Example: x:=x+0 can be removed
x:=y2 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.

UNIT – V
Control flow &Data flow Analysis

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.

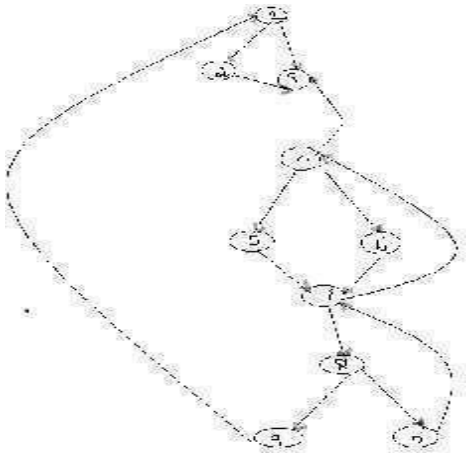
Dominators:

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.

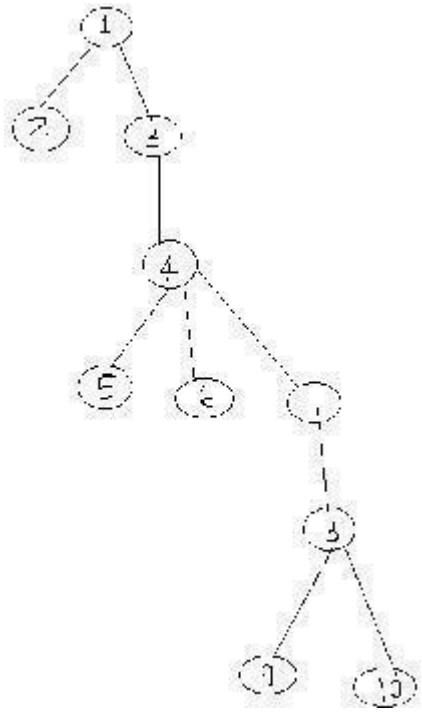
*node 9 and 10 dominates only themselves.



- 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 initialnode 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}

Natural 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

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.

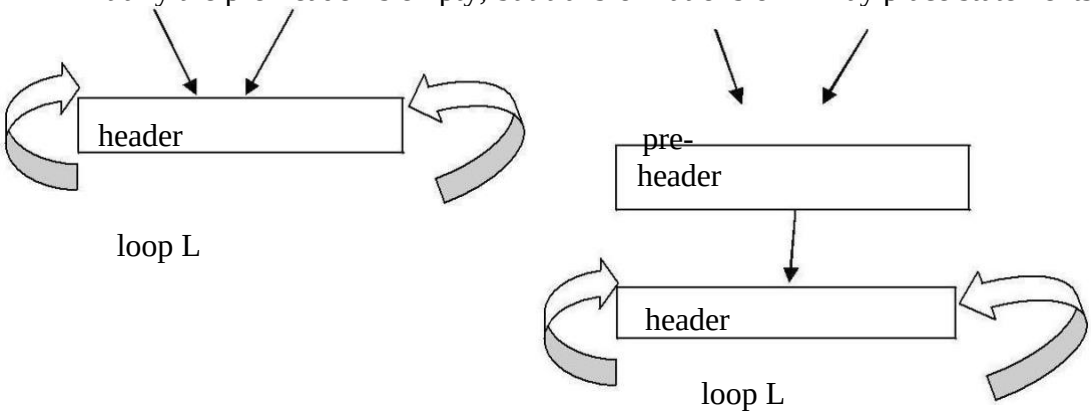
Pre-Headers:

- 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**

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.

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
  …….

OBJECT CODE GENERATION:

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

Code generation can be considered as the final phase of compilation. Through post code generation, optimization process can be applied on the code, but that can be seen as a part of code generation phase itself. The code generated by the compiler is an object code of some lower-level programming language, for example, assembly language. We have seen that the source code written in a higher-level language is transformed into a lower-level language that results in a lower-level object code, which should have the following minimum properties:

- It should carry the exact meaning of the source code.
- It should be efficient in terms of CPU usage and memory management.

