Lecture 16: Conditions, decisions, and loops

David Hovemeyer

October 25, 2023

601.428/628 Compilers and Interpreters

Today

- Conditions
- Decisions
- Loops
- Additional considerations

Conditions

Conditions

- A *condition* is an expression used as a truth value
- In C, any integer or pointer value can be used as a condition
  - Integer: 0 is false, non-zero values are true
  - Pointer: null pointer is false, non-null pointers are true
- *Relational* operators compare integer or pointer values to produce a truth value
  - <, >, ==, !=, etc.
- *Logical* operators operate on truth values
  - &&. ||, !
- All relational and logical operators yield an `int` value which is required to be either 1 (true) or 0 (false)

Values vs. control

Conditions are used for two related but distinct purposes:

1. To compute a truth value (1 or 0) as a data value
2. To control execution (i.e., when used in a control construct such as `if`, `if`/`else`, a `while` loop, etc.)

In general, these uses require somewhat different code generation strategies.

Recommendation: generate code for conditions to compute a boolean data value. When the result of a condition is used in a control structure (decision or loop), check whether the computed data value is true or false.

This approach will generate *slightly* convoluted code, but

- it avoids special cases for purpose #1 vs. #2
- the generated code will be easy to simplify later on

Handling relational operators in high-level IR

The high-level IR has dedicated instructions for relational operators. These operators behave much like other ALU instructions: there are two source operands and one destination operand.

E.g., `cmplt_l` compares two 32-bit signed integers and

- stores the value 1 in the destination if the first source operand is less than the second source operand, and
- stores the value 0 in the destination otherwise

```
/* Store 1 in vr15 if vr10 < vr11, otherwise store 0 in vr15 */
cmplt_l  vr15, vr10, vr11
```

Condition as computing a value

```
/* C code */
int a, b, c;

a = read_i32();
b = read_i32();

c = a < b;

print_i32(c); // prints 0 or 1
```

```
/* generated high-level IR */
call      read_i32
mov_l     vr13, vr0
mov_l     vr10, vr13
call      read_i32
mov_l     vr14, vr0
mov_l     vr11, vr14
cmplt_l   vr15, vr10, vr11
mov_l     vr12, vr15
mov_l     vr1, vr12
call      print_i32
```

Note: a is `vr10`, b is `vr11`, c is `vr12`

Conditional jumps in high-level IR

The high-level IR has two conditional jump instructions, `cjmp_t` (conditional jump if true) and `cjmp_f` (conditional jump if false.)

These instructions consume the boolean value computed by a comparison in order to conditionally transfer control to a target instruction.

Condition as controlling execution

```
/* C code */
int i, n, sum;

n = read_i32();

i = 0;
sum = 0;

while (i < n) {
  sum = sum + i;
  i = i + i;
}

print_i32(sum);
```

```
/* high-level IR */
        call     read_i32
        mov_l    vr13, vr0
        mov_l    vr11, vr13
        mov_l    vr14, $0
        mov_l    vr10, vr14
        mov_l    vr15, $0
        mov_l    vr12, vr15
        jmp      .L1
.L0:    add_l    vr13, vr12, vr10
        mov_l    vr12, vr13
        mov_l    vr14, $1
        add_l    vr15, vr10, vr14
        mov_l    vr10, vr15
.L1:    cmplt_l  vr14, vr10, vr11
        cjmp_t   vr14, .L0
        mov_l    vr1, vr12
        call     print_i32
```

Note: i is vr10, n is vr11, sum is vr12

Computing a boolean value in low-level code

In x86-64, the `set`X instructions set an 8-bit register to 1 or 0 based on testing condition codes. (X represents the equality or inequality being tested.)

For example, the code

```
cmpl    %r11d, %r10d
setl    %al
```

would set the 8-bit `%al` register to 1 if the 32-bit signed value in `%r10d` is less than the 32-bit signed value in `%r11d`, and set `%al` to 0 otherwise.

Zero-extending the 8-bit value resulting from a `set`X instruction yields a 32-bit `int` value that is either 1 or 0, which can be the result of the condition.

Computing a boolean value in low-level code (example)

```
/* in high-level IR */
cmplt_l  vr14, vr10, vr11

/* in low-level IR */
movl     -48(%rbp), %r10d    /* cmplt_l  vr14, vr10, vr11 */
cmpl     -40(%rbp), %r10d
setl     %r10b
movzbl   %r10b, %r11d
movl     %r11d, -16(%rbp)
```

Note that the low-level code generator allocated storage for `vr10`, `vr11`, and `vr14` as (respectively) `-48(%rbp)`, `-40(%rbp)`, and `-16(%rbp)`.

