Unit-05

Code Generation and Optimization

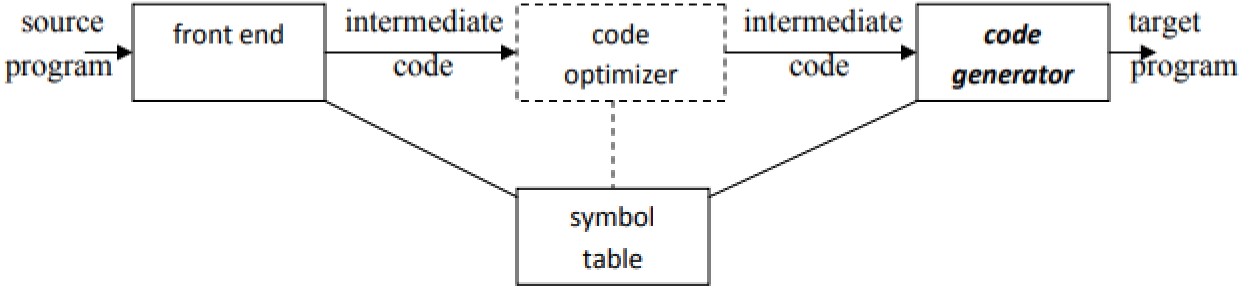
Syllabus

- Code Generation: Code generation Issues. Basic blocks and flow graphs, A Simple Code Generator.
- Code Optimization: Machine Independent: Peephole optimizations: Common Sub-expression elimination, Removing of loop invariants, Induction variables and Reduction in strengths, use of machine idioms, Dynamic Programming Code Generation.
- Machine dependent Issues: Assignment and use of registers, Rearrangement of Quadruples for code optimization.

Code Generation

- The final phase in compiler model is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program.
- Code generator main tasks:
  - Instruction selection
  - Register allocation and assignment
  - Instruction ordering

Position of code generator



Issues In The Design Of A Code Generator

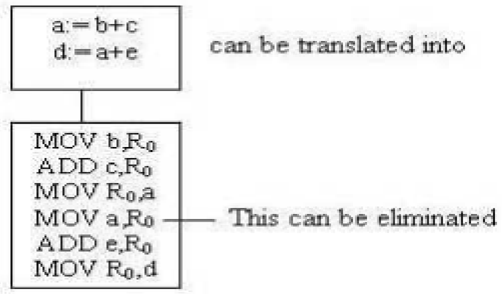
- The following issues arise during the code generation phase :
    1. Input to code generator
    2. Target program
    3. Memory management
    4. Instruction selection
    5. Register allocation
    6. Evaluation order

Issues In The Design Of A Code Generator

- Input to code generator :
    - Prior to code generation, the front end must be scanned, parsed and translated into intermediate representation along with necessary type checking. Therefore, input to code generation is assumed to be error-free.
- Target program -The output of the code generator is the target program. The output may be :
    - Absolute machine language - It can be placed in a fixed memory location and can be executed immediately
    - Relocatable machine language - It allows subprograms to be compiled separately.
    - Assembly language - Code generation is made easier

Issues In The Design Of A Code Generator

- Memory management:
  - Names in the source program are mapped to addresses of data objects in run-time memory by the front end and code generator.
  - It makes use of symbol table, that is, a name in a three-address statement refers to a symbol-table entry for the name.
  - Labels in three-address statements have to be converted to addresses of instruction
- Instruction selection:
  - The instructions of target machine should be complete and uniform.
  - Instruction speeds and machine idioms are important factors when efficiency of target program is considered.
  - The quality of the generated code is determined by its speed and size.
  - The former statement can be translated into the latter statement as shown below



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

can be translated into

```
MOV b,R0
ADD c,R0
MOV R0,a
MOV a,R0
ADD e,R0
MOV R0,d
```

This can be eliminated

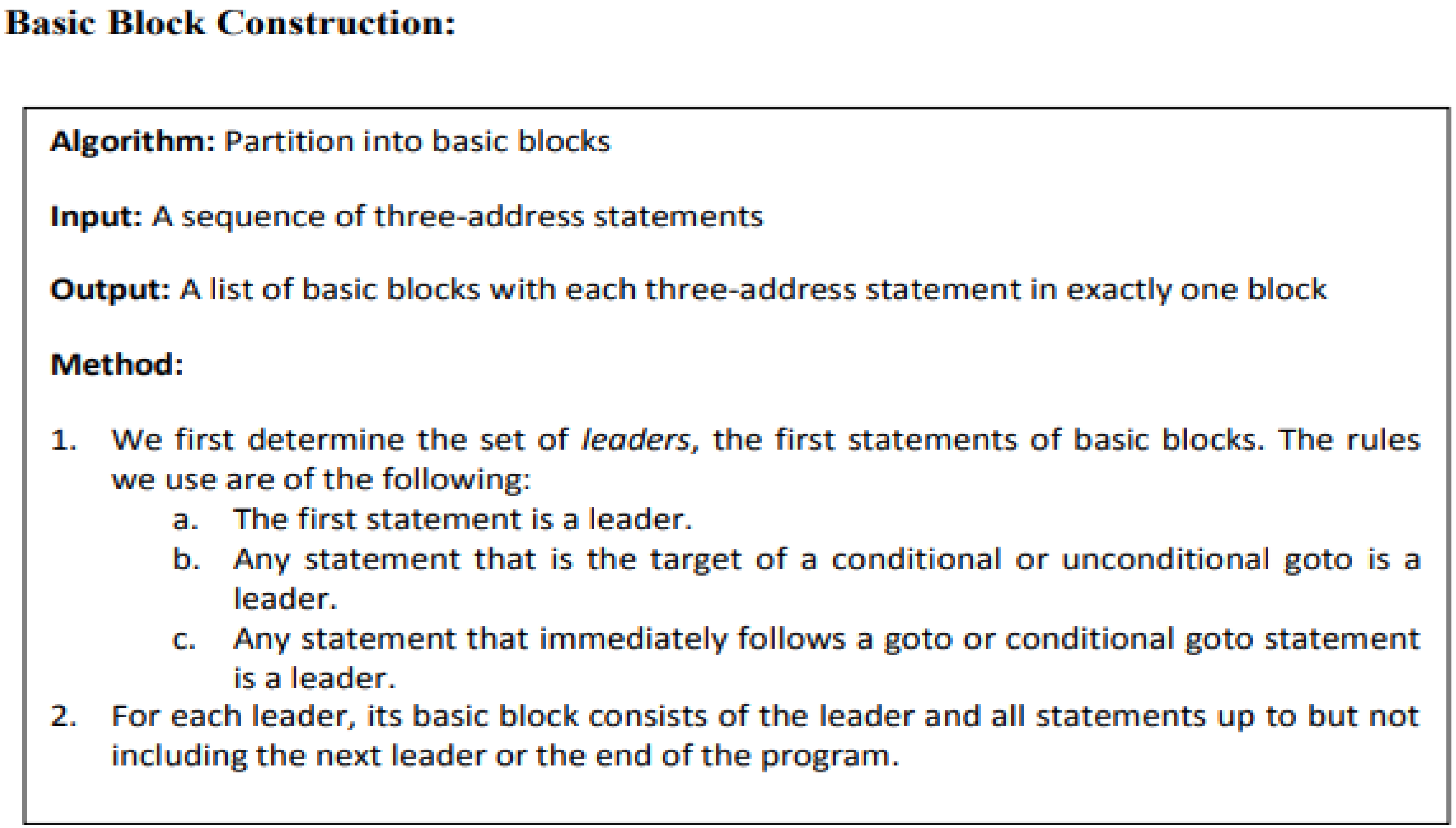
Issues In The Design Of A Code Generator

- Register allocation
  - Instructions involving register operands are shorter and faster than those involving operands in memory.
  - The use of registers is subdivided into two sub problems :
    - Register allocation – the set of variables that will reside in registers at a point in the program is selected.
    - Register assignment – the specific register that a variable will reside in is picked
- Evaluation order
  - The order in which the computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others

Basic blocks and flow graphs

- A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without any halt or possibility of branching except at the end.
- The following sequence of three-address statements forms a basic block:
    - t1 : = a * a
    - t2 : = a * b
    - t3 : = 2 * t2
    - t4 : = t1 + t3
    - t5 : = b * b
    - t6 : = t4 + t5

Algorithm for Basic Block Construction



Basic Block Construction:

