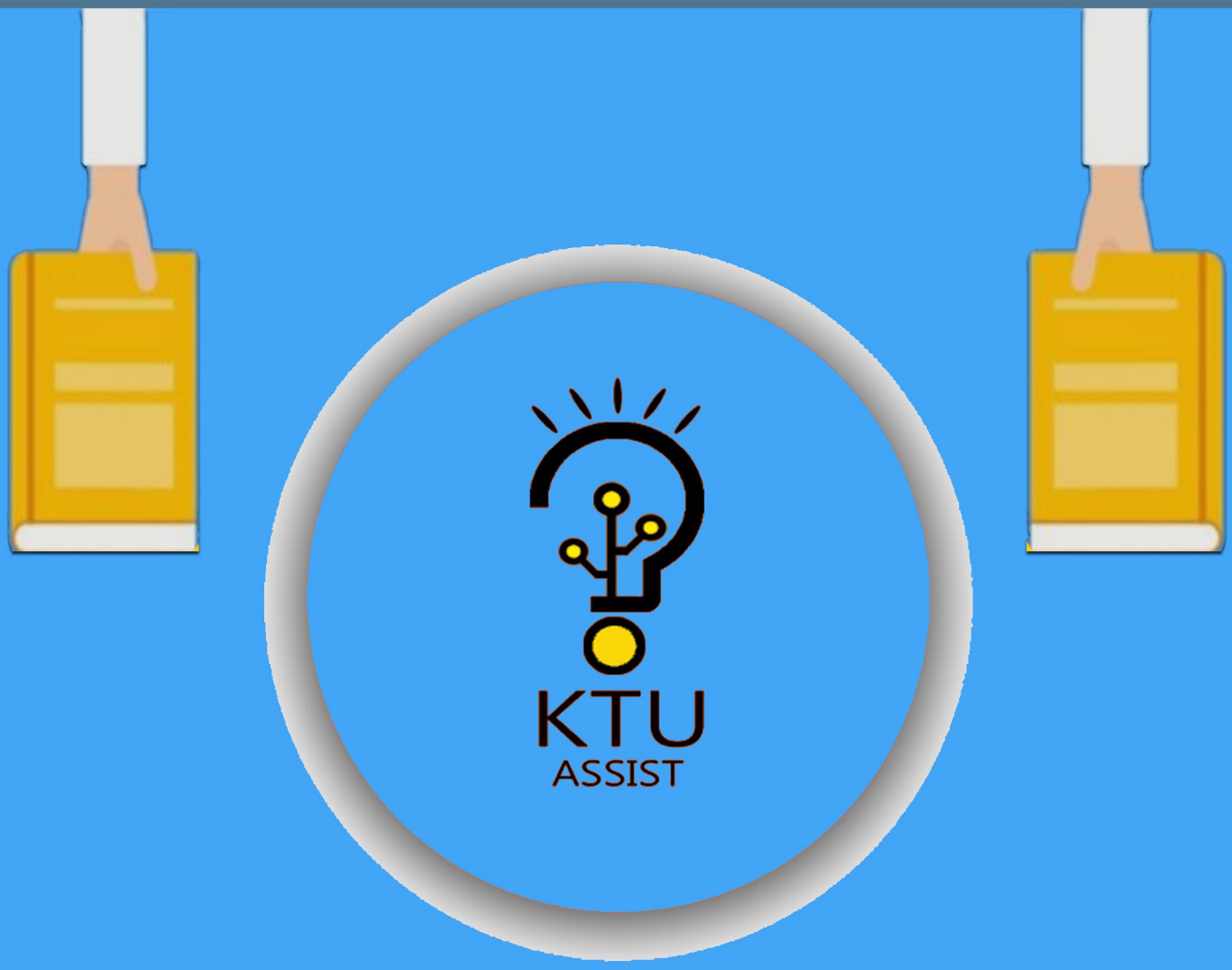
APJ ABDUL KALAM TECHNOLOGICAL UNIVERSITY

STUDY MATERIALS



Module 6

Code Optimization & Code Generation

CODE OPTIMIZATION

Ques 1) What is code optimization? How it is achieved?

Ans: Code Optimization

Code optimization refers to techniques by which a compiler can employ in an attempt to produce a better object language program than the most obvious for a given source program. The source program (the input to a compiler) is a specification of some computation.

The object program (the output of the compiler) is supposed to be another specification of the same computation. For each source program there are infinitely many object programs that implement the same computation, in the sense that they produce the same output when presented with the same input. Some of these object programs may be better than others with regard to such criteria as **size, memory** or **speed**.

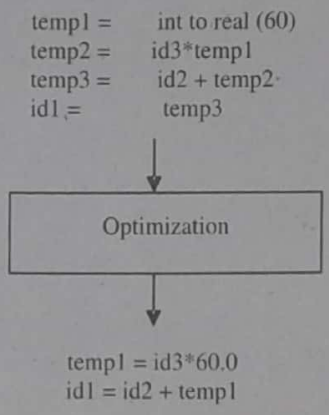


Figure 6.1: Code Optimization

Code optimizations (**figure 6.1**) are achieved by:

1) Eliminating redundancies in a program
2) Rearrangement of computations in a program to make it execute efficiently.

The **primary task** of code optimization phase is to reorganize the intermediate code of the program to make the object code more efficient. The efficiency criteria may be of a different nature, e.g. to increase performance and decrease memory consumption. The transformations that take place at optimization phase should create a program equivalent to the source one.

Ques 2) Describe optimization transformations.

Or

What are the code improving transformations?

Ans: Optimizing Transformations (Code Improving Transformations)

The code improvement can be possible by using the code improving transformations. These are generally called **optimization techniques**. Some of the codes improving transformation are as follows:

1) **Constant Folding:** This is actually compile time evaluation. This makes possible for the computations performed during the execution time itself, and thus avoids the computation during the execution time. Constant folding is nothing but replacing the runtime compilation by compile time computation.

   For example, a := (22/7) * (r * r)

   In this example, 22/7 can be computed during the compilation time itself than computing in each execution.

2) **Constant Propagation:** Propagation means propagating an entity from one statement to another statement. This is done for constants.

For example, consider the following three address statements,

```
Temp1 = 4;
   ...
   ...
temp2 = temp1 * 2;
```

Here, the variable temp1 is propagated. This can be optimized as given below during the compile time itself:

```
temp1 = 4;
   ...
   ...
temp2 = 4 * 2;
```

This is actually constant propagation. Constant propagation is nothing but replacement of a variable by a constant that appears on the RHS of an assignment for that variable. This is done during the compile time. The variable should not be re-defined along the path of its use.

3) **Common Sub-Expression Elimination (CSE):** This can be explained by the following example:

consider the following code:

```
a = x + y;
   ...
   ...
b = x + y + z;
```

The above can be optimized as follows:

```
temp3 = x + y;
   a = temp3;
   ...
   ...
   b = temp3 + z;
```

In the second case, the computation is done for "x + y" only once. But in the previous case twice the computations are done; "x + y" is here common sub-expression. The common sub-expression elimination allows no re-computations of an expression which is evaluated already. The common sub-expression should not be re-defined along the path of its use.

4) **Variable Propagation or Copy Propagation:** Copy propagation means the use of variable v1 in place of v2. Consider the following statements:

```
v1 = v2;
   ...
   ...
h = v1 + f;
g = v2 + f – 6;
```

In statement 2, v2 can be used in place of v1 by copy propagation. The transformed statements will be as listed below:

```
v1 = v2;
   ...
   ...
   ...
h = v2 + f;
g = v2 + f – 6;
```

This leads for common sub-expression elimination further.

5) **Dead Code Elimination:** Dead codes are the portions of the program that can never get executed for any control flow through the program. Thus, these codes can be eliminated from the program without affecting the program behavior.

