CODE GENERATION

The final phase in compiler model is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program. The code generation techniques presented below can be used whether or not an optimizing phase occurs before code generation.

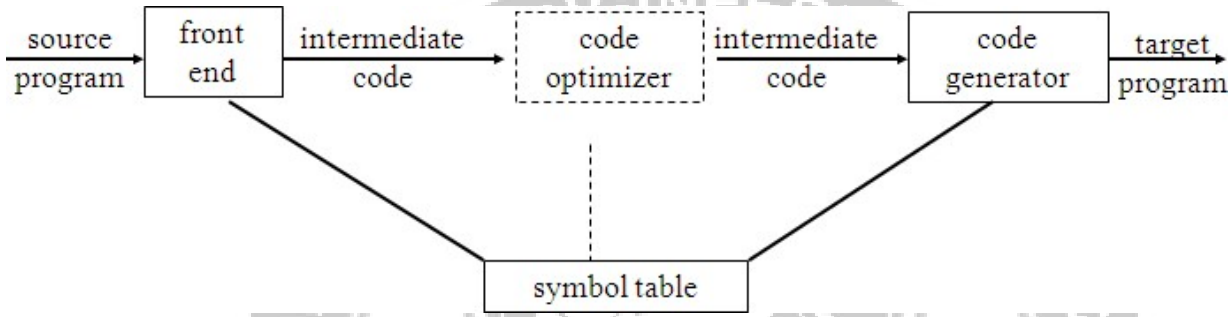

Fig. Position of code generator

ISSUES IN THE DESIGN OF A CODE GENERATOR

The following issues arise during the code generation phase:

1. Input to code generator
2. Target program
3. Memory management
4. Instruction selection
5. Register allocation
6. Evaluation order

1. Input to code generator:

• The input to the code generation consists of the intermediate representation of the source program produced by front end , together with information in the symbol table to determine run- time addresses of the data objects denoted by the names in the intermediate representation.

• Intermediate representation can be :

a. Linear representation such as postfix notation
b. Three address representation such as quadruples
c. Virtual machine representation such as stack machine code
d. Graphical representations such as syntax trees and dags.

• Prior to code generation, the front end must be scanned, parsed and translated into intermediate representation along with necessary type checking. Therefore, input to code generation is assumed to be error-free.

2. Target program:

• The output of the code generator is the target program. The output may be :

a. Absolute machine language

- It can be placed in a fixed memory location and can be executed immediately.

b. Relocatable machine language

- It allows subprograms to be compiled separately.

c. Assembly language

- Code generation is made easier.

3. Memory management:

• Names in the source program are mapped to addresses of data objects in run-time memory by the front end and code generator.

• It makes use of symbol table, that is, a name in a three-address statement refers to a symbol- table entry for the name.

• Labels in three-address statements have to be converted to addresses of instructions. For example,

j:gotoigenerates jump instruction as follows:

* if i < j, a backward jump instruction with target address equal to location of code for quadruple i is generated.

* if i > j, the jump is forward. We must store on a list for quadruple i the location of the first machine instruction generated for quadruple j. When i is processed, the machine locations for all instructions that forward jumps to i are filled.

4. Instruction selection:

• The instructions of target machine should be complete and uniform.

• Instruction speeds and machine idioms are important factors when efficiency of target program is considered.

• The quality of the generated code is determined by its speed and size.

• The former statement can be translated into the latter statement as shown below:

```
a:=b+c
d:=a+e          (a)

MOV  b,R0  ADD
c,R0
MOV R0,a        (b)
MOV a,R0
ADD  e,R0  MOV
R0,d
```

5. Register allocation

• Instructions involving register operands are shorter and faster than those involving operands in memory. The use of registers is subdivided into two subproblems :

1. Register allocation - the set of variables that will reside in registers at a point in the program is selected.

2. Register assignment - the specific register that a value picked.

- Certain machine requires even-odd register pairs for some operands and results.

For example , consider the division instruction of the form :D x, y where, x - dividend even register in even/odd register pair y-divisor even register holds the remainder odd register holds the quotient

6. Evaluation order

- The order in which the computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others.