Controlling Program Flow
Control Flow

Computers execute instructions in sequence.

Except when we change the flow of control

■ Jump and Call instructions
■ Unconditional jump
  ● Direct jump: jmp Label
    ▶ Jump target is specified by a label (e.g., jmp .L1)
  ● Indirect jump: jmp *Operand
    ▶ Jump target is specified by a register or memory location
      (e.g., jmp *%rax)
Conditional statements

Some jumps are *conditional*

- A computer needs to jump if certain a condition is true
- In C, *if, for, and while* statements

  ```c
  if (x) {...} else {...}
  
  while (x) {...}
  
  do {...} while (x)
  
  for (i=0; i<max; i++) {...}
  
  switch (x) {
    case 1: ...
    case 2: ...
  }
  ```
Condition codes

Processor flag register *eflags* (extended flags)

Flags are set or cleared by depending on the result of an instruction

Each bit is a flag, or condition code

**CF** Carry Flag  **SF** Sign Flag

**ZF** Zero Flag  **OF** Overflow Flag

<table>
<thead>
<tr>
<th>Registers</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%r8</td>
</tr>
<tr>
<td>%rbx</td>
<td>%r9</td>
</tr>
<tr>
<td>%rcx</td>
<td>%r10</td>
</tr>
<tr>
<td>%rdx</td>
<td>%r11</td>
</tr>
<tr>
<td>%rsi</td>
<td>%r12</td>
</tr>
<tr>
<td>%rdi</td>
<td>%r13</td>
</tr>
<tr>
<td>%rsp</td>
<td>%r14</td>
</tr>
<tr>
<td>%rbp</td>
<td>%r15</td>
</tr>
<tr>
<td>%rip</td>
<td></td>
</tr>
</tbody>
</table>

Condition codes
Implicit setting

Automatically Set By Arithmetic and Logical Operations

Example: `addq Src, Dest`

C analog: `t = a + b`

- **CF** (for unsigned integers)
  - set if carry out from most significant bit (unsigned overflow)
    - `(unsigned long t) < (unsigned long a)`

- **ZF** (zero flag)
  - set if `t == 0`

- **SF** (for signed integers)
  - set if `t < 0`

- **OF** (for signed integers)
  - set if signed (two’s complement) overflow
    - `(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)`

*Not set by lea, push, pop, mov instructions*
Explicit setting via compare

Setting condition codes via compare instruction

\( \text{cmpq } b, a \)

- Computes \( a - b \) without setting destination
- CF set if carry out from most significant bit
  - Used for unsigned comparisons
- ZF set if \( a == b \)
- SF set if \( (a-b) < 0 \)
- OF set if two’s complement (signed) overflow
  \( (a>0 \land b<0 \land (a-b)<0) \lor (a<0 \land b>0 \land (a-b)>0) \)
- Byte, word, and double word versions \( \text{cmpb, cmpw, cmpl} \)
Explicit setting via test

Setting condition codes via test instruction

`testq b, a`

- Computes `a & b` without setting destination
  - Sets condition codes based on result
  - Useful to have one of the operands be a mask

- Often used to test zero, positive
  - `testq %rax, %rax`

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

- Byte, word and double word versions `testb`, `testw`, `testl`
# Conditional jump instructions

Jump to different part of code based on condition codes

<table>
<thead>
<tr>
<th>jX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>1</td>
<td>Unconditional</td>
</tr>
<tr>
<td>je, jz</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jne, jnz</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>js</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>jns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg</td>
<td>~(SF^OF) &amp;~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jge</td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>jl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jle</td>
<td>(SF^OF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>ja</td>
<td>~CF &amp; ~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>jb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>

Overflow flips result
Jump instructions

What’s the difference between jg and ja?

