| | | |
|---|---|---|
| (4) | Static allocation is simple, but not efficient memory management technique. | Heap allocation does memory management in efficient way. |
| (5) | Static allocation strategy is faster in accessing data as compared to heap allocation. | Heap allocation is slow in accessing as there is chance of creation of holes in reusing the free space. |
| (6) | It is easy to implement. | It is comparatively expensive to implement. |

## UNIT - IV

## GLOBAL OPTIMIZATION

**Q.22. Explain Global Optimization.**

**Ans. Global Optimization:**

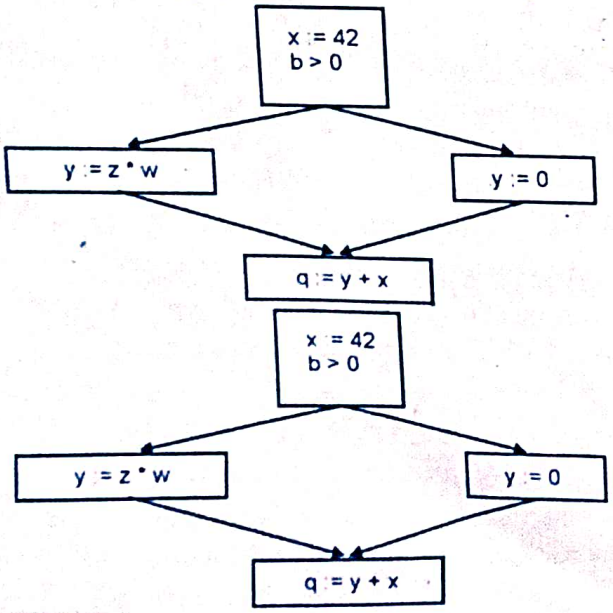
The scope of Global optimization is generally a program unit, usually spanning over number of basic blocks.

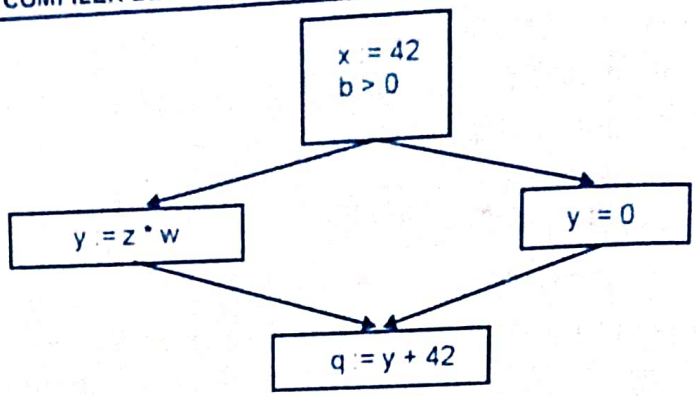
- It can achieved better transformation leading to more optimized.
- Program flow graph normally undergoes two types analysis:

(a) Control Flow Analysis.

(b) Data Flow Analysis.

- These optimizations can be extended to an entire control-flow graph.




## UNIT - V

## ARCHITECTURE DEPENDENT CODE IMPROVEMENT IN COMPILER DESIGN: INSTRUCTION SCHEDULING

**Q.23. Explain Architecture dependent code improvement in compiler design: Instruction Scheduling.**

**OR Explain Code Optimizing Process.**

**OR What is Code Optimization?**

**Ans. Code Optimization:**

- Code optimization is a program transformation approach that aims to enhance code by reducing resource consumption (i.e. CPU and memory) while maintaining high performance.
- In code optimization, high-level generic programming structures are substituted with low-level programming codes. The three guidelines for code optimization are as follows:

(i) In no way should the output code alter the program's meaning.

(ii) The program's speed should be increased, and it should use fewer resources if at all feasible.

(iii) The optimization step should be quick and not hinder the compilation process.

- At several stages of the compilation process, efforts to optimize the code might be made. Users can alter/rearrange the code at first or create the code using better algorithms. The compiler can improve loops and address computations after completing intermediate code. The compiler can leverage Memory Hierarchy and CPU registers while generating the target machine code.

**Q.24.Why Code Optimization is necessary?**

**Ans. Code Optimization:**

- Optimizing code in compiler design is important because it directly affects the performance of the compiled code.
- A well-optimized code runs faster and consumes fewer resources, leading to improved overall system performance and reduced energy consumption.
- Additionally, optimization can reduce the size of the generated code, which is important for embedded systems with limited memory.
- The optimization process can also help identify and eliminate bottlenecks in the code, leading to more efficient algorithms and improved software design. Overall, optimization is a critical step in the compiler design process that can greatly improve the end-user experience.

**Q.25.Explain types of Code Optimization.**

**Ans.** The code optimization process can be broadly classified into two types:

(i) Machine Independent Optimization

(ii) Machine Dependent Optimization

**(i) Machine Independent Optimization:**

This step of code optimization aims to optimize the intermediate code to produce a better target code. No CPU registers or absolute memory addresses are involved in the section of the intermediate code that is translated here.

**(ii) Machine Dependent Optimization:**

- After the target code has been created and converted to fit the target machine architecture, machine-dependent optimization is performed. It may use absolute memory references rather than relative memory accesses and requires CPU registers. Machine-dependent optimizers make a concerted attempt to maximize the memory hierarchy's benefits.
- A code has several statements, loops, branches, etc. So code optimization must be performed on all of them. The code optimization is done differently, considering the following:

**(1) Loop Optimization:**

The majority of programs in the system operate in a loop. It is vital to optimize the loops to save CPU cycles and memory. The following strategies can be used to improve loops.

**(i) Loop-invariant Code:**

It is a piece of code that sits in the loop and computes the same value each time an iteration is performed. This code may be moved out of the loop by storing it to be calculated just once rather than with each iteration.

**(ii) Induction Analysis:**

If a loop-invariant value changes the value of a variable within the loop, it is termed an induction variable.

**(iii) Strength Reduction:**

Some expressions use more CPU cycles, time, and memory than others. These expressions should be replaced with less expensive expressions without sacrificing the expression's output. For example, multiplication (x * 2) uses more CPU cycles than (x << 1) but produces the same output.

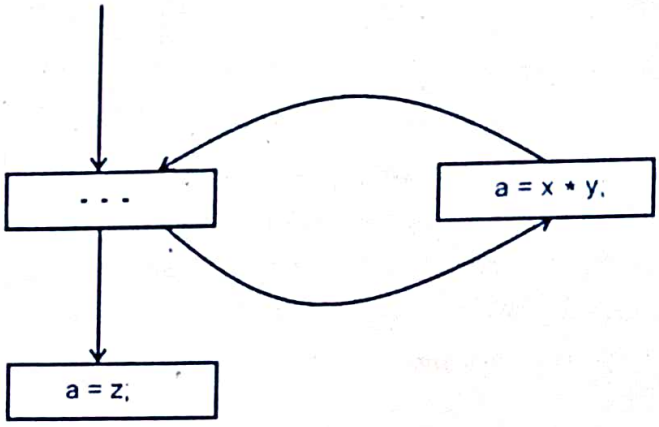



Fig.1. Loop Code flow chart

**(2) Partially Dead Code:**

- Some code statements include calculated values utilized only in particular conditions, i.e. the values are used sometimes and not others. Partially dead-code refers to such codes.
- The control flow diagram below shows a program section in which the variable 'a is utilized to assign the output of the equation 'x * y'. Let's

pretend that the 'a variable's value is never utilized within the loop. 'a' is given the variable 'z' value, which will be utilized later in the program, immediately after the control leaves the loop. We may infer that because the assignment code 'a' is never utilized anywhere, it is suitable for deletion.

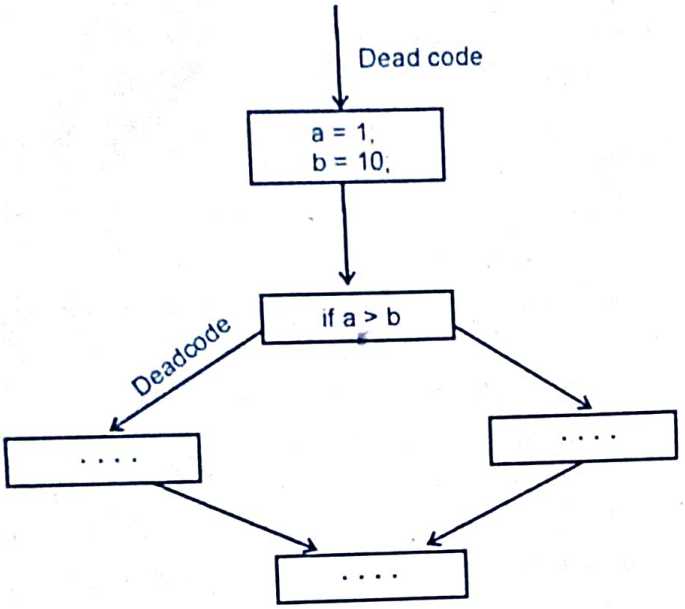



Fig.2. Dead Code flow chart

- Similarly, the conditional statement in the image above is always false, meaning that the code written in the "true" case will never be run and so may be eliminated.

(3) **Unreachable Code Elimination:**
A control flow graph should be created first. An inaccessible code block does not have an incoming edge. The inaccessible branches can be deleted after continual propagation and folding.

(4) **Function Inlining:**
The body of the function takes the place of a function call. This saves a lot of time by eliminating the need to copy all parameters, store the return address, and so on.

(5) **Function Cloning:**
For different calling arguments, specialized codes for a function are constructed. Overloading a function is an example of this.

(6) **Partial Redundancy:**
In a parallel route, redundant expressions are calculated many times without changing the operands. Partial-redundant expressions, on the other hand, are calculated several times along a path without changing the operands. By employing 'a code-motion approach, loop-invariant code may be rendered largely redundant.

An example of a partially redundant code can be:

```
If (condition) {
    a = y OP z;
} else {
    ...
}
c = y OP z;
```

We assume that the operands' values (y and z) do not change when variable an is assigned to variable c. If the condition statement is true, y OP z is calculated twice; otherwise, it is computed once. As stated below, code motion may be utilized to remove redundancy:

```
If (condition) {
    ...
    tmp = y OP z;
    a = tmp;
    ...
} else {
    ...
    tmp = y OP z;
}
c = tmp;
```

y OP z should only be computed once, regardless of whether the condition is true or not.

## TARGET CODE GENERATION

**Q.16.Explain Target Code Generation.**

**Ans. Target Code Generation:**

Target code generation is the final Phase of Compiler:

(i) **Input:** Optimized Intermediate Representation.

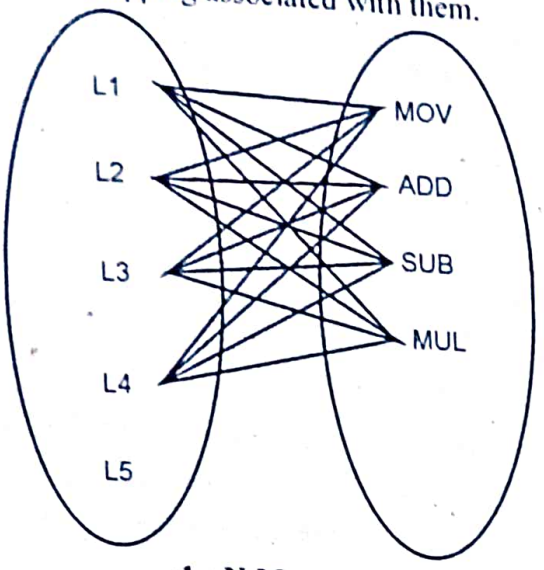
(ii) **Output:** Target Code.

(iii) **Task Performed:** Register allocation methods and optimization, assembly level code.

(iv) **Method:** Three popular strategies for register allocation and optimization.

(v) **Implementation:** Algorithms.

- Target code generation deals with assembly language to convert optimized code into machine understandable format. Target code can be machine readable code or assembly code. Each line in optimized code may map to one or more lines in machine (or) assembly code, hence there is a 1:N mapping associated with them.




**1 : N Mapping**

- Computations are generally assumed to be performed on high speed memory locations, known as registers. Performing various operations on registers is efficient as registers are faster than cache memory. This feature is effectively used by compilers, However registers are not available in large amount and they are costly. Therefore we should try to use minimum number of registers to incur overall low cost.

**Optimized Code:**

**Example 1:**

$L_1 : a = b + c * d$

**Optimization :**

$t_0 = c * d$

$a = b + t_0$

**Example 2:**

$L_2 : e = f - g / d$

**Optimization:**

$t_0 = g / d$

$e = f - t_0$

**Q.19. What are the various issues in design of code generator?**

**Ans. Various issues of Code Generator:**

**(1) Input to the Code Generator:**

- The source program's intermediate representation created by the front end, combined with data from the symbol table, are the inputs to the code generator.
- Graphical representations and three address representations are both included in IR.

**(2) The Target Program:**

- The complexity of building a decent code generator that creates high quality machine code is significantly influenced by the target machine's instruction set architecture.
- The most popular target machine architectures are stack based, CISC (Complex Instruction Set Computer) and RISC (Reduced Instruction Set Computer).

**(3) Instruction Selection:**

- The target machine must be able to execute the code sequence that the code generator creates from the IR program.
- If the IR is high level, the code generator may use code templates to convert each IR statement into a series of machine instructions.

