User Id : weizhe.goh@digipen.edu | Started : 2019.11.12 12:24:49 | Score :

DigiPen CS Vadim Surov CS100 Presentation #16

Assembler - Flow Control

This presentation ia about flow control in assembler code

1 Intro

- Assembler has the ability to change the order in which instructions are executed.
- The Instruction Pointer (IP) register, %rip for x86-64, contains the address of the next instruction to be executed. So to change the flow of control, the programmer must be able to modify the value of IP.
- IP cannot be set directly using, for example, mov instruction:

```
mov label, %rip # Does not work
```

- Special control instructions must be used instead.

2 Jump instructions

- Different jump instructions allow the programmer to set the value of the IP register indirectly.
- The location (usually a label) passed as the argument of such instructions.
- The first instruction executed after the jump is the instruction immediately following the label.
- The jmp instruction is also called **unconditional jump**.

3 jmp

- The following example shows control jumping over 3 instructions making them **unreachable code**, or code that will never be reached regardless of logic flow.

Run

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

    jmp    next

    mov    $str, %rdi
    mov    $10, %rsi
    xor    %eax, %eax # Clear AL
    call   printf

next:
    xor    %eax, %eax # return 0;
    pop    %rbx # For alignment
    ret
```

- Think about:
  - How to make infinite loop with jmp?
  - How to "restart" the program?

4 cmp

- All the rest jump instructions are conditional jumps, meaning that program flow is diverted only if a condition is true.
- These instructions are often used after a comparison instruction cmp, but this order is not required.
- cmp instruction performs a comparison operation between first (subtrahend) and second (minuend) operands.
- cmp subtrahend, minuend
- The comparison is performed by a (signed) subtraction of subtrahend from minuend, the results as flags are saved in flag register, the result as a value is discarded (this is the only difference with **sub** instruction).
- Examples:

```
cmp $0, %rax
```

```
cmp %rax, %rbx
```

```
cmp a, %rax
```

5 je

- je (Jump if Equal) instruction loads IP with the specified address, if zero flag is set, ZF=1.
    - For example, ZF=1 when operands of previous cmp or **sub** instructions are equal.
- So next example, output the same result if replace cmp with subl. Why cmp is better there?

5 je 2

Run

```
     .data
str1: .asciz "true"
str2: .asciz "false"
     .text
     .global main
main:
    push    %rbx # For alignment

    movl    $5, %ecx
    movl    $5, %edx
    cmp     %ecx, %edx
    je      equal

    mov     $str2, %rdi
    xor     %eax, %eax # Clear AL
    call    printf

equal:
    mov     $str1, %rdi
    xor     %eax, %eax # Clear AL
    call    printf

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

```
true
```

- Try jne instruction (Jump if Not Equal)

6 jg

- jg (Jump if Greater) instruction loads IP with the specified address, if sign and zero flags are both reset: SF=0 ,ZF=0) .
    - For example, when the minuend of the previous cmp instruction is greater than the subtrahend (performs signed comparison).

Run

```
    .data
a:    .long  10
b:    .long  20
str1: .asciz "10>20"
str2: .asciz "10<20"
    .text
    .global main
main:
    push    %rbx # For alignment

    movl b, %eax
    cmp a, %eax

    jg next

    mov    $str1, %rdi
    xor    %eax, %eax # Clear AL
    call   printf

next:
    mov    $str2, %rdi
    xor    %eax, %eax # Clear AL
    call   printf

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

```
10<20
```

- Try jng.

7 List 1

- jmp - unconditional jump
- jo - Jump if Overflow
- jno - Jump if Not Overflow
- js - Jump if Signed
- jns - Jump if Not Signed

8 List 2

For signed comparison:

- je - Jump if Equal
- jne - Jump if Not Equal
- jg - Jump if Greater
- jge - Jump if Greater or Equal
- jl - Jump if Lesser
- jle - Jump if Lesser or Equal
- jz - Jump if Zero
- jnz - Jump if Not Zero

9 List 3

For unsigned comparisons:

- ja - Jump if Above
- jea - Jump if Above or Equal
- jb - Jump if Below
- jbe - Jump if Below or Equal

10 Counter

- Next example uses dec and jnz to implement a loop with counter in %ecx register.

Run

```
    .data
str: .asciz "loop\n"
    .text
    .global main
main:
    push   %rbx # For alignment

    movq   $3, %rcx
next:

    push   %rcx

    mov    $str, %rdi
    xor    %eax, %eax # Clear AL
    call   printf

    pop    %rcx

    dec    %rcx
    jnz    next

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

```
loop
loop
loop
```

- Why push and pop of %rcx is used in this code?

11 Loop

- loop instruction decrements %rcx and jumps to the address specified as operand unless decrementing %rcx caused its value to become zero.

Run

```
    .data
str: .ascii "%d "
    .text
    .global main
main:
    push   %rbx # For alignment

    mov    $5, %rcx

repeat:

    push   %rcx

    # Output
    mov    $str, %rdi
    mov    %rcx, %rsi
    xor    %eax, %eax # Clear AL
    call   printf

    pop    %rcx
    loop   repeat

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

```
5 4 3 2 1
```

- How to output in ascending order? What about even numbers? Negative numbers?

★ 12 Conditional loops

- loope (Loop if Equal) and loopz (Loop if Zero) instructions permits a loop to continue while ZF=1 and %rcx>0.
- loopne (Loop if Not Equal) and loopnz (Loop if Not Zero) instructions permits a loop to continue while ZF=0 and %rcx>0.

★ 13 loopne

- What is the output?

Run

```
    .data
str: .ascii "%d "
    .text
    .global main
main:
    push  %rbx # For alignment

    mov   $10, %rcx

repeat:
    push   %rcx

    # Output
    mov   $str, %rdi
    mov   %rcx, %rsi
    xor   %eax, %eax # Clear AL
    call  printf

    pop   %rcx

    cmp    $5, %rcx
    loopne repeat

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

```
10 9 8 7 6 5
```

★ 100 References

[Manual](#) – The GNU Assembler manual

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

Responder sign: