**CODE GENERATION**

Issues involved in code generation – Register allocation – Conversion of three address code to assembly code using code generation algorithm – examples – Procedure for converting assembly code to machine code – Case study

# 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.

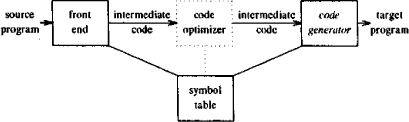


Figure 5.1 Position of code generator

* The code generated by the compiler is an object code of some lower-level programming language, for example, assembly language.
* The source code written in a higher-level language is transformed into a lower-level language that results in a lower-level object code, which should have the following minimum properties:
  + It should carry the exact meaning of the source code.
  + It should be efficient in terms of CPU usage and memory management.

# 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)Instruction selection

4)Register allocation

5)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 to determine run-time addresses of the data objects denoted by the names in the intermediate representation.

Intermediate representation can be :

1)Linear representation such as postfix notation

2)Three address representation such as Quadruples, triples etc.

3)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:

**Absolute Object Code**:

It can be placed in a fixed memory location and can be executed immediately. In absolute object code, the generated machine code contains fixed memory addresses.

It assumes that the program will be loaded into a specific memory location in the computer's memory.

This form is less flexible as the program cannot be easily relocated to different memory addresses.

**Relocatable Object Code:**

It allows subprograms to be compiled separately. Relocatable object code is more flexible as it does not assume a fixed memory location.

It contains information allowing the operating system's loader to adjust memory addresses during the loading process.This facilitates the program's ability to be loaded into different memory locations.

**Symbolic (or Assembly) Code:**

Symbolic code is a human-readable representation of the program that includes mnemonic instructions and symbolic addresses.

It is often used in conjunction with an assembler, which translates symbolic code into machine code.

This form is useful for debugging and understanding the program's logic.

# Instruction selection:

* 1. The instructions of target machine should be complete and uniform.
  2. Instruction speeds and machine idioms are important factors when efficiency of target program is considered.
  3. The quality of the generated code is determined by its speed and size.

# For example:

Every three-address statement of the form

# x=y+z

where x, y and z are statically allocated. Code sequence generated is shown as:

MOV y,R0 /\* load y into register R0 \*/ ADD z,R0 /\* add z to R0 \*/

MOV R0,x /\* store R0 into x \*/

Unfortunately, this kind of statement-by-statement code generation often produces poor code. For example, the sequence of statements,

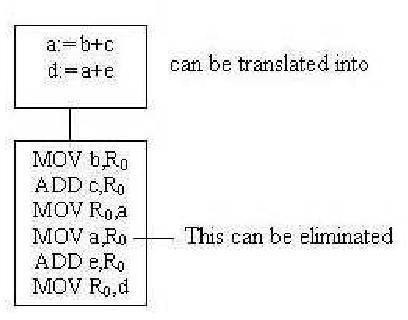


Figure 5.2 Code Translation

The quality of the generated code is determined by its speed and size. A target machine with a rich instruction set may provide several ways of implementing a given operation.

For example:

If the target machine has an “increment” instruction(INC), then the three address statement

# a=a+1 , its equivalent assembly code is given below

# MOV a,R0

# ADD #1,R0

# MOV R0,a

may be implemented more efficiently by the single instruction **a++ ,** its equivalent assembly instruction is given below

# INC a

**Register allocation:**

Instructions involving register operands are shorter and faster than those involving operands in memory. A program has a number of values to be maintained during the execution. The target machine’s architecture may not allow all of the values to be kept in the registers. Code generator decides what values to keep in the registers. Also, it decides the registers to be used to keep these values.

