**UNIT V**

**Basic Optimization**

**Basic Optimization:** Constant-Expression Evaluation–Algebraic Simplifications and Re association–Copy Propagation – Common Sub-expression Elimination – Loop-Invariant Code Motion – Induction Variable Optimization.

**Code Generation:** Issues in the Design of Code Generator–The Target Machine–Runtime Storagemanagement – Next-use Information – A simple Code Generator – DAG Representation of Basic Blocks – Peephole Optimization – Generating Code from DAGs

P a g e | **1** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**2 MARKS**

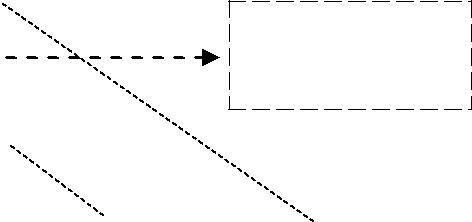
**1. What is code optimization?**

Code optimization techniques are generally applied after syntax analysis, usually both before and during code generation. The techniques consist of detecting patterns in the program and replacing these patterns by equivalent and more efficient constructs. This improvements is achieved by program transformation are called ***optimization.***

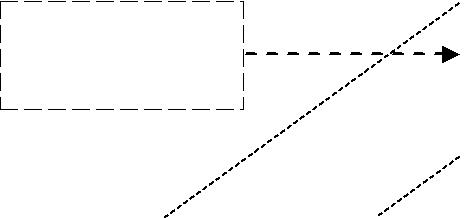
1. **What is optimizing compilers?**

Compilers that apply code-improving transformations are called ***optimizing compilers.***

1. **Give the block diagram of organization of code optimizer.**



|  |  |  |  |
| --- | --- | --- | --- |
| Front end |  | Code | Code |
|  |  | optimizer | generator |
|  |  |  |  |
|  |  |  |  |



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Control |  | Data Flow |  | Transform |
| flow |  | analysis |  | ations |
| analysis |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |



1. **What are the properties of optimizing compilers?** 
   * Transformation must preserve the meaning of programs.
   * Transformation must, on the average, speed up the programs by a measurable amount.
   * A Transformation must be worth the effort.
2. **What are the advantages of the organization of code optimizer?** 
   * The operations needed to implement high level constructs are made explicit in the intermediate code, so it is possible to optimize them.
   * The intermediate code can be independent of the target machine, so the optimizer does not have to change much if the code generator is replaced by one for a different machine
3. **What are the 3 areas of code optimization?** 
   * + Local optimization
     + Loop optimization
     + Data flow analysis

P a g e | **2** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**7. Define local optimization.**

The optimization performed within a block of code is called a local optimization.

1. **Define constant folding.** 
   * Deducing at compile time that the value of an expression is a constant and using the constant instead is known as ***constant folding.***
   * Constant folding is nothing but replacing the run-time compilation by the compile time compilation. This is done generally for the constants.
   * Example: a:=(22/7) \* (r\*r)
2. **What is propagation?** 
   * Propagation means propagating an entity from one statement to another statement. This is done for constants.
   * Evaluation or replace a variable with constant which has been assigned to it earlier.
3. **Define Local transformation & Global Transformation.** 
   * A transformation of a program is called ***Local***, if it can be performed by looking only at the statements in a basic block.
   * Otherwise it is called ***global***.
   * Many transformations can be performed at both local and global levels.
   * Local transformations are usually performed first.
4. **Give the criteria for code-improving transformations. (NOV 2011)** 
   * Common sub expression elimination
   * Copy propagation
   * Dead – code elimination
   * Constant folding
5. **What is meant by Common Sub expressions?**

An occurrence of an expression E is called a ***common sub expression***, if E was previously computed, and the values of variables in E have not changed since the previous computation.

**13. What is copy propagation?**

The assignment of the form *f := g* called copy statements or copies.

1. **What is meant by Dead Code?**

A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point. The statement that computes values that never get used is known ***Dead code*** or ***useless code.***

P a g e | **3** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

1. **What are the techniques used for loop optimization?**

Three techniques are important for loop optimizations are

* + - * 1. Code motion
        2. Induction variable elimination
        3. Reduction in strength

1. **What is code motion?** 
   * ***Code motion***, which moves code outside a loop.
   * *Code motion* is an important modification that decreases the amount of code in a loop.
   * This transformation takes an expression that yields the same result independent of the number of times a loop is executed ( a loop-invariant computation) and places the expression before the loop.
2. **Define Induction variable? (MAY 2013)**

The values of j and t4 remain in lock-step; every time the value of j decreases by 1, that of t4 decreases by 4 because 4\*j is assigned to t4. Such identifiers are called ***induction variables.***

1. **What is meant by Reduction in strength? (MAY 2012)**

*Reduction in strength*, which replaces an expensive operation by a cheaper one, such as a multiplication byan addition.

1. **What is meant by loop invariant computation?**

The transformation takes an expression that yields the same result independent of the number of times the loop is executed is known as loop invariant computation.

1. **What is 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.
2. **What are the issues in the design of a code generator?**

The various issues in design of code generator are

* + - 1. Input to the Code Generator
      2. Target Programs
      3. Memory Management
      4. Instruction Selection
      5. Register Allocation and
      6. Choice of Evaluation Order

P a g e | **4** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

1. **What are the outputs of code generator?**

The output of the code generator is the target program. The output may take on a variety of forms:

* + 1. absolute machine language,
    2. relocatable machine language, or
    3. assembly language.

1. **What is register allocation?** 
   * Instructions involving register operands are usually shorter and faster than those involving operands in memory. Efficient utilization of register is particularly important in generating good code.
   * The use of registers are
     1. Register allocation
     2. Register assignment
2. **What are the types of address mode?**

The types of address modes in assembly-language

* + Absolute
  + Register
  + Indexed
  + Indirect register
  + Indirect indexed

1. **What is meant by activation record?**

Information needed during an execution of a procedure is kept in a block of storage called activation record; storage for names local to the procedure also appears in the activation record.

1. **What are the two standard storage allocation strategies?**

The two standard allocation strategies are

* + 1. Static allocation
    2. Stack allocation

1. **Define static and stack allocation.** 
   * In *static allocation,* the position of an activation record in memory is fixed at compile time.
   * In *stack allocation,* a new activation record is pushed onto the stack for each execution of a procedure. The record is popped when the activation ends.

P a g e | **5** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**28. What are the fields in activation records?**

The activation record for a procedure has fields as

* + - to hold parameters,
    - results
    - machine status information,
    - local data,
    - temporary variables

1. **What are the limitations of using static allocation?** 
   * The size of a data object and constraints on its position in memory must be known at compile time.

 Recursive procedure are restricted, because all activations of a procedure use the same bindings for local name

* + - Data structures cannot be created dynamically since there is no mechanism for storage allocation at run time

1. **When static allocation can become stack allocation? (NOV 2011)** 
   * Static allocation can become stack allocation by using relative addresses for storage in activation records.
   * The position of the activation record for the procedure is not known until run time.
   * In stack allocation, this position is usually stored in a register, so words in the activation record can be accessed as offsets from the value in this register.
2. **What is a basic block? What are the entry points and how do you call the entry instructions?**

**(MAY 2013)**

A ***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 except at the end.

1. **What are descriptors in code generation algorithm?** 
   * The code-generation algorithm uses descriptors to keep track of register contents and addresses for names.
     + 1. Register descriptors
       2. Address descriptors
2. **What is Register Descriptors?** 
   * + A register descriptor keeps track of what is currently in each register.
     + Initially all the registers are empty.
     + Each register will hold the value of zero or more names at any given time.

P a g e | **6** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

1. **What is Address Descriptors?** 
   * Address descriptors keeps track of location where current value of the name can be found at runtime.
   * The location might be a register, a stack location or a memory address.
   * This information can be stored in the symbol table and is used to determine the accessing method for a name.
2. **What is DAG? (NOV 2012, 2013)** 
   * Directed acyclic graphs (DAG) are useful data structure for implementing transformations on basic blocks.
   * A dag gives a picture of how the value computed by each statement in a basic block is used in subsequent statements of the block.
3. **How DAG is constructing?**

DAG is constructing from three-address statements is a good way of

* + - Determining the common sub-expressions
    - Determining which names are used inside the block but evaluated outside the block and
    - Determining which statements of the block could have their computed value outside the block.

1. **Mention the applications of DAG?** 
   * To automatically detect a common sub expressions.
   * To determine which identifiers have their values used in the block.
   * To determine which statements compute values that could be used outside the block.
2. **Define peephole optimization.**

The technique for locally improving the target code is peephole optimization, a method for trying 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 whenever possible.

1. **List the characteristics of peephole optimization.**

The characteristics of peephole optimization are

* + Redundant instruction elimination
  + Unreachable Code
  + Flow of control optimizations
  + Algebraic simplification
  + Reduction in Strength
  + Use of machine idioms

P a g e | **7** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**40. Construct a 3-address code for (B+A) \* (Y-(B+A)). (NOV 2013)**

The 3-address code for (B+A) \* (Y-(B+A)) t1 := B + A;

t2 := Y - t1; t3 := t1 \* t2;

1. **Define Flooding? (MAY 2012)** 
   * A **flooding algorithm** is an  [algorithm](http://en.wikipedia.org/wiki/Algorithm) for distributing material to every part of a  [graph.](http://en.wikipedia.org/wiki/Graph_%28mathematics%29) The name derives from the concept of inundation by a  [flood](http://en.wikipedia.org/wiki/Flood).
   * Flooding algorithms are used in  [computer networking](http://en.wikipedia.org/wiki/Flooding_%28computer_networking%29) and  [graphics.](http://en.wikipedia.org/wiki/Flood_fill) Flooding algorithms are also useful for solving many mathematical problems, including  [maze](http://en.wikipedia.org/wiki/Maze) problems and many problems in  [graph theory](http://en.wikipedia.org/wiki/Graph_theory).
2. **What is translation of symbol? (NOV 2012)** 
   * A translation of symbols mainly a constant folding and constant propagation.
   * A context-free grammar with semantic actions embedded within the right sides of the productions.

P a g e | **8** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

* 1. **MARKS**

1. **Write a short note on Constant, Expression Evaluation? (6 marks)**

**Constant:**

The code improving transformation as

* + - Constant folding
    - Constant propagation

1. **Constant Folding:** 
   * Constant - expressions evaluation or ***constant folding*** refers to the evaluation at compile time of expressions whose operands are known to be constant.
   * Evaluation of an expression with constant operands to replace the expression with single value.
   * This is actually compile - time evaluation.
   * It makes the possible for the computations performed during the compile time itself, and thus avoids the computation during the execution time.
   * Deducing at compile time that the value of an expression is a constant and using the constant instead is known as ***constant folding.***
   * Constant folding is nothing but replacing the run-time compilation by the compile time compilation. This is done generally for the constants.

**Example**: a :=( 22/7) \* (r \* r)a := 3.14286 \* (r \* r)

* The value (22/7) can be computed during the compilation itself than computing it in each execution.

**Example:** i = 320 \* 200 \* 32

* + Most compilers will substitute the computed value at compile time.

1. **Constant Propagation:** 
   * ***Constant propagation*** means propagating an entity from one statement to another statement.This is done for constants.
   * Evaluation or replace a variable with constant which has been assigned to it earlier.
   * Constant propagation is particularly important when procedures or macros are passed constant parameters.
   * Constant propagation is nothing but replacement of a variable by a constant that appears on the

right hand side of an assignment for that variable.

**Example:** a: =2;

P a g e | **9** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

Consider the three-address statements temp1:=4;

………

temp2:=temp1\*2;

Here, the variable temp1 is propagated. This can be optimized during the compile time itself. temp1:=4;

………

temp2:=4\*2;

This is actually ***constant propagation.***

The variable should not be redefined along the path of its use.

**Example:**

pi: =3.14286

Area: = pi \* r \*\* 2;  area: = 3.14286 \* r\*\* 2; This process is done during the compile time.

1. **Expression Evaluation:** 
   * ***Expressions evaluation*** or constant folding refers to the evaluation at compile time of expressionswhose operands are known to be constant.
   * Determine that all operands in an expression are constant value.
   * Perform the evaluation of the expression at compile time.
   * Replace the expression by its value.
   * Identify common sub-expression present in different expression, compute once, and use the result in all the places.
   * The definition of the variables involved should not change.

**Example:**

a := b \* c;

……. …….

x := b \* c + 5;

To generate three-address code for the above statements

temp1 := b \* c; a := temp1;

……….

x := temp1 + 5;

P a g e | **10** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**2. Write a short note on algebraic Simplifications and Re-association? (6 marks)**

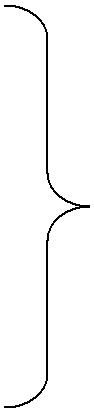
1. **Algebraic Simplifications:** 
   * Algebraic simplification uses algebraic properties of operators or particular operand combinations to simplify expressions.

**Expressions simplification:**

* i +0 = 0 + i = i - 0 = i
* i ^ 2 = i \* i (also strength reduction)
* i\*5 can be done by t := i shl 3; t=t-i
* Associativity and distributive can be applied to improve parallelism (reduce the height of expression trees).
* Algebraic simplifications for floating point operations are seldom applied.
* The reason is that floating point numbers do not have the same algebraic properties as real numbers.

**Examples:**

**1. *Algebraic simplification using the rules***



A\*1 := A

A\*0 := 0

A-0 := A

A/1 := A single instruction with a constant operand A\*2 := A+A

A^2 := A\*A

A\*\*2 := A\*A

**2. *Initial code*** ***Algebraic simplification***

A := X\*\*2; A := X\*X;

B := 3; B := 3;

C := X; C := X;

D := C\*C;  D := C\*C;

E := B\*2; E := B+B;

F := A+D; F := A+D;

G := E\*F; G := E\*F;

P a g e | **11** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

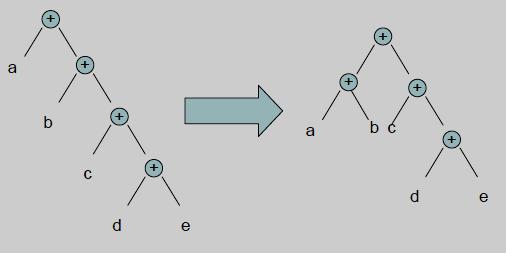
1. Algebraic transformation can be used to change the set of expressions computed by a basic block into an algebraically equivalent set.

b && true := b

b && false := false b || true := true b || false := b

X \* 4 := X<<2 16\* X := X<<4

1. The goal is to reduce height of expression tree to reduce execution time in a parallel environment.



**5. *Common sub-expression elimination***

Transform the program so that the value of a (usually scalar) expression is saved to avoid having to compute the same expression later in the program.

***For example:***

x = e^3+1

…

y= e^3

is replaced (assuming that e is not reassigned in …) with t=e^3

x = t+1

…

y=t

P a g e | **12** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

1. ***Copy propagation*** 
   * Eliminates unnecessary copy operations.

***For example:***

* + 1. = y

<other instructions> t = x + 1

Is replaced (assuming that neither x nor y are reassigned in …) with

<other instructions> t = y + 1

 Copy propagation is useful *after* common sub-expression elimination.

***For example:***

x = a+b

…

y = a+b

Is replaced by common sub-expression elimination into the following code t = a+b

x = t

…

z = x

y = a+b

Here x=t can be eliminated by copy propagation*.*

1. **Algebraic Re-association:** 
   * Re-association refers to using associativity, commutativity, and distributivity to divide an expression into parts that are constant, loop invariant and variable.
   * The optimization that can remove useless instructions entirely via algebraic identities.

**Example:**

|  |  |  |
| --- | --- | --- |
| Consider the assignment statement **b=5+a+10** | |  |
| The three address code for the above sequence | |  |
| temp1=5; | temp1=5; | temp1=15+a; |
| temp2=temp1+a; | temp2=temp1+a; | b=temp1; |
| temp3=temp2+10; | b=temp1; |  |
| b=temp3; |  |  |



P a g e | **13** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**3. Explain principal sources of optimization or Local optimization techniques. (11 marks) (NOV 2012 MAY 2013)**

* A transformation of a program is called ***Local***, if it can be performed by looking only at the statements in a basic block.
* Otherwise it is called ***global***.
* Many transformations can be performed at both local and global levels.
* Local transformations are usually performed first.

**Function Preserving Transformations:**

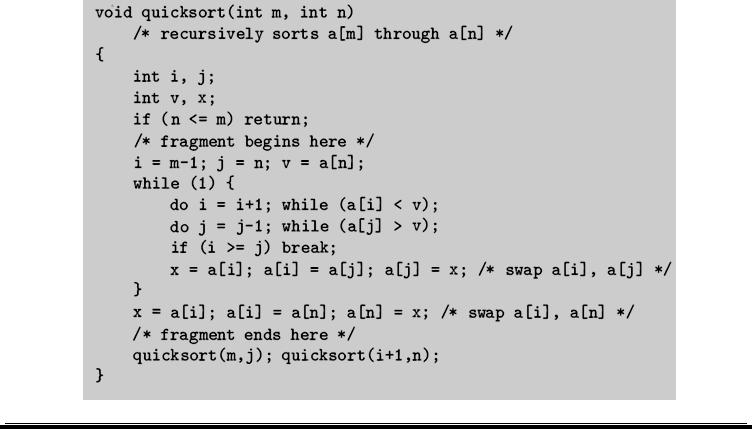
* There are a number of ways in which a compiler can improve a program without changing the function it computes.

The examples of function preserving transformations are

1. Common sub-expression elimination
2. Copy propagation
3. Dead code elimination
4. Constant folding

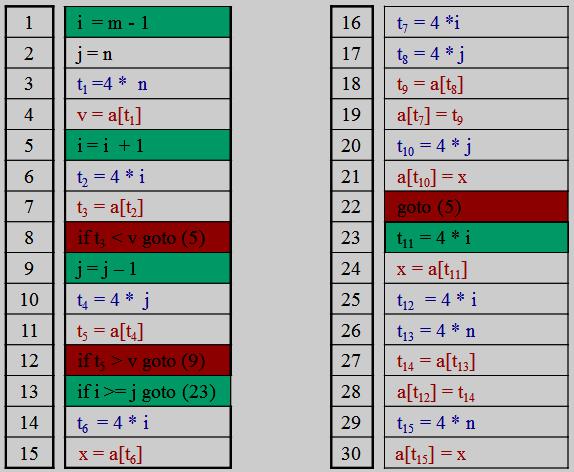
The DAG representation of basic blocks showed how local common sub-expressions could be removed as the DAG for the basic block is constructed.

