ECE 573 Step 8 Report: *Optimization via Loop-Invariant Code Motion*

Name: Hyungjun Doh

For ECE 573 project step 8, I decided to optimize **loop-invariant code motion**, which involves identifying computations within a loop that yield the same result on every iteration and moving them outside the loop. This reduces redundant computations, decreases the instruction count within the loop, and improves runtime efficiency.

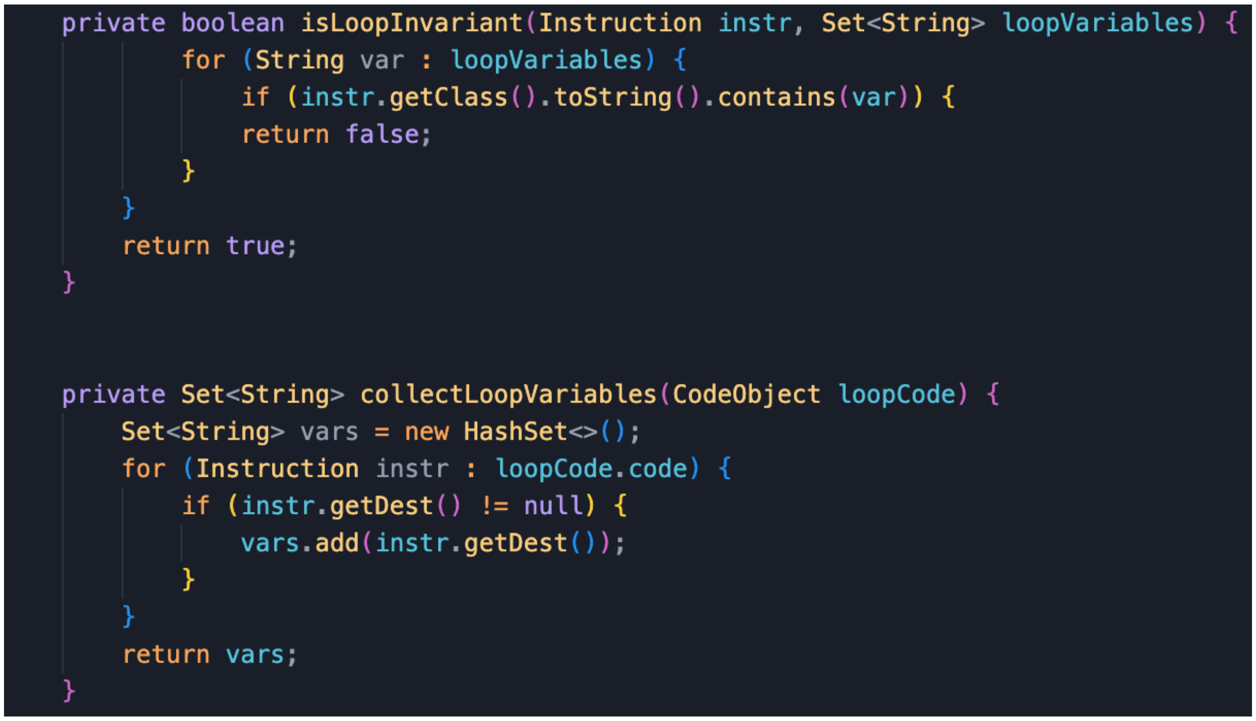
For example, if a value remains constant throughout the iterations of a loop, computing it only once before the loop starts can save processing time and resources.

Content:

