(2)

Prof. Deepali Raikar

2 COP

8

Code Generation

8.1 Introduction

Code generation is the final activity of compiler. Basically code generation is a process of creating **assembly language/machine language** statements which will perform the operations specified by the **source program** when they run.

Various properties desired by an object code generation phase are

- **Correctness** – It should produce a correct code and do not alter the purpose of source code.
- **High quality** – It should produce a high quality object code.
- **Efficient use of resources of the target machine** – While generating the code it is necessary to know the target machine on which it is going to get generated. By this the code generation phase can make an efficient use of resources of the target machine. For instance memory utilization while allocating the registers or utilization of arithmetic logic unit while performing the arithmetic operations.
- **Quick code generation** – This is a most desired feature of code generation phase. It is necessary that the code generation phase should produce the code quickly after compiling the source program.

In this chapter we will discuss the concept of code generation and object code forms. Then we will understand the machine dependant code optimization process which is popularly done by peephole optimization technique. In the later part of this chapter we will see how actual code get generated by using different code generating algorithms.

8.2 Code Generation

As we know, code generation is the final phase in the process of compilation. It takes intermediate code as an input and generates target machine code as output. The position of code generator in compilation process is illustrated by following figure.

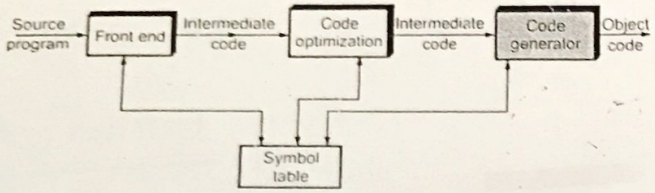


Fig. 8.1 Position of code generator in compiler

8.3 Object Code Forms

The output of code generation is an object code or machine code. This code normally comes in following forms.

1. Absolute code
2. Relocatable machine code
3. Assembler code

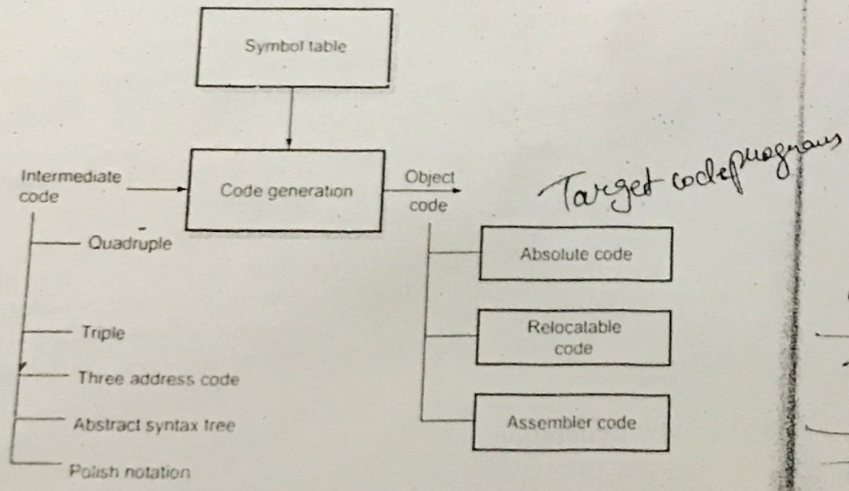


Fig. 8.2 Object code forms

Let us discuss each in detail.

1. **Absolute code** – Absolute code is a machine code that contains reference to actual addresses within program's address space. The generated code can be placed directly in the memory and execution starts immediately. Generally small programs can be compiled and executed quickly because of absolute code generation. A number of "student-job" compilers such as WATFIV and PL/C produce absolute code.

2. **Relocatable code** – Producing a relocatable machine language program as output allows subprograms to be compiled separately. A set of relocatable object modules can be linked together and loaded for execution with the help of a *linking loader*. The advantage of generating relocatable machine language code is that we gain a great deal of flexibility in being able to compile subroutines separately and to call other previously compiled programs from an object module.

   If the target machine does not handle relocation automatically, the compiler must provide explicit relocation information to the loader to link the separately compiled program segments.

3. **Assembler code** – Producing an assembly language program as output makes the process of code generation somewhat easier. We can generate symbolic instructions and use the macro facilities of the assembler to help in generation of code. But generating assembler code as an output makes code generation process slower. (because assembling, linking and loading is required).

8.4 Issues in Code Generation

Let us discuss some common issues in design of code generator

- **Input to the code generator**

The code generation phase takes intermediate code as input. This intermediate code along with the symbol table information is used to determine the runtime addresses of the data objects. These data objects are denoted by the names in the intermediate representation. The intermediate code may be in any form such as three address code, quadruple, posix notation or it may be represented using graphical representations such as syntax trees or DAGs. The intermediate code generated by the front end should be such that the target machine can easily manipulate it, in order to generate appropriate target machine code. In the front end of compiler necessary type checking and type conversion needs to be done. The detection of the semantic errors should be done before submitting the input to the code generator. The code generation phase requires the complete error free intermediate code as input.

- **Target programs**

The output of code generator is target code. Typically, the target code comes in three forms such as : absolute machine language, relocatable machine language and assembly language.

The advantage of producing target code in absolute machine form is that it can be placed directly at the fixed memory location and then can be executed immediately. The benefit of such target code is that small programs can be quickly compiled.

For example : The WATFIV and PL/C are the compilers which produce the absolute code as output.

The advantage of producing the relocatable machine code as output is that the subroutines can be compiled separately. Relocatable object modules can be linked together and loaded for execution by a linking loader. This offers great flexibility of compiling the subroutines separately. If the target machine cannot handle the relocation automatically then the compiler must provide explicit relocation information to the loader in order to link the separately compiled subroutine segments.

