CMPT 295

Unit - Machine-Level Programming

Lecture 13 – Assembly language – Program Control – **cmovX**
Iterative Statements – Loops

Last Lecture

- In C, we can change the execution flow of a program
  1. Conditionaly
     - Conditional statements: if/else, switch
     - Iterative statements: loops
  2. Unconditionally
     - Functions calls
- In x86-64 assembly, we can also change the execution flow of a program
  - `cmp*` instruction (compare)
  - `jX` instructions (jump)
  - `call` and `ret` instructions

Today's Menu

- Introduction
  - C program -> assembly code -> machine level code
- Assembly language basics: data, `move` operation
  - Memory addressing modes
- Operation `leaq` and Arithmetic & logical operations
- Conditional Statement – Condition Code + **cmovX**
- Loops
- Function call – Stack
- Array
- Buffer Overflow
- Floating-point operations

Homework: `int max(int x, int y)`

version 1 – with **`jx`** instruction

In C:

```
int max(int x, int y) {
   int result = x;
   if (y > x)
      result = y;
   return result;
}
```

In Assembly: # x in %edi, y in %esi, result in %eax

```
max:
    movl %edi, %eax # result = x
    cmpl %edi, %esi  # if y <= x then
    jle   endif      # return
    movl %esi, %eax  # result = y (if y > x)
endif:
    ret
```

We branch (jump) when the condition (**`y > x`**) is false, i.e., when (**`y <= x`**)

-> This technique is called *"coding the false condition first"*

or "taking care of …"

Conditional move instruction **cmovX**

What C code looks like when using conditional operator:

```
result = test ? val2 : val1;
return result;
```

What logic of assembly code looks like when using **cmovX** (expressed in C):

```
result = val1;
if (test) result = val2;
return result;
```

- Example: **cmovle** **Src, Dest**

alternative: **int abs(int x)**

in C:

```
int abs(int x){
   if ( x < 0 )
     x = -x;
   return x;
}
```

in assembly:

```
    # x in %edi, result in %eax
abs:
    movl   %edi, %eax   # result = x
    negl   %edi         # x = -x
    cmpl     $0, %eax   # if x < 0 then
    cmovl  %edi, %eax   # result = -x
    ret
```

Advantage of conditional move `cmovX`

Note about **branching**:

- Branches are very disruptive to instruction flow through microprocessor (CPU) pipelines
- However, since conditional moves (`cmovX`) do not require control transfer (no branching/jumping required), they are less disruptive
- So, `gcc` tries to use them, but only when safe

What do we mean by "safe"?

- In `result = test ? aVal : anotherVal;` both values (`aVal` and `anotherVal`) are computed so their computation must be "safe"
- Example of unsafe computations:
  1. Expensive computations `val = Test(x) ? Hard1(x) : Hard2(x);`
     - Only makes sense when computations are very simple
  2. Risky computations `val = p ? *p : 0;`
     - Only makes sense when computations do not crash the application
  3. Computations with side effects `val = x > 0 ? x*=7 : x+=3;`
     - Only makes sense when computations do not have side effects

Homework:

Example: alternate `int max(int x, int y)`

version 2 – with **cmovX** instruction

In C:

```
int max(int x, int y) {
   int result = x;
   if (y > x)
      result = y;
   return result;
}
```

In Assembly: # x in %edi, y in %esi, result in %eax

```
max:
    movl %edi, %eax   # result = x
    cmpl %edi, %esi   # if y > x then
    cmovg %esi, %eax # result = y
    ret
```

While loop – "*coding the false condition first*"

means put code to "exit loop" (i.e., jump when condition is false) right after testing the condition

`int x` and `int y` are arguments to function

in C:

```
while (x < y) {
   // stmts
}
return;
```

① analyze the condition

③ list all 3 outcomes:
- x - y > 0 → x > y → exits loop → g
- < 0 → x < y → loops → l
- = 0 → x = y → exits loop → e

④ which outcomes "exit loop"? → g, e

in assembly: # x in %edi, y in %esi

```
loop:
    cmpl %esi, %edi      # ② write (y, x ← careful of order)
    jge endloop          # ⑤ write; for false condition i.e. for g & e
    # stmts in loop
    jmp loop

endloop:
    ret
```

Loop Pattern 1

```
loop:
   if cond false
     goto done:
   stmts
   goto loop:
done:
```

While loop – "jump-to-middle"

`int x` and `int y` are arguments to function

in C:

```
while (x < y) {
   // stmts
}
return;
```

① ④ consider outcome that "loops"

③ x-y >0 → x >y → exits loop → g
<0 → x <y → loops → l
=0 → x =y → exits loop → e

in assembly: # x in %edi, y in %esi

```
    jmp test
loop:
    # stmts
test:
② cmpl %esi, %edi   (y, x)
⑤ jl loop
    ret
```

Loop Pattern 2

```
  goto test:
loop:
  stmts
test:
  if cond true
    goto loop:
done:
```

Here, we jump back into the loop when condition is true!

Do While loop – "jump-to-middle"

`int x` and `int y` are arguments to function

in C:

```
do {
   stmts
} while (x < y);
return;
```

in assembly:

```
loop:
     # stmts
test:
     cmpl %esi, %edi
     jl   loop

     ret
```

Loop Pattern 2

```
   goto test:
loop:
   stmts
test:
   if cond true
     goto loop:
done:
```

For loop

① Here, are we "coding the false condition first?

② Which loop pattern are we using?

In C:

```
for (i = 0; i < n; i++){
   // stmts
}
return;
```

(initialization: `i = 0`; condition testing: `i < n`; increment: `i++`)

```
i = 0; // initialization
while (i < n) { //condition testing
    // stmts
    i++; // increment
}
return;
```

In Assembly:

```
                    # n in %edi, i in %ecx
    xorl %ecx, %ecx   # initialization
loop:                 # %ecx (i) <- 0
    cmpl %edi, %ecx   # while i < n true
                      #   testing
    jge endloop*      # jump when i >= n
                      #   false condition
    # stmts
    incl %ecx         # i++ increment
    jmp loop          # loop again
endloop:
     ret
```

* In this situation "je" would also work. Do you see why?

Compiler can produce different instruction combinations when assembling the same C code.

Summary

- In x86-64 assembly, there are no conditional statements, however, we can alter the execution flow of a program by using …
  - **`cmp*`** instruction (compare)
  - **`jX`** instructions (jump)
  - **`call`** and **`ret`** instructions
  - **`cmovX`** instructions -> conditional move
- In x86-64 assembly, there are no iterative statements, however, we can alter the execution flow of a program by using …
  - **`cmp*`** instruction
  - **`jX`** instructions (jump)
- CPU uses these *condition codes* to decide whether a …
  - **`jX`** instruction (conditional jump) is to be exectued or a
  - **`cmovX`** instruction (conditional move) is to be exectued
- 2 loop patterns:
  - "*coding the false condition first*" -> `while` loops (hence `for` loops)
  - "*jump-in-middle*" -> `while`, `do-while` (hence `for` loops)

`cmp*` and **`test*`** instructions set *condition codes*

Next Lecture

- Introduction
  - C program -> assembly code -> machine level code
- Assembly language basics: data, `move` operation
  - Memory addressing modes
- Operation `leaq` and Arithmetic & logical operations
- Conditional Statement – Condition Code + `cmovX`
- Loops
- Function call – Stack
  - Overview of Function Call
  - Memory Layout and Stack - x86-64 instructions and registers
  - Passing control
  - Passing data – Calling Conventions
  - Managing local data
- Array
- Buffer Overflow
- Floating-point operations