

Code Optimization

Overview and Examples

Code Optimization

❖ Why

- ❑ Reduce programmers' burden

- Allow programmers to concentrate on high level concept
- Without worrying about performance issues

❖ Target

- ❑ Reduce execution time

- ❑ Reduce space

The optimization can be classified depending on

→ Level of code

Design level

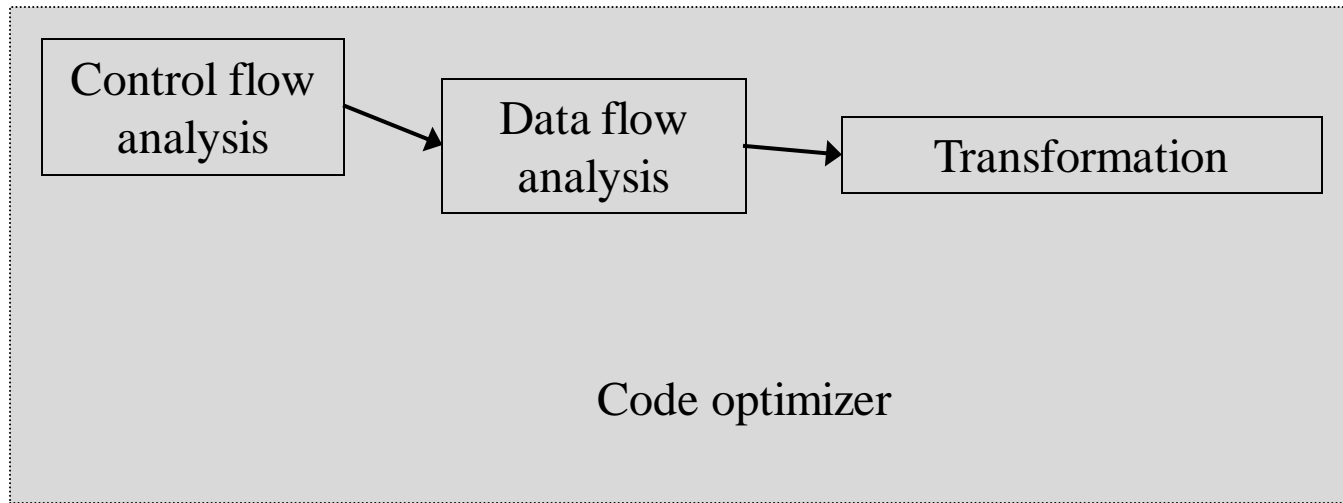
- Source code level
- Compile level
- Assembly level

→ Programming languages

- Machine independent
- Machine dependent

→ Scope

- Local
- Global



Flow graph

-----Graphical representation of three address code

Nodes -represent a single block

Edges -represent flow of control

Basic block

Is a set of consecutive statements tht are executed sequentially.

Procedure to identify basic block

Given three address code identify leader statements and group the leader statement with the statement up to the next leader.

Rules to identify Leader statement:

- 1.First statement in the program is Leader
- 2.Any statement that is target of a conditional or unconditional statement is a leader statement
- 3.Any statement that immediately follows a conditional or unconditional statement is a leader statement

Example 3: Identify the basic blocks for the following code fragment.

```
main ( )
{
    int i = 0, n = 10;
    int a[n];
    while ( i <= (n-1))
    {
        a[i] = i * i;
        i=i+1;
    }
    return;
}
```

The three address code for the initialize function is as follows:

- (1). $i := 0$
- (2). $n := 10$
- (3). $t_1 := n - 1$
- (4). If $i > t_1$ goto (12)
- (5). $t_2 := i * i$
- (6). $t_3 := 4 * i$
- (7). $t_4 := a[t_3]$
- (8). $t_4 := t_2$
- (9). $t_5 := i + 1$
- (10). $i := t_5$
- (11). goto (3)
- (12). return