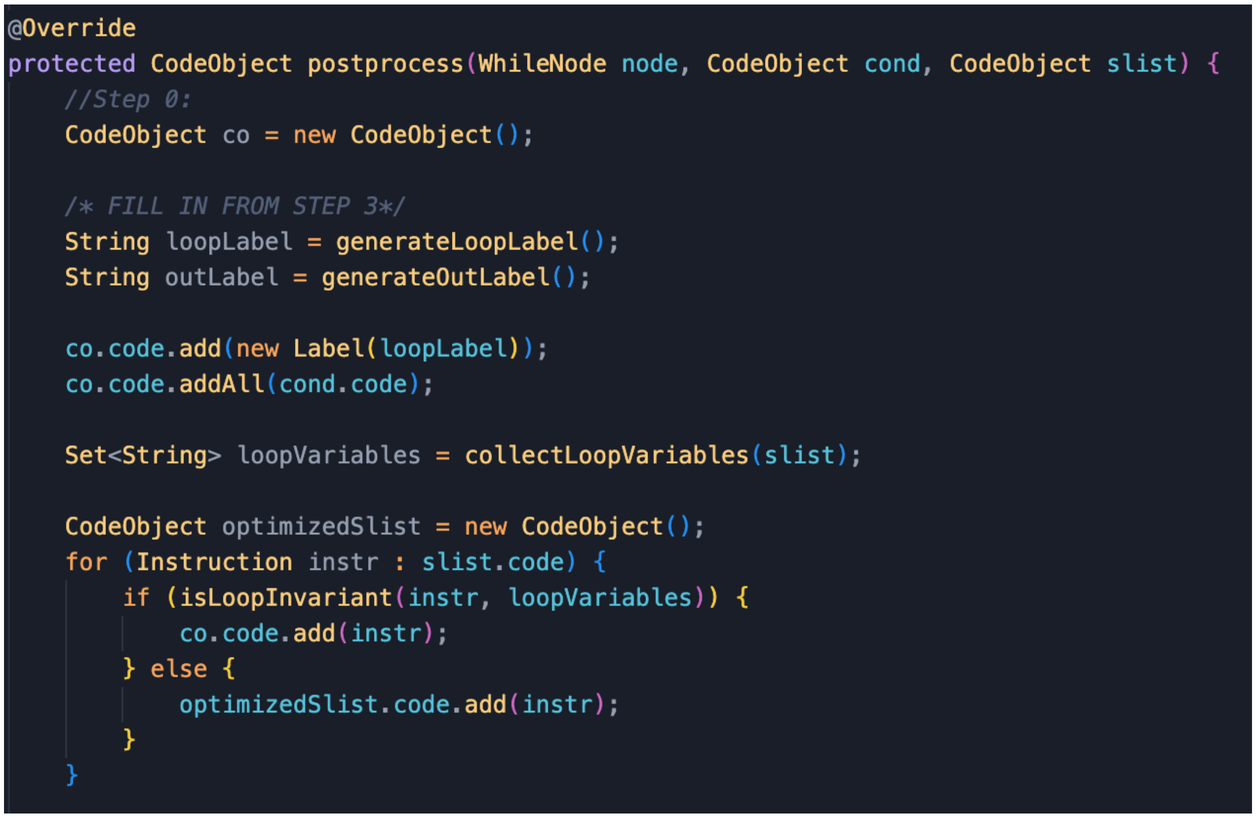
Implementation Details:

To implement loop-invariant code motion in the compiler:

1. **Language:**
    - For this step, I implemented the optimization using the Java programming language, as I have been using Java consistently throughout this semester.
2. **Detection of Invariant Code:**
    - The compiler analyzes the loop body to identify computations that do not depend on variables modified within the loop.
    - A set of "loop variables" is created, containing all variables updated inside the loop. Any computation that does not involve these variables is considered invariant.
    - Implemented Functions: ***isLoopInvariant(), collectLoopVariables()***
3. **Code Motion:**
    - The invariant computation is moved before the loop starts. This is implemented in the ***WhileNode*** function because this function processes the structure of the loop itself. By placing the logic for identifying and moving invariant computations within ***WhileNode***, the compiler ensures that these optimizations are performed during the loop's code generation phase. The rest of the code inside the loop remains unchanged to maintain the correctness of the program's execution.
    - Implemented Functions: ***postprocess(WhileNode)***