Using a condition for control flow in low-level code

If every condition yields a boolean value, control flow can be implemented by

- comparing the computed boolean value to 0, and then
- executing a conditional jump

Using a condition for control flow in low-level code (example)

```
/* in high-level IR */
cmplt_l  vr14, vr10, vr11
cjmp_t   vr14, .L0

/* in low-level IR */
movl     -48(%rbp), %r10d    /* cmplt_l  vr14, vr10, vr11 */
cmpl     -40(%rbp), %r10d
setl     %r10b
movzbl   %r10b, %r11d
movl     %r11d, -16(%rbp)
cmpl     $0, -16(%rbp)       /* cjmp_t   vr14, .L0 */
jne      .L0
```

Note that the low-level code generator allocated storage for `vr10`, `vr11`, and `vr14` as (respectively) `-48(%rbp)`, `-40(%rbp)`, and `-16(%rbp)`.

Simplifying control flow

Peephole optimization can be very effective at simplifying idioms in generated code, including simplifying code generated for control flow. For example:

```
/* Prior to peephole optimization */
movl      %r12d, %r10d         /* cmplt_l  vr14<%r9d>, vr10, vr11 */
cmpl      %r13d, %r10d
setl      %r10b
movzbl    %r10b, %r11d
movl      %r11d, %r9d
cmpl      $0, %r9d             /* cjmp_t   vr14<%r9d>, .L0 */
jne       .L0

/* After peephole optimization */
cmpl      %r13d, %r12d         /* cmplt_l  vr14<%r9d>, vr10, vr11 */
jl        .L0
```

(Note that in the generated code, the register allocator has assigned CPU registers as storage for the virtual registers used.)

Decisions

Decisions

A *decision* makes a choice about a condition or other data value to conditionally-execute code.

Examples: `if` statements, `if`/`else` statements, `switch` statements.

The high-level code generator should generate labels (`.L0`, `.L1`, etc.) for the conditionally-executed code as necessary. These will be targets of unconditional and conditional jump instructions.

if statements

```
/* C code */
int a, b;
a = read_i32();
b = read_i32();
if (a < b) {
  print_i32(42);
}
...rest of code...
```

```
/* high-level IR */
    call     read_i32
    mov_l    vr12, vr0
    mov_l    vr10, vr12
    call     read_i32
    mov_l    vr13, vr0
    mov_l    vr11, vr13
    cmplt_l  vr14, vr10, vr11
    cjmp_f   vr14, .L0
    mov_l    vr12, $42
    mov_l    vr1, vr12
    call     print_i32
.L0:
    ...rest of code...
```

Check condition, conditional branch

Note: a is `vr10`, b is `vr11`

if statements

```
/* C code */
int a, b;
a = read_i32();
b = read_i32();
if (a < b) {
  print_i32(42);
}
...rest of code...
```

Body of if statement

```
/* high-level IR */
    call     read_i32
    mov_l    vr12, vr0
    mov_l    vr10, vr12
    call     read_i32
    mov_l    vr13, vr0
    mov_l    vr11, vr13
    cmplt_l  vr14, vr10, vr11
    cjmp_f   vr14, .L0
    mov_l    vr12, $42
    mov_l    vr1, vr12
    call     print_i32
.L0:
    ...rest of code...
```

Note: a is `vr10`, b is `vr11`

if/else statements

```
/* C code */
int a, b;
a = read_i32();
b = read_i32();
if (a < b) {
  print_i32(42);
} else {
  print_i32(17);
}
...rest of code...
```

Check condition, conditional branch

```
/* high-level IR */
    call     read_i32
    mov_l    vr12, vr0
    mov_l    vr10, vr12
    call     read_i32
    mov_l    vr13, vr0
    mov_l    vr11, vr13
    cmplt_l  vr14, vr10, vr11
    cjmp_f   vr14, .L1
    mov_l    vr12, $42
    mov_l    vr1, vr12
    call     print_i32
    jmp      .L0
.L1:
    mov_l    vr12, $17
    mov_l    vr1, vr12
    call     print_i32
.L0:
    ...rest of code...
```

Note: a is vr10, b is vr11

if/else statements

```
/* C code */
int a, b;
a = read_i32();
b = read_i32();
if (a < b) {
  print_i32(42);
} else {
  print_i32(17);
}
...rest of code...
```