The advantage of producing assembly code as output makes the code generation process easier.The symbolic instructions and Macro facilities of assembler can be used to generate the code.It is advantageous to have assembly language as output for the machines with small memory.

However, out of these three forms of output target code it is always preferable to have relocatable machine code as target code.

- **Memory management**

Both the front end and code generator performs the task of mapping the names in the source program to addresses to the data objects in run time memory. The names used in the three address code refer to the entries in the symbol table. The type in a declaration statement determines the amount of storage (memory) needed to store the declared identifier. Thus using the symbol table information about memory requirements code generator determines the addresses in the target code.

Similarly if the three address code contains the labels then those labels can be converted into equivalent memory addresses. For instance if a reference to *'goto j'* is encountered in three address code then appropriate jump instruction can be generated by computing the memory address for label j.

- **Instruction selection**

The uniformity and completeness of instruction set is an important factor for the code generator. The selection of instruction depends upon the instruction set of target machine. The speed of instruction and machine idioms are two important factors in

selection of instructions. If we do not consider the efficiency of target code then the instruction selection becomes a straightforward task.

For each type of three address code the code skeleton can be prepared which ultimately gives the target code for the corresponding construct.

For example - x := a+b then the code sequence that can be generated as

```
MOV a, R0 /* loads the a to register R0*/
ADD b, R0  /* performs the addition of b to R0*/
MOV R0,x /* stores the contents of register R0 to x */
```

In the above example the code is generated line by line. Such a line by line code generation process generates the poor code because the redundancies can be achieved by subsequent lines and those redundancies cannot be considered in the process of line by line code generation.

For example

```
x := y+z
a := x+t
```

The code for the above statements can be generated as follows :

```
MOV y,R0
ADD z,R0
MOV R0,x
MOV x,R0
ADD t,R0
MOV R0,a
```

The above generated code is a poor code because MOV R0, a is not used and statement MOV a, R0 is redundant. Hence the efficient code can be

```
MOV y,R0
ADD z,R0
ADD t,R0
MOV R0,a
```

The quality of generated code is decided by its speed and size. Simply line by line translation of three address code into target code leads to correct code but it can generate unacceptably non efficient target code.

- **Register allocation**

If the instruction contains register operands then such a use becomes shorter and faster than that of using operands in the memory. Hence while generating a good Code efficient utilization of register is an important factor. There are two important activities done while using registers.

1. Register allocation - During register allocation, select appropriate set of variables that will reside in registers.
2. Register assignment - During register assignment, pick up the specific register in which corresponding variable will reside.

Obtaining the optimal (minimum) assignment of registers to variable is difficult.

Certain machines require register pairs such as even odd numbered registered for some operands and results.

For example in IBM systems integer multiplication requires register pair.

Consider the three address code.

```
t1 := a+b
t1 := t1*c
t1 := t1/d
```

The efficient machine code sequence will be

```
MOV a,R0
ADD b,R0
MUL c,R0
DIV d,R0
MOV R0,t1
```

- **Choice of evaluation order**

The evaluation order is an important factor in generating an efficient target code. Some orders require less number of registers to hold the intermediate results than the others. Picking up the best order is one of the difficulty in code generation. Mostly, we can avoid this problem by referring the order in which the three address code is generated by semantic actions.

- **Approaches to code generation**

The most important factor for a code generation is that it should produce the correct code. With this approach of code generation various algorithms for generating code are designed.

8.5 Target Machine Description

For designing the good code generator it is necessary to have prior knowledge of target machine and instruction set used for this target machine.

In this chapter we will assume that the target machine code is a register machine like minicomputers. Specifically following assumptions are made for code generation.

- We will assume that in the target computer addresses are given in bytes and four bytes form a word.
- There are n general purpose registers R0, R1,....Rn-1.
- The two address instruction is of the form.

op source destination

where op is an opcode and source and destination are data fields.

For instance :

MOV - moves from source to destination.

ADD - add source to destination.

SUB - subtracts source from destination.

The source and destination are specified by registers and memory locations.

- The addressing modes used are as follows.

Addressing mode	Form	Address	Added cost
Absolute	M	M	1
Register	R	R	0
Indexed	c(R)	c + contents(R)	1
Indirect register	*R	contents(R)	0
Indirect indexed	*c(R)	contents(c+contents(R))	1
Literal	#c	c	1

If we have absolute or resister addressing mode we can use M or register for source or destination.

For example, the instruction MOV R1, M stores the contents of register R_1 into memory location M.

For the indexed addressing mode the address offset c from the value of register R0 can be written as

MOV 7(R1), M

Means it stores the value contents (7+ contents(R1)) to the memory location M.

The last two addressing modes represent the indirect addresses indicates by *.

For the instruction MOV *7(R0), M

It stores the value contents(contents (7+ contents(R0))) to the memory location M.

In the literal addressing mode the source becomes constant.

For example MOV #5, R0

By this instruction we can store the constant 5 into register R0.

8.5.1 Cost of the Instruction

The instruction cost can be computed as one plus cost associated with the source and destination addressing modes given by "added cost".

