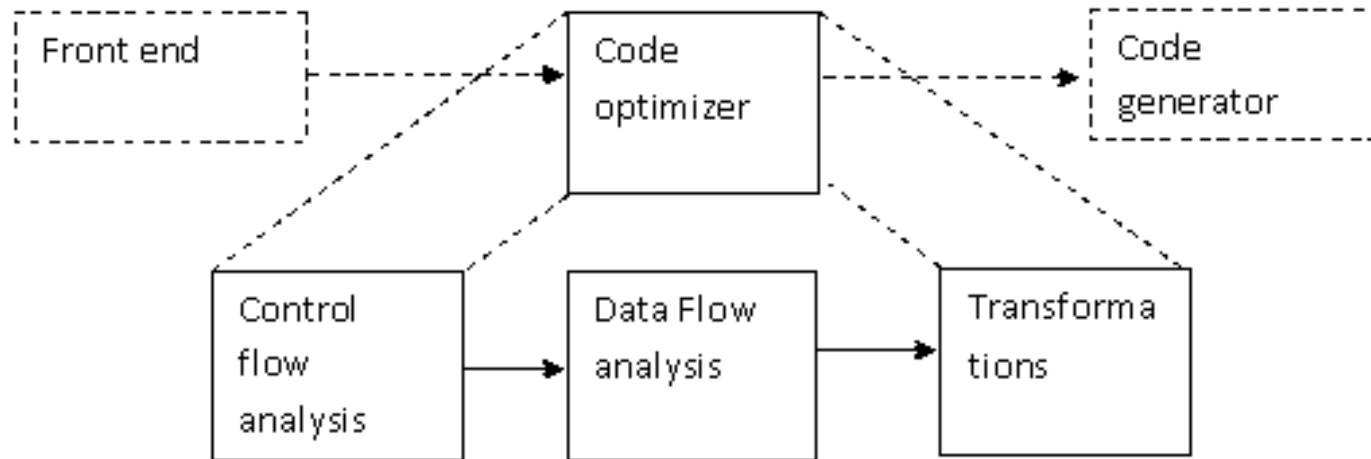


CODE OPTIMIZATION

Code Optimization

- To improve the intermediate code.
- Faster running machine code will result.

Block diagram of organization of code optimizer



Basic Block

- Basic block contains a sequence of statements. The flow of control enters at the beginning and leave at the end without any halt (except may be the last instruction of the block).
- The following sequence of three address statements forms a basic block:

t1:= x * x

t2:= x * y

t3:= 2 * t2

t4:= t1 + t3

t5:= y * y

t6:= t4 + t5

Basic block construction:

Algorithm: Partition into basic blocks

Input: It contains the sequence of three address statements

Output: it contains a list of basic blocks with each three address statement in exactly one block

Method: First identify the leader in the code.

The rules for finding leaders are as follows:

- The first statement is a leader.
- Statement L is a leader if there is a conditional or unconditional goto statement like: if....goto L or goto L (Statement L is the target of goto)
- Instruction L is a leader if it immediately follows a goto or conditional goto statement like: if goto B or goto B

For each leader, its basic block consists of the leader and all statement up to the next leader. It doesn't include the next leader or end of the program.

- Consider the following source code for dot product of two vectors a and b of length 10:

```
begin
  prod := 0;
  i := 1;
  do begin
    prod := prod + a[i] * b[i];
    i := i + 1;
  end
  while i <= 10
end
```

(1) prod := 0

(2) i := 1

(3) t1 := 4 * i

(4) t2 := a[t1]

(5) t3 := 4 * i

(6) t4 := b[t3]

