Compiler Design
Unit-5

Code optimization :-

Elimination of unnecessary instruction in object code or the replacement of one sequence of instructions by a faster sequence of instructions that does the same thing is usually called "code improvement" or "code optimization". (Space & time).

Function Preserving Transformations :-

There are no. of ways in which a compiler can improve a program without changing the function it computes.

Function preserving Transformation examples :-

(i) Common ~~Subsequence~~ expression Problem :-

An occurrence of expression E is called common subexpression if E was previously computed and the values of the variables in E have not changed since the previous computation.

	old	new
a	2	5
b	3	1
c	2	3
d	4	1

value not changed

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

so, we can replace

(ii) Constant folding :-

If the value of an expression is constant, use the constant instead of expression.

Eg → PI = 22/7 or PI = 3.14.

(iii) Copy Propagation :-

Eg →

```
x = a              x = a
y = x * b   ⇒      y = a * b
z = a * c          z = a * c
```

↳ Copy ka time bacha. But it introduced dead code. as x = a is useless now.

(iv) Dead Code Elimination :-

Eg →

```
x = a
y = a * b   ⇒   y = a * b
z = a * c       z = a + c
```

(v) Code Motion :- Loops are very important place for optimizations, especially the inner loops where program tend to spend bulk of their time. The running time of a program may be improved if we decrease the no. of instruction in an inner loop. So, an important modification that decreases the amount of code in a loop is Code Motion.

Ex→
```
while (i<10) {
    x = y+z
    i = i+1
}
```
⇒
```
x = y+z
while (i<10) {
    i = i+1
}
```

(vi) Induction variables elimination & Reduction in Strength :-

Another important optimization is to find induction variables in loops and optimize their computation.

A variable x is said to be an 'induction variable' if there is a positive or negative constant C such that each time x is assigned, its value increases by C. It can be computed with a single increment or decrement per loop iteration.

Eg →
```
i = 1
while (i<10) {
    t = i*4
    i = i+1
    print(t)
}
```
⇒
```
t = 4
while (t<=40) {
    t = t+4;
    print(t)
}
```

Code optimization in Basic Blocks :-

- Structure Preserving transformation
  - Dead Code Elimination
  - Common Subexpression Elimination
  - Renaming of temporary variables
  - Interchange of two independent adjacent statement.
- Algebric transformation
  - Constant folding
  - Copy propagation
  - Strength Reduction

→ Dead Code Elimination :-

```
x = 2
if (x > 2)
print ("Code is Correct");
else
print ("Code is incorrect");
```

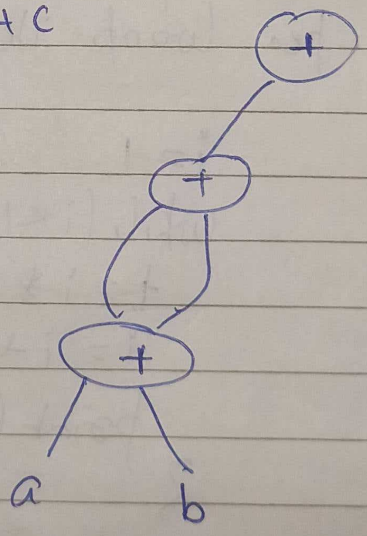
⇒

```
x = 2
print ("Code is incorrect");
```

→ Common ~~Subexpression~~ Subexpression Elimination :-

In this technique, the subexpression which are common and used frequently, are calculated only once and reused when needed. DAG is used to eliminate common subexpressions.

$x = (a+b) + (a+b) + c$



→ Renaming of temporary variables :-

$t = a+b$

$x = a+b$

→ Interchange of two independent adjacent Statement :-
If a block has two adjacent statements which are independent can be interchanged without affecting the basic block value.

