Control

Young W. Lim

2020-04-16 Thr

Outline

Based on

"Computer Architecture: A Programmer's Perspective",
Bryant & O'Hallaron

Compling 32-bit program on 64-bit gcc

- `gcc -v`
- `gcc -m32 t.c`
- `sudo apt-get install gcc-multilib`
- `sudo apt-get install g++-multilib`
- `gcc-multilib`
- `g++-multilib`
- `gcc -m32`
- `objdump -m i386`

Condition Codes

- CF (Carry Flag)
  - the most recent operation generated a carry out of the msb
  - used to detect overflow for unsigned operations
- ZF (Zero Flag)
  - the most recent operation yielded zero
- SF (Sign Flag)
  - the most recent operation yielded a negative value
- OF (Overflow Flag)
  - the most recent operation caused a 2's complement overflow either neagtive or positive

Condition Codes and C Expressions

- CF : `(unsigned t) < (unsigned a)` Unsigend Overflow
- ZF : `(t == 0)`
- SF : `(t < 0)`
- OF : `(a < 0 == b < 0) && (t < 0 != a < 0)`

- `t = a+b` assumed

CMP instruction

- cmpb op1, op2
- cmpw op1, op2
- cmpl op1, op2

- NULL ← op2 - op1
  - subtracts the contents of the src operand op1 from the dest operand
  - discard the results, only the flag register is affected

Condition	Signed Compare	Unsigned Compare
op1 < op2	ZF = 0 && SF = OF	CF = 0 && ZF = 0
op1 <= op2	SF == OF	CF == 0
op1 == op2	ZF == 1	ZF == 1
op1 >= op2	ZF = 1 or SF ! OF	CF == 1 or ZF ==1
op1 > op2	SF != OF	CF ==1

- t = a+b assumed

TEST instruction

- testb src, dest
- testw src, dest
- testl src, dest

- NULL ← dest & src
  - ands the contents of the src operand with the dest operand
  - discard the results, only the flag register is affected

Condition Code Examples

```
addl
t=a+b

CF: (unsigned t) < (unsigned a)
    mag(t) < mag(a) if C=1

ZF: (t == 0)
    zero t
SF: (t < 0)
    negative t

OF: (a<0 == b<0) && (t<0 != a<0)
    sign(a) == sign(b) != sign(t)
```

Compare and Test (1)

`cmpb S2, S1`	compare bytes	`S1 - S2`
`testb S2, S1`	test bytes	`S1 & S2`
`cmpw S2, S1`	compare words	`S1 - S2`
`testw S2, S1`	test words	`S1 & S2`
`cmpl S2, S1`	compare double words	`S1 - S2`
`testl S2, S1`	test double words	`S1 & S2`

Compare and Test (2)

`cmpb S2, S1`	compare bytes	`-S2 + S1`
`testb S2, S1`	test bytes	`S2 & S1`
`cmpw S2, S1`	compare words	`-S2 + S1`
`testw S2, S1`	test words	`S2 & S1`
`cmpl S2, S1`	compare double words	`-S2 + S1`
`testl S2, S1`	test double words	`S2 & S1`

Set

sete D	setz	D ← ZF	(equal / zero)
setne D	setnz	D ← ~ZF	(not equal/ not zero)
sets D		D ← SF	(negative)
setns D		D ← ~SF	(non-negative)
setg D	setle	D ← ~(SF^OF)&~ZF	(greater, signed >)
setge D	setnl	D ← ~(SF^OF)	(greater or equal, signed >=)
setl D	setnge	D ← SF^OF	(less, signed <)
setle D	setng	D ← (SF^OF) | ZF	(less or equal, signed <=)
seta D	setnbe	D ← ~CF&~ZF	(above, usnigned >)
setae D	setnb	D ← ~CF	(above or euqal, unsinged >=)
setb D	setnae	D ← CF	(below, unsigned <)
setbe D	setna	D ← CF&~ZF	(below or equal, unsigned <=)

