Code Generation

Introduction to Code Generation

Compiler Design is an essential aspect of software engineering that enables us to translate high-level programming languages into machine-readable code. One of the most important tasks in compiler design is code generation, which involves transforming the intermediate code generated by the compiler into efficient machine code and challenging process that requires a deep understanding of the target architecture and the programming language being compiled. In this blog, we will explore the principles and techniques of code generation in compiler design, and discuss how compilers can optimize code to produce faster and more efficient programs. Whether you are a seasoned programmer or a curious learner, this blog will provide you with valuable insights into the world of compiler design and code generation.

Code generation can be considered as the final phase of the compilation. It takes as input the intermediate representation (IR) produced by the front end of the compiler, along with the relevant symbol table information, and produces as output a semantically equivalent target program, as shown in Fig 1.

Figure 1. Position of code generator

The requirement imposed on a code generator are severe. The Target Program must preserve the semantic meaning of the source program and be of high quality, that it must make effective use of the available resources of the target machine. Through post code generation, optimization process can be applied on the code, but that can be sees 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. While transforming into a lower-level language that results in a lower-level object code should have following properties:

- It should carry the exact meaning of the source code.
- It should be efficient in terms of CPU usage and memory management.

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 instruction.

Input to the Code Generator

The input to the code generator is the intermediate representation of the source program produced by the front end, along with information in the symbol table that is used to determine the run-time addresses of the data objects denoted by the names in the IR. Intermediate Representation include three-address representation such as quadruples, triples, indirect triples: virtual machine representations such as postfix notation; and graphical representation such as abstract syntax trees (AST) and DAG's.

The Target Program

The architecture of target machine has a significant impact on the difficulty of constructing a good code generator that produces high-quality machine code. The most common target-machine architectures are reduced instruction set computer (RISC), complex instruction set computer (CISC), and stack based.

A RISC machine typically has many registers, three-address instructions, simple addressing modes, and a relatively simple instruction-set architecture. In contrast, a CISC machine typically has few registers, two-address instructions, a variety of addressing modes, several register classes, variable-length instructions and instructions with side effects. One example of a RISC architecture is the ARM architecture, which is commonly used in mobile devices and embedded systems. CISC architecture is the x86 architecture, which is commonly used in desktop and laptop computers.

In a stack-based machine, operations are done by pushing operands onto a stack and then performing the operations on the operands at the top of the stack. Stack-based machines almost disappeared because it was felt that the stack was too limiting and required too many swap and copy operations. However, they were revived with the introduction of Java Virtual Machine.

Instruction Selection

The code generator takes intermediate representation as input and converts it into target machine's instruction set. One representation can have many ways to convert it, so it becomes the responsibility of the code generator to choose the appropriate instruction wisely. The complexity of performing this mapping is determined by a factor such as:

- The level of the Intermediate Representations
- The nature of the instruction-set architecture
- The desired quality of the generated code

The nature of the instruction set of the target machine has a strong effect on the difficulty of instruction selection. For example, the uniformity and completeness of the instruction set are important factors. On some machines, floating-points operations are done using separate registers if target machine does not support each data type in the uniform manner, then each exception to general rule require special handling.

Register Allocation

A key problem in code generation is deciding what values to hold in what registers. A program has a number of values to be maintained during the execution. The targets 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. The use of registers is often subdivided into two subproblems:

1. *Register allocation,* during which we select the set of variables that will reside in registers at each point in the program.
2. *Register assignment,* during which we pick the specific register that a variable will reside in.

Evaluation Order

The order in which computations are performed by the code generator. It creates schedules for instruction to execute them.

A simple Target Machine Model

Our target computer models a three-address machine with load and store operations, computation operations, jump operations, and conditional jumps.

- *Load* operations: The instruction LD *dst, addr* loads the values in location *addr* into location *dst*. This instruction denotes the assignment *dst=addr.* The most common form of this instruction is LD *r, x* which loads the value in location *x* into register *r.*
- *Store* operations: This instruction ST *x,r* stores the value in register *r* into the location *x*. This instruction denotes the assignment *x=r*.
- *Computation* operation of the form OP *dst, src1, src2*, where OP is a operator like ADD or SUB, and *dst, src1, and src2* are locations. For examples, SUB *r1, r2, r3* computes *r1=r2-r3.*
- *Unconditional jumps*: The instruction BR *L* causes control to branch to the machine instruction with label *L.* (BR stands for *branch.*)
- *Conditional jumps* of the form B*cond r, L,* where *r* is a register, *L* is a label, and *cond* stands for any of the common tests on values in the register *r.*

Now we will see how the intermediate code is transformed into target object code (assembly code).