For example,

i) A variable is said to be dead at a place in a program if the value contained in the variable at that place is not used anywhere in the program.

ii) If an assignment is made to a variable v at a place where v is dead, then the assignment is a dead assignment.

For example, consider the flow graph shown in **figure 6.2**. The following observations can be made:

i) The assignment a = c of block 1 is not dead, since a is used in block 4.

ii) The assignment x = y – 5 of block 5 is dead. The expression y – 5 can also be eliminated since it is no longer meaningful. However, an expression capable of producing side effects like function calls cannot be eliminated so simply.

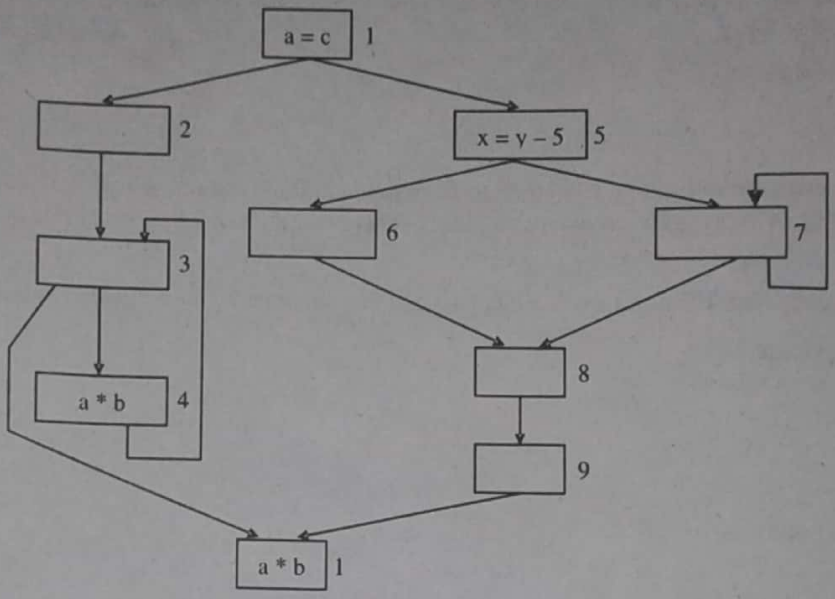


Figure 6.2: Dead Code Elimination

Ques 3) Define principle source of optimization.

Or

Write a note on loop optimization.

Ans: Principle Sources of Optimization

A transformation is local if it is performed by looking at the statements only in the basic blocks. Otherwise it is global. Local transformations are usually performed first.

Some of the sources of optimisations are listed below:

1) **Function-Preserving Transformations:** These transformations improve the performance of the code without changing the actions that it performs. The following are types of function preserving transformations:

   i) **Common Sub-Expression (CSE) Elimination:** If an expression E is computed already and its values are re-used, then it is a common sub-expression. Consider the TAC in **figure 6.3**.

```
1)  i :=m1
2)  j :=n
3)  t1 :=4*n
4)  v :=a[t1]
5)  i := i+1
6)  t2 := 4*i
7)  t3 := a[t2]
8)  if t3 < v goto (5)
9)  j :=j1
10) t4 := 4*j
11) t5 := at4]
12) if t5 > v goto (9)
13) if i>= j goto (23)
14) t6 := 4*i
15) x := a[t6]
```

```
16) t7 := 4*i
17) t8 := 4*j
18) t9 := a[t8]
19) a[t7] := t9
20) t10 := 4*j
21) a[t] [0] := x
22) goto (5)
23) t11 := 4*i
24) x := a[t11]
25) t12 := 4*i
26) t13 := 4*n
27) t14 := a[t][3]
28) a[t12] := t14
29) t15 := 4*n
30) a[t15] := x
```

Figure 6.3: TAC for Quick Sort

Here in Block B5, 4*i is computed in statements 14 and 16. Also 4*j computed in statement 17 is re-computed in statement 20. Hence values in temporary registers t6 and t7 are the same. Also values in registers t8 and t10 are the same. Hence t6 and t8 are used instead of t7 and t10. Hence block B5 is rewritten as shown in **figure 6.4 (a):**

```
(14) t6 := 4 * i
(15) x := a[t6]
(16) t8 := 4 * j
(18) t9 := a[t8]
(19) a[t6]:= t9
(21) a[t8] := x
(22) goto (5) B2
```

Figure 6.4 (a): B5 after Elimination of Common Sub-Expressions

After eliminating common sub-expressions in block B6, the resultant block B6 is shown in **figure 6.4 (b):**

```
(23) t11 := 4 * i
(24)x := a[t11]
(26) t13 := 4 * n
(27) t14:= a[t12]
(28) a[t12] := t14
(30) a[t13] := x
```

Figure 6.4 (b) : B6 after Elimination of Common Sub-Expressions

After elimination of global common sub-expressions, the blocks B5 and B6 are shown in **figure 6.4 (c):**

```
(15) x := t3
(19) a[t2] := t5
(21) a[t4] := x
(22) goto (5) B2
```

```
(24) x := t3
(27) t14 := a[t1]
(28) a[t2] := t14
(30) a[t1] := x
```

Figure 6.4 (c): B5 and B6 after Global Elimination of Common Sub-Expressions

Here t6 and t11 in blocks B5 and B6 are eliminated and replaced by t2. Hence a[t6] and a[t11] is a[t2] and its value is stored in t3 in block B2. Similarly, t8 is replaced by t4. Hence, a[t8] is now a[t4], which is stored in t5. Similarly, t13 is replaced by t1. Hence, a[t13] is now a[t1].

For example, eliminate common sub-expressions in the following:

```
a := b*c;
x := b*c + 5;
```

"b*c" is the sub-expression. It is eliminated as follows:

```
t1 := b*c;
a := t1;
x := t1 + 5;
```

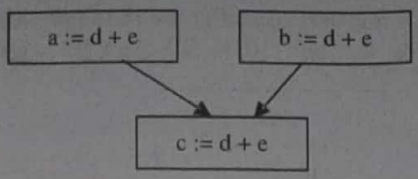
ii) **Copy Propagation:** Assignments of the form f := g are called as copy statements. When common sub-expressions are eliminated, copy statements are introduced. Hence they have to be eliminated. **For example,** in **figure 6.4 (c),** "x" is a copy variable. Hence it can be propagated if the statement "x" in statement (21) is replaced by t3. Hence B5 can be rewritten as follows:

```
(15) x := t3
(19) a[t2] := t5
(21) a[t4] := t3
(22) goto (5) B2
```

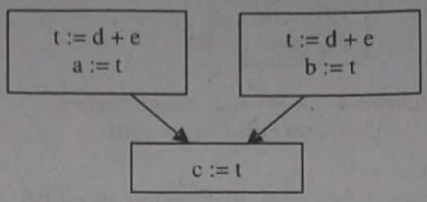
```
(24) x := t3
(27) t14 := a[t1]
(28) a[t2] := t14
(30) a[t1] := t3
```

Figure 6.5: Copy Propagation in B5 and B6

Consider the following flow graph:



Eliminating common sub-expressions, copies are introduced as follows:



iii) **Dead Code Elimination:** A variable is said to be dead at a place in a program if the value contained in the variable at that place is not used anywhere in the program. If an assignment is made to a variable v at a place where v is dead, then the assignment is a dead assignment. Removing a dead assignment makes no difference to the meaning/results of the program. This is also called as useless code. Dead code can result from optimisation transformations.

2) **Loop Optimisation:** Innermost loops are most important sources of optimisation. Even if a few statements are eliminated in the innermost loops, then code improvement is high. Three important techniques of loop optimisations are code motion, elimination of induction variables, and reduction in strength. These methods are explained below:

i) **Code Motion:** In this technique, loop invariants are removed from the loop and placed before it. **For example,** consider "while(i<=n-2)". As 'n' is a loop invariant, it can be rewritten as "t=n -2; while (i<=t)".

