# Control flow

Condition codes

Conditional and unconditional jumps

Loops

Switch statements

# Conditionals and Control Flow

## Familiar C constructs

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

## Two key pieces

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

# 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 |

`%eip`

## Instruction pointer

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

register holds address of next instruction to execute

# 1. ***compare*** and ***test***: conditions

ex

**`cmpl b,a`** computes **`a - b,`** sets flags, discards result

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

**`testl b,a`** computes **`a & b,`** sets flags, discards result

Common pattern:

```
testl %eax, %eax
```

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

# **Aside:** save conditions

***setg:*** set if greater

stores byte:

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

0x00 otherwise

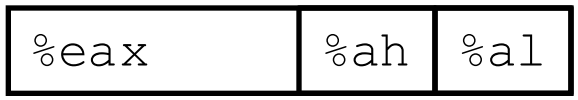
```
int gt (int x, int y) {
  return x > y;
}
```

```
movl 12(%ebp),%eax   # eax = y

cmpl %eax,8(%ebp)    # compare: x - y

setg %al             # al = x > y

movzbl %al,%eax      # zero rest of %eax
```




**Z**ero-extend from **B**yte (8 bits) to **L**ongword (32 bits)

# 2. *jump*: choose next instruction

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

Always jump

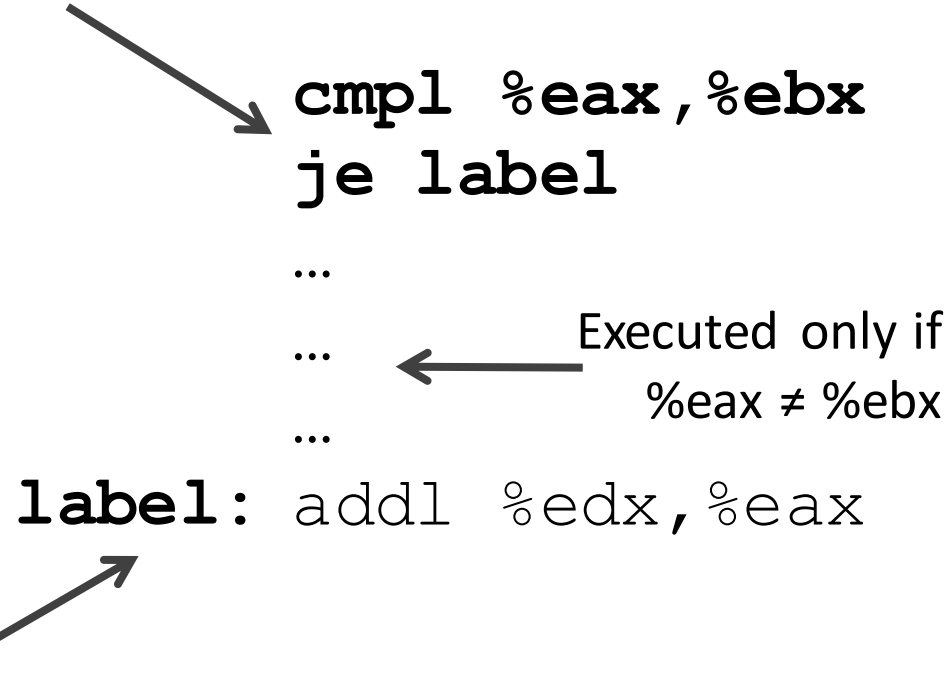
Jump iff *condition*

| jX | 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 %eax = %ebx , jump to *label.*"




***Label***

Name for address of following instruction.

# Conditional Branch Example

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

```
absdiff:
    pushl   %ebp                    } Setup
    movl    %esp, %ebp
    movl    8(%ebp), %edx           } Body
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L7
    subl    %eax, %edx
    movl    %edx, %eax
.L8:
    leave                           } Finish
    ret
.L7:
    subl    %edx, %eax              } Body
    jmp     .L8
```

***Labels***

Name for address of following instruction.

**How did the compiler create this?**

# Control-Flow Graph

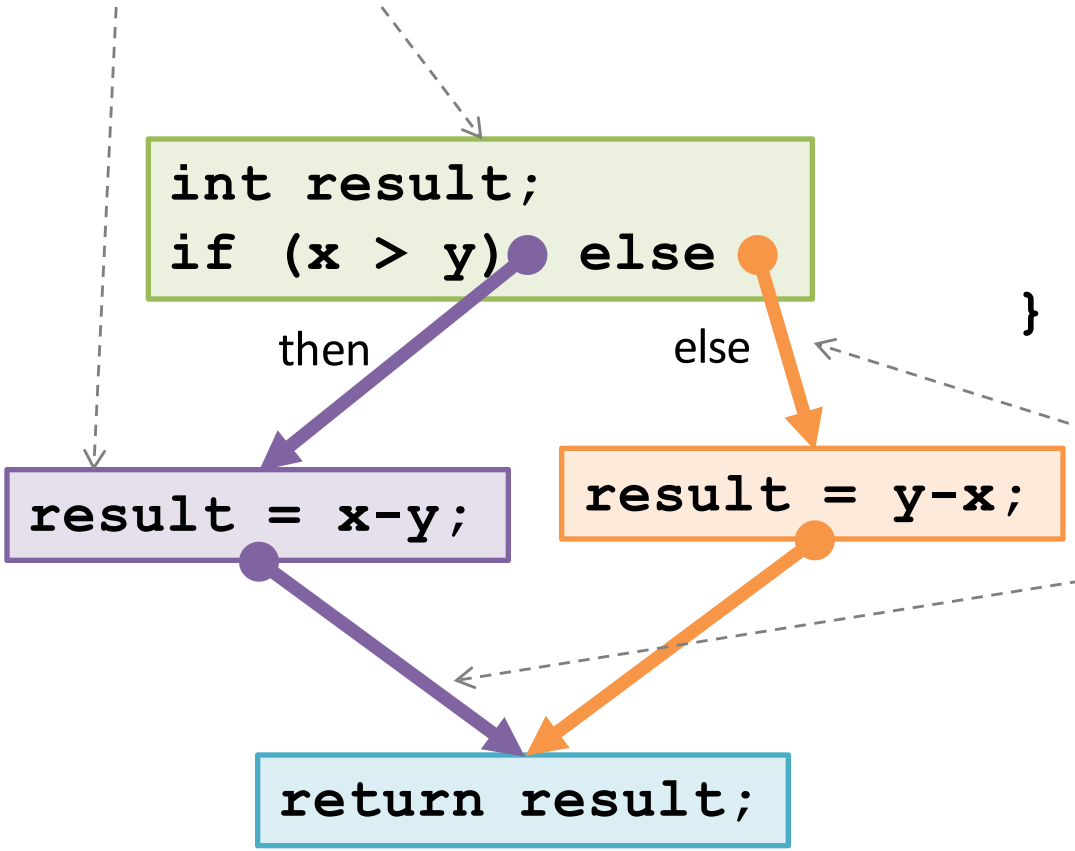
Code flowchart/directed graph.

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.