(7) t5 := t2 * t4

(8) t6 := prod + t5

(9) prod := t6

(10) t7 := i + 1

(11) i := t7

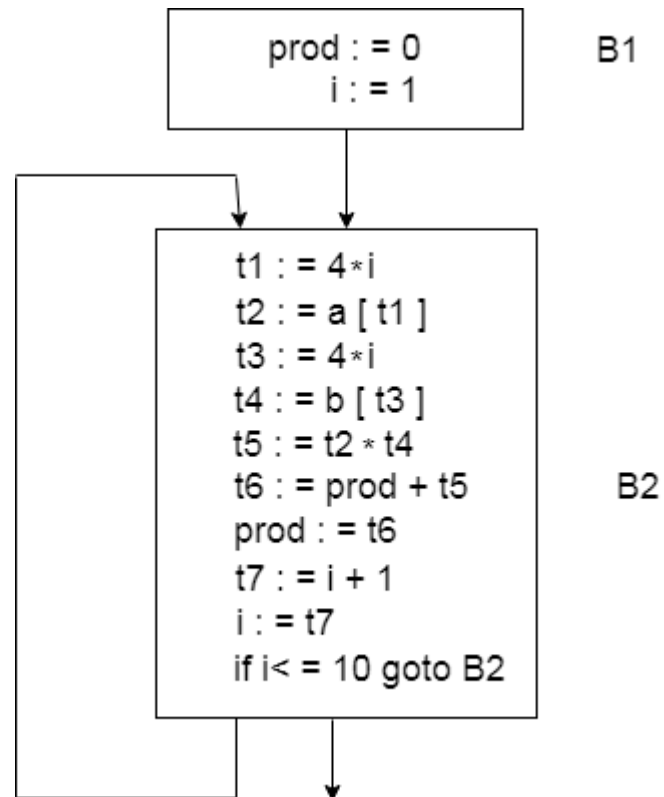
(12) if i <= 10 goto (3)

Basic block B1 contains the statement (1) to (2)

Basic block B2 contains the statement (3) to (12)

Flow Graph

- Flow graph is a directed graph. It contains the flow of control information for the set of basic block.
- A control flow graph is used to depict that how the program control is being passed among the blocks. It is useful in the loop optimization.
- Flow graph for the vector dot product is given as follows:



- Block B1 is the initial node. Block B2 immediately follows B1, so from B1 to B2 there is an edge.
- The target of jump from last statement of B2 is the first statement B2, so from B2 to B2 there is an edge.
- B2 is a successor of B1 and B1 is the predecessor of B2.

Optimization of Basic Blocks / Principle source of Optimization

- Optimization process can be applied on a basic block. While optimization, we don't need to change the set of expressions computed by the block.
- There are two type of basic block optimization. These are as follows:
 1. Structure-Preserving Transformations
 2. Algebraic Transformations

1. Structure preserving transformations:

- The primary Structure-Preserving Transformation on basic blocks is as follows:
 - ✓ Common sub-expression elimination
 - ✓ Dead code elimination
 - ✓ Renaming of temporary variables
 - ✓ Interchange of two independent adjacent statements

Common sub-expression elimination:

- In the common sub-expression, you don't need to be computed it over and over again. Instead of this you can compute it once and kept in store from where it's referenced when encountered again.
 1. $a := b + c$
 2. $b := a - d$
 3. $c := b + c$
 4. $d := a - d$
- In the above expression, the second and forth expression computed the same expression. But we can't consider first and third expression because b value is changing meanwhile. So the block can be transformed as follows:
 1. $a := b + c$
 2. $b := a - d$
 3. $c := b + c$
 4. $d := b$

Dead-code elimination

- It is possible that a program contains a large amount of dead code.
- This can be caused when a variable once declared and defined, it is not used anywhere in the program.
- Suppose the statement $x := y + z$ appears in a block and x is a dead symbol that means it will never subsequently be used. Then without changing the value of the basic block you can safely remove this statement.

```
#define flag 0
-----
-----
if (flag) {
do something ; }
```

Note : The flag has been set to zero. So the true block will not be executed at least once because the code is dead.

Renaming temporary variables

- A statement $t := b + c$ can be changed to $u := b + c$ where t is a temporary variable and u is a new temporary variable. All the instance of t can be replaced with the u without changing the basic block value.

Interchange of statement

- Suppose a block has the following two adjacent statements:
 1. $t1 := b + c$
 2. $t2 := x + y$
- These two statements can be interchanged without affecting the value of block when value of $t1$ does not affect the value of $t2$.