We will now see how the intermediate code is transformed into target object code (assembly code, in this case).

Directed Acyclic Graph

Directed Acyclic Graph (DAG) is a tool that depicts the structure of basic blocks, helps to see the flow of values flowing among the basic blocks, and offers optimization too. DAG provides easy transformation on basic blocks. DAG can be understood here:

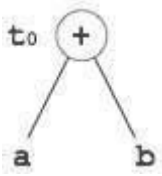
- Leaf nodes represent identifiers, names or constants.
- Interior nodes represent operators.
- Interior nodes also represent the results of expressions or the identifiers/name where the values are to be stored or assigned.

Example:

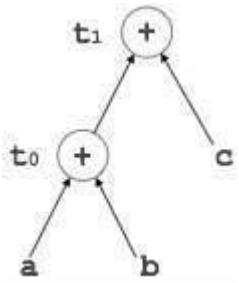
$t_0 = a + b$

$t_1 = t_0 + c$

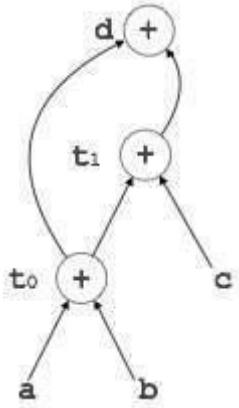
$d = t_0 + t_1$



[$t_0 = a + b$]



[$t_1 = t_0 + c$]



[$d = t_0 + t_1$]

REGISTER ALLOCATION:

Register allocation In the IR, we assumed an unlimited number of registers (to ease IR code generation) This is obviously not the case on a physical machine (typically, from 5 to 10 general- purpose registers) Registers can be accessed quickly and operations can be performed on them directly Using registers intelligently is therefore a critical step in any compiler (can make a di↵erencein orders of magnitude) Register allocation is the process of assigning variables to registers and managing data transfer in and out of the registers

GENERIC CODE GENERATION ALGORITHM:

Assume that for each operator in the statement, there is a corresponding target language operator
The computed results can be left in registers as long as possible, storing them only if the register is needed for another computation or just before a procedure call, jump or labeled statement

Register and Address Descriptors

These are the primary data structures used by the code generator. They keep track of what values are in each registers, as well as where a given value resides

- Each register has a register descriptor containing the list of variables currently stored in this register. At the start of the basic block, all register descriptors are empty. It keeps track ofrecent/current variable in each register. It is constructed whenever a new register is needed
- Each variable has an address descriptor containing the list of locations where this variable is currently stored. Possibilities are its memory location and one or more registers
- The memory location might be in the static area, the stack, r presumably the heap. The register descriptors can be computed from the address descriptors. For each name of the block, an address descriptors is maintained that keep track of location where current value of name is found at runtime

There are basically three aspects to be considered in code generation:

Choosing registers
Generating instructions
Managing descriptors

Register allocation is done in a function getReg(Instruction). The instruction generation algorithms

uses getReg() and the descriptors

The generation of machine instruction is done as follows:

Given a TAC, OP x, Y(i.e., x=x OP y), generation of machine instructions proceeds as follows:

Step 1: Call getReg(OP, x, y) to get R_x and R_y, the registers to be used for x and y respectively. GetReg merely selects the registers, it does not gurantee that the desired values are present in these registers.

Step 2: Check the register descriptor for R_y. If y is not present in Ry, check the address descriptor for y and issue LD R_y, y

Step 3: Similar treatment is done for R_x

Step 4: Generate the instruction OP R_x, R_y

When the TAC is x = y, step 1 and step 2 are same (getReg() will set Rx =Ry). Step 3 is empty and step 4 is omitted. If 'y' was already in a register before the copy instruction, no code is generated at this point. Since the value of 'y' is not in its memory location, we may need to store this value back into 'y' at block exit.