Which one would you use to compare two pointers?
Conditional jump example

Non-optimized

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;
}

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
General Conditional Expression Translation (Using Branches)

C Code

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

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

Goto Version

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

Create separate code regions for then & else expressions

Execute appropriate one
Practice problem 3.18

/* x in %rdi, y in %rsi, z in %rdx */
test:
    leaq (%rdi,%rsi), %rax
    addq %rdx, %rax
    cmpq $-3, %rdi
    jge .L2
    cmp %rdx,%rsi
    jge .L3
    movq %rdi, %rax
    imulq %rsi, %rax
    ret
.L3:
    movq %rsi, %rax
    imulq %rdx,%rax
    ret
.L2
    cmpq $2, %rdi
    jle .L4
    movq %rdi, %rax
    imulq %rdx, %rax
.L4
    ret

long test(long x, long y, long z)
{
    long val = ____________ ;
    if ( ____________ ) {
        if ( ____________ )
            val = ____________ ;
        else
            val = ____________ ;
    } else if ( ____________ )
        val = ____________ ;
    return val;
}
Avoiding conditional branches

Modern CPUs with deep pipelines

- Instructions fetched far in advance of execution
- Mask the latency going to memory
- Problem: What if you hit a conditional branch?
  - Must predict which branch to take!
  - Branch prediction in CPUs well-studied, fairly effective
  - But, best to avoid conditional branching altogether
Conditional moves

Conditional instruction execution

\texttt{cmovXX \ Src, Dest}

- Move value from src to dest if condition \textit{XX} holds
  - No branching
  - Handled as operation within Execution Unit
  - Added with P6 microarchitecture (PentiumPro onward, 1995)

Example

```
# %rdi = x, %rsi = y
# return value in %rax returns max(x,y)
movq %rdi,%rdx       # Get x
movq %rsi,%rax       # rval=y
cmpq %rdx, %rax      # rval:x
cmovl %rdx,%rax      # If <, rval=x
```

Performance

- 14 cycles on all data
- More efficient than conditional branching (single control flow path)
- But overhead: both branches are evaluated
Conditional Move template

- Instruction supports
  - if (Test) Dest ← Src
- GCC attempts to restructure execution to avoid disruptive conditional branch
  - Both values computed
  - Overwrite “then”-value with “else”-value if condition doesn’t hold
- Conditional moves do not transfer control

C Code

```c
val = Test ? Then_Expr : Else_Expr;
result = Then_Expr;
eval = Else_Expr;
nt = !Test;
if (nt) result = eval;
return result;
```

Branch version

```c
ntest = !Test;
if (ntest) goto Else;
val = Then_Expr;
goto Done;
Else:
val = Else_Expr;
Done:
```
Conditional Move example

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

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

Branch version

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
Practice problem 3.21

/* x in %rdi, y in %rsi */
thest:
    leaq  0(,%rdi,8), %rax
    testq %rsi, %rsi
    jle  .L2
    movq %rsi, %rax
    subq %rdi, %rax
    movq %rdi, %rdx
    andq %rsi, %rdx
    cmpq %rsi, %rdi
    cmovge %rdx, %rax
    ret
.L2:
    addq %rsi, %rdi
    cmpq $-2, %rsi
    cmovle %rdi, %rax
    ret

long test(long x, long y)
{
    long val = 8*x ;
    if ( _______ y > 0 _______ ) {
        if ( _______ x < y _______ )
            val = _______ y-x _______ ;
        else
            val = _______ x&y _______ ;
    } else if ( _______ y <= -2 _______ )
        val = _______ x+y _______ ;
    return val;
}
When not to use Conditional Move

Expensive computations

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

- Both Hard1(x) and Hard2(x) computed
- Use branching when “then” and “else” expressions are more expensive than branch misprediction

Computations with side effects

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

- Executing both values causes incorrect behavior

Condition must hold to prevent fault

- Null pointer check
Loops

Implemented in assembly via tests and jumps

- Compilers implement most loops as `do-while`
  - Add additional check at beginning to get “while-do”

```
  do {
      body-statements
  } while (test-expr);
```
long factorial_do(long x)
{
    long result = 1;
    do {
        result *= x;
        x = x - 1;
    } while (x > 1);
    return result;
}

factorial_do:
    movq $1, %rax ; result = 1
.L2:
    imulq %rdi, %rax ; result *= x
    subq $1, %rdi ; x = x - 1
    cmpq $1, %rdi ; if x > 1
    jg .L2 ; goto loop
    ret ; return result
Are these equivalent?