Directed Acyclic Graph

Directed Acyclic Graph (DAG) is a tool that depicts the structure of basic blocks, helps to see the flow of values flowing among the basic blocks, and offers optimization too. DAG provides easy transformation on basic blocks.

Example 1: The two-statement sequence

$x = b * c$

$y = a + x$

The corresponding intermediate code (using TAC) would be:

$t_0 = b * c$

$t_1 = a + t_0$

$y = t_1$

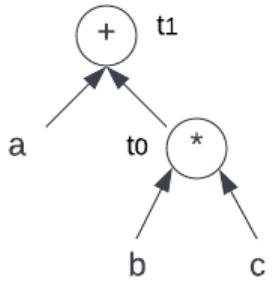

Figure 2. DAG for basic block in example 1

The machine-code equivalent would be something like:

LD R_0, b

LD R_1, c

MUL R_0, R_0, R_1

LD R_2, a

ADD R_2, R_2, R_0

ST y, R_2

Example 2: Three Statement sequence

$t_0 = a + b$

$t_1 = t_0 + c$

$d = t_0 + t_1$

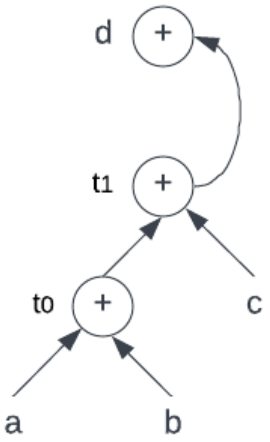

Figure 3. DAG for basic block in example 2

The machine-code equivalent would be something like:

LD R_0, a	// R_0 = a
ADD R_o, R_o, b	// R_0 = R_0+b
LD R_1, c	// R_1 = c
ADD R_1, R_1, R_0,	// R_1 = R_1+R_0
ST d, R_1	// d = R_1

Example 3: The four-statement sequence assuming a and b are arrays whose elements are 4-byte values.

x = a [i]
y = b [j]
a [i] = y
b [j] = x

The corresponding intermediate code (using TAC) would be:

t_0 = 4 * i
t_1 = a[t_0]
t_2 = 4 * j
t_3 = b[t_2]
t_1 = t_3
t_3 = t_1

The machine-code equivalent would be something like:

LD R_0, i
MUL R_0, R_0, #4
LD R_1, a[R_0]
LD R_2, j
MUL R_2, R_2, #4
LD R_3, b[R_2]
ST a[R_0], R_3
ST b[R_2], R_1

Example 4: The four-statement sequence.

a = b + c
b = a – d
c = b + c
d = a – d

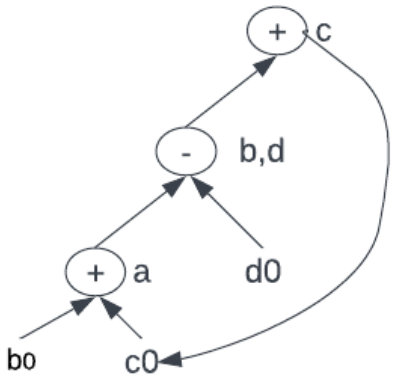

Figure 4. DAG for basic block in example 4

The machine-code equivalent would be something like:

LD R0, a
LD R1, b
LD R2, c
ADD R0, R2, R3
ST a, R0
LD R3, d
SUB R1, R0, R3

```
ADD R2, R1, R2
SUB R3, R0, R3
ST d, R3
```

In conclusion, code generation is an important aspect of compiler design that involves transforming high-level source code into low-level executable code. Code generators automate this process and can be used to generate code for a wide range of applications and programming languages. Examples of code generators include compiler-based code generators, JIT code generators, domain-specific language (DSL) code generators, model-driven code generators, and source-to-source code generators. Each type of code generator has its own strengths and weaknesses and is suited for different types of applications and development environments. Understanding code generation is essential for building efficient and optimized compilers and programming languages.