All variables needed by (dynamically) subsequent blocks (i.e., that live-on-exit) have their current values in their memory locations. Such live variables are identified as follows:

- Temporaries never live beyond a basic block. Hence, they are ignored
- Variables dead on exit are also ignored
- All live on exit variables need to be stored in their memory location on exit from the block. So, check the address descriptor for each live on exit variable. If its own memory location is not listed, generate ST X, R, where R is a register listed in the address descriptor

The management of register and address descriptor is performed as follows:

- For a register R, let Desc(R) be is register descriptor. For a program variable x, let Desc(x) be its address descriptor. The management of descriptor for load, store, operation and copy are given below

Load: LD R, x

Desc(R) = x(removing everything else from Desc(R))

Add R to Desc(x) (leaving alone everything else in Desc(x))

Remove R from Desc(w) for all $w \neq x$

Store: ST x, R

Add the memory location of x to Desc(x)

Operation: OP Rx, Ry implementing the quas OP x, y

Desc(R_x) = x

Desc(x) = R_x

After operation R_x has R_x OP R_y

Copy: for x = y after processing the load

Add x to Desc(R_y) (note y not x)

Desc(x) = R_y

Minimize the number of registers used:

- When a register holds a temporary value and there are no subsequent uses for this value, we reuse that register
- When a register holds the value of a program variable and there are no subsequent uses of this value, we reuse that register providing this value also in the memory location for the variable
- When a register holds the values of a program variable and all subsequent uses of this value are preceded by a redefinition, we could reuse this register. But to know about all subsequent uses, one

may require live/dead-on-exit knowledge

Assume a, b, c and d are program variables and t, u, v are compiler generated temporaries. These are represented as t1,t1,t2 and t$3. The code generated for different TACs is given below:

```
t = a –b
        LD R1, a
        LD R2, b
        SUB R1, R2
U = a – c
        LD R3, a
        LD R2, c
        SUB R3, R2
v = t + u
        ADD R1, R3
a = d
        LD R2, d
        ST a, R2
d = v + u
        ADD R1, R3
        ST d, R1
Exit
```

DAG for Register Allocation:

Code generation from DAG is much simpler than the linear sequence of three address code
With the help of DAG one can rearrange sequence of instructions and generate and efficient code
There exist various algorithms which are used for generating code from DAG. They are:
Code Generation from DAG:-

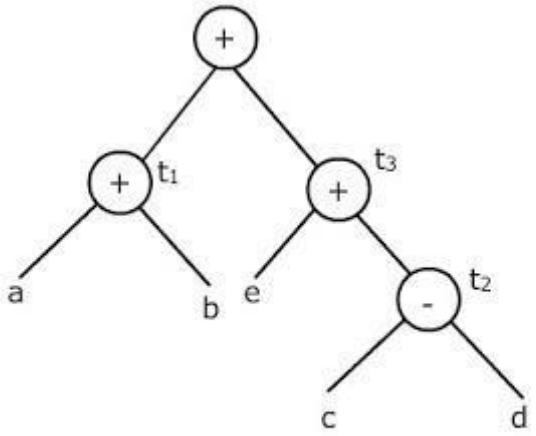
Rearranging Order
Heuristic Ordering
Labeling Algorithm

Rearranging Order

These address code's order affects the cost of the object code which is being generated
Object code with minimum cost can be achieved by changing the order of computations
Example:

```
t1:= a + b
t2:= c – d
t3:= e + t2
t4:= t1 + t3
```

For the expression (a+b) + (e+(c-d)), a DAG can be constructed for the above sequence as shown below



DAG for *(a + b) + (e + (c - d))*

The code is thus generated by translating the three address code line by line

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

Now, if the ordering sequence of the three address code is changed

```
t2:= c - d
t3:= e + t2
t1:= a + b
t4:= t1 + t3
```

Then, an improved code is obtained as:

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

Heuristic Ordering

The algorithm displayed below is for heuristic ordering. It lists the nodes of a DAG such that the node's reverse listing results in the computation order.