WELLESLEY

x86 Control Flow

(**Part A**, Part B)

Condition codes, comparisons, and tests

[Un]Conditional jumps and conditional moves

Translating if-else, loops, and switch statements

Motivation

Recall: instruction memory is a flat list (with the program counter as index)!

Instruction pointer

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

register holds address of next instruction to execute

Address	Contents
0x…	<instruction>
0x…	<instruction>
0x…	<instruction>
0x…	<instruction>
0x…	<instruction>
…	

We don’t get to keep

```
if/while/for/break/continue
```

Conditionals and Control Flow

To implement familiar C **constructs**

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

Similar to `BEQ`, `JMP` in HW ISA (but more options)

Two **key pieces**

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

Most real-world ISAs use similar designs!

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

`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?*

If we calculate "testq %rax, %rax" then check the "ZF" flag, what does the result indicate?

0

`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
```

If ZF = 1, then %rax is negative, otherwise… **(A)**

If ZF = 1, then %rax is positive, otherwise … **(B)**

If ZF = 1, then %rax is zero, otherwise it is … **(C)**

If ZF = 1, then %rax is nonzero, otherwise … **(D)**

None of the above **(E)**

2. Transfer control: choose next instruction

Different **jump/branch instructions** to different part of code by setting `%rip`.

Like `BEQ`, `JMP`

Unconditional jump

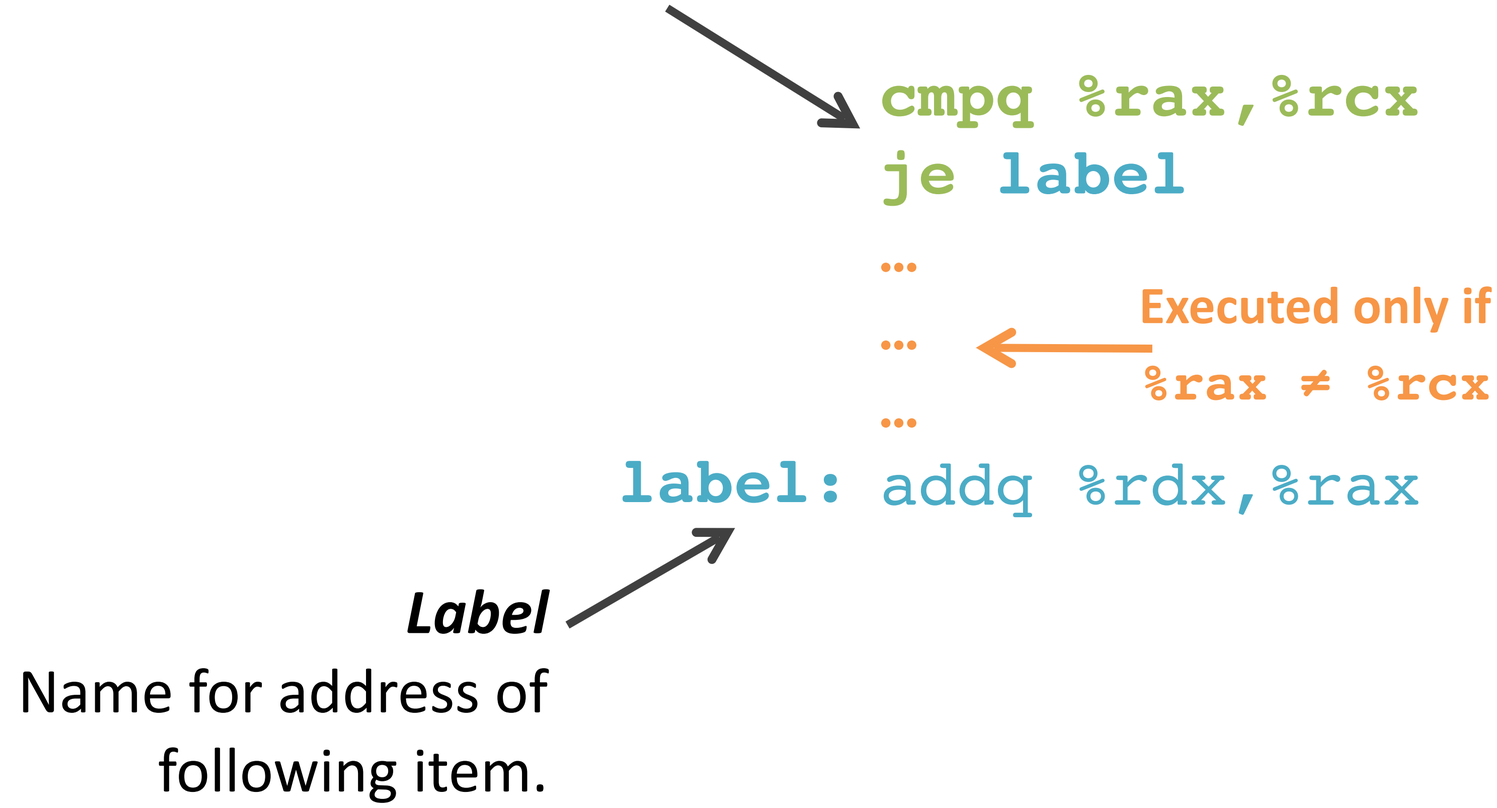
Like `BEQ`, but for more than just "equal"

Conditional jumps

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** `%rcx == %rax` **then jump** to `label`."



Interpreting Conditional Jumps

It is easier to read conditional jumps in x86-64 by comparing b against a instead of looking at condition codes.

		cmp b,a	test b,a
je	"Equal"	a == b	a&b == 0
jne	"Not equal"	a != b	a&b != 0
js	"Sign" (negative)	a-b < 0	a&b < 0
jns	(non-negative)	a-b >= 0	a&b >= 0
jg	"Greater"	a > b	a&b > 0
jge	"Greater or equal"	a >= b	a&b >= 0
jl	"Less"	a < b	a&b < 0
jle	"Less or equal"	a <= b	a&b <= 0
ja	"Above" (unsigned >)	a > b	a&b > 0U
jb	"Below" (unsigned <)	a < b	a&b < 0U

