CPSC-240 Computer Organization and Assembly Language

Chapter 12

Functions

Instructor: Yitsen Ku, Ph.D.
Department of Computer Science,
California State University, Fullerton, USA

CALIFORNIA STATE UNIVERSITY FULLERTON™

Outline

- Updated Linking Instructions
- Debugger Commands
- Stack Dynamic Local Variables
- Function Declaration
- Standard Calling Convention
- Linkage
- Argument Transmission
- Calling Convention

Updated Linking Instructions

Updated Linking Instructions

- When writing and debugging functions, it is easier for the C compiler (either GCC or G++) to link the program as the C compiler is aware of the appropriate locations for the various C/C++ libraries.

```
yasm -g dwarf2 -f elf64 example.asm -l example.lst
gcc -g -o example example.o
```

- Note, Ubuntu 18 will require the no-pie option on the gcc command as shown:

```
gcc -g -no-pie -o example example.o
```

- This will use the GCC compiler to call the linker, reading the example.o object file and creating the example executable file. The "-g" option includes the debugging information in the executable file in the usual manner.

Debugger Commands

Debugger Command, *next*

- With respect to a function call, the debugger next command will execute the entire function and go to the next line.
- When debugging functions, this is useful to quickly execute the entire function and then just verify the results. It will not display any of the function code.

Debugger Command, *step*

- With respect to a function call, the debugger step command will step into the function and go to the first line of the function code.
- It will display the function code. When debugging functions, this is useful to debug the function code.

Stack Dynamic Local Variables

Stack Dynamic Local Variables

- In a high-level language, non-static local variables declared in a function are stack dynamic local variables by default.
- Some C++ texts refer to such variables as automatics. This means that the local variables are created by allocating space on the stack and assigning these stack locations to the variables.
- When the function completes, the space is recovered and reused for other purposes.

Stack Dynamic Local Variables

- This requires a small amount of additional run-time overhead, but makes a more efficient overall use of memory.
- If a function with a large number of local variables is never called, the memory for the local variables is never allocated.
- This helps reduce the overall memory footprint of the program which generally helps the overall performance of the program.

Stack Dynamic Local Variables

- In contrast, statically declared variables are assigned memory locations for the entire execution of the program.
- This uses memory even if the associated function is not being executed. However, no additional run-time overhead is required to allocate the space since the space allocation has already been performed (when the program was initially loaded into memory).

Function Declaration

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. There is no specific order required for how functions are defined. However, functions cannot be nested.

Standard Calling Convention

Standard Calling Convention

- To write assembly programs, a standard process for passing parameters, returning values, and allocating registers between functions is needed.
- If each function did these operations differently, things would quickly get very confusing and require programmers to attempt to remember for each function how to handle parameters and which registers were used.

Standard Calling Convention

- To address this, a standard process is defined and used which is typically referred to as a standard calling convention.
- There are actually a number of different standard calling conventions.
- The 64-bit C calling convention, called System V AMD64 ABI, is described in the remainder of this document.

Standard Calling Convention

- This calling convention is also used for C/C++ programs by default. This means that interfacing assembly language code and C/C++ code is easily accomplished since the same calling convention is used.
- It must be noted that the standard calling convention presented here applies to Linux based operating systems. The standard calling convention for Microsoft Windows is slightly different and not presented in this text.

Linkage

Linkage

- The linkage is about getting to and returning from a function call correctly. There are two instructions that handle the linkage, call and ret instructions.
- The call transfers control to the named function, and ret returns control back to the calling routine.
  - The call works by saving the address of where to return to when the function completes (referred to as the return address). This is accomplished by placing contents of the rip register on the stack. Recall that the rip register points to the next instruction to be executed (which is the instruction immediately after the call).
  - The ret instruction is used in a procedure to return. The ret instruction pops the current top of the stack (rsp) into the rip register. Thus, the appropriate return address is restored.

Linkage

- Since the stack is used to support the linkage, it is important that within the function the stack must not be corrupted. Specifically, any items pushed must be popped.
- Pushing a value and not popping would result in that value being popped off the stack and placed in the rip register.
- This would cause the processor to attempt to execute code at that location. Most likely the invalid location will cause the process to crash.

