COMP 264 Fall 2015 Lecture Notes

R. I. Greenberg

October 12, 2015

Machine-Level Rep. of Programs continued – Control

1 Condition Codes

Four single-bit *condition code* registers in CPU:

- CF: Carry Flag. Most recent op had a carry out.
- ZF: Zero Flag. Most recent op had result of zero.
- SF: Sign Flag. Most recent op had negative result.
- OF: Overflow Flag. Most recent op caused 2's complement overflow.

The book gives C expressions to define how these flags would be set for an *addition*, but one can generally expect the mechanism to differ.

Except for `leaq`, all the arithmetic and logical instrs. we saw in Figure 3.10, set the condition codes. But logical ops. set CF and OF to 0, and shift ops. set OF to 0 and CF to the last bit shifted out.

Also, two special operations (each can be on 1 byte, 2 byte, 4 byte, or 8 byte operands) that just set condition codes without modifying any other registers. 8 byte versions are `cmpq` to set based on difference of operands and `testq` to set based on bitwise AND.

2 Accessing the Condition Codes

- Figure 3.14 shows 12 "set" instrs., e.g., `sete`, `setns`, `setge`, `setl`, `setae`. (Can see table in slide 22 of *05-machine-control slides*.)
- Each sets a 1 byte register or bottom byte of a memory location to 0 or 1.

- Ex.: `setl` sets dest. to $SF \oplus OF$.
- Figure 3.14 explains effect of doing a "set" after a `cmpb`, `cmpw`, `cmpl`, or `cmpq` instr. e.g., w. a in `%rdi` and b in `%rsi`:

`cmpq %rsi,%rdi`	Set codes according to $a - b$.
`setl %al`	Set `%al` to 1 if a less than b; else 0.
`movzbq %al,%rax`	Zero extend to full 64 bits.

  Note: `l` suffix on `setl` is for "less than"; no need for operand size suffix.

3 Jump Instructions and their Encodings

- Figure 3.15 shows a "jump" instr. corresp. to each "set" instr. e.g., `jl` jumps when $SF \oplus OF$ is 1. Such jumps are *direct*, i.e., jump target is encoded as part of the instr. In assembly language, use a symbolic label, e.g.:

```
  jl .label
  movl %rax,%rbx
.label:
  movl %rbx,%rax
```

  (Can see table in slide 26 of *05-machine-control slides*.)

- Also an unconditional direct jump `jmp` used with a label as above.
- Also can do *indirect* jump, where target comes from a register or memory location as in the normal way of specifying operands. Syntax examples :

```
jmp *%rbx
jmp *24(%rcx,%rdx,4)
```

- For direct jumps, assembler and linker must replace symbolic reference with an encoding of jump target. Common is *PC-relative addressing*: encode distance of jump target from what would have been next instr. Pluses:
  - Likely to fit in fewer bits.
  - Position independence: Encoding of program won't need to be changed based on where it is loaded into memory.
- Examples in text.

4 Translating C Control Structures into Assembly Language

4.1 If-Then-Else

A natural C if-then-else example:

```
int absdiff(int x, int y) {
  if (x<y) return y-x;
  else return x-y;
}
```

Since we're stuck with just the "jump" instructions of the assembly language for change of control, must make a more direct correspondence to equivalent C code with gotos, e.g.:

```
int gotodiff(int x, int y) {
  int rval;
  if (x < y) goto less;
  rval = x - y;
  goto done;
 less: rval = y - x;
 done: return rval;
}
```

(In a C subroutine, we could actually return a result from each of the "then" and "else" cases without having the two control paths rejoin, but the style here follows a general approach for an if-then-else construct that could be followed by additional code.)

In general can make the following transformation to make assembly language translation more direct:

```
                                     if (test-expr) goto true;
                                        else-block
    if (test-expr) then-block    ⇒      goto done;
    else else-block                  true: then-block
                                     done:
```

In Figure 3.16, you will see that typical assembly code is actually a little different. The usual approach is to invert the test and place the *then-block* first. This is advantageous in the common case when there is no "else".

4.2 Conditional Move

A newer idea than conditional transfer of control: use condition codes to decide whether or not to do a transfer of data. Can be much more efficient on modern machines with pipelining (which we'll look at in Chapter 5).

Since 1995, IA32 processors have had conditional move that decides whether to copy a value to a register. Often went unused (to maintain backward compatibility). But now typically used when possible.