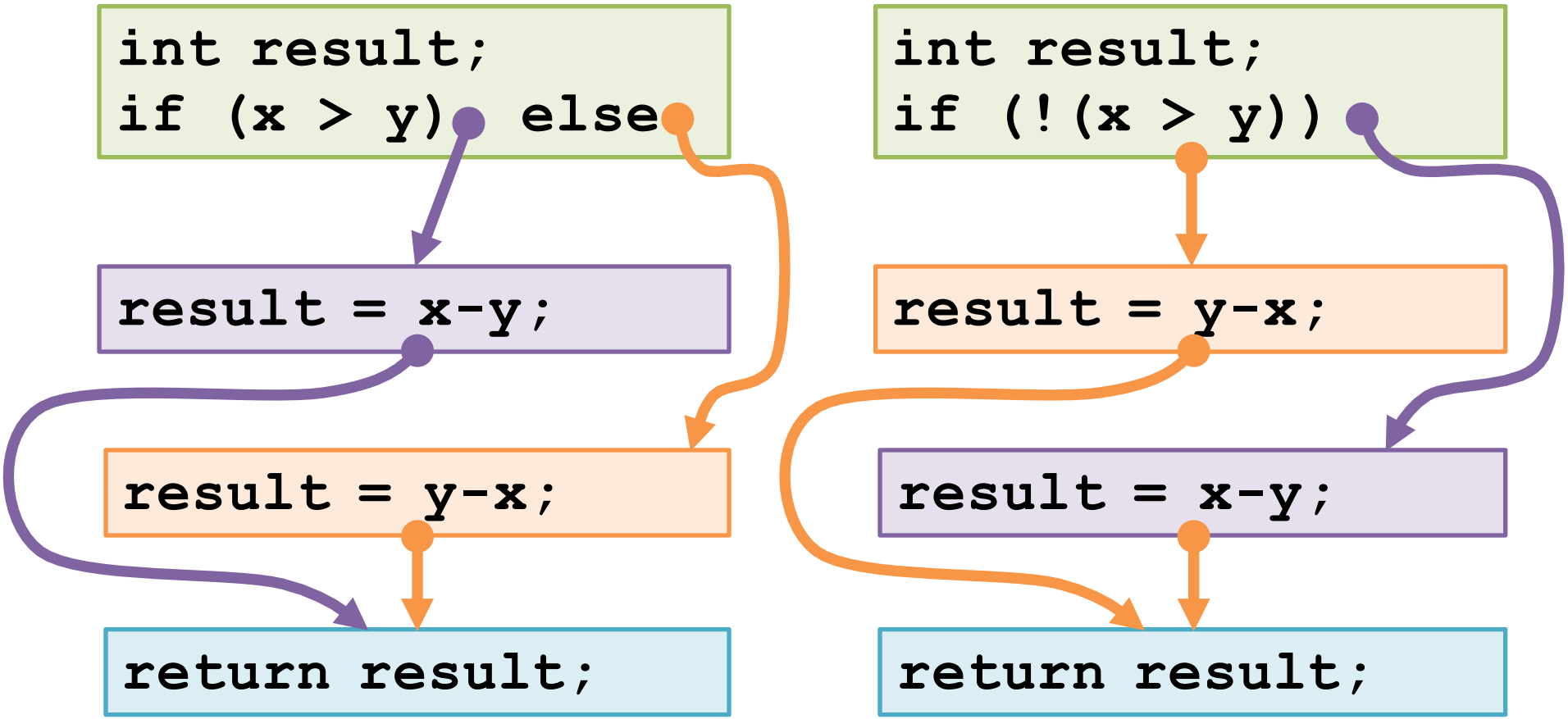
```
int absdiff(int x, int y) {
  int result;
  if (x > y) {
    result = x-y;
  } else {
    result = y-x;
  }
  return result;
}
```



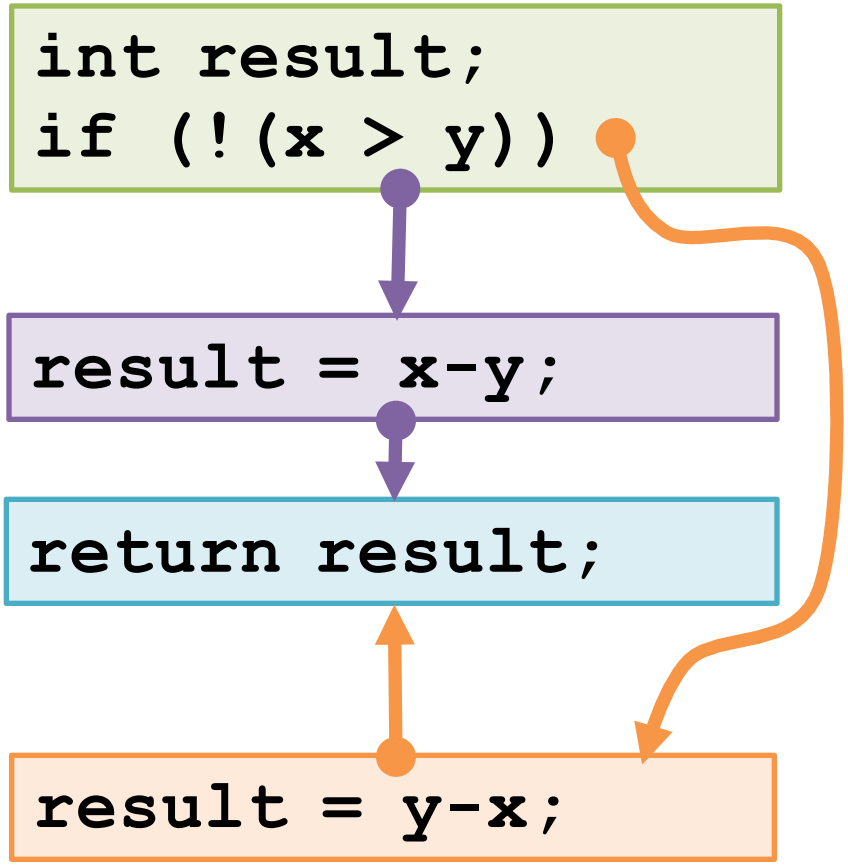

Edges = ***Control Flow***:

Which basic block executes next (under what condition).

# Choose a linear order of basic blocks.

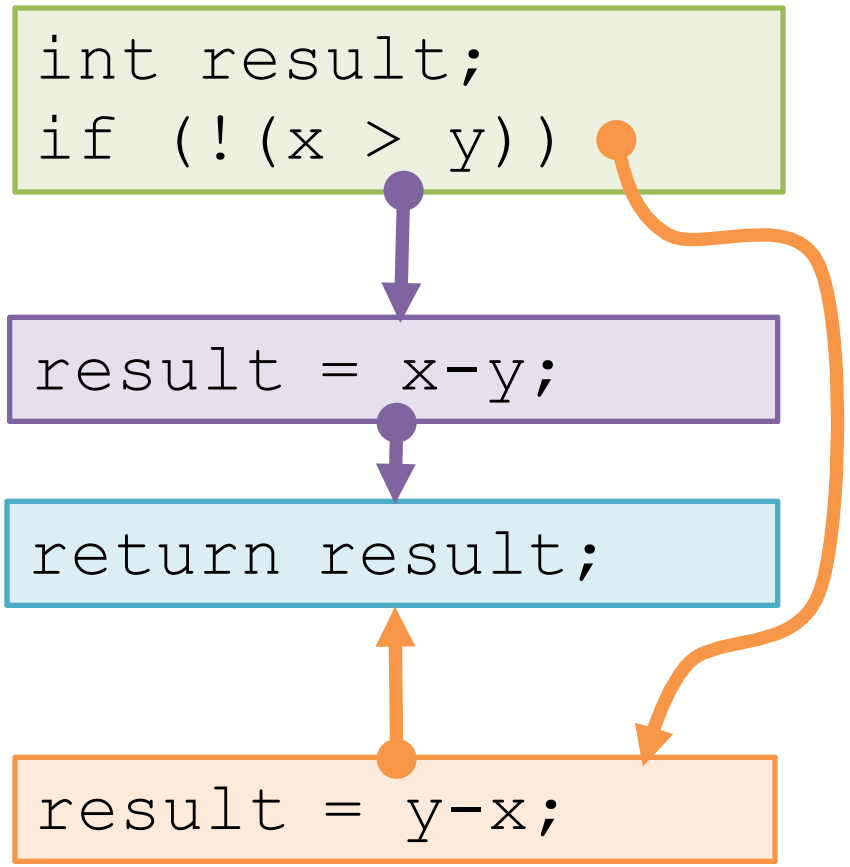


# Choose a linear order of basic blocks.




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

# Translate basic blocks with jumps + labels




```
pushl   %ebp
movl    %esp, %ebp
movl    8(%ebp), %edx
movl    12(%ebp), %eax
cmpl    %eax, %edx
jle Else
```

```
subl    %eax, %edx
movl    %edx, %eax
```

