UNIT - V

UNIT-V
CODE OPTIMIZATION

1. INTRODUCTION

- The code produced by the straight forward compiling algorithms can often be made to run faster or take less space, or both. This improvement is achieved by program transformations that are traditionally called optimizations. Compilers that apply code-improving transformations are called optimizing compilers.

- Optimizations are classified into two categories. They are

- Machine independent optimizations:

- Machine dependant optimizations:

1.1 Machine independent optimizations:

Machine independent optimizations are program transformations that improve the target code without taking into consideration any properties of the target machine.

1.2 Machine dependant optimizations:

Machine dependant optimizations are based on register allocation and utilization of special machine- instruction sequences.

1.3 The criteria for code improvement transformations:

- Simply stated, the best program transformations are those that yield the most benefit for the least effort.

- The transformation must preserve the meaning of programs. That is, the optimization must not change the output produced by a program for a given input, or cause an error such as division by zero, that was not present in the original source program. At all times we take the "safe" approach of missing an opportunity to apply a transformation rather than risk changing what the program does.

- A transformation must, on the average, speed up programs by a measurable amount. We are also interested in reducing the size of the compiled code although the size of the code has less importance than it once had. Not every transformation succeeds in improving every program, occasionally an "optimization" may slow down a program slightly.

- The transformation must be worth the effort. It does not make sense for a compiler writer

to expend the intellectual effort to implement a code improving transformation and to have the compiler expend the additional time compiling source programs if this effort is not repaid when the target programs are executed. “Peephole” transformations of this kind are simple enough and beneficial enough to be included in any compiler.

- Flow analysis is a fundamental prerequisite for many important types of code improvement.
- Generally control flow analysis precedes data flow analysis.
- Control flow analysis (CFA) represents flow of control usually in form of graphs, CFA constructs such as
- A transformation of a program is called local if it can be performed by looking only at the statements in a basic block; otherwise, it is called global.

- Many transformations can be performed at both the local and global levels. Local transformations are usually performed first.

2.1 Function-Preserving Transformations

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

- The transformations

    - Common sub expression elimination, o Copy propagation,
    - Dead-code elimination, and
    - Constant folding, are common examples of such function-preserving transformations. The other transformations come up primarily when global optimizations are performed.
- Frequently, a program will include several calculations of the same value, such as an offset in an array. Some of the duplicate calculations cannot be avoided by the programmer because they lie below the level of detail accessible within the source language.

2.2 Common Sub expressions 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.

- For example

```
t1: =4*i t2: =a [t1] t3: =4*j t4:=4*i t5: =n
t 6: =b [t 4] +t 5
```

The above code can be optimized using the common sub-expression elimination as t1: =4*i

```
t2: =a [t1] t3: =4*j t5: =n
 t6: =b [t1] +t5
```

The common sub expression t 4: =4*i is eliminated as its computation is already in t1. And value of i is not been changed from definition to use.

2.3 Copy Propagation:

Assignments of the form f : = g called copy statements, or copies for short. The idea behind the copy-propagation transformation is to use g for f, whenever possible after the copy statement f: = g. Copy propagation means use of one variable instead of another. This may not appear to be an improvement, but as we shall see it gives us an opportunity to eliminate x.

For example: x=Pi;

```
……
A=x*r*r;
```

The optimization using copy propagation can be done as follows:

```
A=Pi*r*r;
```

Here the variable x is eliminated

2.4 Dead-Code Eliminations:

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. An optimization can be done by eliminating dead code.

Example:

```
i=0;
if(i=1)
{
 a=b+5;
}
```

Here, 'if' statement is dead code because this condition will never get satisfied.

2.5 Constant folding:

- We can eliminate both the test and printing from the object code. More generally, deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding.

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

For example,

a=3.14157/2 can be replaced by

a=1.570 there by eliminating a division operation.

2.6 Loop Optimizations:

- We now give a brief introduction to a very important place for optimizations, namely loops, especially the inner loops where programs tend to spend the bulk of their time. The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop.