**(4) Register Allocation:**

- A key problem in code generation is deciding what values to hold in which registers on the target machine do not have enough space to hold all values.
- Values that are not held in registers need to reside in memory. Instructions involving register operands are invariably shorter and faster than those involving operands in memory, so efficient utilization of registers is particularly important.
- The use of registers is often subdivided into two subproblems:

(i) Register allocation, during which we select the set of variables that will reside in registers at each point in the program.

(ii) Register assignment, during which we pick the specific register that a variable will reside in.

**(5) 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.

**Q.20. Explain various issues of code loop optimization.**

**Ans. Issues of Code Loop Optimization:**

**(1) Function Preserving Transformation:**

The function preserving transformations are basically divided into following types:

**(a) Common Sub-expression Elimination:**

- A common sub-expression is just an expression that has previously been computed and is utilized repeatedly throughout the programme.
- We stop computing the same expression repeatedly if the outcome of the expression is unchanged.

**For example:**

**Before Common Sub-expression Elimination:**

```
a = t * 4 – b + c;
.....
.....
m = t * 4 – b + c;
.....
.....
n = t * 4 – b + c'
```

**After Common Sub-expression Elimination:**

```
temp = t * 4 – b + c;
a = temp;
.....
.....

m = temp;
.....
.....
n = temp;
```

In given example, the equation a = t * 4 – b + c is occurred most of the times. So it is eliminated by storing the equation into temp variable.

**(b) Dead Code Elimination:**

- Dead code means the code which can be emitted from program and still there will be no change in result.
- A variable is live only when it is used in the program again and again. Otherwise, it is declared as dead, because we cannot use that variable in the program so it is useless.
- The dead code occurred during the program is not introduced intentionally by the programmer.

**For example:**

```
# Define False = 0
!False = 1
If(!False)
{
   .....
   .....
}
```

- If false becomes zero, is guaranteed then code in 'IF' statement will never be executed. So, there is no need to generate or write code for this statement because it is dead code.

**(c) Copy Propagation:**

- Copy propagation is the concept where we can copy the result of common sub-expression and use it in the program.
- In this technique the value of variable is replaced and computation of an expression is done at the compilation time.

**For example:**

$p_i = 3.14;$

$r = 5;$

Area = $p_i * r * r;$

Here at the compilation time the value of $p_i$ is replaced by 3.14 and r by 5.

**(d) Constant Folding (Compile Time Evaluation):**

- Constant folding is defined as replacement of the value of one constant in an expression by equivalent constant value at the compile time.
- In constant folding all operands in an operation are constant. Original evaluation can also be replaced by result which is also constant.

For example: a = 3.14157/2 can be replaced by a = 1.570785 thereby eliminating a division operation.

**(3) Algebraic Simplification:**

- Peephole optimization is an effective technique for algebraic simplification.
- The statements such as

  x : = x + 0

  or x : = x * 1

  can be eliminated by peephole optimization.

**Q.21. Generate the three address code for the following code fragment:**

```
while(a > b)
{
  if (c < d)
  x = y + z;
  else
  x = y – z;
}
```

**Ans.** Three address code for the given code:

(1) If (a > b) goto (3)
(2) goto
(3) If (c < d) goto
(4) goto (7)
(5) $t_1 = y + z$
(6) $x = t_1$
(7) $t_2 = y - z$
(8) $x = t_2$

## TYPE SYSTEMS

**Q.30. Explain Type Systems.**

**Ans. Type Systems:**

- A type system is a collection of rules for assigning type expressions to the various parts of a program. A type checker implements a type system. The type systems are specified in a syntax-directed manner.
- Different type systems may be used by different compilers or processors of the same language.
- For example, in Pascal, the type of an array includes the index set of the array, so a function with an array argument can only be applied to arrays with that index set.
- Many Pascal compilers, however, allow the index set to be left unspecified when an array is passed as an argument. Thus these compilers use a different type system than that in the Pascal language definition.
- Similarly, in the UNIX system, the lint command examines C programs for possible bugs using a more detailed type system than the C compiler itself uses.

**Static and Dynamic Checking of Types:**

- Checking done by a compiler is said to be static, while checking done when the target program runs is termed dynamic. In principle, any check can be done dynamically, if the target code carries the type of an element along with the value of the element.
- A sound type system eliminates the need for dynamic checking for type error because it allows us to determine statically that these errors cannot the target program runs. That is, if a sound type system assigns a type other than type_error to a program part, then type errors cannot occur when the target code for the program part is run.
- A language is strongly typed if its compiler can guarantee that the programs it accepts will execute without type errors.
- In practice, some checks can be done only dynamically. For example, if we first declare

  table: array[0. .255] of char;
  i: integer

  and then compute table[i], a compiler cannot in general guarantee that during execution, the value of i will lie in the range 0 to 255.

**Error Recovery:**

- Since type checking has the potential for catching errors in programs, it is important for a type checker to do something reasonable when an error is discovered.
- At the very least, the compiler must report the nature and location of the error. It is desirable for the type checker to recover from errors, so it can check the rest of the input.
- Since error handling affects the type-checking rules, it has to be designed into the type system right from the start; the rules must be prepared to cope with errors.

- The inclusion of error handling may result in a type system that goes beyond the one needed to specify correct programs.
- For example, once an error has occurred, we may not know the type of the incorrectly formed program fragment.
- Coping with missing information requires techniques similar to those needed for languages that do not require identifiers to be declared before they are used.

## DATA ABSTRACTION

**Q.27. Explain Data Abstraction.**

**Ans. Data Abstraction:**

- Abstraction is a mechanism to' hide irrelevant details and represent only the essential features so that one can focus on important things at a time; It allows managing complex systems by concentrating on the essential features only.
- For example, while driving a car, a driver only knows the essential features to drive a car such as how to use clutch, brake, accelerator, gears, steering, etc., and least bothers about the internal details of the car like motor, engine, wiring, etc.
- Abstraction can be of two types, namely, data abstraction and control abstraction. Data abstraction means hiding the details about the data and control abstraction means hiding the implementation details. In object-oriented approach, one can abstract both data and functions.
- However, generally, the classes in OOP are defined in such a way that the data is hidden from the outside world and the functions form the public interface. That is, the functions of the class can be directly accessed by other functions outside the class and the hidden data can be accessed indirectly with the help of these functions.
- Data abstraction refers to supplying the outside world with only basic knowledge about the data, covering the specifics of the context or implementation.
- The abstraction of data enables a programme to ignore the details of how a type of data is represented. Abstraction refers to the act of representing essential characteristics without including the details or explanations of the background.
- Classes use the abstraction method and are specified as a list of abstract attributes to operate on these attributes, such as distance, cost, size, etc. and functions.
- Data abstractions contribute to successful design of programmes:

(i) They assist with encapaulation (Information hiding).

(ii) They help reduce the complexity of interfaces with software.

(iii) They make more modifiable systems.

## COMPILATION OF OBJECT ORIENTED FEATURES

**Q.28. Explain compilation of object oriented features.**

**Ans.**

- A framework of the programming language in which data and their related processing ("methods") are defined as "objects." self-contained entities. Today's traditional, Object-Oriented Programming (OOP) languages, provide a structured set of rules for object creation and management.

  The data is stored in a traditional relational database or, if the data has a complex structure, in an object database.
- In object-oriented programming, there are three key characteristics that make them distinct from non-OOP languages: encapsulation, heritage and polymorphism.

(i) **Encapsulation Enforces Modularity:**

- Encapsulation refers to the development of self-contained modules that bind the data to processing functions. These types of user-defined data are called classes," and an example of a class is a "object."

- For example, a class could be Manager in a payroll system, and two instances (two objects) of the Manager class could be Pat and Jan. Encapsulation guarantees good modularity of code, which keeps routines distinct and less prone to conflict with each other.

**(ii) Inheritance Passes 'Knowledge Down:**

- In hierarchies, classes are created and inheritance enables the hierarchy to be passed down to the structure and techniques in class. That means that when adding functions to complex systems, less programming is needed.
- If a step is introduced at the bottom of a hierarchy, it is only important to include the processing and data associated with that particular step. All else is inherited. The ability to reuse existing objects is considered to be a major benefit of object technology.

**(iii) Polymorphism Takes Any Shape:**

- Object-oriented programming enables procedures to be generated regarding objects whose exact form is not known before runtime.
- For example, depending on the programme mode, a screen cursor will alter its shape from an arrow to a line. In response to mouse movement, the routine to move the cursor on screen will be written for "cursor" and polymorphism allows the cursor to take on whatever shape is required at runtime. It also makes it simple to incorporate new shapes.

## NON-IMPERATIVE LANGUAGE

**Q.29.Differentiate between Imperative and Non-imperative Language.**

**Ans. Non-Imperative Language:**

- Non-imperative programming language describes how the result should look like but not necessarily the procedures to achieve the desired result. Examples of well-known non-imperative language are Ruby, R, Haskell.
- It is also known as Declarative Programming as the name suggests is a type of programming paradigm that describes what programs to be executed. Developers are more concerned with the answer that is received. It declares what kind of results we want and leave programming language aside focusing on simply figuring out how to produce them. In simple words, it mainly focuses on end result. It expresses the logic of computation. Miranda, Erlang, Haskell, Prolog are a few popular examples of declarative programming.
- After getting the basics of understanding of both the languages now let us do discuss the key differences between these two different types of programming.

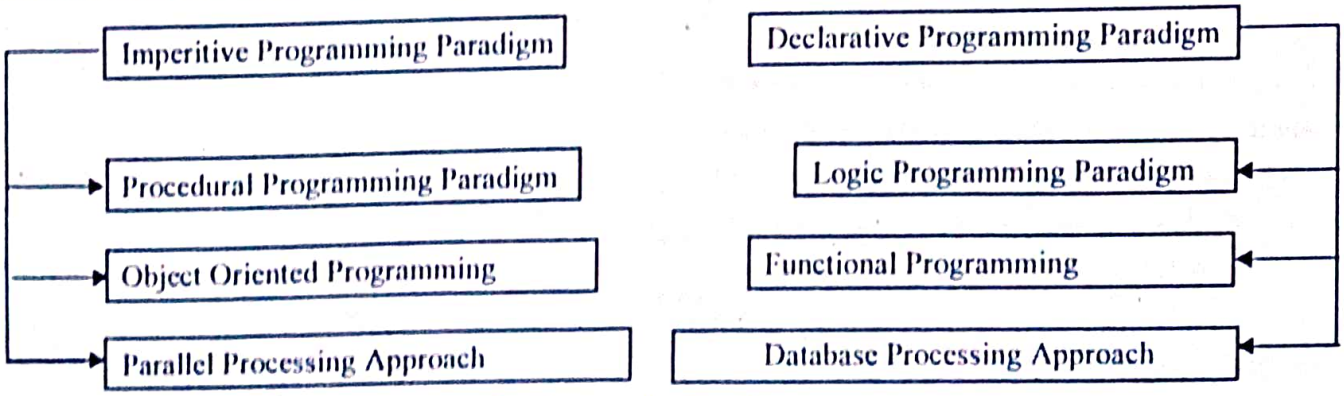



Fig. Programming Paradigms

- A Non-imperative programming focuses on applying a mathematical function to computer programming, functional programming evaluates an expression constructed using functions. A function, for example, is a statement that returns a single value based on the argument or inputs.

- The non-imperative programming language emphasizes the evaluation of expressions rather than the step-by-step execution of program instruction.
- A non-imperative programming language must use a higher level of abstraction than they would if they were learning to program in Pascal or C. Language processing (translation, style checking), pattern matching (vision, genetics) and process optimization are among examples (scheduling). Non Imperative programming languages include Ruby, R, Haskell, Scheme and ML.

**Difference between Imperative and Non-Imperative Programming:**

- The major difference between imperative and non-imperative programming is that the imperative language focus on how the program should follow its tasks, while the non-imperative, functional or logical language tells the program what it needs to do as opposed to how to do it.

Imperative programming in computer science "is a programming paradigm that describes computation in terms of statements that change a program state" (Libflow, nd.). Imperative program define the sequence of commands for the computer to perform. This is in contrast to declarative which expresses what the program should accomplish without prescribing how to do it in terms of sequences of action to be taken.

- The non-imperative programming language would be better in functional and logical programming. A pure functional language like Haskell has state changes that are only represented as functions that transform the state (Foss, 2015). As with functional programming language, logical programming consists of logical statements and programs executed by searching for proofs of the statement. Examples include Prolog and database query languages such as SQL.
- On the other hand, procedural language is an imperative language in which the program is built from one or more procedures (subroutines or functions). Imperative programming can also be used in structured programming and modular programming.

Q.25. Explain method to eliminate global common sub expression.

OR How global common subexpression can be eliminated with the help of "available" information? Explain with the help of example.

Ans.

- Global common subexpression computes value but in different basic blocks. For detection of such expression, we need to compute available expression.

