MCA 18 302
PRINCIPLES OF COMPILERS

MODULE 5

1. **Code generation**
   a) Issues in the design of a code generator
   b) The target language
      i. A simple target machine model
      ii. The program and instruction costs
   c) Address in the target code
      i. Static allocation
      ii. Stack allocation
      iii. Run-time address for names
   d) Basic blocks and flow graphs
      i. Representation of flow graphs.

2. Code optimization

a) The principal sources of optimization

b) Data flow analysis

   i. Data flow analysis Abstraction

   ii. Data flow analysis schema

   iii. Data flow schemas on basic blocks

   iv. Reaching definitions

   v. live variable analysis

   vi. Available expressions.

c) Region based analysis

   i. Regions

   ii. Region hierarchies for reducible flow graphs

   iii. Overview of a region based analysis.

Issues in the design of a code generator

1. Input to code generator:

- The input to code generator is the intermediate code generated by the front end, along with information in the symbol table that determines the run-time addresses of the data-objects denoted by the names in the intermediate representation.
- Intermediate codes may be represented mostly in quadruples, triples, indirect triples, Postfix notation, syntax trees, DAG's, etc.
- The code generation phase just proceeds on an assumption that the input are free from all of syntactic and state semantic errors, the necessary type checking has taken place and the type-conversion operators have been inserted wherever necessary.

2. Target program:

- The target program is the output of the code generator.
- The output may be absolute machine language, relocatable machine language, assembly language.

  a) **Absolute machine language** as output has advantages that it can be placed in a fixed memory location and can be immediately executed.

b) **Relocatable machine languag**e as an output allows subprograms and subroutines to be compiled separately. Relocatable object modules can be linked together and loaded by linking loader. But there is added expense of linking and loading.

c) **Assembly language** as output makes the code generation easier. We can generate symbolic instructions and use macro-facilities of assembler in generating code. And we need an additional assembly step after code generation.

3. Memory Management :

- Mapping the names in the source program to the addresses of data objects is done by the front end and the code generator.
- A name in the three address statements refers to the symbol table entry for name. Then from the symbol table entry, a relative address can be determined for the name.

4. Instruction selection

- Selecting the best instructions will improve the efficiency of the program.
- It includes the instructions that should be complete and uniform.
- Instruction speeds and machine idioms also plays a major role when efficiency is considered. But if we do not care about the efficiency of the target program then instruction selection is straight-forward.

- For example, the respective three-address statements would be translated into the latter code sequence as shown below:

```
P:=Q+R
S:=P+T
```

```
MOV Q, R0
ADD R, R0
MOV R0, P
MOV P, R0
ADD T, R0
MOV R0, S
```

- Here the fourth statement is redundant as the value of the P is loaded again in that statement that just has been stored in the previous statement. It leads to an inefficient code sequence. A given intermediate representation can be translated into many code sequences, with significant cost differences between the different implementations. A prior knowledge of instruction cost is needed in order to design good sequences, but accurate cost information is difficult to predict.

5. Register allocation issues:

- Use of registers make the computations faster in comparison to that of memory, so efficient utilization of registers is important.
- The use of registers are subdivided into two sub problems:
  1. During **Register allocation ,** we select only those set of variables that will reside in the registers at each point in the program.
  2. During a subsequent **Register assignment** phase, the specific register is picked to access the variable.

- As the number of variables increases, the optimal assignment of registers to variables becomes difficult. Mathematically, this problem becomes NP-complete. Certain machine requires register pairs consist of an even and next odd-numbered register.
- For example

  M a, b

  These types of multiplicative instruction involve register pairs where the multiplicand is an even register a and b, the multiplier is the odd register of the even/odd register pair.

6. Evaluation order:

- The code generator decides the order in which the instruction will be executed.
- The order of computations affects the efficiency of the target code.
- Among many computational orders, some will require only fewer registers to hold the intermediate results. However, picking the best order in the general case is a difficult NP-complete problem.

7. Approaches to code generation issues:

- ➢ Code generator must always generate the correct code. It is essential because of the number of special cases that a code generator might face.
- ➢ Some of the design goals of code generator are:

1. Correct
2. Easily maintainable
3. Testable
4. Efficient