The use of registers is subdivided into two sub problems :

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

**Register assignment -** the specific register that a variable will reside is selected.

Certain machine requires even-odd register pairs for some operands and results. For example consider the division instruction of the form :

# Div x, y

where, x – dividend in even register in even/odd register pair, y – divisor in even register holds the remainder odd register holds the quotient

# Evaluation order:

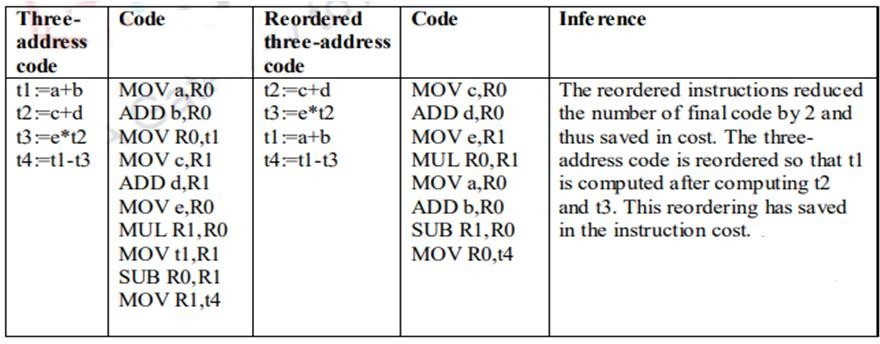
At last, the code generator decides the order in which the instruction will be executed. The order in which the computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others.

* Picking the best order is a difficult task.
* Initially avoid this problem by generating code for the three address statements in the order in which they have been produced by the intermediate code generator.
* It creates schedules for instructions to execute them.

When instructions are independent, their evaluation order can be changed to utilize registers and save on instruction cost. Consider the following instruction:

a+b-(c+d)\*e

The three-address code, the corresponding code and its reordered instruction are given below:



# A SIMPLE CODE GENERATOR: \*\*A code-generation algorithm:

A code generator generates target code for a sequence of three- address statements and effectively uses registers to store operands of the statements.

For example: consider the three-address statement a := b+c can have the following sequence of codes:

ADD Rj, Ri Cost = 1 // if Ri contains b and Rj contains c (or)

ADD c, Ri Cost = 2 // if c is in a memory location (or)

MOV c, Rj Cost = 3 // move c from memory to Rj ADD Rj, Ri

# Register and Address Descriptors:

A **register descriptor** is used to keep track of what is currently in each registers. The register descriptors show that initially all the registers are empty.

An **address descriptor** stores the location where the current value of the name can be found at run time.

code-generation algorithm :

The algorithm takes as input a sequence of three -address statements constituting a basic block. For each three-address statement of the form x : = y op z, perform the following actions:

Invoke a function *getreg* to determine the location L where the result of the computation y op z should be stored.

Consult the address descriptor for y to determine y’, the current location of y. Prefer the register for y’ if the value of y is currently both in memory and a register. If the value of y is not already in L, generate the instruction MOV y’ , L to place a copy of y in L.

Generate the instruction OP z’ , L where z’ is a current location of z. Prefer a register to a memory location if z is in both. Update the address descriptor of x to indicate that x is in location L. If x is in L, update its descriptor and remove x from all other descriptors.

If the current values of y or z have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution of x : = y op z those registers will no longer contain y or z.

# Generating Code for Assignment Statements:

The assignment d : = (a-b) + (a-c) + (a-c) might be translated into the following three- address code sequence:

t : = a – b

u : = a – c v: = t + u d := v + u

# Code sequence for the example is:

Table 5.2 Shows the code sequence and the register allocation

|  |  |  |  |
| --- | --- | --- | --- |
| **Statements** | **Code Generated** | **Register descriptor** | **Address descriptor** |
| t : = a – b | MOV a, R0  SUB b, R0 | Register empty | 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 R0 |
| d : = v + u | ADD R1, R0  MOV R0, d | R0 contains d | d in R0  d in R0 and memory |

# Generating code for Indexed Assignments

|  |  |  |
| --- | --- | --- |
| **Statements** | **Code Generated** | **Cost** |
| a : = b[i] | MOV b(Ri), R | 2 |
| a[i] : = b | MOV b, a(Ri) | 3 |

**Generating code for Pointer Assignments**

|  |  |  |
| --- | --- | --- |
| **Statements** | **Code Generated** | **Cost** |
| a : = \*p | MOV \*Rp, a | 2 |
| \*p: = a | MOV a, \*Rp | 2 |

# \*\*Register Allocation and Assignment:

**Register allocation:**

Instructions involving register operands are shorter and faster than those involving operands in memory. A program has a number of values to be maintained during the execution. The target machine’s architecture may not allow all of the values to be kept in the registers. Code generator decides what values to keep in the registers. Also, it decides the registers to be used to keep these values.

