Computer Systems

CS107

Cynthia Lee

Today's Topics

LECTURE:

- More assembly code!
  - More control flow
  - Function call and return
  - Some misc. instructions you might see in your assign5 binary bomb

Go here!

https://web.stanford.edu/class/cs107/guide_x86-64.html

Conditional jumps

Typical 2-step control flow

1. Compare two values to **write** the condition codes (implicit destination register)
   - cmp, test
2. Conditionally jump based on **reading** the condition codes (implicit source register)
   - je, jne, jl, jg

%eflags

- There is also a 1-step unconditional jump
- Doesn't look at condition code, just goes no matter what
  - jmp [target]

STEP 1 of control flow: cmp, test

Op	Source1	Source2	Dest Comments
`cmp`	`op2`	`op1`	op1 – op2, sets condition codes
`test`	`op2`	`op1`	op1 & op2, sets condition codes

- op1 and op2 can be any of the complex addressing modes we've seen
- Implicit destination %eflags contains condition codes
  - Sequence of Boolean values packed into one register
  - **t** is the result of the cmp or test operation above
    - ZF = zero flag (t = 0)
    - SF = sign flag (t < 0)
    - CF = carry flag (there was a carry out of MSB*, *i.e.* unsigned overflow)
    - OF = overflow flag (MSB* changed from 0 to 1, *i.e.* signed overflow)
    - …

* MSB = "Most Significant Bit"

Code example: `%eflags` and cmp

```
0000000004004f2 <if_then>:
  4004f2:       cmp    $0x6,%edi
  4004f5:       jne    4004fa <if_then+0x8>
  4004f7:       add    $0x1,%edi
  4004fa:       lea    (%rdi,%rdi,1),%eax
  4004fd:       retq
```

Which of these flags are set (i.e., the corresponding bit of `%eflags` is 1) after the `cmp` if we pass **107** to the `if_then` function?

A. ZF = zero flag (t = 0)
B. SF = sign flag (t < 0)
C. CF = carry flag (there was a carry out of MSB*, *i.e.* unsigned overflow)
D. OF = overflow flag (MSB* changed from 0 to 1, *i.e.* signed overflow)

STEP 2 of control flow: jump

Op	Target	Remarks
`jmp`	`target`	`Unconditional jump`
`je`	`target`	`Jump if ZF is 1, in other words op1-op2=0 in previous cmp, in other words op1=op2`

- Target is a memory address: the address of the instruction where we should jump
- Implicit source %eflags contains condition codes
  - Sequence of Boolean values packed into one register
    - ZF = zero flag
    - SF = sign flag
    - CF = carry flag
    - OF = overflow flag
    - …

Control operations

```
cmpl op2, op1     # result = op1 - op2, discards result, sets condition
test op2, op1     # result = op1 & op2, discards result, sets condition

jmp target        # unconditional jump
je  target        # jump equal, synonym jz jump zero (ZF=1)
jne target        # jump not equal, synonym jnz (ZF=0)
jl  target        # jump less than, synonym jnge (SF!=OF)
jle target        # jump less or equal, synonym jng (ZF=1 or SF!=OF)
jg  target        # jump greater than, synonym jnle (ZF=0 and SF=OF)
jge target        # jump greater or equal, synonym jnl (SF=OF)
ja  target        # jump above, synonym jnbe (CF=0 and ZF=0)
jb  target        # jump below, synonym jnae (CF=1)
js  target        # jump signed (SF=1)
jns target        # jump not signed (SF=0)
```

(detail note: in hex, target is specified as an offset from current address)

Control operations

```
cmpl op2, op1     # result = op1 - op2, discards result, sets condition
test op2, op1     # result = op1 & op2, discards result, sets condition

jmp target        # unconditional jump
je  target        # jump equal, synonym jz jump zero (ZF=1)
jne target        # jump not equal, synonym jnz (ZF=0)
jl  target        # jump less than, synonym jnge (SF!=OF)
jle target        # jump less or equal, synonym jng (ZF=1 or SF!=OF)
jg  target        # jump greater than, synonym jnle (ZF=0 and SF=OF)
jge target        # jump greater or equal, synonym jnl (SF=OF)
ja  target        # jump above, synonym jnbe (CF=0 and ZF=0)
jb  target        # jump below, synonym jnae (CF=1)
js  target        # jump signed (SF=1)
jns target        # jump not signed (SF=0)
```