```
    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
```

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

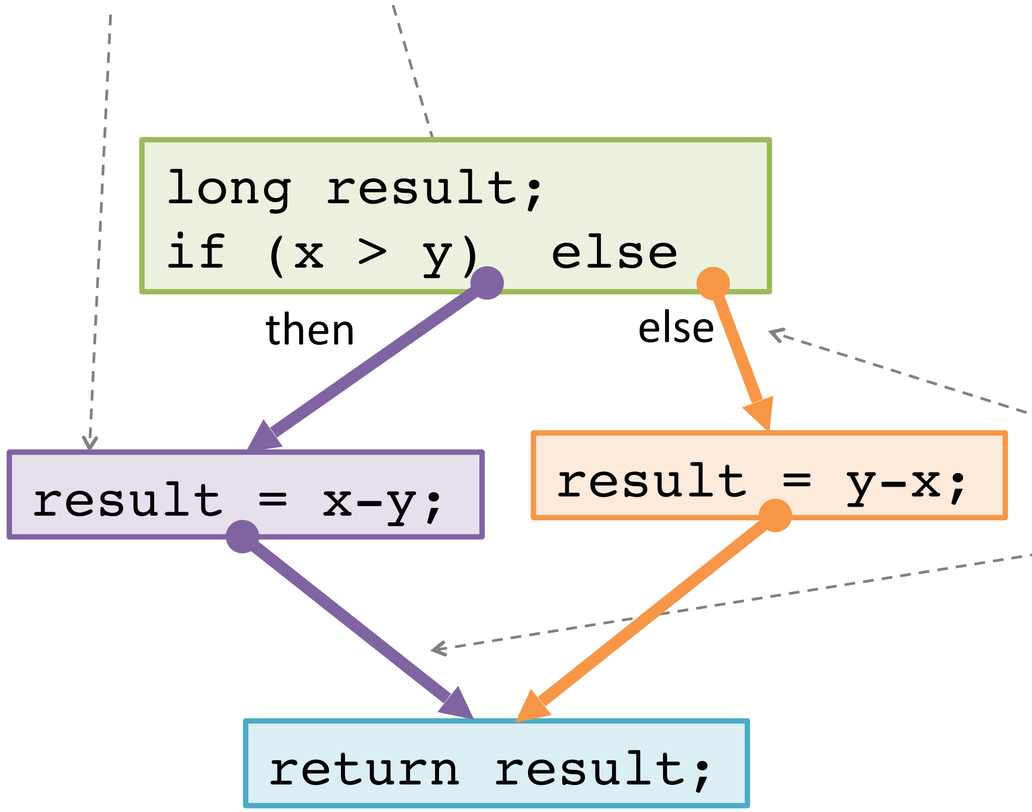
Code flowchart/directed graph.

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

```
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.

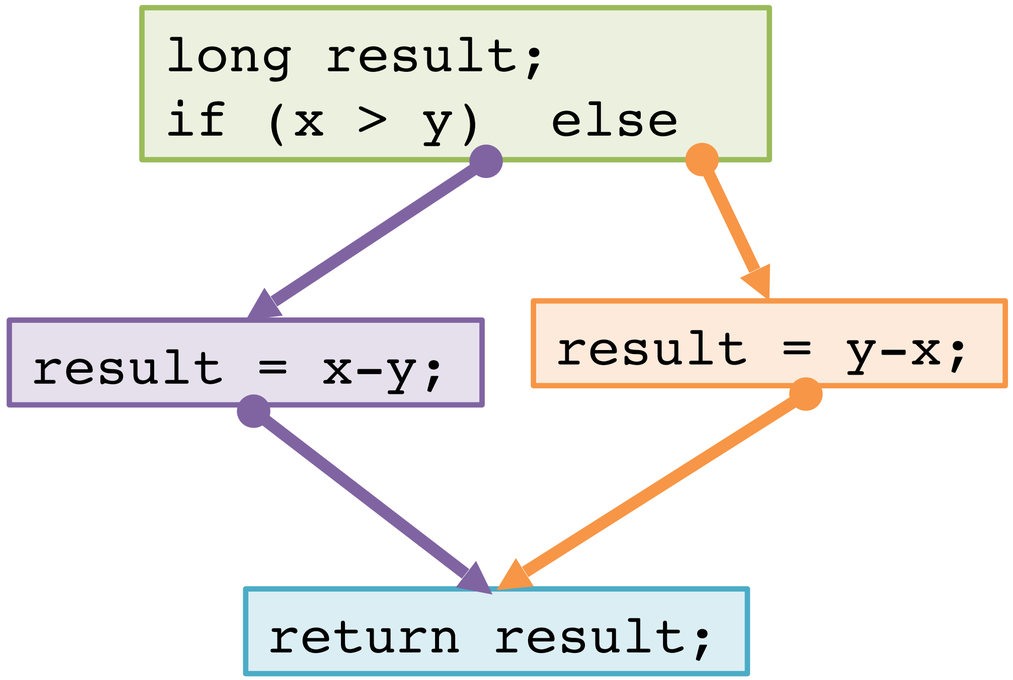


Edges = ***Control Flow***:

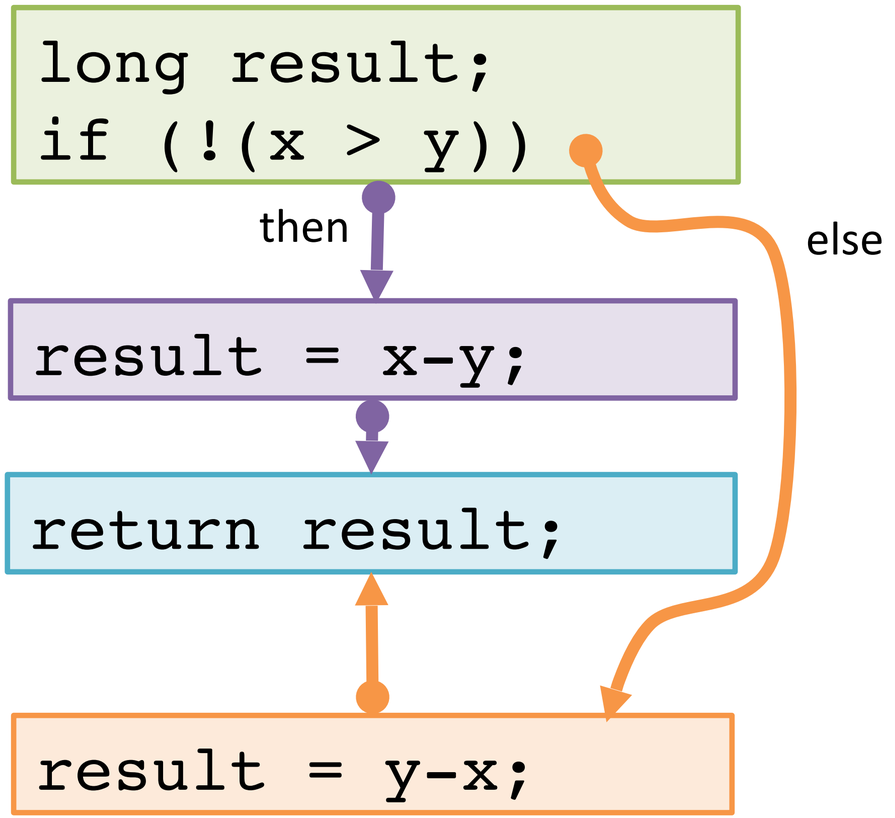
Which basic block executes
next (under what condition).

Control-Flow Graph

How do we represent this non-flat structure in a single instruction memory?



Choose a linear order of basic blocks.

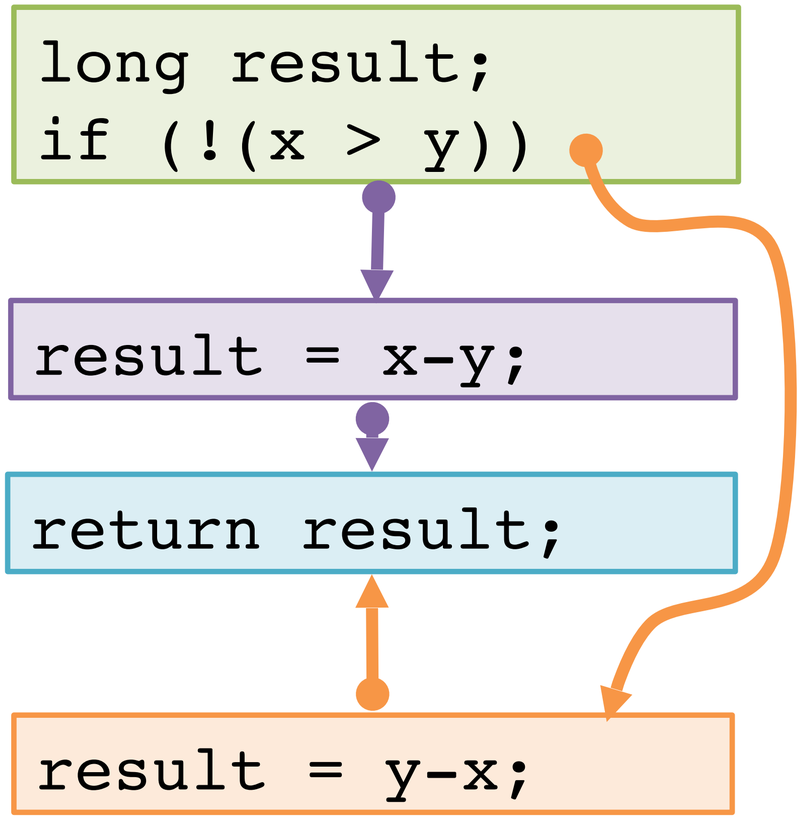


There are many different linear orders!