“If true” and “if false”‘ blocks

```
/* high-level IR */
    call     read_i32
    mov_l    vr12, vr0
    mov_l    vr10, vr12
    call     read_i32
    mov_l    vr13, vr0
    mov_l    vr11, vr13
    cmplt_l  vr14, vr10, vr11
    cjmp_f   vr14, .L1
    mov_l    vr12, $42
    mov_l    vr1, vr12
    call     print_i32
    jmp      .L0
.L1:
    mov_l    vr12, $17
    mov_l    vr1, vr12
    call     print_i32
.L0:
    ...rest of code...
```

Note: a is vr10, b is vr11

if/else statements

```
/* C code */
int a, b;
a = read_i32();
b = read_i32();
if (a < b) {
  print_i32(42);
} else {
  print_i32(17);
}
...rest of code...
```

Avoid fall-through from "if true" to "if false" block

```
/* high-level IR */
    call      read_i32
    mov_l     vr12, vr0
    mov_l     vr10, vr12
    call      read_i32
    mov_l     vr13, vr0
    mov_l     vr11, vr13
    cmplt_l   vr14, vr10, vr11
    cjmp_f    vr14, .L1
    mov_l     vr12, $42
    mov_l     vr1, vr12
    call      print_i32
    jmp       .L0
.L1:
    mov_l     vr12, $17
    mov_l     vr1, vr12
    call      print_i32
.L0:
    ...rest of code...
```

Note: a is vr10, b is vr11

switch statements

A `switch` statement could be translated into an equivalent series of `if`/`else if` statements:

```
int a;
a = ...some value...;
switch (a) {
case 0:
  ...code...
  break;
case 1:
case 2:
  ...code...
  break;
default:
  ...code...
}
```

```
int a;
a = ...some value...;
if (a == 0) {
  ...code...
} else if (a == 1 || a == 2) {
  ...code...
} else {
  ...code...
}
```

Jump tables

If the values of the cases are "dense" within a range, a `switch` statement can be compiled as a *jump table*. The idea:

1. An array is allocated where each entry contains the code address of the first instruction in a case
2. The switched value is converted into an index into this array (generally by subtracting the value of the minimum case value)
3. Executing the correct case means retrieving the code address from the array using the computed index, and jumping to that instruction

A jump table is $O(1)$ rather than $O(N)$, where N is the number of cases.

Loops

`while` loops

A `while` loop is the most general kind of loop in C.

Suggested code generation strategy:

- The code to check loop condition should be labeled and generated at the *end* of the loop body; it conditionally jumps to the beginning of the loop body if the condition evaluates as true
- To enter the loop, jump to the code which checks the loop condition

Example while loop

```
/* C code */
while (i < n) {
  sum = sum + i;
  i = i + 1;
}
...rest of code...

/* High-level IR */
    jmp      .L1
.L0:
    add_l    vr13, vr12, vr10
    mov_l    vr12, vr13
    mov_l    vr14, $1
    add_l    vr15, vr10, vr14
    mov_l    vr10, vr15
.L1:
    cmplt_l  vr14, vr10, vr11
    cjmp_t   vr14, .L0
    ...rest of code...
```

Note: `i` is `vr10`, `n` is `vr11`, `sum` is `vr12`

Example while loop

```
/* C code */
while (i < n) {
  sum = sum + i;
  i = i + 1;
}
...rest of code...
```

```
/* High-level IR */
    jmp       .L1
.L0:
    add_l     vr13, vr12, vr10
    mov_l     vr12, vr13
    mov_l     vr14, $1
    add_l     vr15, vr10, vr14
    mov_l     vr10, vr15
.L1:
    cmplt_l   vr14, vr10, vr11
    cjmp_t    vr14, .L0
    ...rest of code...
```

Enter loop by jumping to the loop condition check

Note: `i` is `vr10`, `n` is `vr11`, `sum` is `vr12`

Example `while` loop

```
/* C code */
while (i < n) {
  sum = sum + i;
  i = i + 1;
}
...rest of code...

/* High-level IR */
    jmp      .L1
.L0:
    add_l    vr13, vr12, vr10
    mov_l    vr12, vr13
    mov_l    vr14, $1
    add_l    vr15, vr10, vr14
    mov_l    vr10, vr15
.L1:
    cmplt_l  vr14, vr10, vr11
    cjmp_t   vr14, .L0
    ...rest of code...
```

