Assembly Programming III
CSE 351 Spring 2017

Guest Lecturer:
Justin Hsia

Instructor:
Ruth Anderson

Teaching Assistants:
Dylan Johnson
Kevin Bi
Linxing Preston Jiang
Cody Ohlsen
Yufang Sun
Joshua Curtis
Administrivia

- Lab 1 due TONIGHT, Friday (4/14)
  - Remember, you have *late days* available if needed.
- Homework 2 due next Wednesday (4/19)
- Lab 2 (x86-64) released soon!
  - Learn to read x86-64 assembly and use GDB
x86 Control Flow

- Condition codes
- Conditional and unconditional branches
- Loops
- Switches
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)
    - Single bit registers:
      - %rip
      - %rax
      - %rbx
      - %rcx
      - %rdx
      - %rsi
      - %rdi
      - %rsp
      - %rbp
      - %r8
      - %r9
      - %r10
      - %r11
      - %r12
      - %r13
      - %r14
      - %r15

- Program Counter (instruction pointer)
  - Condition Codes
    - CF
    - ZF
    - SF
    - OF
Condition Codes (Implicit Setting)

- **Implicitly** set by arithmetic operations
  - (think of it as side effects)
  - **Example**: `addq src, dst ↔ r = d+s`

- **CF=1** if carry out from MSB (unsigned overflow)
- **ZF=1** if \( r = 0 \)
- **SF=1** if \( r < 0 \) (assuming signed, actually just if MSB is 1)
- **OF=1** if two’s complement (signed) overflow
  \[ (s > 0 \land d > 0 \land r < 0) \lor (s < 0 \land d < 0 \land r \geq 0) \]
- **Not set by lea instruction** (beware!)
Condition Codes (Explicit Setting: Compare)

- **Explicitly set by Compare instruction**
  - `cmpq src1, src2`
  - `cmpq a, b` sets flags based on \( b-a \), but doesn’t store

- `CF=1` if carry out from MSB (used for unsigned comparison)
- `ZF=1` if \( a==b \)
- `SF=1` if \((b-a)<0\) (signed)
- `OF=1` if two’s complement (signed) overflow

\[(b>0 \&\& a<0 \&\& (b-a)<0) \text{ or } (b<0 \&\& a>0 \&\& (b-a)>0)\]
Condition Codes (**Explicit Setting: Test**)

- **Explicitly** set by **Test** instruction
  - `testq src2, src1`
  - `testq a, b` sets flags based on `b&a`, but doesn’t store
    - Useful to have one of the operands be a *mask*
  - Can’t have carry out (**CF**) or overflow (**OF**)
  - **ZF=1** if `a&b==0`
  - **SF=1** if `a&b<0` (signed)

- **Example**: `testq %rax, %rax`
  - Tells you if (+), 0, or (−) based on **ZF** and **SF**
Using Condition Codes: Jumping

- \( j * \) Instructions
  - Jumps to `target` (an address) based on condition codes

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp <code>target</code></td>
<td>1</td>
<td>Unconditional</td>
</tr>
<tr>
<td>je <code>target</code></td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jne <code>target</code></td>
<td>(~ZF)</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>js <code>target</code></td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>jns <code>target</code></td>
<td>(~SF)</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg <code>target</code></td>
<td>((SF^OF) &amp; ~ZF)</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jge <code>target</code></td>
<td>((SF^OF))</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>jl <code>target</code></td>
<td>((SF^OF))</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jle <code>target</code></td>
<td>((SF^OF) | ZF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>ja <code>target</code></td>
<td>(~CF&amp;~ZF)</td>
<td>Above (unsigned “&gt;“)</td>
</tr>
<tr>
<td>jb <code>target</code></td>
<td>CF</td>
<td>Below (unsigned “&lt;“)</td>
</tr>
</tbody>
</table>
Using Condition Codes: Setting

*set* Instructions

- Set low-order byte of $dst$ to 0 or 1 based on condition codes
- Does not alter remaining 7 bytes