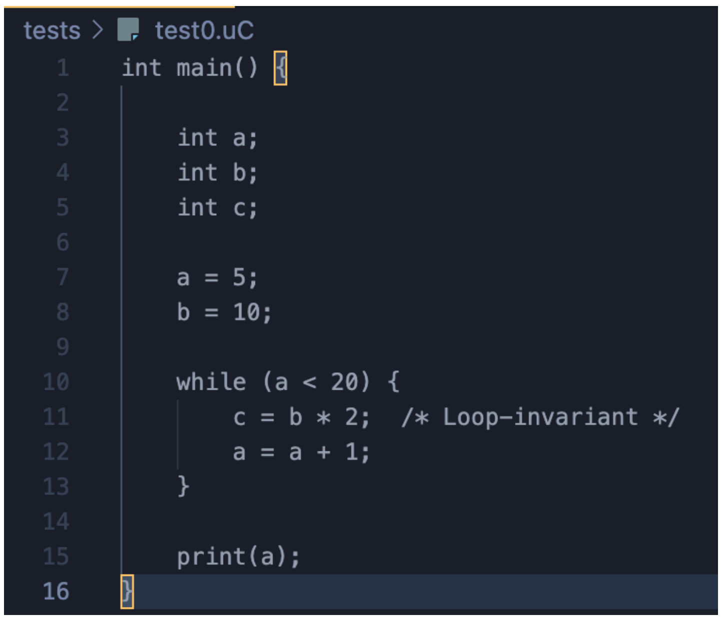
```java
    private boolean isLoopInvariant(Instruction instr, Set<String> loopVariables) {
            for (String var : loopVariables) {
                if (instr.getClass().toString().contains(var)) {
                return false;
            }
        }
        return true;
    }


    private Set<String> collectLoopVariables(CodeObject loopCode) {
        Set<String> vars = new HashSet<>();
        for (Instruction instr : loopCode.code) {
            if (instr.getDest() != null) {
                vars.add(instr.getDest());
            }
        }
        return vars;
    }
```



```java
@Override
protected CodeObject postprocess(WhileNode node, CodeObject cond, CodeObject slist) {
    //Step 0:
    CodeObject co = new CodeObject();

    /* FILL IN FROM STEP 3*/
    String loopLabel = generateLoopLabel();
    String outLabel = generateOutLabel();

    co.code.add(new Label(loopLabel));
    co.code.addAll(cond.code);

    Set<String> loopVariables = collectLoopVariables(slist);

    CodeObject optimizedSlist = new CodeObject();
    for (Instruction instr : slist.code) {
        if (isLoopInvariant(instr, loopVariables)) {
            co.code.add(instr);
        } else {
            optimizedSlist.code.add(instr);
        }
    }
```

Test Cases:

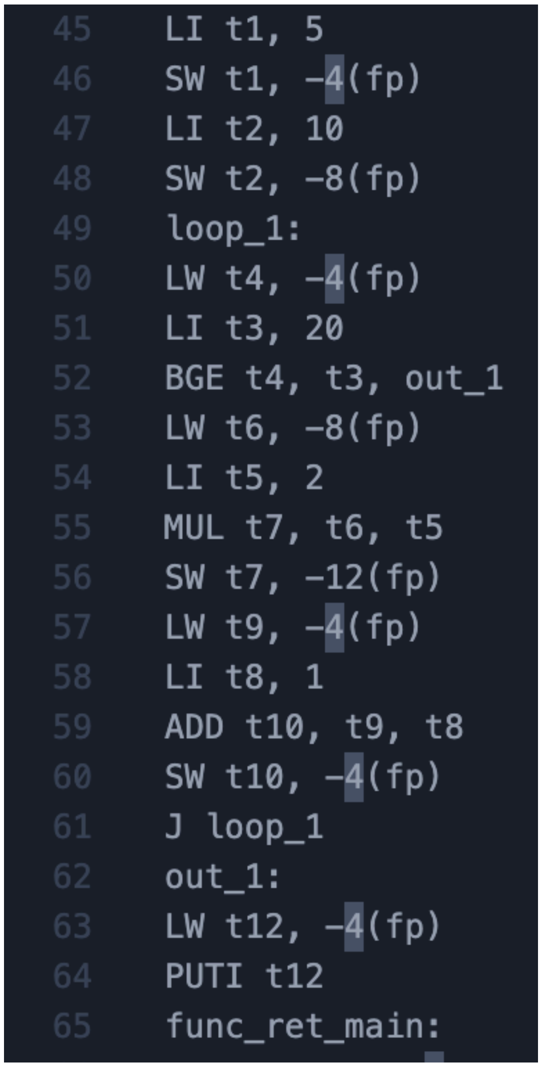
Test Case #0 (test0.uC):



