Machine-Level Programming: Control

Arquitectura de Computadores
Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto

Luís Nogueira (lmn@isep.ipp.pt)

Today: Machine-Level Programming: Control

- **Control: Condition codes**
- **Accessing the condition codes**
  - Conditional jumps
  - Conditional set of a single byte
  - Conditional transfer of data

Introduction

- **Some constructs in C, such as conditionals, loops, and switches, require conditional execution**
  - The sequence of operations that gets performed depends on the outcomes of tests applied to the data

- **Assembly code provides two basic low-level mechanisms for implementing conditional behavior:**
  1. It tests data values
  2. Either alters the control flow or the data flow based on the result of these tests

Processor state (IA32, partial)

- **Information about currently executing program**
  - Temporary data (`%eax`, `%ebx`, …)
  - Location of runtime stack (`%ebp`,`%esp`)
  - Location of current code control point (`%eip`)
  - Monitor execution (`%eflags`)

`%eax`	General purpose registers
`%ecx`	
`%edx`	
`%ebx`	
`%esi`	
`%edi`	
`%esp`	Current stack top
`%ebp`	Current stack frame
`%eip`	Instruction pointer
`%eflags`	Status, control and system flags

Condition codes

- **Monitor the kind of result produced from the execution of instructions by evaluating the bits of the EFLAGS register**

`CF` `ZF` `SF` `OF` Condition codes

  - CF – Carry Flag (for unsigned)
  - ZF – Zero Flag
  - SF – Sign Flag (for signed)
  - OF – Overflow Flag (for signed)

- **Can be set implicitly or explicitly**

Condition codes: Implicit setting

- **Implicitly set by arithmetic operations**
  - Think of it as side effect
- **Some instructions do not change EFLAGS**
  - `push`, `pop`, `call`, `leal`, …
- **Full documentation link on course website**
  - A reference sheet can be found here (Note: it uses Intel syntax)

Condition codes: Implicit setting example

- **`add` Src,Dest**
  - Is like computing `t = a+b`, setting destination to `t`

- **Sets condition codes based on value of 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, whenever

`(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)`

Condition codes: Explicit setting (*cmp*)

- **`cmp b,a`**
  - Is like computing **`a-b`** without setting destination

- **Sets condition codes based on value of a - b**
  - 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, whenever

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

Condition codes: Explicit setting (*test*)

- **`test b,a`**
  - Is like computing **`a & b`** without setting destination
  - Useful when one of the operands is a mask

- **Sets condition codes based on value of src1 & src2**
  - ZF set when **`a&b == 0`**
  - SF set when **`a&b < 0`**

Accessing the condition codes

- **Rather than reading the condition codes directly, there are three common ways of using them:**

1. **Conditionally jump** to some other part of the program

2. **Set a single byte to 0 or 1** depending on some combination of the condition codes

3. **Conditionally transfer** data

Today: Machine-Level Programming: Control

- Control: Condition codes
- **Accessing the condition codes**
  - Conditional jumps
  - Conditional set of a single byte
  - Conditional transfer of data

Transfer of control

- **You can alter the normal order of execution by transferring control to another section of a program unit or a subprogram**

- **Unconditional transfer of control**
  - Occurs each time a certain point is reached in a program unit

- **Conditional transfer of control**
  - Occurs only when specified conditions are met at a certain point in a program unit
  - The program follows one execution path when a condition holds and another when it does not

Jumping

- **A jump instruction can cause the execution to switch to a completely new position in the program**
  - Either conditionally or unconditionally

- **These jump destinations are indicated in assembly code by a label**

- **When generating the object-code file, the assembler determines the addresses of all labeled instructions**
  - It encodes the addresses of the destination instructions as part of the jump instructions

Jumping

- **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)

Conditional branch example

C code

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

Assembly code

```
absdiff:
    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:
   movl %ebp, %esp
   popl %ebp
   ret
```

Setup

Body1

Body2a

Body2b

Finish

- **Note:** the function can actually return a negative value if one of the subtractions overflows. Our interest here is to demonstrate simple Assembly code, not to implement robust code

Conditional branch (goto version)

- **C allows "goto" as means of transferring control**
  - Closer to machine-level programming style

- **Generally considered bad coding style in C**
  - Its use can make code very difficult to read and debug

- **We will use it as a way to construct C programs that describe the control flow of assembly-code programs**
  - We call this style of programming "goto code"

Conditional branch (goto version) example

C code

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

Goto version

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

Conditional branch (goto version) example

Goto version

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

Assembly code

```
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:
    movl %ebp, %esp         } Finish
    popl %ebp
    ret
```

General conditional expression translation

C code

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

Example

```
val = x>y ? x-y : y-x;
```

Goto version

```
  if (!Test)
    goto Else;
  val = Then_Expr;
  goto Done;
Else:
  val = Else_Expr;
Done:
  . . .
```

- **Test is expression returning integer**
  - = 0 interpreted as false
  - ≠ 0 interpreted as true
- **Create separate code regions for *then* and *else* expressions**
- **Execute appropriate one**

General conditional expression translation

Assembly code

```
    cmpX    Arg1, Arg2
    jXX     .Else
    Then_Expr
    jmp     .Done
.Else:
    Else_Expr
.Done:
```