Algorithm: Partition into basic blocks

Input: A sequence of three-address statements

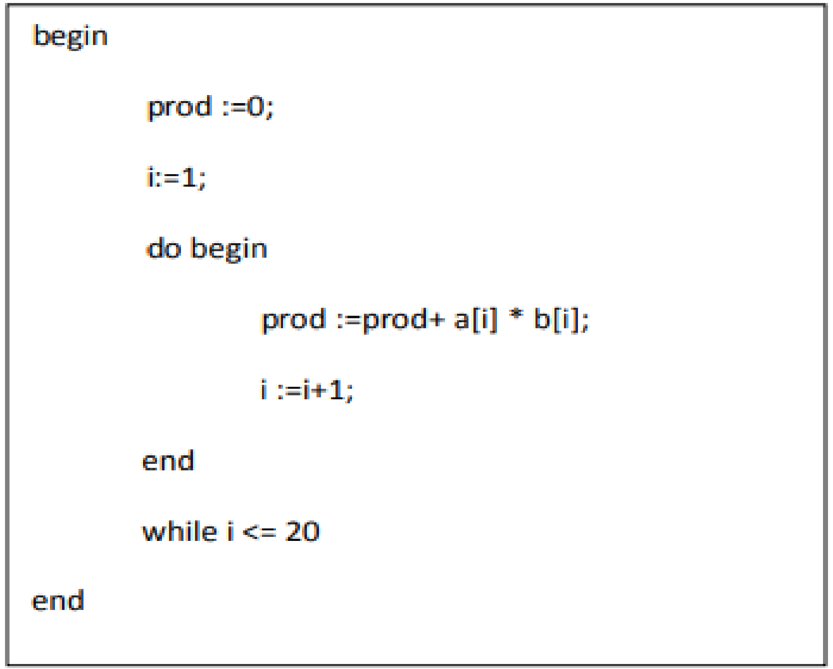
Output: A list of basic blocks with each three-address statement in exactly one block

Method:

1. We first determine the set of *leaders*, the first statements of basic blocks. The rules we use are of the following:
    a. The first statement is a leader.
    b. Any statement that is the target of a conditional or unconditional goto is a leader.
    c. Any statement that immediately follows a goto or conditional goto statement is a leader.
2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

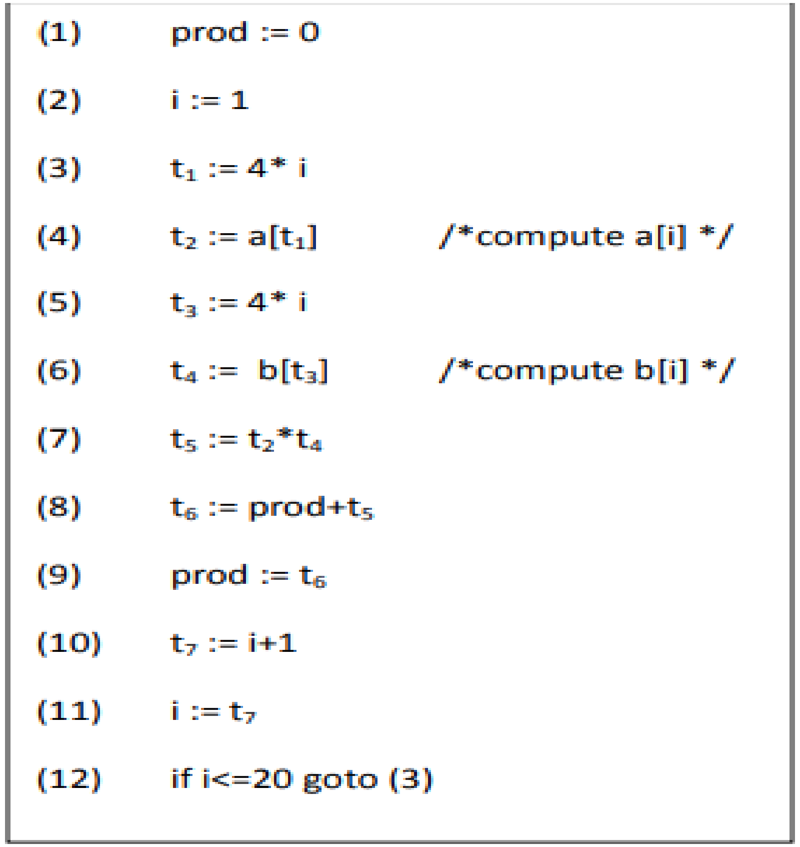
Basic blocks and flow graphs construction

- Consider the following source code for dot product of two vectors a and b of length 20.



```
begin
          prod :=0;
          i:=1;
          do begin
                    prod :=prod+ a[i] * b[i];
                    i :=i+1;
          end
          while i <= 20
end
```

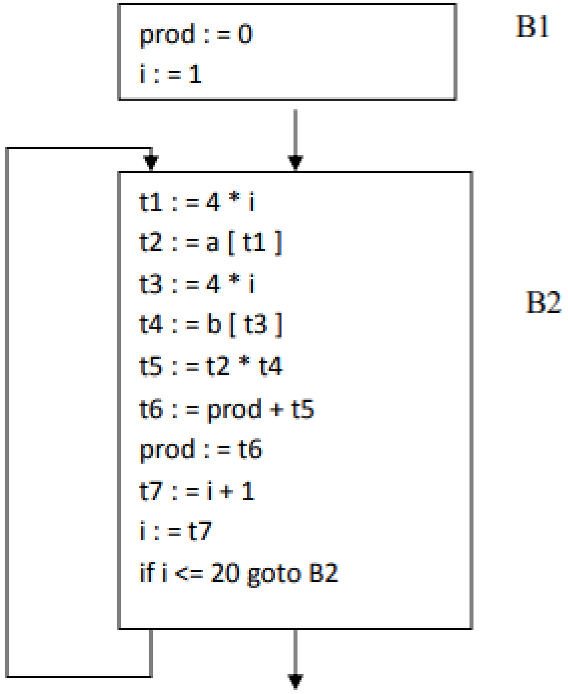
The three-address code for the above source program is given as :



```
(1)     prod := 0
(2)     i := 1
(3)     t_1 := 4* i
(4)     t_2 := a[t_1]        /*compute a[i] */
(5)     t_3 := 4* i
(6)     t_4 :=  b[t_3]       /*compute b[i] */
(7)     t_5 := t_2*t_4
(8)     t_6 := prod+t_5
(9)     prod := t_6
(10)    t_7 := i+1
(11)    i := t_7
(12)    if i<=20 goto (3)
```

Basic blocks and flow graphs construction

- Basic block 1: Statement (1) to (2)
- Basic block 2: Statement (3) to (12)



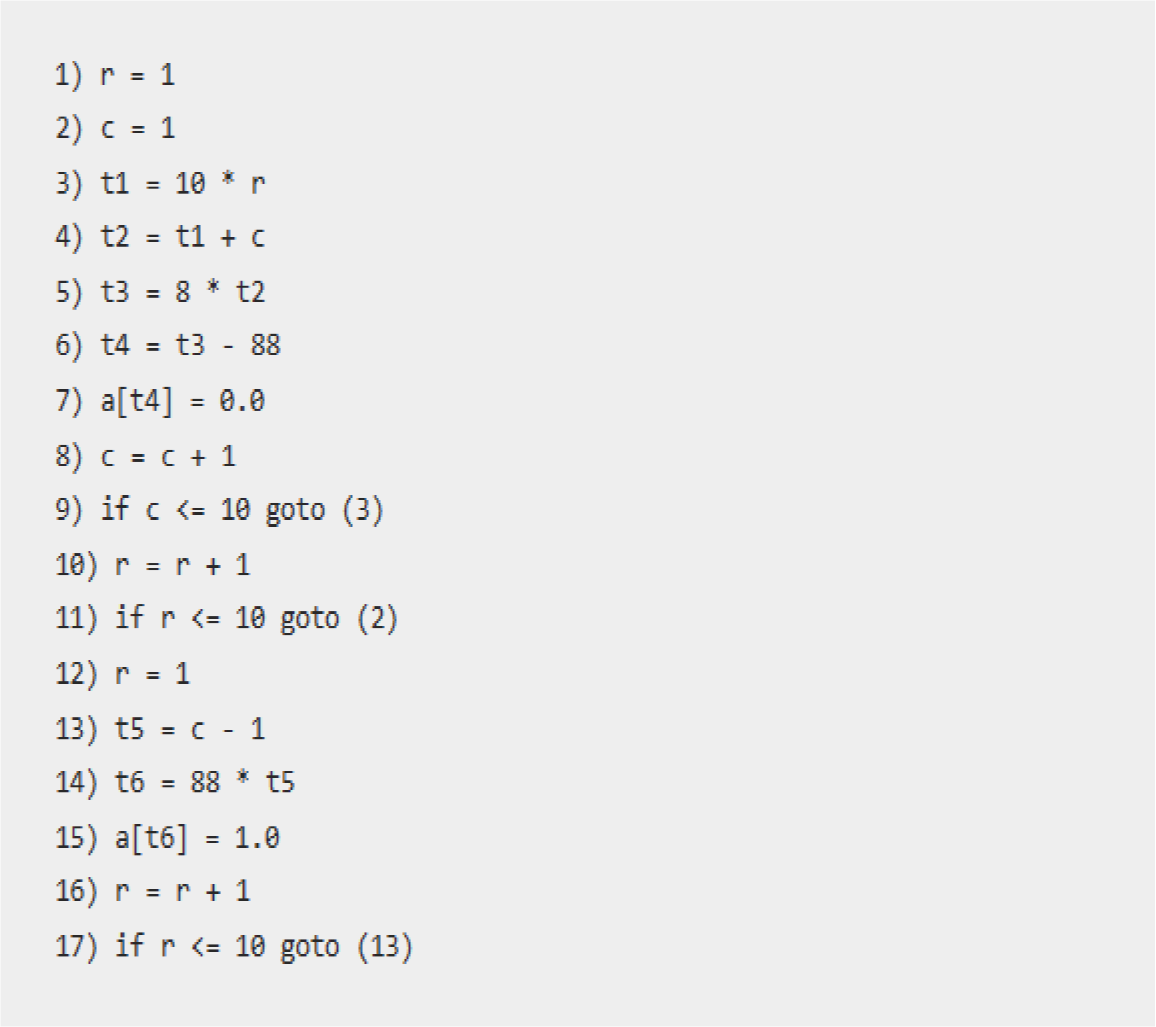
```
prod : = 0
i : = 1
```

```
t1 : = 4 * i
t2 : = a [ t1 ]
t3 : = 4 * i
t4 : = b [ t3 ]
t5 : = t2 * t4
t6 : = prod + t5
prod : = t6
t7 : = i + 1
i : = t7
if i <= 20 goto B2
```