**Quick sort for c program:**

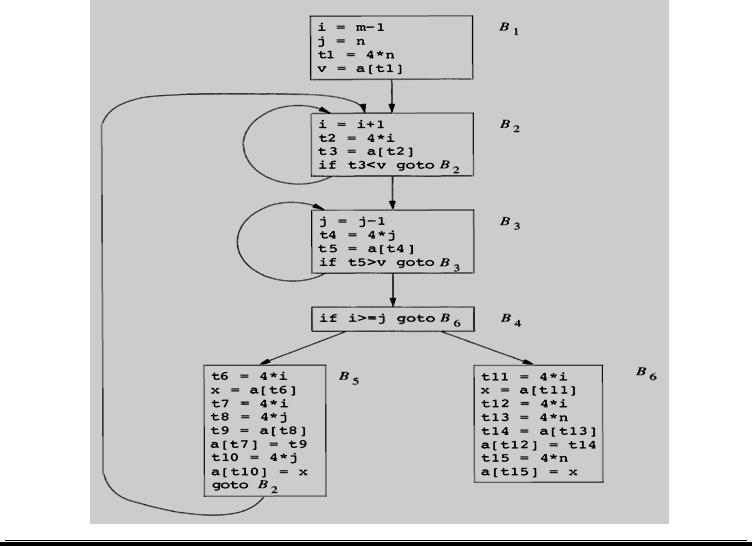


P a g e | **14** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**Three-address code:**

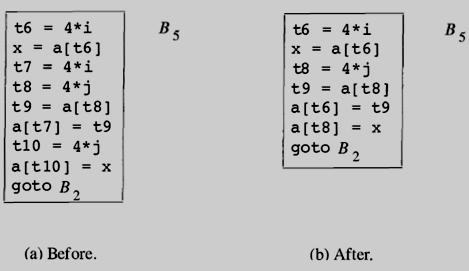


**Flow graph:**



P a g e | **15** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

1. **Common Sub-expression Elimination:** 
   * An occurrence of an expression E is called a ***common sub expression***, if E was previously computed, and the values of variables in E have not changed since the previous computation.
   * DAG representations of basic blocks show how local common sub-expressions could be removed as the DAG for basic block.
   * A program can be calculated the same value such as an offset in an array.
   * Ex: recalculates ***4\*i*** and ***4\*j.***
   * For example, the assignments to ***t7*** and ***t10*** have the common sub-expressions ***4\*i*** and ***4\*j,*** respectively. They have been eliminated by using ***t6*** instead of ***t7*** and ***t8*** instead of ***t10***.



**Local common sub-expression elimination**

**Elimination of global and local common Sub-expression:**

* The result of eliminating both global and local common sub-expressions from blocks B5 and B6 in the flow graph.
* After local common sub-expressions are eliminated B5 still evaluates 4\*i and 4\*j.
* Both are common sub-expressions; in particular, the three statements

**t8:= 4\*j; t9:= a [t8]; a [t8]:= x** in B5 can be replaced by

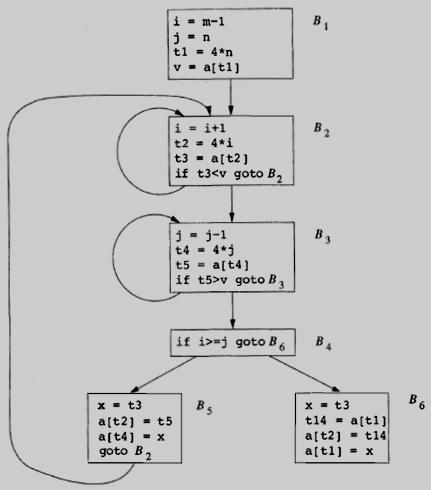
**t9:= a [t4];** **a [t4]:= x** using t4 computed in block B3.

* The control passes from the evaluation of 4\*j in B3 to B5, there is no change in j, so t4 can be used if 4\*j is needed.
* Another common sub-expression comes to light in B5 after ***t4*** ***replaces t8***.
* The new expression a[t4] corresponds to the value of a[j] at the source level.

P a g e | **16** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

The statement

**t9:= a [t4]; a [t6]:= t9** in B5 can therefore be replaced by

**a [t6]:= t5**

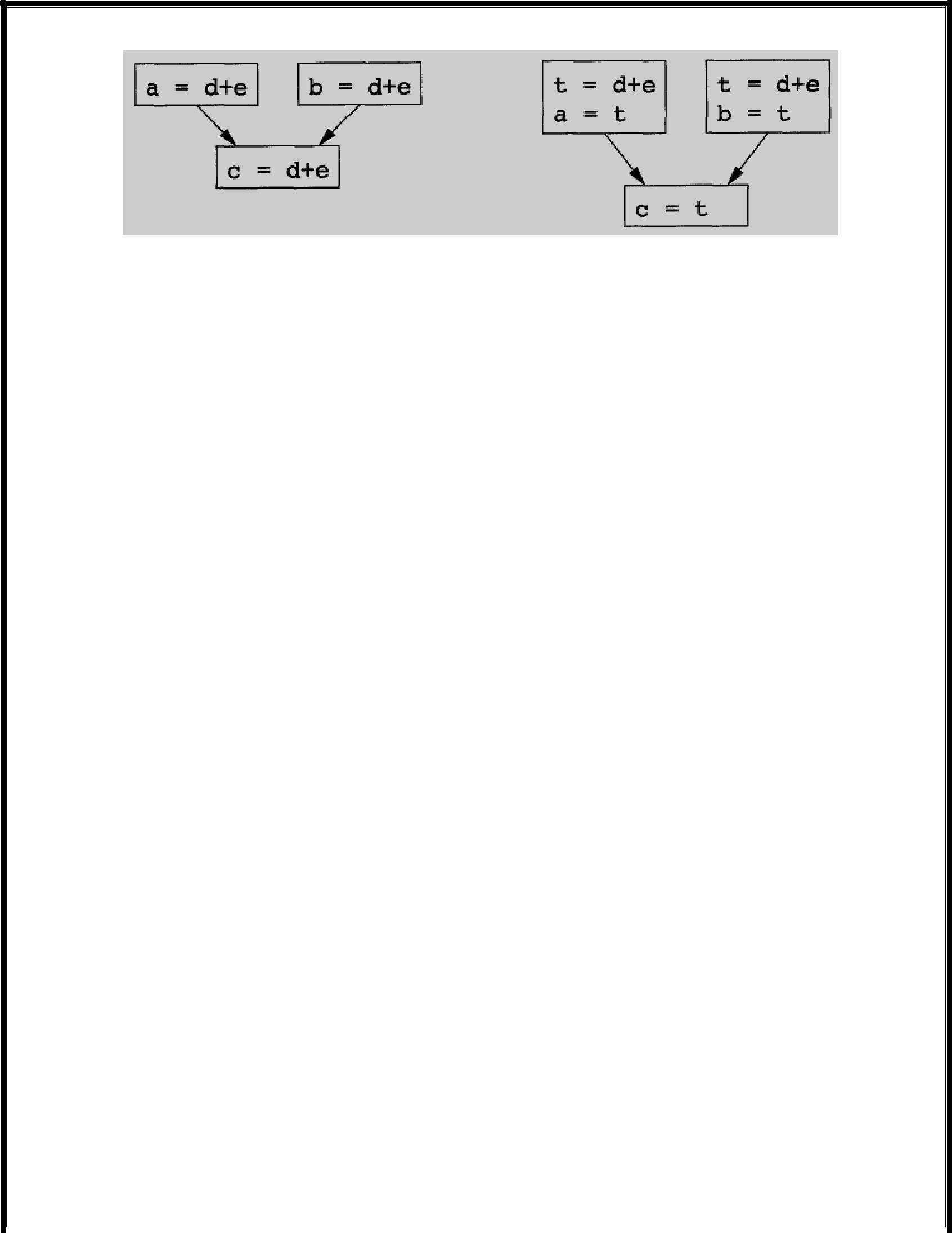
**Elimination of global and local common Subexpression**

1. **Copy Propagation:** 
   * Block B5 can be further improved by eliminating x using two new transformations.
   * The assignments of the form **f := g** called **copy statements**, or ***copies***
   * When the common sub-expression in ***c: = d+e*** is eliminated the algorithm uses a new variable ***t*** to hold the value of ***d+e.***
   * Since control may reach ***c: = d+e*** either after the assignment to a or after the assignment to b, it would be incorrect to replace ***c: = d+e*** by either ***c: = a*** or by ***c: =b.***
   * The idea behind the copy-propagation transformation is to use g for f, wherever possible after the copy statement ***f: =g.***
   * For example, the assignment ***x: = t3*** in block B5 is a copy. Copy propagation applied to B5 yields:

**x := t3**

**a [t2]:=t5 a [t4]:=t3 goto B2**

P a g e | **17** LANGUAGE TRANSLATORS DEPARTMENT OF CSE



**Copies introduces during common sub-expression elimination**

1. **Dead-Code Elimination:** 
   * A variable is live at a point in a program if its value is used subsequently;
   * Otherwise it is dead at that point.
   * Dead or useless code statements that compute values that never get used.
   * Dead Code are portion of the program which will not be executed in any path of the program can be removed.
   * For example, the use of debug that is set to true or false at various points in the program, and used in

statements like

***if (debug) print …***

* By a data-flow analysis, it may be possible to deduce that each time the program reaches this statement, the value of debug is ***false***.
* Usually, it is because there is one particular statement

***debug :=false***

* + If copy propagation replaces debug by false, then the print statement is dead because it cannot be reached. We can eliminate both the test and printing from the object code.

1. **Constant folding:** 
   * More generally, deducing at compile time that the value of an expression is a constant and using the constant instead is known as ***constant folding.***
   * One advantage of copy propagation is that it often turns the copy statement into dead code.
   * For example, copy propagation followed by dead-code elimination removes the assignment to **x** and transforms into

**a [t2 ] := t5 a [t4] := t3 goto B2**

P a g e | **18** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

1. **Explain Loop optimization techniques. (11 marks) (NOV 2011, 2013) (MAY 2012)** 
   * The optimizations, namely loops, especially the inner loops where programs tend to spend the bulk of their time.
   * The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop.

Three techniques are important for loop optimization

* 1. ***Code motion*** which moves code outside a loop;
  2. ***Induction-variable elimination*** which we apply to eliminate i and j from the inner loops B2 and B3.
  3. ***Reduction in strength*** which replaces and expensive operation by a cheaper one, such as amultiplication by an addition.

1. **Code Motion:**

* An important modification that decreases the amount of code in a loop is ***code motion***.
* This transformation takes an expression that yields the same result independent of the number of times a loop is executed ( *a loop-invariant computation*) and places the expression before the loop.
* The notion “before the loop” assumes the existence of an entry for the loop.