Algebraic transformations:

- In the algebraic transformation, we can change the set of expression into an algebraically equivalent simplified/cheaper expression set.
- The expression $x := x + 0$ or $x := x * 1$ can be eliminated from a basic block without changing the set of expression.
- Constant folding

If the value of the variable is identified as constant during compile time, then it is replaced by the constant instead is known as constant folding.

Eg.

```
int i=5;  
---  
---  
k= i+j;  
----  
----  
Before
```



```
int i=5;  
----  
----  
k = 5 +j;  
----  
----  
After
```

- Reduction in strength
 - The strength of certain operators is higher than other. For instance strength of $*$ is higher than $+$
 - In strength reduction technique the higher strength operators can be replaced by lower strength operators.
 - Eg.
 - $a ** 2 \rightarrow a * a$
 - $a * 2 \rightarrow a + a$
- Copy propagation
 - Variable propagation

An assignment of the form $a = b$ followed by any assignments of 'a' will be replaced by 'b'.

$$x=pi$$

$$A=x*r*r \rightarrow A=pi*r*r$$
 - Constant propagation

$$x=3$$

$$y=x+k \rightarrow y=3+k$$

Direct Acyclic Graph (DAG):

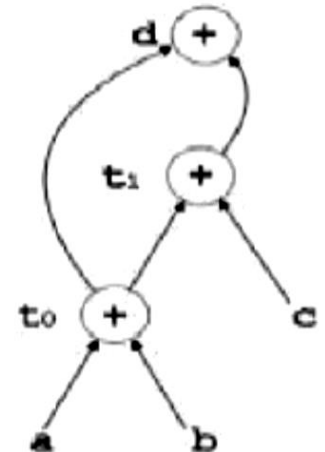
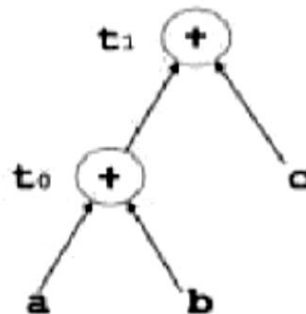
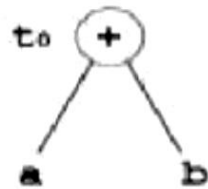
DAG is a tool that depicts the structure of basic blocks, helps to see the flow of values flowing among the basic blocks, and offers optimization too. DAG provides easy transformation on basic blocks. DAG can be understood here:

- 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.

