Lecture 4

<2016-04-06 Wed>

Contents

1 Condition Codes

- Single bit registers
  - CF: carry flag
  - ZF: zero flag
  - SF: sign flag
  - OF: overflow flag

1.1 Not set by `leaq` instruction

1.2 Implicitly Set by Arithmetic Operations

1.2.1 example

```
addq src, dest
   t = a + b
```

- CF set if carried out from most significant bit (unsigned overflow)
- ZF set if `t == 0`
- SF set if `t < 0` (signed)
- OF set if 2's complement overflow
  - `(a<0 && b<0 && t>=0) || (a>0 && b>0 && t<0)`

1.3 Explicitly Set

1.3.1 By Compare Instruction

`cmpq b, a` : computing `a - b` without destination

- CF set if carry out from most significant bit (unsigned comparison)
- ZF set if `a == b`
- SF set if `(a - b) < 0` (signed)
- OF set if 2's complement overflow (signed)
  - `(a<0 && b>0 && (a-b)>0) || (a>0 && b<0 && (a-b)<0)`

1.3.2 By Test Instruction

`testq b, a` : computing `a & b` without setting destination

- ZF set if `a & b == 0`
- SF set if `a & b < 0`

1.4 Reading conditional codes

1.4.1 setX Instructions

- set low-order byte of destination to 0 or 1 based on combinations of condition codes
- does not alter remaining 7 bytes
  - use `movzbl` to set upper bits to 0

instruction	
`movsXY`	move a byte and sign extend it
`movzXY`	move a byte and 0 extend it
example	explanation
`movsbl`	move a byte from src to dest and sign extend it to long
`movzbl`	move a byte from src to dest and 0 extend it to long
`movsbl $0xFF %eax`	`%eax = 0xFFFFFFFF`
`movzbl $0xFF %eax`	`%eax = 0x000000FF`

setX	condition	description
`sete`	`ZF`	equal / zero
`setne`	`~ZF`	not equal / not zero
`sets`	`SF`	negative
`setns`	`~SF`	nonnegative
`setg`	`~(SF^OF) & ~ZF`	greater (signed)
`setge`	`~(SF^OF)`	greater or equal (signed)
`setl`	`(SF^OF)`	less (signed)
`setle`	`(SF^OF) | ZF`	less or equal (signed)
`seta`	`~CF & ~ZF`	above (unsigned)
`setb`	`CF`	below (unsigned)

Table 1: x84-64 integer registers

8 bytes registers	lower-order 4 bytes	lower-order 1 byte	note
%rax	%eax	%al	
%rbx	%ebx	%bl	
%rcx	%ecx	%cl	
%rdx	%edx	%dl	
%rsi	%esi	%sil	
%rdi	%edi	%dil	
%rsp	%esp	%spl	stack pointer
%rbp	%ebp	%bpl	base pointer
%r8	%r8d	%r8b	
%r9	%r9d	%r9b	
%r10	%r10d	%r10b	
%r11	%r11d	%r11b	
%r12	%r12d	%r12b	
%r13	%r13d	%r13b	
%r14	%r14d	%r14b	
%r15	%r15d	%r15b	

1.4.2 example

```
long gt(long x, long y) {
  return x > y;
}
```

converted to assmebly

8 byte register	lower-order 1 byte	variable
%rdi		x
%rsi		y
%rax	%al	return value

```
cmpq    %rsi, %rdi              ;compare x, y
setg    %al                     ;set lower-order byte of %rax to comparison result
movzbl  %al, %rax               ;set upper 7 byte of %rax to 0
```

2 Conditional Branch / Move

2.1 Jumping, Conditional Branch

- jump to different part of code depending on condition codes

jX	condition	description
`jmp`	`1`	unconditional
`je`	`ZF`	equal / zero
`jne`	`~ZF`	not equal / not zero
`js`	`SF`	negative
`jns`	`~SF`	nonnegative
`jg`	`~(SF^OF) & ~ZF`	greater (signed)
`jge`	`~(SF^OF)`	greater or equal (signed)
`jl`	`(SF^OF)`	less (signed)
`jle`	`(SF^OF) | ZF`	less or equal (signed)
`ja`	`~CF & ~ZF`	above (unsigned)
`jb`	`CF`	below (unsigned)

2.1.1 Conditional Branch example

```
long absdiff(long x, long y) {
  long result;
  if (x > y)
    result = x - y;
  else
    result = y - x;
  return result;
}
```

register	variable
`%rdi`	`x`
`%rsi`	`y`
`%rax`	`result`

```
absdiff:
      cmpq      %rsi, %rdi       ;compare x, y
      jle       .L4              ;jump if x <= y
      movq      %rdi, %rax       ;%rax = %rdi (result = x)
      subq      %rsi, %rax       ;%rax = %rax - %rsi (result -= y)
      ret
.L4:
      movq      %rsi, %rax       ;%rax = %rsi (result = y)
      subq      %rdi, %rax       ;%rax = %rax - %rdi (result -= x)
      ret
```

