Control flow

Condition codes

Conditional and unconditional jumps

Loops

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

ZF	Zero Flag	result == 0
SF	Sign Flag	result < 0
CF	Carry Flag	carry-out/unsigned overflow
OF	Overflow Flag	two's complement overflow

`%rip` **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:

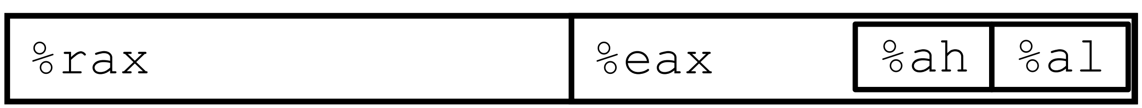
0x01 if `~(SF^OF)&~ZF`

0x00 otherwise

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

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

Zero-extend from **B**yte (8 bits) to **Q**uadword (64 bits)



2. ***jump***: choose next instruction

Jump/branch to different part of code by setting **%eip**.

Always jump: jmp

Jump iff *condition*: je through jb

j__	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)

Jump for control flow

Jump immediately follows comparison/test.

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

```
        cmpq %rax,%rcx
        je label
        …
        …
        …
label:  addq %rdx,%rax
```

Executed only if
%rax ≠ %rcx

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;
}
```

```
absdiff:
        cmpq    %rsi, %rdi
        jle     .L7
        subq    %rsi, %rdi
        movq    %rdi, %rax
.L8:
        retq
.L7:
        subq    %rdi, %rsi
        jmp     .L8
```

Labels

Name for address of following item.

How did the compiler create this?

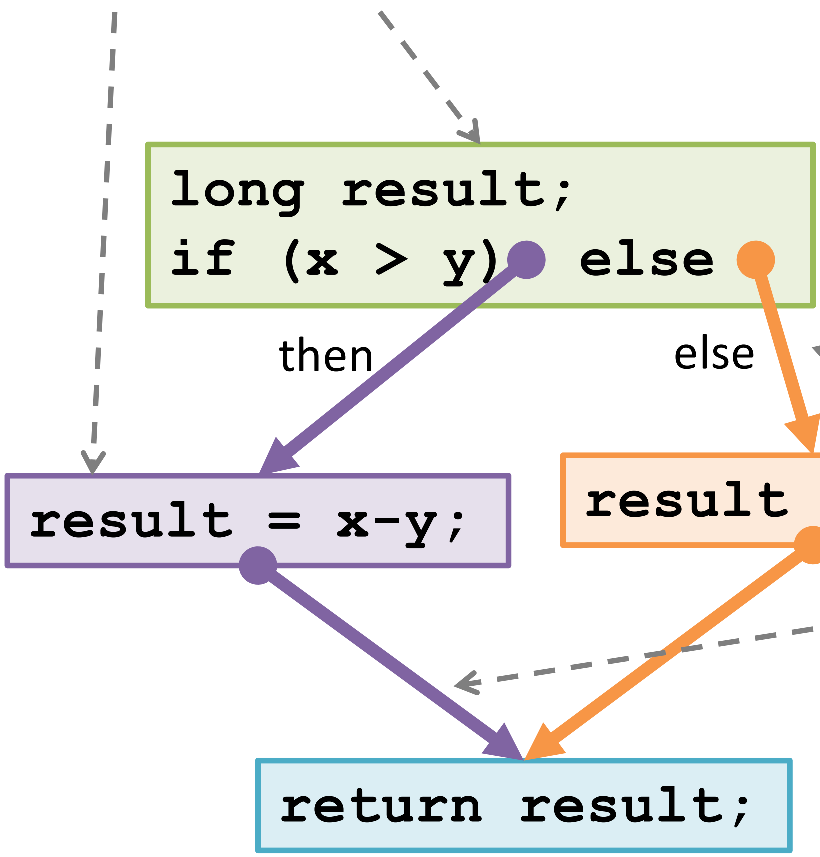