Jump instruction encoding

- a jump instruction can cause the execution
  to switch to a completely new position in the program
- these jump destinations are generally indicated by a label
- in generating the object code file
  - the assembler determines the addresses of all labled instructions
  - and encodes the jump targets as part of the jump instruction

`jmp` instruction

- the `jmp` instruction jumps unconditionally
- direct jump
  - the jump target is enconded as part of instruction
  - give a label as the jump target
- indirect jump
  - the jump target is read from a register or a memory location
  - using * followed by an operand specifier
  - `jmp *%eax` uses the value in register `%eax` as the jump target
  - `jmp *(%eax)` reads the jump target from memory using the value in `%eax` as the read address

Conditional jump instructions

- the other jump instructions either jump or continue executing at the next instruction in the code sequence depending on some combination of the condition codes
- like `set` instruction
- the underlying machine instructions have multiple names
- conditional jumps can only be direct

Jump instructions

jmp Label		(1)	direct
jmp *Operand		(1)	indirect
je Label	jz	(ZF)	equal/zero
jne Label	jnz	(~ZF)	not equal / non-zero
js Label		(SF)	negative
jns Label		(~SF)	non-negative
jg Label	jnle	(~(SF^OF)&~ZF)	greater, signed >
jge Label	jnl	(~(SF^OF))	greater or equal, signed >=
jl Label	jnge	((SF^OF))	less, signed <
jle Label	jng	((SF^OF)	less or equal, signed <=
ja Label	jnbe	(~CF&~ZF)	above, unsigned >
jae Label	jnb	(~CF)	above or equal, unsigned >=
jb Label	jnae	(CF)	below, unsigned <
jbe Label	jna	(CF&~ZF)	below or equal, unsigned <=

PC-relative addressing

- jump relative

jump	.... offset ....

- effective PC address = next instruction address + *offset*
  (offset may be negative)
- particularly useful in connection with jumps,
  because typical jumps are to nearby instructions
- most `if` or `while` statements are reasonably short
- another advantage is +position-independent_ code

https://en.wikipedia.org/wiki/Addressing_mode#PC-relative

Encoding format of object code

- the format of object code
- understanding how the targets of jump instructions are encoded will be important
    - when studying linking process
    - interpreting the output of a disassembler
- In assembly code, jump targets are written using symbolic labels
- the assembler, and later the linker, generate the proper encodings of the jump targets
- there are several different encodings of for jumps, but some of the most commonly used ones are PC-relative

`Computer Architecture : A Programmer's Prespective`

Encoding jump instructions

- PC-reltive
  - encodes the difference between the address of the target instruction the address of the instruction immediately following the jump
  - these offsets can be encoded using 1, 2, or 4 bytes
- Absolute
  - directly specify the target address using 4 bytes
- the assembler and linker select the appropriate encodings

an example of PC-relative addressing

```
  jle .L4           If <=, goto dest2
   .p2align 4,,7    Aligns next instruction to multpile of 8
.L5:                dest1:
  movl %edx, %eax
  sarl $1, %eax
  subl %eax, %edx
  jg .L5            If >, goto dest1
.L4:                dest2:
  movl %edx, %eax
```

`Computer Architecture : A Programmer's Prespective                Computer Arch`

simulations

```
  jle .L4
   .p2align 4,,7
.L5:                 dest1:
  movl %edx, %eax    0111 -> 0111   0100 -> 0100   0010 -> 0010  0001 -> 0001
  sarl $1, %eax      0111 -> 0011   0100 -> 0010   0010 -> 0001  0001 -> 0000
  subl %eax, %edx         -> 0100           -> 0010          -> 0001         -> 0001
  jg .L5             If >, goto dest1
.L4:                 dest2:
  movl %edx, %eax

  jle .L4
   .p2align 4,,7
.L5:                 dest1:
  movl %edx, %eax    0100-> 0100  0010 -> 0010   0001 -> 0001  0001 -> 0001
  sarl $1, %eax      0100-> 0010  0010 -> 0001   0001 -> 0000  0001 -> 0000
  subl %eax, %edx        -> 0010         -> 0001         -> 0001         -> 0001
  jg .L5             If >, goto dest1
.L4:                 dest2:
  movl %edx, %eax
```

