

Control Flow

1 Learning Objectives

- Understand the use of condition codes and jump instructions in x86 assembly language.
- Recognize the components of simple loops in assembly language.
- Infer the C code corresponding to 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:

```
$ wget http://www.cs.cmu.edu/~213/activities/machine-control.tar
$ tar xf machine-control.tar
$ cd machine-control
```

Record your answers to the discussion questions below. You may wish to refer back to the activity from the previous class (<https://www.cs.cmu.edu/~213/activities/gdb-and-assembly.pdf>) which contains a list of relevant GDB commands.

3 Basic Control Flow

This activity introduces the concept of condition codes and branch instructions.

The *condition codes* are four single-bit registers, named ZF, SF, CF, and OF. They are set implicitly by most arithmetic instructions (but not by MOV or LEA) and they have the following meanings:

ZF Result of operation was zero.

SF Result of operation was negative (its *Sign* bit was set)

CF Operation had an unsigned overflow (there was a *Carry* from the leftmost pair of bits)

OF Operation had a signed overflow (the sign bit of both inputs was the same, and the sign bit of the output is not equal to whatever that was)

Control Flow

Jump instructions (also called *branch instructions*) change the program counter (%rip). There are fifteen basic jump instructions:

| Name | A.k.a. | Jump if... | After CMP... |
|------|----------|-------------------------------|-------------------------|
| JMP | | Always | |
| JS | | Negative (SF = 1) | |
| JNS | | Not negative (SF = 0) | |
| JO | | Signed overflow (OF = 1) | |
| JNO | | No signed overflow (OF = 0) | |
| JE | JZ | Zero (ZF = 1) | Equal |
| JNE | JNZ | Not zero (ZF = 0) | Not equal |
| JB | JC, JNAE | Unsigned overflow (CF = 1) | Unsigned below |
| JAE | JNC, JNB | No unsigned overflow (CF = 0) | Unsigned above or equal |
| JA | JNBE | CF = 0 and ZF = 0 | Unsigned above |
| JBE | JNA | CF = 1 or ZF = 1 | Unsigned below or equal |
| JL | JNGE | SF \neq OF | Signed less |
| JGE | JNL | SF = OF | Signed greater or equal |
| JG | JNLE | ZF = 0 and SF = OF | Signed greater |
| JLE | JNG | ZF = 1 or SF \neq OF | Signed less or equal |

The official “name” of a jump instruction is the name that `objdump` and `gdb` will use in disassembly listings. This is usually mnemonic for what the jump instruction will do if used immediately after a `CMP` instruction (as described in the “After CMP...” column). The “a.k.a.” (also known as) names are mnemonic for other interpretations of what the instruction does; people writing assembly by hand can use them for clarity, but the distinction is lost in machine language.

Problem 1. Why is `JZ` (jump if zero) the same instruction as `JE` (jump if equal)?

Remember that `CMP` sets the flags based on the result of subtracting its first argument from its second argument. If the two arguments to `CMP` are equal, the result of the subtraction will be zero, so `ZF` will be set and `SF`, `OF`, `CF` will be clear. Thus, “jump if `ZF = 1`” performs “jump if equal” *when executed immediately after a `CMP` instruction*. The alternative name `JZ` more accurately describes the behavior when this instruction comes immediately after a `TEST` instruction (see below).

Problem 2. Within the `machine-control` directory you created earlier, read the file `jumps.S`. The code in this file doesn’t do anything *useful*, it just demonstrates the syntax of jump instructions. When you understand what’s going on in this file, run these commands:

Control Flow

```
$ as jumps.S -o jumps.o
$ objdump -d jumps.o
```

Examine the output of the second command. (There will be a lot of output. You may want to make your shell window taller, or pipe the output to a “pager”, e.g. `objdump -d jumps.o | less`). Compare it to what you remember from `jumps.S`, and the table above. You will probably notice that all of the “a.k.a.” instructions have changed to the corresponding “name” instruction. What else do the lines for those groups of instructions have in common?

In the hexadecimal representation of machine code, shown to the left of each assembly instruction, when the mnemonic is the same, the first byte of the machine code is also always the same. For example:

```
6: 72 2e          jb     36 <destination>
8: 72 2c          jb     36 <destination>
a: 72 2a          jb     36 <destination>
```

This byte of the machine code is the *opcode* (“operation code”): it identifies the instruction to the CPU. On the x86, the opcode is *not* always the first byte. The jump instructions happen to be short and simple.

Problem 3. In the disassembly listing from the previous question, look at the *second* byte of each machine instruction. This is the part of the instruction that tells the CPU where to find the instruction that will be executed next (if the jump happens). Do you see a pattern to these bytes? What relationship is there among the address of “destination”, the address of each jump instruction, the *length* of each jump instruction, and the value of the second byte?