Identifying leader statements in the above three address code

Statement (1) is leader using rule 1

Statement (3) and (12) are leader using rule 2

Statement (4) and (12) are leaders using rule 3

- | | |
|----------------------------|------------|
| 1. $i := 0$ | → Leader 1 |
| 2. $n := 10$ | |
| 3. $t_1 := n - 1$ | → Leader 2 |
| 4. If $i > t_1$ go to (12) | → Leader 3 |
| 5. $t_2 := i * i$ | |
| 6. $t_3 := 4 * i$ | |
| 7. $t_4 := a[t_3]$ | |
| 8. $t_4 := t_2$ | |
| 9. $t_5 := i + 1$ | |
| 10. $i := t_5$ | |
| 11. go to (3) | → Leader 4 |
| 12. Return | |

Code Optimization

- Basic block 1 includes statements (1) and (2)
- Basic block 2 includes statements (3) and (4)
- Basic block 3 includes statements (5)–(11)
- Basic block 4 includes statement (12)
- Basic blocks are shown in Figure 10.2.

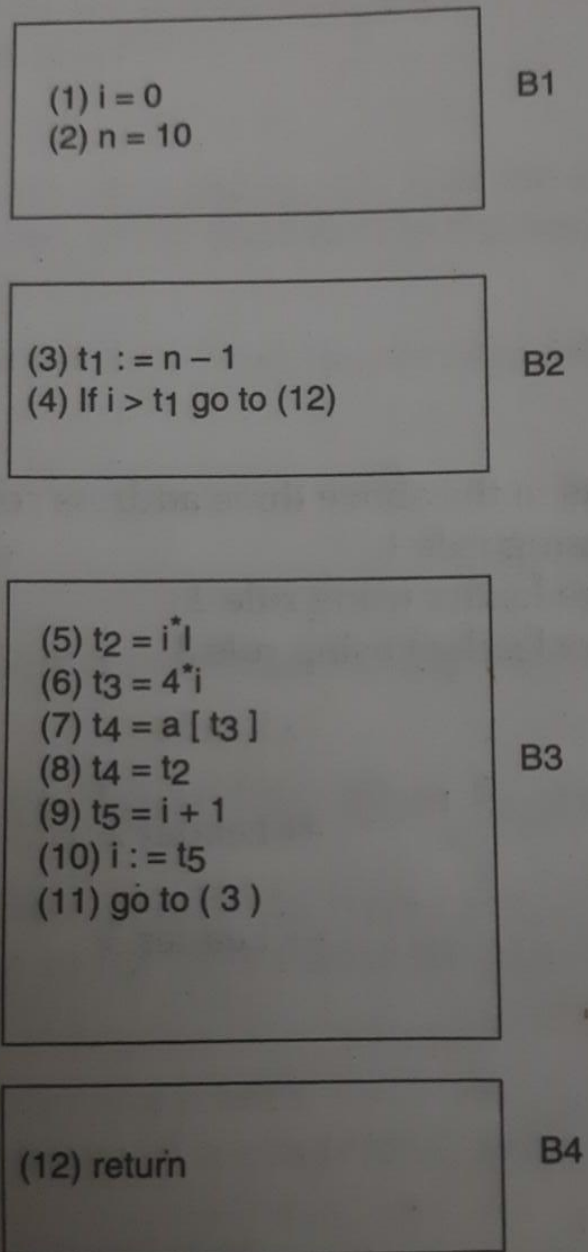


Figure 10.2 Basic Blocks

Flow graph:

An edge is placed from block B1 to B2 if block B2 could immediately follow B1 during execution or satisfies the following

1.

The last statement in B1 is either conditional or unconditional jump statement that is followed by first statement in B2.

Or

2.

The first statement in B2 follows the last statement in B1 and is not an unconditional /conditional jump.