Consider TAC in **figure 6.6**, t2 and t4 are loop invariants.

```
1)  prod :=
2)  i: =1
3)  t1:=4*I        /*This is because it is considered that each array element */
                   /*occupies 4 bytes */
4)  t2:=addr(a)-4  /* since the first array index is a[0] */
5)  t3:=t2[t1]
6)  t4:=addr(b)-4
7)  t5:=t4[t1]
8)  t6:=t3*t5
9)  prod:=prod+t6
10) i:=i+1
11) if i<=20 goto(3)
```

Figure 6.6: TAC for Matrix Multiplication

Hence it is moved out of block B2 and placed in a new block B3 outside it. Hence the flow graph is re-drawn as shown in **figure 6.7:**

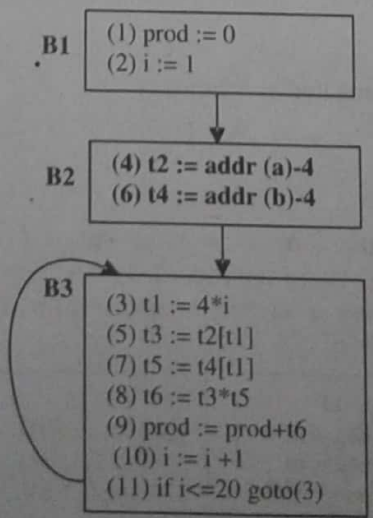


Figure 6.7: Code Motion

ii) **Induction Variable Elimination:** An induction variable v is an integer scalar variable which is only subjected to the following kinds of assignments in a loop: v = v op constant. The controlled variable of a for loop is an induction variable according to this definition.

In **figure 6.8**, variable t1 is an arithmetic progression of i. Hence 'i' can be eliminated. Thus i := i+ 1 becomes, t1 := t1 + 4 (4 bytes for each array element). And i<=20 becomes t1<=76 (t1<=4*20 – 4), as the array index begins from '0'. Now statement 3 can be eliminated and t1 can be initialised to 0. t1 is a loop invariant. Hence it is removed outside B2. The resulting flow graph is given below:

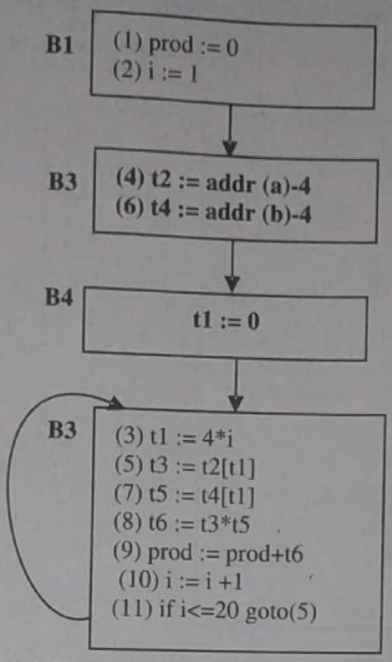


Figure 6.8: Flow Graph for Matrix Multiplication after Induction Variable Elimination

iii) **Reduction in Strength:** The removal of induction variables has resulted in another type of optimisation called reduction in strength. Here, costly operations like '*' are replaced by less costly operations like '+'. Thus i = i * 2 is replaced by i = i + i. Similarly, consider the operation that finds the combined lengths of strings S1 and S2.

L = strlen(S1 || S2) is more costly in terms of machine instructions when compared to L = strlen(S1) + strlen(S2).

Ques 4) What is basic block?

Ans: Basic Block

A basic block is a sequence of program statements (s_1, s_2, s_3 .....s_n) such that only s_n can be a transfer of control statement and only s_1 can be the destination of a transfer of control statement.

Break the code into basic blocks, that is sequences of consecutive statements which may be entered only at the beginning, and when entered are executed in sequence without halt or possibility of branch (except at the end of the basic block).

Algorithm: Partition into basic blocks.
Input: A sequence of three – address statement.

Output: A list of basic blocks with each three – address statement in exactly one block.

Method:

1) We first determine the set of **leaders**, the first statements of basic blocks. The rules we use are the following.
   i) The first statement is a leader.
   ii) Any statement which is the target of a conditional or unconditional goto is a leader.
   iii) Any statement which immediately follows a conditional goto is a leader.
2) For each leader construct its basic block, which consists of the leader and all statements up to but not including the next leader or the end of the program. Any statements not placed in a block can never be executed and may now be removed, if desired.

Ques 5) what is flow graph?

Ans: Flow Graph

It is useful to portray the basic blocks and their successor relationships by a directed graph called a **flow graph.**

The nodes of the flow graph are the basic blocks. One node is distinguished as **initial**; it is the block whose leader is the first statement. There is a directed edge from block B_1 to block B_2 if B_2 could immediately follow B_1 during execution, that is, if

1) There is a conditional or unconditional jump from the last statement of B_1 to the first statement of B_2, or
2) B_2 immediately follows B_1 in the order of the program, and B_1 does not end in an unconditional jump.

We say that B_1 is a **predecessor** of B_2, and B_2 is a successor of B_1.

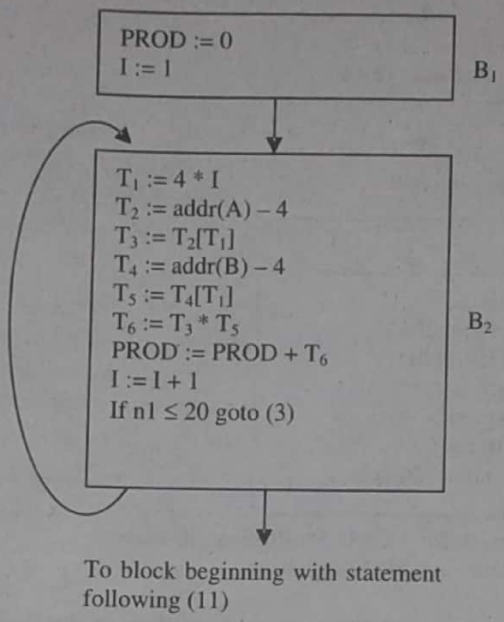


Figure 6.9: Flow Graph

Ques 6) What is loop?

Ans: Loops

Programming languages constructs like while-statements, do-while statements and for statements naturally give rise to loops in programs. Since virtually every program spends most of its time in executing its loop, it is especially important for a compiler to generate good code for loops. Many code transformations depend upon the identification of loops in a flow-graph.

A loop is a collection of nodes that is strongly connected, that is, from any node in the loop to any other, there is a path of length one or more, wholly within the loop, and has a unique entry, a node in the loop such that the only way to reach a node of the loop from a node outside the loop is to first go through the entry.

For example, the flow graph of **figure 6.10** has three loops:

1) B_3 by itself.
2) B_6 by itself.
3) $\{B_2, B_3, B_4\}$

The first two are single nodes with an edge to the node itself.

For example, B_3 forms a loop with B_3 as its entry. Note that the second requirement for a loop is that there is a non-empty path from B_3 to itself. Thus, a single node like B_2, which does not have an edge $B_2 \rightarrow B_2$, is not a loop, since there is no non-empty path from B_2 to itself within $\{B_2\}$.

The third loop, $L = \{B_2, B_3, B_4\}$, has B_2 as its loop entry. Note that among these three nodes, only B_2 has a predecessor, B_1, that is not in L. Further, each of the three nodes has a non-empty path to B_2 staying within L.

For example, B_2 has the path $B_2 \rightarrow B_3 \rightarrow B_4 \rightarrow B_2$.

