User Id : weizhe.goh@digipen.edu | Started : 2019.11.26 11:27:16 | Score :

DigiPen CS Vadim Surov CS100 Presentation #19

# Assembler - Functions

This presentation guides you through working with functions.using assembler.

## 1. Stack

- The following code shows how to work with the stack:

Run

```
.macro PRINT fmt, v
    mov   \fmt, %edi
    mov   \v, %rsi
    xor   %eax, %eax # Clear AL
    call  printf
.endm

    .data
fmt: .asciz "%d"
    .text
    .global main
main:
    push  %rbx # For alignment

    mov  $10, %rax
    mov  $20, %rbx
    mov  $30, %rcx
    push %rax
    push %rbx
    push %rcx
    pop  %rax
    pop  %rbx
    pop  %rcx

    PRINT $fmt, %rax

    xor   %eax, %eax # return 0;
    pop   %rbx
    ret
```

```
30
```

## 1. Comments

- The stack is a special reserved area in memory for placing data.
- The stack is reserved at the end of the memory area, and as data is placed on the stack, it grows downward.
- The %rsp register contains the memory address of the start of the stack.
- Placing new data items in the stack is called pushing. The instruction used to perform this task is the push instruction.
- When you have some data on the stack, you can retrieve the data from the stack using pop instruction.
- The %rsp register decrease or increase with each data element added or removed to the stack, pointing to the new start of the stack.

## 1. Problems

- No problems so far

## 2. Functions

- How to create and use functions in assembly language programs?
- The following code calls foo function to set 2 to the rax register:

Run

```
.macro PRINT fmt, v
    mov   \fmt, %edi
    mov   \v, %rsi
    xor   %eax, %eax # Clear AL
    call  printf
.endm

    .data
fmt: .asciz "%d"
    .text
    .global main
main:
    push  %rbx # For alignment

    mov   $1, %rax
    call  foo
    PRINT $fmt, %rax
    jmp   end
foo:
    mov   $2, %rax
    ret

end:
    xor   %eax, %eax # return 0;
    pop   %rbx
    ret
```

```
2
```

## 2. Comments

- The call instruction is used to pass control from the main program to the function.
    - When the call instruction is performed, the next instruction executed is the first instruction in the function
    - Continuing to step through the program, the next instruction in the function is performed, and so on until the ret instruction, which returns to the main program
- The call instruction places the return address from the calling program onto the top of the stack, so the function knows where to return.

## 2. Problems

- If you are calling a function that modifies registers the main program uses, there is no guarantee that the registers will be in the same state when the function is finished as they were before the function was called.
- So, it is crucial that you save the current state of the registers before calling the function, and then restore them after the function returns.

### 3. Stack As Storage

- The following code store and restore the rax register value in the stack before and after calling the foo function:

Run

```
.macro PRINT fmt, v
    mov   \fmt, %edi
    mov   \v, %rsi
    xor   %eax, %eax # Clear AL
    call  printf
.endm

    .data
fmt: .asciz "%d"
    .text
    .global main
main:
    push  %rbx # For alignment

    mov   $100, %rax
    push  %rax
    mov   $1, %rax
    call  foo
    pop   %rax
    PRINT $fmt, %rax
    jmp   end
foo:
    mov   $2, %rax
    ret

end:
    xor   %eax, %eax # return 0;
    pop   %rbx
    ret
```

```
100
```

### 3. Comments

- The foo function uses rax register to pass and return a parameter.
- rax, rbx, and so on, are 64 bit registers.
- push and pop work with such registers.

### 3. Problems

- How about more parameters?

### 4. Stack As Storage

- The following code stacks 3 registers to use them for parameters and the returning result of a function:

Run

```
.macro PRINT fmt, v
    mov   \fmt, %edi
    mov   \v, %rsi
    xor   %eax, %eax # Clear AL
    call  printf
.endm

    .data
fmt: .asciz "%d"
    .text
    .global main
main:
    push  %rbx # For alignment

    push %rax
    push %rbx
    push %rcx

    mov  $1, %rax
    mov  $2, %rbx
    call add

    PRINT $fmt, %rcx

    pop  %rcx
    pop  %rbx
    pop  %rax

    jmp  end

add:
    add %rax, %rbx
    mov %rbx, %rcx
    ret

end:
    xor   %eax, %eax # return 0;
    pop   %rbx
    ret
```

```
3
```

### 4. Comments

- Note that pop instructions go in the reverse order to push instructions.
- 32-bit assembler has pusha and popa instructions to save and restore all reqisters.
- Those instructions are not supported in 64-bit mode.