Control-Flow Graph

Code flowchart/directed graph.

Introduced by Fran Allen, et al.
Won the 2006 Turing Award
for her work on compilers.

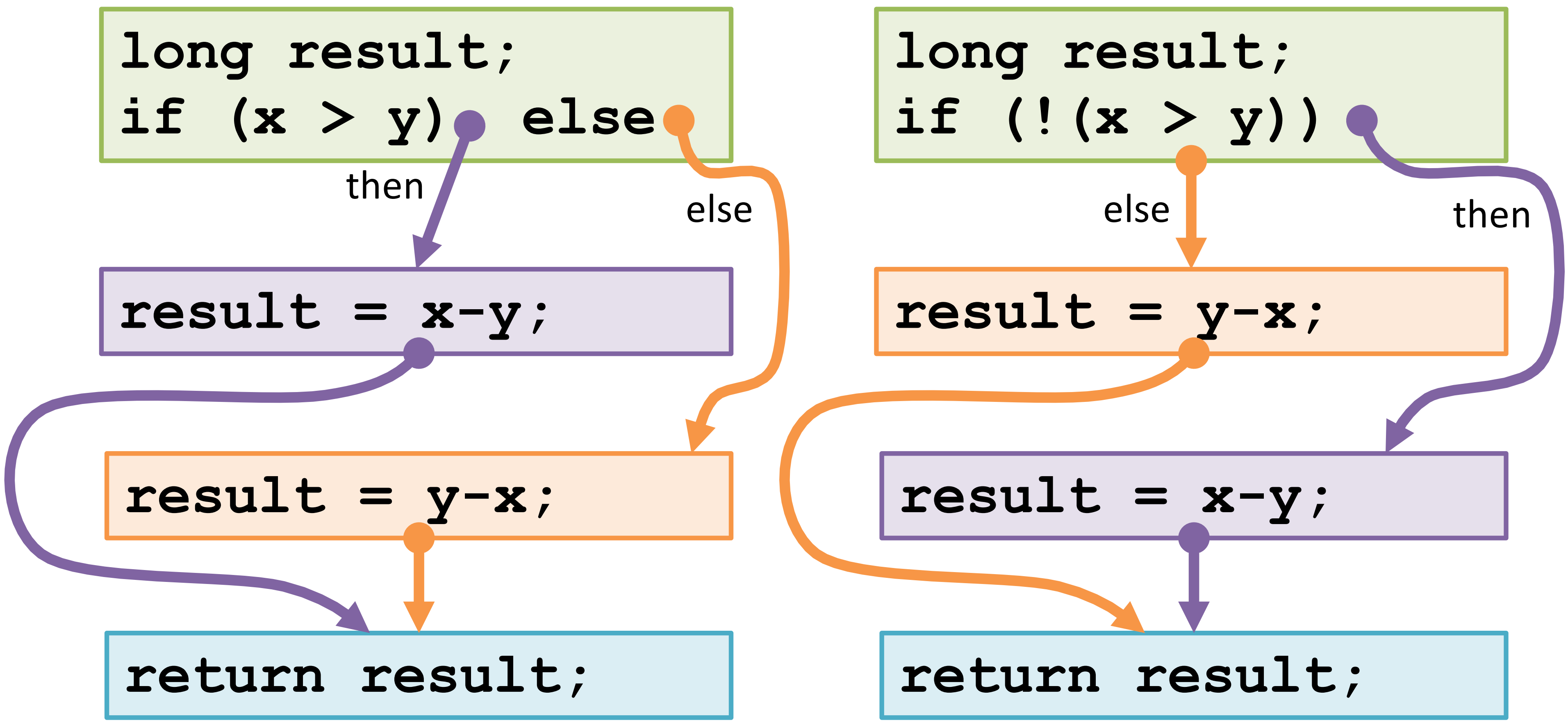
```
long absdiff(long x, long y){
   long result;
   if (x > y) {
      result = x-y;
   } else {
      result = y-x;
   }
   return result;
}
```

Nodes = ***Basic Blocks***:
Straight-line code always executed together in order.

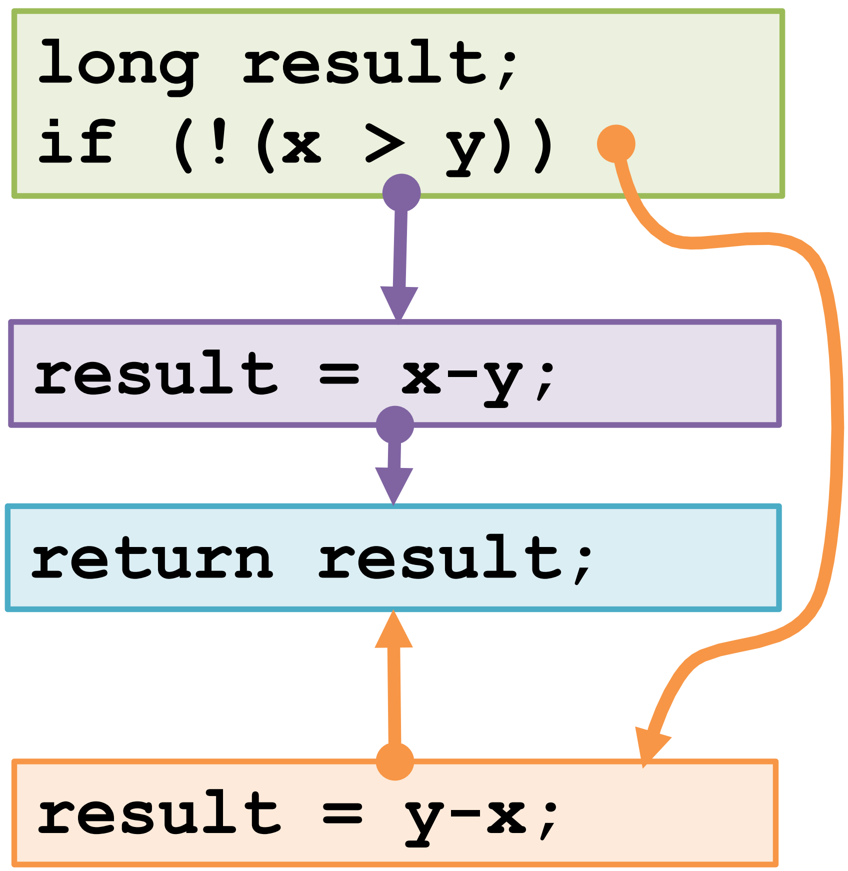


Edges = ***Control Flow***:
Which basic block executes next (under what condition).

Choose a linear order of basic blocks.

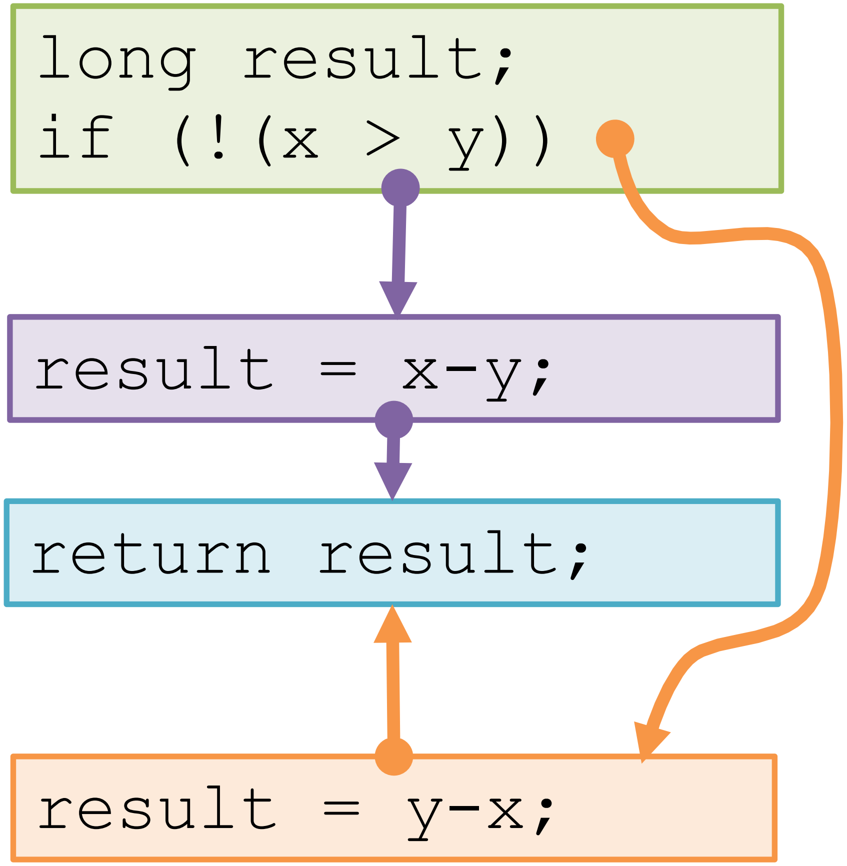


Choose a linear order of basic blocks.



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

ex

