

Control flow (1)

Condition codes

Conditional and unconditional jumps

Loops

Conditional moves

Switch statements

Conditionals and Control Flow

Two key pieces

1. Comparisons and tests: check conditions
2. Transfer control: choose next instruction

Familiar C constructs

- if else
- while
- do while
- for
- break
- continue

Processor Control-Flow State

Condition codes (a.k.a. *flags*)

1-bit registers hold flags set by last ALU operation



Zero Flag result == 0



Sign Flag result < 0



Carry Flag carry-out/unsigned overflow



Overflow Flag two's complement overflow



Instruction pointer
(a.k.a. *program counter*)

register holds address of next instruction to execute

1. *compare* and *test*: conditions

ex

`cmpq b,a` computes $a - b$, sets flags, discards result

Which flags indicate that $a < b$? (signed? unsigned?)

`testq b,a` computes $a \& b$, sets flags, discards result

Common pattern:

`testq %rax, %rax`

*What do **ZF** and **SF** indicate?*

Aside: save conditions

setg: set if greater

stores byte:

0x01 if $\sim(SF \wedge OF) \& \sim ZF$

0x00 otherwise

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

```
cmpq %rsi,%rdi          # compare: x - y  
setg %al                # al = x > y  
movzbq %al,%rax       # zero rest of %rax
```

Zero-extend from **Byte** (8 bits) to **Quadword** (64 bits)

%rax

%eax

%ah

%al

2. *jump*: choose next instruction

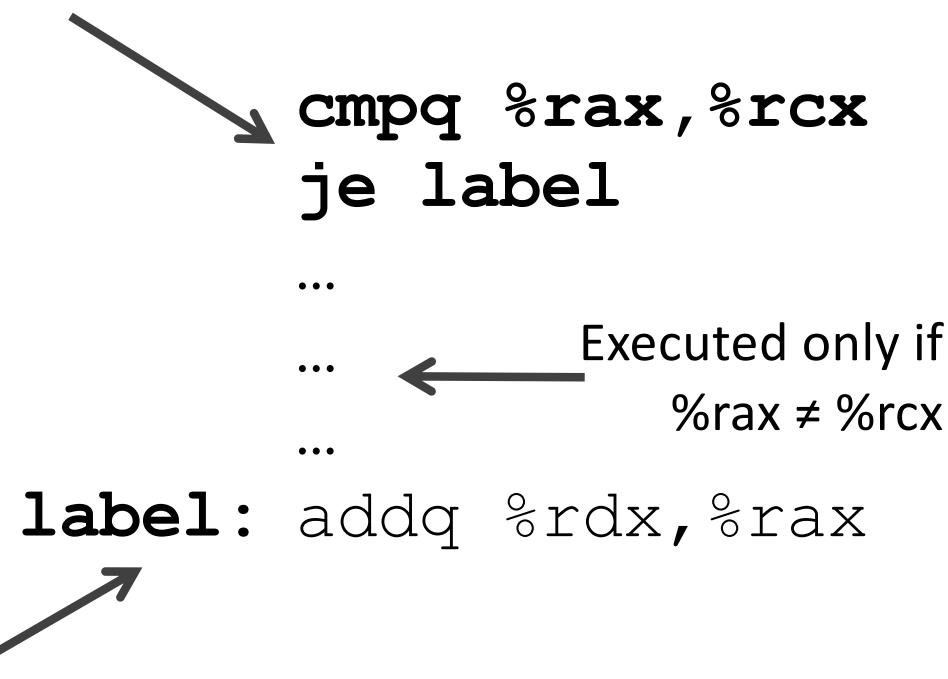
Jump/branch to different part of code by setting `%rip`.

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

Jump for control flow

Jump immediately follows comparison/test.

Together, they make a decision:
"if $\%rcx == \%rax$, jump to *label*."



Label

Name for address of
following item.

Conditional Branch Example

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

Labels
Name for address of
following item.

absdiff:

cmpq %rsi, %rdi

jle .L7

subq %rsi, %rdi

movq %rdi, %rax

.L8:

retq

.L7:

subq %rdi, %rsi

movq %rsi, %rax

jmp .L8

How did the compiler create this?

Control-Flow Graph

Code flowchart/directed graph.