The use of registers is subdivided into two sub problems :

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

**Register assignment -** the specific register that a variable will reside is selected.

Certain machine requires even-odd register pairs for some operands and results. For example consider the division instruction of the form :

# Div x, y

where, x – dividend in even register in even/odd register pair, y – divisor in even register holds the remainder odd register holds the quotient

Register allocation is only within a basic block. It follows top-down approach.

# Local register allocation

* Register allocation is only within a basic block. It follows top-down approach.
* 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

# Need of global register allocation:

* Local allocation does not take into account that some instructions (e.g. those in loops) execute more frequently. It forces us to store/load at basic block endpoints since each block has no knowledge of the context of others.
* To find out the live range(s) of each variable and the area(s) where the variable is used/defined global allocation is needed. Cost of spilling will depend on frequencies and locations of uses.

Register allocation depends on:

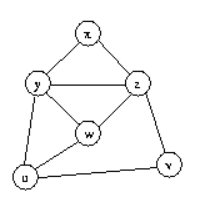
* + Size of live range
  + Number of uses/definitions
  + Frequency of execution
  + Number of loads/stores needed.
  + Cost of loads/stores needed.

# Register allocation using Graph Coloring :

# The interference graph is used for assigning registers to temporary variables. If two variables do not interfere (ie, there is no edge between these two variables in the interference graph) then we can use the same register for both of them, thus reducing the number of registers needed. On the other hand, if there is a graph edge between two variables, then we should not assign the same register to them since this register needs to hold the values of both variable at one point of time (because the lives of these variables overlap at one point of time - this is what interference means).

# Suppose that we have n available registers: r1, r2,..., rn. If we view each register as a different color, then the register allocation problem for the interference graph is equivalent to the graph coloring problem where we try to assign one of the n different colors to graph nodes so that no two adjacent nodes have the same color.

Consider the following interference graph from the previous section:



The nodes are pushed in the stack in the order of xvuzyw. Then, at the selection phase, xyzwuv variables are assigned the registers R0R2R1R0R1R0.

**\*\* THE DAG REPRESENTATION FOR BASIC BLOCKS**

A DAG for a basic block is a directed acyclic graph with the following labels on nodes:

Leaves are labeled by unique identifiers, either variable names or constants.

Interior nodes are labeled by an operator symbol.

Nodes are also optionally given a sequence of identifiers for labels to store the

computed values.

DAGs are useful data structures for implementing transformations on basic blocks.

It gives a picture of how the value computed by a statement is used in subsequent

statements.

**It provides a good way of determining common sub - expressions**

Input: A basic block

Output: A DAG for the basic block containing the following information:

A label for each node. For leaves, the label is an identifier. For interior nodes, an

operator symbol.

For each node a list of attached identifiers to hold the computed values.

Case (i) x : = y OP z

Case (ii) x : = OP y

Case (iii) x : = y

Method:

Step 1: If y is undefined then create node(y).

If z is undefined, create node(z) for case(i).

Step 2: For the case(i), create a node(OP) whose left child is node(y) and right child is

node(z). ( Checking for common sub expression). Let n be this node.

For case(ii), determine whether there is node(OP) with one child node(y). If not create such

a node.

For case(iii), node n will be node(y).

Step 3: Delete x from the list of identifiers for node(x). Append x to the listof attached identifiers for the node n found in step 2 and set node(x) to n.

