**UNIT-5**

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



**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

**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

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.

**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.

**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.

**Instruction selection:**

The instructions of target machine should be complete and uniform The quality of the generated code is speed and size.

**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 :

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

***Register assignment – the specific register that a variable will reside in is***

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

**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

**TARGET MACHINE**

Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator The target computer is a byte-addressable machine with 4 bytes to a word It has *n general-purpose registers, R0, R1, . . . , Rn-1.*

It has two-address instructions of the form:

*op source, destination* where, *op is an op-code, and source and destination are data fields.*

It has the following op-codes :

MOV (move *source to destination)*

ADD (add *source to destination)*

SUB (subtract *source from destination)*

For EX:MOV R0, M stoMOV R0, R1 copies the contents of register R0 into R1. It has cost one,since it occupies only one word of memory

The three-address statement **a : = b + c can be implemented by many different instruction**

sequences :

i) MOV b, R0

ADD c, R0 cost = 6

MOV R0, a

ii) MOV b, a

ADD c, a cost = 6

res contents of Register R0 into memory location M ;

iii) Assuming R0, R1 and R2 contain the addresses of a, b, and c :

MOV \*R1, \*R0

ADD \*R2, \*R0 cost = 2

**RUN-TIME STORAGE MANAGEMENT**

Information needed during an execution of a procedure is kept in a block of storage

called an activation record, which includes storage for names local to the procedure.

· The two standard storage allocation strategies are:

1. Static allocation

2. Stack allocation

In static allocation, the position of an activation record in memory is fixed at compiletime.

· In stack allocation, a new activation record is pushed onto the stack for each execution ofa procedure. The record is popped when the activation ends.

**BASIC BLOCKS AND FLOW GRAPHS**

**Basic Blocks**

· A *basic block is a sequence of consecutive statements in which flow of control enters at* the beginning and leaves at the end without any halt or possibility of branching except at the end.

· The following sequence of three-address statements forms a basic block:

t1 : = a \* a

t2 : = a \* b

t3 : = 2 \* t2

t4 : = t1 + t3

t5 : = b \* b

t6 : = t4 + t5



Basic block 1: Statement (1) to (2)

Basic block 2: Statement (3) to (12)

**Transformations on Basic Blocks:**

A number of transformations can be applied to a basic block without changing the set of

expressions computed by the block. Two important classes of transformation are :

· Structure-preserving transformations

· Algebraic transformations

**Structure preserving transformations:**

**a) Common subexpression elimination:**

a : = b + c a : = b + c

b : = a – d b : = a - d

c : = b + c c : = b + c

d : = a – d d : = b

Since the second and fourth expressions compute the same expression, the basic block can be transformed as above.

**b) Dead-code elimination:**

Suppose *x is dead, that is, never subsequently used, at the point where the statement x : =*y + z appears in a basic block. Then this statement may be safely removed without changing the value of the basic block.

**Renaming temporary variables:**

A statement **t : = b + c ( t is a temporary ) can be changed to u : = b + c (u is a new**

temporary) and all uses of this instance of **t can be changed to u without changing the value of** the basic block.Such a block is called a *normal-form block.*

**Interchange of statements:**

Suppose a block has the following two adjacent statements:

t1 : = b + c

t2 : = x + y

We can interchange the two statements without affecting the value of the block if and

only if neither **x nor y is t1 and neither b nor c is t2.**

**Algebraic transformations:**

Algebraic transformations can be used to change the set of expressions computed by a basic block into an algebraically equivalent set.

Examples:

i) x : = x + 0 or x : = x \* 1 can be eliminated from a basic block without changing the set of

expressions it computes.

ii) The exponential statement x : = y \* \* 2 can be replaced by x : = y \* y.

**Flow Graphs**

· Flow graph is a directed graph containing the flow-of-control information for the set of

basic blocks making up a program.

· The nodes of the flow graph are basic blocks. It has a distinguished initial node.

· E.g.: Flow graph for the vector dot product is given as follows:



B1 is the *initial node. B2 immediately follows B1, so there is an edge from B1 to B2. The*

target of jump from last statement of B1 is the first statement B2, so there is an edge fromB1 (last statement) to B2 (first statement).

· B1 is the *predecessor of B2, and B2 is a successor of B1.*

**Loops**

· A loop is a collection of nodes in a flow graph such that

1. All nodes in the collection are *strongly connected.*

2. The collection of nodes has a unique *entry.*

· A loop that contains no other loops is called an inner loop.

**NEXT-USE INFORMATION**

If the name in a register is no longer needed, then we remove the name from the register and the register can be used to store some other names.

**CODE OPTIMIZATION**

The code produced by the straight forward compiling algorithms can often be made to run faster or take less space, or both. This improvement is achieved by program transformations that are traditionally called optimizations. Compilers that apply code-improving

transformations are called optimizing compilers.

 Optimizations are classified into two categories. They are

 Machine independent optimizations:

 Machine dependant optimizations

**Machine independent optimizations:**

 Machine independent optimizations are program transformations that improve the target code without taking into consideration any properties of the target machine.

**Machine dependant optimizations:**

 Machine dependant optimizations are based on register allocation and utilization of special machine-instruction sequences

**PRINCIPAL SOURCES OF OPTIMISATION**

A transformation of a program is called local if it can be performed by looking only at the

statements in a basic block; otherwise, it is called global.