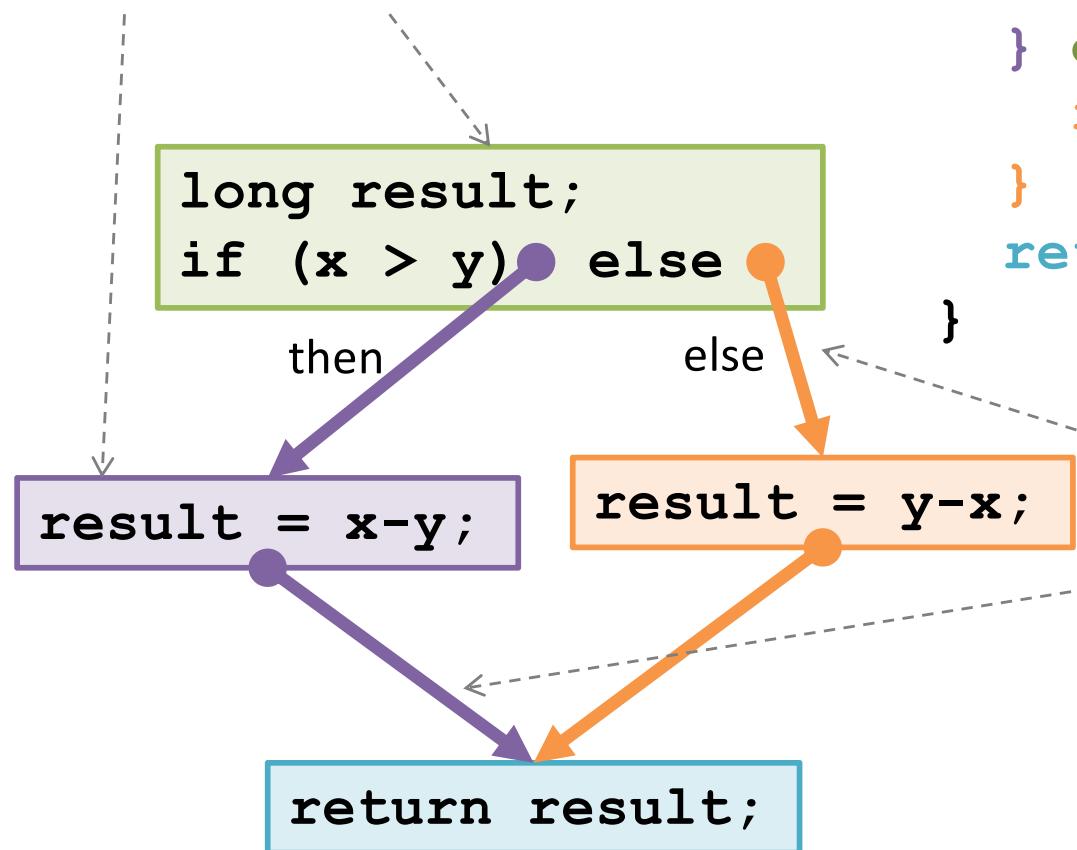
Introduced by Fran Allen, et al.

Won the 2006 Turing Award
for her work on compilers.



Nodes = **Basic Blocks**:

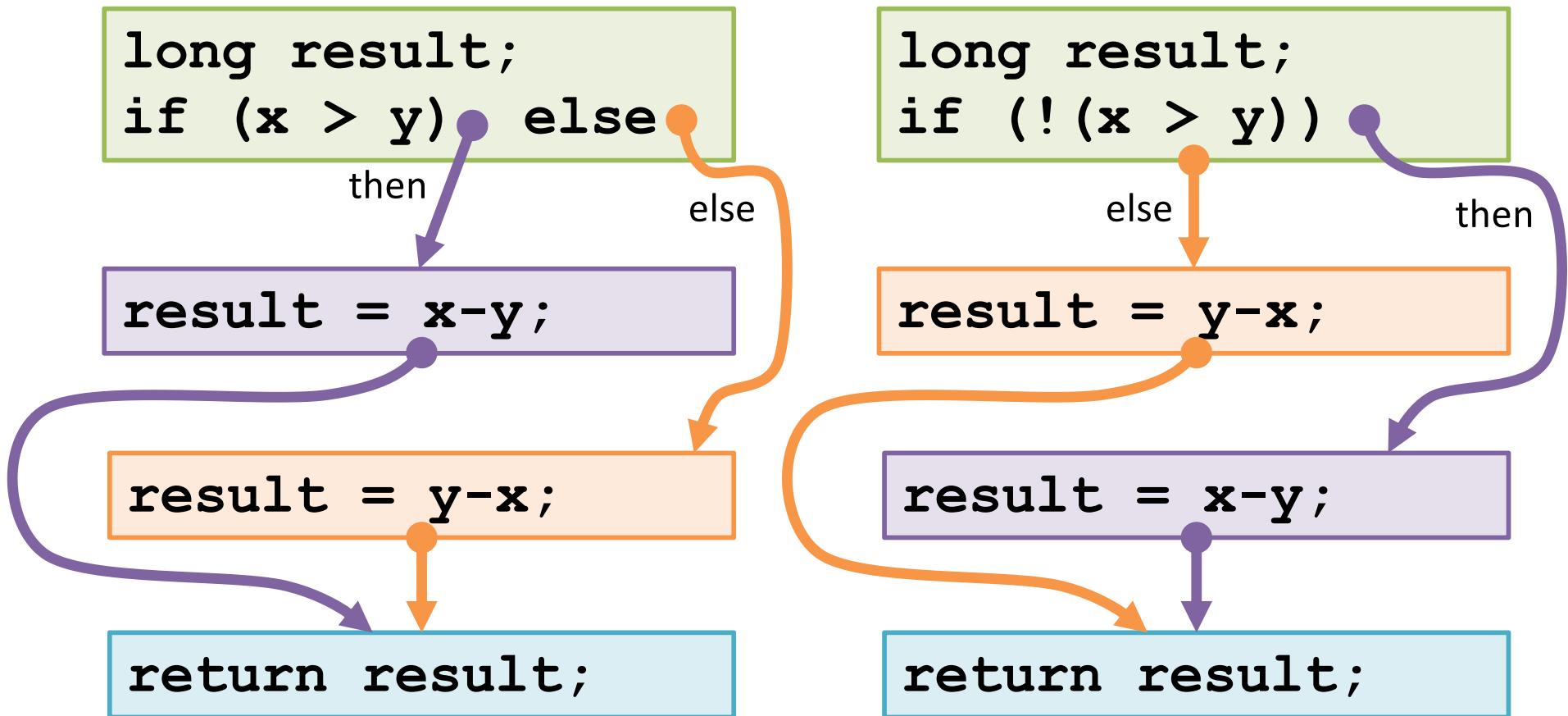
Straight-line code always
executed together in order.



