Machine-Level Programming: Branching and Loops

CSCI 237: Computer Organization
10^{th} Lecture, Mar 3, 2025

Jeannie Albrecht

Administrative Details

- Lab 2: first three phases due this week
  - Any questions? How's it going??
- HW 3 due Friday on Glow
- I will be out this Friday. Sorry!
  - I will post a lecture video instead
  - Please watch it!

Last Time

- Data Dependent Control
  - Control: Condition codes
  - Conditional branches and moves
  - Loops
  - Switch Statements

- Followup from Friday:
  - What exactly does `cmpq` do?
    - `cmpq a, b` sets flags based on `b-a`, but doesn't store result (just sets condition codes)

Recap: Jumping

- Jump Instructions: **`jX address`**
  - Jump to different part of code depending on condition codes

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)

Conditional Branch Example (with jumps)

- Generation

```
> gcc –Og -S –fno-if-conversion absdiff.c
```

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

```
absdiff:
   cmpq    %rsi, %rdi   # x:y
   jle     .L4
   movq    %rdi, %rax
   subq    %rsi, %rax
   ret
.L4:       # x <= y
   movq    %rsi, %rax
   subq    %rdi, %rax
   ret
```

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

Today

- Data Dependent Control
  - Control: Condition codes
  - Finish up conditional branches and moves
  - Loops
  - Switch Statements

Expressing with Goto Code

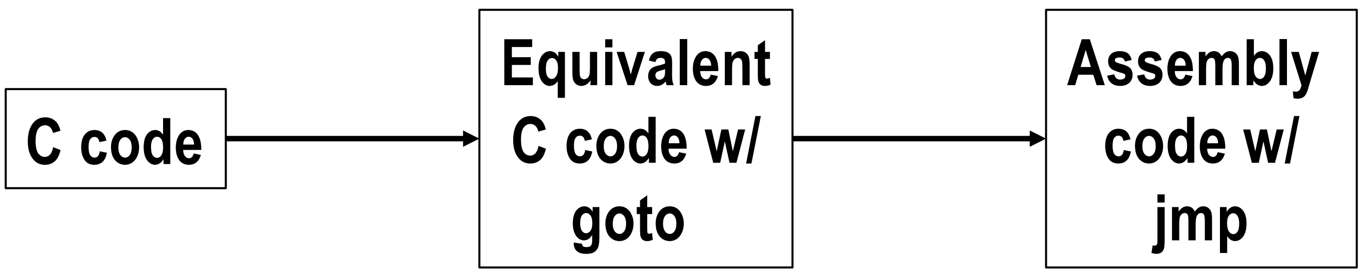
- C allows **`goto`** statements (typically considered very bad programming style!)
- Jump to (or goto) position designated by label
- Goto version of C code is similar to x86-64 with jumps

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

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

jX Is a Powerful Tool

- We can convert many C control flow constructs to equivalent C code that contains goto statements
- We can convert C code with goto statements into x86-64 with **jX**
- Let's look closer at this mapping



General Conditional Expression Translation (Using Branches/Jumps)

C Code (w/ ternary operator)

```
val = Test ? val_if_true : val_if_false;
```

```
val = Test ? Then_Expr : Else_Expr;
```

```
val = x > y ? x - y : y - x;
```

Goto Version

```
  ntest = !Test;
  if (ntest) goto Else;
  val = Then_Expr;
  goto Done;
Else:
  val = Else_Expr;
Done:
  . . .
```

- Create separate code regions for then & else expressions
- Execute appropriate one

General Conditional Expression Translation (Using Conditional Moves instead of Jumps)

- Conditional Move Instructions: `cmovX src, dest`
  - Instruction supports:

    `if (Test) Dest ← Src`
  - Supported in post-1995 x86 processors
  - GCC tries to use them
    - But, only when known to be **safe**
- Benefits of `cmov` vs `jump`
  - Branches (jumps) are very disruptive to instruction flow through pipelines
  - Relies on good *prediction* for good performance
  - Conditional moves do not require *control transfer* (better for processor)

C Code

```
val = Test
      ? Then_Expr
      : Else_Expr;
```

"Goto" Version

```
result = Then_Expr;
eval = Else_Expr;
nt = !Test;
if (nt) result = eval;
return result;
```

Conditional Move Example

Generation

```
> gcc –O1 –S absdiff.c
```

Notice the compile flags!