- Three techniques are important for loop optimization:
  - code motion, which moves code outside a loop;
  - Induction -variable elimination, which we apply to replace variables from inner loop.
  - Reduction in strength, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.

2.7 Code Motion:

- An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and places the expression before the loop. 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 */
```

2.8 Induction Variables :

- Loops are usually processed inside out. For example consider the loop around B3.
- Note that the values of j and t4 remain in lock-step; every time the value of j decreases by 1, that of t4 decreases by 4 because 4*j is assigned to t4. Such identifiers are called induction variables.

- When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction-variable elimination. For the inner loop around B3 in Fig. we cannot get rid of either j or t4 completely; t4 is used in B3 and j in B4.
- However, we can illustrate reduction in strength and illustrate a part of the process of induction-variable elimination. Eventually j will be eliminated when the outer loop of B2 - B5 is considered.

Example:

As the relationship t 4:=4*j surely holds after such an assignment to t 4 in Fig. and t4 is not changed elsewhere in the inner loop around B3, it follows that just after the statement j:=j -1 the relationship t4:= 4*j-4 must hold. We may therefore replace the assignment t 4:= 4*j by t4:= t4-4. The only problem is that t 4 does not have a value when we enter block B3 for the first time. Since we must maintain the relationship t4=4*j on entry to the block B3, we place an initializations of t4 at the end of the block where j itself is initialized, shown by the dashed addition to block B1 in second Fig.

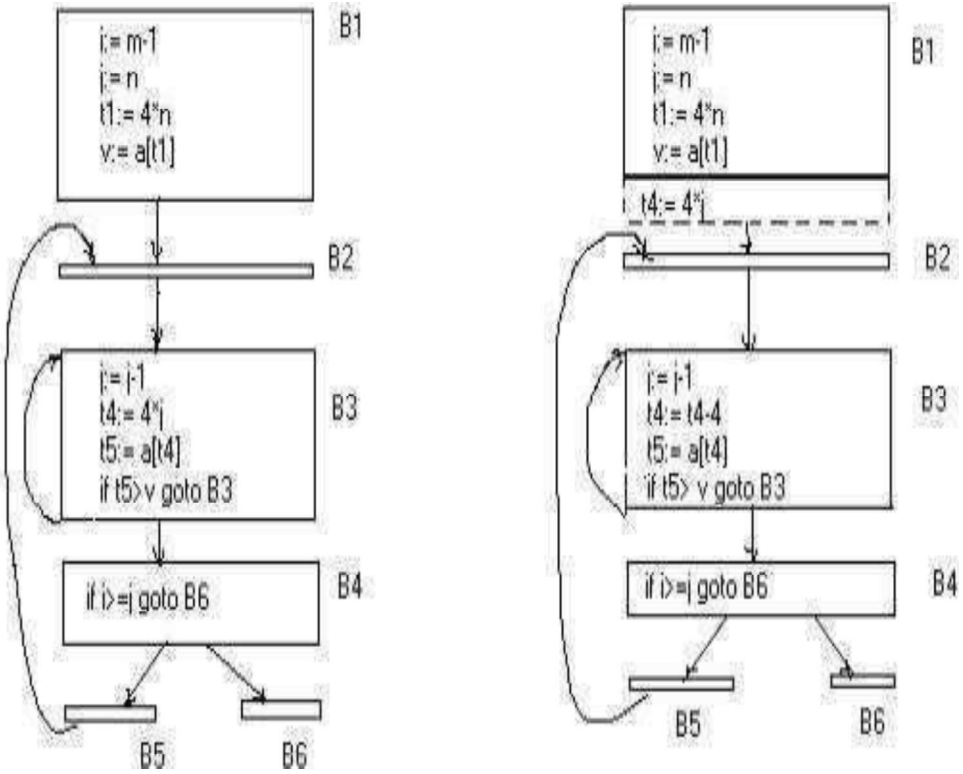
The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction, as is the case on many machines.

2.9 Reduction in Strength:

- Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.

- For example, x^2 is invariably cheaper to implement as x*x than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

3. OPTIMIZATION OF BASIC BLOCKS



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

Structure -Preserving Transformations
Algebraic Transformations

3.1 Structure- Preserving Transformations:

The primary Structure-Preserving Transformation on basic blocks are:

- Common sub-expression elimination
- Dead code elimination
- Renaming of temporary variables
- Interchange of two independent adjacent statements.

3.2 Common sub-expression elimination:

Common sub expressions need not be computed over and over again. Instead they can be computed once and kept in store from where it's referenced when encountered again – of course providing the variable values in the expression still remain constant.

Example:

a: =b+c
b: =a-d
c: =b+c
d: =a-d

The 2^{nd} and 4^{th} statements compute the same expression: b+c and a-d

Basic block can be transformed to

a: =b+c
b: =a-d
c: =a
d: =b

3.3 Dead code elimination:

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

3.4 Renaming of temporary variables:

- A statement t:=b+c where t is a temporary name can be changed to u:=b+c where u is another temporary name, and change all uses of t to u.

- In this we can transform a basic block to its equivalent block called normal-form block.

3.5 Interchange of two independent adjacent statements:

Two statements

t1:=b+c

t2:=x+y

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

3.6 Algebraic Transformations:

- Algebraic identities represent another important class of optimizations on basic blocks. This includes simplifying expressions or replacing expensive operation by cheaper ones i.e. reduction in strength.

- Another class of related optimizations is constant folding. Here we evaluate constant expressions at compile time and replace the constant expressions by their values. Thus the expression 2*3.14 would be replaced by 6.28.

- The relational operators <=, >=, <, >, + and = sometimes generate unexpected common sub expressions.

- Associative laws may also be applied to expose common sub expressions. For example, if the source code has the assignments

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

  the following intermediate code may be generated:

  a :=b+c t :=c+d

  e :=t+b

Example:

x:=x+0 can be removed

x:=y**2 can be replaced by a cheaper statement x:=y*y

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

4.LOOPS IN FLOW GRAPH

A graph representation of three-address statements, called a **flow graph**, is useful for understanding code-generation algorithms, even if the graph is not explicitly constructed by a code-generation algorithm. Nodes in the flow graph represent computations, and the edges represent the flow of control.

4.1 Dominators:

In a flow graph, a node d dominates node n, if every path from initial node of the flow graph to n goes through d. This will be denoted by *d dom n*. Every initial node dominates all the remaining nodes in the flow graph and the entry of a loop dominates all nodes in the loop. Similarlyeverynode dominates itself.

Example:

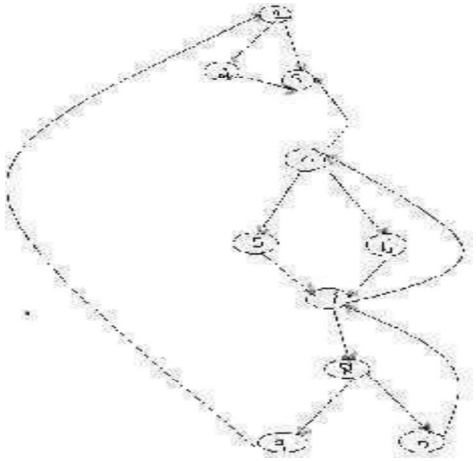
*In the flow graph below,

*Initial node,node1 dominates every node. *node 2 dominates itself *node 3 dominates all but 1 and 2. *node 4 dominates all but 1,2 and 3.

*node 5 and 6 dominates only themselves,since flow of control can skip around either by goin through the other.

*node 7 dominates 7,8 ,9 and 10. *node 8 dominates 8,9 and 10.

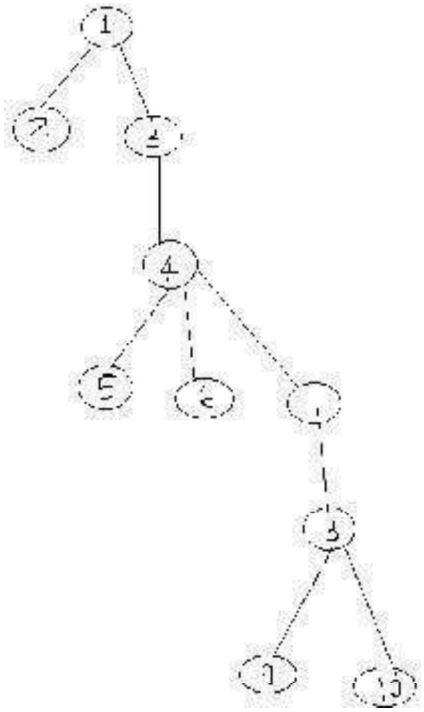
*node 9 and 10 dominates only themselves.



- The way of presenting dominator information is in a tree, called the dominator tree in which the initial node is the root.

- The parent of each other node is its immediate dominator.
- Each node d dominates only its descendents in the tree.
- The existence of dominator tree follows from a property of dominators; each node has a unique immediate dominator in that is the last dominator of n on any path from the initial node to n.

- In terms of the dom relation, the immediate dominator m has the property is d=!n and d dom n, then d dom m.



D(1)={1}

D(2)={1,2}

D(3)={1,3}

D(4)={1,3,4}

D(5)={1,3,4,5}

D(6)={1,3,4,6}