 Many transformations can be performed at both the local and global levels. Local

transformations are usually performed first.

**Function-Preserving Transformations**

 There are a number of ways in which a compiler can improve a program without

changing the function it computes.

 The transformations

 Common sub expression elimination,

 Copy propagation,

 Dead-code elimination, and

 Constant folding are common examples of such function-preserving transformations. The other transformations come up primarily when global optimizations are performed Frequently, a program will include several calculations of the same value, such as an offset in an array. Some of the duplicate calculations cannot be avoided by the programmer because they lie below the level of detail accessible within the source language.

 **Common Sub expressions elimination:**

An occurrence of an expression E is called a common sub-expression if E was previously

computed, and the values of variables in E have not changed since the previous computation. We can avoid recomputing the expression if we can use the previously computed value.

 For example

t1: = 4\*i

t2: = a [t1]

t3: = 4\*j

t4: = 4\*i

t5: = n

t6: = b [t4] +t5

The above code can be optimized using the common sub-expression elimination as

t1: = 4\*i

t2: = a [t1]

t3: = 4\*j

t5: = n

t6: = b [t1] +t5

The common sub expression t4: =4\*i is eliminated as its computation is already in t1. And value of i is not been changed from definition to use.

**Copy Propagation:**

 Assignments of the form f : = g called copy statements, or copies for short. The idea

behind the copy-propagation transformation is to use g for f, whenever possible after the

copy statement f: = g. Copy propagation means use of one variable instead of another.

This may not appear to be an improvement, but as we shall see it gives us an opportunity

to eliminate x.

 For example:

x=Pi;

……

A=x\*r\*r;

The optimization using copy propagation can be done as follows:

A=Pi\*r\*r;

Here the variable x is eliminated

**Dead-Code Eliminations:**

 A variable is live at a point in a program if its value can be used subsequently; otherwise,

it is dead at that point. A related idea is dead or useless code, statements that compute values that never get used. While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations. An optimization can

be done by eliminating dead code.

Example:

i=0;

if(i=1)

{

a=b+5;

}

Here, ‘if’ statement is dead code because this condition will never get satisfied.

**Constant folding**:

 We can eliminate both the test and printing from the object code. More generally,

deducing at compile time that the value of an expression is a constant and using the

constant instead is known as constant folding.

 One advantage of copy propagation is that it often turns the copy statement into dead

code.

 For example,

a=3.14157/2 can be replaced by

a=1.570 there by eliminating a division operation

**Loop Optimizations:**

 We now give a brief introduction to a very important place for optimizations, namely

loops, especially the inner loops where programs tend to spend the bulk of their time. The

running time of a program may be improved if we decrease the number of instructions in

an inner loop, even if we increase the amount of code outside that loop.

 Three techniques are important for loop optimization:

 code motion, which moves code outside a loop;

 Induction-variable elimination, which we apply to replace variables from inner loop.

 Reduction in strength, which replaces and expensive operation by a cheaper one, such as

a multiplication by an addition

**Code Motion:**

 An important modification that decreases the amount of code in a loop is code motion.

This transformation takes an expression that yields the same result independent of the

number of times a loop is executed ( a loop-invariant computation) and places the

expression before the loop. Note that the notion “before the loop” assumes the existence

of an entry for the loop. For example, evaluation of limit-2 is a loop-invariant

computation in the following while-statement:

while (i <= limit-2) /\* statement does not change limit\*/

Code motion will result in the equivalent of t= limit-2;

while (i<=t) /\* statement does not change limit or t \*/\

 **Induction Variables :**

 Loops are usually processed inside out. For example consider the loop around B3.

 Note that the values of j and t4 remain in lock-step; every time the value of j decreases by

1, that of t4 decreases by 4 because 4\*j is assigned to t4. Such identifiers are called

induction variables.

 When there are two or more induction variables in a loop, it may be possible to get rid of

all but one, by the process of induction-variable elimination. For the inner loop around

B3 in Fig. we cannot get rid of either j or t4 completely; t4 is used in B3 and j in B4.

However, we can illustrate reduction in strength and illustrate a part of the process of

induction-variable elimination. Eventually j will be eliminated when the outer loop of B2

- B5 is considered.

Example:

As the relationship t4:=4\*j surely holds after such an assignment to t4 in Fig. and t4 is not

changed elsewhere in the inner loop around B3, it follows that just after the statement

j:=j-1 the relationship t4:= 4\*j-4 must hold. We may therefore replace the assignment t4:=

4\*j by t4:= t4-4. The only problem is that t4 does not have a value when we enter block B3

for the first time. Since we must maintain the relationship t4=4\*j on entry to the block B3,



we place an initializations of t4 at the end of the block where j itself is

initialized, shown by the dashed addition to block B1 in second Fig.

**Reduction In Strength:**

 Reduction in strength replaces expensive operations by equivalent cheaper ones on the

target machine. Certain machine instructions are considerably cheaper than others and

can often be used as special cases of more expensive operators.

 For example, x² is invariably cheaper to implement as x\*x than as a call to an

exponentiation routine. Fixed-point multiplication or division by a power of two is

cheaper to implement as a shift. Floating-point division by a constant can be implemented

as multiplication by a constant, which may be cheaper