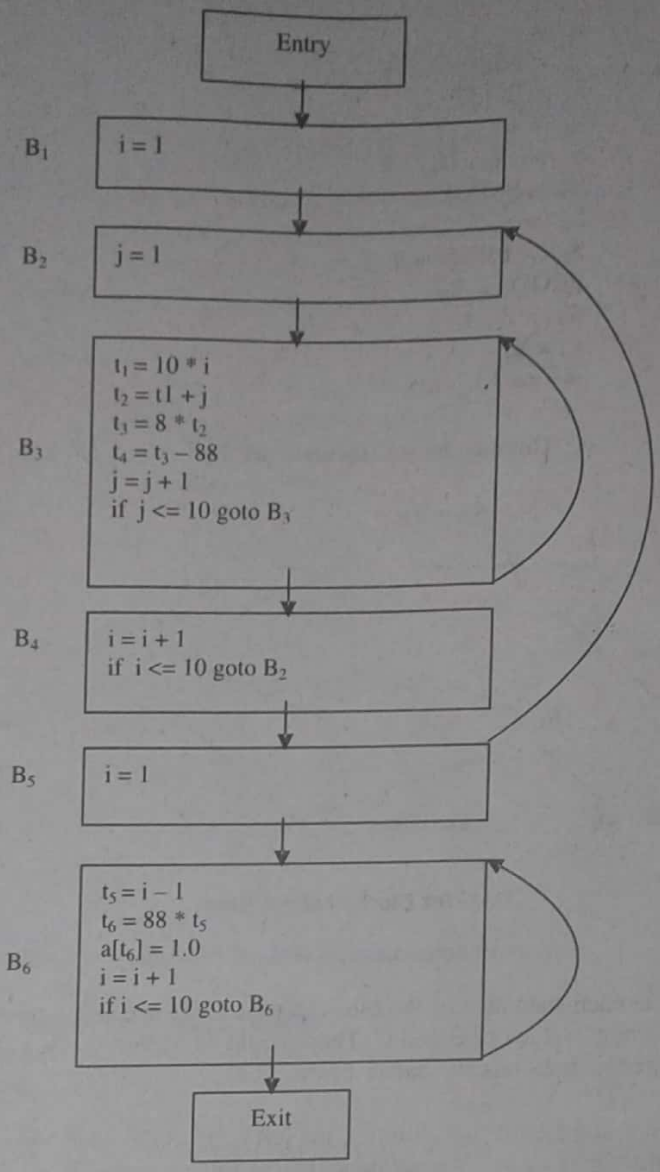


Figure 6.10: Flow Graph

Ques 7) What is the meaning of DAG representation of basic blocks? How DAG is constructed?

Ans: DAG Representation of Basic Blocks

A useful data structure for automatically analysing basic blocks is a **directed acyclic graph** (hereafter called a **DAG**). A DAG is a directed graph with no cycles, which gives a picture of how the value computed by each statement in a basic block is used in subsequent statements in the block.

Constructing a DAG from three – address statements is a good way of determining common sub expressions within a block, determining which names are used inside the block but evaluate outside the block, and determining which statements of the block could have their value used outside the block.

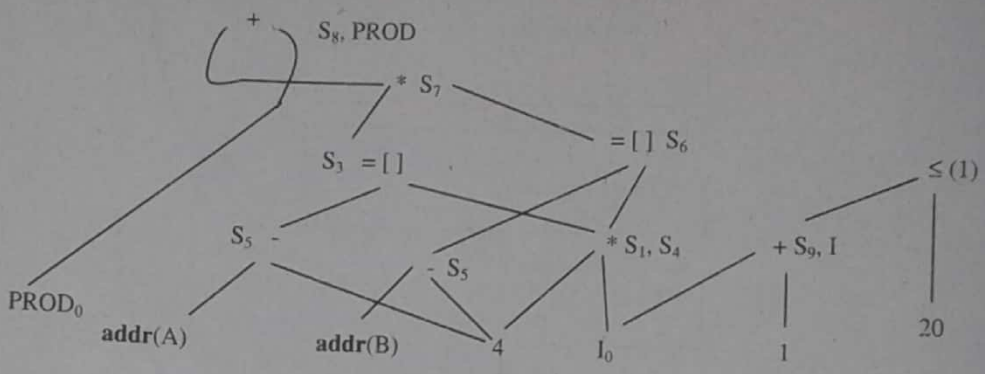
A computation DAG (or just DAG) is a directed acyclic graph with the following labels on nodes:

1) Leaves are labeled by unique identifiers, either variable names or constants. It is convenient to use leaf labels like **addr**(A) to denote the l – value of A, while other identifiers are assumed to denote r – values. The leaves represent initial values of names, and we shall subscript them with 0 to avoid confusion with labels denoting "current" values of names as in (3) below.
2) Interior nodes are label by an operator symbol.
3) Nodes are also optionally given an extra set of identifiers for labels. The intention is that interior nodes represent computed values, and identifiers labeling a node are deemed to have that value.

The DAG representation of a basic block is nothing more nor less than an optimized version of the "triples".

$S_1 := 4 * I$
$S_2 := \textbf{addr}(A) - 4$
$S_3 := S_2 [S_1]$
$S_4 := 4 * I$
$S_5 := \textbf{addr}(B) - 4$
$S_6 := S_5 [S_4]$
$S_7 := S_3 * S_6$
$S_8 := PROD + S_7$
$PROD := S_8$
$S_9 := I + 1$
$I := S_9$
if $I <= 20$ goto (1)

Three-address Code for Block B_2



DAG for Block of above figure

DAG Construction

To construct a DAG we look at each statement of the block in turn. If we see a statement like A : = B + C we look for the nodes which represent the "current" values of B and C. These could be leaves, or they could be interior nodes of the DAG if B and/or C had been evaluated by previous statements of the block.

We then create a node labeled + and give it two children; the left child is the node for B, the right the node for C. then we label this node A. However, if there is already a node denoting the same value as B + C, we do not create a node, but rather gibe this existing node the additional label A.

Two details should be mentioned. First, if a (not A) had previously labeled some other node, we remove that label, since the "current" value of A is the node just created. Second, for an assignment such as A: = B we do not create a new node. Rather, we append label A to the node for the "current" value of B.

We shall now give the algorithm to compute a DAG from a block. The reader should be warned that this algorithm may not operate correctly if there are assignments to arrays, if there are indirect assignments through pointers in the block, or if one memory location can be referred to by two or more names, due to EQUIVALENCE statement of a procedure call.

Algorithm for Constructing DAG

Input: A basic block.

Output: A DAG with the following information:

1) A label for each node. For leaves the label is an identifier (constants permitted), and for interior nodes, an operator symbol.
2) For each node a (possibly empty) list of **attached identifiers** (constants not permitted here).

Method:

The DAG construction process is to do the following steps (1) through (3) for each statement of the block, in turn. Initially we assume there are no nodes, and NODE ( ) is undefined for all arguments. Suppose the "current" three address statement is either (i) A : = B **op** C, (ii) A : = **op** B, or (iii) A : = B. We refer to these as cases (i), (ii), and (iii).

We treat a relational operator like **if** I <= 20 **goto** as case (i) , with a undefined.

1) If NODE(B) is undefined, create a leaf labeled B, and let NODE(B) be this node. In case (i), if NODE(C) is undefined, create a leaf labeled C and let that leaf be NODE(C).

2) In case (i), determine if there is a node labeled op, whose left child is NODE(B) and whose right child is NODE(C). (This is to catch common sub expressions.) If not, create such a node. In either event, lit n be the node found or created. In case (ii), determine whether there is a node labeled op, whose lone child is NODE(B). If not, create such a node, and let n be the node found or created. In case (iii), let n be NODE(B).

