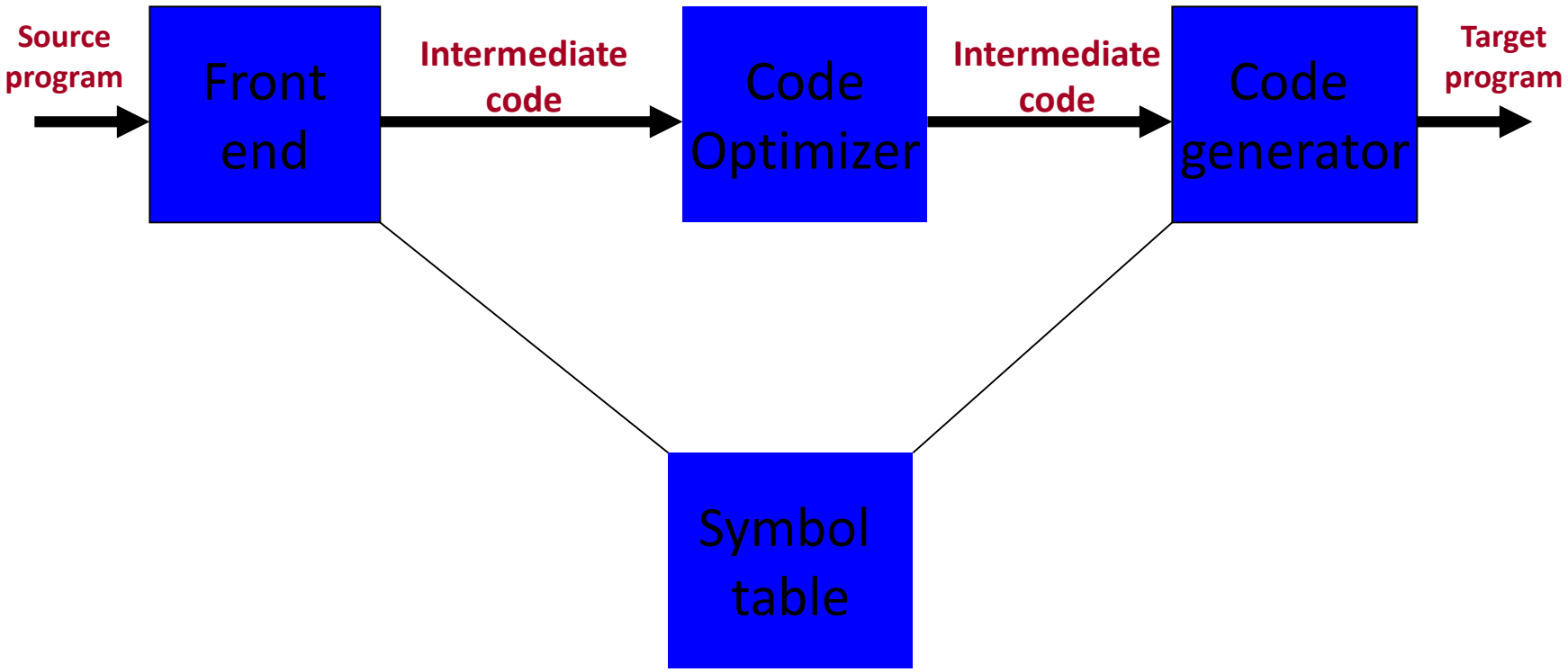
Code Generation

Introduction



Position of code generator

Issues in the Design of a Code Generator

- The most important criteria for the code gen is that it produces correct codes.

- Depend on
  - Input to the code generation(IR)
  - Target program(language)
  - Operating System
  - Memory management
  - Instruction Selection
  - Register allocation and assignment
  - Evaluation order

Basic Blocks and Flow Graphs

Basic Block:

A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibly of the branching except at the end.