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

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

```
absdiff:
   movq    %rdi, %rax  # x
   subq    %rsi, %rax  # result = x-y
   movq    %rsi, %rdx
   subq    %rdi, %rdx  # eval = y-x
   cmpq    %rsi, %rdi  # x:y
   cmovle  %rdx, %rax  # if <=, result = eval
   ret
```

Bad Cases for Conditional Move

Expensive Computations

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

- Both values get computed!
- Only makes sense to do this when both computations are very simple

Bad Performance

Risky Computations

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

- Both values get computed
- May have undesirable effects (like making p NULL)

Unsafe

Computations with side effects

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

- Both values get computed
- Must be side-effect free (can't change x)

Illegal

Note about Compile flags

- If-else statements executed using conditional branches or moves
  - `–Og –S –fno-if-conversion`
    - Used to generate **conditional branch** code (with jumps)
    - –fno-if-conversion not always needed, but often is
  - `–O1`
    - Used to generate **conditional move** code (no jumps)

Moving on to loops

- Data Dependent Control
  - Control: Condition codes
  - Finish up conditional branches and moves
  - Loops
  - Switch Statements

“Do-While” Loop Example

C Code

```
long pcount_do
       (unsigned long x) {
  long result = 0;
  do {
    result += x & 0x1;
    x >>= 1;
  } while (x);
  return result;
}
```

Goto Version

```
long pcount_goto
       (unsigned long x) {
   long result = 0;
 loop:
   result += x & 0x1;
   x >>= 1;
   if(x) goto loop;
   return result;
}
```

- Count number of 1’s in argument **x** (aka bitcount or “popcount”)
- Use conditional jump to either continue looping or to exit loop

"Do-While" Loop Compilation

Goto Version

```
long pcount_goto
       (unsigned long x) {
   long result = 0;
 loop:
   result += x & 0x1;
   x >>= 1;
   if(x) goto loop;
   return result;
}
```

```
    movl    $0, %eax      #  result = 0
.L2:                      #  loop:
    movq    %rdi, %rdx
    andl    $1, %edx      #  t = x & 0x1
    addq    %rdx, %rax    #  result += t
    shrq    %rdi          #  x >>= 1
    jne     .L2           #  if(x) goto loop
    ret
```

Register	Use(s)
`%rdi`	Argument `x`
`%rax`	`result`

General “Do-While” Translation

C Code

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

Goto Version

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

- Body:

```
{
  Statement1;
  Statement2;
    …
  Statementn;
}
```

General “While” Translation #1

- “Jump-to-middle” translation
- Used with **-Og** compile flag

While version

```
while (Test)
  Body
```

Goto Version

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

While Loop Example #1

C Code

```
long pcount_while
       (unsigned long x) {
  long result = 0;

  while (x) {
    result += x & 0x1;
    x >>= 1;
  }

  return result;
}
```

Jump to Middle

```
long pcount_goto_jtm
       (unsigned long x) {
   long result = 0;
   goto test;
 loop:
   result += x & 0x1;
   x >>= 1;
 test:
   if(x) goto loop;
   return result;
}
```

- Compared to "do-while" version of function, execution seems to "jump to the middle" (hence the name)
- Initial goto starts while loop execution at "test" (which is in the middle of the code)

General "While" Translation #2

While version

```
while (Test)
  Body
```

- "Do-while" conversion (aka "guarded-do")
- Used with `–O1` compile flag

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:
```

While Loop Example #2

C Code

```
long pcount_while
        (unsigned long x) {
  long result = 0;

  while (x) {
    result += x & 0x1;
    x >>= 1;
  }

  return result;
}
```

Do-While Version

```
long pcount_goto_dw
        (unsigned long x) {
    long result = 0;
    if (!x) goto done;
 loop:
    result += x & 0x1;
    x >>= 1;
    if(x) goto loop;
 done:
    return result;
}
```

- Compared to “do-while” version of function, looks very similar *except* for initial test
- Initial conditional “guards” entrance to loop

For Loop Form

General Form

```
for (Init; Test; Update )