End:

```
leave
ret
```

Else:

```
subl    %edx, %eax
jmp End
```

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

# Execute absdiff

ex

```
pushl   %ebp
movl    %esp, %ebp
movl    8(%ebp), %edx
movl    12(%ebp), %eax
cmpl    %eax, %edx
jle Else
```

Start here.

```
subl    %eax, %edx
movl    %edx, %eax
```

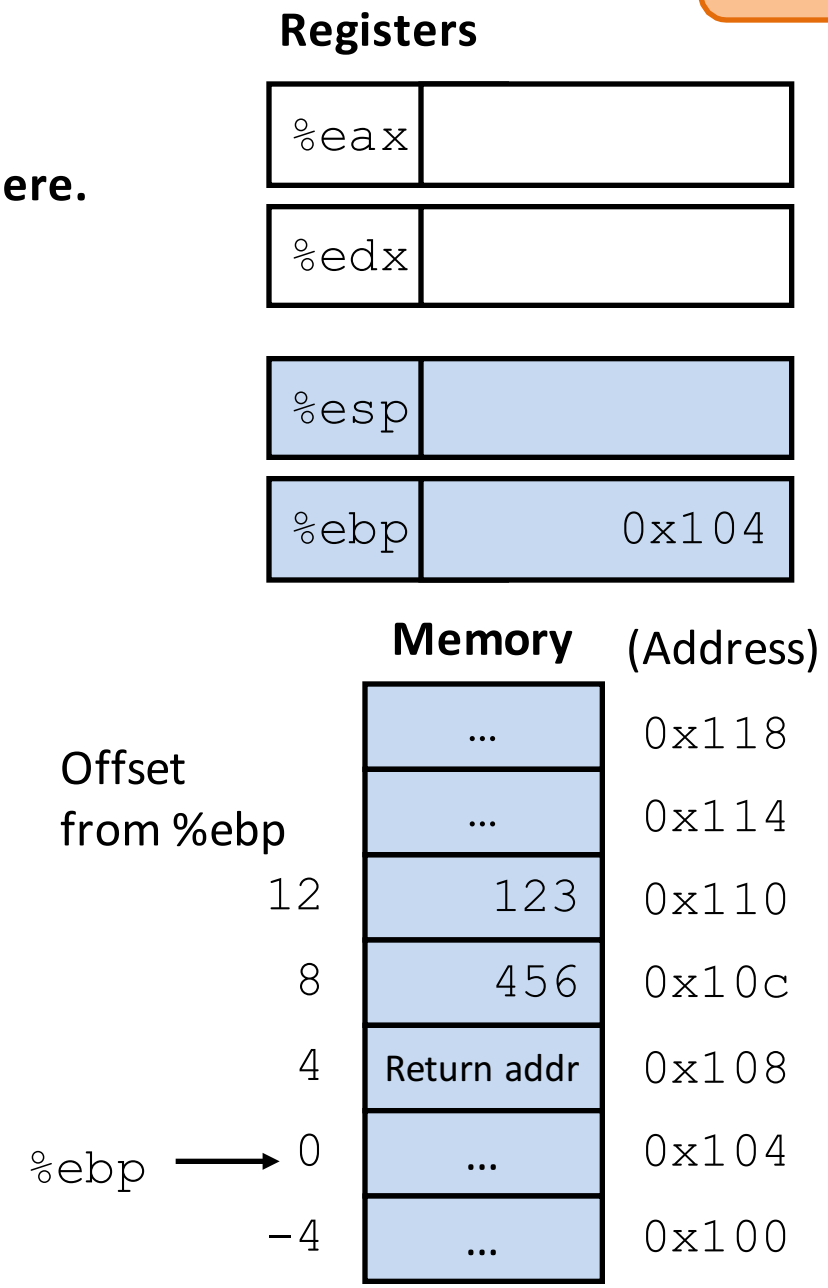
End:

```
leave
ret
```

Stop here.
What is in %eax?

Else:

```
subl    %edx, %eax
jmp End
```



# **Note:** CSAPP shows translation with **goto**

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

```
int goto_ad(int x, int 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**

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

Close to assembly code.

```
absdiff:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L7
    subl    %eax, %edx
    movl    %edx, %eax
.L8:
    leave
    ret
.L7:
    subl    %edx, %eax
    jmp     .L8
```

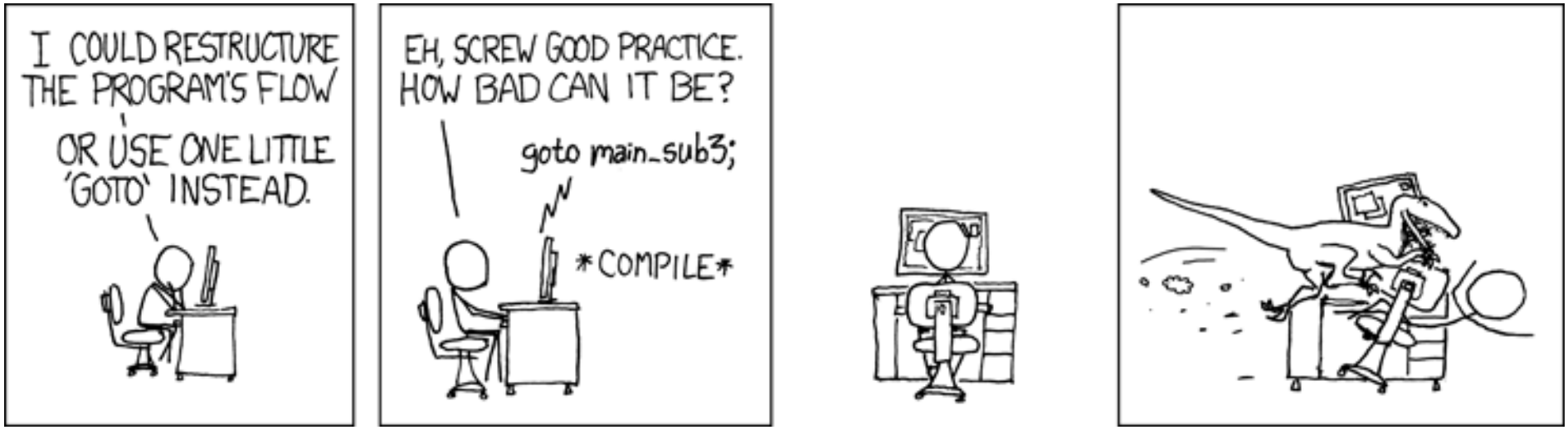
Setup

Body

Finish

Body

# But never use goto in your source code!




http://xkcd.com/292/

# compile if-else

ex

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

Assume x available in 8(%ebp),
y available in 12(%ebp).

Place result in %eax.

# PC-relative addressing

```
0x100      cmp   %eax, %ebx      0x1000
0x102      je    0x70            0x1002
0x104      …                     0x1004
…          …                     …
0x174      add   %eax, %ebx      0x1074
```

- **Jump instruction encodes *offset* from next instruction to destination PC.**
  - (Not the absolute address of the destination.)
  - PC relative branches are relocatable
  - Absolute branches are not (or they take a lot work to relocate)

# PC-relative addressing

**objdump output:**