- **Flow Graph:** A graph representation of three address statements, called flow graph.
- Nodes in the flow graph represent computations.
- Edges represent the flow of control.
- Used to do better job of register allocation and instruction selection.

Basic Blocks

- **Algorithm: Partitioning three address instructions into basic blocks**
- **Input: a sequence of three address instructions.**
- **Output: a list of basic block for that sequence in which each instruction is assigned to exactly one basic block.**
  - **Method**
    - **We first determine the leader(first instruction in some basic block)**
      1) **The first instruction is a leader**
      2) **Any instruction that is the target of a conditional or unconditional goto is a leader**
      3) **Any instruction that immediately follows a goto or unconditional goto instruction is a leader**
    - **For each leader, its basic block consists of the leader and all the instructions up to but not including the next leader or the end of the program.**

Basic Blocks

- Example : Consider the source code where 10 x 10 matrix a is converted into an identity matrix.

```
for i from 1 to 10 do
  for j from 1 to 10 do
        a[i,j) = 0.0;
for i from 1 to 10 do
  a[i, i] = 1.0;
```

- In generating the intermediate code, we have assumed that the real-valued array elements take 8 bytes each, and that the matrix a is stored in row-major form.

Intermediate code to set a 10 x 10 matrix to an identity matrix

- 1) i = 1
- 2) j = 1
- 3) t l = 10 * i
- 4) t 2 = t l + j
- 5) t 3 = 8 * t2
- 6) t 4 = t3 - 88
- 7) a [t 4] = 0.0
- 8) j = j + 1
- 9) i f j <= 10 goto (3)
- 10) i = i + 1
- 11) i f i <= 10 goto (2)
- 12) i = 1
- 13) t 5 = i - 1
- 14) t 6 = 88 * t5
- 15) a [t 6] = 1.0
- 16) i = i + 1
- 17) i f i <= 10 goto (13)

Basic Blocks

- The leaders are instructions:-
  1) By rule 1 of the algorithm
  2) By rule 2 of the algorithm
  3) By rule 2 of the algorithm
  10) By rule 3 of the algorithm
  12) By rule 3 of the algorithm
  13) By rule 2 of the algorithm

  We conclude that the leaders are instructions 1, 2, 3, 10, 12, and 13.

Basic Blocks

- The basic block of each leader contains all the instructions from itself until just before the next leader.
- Thus,the basic block 1 is just having 1)

  the basic block 2 is having 2)

  the basic block 3 is having 3) to 9)

  the basic block 4 is having 10) to 11)

