Date:
16-09-2014

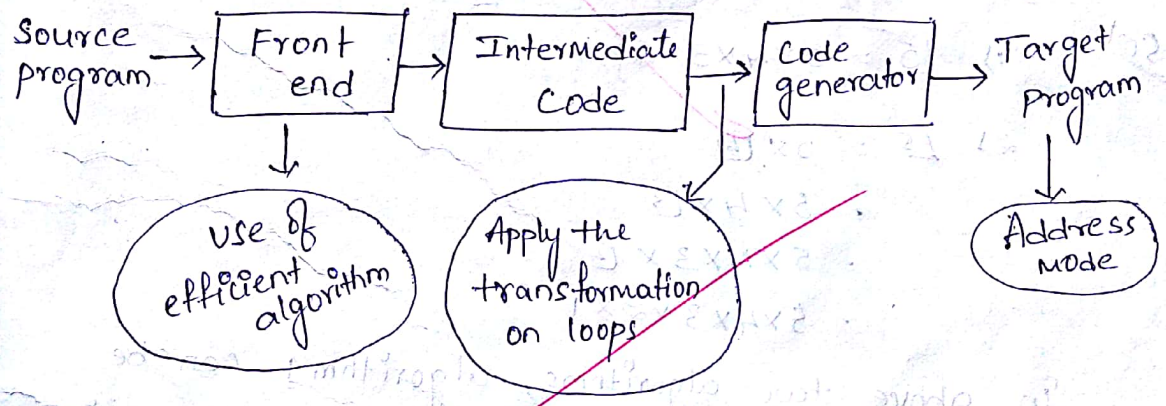
UNIT-6

CODE OPTIMIZATION

- → Consideration for optimization
- → Scope of optimization.
- → Local optimization
- → Loop optimization
- → Frequency Reduction
- → Folding
- → DAG representation.

Consideration for optimization (or) code optimization :-

Code optimization means reducing (or) optimizing intermediate code. Code optimization is used to generate efficient target program. We can apply the code optimization in different phases in compilation process.



Any compiler (or) programmer can convert the source program into target program. And when compiler converts the source program into target program we are using front end, Intermediate code, code generator.

→ In front end source program act as machine dependent & traget program act as independent.

→ In be

After compilation of front end we impliment the intermediate code generator. Intermediate code generator can hold the what compiler can send. After complition of intermediate code generation apply the transformation on loops for the code and proceed to generate code generator after completion all these phases we reach the target program. In target program we are using registers, instructions, and addressing modes. And target program act as a back end.

→ In back end source program act as a machine independent and Target program act as a machine dependent.

Eg: To find the factorial of 5 we have used two algorithms

Sol

1) $\lfloor 5 = 5 \times 4 \times 3 \times 2 \times 1$

2) $\lfloor 5 = 5 \times \lfloor 4$

$= 5 \times 4 \times \lfloor 3$

$= 5 \times 4 \times 3 \times \lfloor 2$

$= 5 \times 4 \times 3 \times 2 \times 1$

In above two algorithms, algorithm 1 can be optimized to one statement. And algorithm 1 required less amount of space and less time. Therefore algorithm 1 is efficient. and we can use algorithm 1 to develop the source program. After generating the intermediate code the code optimization can be done by applying transformations on loops.

→ After generating the target program the code optimization can be done by using appropriate instruction, registers and addressing modes.

Eg:- $P = i + r * 100$

Intermediate code

$t_1 = 100$

$t_2 = r * t_1$

$t_3 = i + t_2$

$P = t_3$

$\Rightarrow$

code optimization

$t_1 = r * 100$

$t_2 = i + t_1$

$P = t_2$

Target program.

```
mul  r, 100
mov  t1, r
ADD  i, t1
mov  t2, i
mov  i, t2
```

14M

Scope of optimization :- (or) [classification of code optimization based on scope]

Based on the scope optimization we implement three techniques

1) Local optimization
2) Loop optimization
3) global optimization.

Local optimization :-

If the optimization can be applied to specific block then this called Local optimization. This can be performed by using following techniques.

→ Elemenation of common sub expression
→ copy propagation
→ constant folding
→ Dead-code elemination.

→ Elemenation of common sub expression :-

It means elemenating the common subexpression from the block (or) code. It avoiding recomputation of sub expression which is already computed in previously.

Eg:-
$t_1 = a*b$
$t_2 = a+b$
$t_3 = a*b$
$t_4 = a-b$
$x = t_4$

In above block we have common sub expression i.e., $t_3 = a*b$ so we can eleminate the common sub expression as

$t_1 = a*b$
$t_2 = a+b$
$t_4 = a-b$
$x = t_4$

∴ The given block is optimized as recently above obtained block.

→ Copy propagation :-

copy propagation can be done in the following two technique.

⇒ constant propagation
⇒ variable propagation.

Constant propagation :-

constant propagation is a copy propagation of a local optimization and in which variable is replaced by constant.

Eg:- $pi = 3.14$, $r=5$

$$a = Pi * r * r$$
$$= 3.14 * 5 * 5$$
$$a = 78.5$$

The above evaluation can be done at the time of compilation. This compilation is called as compile time evaluation.

variable propagation:-

variable propagation is a copy propagation of local optimization and in which the variable is replaced by another variable.