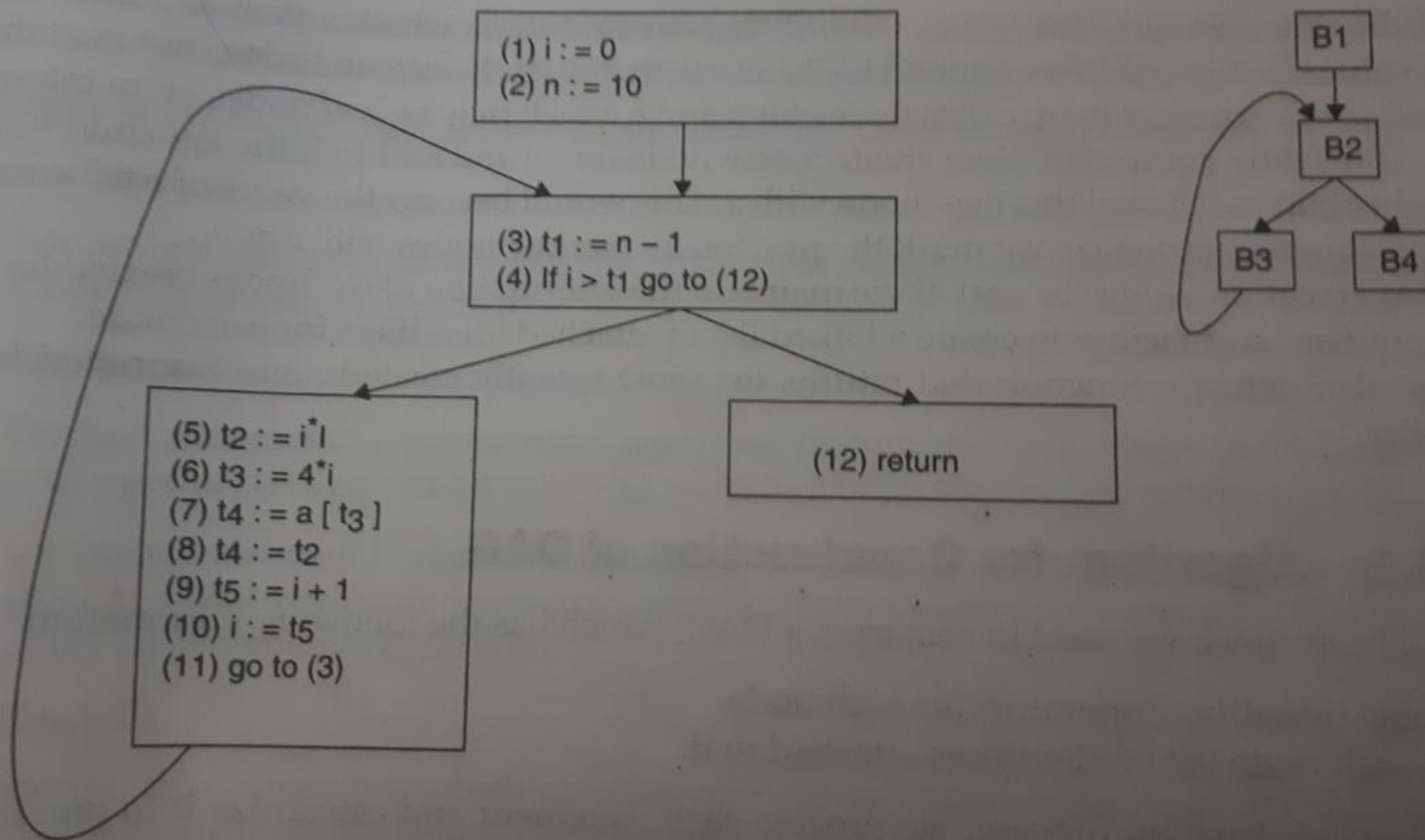


Figure 10.3 Flow Graph for Example 3

Code optimization techniques:

Code Optimization Techniques

❖ Constant propagation

□ If the value of a variable is a constant, then replace the variable by the constant

- It is not the constant definition, but a variable is assigned to a constant
- The variable may not always be a constant

□ E.g.