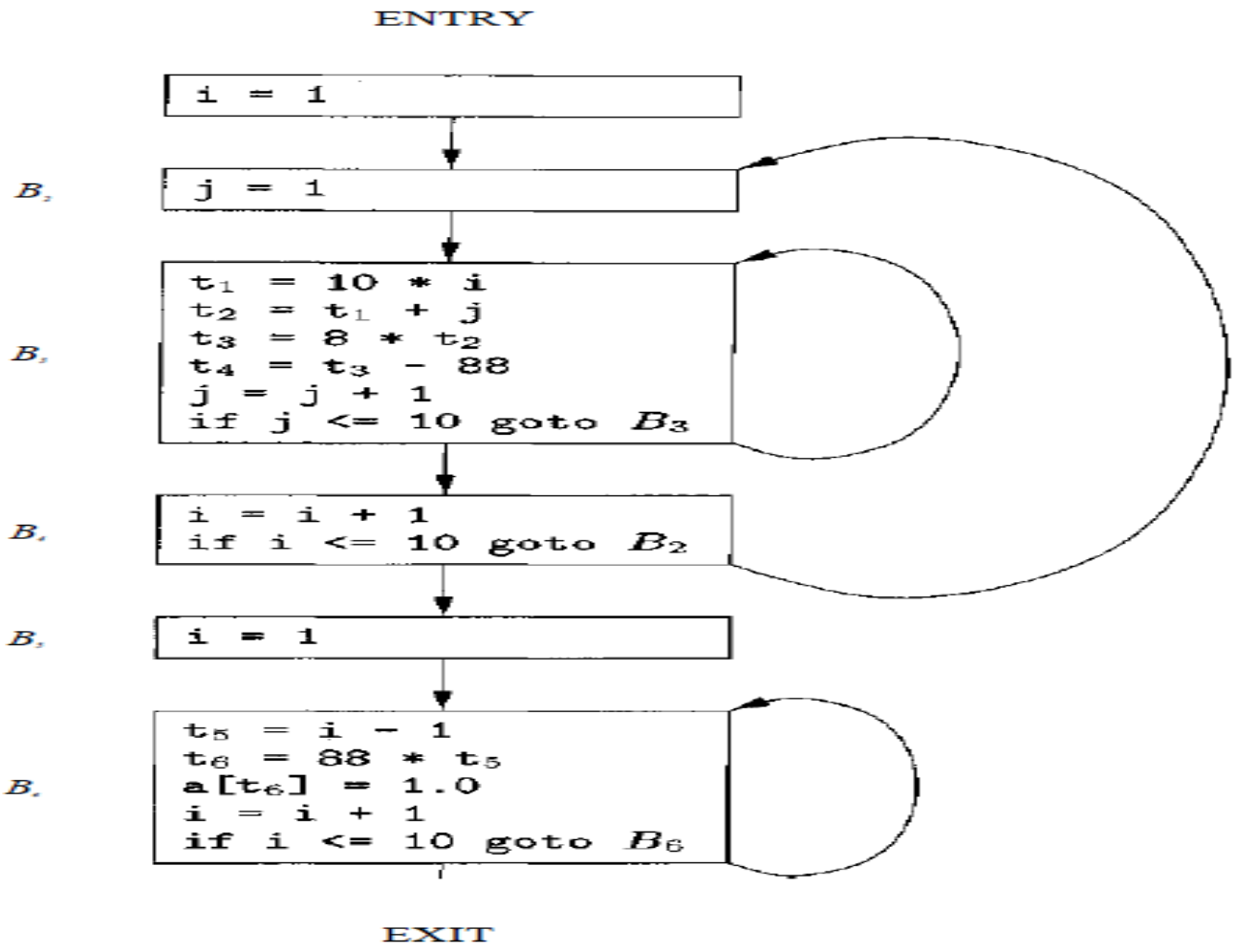
  the basic block 5 is having 12)

  the basic block 6 is having 13) to 17)

Flow Graphs

- Once an intermediate-code program is partitioned into basic blocks, we represent the flow of control between them by a flow graph.
- The nodes of the flow graph are the basic blocks.
- we add two nodes, called the entry and exit, that do not correspond to executable intermediate instructions.
- There is an edge from the entry to the first executable node of the flow graph, that is, to the basic block that comes from the first instruction of the intermediate code.
- There is an edge to the exit from any basic block that contains an instruction that could be the last executed instruction of the program.

Flow Graphs


ENTRY
i = 1
B_2
j = 1
B_3
t_1 = 10 * i
t_2 = t_1 + j
t_3 = 8 * t_2
t_4 = t_3 - 88
j = j + 1
if j <= 10 goto B_3
B_4
i = i + 1
if i <= 10 goto B_2
B_5
i = 1
B_6
t_5 = i - 1
t_6 = 88 * t_5
a[t_6] = 1.0
i = i + 1
if i <= 10 goto B_6
EXIT

Representation of Flow Graphs

- Flow graphs, being quite ordinary graphs, can be represented by any of the data structures appropriate for graphs.
- It is likely to be more efficient to create a linked list of instructions for each basic block.

Loops

- Every program spends most of its time in executing its loops, it is especially important for a compiler to generate good code for loops.
- Many code transformations depend upon the identification of "loops" in a flow graph.