$$t_0 = a + b$$

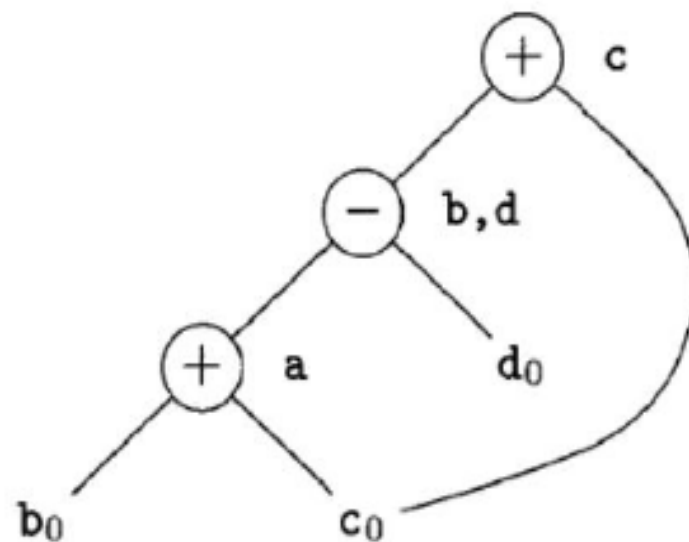
$$t_1 = t_0 + c$$

$$d = t_0 + t_1$$



Finding Local Common Subexpressions

$a = b + c$
 $b = a - d$
 $c = b + c$
 $d = a - d$

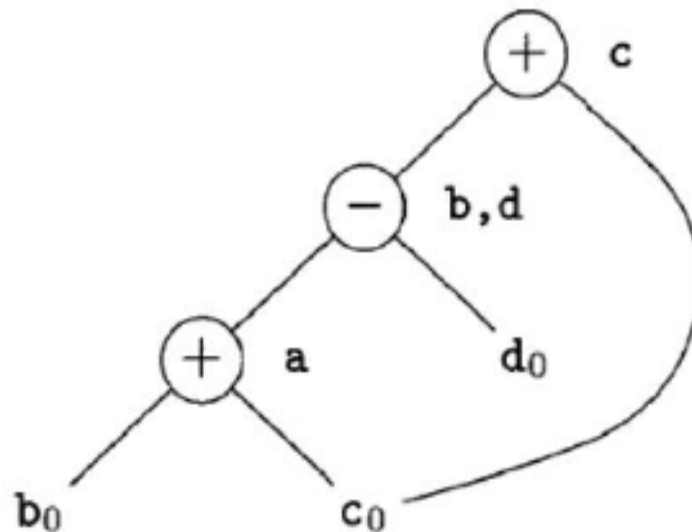


Common subexpressions can be detected by noticing, as a new node *M* is about to be added, whether there is an existing node *N* with the same children, in the same order, and with the same operator.


```

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

```



Since there are **only three nonleaf nodes** in the DAG, the basic block in can be replaced by a block with only **three statements**.

```

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

```

```

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

```

- If **b** is not live on exit from the block, then we do not need to compute **b** variable, and can use **d** to receive the value
- 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

Loop Optimization

Much of the compiling time is spend in inner loops. The running time can be minimized if the number of instructions inside the loop can be minimized by moving the code outside the loop.

- Code motion
- Induction variable elimination
- Reduction in strength
- Loop merging
- Loop unrolling


Code motion

- In code motion we move code outside a loop.
- A loop invariant computation is one that computes the same value every time a loop is executed. Therefore , moving such a computation outside the loop leads to a reduction in the execution time.
- Identification of loop invariant computation requires the detection of loops in the program.
- Whether a loop exists in the program or not depends on the program's control flow ,therefore, requiring a control flow analysis.
- For loop detection, a graphical representation, called a “program flow graph” shows how the control is flowing in the program and how the control is being used.

Example 1

- Move loop invariant computation out side the loops

```
• While ( i < 100)
{
    *p = x / y + 1
    i = i + 1
}
```




```
    t = x / y
    while ( i < 100)
    {
        *p = t + 1
        i = i + 1
    }
```

Example 2

```
while ( i <= limit - 2) /*statement does not change limit */
```

```
{
    -----
}
```



```
    t= limit - 2
    while( i <= t) /*statement does not change
    limit */
    {
        -----
    }
```

Induction variable elimination

- Typical loop optimization strategies focus on identifying variables that are incremented by fixed amounts with each iteration called induction variable. These include loop control variables and other variables that depend on the loop control variables in a fixed ways

```
i1=0  
i2=0  
For(i=0;i<10;i++)  
{  
  A[i1++]=B[i2++]  
}
```



```
For(i=0;i<10;i++)  
{  
  A[i]=B[i]  
}
```

-Reduction in strength

- The strength of certain operators is higher than other. For instance strength of * is higher than +
- In strength reduction technique the higher strength operators can be replaced by lower strength operators.
- Eg.


$$a ** 2 \rightarrow a * a$$

$$a * 2 \rightarrow a + a$$

Loop merging/fusion/jamming

- Loop jamming is a technique that merges the bodies of two loops if the two loops have the same number of iterations and they use the same indices. This eliminates the test of one loop. For example, consider the following loop

```
for ( j = 0 ; j < 10 ; j++ )  
    x [ j ] = j;  
for ( i = 0 ; i < 10 ; i++ )  
    a [ i ] = i-1;
```



```
for ( i = 0 ; i < 10 ; i++ ) {  
    x[i]=i  
    a [ i ] = i-1;  
}
```

Loop unrolling

- Loop unrolling involves replicating the body of the loop to reduce the required number of tests if the number of iterations are constant. For example consider the following loop;

```
for ( i = 0 ; i < 200 ; i ++ )
```

```
    calc()
```



```
for ( i = 0 ; i < 100 ; i ++ )
```