Computer Architecture : A Programmer's Prespective

disassembled version

```
 8: 7e 11                jle  1b <silly+0x1b>    Target = dest2
 a: 8d b6 00 00 00 00    lea  0x0(%esi),%esi     Added nops
10: 89 d0                mov  %edx,%eax          dest1:
12: c1 f8 01             sar  $0x1,%eax
15: 29 c2                sub  %eax,%edx
17: 85 d2                test %edx,%edx          %edx & %edx = %edx
19: 7f f5                jg   10 <silly+0x10>    Target = dest1
1b: 89 d0                mov  %edx,%eax          dest2:
```

Computer Architecture : A Programmer's Prespective

disassembled version notes (1)

```
1.   8: 7e 11                jle  1b <silly+0x1b>    Target = dest2
```

- no real effects, serves as 6-byte `nop`
- to make the address of the next instruction a multiple of 16

`Computer Architecture : A Programmer's Prespective`

disassembled version notes (2)

```
1.    8: 7e 11                jle  1b <silly+0x1b>    Target = dest2
2.    a: 8d b6 00 00 00 00    lea  0x0(%esi),%esi     Added nops
```

- jump target : `0x1b (27)`
- jump target encoding : `0x11 + 0xa = 0x1b (17 + 10 =27)`
- next instruction address : `0xa (10)`

`Computer Architecture : A Programmer's Prespective`

disassembled version notes (3)

```
7.   19: 7f f5                jg   10 <silly+0x10>    Target = dest1
8.   1b: 89 d0                mov  %edx,%eax          dest2:
```

- jump target : `0x10 (16)`
- jump target encoding : `0xf5 + 0x1b = 0xf5 (-11 + 27 =16)`
- next instruction address : `0x1b (27)`

`Computer Architecture : A Programmer's Prespective`

the value of PC

- the value of the program counter
  when performing PC-relative addressing
  is the address of the instruction following the jump
  not the address of the jump instruction
- the processor would update the program counter
  as its frist step in executing an instruction

`Computer Architecture : A Programmer's Prespective`

disassembled version after linking

```
80483c8: 7e 11                 jle  79473db <silly+0x1b>  Target = dest2
80483ca: 8d b6 00 00 00 00     lea  0x0(%esi),%esi        Added nops
80483d0: 89 d0                 mov  %edx,%eax             dest1:
80483d2: c1 f8 01              sar  $0x1,%eax
80483d5: 29 c2                 sub  %eax,%edx
80483d7: 85 d2                 test %edx,%edx             %edx & %edx = %edx
80483d9: 7f f5                 jg   80483d0 <silly+0x10>  Target = dest1
80483db: 89 d0                 mov  %edx,%eax             dest2:
```

`Computer Architecture : A Programmer's Prespective`

disassembled version after linking notes

```
1.   8: 7e 11             jle  1b <silly+0x1b>    Target = dest2

7.  19: 7f f5             jg   10 <silly+0x10>    Target = dest1

1.  80483c8: 7e 11        jle  79473db <silly+0x1b>  Target = dest2

7.  80483d9: 7f f5        jg   80483d0 <silly+0x10>  Target = dest1
```

- the instruction have been relocated to different addresses, but the encondings of the jump targets in line 1 and line 7 remain unchanged
- by using PC-relative encoding of the jump targets, the instructions can be completely encoded (requiring just two bytes) and the object code can be shifted to different positions in memory without modification.