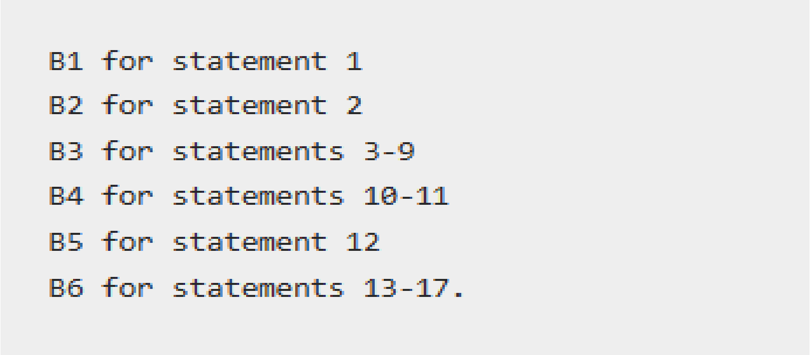
Prod:=0
i:=1

t1=4*I
t2=a[t1]
t3=4*I
t4=b[t3]
t5=t2*t4
t6=prod+t5
Prod=t6
t7=i+1
i=t7
If i<=10 goto (3)

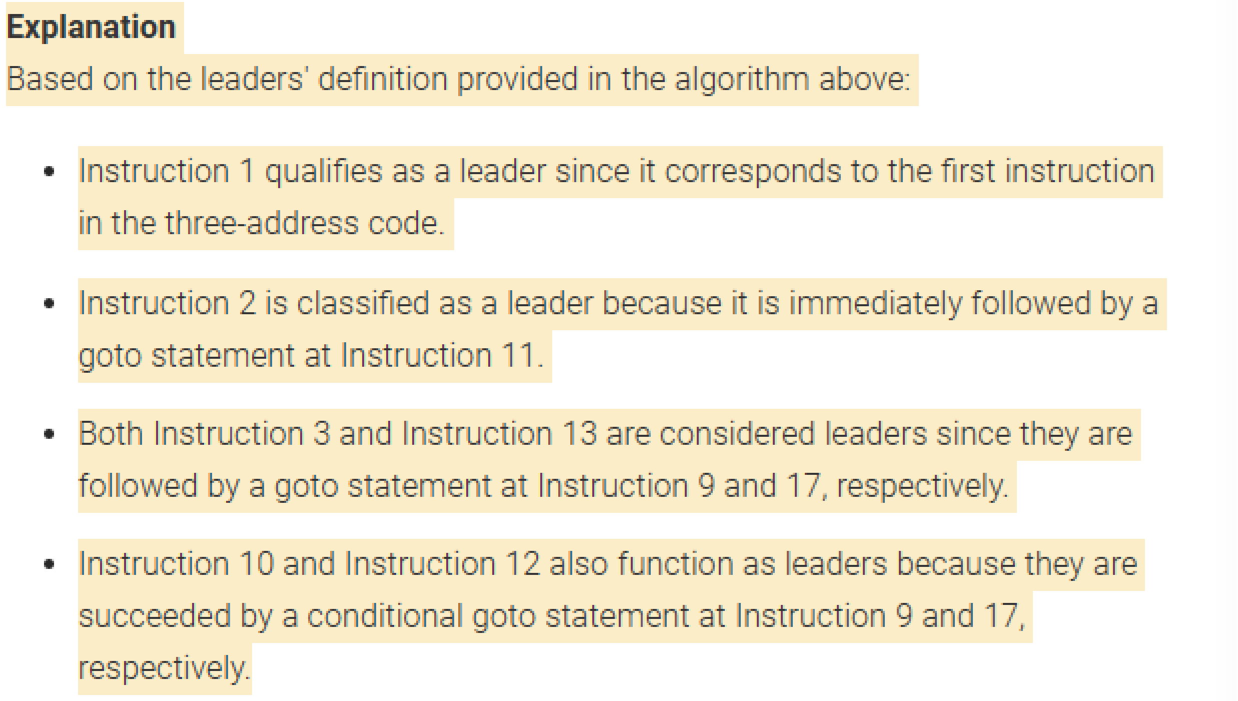
Example of Basic Block



```
1) r = 1
2) c = 1
3) t1 = 10 * r
4) t2 = t1 + c
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) c = c + 1
9) if c <= 10 goto (3)
10) r = r + 1
11) if r <= 10 goto (2)
12) r = 1
13) t5 = c - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) r = r + 1
17) if r <= 10 goto (13)
```



```
B1 for statement 1
B2 for statement 2
B3 for statements 3-9
B4 for statements 10-11
B5 for statement 12
B6 for statements 13-17.
```

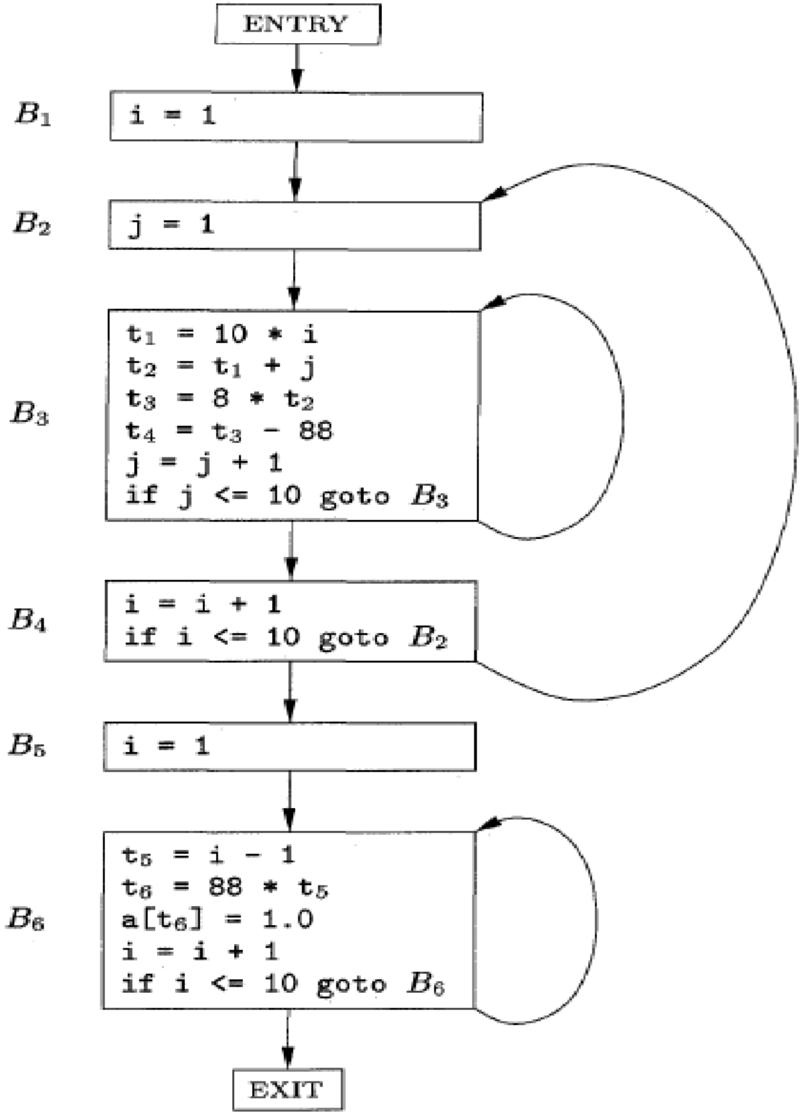


Explanation

Based on the leaders' definition provided in the algorithm above:

- Instruction 1 qualifies as a leader since it corresponds to the first instruction in the three-address code.
- Instruction 2 is classified as a leader because it is immediately followed by a goto statement at Instruction 11.
- Both Instruction 3 and Instruction 13 are considered leaders since they are followed by a goto statement at Instruction 9 and 17, respectively.
- Instruction 10 and Instruction 12 also function as leaders because they are succeeded by a conditional goto statement at Instruction 9 and 17, respectively.

Flow graph based on Basic Blocks



B_1

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

DAG representation of basic blocks

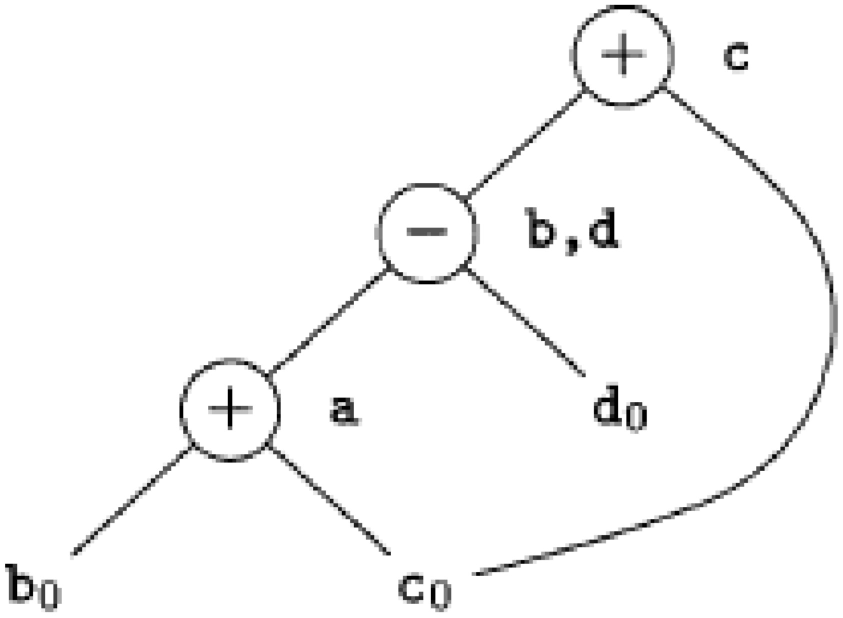
- There is a node in the DAG for each of the initial values of the variables appearing in the basic block.
- There is a node N associated with each statement s within the block. The children of N are those nodes corresponding to statements that are the last definitions, prior to s, of the operands used by s.
- 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.
- Certain nodes are designated *output nodes*. These are the nodes whose variables are *live on exit* from the block.

The Dag Representation For Basic Blocks

- A DAG for a basic block is a directed acyclic graph with the following labels on nodes:
    - 1. Leaves are labeled by unique identifiers, either variable names or constants.
    - 2. Interior nodes are labeled by an operator symbol.
    - 3. Nodes are also optionally given a sequence of identifiers for labels to store the computed values.
- DAGs are useful data structures for implementing transformations on basic blocks.
- It gives a picture of how the value computed by a statement is used in subsequent statements.
- It provides a good way of determining common sub - expressions

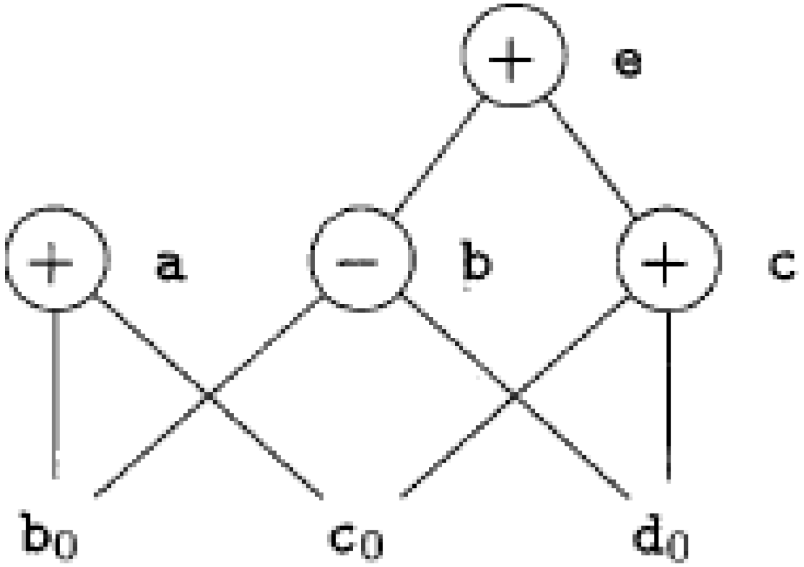
DAG for basic block

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

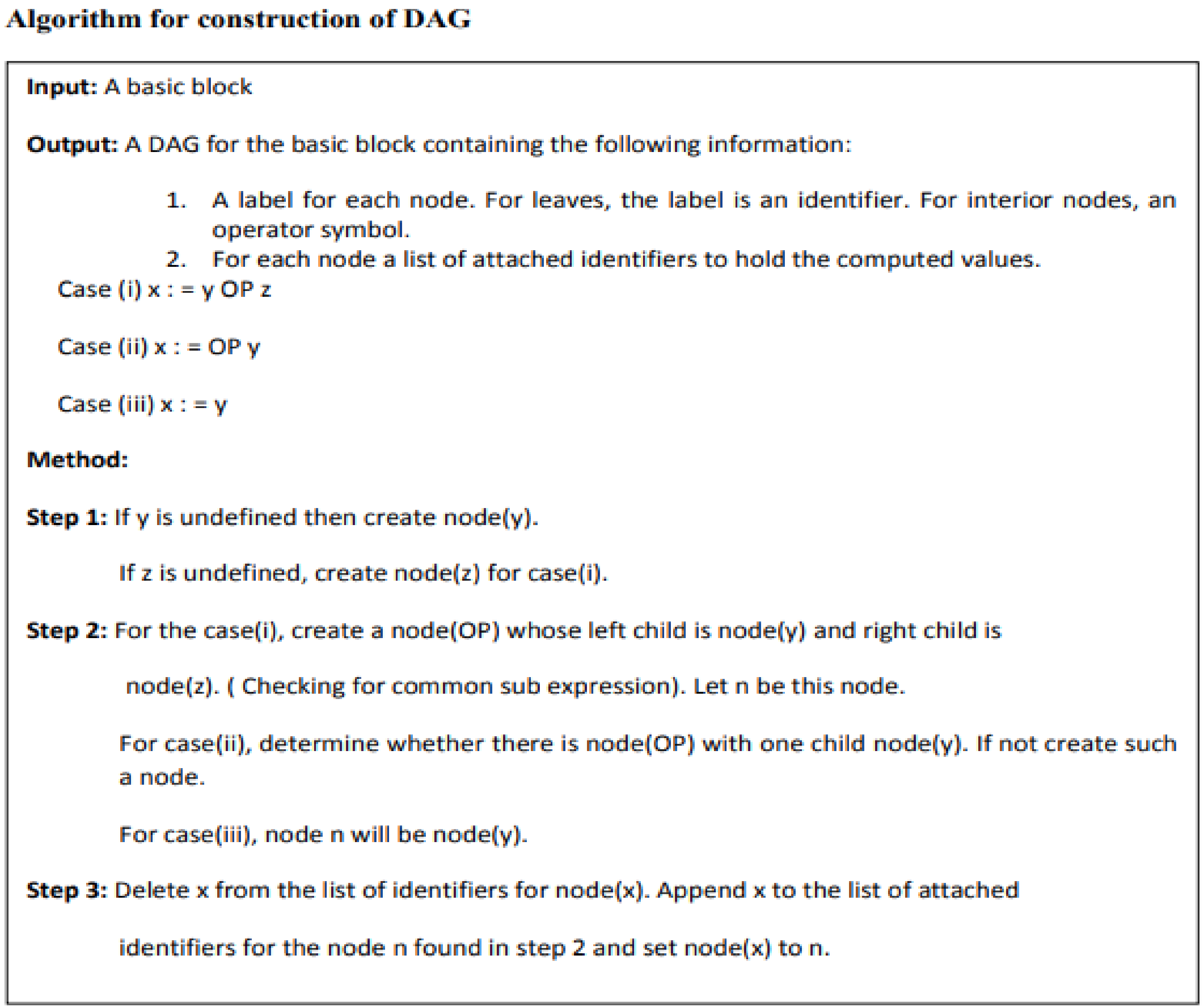


DAG for basic block

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



Algorithm for construction of DAG



Algorithm for construction of DAG

Input: A basic block

Output: A DAG for the basic block containing the following information:

1. A label for each node. For leaves, the label is an identifier. For interior nodes, an operator symbol.
2. For each node a list of attached identifiers to hold the computed values.