```
00000000 <absdiff>:
0:  55                      push   %ebp
1:  89 e5                   mov    %esp,%ebp
3:  83 ec 10                sub    $0x10,%esp
6:  8b 45 08                mov    0x8(%ebp),%eax
9:  3b 45 0c                cmp    0xc(%ebp),%eax
c:  7e 0b                   jle    19 <absdiff+0x19>
e:  8b 45 08                mov    0x8(%ebp),%eax
11: 2b 45 0c                sub    0xc(%ebp),%eax
14: 89 45 fc                mov    %eax,-0x4(%ebp)
17: eb 09                   jmp    22 <absdiff+0x22>
19: 8b 45 0c                mov    0xc(%ebp),%eax
1c: 2b 45 08                sub    0x8(%ebp),%eax
1f: 89 45 fc                mov    %eax,-0x4(%ebp)
22: 8b 45 fc                mov    -0x4(%ebp),%eax
25: c9                      leave
26: c3                      ret
```

**How are the jump targets encoded?** Why?

# Compiling Loops

C/Java code:

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

Machine code:

```
loopTop:    cmpl   $0, %eax
            je     loopDone
              <loop body code>
            jmp    loopTop
loopDone:
```

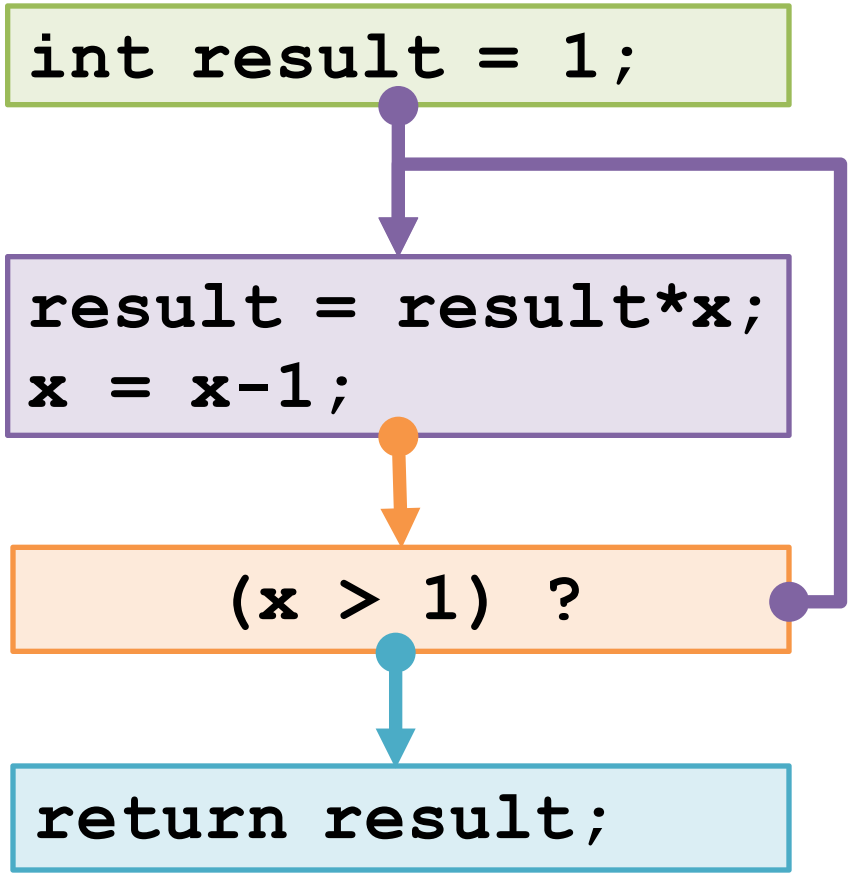
**How to compile other loops should be straightforward**

The only slightly tricky part is to where to put the conditional branch: top or bottom of the loop

# “Do-While” Loop Example

C Code

```
int fact_do(int x) {
  int 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 Example

## C Code

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

## Goto Version

```
int fact_goto(int x) {
  int result = 1;
loop:
  result = result * x;
  x = x-1;
  if (x > 1) goto loop;
  return result;
}
```

**Keys:**

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

# "Do-While" Loop Compilation

## Goto Version

```
int fact_goto(int x) {
  int result = 1;


loop:
  result = result * x;
  x = x-1;
  if (x > 1)
    goto loop;

  return result;
}
```

## Assembly

```
fact_goto:
   pushl %ebp
   movl %esp,%ebp
   movl $1,%eax
   movl 8(%ebp),%edx

.L11:
   imull %edx,%eax
   decl %edx
   cmpl $1,%edx
   jg .L11

   movl %ebp,%esp
   popl %ebp
   ret
```

| Register | Variable |
|---|---|
| %edx | |
| %eax | |

Translation?

**Why?**

Why put the loop condition at the end?

# General "Do-While" Translation

## C Code

```
do
  Body
  while (Test);
```

## Goto Version

```
loop:
  Body
  if (Test)
    goto loop
```

***Body:***

{

$Statement_1$;

$Statement_2$;

…

$Statement_n$;

