x86-64 Control Flow
Topics

- Condition codes
- Jumps
- Conditional moves
- Jump tables
Motivation

- We cannot translate the following C function to assembly, using only data movement and arithmetic operations
  - Fundamental requirement: ability to control the flow of program execution (i.e., decision-making)
  - Necessary for translating structured code

```c
int min (int x, int y)
{
    if (x < y) {
        return x;
    } else {
        return y;
    }
}
```
Control flow

- The **program counter (PC)** tracks the address of the next instruction to be executed
  - To change the PC in assembly, use a **jump** instruction
    - Often the jump will be relative to the current PC value
  - In assembly, the target of a jump is usually a **label**, which is converted to a code address by the assembler
    - Labels are written using colon notation (e.g., “L1:" )
  - However, **unconditional** jumps aren’t sufficient for decision-making
    - In fact, the compiler can just re-arrange code to avoid them

```
movl $2, %eax
jmp L1
movl $3, %eax    # never executed!
L1:
movl $4, %eax
```
Conditional jumps

- Conditional jumps only jump under certain conditions
- In machine/assembly code, conditional jumps are often encoded using a pair of instructions
  - The first sets the condition codes of the CPU
    - On x86-64, the FLAGS register
    - Arithmetic/logical instructions do this as a side effect
    - Special-purpose instructions `cmp` and `test`
  - The second jumps base on the value of the condition codes
    - On x86-64, many variants: “jump-if-equal”, “jump-if-less-than”, etc.

```
cmpl %eax, %ecx       # means “compare %ecx with %eax”
jle pos1              # means “jump-if-less-than-or-equal”
```
Condition codes

• x86-64: special %flags register stores bits for these condition codes:
  – **CF** (carry): last operation resulted in a carry out or **borrow in**
    • (e.g., overflow for unsigned arithmetic)
  – **ZF** (zero): last operation resulted in a zero
  – **SF** (sign): last operation resulted in a negative value
  – **OF** (overflow): last operation caused a two’s complement overflow (negative or positive)

• As well as a few we won’t use:
  – **PF** (parity): last operation resulted in an even number of 1 bits in the eight least significant bits
  – **AF** (adjust): last operation resulted in a carry out for the four least significant bits
  – **IF** (interrupt): CPU will handle interrupts

• Use $eflags to reference this register in GDB
  – E.g., “print $eflags” or “display $eflags”
Aside: Subtraction

- **In addition**, the carry flag is set if a subtraction requires a carry out of the most significant (leftmost) bit
  - Basically, it’s the “extra bit” necessary to represent the result
  - E.g., $\text{1001} + \text{0001} = \text{1010}$ (CF=0)
  - E.g., $\text{1111} + \text{0001} = \text{0000}$ (CF=1)

- **In subtraction**, the carry (borrow) flag is set if a subtraction requires a borrow into the most significant (leftmost) bit
  - E.g., $\text{1000} - \text{0001} = \text{0111}$ (CF=0)
  - E.g., $\text{0000} - \text{0001} = \text{1111}$ (CF=1)
Condition codes

• Special `cmp` and `test` instructions
  - `cmp` compares two values (computes \( \text{arg}_2 - \text{arg}_1 \))
    - **NOTE REVERSED ORDERING** – also, the result is not saved
    - Type-agnostic: all flags are set, but not all are relevant!
    - Does not change either operand
  - `test` checks for non-zero values (computes \( \text{arg}_2 \& \text{arg}_1 \))
    - Often, the arguments are the same (or one is a bit mask)
    - Always sets carry and overflow flags to zero
    - Does not change either operand

```
cmpl %eax, %ecx  # means “compare %ecx with %eax”
testl $0xFF, %edx  # means “check low-order 8 bits of %edx”
```
Jump instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Synonym</th>
<th>Jump condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>Label</td>
<td>1</td>
<td>Direct jump</td>
</tr>
<tr>
<td>jmp</td>
<td>*Operand</td>
<td>1</td>
<td>Indirect jump</td>
</tr>
<tr>
<td>je</td>
<td>Label</td>
<td>jz</td>
<td>ZF</td>
</tr>
<tr>
<td>jne</td>
<td>Label</td>
<td>jnz</td>
<td>~ZF</td>
</tr>
<tr>
<td>js</td>
<td>Label</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>jns</td>
<td>Label</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg</td>
<td>Label</td>
<td>jnle</td>
<td>~(SF ~ OF) &amp; ~ZF</td>
</tr>
<tr>
<td>jge</td>
<td>Label</td>
<td>jnl</td>
<td>~(SF ~ OF)</td>
</tr>
<tr>
<td>jl</td>
<td>Label</td>
<td>jnge</td>
<td>SF ~ OF</td>
</tr>
<tr>
<td>jle</td>
<td>Label</td>
<td>jng</td>
<td>(SF ~ OF)</td>
</tr>
<tr>
<td>ja</td>
<td>Label</td>
<td>jnbe</td>
<td>~CF &amp; ~ZF</td>
</tr>
<tr>
<td>jae</td>
<td>Label</td>
<td>jnb</td>
<td>~CF</td>
</tr>
<tr>
<td>jb</td>
<td>Label</td>
<td>jnae</td>
<td>CF</td>
</tr>
<tr>
<td>jbe</td>
<td>Label</td>
<td>jna</td>
<td>CF</td>
</tr>
</tbody>
</table>