Available expression :

- An expression x op y is available at point P if every path from the initial node of x op y and if after the last such evaluation and prior to reaching p there are no subsequent assignment, to x or y.
- To eliminate such global common subexpression we need to compute the set of all the expression available at the point just before the start of every block and end of every block. This is called as IN(B) and OUT(B).
- The computation of IN(B) and OUT(B) required the set of expression generated and killed by the basic block.
- To compute basic block solve the following expression

OUT(B) = IN(B) – KILL(B) ∪ GEN(B)

IN(B) = OUT(B)

**Algorithm :**

(1) $IN(B_1) = \phi$

$OUT(B_1 = GEN(B_1))$

(2) For (i = 2; i ≤ n; i + +)

{

IN(B) = U

OUT(B) = U – GEN (B)

}

(3) Flag = True

(4) While (flag) do

{

flag = false

for (i = 2; i ≤ n; i + +)

{

$IN_{new} = \phi$

For each predecessor P of $B_i$

$IN_{new}(B_i) = IN_{new}(B_i) \cap OUT(P)$

If $IN_{new}(B_i)$ is not equal to $IN(B_i)$ then

{

flag = true

$IN(B_i) = IN_{new}(B_i)$

$OUT(B_i) = IN(B_i) - KILL(B_i) \cup GEN(B_i)$

}

}

}

- Now after computing IN and OUT we will start eliminating global common subexpression
- For every statement 'S' of the form x = y op z such that y or z available at the beginning of the block containing S and neither y nor z is defined prior to the statement x = y op z in that block.

(a) Find all the definition reaching up to the 'S' statement that have y op z on the right.

(b) Create new temp.

(c) Replace each statement U = y op z found in the step (1) by

temp = y op z

U = temp

(d) Replace the statement x = y op z in block by x = temp

## LOOP OPTIMIZATION

Q.26. Write short note on loop optimization.

OR Explain loop optimization with an example.

OR What is the role of loop optimization in code optimization process?

OR What are different loop optimizations? Explain in brief.

**VTU**

**Q.25. Explain method to eliminate global common sub expression.**

**CS : 5-09(7M)**

**OR How global common subexpression can be eliminated with the help of "Available" information? Explain with the help of example.**

**CS : 5-04(10M)**

**Ans.**

- Global common subexpression computes value for in different basic blocks. For detection of such expression, we need to compute available expression.

**Available expression :**

- An expression x op y is available at point 'P' if every path from the initial node of x op y and, if after the last such evaluation and prior to reaching p there are no subsequent assignment to x or y.
- To eliminate such global common subexpression we need to compute the set of all the expression available at the point just before the start of every block and end of every block. This is called as IN(B) and OUT(B).
- The computation of IN(B) and OUT(B) required the set of expression generated and killed by the basic block.
- To compute basic block solve the following expression :

$$OUT(B) = IN(B) - KILL(B) \cup GEN(B)$$

$$IN(B) = OUT(B)$$

**Algorithm :**

(1) $IN(B_1) = \phi$

$OUT(B_1) = GEN(B_1)$

(2) For (i = 2; i <= n; i++)

{

$IN(B_i) = U$

$OUT(B_i) = U - GEN(B_i)$

}

(3) Flag = True

(4) While (flag) do

{

flag = false

for (i = 2; i <= n; i++)

{

$IN_{new} = \phi$

For each predecessor P of B

$IN_{new}(B_i) = IN_{new}(B_i) \cap OUT(P)$

If $IN_{new}(B_i)$ is not equal to $IN_{(B_i)}$ then

{

flag = true

$IN(B_i) = IN_{new}(B_i)$

$OUT(B_i) = IN(B_i) - KILL(B_i) \cup GEN(B_i)$

}

}

}

- Now after computing IN and OUT we will start eliminating of global common subexpression.
- For every statement 'S' of the form x = y op z such that y op z is available at the beginning of the block containing S and neither y nor z is defined prior to the statement x = y op z in the block do :

(a) Find all the definition reaching up to the 'S' statement that assigns y op z on the right.

(b) Create new temp.

(c) Replace each statement U = y op z found in the step (1) by

temp = y op z

U = temp

(d) Replace the statement x = y op z in block by x = temp.

## LOOP OPTIMIZATION

**Q.26. Write short note on loop optimization.**

**CS : 5-09(10M), CS : 5-04(10M)**

**OR Explain loop optimization with an example.**

**OR What is the role of loop optimization in code optimization process?**

**CS : 5-04(10M)**

**OR What are different loop optimizations? Explain in brief.**

**CS : 5-04(10M)**

**VTU**

**Q.25. Explain method to eliminate global common sub expression.**

**CS : S-09(7M)**

**OR How global common subexpression can be eliminated with the help of "Available" information? Explain with the help of example.**

**CS : S-14(13M)**

Ans.

- Global common subexpression computes value but in different basic blocks. For detection of such expression, we need to compute available expression.

**Available expression :**

- An expression x op y is available at point P if every path from the initial node of x op y and if after the last such evaluation and prior to reaching p there are no subsequent assignment, to x or y.
- To eliminate such global common subexpression are need to compute the set of all the expression available at the point just before the start of every block and end of every block. This is called as IN(B) and OUT(B).
- The computation of IN(B) and OUT(B) required the set of expression generated and killed by the basic block.
- To compute basic block solve the following expression :

$OUT(B) = IN(B) - KILL(B) \cup GEN(B)$

$IN(B) = OUT(B)$

**Algorithm :**

**(1)** $IN(B_1) = \phi$

$OUT(B_1 = GEN(B_1))$

**(2) For** $(i = 2; i \le n; i++)$

{

$IN(B) = U$

$OUT(B) = U - GEN(B)$

}

**(3) Flag = True**

**(4) While (flag) do**

{

**flag = false**

for $(i = 2;\ i \le n; i++)$

{

$IN_{new} = \phi$

For each predecessor P of $B_i$

$IN_{new}(B_i) = IN_{new}(b_i) \cup OUT(P)$

If $IN_{new}(B_i)$ is not equal to $IN_{(B_i)}$ then

{

flag = true

$IN(B_i) = IN_{new}(B_i)$

$OUT(B_i) = IN(B_i) - KILL(B_i) \cup GEN(b_i)$

}

}

}

- Now after computing IN and OUT we will start eliminating global common subexpression.
- For every statements 'S' of the form x = y op z such that y op z available at the beginning of the block containing S and neither nor z is defined prior to the statement x = y op z in that block do

(a) Find all the definition reaching up to the 'S' statement block have y op z on the right.

(b) Create new temp.

(c) Replace each statement U = y op z found in the step (1) by

temp = y op z

U = temp

(d) Replace the statement x = y op z in block by x = temp.

## LOOP OPTIMIZATION

**Q.26. Write short note on loop optimization.**

**CS : S-10,11(4M), CT : W-12(7M)**

**OR Explain loop-optimization with an example.**

**OR What is the role of loop-optimization in code optimization process?** **CT : W-12(7M)**

**OR What are different loop optimizations? Explain in brief.** **CS : W-18(7M)**

VBD

**Loop optimization :**

Loop optimization is the most valuable machine independent optimization because a program's inner loops are good candidates for improvement.

A loop invariant computation is one that computes the same value every time a loop is executed. Therefore, moving such a computation outside the loop leads to a reduction in the execution time.

Induction variables are those variables used in a loop, their values are in lock-step, and hence, it may be possible to eliminate all except one.

The important techniques of loop optimization are, code motion, induction variable elimination and reduction in strength.

**Code Motion :**

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

This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop invariant computation) and places the expression before the loop.

Note that the notion "before the loop" assumes the existence of an entry for the loop.

For example, evaluation of limit – 2 is a loop-invariant computation in the following while statement.

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

code motion will result in the equivalent of

t = limit – 2;

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

**Induction variables and reduction in strength :**

While code motion is not applicable to the quick sort example we have been considering the other two transformations.

Loops are usually processed inside out. For example, consider the loop around $B_3$. Only the portion of the flow graph relevant to transformations on $B_3$ is shown in fig.

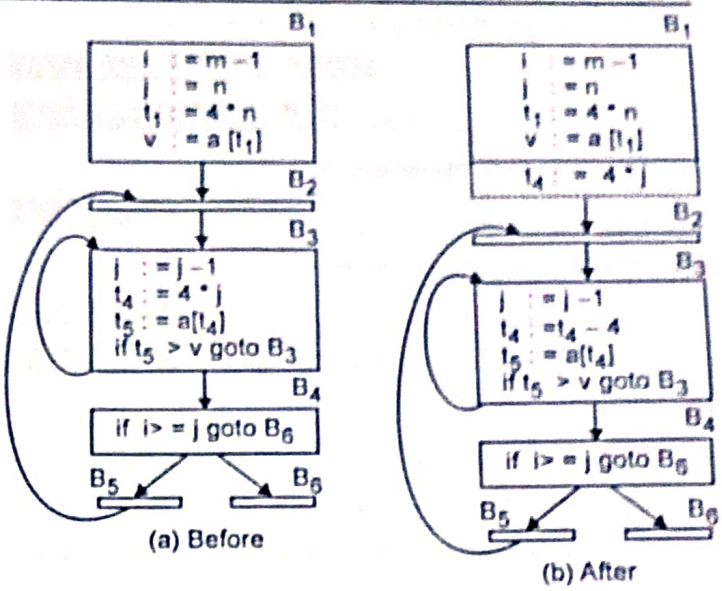



**Fig. Strength reduction applied to 4 * j in block $B_3$**

- Note that the values of j and $t_4$ remain in lock-step; every time the value of j decreases by 1, that of $t_4$ decreases by 4 because 4*j is assigned to $t_4$. Such identifiers are called induction variables.
- When there are two or more induction variables in a loop, it may be possible to get rid of all but one by the process of induction variable elimination. For the inner loop around $B_3$ in Fig. we cannot get rid of either j or $t_4$ completely; $t_4$ is used in $B_3$ and j in $B_4$.
- However, we can illustrate reduction in strength and illustrate a part of process of induction variable elimination.
- Eventually j will be eliminated when the outer loop of $B_2 - B_5$ is considered.

**Example :** As the relationship $t_4 = 4 * j$ surely holds after such an assignment to $t_4$ in Fig. and $t_4$ is not changed elsewhere in the inner loop around $B_3$, it follows that just after the statement j := j – 1 the relationship $t_4 = 4 * j - 4$ must hold.

**Q.27. Define dominator.** **CS : S-09 (5M)**

**OR Write short note on dominators and dominator computing algorithm.** **CT : S-10 (4M)**

VTU

**OR What is dominator? How is it used to detect a loop?**

**CT : S-14(3M), W-10(4M), S-12(5M).**

**CS : W-14(2M), S-11, W-11(3M)**

**OR How does dominators help to detect the loops in program?**

**CT : S-11(3M), W-12(2M)**

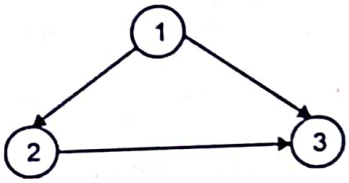
**OR Illustrate steps to be carried out for loop detection with the help of an example.** **CS : W-13(7M)**

**OR How a loop can be detected in given TAC?** **CT : W-09(7M)**

**Ans. Dominator :**

- If every path from the initial node of the flow graph to n goes through node d then we can say that node d dominates node n.
- Under this definition, every node dominates itself and the entry of the loop dominates all node in the loop.

**Example :**

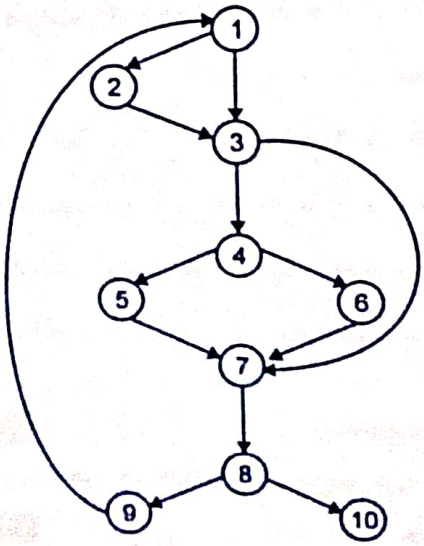



- In this flow graph, the dominator of node 3 is node 1 only, since all the paths reaching upto the node 3 from node 1 does not go through node 2.
- The properties of the Dominator (DOM) are as follows :

(1) They are reflexive, i.e. every node dominates itself.

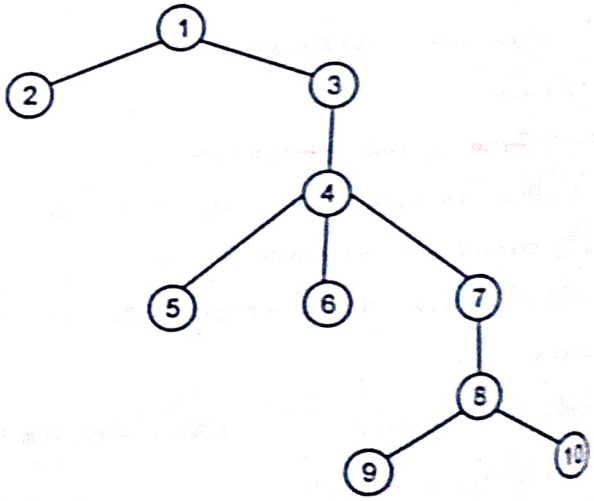
(2) They are transitive, i.e. if a dom b and b dom c implies a dom c.