In CS240, we'll accept any one that "works"

The compiler tries to choose the "best" one

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
```

Insert label if incoming edge from a block other than the block above

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	5
%rsi	3

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	2
%rdi	~~5~~ 2
%rsi	3

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	2
%rdi	~~5~~ 2
%rsi	3

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	4
%rsi	7

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	3
%rdi	4
%rsi	~~7~~ 3

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	3
%rdi	4
%rsi	~~7~~ 3

Compile if-else

ex

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

```
wacky:
```

Recall:

`x` is available in `%rdi`

`y` is available in `%rsi`

Calculate `result` in `%rax` for return

Instructions to use (one or more of)

`movq, addq, cmpq, jg or jle, leaq`

Hint: be careful not to overwrite `x, y`!

Compile if-else

ex

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

Recall:

- `x` is available in `%rdi`
- `y` is available in `%rsi`
- Calculate `result` in `%rax` for return

Instructions to use (one or more of)

`movq, addq, cmpq, jg or jle, leaq`

Hint: be careful not to overwrite `x, y`!

`wacky:` Incomplete first attempt

```
  addq %rdi, %rsi
  cmpq $7, %rsi
  jle Else

Then:


Else:
```

Oops, we overwrote y!
Now can't compute y + 2

Group task: fix/complete this code by filling in x86 for each block

Compile if-else (solution #1)

ex

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

Recall:

- `x` is available in `%rdi`
- `y` is available in `%rsi`
- Calculate `result` in `%rax` for return

```
wacky:
  movq %rdi, %rdx
  addq %rsi, %rdx
  cmpq $7, %rdx
  jle Else

  movq %rdi, %rax

 End:
  retq

 Else:
  addq $2, %rsi
  movq %rsi, %rax
  jmp End
```

Compile if-else (solution #2) - `leaq`

ex

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

