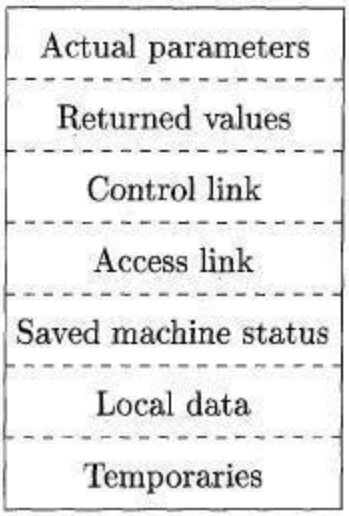
UNIT - IV

RUN TIME STORAGE ORGANIZATION

Activation records

- Procedure calls and returns are usually managed by a run-time stack called the control stack.
- Each live activation has an activation record (sometimes called a frame)
- The root of activation tree is at the bottom of the stack
- The current execution path specifies the content of the stack with the last
- Activation has record in the top of the stack.

A General Activation Record



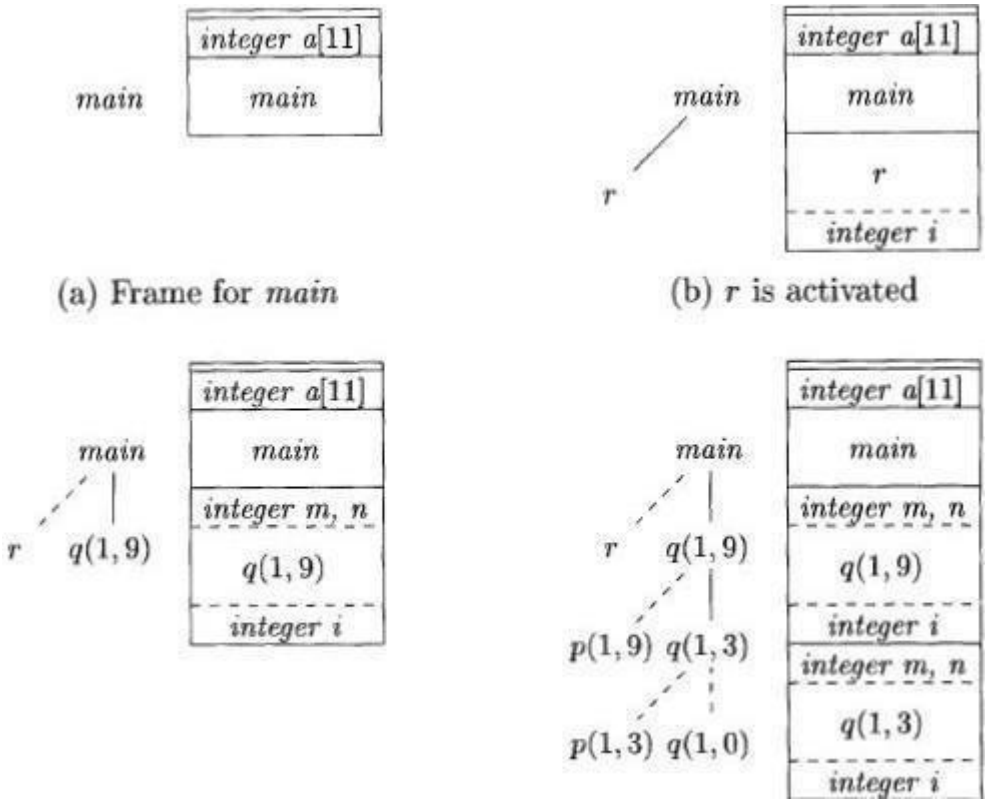
Activation Record

- Temporary values
- Local data
- A saved machine status
- An "access link"
- A control link
- Space for the return value of the called function
- The actual parameters used by the calling procedure

- Elements in the activation record:

- Temporary values that could not fit into registers.
- Local variables of the procedure.
- Saved machine status for point at which this procedure called. Includes return address and contents of registers to be restored.
- Access link to activation record of previous block or procedure in lexical scope chain.
- Control link pointing to the activation record of the caller.
- Space for the return value of the function, if any.
- actual parameters (or they may be placed in registers, if possible)

Downward-growing stack of activation records:



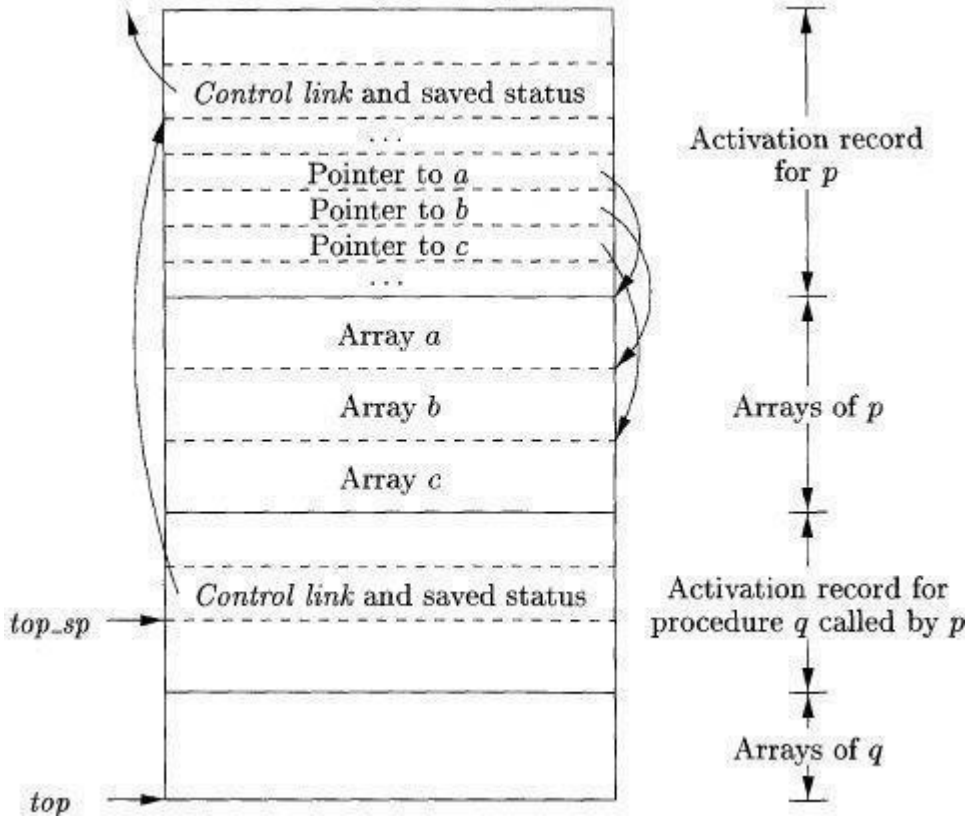
(a) Frame for *main* (b) *r* is activated

Designing Calling Sequences:

- Values communicated between caller and callee are generally placed at the beginning of callee's activation record
- Fixed-length items: are generally placed at the middle
- Items whose size may not be known early enough: are placed at the end of activation record

- We must locate the top-of-stack pointer judiciously: a common approach is to have it point to the end of fixed length fields

Access to dynamically allocated arrays:



ML:

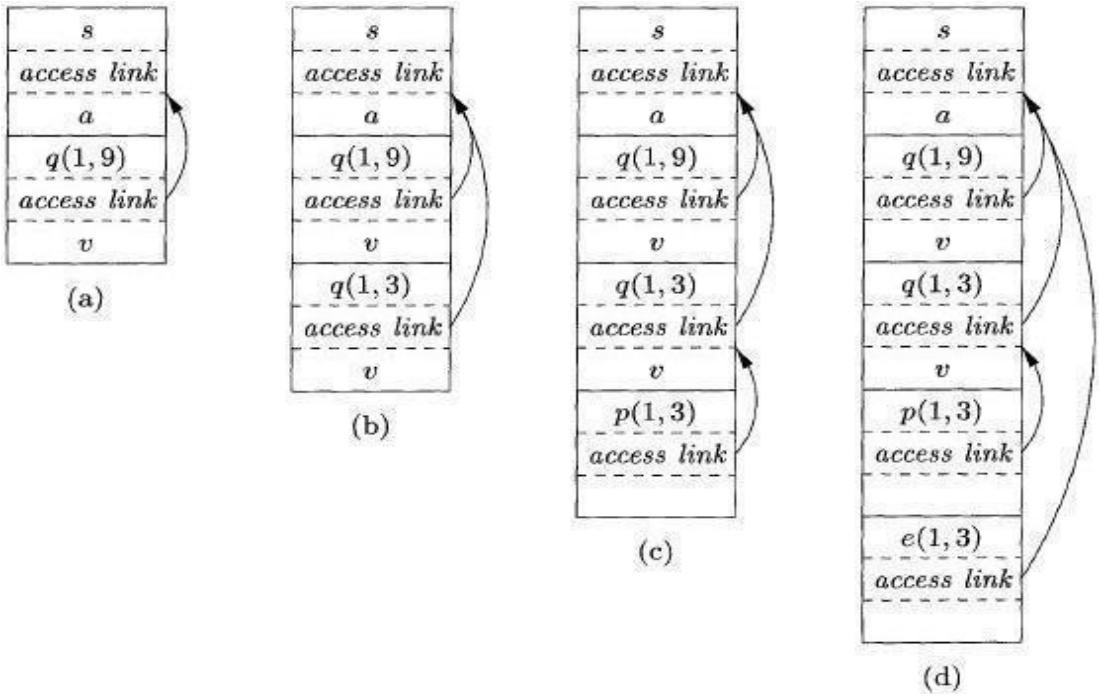
- ML is a functional language
- Variables are defined, and have their unchangeable values initialized, by a statementof the form:
  val (name) = (expression)
- Functions are defined using the syntax:

  fun (name) ((arguments)) = (body)
- For function bodies we shall use let-statements of the form: let (list of definitions) in (statements) end

A version of quick sort, in ML style, using nested functions:

```
1) fun sort(inputFile, outputFile) =
       let
2)         val a = array(11,0);
3)         fun readArray(inputFile) = ··· ;
4)               ··· a ··· ;
5)         fun exchange(i,j) =
6)               ··· a ··· ;
7)         fun quicksort(m,n) =
               let
8)                 val v = ··· ;
9)                 fun partition(y,z) =
10)                      ··· a ··· v ··· exchange ···
               in
11)                ··· a ··· v ··· partition ··· quicksort
               end
       in
12)        ··· a ··· readArray ··· quicksort ···
       end;
```

Access links for finding nonlocal data:

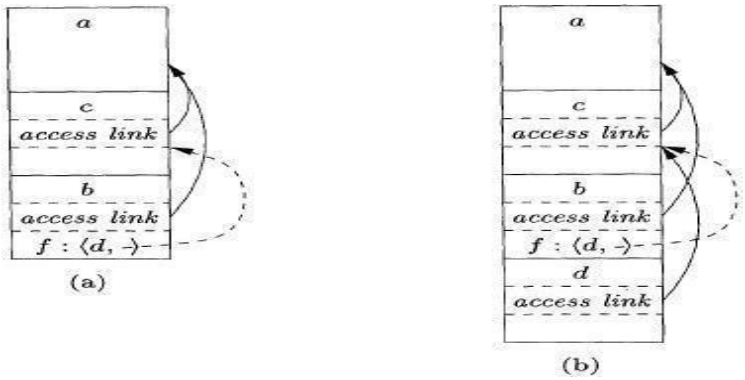


Sketch of ML program that uses function-parameters:

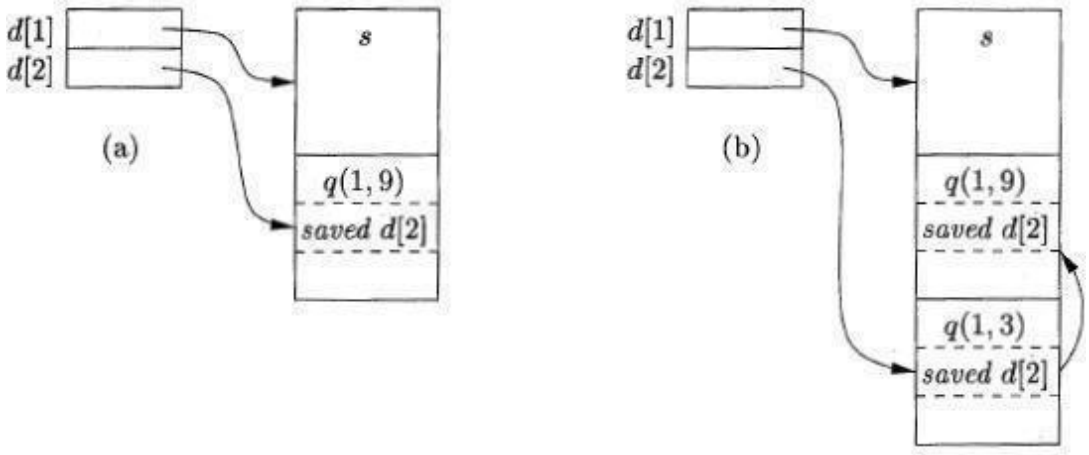
```
fun a(x) =
    let
        fun b(f) =
            ··· f ··· ;
        fun c(y) =
            let
                fun d(z) = ···
            in
                ··· b(d) ···
            end
    in
        ··· c(1) ···
    end;
```

\

Actual parameters carry their access link with them:



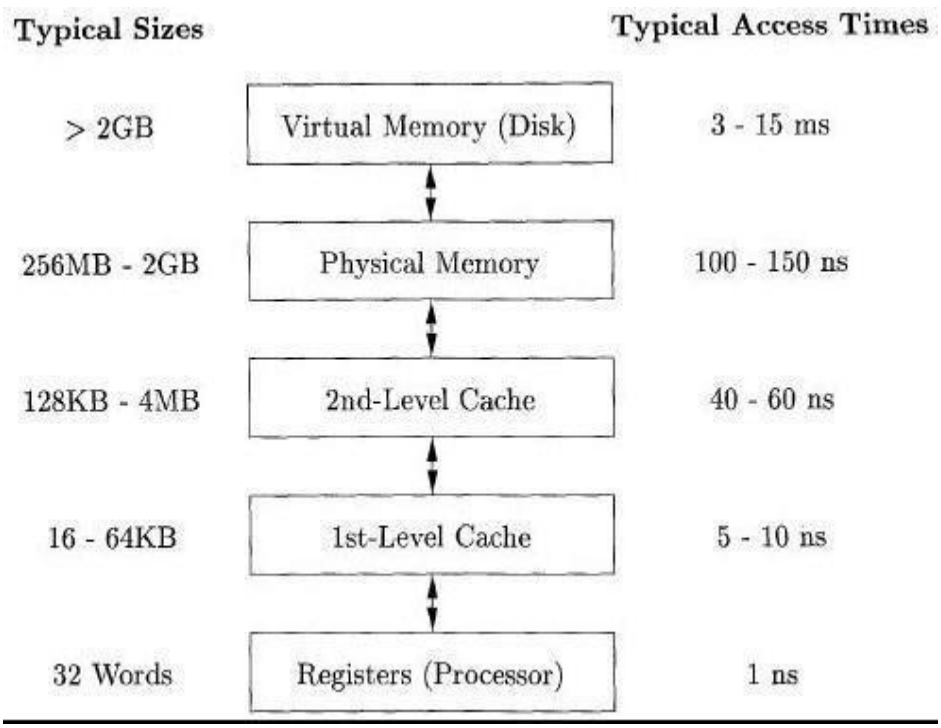
Maintaining the Display:



Memory Manager:

- Two basic functions:
  - Allocation
  - Deallocation
- Properties of memory managers:
  - Space efficiency
  - Program efficiency
  - Low overhead

Typical Memory Hierarchy Configurations:



Locality in Programs:

The conventional wisdom is that programs spend 90% of their time executing 10% of the code:

- Programs often contain many instructions that are never executed.
- Only a small fraction of the code that could be invoked is actually executed in atypical run of the program.
- The typical program spends most of its time executing innermost loops and tight recursive cycles in a program.

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:

Machine independent optimizations:

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

Machine dependant optimizations:

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

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.

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, ○ Copypropagation,
    - 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.

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.

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

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.

Exampl
e: i=0;
if(i=1)
{
a=b+5;
}

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

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.

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.

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 */
```

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.

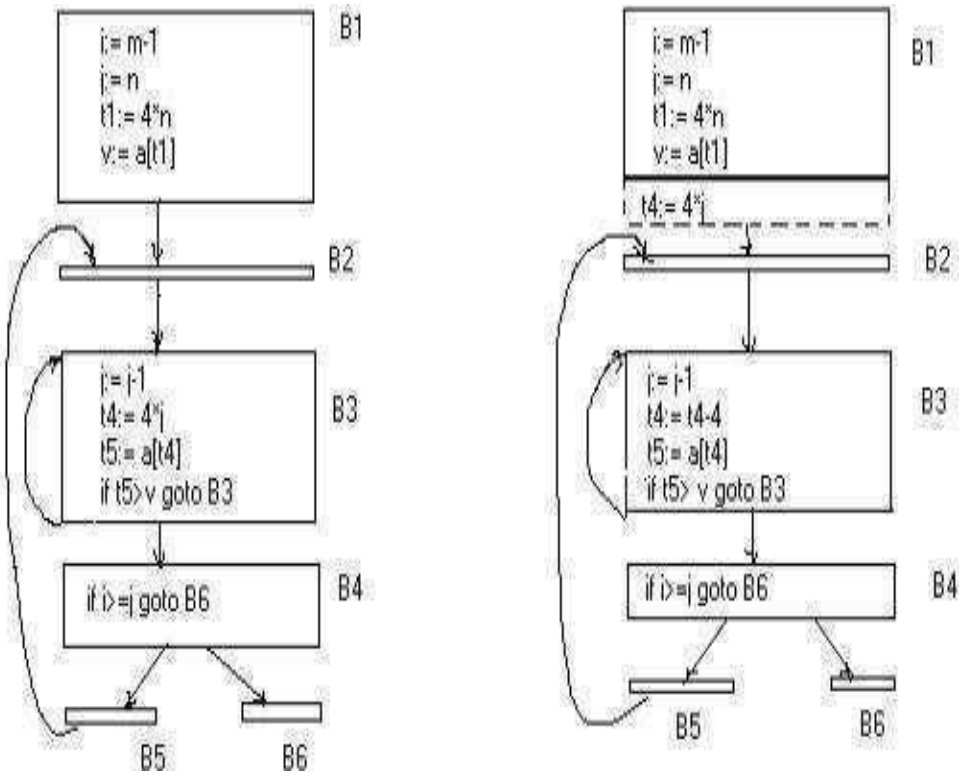
The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction, as is the case on many machines.

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

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.

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 2nd and 4th statements compute the same expression: b+c

and a-d Basic block can be transformed to

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

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.

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.

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.

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:=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.