**Figure 3.15** The jump instructions. These instructions jump to a labeled destination when the jump condition holds. Some instructions have “synonyms,” alternate names for the same machine instruction.
C code:

```c
int min (int x, int y) {
    if (x < y) {
        return x;
    } else {
        return y;
    }
}
```

x86-64 assembly:

```assembly
(x in %edi, y in %esi)

min:
    cmpl %esi, %edi
    jge .L3
    movl %edi, %eax
    ret

.L3:
    movl %esi, %eax
    ret
```
Example

C code:

```c
int min (int x, int y) {
    if (x < y) {
        return x;
    } else {
        return y;
    }
}
```

x86-64 assembly:

```
(x in %edi, y in %esi)

min:
    cmpl %esi, %edi
    jge .L3
    movl %edi, %eax
    ret

.L3:
    movl %esi, %eax
    ret
```
C code:

```c
int min (int x, int y)
{
    if (x < y) {
        return x;
    } else {
        return y;
    }
}
```

x86-64 assembly:

```
(x in %edi, y in %esi)

min:
    cmpl %esi, %edi
    jge .L3
    movl %edi, %eax
    ret
.L3:
    movl %esi, %eax
    ret
```
Example

C code:

```c
int min (int x, int y)
{
    if (x < y) {
        return x;
    } else {
        return y;
    }
}
```

x86-64 assembly:

```assembly
(x in %edi, y in %esi)

min:
    cmpl %esi, %edi
    jge .L3
    movl %edi, %eax
    ret

.L3:
    movl %esi, %eax
    ret
```
Control flow

• Compilers translate block-structured code to linear code using conditional jumps
  - We can simulate conditional jumps in C using the `goto` statement
    • General template: "if (<cond>) goto <label>;
    • Syntax for labels is the same in C and assembly (colon notation)

• CS:APP: C “goto code” is code that uses only `if/goto` and `goto`
  - No blocks (and therefore no “else” blocks or explicit loops)
  - Not a good idea in general!
    • Famous letter by Dijkstra: "Go To Statement Considered Harmful"
  - However, it is useful for pedagogical purposes (closer to assembly)
Example

C code:

```c
if (x < y) {
    printf("A");
} else {
    printf("B");
}
printf("C");
```

C goto code:

```c
goto L1;
printf("A");
goto L2;
L1:
printf("B");
L2:
printf("C");
```

note inverted condition!

C code:

```c
while (x < 5) {
    x = x - 1;
}
```

C goto code:

```c
goto L2;
L1:
x = x - 1;
L2:
if (x < 5) goto L1;
```
**Conditionals (in goto code)**

```plaintext
if (<test-expr>)
    <true-branch>
else
    <false-branch>
```

```plaintext
if (!<test-expr>)
    goto false;
<true-branch>
goto done;
false:
    <false-branch>
done:
```

```plaintext
if (<top-expr>)
    if (<test-expr>)
        <true-branch>
    else
        <false-branch>
else
    <else-branch>
```

```plaintext
if (!<top-expr>)
    goto else;
if (!<test-expr>)
    goto false;
<true-branch>
goto done;
false:
    <false-branch>
done:
    goto end;
else:
    <else-branch>
end:
```

If/else

Nested if/else
Conditionals (in goto code)

if (<test-expr>)
  <true-branch>
else
  <false-branch>

if (!<test-expr>)
  goto false;
  <true-branch>
  goto done;
false:
  <false-branch>
done:

if (<top-expr>)
  if (<test-expr>)
    <true-branch>
  else
    <false-branch>
else
  <else-branch>

if (!<top-expr>)
  goto else;
  if (!<test-expr>)
    goto false;
    <true-branch>
    goto done;
false:
    <false-branch>
done:
  goto end;
else:
  <else-branch>
end:

If/else

Nested if/else
Conditional moves

- Similar to conditional jumps, but they move data if certain condition codes are set
  - Benefit: no *branch prediction* penalty
  - We'll see how this produces faster code in a few weeks
- In C code: "\( x = ( \text{<cond>} ? \text{<tvalue>} : \text{<fvalue>}) \)"

```
cmpq %rax, %rbx
jg L01
movq %rax, %rcx
jmp L02
L01:
movq %rbx, %rcx
L02:
```

```
movq %rax, %rcx
cmpq %rax, %rbx
cmovg %rbx, %rcx
```
Conditional moves