```
    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;
}
```

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

        subq    %rsi, %rdi
        movq    %rdi, %rax
End:
        retq
Else:
        subq    %rdi, %rsi
        movq    %rsi, %rax
        jmp End
```

Close to assembly code.

But never use goto in your source code!

http://xkcd.com/292/

compile if-else

ex

```
long wacky(long x, long y){
  int 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).

Compiling Loops

C/Java code:

```
while ( sum != 0 ) {
   <loop body>
}
```

Machine code:

```
loopTop:    testq %rax, %rax
            je    loopDone
              <loop body code>
            jmp   loopTop
loopDone:
```

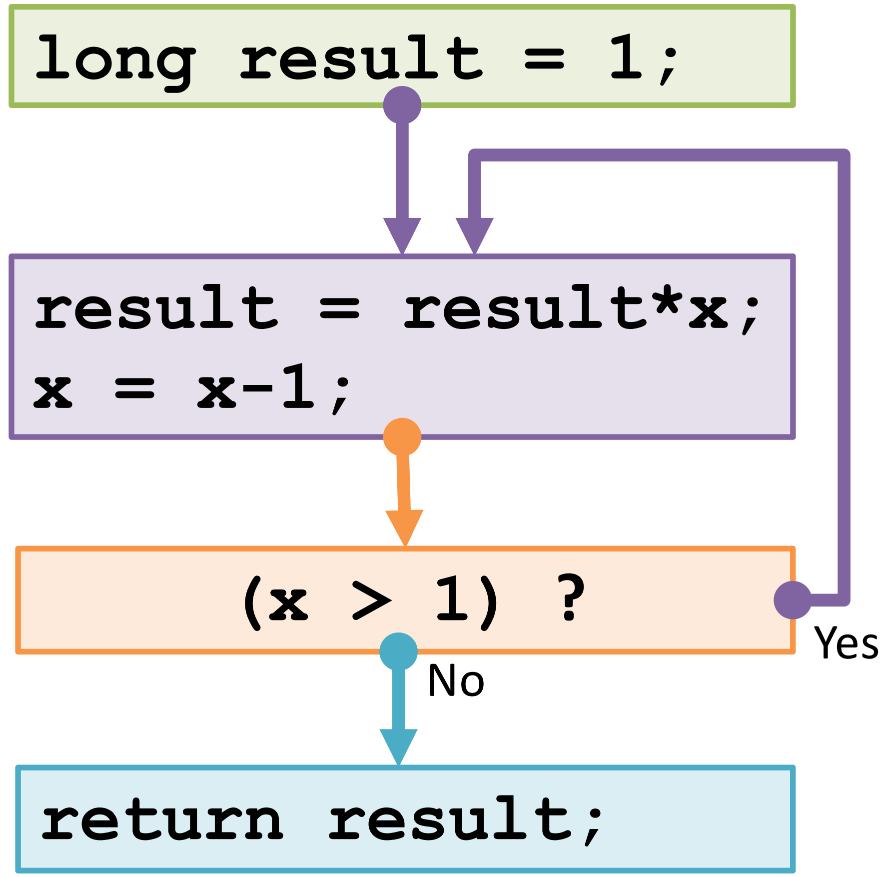
Compilation of other loops should be straightforward

Interesting part: put the conditional branch at top or bottom of loop?

do while loop example

C Code

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

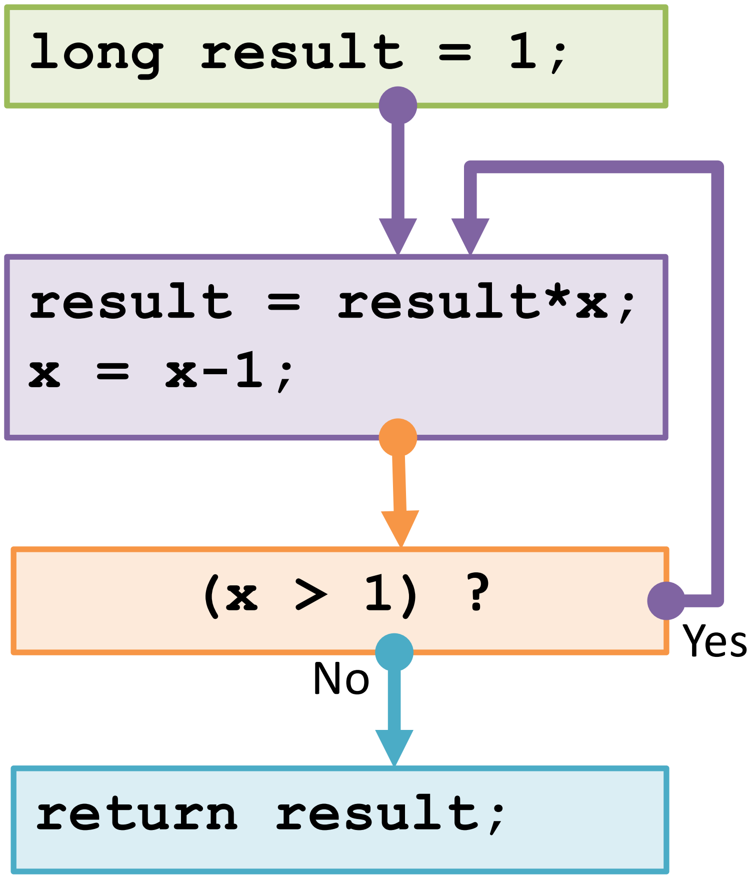


Keys:

- Use backward branch to continue looping
- Only take branch when “while” condition holds

do while loop translation

Register	Variable
%rdi	
%rax	



Assembly