`N := 10; C := 2;`

`for (i:=0; i<N; i++) { s := s + i*C; }`

\Rightarrow `for (i:=0; i<10; i++) { s := s + i*2; }`

`If (C) go to ...` \Rightarrow `go to ...`

- The other branch, if any, can be eliminated by other optimizations

□ Requirement:

- After a constant assignment to the variable
- Until next assignment of the variable
- Perform data flow analysis to determine the propagation

Code Optimization Techniques

❖ Constant folding

- ❑ In a statement $x := y \text{ op } z$ or $x := \text{op } y$
- ❑ If y and z are constants
- ❑ Then the value can be computed at compilation time
- ❑ Example

`#define M 10`

`x := 2 * M \Rightarrow x := 20`

`If (M < 0) goto L \Rightarrow can be eliminated`

`y := 10 * 5 \Rightarrow y := 50`

- ❑ Difference: constant propagation and folding
 - Propagation: only substitute a variable by its assigned constant
 - Folding: Consider variables whose values can be computed at compilation time and controls whose decision can be determined at compilation time

Code Optimization Techniques

❖ Algebraic simplification

□ More general form of constant folding, e.g.,

- $x + 0 \Rightarrow x$ $x - 0 \Rightarrow x$
- $x * 1 \Rightarrow x$ $x / 1 \Rightarrow x$
- $x * 0 \Rightarrow 0$

□ Repeatedly apply the rules

- $(y * 1 + 0) / 1 \Rightarrow y$

❖ Strength reduction

□ Replace expensive operations

- E.g., $x := x * 8 \Rightarrow x := x \ll 3$

Code Optimization Techniques

❖ Copy propagation

- ❑ Extension of constant propagation

- ❑ After y is assigned to x, use y to replace x till x is assigned again

- ❑ Example

$x := y; \quad \Rightarrow \quad s := y * f(y)$

$s := x * f(x)$

- ❑ Reduce the copying

- ❑ If y is reassigned in between, then this action cannot be performed

Code Optimization Techniques

❖ Common subexpression elimination

□ Example:

$a := b + c$		$a := b + c$
$c := b + c$	\Rightarrow	$c := a$
$d := b + c$		$d := b + c$

Code Optimization Techniques

❖ Unreachable code elimination

- ❑ Construct the control flow graph
- ❑ Unreachable code block will not have an incoming edge
- ❑ After constant propagation/folding, unreachable branches can be eliminated

❖ Dead code elimination

- ❑ Ineffective statements
 - $x := y + 1$ (immediately redefined, eliminate!)
 - $y := 5 \Rightarrow y := 5$
 - $x := 2 * z \quad x := 2 * z$
- ❑ A variable is dead if it is never used after last definition
 - Eliminate assignments to dead variables
- ❑ Need to do data flow analysis to find dead variables

Code Optimization Techniques

❖ Loop optimization

- ❑ Consumes 90% of the execution time

⇒ a larger payoff to optimize the code within a loop

❖ Techniques

- ❑ Loop invariant detection and code motion
- ❑ Induction variable elimination
- ❑ Strength reduction in loops
- ❑ Loop unrolling

Code Optimization Techniques

❖ Loop invariant detection and code motion

- ❑ If the result of a statement or expression does not change within a loop, and it has no external side-effect
- ❑ Computation can be moved to outside of the loop
- ❑ Example

for (i=0; i<n; i++) a[i] := a[i] + x/y;

- Three address code

for (i=0; i<n; i++) { c := x/y; a[i] := a[i] + c; }

\Rightarrow c := x/y;

for (i=0; i<n; i++) a[i] := a[i] + c;

Code Optimization Techniques

❖ Strength reduction in loops

□ Example

$s := 0; \text{ for } (i=0; i<n; i++) \{ v := 4 * i; s := s + v; \}$
 $\Rightarrow s := 0; \text{ for } (i=0; i<n; i++) \{ v := v + 4; s := s + v; \}$

❖ Induction variable elimination

□ If there are multiple induction variables in a loop, can eliminate the ones which are used only in the test condition

□ Example

$s := 0; \text{ for } (i=0; i<n; i++) \{ s := 4 * i; \dots \}$ -- i is not referenced in loop
 $\Rightarrow s := 0; e := 4*n; \text{ while } (s < e) \{ s := s + 4; \}$

Code Optimization Techniques

❖ Loop unrolling

- ❑ Execute loop body multiple times at each iteration
- ❑ Get rid of the conditional branches, if possible
- ❑ Allow optimization to cross multiple iterations of the loop
 - Especially for parallel instruction execution
- ❑ Space time tradeoff
 - Increase in code size, reduce some instructions

2. Loop Unrolling:

Loop unrolling is a loop transformation technique that helps to optimize the execution time of a program.

