CSD1100

Assembler - Flow Control

Vadim Surov

Introduction

- 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 rip, label          ; Does not work
```

- Special **control instructions must be used** instead.

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 followed the label.
- The **jmp** instruction is simplest one, and it is what is called the **unconditional jump** instruction:

```
jmp end   ; Same as goto end in c
```

jmp

- The following example shows control jumping over 4 output instructions making them **unreachable code.**
- **Unreachable code** is a code that will never be reached regardless of logic flow.

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

jmp

fc_jmp.asm

```
1   ; Compilation and debugging:
2   ; $ nasm -f elf64 -g -F dwarf fc_jmp.asm
3   ; $ ld -dynamic-linker /lib64/ld-linux-x86-64.so.2
4   ; $ gdb fc_jmp
5   ; (gdb) b _start
6   ; (gdb) run
7   ; (gdb) s
8   ; (gdb) q
9
10  section .data
11
12  format db '%lld',10,0
13
14  section .text
15      global _start
16      extern printf
17
```

jmp

```
17
18  _start:
19      jmp     next        ; unconditional jump
20
21      ; output
22      mov     rdi, format
23      mov     rsi, 10
24      xor     rax, rax
25      call    printf
26
27  next:
28      mov rax, 60         ; syscall number for exit
29      mov rdi, 0          ; int status 0
30      syscall
```

jmp

```
$ gdb -q fc_jmp
Reading symbols from fc_jmp...
(gdb) b _start
Breakpoint 1 at 0x401020: file fc_jmp.asm, line 19.
(gdb) run
Starting program: /mnt/c/Users/vadim/OneDrive/Desktop/Pr

Breakpoint 1, _start () at fc_jmp.asm:19
19          jmp    next     ; unconditional jump
(gdb) s
next () at fc_jmp.asm:28
28          mov rax, 60     ; syscall number for exit
(gdb) s
29          mov rdi, 0      ; int status 0
(gdb)
```

cmp

- All the rest jump instructions are **conditional jumps**, meaning that program flow is diverted only if some condition is true.
- These instructions are often used after a comparison instruction **cmp**, but this order is not required.

```
...
cmp rax, rbx
...
je next
...
```

cmp

- **cmp** instruction performs a comparison operation between first (subtrahend) and second (minuend) operands.

cmp minuend, subtrahend

- The comparison is performed by a (signed) subtraction of subtrahend from minuend, the **results as flags** are saved in flag register.

```
(gdb) i r eflags
eflags         0x246                [ PF ZF IF ]
(gdb)
```

- Using **cmp** the result as a value is discarded (this is the only difference with **sub** instruction).

je

- **je** (Jump if Equal) instruction loads IP with the specified address, if zero flag is set, ZF=1.
- ZF=1 when operands of previous cmp or sub instructions are equal.

- Next example outputs the same result if replace cmp with sub. Why cmp is better there?
- Try jne instruction (Jump if Not Equal)

je

fc_je.asm

```
1   ; Compilation and debugging:
2   ; $ nasm -f elf64 -g -F dwarf fc_je.asm
3   ; $ ld -dynamic-linker /lib64/ld-linux-x86-64.so.2 -lc fc_je.o -o fc_je
4   ; $ gdb -q fc_je
5   ; (gdb) b _start
6   ; ...
7
8   section .data
9
10  true  db "true",10,0
11  false db "false",10,0
12
13  section .text
14      global _start
15      extern puts
```

je

```
17  _start:
18      mov     rcx, 5
19      mov     rdx, 5
20      cmp     rcx, rdx
21      je      equal       ; conditional jump
22
23      ; Output false
24      mov     rdi, false
25      xor     rax, rax
26      call    puts
27      jmp     end
28
29  equal:
30      ; Output true
31      mov     rdi, true
32      xor     rax, rax
33      call    puts
34
35  end:
36      mov rax, 60         ; syscall number for exit
37      mov rdi, 0          ; int status 0
38      syscall
39
```

je

```
(gdb) run
Starting program: /mnt/c/Users/vadim/OneDrive/Desktop/Pr

Breakpoint 1, _start () at fc_je.asm:23
23          mov     rcx, 5
(gdb) s
24          mov     rdx, 5
(gdb) s
25          cmp     rcx, rdx
(gdb) s
26          je      equal     ; conditional jump
(gdb) s
equal () at fc_je.asm:36
36          mov     rdi, true
(gdb) s
37          xor     rax, rax
(gdb) s
38          call    puts
(gdb) b 41
Breakpoint 2 at 0x401055: file fc_je.asm, line 41.
(gdb) n
true


Breakpoint 2, end () at fc_je.asm:41
41          mov rax, 60     ; syscall number for exit
(gdb) s
42          mov rdi, 0      ; int status 0
(gdb) s
43          syscall
```

jg

- jg (Jump if Greater) instruction loads IP with the specified address, if sign and zero flags are both reset:

  SF=0, ZF=0

- SF=0, ZF=0 when the minuend of the previous cmp instruction is greater than the subtrahend.
- Try jng.

jg

fc_jg.asm

