Code Generator

Zakia Zinat Choudhury
Lecturer
Department of Computer Science & Engineering
University of Rajshahi

1

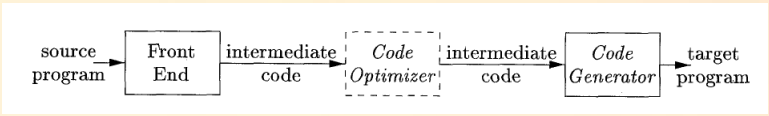
Code Generation

- Code generation can be considered as the final phase of compilation.
- Through post code generation, optimization process can be applied on the code, but that can be seen as a part of code generation phase itself. The code generated by the compiler is an object code of some lower-level programming language, for example, assembly language. We have seen that the source code written in a higher-level language is transformed into a lower-level language that results in a lower-level object code, which should have the following minimum properties:
  - It should carry the exact meaning of the source code.
  - It should be efficient in terms of CPU usage and memory management.

2

Code Generation

- The final phase in compiler model is the code generator. It takes as input the intermediate representation (IR) produced by the front end of the compiler, along with relevant symbol table information, and produces as output a semantically equivalent target program, as shown in below:



- A code generator has three primary tasks: **instruction selection, register allocation and assignment, and instruction ordering**.
- Instruction selection involves choosing appropriate target-machine instructions to implement the IR statements. Register allocation and assignment involves deciding what values to keep in which registers. Instruction ordering involves deciding in what order to schedule the execution of instructions.

3

Issues in the Design of a Code Generator

- **Target language** : The code generator has to be aware of the nature of the target language for which the code is to be transformed. That language may facilitate some machine-specific instructions to help the compiler generate the code in a more convenient way. The target machine can have either CISC or RISC processor architecture.
- **IR Type** : Intermediate representation has various forms. It can be in Abstract Syntax Tree (AST) structure, Reverse Polish Notation, or 3-address code.
- **Selection of instruction** : The code generator takes Intermediate Representation as input and converts (maps) it into target machine's instruction set. One representation can have many ways (instructions) to convert it, so it becomes the responsibility of the code generator to choose the appropriate instructions wisely.
- **Register allocation** : A program has a number of values to be maintained during the execution. The target machine's architecture may not allow all of the values to be kept in the CPU memory or registers. Code generator decides what values to keep in the registers. Also, it decides the registers to be used to keep these values.
- **Ordering of instructions** : At last, the code generator decides the order in which the instruction will be executed. It creates schedules for instructions to execute them.

4

Peephole Optimization

- While most production compilers produce good code through careful instruction selection and register allocation, a few use an alternative strategy: they generate naive code and then improve the quality of the target code by applying "optimizing" transformations to the target program. The term "optimizing" is somewhat misleading because there is no guarantee that the resulting code is optimal under any mathematical measure.
- A simple but effective technique for locally improving the target code is peephole optimization, which is done by examining a sliding window of target instructions (called the peephole) and replacing instruction sequences within the peephole by a shorter or faster sequence, whenever possible. Peephole optimization can also be applied directly after intermediate code generation to improve the intermediate representation.

5

Peephole Optimization

A bunch of statements is analyzed and are checked for the following possible optimization:

- **Eliminating Redundant Loads and Stores**

At compilation level, the compiler searches for instructions redundant in nature. Multiple loading and storing of instructions may carry the same meaning even if some of them are removed.

For example:

```
MOV x, R0
MOV R0, R1
```

We can delete the first instruction and re-write the sentence as:

```
MOV x, R1:
```

6

Peephole Optimization

- **Eliminating Unreachable Code**

Unreachable code is a part of the program code that is never accessed because of programming constructs. Programmers may have accidently written a piece of code that can never be reached.

For example,

```
void add_ten(int x)
{
   return x + 10;
   printf("value of x is %d", x);
}
```

In this code segment, the **printf** statement will never be executed as the program control returns back before it can execute, hence **printf** can be removed.

7

Peephole Optimization

- **Flow of Control Optimization**

There are instances in a code where the program control jumps back and forth without performing any significant task. These jumps can be removed. Consider the following chunk of code:

For example:

```
...
MOV R1, R2
GOTO L1
...
L1 :   GOTO L2
L2 :   INC R1
```

In this code, label L1 can be removed as it passes the control to L2. So instead of jumping to L1 and then to L2, the control can directly reach L2, as shown below:

```
...
MOV R1, R2
GOTO L2
...
L2 :   INC R1
```

8

Peephole Optimization

- **Algebraic Expression Simplification**

There are occasions where algebraic expressions can be made simple.

For example,

The expression a = a + 0 can be replaced by a itself and the expression a = a + 1 can simply be replaced by INC a.

- **Strength Reduction**

There are operations that consume more time and space. Their 'strength' can be reduced by replacing them with other operations that consume less time and space but produce the same result.

For example,

The output of a * a and a^2 is same, a^2 is much more efficient to implement.

9

Peephole Optimization

- **Accessing Machine Instructions**

The target machine may have hardware instructions to implement certain specific operations efficiently. Detecting situations that permit the use of these instructions can reduce execution time significantly.

For example,

Some machines have auto-increment and auto-decrement addressing modes. These add or subtract one from an operand before or after using its value. The use of the modes greatly improves the quality of code.

10