}

***Test* returns integer**

= 0 interpreted as false

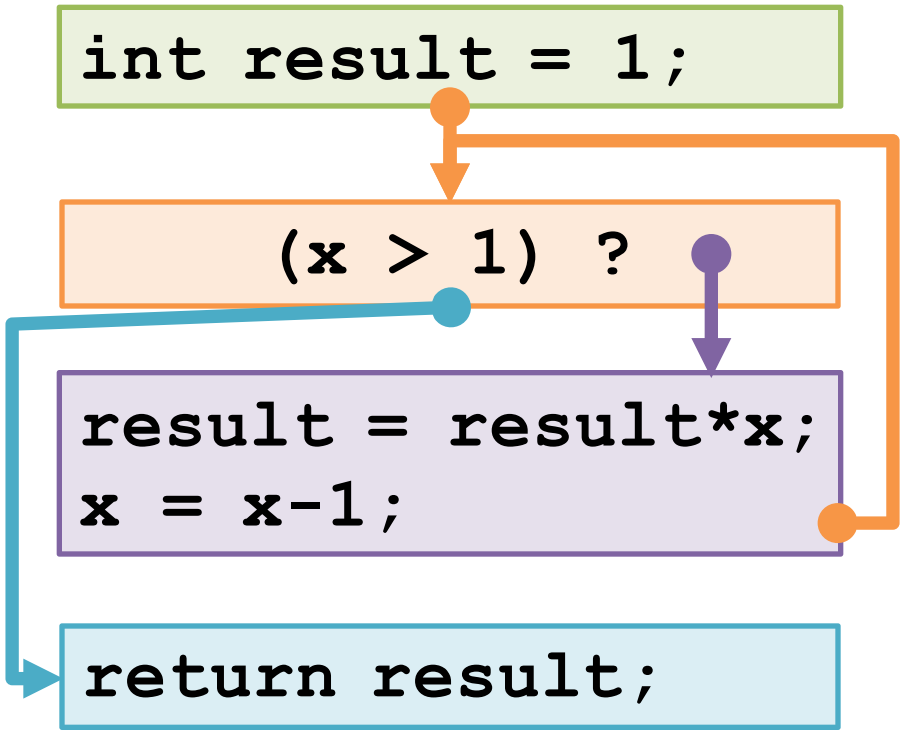
≠ 0 interpreted as true

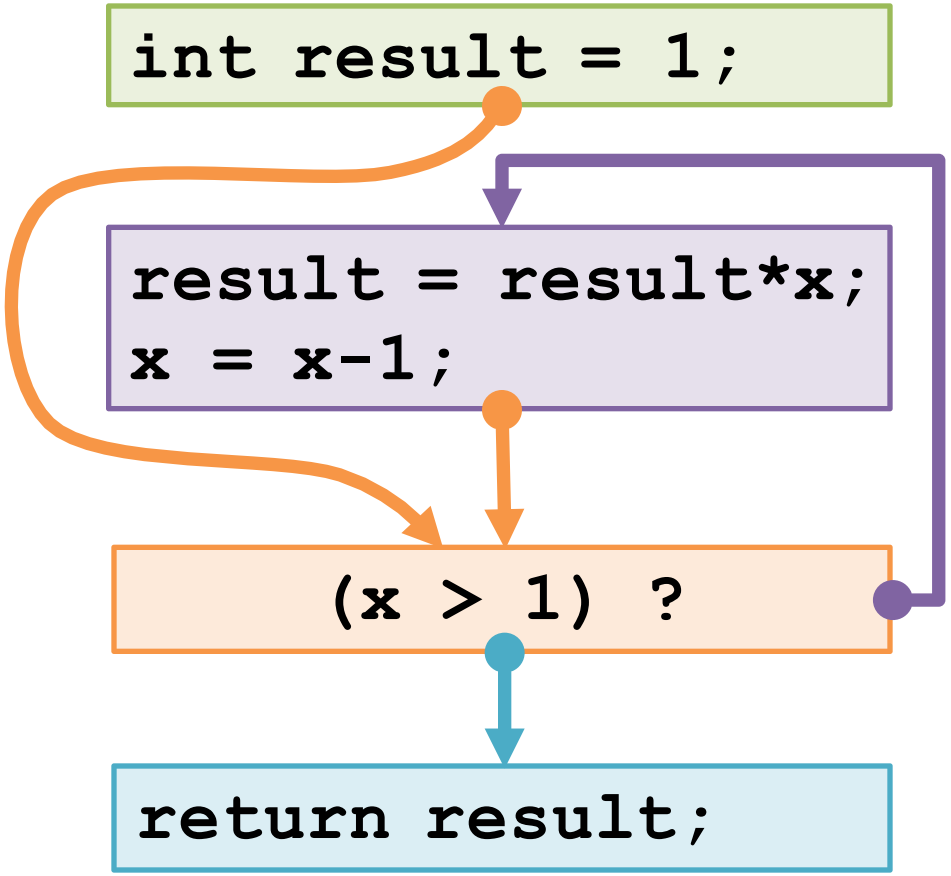
# "While" Loop Translation

Why?

C Code

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



# "While" Loop Translation

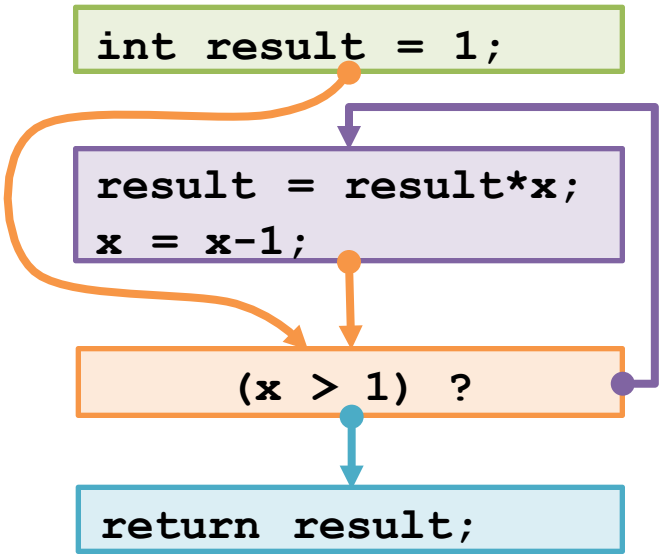
Why?

## C Code

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

## Goto Version

```
int fact_while_goto(int x) {
  int result = 1;
  goto middle;
loop:
  result = result * x;
  x = x-1;
middle:
  if (x > 1)
    goto loop;
  return result;
}
```



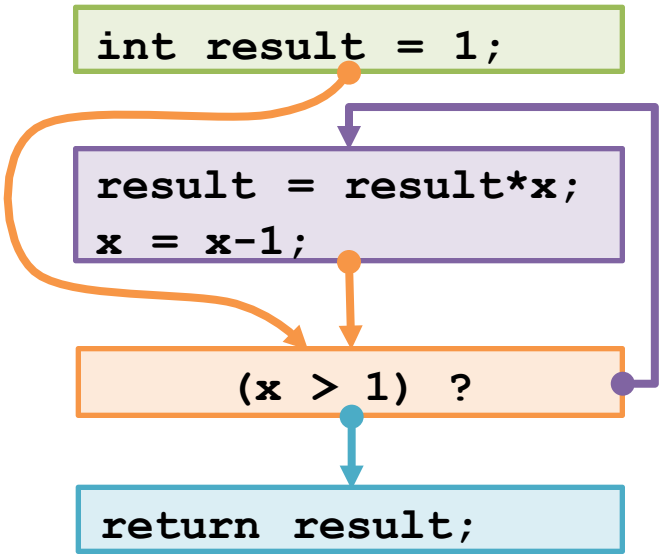

**This order is used by GCC for both IA32** and x86-64

**Test at end, first iteration jumps over body to test.**

# "While" Loop Example

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

```
# x in %edx, result in %eax
  jmp   .L34        #   goto Middle
.L35:               # Loop:
  imull %edx, %eax  #   result *= x
  decl  %edx        #   x--
.L34:               # Middle:
  cmpl  $1, %edx    #   x:1
  jg    .L35        #   if >, goto
                    #           Loop
```



[optional]

# "For" Loop Example: Square-and-Multiply

```
/* 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 1 0 1 = 13

$1^{2\text{^}31} * \ldots * 1^{16} * x^8 * x^4 * 1^2 * x^1 = x^{13}$

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

## Algorithm

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

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

$z_i = 1$ when $p_i = 0$

$z_i = x$ when $p_i = 1$

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

Example

$3^{10} = 3^2 * 3^8$

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

# `power` Computation

```
/* 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;
}
```

| before iteration | result | x=3 | p=10 |
|---|---|---|---|
| 1 | 1 | 3 | $10=1010_2$ |
| 2 | 1 | 9 | $5=\ 101_2$ |
| 3 | 9 | 81 | $2=\ \ 10_2$ |
| 4 | 9 | 6561 | $1=\ \ \ 1_2$ |
| 5 | 59049 | 43046721 | $0_2$ |

# "For" Loop Example

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

General Form

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

*Init*

```
result = 1
```

*Test*

```
p != 0
```

*Update*

```
p = p >> 1
```

*Body*

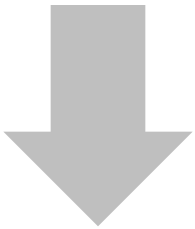
```
{
  if (p & 0x1) {
    result = result*x;
  }
  x = x*x;
}
```

# "For" → "While"

derive

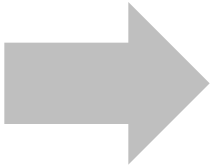
For Version

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



While Version

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



Goto Version

