5.1 ISSUES IN THE DESIGN OF A CODE GENERATOR

A Code Generator is an important phase of a compiler that converts the Intermediate Representation (IR) of a program into target machine code or assembly code.

The main goal of a code generator is to produce correct and efficient target code with minimum execution time and memory usage.

Basic Flow of Code Generation

Source Program

|

Lexical Analysis

|

Syntax Analysis

|

Semantic Analysis

|

Intermediate Code

|

Code Generator

|

Target / Machine Code

The design of a code generator involves several important issues that affect the correctness, speed, memory usage, and quality of the generated code.

1. Input to the Code Generator

The first issue is deciding the form of input that the code generator receives.

The input is usually an Intermediate Representation (IR), such as:

- Three-address code
- Syntax tree
- DAG (Directed Acyclic Graph)
- Quadruples
- Triples

Example:

Three-address code:

t1 = a + b

t2 = t1 * c

Target code:

MOV R1, a

ADD R1, b

MUL R1, c

Therefore, the code generator must understand the structure and meaning of the intermediate representation.

2. Target Machine

The code generator must consider the characteristics of the target machine.

Different processors have different:

- Instruction sets.
- Registers.
- Addressing modes.
- Memory organizations.
- Instruction formats.

Example:

One machine may have registers R0, R1, R2, R3, while another machine may have a different register architecture.

Therefore, the generated code must be suitable for the target machine.

3. Instruction Selection

Instruction selection means choosing suitable machine instructions for each intermediate-code operation.

Example:

Intermediate code:

x = a + b

Target code:

MOV R1, a

ADD R1, b

MOV x, R1

The code generator must select instructions that perform the required operation correctly and efficiently.

Good instruction selection should:

- Use fewer instructions.
- Execute quickly.
- Make effective use of machine instructions.
- Use special instructions provided by the processor.

4. Register Allocation

Registers are small and fast storage locations inside the CPU.

Since the number of registers is limited, the code generator must decide which variables and temporary values should be stored in registers and which should be stored in memory.

Example:

Suppose the machine has only four registers:

R0 R1 R2 R3

But the program needs many variables:

a, b, c, d, e, f, g

The code generator must decide which variables should remain in registers and which should be stored in memory.

Importance:

- Reduces memory access.
- Improves execution speed.
- Produces efficient machine code.

5. Register Assignment

Register allocation decides which values should be placed in registers, while register assignment decides which particular register is used.

Example:

a → R1

b → R2

t1 → R3

The code generator must ensure that the same register is not incorrectly assigned to two values that are needed at the same time.

6. Evaluation Order

The code generator must determine the order in which expressions should be evaluated.

Example:

Expression:

x = (a + b) * (c + d)

Intermediate code:

t1 = a + b

t2 = c + d

x = t1 * t2

The evaluation order should try to:

- Minimize the number of registers required.
- Reduce temporary storage.
- Reduce memory operations.
- Produce efficient code.

7. Memory Management

The code generator must decide how variables and temporary values are stored in memory.

Memory may be required for:

- Local variables.
- Global variables.
- Arrays.
- Structures.
- Function parameters.
- Temporary values.

Example:

int a;

int b;

int c;

The code generator must assign appropriate memory locations or stack locations to these variables.

8. Addressing Modes

Modern processors support different addressing modes to access data from memory and registers.

Common addressing modes include:

1. Immediate addressing.
2. Register addressing.
3. Direct addressing.
4. Indirect addressing.
5. Indexed addressing.

Example:

MOV R1, #10

Here, #10 represents an immediate value.

Using the correct addressing mode can reduce the number of instructions and improve performance.

9. Instruction Scheduling

Instruction scheduling determines the order of machine instructions to improve processor performance.

If some instructions can be executed independently, the code generator can arrange them in an efficient order.

Objectives:

- Reduce delays.
- Improve CPU utilization.
- Reduce pipeline stalls.

- Improve execution speed.

10. Handling Control Flow

The code generator must correctly translate control statements such as:

- if
- if-else
- while
- for
- switch
- break
- continue

It must also handle function calls and jump instructions.

Example:

Intermediate code:

if a < b goto L1

goto L2

L1: x = 10

L2:

Possible target code:

CMP a, b

JL L1

JMP L2

L1: MOV x, 10

L2:

Correct labels and jump instructions are essential for proper program execution.

11. Procedure and Function Calls

The code generator must correctly handle function calls.

It has to manage:

- Passing arguments.

- Returning values.
- Saving registers.
- Creating activation records.
- Managing the stack.
- Returning control to the calling function.

Example:

result = add(a, b)

The generated code must arrange a and b according to the target machine's calling convention and obtain the returned result.

12. Code Quality

Another important issue is the quality of generated code.

Good generated code should:

- Be correct.
- Execute quickly.
- Use less memory.
- Contain fewer instructions.
- Make efficient use of registers.

There is often a trade-off between compilation time and code quality. More optimization may produce better code but can require more compilation time.

13. Machine-Dependent and Machine-Independent Issues

Some code-generation decisions depend on the target machine, while others do not.

Machine-Dependent Issues:

- Number of registers.
- Instruction set.
- Addressing modes.
- Processor architecture.

Machine-Independent Issues:

- Basic optimization.
- Expression analysis.

- Intermediate-code organization.

A well-designed compiler separates these concerns whenever possible.