We basically remove or reduce iterations.

Loop unrolling increases the program's speed by eliminating loop control instruction and loop test instructions.

Example:

Initial code:

```
for (int i=0; i<5; i++)  
    printf("CSE");
```

Optimized code:

```
printf("CSE\n");  
printf("CSE\n");  
printf("CSE\n");  
printf("CSE\n");  
printf("CSE\n");
```

Loop Jamming:

Loop jamming is the combining the two or more loops in a single loop.

It reduces the time taken to compile the many number of loops.

Example:

Initial Code:

```
for(int i=0; i<5; i++)  
    a = i + 5;  
for(int i=0; i<5; i++)  
    b = i + 10;
```

Optimized code:

```
for(int i=0; i<5; i++)  
{  
    a = i + 5;  
    b = i + 10;  
}
```

Code Optimization Techniques

❖ Loop fusion

□ Example

```
for i=1 to N do
    A[i] = B[i] + 1
endfor
for i=1 to N do
    C[i] = A[i] / 2
endfor
for i=1 to N do
    D[i] = 1 / C[i+1]
endfor
```

```
for i=1 to N do
    A[i] = B[i] + 1
    C[i] = A[i] / 2
    D[i] = 1 / C[i+1]
endfor
```

Is this correct?
Actually, cannot fuse
the third loop

Before Loop Fusion

Code Optimization -- Summary

❖ Read Section 9.1

Example:

a=b+c	a=b+c	a=b+c	a=b+c
z=a**2	z=a*a	z=a*a	z=a*a
X=0*2	x=0	x=0	x=0
y=b+c	y=b+c	y=a	
w=y*y	w=y*y	w=a*a	w=z
u=x+3	u=3	u=3	u=3
v=u+w	v=3+w	v=3+w	v=3+z

GIVEN CODE

a=b+c
z=a*a
v=3+z

OPTIMIZED CODE

Machine dependent Optimization

This optimization can be applied on target machine instructions.

This includes

Register allocation

Use of addressing modes

Peephole optimization

Peephole optimization

→ Redundant Loads and stores

Mov y,R0

ADD z,R0

MOV R0,x

a=b+c

d=a+e

Then it generates the code given below

1.MOV b,R0

2.ADD c,R0

3.MOV R0,a

4.MOV a,R0

5.ADD e,R0

6.MOV R0,d

1.MOV b,R0

2.ADD c,R0

3.ADD e,R0

4.MOV R0,d

Peephole optimization:

→ Algebraic simplification

$x = x + 0$

$x = x * 1$

→ Dead code elimination

#define x 0

→ If(x)

→ {

→ ...