Case (i) x : = y OP z

Case (ii) x : = OP y

Case (iii) x : = y

Method:

Step 1: If y is undefined then create node(y).

If z is undefined, create node(z) for case(i).

Step 2: For the case(i), create a node(OP) whose left child is node(y) and right child is

node(z). (Checking for common sub expression). Let n be this node.

For case(ii), determine whether there is node(OP) with one child node(y). If not create such a node.

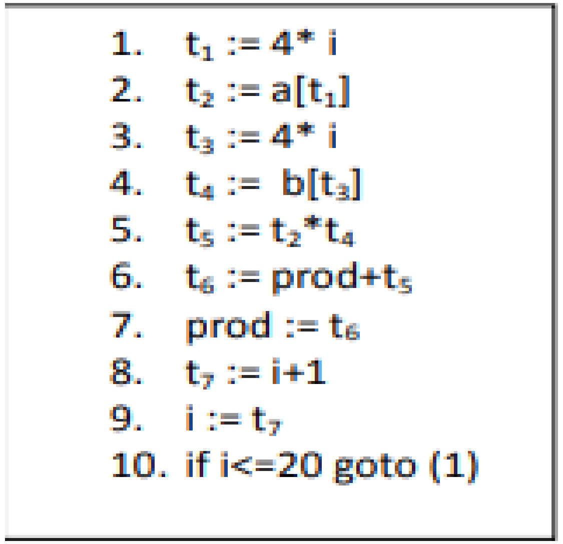
For case(iii), node n will be node(y).

Step 3: Delete x from the list of identifiers for node(x). Append x to the list of attached

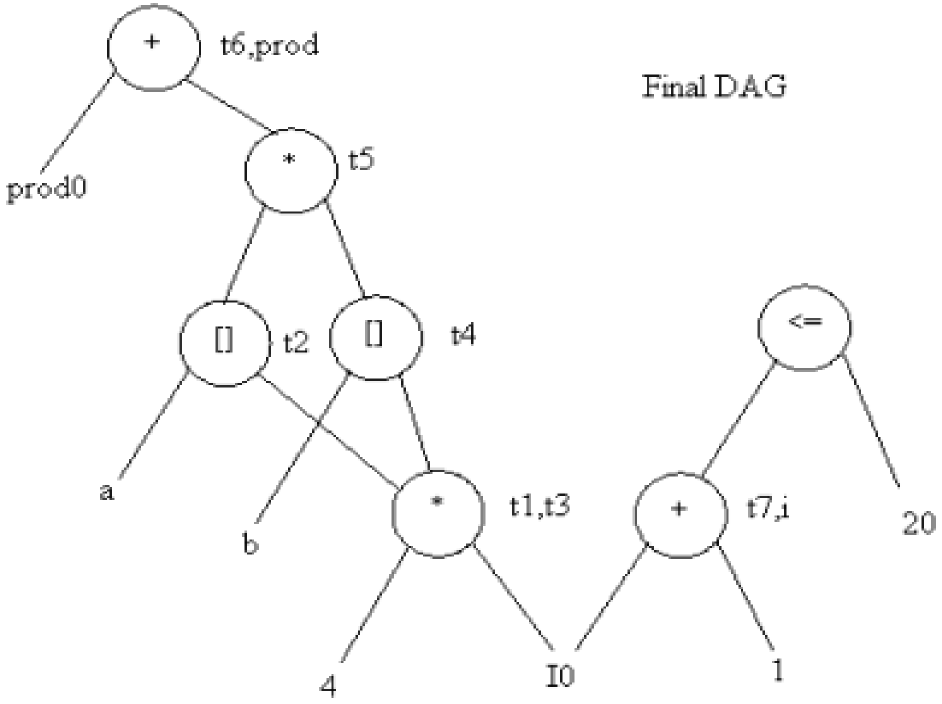
identifiers for the node n found in step 2 and set node(x) to n.

construction of DAG

- Example: Consider the block of three- address statements:



1. $t_1 := 4* i$
2. $t_2 := a[t_1]$
3. $t_3 := 4* i$
4. $t_4 := b[t_3]$
5. $t_5 := t_2*t_4$
6. $t_6 := prod+t_5$
7. $prod := t_6$
8. $t_7 := i+1$
9. $i := t_7$
10. if $i<=20$ goto (1)



Application of DAGs

- We can automatically detect common sub expressions.
- We can determine which identifiers have their values used in the block.
- We can determine which statements compute values that could be used outside the block

Transformations on Basic Blocks

- A number of transformations can be applied to a basic block without changing the set of expressions computed by the block.
- Two important classes of transformation are :
    - Structure-preserving transformations
    - Algebraic transformations

Structure preserving transformations:

- Common subexpression elimination:
  -
a : = b + c		a : = b + c
b : = a – d	⇨	b : = a - d
c : = b + c		c : = b + c
d : = a – d		d : = b
  - Since the second and fourth expressions compute the same expression, the basic block can be transformed as above.
- Dead-code elimination:
  - Suppose x is dead, that is, never subsequently used, at the point where the statement x : = y + z appears in a basic block. Then this statement may be safely removed without changing the value of the basic block

Structure preserving transformations:

- Renaming temporary variables:
  - A statement t : = b + c (t is a temporary) can be changed to u : = b + c (u is a new temporary) and all uses of this instance of t can be changed to u without changing the value of the basic block. Such a block is called a normal-form block.
- Interchange of statements:
  - Suppose a block has the following two adjacent statements:
    - t1 : = b + c
    - t2 : = x + y
    - We can interchange the two statements without affecting the value of the block if and only if neither x nor y is t1 and neither b nor c is t2

Algebraic transformations:

- Algebraic transformations can be used to change the set of expressions computed by a basic block into an algebraically equivalent set.
  - Examples:
  - x : = x + 0 or x : = x * 1 can be eliminated from a basic block without changing the set of expressions it computes.
  - The exponential statement x : = y * * 2 can be replaced by x : = y * y

Code generation algorithm: A SIMPLE CODE GENERATOR

- A code generator generates target code for a sequence of three-address statements and effectively uses registers to store operands of the statements.
    - For example: consider the three-address statement
    - a := b+c It can have the following sequence of codes:
        - ADD Rj, Ri Cost = 1 // if Ri contains b and Rj contains c (or)
        - ADD c, Ri Cost = 2 // if c is in a memory location (or)
        - MOV c, Rj Cost = 3 // move c from memory to Rj and add
        - ADD Rj, Ri

A code-generation algorithm

- **Register Descriptors:** A register descriptor is used to keep track of what is currently in each registers. The register descriptors show that initially all the registers are empty.
- **Address Descriptors -**An address descriptor stores the location where the current value of the name can be found at run time.
- **A code-generation algorithm:**

  1.The algorithm takes as input a sequence of three-address statements constituting a basic block. For each three-address statement of the form x : = y op z, perform the following actions:

  Invoke a function getreg to determine the location L where the result of the computation y op z should be stored.

  2. Consult the address descriptor for y to determine y’, the current location of y. Prefer the register for y’ if the value of y is currently both in memory and a register. If the value of y is not already in L, generate the instruction MOV y’ , L to place a copy of y in L.