- For example, consider a flow graph ;




| Node no. | Dominator set |
|---|---|
| 1 | 1 |
| 2 | 1, 2 |
| 3 | 1, 3 |
| 4 | 1, 3, 4 |
| 5 | 1, 3, 4, 5 |
| 6 | 1, 3, 4, 5, 6 |
| 7 | 1, 3, 4, 7 |
| 8 | 1, 3, 4, 7, 8 |
| 9 | 1, 3, 4, 7, 8, 9 |
| 10 | 1, 3, 4, 7, 8, 10 |

- For storing the set of dominator, static or dynamic dist are used.
- But this requires large amount of space. Thus an alternate representation is used i.e. tree representation.




- The existence of dominator tree is based on the property that each node n has unique immediate dominator m, such that m is the last dominator of n on any path from the initial node to n.
- In terms of dom relation, the immediate dominator M has property that if d ≠ n and d dom n, then d dom m.
- Thus, the immediate dominator in above flow graph will be,

| Node no. | Immediate dominator |
|---|---|
| 1 | 1 |
| 2 | 1 |
| 3 | 1 |

| | |
|---|---|
| 4 | 3 |
| 5 | 4 |
| 6 | 4 |
| 7 | 4 |
| 8 | 7 |
| 9 | 8 |
| 10 | 8 |

**Application of dominator :**

(1) One of the important application of dominator is to determine the loops of a flow graph. There are 2 properties of such loops :

(a) A loop must have a single entry point called the 'Header'.

(b) There must be at least one way to iterate the loop, i.e. at least one path back to the header.

(2) The best way of finding all loops in a flow graph is to search for edges in the flow graph always heads dominators their tails.

**Loop detection :**

- In a flow graph loop can be defined as a cycle that satisfies two properties :

(a) The loop should have a single node or header, so that it will be possible to move all of the loop invariant computation in a unique place i.e. in front of header.

(b) Loop should be strongly connected in other words it should be possible to go from any node of loop to any other node while staying within the loop. This is needed until at least some of the loops get executed repeatedly.

- When the flow graph contains one or more back edges, then only one or more loops / cycles, are there in flow graph.
- Thus for detection of loop identification of back edges is required.

**Identification of the back edges :**

- To detect loop, it is necessary to identify the back edges in the flow graph.
- If there is no back edge then there is no loop and if there is back edge then there may exist one or more loop formed by back edge.

**Back edge detection :**

- For back edge section, we do depth first search of the program flow graph and during the search when a node is visited a unique number associated with it is called as depth first number.
- This number will then be used to decodes whether a → b is back edge or not.

**Algorithm for back edge detection :**

```
Number = Total number of nodes in a program
                flow graph
search (m)
{
    if (m is not visited) then
    {  mark m = visited ;
    DFN (m) = Number ;
    Number = Number – 1;
    For every successor s of m do
    Search (s) :
    } // if end.
main( )
    {
      Search (n);          /*no. is initial node of graph */
    } // main end
}  // search end
```

**Q.28. Find the dominator of each node of following flow graph. Also construct dominator tree.** CS : W-14(5M)

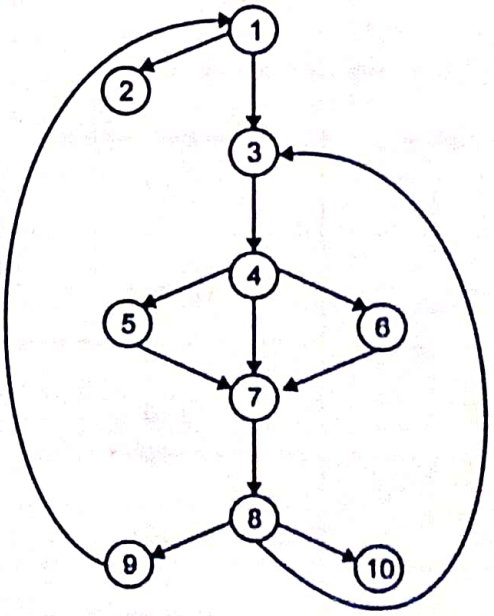
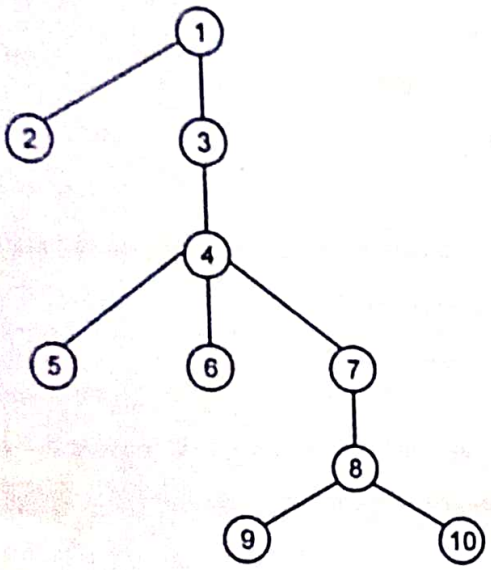



**Fig. Flow graph**

**Ans.** Dominator set are as follows :

| Node no. | Dominator set |
|---|---|
| 1 | 1 |
| 2 | 1, 2 |
| 3 | 1, 3 |
| 4 | 1, 3, 4 |
| 5 | 1, 3, 4, 5 |
| 6 | 1, 3, 4, 6 |
| 7 | 1, 3, 4, 7 |
| 8 | 1, 3, 4, 7, 8 |
| 9 | 1, 3, 4, 7, 8, 9 |
| 10 | 1, 3, 4, 7, 8, 10 |

Dominator tree is as follows :




Immediate dominator in flow graph is as follows :

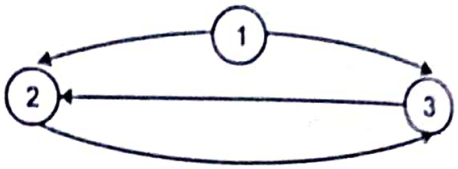
| Node no. | Immediate dominator |
|---|---|
| 1 | 1 |
| 2 | 1 |
| 3 | 1 |
| 4 | 3 |
| 5 | 4 |
| 6 | 4 |
| 7 | 4 |
| 8 | 7 |
| 9 | 8 |
| 10 | 8 |

**Q.29. Define reducible flow graph.** CS : S-09(2M)

**OR What do you mean by reducible flow graph? Explain with suitable example.** CS : W-10(6M), W-12(4M)

**OR What is need of reducible flow graph?** CT : S-12(2M)

**OR What is meant by reducible flow graph? Is following graph reducible? Justify your answer.** CT : W-12(5M), S-11(4M)




**OR Write short note on reducible flow graph.** CT : S-14(4M)

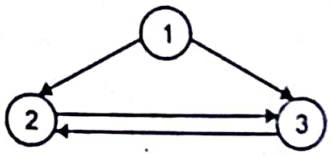
**Ans. Reducible flow graph :**

- A flow graph G is said to be reducible, if its edges can be partitioned into two disjoint groups, forward edges and backward edges.
- Reducible flow graph has two properties :

  **(1)** The forward edges form an acyclic graph in which every node can be reached from the initial node G.

  **(2)** The back edges are those edges whose heads dominates their tails.

- For example, consider the flow graph.




- In above flow graph, no edges head dominates the tail of that edge, thus there is no back edge.
- It could has been a reducible graph, if the entire graph had been acyclic.
- But that is not the case. Thus it is not a reducible flow graph.

**Q.30. Define induction variable.** CS : S-09(2M)

**OR Write short note on elimination of induction variable.** CS : W-14(5M)

**Ans. Induction variable :**

- We define induction variable of a loop as those name whose only assignments within the loop are of the form

I = I ± C, where C is a constant or a name whose value does not change within the loop.

A basic induction variable may or may not form an arithmetic progression at the loop header.

For example, consider the flow graph shown in Fig. In the loop formed by $B_2$, I is a basic induction variable.

**Algorithm for detecting and eliminating induction variables :**

An algorithm exists that will detect and eliminate induction variables. Its method is as follows :

(1) Find all of the basic induction variables by scanning the statements of loop L.

(2) Find any additional induction variables and for each such additional induction variable A, find the family of some basic induction B to which A belongs. (If the value of A at the point of assignment is expressed as $C_1B + C_2$, then A is said to belong to the family of basic induction variable B). Specifically, we search for names A with single assignments to A within loop L, and which have one of the following forms :

A = B * C

A = C * B

A = B / C

A = B ± C

A = C ± B

Where, C is a loop constant, and B is an induction variable, basic or otherwise. If B is basic, then A is in the family of B. If B is not basic, let B be in the family of D, then the additional requirements to be satisfied are :

(a) There must be no assignment to D between the lone point of assignment to B in L and the assignment to A.

(b) There must be no definition of B outside of L reaches A.

(3) Consider each basic induction variable B in turn. For every induction variable A in the family of B :

(a) Create a new name, temp.

(b) Replace the assignment to A in the loop with A = temp.

(c) Set temp to $C_1B + C_2$ at the end of the preheader by adding the statements :

temp = $C_1$ * B

temp = temp + $C_2$ */* ***omit if $C_2 = 0$*** */*

(d) Immediately after each assignment B = B + D, where D is a loop invariant, append :

temp = temp + $C_1$ * D

If D is a loop invariant name, and if $C_1 \neq 1$, create a new loop invariant name for $C_1$ * D, and add the statements :

temp1 = $C_1$ * D

temp = temp + temp1

(e) For each basic induction variable B whose only uses are to compute other induction variables in its family and in conditional branches, take some A in B's family, preferably one whose function expresses its value simply, and replace each test of the form B reloop X goto Y by :

temp2 = $C_1$ * X

temp2 = temp2 + $C_2$ */* ***omit if $C_2 = 0$*** */*

if temp reloop temp2 goto Y

Delete all assignments to B from the loop, as they will now be useless.

(f) If there is no assignment to temp between the introduced statement A = temp (step 1) and the only use of A, then replace all uses of A by temp and delete the statement A = temp.

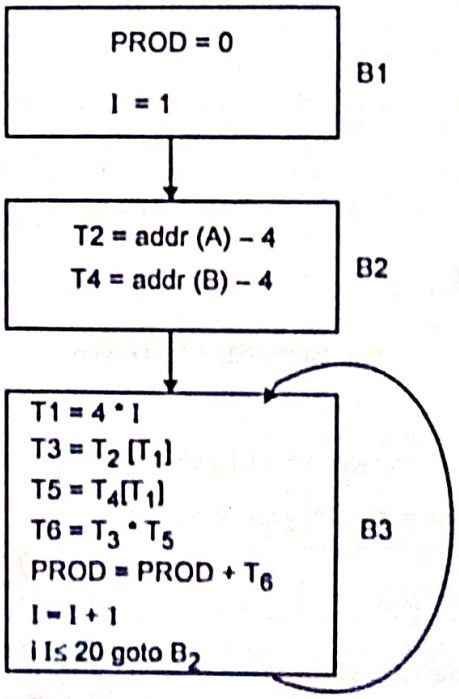



**Fig. (a) Flow graph where I is a basic induction variable**

In the flow graph shown in Fig. (a) we see that I is a basic induction variable, and T1 is the additional induction variable in the family of I, because the value of T1 at the point of assignment in the loop is expressed as T1 = 4 * I. Therefore, according to step 3b, we replace T1 = 4 * I by T1 = temp. And according to step 3c, we add temp = 4 * I to the preheader. We then append the statement temp to temp + 4 after statement (10), as shown in Fig. (a) as per step 3d. And according to step 3e, we replace the statement if I ≤ 20 goto B2 by

temp1 = 80

if (temp ≤ temp1) goto B2, and delete I = I + 1

The results of these modifications are shown in Fig. (b)

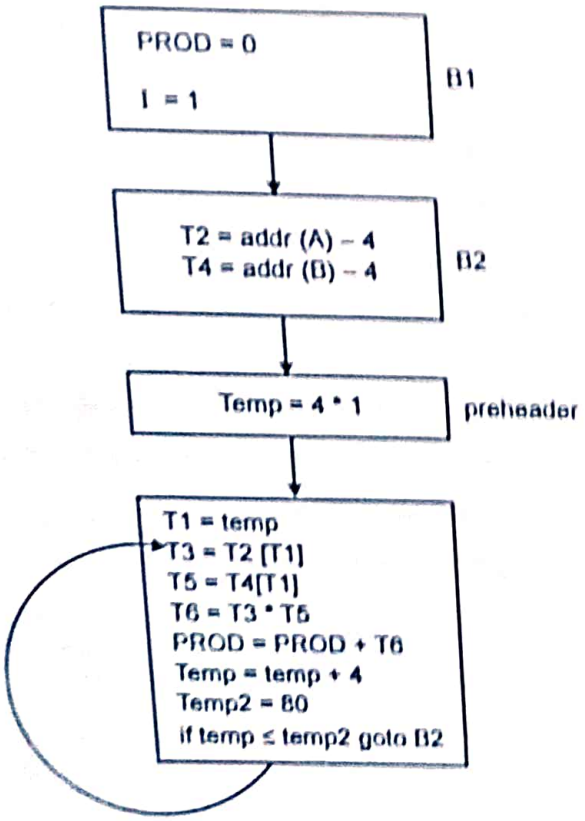



Fig. (b) Modified flow graph

**Q.31. How does strength reduction and dead code elimination help for code optimization? Explain with suitable example.**

**CT 1 & 2 (2M)**

Ans.