→ Printvalue

→ ...}

If x=1 goto L1

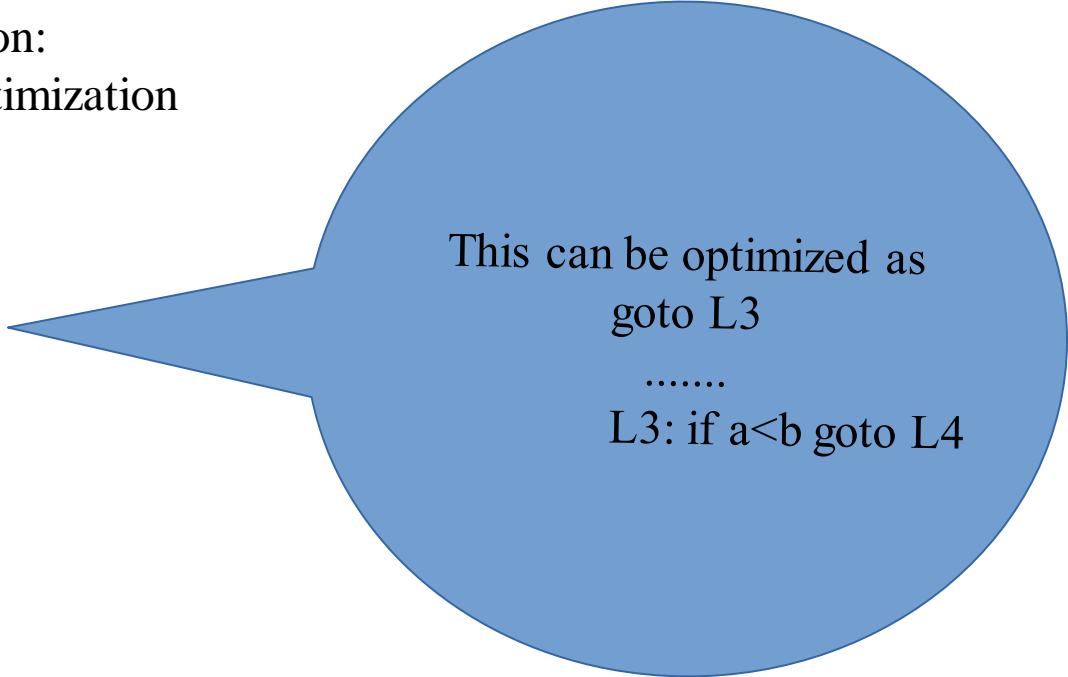
goto L2

L1:print value

L2:....

Peelhole optimization:
Flow of Control optimization

Goto L1
L1: goto L2
.....
L2: goto L3
....
L3: if a<b goto L4
L4:



This can be optimized as
goto L3
.....
L3: if a<b goto L4

Peelhole optimization:

Reduction in strength

This optimization mainly deals with replacing expensive operations by cheaper ones.

For example

X^2 ----- $x * x$

Use of Machine Idioms

$x = x + 1$

INR x

Data flow analysis of flow graphs

There are two types of analysis performed for global optimizations.

Control flow analysis and data flow analysis.

Data flow analysis is a process of computing values of data flow properties.

Data flow properties:

Available expression

Reaching definition

Live variables

Busy variables

w1:x=3-----definition point

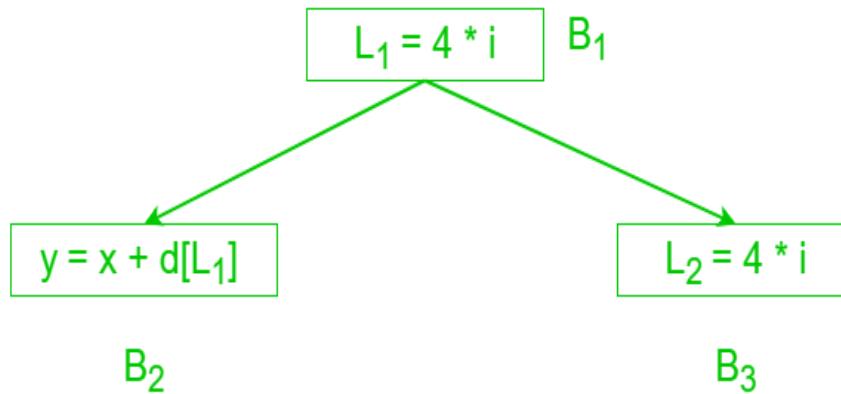
W2:y=x-----Reference point

W3:z=a*b-----Evaluation point

Data Flow Properties –

Available Expression – An expression is said to be available at a program point x iff along all paths reaching to x , the expression is available at its evaluation point.

An expression $a+b$ is said to be available if none of the operands gets modified before their use.