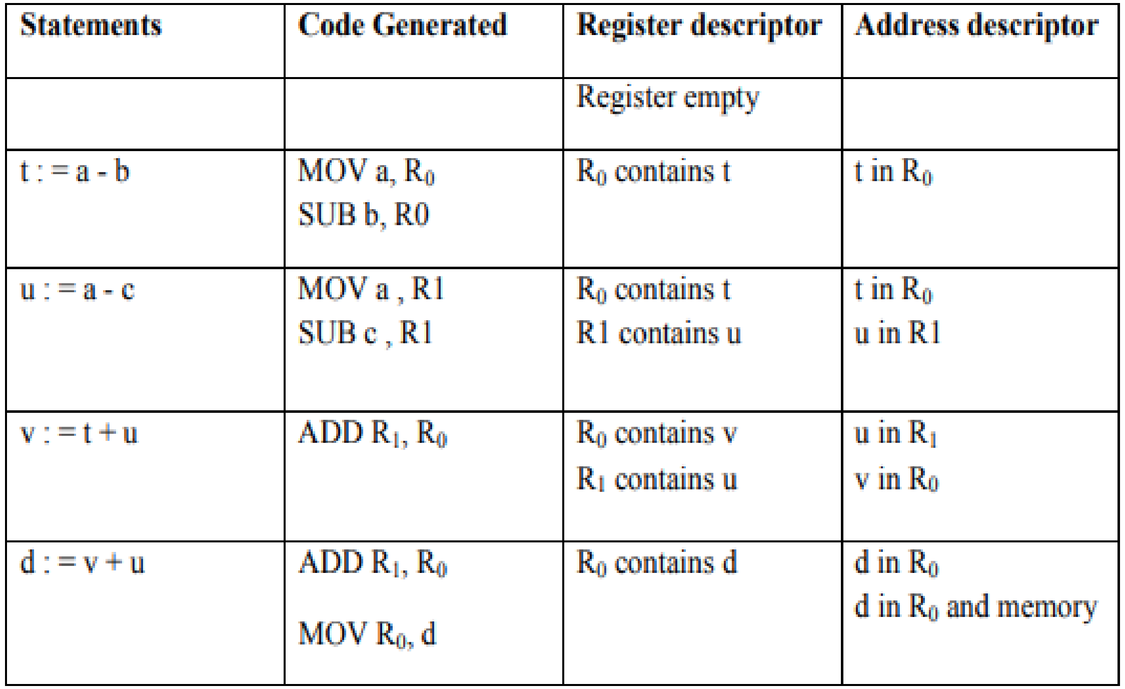
  3. Generate the instruction OP z’ , L where z’ is a current location of z. Prefer a register to a memory location if z is in both. Update the address descriptor of x to indicate that x is in location L. If x is in L, update its descriptor and remove x from all other descriptors.

  4. If the current values of y or z have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution of x : = y op z , those registers will no longer contain y or z.

Generating Code for Assignment Statements:

- The assignment d : = (a-b) + (a-c) + (a-c) might be translated into the following
- Three address code sequence:
    - t : = a – b
    - u : = a – c
    - v : = t + u
    - d : = v + u
        - with d live at the end.

Code sequence for the example is



Statements	Code Generated	Register descriptor	Address descriptor
		Register empty	
t : = a - b	MOV a, R_0 SUB b, R0	R_0 contains t	t in R_0
u : = a - c	MOV a , R1 SUB c , R1	R_0 contains t R1 contains u	t in R_0 u in R1
v : = t + u	ADD R_1, R_0	R_0 contains v R_1 contains u	u in R_1 v in R_0
d : = v + u	ADD R_1, R_0 MOV R_0, d	R_0 contains d	d in R_0 d in R_0 and memory

Rearranging the order

- The order in which computations are done can affect the cost of resulting object code.
- For example, consider the following basic block:
  - t1 : = a + b
  - t2 : = c + d
  - t3 : = e – t2
  - t4 : = t1 – t3
- Generated code sequence for basic block:
  - MOV a , R_0
  - ADD b , R_0
  - MOV c , R_1
  - ADD d , R_1
  - MOV R_0 , t_1
  - MOV e , R_0
  - SUB R_1 , R_0
  - MOV t_1 , R_1
  - SUB R_0 , R_1
  - MOV R_1 , t_4

- Rearranged basic block:
- Now t1 occurs immediately before t4.
    - t2 : = c + d
    - t3 : = e – t2
    - t1 : = a + b
    - t4 : = t1 – t3
- Revised code sequence:
    - MOV c , R0
    - ADD d , R0
    - MOV a , R0
    - SUB R0 , R1
    - MOV a , R0
    - ADD b , R0
    - SUB R1 , R0
    - MOV R0 , t4
- In this order, two instructions MOV R0 , t1 and MOV t1 , R1 have been saved.

Code Optimization

- A code optimizing process must follow the three rules given below:
  1. The output code must not, in any way, change the meaning of the program.
  2. Optimization should increase the speed of the program and if possible, the program should demand less number of resources.
  3. Optimization should itself be fast and should not delay the overall compiling process.
- Efforts for an optimized code can be made at various levels of compiling the process.
  1. At the beginning, users can change/rearrange the code or use better algorithms to write the code.
  2. After generating intermediate code, the compiler can modify the intermediate code by address calculations and improving loops.
  3. While producing the target machine code, the compiler can make use of memory hierarchy and CPU registers.

Code Optimization

- Optimization can broadly be categorized into two- Machine Independent and Machine dependent.
  - **Machine-independent optimization** phase tries to improve the intermediate code to obtain a better output. The optimized intermediate code does not involve any absolute memory locations or CPU registers.
  - **Machine-dependent optimization** is done after generation of the target code which is transformed according to target machine architecture. This involves CPU registers and may have absolute memory references.

Main Types of Code Optimization: High-level optimizations

1. High-level optimizations, intermediate level optimizations, and low-level optimizations - High-level optimization is a language dependent type of optimization that operates at a level in the close vicinity of the source code. High-level optimizations include in lining where a function call is replaced by the function body and partial evaluation which employs reordering of a loop, alignment of arrays, padding, layout, and elimination of tail recursion.

Most of the code optimizations performed fall under intermediate code optimization which is language independent. This includes:

1. **The elimination of common sub-expressions** – This type of compiler optimization probes for the instances of identical expressions by evaluating to the same value and researches whether it is valuable to replace them with a single variable which holds the computed value.

2**. Constant propagations** – Here, expressions which can be evaluated at compile time are identified and replaced with their values.

3. **Jump threading** – This involves an optimization of jump directly into a second one. The second condition is eliminated if it is an inverse or a subset of the first which can be done effortlessly in a single pass through the program. Acyclic chained jumps are followed till the compiler arrives at a fixed point.

4. **Loop invariant code motion** – This is also known as hoisting or scalar promotion. A loop invariant contains expressions that can be taken outside the body of a loop without any impact on the semantics of the program. The above-mentioned movement is performed automatically by loop invariant code motion.

5. **Dead code elimination** – Here, as the name indicates, the codes that do not affect the program results are eliminated. It has a lot of benefits including reduction of program size and running time. It also simplifies the program structure. Dead code elimination is also known as DCE, dead code removal, dead code stripping, or dead code strip.

6. **Strength reduction** – This compiler optimization replaces expensive operations with equivalent and more efficient ones, but less expensive. For example, replace a multiplication within a loop with an addition

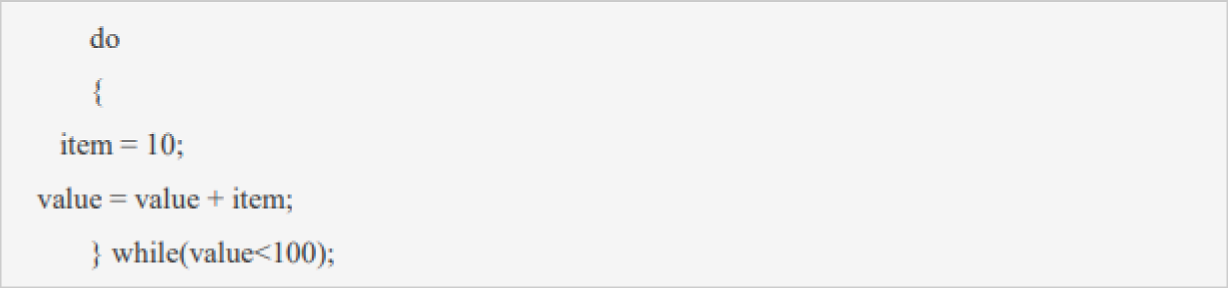
Main Types of Code Optimization: Low-level Optimization