**(1) Strength reduction :**

- Loops are usually processed inside-out. For example, consider the loop around $B_3$. Only the portion of the flow graph relevant to transformations on $B_3$ is shown in Fig. below.
- The values of j and $t_4$ remain in lock-step; every time the value of j decreases by 1, that $t_4$ decreases by 4 because 4*j is assigned to $t_4$. such identifiers are called induction variables.
- When there are two or more induction variables in a loop, it may be possible to get rid of all but one by the process of induction variable elimination.

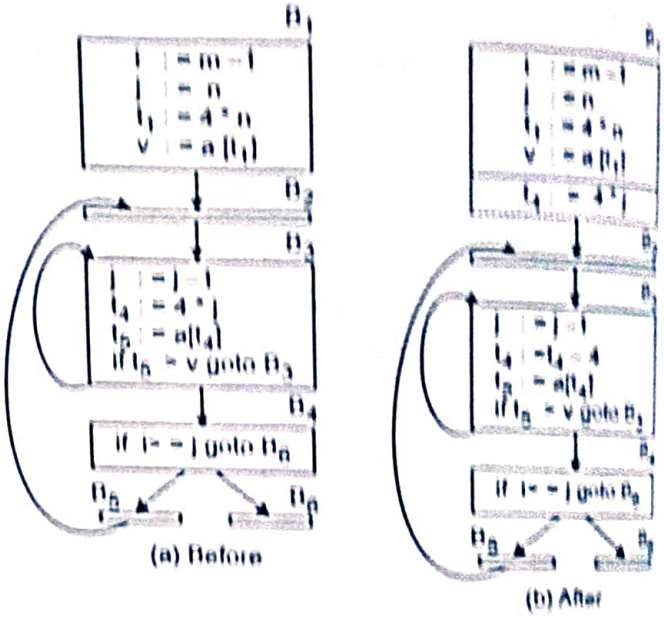



Fig. Strength reduction applied to 4 * j in block $B_3$

- For the inner loop around $B_3$ in Fig. (a), we cannot get rid of either j or $t_4$ completely; $t_4$ is used in $B_3$ and j in $B_4$.
- However, we can illustrate reduction in strength and illustrate a part of process of induction variable elimination. Eventually j will be eliminated when the outer loop of $B_2 - B_5$ is considered.

**(2) Dead code elimination :**

- A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point.
- A related idea is dead or useless code, statements that compute values that never get used.
- While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations.
- If we use following statement at various points in the program, the statement is like

If (debug) print .....

By a data-flow analysis, it may be possible to deduce that each time the program reaches this statement

Deducing at compile time that the value of an expression is constant and using the constant instead is known as constant flooding.

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

Consider the basic block which is considered previously.

$x := t_3$

$a[t_2] := t_5$

$a[t_4] := t_3$

goto $B_2$

The above basic block is after copy propagation. The dead code elimination removes the assignment to x and transforms into

$a[t_2] := t_5$

$a[t_4] := t_3$

goto $B_2$.

**Q.12. Write short note on function preserving transformation in code optimization.** **CT : S-09,12(4M)**

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

**Function preserving transformations :**

Common sub-expression elimination, copy propagation, dead-code elimination and constant folding are common examples of such function preserving transformations.

**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.

We can avoid recomputing the expression if we can use the previously computed value. This can be eliminated by using DAG representation of basic block.

- For example, block $B_5$ shown in Fig. below recalculates 4 * i and 4 * j.
- The assignment to $t_7$ and $t_{10}$ have the common subexpression 4 * i and 4 * j, respectively.
- There is elimination of common subexpression using $t_6$ instead of $t_7$ and $t_8$ instead of $t_{10}$.

| $B_5$ | $B_5$ |
|---|---|
| $t_6 := 4 * i$ | $t_6 := 4 * i$ |
| $x := a[t_6]$ | $x := a[t_6]$ |
| $t_7 := 4 * i$ | $t_8 := 4 * j$ |
| $t_8 := 4 * j$ | $t_9 := a[t_8]$ |
| $t_9 := a[t_8]$ | $a[t_6] := t_9$ |
| $a[t_7] := t_9$ | $a[t_8] := X$ |
| $t_{10} := 4 * j$ | goto $B_2$ |
| $a[t_{10}] := X$ | |
| goto $B_2$ | |

**Fig. Local common sub-expression elimination.**

(2) **Copy propagation :**

- The assignment statements of the form $f := g$ called copy statement, or copies for short.

Consider a basic block

$x := t_3$

$a[t_2] := t_5$

$a[t_4] := x$

goto $B_2$

This can be further improved by eliminating x using two new transformations.

- The idea behind the copy-propagation transformation is to use g for f, wherever possible after the copy statement $f := g$.
- The assignment $x := t_3$ in block $B_1$ is a copy.

- Copy propagation applied to block yields

$x = t_1$

$a[t_2] = t_5$

$a[t_4] = t_3$

goto $B_2$

**(3) Dead code elimination :**

- A variable is live at a point in a program if its value can be used subsequently, otherwise, it is dead at that point
- A related idea is dead or useless code, statements that compute values that never get used.
- While the propogation unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations.
- If we use following statement at various points in the program, the statement is like

If (debug) print .....

- By a data-flow analysis, it may be possible to deduce that each time the program reaches this statement.
- Deducing at compile time that the value of an expression is constant and using the constant instead is known as constant flooding.

**Advantage :**

It turns the copy statement into dead code.

**Q.33. Consider the following code :**

```
for (j = 0 ; j < 20 ; j++)
{
    for (k = 0 ; k <= 20, k++)
    {
        x [j] [k] = y[j] [k] + z [j] [k];
    }
}
```

**Perform the following loop optimizations :**

**(i) Elimination of common sub-expression**

**(ii) Code motion**

**(iii) Reduction in strength.** **CT:W-13(9M)**

**Ans. (i) Common sub-expression elimination :**

- The offset for each of the arrays is the same. So, we can eliminate that common sub-expression (i.e. compute the value in $t_1$ only once and use it 3 times)
- $C_1$ is a constant representing W * N. W is the width of an element and N is the number of columns in each of the arrays

```
for (j = 0 ; j <= 30 ; j ++) {
for (k = 0 ; k <= 30 , k++) {
t2 = C1 * j + W * k ;
X [t2] = y [t2] + z [t2] ;
}
}
```

**(ii) Code Motion :**

The expression $C_1$ * j is constant in the inner loop and can be taken out :

```
for (j = 0 ; j <= 30 ; j ++) {
t1 = C1 * j ;
for (k = 0 ; k <= 30 ; k++) {
t2 = t1 * j + W * k ;
X [t2] = y [t2] + z [t2] ;
}
}
```

**(iii) Loop reduction in strength :**

We can replace multiplications by additions in the inner and outer loop since $t_1$ and $t_2$ are induction variables.

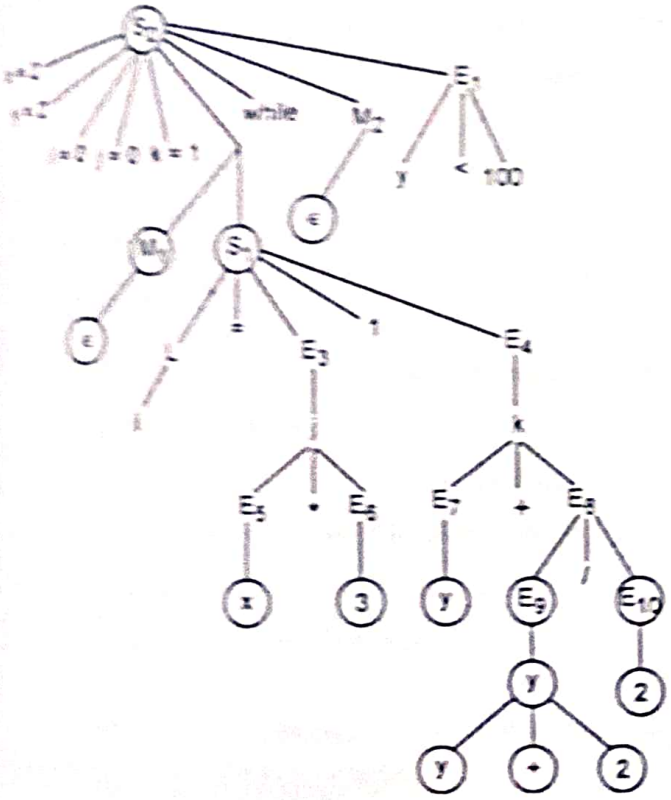
```
t1 = 0 ;
for (j = 0 ; j <= 30 ; j ++) {
t2 = t1 + 0 ;
for (k = 0 ; k <= 30 ; k++) {
x [t2] = y [t2] + z [t2] ;
t2 = t2 + W ;
}
t1 = t1 + C1 ;
}
```

**Q.4** Optimize following code using appropriate techniques of code optimization :

```
x=2; y=2; i=0; j=0; k=1;
while (y < 100)
{
    i=j/k;
    j=x*3;
    k=j+y/2;
    y=y+2;
}
```

**CS : S-14, 15, 16**

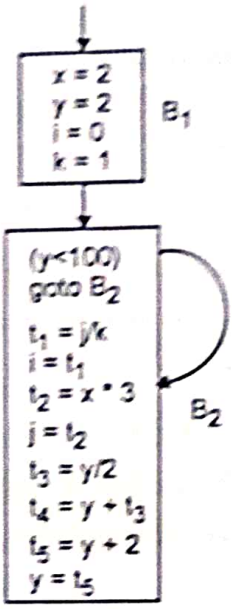
**Ans** Basic representation of a code :




TAC is as follows :

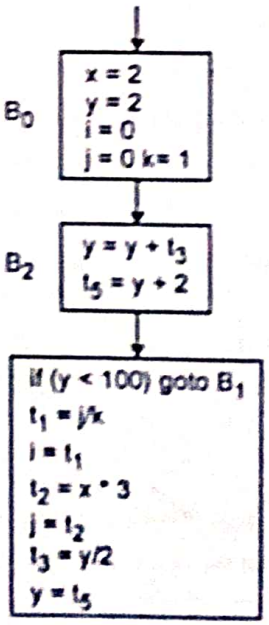
| | |
|---|---|
| 100 | x = 2 |
| 101 | y = 2 |
| 102 | i = 0 |
| 103 | j = 0 |
| 104 | k = 1 |
| 105 | if (y < 100) goto 117 |
| 106 | go to 117 |
| 107 | $t_1 = j/k$ |
| 108 | $i = t_1$ |
| 109 | $t_2 = x * 3$ |
| 110 | $j = t_2$ |
| 111 | $t_3 = y/2$ |
| 112 | $t_4 = y + t_3$ |
| 113 | $t_5 = y + 2$ |
| 114 | $y = t_5$ |
| 117 | exit |

Basic block computation is as follows :




(i) **Elimination of loop invariant computation :**

- In block $B_2$, $yt_4 = y + t_3$ and $t_5 = y + 2$ are loop invariant computations.
- In this, moving these loop invariant computation outside the block would not change the meaning of code.




**VTU**

(ii) Removal of common sub-expression :

- A common sub-expression is an expression having same or common value.
- For identifying the common sub-expression, we use syntax tree or graph called as directed acyclic graph.
- Here, there is no common sub-expression so cannot eliminate it.

(iii) Elimination of induction variable :

As there is no induction variable present, we can't eliminate it.

**Q.35. Write short note on loop invariant computations.**

**GTU : [illegible]**

**Ans.** Loop invariant computations :

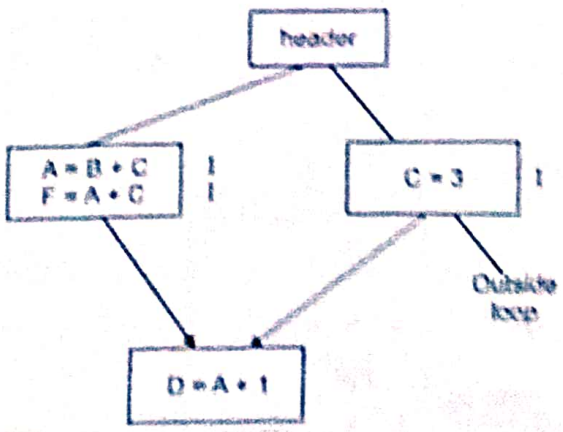
- Loop invariant computations means the computations computing a value which is not dependent on how many times loop gets executed.
- Identification of loop invariant computation requires the detection of loops in the program.
- Whether a loop exists in the program or not depends on the programs control flow.
- To eliminate loop invariant computations, we first identify the invariant computations and then move them outside loop if the move does not lead to a change in the program's meaning.

**Q.36. Write short note on three conditions of code motion in loop invariant computation.**

**GTU : [illegible]**

**Ans.**

- Loop invariant code consist of statements or expressions which can be moved outside of the body of a loop without affecting the semantic of program.
- Loop invariant computation and code motion (LICM) move a loop invariant statement within a loop to preheader of the loop.




Where, 1-2 shortest statements are basic, the loop invariant.

- Observations are as follows.

(1) Loop invariants :

Operands are defined outside loop or are defined by invariant.

(2) Code motion : Not all invariant statements can be moved to preheader.

