3.6 CONTROL

Spring, 2014
System Programming
Euiseong Seo (euiseong@skku.edu)
Today

- Complete addressing mode, address computation (leal)
- Arithmetic operations
- x86-64
- Control: Condition codes
- Conditional branches
- While loops
int absdiff(int x, int y) {
    int result;
    if (x > y) {
        result = x-y;
    } else {
        result = y-x;
    }
    return result;
}
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;
}

C allows “goto” as means of transferring control
° Closer to machine-level programming style

Generally considered bad coding style
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;
}
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;
}
```c
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;
}
```

```
calldiff:
    pushl %ebp
    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
    ret
```
General Conditional Expression Translation

C Code

\[
\text{val} = \text{Test} \ ? \ \text{ThenExpr} : \ \text{ElseExpr};
\]

\[
\text{val} = x>y \ ? \ x-y : y-x;
\]

Goto Version

\[
\text{nt} = \neg \text{Test};
\]

\[
\text{if} (\text{nt}) \ \text{goto} \ \text{Else};
\]

\[
\text{val} = \text{ThenExpr};
\]

\[
\text{goto} \ \text{Done};
\]

Else:

\[
\text{val} = \text{ElseExpr};
\]

Done:

\[
\ldots
\]

- Test is expression returning integer
  - \( = 0 \) interpreted as false
  - \( \neq 0 \) interpreted as true

- Create separate code regions for then & else expressions

- Execute appropriate one
**Using Conditional Moves**

**Conditional Move Instructions**
- Instruction supports:
  - if (Test) Dest ← Src
- Supported in post-1995 x86 processors
- GCC does not always use them
  - Wants to preserve compatibility with ancient processors
  - Enabled for x86-64
  - Use switch -march=686 for IA32

**Why?**
- Branches are very disruptive to instruction flow through pipelines
- Conditional move do not require control transfer

**C Code**

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

**Goto Version**

```c
tval = Then_Expr;
result = Else_Expr;
t = Test;
if (t) result = tval;
return result;
```
int absdiff(int x, int y) {
    int result;
    if (x > y) {
        result = x-y;
    } else {
        result = y-x;
    }
    return result;
}
**BAD CASES FOR CONDITIONAL MOVE**

**Expensive Computations**

```c
val = Test(x) ? Hard1(x) : Hard2(x);
```

- Both values get computed
- Only makes sense when computations are very simple

**Risky Computations**

```c
val = p ? *p : 0;
```

- Both values get computed
- May have undesirable effects

**Computations with side effects**

```c
val = x > 0 ? x*=7 : x+=3;
```

- Both values get computed
- **Must be side-effect free**
Today

- Complete addressing mode, address computation (leal)
- Arithmetic operations
- x86-64
- Control: Condition codes
- Conditional branches and moves
- Loops
Count number of 1’s in argument x ("popcount")

Use conditional branch to either continue looping or to exit loop
“Do-While” Loop Compilation

Goto Version

```c
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

```
.L2:
    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
```
**General “Do-While” Translation**

**C Code**

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

**Goto Version**

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

- **Body:**
  
  ```
  { 
      Statement_1;
      Statement_2;
      ...
      Statement_n;
  }
  ```

- **Test returns integer**
  
  - = 0 interpreted as false
  - ≠ 0 interpreted as true
Is this code equivalent to the do-while version?
While version

\[ \text{while } (Test) \]

\[ \text{Body} \]

Do-While Version

\[ \text{if } (!Test) \]
\[ \text{goto done; } \]
\[ \text{do } \]
\[ \text{Body} \]
\[ \text{while}(Test); \]
\[ \text{done:} \]

Goto Version

\[ \text{if } (!Test) \]
\[ \text{goto done; } \]
\[ \text{loop: } \]
\[ \text{Body} \]
\[ \text{if } (Test) \]
\[ \text{goto loop; } \]
\[ \text{done:} \]
C Code

```c
#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;
}
```

► Is this code equivalent to other versions?
General Form

for (Init; Test; Update)

Body

for (i = 0; i < WSIZE; i++) {
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
}
“For” Loop → While Loop

For Version

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

While Version

```
Init;
while (Test) {
  Body
  Update ;
}
```
"For" Loop → ... → Goto

For Version

```c
for (Init; Test; Update) {
  Body
}
```

While Version

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

Goto Version

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

```c
#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

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

► Today
  ◦ Complete addressing mode, address computation (leal)
  ◦ Arithmetic operations
  ◦ Control: Condition codes
  ◦ Conditional branches & conditional moves
  ◦ Loops

► Next Time
  ◦ Switch statements
  ◦ Stack
  ◦ Call / return
  ◦ Procedure call discipline