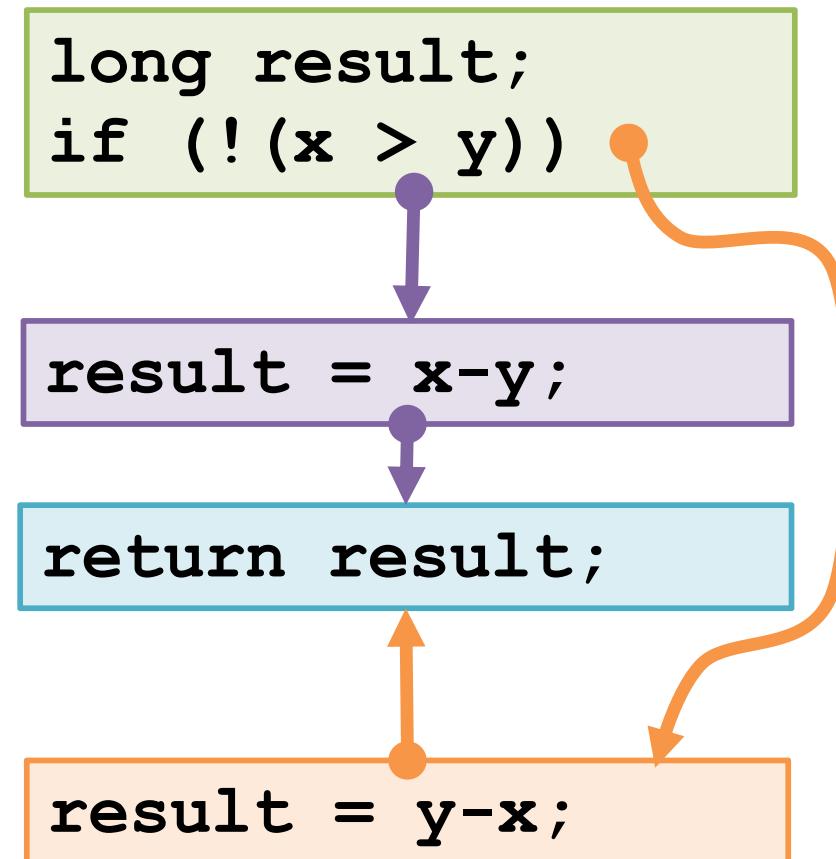
```
long absdiff(long x, long y) {
    long result;
    if (x > y) {
        result = x-y;
    } else {
        result = y-x;
    }
    return result;
}
```

Edges = **Control Flow**:
Which basic block executes
next (under what condition).

Choose a linear order of basic blocks.

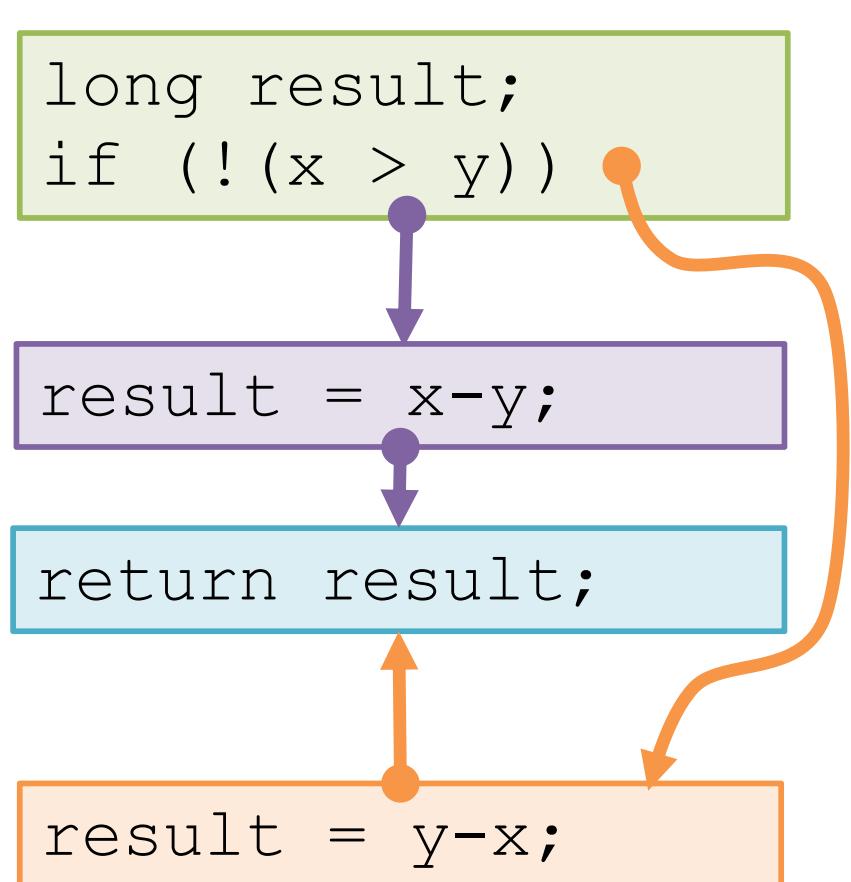


Choose a linear order of basic blocks.



Why might the compiler choose this basic block order instead of another valid order?

Translate basic blocks with jumps + labels



cmpq %rsi, %rdi
jle Else

subq %rsi, %rdi
movq %rdi, %rax

End:

retq

Else:

subq %rdi, %rsi
movq %rsi, %rax
jmp End

Why might the compiler choose this basic block order instead of another valid order?