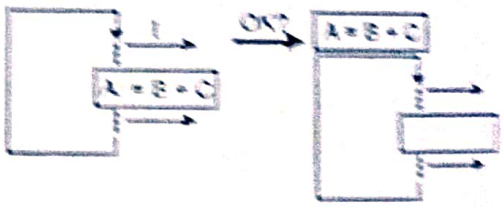
Algorithm is as follows :

- Detect loop invariant computations.
- Check condition for code motion.
- Code transformation conditions of code motion are as follow.

(i) Correctness : Movement does not change the program semantic.

(ii) Performance : Code should not be slowed down.

Example :




(3) Need information :

Control flow of the loop such that dominate all the exits.

Other definition :

No other definition of A exist.

Other uses :

All uses of A are dominated by block b.

**Q.37. Write short note on loop jamming and loop unrolling.**

**GTU : [illegible]**

**GTU : [illegible]**

**Ans.** Loop unrolling :

- Loop unrolling involves replicating the body of the loop [illegible] the required number of times if the number of iterations are [illegible]
- For example, consider following loop.

```
I = 1
While (I <= 100)
{
x[I] = 0
I++;
}
```

**(ii) Removal of common sub-expression :**

- A common sub-expression is an expression having same or common value.
- For identifying the common sub-expression, we use special type of graph called as directed acyclic graph.
- Here there is no common sub-expression so cannot eliminate it

**(iii) Elimination of induction variable :**

As there is no induction variable present, so can't eliminate it

**Q.35. Write short note on loop invariant computations.**

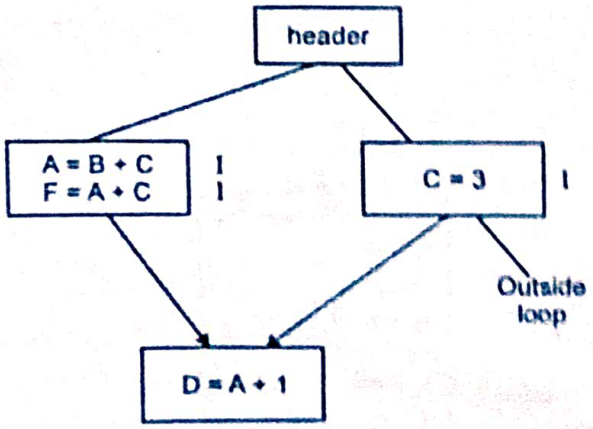
**Ans. Loop invariant computations :**

- Loop invariant computation means the computations computing a value which is not dependent on how many times loop gets executed.
- Identification of loop invariant computation requires the detection of loops in the program.
- Whether a loop exists in the program or not depends on the programs control flow.
- To eliminate loop invariant computations, we first identify the invariant computations and then move them outside loop if the move does not lead to a change in the program's meaning.

**Q.36. Write short note on three conditions of code motion in loop invariant computation.**

**Ans.**

- Loop invariant code consist of statements or expressions which can be moved outside of the body of a loop without affecting the semantic of program.
- Loop invariant computation and code motion (LICM) move a loop invariant statement within a loop to preheader of the loop.




Where, I - Labelled statements are looks like loop invariant

- Observations are as follows

**(1) Loop invariants :**

Operands are defined outside loop or are defined by loop invariant

**(2) Code motion :** Not all invariant statements can be moved to the preheader.

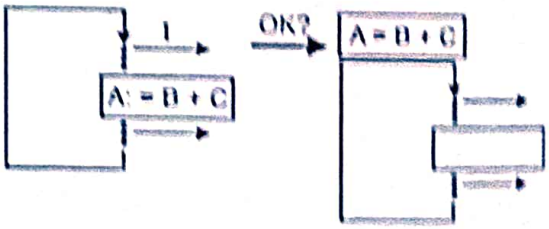
Algorithm is as follows

- Detect loop invariant computations
- Check condition for code motion
- Code transformation conditions of code motion are as follows

**(i) Correctness :** Movement does not change the program semantics

**(ii) Performance :** Code should not be slowed down

**Example :**




**(3) Need information :**

Control flow of the loop such that dominate all the exits

**Other definition :**

No other definition of A exist

**Other uses :**

All uses of A are dominated by block b.

**Q.37. Write short note on loop jamming and loop unrolling**

**Ans. Loop unrolling :**

- Loop unrolling involves replicating the body of the loop to make the required number of tests if the number of iterations are known
- For example, consider following loop

```
I = 1
Where (I < = 100)
{
x [ I ] = 0
I ++;
}
```

In above case I <= 100 will be performed 100 times

But if the body of the loop is replicated, then the number of times this test will need to be performed will be 50.

After replication of the body, the loop will be :

```
I = 1
While ( I ≤ 100 )
{
x [ I ] = 0.
I ++;
x [ I ] = 0.
I ++;
}
```

It is possible to close always any divisor for the number of times the loop is executed and the body will be replicated many times.

Unrolling once i.e. replicating the body to form two copies of the body it saves 50% of the maximum possible execution.

**Loop Jamming :**

Loop jamming is a technique that merges the bodies of two loops if the two loops have the same number of iterations and they use the same indices.

This eliminates the test of one loop.

For example, consider the following loop :

```
{
for (I = 0; I < 10; I++)
for (J = 0; J < 10; J++)
X [I, J] = 0.
for (I = 0; I < 10, I ++)
X [X, J] = 1;
}
```

Here, the bodies, of the loops on I can be concatenated. Hence the result of loop jamming will be :

```
{
for (I = 0; I < 10; I ++)
    {
    for (J = 0; J < 10; J ++)
    {
    X [I, J] = 0;
    X [I, J] = 1;
    }
}
```

- The following conditions are sufficient for making loop jamming legal :

(i) No quantity is computed by the second loop at the iteration I if it is computed by the first loop at iteration $J \geq I$.

(ii) If a value is computed by the first loop at iteration $J \geq I$ then this value should not be used by second loop at iteration I.

**Q.38. What do you mean by u-d chaining? What are the applications of u-d chaining?** **CT : S-09(4M)**

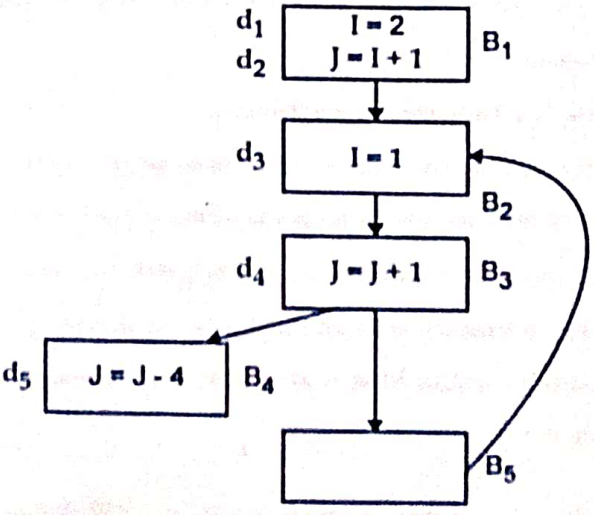
**Ans. u-d chaining :**

- If the use of A in block B is preceded by its definition, then the u-d chain of A contains only the last definition prior to this use of A.
- If the use of A in block B is not preceded by any definition of A, then the u-d chain for this use consist of all definition of A in IN (B).
- The u-d chain take up much space, it is important for an optimizing compiler to format them completely.

**Applications of u-d chains :**

- There are variety of uses to which u-d chain information can be put.
- It is used to identify the loop invariant computations.

**Example :**




The IN and OUT of above flow graph will be

| Block | IN | OUT |
|---|---|---|
| $B_1$ | $\phi$ | $\{d_1, d_2\}$ |
| $B_2$ | $\{d_1, d_2, d_3, d_4, d_5\}$ | $\{d_2, d_3, d_4, d_5\}$ |
| $B_3$ | $\{d_2, d_3, d_4, d_5\}$ | $\{d_3, d_4\}$ |
| $B_4$ | $\{d_3, d_4\}$ | $\{d_5, d_3\}$ |
| $B_5$ | $\{d_3, d_4, d_5\}$ | $\{d_3, d_4, d_5\}$ |

- In above the flow graph $d_2$ uses I, $d_4$ and $d_5$ uses J.
- The use of I at $d_2$ in block $B_1$ is preceded by definition of I in $B_1$ i.e. $d_1$. Hence ud-chain for I in $d_2$ consist $\{d_1\}$
- Similarly use of J at $d_4$ in block $B_3$ is not preceded by the definition of J in $B_3$.
- Thus, we have to consider IN $[B_3]$ i.e. $\{d_2, d_3, d_4, d_5\}$
- Out of these except $d_3$, all contains definition of J. Thus the ud chain for J in $d_4$ is $\{d_2, d_4, d_5\}$
- The use of J at $d_5$ in block $B_4$ is not preceded by definition of J, so consider IN $[B_4] = \{d_3, d_4\}$
- Of these $d_4$ contains definition of j. Hence ud-chain for J and $d_5$ consists in $\{d_4\}$.
- To obtain such a graph, we must partition the intermediate code into basic blocks
- This requires identifying leader statements which are defined as follows :

(1) The first statement is a leader statement.

(2) The target of a conditional or unconditional goto is a leader.

(3) A statement immediately follows a conditional goto is a leader.

- A basic block is a sequence of TAC statements that can be entered only at beginning and control ends after the execution of the last statement, without a halt or any possibility of branching, except at the end

**Q.39. Calculate u-d chain for the statement "a = a * I" in following code :**

(i) a = 1

(ii) i = 2

(iii) if i >= 10 goto (8)

(iv) a = a * i

(v) $t_1$ = i + 1

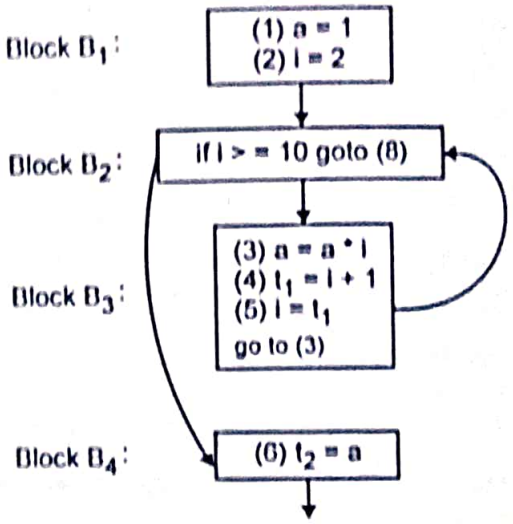
(vi) i = $t_1$

(vii) goto (3)

(viii) $t_2$ = a

CS : S-07(14M)

**Ans.** **Step (1) : The program flow graph will be :**




**Step (2) : To find GEN and KILL code :**

| Block | GEN | KILL |
|---|---|---|
| $B_1$ | {1, 2} | {3, 5} |
| $B_2$ | {$\phi$} | {$\phi$} |
| $B_3$ | {3, 4, 5} | {1, 3} |
| $B_4$ | {6} | {$\phi$} |

**Step (3) : To find predecessor :**

| Block | Predecessor |
|---|---|
| $B_1$ | $\phi$ |
| $B_2$ | $B_1, B_4$ |
| $B_3$ | $B_2$ |
| $B_4$ | $B_2, B_3$ |
| $B_5$ | $B_4$ |

Step (4) : To find iteration

Iteration (1) :

| Block | IN | OUT |
|---|---|---|
| $B_1$ | {ϕ} | {1, 2} |
| $B_2$ | {ϕ} | {ϕ} |
| $B_3$ | {ϕ} | {3, 4, 5} |
| $B_4$ | {ϕ} | {6} |

Iteration (2) :

| Block | IN | OUT |
|---|---|---|
| $B_1$ | {ϕ} | {1, 2} |
| $B_2$ | {1, 2, 3, 4, 5} | {1, 2, 3, 4, 5} |
| $B_3$ | {ϕ} | {3, 4, 5} |
| $B_4$ | {ϕ} | {6} |

Iteration (3) :

| Block | IN | OUT |
|---|---|---|
| $B_1$ | {ϕ} | {1, 2} |
| $B_2$ | {1, 2, 3, 4, 5} | {1, 2, 3, 4, 5} |
| $B_3$ | {1, 2, 3, 4, 5} | {3, 4, 5} |
| $B_4$ | {1, 2, 3, 4, 5} | {1, 2, 3, 4, 5} |

Iteration (4) :

| Block | IN | OUT |
|---|---|---|
| $B_1$ | {ϕ} | {1, 2} |
| $B_2$ | {1, 2, 3, 4, 5} | {1, 2, 3, 4, 5} |
| $B_3$ | {1, 2, 3, 4, 5} | {3, 4, 5} |
| $B_4$ | {1, 2, 3, 4, 5} | {1, 2, 3, 4, 5, 6} |

OUT of iteration III and IV are same

The use of $x = x * i$ at 3 in $B_3$ is not preceded by definition of x and i. Thus we consider IN ($B_3$), which is from iteration IV, equal to {1, 2, 3, 4, 5} of these all for 4 are definition of x and i, so our claim for $x = x * i$ at 3 is 1, 2, 3, 4, 5

Q.46 What is basic block? Explain with suitable example. Also define the GEN and KILL set for a basic block.

