This code will increment `i` 32 times and test if `i` is less than 32 the same number of times. It's obvious that the test will fail the first 31 times and succeed the last time. You can eliminate the tests by "unrolling" the loop, like this:

```
iload_2     ; Push m
iconst_0    ; Push 0
ishr        ; Compute m >> 0
iconst_1    ; Push 1
iand        ; Compute m >> 0 & 1 (the first bit)
iload_3     ; Push n
iadd        ; Add the result to n
istore_3    ; store n

iload_2     ; Push m
iconst_1    ; Push 1
ishr        ; Compute m >> 1
iconst_1    ; Push 1
iand        ; Compute m >> 1 & 1 (the second bit)
iload_3     ; Add n
iadd
istore_3    ; Store n

;; Repeat this pattern 29 more times

iload_2     ; Push m
bipush 31   ; Push 31
ishr        ; Compute m >> 31 (the leftmost bit)
iconst_1    ; Push 1
iand        ; Compute m >> 31 & 1
iload_3     ; Push n
iadd        ; Add the last bit to n
istore_3    ; Store n
```

Although this greatly expands the size of the resulting code, the total number of instructions executed is reduced. It also allowed us to completely eliminate local variable 1, which had been used as the loop counter. This results in a speedup of 200% to 400%, depending on the virtual machine implementation used.

14.2.5 Peephole Optimization

Compilers often generate code that has redundant instructions. A peephole optimizer looks at the resulting bytecodes to eliminate some of the redundant instructions. This involves looking at the bytecodes a few instructions at a time, rather than doing wholesale reorganization of the code. The name "peephole optimization" comes from the tiny "window" used to view the code.