Recall the `absdiff` procedure from above where we need to decide whether to return $x - y$ or $y - x$. Assembly code can place one of the values in `%rax` and then decide whether to move the other value to overwrite `%rax`. Such code is shown in Figure 3.17 along with C code showing the form of this translation. Figure 3.18 gives a table of `cmov__` instructions, just like the `j__` instructions of Figure 3.15 and the `set__` instructions of Figure 3.14. As before, a suffix of "l" is not for "long" but rather "less than". Transfers can be of 16-bit, 32-bit, or 64-bit values, which the compiler infers from the destination register.

The general form of translation with conditional move can be viewed as converting

```
v = test-expr ? then-expr : else-expr
```

to

```
vt = then-expr;
v = else-expr;
t = test-expr;
if (t) v = vt;
```

rather than

```
if (!test-expr)
    goto false;
v = true-expr;
goto done;
false: v = else-expr
done:
```

The translation involving conditional move is only acceptable when it's ok to always evaluate both *then-expr* and *else-expr*, i.e., neither can cause an error or side-effect. It's also only useful when these expressions are easily computed so there isn't a lot of time wasted doing the extra computation.

4.3 Loops

C includes three different loop constructs: "for", "while", and "do-while". Most compilers effectively transform loops to do-while form and then into machine code.

4.3.1 Do-While Loops

- General form:

  ```
  do
      body-statements
  while (test-expr);
  ```

  Always executes *body-statements* at least once, and then repeats as long as *test-expr* evaluates to nonzero.

- Equivalent goto form:

```
    loop:
        body-statements
    if (test-expr) goto loop;
```

- Ex.: Code to compute nth Fibonacci no. ($n \geq 1$):

```
long fib_dw(long n)
{
  long i = 0;
  long val = 0;
  long nval = 1;

    do {
      long t = val + nval;
      val = nval;
      nval = t;
      i++;
    } while (i < n);

    return val;
}
```

The following table showing register usage helps for understanding the assembly code:

Reg.	Var.	Init.
`%rdx`	`i`	0
`%rdi`	`n`	n
`%rcx`	`val`	0
`%rax`	`nval`	1
`%rsi`	`t`	—

On this basis, the core part of the assembly language code from our `gcc` (with `-O2`):

```
# %rdi holds n
movl $1, %eax # 1 -> %eax and thus %rax (nval)
xorl %ecx, %ecx # 0 -> %ecx and thus %rcx (val)
xorl %edx, %edx # 0 -> %edx and thus %rdx (i)
jmp .L2 # goto .L2
.L3:
movq %rax, %rcx # nval -> val
movq %rsi, %rax # t -> nval
.L2:
addq $1, %rdx # i++
leaq (%rcx,%rax), %rsi # val + nval -> t
```

```
cmpq %rdi, %rdx # Set codes per i-n
jl .L3 # if (i<n) goto .L3
```

Note the clever use of `leaq` in the code to add contents of two registers and place result in a third

- (Text shows an example for factorial instead of Fibonacci.)

4.3.2 While Loops

- General form:
  ```
      while (test-expr);
          body-statements
  ```

  Due to test at top, might execute *body-statements* zero times.

- One type of transformation gcc sometimes uses may be referred to as "jump to middle" and is illustrated in our text.

- Another approach that might seem inatural is to consider the following equivalent goto form:
  ```
      loop: if (!test-expr) goto done;
          body-statements
          goto loop;
      done:
  ```
  But it is probably more efficient if we can avoid having two possible changes of control inside the loop.

  Thus, a better equivalent goto form gcc sometimes uses is what may be refered to as "guarded do":
  ```
      if (!test-expr) goto done;
      do
          body-statements
      while (test-expr);
      done:
  ```
  From here, we can transform "if" and " do-while" as before.

- Fibonacci code written with a "while" is here:

  ```
  long fib_w(long n)
  {
    long i = 1;
    long val = 1;
    long nval = 1;
  ```

```
  while (i < n) {
    long t = val+nval;
    val = nval;
    nval = t;
    i++;
  }

  return val;
}
```

Compiling this directly on our gcc did not yield the interesting assembly language implementation I would have liked, but I was able to coerce the compiler by writing the C code in this goto form:

```
long fib_w_goto(long n)
{
  long val = 1;
  long nval = 1;
  long nmi, t;

  if (val >= n)
    goto done;
  nmi = n-1;

 loop:
  t = val+nval;
  val = nval;
  nval = t;
  nmi--;
  if (nmi)
    goto loop;

 done:
  return val;
}
```

which generated this for the core part of the assembly code:

```
# %rdi initially holds value of n ; becomes nmi.
cmpq $1, %rdi # Set codes per n-1
movl $1, %eax # 1 -> %eax and thus %rax (nval)
jle .L2 # if (n<=1) goto .L2
subq $1, %rdi # n-1 -> nmi
```

```
movl $1, %edx # 1 -> %edx and thus %rdx (val)
jmp .L3 # goto .L3
.L5:
movq %rax, %rdx # nval -> val
movq %rcx, %rax # t -> nval
.L3:
subq $1, %rdi # nmi--
leaq (%rdx,%rax), %rcx # val + nval -> t
jne .L5 # if (nmi) goto .L5
.L2:
```