```
fact_do:
  movq $1,%rax

.L11:
  imulq %rdi,%rax
  decq %rdi
  cmpq $1,%rdi
  jg .L11

  retq
```

Why?

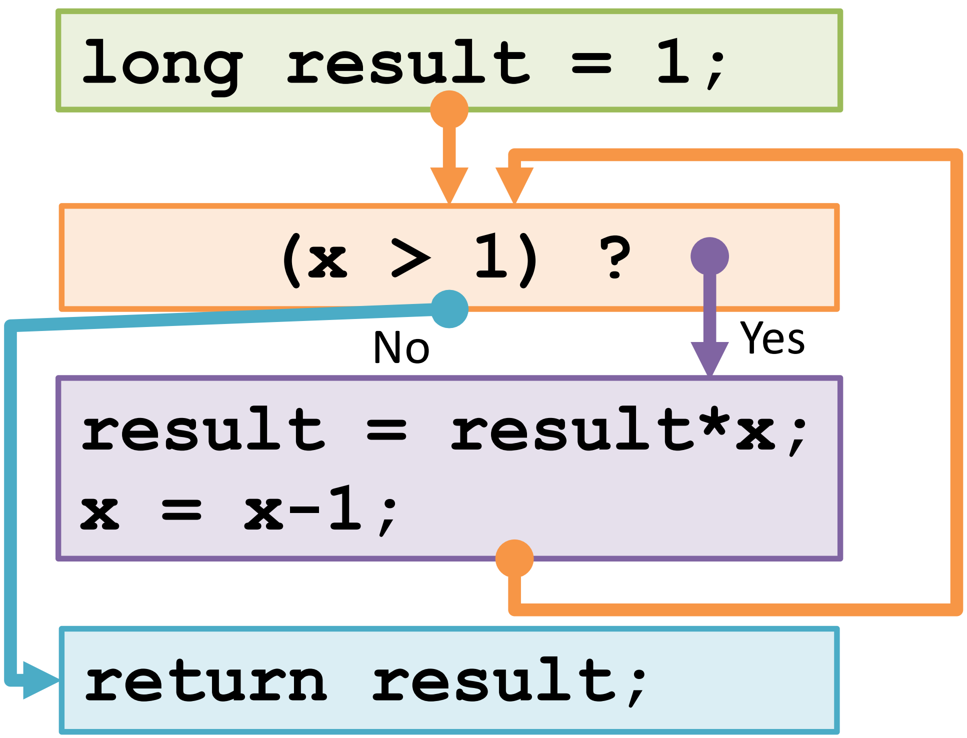
Why put the loop condition at the end?

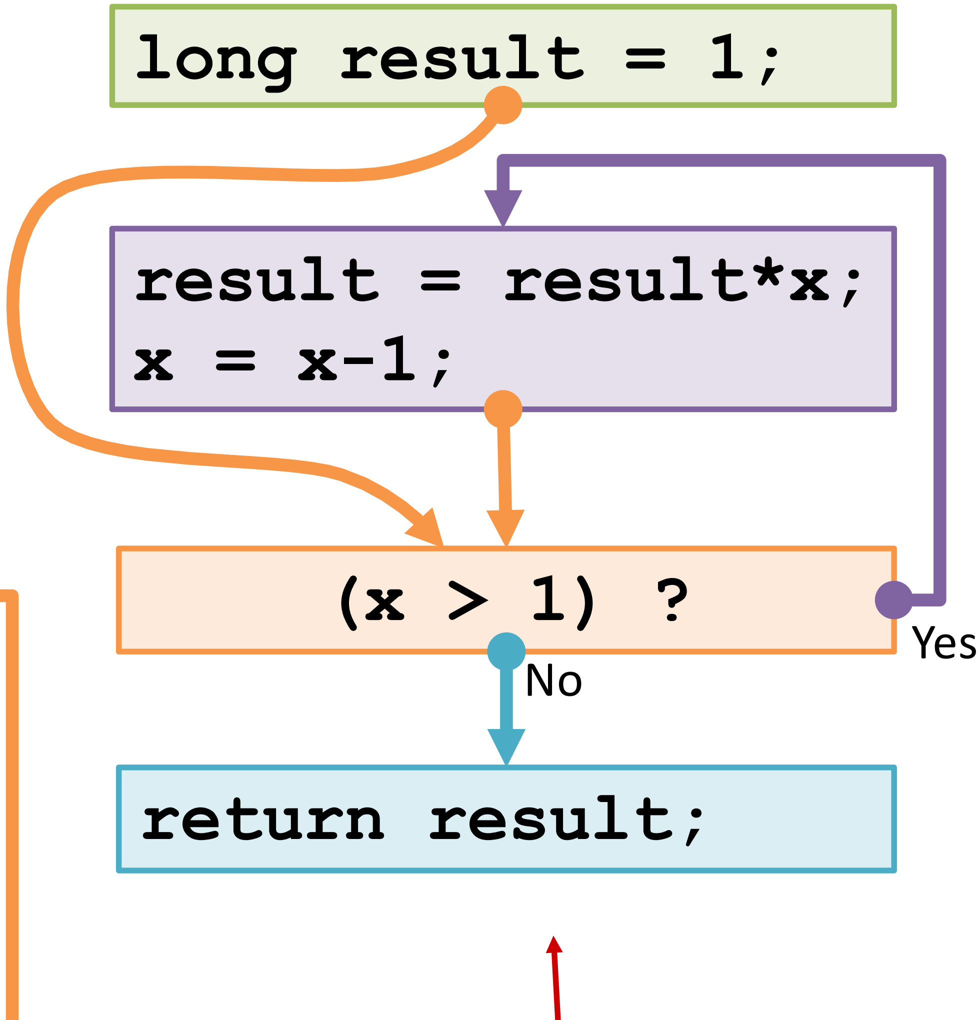
while loop translation

Why?

C Code

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



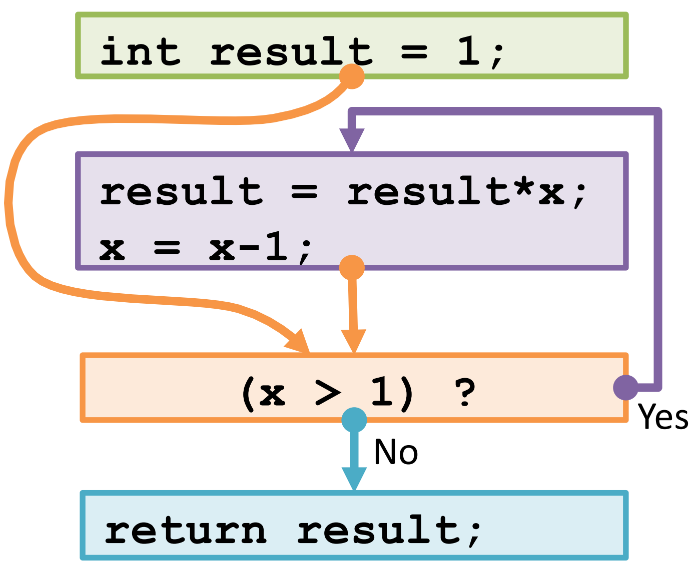


This order is used by GCC for x86-64

while loop example

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