Execute absdiff

```
cmpq    %rsi, %rdi  
jle    Else
```

```
subq    %rsi, %rdi  
movq    %rdi, %rax
```

End:

```
retq
```

Else:

```
subq    %rdi, %rsi  
movq    %rsi, %rax
```

```
jmp End
```

Registers

%rax	
%rdi	
%rsi	

Note: CSAPP shows translation with goto

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

```
long goto_ad(long x,long y){  
    int result;  
    if (x <= y) goto Else;  
    result = x-y;  
End:  
    return result;  
Else:  
    result = y-x;  
    goto End;  
}
```

Note: CSAPP shows translation with goto

```
long goto_ad(long x, long y) {  
    long result;  
    if (x <= y) goto Else;  
    result = x-y;  
End:  
    return result;  
Else:  
    result = y-x;  
    goto End;  
}
```

Close to assembly code.

absdiff:

```
cmpq    %rsi, %rdi  
jle    Else  
  
subq    %rsi, %rdi  
movq    %rdi, %rax
```

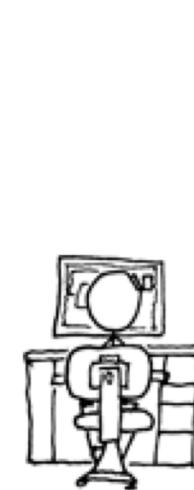
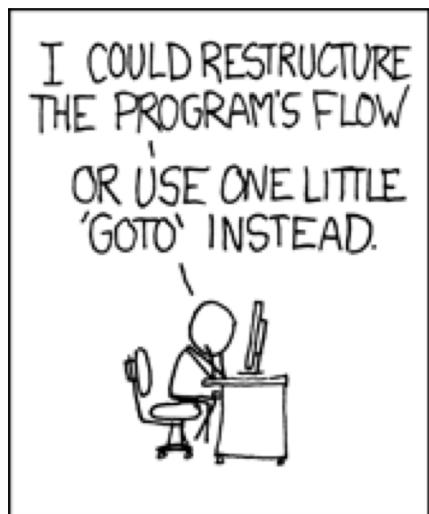
End:

```
retq
```

Else:

```
subq    %rdi, %rsi  
movq    %rsi, %rax  
jmp End
```

But never use goto in your source code!



<http://xkcd.com/292/>

ex

compile if-else

```
long wacky(long x, long y) {
    long result;
    if (x + y > 7) {
        result = x;
    } else {
        result = y + 2;
    }
    return result;
}
```

Assume x available in %rdi,
y available in %rsi.

Place result in %rax.

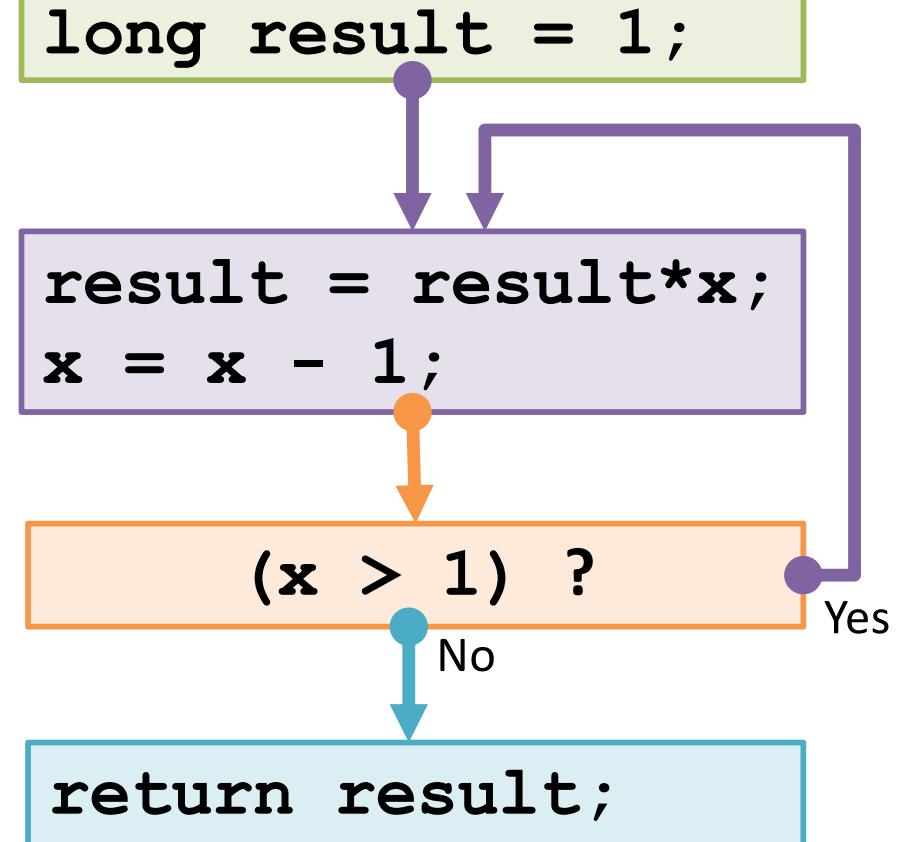
Encoding Jumps: PC-relative addressing

0x100	cmpq %rax, %rbx	0x1000
0x102	je 0x70	0x1002
0x104	...	0x1004
...
0x174	addq %rax, %rbx	0x1074