Instruction	Cost	Interpretation
MOV R0, R1	1	Cost of register mode +1=0+1=1
MOV R1, M	2	Use of memory variable +1=1+1=2
SUB 5(R0), *10(R1)	3	Use of first constant +use of second constant +1 =3

➠ **Example 8.1** : *Compute the cost of following set of instructions.*

```
MOV a,R0
ADD b,R0
MOV R0,c
```

Solution : The cost of this set is 6 because

```
MOV a,R0          2
ADD b,R0          2
MOV R0,c          2
-----------------------
Total cost    =   6
```

➠ **Example 8.2** : *Compute the cost of following set of instructions.*

```
MOV *R1,*R0
ADD *R2,*R0
```

Solution : The cost of this set is 2 because

```
MOV *R1,*R0   1
ADD *R2,*R0   1
-----------------
Total cost = 2
```

8.6 Basic Blocks and Flow Graphs

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

An example of the basic block is as shown below.

```
t1 := a*5
t2 := t1+7
t3 := t2-5
t4 := t1+t3
t5 := t2+b
```

8.6.1 Some Terminologies used in Basic Blocks

- **Define and use** - The three address statement a := b+c is said to define a and to use b and c.

- **Live and dead** - The name in the basic block is said to be live at a given point if its value is used after that point in the program. And the name(variable) in the basic block is said to be dead at a given point if its value is never used after that point in the program.

8.6.2 Algorithm for Partitioning into Blocks

Any given program can be partitioned into basic blocks by using following algorithm. We assume that an intermediate code is already generated for the given program.

1. First determine the **leaders** by using following rules.
   (a) The **first statement** is a leader.
   (b) Any target statement of conditional or unconditional **goto** is a leader.
   (c) Any statement that immediately **follow a goto** or unconditional goto is a leader.
2. The basic block is formed starting at the leader statement and ending just before the next leader statement appearing.

➠ **Example 8.3** : *Consider the following program code for computing dot product of two vectors a and b of length 10 and partition it into basic blocks.*

```
prod = 0;
i = 1;
do
{
  prod = prod + a[i] * b[i];
  i=i+1;
}while(i<=10);
```

Solution : First we will write the equivalent three address code for the above program.

1. prod := 0
2. i := 1
3. t_1 := 4 * i
4. t_2 := a[t_1] /* computation of a[i] */
5. t_3 := 4 * i
6. t_4 := b[t_3] /* computation of b[i] */
7. t_5 := t_2 * t_4
8. t_6 := prod+t_5
9. prod := t_6
10. t_7 := i+1

11. i := t_7
12. if i< = 10 goto (3)

According to the algorithm

Statement 1 is a leader by rule 1(a).

Statement 3 is also leader by rule 1(b).

Hence, statement 1 and 2 form the basic block. Similarly statement 3 to 12 form another basic block.

Block 1

1. prod := 0
2. i := 1

Block 2

3. t_1 := 4 * i
4. t_2 := a[t_1]
5. t_3 := 4 * i
6. t_4 := b[t_3]
7. t_5 := t_2 * t_4
8. t_6 := prod+t_5
9. prod := t_6
10. t_7 := i + 1
11. i := t_7
12. if i<=10 goto (3)

8.6.3 Flow Graph

A flow graph is a directed graph in which the flow control information is added to the basic blocks.

- The nodes to the flow graph are represented by **basic blocks**.
- The block whose leader is the first statement is called **initial block**.
- There is a directed edge from block B_1 to block B_2 if B_2 immediately follows B_1 in the given sequence. We can say that B_1 is a predecessor of B_2.

For example, consider the three address code as

1. prod := 0
2. i := 1

3. t_1 := 4 * i
4. t_2 := a[t_1] /* computation of a[i] */
5. t_3 := 4 * i
6. t_4 := b[t_3] /* computation of b[i] */
7. t_5 := t_2 * t_4
8. t_6 := prod+t_5
9. prod := t_6
10. t_7 := i+1
11. i := t_7
12. if i<=10 goto (3)

The flow graph for the above code can be drawn as follows.

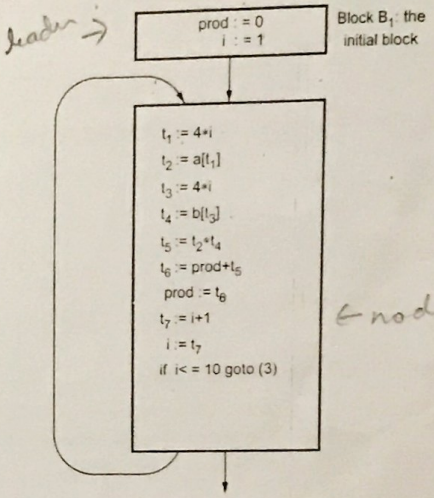


Fig. 8.3 Flow graph

In this flow graph block B_1 is a initial block.

Loop

Loop is a collection of nodes in the flow graph such that,

i) All such nodes are strongly connected. That means always there is a path from any node to any other node within that loop.

ii) The collection of nodes has unique entry. That means there is only one path from a node outside the loop to the node inside the loop.

iii) The loop that contains no other loop is called inner loop.

8.7 Next-Use Information

The next-use information is a collection of all the names that are useful for next subsequent statement in a block. The use of a name is defined as follows.

Consider a statement,

x := i

j := x op y

that means the statement j uses value of x

The next-use information can be collected by making the backward scan of the programming code in that specific block. Suppose the three address statement is as given below.

L: a := b op c then the steps in backward scan are -

i) The currently found information in symbol table regarding the next use and liveness of a, b and c is associated with the statement L.

ii) In the symbol table set 'a' to "not live" and "no next use".

iii) Set 'b' and 'c' to live and next uses of b and c in symbol table.

8.7.1 Storage for Temporary Names

For the distinct names each time a temporary is needed. And each time a space gets allocated for each temporary. To have optimization in the process of code generation we can pack two temporaries into the same location if they are not live simultaneously.

Consider three address code as,

$t_1 := a * a$

$t_2 := a * b$

$t_3 := 4 * t_2$

$t_4 := t_1 + t_3$

$t_5 := b * b$

$t_6 := t_4 + t_5$

This can be packed into two temporaries as follows.