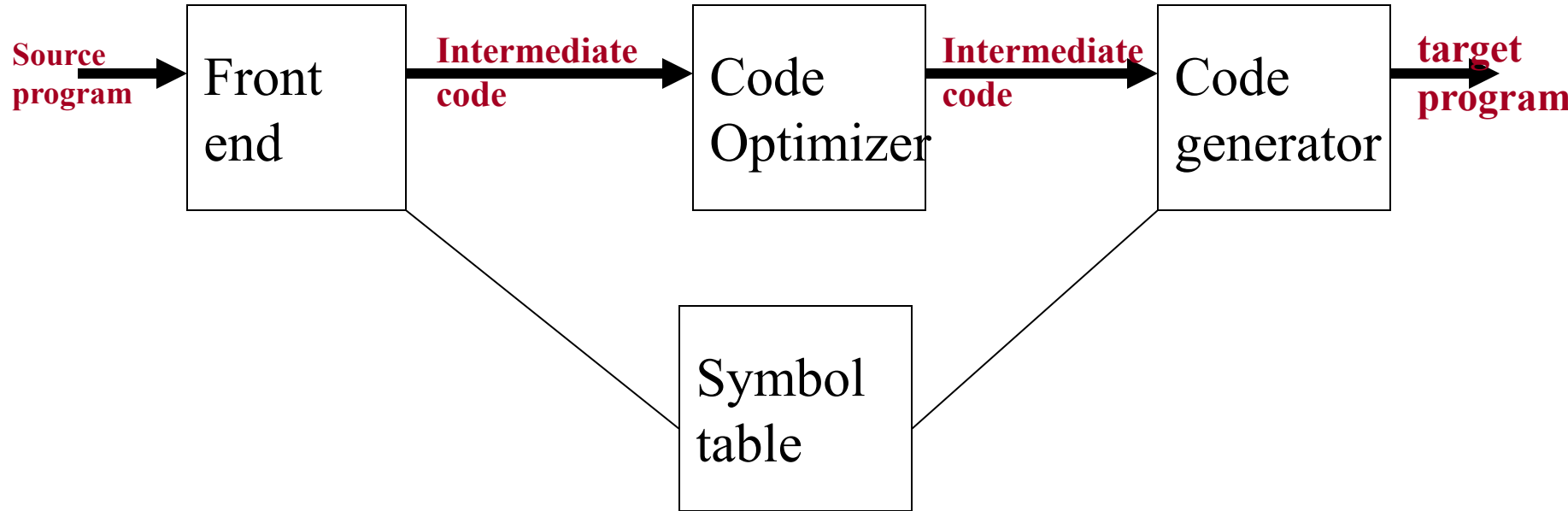
```
{
```

```
    calc()
```

```
    calc()
```

```
}
```


Introduction



Position of code generator

- ✓ output code must be correct
- ✓ output code must be of high quality
- ✓ code generator should run efficiently

Issues in the Design of a Code Generator

- Input to the code generator
- Target language
- Memory management
- Instruction Selection
- Register allocation
- Choice of Evaluation order

Input to the Code Generator

- It consists of the intermediate representation of the source program, together with information in the symbol table.
 - ie. used to determine the runtime addresses of the data objects.
- Intermediate representations may be
 - Linear representation - Postfix notations
 - Three address representations - quadruples
 - Graphical representation - Syntax tree , DAG

Target Programs

- The output of the code generator is the target program.
- Target program may be
 - Absolute machine language
 - It can be placed in a fixed location of memory and immediately executed
 - Re-locatable machine language
 - Subprograms to be compiled separately
 - A set of re-locatable object modules can be linked together and loaded for execution by a linker
 - Assembly language
 - Easier

Memory Management

- Mapping names in the source program to addresses of the data objects in runtime memory. Co-operatively by the front end and the code generator.
- A name in a three address statement refers to a symbol table entry for the name.
- From the symbol table information, a relative address can be determined for the name in a data area for the procedure.