    Body
```

```
#define WSIZE 8*sizeof(int)
long pcount_for
       (unsigned long x){
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++) {
    unsigned bit =
       (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned bit =
       (x >> i) & 0x1;
  result += bit;
}
```

For Loop → While Loop

For Version

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

While Version

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

For-While Conversion

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned bit =
       (x >> i) & 0x1;
  result += bit;
}
```

```
long pcount_for_while
       (unsigned long x) {
  size_t i;
  long result = 0;
  i = 0;
  while (i < WSIZE) {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
    i++;
  }
  return result;
}
```

For Loop → Do While Loop

For Version

```
for (Init; Test; Update)

    Body
```

Do While Version

```
  Init;

  if (!Test) goto done;

loop:

  Body;

  Update;

  if (Test) goto loop;

done:
```

For Loop Do-While Conversion

C Code

```
long pcount_for
   (unsigned long x) {
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++){
    unsigned bit =
       (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

Goto Version

```
long pcount_for_goto_dw
   (unsigned long x) {
  size_t i;
  long result = 0;
  i = 0;                                  Init
  if (!(i < WSIZE))                      ! Test
    goto done;
 loop:
  {
    unsigned bit =
       (x >> i) & 0x1;                    Body
    result += bit;
  }
  i++;                                    Update
  if (i < WSIZE)                          Test
    goto loop;
 done:
  return result;
}
```

- Initial test can be optimized away since WSIZE is multiple of sizeof(int) and is never 0

Loops Summary

- Looked at do-while, while, for loops
- General strategy: rewrite C code into "goto" version
- x86-64 is very similar to goto version

- Next up: switch statements

Switch Statements

- Switch statements evaluate a given expression and execute statements based on the evaluated value
- Used to perform different actions based on different conditions (or cases)
- Alternative to long if-else if statements that compare a single variable to several values
- A “multiway branch” statement that provides an easy way to dispatch execution to different parts of code based on the value of the expression

Switch Statements

```
switch(expression) {
   case value1:
       statement_1;
       break;
   case value2:
       statement_2;
       break;
   .
   .
   .
   case value_n:
       statement_n;
       break;
   default:
       default_statement;
}
```

- Rules
  - The “case value” must be of “char” or “int” type in C
  - There can be one or N number of cases
  - The values in the case must be unique
  - Each statement of the case can have an *optional* break statement
  - The default statement is also optional

Switch Statement

```
int var = 1;

switch (var) {
   case 1:
       printf("Case 1 is Matched.");
       break;

   case 2:
       printf("Case 2 is Matched.");
       break;

   case 3:
       printf("Case 3 is Matched.");
       break;

   default:
       printf("Default case is Matched.");
}
```

Switch Statements

Switch Form

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

Jump Table

```
JTab: Targ0
      Targ1
      Targ2
      •
      •
      •
      Targn-1
```

Jump Targets

```
Targ0:   Code Block 0
Targ1:   Code Block 1
Targ2:   Code Block 2
  •
  •
  •
Targn-1: Code Block n–1
```

Translation (Extended C)

```
goto *JTab[x];
```

gcc often translates switch statements into "extended C" with jump tables (when # cases > 4) -> *indirect jumping*

Switch Statement Example

C code

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

Jump table in x86-64

```
.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 statement in x86-64:

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi           # x:6
    ja      .L8                # Use default, goto L8
    jmp     *.L4(,%rdi,8)      # goto *JTab[x]
```

Direct jump → ja

Indirect jump → jmp

Interpreting the Jump Table

Jump table in x86-64

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

```
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi        # x:6
    ja      .L8             # Use default
    jmp     *.L4(,%rdi,8)   # goto *JTab[x]
```

Code Blocks (x == 1)

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

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

```
.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
```

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

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

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

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

```
.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
```

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

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

```
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       #  w = 2
  ret
```

```
.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
```

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

Summary of Ch 3.6

- C Data-dependent Control
  - if-then-else, do-while, while, for, switch
- Assembler Control
  - Conditional jump, Conditional move
  - Indirect jump (via jump tables)
  - Compiler generates code sequence to implement more complex control
- Standard Techniques
  - Loops converted to do-while or jump-to-middle form
  - Large switch statements use jump tables
  - Sparse switch statements may use “decision trees” (if-else if-else if-else)