$t_1 := a * a$

$t_2 := a * b$

$t_2 := 4 * t_2$

$t_1 := t_1 + t_2$

$t_2 := b * b$

$t_1 := t_1 + t_2$

Many times the temporaries can be packed into registers rather than memory locations.

8.8 Register Allocation and Assignment

As we know the use of register operands instead of memory operands is always the faster and shorter. This also means that proper use of registers help in generating the good code. There are various strategies for deciding how to use these registers and what values are to be stored in the registers. In other words certain strategies are adopted by the compiler for register allocation and assignment (as we know that register handling in code generation can be done using register allocation and assignment).

- The most commonly used strategy to register allocation and assignment is to **assign specific values** to specific registers. For instance for base addresses separate set of registers can be used. Similarly for storing the stack pointers again a separate set of registers can be assigned; arithmetic computations can be done using separate set of registers. Remaining set of registers are used by the compiler for suitable purposes.
- The **advantage** of this approach is that the design process of compiler for code generation becomes **simplified**.
- The **disadvantage** of this method is that the design of compiler becomes **complicated** because of restrictive use of registers. At the same time certain set of registers remain totally unused over substantial portions of code and some set of registers get overloaded. But this disadvantage can be tolerable by most of the compiler and this approach can be adopted in most of the computing environment.

Now we will discuss various strategies used in register allocation and assignment and those are

1. Global register allocation
2. Usage count
3. Register assignment for outer loop
4. Graph coloring for register assignment

8.8.1 Global Register Allocation

While generating the code the registers are used to hold the values for the duration of single block. All the live variables are stored at the end of each block. For the variables that are used consistently we can allocate specific set of registers. Hence allocation of variables to specific registers that is consistent across the block boundaries is called **global register allocation**.

Following are the strategies adopted while doing the global register allocation.

- The global register allocation has a strategy of storing the most frequently used variables in fixed registers throughout the loop.
- Another strategy is to assign some fixed number of global registers to hold the most active values in each inner loop.
- The registers not already allocated may be used to hold values local to one block.
- In certain languages like C or Bliss programmer can do the register allocation by using register declaration.

8.8.2 Usage Count

The usage count is the count for the use of some variable x in some register used in any basic block. The usage count gives the idea about how many units of cost can be saved by selecting a specific variable for global register allocation. The approximate formula for usage count for the Loop L in some basic block B can be given as,

$$\sum_{\text{block B in L}} (\text{use}(x, B) + 2 * \text{live}(x, B))$$

Where use (x, B) is number of times x used in block B prior to any definition of x and live (x, B) =1 is x is live on exit from B; otherwise live(x)=0.

For example :

Consider a block B1, B2, B3, B4 and count the usage count for block B in following loop L.

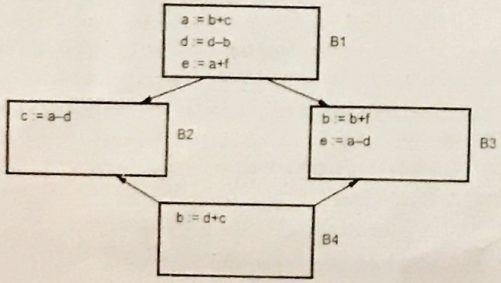


Fig. 8.4

The usage count for block B1 for variable a is

use(a, B1) = 0 ∵ a is defined in B1 before use.

2 * live(a, B1) = 2 ∵ a is live on exit of B1 hence live(a, B1) = 1

∴ (use(a, B1) + 2 * live(a, B)) = 2

The usage count for block B2 and B3 for variable a is

use(a, B2) = use(a, B2) = 1 ∵ a is used in B1 and B2 before definition.

2 * live(a, B1) = 2 * live(a, B2) = 0 ∵ a is not live on exit of B1 and B2

$$\therefore \sum_{\text{Bin L}} \text{use}(a, B) = 2$$

$$\therefore \sum_{\text{Bin L}} \text{use}(a, B) + 2 * \text{live}(a, b) = 2 + 2$$

$$= 4$$

Hence the usage count of a is 4. That means compiler can save 4 units of cost by selecting a for the global register allocation.

8.8.3 Register Assignment for Outer Loop

Consider that there are two loops L1 is outer loop and L2 is an inner loop. And allocation of variable a is to be done to some register. The approximate scenario is as given below.

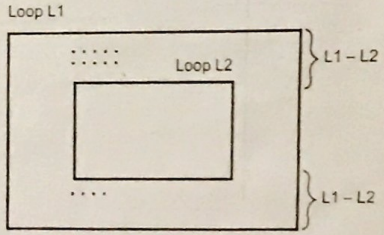


Fig. 8.5 Nested loops

Following criteria should be adopted for register assignment for outer loop.

1) If a is allocated in loop L2 then it should not be allocated in L1 – L2.
2) If a is allocated in L1 and it is not allocated in L2 then store a on a entrance to L2 and load a while leaving L2.
3) If a is allocated in L2 and not in L1 then load a on entrance of L2 and store a on exit from L2.

8.9 DAG Representation of Basic Blocks

The directed acyclic graph is used to apply transformations on the basic block. To apply the transformations on basic block a DAG is constructed from three address statement.

A DAG can be constructed for the following type of labels on nodes.

1) Leaf nodes are labelled by identifiers or variable names or constants. Generally leaves represent r-values.
2) Interior nodes store operator values.

The DAG and flow graphs are two different pictorial representations. Each node of the flow graph can be represented by DAG because each node of the flow graph is a basic block.

Example 8.4 : *Consider*

```
sum = 0;
for (i = 0;i<=10;i++)
sum = sum+a[i];
```

Solution : The three address code for above code is

1) sum := 0
2) i := 0
3) t_1 := 4 * i
4) t_2 := a[t_1]
5) t_3 := sum+t_2
6) sum := t_3
7) t_4 := i+1;
8) i := t_4
9) if i<=10 goto (3)

We can partition above code into basic blocks as follows.

