Jump Tables

There are a number of ways to rewrite a ‘switch’ statement in C to assembler. The obvious way is to use ‘if .... then .... else’ conditionals. If you have *n* elements, then this leads to an O(n) solution. If the switch is sparse with the cases sequential or near sequential, then an O(1) solution is possible by using jump tables.

Consider the following example shown in class:

```
typedef enum {ADD, MULT, MINUS, DIV, MOD, BAD} op_type;

char unparse_symbol(op_type op)
{
  switch (op) {
  case ADD :
    return '+';
  case MULT:
    return '*';
  case MINUS:
    return '-';
  case DIV:
    return '/';
  case MOD:
    return '%';
  case BAD:
    return '?';
  }
}
```

This is assembled into the following code with comments shown in class:

```
unparse_symbol:
        pushl %ebp                  # Stack Setup
        movl %esp,%ebp              # Stack Setup

        movl 8(%ebp),%eax           # eax = op
        cmpl $5,%eax                # Compare op : 5
        ja .L49                     # If > goto done
        jmp *.L57(,%eax,4)          # goto Table[op]

.L51:
        movl $43,%eax       # ’+’
        jmp .L49
.L52:
        movl $42,%eax       # ’*’
```

```
        jmp .L49
.L53:
        movl $45,%eax       # '-'
        jmp .L49
.L54:
        movl $47,%eax       # '/'
        jmp .L49
.L55:
        movl $37,%eax       # '%'
        jmp .L49
.L56:
        movl $63,%eax       # '?'
        # Fall Through to .L49

.L49:   # Done:
        movl %ebp,%esp      # Stack clean-up
        popl %ebp           # Stack clean-up
        ret                 # Finish - return value in %eax
```

The compiler also created a data segment (.rodata) where it created a 'jump table' that looks like this:

```
.L57:
        .long .L51      #Op = 0
        .long .L52      #Op = 1 / Memory address is .L57 +4
        .long .L53      #Op = 2 / Memory address is .L57 +8
        .long .L54      #Op = 3 / Memory address is .L57 +12
        .long .L55      #Op = 4 / Memory address is .L57 +16
        .long .L56      #Op = 5 / Memory address is .L57 +20
```

Remembering that our enumerated values are:

ADD	0
MULT	1
MINUS	2
DIV	3
MOD	4
BAD	5

Note that the memory address of an OP is .L57 + Opvalue * 4. Why 4? That's the size of a word on our process, and the size of a memory address. Each entry in the table above is a memory address denoted by a symbol. We use this fact to optimize our switch statement.

The first line moves the OP argument into %eax off of the stack. The OP is next compared to 5 to see if it is within the bounds (0-5) of a legal OP. If OP is greater than 5, then we jump to the end - .L49 and return from the function. Otherwise, we use the jump table. The instruction *jmp *.L57(,%eax,4)* is an absolute (non-PC relative) jump. It can be rewritten in Intel format as *JMP [.L57+EAX*4]*. We are multiplying the OP value by 4, adding it to the location of label .L57 and then taking the value at that memory location and jumping to it. So let's walk through an example- Let OP = 2. We get to the JMP instruction and we have JMP [.L57+8]. There we find the value located at .L57+8 which is .L53, and we do an absolute jump to .L53. The instruction at .L53 moves $45 into %eax, and execution jumps to .L49 and returns with the value $45 in %eax to the calling function.