Express with goto code

```
long absdiff_j(long x, long y) {
  long result;
  int ntest = (x <= y);
  if (ntest)
    goto Else;

  result = x - y;
  goto Done;

Else:
  result = y - x;

Done:
  return result;
}
```

2.2 Conditional Move

- Conditional Move Instructions
  - instruction supports
    - `if (Test) Dest <= Src`
  - branches are disruptive to instruction flow through pipelines
  - conditional moves do not require control transfer

2.2.1 Bad cases for conditional move

Both values get computed

- expensive computations
  - `test(x) ? Hard1(x) : Hard2(x)`
  - both values get computed
  - only make sense when computations are simple
- risky computation
  - `p ? *p : 0`
  - both values get computed
  - may have undesirable result

- computation with side effect
  - `x > 0 ? x *= 7 : x += 3`
  - both values get computed
  - must be side effect free

2.2.2 conditional move example

```
long absdiff_move(long x, long y) {
  long result;
  result = (x > y) ? (x - y) : (y - x);
  return result;
}
```

- `cmovle` : conditional move when less than or equal to
- using conditional move

register	variable
`%rdi`	`x`
`%rsi`	`y`
`%rax`	`result`
`%rdx`	intermediate value

```
absdiff_move:
      movq     %rdi, %rax      ;%rax = %rdi (result = x)
      subq     %rsi, %rax      ;%rax = %rax - %rsi (result -= y)
      movq     %rsi, %rdx      ;%rdx = %rsi (alternate_result = y)
      subq     %rdi, %rdx      ;%rdx = %rdx - %rdi (alternate_result -= x)
      cmpq     %rsi, %rdi      ;compare x, y
      cmovle   %rdx, %rax      ;move %rdx to %rax only when x <= y
      ret
```

3 Loop

3.1 Do-While Loop

- do-while

```
do {
  body;
} while (test);
```

- goto

```
loop:
  body;
  if (test)
    goto loop;
```

3.1.1 example

- count number of 1's in argument x
- do-while version

```
long pcount_do(unsigned long x) {
  long result = 0;
  do {
    result += x & 0x1;
    x >> 1;
  } while (x);
  return result;
}
```

- goto version

```
long pcount_goto(unsigned long x) {
  long result = 0;
loop:
  result += x & 0x1;
  x >> 1;
  if (x)
    goto loop;
  return result;
}
```

register	variable
`%rdi`	`x`
`%rax`	`result`

```
      movl      $0, %rax        ;result = 0
.L2:
      movq      %rdi, %rdx      ;%rdx = %rax
      andl      $1,   %edx      ;t = x & 0x1
```

```
    addq      %rdx, %rax        ;result += t
    shrq      %rdi              ;x >>= 1
    jne       .L2               ;if (x) goto loop
    ret
```

3.2 While Loop

- while loop

```
while (test)
    body;
```

- goto

```
goto test;
loop:
  body;
test:
  if (test)
    goto loop;
done:
```

3.3 For Loop

```
for (init; test; update)
  body;
```

equivalent to

```
init;
while (test) {
  body;
  update;
}
```

4 Switch

- multiple case label
- fall through cases
- missing cases

```
long switch_eg(long x, long y, long z) {
  long w = 1;
  switch (x) {
  case 1:
    w = y*z;
    break;
  case 2:
    w = y/z;
    /* fall through */
  case 3:
    w += z;
    break;
  case 5:
  case 6:
    w -= z;
    break;
  default:
    w = 2;
  }
  return w;
}
```

register	variable
%rdi	x
%rsi	y
%rdx	z
%rax	return value

```
        ;; jump table
.section        .rodata
      .align 8
.L4:
      .quad     .L8             ;x = 0
      .quad     .L3             ;x = 1
      .quad     .L5             ;x = 2
      .quad     .L9             ;x = 3
      .quad     .L8             ;x = 4
      .quad     .L7             ;x = 5
      .quad     .L7             ;x = 6

switch_eg:
```

```
        movq      %rdx, %rcx
        cmpq      $6,   %rdi
        ja        .L8               ;default
        jmp       *.L4(,%rdi,8)

          ;; (x == 1)
.L3:
        movq      %rsi, %rax
        imulq     %rdx, %rax
        ret
          ;; x == 2
.L5:
        movq      %rsi, %rax
        cqto
        idivq     %rcx
        jmp       .L6
          ;; x == 3
.L9:
        movl      $1, %eax
          ;; fall through
.L6:
        addq      %rcx, %rax
        ret
          ;; x == 5, x == 6
.L7:
        movq      $1, %eax
        subq      %rdx, %rax
        ret
          ;; default
.L8:
        movl      $2, %eax
        ret
```

- Explanation
  - table structure
    - each target requires 8 bytes
    - base address at .L4
  - jumping
    - direct: `jmp .L8`

    - go to instruction at address (label) .L8
  - indirect: `jmp *.L4(,%rdi,8)`
    - go to instruction at address as computed by address computation