tests > test0.uC

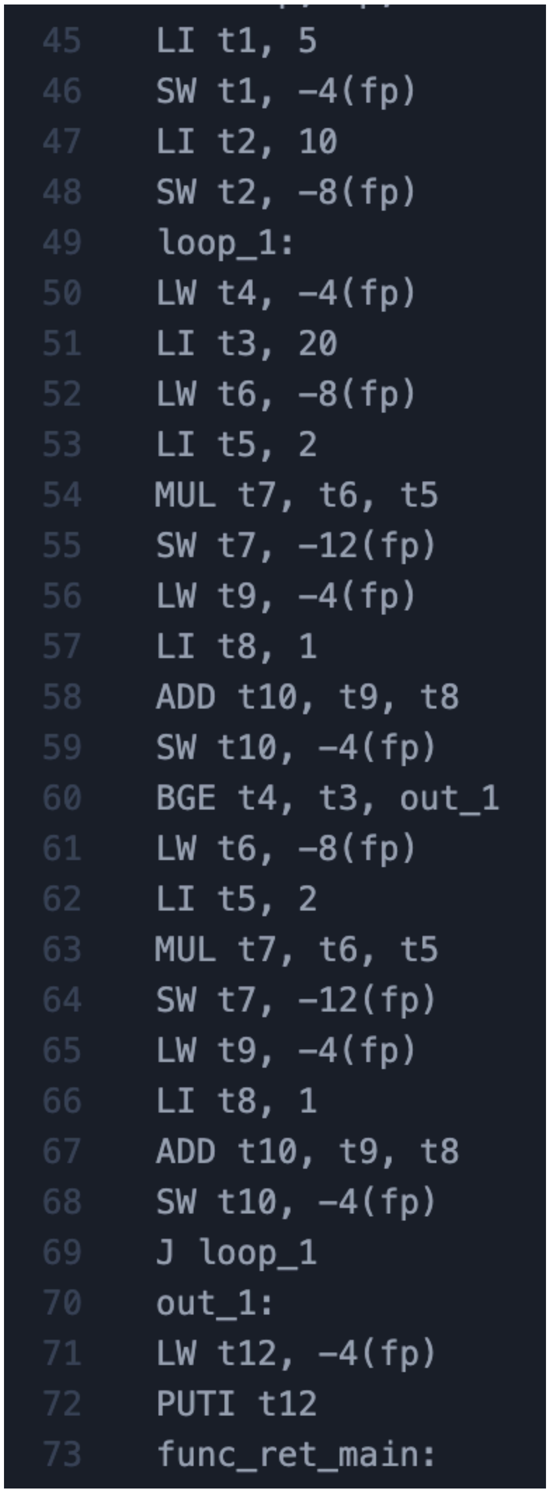
```
int main() {

    int a;
    int b;
    int c;

    a = 5;
    b = 10;

    while (a < 20) {
        c = b * 2;  /* Loop-invariant */
        a = a + 1;
    }

    print(a);
}
```

Before Optimization Assembly:



```
LI t1, 5
SW t1, -4(fp)
LI t2, 10
SW t2, -8(fp)
loop_1:
LW t4, -4(fp)
LI t3, 20
BGE t4, t3, out_1
LW t6, -8(fp)
LI t5, 2
MUL t7, t6, t5
SW t7, -12(fp)
LW t9, -4(fp)
LI t8, 1
ADD t10, t9, t8
SW t10, -4(fp)
J loop_1
out_1:
LW t12, -4(fp)
PUTI t12
func_ret_main:
```