`Computer Architecture : A Programmer's Prespective`

if-else statement

```
if (expr)
  then-statement
else
  else-statement
```

```
t = exorl
if (t)
  goto true;
  // else-statement
  goto done;
true:
  // then-statement
done:
```

if-else exmple (1)

```
int abs(int x, int y)
{
  if (x<y)
    return y-x;
  else
    return x-y;
}
```

```
int abs_goto(int x, int y)
{
  int val;

  if (x < y)
    goto true;
  val = x - y;
  goto done;
true:
  val = y - x;
done:
  return val;
}
```

if-else exmple (2)

```
movl 8(%ebp), %edx
movl 12(%ebp), %3ax
cmpl %eax, %edx
jl   .L3
subl %eax, %edx
movl %edx, %eax
jmp  .L5
.L3:
subl %edx, %eax
.L5:
```

```
int abs_goto(int x, int y)
{
  int val;

  if (x < y)
    goto true;
  val = x - y;
  goto done;
true:
  val = y - x;
done:
  return val;
}
```

do-while statement

```
do
  // body-statement
while (expr);
```

```
loop:
  // body-statement
  t = expr;
  if (t)
    goto loop:
```

while statement (1)

```
while (expr)
{
  // body-statement
}
```

```
loop:
  t = expr;
  if (!t)
    goto done;
  // body-statement
  goto loop;
done:
```

while statement (2)

```
if (!expr)
  goto done;
do
  // body-statement
while (expr);
done:
```

```
t = expr;
if (!t)
  goto done;
loop:
  // body-statement
  t = expr;
  if (t)
    goto loop;
done;
```

for statement (1)

```
for (init; test; update)
  // body-statement
```

```
init_expr;
while (test) {
  // body-statement
  update;
}
```

for statement (2)

```
init_expr;
if (!test)
  goto done;
do {
  // body-statement
  update;
} while (test);
done:
```

```
init_expr;
t = test;
if (!t)
  goto done;
loop:
  // body-statement
  update_expr;
  t = test;
  if (t)
    goto loop;
done;
```

Switch statement (1)

```
int switch_example(int x)
{
  int result = x;

  switch (x) {
    case 100: result *= 13; break;
    case 102: result += 10;
    case 103: result += 11; break;
    case 104:
    case 105: result *= result; break;
    default:  result = 0;
  }

  return result;
}
```

Switch statement (2)

```
code *jt[7] =
{loc_A, loc_def, loc_B, loc_C, loc_D, loc_def, loc_D };

int switch_tanslated(int x)
{
  unsigned xi = x - 100;
  int result = x;
  if (xi>6) goto loc_def;
  goto jt[xi];
loc_A: result *= 13; goto done;
loc_B: result += 10; goto done;
loc_C: result += 11; goto done;
loc_D: result *= result; goto done;
loc_def: result = 0;
done: return result;
}
```

Switch statement (3)

```
leal -100(%edx), %eax    ;; xi = x-100
cmpl $6, %eax             ;; compare xi:6
ja .L9                    ;; if >, go to loc_def
jmp *.L10(,%eax,4)        ;; goto jt[xi]

;; Case 100
.L4:                     ;; loc_A
leal (%edx,%edx,2), %eax ;; 3*x
leal (%edx,%eax,4), %edx ;; x+4*3x
jmp .Le                  ;; goto done

;; Case 102
.L5:                     ;; loc_B
addl $10, %edx           ;; result += 10
```

Switch statement (4)

```
;; Case 103
.L6:                        ;; loc_C
addl $11, %edx              ;; result += 11
jmp .L3                     ;; goto done

;; Cases 104, 106
.L8:                        ;; loc_D
imull %edx, %edx            ;; result *= result
jmp .L3                     ;; goto done

;; Default Case
.L9:                        ;; loc_defa
xorl %edx, %edx             ;; result = 0
```

Switch statement (5)

```
;; Return Result
.L3:                        ;; done
movl %edx, %eax             ;; set return value
```