Loops

- We say that a set of nodes *L in a flow graph is a loop if*
- 1. There is a node in L called the loop entry with the property that no other node in L has a predecessor outside L. That is, every path from the entry of the entire flow graph to any node in L goes through the loop entry.
- 2. Every node in L has a nonempty path, completely within L, to the entry of L.

Loops

Example: The flow graph has three loops:

1. *B3 by itself.*
2. *B6 by itself.*
3. *{B2, B3, B4}.*

Flow Graphs

- The successor of *B1 is B2.*
- The successor of *B3 is B3 and B4.*
- The successor of *B4 is B2,B3,B4 and B5.*
- The successor of *B5 is B6.*

Next-Use Information

- If the value of a variable that is currently in a register will never be referenced subsequently, then that register can be assigned to another variable.
- Suppose three-address statement *i assigns a value to x. If statement j has x as an* operand, and control can flow from statement *i to j along a path that has* no intervening assignments to *x, then we say statement j uses the value of x* computed at statement *i. We further say that x is live at statement i.*
- We wish to determine for each three-address statement *x = y + z what the* next uses of *x, y, and z are.*

Next-Use Information

Algorithm to determining the liveness and next-use information for each statement in a basic block.

- **INPUT:** A basic block *B of three-address statements. We assume that the*
- symbol table initially shows all nontemporary variables in *B as being live on* exit.
- **OUTPUT:** At each statement *i: x = y + z in B, we attach to i the liveness and*
- next-use information of *x, y, and z.*
- **METHOD:** We start at the last statement in *B and scan backwards to the*
- beginning of *B. At each statement i: x = y + z in B, we do the following:*
  - 1. Attach to statement *i the information currently found in the symbol table* regarding the next use and liveness of *x, y, and y.*
  - 2. In the symbol table, set *x to "not live" and "no next use."*
  - 3. In the symbol table, set y and z *to "live" and the next uses of y and z to i.*

Next-Use Information

- Here we have used + as a symbol representing any operator. If the three-address statement *i is of the form x = + y or x = y, the steps are the same as above,* ignoring *z.*

- *Note that the order of steps (2) and (3) may not be interchanged* because *x may be y or z.*
- For example :-quadruple i: x := y op z;
  - Record next uses of x, y ,z into quadruple
  - Mark x dead (previous value has no next use)
  - Next use of y is i; next use of z is i; y, z are live

Transformation on Basic Block

- A basic block computes a set of expressions.
- Transformations are useful for improving the quality of code.
- Two important classes of local optimizations that can be applied to a basic blocks
  - Structure Preserving Transformations
  - Algebraic Transformations

The DAG Representation of Basic Blocks

- Many important techniques for local optimization begin by transforming a basic block into a DAG (directed acyclic graph).
- Construction of a DAG for a basic block is as follows:

1. There is a node in the DAG for each of the initial values of the variables appearing in the basic block.

2. There is a node *N associated with each statement s within the block.* The children of *N are those nodes corresponding to statements that are* the last definitions, prior to s, of the operands used by *s.*

- 3. Node *N is labeled by the operator applied at s, and also attached to N* is the list of variables for which it is the last definition within the block.