Check loop condition, jump to top of loop if condition is true

Note: `i` is `vr10`, `n` is `vr11`, `sum` is `vr12`

Example `while` loop

```
/* C code */
while (i < n) {
  sum = sum + i;
  i = i + 1;
}
...rest of code...

/* High-level IR */
    jmp      .L1
.L0:
    add_l    vr13, vr12, vr10
    mov_l    vr12, vr13
    mov_l    vr14, $1
    add_l    vr15, vr10, vr14
    mov_l    vr10, vr15
.L1:
    cmplt_l  vr14, vr10, vr11
    cjmp_t   vr14, .L0
    ...rest of code...
```

Execute body of loop

Note: `i` is `vr10`, `n` is `vr11`, `sum` is `vr12`

`do`/`while` loops

A `do`/`while` loop is mostly the same as a while loop. The main difference is that you would omit the unconditional jump to the loop condition check that you would use to enter a `while` loop.

do/while example

```
/* C code */
do {
  sum = sum + i;
  i = i + i;
} while (i < n);
...rest of code...
```

```
/* High-level IR */
.L0:
    add_l     vr13, vr12, vr10
    mov_l     vr12, vr13
    add_l     vr14, vr10, vr10
    mov_l     vr10, vr14
    cmplt_l   vr15, vr10, vr11
    cjmp_t    vr15, .L0
    ...rest of code...
```

Note: `i` is `vr10`, `n` is `vr11`, `sum` is `vr12`

do/while example

```
/* C code */
do {
  sum = sum + i;
  i = i + i;
} while (i < n);
...rest of code...
```

```
/* High-level IR */
.L0:
    add_l     vr13, vr12, vr10
    mov_l     vr12, vr13
    add_l     vr14, vr10, vr10
    mov_l     vr10, vr14
    cmplt_l   vr15, vr10, vr11
    cjmp_t    vr15, .L0
    ...rest of code...
```

Note: `i` is `vr10`, `n` is `vr11`, `sum` is `vr12`

Execute body of loop

do/while example

```
/* C code */
do {
  sum = sum + i;
  i = i + i;
} while (i < n);
...rest of code...
```

```
/* High-level IR */
.L0:
    add_l     vr13, vr12, vr10
    mov_l     vr12, vr13
    add_l     vr14, vr10, vr10
    mov_l     vr10, vr14
    cmplt_l   vr15, vr10, vr11
    cjmp_t    vr15, .L0
    ...rest of code...
```

Note: `i` is `vr10`, `n` is `vr11`, `sum` is `vr12`

Check loop condition

`for` loops

A `for` loop is essentially the same as a `while` loop. The only difference is that a variable can be initialized before the loop starts, and an update is automatically executed at the end of each loop iteration.

Equivalence of `for` and `while` loops

```
/* for loop */
for (initialization; condition; update) {
  body
}

/* equivalent while loop */
initialization
while (condition) {
  body
  update
}
```

Additional considerations

Additional considerations

In general, if a conditional branch (e.g., `cjmp_t`) is not taken, control will "fall through" to the next instruction sequentially.

When an `InstructionSequence` is converted to a control-flow graph, these "fall through" control edges are potentially problematic.

- The reason is that basic blocks connected by a fall-through edge must be adjacent when converted from a graph back to a linear sequence of instructions

It's not a bad idea to insert explicit `jmp` instructions and labels to make fall-through edges explicit.

- That way, the code works even if the basic blocks involved in the fall-through are not sequential when converted to a linear representation

Making fall-through edges explicit

```
/* high-level IR with
 * implicit fall-through */
    cmplt_l  vr14, vr10, vr11
    cjmp_f   vr14, .L1
    mov_l    vr12, $42
    mov_l    vr1, vr12
    call     print_i32
    jmp      .L0
.L1:
    mov_l    vr12, $17
    mov_l    vr1, vr12
    call     print_i32
.L0:
    ...rest of code...
```

```
/* high-level IR with
 * explicit fall-through */
    cmplt_l  vr14, vr10, vr11
    cjmp_f   vr14, .L1
    jmp .L2
.L2:
    mov_l    vr12, $42
    mov_l    vr1, vr12
    call     print_i32
    jmp      .L0
.L1:
    mov_l    vr12, $17
    mov_l    vr1, vr12
    call     print_i32
.L0:
    ...rest of code...
```

Removing unnecessary jumps

The unnecessary `jmp` instructions inserted to make fall-through explicit can be easily detected and removed during code optimization.