```
  movq $1, %rax
  jmp    .L34
.L35:
  imulq %rdi, %rax
  decq  %rdi
.L34:
  cmpq  $1, %rdi
  jg    .L35
```



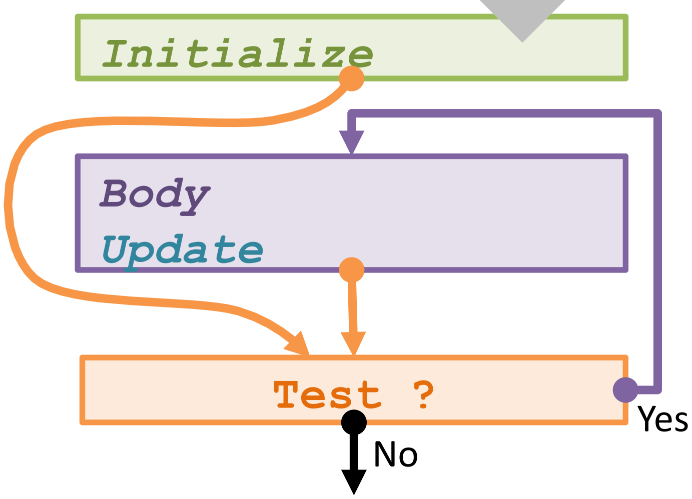
`for` loop translation

For Version

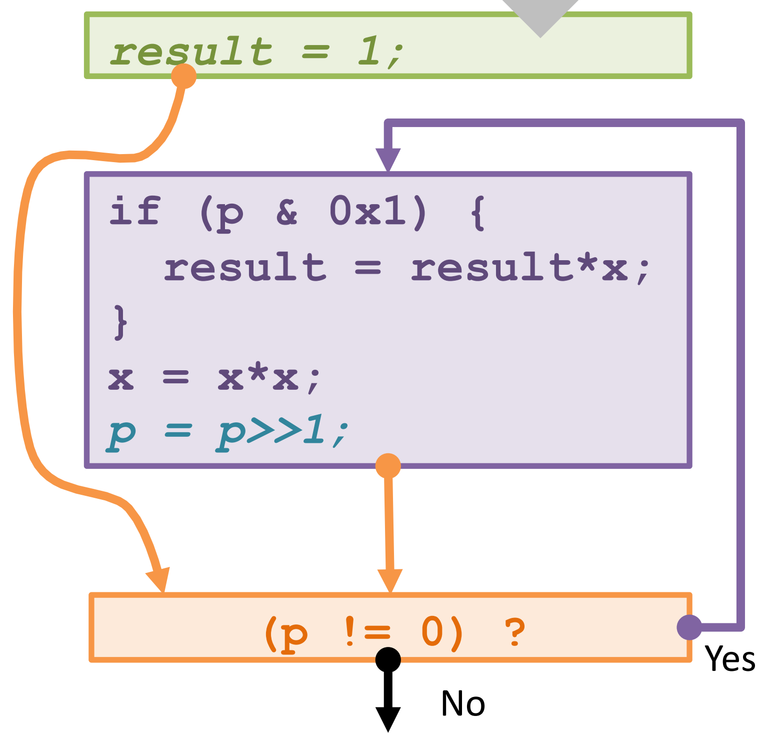
```
for (Initialize; Test; Update )
    Body
```

While Version

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




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



(Aside) **Conditional Move**

`cmov` src, dest   if (*Test*) *Dest* ← *Src*

Why? Branch prediction in pipelined/OoO processors.