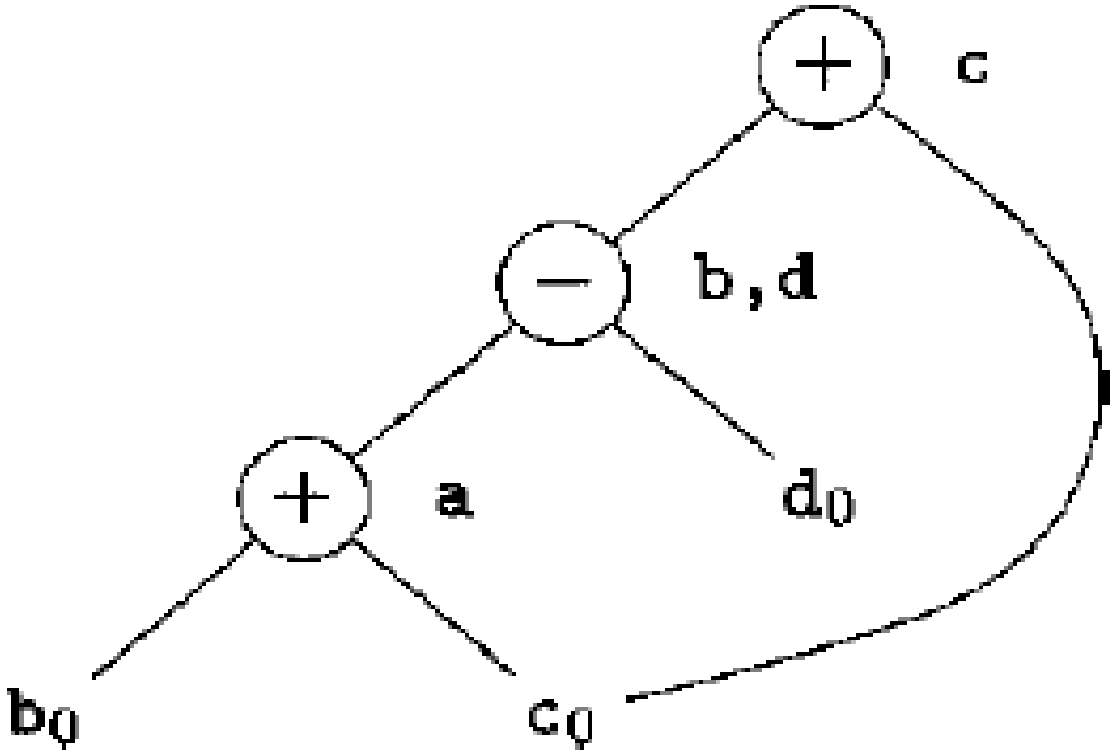
- 4. Certain nodes are designated *output nodes. These are the nodes whose* variables are *live on exit from the block; that is, their values may be* used later, in another block of the flow graph. Calculation of these "live variables" is a matter for global flow analysis.

- The DAG representation of a basic block lets us perform several code improving transformations on the code represented by the block.
- a) We can eliminate *local common subexpressions, that is, instructions that* compute a value that has already been computed.
- b) We can eliminate *dead code, that is, instructions that compute a value* that is never used.
- c) We can reorder statements that do not depend on one another; such reordering may reduce the time a temporary value needs to be preserved in a register.
- d) We can apply algebraic laws to reorder operands of three-address instructions, and sometimes thereby simplify the computation.

Finding Local Common Subexpressions

- Common subexpressions can be detected by using "value-number" method.
- As a new node *M is about* to be added, whether there is an existing node *N with the same children, in* the same order, and with the same operator.
- If so, N computes the same value as *M and may be used in its place.*
- Consider a block **a = b + c**
  b = a - d
  c = b + c
  d = a - d

The DAG for the basic block is



- The node corresponding to the fourth statement d = **a - d has the** operator - and the nodes with attached variables **a and do as children.**
- **Since** the operator and the children are the same as those for the node corresponding to statement two, we do not create this node, but add d to the list of definitions for the node labeled —.
- In fact, if b is not live on exit from the block, then we do not need to compute that variable, and can use d to receive the value represented by the node labeled —.