Example of what this means: For jne following cmp, we will jump to `target` if ZF=0 (and will continue to next instruction if ZF=1)

Code example: `%eflags` and cmp together with jne

```
0000000004004f2 <if_then>:
  4004f2:       cmp    $0x6,%edi
  4004f5:       jne    4004fa
  4004f7:       add    $0x1,%edi
  4004fa:       lea    (%rdi,%rdi,1),%eax
  4004fd:       retq
```

What is the value of `%rip` after the `jne` instruction, if the input to the function is **5**?

A. `4004f5`
B. `4004f7`
C. `4004fa`
D. Something else

Code example: `%eflags` and cmp together with jne

```
0000000004004f2 <if_then>:
  4004f2:       cmp    $0x6,%edi
  4004f5:       jne    4004fa
  4004f7:       add    $0x1,%edi
  4004fa:       lea    (%rdi,%rdi,1),%eax
  4004fd:       retq
```

What value is returned from this function if the input to the function is **5**? (need to parse that `lea` instruction!)

A. 5
B. 6
C. 10
D. 11
E. 12
F. Something else

`lea` instruction
(gcc likes to use this, so you might see it in your assign5 bomb)

- "Load effective address"
  - This instruction does some math, usually piecing together a memory address in preparation to do a move

```
lea 0x20(%rsp), %rdi # register %rdi = %rsp + 0x20
```

  - Unlike what we may expect from mov with indirect addressing mode, this does NOT do any memory access

- Use for simple addition
  - Because it just does math, not a dereference, sometimes you'll see gcc use it for addition that has nothing to do with memory addresses

```
lea (%rdi,%rdi,1), %rax # register %rax = %rdi + %rdi
```

  - Why wouldn't gcc just use add? ¯_(ツ)_/¯
  - Actually, there is a reason having to do with hardware

13

If statement construction (`if` **without** `else`)

```
int if_then(int param1)
{
    if (param1 == 6)
        param1++;
    param1 *= 2;
    return param1
}
```

```
# gcc output
0000000004004f2 <if_then>:
  4004f2:       cmp    $0x6,%edi
  4004f5:       jne    4004fa
  4004f7:       add    $0x1,%edi
  4004fa:       lea    (%rdi,%rdi,1),%eax
  4004fd:       retq
```

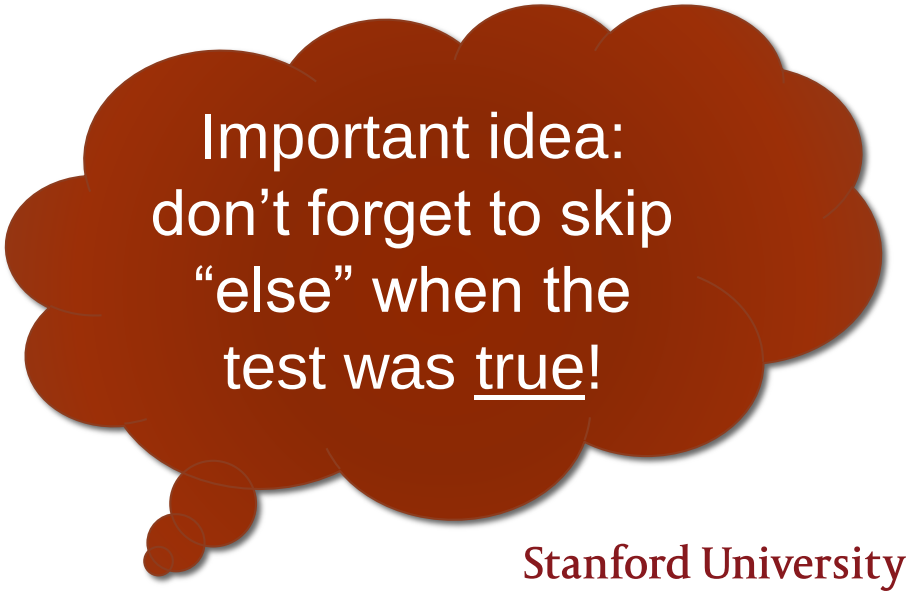
Control flow
C translation examples

15

If-else construction (`if` with `else`)

```
/* if-else */
if (num > 3) {
    x = 10;
} else {
    x = 7;
}
```

```
# equivalent assembly pseudocode
Test
Skip past if-body if test fails
If-body
Skip past else-body
Else-body
[PAST ELSE-BODY]
```