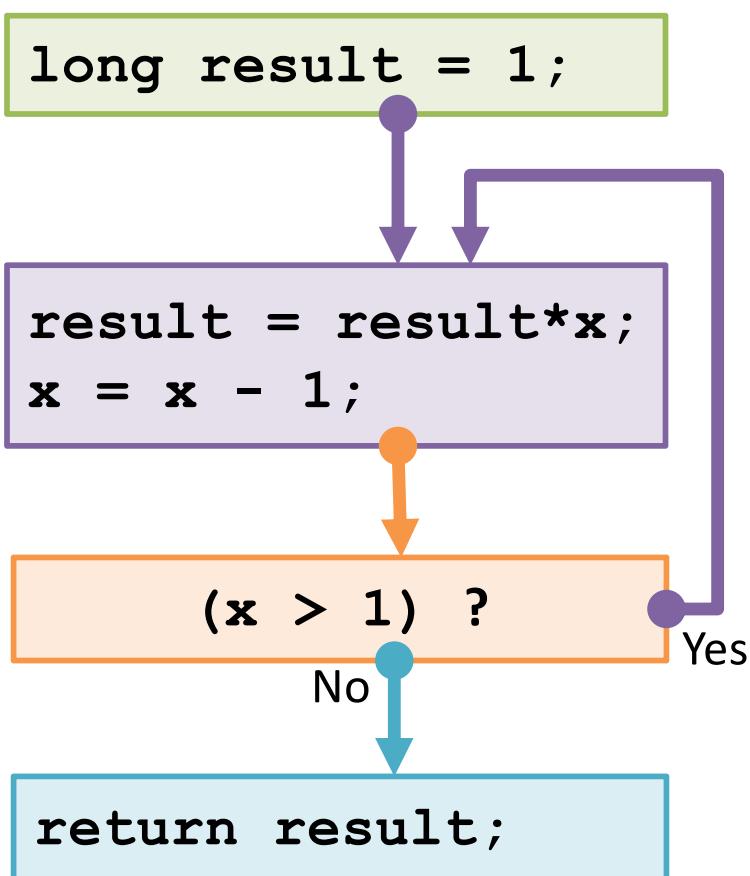
- PC-relative *offsets* support relocatable code.
- Absolute branches do not (or it's hard).

do while loop

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



do while loop



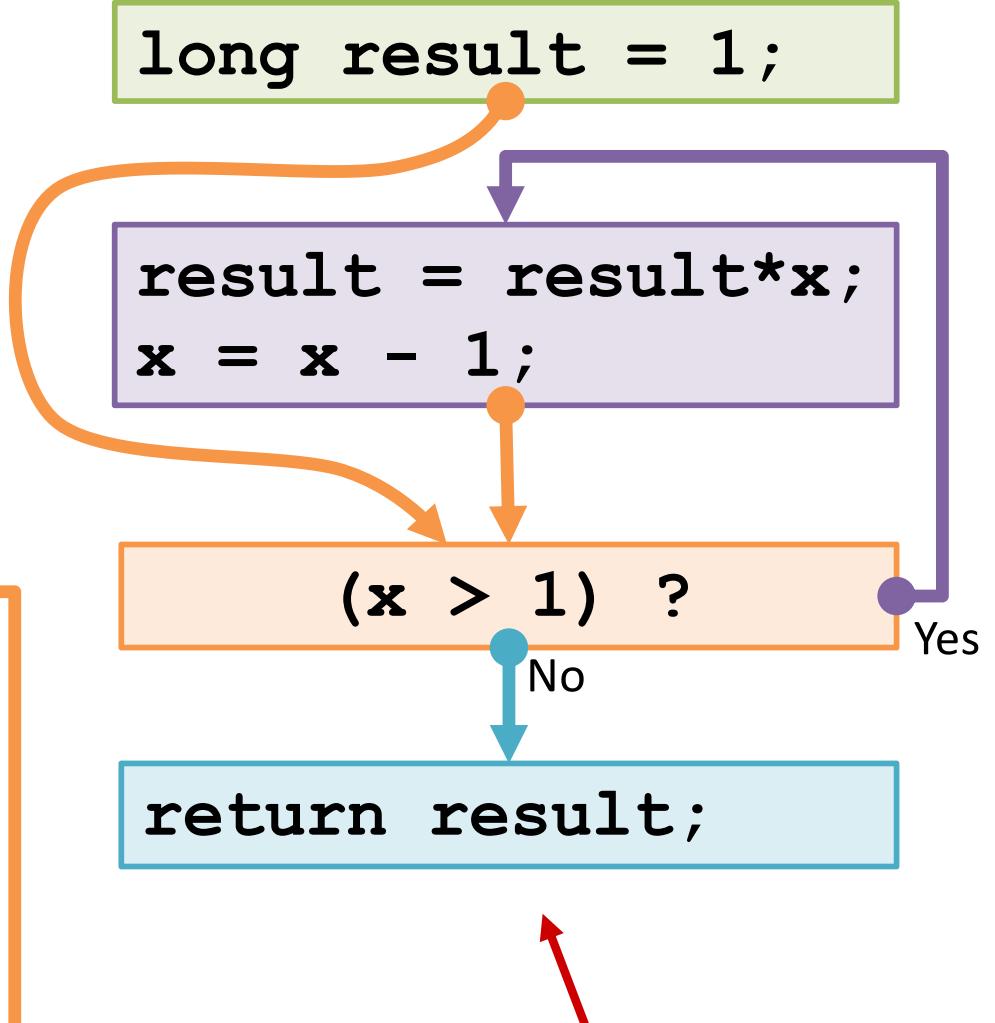
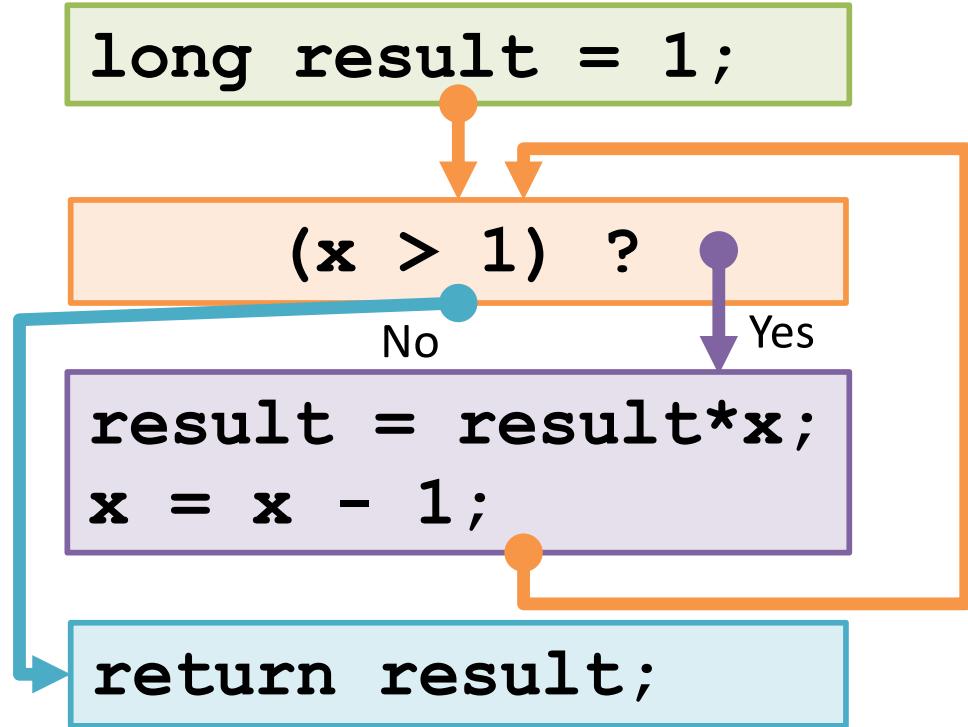
```
fact_do:  
    movq $1,%rax  
  
.L11:  
    imulq %rdi,%rax  
    decq %rdi  
  
    cmpq $1,%rdi  
    jg .L11  
  
    retq
```