```
wacky:
  leaq (%rdi, %rsi), %rdx
  cmpq $7, %rdx
  jle Else

  movq %rdi, %rax

 End:
  retq

 Else:
  leaq 2(%rsi), %rax
  jmp End
```

Recall:

- `x` is available in `%rdi`
- `y` is available in `%rsi`
- Calculate `result` in `%rax` for return

x86 Control Flow

(Part A, **Part B**)

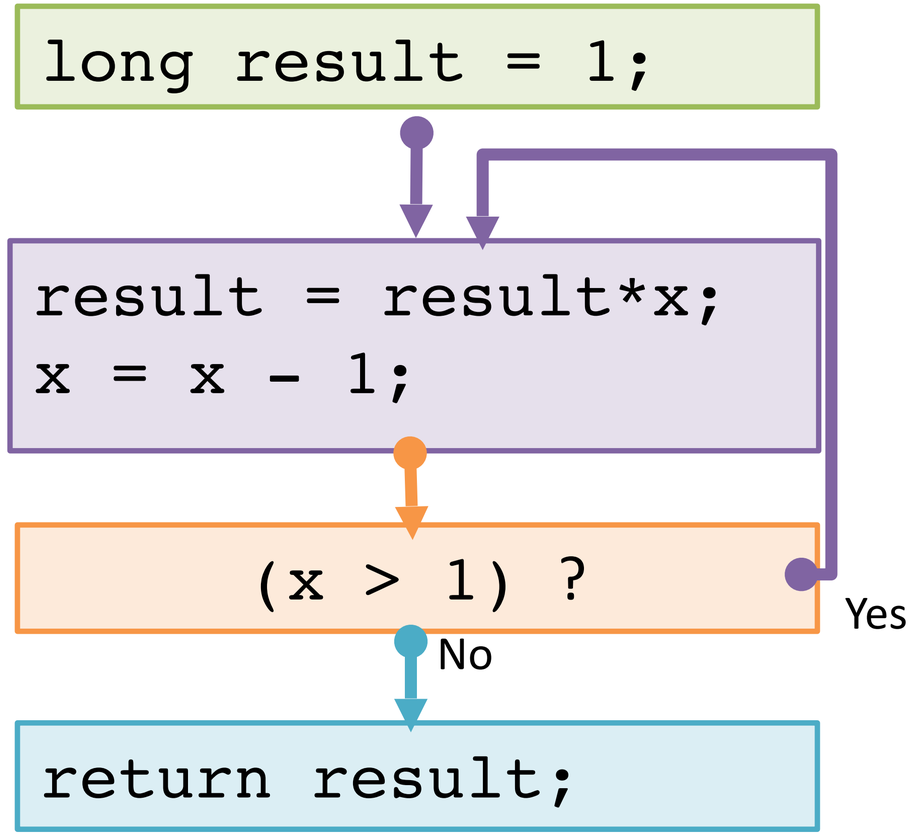
Condition codes, comparisons, and tests

[Un]Conditional jumps and **conditional moves**

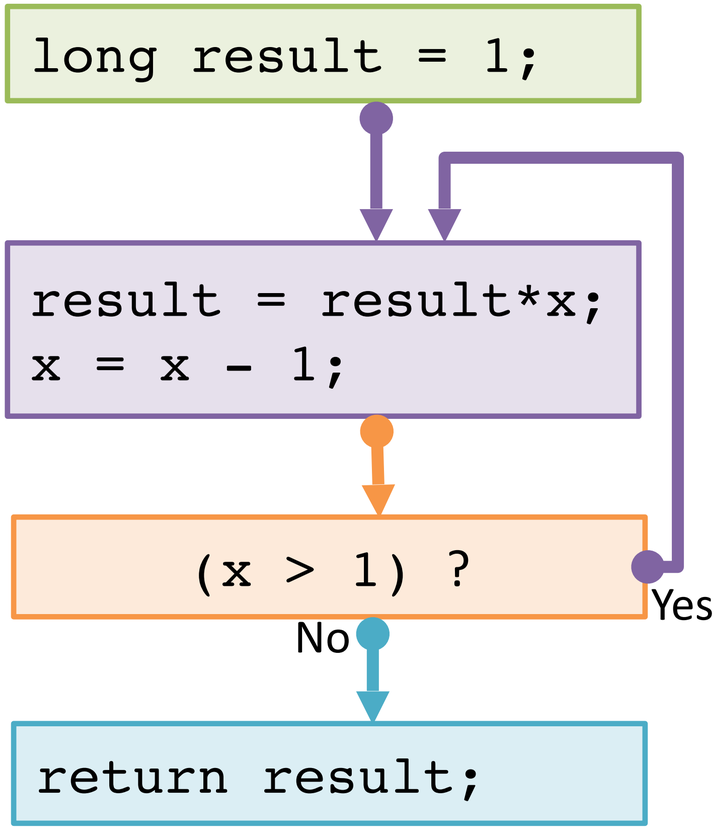
Translating if-else, **loops, and switch statements**

do while loop

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



do while loop

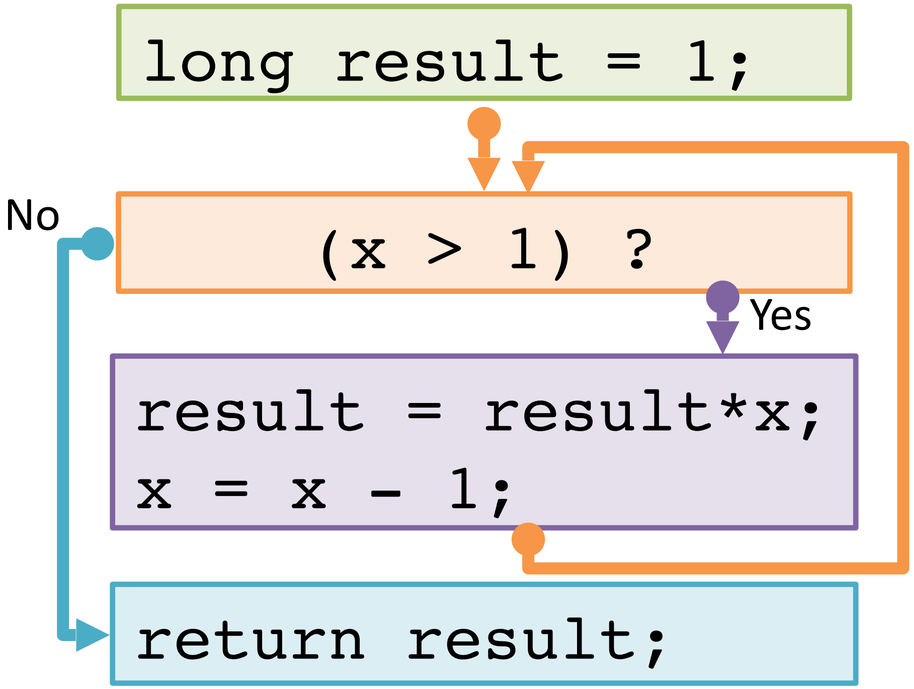


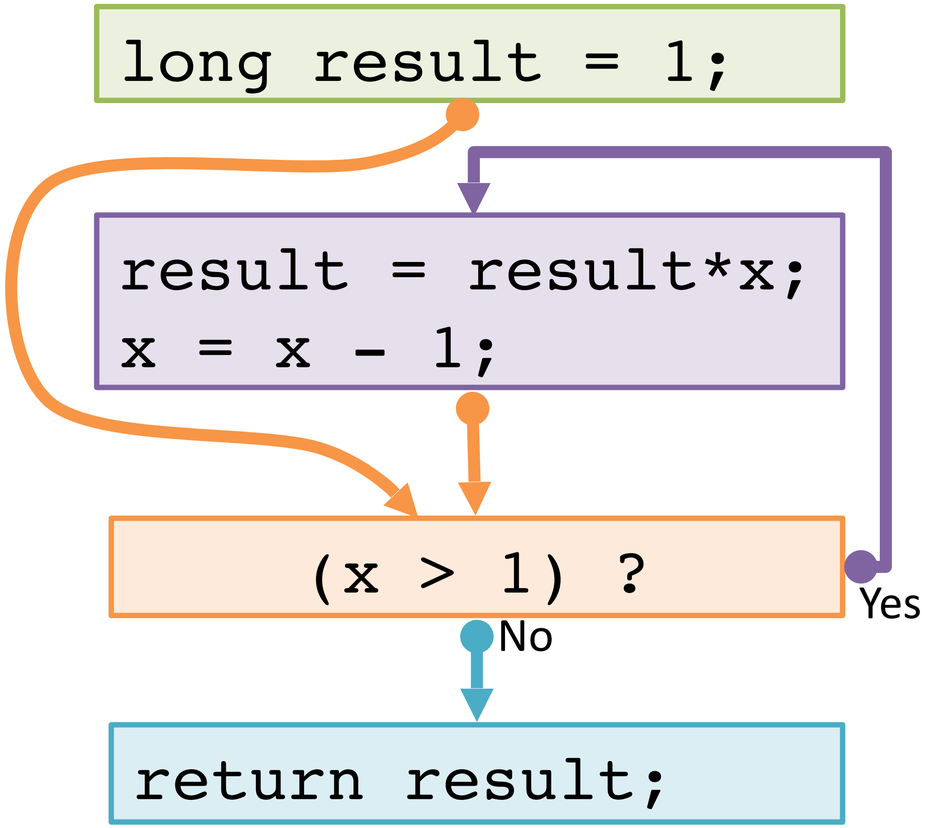