- The assembly implementation typically adheres to this form
- Conditional and unconditional branches make sure the correct block is executed

Loops

- **C provides several looping constructs - namely, *do-while*, *while*, and *for***

- **No corresponding instructions exist in machine code**
  - Instead, combinations of conditional tests and jumps are used to implement the effect of loops

- **Most compilers generate loop code based on the *do-while* form of a loop**
  - See the document “Loops and switch statements” for details

"Do-While" loop example

- **Count number of 1's in argument x**

C code

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

Goto version

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

- **Use conditional branch to either continue or exit loop**

“Do-While” loop in Assembly

- **Assume:**

%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
```

Goto version

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

General "Do-While" translation

C code

```
do{
  statement1
  ...
  statementn
}while(Test);
```

Body

Goto version

```
loop:
   statement1
   ...
   statementn
   if(Test)
    goto loop
```

Body

- **Test returns integer**
  - = 0 interpreted as false
  - ≠ 0 interpreted as true

Today: Machine-Level Programming: Control

- Control: Condition codes
- **Accessing the condition codes**
  - Conditional jumps
  - Conditional set of a single byte
  - Conditional transfer of data

Set a single byte

- ***setX* instructions**
  - Set a **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)

Set a single byte (cont.)

- **One of 8 addressable byte registers**
  - Does not alter remaining 3 bytes
  - Typically use **`movzbl`** to finish job
    - Moves and zero extends remaining bytes

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

%esp
%ebp

C code

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

Assembly code

```
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
```

Set a single byte (cont.)

- One of 8 addressable byte registers

%eax	%ah	%al
		%cl
		%dl
		%bl

C co

```
int
{
   re
}
```

Asse

```
mov
cmp
set
movzbl %al,%eax        # Zero rest of %eax
```

Move zero byte long

`movzbl %al,%eax`

0x000000	**%al**

Several other data movement instruction, e.g.:

`movsbl` (move sign byte long) extends signal of %al

Today: Machine-Level Programming: Control

- Control: Condition codes
- **Accessing the condition codes**
  - Conditional jumps
  - Conditional set of a single byte
  - Conditional transfer of data

Conditional transfer of data

- **An alternate strategy to implement conditional operations**

- **Conditional move instructions**
  - Instruction supports ***if (Test) Dest*** **←** ***Src*** in a **single instruction**
  - 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 moves do not require transfer of control

Using conditional moves

C code

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

Conditional move

```
tval = Then_Expr;
result = Else_Expr;
t = Test;
/* single instruction */
if (t) result = tval;
return result;
```

- **This approach computes both outcomes of a conditional operation**
- **It only performs the data movement if the specified condition holds**

IA32 Conditional move instructions

- **Copy the source value *S* to its destination *R* depending on condition codes**

cmovX	Condition	Description
`cmove S,R`	`ZF`	Equal / Zero
`cmovne S,R`	`~ZF`	Not Equal / Not Zero
`cmovs S,R`	`SF`	Negative
`cmovns S,R`	`~SF`	Nonnegative
`cmovg S,R`	`~(SF^OF)&~ZF`	Greater (signed)
`cmovge S,R`	`~(SF^OF)`	Greater or Equal (signed)
`cmovl S,R`	`(SF^OF)`	Less (signed)
`cmovle S,R`	`(SF^OF)|ZF`	Less or Equal (signed)
`cmova S,R`	`~CF&~ZF`	Above (unsigned)
`cmovae S,R`	`~CF`	Above or equal (unsigned)
`cmovb S,R`	`CF`	Below (unsigned)
`cmovbe S,R`	`CF|ZF`	Below or equal (unsigned)

Conditional move example

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

```
int cmovdiff(int x, int y)
{
    int tval = y-x;
    int rval = x-y;
    int test = x<y;

    /* single instruction */
    if (test) rval = tval;
    return rval;
}
```

- **Assume:**

%edi	x
%esi	y

```
cmovdiff:
  movl    %edi,%edx
  subl    %esi,%edx      # tval = x-y
  movl    %esi,%eax
  subl    %edi,%eax      # result = y-x
  cmpl    %esi,%edi      # compare x:y
  cmovg   %edx,%eax      # if >, result = tval
  ret
```

Understanding performance impact

- **In a typical application, the outcome of the test *x* < *y* is highly unpredictable**
  - Even the most sophisticated branch prediction hardware will guess correctly only around 50% of the time
- **In the previous example, the computations performed in each of the two code sequences require only a single clock cycle**
- **As a consequence, the branch misprediction penalty dominates the performance of this function**
  - On an Intel Core i7, the time required for the conditional jump version ranges between around 13 and 57 cycles, depending on whether or not the branch is predicted correctly

Bad cases for conditional moves

Expensive computations

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

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

Risky computations

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

- Both values get computed
- May have undesirable effects (e.g., if p is NULL)

Computations with side effects

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

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

Machine-Level Programming: Control: Summary

- **C control**
  - if-then-else

- **Assembler control**
  - Conditional jumps
  - Conditional set of a single byte
  - Conditional transfer of data