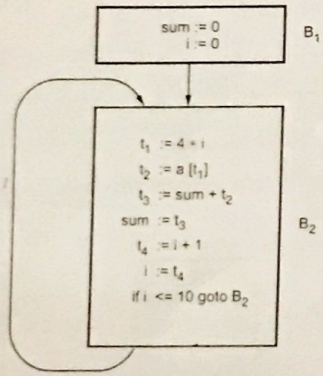
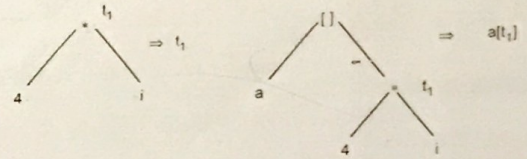


Fig. 8.6 Basic blocks

Now let us consider block B_2 for construction of DAG by numbering it.

1) t_1 := 4 * i
2) t_2 := a[t_1]
3) t_3 := sum + t_2
4) sum := t_3
5) t_4 := i + 1
6) i := t_4
7) if i<10 goto (1)



Continuing in this fashion,

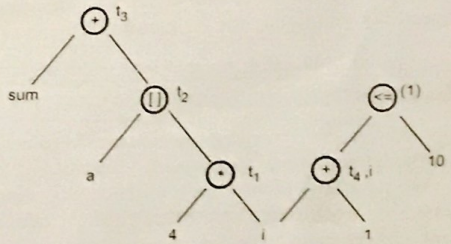


Fig. 8.7 DAG for block B_2

At node <= we have specified the goto 1 condition by mentioning statement number (1).

Algorithm for construction of DAG

We assume the three address statement could of following types.

Case (i) x := y op z

Case (ii)x := op y

Case (iii) x := y

With the help of following steps the DAG can be constructed.

Step 1 : If y is undefined then create node(y). Similarly if z is undefined create a node(z).

Step 2 : For the case(i) create a node(op) whose left child is node(y) and node(z) will be the right child. Also check for any common subexpressions. For the case(ii) determine whether is a node labeled op, such node will have a child node(y). In case(iii) node n will be node(y).

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

Applications of DAG

The DAG is used in

1. Determining the common sub-expressions (expressions computed more than once).
2. Determining which names are used inside the block and computed outside the block.
3. Determining which statements of the block could have their computed value outside the block.
4. Simplifying the list of quadruples by eliminating the common sub-expressions and not performing the assignment of the form x := y unless and until it is a must.

8.10 Peephole Optimization

If we apply the statement by statement code generation strategy then the generated target code may contain many redundant instructions. The quality of such code is very poor. To optimize such a target code certain transformations need to be applied on the target code. These transformations ultimately will result in getting the significant improvement over the running time or space requirement of the target program.

Definition

Peephole optimization is a simple and effective technique for locally improving target code. This technique is applied to improve the performance of the target program by examining the short sequence of target instructions and replacing these instructions by shorter or faster sequence. [Note that : a peephole –a small window is moved over the target code and transformations can be made. And hence is the name!]

8.10.1 Characteristics of Peephole Optimization

The peephole optimization can be applied on the target code using follow characteristic.

1. Redundant instruction elimination.
2. Flow of control optimization.
3. Algebraic simplification.
4. Use of machine idioms.

Let us discuss each one by one.

1. Redundant instruction elimination

- Especially the redundant loads and stores can be eliminated in this type of transformations.

For example :

```
MOV R0,x
MOV x,R0
```

We can eliminate the second instruction since x is already in R0. But if (MOV x, R0) is a label statement then we cannot remove it.

- We can eliminate the unreachable instructions. For example, following is a piece of C code.

```
sum=0
if(sum)
 printf("%d",sum);
```

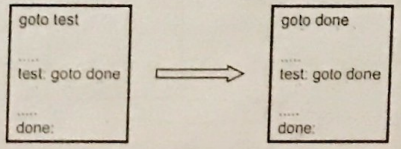
Now this if statement will never get executed hence we can eliminate such a unreachable code, similarly

```
int fun(int a,int b)
{
c=a+b;
return c;
printf("%d",c);   /* unreachable code and hence can be eliminated */
}
```

2. Flow of control optimization

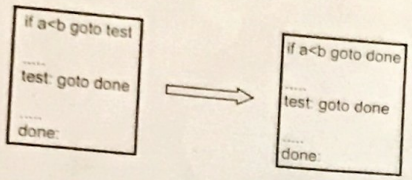
Using peephole optimization unnecessary jumps on jumps can be eliminated.

For example,



Thus we reduce one jump by this transformation.

Another example could be



3. Algebraic simplification

Peephole optimization is an effective technique for algebraic simplification.

The statements such as

```
x := x+0
```

or

```
x := x*1
```

can be eliminated by peephole optimization.

4. Reduction in strength

Certain machine instructions are cheaper than the other. In order to improve the performance of the intermediate code we can replace these instructions by equivalent cheaper instruction. For example, x^2 is cheaper than x*x. Similarly addition and subtraction is cheaper than multiplication and division. So we can effectively use equivalent addition and subtraction for multiplication and division.

5. Machine idioms

The target instructions have equivalent machine instructions for performing some operations. Hence we can replace these target instructions by equivalent machine instructions in order to improve the efficiency. For example, some machines have auto-increment or auto-decrement addressing modes that are used to perform add or subtract operations.

8.11 A simple Code Generator

In this section we will discuss the method of generating target code from three address statement.

- In this method computed results can be kept in registers as long as possible.