Register	Variable
%rdi	
%rax	

Why put the loop condition at the end?

while loop

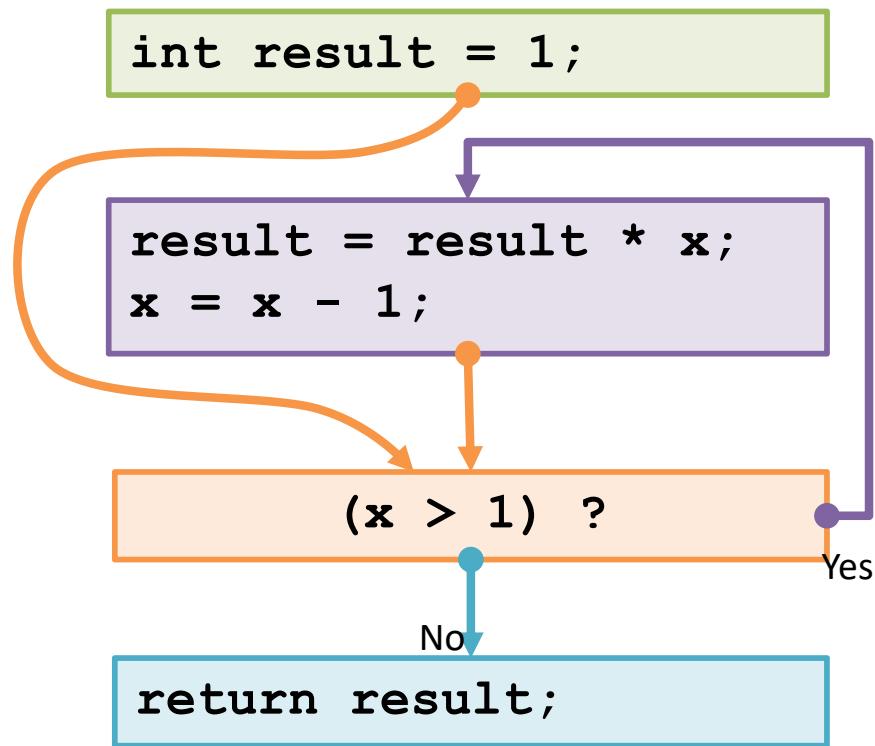
```
long fact_while(long x) {  
    long result = 1;  
    while (x > 1) {  
        result = result * x;  
        x = x - 1;  
    }  
    return result;  
}
```



This order is used by GCC for x86-64. Why?