Each second byte’s value is 2 less than the value of the previous instruction’s second byte. The last jump instruction in the list has a second byte whose value is zero.

The second byte’s value is equal to the number of bytes in between the address of “destination” and the address of the first byte *after* the jump instruction. Put another way, it’s the value to *add* to `%rip` if the jump is taken. (For “microarchitectural” reasons—reasons having to do with how the CPU works internally—`%rip` always holds the address of the *next* instruction to execute, *assuming the jump, if any, is not taken.*)

Problem 4. (*Advanced*) Uncomment the line of `jumps.S` that reads

```
// .skip 97
```

Repeat the `as` and `objdump` commands. What happened to the first several jump instructions? Why do you think this is? What would happen if you changed 97 to 98? Experiment with various other numbers. Do you see a pattern?

Control Flow

The first several jump instructions' machine code has become longer. JMP is now `e9 xx xx xx xx` instead of `eb xx`, and some of the conditional jumps are `0f 8x xx xx xx xx` instead of `7x xx`. This happens for each instruction that needs to encode a change in `%rip` larger than `0x7F`.

Problem 5. (*Advanced*) Based on your answer to the previous question, what do you think the machine instruction `eb f0` would do?

The value added to `%rip` is *signed*, so this instruction jumps (unconditionally) to the instruction beginning 14 bytes *before* this instruction.

4 Comparisons and Conditional Set Instructions

In this activity you will experiment with the CMP instruction, which sets the condition codes based on *comparing* two integers, and see how some of the conditional jump mnemonics correspond to some of C's relational operators. You will also be introduced to the *conditional set* instructions, which set a register to 0 or 1 based on the condition codes.

To begin this activity, run these commands (again, within the "machine-control" directory):

```
$ gdb ./cmp-set
(gdb) r
```

Read and follow the instructions that are printed, until it tells you to come back to this handout.

Problem 6. Based on the disassembly of `sete`, `seta`, and `setg`, which registers contain function arguments? Can you tell which is the first and which is the second argument?

`%rsi` and `%rdi` contain function arguments. (These functions only use the low 16 bits of each, so the disassembly refers to `%si` and `%di`, but conventionally we talk about an entire register using its R-name.) It's not possible to tell from this code which is the first argument and which is the second.

Problem 7. Based on the disassembly of `sete`, `seta`, and `setg`, which register contains the return value?

`%rax` contains the return value. (Here, only the low 32 bits are used, so the disassembly refers to `%eax`.)

Control Flow

You can call functions from the debugger with the `call` command.¹ For example,

```
(gdb) call sete(0, 1)
```

calls the function `sete` with arguments 0 and 1, and prints the result, like this:

```
(gdb) call sete(0, 1)
$1 = 0
```

In this case, `sete` returned zero. The “`$1 =`” prefix is to remind you that you can use `$1` in future function calls, or any other place GDB wants an arithmetic expression, to refer back to the number that was returned. (This is more useful with functions that can return many different values.)

Problem 8. Call `sete`, `seta`, and `setg` with each of the following pairs of values. Fill in the table.

| Arg 1 | Arg 2 | sete | setg | seta |
|--------|--------|------|------|------|
| 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 1 |
| -1 | 0 | 0 | 0 | 1 |
| 0 | -1 | 0 | 1 | 0 |
| 32768 | 32767 | 0 | 0 | 1 |
| 32767 | 32768 | 0 | 1 | 0 |
| 32768 | 32768 | 1 | 0 | 0 |
| 32768 | -32768 | 1 | 0 | 0 |
| -32768 | 32768 | 1 | 0 | 0 |
| -32768 | -32768 | 1 | 0 | 0 |

Problem 9. Assuming `%rdi` is the first and `%rsi` the second argument register, fill in the blanks in the C source code for `sete`, `setg`, and `seta`. (Hint: `stdint.h` defines the type name `int16_t` for 16-bit signed integers, and the type name `uint16_t` for 16-bit unsigned integers.)

```
#include <stdint.h>
int sete(uint16_t a,
         uint16_t b) {
    return a == b;
}
```

¹**Caution:** Do not do this in bomb lab or attack lab. If you do, your bomb will explode, and your attack will not count.

Control Flow

```
int setg( int16_t a,
         int16_t b) {
    return a > b;
}

int seta( uint16_t a,
         uint16_t b) {
    return a > b;
}
```

5 Tests and Conditional Move Instructions

In this activity you will experiment with the TEST instruction, which sets the condition codes based on the *bitwise and* of two integers. You will also be introduced to the *conditional move* instructions, which, based on the condition codes, either do or do not copy one register into another.

To begin this activity, run these commands (again, within the “machine-control” directory):

```
$ gdb ./test-cmov
(gdb) r
```

Read and follow the instructions that are printed, until it tells you to come back to this handout.

