Machine-Level Programming II: Arithmetic &Control

Kai Shen

1

Outline

- Arithmetic operations
- Control: Condition codes
- Conditional branches
- Loops
- Switch

2

Some Arithmetic Operations

- Two Operand Instructions:

Format	*Computation*		
`addl`	*Src,Dest*	Dest = Dest + Src	
`subl`	*Src,Dest*	Dest = Dest – Src	
`imull`	*Src,Dest*	Dest = Dest * Src	
`sall`	*Src,Dest*	Dest = Dest << Src	*Also called shll*
`sarl`	*Src,Dest*	Dest = Dest >> Src	*Arithmetic*
`shrl`	*Src,Dest*	Dest = Dest >> Src	*Logical*
`xorl`	*Src,Dest*	Dest = Dest ^ Src	
`andl`	*Src,Dest*	Dest = Dest & Src	
`orl`	*Src,Dest*	Dest = Dest | Src	

- One Operand Instructions

`incl`	*Dest*	*Dest = Dest + 1*
`decl`	*Dest*	*Dest = Dest – 1*
`negl`	*Dest*	*Dest = – Dest*
`notl`	*Dest*	*Dest = ~Dest*

3

Arithmetic Example

```
int arith(int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```

Stack

Offset	
	•
	•
	•
16	z
12	y
8	x
4	Rtn Addr
0	Old %ebp ← %ebp

```
movl   8(%ebp), %ecx          # ecx = x
movl   12(%ebp), %edx         # edx = y
leal   (%edx,%edx,2), %eax    # eax = y*3
sall   $4, %eax               # eax *= 16 (t4)
leal   4(%ecx,%eax), %eax     # eax = t4 +x+4 (t5)
addl   %ecx, %edx             # edx = x+y (t1)
addl   16(%ebp), %edx         # edx += z (t2)
imull  %edx, %eax             # eax = t2 * t5 (rval)
```

4

Outline

- Arithmetic operations
- Control: Condition codes
- Conditional branches
- Loops
- Switch

5

Processor State (IA32, Partial)

- Information about currently executing program
  - Temporary data (`%eax`, …)
  - Location of runtime stack (`%ebp`,`%esp`)
  - Location of current code control point (`%eip`, …)
  - Status of recent tests (**CF, ZF, SF, OF**)

`%eax`	General purpose registers
`%ecx`	
`%edx`	
`%ebx`	
`%esi`	
`%edi`	
`%esp`	Current stack top
`%ebp`	Current stack frame
`%eip`	Instruction pointer
CF ZF SF OF	Condition codes

6

Condition Codes (Arithmetic Setting)

- Single bit registers
  - **CF** Carry Flag (for unsigned) **SF** Sign Flag (for signed)
  - **ZF** Zero Flag **OF** Overflow Flag (for signed)
- Set by arithmetic operations

  Example: **`addl/addq`** *Src,Dest* ↔ **`t = a+b`**

  CF set if carry out from most significant bit (unsigned overflow)

  ZF set if **`t == 0`**

  SF set if **`t < 0`** (as signed)

  OF set if two's-complement (signed) overflow

  `(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)`
- Not set by **`lea`** instruction

7

Condition Codes (Setting by Compare)

- Setting by compare instruction
  - **`cmpl`** *Src2, Src1*
  - **`cmpl b,a`** like computing **`a-b`** without setting destination

  - **CF set** if carry out from most significant bit (used for unsigned comparisons)
  - **ZF set** if **`a == b`**
  - **SF set** if **`(a-b) < 0`** (as signed)
  - **OF set** if two's-complement (signed) overflow

  `(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)`

8

Condition Codes (Setting by Test)

- Setting by Test instruction
  - **testl** *Src2, Src1*
  testl b,a like computing **a&b** without setting destination

  - Sets condition codes based on value of *Src1* & *Src2*
  - Useful to have one of the operands be a mask

  - **ZF set** when **a&b == 0**
  - **SF set** when **a&b < 0**

9

Reading Condition Codes

- SetX Instructions:
  - Set single byte based on combination of condition codes
- One of 8 addressable byte registers
  - Does not alter remaining 3 bytes
  - Typically use **movzbl** to finish job

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