while loop

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



`fact_while:`

```
    movq $1, %rax
    jmp .L34
```

`.L35:`

```
    imulq %rdi, %rax
    decq %rdi
```

`.L34:`

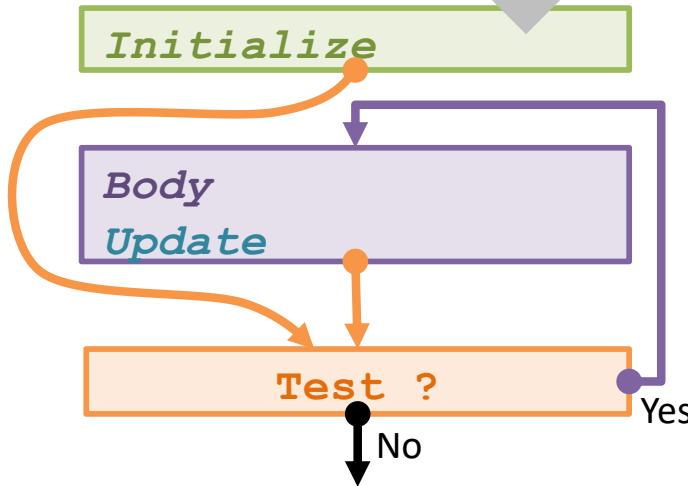
```
    cmpq $1, %rdi
    jg .L35
```

`retq`

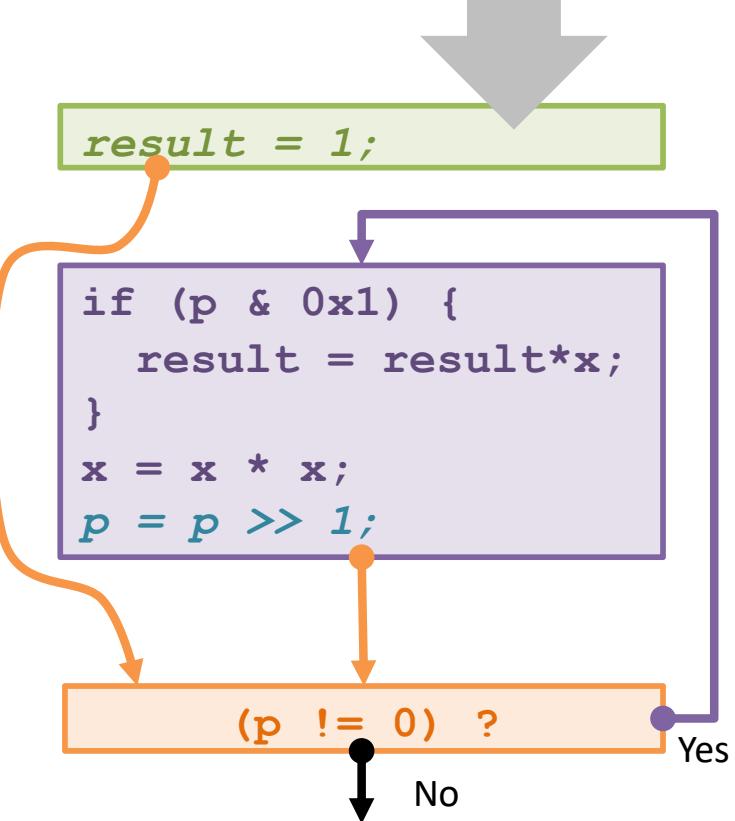
for loop translation

```
for (Initialize; Test; Update) {
    Body
}
```

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



```
for (result = 1; p != 0; p = p>>1) {
    if (p & 0x1) {
        result = result * x;
    }
    x = x * x;
}
```



Control flow (2)

Condition codes

Conditional and unconditional jumps

Loops

Conditional moves

Switch statements

(Aside) Conditional Move

Why? Branch prediction in pipelined/OoO processors.

cmove_ src, dest

if (*Test*) Dest \leftarrow Src

```
long absdiff(long x, long y) {  
    return x>y ? x-y : y-x;  
}
```

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

absdiff:

```
movq    %rdi, %rax  
subq    %rsi, %rax  
movq    %rsi, %rdx  
subq    %rdi, %rdx  
cmpq    %rsi, %rdi  
cmovle %rdx, %rax  
ret
```

(Aside) Bad Cases for Conditional Move

Expensive Computations

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

Risky Computations

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

Computations with side effects

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

switch statement

```
long switch_eg (long x, long y, long z) {  
    long w = 1;  
    switch(x) {  
        case 1:  
            w = y * z;  
            break;  
        case 2:  
            w = y - z; // ←  
        case 3:  
            w += z;  
            break;  
        case 5:  
        case 6:  
            w -= z;  
            break;  
        default: // ←  
            w = 2;  
    }  
    return w;  
}
```

Fall through cases

Multiple case labels

Missing cases use default

**Lots to manage:
use a *jump table*.**

switch jump table structure

C code:

```
switch(x) {  
    case 1: <some code>  
        break;  
    case 2: <some code>  
    case 3: <some code>  
        break;  
    case 5:  
    case 6: <some code>  
        break;  
    default: <some code>  
}
```