Problem 10. Why does the `cmovc` function contain a `CMOVB` instruction rather than a `CMOVC` instruction? (Hint: look carefully at the table on page 2.)

`CMOVB` and `CMOVC` are two names for the same machine instruction. The disassembler always prints “`cmovb`” for this instruction, even though it probably says “`cmovc`” in the source code for `test-cmov`.

Problem 11. In the disassembly of `cmove`, `cmovs`, and `cmovc`, what do you notice about the arguments to the TEST instruction?

Both arguments are the same—TEST is being asked to set condition codes based on the bitwise AND of a register with itself. (This is actually more common than any other use of TEST. $x \& x == x$ for all x , so the condition codes are simply set based on the value of x .)

Problem 12. Call `cmove`, `cmovs`, and `cmovc` with each of the following pairs of values. Fill in the table.

| Arg 1 | Arg 2 | cmove | cmovs | cmovc |
|-------|-------|-------|-------|-------|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 2 | 2 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 2 | 0 | 0 | 0 |
| -1 | 0 | 0 | 0 | 0 |
| -1 | 1 | 0 | 1 | 0 |
| -1 | 2 | 0 | 2 | 0 |
| 32767 | 1 | 0 | 0 | 0 |
| 32768 | 1 | 0 | 1 | 0 |

Problem 13. Why does `cmovs(32768, 1)` return 1?

32768 is larger than T_{max} for a 16-bit twos-complement number, so it is interpreted as negative. Put another way, when test `%di,%di` examines the result of the AND, its sign bit is set, and so it sets SF.

Problem 14. Is it possible to make `cmovc` return anything other than 0? Explain.

No. Bitwise AND never has a carry out, so TEST always clears CF. CMOVC after TEST will never do the move.

6 Loops

In Problem 5 we saw that jump instructions can jump both forward and backwards within the machine code. Backward jumps enable us to implement *loops*, in which part of the code is executed repeatedly.

Problem 15. You have been provided a file `loops.o`, containing machine code for three functions. The body of each function is a loop. Run the command

```
$ objdump -d loops.o
```

Translate the assembly language back into C and fill in the blanks in the functions below.

Control Flow

```
int forLoop(int* x, int len) {
    int ret = 0;
    for (i = 0; i < len; i++) {
        ret += x[i];
    }
    return ret;
}

int whileLoop(int* x, int len) {
    int ret = 0;
    while (i < len) {
        ret += x[i];
        i++;
    }
    return ret;
}

int doWhileLoop(int* x, int len) {
    do {
        ret += x[i];
        i++;
    } while (i < len);
    return ret;
}
```

Problem 16. While you were working out the previous problem, how did you identify which register was used as the counter variable `i`?

One way is to look for a register that's being incremented by 1 each time around the loop. (Caution: we used `-Og` mode to compile `loops.o`. With more aggressive optimization, like what you get with `-O2`, there might not *be* any such register.)

Problem 17. If we hadn't told you, and the names didn't give it away, could you have known that `forLoop`'s C source contained a for loop and `whileLoop`'s C source contained a while loop?

Control Flow

No, because both functions were compiled to exactly the same machine code. This is not an accident; *any* for loop

```
    for (setup; condition; increment) {
        body;
    }
```

can be rewritten as an equivalent while loop

```
    setup;
    while (condition) {
        body;
        increment;
    }
```

7 Switch Statements

Switch statements in C are often compiled to *computed jumps* in assembly language. A jump instruction with an argument like

```
    jmp    *.L4(,%rdi,8)
```

looks up the %rdi'th entry in the array beginning at .L4, and jumps to the address *stored in* that array entry. So, for instance, if %rdi is 2, and array entry 2 (counting from zero, as always) contains the address of label .L5, then the CPU will jump to .L5.

Here is a complete example of what this looks like in assembly.

```
switcher:
    cmpq $7, %rdi
    ja   .L2
    jmp  *.L4(,%rdi,8)
.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)
```

Control Flow

```
ret

.section .rodata
.L4:
.quad .L3
.quad .L2
.quad .L5
.quad .L2
.quad .L6
.quad .L7
.quad .L2
.quad .L5
```

Problem 18. The C code below is a partial translation (“decompilation”) of the assembly code above. Fill in the case labels with the appropriate numbers.

```
// %rdi = a and val, %rsi = b, %rdx = c, %rcx = dest
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;
}
```

Control Flow

The key to figuring out switch statements is to combine information from the assembly and the jump table to determine the different cases. The `ja .L2` instruction tells us that `.L2` is the default case, since all values not within 0 to 7 go to this case. We can then look in the table and see that values 1 and 3 also go to `.L2`, so they must not have case labels of their own. The value `.L5` is also repeated in the jump table, which means this must correspond to the pair of case labels next to each other: 2 and 7. Then we match up the remaining labels with the remaining C cases. This problem is example 3.31 in the textbook.