For example, evaluation of ***limit-2*** is a loop-invariant computation in the following while-statement: ***while (i<= limit-2 )*** /\* statement does not change limit \*/

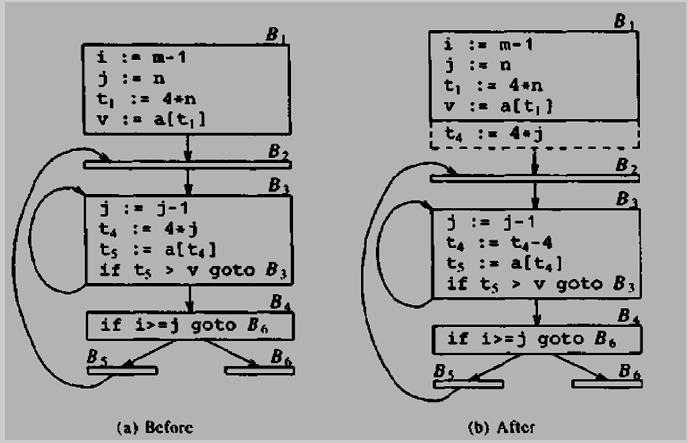
Code motion will result in the equivalent of ***t= limit-2;***

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

P a g e | **19** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

1. **Induction Variables:** 
   * The values of j and t4 remain in lock-step; every time the value of j decreases by 1, that of t4 decreases by 4 because 4\*j is assigned to t4.Such identifiers are called ***induction variables***.
   * When there are two or more induction variables in a loop, by the process of induction-variable elimination.
   * For the inner loop around B3, t4 is used in B3 and j in B4.

**Strength reduction applied to 4\*j in block B3**



* In block B3, t4 := 4 \* j holds assignment to t4 and t4 is not changed elsewhere in the inner loop around B3.
* It follows that the statement j := j - 1 the relationship t4 := 4 \*j - 4 must hold.

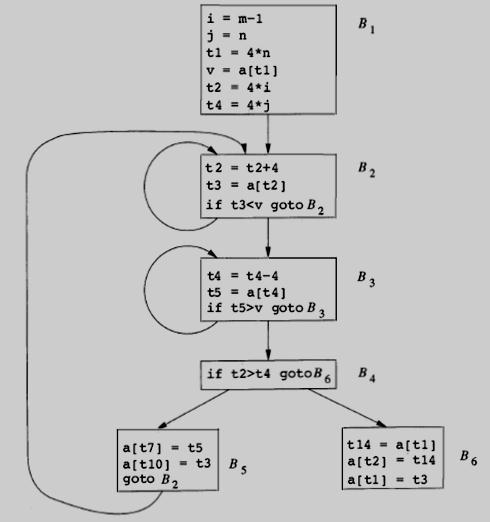
 The replace the assignment **t4** **:= 4 \* j** by **t4** **:= t4** **- 4.**

* The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction.

P a g e | **20** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

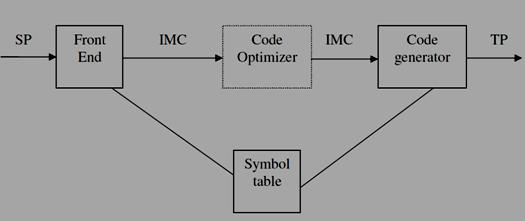
1. **Reduction in Strength:** 
   * **Reduction in strength,** which replaces and expensive operation by a cheaper one, such as amultiplication by an addition.
   * After reduction in strength is applied to the inner loop around B2 and B3, the only use in i and j to determine the outcome of test in block B4.
   * The value of **i** and **t2** **= 4 \* i** and **j** for **t4** **= 4 \* j** , so the test **t2** **>= t4** is equivalent to i >= j.
   * Replacement i in block B2 and j in Block B3 become dead variables and assignment to the blocks, a dead code that can be eliminated.

**Flow graph for induction variable elimination:**



P a g e | **21** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

1. **Explain the issues in design of a code generator? (11 marks) (NOV 2013)** 
   * 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.



**Issues in the design of a code generator:**

While the details are dependent on the target language and the operating system, issues such as

1. Input to the code generator
2. Target Programs
3. memory management
4. instruction selection
5. register allocation and
6. evaluation order

**(i)Input to the 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.

* The intermediate language, including:
  + ***linear*** representations such as ***postfix notation,***
  + ***three address*** representations such as ***quadruples,***
  + ***virtual*** representations such as ***stack machine code.***
  + ***graphical*** representations such as ***syntax trees and dags.***
* The code generation the front end has scanned, parsed, and translated the source program into a intermediate representation, 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.).

P a g e | **22** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

1. **Target Programs:** 
   * The output of the code generator is the target program.
   * The intermediate code, this output may take on a variety of forms:
     1. absolute machine language,
     2. relocatable machine language, or
     3. assembly language

**Absolute machine language**

* Producing an absolute machine language program as output that it can be placed in a location in memory and immediately executed.
* A small program can be compiled and executed quickly.
* A number of “student-job” compilers, such as WATFIV and PL/C, produce absolute code.

**Relocatable machine language**

* 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.

**Assembly language**

* + 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

1. **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 the code generator.
   * We assume that a name in a three-address statement refers to a symbol table entry for the name.
   * Symbol-table entries are created as the declarations in a procedure.
   * The type of declaration determines the width, the amount of storage, needed for the declared name.
   * The symbol-table information, needed for the determined for the name in a data area for the procedure.
   * The static and stack allocation of data areas , and show how names in the intermediate representation can be converted into addresses in the target code.

P a g e | **23** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

1. **Instruction Selection:** 
   * The instruction set of the target machine determines the difficulty of instruction selection.
   * The instructions set are important factors
     1. ***uniformity***
     2. ***completeness***
     3. ***Instruction speeds***
     4. ***and machine idioms***
   * If the target machine does not support each data type in a uniform manner, then each exception to the general rule requires special handling.
   * The efficiency of the target program, instruction selection is straightforward.
   * Three- address statement can design a code skeleton that the target code to be generated.
   * For example, three address statement of the form **x := y + z,** where x, y, and z are statically allocated, can be translated into the code sequence

**MOV y, R0** /\* load y into register R0 \*/

**ADD z, R0** /\* add z to R0 \*/

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

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

**a := b + c d := a + e**

would be translated into

**MOV b, R0**

**ADD c, R0**

**MOV R0, a**

**MOV a, R0**

**ADD e, R0**

**MOV R0, d**

* Here the fourth statement is redundant, and so is the third if ‘a’ is not subsequently used.

P a g e | **24** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

* The quality of the generated code is determined by its speed and size.
* Instruction speeds are needed to design good code sequence but unfortunately, accurate timing information is often difficult to obtain.

For example if the target machine has an ***“*increment” instruction (INC),** then the three address statement **a := a+1** may be implemented more efficiently by the single instruction ***INC a***, sequence that loads a into a register, add one to the register, and then stores the result back into a.

**MOV a, R0**

**ADD #1, R0**

**MOV R0, a**

1. **Register Allocation:** 
   * Instructions involving register operands are usually shorter and faster than those involving operands in memory.
   * Efficient utilization of register is particularly important in generating good code.
   * The use of registers is often subdivided into two sub-problems:
     + 1. During **register allocation,** we select the set of variables that will reside in registers at a point in the program.
       2. During a **register assignment** phase, we 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 or the operating system of the target machine may require that certain register usage.
   * Certain machines require **register pairs** (an even and next odd numbered register) for some operands and results.
   * For example, in the IBM System/370 machines integer multiplication and integer division involve register pairs.

The ***multiplication instruction*** is of the form

* 1. **x, y**
* where x, is the multiplicand, is the even register of an even/odd register pair.
* The multiplicand value is taken from the odd register pair. The multiplier y is a single register. The product occupies the entire even/odd register pair.

P a g e | **25** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

The ***division instruction*** is of the form

* 1. **x, y**
* where the 64-bit dividend occupies an even/odd register pair whose even register is x; y represents the divisor.
* After division, the even register holds the remainder and the odd register the quotient.

Consider the two three address code sequences (a) and (b) in which the only difference is the operator in the second statement. The shortest assembly sequence for (a) and (b) are given in(c).

* **Ri** stands for **register i**.
* **L**, **ST** and **A** stand for **load, store and add** respectively.

The optimal choice for the register into which ‘a’ is to be loaded depends on what will ultimately happen to e.

|  |  |  |  |
| --- | --- | --- | --- |
| **t := a + b** | | **t := a + b** | |
| **t := t \* c** | | **t := t + c** | |
| **t := t / d** | | **t := t / d** | |
| **(a)** | | **(b)** |  |
|  | **Two three address code sequences** | |  |
| **L** | **R1, a** | **L** | **R0, a** |
| **A** | **R1, b** | **A** | **R0, b** |
| **M** | **R0, c** | **A** | **R0, c** |
| **D** | **R0, d** | **SRDA** | **R0, 32** |
| **ST** | **R1, t** | **D** | **R0, d** |
|  |  | **ST** | **R1, t** |
| **(a)** | | **(b)** |  |

**Optimal machine code sequence**

1. **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, to 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.

P a g e | **26** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

1. **Explain the Target machine in code generator? (6 marks)** 
   * Familiarity with the target machine and its instruction set is for designing a good code generator.
   * Our target computer is a byte addressable machine with four bytes to a word and n general purpose registers, R0,R1,…., Rn-1.

The has two address instruction is of the form **op source, destination**

in which **op** is an **op-code** and **source** and **destination** are **data fields.**

The op-codes are

***MOV (move source to destination)***

***ADD (add source to destination)***

***SUB (subtract source from destination)***

* The source and destination fields are not long enough to hold the memory addresses, so certain bit patterns in these fields specify that words following an instruction contain operands or addresses.
* The source and destination of an instruction are specified by combining register and memory locations with addressing modes.
* The description, **contents (a)** denotes the contents of the register or memory address represented by a.

