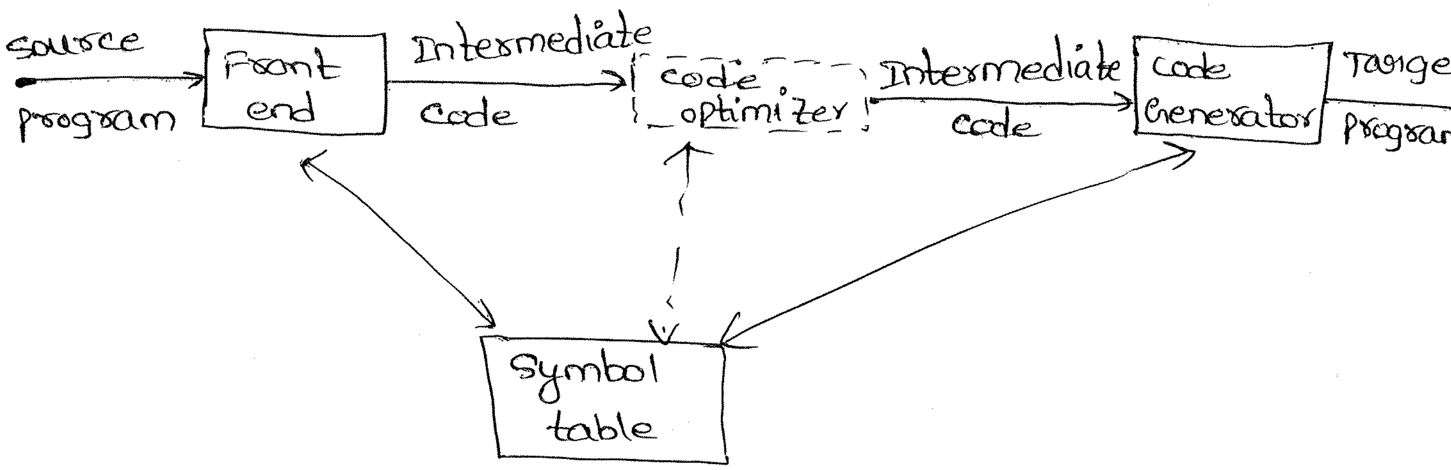
Unit – V

Code Generation

Issues in the design of a code Generator, object code forms, Code Generation Algorithm, Register Allocation and Assignmen

* Issues in the design of a code Generator.



1. Input to the code Generator
2. Target program
3. memory management
4. Instruction selection
5. Register Allocation
6. Choice of Evaluation order

1. Input to the Code Generator:

i) It contains the intermediate representation of the source program and the information of the symbol table.

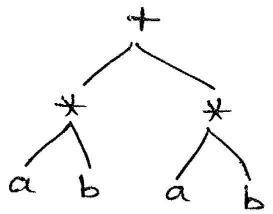
ii) The source program is produced by the front-end.

iii) we assume front end produces low-level intermediate representation.

iv) Intermediate representation has the several choices:

a) postfix notation → ab*

b) Syntax tree → a*b + a*b



c) Three address code.

2. Target program: The target program is the output of the code generator.

a) Assembly language: It allows subprogram to be separately compiled

b) Relocatable language: It makes the process of code generation easier.

c) Absolute language: It can be placed in a fixed location in memory and can be executed immediately.

3. memory management: Local variables are stack allocation in the activation record while global variables are in static area

4. Instruction selection: i) Nature of instruction set of the target machine should be complete and uniform.

ii) The quality of the generated code can be determined by its speed and size.

Ex: Three address code is:

a:= b+c

d: = a+e

5. Register allocation: i) Register can be accessed faster than memory
ii) In register allocation, we select the set of variables that will reside in register.

Ex: D x,y
'x' is the dividend even register in even/odd
'y' is the divisor.
Even register is used to hold the reminder
old register is used to hold the quotient.

6. Evaluation order: i) The efficienly of the target code can be effected by the order in which the computations are performed.
ii) some computation orders need fewer registers to hold results of intermediate than others.