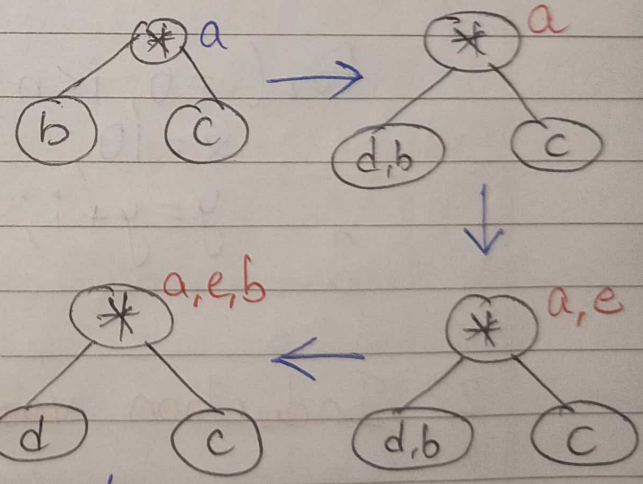
$t1 = a + b$
$t2 = c + b$

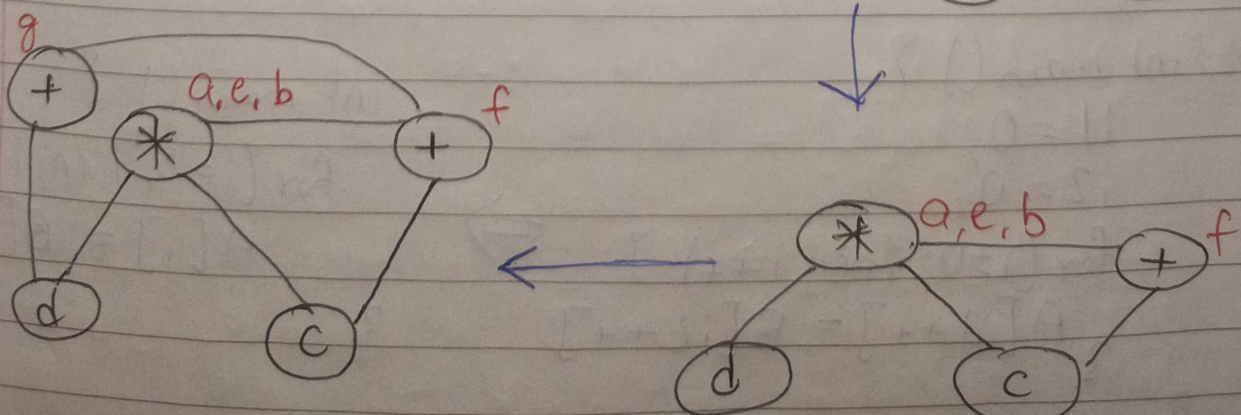
→ Algebraic Transformation :-
- → Constant folding :- $x = 2*3+y \Rightarrow x = 6+y$
- → Copy propagation :- $x = 3$, $y = x+c \Rightarrow x = 3$, $y = 3+c$
- → Strength Reduction :- $x = y*2$, $x = y+y$.

Building Expressions of DAG :-

$a = b*c$
$d = b$
$e = d*c$
$b = e$
$f = b+c$
$g = d+f$

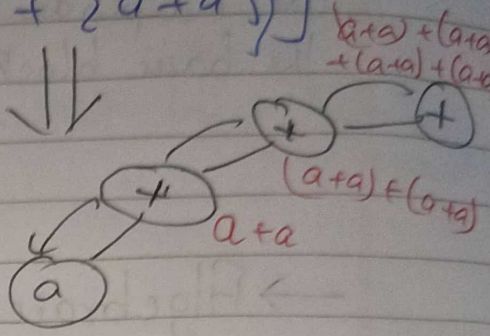
Draw optimized DAG :-



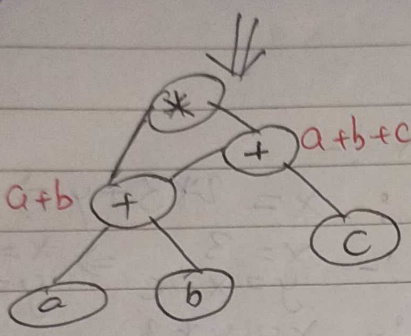


Question to practise :-

1) $[\{(a+a)+(a+a)\}+\{\{a+a\}+\{a+a\}\}]$



2) $(a+b)*(a+b+c)$.