**Addressing modes** together with their assembly-language forms and associated costs are as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| **MODE** | **FORM** | **ADDRESS** | **ADDED COST** |
|  |  |  |  |
| **Absolute** | **M** | M | **1** |
|  |  |  |  |
| **Register** | **R** | R | **0** |
|  |  |  |  |
| **Indexed** | **c(R)** | c + contents(R) | **1** |
|  |  |  |  |
| **Indirect indexed** | **\*R** | contents(R) | **0** |
|  |  |  |  |
| **Indirect indexed** | **\*c(R)** | contents(c + contents(R)) | **1** |
|  |  |  |  |
| **Literal** | **#c** | c | **1** |
|  |  |  |  |

P a g e | **27** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

The following instructions are

1. A memory location M or a register R represents itself when used as a source or destination. For example, the instruction

**MOV R0, M**

stores the contents of register R0 into memory location M.

**2.** An address offset c from the value in register R is written as c(R).Thus,

**MOV 4(R0), M**

stores the value **contents (4 + contents (R0))** into memory location M. **3.** Indirect versions of the last two modes are indicated by prefix \*. Thus,

**1. MOV \*4(R0), M**

stores the value **contents(contents(4 + contents(R0))** into memory location M. **4.** A final address mode allows the source to be a constant:

**1. MOV #1, R0** loads the constant 1 into register R0.

**Instruction Costs:**

* + The cost of an instruction is one plus cost associated with the source and destination modes.
  + The cost corresponds to the length of the instruction.
  + Address mode involving registers have ***cost zero***, while those with a memory location or literal in them have cost one, because such operands have to be stored with the instruction.
  + The most instructions, the time taken to fetch an instruction from memory exceeds the time spent executing the instruction.
  + By minimizing the instruction length is to minimize the time taken to perform the instruction.

1. The instruction **MOV R0, R1** copies the contents of register R0 into register R1.***This instruction has*** ***cost one.***
2. The instruction **MOV R5, M** copies the contents of register R5 into memory location M. ***This*** ***instruction has cost two.***
3. The instruction **ADD #1, R3** adds the constant 1 to the contents of register 3, and ***cost has two.***
4. The instruction **SUB 4(R0), \*12(r1)** stores the value

**contents (contents (12 + (contents (R1)) – contents (contents (4+R0))** into the destination \***12(r1).** ***The cost of the instruction is three.***

P a g e | **28** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

|  |  |  |  |
| --- | --- | --- | --- |
|  | MOV R0, R1 |  | **cost = 1** |
|  | MOV \*R0,\*R1 |  | **cost = 1** |
|  | MOV R0, a |  | **cost = 2** |
|  | MOV a, R0 |  | **cost = 2** |
|  | MOV #1, R0 |  | **cost = 2** |
|  | MOV a, b |  | **cost = 3** |

The three- address statement of the form a**: = b + c,** where b and c are simple variables in distinct memory location denoted by these names.

|  |  |  |
| --- | --- | --- |
| **1.** | MOV b, R0 |  |
|  | ADD c, R0 | **cost = 6** |
|  | MOV R0, a |  |
| **2.** | MOV b, a | **cost = 6** |
|  | ADD c, a |  |

Assume R0 = a, R1 = b, R2 = c, respectively

|  |  |  |
| --- | --- | --- |
| **3.** | MOV \*R1,\*R0 | **cost = 2** |
|  | ADD \*R2,\*R0 |  |
| Assume R1 = b, R2 = c | |  |
| **4.** | ADD R1, R2 | **cost = 3** |
|  | MOV R1, a |  |

P a g e | **29** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

1. **Explain the run-time storage management? (11 marks)** 
   * The semantic of procedures in a language determines how names are bound to storage during execution.
   * Information needed during an execution of a procedure is kept in a block of storage called ***activation*** ***record;*** storage for names local to the procedure also appears in the activation record.

The two standard allocation strategies are

* 1. ***Static allocation***
  2. ***Stack allocation***
* In ***static allocation,*** the position of an activation record in memory is fixed at compile time.
* In ***stack allocation,*** a new activation record is pushed onto the stack for each execution of a procedure. The record is popped when the activation ends.

The activation record for a procedure has fields as

* to hold parameters,
* results
* machine status information,
* local data,
* temporary variables.

The run time allocation and de-allocation of activation record occurs as part of the

* procedure calls and
* return sequences

The three- address statement as

1. call
2. return
3. halt and
4. action

P a g e | **30** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

1. **Static Allocation:** 
   * Consider the code needed to implement static allocation.
   * A **call** statement in the intermediate code is implemented by the sequence of two target-machine instructions.
   * A ***MOV*** instruction to save the return address and a ***GOTO*** transfers control to the target code for the

called procedure:

**MOV #here+20, callee.static\_area GOTO callee.code\_area**

Where

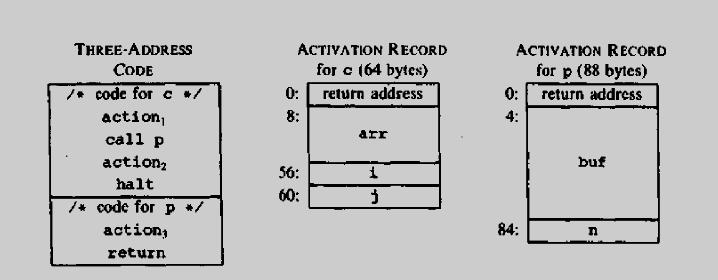
* The ***callee.static\_area*** and ***callee.code\_area*** are constants referring to the address of the activation record and the first instruction for the called procedure respectively.
* The source **#here+20** in the **MOV** instruction is the literal return address.
* The code for a procedure ends with a return to the calling procedure, except that the first procedure has no caller, so its final instruction is HALT.

Return from procedure ***callee*** is implemented by

**GOTO \*callee.code\_area**

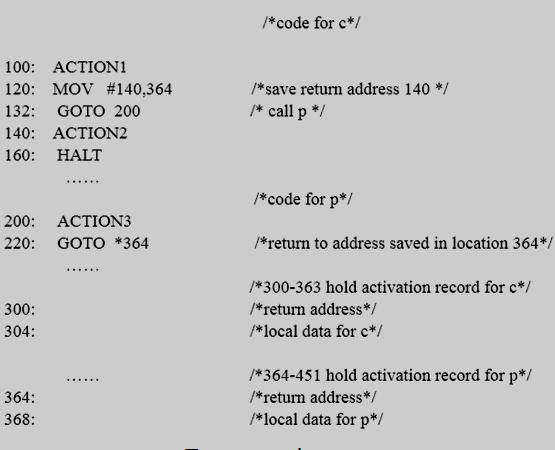
* which transfers control to the address saved at the beginning of the activation record.

**Input to code generator:**



P a g e | **31** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**Target code for the input:**



1. **Stack Allocation:** 
   * Static allocation can become stack allocation by using relative addresses for storage in activation records.
   * The position of the record for an activation of a procedure is not known until run time.
   * In ***stack allocation,*** this position is usually stored in a register, so words in the activation record can be accessed as offsets from the value in this register.
   * Relative addresses in an activation record can be taken as offsets from any known position in the activation record.
   * A register SP a pointer to the beginning of the activation record on the top of the stack.
   * When a procedure call occurs, the calling procedure increments SP and transfers control to the called procedure.
   * After control returns to the caller, it decrements SP, thereby de-allocating the activation record of the called procedure.

P a g e | **32** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

The code for the first procedure initializes the stack by setting SP to the start of the stack area in memory:

**MOV #*stackstart*, SP** /\* initialize the stack \*/

code for the first procedure

**HALT** /\* terminate execution \*/

A procedure call sequence increments SP, saves the return address, and transfers control to the called procedure:

**ADD #*caller.recordsize*, SP**

**MOV #*here+*16, SP** /\* save return address \*/

**GOTO *callee.code\_area***

* The attribute ***caller.recordsize*** represents the size of an activation record, so the ADD instruction leaves SP pointing to the beginning of the next activation record.
* The source ***#here+16*** in the ***MOV instruction*** is the address of the instruction following the GOTO; it is saved in the address pointed to by SP.

The return sequence consists of two parts. The called procedure transfers control to the return address using

**GOTO \*0(SP)** /\*return to caller\*/

The reason for using \*0(SP) in the GOTO instruction is that we need two levels of indirection:

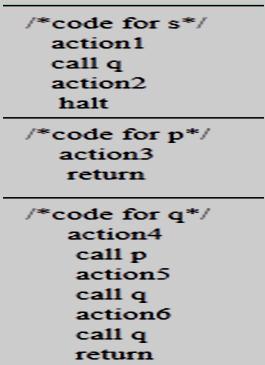
* + 0(SP) is the address of the first word in the activation record and
  + \*0(SP) is the return address saved there.
* The second part of the return sequence is in the caller, which decrements SP, thereby restoring SP to its previous value.
* That is, after the subtraction SP points to the beginning of the activation record of the caller:

**SUB #*caller.recordsize*, SP**

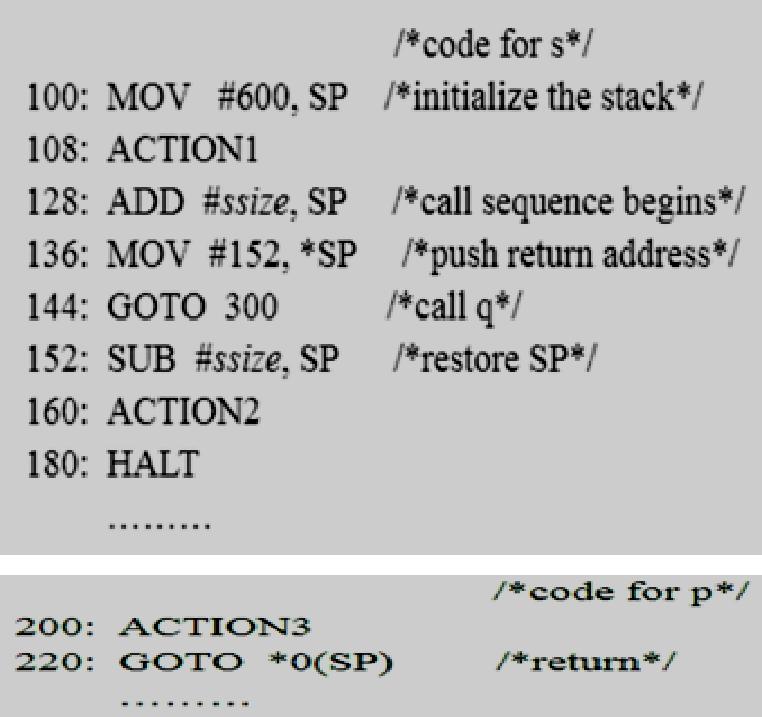
P a g e | **33** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**Target code for stack allocation:**