D(7)={1,3,4,7}

D(8)={1,3,4,7,8}

D(9)={1,3,4,7,8,9}

D(10)={1,3,4,7,8,10}

4.2Natural Loop:

- One application of dominator information is in determining the loops of a flow graph suitable for improvement.

- The properties of loops are

    - A loop must have a single entry point, called the header. This entry point-dominates all nodes in the loop, or it would not be the sole entry to the loop.
    - There must be at least one wayto iterate the loop(i.e.)at least one path back to the header.

- One way to find all the loops in a flow graph is to search for edges in the flow graph whose heads dominate their tails. If a→b is an edge, b is the head and a is the tail. These types of edges are called as back edges.

Example:

In the above graph,

→ 4 4 DOM 7

→7 7 DOM 10

→ 3

→ 3

9 →1

- The above edges will form loop in flow graph.
- Given a back edge n → d, we define the natural loop of the edge to be d plus the set of nodes that can reach n without going through d. Node d is the header of the loop.

Algorithm: Constructing the natural loop of a back edge.
Input: A flow graph G and a back edge n→d

Output: The set loop consisting of all nodes in the natural loop n→d.

Method: Beginning with node n, we consider each node m*d that we know is in loop, to make sure that m's predecessors are also placed in loop. Each node in loop, except for d, is placed once on stack, so its predecessors will be examined. Note that because d is put in the loop initially, we never examine its predecessors, and thus find only those nodes that reach n without going through d.

Procedure insert(m);
if m is not in *loop* **then begin** *loop* := *loop* U {m}; push *m* onto *stack*
end;
stack : =empty; *loop* : ={*d*}; *insert*(*n*);

while *stack* is not empty **do begin**

pop *m*, the first element of *stack*, off *stack*; **for** each predecessor *p* of *m* **do** *insert(p)*

end Inner

5.LOOP:

- If we use the natural loops as "the loops", then we have the useful property that unless two loops have the same header, they are either disjointed or one is entirely contained in the other. Thus, neglecting loops with the same header for the moment, we have a natural notion of inner loop: one that contains no other loop.

- When two natural loops have the same header, but neither is nested within the other, they are combined and treated as a single loop.

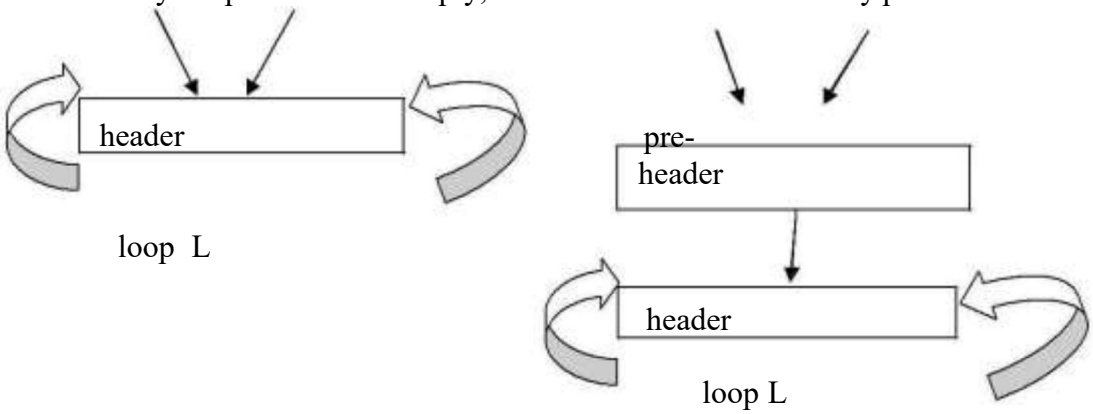
5.1 Pre-Headers:

- Several transformations require us to move statements "before the header". Therefore begin treatment of a loop L by creating a new block, called the preheater.

- The pre -header has only the header as successor, and all edges which formerly entered the header of Lfrom outside L instead enter the pre-header.

- Edges from inside loop L to the header are not changed.

- Initially the pre-header is empty, but transformations on L may place statements in it.



(a) Before **(b) After**

5.2 Reducible flow graphs:

- Reducible flow graphs are special flow graphs, for which several code optimization transformations are especially easy to perform, loops are unambiguously defined, dominators can be easily calculated, data flow analysis problems can also be solved efficiently.