3) Append A to the list of attached identifiers for the node n found in (2). Delete A from the list of attached identifiers for node (A). Finally set NODE(A) to n.

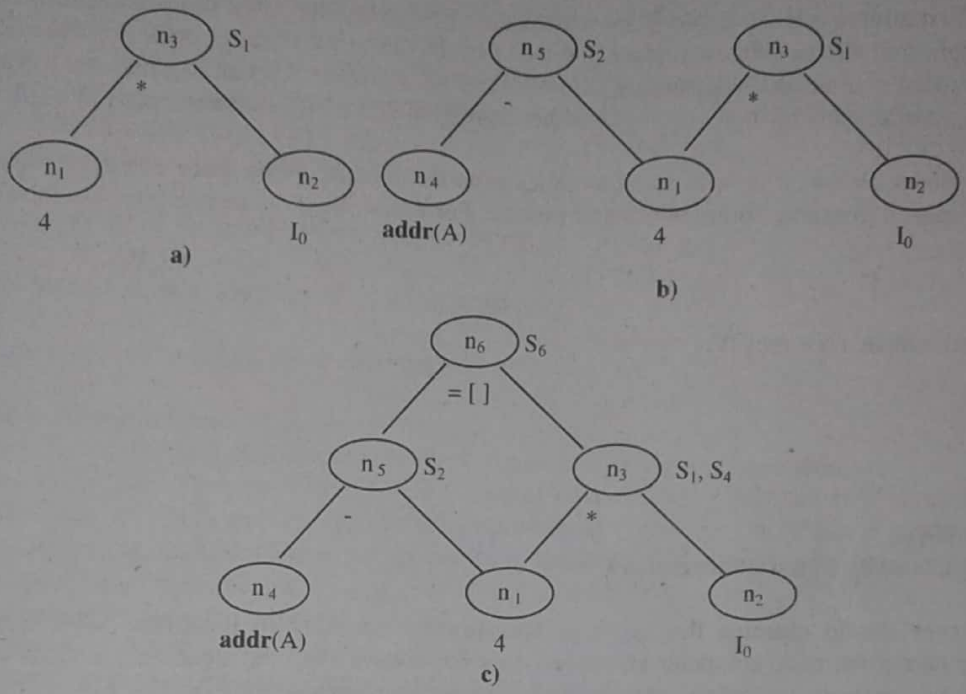


Figure 6.11: Steps in the DAG Construction Process

Ques 8) Discuss about the optimization of basic blocks.

Ans: optimization of basic blocks

There are two types of basic block optimizations. They are:

1) **Structure-Preserving Transformations:** The primary Structure-Preserving Transformation on basic blocks are:
   i) **Common Sub-Expression Elimination:** Common sub expressions need not be computed over and over again. Instead they can be computed once and kept in store from where it's referenced.

Example:

a: =b + c
b: =a – d
c: =b + c
d: =a – d

The 2nd and 4th statements compute the same expression: b + c and a – d

Basic block can be transformed to:

a: = b + c
b: = a – d
c: = a
d: = b

ii) **Dead Code Elimination:** It is possible that a large amount of dead (useless) code may exist in the program. This might be especially caused when introducing variables and procedures as part of construction or error-correction of a program - once declared and defined, one forgets to remove them in case they serve no purpose. Eliminating these will definitely optimize the code.

iii) **Renaming of Temporary Variables:** A statement t:=b + c where t is a temporary name can be changed to u:=b + c where u is another temporary name, and change all uses of t to u. In this a basic block is transformed to its equivalent block called normal-form block.

iv) **Interchange of Two Independent Adjacent Statements:** Two statements,
t1:=b + c
t2:=x + y

can be interchanged or reordered in its computation in the basic block when value of t1 does not affect the value of t2.

2) **Algebraic Transformations:** Algebraic identities represent another important class of optimizations on basic blocks. This includes simplifying expressions or replacing expensive operation by cheaper ones i.e. reduction in strength. Another class of related optimizations is constant folding. Here we evaluate constant expressions at compile time and replace the constant expressions by their values. Thus the expression 2*3.14 would be replaced by 6.28.

The relational operators <=, >=, <, >, + and = sometimes generate unexpected common sub expressions. Associative laws may also be applied to expose common sub expressions. **For example,** if the source code has the assignments:
a :=b + c
e :=c + d + b

The following intermediate code may be generated:
a : = b + c
t :=c + d
e :=t + b

Example:
x:=x+0 can be removed
x:=y**2 can be replaced by a cheaper statement x:=y*y

The compiler writer should examine the language specification carefully to determine what rearrangements of computations are permitted, since computer arithmetic does not always obey the algebraic identities of mathematics. Thus, a compiler may evaluate x*y–x*z as x*(y-z) but it may not evaluate a + (b – c) as (a + b) – c.

CODE GENERATION

Ques 9) What do you mean by code generation? What are the problems with it?

Ans: Code Generation
The final phase in our 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 requirements traditionally imposed on a code generator are severe. The output code must be correct and of high quality, meaning that it should make effective use of the resources of the target machine. Moreover, the code generator itself should run efficiently.

Code generation is the process by which a compiler's code generator converts some intermediate representation of source code into a form (e.g., machine code) that can be readily executed by a machine (often a computer).

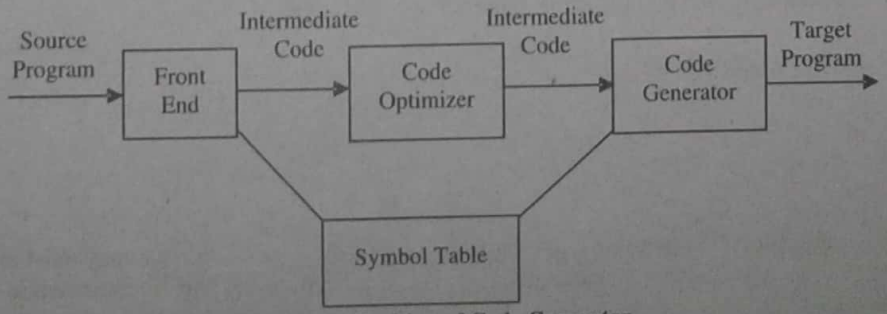


Figure 6.12: Position of Code Generator

Sophisticated compilers typically perform multiple passes over various intermediate forms. This multi-stage process is used because many algorithms for code optimization are easier to apply one at a time, or because the input to one optimization relies on the processing performed by another optimization. This organization also facilitates the creation of a single compiler that can target multiple architectures, as only the last of the code generation stages (**backend**) needs to change from target to target.

The input to the code generator typically consists of a parse tree or an abstract syntax tree. The tree is converted into a linear sequence of instructions, usually in an intermediate language such as three address code. Further stages of compilation may or may not be referred to as "code generation", depending on whether they involve a significant change in the representation of the program.

Problems in Code Generation

There are three problems that arise when we attempt to generate an efficient target code.

1) Selection of instructions to represent the computation that is specified by the three-address statement.
2) To decide the order of computation that leads to generate more efficient code.
3) Deciding the registers for computation purpose.

Ques 10) Discuss the issues in the design of a code generator.
Or
How memory will be managed at the time of code generation?
Or
How instructions are selected at the time of code generation?

Ans: Issues in the Design of a Code Generator

Tasks which are typically part of a sophisticated compiler's "code generation" phase include:

1) **Input to Code Generator:** The input to the code generator consists of the intermediate representation of the source program produced by the front end, together with information in the symbol table that is used to determine the run time addresses of the data objects denoted by the names in the intermediate representation. There are several choices for the intermediate language, including:
   i) **Linear representations** such as postfix notation,
   ii) **Three address representations** such as quadruples,
   iii) **Virtual machine representations** such as syntax trees and dags.