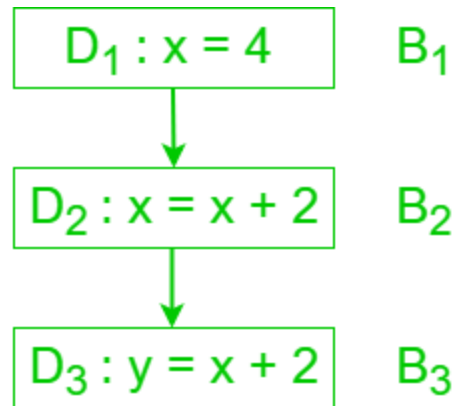
Expression $4 * i$ is available for block B_2, B_3

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

Example –

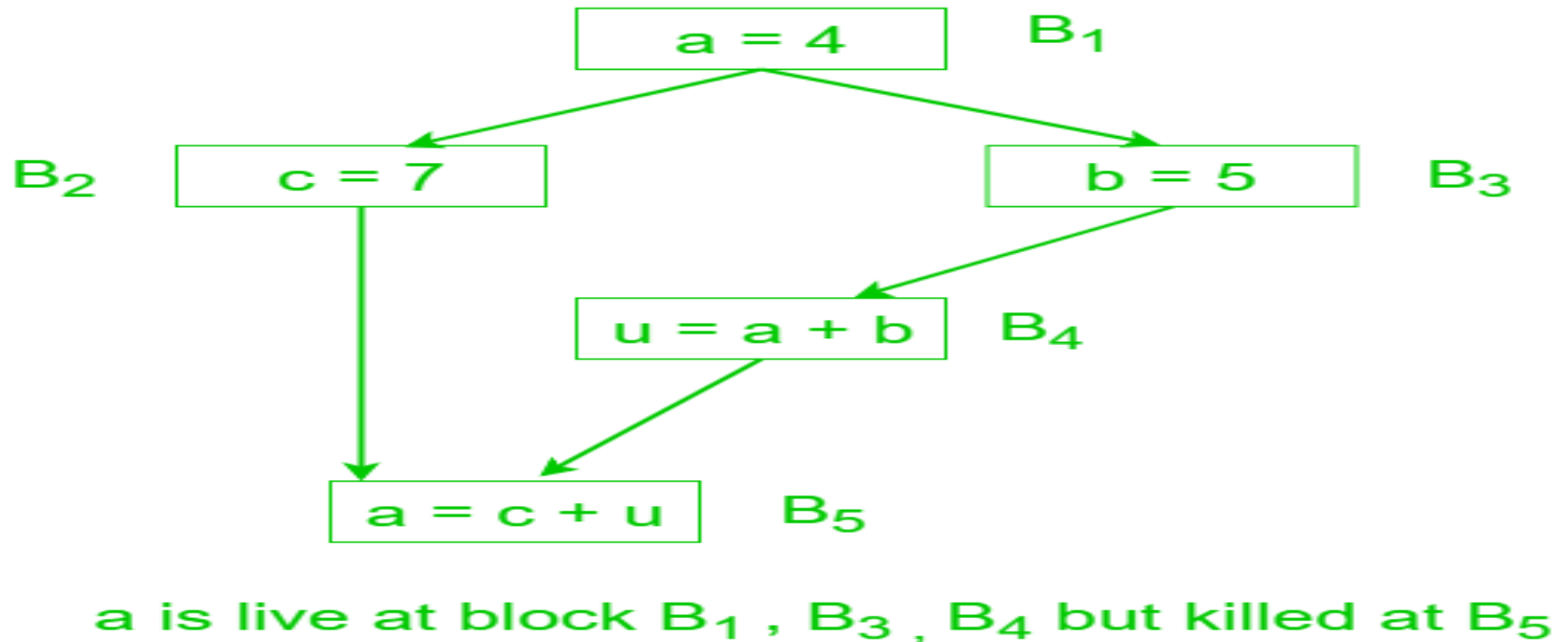
Advantage –

It is used in constant and variable propagation.



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.



Advantage –

It is useful for register allocation.

It is used in dead code elimination.

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.

Advantage –

It is used for performing code movement optimization.

Issues in the design of a code generator

Input to code generator –

Target program –

Memory Management –

Instruction selection –

$P := Q + R$

$S := P + T$

MOV Q, R0

ADD R, R0

MOV R0, P

MOV P, R0

ADD T, R0

MOV R0, S

A prior knowledge of instruction cost is needed in order to design good sequences, but accurate cost information is difficult to predict.