For example :

```
x := a+b;
```


The corresponding target code is

ADD b, R_i	Here R_i holds value of a Here cost =2.

OR

MOV b, R_1 ADD R_1, R_0	Here R_0 holds value of a Here cost =2.

- The code generator algorithm uses descriptors to keep track of register contents and addresses for names.

1. A **register descriptor** is used to keep track of what is currently in each register. The register descriptors show that initially all the registers are empty. As the code generation for the block progresses the registers will hold the values of computations.
2. The address descriptor stores the location where the current value of the name can be found at run time. The information about locations can be stored in the symbol table and is used to access the variables.

- The modes of operand addressability are as given below.

S is used to indicate value of operand in storage.

R is used to indicate value of operand in register.

IS indicates that the address of operand is stored in storage i.e. indirect accessing.

IR indicates that the address of operand is stored in register i.e. indirect accessing.

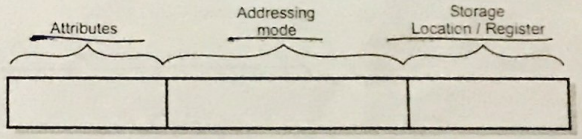


Fig. 8.8 Address descriptor

- The address descriptor has following fields

The **attributes** mean type of the operand. It generally refers to the name of temporary variables.

The **addressing mode** indicates whether the addresses are of type 'S', 'R', 'IS', 'IR'.

The third field is **location field** which indicates whether the address is in storage location or in register.

- Similarly the register descriptor can be shown as below.

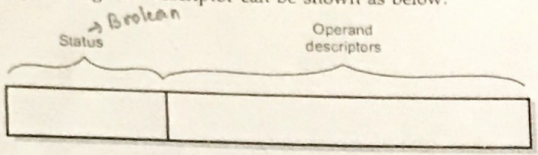


Fig. 8.9 Register descriptor

By using register descriptors we can keep track of the registers which are currently occupied. The status field is of Boolean type which is used to check whether the register is occupied with some data or not. When the status field holds the value 'True' then operand descriptors fields contains the pointer to the operand descriptor who is having the latest value in the register.

Algorithm for code generation

Read the expression in the form of operator, operand1 and operand2 and generate code using following algorithm.

```
Gen_Code(operator, operand1, operand2)
{
    if (operand1.addressmode = 'R')
    {
        if (operator = '+')
        Generate('ADD operand2,R0');
        else if(operator = '-')
            Generate('SUB operand2,R0');
        else if(operator = '*')
            Generate('MUL operand2,R0');
        else if(operator = '/')
            Generate('DIV operand2,R0');
    }
        else     if (operand2.addressmode = 'R')
        {
        if (operator = '+')
        Generate('ADD operand1,R0');
        else if(operator = '-')
            Generate('SUB operand1,R0');
        else if(operator = '*')
            Generate('MUL operand1,R0');
        else if(operator = '/')
            Generate('DIV operand1,R0');
        }
        else
        {
        Generate('MOV operand2,R0);
            if (operator = '+')
        Generate('ADD operand2,R0');
```



```
        else if(operator = '-')
            Generate('SUB operand2,R0');
        else if(operator = '*')
            Generate('MUL operand2,R0');
        else if(operator = '/')
            Generate('DIV operand2,R0');
    }
}
```

Example

We will generate code for following expression.

$$x := (a + b) * (c - d) + ((e/f) * (a + b))$$

The corresponding three address code can be given as,

$t_1 := a + b$

$t_2 := c - d$

$t_3 := e/f$

$t_4 := t_1 * t_2$

$t_5 := t_3 * t_1$

$t_6 := t_4 + t_5$

Using the simple code generation algorithm the sequence target code can be generated as,

Three address code	Target code sequence	Register descriptor	Operand descriptor
$t_1 := a + b$	MOV a, R_0 ADD b, R_0	Empty R_0 contains t_1	t_1 | R | R_0
$t_2 := c - d$	MOV c, R_1 SUB d, R_1	R_1 contains c R_1 contains t_2	t_2 | R | R_1
$t_3 := e/f$	MOV e, R_2 DIV f, R_2	R_2 contains e R_2 contains t_3	t_3 | R | R_2
$t_4 := t_1 * t_2$	MUL R_0, R_1	R_0 contains t_1 R_1 contains t_2 R_1 contains t_4	t_4 | R | R_1
$t_5 := t_3 * t_1$	MUL R_2, R_0	R_2 contains t_3 R_0 contains t_1 R_0 contains t_5	t_5 | R | R_0
$t_6 := t_4 + t_5$	ADD R_1, R_0	R_1 contains t_4 R_0 contains t_5 R_0 contains t_6	t_6 | R | R_0

8.12 Generating Code from DAG

Generating code from DAG is much simpler than the linear sequence of three address code. Using DAG we can rearrange some sequence of instructions and generate an efficient code. There are various algorithms used generating code from DAG as shown in Fig. 8.10.

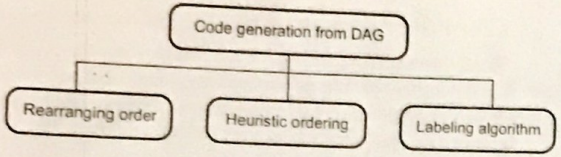


Fig. 8.10 Code generation from DAG

Let us understand how effective it is to generate code using DAG instead of using a three address code sequence.

8.12.1 Rearranging Order

The order of three address code affects the cost of the object code being generated. In the sense that by changing the order in which computations are done we can obtain the object code with minimum cost.

For example :

t_1:=a+b

t_2:=c–d

t_3:=e+t_2

t_4:=t_1+t_3

For the expression (a+b)+(e+(c–d))

A DAG can be constructed for the above sequence as follows.