Ans : Basic block :

- It is the sequence of three address statements that is entered at the beginning and control exits after the execution of last statement without any halt or any branching, except at the end.
- Some of the loop invariant computations are as follows

(a) If there is an expression inside the loop such as $I = constant$ or $I = C$ where C = constant the expression is loop invariant.

(b) A variable initialized in a variable which is not manipulated inside the loop.

- The set of all definitions capable of reaching to a point immediately after the last statement of block B will be called as OUT (B)
- We compute both IN (B) and OUT (B) for every block B. GEN (B) and KILL (B) are defined as follows

GEN (B) = The set of all the definitions generated in block B

KILL (B) : The set of all definitions outside block B that define the same variable as are defined in block B

- IN (B) and OUT (B) are defined by the following set of equations, which are called "data flow equations"

IN (B) = OUT (p)

OUT (B) = IN (B) – KILL (B) ∪ GEN (B)

Algorithm :

For computing smallest IN (B) and OUT (B) is as follows

(1) For each block B do

{

IN (B) = ϕ

OUT (B) = GEN (B)

}

(2) flag = true

(3) while (flag) do

{

flag = false

for each block B do

{

$IN_{new}$ (B) = ϕ

for each predecessor P of B

$IN_{new}(B) = IN_{new}(B) \cup OUT(P)$

if $IN_{new}(B) \neq IN(B)$ then

{

flag = true

$IN(B) = IN_{new}(B)$

OUT (B) = IN (B) – KILL (B) ∪

GEN (B)

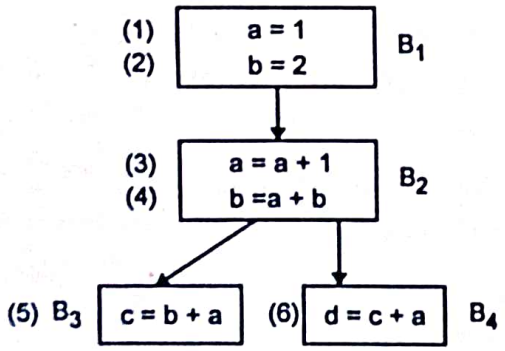
}

}

}

- Initially we take IN (B) for every block that is to be an empty set and we take OUT (B) to be GEN (B), and we compute $IN_{new}(B)$.
- If it is different from IN (B), we compute a new OUT (B) and go for the next iteration.
- This is continued until IN (B) comes out to be the same for every B in a previous and current iteration.
- For example, Consider following flow graph.

**Step (1) :**




**Step (2) : Compute GEN and KILL :**

| Block | GEN | KILL |
|---|---|---|
| $B_1$ | {1, 2} | {3, 4} |
| $B_2$ | {3, 4} | {1, 2} |
| $B_3$ | {5} | { ϕ } |
| $B_4$ | {6} | { ϕ } |

**Step (3) : To find predecessor :**

| Block | Predecessor |
|---|---|
| $B_1$ | ϕ |
| $B_2$ | $B_1$ |
| $B_3$ | $B_1$ |
| $B_4$ | $B_3$, $B_2$ |

**Step (4) :**

**Compute IN and OUT in iteration until IN and OUT are same :**

**Iteration (1) :**

| Block | IN | OUT |
|---|---|---|
| $B_1$ | ϕ | {1, 2} |
| $B_2$ | ϕ | {3, 4} |
| $B_3$ | ϕ | {5} |
| $B_4$ | ϕ | {6} |

**Iteration (2) :**

| Block | IN | OUT |
|---|---|---|
| $B_1$ | { ϕ } | {1, 2} |
| $B_2$ | {1, 2} | {3, 4} |
| $B_3$ | {3, 4} | {3, 4, 5} |
| $B_4$ | {3, 4, 5} | {3, 4, 5, 6} |

**Iteration (3) :**

| Block | IN | OUT |
|---|---|---|
| $B_1$ | ϕ | {1, 2} |
| $B_2$ | {1, 2} | {3, 4} |
| $B_3$ | {3, 4} | {3, 4, 5} |
| $B_4$ | {3, 4, 5} | {3, 4, 5, 6} |

As OUT of iteration (2) and (3) are same. So stop iteration.

{

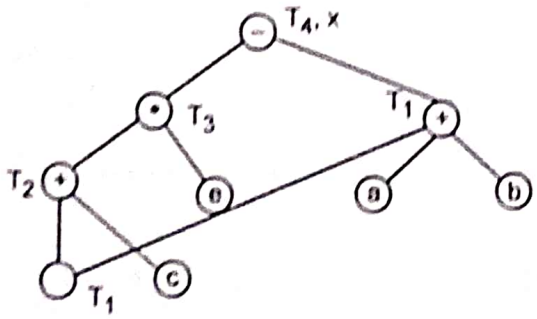
Store all the results

}

**Step 3 : Apply this algorithm for above example :**

| Statement | I. | Instruction generated | Register descriptor | Address descriptor |
|---|---|---|---|---|
| $T_1 = a + b$ | $R_0$ | MOV a, $R_0$<br>ADD b, $R_0$ | $R_0$ hold $T_1$ | $T_1$ is in $R_0$ |
| $T_2 = T_1 + c$ | $R_1$ | MOV C, $R_1$<br>ADD $T_1$, $R_1$ | $R_1$ hold $T_2$ | $T_2$ is in $R_1$ |
| $T_3 = e * T_2$ | $R_0$ | MOV $R_1$, $T_1$<br>MOV e, $R_0$<br>MUL $R_1$, $R_0$ | $R_1$ hold $T_3$ | $T_3$ is in $R_0$ |
| $T_4 = T_1 - T_3$<br>$x = T_1 - T_3$ | $R_1$ | MOV $T_3$, $R_0$<br>SUB $R_0$, $R_1$ | $R_0$ hold value of x | X is in $R_1$ |

**Step 4 :** The DAG can be constructed as follows :




**Step 5 : Optimize code will be :**

100 $T_1 = a + b$

101 $T_2 = T_1 + c$

102 $T_3 = e * T_2$

103 $x = T_1 - T_3$

## REGISTER ALLOCATION AND ASSIGNMENT

**Q.11. Write short note on register allocation and assignment.**

**Ans. Register allocation and assignment :**

- Instructions involving only register operands are shorter and faster than those involving memory operands.
- Therefore, efficient utilization of register is important in generating good code.
- One approach to register allocation and assignment is to assign specific values in an object program to certain registers.
- For example, a decision can be made to assign base addresses to one group of registers, arithmetic computation to another, the top of the run-time stack to a fixed register, and so on.
- This approach has the advantage that it simplifies the design of a compiler. Its disadvantage is that, applied too strictly it uses register inefficiently; certain registers may go unused over substantial portions of code, while unnecessary loads and stores are generated.
- Nevertheless, it is reasonable in most computing environments to reserve a few registers for base registers, stack pointers and the like, and to allow the remaining registers to be used by the compiler as it sees fit.

**Q.12. Write short note on global register allocation.**

**Ans. Global register allocation :**

- The code-generation algorithm uses registers to hold values for the duration of a single basic block.
- However, all live variables were stored at the end of each block.
- To save some of these stores and corresponding loads, we might arrange to assign registers to frequently used variables and keep these registers consistent across block boundaries (globally).
- Since programs spend most of their time in inner loops, a natural approach to global register assignment is to try to keep a frequently used value in a fixed register throughout a loop.
- For the time being assume that we know the loop structure of a flow graph, and that we know what values computed in a basic block are used outside that block.
- One strategy for global register allocation is to assign some fixed number of registers to hold the most active values in each inner loop. The selected values may be different in different loops.
- Registers not already allocated may be used to hold values local to one block.
- This approach has the drawback that the fixed number of registers

not always the right number to make available for global register allocation.

Yet the method is simple to implement and it has been used in Fortran H, the optimizing Fortran computer compiler for the IBM 360 series machines.

In languages like C and Bliss a programmer can do some register allocation directly by using register declarations to keep certain values in registers for the duration of procedure.

Judicious use of register declarations can speed up many programs, but a programmer should not do register allocation without first profiling the program.

Q.13. **Explain how usage counts is used in register allocation.**

Ans. **Usage counts :**

A simple method for determining the savings to be realized by keeping variable x in a register for the duration of loop L is to recognize that in our machine model we save one unit of cost for each reference to x if x is in a register.

However, if we use the approach of code generation algorithm to generate code for a block, there is a good chance that after x has been computed in a block it will remain in a register if there are subsequent uses of x in that block.

Thus we count a savings of one for each use of x in loop L that is not preceded by an assignment to x in the same block.

We also save two units if we can avoid a store of x at the end of a block.

Thus if x is allocated a register, count a savings of two for each block of L for which x is live on exit and in which x is assigned a value.

On the debit side, if x is live on entry to the loop header, we must load x into its register just before entering loop L. This load costs two units.

Similarly, for each exit block B of loop L at which x is live on entry to some successor of B outside of L, we must store x at a cost of two.

However, on the assumption that the loop is iterated many times, we may neglect these debits since they occur only once each time we enter the loop.

- Thus an approximate formula for the benefit to be realized from allocating a register to x within loop L is :

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

where use (x, B) is the number of times x is used in B prior to any definition of x, live (x, B) is 1 if x is live on exit from B and is assigned a value in B and live (x, B) is 0 otherwise.

- Note that equation is approximate because not all blocks in a loop are executed with equal frequency and also because equation was based on the assumption that a loop is iterated "many" times.
- On other machines a formula analogous to equation, but possibly quite different from it, would have to be developed.

Q.14. **Explain the method of graph coloring for register allocation.**

Ans. **Register allocation by Graph Coloring :**

- When a register is needed for a computation but all available registers are in use, the contents of one of the used registers must be stored (spilled) into a memory location in order to free up a register.
- Graph coloring is a simple systematic technique for allocating registers and managing register spills.
- In this method, two passes are used. In the first, target-machine instructions are selected as though there were an infinite number of symbolic registers, in effect, names used in the intermediate code become names of registers and the three-address statements become machine-language statements.
- If access to variables requires instructions that use stack pointers, display pointers, base registers or other quantities that assist access, then we assume that these quantities are held in registers reserved for each purpose.
- Normally, their use is directly translatable into an access mode for an address mentioned in a machine instruction.
- If access is more complex, the access must be broken into several machine instructions and a temporary symbolic register (or several) may need to be created.
- Once the instructions have been selected a second pass assigns physical registers to symbolic ones.

- The goal is to find an assignment that minimizes the cost of the spills.
- In the second pass, for each procedure a register-interference graph is constructed in which the nodes are symbolic registers and an edge connects two nodes if one is live at a point where the other is defined.
- For example a register-interference graph for figure would have nodes for names a and d. In block $B_1$, a is live at the second statement, which defines d, therefore in the graph there would be an edge between the nodes for a and d.

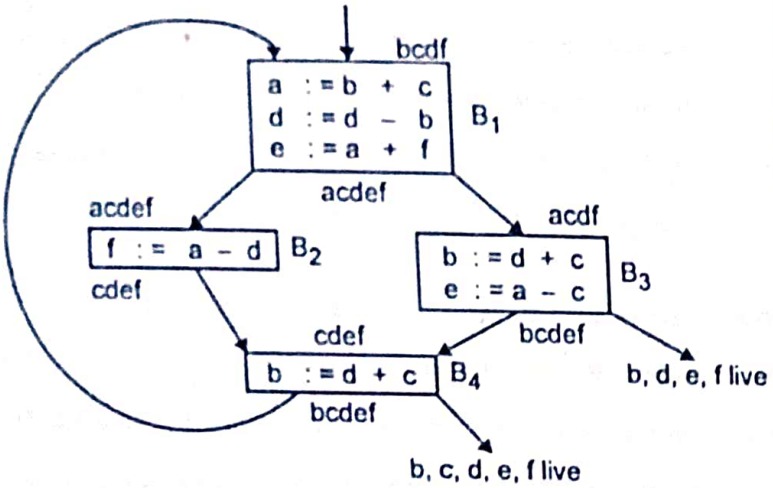



**Fig. Flow graph of an inner loop.**

- An attempt is made to color the register-interference graph using k colors, where k is the number of assignable register.
- A graph is said to be colored if each node has been assigned a color in such a way that no two adjacent nodes have the same color.
- A color represents a register and the coloring makes sure that no two symbolic registers that can interfere with each other are assigned the same physical register.
- Although the problem of determining whether a graph is k-colorable is NP-complete in general, the following heuristic technique can usually be used to do the coloring quickly in practice.
- Suppose a node n in a graph G has fewer than k neighbors (nodes connected to n by an edge). Remove n and its edges from G to obtain a graph G'.
- A k-coloring of G' can be extended to a k-coloring of G by assigning n a color not assigned to any of its neighbors.
- By repeatedly eliminating nodes having fewer than k edges from the register-interference graph, either we obtain the empty graph, in which case we can produce a k-coloring for the original graph by coloring the nodes in th reverse order in which they were removed or we obtain a graph in which each node has k or more adjacent nodes.
- In the latter case a k-coloring is no longer possible.
- At this point a node is spilled by introducing code to store and reload the register. Then the interference graph is appropriately modified and the coloring process resumed.