   We assume that prior to code generation the front end has scanned, parsed, and translated the source program into a reasonably detailed intermediate representation, so the values of names appearing in the intermediate language can be represented by quantities that the target machine can directly manipulate (bits, integers, reals, pointers, etc.).

   We also assume that the necessary type checking has take place, so type conversion operators have been inserted wherever necessary and obvious semantic errors (e.g., attempting to index an array by a floating point number) have already been detected. The code generation phase can therefore proceed on the assumption that its input is free of errors. In some compilers, this kind of semantic checking is done together with code generation.
2) **Target Programs:** The output of the code generator is the target program. The output may take on a variety of forms:
   i) Absolute machine language,
   ii) Relocatable machine language or assembly language.

   Producing an absolute machine language program as output has the advantage that it can be placed in a location in memory and immediately executed. A small program can be compiled and executed quickly.

   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 by a linking loader.

   Although we must pay the added expense of linking and loading if we produce relocatable object modules, 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.

   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 generate code. The price paid is the assembly step after code generation. Because producing assembly code does not duplicate the entire task of the assembler, this choice is another reasonable alternative, especially for a machine with a small memory, where a compiler must uses several passes.

3) **Memory Management:** Mapping names in the source program to addresses of data objects in run time memory is done cooperatively by the front end and code generator. A name in a three-address statement refers to a symbol table entry for the name.

If machine code is being generated, labels in three address statements have to be converted to addresses of instructions. This process is analogous to the **"back patching"**.

Let suppose that labels refer to quadruple numbers in a quadruple array. As one scan each quadruple in turn he/she can deduce the location of the first machine instruction generated for that quadruple, simply by maintaining a count of the number of words used for the instructions generated so far. This count can be kept in the quadruple array (in an extra field), so if a reference such as j: goto i is encountered, and i is less than j, the current quadruple number, one may simply generate a jump instruction with the target address equal to the machine location of the first instruction in the code for quadruple i.

If, however, the jump is forward, so i exceeds j, one must store on a list for quadruple i the location of the first machine instruction generated for quadruple j. Then we process quadruple i, fill in the proper machine location for all instructions that are forward jumps to i.

4) **Instruction Selection:** This task tells which instructions to use. The nature of the instruction set of the target machine determines the difficulty of instruction selection. The uniformity and completeness of the instruction set are important factors. If the target machine does not support each data type in a uniform manner, then each exception to the general rule requires special handling.

Instruction selection is typically carried out by doing a recursive postorder traversal on the abstract syntax tree, matching particular tree configurations against templates.

For example, the tree W:= ADD(X, MUL(Y, Z)) might be transformed into a linear sequence of instructions by recursively generating the sequences for t_1 := X and t_2 := MUL(Y,Z), and then emitting the instruction ADD W, t_1, t_2.

In a compiler that uses an intermediate language, there may be two instruction selection stages:
i) One to convert the parse tree into intermediate code,
ii) A second phase much later to convert the intermediate code into instructions from the instruction set of the target machine.

5) **Register Allocation:** The allocation of variables to processor registers. Instructions involving register operands are usually shorter and faster than those involving operands in memory. Therefore, efficient utilization of register is particularly important in generating good code. The use of registers is often subdivided into two subproblems:
i) During **register allocation,** select the set of variables that will reside in registers at a point in the program.
ii) During a subsequent **register assignment** phase, pick the specific register that a variable will reside in.

Finding an optimal assignment of registers to variables is difficult, even with single register values. Mathematically, the problem is NP-complete. The problem is further complicated because the hardware and/or the operating system of the target machine may require that certain register usage conventions be observed.

6) **Choice of Evaluation Order:** The order in which computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others. Picking a best order is another difficult, NP-complete problem. Initially, one shall avoid the problem by generating code for the three-address statements in the order in which they have been produced by the intermediate code generator.

Ques 11) Write the algorithm for code generation.

Ans: Algorithm for Code Generation

An algorithm for code generation is as below. For every three-address statement of the form x = y op z in the basic block do:

1) Call getreg() to obtain the location L in which the computation y op z should be performed. /* This requires passing the three-address statement x = y op z as a parameter to getreg(), which can be done by passing the index of this statement in the quadruple array.

2) Obtain the current location of the operand y by consulting its address descriptor, and if the value of y is currently both in the memory location as well as in the register, then prefer the register. If the value of y is currently not available in L, then generate an instruction MOV y, L (where y as assumed to represent the current location of y).

3) Generate the instruction OP z, L, and update the address descriptor of x to indicate that x is now available in L, and if L is in a register, then update its descriptor to indicate that it will contain the run-time value of x.

4) If the current values of y and/or z are in the register, and we have no further uses for them, and they are not live at the end of the block, then alter the register descriptor to indicate that after the execution of the statement x = y op z, those registers will no longer contain y and/or z.

That Store all the results.

The function getreg(), when called upon to return a location where the computation specified by the three-address statement x = y op z should be performed, returns a location L as follows:

1) It searches for a register already containing the name y. If such a register exists, and if y has no further use after the execution of x = y op z, and if it is not live at the end of the block and holds the value of no other name, then return the register for L.

2) Otherwise, getreg() searches for an empty register; and if an empty register is available, then it returns it for L.

3) If no empty register exists, and if x has further use in the block, or op is an operator such as indexing that requires a register, then getreg() finds a suitable, occupied register. The register is emptied by storing its value in the proper memory location M, the address descriptor is updated, the register is returned for L. (The least-recently used strategy can be used to find a suitable, occupied register to be emptied.)

4) If x is not used in the block or no suitable, occupied register can be found, getreg() selects a memory location and returns it for L.

Example: Consider the expression:
x = (a + b) – ((c + d) – e)

The three-address code for this is:
t1 = a + b
t2= c + d
t3= t2 – e
x = t1 – t3

Solution: Computation for the Expression x = (a + b) – ((c + d) – e)

Table 6.1: Computation for the Expression x = (a + b) – ((c + d) – e)

Statement	L	Instructions Generated	Register Descriptor	Address Descriptor
			All registers empty	
t1 = a + b	R0	MOV a, R0 ADD b, R0	R0 will hold t1	t1 is in R0
t2 = c + d	R1	MOV c, R1 ADD d, R1	R1 will hold t2	t2 is in R1
t3 = t2 – e	R1	SUB e, R1	R1 will hold t3	t3 is in R1
x = t1 – t3	R0	SUB R1, R0	R0 will hold x	x is in R0
		MOV R0, x		x is in R0 and memory

Ques 12) Define the code generation of if and while loop.

Ans: Code Generation for If and While Statements
Now consider the following two forms of the if- and while-statements, which are similar in many different languages (but which we give here in a C-like syntax):

if-stmt → if (exp) stmt | if (exp).stmt else stmt
while-stmt → while (exp) stmt

The chief problem in generating code for such statements is to translate the structured control features into an "unstructured" equivalent involving jumps, which can be directly implemented. Compilers arrange to generate code for such statements in a standard order that allows the efficient use of a subset of the possible jumps that a target architecture might permit.

Typical code arrangements for each of these statements are shown in **figures 6.13** and **6.14**.