2. Low-level Optimization is highly specific to the type of architecture. This includes the following:

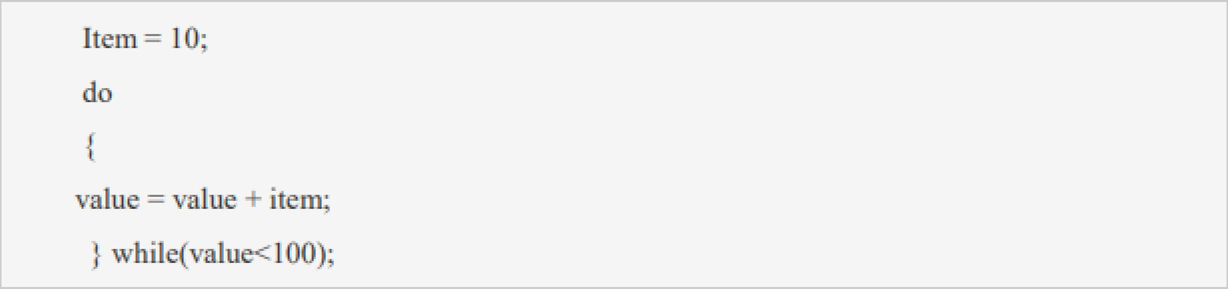
1. **Register allocation** – Here, a big number of target program variables are assigned to a small number of CPU registers. This can happen over a local register allocation or a global register allocation or an inter-procedural register allocation.
2. **Instruction Scheduling** – This is used to improve an instruction level parallelism that in turn improves the performance of machines with instruction pipelines. It will not change the meaning of the code but rearranges the order of instructions to avoid pipeline stalls. Semantically ambiguous operations are also avoided.
3. **Floating-point units utilization** – Floating point units are designed specifically to carry out operations of floating point numbers like addition, subtraction, etc. The features of these units are utilized in low-level optimizations which are highly specific to the type of architecture.
4. **Branch prediction** – Branch prediction techniques help to guess in which way a branch functions even though it is not known definitively which will be of great help for the betterment of results.
5. **Peephole and profile-based optimization** – Peephole optimization technique is carried out over small code sections at a time to transform them by replacing with shorter or faster sets of instructions. This set is called as a peephole. Profile-based optimization is performed on a compiler which has difficulty in the prediction of likely outcome of branches, sizes of arrays, or most frequently executed loops. They provide the missing information, enabling the compilers to decide when needed.

Machine Independent Optimization

- In this optimization, the compiler takes in the intermediate code and transforms a part of the code that does not involve any CPU registers and/or absolute memory locations.
- For example: should not only save the CPU cycles, but can be used on any processor.



```
      do
      {
   item = 10;
value = value + item;
      } while(value<100);
```



```
  Item = 10;
  do
  {
 value = value + item;
   } while(value<100);
```

Machine Dependent Optimization

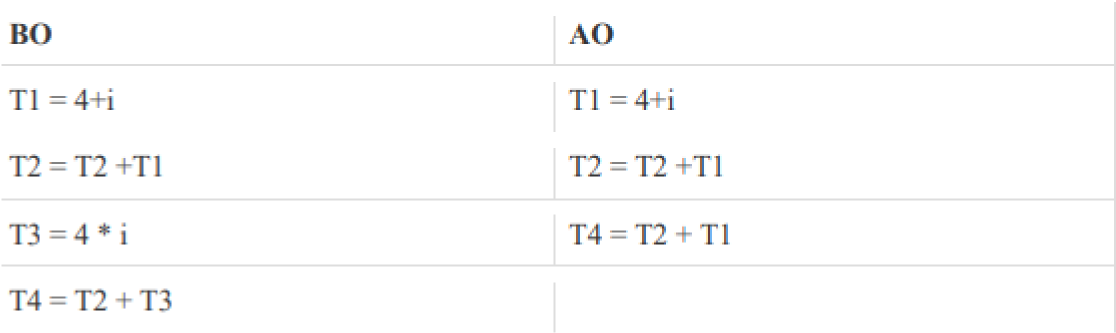
- Machine-dependent optimization is done after the target code has been generated and when the code is transformed according to the target machine architecture.
    - It involves CPU registers and may have absolute memory references rather than relative references.
    - Machine-dependent optimizers put efforts to take maximum advantage of memory hierarchy. Machine independence includes two types

        i. Function Preserving

        ii. Loop optimization

Function preserving: Common Sub Expression Elimination

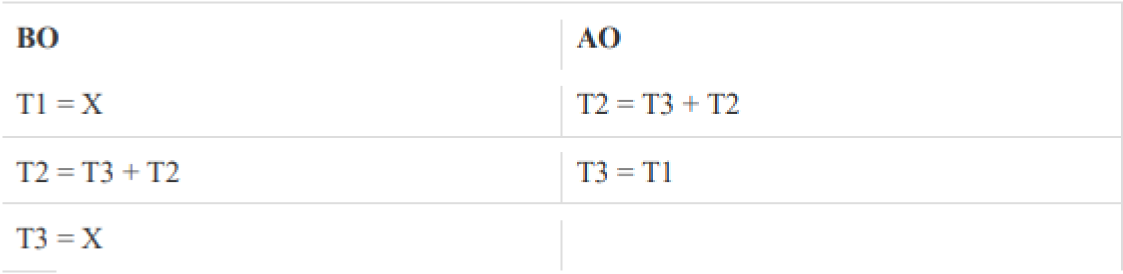
The expression that produces the same results should be removed out from the code
Example:



BO	AO
T1 = 4+i	T1 = 4+i
T2 = T2 +T1	T2 = T2 +T1
T3 = 4 * i	T4 = T2 + T1
T4 = T2 + T3	

Constant folding and Copy Propagation

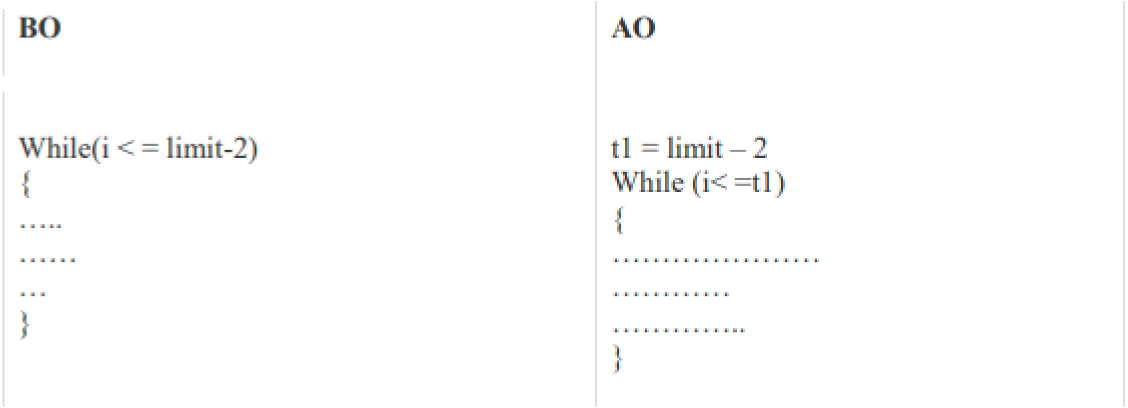
- Constant folding- If expression generates a constant value then instead of performing its calculation again and again we calculate it once and assign it.
- Copy Propagation -In this propagation a F value is been send to G and G value is been send to H We can eliminate G variable directly assigning the value of F to H.



BO	**AO**
T1 = X	T2 = T3 + T2
T2 = T3 + T2	T3 = T1
T3 = X	

Dead Code Elimination & Loop Optimization

- Dead code is one or more than one code statements, which are:
    - Either never executed or unreachable,
    - Or if executed, their output is never used. Thus, dead code plays no role in any program operation and therefore it can simply be eliminated.
- Loop Optimization : perform optimization on loops.
    - Code Motion It specifies on a condition if we perform some operations to be carried out and then compare for a condition. Instead of that perform the calculation outside the loop and assign a value in the calculation.

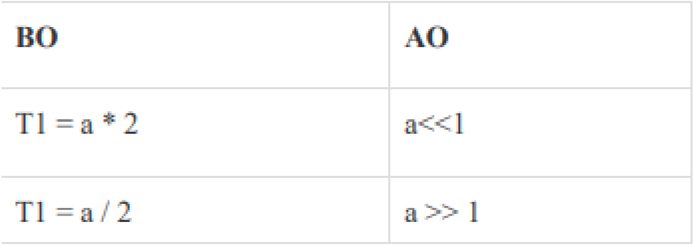


BO	**AO**

While(i < = limit-2)
{
…..
……
…
}
	t1 = limit – 2
	While (i< =t1)
	{
	…………………
	…………
	…………..
	}

Strength Reduction

- Strength Reduction It specifies the operators such as multiplication and division can be replaced by a addition and subtraction respectively. The multiplication operator can be easily replaced by left shift operator a<>1 operator.