```
   Initialize;
   goto middle;
loop:
   Body
   Update ;
middle:
   if (Test)
     goto loop;
done:
```

# For-Loop: Compilation

derive

For Version

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

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

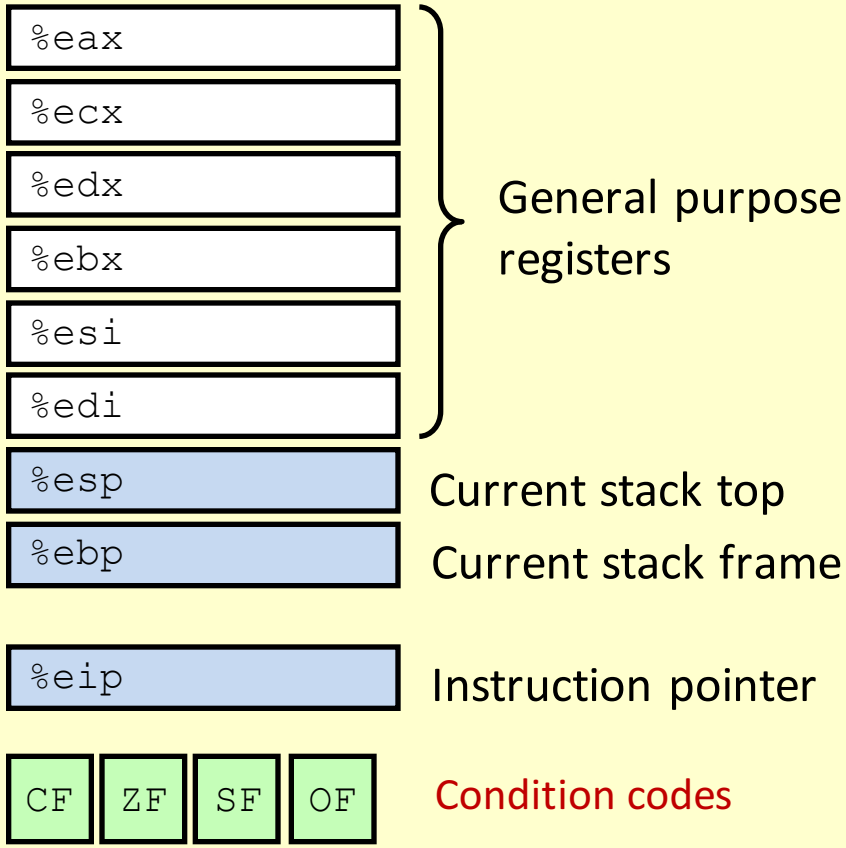
Goto Version

```
  Initialize;
  goto middle;
loop:
  Body
  Update ;
middle:
  if (Test)
    goto loop;
done:
```

```
  result = 1;
  goto middle;
loop:
  if (p & 0x1)
    result *= x;
  x = x*x;
  p = p >> 1;
middle:
  if (p != 0)
    goto loop;
done:
```

# Review

**Processor State**




**Memory addressing modes**

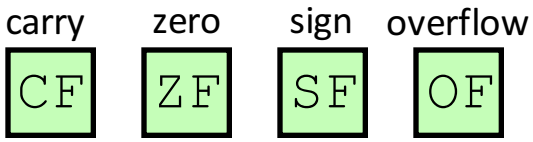
```
(%eax)
17(%eax)
12(%edx, %eax)
2(%ebx, %ecx, 8)
```

**Immediate (constant), Register, and Memory Operands**

```
subl %eax, %ecx              # ecx = ecx + eax
sall $4,%edx                 # edx = edx << 4
addl 16(%ebp),%ecx           # ecx = ecx + Mem[16+ebp]
imull %ecx,%eax              # eax = eax * ecx
```

# Review

## Control




1-bit condition code/flag registers

Set by arithmetic instructions (`addl`, `shll`, etc.), `cmp`, `test`

Access flags with `setg`, `setle`, … instructions

**Conditional jumps** use flags for decisions (`jle .L4`, `je .L10`, …)

**Unconditional jumps** always jump: `jmp`

Direct or indirect jumps

## Standard Techniques

Loops converted to do-while form

Large switch statements use jump tables

# Review

## Do-While loop

C Code

```
do
  Body
  while (Test);
```

Goto Version

```
loop:
  Body
  if (Test)
    goto loop
```

## While-Do loop

While version

```
while (Test)
  Body
```

Do-While Version

```
  if (!Test)
    goto done;
  do
    Body
    while(Test);
done:
```

Goto Version

```
  if (!Test)
    goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```

*or*

```
goto middle;
loop:
  Body
middle:
  if (Test)
    goto loop;
```

```
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;
}
```

# Switch Statements

**Multiple case labels**

Here: 5, 6

**Fall through cases**

Here: 2

**Missing cases**

Here: 4

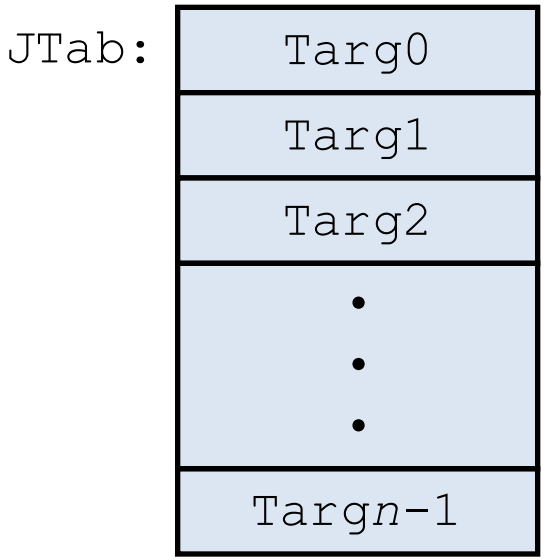
**Lots to manage,**
**we need a *jump table***

# Jump Table Structure

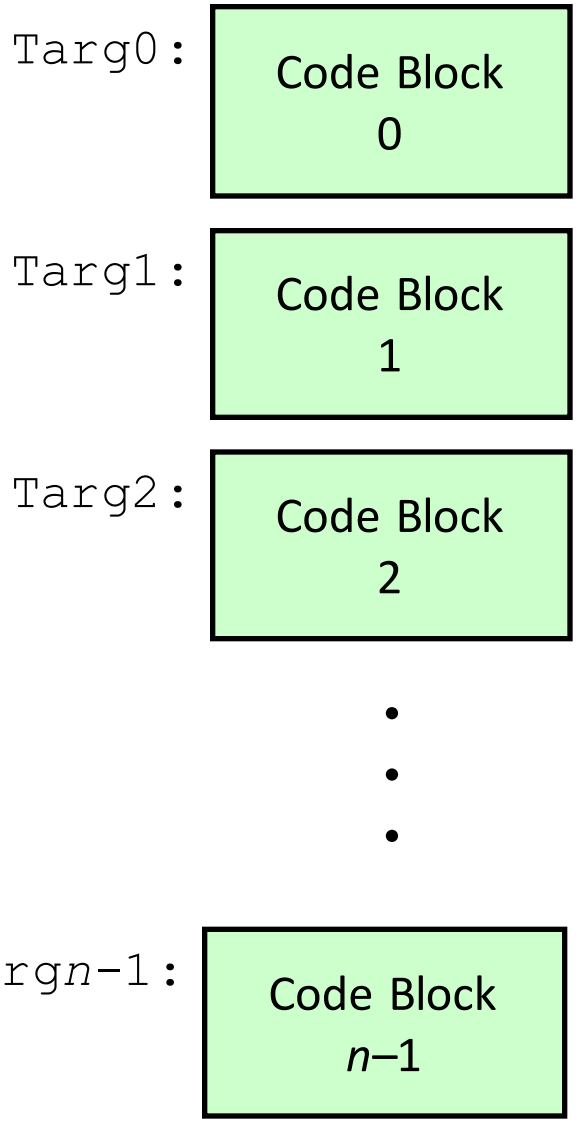
Switch Form

```
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```

Jump Table




Jump Targets




Approximate Translation