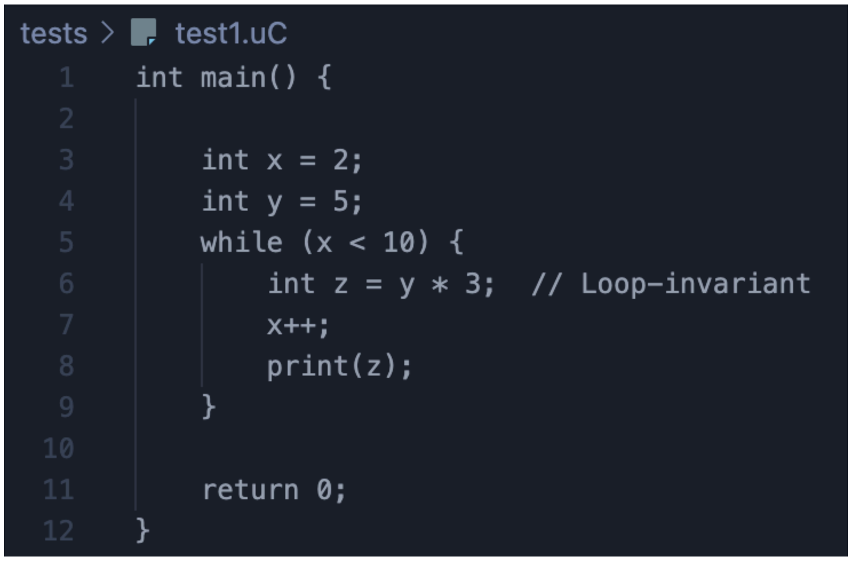
Optimized Assembly:



```
LI t1, 5
SW t1, -4(fp)
LI t2, 10
SW t2, -8(fp)
loop_1:
LW t4, -4(fp)
LI t3, 20
LW t6, -8(fp)
LI t5, 2
MUL t7, t6, t5
SW t7, -12(fp)
LW t9, -4(fp)
LI t8, 1
ADD t10, t9, t8
SW t10, -4(fp)
BGE t4, t3, out_1
LW t6, -8(fp)
LI t5, 2
MUL t7, t6, t5
SW t7, -12(fp)
LW t9, -4(fp)
LI t8, 1
ADD t10, t9, t8
SW t10, -4(fp)
J loop_1
out_1:
LW t12, -4(fp)
PUTI t12
func_ret_main:
```

—---

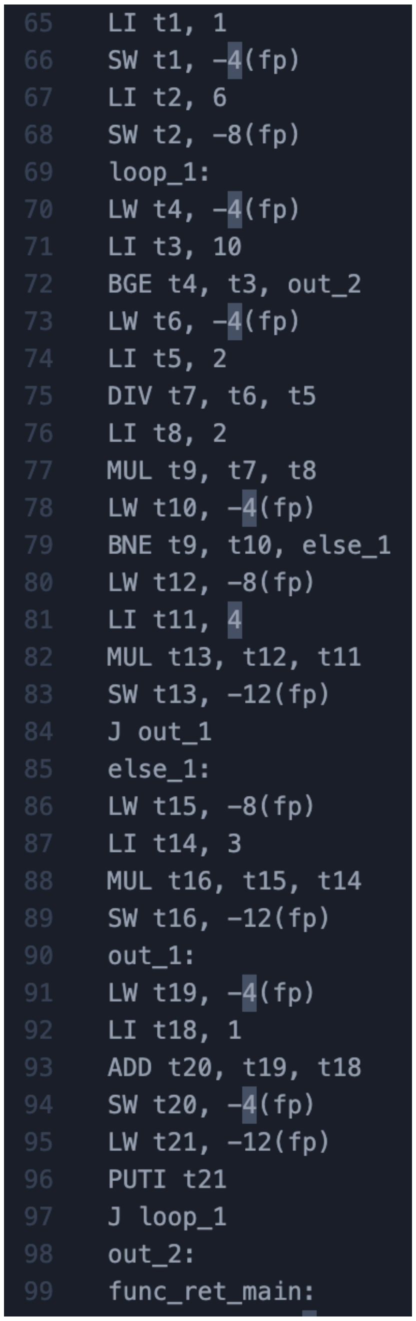
Test Case #2 (test2.uC):