```
fact_do:
  movq $1,%rax

.L11:
  imulq %rdi,%rax
  decq %rdi


  cmpq $1,%rdi
  jg .L11

  retq
```

Why put the loop condition at the end?

while loop

```
long fact_while(long x){
  // Assume >= 0
  long result = 1;
  while (x > 1) {
    result = result * x;
    x = x - 1;
  }
  return result;
}
```

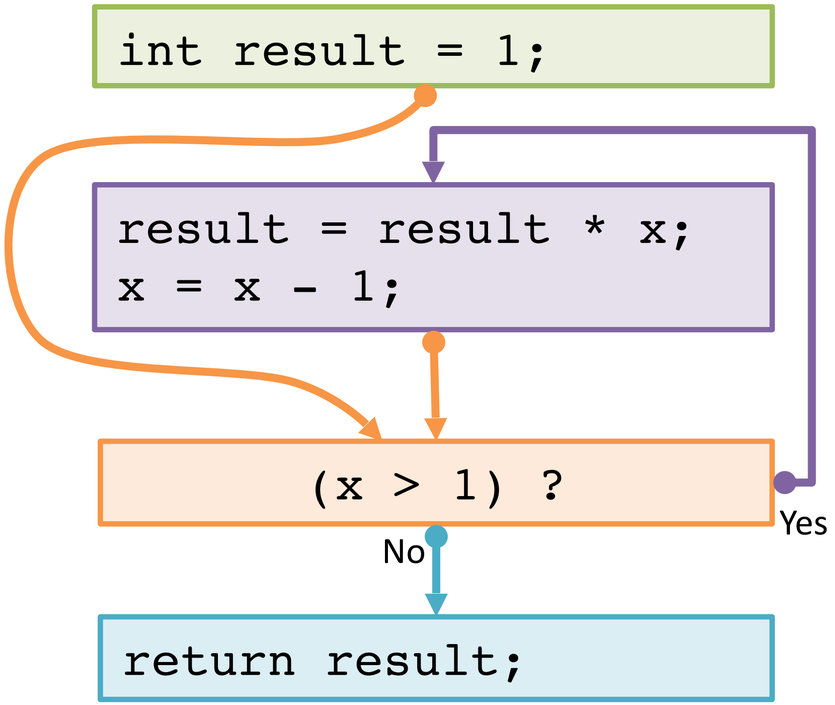




This order is used by GCC for x86-64. Why?

while loop

```
long fact_while(long x){
  // Assume >= 0
  long result = 1;
  while (x > 1) {
    result = result * x;
    x = x - 1;
  }
  return result;
}
```



Full x86-64:

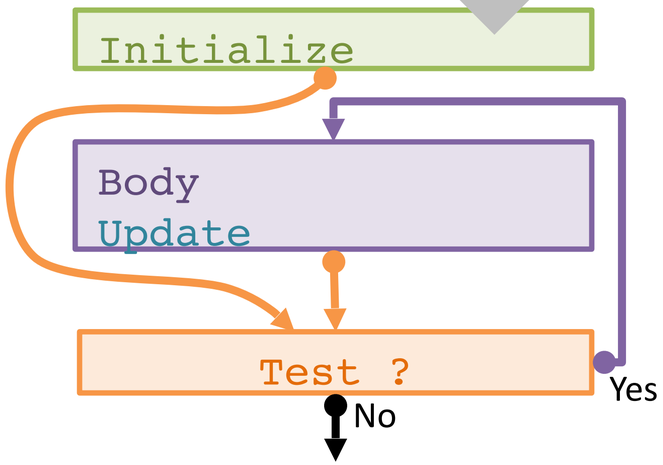
```
fact_while:
  movq $1, %rax
  jmp   .L34