- The block then become

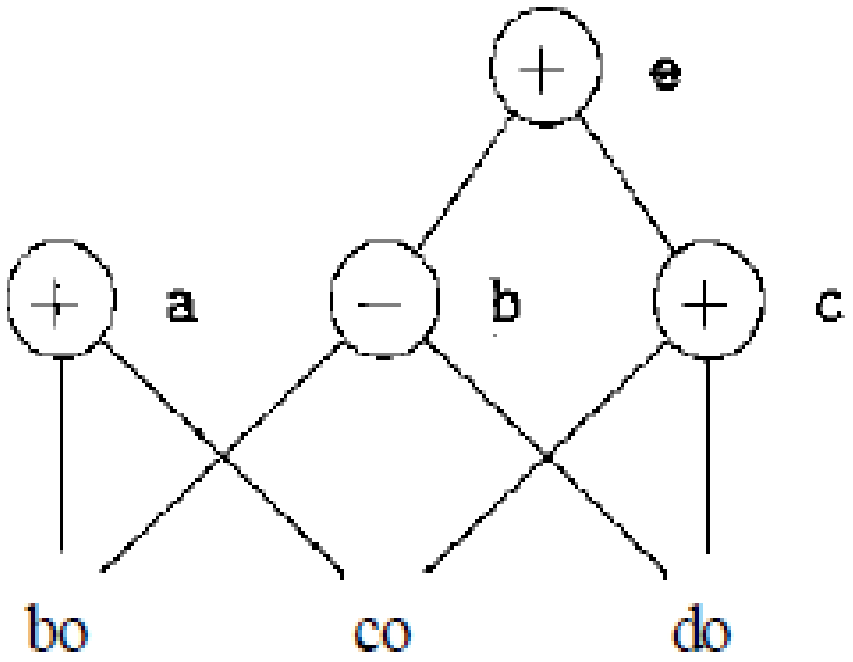
a = b + c

d = a - d

c = d + c

- However, if both b and d are live on exit, then a fourth statement must be used to copy the value from one to the other.

- a = b + c;
- b = b - d
- c = c + d
- e = b + c



- When we look for common subexpressions we really are looking for expressions that are guaranteed to compute the same value, no matter how that value is computed.
- Thus, the DAG method will miss the fact that the expression computed by the first and fourth statements in the sequence is the same b0+c0.

- That is, even though b and c both change between the first and last statements, their sum remains the same, because

  $b + c = (b - d) + (c + d).$

- *The DAG does* not exhibit any common subexpressions.
- However, algebraic identities applied
- to the DAG, may expose the equivalence.

Dead Code Elimination

- Delete from a DAG any root (node with no ancestors) that has no live variables attached.
- Repeated application of this transformation will remove all nodes from the DAG that correspond to dead code.

Structure Preserving Transformations

- **Dead – Code Elemination**

  Say, x is dead, that is never subsequently used, at the point where the statement x = y + z appears in a block. We can safely remove x

- **Renaming Temporary Variables**
  - **say, t = b+c where t is a temporary var.**
  - **If we change u = b+c, then change all instances of t to u.**
- **Interchange of Statements**
  - **t_1 = b + c**
  - **t_2 = x + y**
  - **We can interchange iff neither x nor y is t_1 and neither b nor c is t_2**

Algebraic Transformations

- Replace expensive expressions by cheaper one
  - X = X + 0 eliminate
  - X = X * 1 eliminate
  - X = y**2 (why expensive? Answer: Normally implemented by function call)
    - by X = y * y
- **Flow graph:**
  - We can add flow of control information to the set of basic blocks making up a program by constructing directed graph called flow graph.
  - There is a directed edge from block B_1 to block B_2 if
    - **There is conditional or unconditional jump from the last statement of B_1 to the first statement of B_2 or**
    - **B_2 is immediately follows B_1 in the order of the program, and B_1 does not end in an unconditional jump.**

Peephole Optimization

- The peephole is a small, sliding window on a program. The code in the
- peephole need not be contiguous, although some implementations do require this

characteristic of peephole optimizations:

1 Redundant-instruction elimination

2 Flow-of-control optimizations

3 Algebraic simplications

4 Use of machine idioms

Eliminating Redundant Loads and Stores:

If we see the instruction sequence

LD R0, a

ST a, R0

in a target program, we can delete the store instruction because whenever it is executed, the first instruction will ensure that the value of a has already been loaded into register R0.