```
target = JTab[x];
goto target;
```

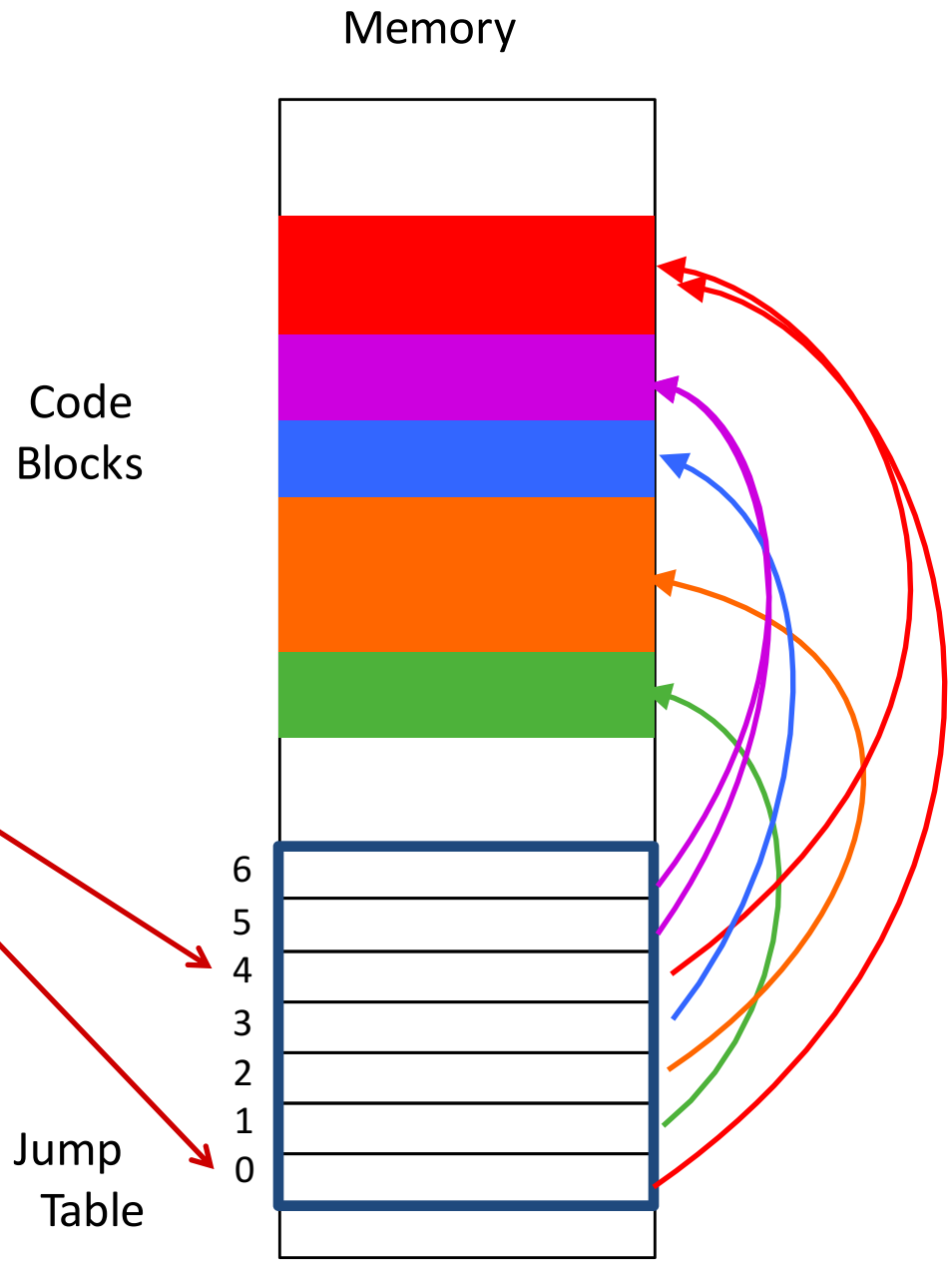
# Jump Table Structure

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 (IA32)

declaring data, not instructions

4-byte memory alignment

Jump table

```
.section .rodata
   .align 4
.L62:
  .long    .L61   # x = 0
  .long    .L56   # x = 1
  .long    .L57   # x = 2
  .long    .L58   # x = 3
  .long    .L61   # x = 4
  .long    .L60   # x = 5
  .long    .L60   # x = 6
```

"long" as in mov*l*: 4 bytes

```
    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 (IA32)

ex

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

Setup:

```
switch_eg:
   pushl %ebp              # Setup
   movl  %esp, %ebp        # Setup
   pushl %ebx              # Setup
   movl  $1, %ebx          # w = 1
   movl  8(%ebp), %edx     # edx = x
   movl  16(%ebp), %ecx    # ecx = z
   cmpl  $6, %edx
   ja    .L61
   jmp   *.L62(,%edx,4)
```

Jump table

```
.section .rodata
   .align 4
.L62:
  .long   .L61   # x = 0
  .long   .L56   # x = 1
  .long   .L57   # x = 2
  .long   .L58   # x = 3
  .long   .L61   # x = 4
  .long   .L60   # x = 5
  .long   .L60   # x = 6
```

*Translation?*

# Switch Statement Example (IA32)

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

Jump table

```
.section .rodata
   .align 4
.L62:
  .long   .L61   # x = 0
  .long   .L56   # x = 1
  .long   .L57   # x = 2
  .long   .L58   # x = 3
  .long   .L61   # x = 4
  .long   .L60   # x = 5
  .long   .L60   # x = 6
```

Setup:

```
switch_eg:
   pushl %ebp              # Setup
   movl  %esp, %ebp        # Setup
   pushl %ebx              # Setup
   movl  $1, %ebx          # w = 1
   movl  8(%ebp), %edx     # edx = x
   movl  16(%ebp), %ecx    # ecx = z
   cmpl  $6, %edx          # x:6
   ja    .L61              # if > 6 goto
default
   jmp   *.L62(,%edx,4)    # goto JTab[x]
```

jump above (like jg, but unsigned)

*Indirect jump*

# Assembly Setup Explanation (IA32)

## Table Structure

Each target requires 4 bytes

Base address at **`.L62`**

Jump table

```
.section .rodata
   .align 4
.L62:
  .long   .L61  # x = 0
  .long   .L56  # x = 1
  .long   .L57  # x = 2
  .long   .L58  # x = 3
  .long   .L61  # x = 4
  .long   .L60  # x = 5
  .long   .L60  # x = 6
```

## Jump target address modes

**Direct:** **`jmp .L61`**

Jump target is denoted by label **`.L61`**

**Indirect:** **`jmp *.L62(,%edx,4)`**

Start of jump table: **`.L62`**

Must scale by factor of 4 (labels are 32-bits = 4 bytes on IA32)

Fetch target from effective address **`.L62 + edx*4`**

**`target = JTab[x]; goto target;`** (only for $0 \le \mathbf{x} \le 6$)

# Code Blocks (Partial)

```
switch(x) {
   . . .
case 2:         // .L57
    w = y/z;
    /* Fall Through */
case 3:         // .L58
    w += z;
    break;
   . . .
default:        // .L61
    w = 2;
}
return w;
```

```
.L61:  // Default case
   movl  $2, %ebx     # w = 2
   jmp .L63
.L57:  // Case 2:
   movl  12(%ebp), %eax  # y
   cltd                  # Div prep
   idivl %ecx            # y/z
   movl  %eax, %ebx # w = y/z
# Fall through - no jmp
.L58:  // Case 3:
   addl  %ecx, %ebx # w+= z
   jmp .L63
...
.L63
   movl  %ebx, %eax  # return w
   popl  %ebx
   leave
   ret
```