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sete $dst$</td>
<td>$ZF$</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>setne $dst$</td>
<td>$\neg ZF$</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>sets $dst$</td>
<td>$SF$</td>
<td>Negative</td>
</tr>
<tr>
<td>setns $dst$</td>
<td>$\neg SF$</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>setg $dst$</td>
<td>$\neg (SF^\wedge OF) &amp; \neg ZF$</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>setge $dst$</td>
<td>$\neg (SF^\wedge OF)$</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>setl $dst$</td>
<td>$SF^\wedge OF$</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>settle $dst$</td>
<td>$(SF^\wedge OF) \mid ZF$</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>seta $dst$</td>
<td>$\neg CF &amp; \neg ZF$</td>
<td>Above (unsigned “&gt;”)</td>
</tr>
<tr>
<td>setb $dst$</td>
<td>$CF$</td>
<td>Below (unsigned “&lt;”)</td>
</tr>
</tbody>
</table>
Reminder: x86-64 Integer Registers

- Accessing the low-order byte:

<table>
<thead>
<tr>
<th>Register</th>
<th>Low-order Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%al</td>
</tr>
<tr>
<td>%rbx</td>
<td>%bl</td>
</tr>
<tr>
<td>%rcx</td>
<td>%cl</td>
</tr>
<tr>
<td>%rdx</td>
<td>%dl</td>
</tr>
<tr>
<td>%rsi</td>
<td>%sil</td>
</tr>
<tr>
<td>%rdi</td>
<td>%sil</td>
</tr>
<tr>
<td>%rsp</td>
<td>%spl</td>
</tr>
<tr>
<td>%rbp</td>
<td>%bpl</td>
</tr>
<tr>
<td>%r8</td>
<td>%r8b</td>
</tr>
<tr>
<td>%r9</td>
<td>%r9b</td>
</tr>
<tr>
<td>%r10</td>
<td>%r10b</td>
</tr>
<tr>
<td>%r11</td>
<td>%r11b</td>
</tr>
<tr>
<td>%r12</td>
<td>%r12b</td>
</tr>
<tr>
<td>%r13</td>
<td>%r13b</td>
</tr>
<tr>
<td>%r14</td>
<td>%r14b</td>
</tr>
<tr>
<td>%r15</td>
<td>%r15b</td>
</tr>
</tbody>
</table>
Reading Condition Codes

- **set**\* Instructions
  - Set a low-order byte to 0 or 1 based on condition codes
  - Operand is byte register (e.g. al, dl) or a byte in memory
  - Do not alter remaining bytes in register
    - Typically use `movzbl` (zero-extended `mov`) to finish job

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

```plaintext
cmpq %rsi, %rdi      #
setg %al            #
movzbl %al, %eax    #
ret
```
Reading Condition Codes

- **set* Instructions**
  - Set a low-order byte to 0 or 1 based on condition codes
  - Operand is byte register (e.g. `al`, `dl`) or a byte in memory
  - Do not alter remaining bytes in register
    - Typically use `movzbl` (zero-extended `mov`) to finish job

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

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

\texttt{movz} \_\_ \texttt{src, regDest} \textit{Move with zero extension}
\texttt{movs} \_\_ \texttt{src, regDest} \textit{Move with sign extension}

- Copy from a \textit{smaller} source value to a \textit{larger} destination
- Source can be memory or register; Destination \textit{must} be a register
- Fill remaining bits of dest with \texttt{zero} (\texttt{movz}) or \texttt{sign bit} (\texttt{movs})

\texttt{movzSD} / \texttt{movsSD}:

\textit{S} – size of source ($b = 1$ byte, $w = 2$)
\textit{D} – size of dest ($w = 2$ bytes, $l = 4$, $q = 8$)

Example:
\texttt{movzbq} %al, %rbx

\begin{verbatim}
0x?? 0x?? 0x?? 0x?? 0x?? 0x?? 0x?? 0xFF \rightarrow %rax
0x00 0x00 0x00 0x00 0x00 0x00 0x00 0xFF \rightarrow %rbx
\end{verbatim}
Aside: movz and movs

movzsrc, regDest   Move with zero extension
movssrc, regDest   Move with sign extension

- Copy from a smaller source value to a larger destination
- Source can be memory or register; Destination must be a register
- Fill remaining bits of dest with zero (movz) or sign bit (movs)

movzSD / movsSD:
S – size of source (b = 1 byte, w = 2)
D – size of dest (w = 2 bytes, l = 4, q = 8)

Example:
movsbl (%rax), %ebx

Copy 1 byte from memory into 8-byte register & sign extend it

Note: In x86-64, any instruction that generates a 32-bit (long word) value for a register also sets the high-order portion of the register to 0. Good example on p. 184 in the textbook.
Choosing instructions for conditionals

- All arithmetic instructions set condition flags based on result of operation (op)
  - Conditionals are comparisons against 0