⇒ Object Code Forms

1. Let assume that, you have a 'c' program, then you give the 'c' program to compiler and compiler will produce the output in assembly code.
2. Now, that assembly language code will give to the assembler and assembler is going to produce you some code, that is known as object code.

Text code
↓
Compiler
↓
Assembly code
↓
Assembler
↓
object code

3. once you call the compiler, then your object code is going to present in Hard-disk. This object code contains various part

Header
Text segment
Data segment
Relocation information
Symbol table
Debugging information

OBJECT CODE

* Header: It is nothing but like an index, like you have a textbook, there is an index page will contain at what page number each topic present.

* Text segment: It is nothing but the set of instruction.

* Data segment: Data segment will contain whatever data you have used.

* Relocation Information: Whenever you try to write a program, we generally use symbol to specify anything.
  - The original address is known as Relocatable address.

1000	L0	level
1001	L1	level
1002	L2	level
1003	L3	level
1004	L4	level
1005	L5	level
	Goto 1004	

Absolute Address

Relocation

0	L0	level
1	L1	level
2	L2	level
3	L3	level
4	L4	level
5	L5	level
	Goto L4	
	Goto 4	

Relocatable address

* Symbol table: It contains every symbol that you have in your program.

Ex: int a,b,c then a,b,c are the symbol.

* Debugging Information: It will help to find how a variable is keeping on changing.

⇒ Code Generation Algorithm

Code Generator: Code Generator is used to produce the target code for three-address statements. It uses registers to store the operands of the three address statement.

The algorithm takes a sequence of three-address statements as input.

- machine instructions for operations

1. Use getReg(x=y+z) to select registers for x, y, and z. call these Rz, Ry and Rx
2. If 'y' is not in Ry, then issue an instruction LD Ry, y', where y' is one of the memory locations for y.
3. similarly, if z is not in Rz, issue and instruction LD RZ, z', where z' is a location for z.
4. Issue the instruction ADD Rx, Ry, RZ

- machine Instructions for copy statements

There is an important special case: a three-address copy statement of the form $x=y$.

2. we assume that getReg will always choose the same register for both x and y. If
3. If 'y' is not already in that register R_y, then generate the machine instruction LD R_y, y.
4. If 'y' was already in R_y, we do nothing.
5. It is only necessary that we adjust the register description for R_y, so that it includes 'x' as one of the values found there.

* Generating code for Assignment statements:

let us take $d:=(a-b)+(a-c)+(a-c)$ can be translated into the following sequence of three-address code.

$t:=a-b$

$U:=a-c$

$V:=t+U$

$d:=V+U$

statement	Code Generated	Register descriptor Register empty	Address descriptor
t: =a-b	mov a, R0 SUB b, R0	R0 contains t	t in R0
u: =a-c	mov a, R1 SUB c, R1	R0 contains t R1 contains u	t in R0 u in R1
v: =t+u	ADD R1, R0	R0 contains v R1 contains u	u in R1 v in R1
d: =v+u	ADD R1, R0 mov R0, d	R0 contains d	d in R0 d in R0 and memory.

⇒ Register Allocation And Assignment:

1. Registers are the faster locations in the memory hierarchy.
2. But unfortunately, this resource is limited.
3. It comes under the most constrained resources of the target processor.
4. Register allocation is an Np-complete problem.
5. However, this problem can be reduced to graph coloring to achieve allocation and assignment.
6. A good register allocator computes an effective approxim solution to a hard problem.

Input program $\xrightarrow[\text{n Registers}]{}$ **Register Allocator** $\xrightarrow[\text{m Registers}]{}$ output program

n Registers | Register Allocator | m Registers

7. The register allocator determines which value will reside in the register and which register will hold each of those values.

8. Register allocation is only within a basic block.

9. It follows top-down approach

10. Assign registers to the most heavily used variables.

Traverse the block

count uses

use count as a priority function

Assign registers to higher priority variables first

11. Register allocation depends on:

i) size of live range

ii) Number of uses / definitions

iii) Frequency of execution

iv) Number of loads

v) cost of loads

Advantages

1. Heavily used values reside in registers

Disadvantages

1. It uses registers inefficiencly
2. Too strictly
3. Does not consider non-uniform distribution of uses