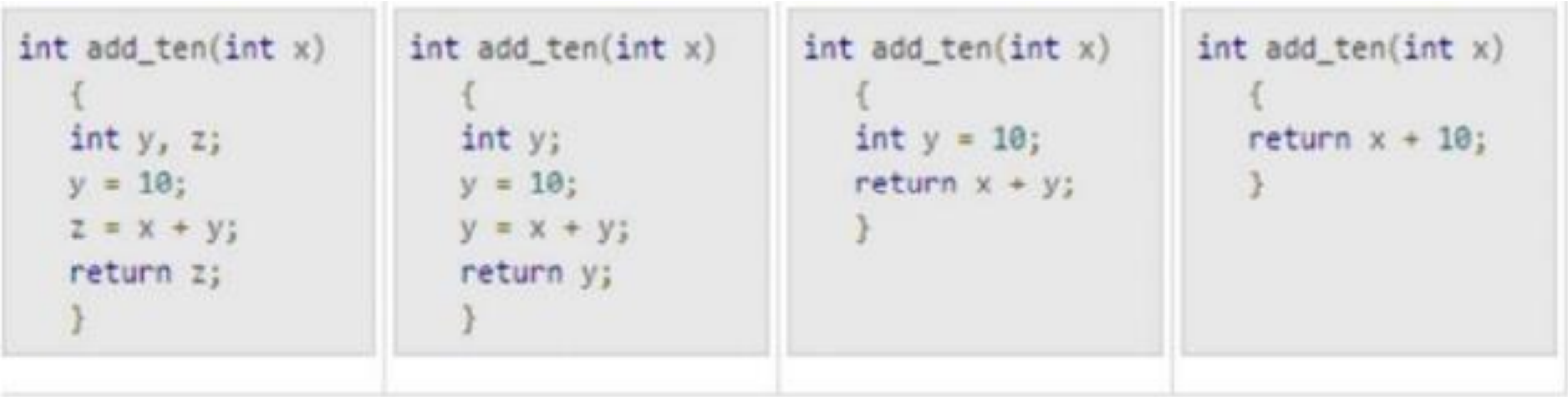


BO	AO
T1 = a * 2	a<<1
T1 = a / 2	a >> 1

Peephole Optimization

- This optimization technique works locally on the source code to transform it into an optimized code. By locally, we mean a small portion of the code block at hand.
- These methods can be applied on intermediate codes as well as on target codes.
- A bunch of statements is analyzed and are checked for the following possible optimization:

1. **Redundant instruction elimination :**
    - At source code level, the following can be done by the user:



```
int add_ten(int x)
   {
   int y, z;
   y = 10;
   z = x + y;
   return z;
   }
```

```
int add_ten(int x)
   {
   int y;
   y = 10;
   y = x + y;
   return y;
   }
```

```
int add_ten(int x)
   {
   int y = 10;
   return x + y;
   }
```

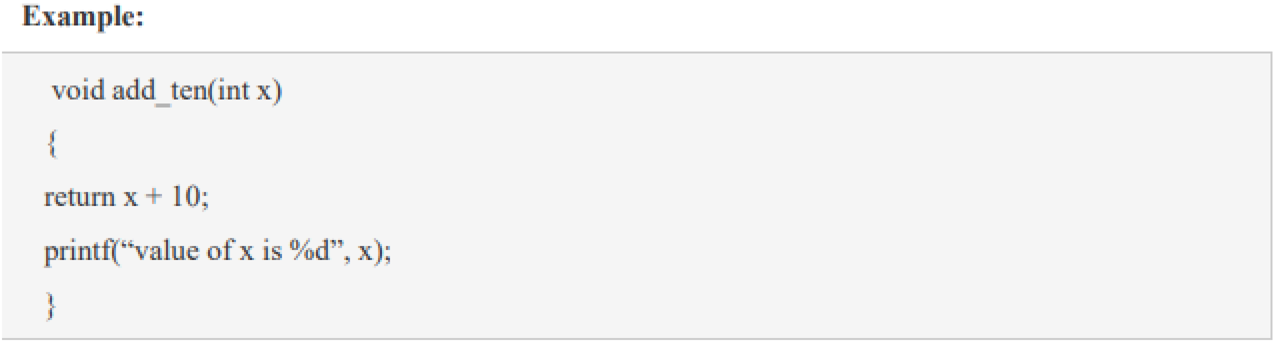
```
int add_ten(int x)
   {
   return x + 10;
   }
```

- At
  inst
- For
    - MOV R0, R1
- We can delete the first instruction and re-write the sentence as:
    - MOV x, R1

Unreachable code

2.Unreachable code is a part of the program code that is never accessed because of programming constructs.

- Programmers may have accidently written a piece of code that can never be reached.



Example:

```
 void add_ten(int x)
{
return x + 10;
printf("value of x is %d", x);
}
```

In this code segment, the printf statement will never be executed as the program control returns back before it can execute, hence printf can be removed.

Strength reduction & Algebraic expression simplification

3. Strength reduction :

- There are operations that consume more time and space. Their ‘strength’ can be reduced by replacing them with other operations that consume less time and space, but produce the same result.
- For example, x * 2 can be replaced by x << 1, which involves only one left shift.
- Though the output of a * a and a2 is same, a2 is much more efficient to implement

4. Algebraic expression simplification :

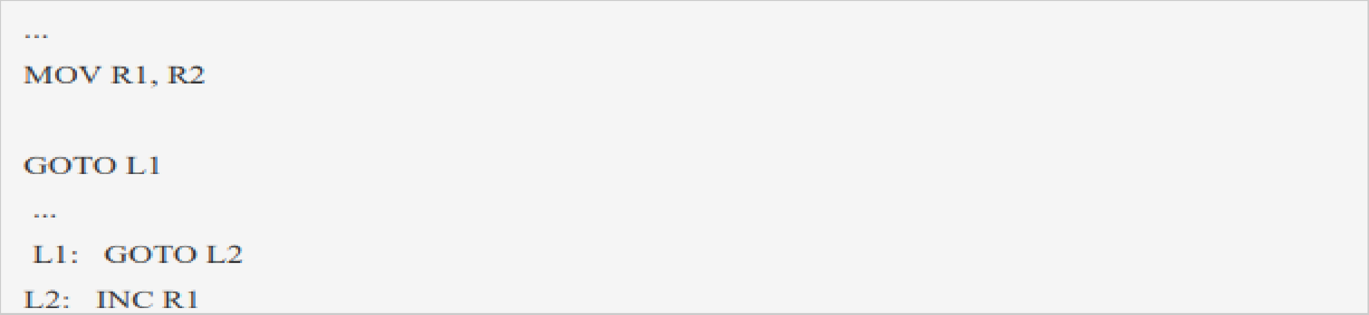
- There are occasions where algebraic expressions can be made simple.

For example, the expression a = a + 0 can be replaced by a itself and the expression a = a + 1 can simply be replaced by INC a.

Flow of control optimization

5. Flow of control optimization

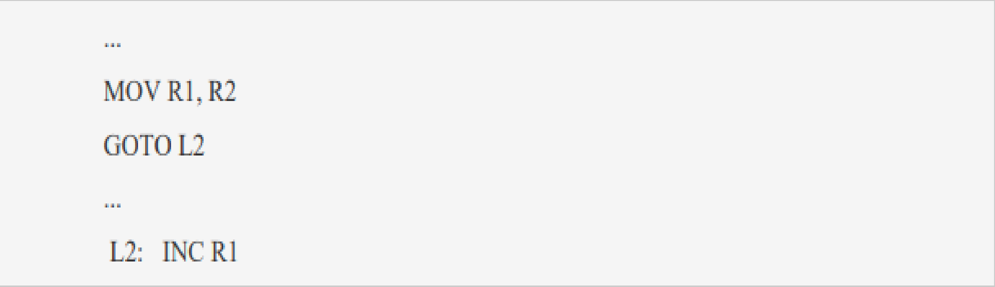
- There are instances in a code where the program control jumps back and forth without performing any significant task.
- These jumps can be removed. Consider the following chunk of code:



```
...
MOV R1, R2

GOTO L1
 ...
 L1:   GOTO L2
L2:   INC R1
```

In this code, label L1 can be removed as it passes the control to L2. So instead of jumping to L1 and then to L2, the control can directly reach L2, as shown below:



```
...
MOV R1, R2
GOTO L2
...
 L2:   INC R1
```