Instruction Selection

- The nature of the instruction set of the target machine determines the difficulty of the instruction selection.
- Uniformity and completeness of the instruction set are important
- Instruction speeds is also important

– Say, $x = y + z$

Mov y, R0

Add z, R0

Mov R0, x

$a = b + c$
 $d = a + e$

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

Instruction Selection

- The quality of the generated code is determined by its speed and size.
- Cost difference between the different implementation may be significant.
 - Say $a = a + 1$
 - Mov a, R0
 - Add #1, R0
 - Mov R0, a
 - If the target machine has increment instruction (INC), we can write
 - inc a**

Register Allocation

- Instructions involving register operands are usually shorter and faster than those involving operands in memory.
- Efficient utilization of register is particularly important in code generation.
- The use of register is subdivided into two sub problems
 - During register allocation, we select the set of variables that will reside in register at a point in the program.
 - During a subsequent register allocation phase, we pick the specific register that a variable will reside in.

Choice of evaluation order

- The order in which computations are performed can affect the efficiency of the target code.
- Some computation order require fewer register to hold intermediate results than others.

TAC \Rightarrow

$$\begin{aligned}t_1 &= a + b \\t_2 &= c + d \\t_3 &= e * t_2 \\t_4 &= t_1 - t_3\end{aligned}$$

MOV	R0, a
ADD	R0, b
MOV	R0, t1
ADD	d, R1
MOV	e, R0
MUL	R1, R0
MOV	t1, R1
Sub	R0, R1
MOV	R1, t4

reorder

$$\begin{aligned}t_2 &= c + d \\t_3 &= e * t_2 \\t_1 &= a + b \\t_4 &= t_1 - t_3\end{aligned}$$

MOV	c, R0
ADD	d, R0
MOV	e, R1
MUL	R0, R1
MOV	a, R0
ADD	t1, R0
Sub	R1, R0
MOV	R0, t4

Target Machine

- Byte addressable with 4 bytes per word
- It has n registers R_0, R_1, \dots, R_{n-1}
- Two address instructions of the form
opcode source, destination
- Usual opcodes like move, add, sub etc.

- **Addressing modes**

MODE	FORM	ADDRESS	Added Cost
Absolute	M	M	1
register	R	R	0
index	c(R)	c+cont(R)	1
indirect register	*R	cont(R)	0
indirect index	*c(R)	cont(c+cont(R))	1
literal	#c	c	1

Target Machine

Instruction costs :

- Instruction cost = 1+cost for source and destination address modes.
 - Address modes involving registers have cost zero.
 - Address modes involving memory location or literal have cost one.
- For example : MOV R0, R1 copies the contents of register R0 into R1. It has cost one, since it occupies only one word of memory.
- The three-address statement **a := b + c** can be implemented by many different instruction sequences :

i) MOV b, R0

ADD c, R0 cost = 6

MOV R0, a

ii) MOV b, a

ADD c, a cost = 6

iiii) Assuming R0, R1 and R2 contain the addresses of a, b, and c :

MOV *R1, *R0

ADD *R2, *R0 cost = 2

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

(or)

ADD c, Ri Cost = 2

(or)

MOV c, Rj Cost = 3

ADD Rj, Ri

Register and Address 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.
- An address descriptor stores the location where the current value of the name can be found at run time.

A code-generation algorithm:

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

1. Invoke a function `getreg` to determine the location L where the result of the computation $y \text{ op } z$ 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 $x := y \text{ op } z$, 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:

Code sequence for the example is:


```

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

Generating Code for Indexed Assignments

The table shows the code sequences generated for the indexed assignments $a := b[i]$ and $a[i] := b$

Statements	Code Generated	Cost
$a := b[i]$	MOV b(Ri), R	2
$a[i] := b$	MOV b, a(Ri)	3

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 that try to improve 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.

Peephole Optimization

- **Typical optimization technique**
 - Redundant – instruction elimination
 - Unreachable code elimination
 - Flow of control optimization
 - Algebraic simplification
 - Use of machine idioms

Peephole Optimization

- Redundant – instruction elimination

The instruction sequence

MOV R0, a // store (1)

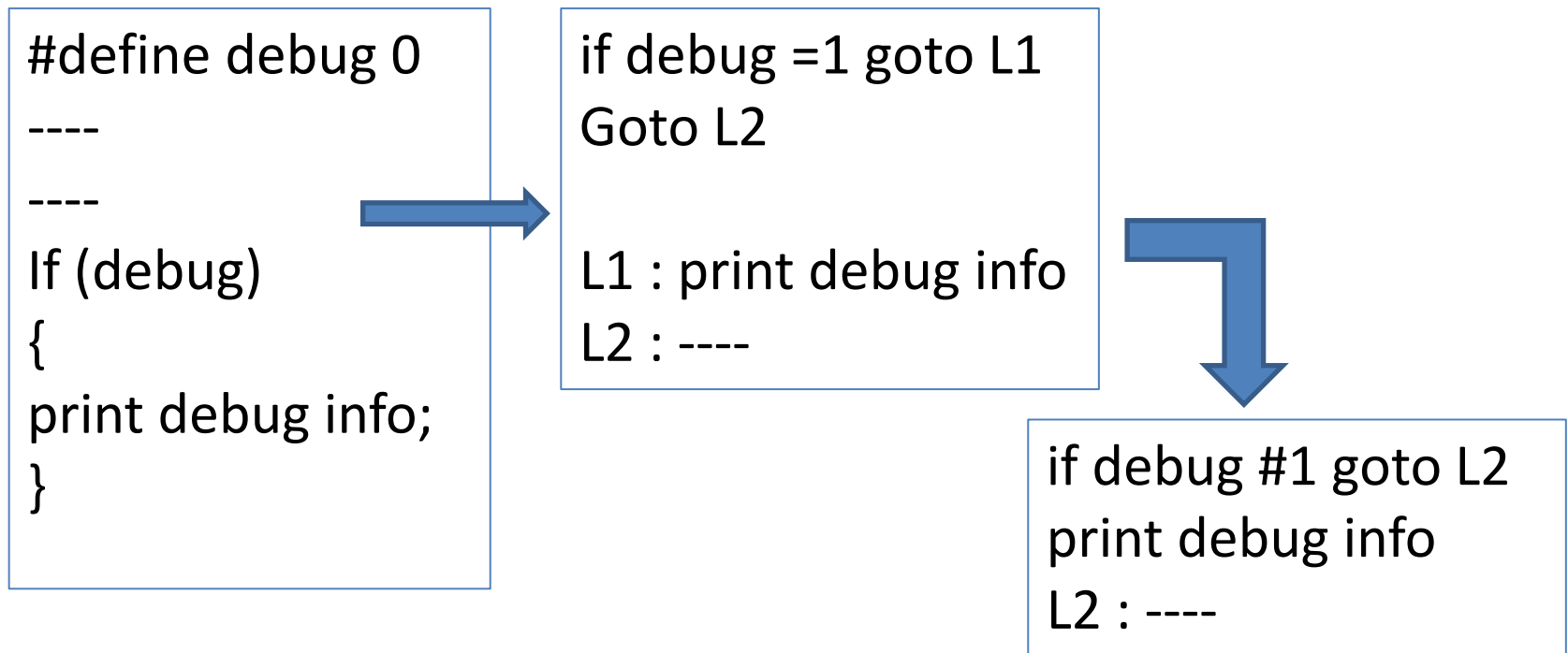
MOV a, R0 // load (2)

- 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).

Peephole Optimization

-Unreachable code elimination

A code which cannot be executed once during execution is called as unreachable code.



- the code here is unreachable so we can eliminate the code

Peephole Optimization

-Flow of control optimization

The unnecessary jumps like jumps to jumps, jumps to conditional jumps, conditional jumps to jumps, can be eliminated.

Eg1.

goto L1

L1: goto L2

L2 : Print S

goto L2

L2 : Print S



Eg2.

goto L1

L1: if a>b goto L2

L3:

if a>b goto L2
goto L3



Peephole Optimization

-Algebraic simplification

The Algebraic identifiers occur in the three address code can be simplified as follows without changing the meaning of the code.

Eg.

$$x = x + 0$$

$$x = x * 1$$

Peephole Optimization

-Use of machine idioms

- Some operations are replaced by hardware instructions which implements efficiently.
- usage of hardware instructions reduces execution time.

-Eg.

$i = i + 1$ **➔** `INC R1`