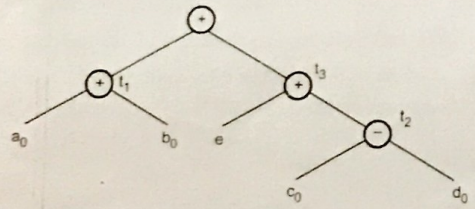


Fig. 8.11 DAG for (a+b) + (e+(c – d))

The code can be generated by translating the three address code line by line.

```
MOV a, R0
ADD b, R0
MOV c, R1
SUB d, R1
MOV R0, t1          t1 := a+b
MOV e, R0           R1 has c-d
ADD R0, R1          /*R1 contains e+(c-d)*/
MOV t1, R0          /*R0 contains a+b*/
ADD R1, R0
MOV R0, t4
```

Now if we change the ordering sequence of the above three address code

t_2:=c–d

t_3:=e+t_2

t_1:=a+b

t_4:=t_1+t_3

then we get an improved code as

```
MOV c, R0
SUB d, R0
MOV e, R1
ADD R0, R1
MOV a, R0
ADD b, R0
ADD R1, R0
MOV R0, t4
```

8.12.2 Heuristic Ordering

The heuristic ordering algorithm is as follows.

1) Obtain all the interior nodes. Consider these interior nodes as unlisted interior nodes.
2) while (unlisted interior nodes remain)
   {
3) pick up an unlisted node n, whose parents have been listed.

4) list n;
5) while(the leftmost child m of n has no unlisted parent AND is not leaf
{
6) list m;
7) n=m;
}
}

First we will draw a DAG for some given expression.

Consider a DAG as shown in the following Fig. 8.12.

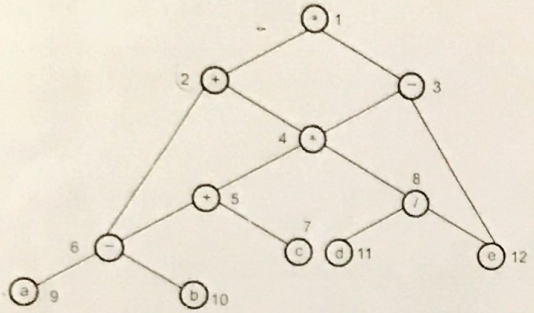


Fig. 8.12 DAG

The DAG is first numbered from top to bottom and from left to right. Then consider the unlisted interior nodes 1 2 3 4 5 6 8.

Initially the only node with unlisted parent is 1

∴ set n=1 by line 4) of algorithm

Now left argument of 1 is 2 and parent of 2 is 1 which is listed. Hence list 2.

∴ set n=2 by line 7) of algorithm

1 2 3 4 5 6 8

Now we will find the leftmost node of 2 and that is 6. But 6 has unlisted parent 5. Hence we cannot select 6.

We therefore can switch to 3. The parent of 3 is 1 which is listed one. Hence list 3 set n=3.

1 2 3 4 5 6 8

The left of 3 is 4. As parent of 4 is 3 and that is listed hence list 4. Left of 4 is 5 which has listed parent (i.e. 4) hence list 5. Similarly list 6.

1 2 3 4 5 6 8

As now only 8 is remaining from the unlisted interior nodes we will list it.

Hence the resulting list is 1 2 3 4 5 6 8. Then the order of computation is decided by reversing this list. We get the order of evaluation as 8 6 5 4 3 2 1. That also means that we have to perform the computations at these nodes in the given order.

$t_8 := d/e$

$t_6 := a - b$

$t_5 := t_6 + c$

$t_4 := t_5 * t_8$

$t_3 := t_4 - e$

$t_2 := t_6 + t_4$

$t_1 := t_2 * t_3$

This gives the optimized code for DAG even though there are any number of registers.

8.12.3 Labelling Algorithm

The labelling algorithm generates the optimal code for given expression in which minimum registers are required. Using labelling algorithm the labelling can be done to the tree by visiting nodes in bottom-up order. By this all the child nodes will be labelled before its parent nodes.

For computing the label at node n with the label L1 to left child and label L2 to the right child as

$$\text{Label (n)} = \begin{cases} \max(L1, L2) & \text{if } L1 \neq L2 \\ L1 + 1 & \text{if } L1 = L2 \end{cases}$$

- We start in bottom-up fashion and label left leaf as 1 and right leaf as 0.
- If labels of the children of a node n are L1 and L2 respectively then

$$\text{Label (n)} = \begin{cases} \max(L1, L2) & \text{if } L1 \neq L2 \\ L1 + 1 & \text{if } L1 = L2 \end{cases}$$

Example 8.5 : *Label the following tree*

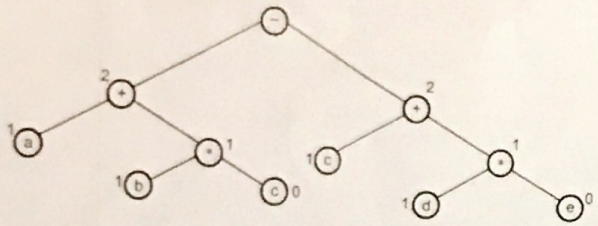


Fig. 8.13 Labelled tree for $((a+(b*c))-(c+(d*e)))$

Solution :

Algorithm

We will first discuss the important notations used in this algorithm.