```
1   ; Compilation and debugging:
2   ; $ nasm -f elf64 -g -F dwarf fc_jg.asm
3   ; $ ld -dynamic-linker /lib64/ld-linux-x86-64.
4   ; $ gdb -q fc_jg
5   ; (gdb) b _start
6   ; ...
7
8   section .data
9   a    dq 10
10  b    dq 20
11  str1 db "10<20",10,0
12  str2 db "10>20",10,0
13
14  section .text
15      global _start
16      extern printf
```

jg

```
18  _start:
19      mov     rax, [a]
20      cmp     rax, [b]
21      jg      next
22
23      mov     rdi, str1
24      xor     rax, rax
25      call    printf
26      jmp     end
27
28  next:
29      mov     rdi, str2
30      xor     rax, rax
31      call    printf
32
33  end:
34      mov rax, 60         ; syscall number for exit
35      mov rdi, 0          ; int status 0
36      syscall
```

jg

```
Reading symbols from fc_jg...
(gdb) b _start
Breakpoint 1 at 0x401020: file fc_jg.asm, line 19.
(gdb) run
Starting program: /mnt/c/Users/vadim/OneDrive/Desktop/Pr

Breakpoint 1, _start () at fc_jg.asm:19
19          mov     rax, [a]
(gdb) s
20          cmp     rax, [b]
(gdb)
21          jg      next
(gdb)
23          mov     rdi, str1
(gdb)
24          xor     rax, rax
(gdb)
25          call    printf
(gdb) b 26
Breakpoint 2 at 0x401044: file fc_jg.asm, line 26.
(gdb) c
Continuing.
10<20

Breakpoint 2, _start () at fc_jg.asm:26
26          jmp     end
```

List of conditional jumps

- jo - Jump if Overflow (the result of 2 positive numbers results in a negative number or if the sum of 2 negative numbers result in a positive number)
- jno - Jump if Not Overflow
- js - Jump if Signed (the result of an operation is negative)
- jns - Jump if Not Signed

List of conditional jumps (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

List of conditional jumps (for unsign. comparison)

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

Counter

- **dec** instruction is used to decrement the operand by 1.
- Next example uses both **dec** and **jnz** to implement a loop with counter in **rcx** register.
- Why push and pop of **rcx** is used in this code?

Counter

ASM fc_counter.asm

```
1   ; Compilation and debugging:
2   ; $ nasm -f elf64 -g -F dwarf fc_counter.asm
3   ; $ ld -dynamic-linker /lib64/ld-linux-x86-64.so.2 -
4   ; ./fc_counter
5
6   section .data
7   str1  db "loop",0
8
9   section .text
10      global _start
11      extern puts
```

Counter

```
13  _start:
14      mov     rcx, 3
15
16  next:
17      push    rcx         ; Save rcx (puts uses it too)
18
19      mov     rdi, str1
20      call    puts
21
22      pop     rcx         ; Restore rcx
23
24      dec     rcx
25      jnz     next
26
27      mov rax, 60         ; syscall number for exit
28      mov rdi, 0          ; int status 0
29      syscall
```

Loop

- **loop** instruction decrements `rcx` and jumps to the address specified as operand unless decrementing `rcx` caused its value to become zero.
- Next example shows how to output in descending order from 5 to 1.
- How to output in ascending order? What about even numbers? Negative numbers?

loop

fc_loop.asm

```asm
;·Compilation·and·debugging:
;·$·nasm·-f·elf64·fc_loop.asm
;·$·ld·-dynamic-linker·/lib64/ld-linux-x86-64.so.2·-lc
;·$·./fc_loop

section·.data
str1··db·"%lli·",10,0

section·.text
····global·_start
····extern·printf

```

loop

```
13  _start:
14      mov     rcx, 5      ; Set counter to 5
15
16  repeat:
17      push    rcx         ; Save
18
19      ; Output
20      mov     rdi, str1
21      mov     rsi, rcx
22      xor     rax, rax
23      call    printf
24
25      pop     rcx         ; Restore
26      loop    repeat
27
28      mov rax, 60         ; syscall number for exit
29      mov rdi, 0          ; int status 0
30      syscall
```

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.

- What is the output of the following code?

loopnz

ASM fc_loopnz.asm

```
1    ; Compilation and debugging:
2    ; $ nasm -f elf64 fc_loopnz.asm
3    ; $ ld -dynamic-linker /lib64/ld-linux-x
4    ; $ ./fc_loopnz
5
6    section .data
7    fmt  db "%lli",10,0
8
9    section .text
10       global _start
11       extern printf
```

loopnz

```
13  _start:
14      mov     rcx, 10     ; Set counter to 10
15
16  repeat:
17      push    rcx         ; Save
18
19      ; Output
20      mov     rdi, fmt
21      mov     rsi, rcx
22      xor     rax, rax
23      call    printf
24
25      pop     rcx         ; Restore
26
27      cmp     rcx, 5
28      loopne repeat
29
30      mov rax, 60         ; syscall number for exit
31      mov rdi, 0          ; int status 0
32      syscall
```

References

1. NASM documentation
   https://www.nasm.us/doc/
2. An official list of the instruction codes can be found in appendix B:
   https://www.nasm.us/doc/nasmdocb.html