An interesting potential optimization may be noted here: Keep track of $n - i$ instead of i, and use equality to 0 as stopping condition (after initial test that $n \geq i$). This looks to me superficially a bit better (fewer instructions in the loop) than what our gcc does produce in a direct compilation of the Fibonacci "while" code:

```
# %rdi initially holds value of n ; becomes nmi.
cmpq $1, %rdi # Set codes per n-1
movl $1, %eax # 1 -> %eax and thus %rax (nval)
movl $1, %ecx # 1 -> %ecx and thus %rcx (val)
movl $1, %edx # 1 -> %edx and thus %rdx (i)
jg .L3 # if (n<=1) goto .L3
jmp .L2 # goto .L2
.L5:
movq %rsi, %rax # t -> nval
.L3:
addq $1, %rdx # i++
leaq (%rcx,%rax), %rsi # val + nval -> t
movq %rax, %rcx # nval -> val
cmpq %rdi, %rdx # Set codes according to i-n
jne .L5 # if (i<>n) goto .L5
.L2:
```

But it is possible that our version of gcc is being smarter than me, because the execution time depends on other complicated considerations that we will talk about when we discuss "pipelining".

4.3.3 For Loops

- General form:
  `for` (*init-expr*; *test-expr*; *update-expr*)
  body-statements

- C standard defines this as equivalent to:

  ```
  init-expr;
  while (test-expr) {
      body-statements
      update-expr;
  }
  ```

- From here, treat "while" loop as before.

4.4 Switch

Multiway branching based on expression evaluating to an integer.

- General form:

  ```
  switch (expr) {
  case first-result:
      first-case-body
  case second-result:
      second-case-body
    ⋮
  default:
      default-body
  }
  ```

  Each of *first-result*, *second-result*, etc., is just an integer that *expr* might evaluate to. Each case body can "fall through" to the next or conclude with `break;` to jump out of the switch. For example, one could check that

  ```
  void switch2(long x, long *dest) {
    long val=0;
    switch(x) {
    case -1:
      /* Code ending in a break for -1 case here */
    case 0:
    case 7:
      /* Code ending in a break for 0 and 7 cases here */
    case 1:
      /* Code ending in a break for 1 case here */
    case 2:
    case 4:
      /* Code ending in a break for 2 and 4 cases here */
    case 5:
      /* Code ending in a break for 5 case here */
  ```

```
  default:
    /* Code for default case here (3, 6, <-1, or >7 */
  }
  *dest = val;
}
```

is a skeleton of C code that could be used by the compiler to generate the assembly code in Practice Problem 3.30. (Jump tables discussed below for this.)

- Could translate into assembly by considering an equivalent form with "if"s and "goto"s, e.g. (with gotos at the end of case bodies that end with a `break;`):

```
t=expr;
if (t!=first-result) goto second_case
first_case:
    first-case-body
second-case:
    if (t!=second-result) goto third_case
    second-case-body
  ⋮
default:
    default-body
}
```

- But can produce faster code by using an indirect jump with a "jump table". The jump table is a list of the memory locations where the code for the various cases starts. An array reference based on the switch expression takes one to the correct entry in the jump table, which tells what location to jump to. For example assembly code for the switch2 procedure shown above is stated in Practice Problem 3.30 to begin:

```
# x in %rdi
switch2:
addq $1, %rdi # Brings smallest x value of -1 to 0
cmpq $8, %rdi # Compares original x value to 7
ja .L2 # x>7 means go to default case
jmp *.L4(,%rdi,8) # Use jump table to go to correct case body
```

and then the jump table looks like this:

```
.L4:
.quad .L9 # Address of Code for -1 case
.quad .L5 # Address of Code for 0 case
.quad .L6 # Address of Code for 1 case
.quad .L7 # Address of Code for 2 case
```

```
.quad .L2 # Address of Code for 3 case (default)
.quad .L7 # Address of Code for 4 case (same as 2 case)
.quad .L8 # Address of Code for 5 case
.quad .L2 # Address of Code for 6 case (default)
.quad .L5 # Address of Code for 7 case (same as 0 case)
```

A additional condition for the jump table to occupy modest space is that the range of *expr* results that the cases correspond to is not too large.

(Can also see detailed jump table example in *06-machine-procedures slides* 2–12.)