**Example:** Consider the block of three- address statements:

t1 := 4\* i

t2 := a[t1]

t3 := 4\* i

t4 := b[t3]

t5 := t2\*t4

t6 := prod+t5

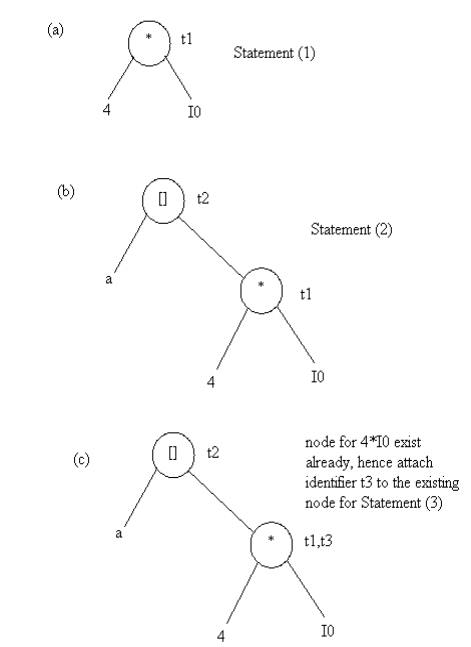
prod := t6

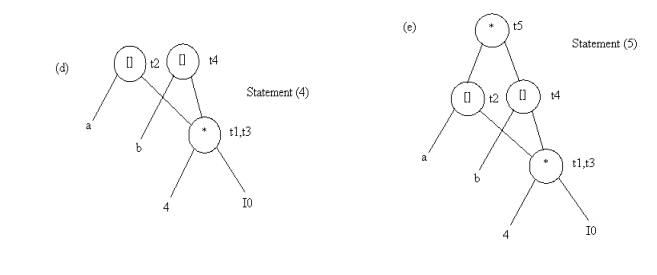
t7 := i+1

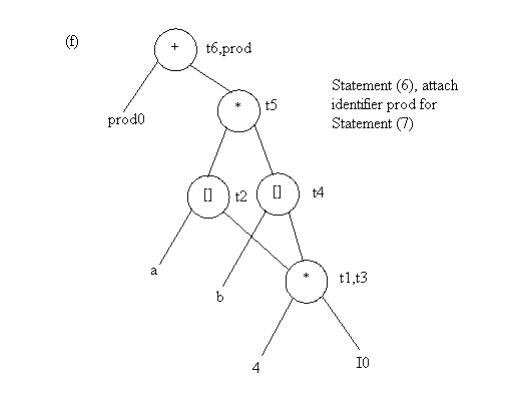
i := t7

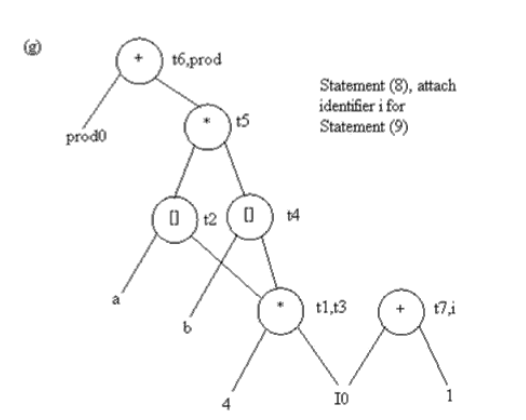
if i<=20 goto (1)

Stages in DAG Construction









# 

# \*\* Object Code forms :

# Object code forms in the context of compiler design refer to the different representations of machine code generated by a compiler for a specific target architecture. These forms include:

# Absolute Object Code:

# In absolute object code, the generated machine code contains fixed memory addresses.

# It assumes that the program will be loaded into a specific memory location in the computer's memory.

# This form is less flexible as the program cannot be easily relocated to different memory addresses.

# Relocatable Object Code:

# Relocatable object code is more flexible as it does not assume a fixed memory location.

# It contains information allowing the operating system's loader to adjust memory addresses during the loading process.

# This facilitates the program's ability to be loaded into different memory locations.

# Symbolic (or Assembly) Code:

# Symbolic code is a human-readable representation of the program that includes mnemonic instructions and symbolic addresses.

# It is often used in conjunction with an assembler, which translates symbolic code into machine code.

# This form is useful for debugging and understanding the program's logic.

# Bytecode:

# Bytecode is a form of intermediate code that is typically designed to be executed by a virtual machine.

# It is commonly used in languages like Java and C#.

# The bytecode is interpreted or compiled by a virtual machine at runtime.

# The choice of object code form depends on factors such as the target platform, the desired level of optimization, and the execution model of the programming language. Different forms may be used at different stages of the compilation process, and some forms may be more suitable for certain types of applications or deployment scenarios.