Loop Optimization :-

1) Code Motion and Frequency reduction :-

```
for(i=0; i<n; i++){
    x=10;
    y=y+i;
}
```

⇒

```
x=10;
for(i=0; i<n; i++){
    y=y+i;
}
```

2) Induction ~~various~~ Variable Elimination :-

```
int main(){
    i1=0;
    i2=0;
    for(i=0; i<n; i++){
        A[i1++]=B[i2++]
    }
}
```

⇒

```
int main(){
    for(i=0; i<n; i++)
        A[i]=B[i]
}
```

3. Loop Merging :-

```
for(i=0; i<n; i++)
    A[i] = i+1;
for(j=0; j<n; j++)
    B[i] = j-1;
```

⇒

```
for(i=0; i<n; i++)
{
    A[i] = i+1;
    B[i] = i-1;
```

4. Loop Unrolling :-

```
main() {
    for(i=0; i<3; i++)
        Cout<<'cd';
}
```

⇒

```
main() {
    Cout<<'cd';
    Cout<<'cd';
    Cout<<'cd';
```

Peephole Optimization :-

1) Redundant load and store elimination
2) Constant folding
3) Strength Reduction
4) Null Sequence / Simplify Algebric Expression
5) Combine operations
6) Dead Code elimination

1) In this redundancy is eliminated.

```
y = x+5
i = y
z = i
w = z*3
```

⇒

```
y = x+5;
w = y*3;
```

2) PI = 22/7 ⇒ PI = 3.14

3) y = x*2 ⇒ y = x+x

4) a = a+0 a = a/1
a = a*1 a = a-0

avoid using these. delete them if there

5) a = b+c ⇒ a = b+c+e
a = a+e

6) Dead code eliminat

Global Data flow Analysis

Steps :-

1) Assign a distinct number to each definition as d_1, d_2, d_3, etc
2) for each variable k, make a list of all definition in the entire diagram where it is used.
3) Calculate following :-
   a) GEN[B] = the set GEN[B] consist of all definition that are generated in blocks.
   b) KILL[B] → the set of all definitions that are outside of block & having same definition as block B.

Q4) Compute the following :-

a. IN[B] :- the set of all definitions reaching the point just before the first stmt of block B.

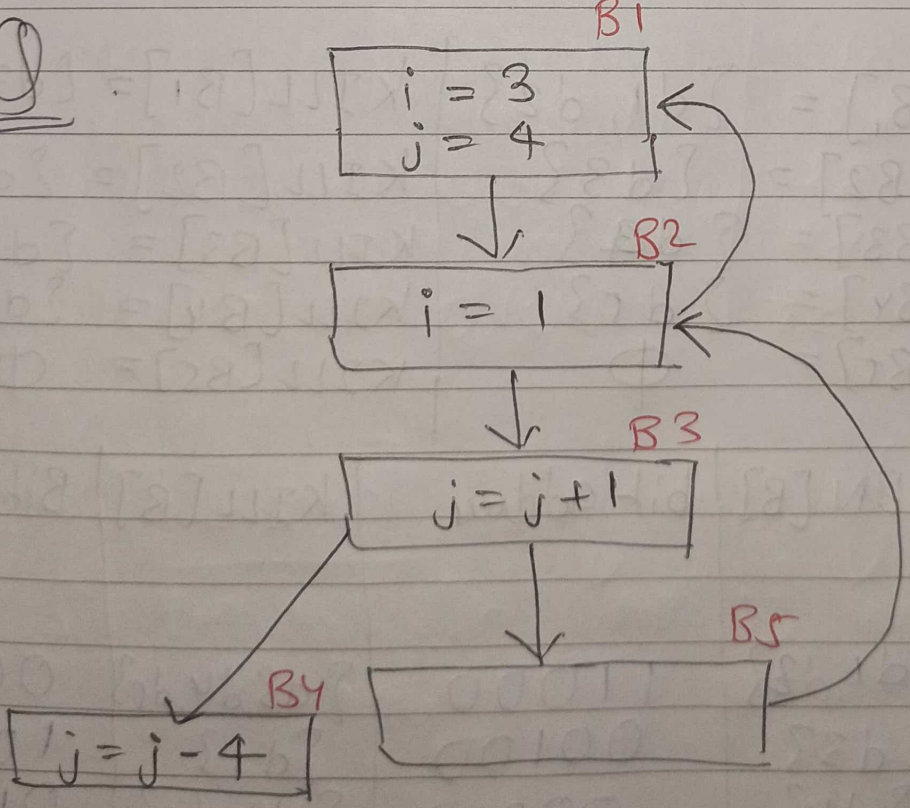
b. OUT[B] = ~~ending~~ the set of all definitions reaching the point just after the ending stmt of block B.