.L35:
  imulq %rdi, %rax
  decq  %rdi

.L34:
  cmpq  $1, %rdi
  jg    .L35

  retq
```

`for` loop translation

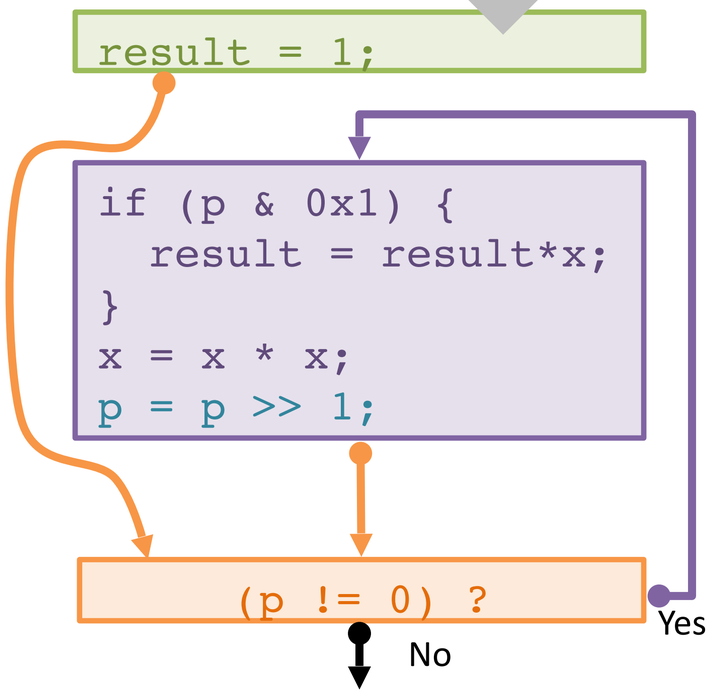
`for` loops are *syntactic sugar* for while loops: we can just translate `for` to `while`

```
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;
}
```



`for` loop: square-and-multiply

optional

```
/* Compute x raised to nonnegative power p */
int power(int x, unsigned int p) {
   int result;
   for (result = 1; p != 0; p = p>>1) {
      if (p & 0x1) {
         result = result * x;
      }
      x = x*x;
   }
   return result;
}
```

$x^m * x^n = x^{m+n}$

0 … 0 1 0 1 1 = 11

$1^{2\text{^}31} * \ldots * 1^{16} * x^8 * 1^4 * x^2 * x^1 = x^{11}$

$1 = x^0$ $x = x^1$

Algorithm

Exploit bit representation: $p = p_0 + 2p_1 + 2^2p_2 + \ldots 2^{n-1}p_{n-1}$

Gives: $x^p = z_0 \cdot z_1^{\,2} \cdot (z_2^{\,2})^{\,2} \cdot \ldots \cdot \underbrace{(\ldots((z_{n-1}^{\,2})^{\,2})\ldots)^{\,2}}_{n-1 \text{ times}}$

$z_i = 1$ when $p_i = 0$

$z_i = x$ when $p_i = 1$

Complexity $O(\log p) = O(\text{sizeof}(p))$

Example

$3^{11} = 3^1 * 3^2 * 3^8$

$= 3^1 * 3^2 * ((3^2)^2)^2$

`for` loop: `power` iterations

optional

```
/* Compute x raised to nonnegative power p */
int power(int x, unsigned int p) {
   int result;
   for (result = 1; p != 0; p = p>>1) {
      if (p & 0x1) {
         result = result * x;
      }
      x = x*x;
   }
   return result;
}
```

iteration	result	x	p
0	1	3	$11 = 1011_2$
1	3	9	$5 = 101_2$
2	27	81	$2 = 10_2$
3	27	6561	$1 = 1_2$
4	177147	430467	0_2

(Aside) **Conditional Move**

`cmov_ src, dest`

if (***Test***) ***Dest*** ← ***Src***

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

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

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

Why? Branch prediction in pipelined/OoO processors.

(Aside) **Bad uses of conditional move**

Expensive Computations

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

Risky Computations

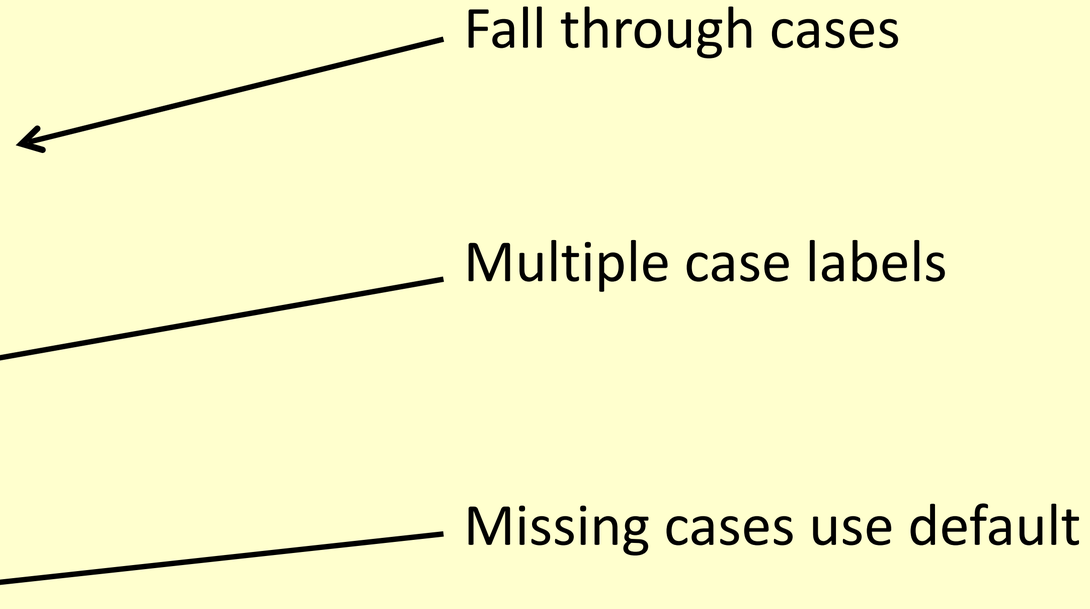
```
val = p ? *p : 0;
```

Computations with side effects

