

Machine-Level Programming : Controls

Computer Systems

Friday, October 4 2024

Today

Basics of control flow

Condition codes

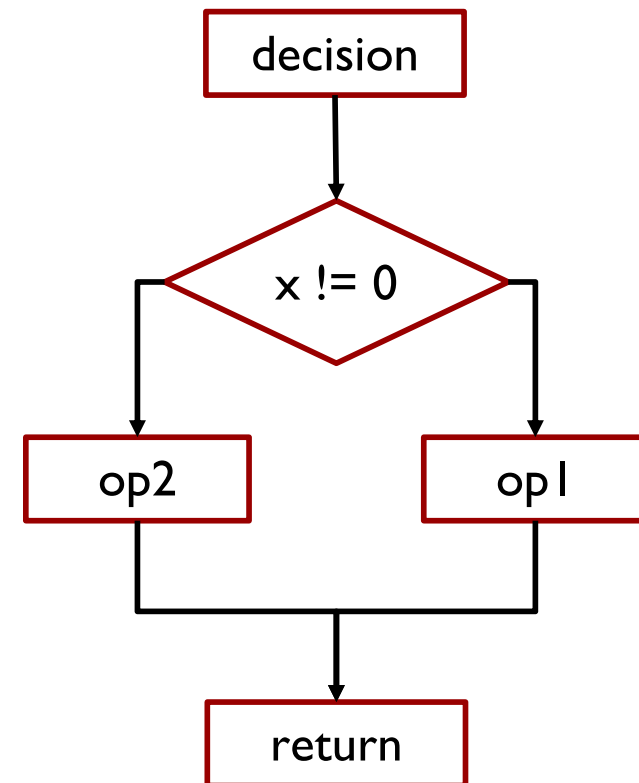
Conditional operations

Loops

If we have time: switch statements

Control flow

```
extern void op1(void) ;  
extern void op2(void) ;  
  
void decision(int x) {  
    if (x) {  
        op1() ;  
    } else {  
        op2() ;  
    }  
}
```



Control flow in assembly language

```
extern void op1(void);
extern void op2(void);

void decision(int x) {
    if (x) {
        op1();
    } else {
        op2();
    }
}
```

```
decision:
    subq    $8, %rsp
    testl   %edi, %edi
    je      .L2
    call    op1
    jmp     .L1
.L2:
    call    op2
.L1:
    addq    $8, %rsp
    ret
```



It's all done with
GOTO!

Processor State (x86-64, Partial)

Information about currently executing program

Temporary data
(**%rax**, ...)

Location of runtime stack
(**%rsp**)

Location of current code control point
(**%rip**, ...)

Status of recent tests
(**CF**, **ZF**, **SF**, **OF**)

Current stack top

Registers

%rax	%r8
%rbx	%r9
%rcx	%r10
%rdx	%r11
%rsi	%r12
%rdi	%r13
%rsp	%r14
%rbp	%r15

%rip Instruction pointer

CF	ZF	SF	OF
-----------	-----------	-----------	-----------

Condition codes

Condition Codes (Implicit Setting)

Single bit registers

CF Carry Flag (for unsigned) **SF** Sign Flag (for signed)

ZF Zero Flag **OF** Overflow Flag (for signed)

GDB prints these as one “eflags” register

```
eflags    0x246    [ PF  ZF  IF ]  Z set, CSO clear
```

Implicitly set (as side effect) of arithmetic operations

Example: **addq** *Src, Dest* \leftrightarrow **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 `leaq` instruction

ZF set when

00000000000000...000000000000

SF set when

1xxxxxxxxxxxx...xxxxxxxxxxxxx

CF set when

$$\begin{array}{r} \text{+} \begin{array}{|l|} \hline yxxxxxxxxxxxxxxxxx \dots \\ \hline yxxxxxxxxxxxxxxxxx \dots \\ \hline \end{array} \\ \hline 1 \quad \begin{array}{|l|} \hline zxxxxxxxxxxxxxxxxx \dots \\ \hline \end{array} \end{array}$$

OF set when

$$\begin{array}{r}
 \text{wxxxxxxxxxxxxxxxxx} \dots \\
 + \text{yxxxxxxxxxxxxxxxxx} \dots \\
 \hline
 \text{zxxxxxxxxxxxxxxxxx} \dots
 \end{array}$$

$w == y \ \&\& \ w \neq z$

Compare Instruction

cmp a, b

Computes $b - a$ (just like **sub**)

Sets condition codes based on result, but...

Does not change b

Used for **if (a < b) { ... }**

whenever $b - a$ isn't needed for anything else

Test Instruction

`test a, b`

Computes $b \& a$ (just like **and**)

Sets condition codes (only SF and ZF) based on result, but...