# Code Blocks (Rest)

```
switch(x) {
case 1:         // .L56
    w = y*z;
    break;
  . . .
case 5:
case 6:         // .L60
    w -= z;
    break;
  . . .
}
return w;
```

```
.L60: // Cases 5&6:
   subl  %ecx, %ebx  # w -= z
   jmp .L63
.L56: // Case 1:
   movl  12(%ebp), %ebx # w = y
   imull %ecx, %ebx     # w*= z
   jmp .L63


...
.L63
   movl  %ebx, %eax  # return w
   popl  %ebx
   leave
   ret
```

# Code Blocks (Partial, return inlined)

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

The compiler might choose to pull the return statement in to each relevant case rather than jumping out to it.

```
.L61:  // Default case
   movl  $2, %ebx     # w = 2
   movl  %ebx, %eax   # Return w
   popl  %ebx
   leave
   ret
.L57:  // Case 2:
   movl  12(%ebp), %eax  # y
   cltd                  # Div prep
   idivl %ecx            # y/z
   movl  %eax, %ebx # w = y/z
# Fall through - no jmp
.L58:  // Case 3:
   addl  %ecx, %ebx # w+= z
   movl  %ebx, %eax # Return w
   popl  %ebx
   leave
   ret
```

# Code Blocks (Rest, return inlined)

```
    switch(x) {
    case 1:       // .L56
        w = y*z;
        break;
      . . .
    case 5:
    case 6:       // .L60
        w -= z;
        break;
      . . .
    }
```

```
.L60: // Cases 5&6:
   subl  %ecx, %ebx  # w -= z
   movl  %ebx, %eax  # Return w
   popl  %ebx
   leave
   ret
.L56: // Case 1:
   movl  12(%ebp), %ebx # w = y
   imull %ecx, %ebx     # w*= z
   movl  %ebx, %eax  # Return w
   popl  %ebx
   leave
   ret
```

The compiler might choose to pull the return statement in to each relevant case rather than jumping out to it.

# Switch machine code

## Setup

Label **`.L61`** will mean address **`0x08048630`**

Label **`.L62`** will mean address **`0x080488dc`**

## Assembly Code

```
switch_eg:
  . . .
   ja     .L61             # if > goto default
   jmp    *.L62(,%edx,4)   # goto JTab[x]
```

## Disassembled Object Code

```
08048610 <switch_eg>:
 . . .
08048622:  77 0c                    ja     8048630
08048624:  ff 24 95 dc 88 04 08     jmp    *0x80488dc(,%edx,4)
```

# Switch machine code

## Jump Table

Doesn't show up in disassembled code

Can inspect using GDB if we know its address.

```
(gdb) x/7xw 0x080488dc
```

E***x***amine **7** he***x***adecimal format "***w***ords" (4 bytes each)

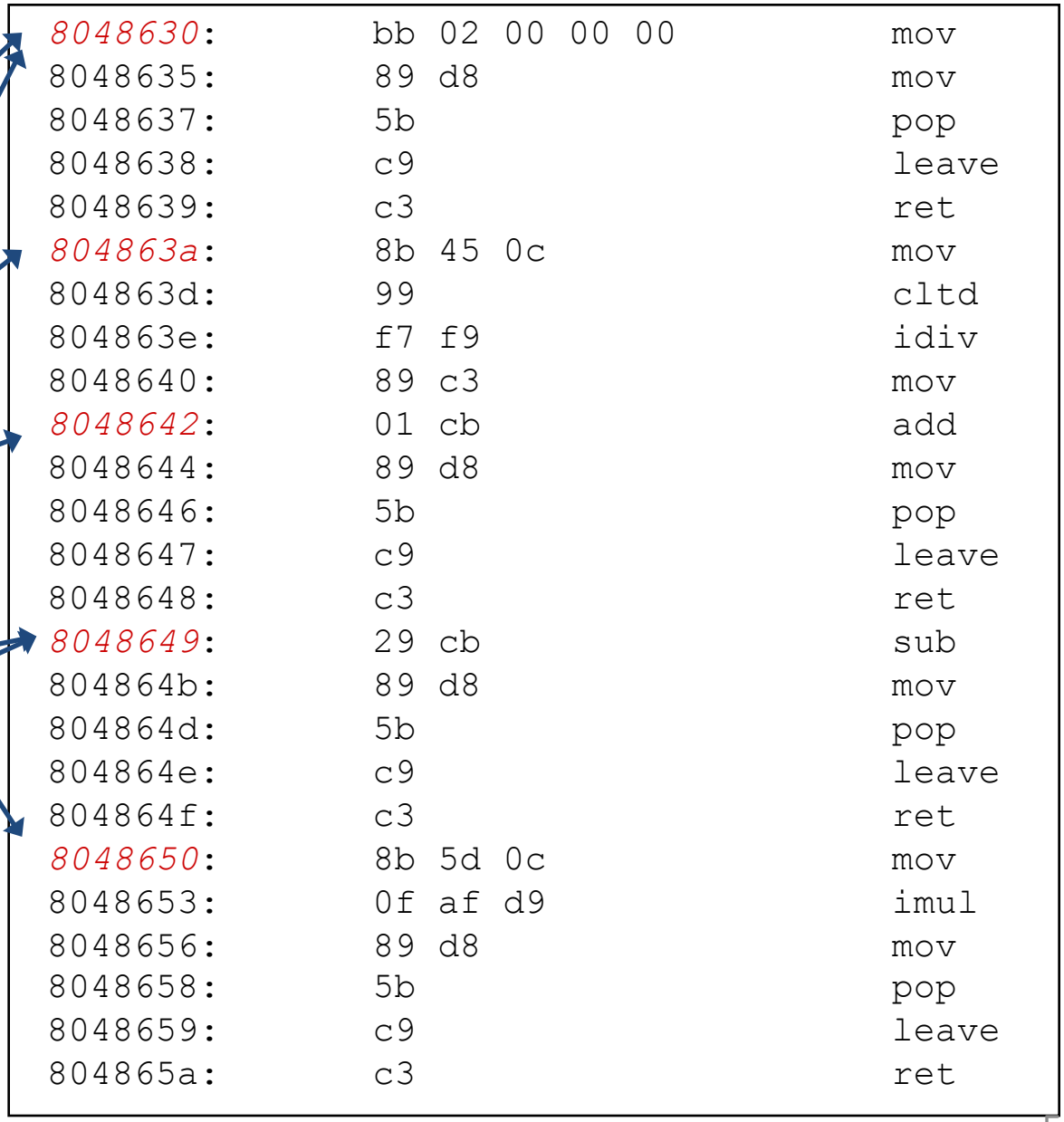
Use command "**`help x`**" to get format documentation

```
0x080488dc:
  0x08048630
  0x08048650
  0x0804863a
  0x08048642
  0x08048630
  0x08048649
  0x08048649
```

# Matching Disassembled Targets

**0x080488dc:**

0x08048630
0x08048650
0x0804863a
0x08048642
0x08048630
0x08048649
0x08048649



```
8048630:        bb 02 00 00 00          mov
8048635:        89 d8                   mov
8048637:        5b                      pop
8048638:        c9                      leave
8048639:        c3                      ret
804863a:        8b 45 0c                mov
804863d:        99                      cltd
804863e:        f7 f9                   idiv
8048640:        89 c3                   mov
8048642:        01 cb                   add
8048644:        89 d8                   mov
8048646:        5b                      pop
8048647:        c9                      leave
8048648:        c3                      ret
8048649:        29 cb                   sub
804864b:        89 d8                   mov
804864d:        5b                      pop
804864e:        c9                      leave
804864f:        c3                      ret
8048650:        8b 5d 0c                mov
8048653:        0f af d9                imul
8048656:        89 d8                   mov
8048658:        5b                      pop
8048659:        c9                      leave
804865a:        c3                      ret
```

# 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;
}
```