- Come in instruction *pairs*

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>JE</td>
<td>Equal</td>
</tr>
<tr>
<td>JNE</td>
<td>Not equal</td>
</tr>
<tr>
<td>JS</td>
<td>Sign (negative)</td>
</tr>
<tr>
<td>JNS</td>
<td>Non-negative</td>
</tr>
<tr>
<td>JG</td>
<td>Greater</td>
</tr>
<tr>
<td>JGE</td>
<td>Greater or equal</td>
</tr>
<tr>
<td>JL</td>
<td>Less</td>
</tr>
<tr>
<td>JLE</td>
<td>Less or equal</td>
</tr>
<tr>
<td>JA</td>
<td>Above (unsigned &gt;)</td>
</tr>
<tr>
<td>JB</td>
<td>Below (unsigned &lt;)</td>
</tr>
</tbody>
</table>

Example:

```assembly
addq 5, (p)
je:   *p+5 == 0
jne:  *p+5 != 0
jg:   *p+5 > 0
jl:   *p+5 < 0
```

```assembly
orq a, b
je:   b|a == 0
jne:  b|a != 0
jg:   b|a > 0
jl:   b|a < 0
```
Choosing instructions for conditionals

- Reminder: `cmp` is like `sub`, `test` is like `and`
  - Result is not stored anywhere

<table>
<thead>
<tr>
<th>Instruction</th>
<th>cmp a, b</th>
<th>test a, b</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>je</code> &quot;Equal&quot;</td>
<td><code>b == a</code></td>
<td><code>b&amp;a == 0</code></td>
</tr>
<tr>
<td>`jne&quot; Not equal&quot;</td>
<td><code>b != a</code></td>
<td><code>b&amp;a != 0</code></td>
</tr>
<tr>
<td>`js&quot; Sign&quot; (negative)</td>
<td><code>b-a &lt; 0</code></td>
<td><code>b&amp;a &lt; 0</code></td>
</tr>
<tr>
<td>`jns&quot; (non-negative)</td>
<td><code>b-a &gt;= 0</code></td>
<td><code>b&amp;a &gt;= 0</code></td>
</tr>
<tr>
<td>`jg&quot; Greater&quot;</td>
<td><code>b &gt; a</code></td>
<td><code>b&amp;a &gt; 0</code></td>
</tr>
<tr>
<td>`jge&quot; Greater or equal&quot;</td>
<td><code>b &gt;= a</code></td>
<td><code>b&amp;a &gt;= 0</code></td>
</tr>
<tr>
<td>`jl&quot; Less&quot;</td>
<td><code>b &lt; a</code></td>
<td><code>b&amp;a &lt; 0</code></td>
</tr>
<tr>
<td>`jle&quot; Less or equal&quot;</td>
<td><code>b &lt;= a</code></td>
<td><code>b&amp;a &lt;= 0</code></td>
</tr>
<tr>
<td>`ja&quot; Above&quot; (unsigned &gt;)</td>
<td><code>b &gt; a</code></td>
<td><code>b&amp;a &gt; 0U</code></td>
</tr>
<tr>
<td>`jb&quot; Below&quot; (unsigned &lt;)</td>
<td><code>b &lt; a</code></td>
<td><code>b&amp;a &lt; 0U</code></td>
</tr>
</tbody>
</table>

```asm
cmpq 5, (p)
    je: *p == 5
    jne: *p != 5
    jg:  *p > 5
    jl:  *p < 5

testq a, a
    je:  a == 0
    jne: a != 0
    jg:  a > 0
    jl:  a < 0

testb a, 0x1
    je:  a_{LSB} == 0
    jne: a_{LSB} == 1
```
Choosing instructions for conditionals

<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>

<table>
<thead>
<tr>
<th></th>
<th>cmp a,b</th>
<th>test a,b</th>
</tr>
</thead>
<tbody>
<tr>
<td>je</td>
<td>“Equal”</td>
<td>b == a</td>
</tr>
<tr>
<td>jne</td>
<td>“Not equal”</td>
<td>b != a</td>
</tr>
<tr>
<td>js</td>
<td>“Sign” (negative)</td>
<td>b-a &lt; 0</td>
</tr>
<tr>
<td>jns</td>
<td>(non-negative)</td>
<td>b-a &gt;=0</td>
</tr>
<tr>
<td>jg</td>
<td>“Greater”</td>
<td>b &gt; a</td>
</tr>
<tr>
<td>jge</td>
<td>“Greater or equal”</td>
<td>b &gt;= a</td>
</tr>
<tr>
<td>jl</td>
<td>“Less”</td>
<td>b &lt; a</td>
</tr>
<tr>
<td>jle</td>
<td>“Less or equal”</td>
<td>b &lt;= a</td>
</tr>
<tr>
<td>ja</td>
<td>“Above” (unsigned &gt;)</td>
<td>b &gt; a</td>
</tr>
<tr>
<td>jb</td>
<td>“Below” (unsigned &lt;)</td>
<td>b &lt; a</td>
</tr>
</tbody>
</table>

if (x < 3) {
    return 1;
}
return 2;