```
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;
  }
}
```

```
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
```

(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 statements

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

Fall through cases

Missing cases

Multiple case labels

Lots to manage, let's use a *jump table*

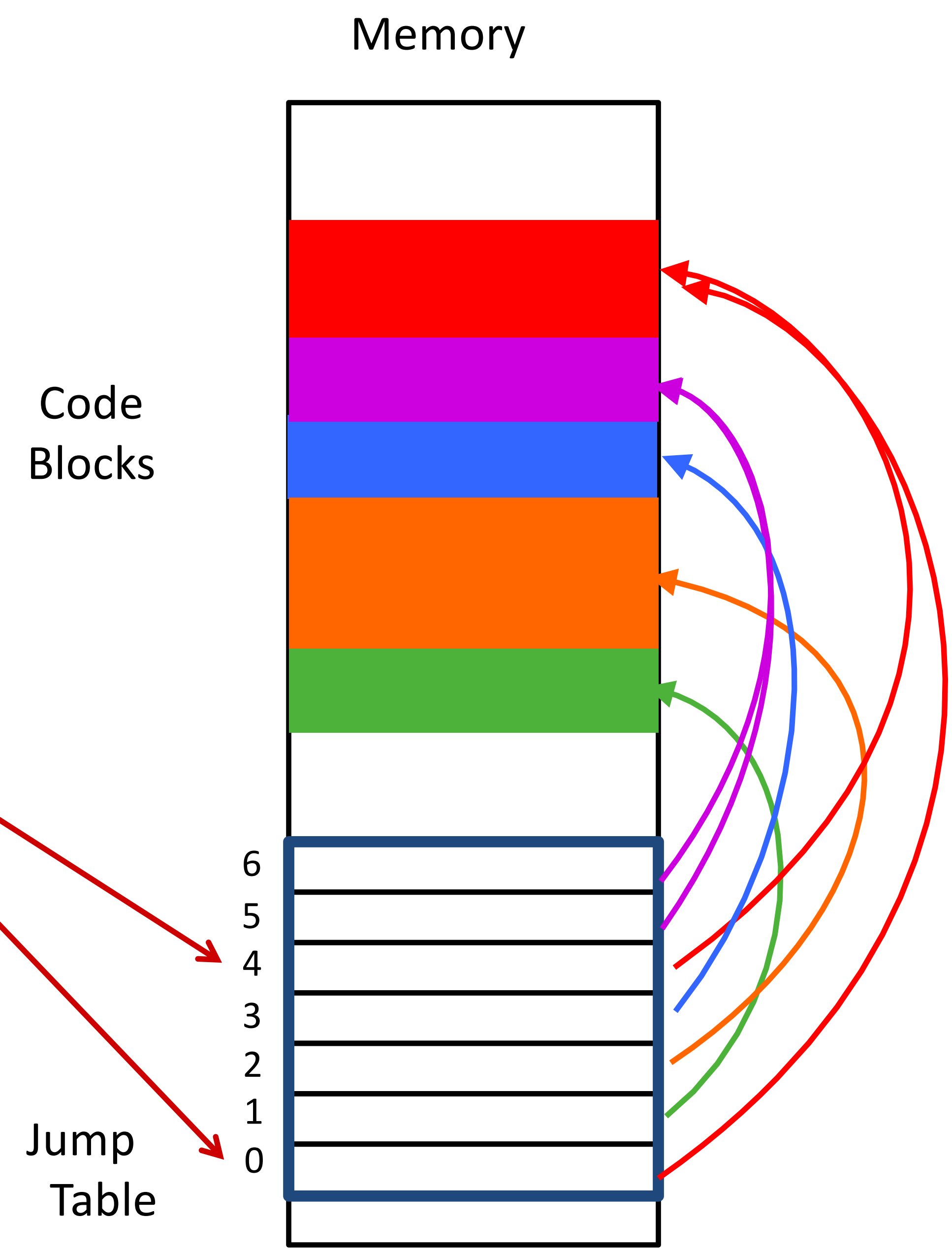
Jump Table Structure

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>
}
```

