d, dword [rdi+rax*4]       ; get arr[len/2]
    mov     dword [r8], r12d              ; return med2
    dec     rax
    mov     r12d, dword [rdi+rax*4]       ; get arr[len/2-1]
    mov     dword [rcx], r12d             ; return med1
medDone:

; -----
;   Find sum

    mov     r12, 0                        ; counter/index
    mov     rax, 0                        ; running sum

sumLoop:
    add     eax, dword [rdi+r12*4]        ; sum += arr[i]
    inc     r12
    cmp     r12, rsi
    jl      sumLoop

    mov     r12, qword [rbp+16]           ; get sum addr
    mov     dword [r12], eax              ; return sum

; -----
;   Calculate average.

    cdq
    idiv    rsi                           ; average = sum/le
    mov     r12, qword [rbp+24]           ; get ave addr
    mov     dword [r12], eax              ; return ave
```

- The call frame for this function would be as follows:
- In this example, the preserved registers **rpb** and then **r12** is pushed. When popped, they must be popped in the exact reverse order **r12** and then **rpb** in order to correctly restore their original values.

. . .	
<8th Argument>	← rbp + 24
<7th Argument>	← rbp + 16
rip	(return address)
rbp	← rbp
r12	← rsp
. . .	