T1: # x < 3:
    movq $1, %rax
    ret
T2: # !(x < 3):
    movq $2, %rax
    ret
Question

A. `cmpq %rsi, %rdi
    jle .L4`
B. `cmpq %rsi, %rdi
    jg  .L4`
C. `testq %rsi, %rdi
    jle .L4`
D. `testq %rsi, %rdi
    jg  .L4`
E. We’re lost...

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

**Register Use(s)**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>1st argument (x)</td>
</tr>
<tr>
<td>%rsi</td>
<td>2nd argument (y)</td>
</tr>
<tr>
<td>%rax</td>
<td>return value</td>
</tr>
</tbody>
</table>

```
absdiff:__________________________  # x > y:
          __________________________
            movq %rdi, %rax
            subq %rsi, %rax
            ret
  .L4: # x <= y:
            movq %rsi, %rax
            subq %rdi, %rax
            ret
```
Choosing instructions for conditionals

<table>
<thead>
<tr>
<th></th>
<th>cmp a,b</th>
<th>test a,b</th>
</tr>
</thead>
<tbody>
<tr>
<td>je</td>
<td>“Equal”</td>
<td>b == a</td>
</tr>
<tr>
<td>jne</td>
<td>“Not equal”</td>
<td>b != a</td>
</tr>
<tr>
<td>js</td>
<td>“Sign” (negative)</td>
<td>b-a &lt; 0</td>
</tr>
<tr>
<td>jns</td>
<td>(non-negative)</td>
<td>b-a &gt;=0</td>
</tr>
<tr>
<td>jg</td>
<td>“Greater”</td>
<td>b &gt; a</td>
</tr>
<tr>
<td>jge</td>
<td>“Greater or equal”</td>
<td>b &gt;= a</td>
</tr>
<tr>
<td>jl</td>
<td>“Less”</td>
<td>b &lt; a</td>
</tr>
<tr>
<td>jle</td>
<td>“Less or equal”</td>
<td>b &lt;= a</td>
</tr>
<tr>
<td>ja</td>
<td>“Above” (unsigned &gt;)</td>
<td>b &gt; a</td>
</tr>
<tr>
<td>jb</td>
<td>“Below” (unsigned &lt;)</td>
<td>b &lt; a</td>
</tr>
</tbody>
</table>

if (x < 3 && x == y) {
    return 1;
} else {
    return 2;
}

T1: # x < 3 && x == y:
    movq $1, %rax
    ret
T2: # else
    movq $2, %rax
    ret
Choosing instructions for conditionals

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
<th>cmp a,b</th>
<th>test a,b</th>
</tr>
</thead>
<tbody>
<tr>
<td>je</td>
<td>“Equal”</td>
<td>b == a</td>
<td>b&amp;a == 0</td>
</tr>
<tr>
<td>jne</td>
<td>“Not equal”</td>
<td>b != a</td>
<td>b&amp;a != 0</td>
</tr>
<tr>
<td>js</td>
<td>“Sign” (negative)</td>
<td>b-a &lt; 0</td>
<td>b&amp;a &lt; 0</td>
</tr>
<tr>
<td>jns</td>
<td>(non-negative)</td>
<td>b-a &gt;=0</td>
<td>b&amp;a &gt;= 0</td>
</tr>
<tr>
<td>jg</td>
<td>“Greater”</td>
<td>b &gt; a</td>
<td>b&amp;a &gt; 0</td>
</tr>
<tr>
<td>jge</td>
<td>“Greater or equal”</td>
<td>b &gt;= a</td>
<td>b&amp;a &gt;= 0</td>
</tr>
<tr>
<td>jl</td>
<td>“Less”</td>
<td>b &lt; a</td>
<td>b&amp;a &lt; 0</td>
</tr>
<tr>
<td>jle</td>
<td>“Less or equal”</td>
<td>b &lt;= a</td>
<td>b&amp;a &lt;= 0</td>
</tr>
<tr>
<td>ja</td>
<td>“Above” (unsigned &gt;)</td>
<td>b &gt; a</td>
<td>b&amp;a &gt; 0U</td>
</tr>
<tr>
<td>jb</td>
<td>“Below” (unsigned &lt;)</td>
<td>b &lt; a</td>
<td>b&amp;a &lt; 0U</td>
</tr>
</tbody>
</table>