C code of do-while

```c
long factorial_do(long x) {
    long result = 1;
    do {
        result *= x;
        x = x - 1;
    } while (x > 1);
    return result;
}
```

C code of while-do

```c
long factorial_while(long x) {
    long result = 1;
    while (x > 1) {
        result *= x;
        x = x - 1;
    }
    return result;
}
```
Assembly of do-while

```
factorial_do:
  movq $1, %rax
.L2:
  imulq %rdi, %rax
  subq $1, %rdi
  cmpq $1, %rdi
  jg .L2
  ret
```

Assembly of while-do

```
factorial_while:
  movq $1, %rax
  jmp .L2
.L3:
  imulq %rdi, %rax
  subq $1, %rdi
  cmpq $1, %rdi
  jg .L3
  rep ret
```
“For” Loop Example

```c
long factorial_for(long x) {
    long result;
    for (result=1; x > 1; x=x-1) {
        result *= x;
    }
    return result;
}
```

General Form

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

**Init**
- `result = 1`

**Test**
- `x > 1`

**Update**
- `x = x - 1`

**Body**
- `{ result *= x; }
```

Is this code equivalent to the do-while version or the while-do version?
“For” Loop Example

factorial_while:

    movq    $1, %rax
    jmp     .L2
.L3:
    imulq   %rdi, %rax
    subq    $1, %rdi
.L2:
    cmpq    $1, %rdi
    jg      .L3
    ret

factorial_for:

    movq    $1, %rax
    jmp     .L2
.L3:
    imulq   %rdi, %rax
    subq    $1, %rdi
.L2:
    cmpq    $1, %rdi
    jg      .L3
    ret

http://thefengs.com/wuchang/courses/cs201/class/07
diff factorial_for.s factorial_while.s
Problem 3.26

fun_a:
    movq  $0, %rax
    jmp  .L5
.L6:
xorq  %rdi, %rax
shrq  %rdi
.L5:
testq %rdi, %rdi
jne  .L6
andq  $1, %rax
ret

long fun_a(unsigned long x) {
    long val = 0;
    while ( _____ ) {
        val = val ^ x ;
        x = x >> 1 ;
    }
    return  val & 0x1  ;
}
C switch Statements

Test whether an expression matches one of a number of constant integer values and branches accordingly

Without a “break” the code falls through to the next case

If x matches no case, then “default” is executed

```c
long switch_eg(long x)
{
    long result = x;
    switch (x) {
        case 100:
            result *= 13;
            break;
        case 102:
            result += 10;
            /* Fall through */
        case 103:
            result += 11;
            break;
        case 104:
        case 106:
            result *= result;
            break;
        default:
            result = 0;
    }
    return result;
}
```
C switch statements

Implementation options

- Series of conditionals
  - `testq/cmpq` followed by `je`
  - Good if few cases
  - Slow if many cases

- Jump table (example below)
  - Lookup branch target from a table
  - Possible with a small range of integer constants

GCC picks implementation based on structure

Example:

```c
switch (x) {  
  case 1:    
  case 5:  
    code at L0
  case 2:    
  case 3:  
    code at L1
  default:  
    code at L2
}
```

1. init jump table at .L3
2. get address at .L3+8*x
3. jump to that address
Example revisited

```c
long switch_eg(long x) {
    long result = x;
    switch (x) {
        case 100:
            result *= 13;
            break;

        case 102:
            result += 10;
            /* Fall through */

        case 103:
            result += 11;
            break;

        case 104:
        case 106:
            result *= result;
            break;

        default:
            result = 0;
    }
    return result;
}
```
long switch_eg(long x)
{
    long result = x;
    switch (x) {
        case 100:
            result *= 13;
            break;
        case 102:
            result += 10;
            /* Fall through */
        case 103:
            result += 11;
            break;
        case 104:
        case 106:
            result *= result;
            break;
        default:
            result = 0;
    }
    return result;
}

Key is jump table at L4
Array of pointers to jump locations

http://thefengs.com/wuchang/courses/cs201/class/07/switch_code.c
Practice problem 3.30

The switch statement body has been omitted in the C program. GCC generates the code shown when compiled.

- What were the values of the case labels in the switch statement?
- What cases had multiple labels in the C code?