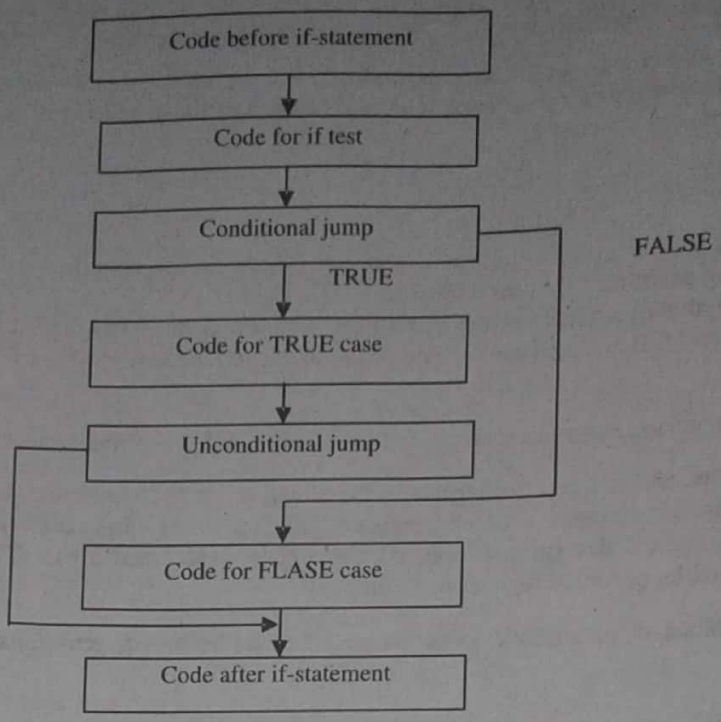


Figure 6.13: Typical Code Arrangement for an If-Statement

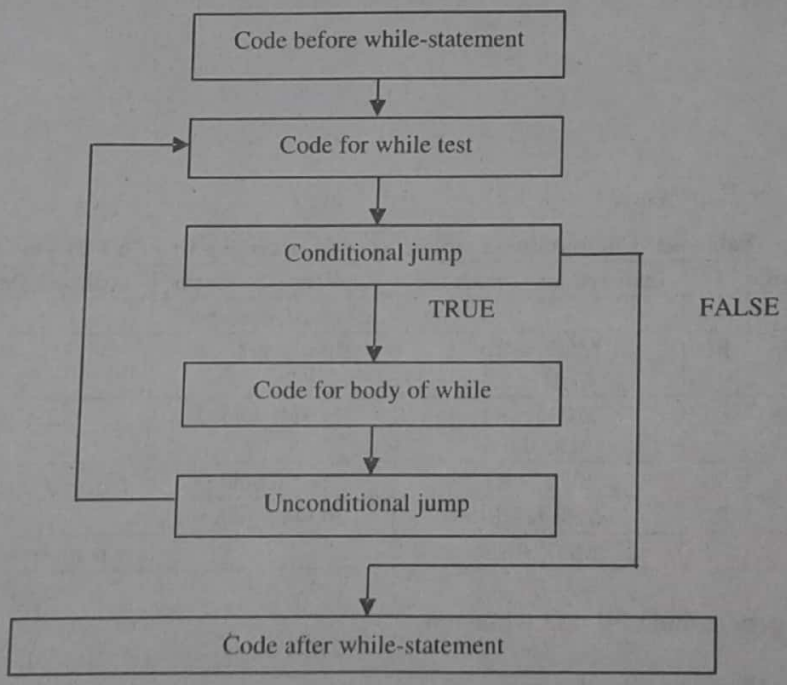


Figure 6.14: Typical Code Arrangement for a While-Statement

Ques 13) Explain the machine model with respect to the code generator.

Ans: Machine Model

Consider a byte-addressable machine with 2^{16} bytes (2^{15} 16-bit words) of memory. We have eight general-purpose registers R0, RI,..., R7 each capable of holding a 16-bit quantity. We have binary operators of the form:

OP source, destination

In which OP is a 4-bit op-code, and source and destination are 6-bit fields. Since these 6-bit fields are not long enough to hold memory addresses, certain bit patterns in these fields specify that words following an instruction will contain operands and/or addresses.

The following addressing modes will be assumed. They are given with their assembly-language mnemonic forms.

1) **r (register mode):** Register r contains the operand.
2) ***r (indirect register mode):** Register r contains the address of the operand.
3) **X(r) (indexed mode):** Value X, which is found in the word following the instruction, is added to the contents of register r to produce the address of the operand.
4) ***X(r) (indirect indexed mode):** Value X, stored in the word following the instruction, is added to the contents of register r to produce the address of the word containing the address of the operand.
5) **#X (immediate):** The word following the instruction contains the literal operand X.
6) **X (absolute):** The address of X follows the instruction.

Use the following op-codes (among others):

MOV (move source to destination)
ADD (add source to destination)
SUB (subtract source from destination

Examples of machine instructions follow:

1) The instruction MOV R0, R1 copies the contents of register 0 into register 1. This instruction has cost one, since it occupies only one word of memory.
2) The (store) instruction MOV R5, M copies the contents of R5 into memory location M. This instruction has cost two, since the address of memory location M is in the word following the instruction.
3) The instruction ADD #1, R3 adds the constant 1 to the contents of R3, and has cost 2, since the constant 1 must appear in the next word.
4) The instruction SUB 4(R0),*5(R1) subtracts ((R0) + 4) from (((Rl) + 5)) where (X) denotes the contents of register or location X. The result is stored at the destination *5(R1). The cost of this instruction is 3, since the constants 4 and 5 are stored in the next two words following the instruction.

One should immediately see some of the difficulties in generating code for this machine. For a quadruple of the form A := B + C where B and C are simple variables in distinct memory locations of the same name, one can generate a variety of code sequences:

1) MOV B, R0
   ADD C, R0 cost = 6
   MOV R0, A
2) MOV B, A cost = 6
   ADD C, A
3) MOV *R1, *R0 cost = 2
   ADD *R2, *R0
   Assuming R0, Rl and R2 contain the addresses of A, B, and C respectively.
4) ADD R2, Rl cost = 3
   MOV Rl, A
   Assuming Rl and R2 contain the values of B and C respectively, and the value of B is not live after the assignment.

One can see that in order to generate good code for this machine, he/she must utilise its addressing capabilities efficiently. There is a premium on keeping the l- or r-value of a name in a register, if possible, if it is going to be used in the near future.

Ques 14) What is target machine? What is the instruction costs?

Ans: Target Machine

The target machine, after being built can execute the user's program directly by hardwired logic or microprogramming logic. However, if the target machine is not available, one can instruct the interpreter to interpret its instruction set as long as he/she understand the target machine spec thoroughly.

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:
1) MOV (move source to destination)
2) ADD (add source to destination)
3) SUB(Subtract source from destination)

The **source** and **destination** of an instruction are specified by combining registers and memory locations with address modes.

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

For example, MOV R0, M stores contents of Register R0 into memory location M.

Instruction Costs
Instruction cost = 1+ cost for source and destination address modes. This cost corresponds to the length of the instruction.

Address modes involving registers have cost zero. Address modes involving memory location or literal have cost one.

Instruction length should be minimized if space is important. Doing so also minimizes the time taken to fetch and perform the instruction.

For example, MOV R0, R1 copies the contents of register R0 into R1. It has cost one, since occupies only one word of memory.

The three-address statement a: = b + c can be implemented by many different instruction sequences:
1) MOV b, R0

   ADD c, R0 cost = 6 MOV R0, a

2) MOV b, a

   ADD c, a cost = 6

3) Assuming R0, R1 and R2 contain the addresses of a, b, and c: MOV *R1, *R0

   ADD *R2, *R0 cost = 2

In order to generate good code for target machine, one must utilise its addressing capabilities efficiently.

Examples
1) Move register to memory R0 ← M.
   MOV R0, M

   Cost = 1 + 1 = 2.

2) Indirect indexed mode:
   MOV * 4 (R0), M

   Cost = 1 plus indirect index plus instruction word
   = 1 + 1 + 1 = 3

3) Indexed mode:
MOV 4(R0), M

Cost = 1 + 1 + 1 = 3

4) Literal mode:
MOV #1, R0

Cost = 1 + 1 = 2

Ques 15) Discuss in detail about a simple code generator with the appropriate algorithm.

Or

Explain code generation phase with simple code generation phase with simple code generation algorithm.