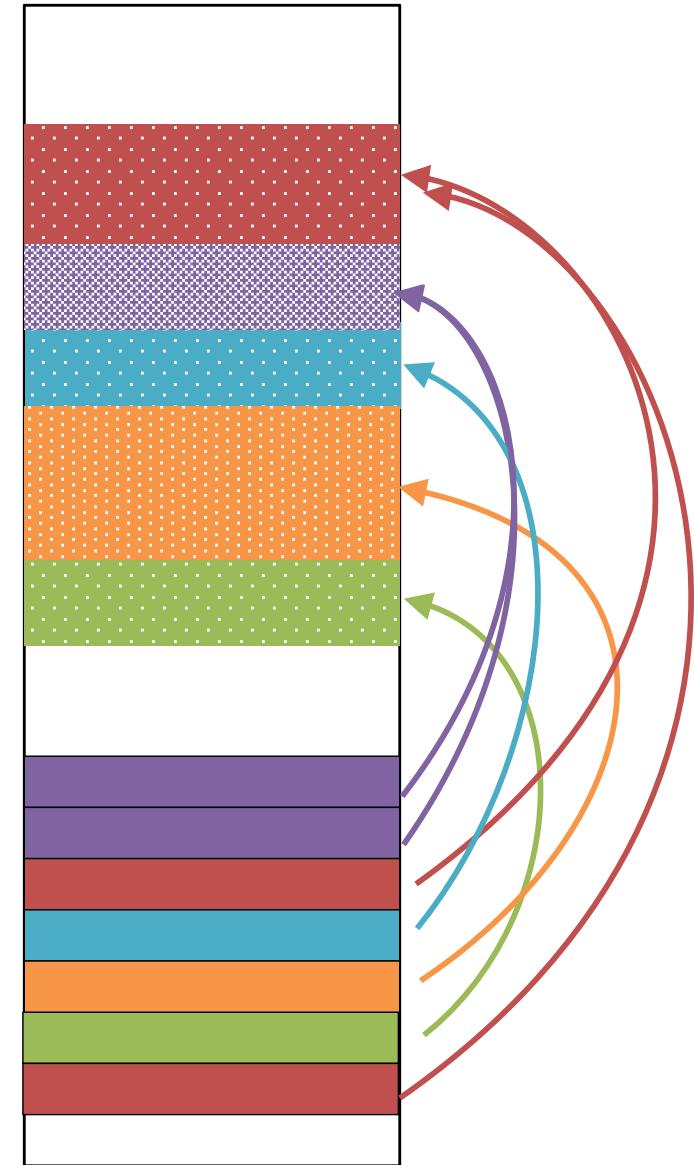
Code
Blocks

Jump
Table

Translation sketch:

```
if (0 <= x && x <= 6)  
    addr = jumptable[x] ;  
    goto addr;  
else  
    goto default;
```

Memory



switch jump table assembly declaration

*read-only data
(not instructions)*

```
.section .rodata      8-byte alignment
.align 8 ←
.L4:
.quad .L8 # x == 0
.quad .L3 # x == 1
.quad .L5 # x == 2
.quad .L9 # x == 3
.quad .L8 # x == 4
.quad .L7 # x == 5
.quad .L7 # x == 6
```

“quad” = *q* suffix = 8-byte value

```
switch(x) {
    case 1: // .L3
        w = y * z;
        break;
    case 2: // .L5
        w = y - z;
    case 3: // .L9
        w += z;
        break;
    case 5:
    case 6: // .L7
        w -= z;
        break;
    default: // .L8
        w = 2;
}
```

switch case dispatch

```
long switch_eg(long x, long y, long z) {  
    long w = 1;  
    switch(x) {  
        . . .  
    }  
    return w;  
}
```

Jump if above (unsigned, but...)

```
switch_eg:  
    movl $1, %eax  
    cmpq $6, %rdi  
    ja .L8  
    jmp * .L4(,%rdi,8)
```

indirect jump

Jump table

```
.section .rodata  
.align 8  
.L4:  
.quad .L8 # x == 0  
.quad .L3 # x == 1  
.quad .L5 # x == 2  
.quad .L9 # x == 3  
.quad .L8 # x == 4  
.quad .L7 # x == 5  
.quad .L7 # x == 6
```

switch cases

```

switch(x) {
    case 1:      // .L3
        w = y * z;
        break;
    case 2:      // .L5
        w = y - z;
    case 3:      // .L9
        w += z;
        break;
    case 5:      // .L7
    case 6:      // .L7
        w == z;
        break;
default:     // .L8
        w = 2;
}
return w;

```

Reg.	Use
%rdi	x
%rsi	y
%rdx	z
%rax	w

.L3: movq %rsi, %rax
 imulq %rdx, %rax
 retq ← "inlined"

.L5: movq %rsi, %rax
 subq %rdx, %rax

.L9: ← Fall-through.
 addq %rcx, %rax
 retq

.L7:
 subq %rdx, %rax
 retq

.L8:
 movl \$2, %eax
 retq

Aside: movl is used because 2 is a small positive value that fits in 32 bits. High order bits of %rax get set to zero automatically. It takes *one byte fewer* to encode a literal movl vs a movq.

switch machine code

Disassembled Object Code

```
00000000004004f6 <switch_eg>:
```

```
    . . .
```

```
4004fd: 77 2b
```

```
        ja 40052a <switch_eg+0x34>
```

```
4004ff: ff 24 fd d0 05 40 00
```

```
        jmpq *0x4005d0 (,%rdi,8)
```

Assembly Code

```
switch_eg:
```

```
    . . .
```

```
    cmpq $6, %rdi
```

```
    ja .L8
```

```
    jmp * .L4 (,%rdi,8)
```

Inspect jump table contents using GDB.

Examine contents as 7a addresses

(gdb) x/7a 0x4005d0

	<i>Address of code for case 0</i>	<i>Address of code for case 1</i>	<i>Address of code for case 6</i>
0x4005d0:	0x40052a <switch_eg+52>	0x400506 <switch_eg+16>	
0x4005e0:	0x40050e <switch_eg+24>	0x400518 <switch_eg+34>	
0x4005f0:	0x40052a <switch_eg+52>	0x400521 <switch_eg+43>	
0x400600:	0x400521 <switch_eg+43>		

Diagram showing arrows pointing from the assembly code to the memory dump:

- An arrow points from the `ja` instruction to the address `0x40052a` in the memory dump.
- An arrow points from the `jmpq` instruction to the address `0x4005d0` in the memory dump.
- An arrow points from the `jmp` instruction to the address `0x400521` in the memory dump.

Would you implement this with a jump table?

```
switch(x) {  
    case 0:      <some code>  
        break;  
    case 10:     <some code>  
        break;  
    case 52000:  <some code>  
        break;  
    default:    <some code>  
        break;  
}
```