```c
void switch2(long x, long *dest) {
    long val = 0;
    switch (x) {
        
    }
    *dest = val
}

/* x in %rdi */
switch2:
    addq $1, %rdi
    cmpq $8, %rdi
    ja .L2
    jmp *.*.L4(%rdi,8)
.L4
    .quad .L9
    .quad .L5
    .quad .L6
    .quad .L7
    .quad .L2
    .quad .L7
    .quad .L8
    .quad .L2
    .quad .L5
```
void switch2(long x, long *dest) {
    long val = 0;
    switch (x) {
    
    } 
    *dest = val 
}

/* x in %rdi */
switch2:
    addq $1, %rdi
    cmpq $8, %rdi
    ja .L2
    jmp *.L4(%rdi,8)
    .L4
    .quad .L9
    .quad .L5
    .quad .L6
    .quad .L7
    .quad .L2
    .quad .L7
    .quad .L8
    .quad .L2
    .quad .L5
Homework A3
Extra slides
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

<table>
<thead>
<tr>
<th>SetX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sete</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>setne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>sets</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>setns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>setg</td>
<td>~(SF^OF) &amp; ~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>setge</td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>setl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>setle</td>
<td>(SF^OF)</td>
<td>ZF</td>
</tr>
<tr>
<td>seta</td>
<td>~CF &amp; ~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>setb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
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

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

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument (x)</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument (y)</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

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

http://thefengs.com/wuchang/courses/cs201/class/07/setg_code.c
x86 REP prefixes

Loops require decrement, comparison, and conditional branch for each iteration

Incur branch prediction penalty and overhead even for trivial loops

REP, REPE, REPNE

- Instruction prefixes can be inserted just before some instructions (movsb, movsw, movsd, cmpsb, cmpsw, cmpsd)
- REP (repeat for fixed count)
  - Direction flag (DF) set via cld and std instructions
  - esi and edi contain pointers to arguments
  - ecx contains counts
- REPE (repeat until zero), REPNE (repeat until not zero)
  - Used in conjunction with cmpsb, cmpsw, cmpsd
x86 REP example

.data
  source DWORD 20 DUP ()
  target DWORD 20 DUP ()

.code
  cld    ; clear direction flag = forward
  mov ecx, LENGTHOF source
  mov esi, OFFSET source
  mov edi, OFFSET target
  rep movsd
x86 SCAS

Searching

■ Repeat a search until a condition is met
■ SCASB  SCASW  SCASD
  • Search for a specific element in an array
  • Search for the first element that does not match a given value
x86 SCAS

.data
alpha BYTE "ABCDEFGHIJKLMNOPQRSTUVWXYZ",0

.code
    mov edi,OFFSET alpha
    mov al,'F' ; search for 'F'
    mov ecx,LENGTHOF alpha
    cld
    repne scasb ; repeat while not equal
    jnz quit
    dec edi ; EDI points to 'F'
x86 LODS/STOS

Storing and loading

- Initialize array of memory or sequentially read array from memory
- Can be combined with other operations in a loop
- **LODSB LODSW LODSD**
  - Load values from array sequentially
- **STOSB STOSW STOSD**
  - Store a specific value into all entries of an array
x86 LODS/STOS

.data
   array DWORD 1,2,3,4,5,6,7,8,9,10
   multiplier DWORD 10

.code
  cld ; direction = up
  mov esi,OFFSET array ; source index
  mov edi,esi ; destination index
  mov ecx,LENGTHOF array ; loop counter

L1: lodsd ; copy [ESI] into EAX
  mul multiplier ; multiply by a value
  stosd ; store EAX at [EDI]
  loop L1h