**Three-address code**



**Three address code for stack allocation:**



P a g e | **34** LANGUAGE TRANSLATORS DEPARTMENT OF CSE



**Run time addresses for names:**

* The storage allocation strategy and the layout of local data in an activation record for a procedure determine how the storage for names is accessed.
* Assume that a name in a three-address statement is really a pointer to a symbol-table entry for the name; it makes the compiler more portable, since the front end need not be changed even if the compiler is moved to a different machine where a different run-time organization is needed.
* Names must be replaced by code to access storage locations. Consider the simple three-address statement x: = 0.
* After the declarations in a procedure are processed, suppose the symbol-table entry for x contains a relative address 12 for x.
* First consider the case in which x is in a statically allocated area beginning at address *static.* Then the actual run-time address for x is *static*+12.
* The assignment x := 0 then translates into

**static [12] := 0**

* If the static area starts at address 100, the target code for this statement is

**MOV #0, 112**

* Suppose x is local to an active procedure whose display pointer is in register R3. Then we may translate the copy x := 0 into the three-address statements

**t1 := 12+R3**

**\*t1 := 0**

in which t1 contains the address of x. This sequence can be implemented by the single machine instruction

**MOV #0, 12 (R3)**

The value in R3 cannot be determined at compile time.

P a g e | **35** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

1. **What is Next use information? Discuss (5 marks) (MAY 2013)** 
   * Next use information about names in basic block.
   * If the name in a register is no longer needed, then the register can be assigned for the some other name.
   * The keeping a name in storage only it will be used subsequently can be applied in a number of contexts.

**Computing Next Uses:**

* The use of a name in a three-address statement as follows:
* Suppose three-address statements i assigns a value to x.
* If statement j has x as an operand and control can flow from statement i to j along a path that has no assignments to x, then the statement j uses the value of x computed at i.
* The three address statement ***X = Y op Z*** the next uses of X, Y and Z.
* The algorithm to determine next uses makes a backward pass over each basic block.
* Assume all temporaries are dead on exit and all user variables are live on exit.

**Algorithm to compute next use information:**

Suppose three address statement **i : X := Y op Z** in backward scan, the following

1. Attach to statement i, information currently found in the symbol table regarding the next use and live-ness of X, Y and Z.
2. In symbol table, set X to “not live” and “no next use “.
3. In symbol table, set Y and Z to be “live” and next use of Y and Z to i.

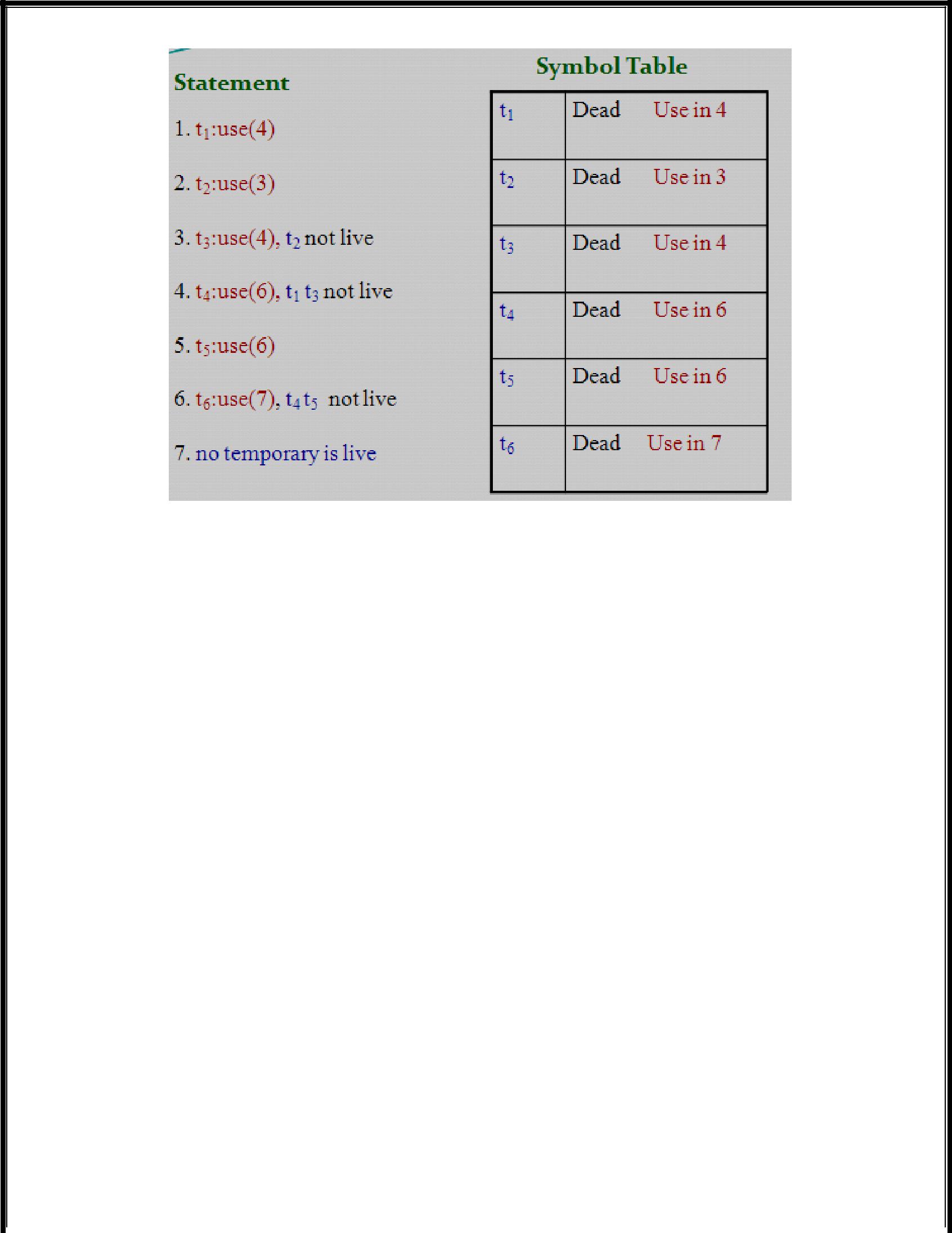
**Storage for Temporary Names:**

* The two temporaries into the same location if they are not live simultaneously.
* All temporaries are defined and used within basic blocks; next-use information can be applied to pack temporaries.
* In the basic block can be packed into two locations. These locations correspond to t1 and t2:

**Example: X = a \* a + 2(a \* b) + (b \* b)**

1. t1 = a \* a
2. t2 = a \* b
3. t3 = 2 \* t2
4. t4 = t1 + t3
5. t5 = b \* b
6. t6 = t4 + t5
7. X = t6

P a g e | **36** LANGUAGE TRANSLATORS DEPARTMENT OF CSE



**9. Explain the four issues in the design of a simple code generator. Generate the code for a simple statement. (NOV 2013) (11 marks)**

* The code-generation generates target code for a sequence of three address statements.
* Each three-address statement operands are currently stored in register.
* Assume that computed results can be left in registers as long as possible, storing them only
  1. If their register is needed for another computation
  2. Just before a procedure call, Jump or labeled statement.

**Register and Address Descriptors:**

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

1. Register descriptors
2. Address descriptors

**Register Descriptors:**

* A register descriptor keeps track of what is currently in each register.
* Initially all the registers are empty.
* We assume that initially the register descriptor shows that all register are empty.
* The code generator for the block progresses, each register will hold the value of zero or more names at any given time.

P a g e | **37** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**Address Descriptors:**

* Address descriptors keeps track of location where current value of the name can be found at runtime.
* The location might be a register, a stack location or a memory address.
* This information can be stored in the symbol table and is used to determine the accessing method for a name.

**Code- Generation Algorithm:**

The code generation algorithm takes as input a sequence of three address statement constituting a basic block.

The three address statement of the form **X = Y op Z**

**STEP 1:**

* Invoke a function **getreg ()** to determine the location L, where the result of computation Y op Z should be stored.
* L will usually be a register or memory location.

**STEP 2:**

* The address descriptor for Y to determine Y’, the current location of Y.
* Prefer the register for Y’, if the value Y is currently both in register and memory location.
* Generate the instruction MOV Y’ in L.

**STEP 3:**

* Generate the instruction op Z’, L where Z’ is a current location of Z.
* The value Z is currently both in register and memory location.
* Update address descriptor of X to indicate that X is in location L.

**STEP 4:**

* If the current values of Y and Z have no next use, are not live on exit from the block.
* Register descriptor to indicate after execution of X=Y op Z.