```
val = x > 0 ? x++ : x--;
```

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;
}
```



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

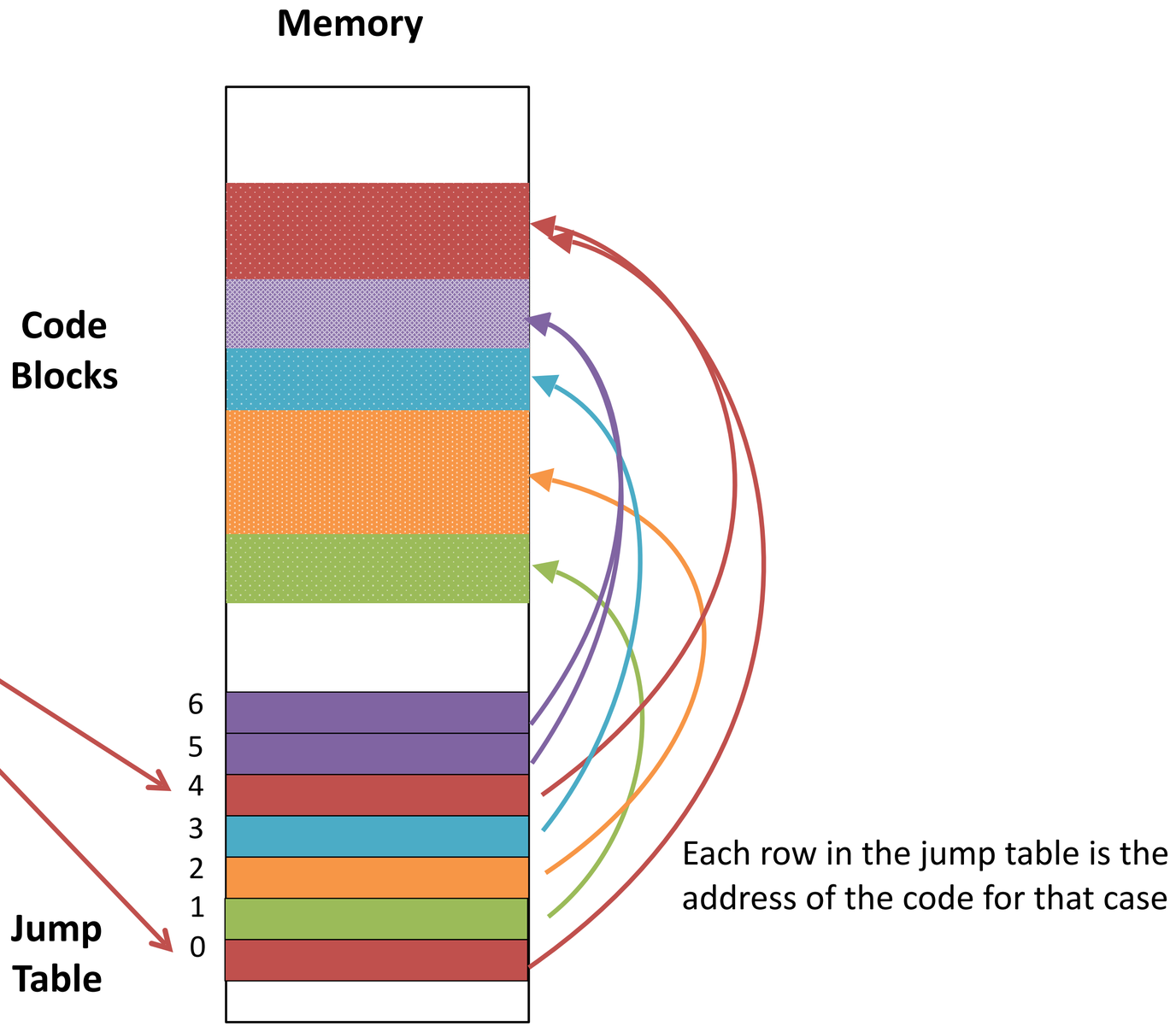
`switch` jump table structure

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>
}
```

Translation sketch:

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



`switch` jump table assembly declaration

read-only ***data***
(not instructions)

8-byte alignment

```
.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" = `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 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
```

Jump if above *(unsigned, but…)*

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

indirect jump

switch cases

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

```
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;
```

```
.L3:    movq    %rsi, %rax
        imulq   %rdx, %rax
        retq                    <-- "inlined" return
.L5:    movq    %rsi, %rax
        subq    %rdx, %rax
.L9:                            <-- Fall-through
        addq    %rdx, %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 *fewer bytes* to encode a literal `movl` vs a `movq`.

`switch` machine code

Assembly Code

```
switch_eg:
  . . .
  cmpq   $6, %rdi
  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)
```

When looking as disassembled code: an indirect jump like this is a sign it's a jump table encoding a switch

Inspect jump table contents using GDB.

Examine contents as 7 addresses

Address of code for case 0

Address of code for case 1

```
(gdb) x/7a 0x4005d0
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>
```

Address of code for case 6

Would you implement this with a jump table?

ex

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

Would it be a good idea to implement this switch statement with a jump table?

Yes

No

Maybe