### 4. Problems

- Trying to keep track of which function uses which registers and global variables, or which registers and global variables are used to pass which parameters, can be a nightmare.
- The C solution for passing input values to functions is to use the stack.
- The stack is accessible from the main program as well as from any functions used within the program. This creates a clean way to pass data between the main program and the functions in a common location, without having to worry about using registers or defining global variables.

### 5. Parameters On Stack

- The following code uses stack to pass parameters:

### 5. Comments

- All of the input parameters for the function are located “underneath” the return address on the stack.
- To retrieve the input parameters from the stack, indirect addressing is used.
  - 8(%rsp) - the first parameter,
  - 16(%rsp) - the second, and so on.

Run

```
.macro PRINT fmt, v
    mov   \fmt, %edi
    mov   \v, %rsi
    xor   %eax, %eax # Clear AL
    call  printf
.endm

    .data
fmt: .asciz "%d"
    .text
    .global main
main:
    push  %rbx # For alignment

    push %rax
    push %rbx
    push %rcx

    push $1
    push $2
    call add
    add $16, %rsp # pop, pop

    PRINT   $fmt, %rax

    pop %rcx
    pop %rbx
    pop %rax

    jmp  end

add:
    mov 8(%rsp), %rax
    add 16(%rsp), %rax
    ret

end:
    xor   %eax, %eax # return 0;
    pop   %rbx
    ret
```

```
3
```

**6. Function prologue and epilogue**

## 5. Problems

- There is a problem with this technique, however. While in the function, it is possible that part of the function process will include pushing data onto the stack. If this happens, it would change the location of the %rsp stack pointer and throw off the indirect addressing values for accessing the parameters in the stack.
- To avoid this problem, it is common practice to copy the %rsp register value to the %rbp register when entering the function.

- The following code program demonstrates how to ensure that there is a register (let it be %rbp) that always contains the correct pointer to the top of the stack when the function is called:

Run

```
.macro PRINT fmt, v
    mov   \fmt, %edi
    mov   \v, %rsi
    xor   %eax, %eax # Clear AL
    call  printf
.endm

    .data
fmt: .asciz "%d"
    .text
    .global main
main:
    push  %rbx # For alignment

    push %rax
    push %rbx
    push %rcx

    push $1
    push $2
    call add
    add $16, %rsp # pop, pop

    PRINT   $fmt, %rax

    pop %rcx
    pop %rbx
    pop %rax

    jmp  end

add:
    push %rbp
    mov  %rsp, %rbp

    mov 16(%rbp), %rax
    add 24(%rbp), %rax

    mov %rbp, %rsp
    pop %rbp
    ret

end:
    xor   %eax, %eax # return 0;
    pop   %rbx
    ret
```

3

### 6. Comments

- The first two instructions at the top of the function code save the original value of %rbp to the top of the stack, and then copy the current %rsp stack pointer (now pointing to the original value of %rbp in the stack) to the %rbp register.
- After the function processing completes, the last two instructions in the function retrieve the original value in the %rsp register that was stored in the %rbp register, and restore the original %rbp register value.
- %rbp is calles the stack pointer.

### 7. System V

- Both Mac OS X and Linux follow the System V ABI for their x86-64 calling conventions.
- There are three x86-64 instructions used to implement procedure calls and returns.
  - The call instruction pushes the address of the next instruction (i.e., the return address) onto the stack and then transfers control to the address specified by its operand.
  - The leave instruction sets the stack pointer (%rsp) to the frame pointer (%rbp) and then sets the frame pointer to the saved frame pointer, which is popped from the stack.
    - Value (%rsp+8) must always be 16-byte aligned when control is transfered to a function entry point.
  - The ret instruction pops the return address off the stack and jumps to it.

### 8. System V

- Integer arguments (up to the first six) are passed in registers, namely: %rdi, %rsi, %rdx, %rcx, %r8, %r9. Additional arguments, if needed, are passed in stack slots immmediately above the return address.
- An integer-valued function returns its result in %rax.
- Registers %rbx,%rbp,%r12,%r13,%r14,%r15 are callee-save; that is, the callee is responsible for making sure that the values in these registers are the same at exit as they were on entry.
- Floating arguments (up to 8) are passed in special registers %xmm0, %xmm1, ..., %xmm7.
  - Byte register %al must be set before the call to indicate how many of the %xmm registers are used.
  - A floating point return value is returned in %xmm0.
  - All the %xmm registers are caller-save.

### References

Conventions – - Calling conventions

By signing this document you fully agree that all information provided therein is complete and true in all respects.

Responder sign: