15213 Lecture 6: Assembly Control Flow

1 Learning Objectives

- Identify the use of condition codes and jump instructions in assembly.
- Recognize the components of simple loops in assembly.
- Infer the C code corresponding to `for`, `do...while`, and `while` loops in assembly.
- Apply knowledge of the TEST and CMP instructions in the context of loops to trace switch statements in assembly.

2 Getting Started

To obtain a copy of today's activity, log into a shark machine and do the following:

1. `$ wget http://www.cs.cmu.edu/~213/activities/lec6.tar`
2. `$ tar xf lec6.tar`
3. `$ cd lec6`

Now run the following commands.

1. `$ ./act4`
2. `$ (gdb) r 5`

Record your answers to the discussion questions below. Recall the activity from last class contains a list of relevant GDB commands.

3 Basic Control Flow

Use GDB's `c` command to progress through the stages. These questions accompany the program. Answer each individually and discuss afterward with your group.

From the activity, we are introduced to the concepts of condition codes and jump instructions. Condition codes refer to four single-bit registers: CF, SF, ZF, and OF, which have the following meanings:

- CF: carry flag (unsigned overflow)
- SF: sign flag (signed result has signed bit set)
- ZF: zero flag (result is zero)
- OF: overflow flag (signed result has changed the sign bit)

Jump instructions have the form `Jx` where the jump is based on the condition `x`. Jump instructions are often called branches, and they change the program counter (`%rip`) to the specified destination.

1. Describe what the instructions TEST and CMP do.
   TEST a, b performs a & b, only modifying condition codes based on the result.
   CMP a, b performs b - a, only modifying condition codes based on the result.

2. Consider the disassembly of the testFlag function. In what case will control jump to testFlag + 29?
   Execution will jump to `testFlag + 29` if the result of `%esi & %edi` equals 0.

3. Consider the disassembly of the `carryAdd` function. Which register corresponds to each of the function's parameters? Which register corresponds to the return value?
   In the function signature char `carryAdd(unsigned long x, unsigned long y, unsigned long* sum)`, `%rdi` corresponds to `unsigned long x`, `%rsi` corresponds to `unsigned long y`, and `%rdx` corresponds to `unsigned long* sum`. `%al` is the char return value.
4. What does the `carryAdd` function do?
   The function adds two values, checks the carry bit after this addition, and stores the value of the carry bit in `%rax`. Afterwards, it stores the result of the addition in a memory location. `add %rsi, %rdi` adds the first and second arguments, `setb %al` stores the carry bit in the least significant bits of `%rax`, `mov %rdi, (%rdx)` moves the result of the addition to the memory location pointed to by the third argument, and `retq` returns from the function.
5. If the tested condition has not changed following a backward jump, what will happen to the program?
   The program will enter an infinite loop.

6. Consider the disassembly of the multThis function. What does this function do?
   The function `multThis` computes the factorial of the input argument.

4 Loops

Now exit out of GDB and run the following command.

```
$ head -n 7 act5.c
```

Fill in the for loop, while loop and do-while loop templates below based on the output. The function prototypes are listed above the template.

```
/* int forLoop(int* x, int len) */
for (i = 0; i < len; i++) {
  . . .
}

/* int whileLoop(int* x, int len) */
while (i < len) {
  . . .
```

```
  i++; // how does the variable i change per iteration?
}

/* int doWhileLoop(int* x, int len) */
do {
  . . .
  i++; // how does the variable i change per iteration?
} while (i < len);
```

In each of the above templates, which register corresponds to the counter variable `i`?

The register `%edx` represents the variable `i`. We can determine this by inspecting which register is incremented with each iteration of the loop.

5 Advanced Control Flow

Recall the assembly for the `carryAdd` function. Do you think it would be easier to implement this function in C? Why or why not?

The function `carryAdd` actually cannot be implemented in C, since it requires checking the condition flags, which are specific to the x86 ISA but not C in general.

Consider the following assembly representing a switch statement. Fill in the template below.

```
/** void switcher(long a, long b, long c, long *dest)
  * a in %rdi, b in %rsi, c in %rdx, dest in %rcx */

switcher:
    cmpq $7, %rdi
    ja     .L2
    jmp     *.L4(, %rdi, 8)
    .section         .rodata
.L7:
    xorq    $15, %rsi
    movq    %rsi, %rdx
.L3:
    leaq     112(%rdx), %rdi
    jmp     .L6
.L5:
    leaq    (%rdx, %rsi), %rdi
    salq     $2, %rdi
    jmp     .L6
.L2:
    movq     %rsi, %rdi
.L6:
    movq    %rdi, (%rcx)
    ret
```

```
/** Jump table */
.L4
    .quad        .L3
    .quad        .L2
    .quad        .L5
    .quad        .L2
    .quad        .L6
    .quad        .L7
    .quad        .L2
    .quad        .L5

void switcher(long a, long b, long c, long *dest) {
    long val;
    switch (a) {
case 5:
    c = b ^ 15;
case 0:
    val = c + 112;
    break;
case 2:
case 7:
    val = (c + b) << 2;
    break;
case 4:
    val = a;
    break;
default:
    val = b;
    }
    *dest = val;
}
```

The key to figuring out switch statements is to combine information from the assembly and the jump table to determine the different cases. We see from the ja instruction (line 3) that .L2 corresponds to the default case since all values not within 0 to 7 go to this case. We then see the value .L5 is repeated in the jump table which means this must correspond to the third and fourth cases. We see the case with xorq falls through which means .L7 must correspond to the first case and .L3 must correspond to the second case. This leaves .L6 to match the 5th case. This problem is example 3.31 in the textbook.