CSE 351 Lecture 10 – x86 Programming III

Conditionals and Labels

In the simplest case, a conditional in assembly is made up of two assembly instructions:

1) The instruction that sets/changes the condition codes
2) A conditional jump instruction

Recall that the condition codes can be set implicitly by arithmetic and logical instructions or explicitly using `cmq`/`test` and that conditional jumps are determined by the previous result *compared to zero*. Conditionals can be massaged into the appropriate format to make it clear which instructions to choose.

Examples:	`x >= y` →	`x - y >= 0` →	1) `cmp  y, x` 2) `jge` *target*
	`x == 3` →	`x - 3 == 0` →	1) `cmp  $3, x` 2) `je` *target*
	`x & 1` →	`x & 1 != 0` →	1) `test $1, x` 2) `jne` *target*

The operand to a jump instruction (the *target*) is known as a ***label*** in assembly. A *label* is a symbolic representation of an instruction's address and is indicated by an identifier (symbolic string) followed by a colon (*e.g.* `main:`, `foo:`, `loop:`). Labels can be freely introduced in assembly and will be associated with the *next* instruction found, ignoring whitespace. When the name of a label (without the '`:`') is used as a target, program execution will jump to the associated instruction if the jump's condition is met.

If-else statements can be constructed by adding labels and jump statements to guarantee that at most one "branch" gets executed; jump targets are almost always to later/forward addresses in the code.

Loops

Loops (while, do-while, for) can similarly be constructed with labels and jump statements, except the main difference is that now one of the jump targets *must* go backwards to the beginning of the loop body. The differences between loops boil down to differences in when you evaluate the test conditional (top or bottom of the loop) and efficiency in terms of how many jump instructions you need to evaluate.

In the following pseudocode, "`<CC instr>`" means the instruction that sets/changes the condition codes (*i.e.* the first instruction in a conditional) and `j*` means the appropriate conditional jump instruction (*i.e.* the second instruction in a conditional) and `j*'` is the opposite jump instruction:

Do-while:

```
loopTop:
  <body code>
  <CC instr>
  j*  loopTop
loopDone:
```

While ver. 1:

```
loopTop:
  <CC instr>
  j*' loopDone
  <body code>
  jmp loopTop
loopDone:
```

While ver. 2:

```
  <CC instr>
  j*' loopDone
loopTop:
  <body code>
  <CC instr>
  j*  loopTop
loopDone:
```

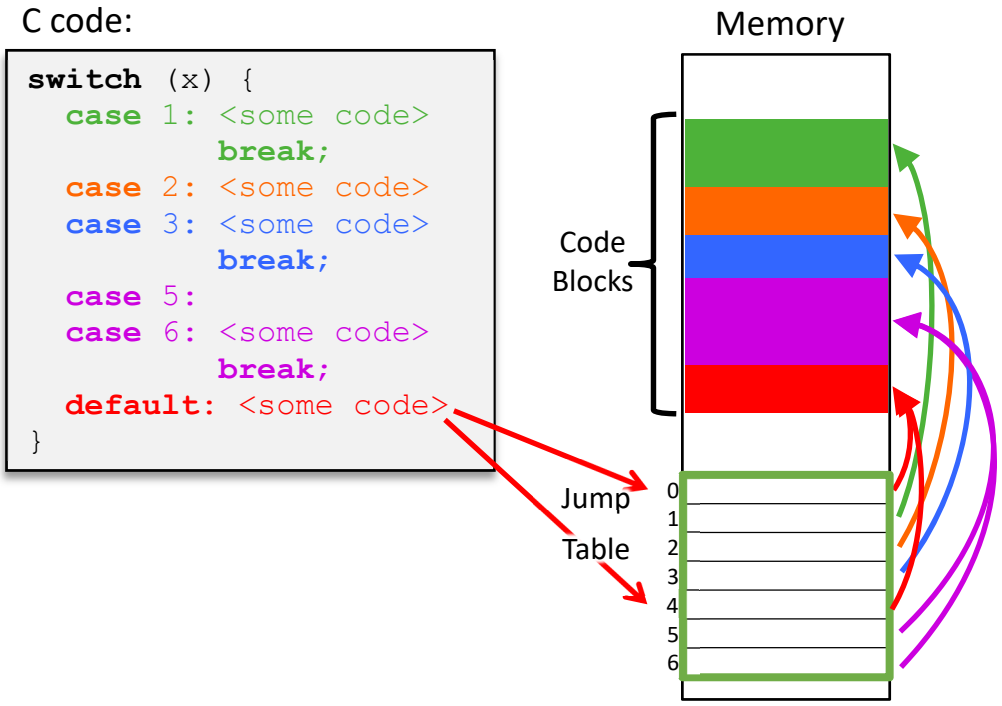
For:

```
<Init>
while (Test) {
  <Body>
  <Update>
}
```

Switch Statements

Sometimes *switch* statements can be thought of as specialized syntax for chains of if-else statements, but the they are actually implemented more efficiently using *jump tables* and *indirect jump* instructions, which also handle the special cases (*e.g.* multiple case labels, fall-through, default case) well.

A ***jump table*** is a data structure used to help branch to different parts of a program. Since all instructions (and therefore, blocks of code) have addresses associated with them, a *jump table* is nothing more than an array of pointers where the pointers are to code blocks instead of program data. The addresses of the code blocks are put into the jump table in the index that matches the associated case's value. Multiple case labels are handled by storing the same code block address in multiple jump table entries. Fall-through is handled by placing the code blocks for those cases one after the other. All non-specified indices should point to the default code block. An example is shown below:



The ***program counter*** (`%rip`) is a special register that holds the address of the next instruction to execute in your program. In order to use the jump table, we need to update the *program counter* to one of the addresses stored in the table. This requires a special instruction called an ***indirect jump***:

```
jmp *Loc
```

The asterisk (`*`) differentiates an indirect jump from a normal jump and tells your program that the address to jump to is found in `Loc`, instead of directly given as a label. The location is specified as a register or memory operand, and we will need to use the memory operand form with our jump table.

Example: **`jmp *.L4(,%rdi,8)`** updates `%rip` to the address of the appropriate code block assuming that `.L4` is the label of the beginning of the jump table and `%rdi` holds the value of the switch variable.