if (x < 3 && x == y) {
    return 1;
} else {
    return 2;
}

T1: # x < 3 && x == y:
    movq $1, %rax
    ret
T2: # else
    movq $2, %rax
    ret

[https://godbolt.org/g/Ovh3jN](https://godbolt.org/g/Ovh3jN)
Summary

- Control flow in x86 determined by status of Condition Codes
  - Showed Carry, Zero, Sign, and Overflow, though others exist
  - Set flags with arithmetic instructions (implicit) or Compare and Test (explicit)
  - Set instructions read out flag values
  - Jump instructions use flag values to determine next instruction to execute
BONUS SLIDES

Bonus content (nonessential). Does contain examples.

- Conditional Operator with Jumps
- Conditional Move
Conditional Operator with Jumps

C Code

\[
\text{val} = \text{Test} \ ? \ \text{Then-Expr} : \text{Else-Expr};
\]

Example:

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

Goto Version

\[
\text{ntest} = !\text{Test};
\text{if} (\text{ntest}) \text{ goto Else;}
\text{val} = \text{Then_Expr;}
\text{goto Done;}
\text{Else:}
\text{val} = \text{Else_Expr;}
\text{Done:}
\ldots
\]

- Ternary operator \( ? : \)
- \( \text{Test} \) is expression returning integer
  - = 0 interpreted as false
  - \( \neq 0 \) interpreted as true
- Create separate code regions for then & else expressions
- Execute appropriate one
Conditional Move

- Conditional Move Instructions: \texttt{cmovC src, dst}
  - Move value from \texttt{src} to \texttt{dst} if condition \texttt{C} holds
  - \texttt{if(Test) Dest} ← \texttt{Src}
  - GCC tries to use them (but only when known to be \texttt{safe})

- Why is this useful?
  - Branches are very disruptive to instruction flow through \texttt{pipelines}
  - Conditional moves do not require control transfer

\begin{Verbatim}
long absdiff(long x, long y)
{
    \textbf{return} x>y ? x-y : y-x;
}
\end{Verbatim}

\begin{Verbatim}
absdiff:
movq    %rdi, %rax # x
subq    %rsi, %rax # result=x-y
movq    %rsi, %rdx
subq    %rdi, %rdx # else_val=y-x
cmpq    %rsi, %rdi # x:y
cmovle  %rdx, %rax # if <=,
ret     # result=else_val
\end{Verbatim}

Bonus Content (nonessential)
more details at end of slides
Using Conditional Moves

- Conditional Move Instructions
  - `cmovC src, dest`
  - Move value from src to dest if condition `C` holds
  - Instruction supports:
    - if (Test) Dest ← Src
  - Supported in post-1995 x86 processors
  - GCC tries to use them
    - But, only when known to be safe

- Why is this useful?
  - Branches are very disruptive to instruction flow through pipelines
  - Conditional moves do not require control transfer

C Code

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

“Goto” Version

```c
result = Then_Expr;
else_val = Else_Expr;
nt = !Test;
if (nt) result = else_val;
return result;
```
Conditional Move Example

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

### Register Use(s)

<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>

### Bonus Content

**absdiff:**

- `movq %rdi, %rax`  
  - # x
- `subq %rsi, %rax`  
  - # result = x - y
- `movq %rsi, %rdx`  
- `subq %rdi, %rdx`  
  - # else_val = y - x
- `cmpq %rsi, %rdi`  
  - # x:y
- `cmovle %rdx, %rax`  
  - # if <=, result = else_val
- `ret`
Bad Cases for Conditional Move

Expensive Computations

\[
\text{val} = \text{Test}(x) \ ? \ \text{Hard1}(x) : \ \text{Hard2}(x);
\]

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

Risky Computations

\[
\text{val} = p \ ? \ *p : 0;
\]

- Both values get computed
- May have undesirable effects

Computations with side effects

\[
\text{val} = x > 0 \ ? \ x*=7 : x+=3;
\]

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