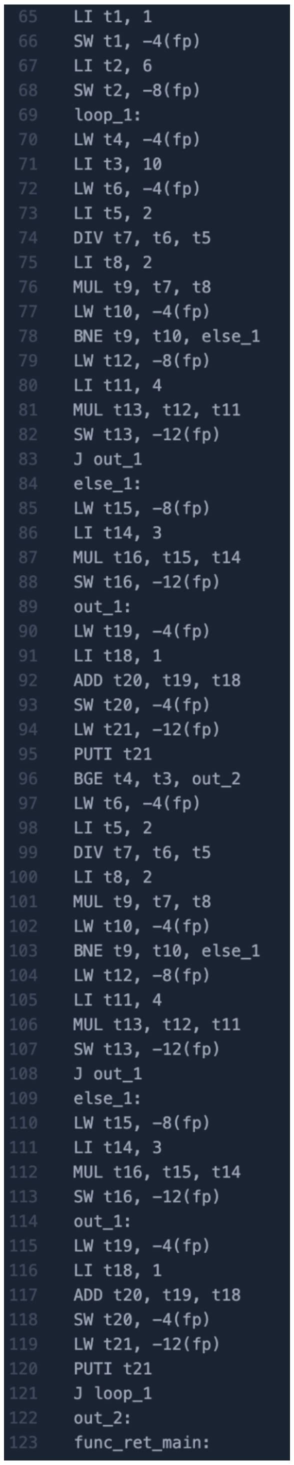
tests > test1.uC

```
int main() {

    int x = 2;
    int y = 5;
    while (x < 10) {
        int z = y * 3;  // Loop-invariant
        x++;
        print(z);
    }

    return 0;
}
```

Generated Assembly (Before Optimization):


```
LI t1, 1
SW t1, -4(fp)
LI t2, 6
SW t2, -8(fp)
loop_1:
LW t4, -4(fp)
LI t3, 10
BGE t4, t3, out_2
LW t6, -4(fp)
LI t5, 2
DIV t7, t6, t5
LI t8, 2
MUL t9, t7, t8
LW t10, -4(fp)
BNE t9, t10, else_1
LW t12, -8(fp)
LI t11, 4
MUL t13, t12, t11
SW t13, -12(fp)
J out_1
else_1:
LW t15, -8(fp)
LI t14, 3
MUL t16, t15, t14
SW t16, -12(fp)
out_1:
LW t19, -4(fp)
LI t18, 1
ADD t20, t19, t18
SW t20, -4(fp)
LW t21, -12(fp)
PUTI t21
J loop_1
out_2:
func_ret_main:
```

<u>Optimized Assembly:</u>


```
LI t1, 1
SW t1, -4(fp)
LI t2, 6
SW t2, -8(fp)
loop_1:
LW t4, -4(fp)
LI t3, 10
LW t6, -4(fp)
LI t5, 2
DIV t7, t6, t5
LI t8, 2
MUL t9, t7, t8
LW t10, -4(fp)
BNE t9, t10, else_1
LW t12, -8(fp)
LI t11, 4
MUL t13, t12, t11
SW t13, -12(fp)
J out_1
else_1:
LW t15, -8(fp)
LI t14, 3
MUL t16, t15, t14
SW t16, -12(fp)
out_1:
LW t19, -4(fp)
LI t18, 1
ADD t20, t19, t18
SW t20, -4(fp)
LW t21, -12(fp)
PUTI t21
BGE t4, t3, out_2
LW t6, -4(fp)
LI t5, 2
DIV t7, t6, t5
LI t8, 2
MUL t9, t7, t8
LW t10, -4(fp)
BNE t9, t10, else_1
LW t12, -8(fp)
LI t11, 4
MUL t13, t12, t11
SW t13, -12(fp)
J out_1
else_1:
LW t15, -8(fp)
LI t14, 3
MUL t16, t15, t14
SW t16, -12(fp)
out_1:
LW t19, -4(fp)
LI t18, 1
ADD t20, t19, t18
SW t20, -4(fp)
LW t21, -12(fp)
PUTI t21
J loop_1
out_2:
func_ret_main:
```

Results:

The optimization eliminates redundant computations from the loop, reducing the instruction count within the loop. This leads to faster execution since the invariant code is computed only once.

Conclusion:

Loop-invariant code motion is a simple yet effective optimization technique that improves runtime performance by minimizing redundant computations in loops. It complements other optimizations, such as strength reduction or common subexpression elimination, to further enhance efficiency.

By applying this optimization, the compiler generates more efficient code, making programs faster and reducing resource usage.