Eliminating Unreachable Code :

Another opportunity for peephole optimization is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed

```
    if debug != 1 goto L2
       Print debugging information
    L2:
```

If debug is set to 0 at the beginning of the program, constant propagation would transform this sequence into

```
    if 0 != 1 goto L2
       print debugging information
    L2:.
```

Flow-of-Control Optimizations :

Simple intermediate code-generation algorithms frequently produce jumps to jumps, jumps to conditional jumps, or conditional jumps to jumps. These 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 sequence

goto L1

……

L1: goto L2

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.

Algebraic Simplication and Reduction in Strength :

These algebraic identities can also be used by a peephole optimizer to

eliminate three-address statements such as

x = x + 0

or

x = x * 1

in the peephole.

Use of Machine Idioms :

some machines have auto-increment and auto-decrement addressing modes. These add or subtract one from an operand before or after using its value. The use of the modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like x = x+1.

Register Allocation and Assignment

- Instruction involving only register operands are shorter and faster then those involving memory operands. This also means that proper use of register help in generating the good code. This section presents various strategies for deciding what values in a program should reside in registers(register allocation)and in which register each value should reside (register assignment).
- One approach to register allocation and assignment is to assign specific value in an object program to certain registers.
- This approach has the advantage that it simplifies the design of a compiler.

- Disadvantage is that , applied too strictly , it uses registers inefficiently; certain registers may go unused over substantial portions of code, while unnecessary loads and stores are generated.

- Now we will discuss various strategies used in register and assignment and those
  - Global Register Allocation
  - Usage Counts
  - Register Assignment for Outer Loops
  - Register Allocation by Graph Coloring

Global Register Allocation

- Generating the code the registers are used to hold the value for the duration of single block.
- All the live variables are stored at the end of each block.
- For the variables that are used consistently we can allocate specific set of registers.
- Hence allocation of variables to specific registers that is consistent across the block boundaries is called global register allocation.

Usage Counts

- The usage count is the count for the use of some variable x in some register used in any basic block.
- The usage count gives the idea about how many units of cost can be saved by selecting a specific variable for global register allocation.
- The approximate formula for usage count for the loop L in some basic block B can be given as,
- $\sum_{\text{Block B in L}}$ (use(x , B)+2*live(x , B))

Where use (x, B) is number of times x used in block B prior to any definition of x and live (x, B)=1 is live on exit from B; otherwise live(x)=0.

Register Assignment for Outer Loops

- Having assigned registers and generated code for inner loops, we may apply the same idea to progressively loops.
- If an outer loop L1, contains an inner loop L2, the names allocated registers in L2 need not be allocated registers in L1-L2.
- However, if name x is allocated a register in loop L1 but not L2, we must store x on entrance to L2 and load x if we leave L2, and enter a block of L1-L2.
- Similarly, if we choose to allocate x a register in L2, but not L1 must load x on entrance to L2 and store x on exit from L2.

Register Allocation by Graph Coloring

- 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 register.
- Graph coloring is a simple systematic technique for allocating registers and managing register spills.
- In this method, two passes are used.
- In first , target-machine instruction are selected as though there were as infinite number of symbolic registers; in effect, names used un the intermediate code become names of registers and the three-address statements become machine- language statements.

  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.

Machine-Independent Optimization

- The main aim of machine-independent optimization is to improve the generated intermediate code so that compiler can get better target code.
- Eliminating unwanted code from the object code or replacing one set of code with another set of code, which makes the object code faster without changing the result of object code, is generally called **code improvement** or **code optimization**.

The Principle Sources of Optimization :

- Common sub expression elimination,
- Copy propagation,
- Dead-code elimination, and
- Constant folding

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

t6: = b [t4] +t5

CONT.....

- 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 t4: =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.

- Example:
- i=0;
- if(i=1)
- {
- a=b+5;
- }

CONT.....