Ans: Simple Code Generator

A simple code generator generates the target code for the three-address statements. The main issue during code generation is the utilisation of registers since the number of registers available is limited. The code generation algorithm takes the sequence of three-address statements as input and assumes that for each operator, there exists a corresponding operator in target language.

The machine code instruction takes the required operands in registers, performs the operation and stores the result in a register. Register and address descriptors are used to keep track of register contents and addresses.

1) Register descriptors are used to keep track of the contents of each register at a given point of time. Initially, we assume that a register descriptor shows that all registers are empty and as the code generation proceeds, each register holds the value of zero or more names at some point.

2) Address descriptors are used to trace the location of the current value of the name at run time. The location may be memory address, register, or a stack location, and this information can be stored in the symbol table to determine the accessing method for a name.

The code generation algorithm for a three-address statement X: = Y op Z is given below:

Algorithm for Code Generation

An algorithm for code generation is as below. For every three-address statement of the form X = Y op Z in the basic block do:

i) Call getreg() to obtain the location L in which the computation Y op Z should be performed. This requires passing the three-address statement X = Y op Z as a parameter to getreg(), which can be done by passing the index of this statement in the quadruple array.

ii) Obtain the current location of the operand Y by consulting its address descriptor, and if the value of Y is currently both in the memory location as well as in the register, then prefer the register. If the value of Y is currently not available in L, then generate an instruction MOV Y, L (where y as assumed to represent the current location of Y).

iii) Generate the instruction op Z, L, and update the address descriptor of X to indicate that x is now available in L, and if L is in a register, then update its descriptor to indicate that it will contain the run-time value of X.

iv) If the current values of Y and/or Z are in the register, and we have no further uses for them, and they are not live at the end of the block, then alter the register descriptor to indicate that after the execution of the statement X = Y op Z, those registers will no longer contain Y and/or Z.

That Store all the results. The function **getreg()**, when called upon to return a location where the computation specified by the three-address statement X = Y op Z should be performed, returns a location L as follows:

i) It searches for a register already containing the name Y. If such a register exists, and if Y has no further use after the execution of X = Y op Z, and if it is not live at the end of the block and holds the value of no other name, then return the register for L.

ii) Otherwise, getreg() searches for an empty register; and if an empty register is available, then it returns it for L.

iii) If no empty register exists, and if X has further use in the block, or op is an operator such as indexing that requires a register, then getreg() finds a suitable, occupied register. The register is emptied by storing its value in the proper memory location M, the address descriptor is updated, the register is returned for L. (The least-recently used strategy can be used to find a suitable, occupied register to be emptied.)

iv) If X is not used in the block or no suitable, occupied register can be found, getreg() selects a memory location and returns it for L.

3) If Y is in a register then the register and address descriptors are altered to record that from now onwards the value of Y is found only in the register that holds the value of Y.

4) If Y is in the memory, the getreg ( ) function is used to determine a register in which the value of Y is to be loaded, and that register is now made as the location of X.

Thus, the instruction of the form X: = Y could cause the register to hold the value of two or more variables simultaneously.

Implementing the Function getreg()

For the three-address statement, X: = Y op Z, the function getreg ( ) returns a location L as follows:

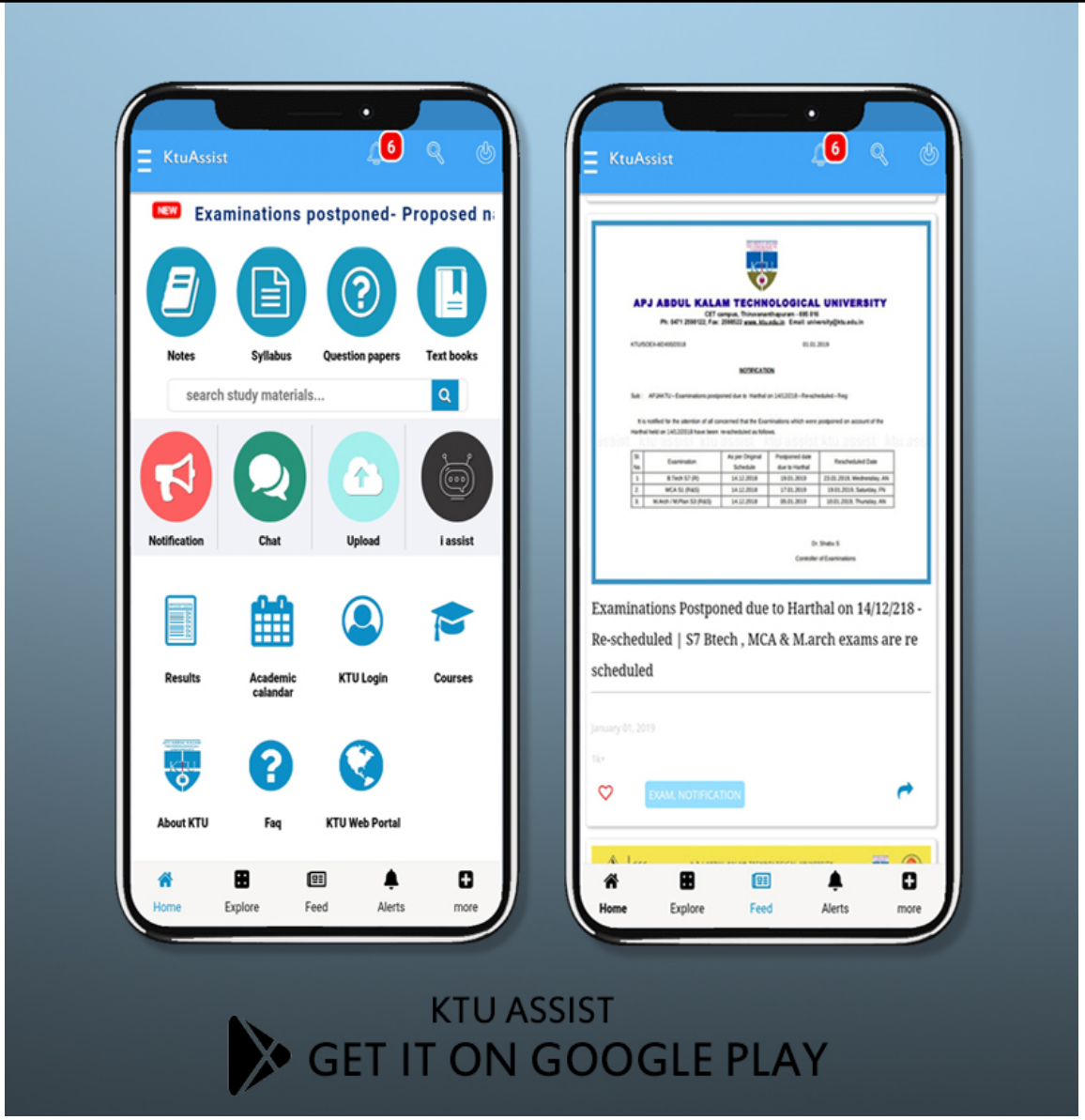
1) If Y is in a register and it is not live and has no next uses after the execution of three-address statement, then return the register of Y for L. The address descriptor of Y is then updated to indicate that Y is no more present in L.

2) If Y is not in a register, then return an empty register for L (if exists).

3) If X is to be used further in the block or OP is the operator that requires a register, then find an occupied register R_0 that may contain one or more values. For each variable in R_0, issue a MOV R_0, M instruction to store the value of R_0 into a memory location M. Then, update the address descriptor for M, and return R_0.

Though there are several ways to choose a suitable occupied register, the simplest way is to choose the one whose data values are to be referenced furthest in the future.

4) If X is not used in the block or an occupied register could not be found, then select the memory location of X as L.



END

facebook.com/ktuassist

instagram.com/ktu_assist