CS179E: Project in Computer Science – Compilers
Document for Phase 4

Jiamin Pan, Yunqing Xiao

1 Introduction

In phase 3, we accomplished register allocation, which is an essential prerequisite for us to eventually obtain assembly language. In fact, the Vapor-M source file we derived in phase 3 is sufficient for us to complete the final translation. In this document, we will introduce some details about the final work.

2 Dependencies

1. JDK version = JDK-11.0.2.
2. Vapor parser. [1]
3. MIPS interpreter.

3 Methods for Final Translation

3.1 Register Operations for Function Call

Every time when we enter a function, we need to store some registers and restore them when we are about to exit. When we enter a function, we complete the following steps:

1. Save \$fp to location \$sp - 2.
2. Move \$fp to \$sp.
3. Decrease \$sp by the size which is equal to the size of local stack and out stack plus 2. The reason why we plus 2 is that we need to store return address and frame pointer here.
4. Save the return register at \$fp - 1.

Suppose the size we use to decrease \$sp is *k*. We can compute *k* easily since we can directly get the size of stacks from the frame of the function. The assemly code for this register operation is

```
sw $fp -8($sp)
move $fp $sp
subu $sp $sp 4k
sw $ra -4($fp)
```

[1]It can be downloaded from the homepage.

Similarly, we will restore these registers when we are about to exit the function, so the step is

1. Restore the return register $ra from $fp - 1.
2. Restore $fp from $fp - 2.
3. Increase $sp by the size *k*.
4. Jump to the return register.

We can perceive that the sequence of restoring is the inverse of the storing sequence. Hence, the assembly code of these steps are

```
lw $ra -4($fp)
lw $fp -8($fp)
addu $sp $sp 4k
jr $ra
```

These works will be done before or after we traverse the body of the function. We will also need a printer to print these statements.

3.2 Type of Literals

The translation of the function body is analogous to the previous translation. We just need to pay attention the type of the statement and create a new visitor to deal with them separately. The general idea is easy, but there is also something we need to pay attention to.

In assign statements, the source could a label, a register, or an integer. We are supposed to treat them differently. Similarly, we will also face this problem in built-in statements. For example, the arguments of Sub statement could also be a register or an integer. If both of them are integers, we just need to compute them at the beginning and translate this line to an “li” assignment statement instead of a “sub” arithmetic statement. If the first argument is an integer, we should store it in $t9 before we call “sub” because “sub” will not accept an immediate value as the substracted part. The implementation is like this:

```
if (opName == "Sub") {
    String dest = stmt.dest.toString(); // Store the destination
    if (stmt.args[0].getClass() != stmt.args[1].getClass()) {   // If one of the arguments is an
    integer
        if (stmt.args[0] instanceof VLitInt) {
            printer.println("li␣$t9␣" + stmt.args[0].toString());   // Store the integer in $t9
            String num = stmt.args[1].toString();
            printer.println("sub␣" + dest + "␣$t9␣" + num);
        } else {
            String reg = stmt.args[0].toString();
            String num = stmt.args[1].toString();
            printer.println("sub␣" + dest + "␣" + reg + "␣" + num); // Direct translation
        }
    } else {
        if (stmt.args[0] instanceof VLitInt) {
            int num1 = ((VLitInt)stmt.args[0]).value;
            int num2 = ((VLitInt)stmt.args[1]).value;
            int num = num1 - num2;  // We compute it before the assignment
            printer.println("li␣" + dest + "␣" + Integer.toString(num));
        } else {
            String reg1 = stmt.args[0].toString();
            String reg2 = stmt.args[1].toString();
            printer.println("sub␣" + dest + "␣" + reg1 + "␣" + reg2);   // Direct translation
        }
    }
}
```

The other types of built-in operations are analogous. As long as we could cope with this problem carefully in our code, we will avoid many unexpected faults in the program. The translation of other statements are similar.

4 Review for Phase 4

```
==== Results ====
Passed 16/16 test cases
- Submission Size = 13 kB
```

To be honest, we did not expect that this phase should be so easy, but it was. Actually we spent just around five hours to accomplish this extra phase. God blessing, we passes all the test cases. This success makes a superior end for our project. We completed all four phases from the beginning to the end, and we do admit that we have benefited a lot from them.

Here we would like to thank our professor, Mr. Mohsen Lesani, and our teaching assistant, Miss Lian Gao, again for their help and consideration. This project is fantastic!

References

[1] Course's Homepage: Compiler Project.

[2] Andrew W. Appel, Jens Palsberg, *Modern Compiler Implementation in Java*, Cambridge University Press, Second Edition, 2002.