16

For loop construction

```
/* for loop */
for (int i=0; i<n; i++) {
    /* body */
}
```

```
/* equivalent while loop */
int i=0;
while (i<n) {
    /* body */
    i++;
}
```

```
# pseudocode of what gcc actually emits
Initialization
Skip loop Body down to Test
Body
Increment
Test
Return to Body if test succeeds
```

17

For loop construction

```
# pseudocode of what gcc
# actually emits
Initialization
Skip loop Body down to Test
Body
Increment
Test
Return to Body if test succeeds
```

```
# simpler code?
Initialization
Test
Skip past loop if Test fails
Body
Increment
Return back up to Test
```

- Same length! Why does gcc use the format on the left?

Say for loop "`for (int i=0; i<n; i++)`" and **n=0**, **n=1000**

Compare the instructions executed in the left and right

A. LEFT and RIGHT have same number of instructions
B. LEFT has more instructions (bad for left)
C. RIGHT has more instructions (bad for right)
D. Other/help

Computer Architecture BIG IDEA:
Code with Smaller Static Instruction Count != Code with Smaller Dynamic Instruction Count

- Our two codes had the same number of instructions
  - **Same** static instruction count
- If loop never executes, right had **higher** dynamic instruction count (bad for right)
- If loop executes many times, left had **higher** dynamic instruction count (bad for left)
- **This lack of correlation is very common!**
  - There are even cases where the compiler emits a static instruction count that is *several times* longer than an alternative, yet still more efficient assuming loops execute many times (e.g. loop unrolling)

Discussion question:

- Does the compiler **know** that the loop will execute many times?
  - In general, no!
- So…what if our code has loops that always execute a small number of times? Did gcc make a bad decision?

(take EE108, EE180, CS316 for more)

Some instructions you might see in your bomb

ASSIGN5 HOMEWORK HELP

movbz/movbs instructions

- "Move byte **zero**-extend" and "Move byte **sign**-extend"

```
movzbl %al, %edx
```

  - Copy low (least-significant) byte from register %eax, zero-extend to 4 bytes wide in %edx

```
movsbl %al, %edx
```

  - Copy low (least-significant) byte from register %eax, sign-extend to 4 bytes wide in %edx

  - Sometimes you'll see this as a way to zero out the top bytes of a register

```
movzbl %al, %eax  # notice src, dst are the same
```

`nop/nopl` instruction

- This instruction is pronounced “no-op,” which is short for “no operation”
- Literally it means to do nothing
  - Only increments `%rip`
- gcc sometimes inserts them because ¯_(ツ)_/¯
  - Actually the reason is for padding to make functions align on nice multiple-of-8 memory address boundaries or something like that

- Also gives rise to a derogatory slang usage you may have heard from computer scientists (e.g., “That person/thing is kind of a nop to me.”) meaning someone or something that doesn’t necessarily do harm, but is useless or unhelpful

Nuance of mov instruction

- Sometimes you'll see this puzzlement in your code:

```
mov %ebx, %ebx
```

  - What is that doing? Looks like a `nop`!

- gcc is likely using it to zero out the top 32 bits of the register
- When mov insruction is performed on a register whose name starts with "e" (the 32-bit portion), the rest of the 64 bits (the part of the corresponding "r"-named register beyond the "e" part) are cleared out to all zeros
- Same as `movbzl`

Another strangely used instruction: xor

- Sometimes you'll see this puzzlement in your code:

```
xor %ebx, %ebx
```

  - What is that doing? XOR of a value with itself is always 0.
  - So it's setting ebx to zero? Why not:

```
mov $0, %ebx
```

- For strange processor hardware reasons, this may be faster (similar reasons as to why gcc would choose `lea` instead of `add`)