**Q.15. What is DAG? What are applications of DAG?**

CS : S-10, W-11(4M), S-11(6M)
CT : W-12(6M), S-13(2M)

**OR Write short note on DAG.** CT : W-11(3M)

**OR Explain DAG with suitable example.** CS:W-12(3M)

**Ans. DAG :**

- DAG (Directed acyclic graph) is a directed graph with no cycle which gives a picture of how the value is computed by each statement in a basic block is used in subsequent statement in the block.
- The optimization of basic block is done by using DAG.
- A DAG is constructed from three address statements which determine common sub expression within a block.

**Construction :**

- To construct a DAG we look at each statement in the block one by one.
- So if A = B + C occurrence look for the nodes which represents current value of B and C.
- This will be leaves or interior nodes of DAG.
- Then create a node labeled '+' and give child B and C. Label the node by A.
- If there is already a node present computing value B + C, then there is no need to create a node but just give an existing node an additional label 'A'.

**Example :**

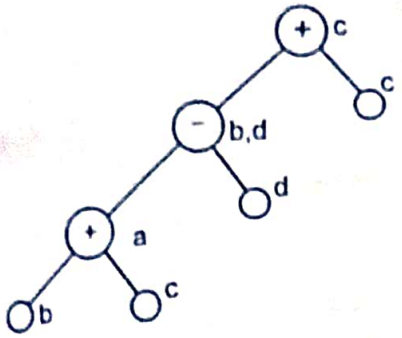
a = b + c

b = a – d

c = b + c

d = a – d

The DAG for above block will be




- In above DAG, we have to create the node for the third statement c = b + c we can use the node a since the value of b is changing in second statement.
- The node corresponding to the fourth statement d = a – d has the operator '–' and nodes labelled d and c as children
- Since the operator and children are same as those for the node corresponding to statement two, thus we add d to the same node b, instead of creating separate node

**Application of DAG :**

(1) Determination of which identifier has their values used in block.

(2) Automatic detection of common sub expression.

(3) Deconstruction of simplified list of quadruple taking advantages of common sub expression and not performing assignment of form A = B unless it is necessary, x := y unless absolutely unnecessary.

- Determination of statement which compute value that could be used outside the block.

**Q.16. Write and explain the algorithm for constructing a DAG.**

**Ans. Algorithm :** Constructing a dag.

**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 here).

**Method :**

- We assume the appropriate data structures are available to create nodes with one or two children, with a distinction between "left" and "right" children in the latter case.
- Also available in the structure is a place for a label for each node and the facility to create a linked list of attached identifiers for each node.
- In addition to these components, we need to maintain the set of all identifiers (including constants) for which there is a node associated.
- The node could be either a leaf labeled by that identifier or an interior node with that identifier on its attached identifier list.
- We assume the existence of a function node (identifier), which, as we build the dag, returns the most recently created node associated with identifier.
- Intuitively, node (identifier) is the node of the dag that represents the value that identifier has at the current point in the dag construction process.
- In practice, an entry in the symbol-table record for identifier would indicate the value of node (identifier).
- 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) x : = y op z, (ii) x : = op y, or (iii) x : = y.
- We treat a relational operator like if I <= 20 goto as case (i), with x undefined.

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

(2) In case (i), determine if there is a node labeled op, whose left child is node (y) and whose right child is node (z). (This check is to catch common subexpressions.) If not, create such a node. In either event, let n be the node found or created. In case (ii), determine whether there is a node labeled op, whose lone child is node (y). If not, create such a node, and let n be the node found or created. In case (iii), let n be node (u).

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

**Q.17. Explain how DAG is useful in generation of efficient code.**

**CT : S-14(3M)**

**Ans.**

- A good code generator is an algorithm which first obtains better

order of computation if one exists and then generates a code by going in that order.

- To find an order of computation, a DAG representation of basic block is used.
- If a DAG representation of basic block does not come out to be a tree, we use an algorithm based upon the heuristic used for ordering the nodes of the DAG to obtain a better order of computation if one exist, whereas when a DAG representation comes out to be a tree, we use a labelling algorithm to label the nodes of DAG and then using these labels we traverse the tree and get the code generated during the traversal itself which takes into account an order of computation.
- In this case, a shortest instruction sequence is generated, whereas when we use heuristic for ordering the nodes of the DAG, there is no guarantee that it will always generate optional object.

**Q.18. Construct DAG for following three address code.**

$S_1 = a + b$

$S_2 = c * d$

$S_3 = e + S_2$

$S_4 = S_1 + S_3$

CS : W-11 (4M)

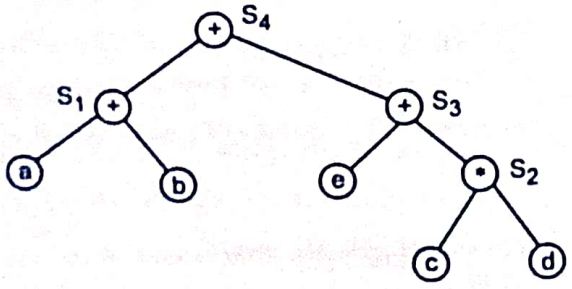
**Ans.** $S_1 = a + b$

$S_2 = c * d$

$S_3 = e + S_2$

$S_4 = S_1 + S_3$

**DAG for above code will be,**




**Q.19. Construct DAG for following code :**

A[I] = B

* P = C

D = A [J]

E = *P

*P = A[I]

CS : W-10, S-12 (7M)

**Ans. TAC is as follows :**

100 $T_1 = I * 4$

101 $T_2 = \text{addr}(A) - 4$

102 $T_3 = T_2 [T_1]$

103 $T_3 = B$

104 $T_4 = *P$

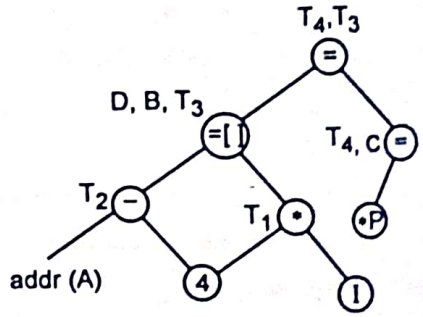
105 $T_4 = C$

106 $0 = T_3$

107 $E = T_4$

108 $T_4 = T_3$

**The DAG for above code will be,**




**Q.20. Construct DAG for the following expression and determine the minimum registers required :**

$Z = (A + B) * (C - D) / E + D$

CT : S-12 (4M)

**Ans. Step 1 : TAC is as follows :**

100 $T_1 = C - D$

101 $T_2 = E + D$

102 $T_3 = T_1 / T_2$

103 $T_4 = A * B$

104 $T_5 = T_4 * T_3$

105 $Z = T_5$

Step 2 : The optimal code :

100 $T_1 = C - D$

101 $T_2 = E + D$

102 $T_3 = T_1 / T_2$

103 $T_4 = A * B$

$Z = T_4 * T_3$

Step 3 : Now generate the no. of register required :

| Statement | L | Instruction generated | Cost | Register descriptor | Address descriptor |
|---|---|---|---|---|---|
| | | | $R_0$ and $R_1$ are empty | | |
| $T_1 = C - D$ | $R_0$ | MOV C, $R_0$<br>SUB $R_0$, D | 2 word<br>2 word | $R_0$ will hold $T_1$ | $T_1$ is in $R_0$ |
| $T_2 = C + D$ | $R_1$ | MOV E, $R_0$<br>ADD D, $R_0$ | 2 word<br>2 word | $R_1$ will hold $T_2$ | $T_2$ is in $R_1$ |
| $T_3 = T_1 / T_2$ | $R_0$ | MOV $R_0$, $T_1$<br>DIV $R_0$, $R_1$ | 2 word<br>2 word | $R_0$ will hold $T_3$ | $T_3$ is in $R_0$ |
| $T_4 = A * B$ | $R_1$ | MOV A, $R_0$<br>MUL $R_1$, B | 2 word<br>1 word | $R_1$ will hold $T_4$ | $T_4$ is in $R_1$ |
| $Z = T_4 * T_3$ | $R_1$ | MOV $T_1$, $R_0$<br>MUL $R_0$, $R_1$ | 2 word<br>1 word | $R_1$ will hold x | x is in $R_1$ |
| | | MOV $R_1$, x | 2 words | | z is in $R_1$ & memory |

Total cost of memory instruction will be 20 memory words.

Q.21. Construct DAG for following TAC :

(i) a = b + c
b = a – d
c = b + c
d = a – d

(ii) a = b + c
b = b – d
c = c + d
d = b + c



Ans. (i) TAC is as follows :

100 $t_1 = b + c$

101 $a = t_1$

102 $t_2 = a + d$

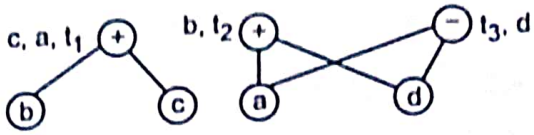
103 $b = t_2$

104 $c = t_1$

105 $t_3 = a - d$

106 $d = t_3$

DAG is as follows :




(ii) TAC is as follows :

100 $t_1 = b + c$

101 $a = t_1$

102 $t_2 = a + d$

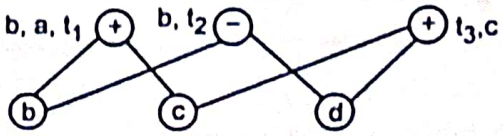
103 $b = t_2$

104 $t_3 = c + d$

105 $c = t_3$

106 $d = t_1$

DAG is as follows :




Q.22. Construct DAG for the following TAC :

$S_1 = h * 1$

$S_2 = adder (A) - 4$

$S_3 = S_2 (S_1)$

$S_4 = h * 1$

$S_5 = adder (B) - 4$

$S_6 = S_5 (S_4)$

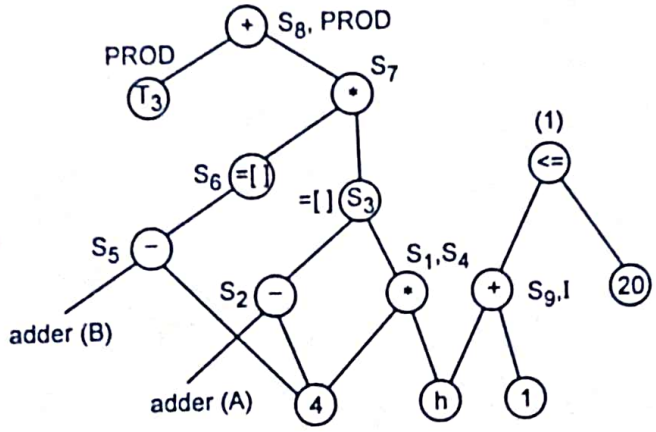
$S_7 = S_3 * S_6$

$S_8 = PROD + S_7$

$PROD = S_8$

$I = S_9$ **CS:W-14 (7M)**

**Ans. DAG can be computed as follows :**




## CODE GENERATION FROM DAG

**Q.23. Write heuristic algorithm to find the optimal order of computation.** **CS:W-09(3M), W-13(7M), CT : S-11,13(6M)**

**OR Explain heuristic algorithm for ordering nodes of DAG.** **CT : S-10(6M), S-14, W-10(4M)**

**Ans. Heuristic algorithm for ordering nodes of DAG :**

- The algorithm for heuristic ordering is given below. It lists the nodes of a DAG such that the nodes reverse listing results in the computation order.

```
{
 while there exists an unlisted interior node do
 {
 select an unlisted node n whose parents have been listed.
 list n
 while there exists a leftmost child m of n that has no unlisted
parents and m is not a leaf do.
 {
 list m
 m = n
 }
}
order = reverse of the order of listing of nodes
}
```

**Example :** Consider the DAG shown in figure below

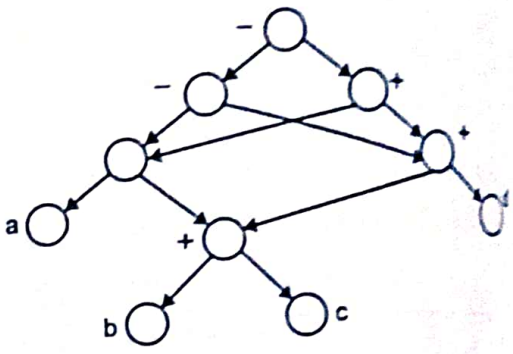



**Fig.(a) DAG Representation**

- The order in which the nodes are listed by the heuristic ... shown in figure below.

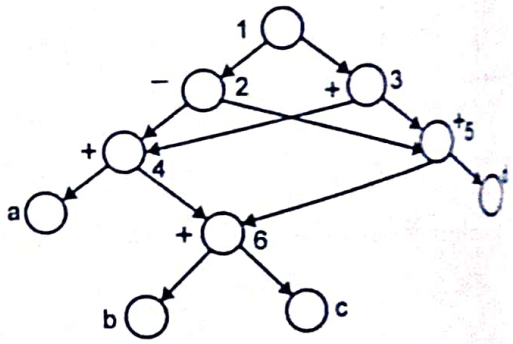



**Fig.(b) DAG Representation using heuristic ordering**

- So the computation order is :

$t_6 = b + c$

$t_5 = t_6 + d$

$t_4 = a + t_6$

$t_3 = t_4 + t_5$

$t_2 = t_4 - t_5$

$t_1 = t_2 - t_3$

- If DAG representation turns out to be a tree, then for each described above, can obtain optimal order using the algo...

**Q.24. State whether the order of computation affects the generated code. If yes, then find an optimal computation for the following DAG.**