Summary of the function calling or linkage

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

- The standard terminology for transmitting values to a function is referred to as call-by-value. The standard terminology for transmitting addresses to a function is referred to as call-by-reference. This should be a familiar topic from a high-level language.
- In general, the calling routine is referred to as the ***caller*** and the routine being called is referred to as the ***callee***.

Argument Transmission

- 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.

Parameter Passing

- 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`

Parameter Passing

- The seventh and any additional arguments are passed on the stack. The standard calling convention requires that, when passing arguments (values or addresses) on the stack, the arguments should be pushed in reverse order.
- That is "**someFunc (one, two, three, four, five, six, seven, eight, nine)**" would imply a push order of: *nine*, *eight*, and then *seven*.

Parameter Passing

- For floating-point arguments, the floating-point registers **xmm0** to **xmm7** are used in that order for the first eight float arguments.
- Additionally, 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**.

Parameter Passing

- For value returning functions, the result is placed in the **A** register based on the size of the value being returned.

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

Register Usage

- For value returning functions, the result is placed in the **A** register based on the size of the value being returned.

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

Register Usage

- The standard calling convention specifies the usage of registers when making function calls. Specifically, 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 calling routine.
- This register preservation is typically performed in the prologue and the restoration is typically performed in the epilogue.

The following table summarizes the register usage.

Register	Usage
`rax`	Return Value
`rbx`	Callee Saved
`rcx`	4^{th} Argument
`rdx`	3^{rd} Argument
`rsi`	2^{nd} Argument
`rdi`	1^{st} Argument
`rbp`	Callee Saved
`rsp`	Stack Pointer
`r8`	5^{th} Argument
`r9`	6^{th} Argument
`r10`	Temporary
`r11`	Temporary
`r12`	Callee Saved
`r13`	Callee Saved
`r14`	Callee Saved
`r15`	Callee Saved

Call 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).

Call Frame

- 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
...	

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:

...	
<8th Argument>	← rbp + 24
<7th Argument>	← rbp + 16
rip	(return address)
rbp	← rbp
rbx	
r10	
r12	← rsp
. . . 128 bytes . . .	*Red Zone*
. . .	

Example, Statistical Function 1 (leaf)

- **Caller**

```
; 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**

```
; Simple example function to find and return
; the sum and average of an array.
; HLL call:
; stats1(arr, len, sum, ave);
; -----
; Arguments:
; arr, address – rdi
; len, dword value – esi
; sum, address – rdx
; ave, address - rcx
```

Example, Statistical Function 1 (leaf)

```
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
    pop     r12                         ; epilogue
    ret
```

Call Frame of Example 1

- The choice of the **r12** register is arbitrary, however a 'saved register' was selected. The call frame for this function would be as follows:

...	
rip	(return address)
r12	← rsp
...	

Example, Statistical Function 2 (non-leaf)

- **Caller**

```
; 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
```

Example, Statistical Function 2 (non-leaf)

- **Callee**

```
; Simple example function to find and return the minimum,
; maximum, sum, medians, and average of an array.
; -----
; HLL call:
; stats2(arr, len, min, med1, med2, max, sum, ave);
; Arguments:
; 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)
```

Example, Statistical Function 2 (non-leaf)

```
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
```

Example, Statistical Function 2 (non-leaf)

```
    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
```

Example, Statistical Function 2 (non-leaf)

```
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/len
    mov     r12, qword [rbp+24]         ; get ave addr
    mov     dword [r12], eax            ; return ave
    pop     r12                         ; epilogue
    pop     rbp
    ret
```

Call Frame of Example 2

- The choice of the registers is arbitrary with the bounds of the calling convention. The call frame for this function would be as follows:

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

Lab Activity

Lab Activity

Given the following variable declarations and code fragment:

```
        lst     dq      1, 3, 5, 7, 9
        mov     rsi, 0
        mov     rcx, 5
lp1:    push    qword [lst+rsi*8]
        inc     rsi
        loop    lp1
        mov     rsi, 0
        mov     rcx, 5
lp2:    pop     qword [lst+rsi*8]
        inc     rsi
        loop    lp2
        mov     rbx, qword [lst]
```

Explain what would be the ***result*** of the code (after execution)?

CALIFORNIA STATE UNIVERSITY FULLERTON

End of Chapter 12