- Exclusive use of structured flow-of-control statements such as if-then-else, while-do, continue, and break statements produces programs whose flow graphs are always reducible. The most important properties of reducible flow graphs are that there are no jumps into the middle of loops from outside; the only entry to a loop is through its header.

- ***Definition***:

- A flow graph G is reducible if and only if we can partition the edges into two disjoint groups, *forward* edges and *back* edges, with the following properties.

- The forward edges from an acyclic graph in which every node can be reached from initial node of G.

- The back edges consist only of edges where heads dominate theirs tails.

- Example: The above flow graph is reducible.

- If we know the relation DOM for a flow graph, we can find and remove all the back edges.

- The remaining edges are forward edges.
- If the forward edges form an acyclic graph, then we can say the flow graph reducible.
- In the above example remove the five back edges 4→3, 7→4, 8→3, 9→1 and 10→7 whose heads dominate their tails, the remaining graph is acyclic.
- The key property of reducible flow graphs for loop analysis is that in such flow graphs every set of nodes that we would informally regard as a loop must contain a back edge.

CODE GENERATION

PEEPHOLE OPTIMIZATION

- A statement-by-statement code-generations strategy often produce target code that contains redundant instructions and suboptimal constructs .The quality of such target code can be improved by applying "optimizing" transformations to the target program.

- A simple but effective technique for improving the target code is peephole optimization, a method for trying to improving the performance of the target program by examining a short sequence of target instructions (called the peephole) and replacing these instructions by a shorter or faster sequence, whenever possible.

- The peephole is a small, moving window on the target program. The code in the peephole need not contiguous, although some implementations do require this.it is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements.

- We shall give the following examples of program transformations that are characteristic of peephole optimizations:

  - Redundant-instructions elimination
  - Flow-of-control optimizations
  - Algebraic simplifications
  - Use of machine idioms
  - Unreachable Code

Redundant Loads And Stores:

If we see the instructions sequence

(1) MOV R0,a
(2) MOV a,R0

we can delete instructions (2) because whenever (2) is executed. (1) will ensure that the value of **a** is already in register R0.If (2) had a label we could not be sure that (1) was always executed immediately before (2) and so we could not remove (2).

Unreachable Code:

- Another opportunity for peephole optimizations is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions. For example, for debugging purposes, a large program may have within it certain segments that are executed only if a variable **debug** is 1. In C, the source code might look like:

```
#define debug
0 ….
If ( debug ) {

Print  debugging information

}
```

In the intermediate representations the if-statement may be translated as:

```
debug =1 goto L2

goto  L2

L1: print debugging information

L2: ..... (a)
```

- One obvious peephole optimization is to eliminate jumps over jumps .Thus no matter what the value of **debug**; (a) can be replaced by:

```
If debug ≠1 goto L2

Print debugging information

L2: ..... (b)
```

- As the argument of the statement of (b) evaluates to a constant **true** it can be replaced by

```
If debug ≠0 goto L2

Print debugging information

L2: .....(c)
```

- As the argument of the first statement of (c) evaluates to a constant true, it can be replaced by goto L2. Then all the statement that print debugging aids are manifestly unreachable and can be eliminated one at a time.

Flows-Of-Control Optimizations:

- The unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations. We can replace the jump sequence

```
 goto L1
       ….
L1: gotoL2 by the sequence
  goto L2
     ….
L1: goto  L2
```

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

```
if a < b goto L1
     ….
L1: goto L2
can be replaced by Ifa < b goto L2
   ….
   L1: goto L2
```

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

```
 goto L1

 ……..
 L1:    if     a     <b     goto     L2

 L3: ..... (1)
```

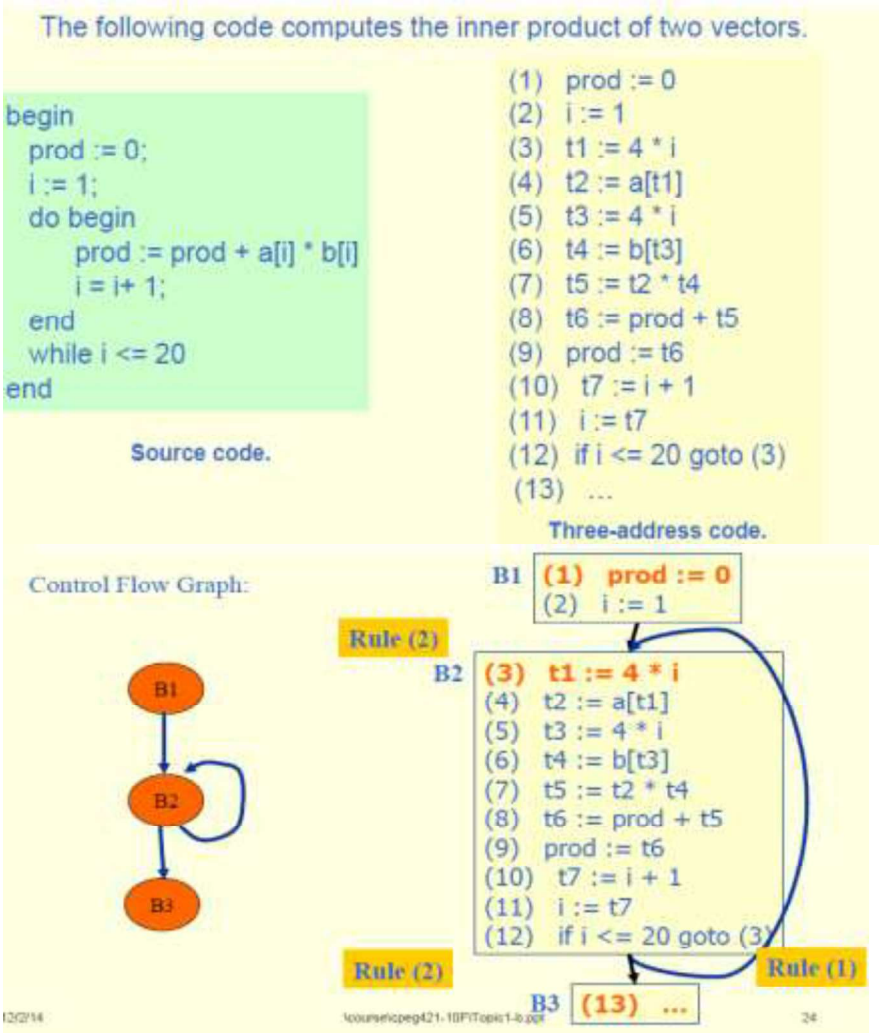
- Maybe replaced by Ifa<b goto L2