Register allocation –

Evaluation order –

Code generation using DAG

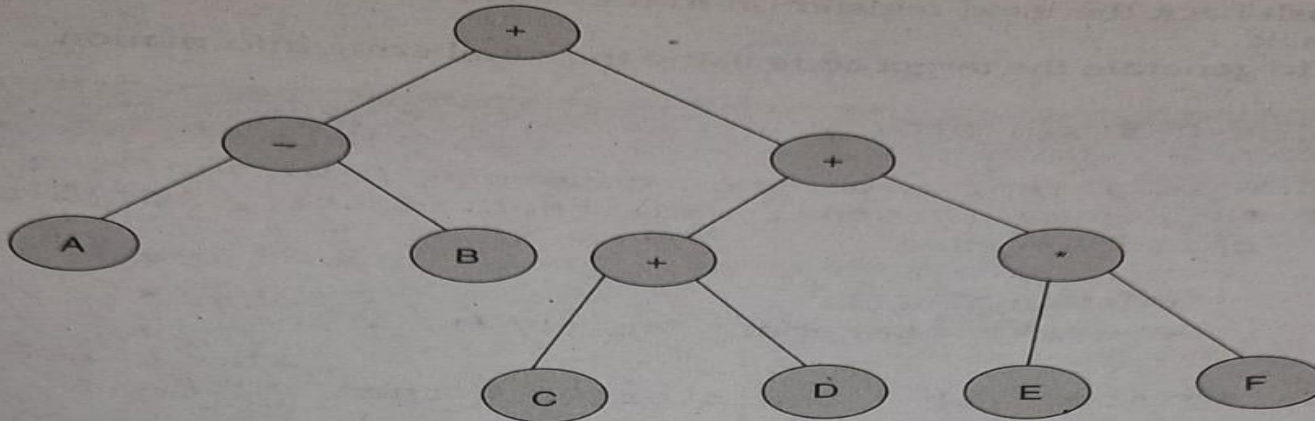


Figure 11.3 AST for $(A-B) + ((C+D) + (E * F))$

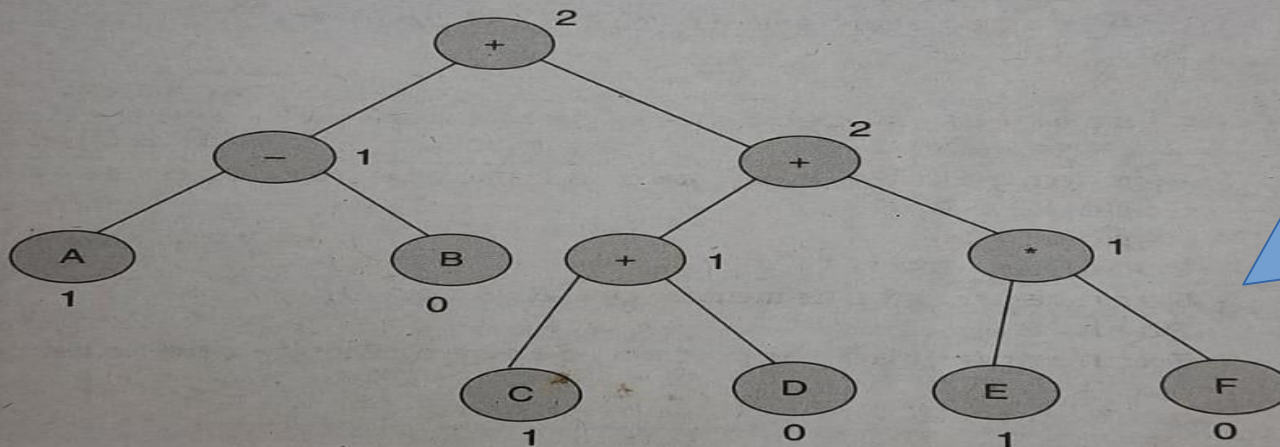


Figure 11.4 Register Allocation

Load R2,c
Add R2,D
Load R1,E
Mult R1,F
Add R2,R1
Load R1,A
Sub R1,B
Add R1,R2