We can use the jump table when x <= 6:

```
if (x <= 6)
  target = JTab[x];
  goto target;
else
  goto default;
```



Jump Table

declaring data, not instructions

8-byte memory alignment

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
```

“quad” as in 4 1978-era 16-bit words

```
switch(x) {
case 1:         // .L56
    w = y*z;
    break;
case 2:         // .L57
    w = y/z;
    /* Fall Through */
case 3:         // .L58
    w += z;
    break;
case 5:
case 6:         // .L60
    w -= z;
    break;
default:        // .L61
    w = 2;
}
```

`switch` statement example

ex

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

but this is signed...

Jump if above (like jg, but unsigned)

Indirect jump

```
switch_eg:
   movq   %rdx, %rcx
   cmpq   $6, %rdi
   ja     .L8
   jmp    *.L4(,%rdi,8)
```

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
```

Code Blocks (x == 1)

```
switch(x) {
case 1:         // .L3
     w = y*z;
     break;
   . . .
}
return w;
```

```
.L3:
   movq    %rsi, %rax   # y
   imulq   %rdx, %rax   # y*z
   ret
```

Compiler has "inlined" the return.

Register	Use(s)
`%rdi`	Argument **x**
`%rsi`	Argument **y**
`%rdx`	Argument **z**
`%rax`	Return value

Handling Fall-Through

```
long w = 1;
switch (x) {
. . .
case 2:   // .L5
    w = y/z;
    /* Fall Through */
case 3:   // .L9
    w += z;
    break;
. . .
}
```

```
case 2:
    w = y/z;
    goto merge;
```

```
case 3:
    w = 1;
```

```
merge:
    w += z;
```

Compiler has inlined "w = 1" only where necessary.

Code Blocks (x == 2, x == 3)

```
long w = 1;
switch (x) {
  . . .
case 2: // .L5
    w = y/z;
    /* Fall Through */
case 3: // .L9
    w += z;
    break;
   . . .
}
```

```
.L5:                       # Case 2
   movq    %rsi, %rax  #  y in rax
   cqto                   #  Div prep
   idivq   %rcx           #  y/z
   jmp     .L6            #  goto merge
.L9:                       # Case 3
   movl    $1, %eax     #  w = 1
.L6:                       # merge:
   addq    %rcx, %rax  #  w += z
   ret
```

Compiler has inlined "w = 1" only where necessary.

Register	Use(s)
`%rdi`	Argument **`x`**
`%rsi`	Argument **`y`**
`%rdx`	Argument **`z`**
`%rax`	Return value

Aside: `movl` is used because 1 is a small positive value that fits in 32 bits. High order bits of `%rax` get set to zero automatically. It takes *one less byte* to encode a `movl` than a `movq`.

Code Blocks (x == 5, x == 6, default)

```
long w = 1;
switch (x) {
  . . .
case 5:  // .L7
case 6:  // .L7
    w -= z;
    break;
default: // .L8
    w = 2;
}
```

```
.L7:                       # Case 5,6
   movl   $1, %eax         #  w = 1
   subq   %rdx, %rax       #  w -= z
   ret
.L8:                       # Default:
   movl   $2, %eax         #  2
   ret
```

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

`switch` machine code

Setup

- Label **.L8:** 0x000000000040052a
- Label **.L4:** 0x00000000004005d0

Assembly Code

```
switch_eg:
  . . .
  ja      .L8
  jmp     *.L4(,%rdi,8)
```

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)
```

Inspect jump table using GDB.

- E**x**amine contents as **7 a**ddresses
- Use command "**help x**" to get format documentation

```
(gdb) x/7a 0x00000000004005d0
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>
```

Matching Disassembled Targets

Jump table contents:

0x4005d0:

```
0x40052a
0x400506
0x40050e
0x400518
0x40052a
0x400521
0x400521
```

Section of disassembled `switch_eg`:

```
400506:  48 89 f0          mov    %rsi,%rax
400509:  48 0f af c2       imul   %rdx,%rax
40050d:  c3                retq
40050e:  48 89 f0          mov    %rsi,%rax
400511:  48 99             cqto
400513:  48 f7 f9          idiv   %rcx
400516:  eb 05             jmp    40051d <switch_eg+0x27>
400518:  b8 01 00 00 00    mov    $0x1,%eax
40051d:  48 01 c8          add    %rcx,%rax
400520:  c3                retq
400521:  b8 01 00 00 00    mov    $0x1,%eax
400526:  48 29 d0          sub    %rdx,%rax
400529:  c3                retq
40052a:  b8 02 00 00 00    mov    $0x2,%eax
40052f:  c3                retq
```

Question

ex

- **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;
}
```