1. The function Gen_code(n) is used that produces the code to evaluate a node labelled as n.
2. For the register allocation a stack is used called 'Reg_Stack'. Initially this stack contains all the available registers R0, R1, ...Rk such that the register R0 is on the top of the stack.
3. The Gen_code(n) evaluates n in the register which is on the top of the stack.
4. Another stack is used to store the temporaries called 'Temp_stack' it contains all the temporaries T0, T1,...Tk having T0 on the top of the stack.
5. The function swap(Reg_Stack) is used to swap top two registers on the stack.
6. The function Print is used to concatenate the arguments in the order in which they are coming. The || is used to indicate the concatenation. In this function the non-quoted arguments need to be evaluated first.
7. The function Gen_code is a function that generates the code. This function consists of various cases. Let us consider these cases.

Case 1: If n is a left node and it is a leaf node.

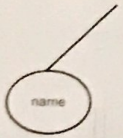


Fig. 8.14

```
Print('MOV'||name||',' top(Reg_Stack));
```

Here the node n is named by an identifier *name*. This also means load *name* into register.

Case 2: If node's right child is a leaf n_2 then,

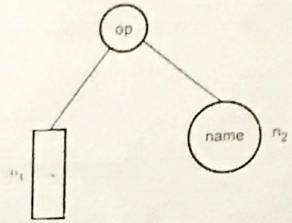


Fig. 8.15

```
Gen_code(n1)
Print(op||name||',',||top(Reg_stack));
```

Here n_1 is first evaluated in the register on the top of the stack and then the operation on identifier *name* is performed with the same register which is on the top of the stack.

Case 3:

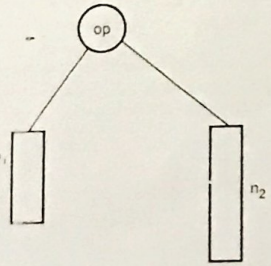


Fig. 8.16

If left child of n i.e. n_1 requires less number of registers than n_2 then we swap top two registers on Reg_Stack.

Then evaluate n_2 into R = top (Reg_Stack). We remove R from the Reg_stack and evaluate n_1 into then we generate an instruction

```
Print(op||R||',',||top(Reg_Stack));
```

Then we push R onto the Reg_stack and then call for swap.

Thus in this case we evaluate the right subtree into register and store this register just below the top of the stack. Then we evaluate the left subtree into the register which is on the top of the stack. The register on the top of the stack contains the left operand and register below the top of the stack contains right operand. Then n is evaluated in the register on the top of the stack.

Case 4 : If the left child n_1 requires more number of registers than the right child n_2, then,

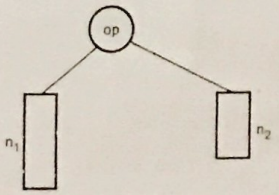


Fig. 8.17

In this case we evaluate left subtree first then the right subtree. There is no need to swap the registers here.

Case 5: Both the children require equal or more number of registers than the available number of registers. Then,

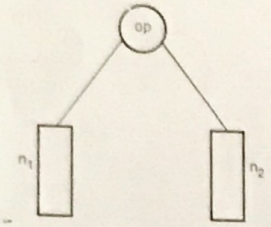


Fig. 8.18

In this case the right subtree is evaluated first and stored in a temporary. Then the evaluation of left subtree is done. And finally the root is evaluated.

Code generation from labelled tree

The function Gen_code(n) can be written as

```
Gen_code(n)
{
/*   case 1: */
if n is a left leaf node then
  Print('MOV'||name||','||top(Reg_Stack);
else
if n is an interior node with operator op and left, right child as
n1 and n2 respectively then
/* case 2 */
if Label(n2)=0
{
n2 represents the operand name
Gen_code(n1);
Print(op||name||','||top[Reg_Stack);
}
/*case 3 */
else if(Label(n1)< Label(n2) AND Label(n1) < k
{
swap(Reg_Stack);
Gen_code(n1);
R=pop(Reg_Stack)   /* n2 is evaluated in reg.R*/
Gen_code(n1);
Print(op||R||','||top(Reg_Stack);
Push(Reg_Stack,R);
swap(Reg_Stack);
```



```
}
/* Case 4 */
else if(Label(n2)< Label(n1) AND Label(n2) < k
 /* k is total number of registers*/
{
Gen_code(n1);
R pop(Reg_Stack)   /* n1 is evaluated in reg.R*/
Gen_code(n2);
Print(op||R||','||top(Reg_Stack);
Push(Reg_Stack,R);
}
/* case 5*/
else
 {
  Gen_code(n2);
  T:=pop(Temp_Stack);
  Print('MOV'||top(Reg_Stack)||',',||T);
  Gen_code(n1);
  Push(Temp_Stack,T);
  Print(op||T||',',||top(Reg_Stack));
 }
}
```

Let us generate a code from this labelling algorithm.

Consider the labelled tree as follows

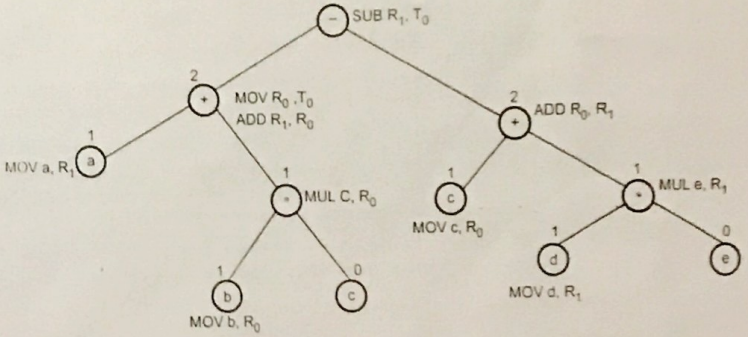


Fig. 8.19 Labelled tree with generated code for $((a+(b*c))-(c+(d*e)))$

```
MOV b, R0
MUL c, R0
MOV a, R1
ADD R1, R0
MOV R0, T0
```