Consider the assignment statement of the form **d: = (a - b) + (a - c) + (a - c**) might be translated into three address code sequence

**t := a - b u := a - c v := t + u d := v + u**

P a g e | **38** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

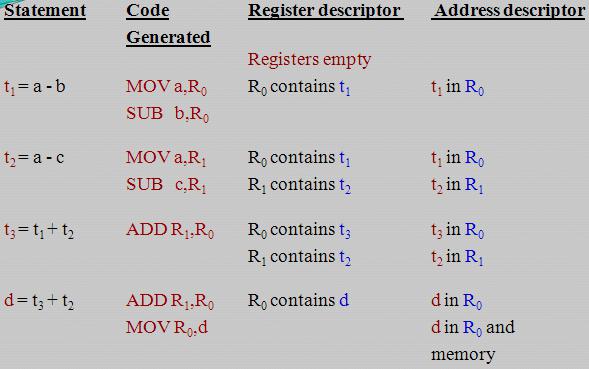
**The *getreg ()* function:**

The function getreg returns the location L to hold the value of x for the assignment **x: =y** ***op*** **z.**

The algorithm for getreg:

1. If the name y is in a register, that holds the value of no other names and y is not live and has no next use after the execution of y = x *op* z, then a. return L. Update the address descriptor of y , so that y is no longer in L.
2. Failing (1), return an empty register for L if there is one.
3. Failing (2), if x has a next use in the block, or if *op* requires a register then a. find an occupied register R. MOV(R,M) if value of R is not in proper M. If R holds value of many variables, generate a MOV for each of the variables.
4. Failing (3), select the memory location of x as L.

**Code Sequence:**



P a g e | **39** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

S

**Conditional Statements:**

* Machines implement conditional jumps in one of two ways.
* One way is to branch if the value of a designated register have six conditions: negative, zero, positive, non-negative, non-zero and non-positive.
* The machine three address statement such as **if X < Y goto Z** can be implemented by
  1. Subtracting Y from X in register R and
  2. Then jumping to Z, if the value in register R is negative.
* A second approach, common to many machines, uses a set of ***condition code*** to indicate whether last quantity computed or loaded into a location is negative, zero, or positive.
* ***Compare instruction (CMP)*** sets the codes without actually computing the value.
* ***CMP X, Y*** sets condition codes to positive if X > Y and so on.
* A conditional-jump machine instruction makes the jump if a designated condition <, +,>, <, ≤, ≠ or ≥.
* The instruction ***CJ<=Z*** to mean “jump to Z if the condition code is negative or zero”.

For example, **if X < Y goto Z** could be implemented by

**Cmp X, Y**

**CJ < Z**

The condition code descriptor

**X := Y + Z**

**if X < 0 goto L**

by

**MOV Y, R0**

**ADD Z, R0**

**MOV R0, X CJ < L**

The condition code is determined by x after ADD Z, R0...

P a g e | **40** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

1. **Explain the DAG representation of basic block? (11 marks) (MAY 2013)** 
   * ***Directed acyclic graphs (DAG)*** are useful data structure for implementing transformations on basicblocks.
   * A ***dag*** gives a picture of how the value computed by each statement in a basic block is used in subsequent statements of the block.

Constructing a dag from three-address statements is a good way of

* Determining the common sub-expressions (expressions computed more than once) within a block.
* Determining which names are used inside the block but evaluated outside the block and
* Determining which statements of the block could have their computed value outside the block.

A ***dag for a basic block*** is a directed acyclic graph has following labels on the nodes:

1. Leaves are labeled by unique identifiers, either variable names or constants. From the operator applied to name we determine whether the L-value or R-value name is created; most leaves represent R-values. The leaves represent initial values of names and we subscript them with 0 to avoid confusion.
2. Interior nodes are labeled by an operator symbol.
3. Nodes are also optionally given a sequence of identifiers for labels. The interior nodes represent computed values and the identifiers labeling a node.

Each node of a flow graph can be represented by a dag, since each node of the flow graph stands for a basic block.

**The source program**

**begin**

prod := 0; i :=1;

**do begin**

prod := prod + a[ i ] \* b[ i ]; i := i + 1;

**end**

**while** i <= 20

**end**

P a g e | **41** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

Three address code as

1. t1 := 4 \* i
2. t2 := a[t1]
3. t3 := 4 \* i
4. t4 := b[t3]
5. t5 := t2 \* t4
6. t6 := prod + t5
7. prod := t6
8. t7 := i + 1
9. i := t7
10. if i <= 20 goto (1)

**DAG Representation:**



**Dag Construction:**

**Input:** a basic block.

**Output**: a dag for the basic block containing 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).

P a g e | **42** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**Method:** Initially assume there are no nodes, andthree address statements in either **(i) x:= y op z** goto case (i), with X undefined.

***node*** is undefined for all arguments. Suppose the current**(ii) x:= op y (iii) x:= y.** A relational operator like if i<=20

1. If ***node(y)*** is undefined, created a leaf labeled y, let ***node(y)*** be this node. In case (i) if ***node (z)*** is undefined, create a leaf labeled z and that leaf be ***node (z).***
2. In case (i) determine if there is a node labeled ***op,*** whose left child is ***node(y)*** and right child is ***node(z).*** If not create such a node, let be n. case (ii), (iii) similar.
3. Delete x from the list attached identifiers for ***node(x).*** Append x to the list of identify for node n and set ***node(x)*** to n.

**Application of Dags:**

The applications of DAGs are

* + To automatically detect a common sub expressions.
  + To determine which identifiers have their values used in the block.
  + To determine which statements compute values that could be used outside the block.

1. **Explain the peephole optimization. (11 marks) (NOV 2011, 2012)** 
   * The technique for locally improving the target code is ***peephole optimization***, a method for trying 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 whenever possible.
   * Peephole optimization as a technique for improving the quality of the target code, the technique can also be applied directly after intermediate code generation to improve the intermediate representation.
   * **Peephole** is a small, moving window on the target program.

The characteristics of peephole optimization are

* Redundant instruction elimination
* Unreachable Code
* Flow of control optimizations
* Algebraic simplification
* Reduction in Strength
* Use of machine idioms

P a g e | **43** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

1. **Redundant loads and stores:**

Consider the instruction sequence

* + - 1. **MOV R0, a**
      2. **MOV a, R0**
  + We can delete instruction (2) because whenever (2) is executed.
  + Instruction (1) the value of a is already in register R0.

1. **Unreachable code:** 
   * + Peephole optimization is the removal of unreachable instructions.
     + An unlabeled instructions immediately following unconditional jump may be removed.
     + This operation can be repeated to eliminate a sequence of instructions.

Consider following code segments that are executed only if a variable debug is 1. In C , the source code as

***# define debug 0***

………

***if (debug)*** {

**print debugging information**

}

In the intermediate representation the if statement may be translated as **if debug = 1 goto L1**

**goto L2**

**L1: print debugging information L2:**

Peephole optimization is to eliminate jump over jumps **if debug ≠ 1 goto L2**

**print debugging information**

**L2:**

Since debug is set to 0 at the beginning of the program, constant propagation should replace **if 0 ≠ 1 goto L2**

**print debugging information**

**L2:**

The argument of the first statement evaluates to a constant true, it can be replaced by goto L2. Then all the statements that print debugging are unreachable and can be eliminated one at a time.

P a g e | **44** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

1. **Flow of control optimizations:** 
   * The intermediate code generation algorithms are frequently produces
     + Jumps to jumps
     + Jumps to conditional jumps or
     + Conditional jumps to jumps
   * The unnecessary jumps can be eliminated either in intermediate code or target code in peephole optimization.

**Replace the jump sequence** goto L1

…..

L1: goto L2 by the sequence

goto L2

…..

L1: goto L2

If there are no jumps to L1, then it may be possible to eliminate the statement **L1: goto L2** provided it is preceded by an unconditional jump.

|  |  |  |  |
| --- | --- | --- | --- |
| **Similarly, the sequence** | |  |  |
|  | if a < b goto L1 |  | if a < b goto L2 |
|  | ……… | can be replaced by | ………. |
| L1 : | goto L2 | L1 : | goto L2 |

Finally, there is only one jump to L1 and L1 is preceded by an unconditional goto. Then the sequence goto L1

………

L1: if a < b goto L1 L3:

may be replaced by

if a < b goto L2 goto L3

…….

L3:

P a g e | **45** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

1. **Algebraic Simplification:** 
   * The amount of algebraic simplification that can be attempt through peephole optimization.
   * For example, statements such as
     1. **:= x + 0**

or

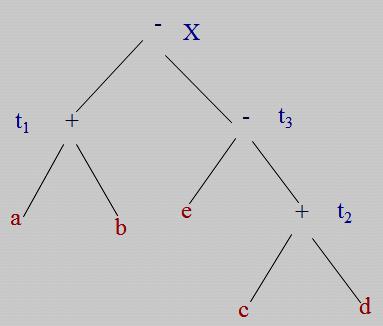
**x := x \* 1**

are produced by straightforward intermediate code generation algorithms and they can be eliminated easily through peephole optimization.

1. **Strength reduction:** 
   * Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine.
   * Certain machine instructions are cheaper than others and can be used as special cases of more expensive operators.
   * For example
     + Replace X^2 by X \* X
     + Fixed point multiplication or division by a power of two is cheaper to implement as a shift.
     + Fixed point division by a constant can be implemented as multiplication by a constant, which may be cheaper.
2. **Use of Machine idioms:** 
   * The target machines have hardware instructions to implement specific operations efficiently.
   * The use of instruction can reduce execution time significantly.
   * For example, machines have ***auto-increment and auto*** ***–decrement addressing modes.***
   * These add or subtract one from an operand before or after using its value.
   * The use of these modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing.
   * These modes can also be used in code for statements i: = i+1.