```
movl 12(%ebp),%eax    # eax = y
cmpl %eax,8(%ebp)     # Compare x : y
setg %al              # al = x > y
movzbl %al,%eax       # Zero rest of %eax
```

%eax	%ah	%al
%ecx	%ch	%cl
%edx	%dh	%dl
%ebx	%bh	%bl
%esi		
%edi		
%esp		
%ebp		

10

Reading Condition Codes

- SetX instructions
  - Set single byte based on combinations of condition codes

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)

11

Outline

- Arithmetic operations
- Control: Condition codes
- Conditional branches & Moves
- Loops
- Switch

12

Jumping

- jX Instructions
  - 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)

13

Conditional Control

- In assembly/machine code, jump is pretty much all we got
- But there are a variety of conditional control in high-level languages
  - if … else …
  - switch(…) {}
  - while loop
  - for loop
  - … …

14

Conditional Branch Example

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

```
absdiff:
    pushl   %ebp             } Setup
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L6
    subl    %eax, %edx
    movl    %edx, %eax
    jmp .L7
.L6:
    subl %edx, %eax
.L7:
    popl %ebp                } Finish
    ret
```

15

Conditional Branch Example (Cont.)

```
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
  goto Exit;
Else:
  result = y-x;
Exit:
  return result;
}
```

- Better resemblance to “goto”-style C code
- Bad coding style, but that’s not our concern here

```
absdiff:
    pushl   %ebp             } Setup
    movl    %esp, %ebp
    movl    8(%ebp), %edx    } Body1
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L6
    subl    %eax, %edx       } Body2a
    movl    %edx, %eax
    jmp .L7
.L6:
    subl %edx, %eax          } Body2b
.L7:
    popl %ebp                } Finish
    ret
```

16

Using Conditional Moves

- Conditional move instructions
  - if (Test) Dest ← Src
- Why?
  - Branches are very disruptive to instruction flow through pipelines
  - Cond-mov require no control transfer

C Code

```
val = Test
   ? Then_Expr
   : Else_Expr;
```

Cond-mov Version

```
tval = Then_Expr;
result = Else_Expr;
t = Test;
if (t) result = tval;
return result;
```

17

Outline

- Arithmetic operations
- Control: Condition codes
- Conditional branches and moves
- Loops
- Switch

18

Loop Control

- In assembly/machine code, jump is pretty much all we got for conditional control (similar to "goto" in C)
- Convert typical loop control into goto-based C code.
  - Familiarize with assembly/machine programming
  - Understand compiler transformation
  - Able to read/comprehend assembly/machine code

19

"Do-While" Loop Example

C Code

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

Goto Version

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

- Count number of 1's in argument x
- Use conditional branch to either continue looping or to exit loop

20

"Do-While" Loop Compilation

Goto Version

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

- Registers:
  %edx   x
  %ecx   result

```
    movl   $0, %ecx     #   result = 0
.L2:       # loop:
    movl   %edx, %eax
    andl   $1, %eax     #   t = x & 1
    addl   %eax, %ecx   #   result += t
    shrl   %edx         #   x >>= 1
    jne    .L2          #   If !0, goto loop
```

21

General "Do-While" Translation

C Code

```
do
  Body
  while (Test);
```

Goto Version

```
loop:
  Body
  if (Test)
    goto loop
```

22

General "While" Translation

While version

```
while (Test)
  Body
```

Do-While Version

```
  if (!Test)
    goto done;
  do
    Body
    while(Test);
done:
```

Goto Version

```
  if (!Test)
    goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```

23

General "For" Translation

For Version

```
for (Init; Test; Update )
    Body
```

While Version

```
Init;
while (Test ) {
    Body
    Update;
}
```

Do-While Version

```
  Init;
  if (!Test)
    goto done;
  do
    Body
    Update
  while(Test);
done:
```

Goto Version

```
  Init;
  if (!Test)
    goto done;
loop:
  Body
  Update
  if (Test)
    goto loop;
done:
```

24

"For" Loop Conversion Example

C Code

```
#define WSIZE 8*sizeof(int)
int pcount_for(unsigned x) {
  int i;
  int result = 0;
  for (i = 0; i < WSIZE; i++) {
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
  return result;
}
```

- Initial test can be optimized away

Goto Version

```
int pcount_for_gt(unsigned x) {
  int i;
  int result = 0;
  i = 0;                          Init
  if (!(i < WSIZE))               ! Test
    goto done;
 loop:
  {                               Body
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
  i++;  Update
  if (i < WSIZE)  Test
    goto loop;
 done:
  return result;
}
```

25

Outline

- Arithmetic operations
- Control: Condition codes
- Conditional branches and moves
- Loops
- Switch

26

Switch Statement

```
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;
}
```