• Similar to conditional jumps, but they move data if certain condition codes are set
  - Benefit: no branch prediction penalty
    • We'll see how this produces faster code in a few weeks
  - In C code: "x = ( <cond> ? <tvalue> : <fvalue>)"

```assembly
cmpq %rax, %rbx
jg L01
movq %rax, %rcx
jmp L02
L01:
   movq %rbx, %rcx
L02:
```

```assembly
movq %rax, %rcx
cmpq %rax, %rbx
cmovg %rbx, %rcx
```
Loops

- Basic idea: jump back to an earlier label
- Three basic forms:
  - Do-while loops
  - Jump-to-middle loops
  - Guarded-do loops
- Note: we’ll use goto code in C first
  - Just to avoid unnecessary complication
  - If you can translate a loop into goto code, it's then much easier to convert to assembly
Loops

do
  \textit{<body-statement>}
while (\textit{<test-expr>});

while (\textit{<test-expr>})
  \textit{<body-statement>}

\textbf{Do-while loop}

\textbf{Jump-to-middle loop}

\textbf{Guarded-do loop}

loop:
  \textit{<body-statement>}
  \textbf{if} (\textit{<test-expr>})
  \textbf{goto} loop;

goto test;
loop:
  \textit{<body-statement>}

\textit{test:}
  \textbf{if} (\textit{<test-expr>})
  \textbf{goto} loop

\textbf{if} (!\textit{<test-expr>})
\textbf{goto} done
loop:
  \textit{<body-statement>}
  \textbf{if} (\textit{<test-expr>})
  \textbf{goto} loop

\textbf{done:}
Loops

**Do**
<body-statement>
while (<test-expr>);

**while** (<test-expr>)
<body-statement>

**do**
<body-statement>
while (<test-expr>);

loop:
<body-statement>
if (<test-expr>)
go to loop;

done:

Do-while loop

Jump-to-middle loop

Guarded-do loop
Loops

\[
\textbf{for} \ (<\textit{init-expr}>; \ <\textit{test-expr}>; \ <\textit{update-expr}>) \\
<\textit{body-statement}>
\]

goto \ test;  \\
\textbf{loop}: \\
<\textit{body-statement}>
\textbf{test}: \\
\textbf{if} \ (<\textit{test-expr}>) \\
\textbf{goto} \ \textbf{loop}

if \ (!<\textit{test-expr}>)
\textbf{goto} \ \textbf{done}
\textbf{loop}:
<\textit{body-statement}>
\textbf{if} \ (<\textit{test-expr}>)
\textbf{goto} \ \textbf{loop}
\textbf{done}:

Jump-to-middle loop  \\
Guarded-do loop
Loops

for (<init-expr>; <test-expr>; <update-expr>)
<body-statement>

Jump-to-middle loop

Guarded-do loop
Switch statements

- One approach: convert to if/elseif code
  - Problem: performance varies based on ordering and actual runtime values!

```java
switch (x) {
    case 10: do_blah();
             break;
    case 11: do_foo();
             break;
    case 13: do_bar();
             break;
    case 15: do_baz();
             break;
    default: error();
}
```

```java
if (x == 10) {
    do_blah();
} else if (x == 11) {
    do_foo();
} else if (x == 13) {
    do_bar();
} else if (x == 15) {
    do_baz();
} else {
    error();
}
```
Switch statements

- Indexed **indirect jump** ("computed goto")
  - Jump to an address stored in a register
  - Implemented using a data structure called a jump table
  - Efficient when # of options is high and the value range is small

```c
switch (x) {
    case 10: do_blah();
             break;
    case 11: do_foo();
             break;
    case 13: do_bar();
             break;
    case 15: do_baz();
             break;
    default: error();
}
```

Jump Table

<table>
<thead>
<tr>
<th>Index</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x400e51</td>
</tr>
<tr>
<td>1</td>
<td>0x400e88</td>
</tr>
<tr>
<td>2</td>
<td>0x401900</td>
</tr>
<tr>
<td>3</td>
<td>0x400f12</td>
</tr>
<tr>
<td>4</td>
<td>0x401900</td>
</tr>
<tr>
<td>5</td>
<td>0x400f34</td>
</tr>
</tbody>
</table>

*x is in %rdx, address of jump table is in %rbx*

```c
subq $0xA, %rdx
movq (%rbx,%rdx,0x8), %rcx
jmp  *%rcx
```
Related coursework

- We can **always** (and automatically!) translate from structured code to linear/goto code
  - This is what a compiler does!
  - If you’re interested in learning more about how this works, plan to take CS 432 as your systems elective
Exercise

• Convert the following C function into x86-64 assembly:

```c
int sum = 0;
int x = 1;
while (x < 10) {
    sum = sum + x;
    x = x + 1;
}
```