$$IN[B] = \bigcup OUT[B]$$

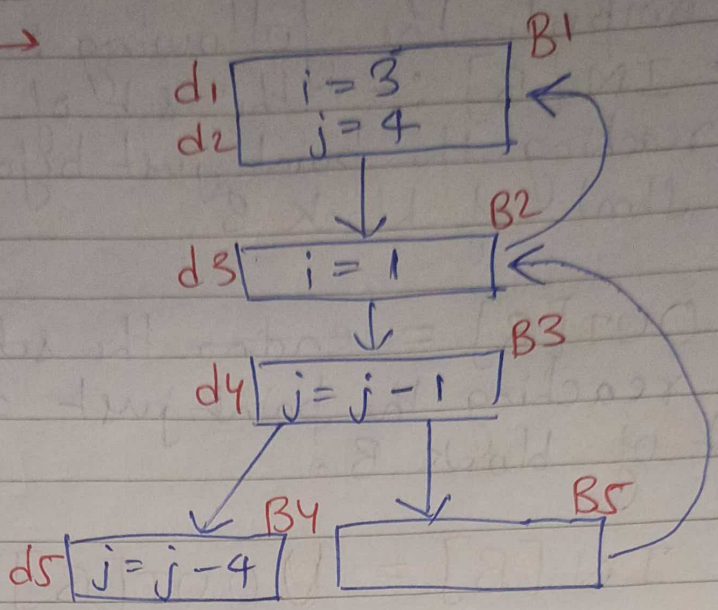
$$OUT[B] = IN[B] - KILL[B] \cup GEN[B]$$

$$= \{IN[B] \text{ AND } (\sim KILL[B])\} \text{ OR } GEN[B]$$

Q.



Sol → Step 1 →



Step 2 :- $i = \{d1, d3\}$
$j = \{d2, d4, d5\}$

Step 3 :-

GEN[B1] = {d1, d2}	KILL[B1] = {d3, d4, d5}
GEN[B2] = {d3}	KILL[B2] = {d1}
GEN[B3] = {d4}	KILL[B3] = {d2, d5}
GEN[B4] = {d5}	KILL[B4] = {d2, d4}
GEN[B5] = ϕ	KILL[B5] = ϕ

Block	GEN[B]	bit address	KILL[B]	Bit Analysis
B1	{d1, d2}	11000	{d3, d4, d5}	00111
B2	{d3}	00100	{d1}	10000
B3	{d4}	00010	{d2, d5}	01001
B4	{d5}	00001	{d2, d4}	01010
B5	ϕ	00000	ϕ	00000

Step 4 :-

Pass 0 :-

Block	IN[B]	OUT[B]
B1	00000	11000
B2	00000	00100
B3	00000	00010
B4	00000	00001
B5	00000	00000

Pass 1 :-

→ IN[B1] = OUT[B2]
= 00100

→ IN[B2] = OUT[B1] ∪ OUT[B5]
= 11000 ∪ 00000 ⇒ 11000

→ IN[B3] = OUT[B2]
= 00100

→ IN[B4] = OUT[B3]
= 00010

→ IN[B5] = OUT[B3]
= 00010

→ OUT[B1] = [IN[B1] AND (~KILL[B1])] OR GEN[B1]
= [00100 AND (~00111)] OR 11000
= [00100 AND 11000] OR 11000
= [00000 OR 11000]
= 11000

OUT[B2] = [IN[B2] AND (~KILL[B2])] OR GEN[B2]
= (11000 AND 01111) or 00100
= 01000 or 00100
= 01100

OUT[B3] = [IN[B3] and (~KILL[B2])] OR GEN[B3]
= (00100 AND 10110) OR 00010
= 00100 OR 00010
= 00110

OUT[B4] = [IN[B4] and (~KILL[B4])] OR GEN[B4]
= (00010 AND 10101) or 00001
= 00000 or 00001 = 00001

OUT[B5] = [IN[B5] and (~KILL[B5])] OR GEN[B5]
= (00010 AND 11111) OR 00000
= 00010

Block	IN[B]	OUT[B]
B1	00100	11000
B2	11000	01100
B3	00100	00110
B4	00010	00001
B5	00010	00010

Similarly, we find Pass 2, Pass 3 — and so on, till we get 2 pass same.

	Pass 2		Pass 3		Pass 4	
Block	IN[B]	OUT[B]	IN[B]	OUT[B]	IN[B]	OUT[B]
B1	01100	11000	01110	11000	01110	11000
B2	11010	01110	01110	01110	01110	01110
B3	01100	00110	01110	00110	01110	00110
B4	00110	00101	00110	00101	00110	00101
B5	00110	00110	00110	00110	00110	00110

↓

These are value I got. You can verify by doing it on your own.