- An example
  - Multiple case labels (5 & 6)
  - Fall through case (2)
  - Missing case (4)
- How to implement switch statement in machine code?
  - Using jumps (or "goto")
- Jump table avoids sequencing through cases
  - constant time, rather than linear

27

Jump Table Structure

Switch Form

```
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```

Jump Table

```
jtab: Targ0
      Targ1
      Targ2
        •
        •
        •
      Targn-1
```

Jump Targets

```
Targ0:   Code Block 0
Targ1:   Code Block 1
Targ2:   Code Block 2
  •
  •
  •
Targn-1: Code Block n–1
```

Approximate Translation

```
target = JTab[x];
goto *target;
```

28

Switch Statement Example (IA32)

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

Jump table

```
.section    .rodata
  .align 4
.JT:
  .long     .L2  # x = 0
  .long     .L3  # x = 1
  .long     .L4  # x = 2
  .long     .L5  # x = 3
  .long     .L2  # x = 4
  .long     .L6  # x = 5
  .long     .L6  # x = 6
```

Setup:

```
switch_eg:
   pushl  %ebp            # Setup
   movl   %esp, %ebp      # Setup
   movl   8(%ebp), %eax   # eax = x
   cmpl   $6, %eax        # Compare x:6
   ja     .L2             # If unsigned > goto default
   jmp    *.JT(,%eax,4)   # Goto *JTab[x]
```

Indirect jump

29

Jump Table

```
.section   .rodata
  .align 4
.JT:
  .long     .L2 # x = 0
  .long     .L3 # x = 1
  .long     .L4 # x = 2
  .long     .L5 # x = 3
  .long     .L2 # x = 4
  .long     .L6 # x = 5
  .long     .L6 # x = 6
```

```
switch(x) {
case 1:       // .L3
    w = y*z;
    break;
case 2:       // .L4
    w = y/z;
    /* Fall Through */
case 3:       // .L5
    w += z;
    break;
case 5:
case 6:       // .L6
    w -= z;
    break;
default:      // .L2
    w = 2;
}
```

30

Discover Jump Table in Machine Code

- First, find the address of the jump table

Disassembled Object Code

```
08048410 <switch_eg>:
 . . .
 8048419:77 07         ja     8048422 <switch_eg+0x12>
 804841b:ff 24 85 60 86 04 08  jmp    *0x8048660(,%eax,4)
```

- Then read into the jump table
  - But the table doesn't show up in disassembled code
  - Can inspect using GDB: `x/7xw 0x8048660`
    - Examine 7 hexadecimal format "words" (4-bytes each)
    - Use command "`help x`" to get format documentation

```
0x8048660:   0x08048422   0x08048432   0x0804843b   0x08048429
0x8048670:   0x08048422   0x0804844b   0x0804844b
```

31

Matching Disassembled Targets

Value
0x8048422
0x8048432
0x804843b
0x8048429
0x8048422
0x804844b
0x804844b

```
8048422:   mov    $0x2,%eax
8048427:   jmp    8048453 <switch_eg+0x43>
8048429:   mov    $0x1,%eax
804842e:   xchg   %ax,%ax
8048430:   jmp    8048446 <switch_eg+0x36>
8048432:   mov    0x10(%ebp),%eax
8048435:   imul   0xc(%ebp),%eax
8048439:   jmp    8048453 <switch_eg+0x43>
804843b:   mov    0xc(%ebp),%edx
804843e:   mov    %edx,%eax
8048440:   sar    $0x1f,%edx
8048443:   idivl  0x10(%ebp)
8048446:   add    0x10(%ebp),%eax
8048449:   jmp    8048453 <switch_eg+0x43>
804844b:   mov    $0x1,%eax
8048450:   sub    0x10(%ebp),%eax
8048453:   pop    %ebp
8048454:   ret
```

32

Summary on Execution Control

- Control in high-level language
  - if-then-else
  - do-while
  - while, for
  - switch
- Control in machine code
  - Conditional jump
  - Conditional move
  - Indirect jump
  - Compiler generates code sequence to implement more complex control
- Standard techniques
  - Loops converted to go-to form
  - Large switch statements use jump tables
  - Sparse switch statements may use decision trees

33

Disclaimer

These slides were adapted from the CMU course slides provided along with the textbook of "Computer Systems: A programmer's Perspective" by Bryant and O'Hallaron. The slides are intended for the sole purpose of teaching the computer organization course at the University of Rochester.

34