- Here, ‘if ’ statement is dead code because this condition will never get satisfied.
- 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:

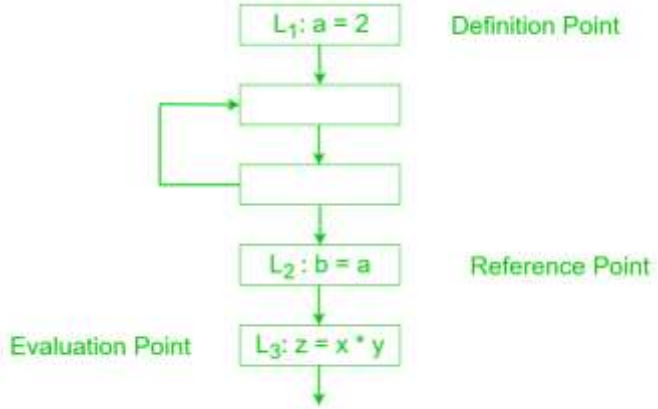
- An important modification that decreases the amount of code in a loop is code motion.
- This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and places the expression before the loop.
- Note that the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

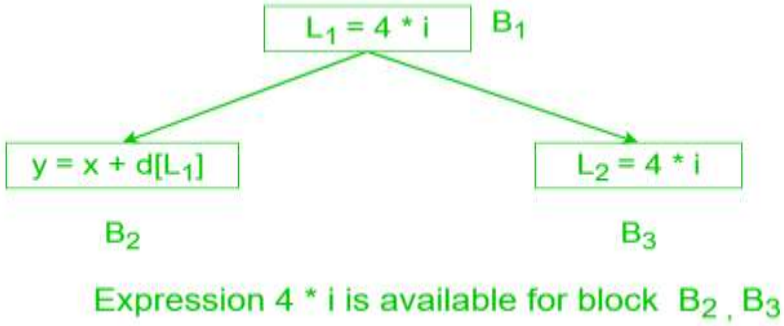
Introduction to Data- flow analysis

- It is the analysis of flow of data in control flow graph, i.e., the analysis that determines the information regarding the definition and use of data in program. With the help of this analysis optimization can be done. In general, its process in which values are computed using data flow analysis.The data flow property represents information which can be used for optimization.
- **Basic Terminologies –**
- **Definition Point:** a point in a program containing some definition.
- **Reference Point:** a point in a program containing a reference to a data item.
- **Evaluation Point:** a point in a program containing evaluation of expression.

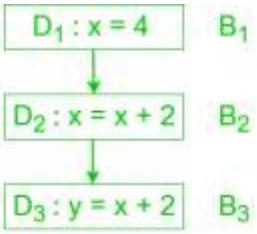


Data Flow Properties –

•**Available Expression –** A expression is said to be available at a program point x iff along paths its reaching to x. A Expression is available at its evaluation point.
A expression a+b is said to be available if none of the operands gets modified before their use.

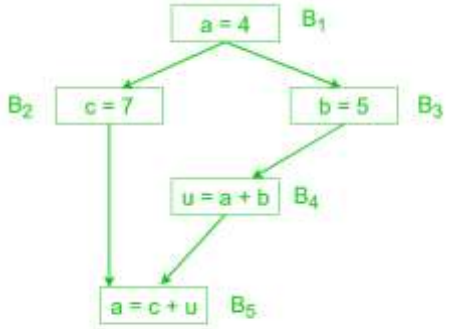


Reaching Definition – A definition D is reaches a point x if there is path from D to x in which D is not killed, i.e., not redefined.



D_1 is reaching definition for B_2 but not for B_3 since it is killed by D_2

Live variable – A variable is said to be live at some point p if from p to end the variable is used before it is redefined else it becomes dead.



a is live at block B_1 , B_3 , B_4 but killed at B_5

Busy Expression – An expression is busy along a path iff its evaluation exists along that path and none of its operand definition exists before its evaluation along the path.