Does not change b

Most common use: `test %rX, %rX`
to compare `%rX` to zero

Second most common use: `test %rX, %rY`
tests if any of the 1-bits in `%rY` are also 1 in `%rX` (or vice versa)

Today

Basics of control flow

Condition codes

Conditional operations

Loops

Jumping

jX Instructions

Jump to different part of code depending on condition codes

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	$\sim ZF$	Not Equal / Not Zero
js	SF	Negative
jns	$\sim SF$	Nonnegative
jg	$\sim (SF \wedge OF) \ \& \ \sim ZF$	Greater (Signed)
jge	$\sim (SF \wedge OF)$	Greater or Equal (Signed)
jl	$(SF \wedge OF)$	Less (Signed)
jle	$(SF \wedge OF) \ \ ZF$	Less or Equal (Signed)
ja	$\sim CF \ \& \ \sim ZF$	Above (unsigned)
jb	CF	Below (unsigned)

Reading Condition Codes

SetX Instructions

Set low-order byte of destination to 0 or 1 based on *combinations* of condition codes

Does not alter remaining 7 bytes

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)

x86-64 Integer Registers

%rax	%al
%rbx	%bl
%rcx	%cl
%rdx	%dl
%rsi	%sil
%rdi	%dil
%rsp	%spl
%rbp	%bpl

%r8	%r8b
%r9	%r9b
%r10	%r10b
%r11	%r11b
%r12	%r12b
%r13	%r13b
%r14	%r14b
%r15	%r15b

SetX argument is always a low byte (%al, %r8b, etc.)

Reading Condition Codes (Cont.)

SetX Instructions:

Set single byte based on combination of condition codes

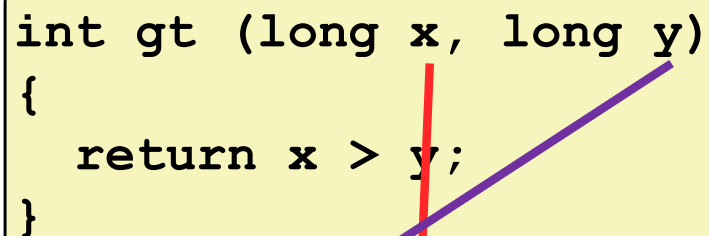
One of addressable byte registers

Does not alter remaining bytes

Typically use `movzbl` to finish job

32-bit instructions also set upper 32 bits to 0

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



```
cmpq    %rsi, %rdi    # Compare x:y
setg     %al          # Set when >
movzbl  %al, %eax     # Zero rest of %rax
ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

Conditional Branch Example (Old Style)

Generation

```
linux> gcc -Og -S -fno-if-conversion control.c
```

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

```
absdiff:
    cmpq    %rsi, %rdi    # x:y
    jle     .L4
    movq    %rdi, %rax
    subq    %rsi, %rax
    ret
.L4:      # x <= y
    movq    %rsi, %rax
    subq    %rdi, %rax
    ret
```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

Expressing with Goto Code

C allows goto statement

Jump to position designated by label

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

```
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;
}
```

General Conditional Expression Translation (Using Branches)

C Code

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

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

Goto Version

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

Create separate code regions for
then & else expressions

Execute appropriate one

Using Conditional Moves

Conditional Move Instructions

Instruction supports:

if (Test) Dest \leftarrow Src

Supported in post-1995 x86 processors

GCC tries to use them

But, only when known to be safe

Why?

Branches are very disruptive to instruction flow through pipelines

Conditional moves do not require control transfer

C Code

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

Goto Version

```
result = Then_Expr;  
eval = Else_Expr;  
nt = !Test;  
if (nt) result = eval;  
return result;
```

Conditional Move Example

```

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

```

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

absdiff:

```

movq    %rdi, %rax    # x
subq    %rsi, %rax    # result = x-y
movq    %rsi, %rdx
subq    %rdi, %rdx    # eval = y-x
cmpq    %rsi, %rdi    # x:y
cmovle  %rdx, %rax    # if <=, result = eval
ret

```

Bad Cases for Conditional Move

Expensive Computations

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

Both values get computed

Only makes sense when computations
are very simple

Bad Performance

Risky Computations

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

Both values get computed

May have undesirable effects

Unsafe

Computations with side effects

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

Both values get computed

Must be side-effect free

Illegal

Today

Basics of control flow

Condition codes

Conditional operations

Loops

“Do-While” Loop Example

C Code

```
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;
}
```

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

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

Register	Use(s)
%rdi	Argument x
%rax	result