P a g e | **46** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

1. **Write note on Generating code from DAGs. (11 marks) (MAY 2012)** 
   * To generate code for a basic block from its DAG representation.
   * Dag shows how to rearrange the order of final computation sequence from linear representation of three - address statements or quadruples.
   * We can improve the program length or few no.of temporaries used.
   * This algorithm for optimal code generation from a tree is also useful when the intermediate code is a parse tree.
2. **Rearranging the Order:** 
   * Consider how the order in which computations are done can affect the cost of resulting object code.
   * Consider the following basic block whose dag representation

t1 := a + b t2 := c + d t3 := e – t2 X := t1 – t3

**Dag for basic block**

* The syntax directed translation of the expression X: = ( a + b ) - ( e - ( c + d ) ) by the algorithm.
* Generate code for the three address statement using the algorithm

MOV a, R0

ADD b, R0

MOV c, R1

ADD d, R1

MOV R0, t1

MOV e, R0

SUB R1, R0

MOV t1, R1

SUB R0, R1

MOV R1, X

P a g e | **47** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

We rearranged the order of the statements so that the computation of t1 occurs immediatedly before that of t4 as:

t2 := c + d t3 := e –t2 t1 := a + b X := t1 – t3

Using the code generation algorithm, the code sequence as

MOV c, R0

ADD d, R0

MOV e, R1

SUB R0, R1

MOV a, R0

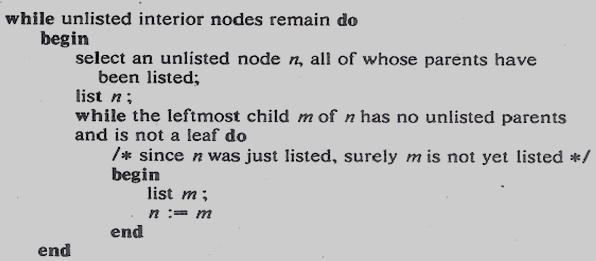
ADD b, R0

SUB R1, R0

MOV R1, X

1. **A Heuristic Ordering for Dags:** 
   * The heuristic ordering algorithm which attempts as far as possible to make the evaluation of a node immediately follows the evaluation of its leftmost argument.
   * The order of node can be edge relationship of the DAG.
   * The edges are procedure calls, array or pointer assignments.

**Node listing Algorithm:**

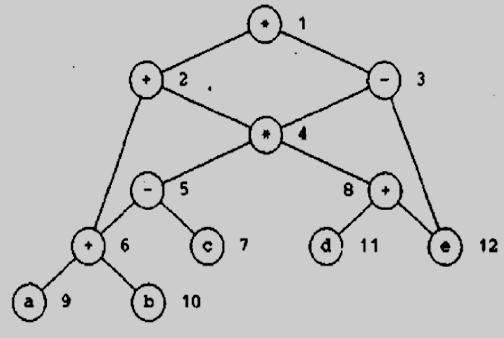


P a g e | **48** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

The ordering corresponds to the sequence of three-address statements:

t8 : = d +e t6 : = a + b t5 : = t6 - c t4 : = t5 \* t8 t3 : = t4 - e t2 : = t6 + t4 t1 : = t2 \* t3

**A DAG:**



**(iii) Optimal Ordering for Trees:**

* A simple algorithm to determine the optimal order in which to evaluate a sequence of quadruples is tree.
* Optimal ordering means the order that yields the shortest instruction sequence, over all instructions sequences that evaluate the tree.

The algorithm has two parts.

1. The first part labels each node of the tree, bottom-up, with an integer that denotes the fewest number of registers required to evaluate the tree with no stores of intermediate results.
2. The second part of the algorithm is a tree traversal whose order is governed by the computed node labels. The output code is generated during the tree traversal.

P a g e | **49** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**The Labeling Algorithm:**

* The term “Left leaf “to mean node that is a leaf and left descendent of the parent.
* All other leaves are referred to as “right leaves”.
* The labeling can be done by visiting nodes in a bottom-up order so that a node is not visited until all its children are labeled.
* The order in which parse tree nodes are created is suitable if the parse tree is used as intermediate code.
* In this case, the labels can be computed as a syntax-directed translation.

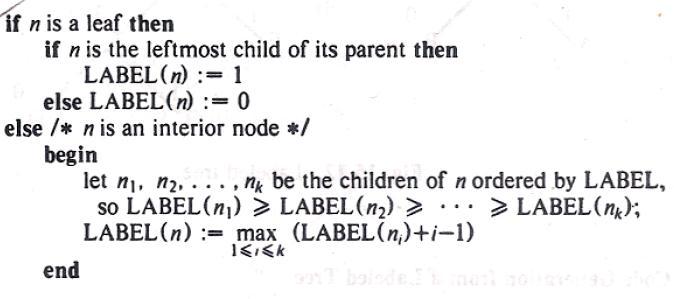
The important ***n*** is a ***binary node*** and its children have labels l1 and l2,



**label(n) = max(l1, l2)** **if l1 ≠l2**

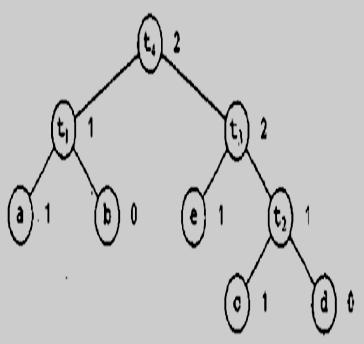
**l1+1** **if l1=l2**

**Label computations:**



 A post-order traversal of the nodes visits the nodes in the order **a b t1 e c d t2 t3 t4.**

* ***Node a*** is labeled 1 ***left leaf.***
* ***Node b*** is labeled 0 ***right leaf.***
* Node t1 is labeled 1 because the labels of its children are unequal and the maximum label of a child is 1.



P a g e | **50** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

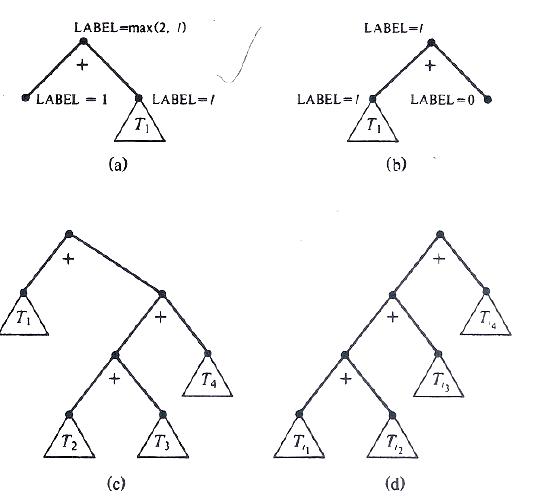
1. **Multi-register Operations:** 
   * The labeling algorithm to handle operations like multiplication, division or a function call, which normally require more than one register to perform.
   * The labeling algorithm ***label (n)*** is always at least the number of registers required by the operation.
   * If multiplication requires two registers, in the binary case use



|  |  |  |
| --- | --- | --- |
| **label (n) =** | **max(2, l1, l2)** | **if l1 ≠l2** |
|  | **l1+1** | **if l1=l2** |

* + where l1 and l2 are the labels of the children of n.

1. **Algebraic Properties:** 
   * The algebraic laws for various operators, the opportunity to replace a given tree T by one with smaller labels and fewer left leaves.



1. **Common Sub-expressions:** 
   * When there are common sub-expressions in a basic block, the corresponding dag will no longer be a tree.
   * The common sub-expressions will corresponds to nodes with more than one parent called **shared** **nodes.**

P a g e | **51** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**UNIVERSITY QUESTIONS**

* 1. **MARKS**

1. When static allocation can become stack allocation?**(NOV 2011) (Ref.Qn.No.30, Pg.no.6)**
2. Give the criteria for code-improving transformations. **(NOV 2011) (Ref.Qn.No.11, Pg.no.3)**
3. Define Flooding.**(MAY 2012) (Ref.Qn.No.41, Pg.no.8)**
4. What is mean by Reduction in Strength? **(MAY 2012) (Ref.Qn.No.18, Pg.no.4)**
5. Define DAG. **(NOV 2012) (NOV 2013) (Ref.Qn.No.35, Pg.no.7)**
6. What is translation of symbol? **(NOV 2012) (Ref.Qn.No.42, Pg.no.8)**
7. What is a basic block? What are the entry points and how do you call the entry instructions? **(MAY 2013)**

**(Ref.Qn.No.31, Pg.no.6)**

1. Define Induction variables? **(MAY 2013) (Ref.Qn.No.17, Pg.no.4)**
2. Construct a 3-address code for (B+A) \* (Y-(B+A)). **(NOV 2013) (Ref.Qn.No.40, Pg.no.8)**

**11 MARKS**

**NOV 2011(REGULAR)**

**1.** Describe the procedure for elimination of induction variables. **(Ref.Qn.No.4, Pg.no.19)**

**(OR)**

**2.** Explain the peephole optimization. **(Ref.Qn.No.11, Pg.no.43)**

**MAY 2012(ARREAR)**

**1.** Write note on Generating code from DAGs. **(Ref.Qn.No.12, Pg.no.47)**

**(OR)**

**2.** Explain Loop optimization techniques. **(Ref.Qn.No.4, Pg.no.19)**

**NOV 2012(REGULAR)**

**1.** Write note on peephole optimization. **(Ref.Qn.No.11, Pg.no.43)**

**(OR)**

**2.** Explain Local optimization techniques. **(Ref.Qn.No.3, Pg.no.14)**

P a g e | **52** LANGUAGE TRANSLATORS DEPARTMENT OF CSE

**MAY 2013(ARREAR)**

**1.** a) Explain Elimination of common sub expression during code optimization. Define for the expression

|  |  |  |
| --- | --- | --- |
| (a+b)-(a+b)/4 | (6) | **(Ref.Qn.No.3, Pg.no.14)** |
| b) What is Next use information? Discuss | (5) | **(Ref.Qn.No.8, Pg.no.36)** |
| **(OR)** |  |  |

**2.** Define Directed Acyclic Graph. How is it related to Basic blocks? Construct a DAG representation forthe following Basic block stating their steps. **(Ref.Qn.No.10, Pg.no.41)**

1. =B\*C
2. =A+B
3. =B\*C
4. =E-D.

**NOV 2013 (REGULAR)**

**1.** Explain briefly any three of the commonly used code optimization techniques. **(Ref.Qn.No.4, Pg.no.19)**

**(OR)**

**2.** Explain the four issues in the design of a simple code generator. Generate the code for a simple statement.

**(Ref.Qn.No.5, Pg.no.22)**

P a g e | **53** LANGUAGE TRANSLATORS DEPARTMENT OF CSE