```
goto L3
 …….
```

Algorithm for partition of basic blocks

1. Find header statements of all the basic blocks from where a basic block starts using the following rules:
   - First statement of intermediate code is a leader.
   - Statements that are target of any branch statements are leaders.
   - Statements that follow any branch statement are leaders.
2. Header statements and the statements following them form a basic block.

Example:----------[3M]



Issues of code generation

1. Input to code generator
2. Target program
3. Memory management
4. Instruction selection
5. Register allocation
6. Choice of Evaluation order

Code generation algorithm

- Code generation algorithm takes a sequence of three address instructions as input and generates target code.
- An essential part of this algorithm is a function getReg(I)which selects registers for each memory location associated with the three address instruction l.
- This algorithm uses descriptors to keep track of registers addresses for variables.
- The register descriptor is used to keep track of registers where the variables are stored.
- The address descriptor keeps track of the locations where the current value of the variable can be found.

getReg() *works as follows:*

- If variable Y is already in register R, it uses that register.
- Else if some register R is available, it uses that register.
- Else if both the above options are not possible, We need to pick one of the allowable registers anyway, and we need to make it safe to reuse.

Example:

By using code generation algorithm, obtain the code for the following three address code.

T=A-B
U=A-C
V=T+U
D=V+U

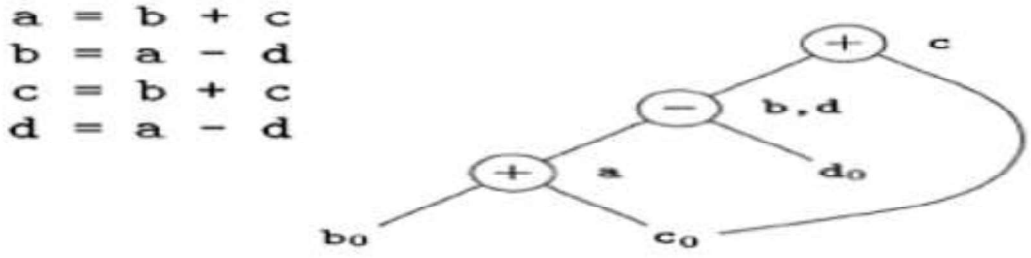
MOV A,R0
SUB B,R0
MOV A,R1
SUB C,R1
ADD R1,R0
ADD R1,R0

Directed Acyclic Graph

- DAG is a directed graph with no cycles.
- DAG is a data structure used to implement transformations on basic blocks.
- We can optimize a basic block by constructing a DAG for it.
- In DAG:

  Leaf nodes represent identifiers i.e., names or constants.

  Interior nodes represent operators



Register allocation

- A Key problem in code generation is deciding what values to hold in what registers.
- Use of registers make the computations faster in comparison to that of memory, so efficient utilization of registers is important.
- The usage of registers is subdivided into two sub problems:

  Register allocation – select the set of variables that will reside in the registers at each point in the program.

  Register assignment- select a specific register that a variable reside in.

Example: register pairs for multiplication and division

```
t=a+b
t=t*c
T=t/d

L     R1, a
A     R1, b
M     R0, c
D     R0, d
ST    R1, t
```

```
t=a+b
t=t+c
T=t/d

L     R0, a
A     R0, b
M     R0, c
SRDA  R0, 32
D     R0, d
ST    R1, t
```