```

movl    $0, %eax           # result = 0
.L2:
                                # loop:
    movq    %rdi, %rdx
    andl    $1, %edx        # t = x & 0x1
    addq    %rdx, %rax      # result += t
    shrq    %rdi            # x >>= 1
    jne     .L2             # if (x) goto
loop
    rep; ret
```

General “Do-While” Translation

C Code

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

```
Body: {  
    Statement1;  
    Statement2;  
    ...  
    Statementn;  
}
```

Goto Version

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

General “While” Translation #1

“Jump-to-middle” translation

Used with -Og

While version

```
while (Test)  
    Body
```



Goto Version

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

While Loop Example #1

C Code

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

Jump to Middle

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

Compare to do-while version of function

Initial goto starts loop at test

General “While” Translation #2

While version

```
while (Test)  
    Body
```



Do-While Version

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

“Do-while” conversion
Used with -O1

Goto Version

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



While Loop Example #2

C Code

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

Do-While Version

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

Compare to do-while version of function

Initial conditional guards entrance to loop

“For” Loop Form

General Form

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

```
#define WSIZE 8*sizeof(int)  
long pcount_for  
    (unsigned long x)  
{  
    size_t i;  
    long result = 0;  
    for (i = 0; i < WSIZE; i++)  
    {  
        unsigned bit =  
            (x >> i) & 0x1;  
        result += bit;  
    }  
    return result;  
}
```

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{  
    unsigned bit =  
        (x >> i) & 0x1;  
    result += bit;  
}
```


“For” Loop → While Loop

For Version

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



While Version

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

For-While Conversion

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{  
    unsigned bit =  
        (x >> i) & 0x1;  
    result += bit;  
}
```

```
long pcount_for_while  
(unsigned long x)  
{  
    size_t i;  
    long result = 0;  
    i = 0;  
    while (i < WSIZE)  
    {  
        unsigned bit =  
            (x >> i) & 0x1;  
        result += bit;  
        i++;  
    }  
    return result;  
}
```

“For” Loop Do-While Conversion

C Code

```
long pcount_for
(unsigned long x)
{
    size_t i;
    long result = 0;
    for (i = 0; i < WSIZE; i++)
    {
        unsigned bit =
            (x >> i) & 0x1;
        result += bit;
    }
    return result;
}
```

Goto Version

```
long pcount_for_goto_dw
(unsigned long x) {
    size_t i;
    long result = 0;
    i = 0;
    if (!(i < WSIZE)) Ini
    goto done; !Test
loop:
{
    unsigned bit =
        (x >> i) & 0x1; Body
    result += bit;
}
i++; Update
if (i < WSIZE) Test
    goto loop;
done:
    return result;
}
```

Initial test can be optimized away

Summary: Condition Codes

Single bit registers

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

jX and SetX instructions

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	$\sim ZF$	Not Equal / Not Zero
js	SF	Negative
jns	$\sim SF$	Nonnegative
jg	$\sim (SF \wedge OF) \ \& \ \sim ZF$	Greater (Signed)
jge	$\sim (SF \wedge OF)$	Greater or Equal (Signed)
jl	$(SF \wedge OF)$	Less (Signed)
jle	$(SF \wedge OF) \mid ZF$	Less or Equal (Signed)
ja	$\sim CF \ \& \ \sim ZF$	Above (unsigned)
jb	CF	Below (unsigned)

SetX	Condition	Description
sete	ZF	Equal / Zero
setne	$\sim ZF$	Not Equal / Not Zero
sets	SF	Negative
setns	$\sim SF$	Nonnegative
setg	$\sim (SF \wedge OF) \ \& \ \sim ZF$	Greater (Signed)
setge	$\sim (SF \wedge OF)$	Greater or Equal (Signed)
setl	$(SF \wedge OF)$	Less (Signed)
setle	$(SF \wedge OF) \mid ZF$	Less or Equal (Signed)
seta	$\sim CF \ \& \ \sim ZF$	Above (unsigned)
setb	CF	Below (unsigned)

Machine Level Programming – Control

C Control

if-then-else

do-while

while, for

switch

Assembler Control

Conditional jump

Conditional move

Indirect jump (via jump tables)

Compiler generates code sequence to implement more complex control

Standard Techniques

Loops converted to do-while or jump-to-middle form

Large switch statements use jump tables

Sparse switch statements may use decision trees (if-elseif-elseif-else)

Homework #4

Patch a binary

➤ Homework #04

- Overview

- **Released date:** 10/4 (Fri.)
- **Due date:** 10/11 (Fri.)
- **Where to submit:** to e-class (<http://eclass.seoultech.ac.kr>)
 - Late submission is not allowed.
- **Assigned score:** 1 points

Decompile `hello` binary and change the string value to print your STUDENT ID

- Submissions

- Explain how to change the string in the bss section.
- Captured images to show the result