Eg:- $x = Pi$

$a = x * r * r$

$a = pi * r * r$.

→ constant folding (or) (folding):-

Constant folding is technique in which eleminating the constant expressions ~~are~~ (or) contants. To declare the constants we have to use preprocessors that is "# define" it is used to declare the constant in the program.

Eg:- # define Max, 10

$y = MAX * 10$

$x = MAX + 3$

In above code we are using pre processor declaration and it require more space to avoiding this problem we can eleminate pre processor statements i.e., $y = 100$, $x = 13$

→ Dead Code elemination:-

Here we are using two variables that is

⇒ Live variable.

⇒ dead variable.

Live variable:-

A variable is said to be live variable if it can be used in furture after its declaration

Dead variable:-

A variable is said to be dead variable if it cannot be used in furture after its declaration

→ Dead code elemination is a technique in which eleminating the dead code means that which having the dead variable

Eg:-
```
int i=0, j=0
for (i=1; i<=10; i++)
  {
  printf ("Hai");
  }
```

In above code 'i' variable is called as live variable, 'j' variable is called as dead variable. Since j cannot be used in furture after its declaration.

Now we want to delete the dead code i.e., j=0 from above code. as

```
int i=0;
for (i=1; i<=10; i++)
  {
  printf ("Hai");
  }
```

14M Loop optimization:-

If we are applying the optimization to the loops then this optimization is called loop optimization.

If the loop having more no. of statements. then this loop will require more space and more time and to execute to avoiding this two problems we have to apply optimization to the loops. Loop optimization can be done by using the following techniques.

1) Code motion
2) Induction variable
3) Reduction of strength
4) Loop invarient
5) Loop Unrolling
6) Loop fusion

Code motion:-

It is a technique in which sending the expression which is used at the entry of the loop and can be executed every time

Eg:-
```
#include<stdio.h>
main()
{
max=10
for(i=0;i<=max-1;i++)
{
printf("hai");
}
getch();
}
```

In the above for loop we have the expression max-1, this expression executed by the compiler every time, but this expression value should be constant.

i.e., 9. To over come the reapeated execution of max-1. We have to send max-1 to out side of the loop

```
#include<stdio.h>
main()
{
    max=10;
    t=max-1;
for(i=0; i<=t; i++)
{
printf("hai");
}
getch();
}
```

In above loop optimization can be done i.e., max-1 expression can be executed only once.

Induction variable:-

A variable in a loop is said to be an induction variable, if this variable value can be changed by using some constant. Here changing of the variable is may be increament (or) decrement.

Eg:-

```
#include<stdio.h>
main()
{
for(i=0; i<=10; i++)
{
i=i+10;   // induction variable
Printf(";");
}
getch();
}
```

In the above loop 'i' variable can be changed every time by the constant '10'. Therefore 'i' variable is called as induction variable.

Reduction of Strength :-

It is a technique in which replacing the higher strength operator by lower strength operator.

Higher strength operator means the operator which having the highest priority.

Lower strength operator means the operator which having the lower priority.

Eg:- +, *

In above example '*' is higher strength operator '+' is lower strength operator.

```
#includ <stdio.h>
    main()
    {
    for (i=1; i<=10; i++)
        {
        count = i * 7;
    printf ("count");
    }
    getch();
    }
```

In the above loop we have higher strength operator i.e., '*' can be replaced by lower strength operator '+' as shown in below.

```
# include <stdio.h>
    main()
{ for (i=1; i<=10; i++)
    {
    count = i + 7;
    printf ("count");
    }
getch();
}
```

Loop invarient :-

It is a technique in which sending the expression to out side of the loop which is used in in side the loop and executed by the compiler every time but this expression value should not be change

Eg:-

```
#include <stdio.h>
main()
 {
max = 10;
for (i=1; i≤10; i++)
  {
  t = max - 1;
  printf ("t");
  }
getch();
 }
```

In the above loop the expression max-1 can be used inside the loop, this expression executed every time by the compiler but this expression value should not be changed. So we can send this expression to outside of the loop to execute only one time as

```
#include <stdio.h>
 {
main()
  {
  max = 10;
  t = max - 1;
for (i=1; i≤ t; i++)
  {
  printf ("t");
  }
getch();
 }
```

Loop unrolling :- It is a technique in which reducing the iterations or jumps of the loop by writting loop statement twice.

```
for (i=1; i<=100; i++)
  {
  printf ("Hai");
  }
```

In the above loop the loop having NO iterations This hundred iterations are reduced by the fifthy (50) by writting the loop statement printf twice as

```
for (i=1; i<=50; i++)
  {
  printf ("Hai");
  printf ("Hai");
  }
```

Loop fusion :- It is a technique in which grouping several loops into singal loop

```
for (i=1; i<=5; i++)
{
for (j=1; j<=5; j++)
  {
  printf ("Hai");
  }
}
```

In the above code we have two for loops i.e., several loops. Now we want combine this 2 loops into single loop.

```
for (i=1; i<=25; i++)
  {
  printf ("Hai");
  }
```

Optimization of basic block:-

Basic block:- Basic block is a sequence of statements in which flow control can enter at one entry and leaves at another entry is called as basic block.

→ To partician the given program into basic blocks we have to use following algorithm.

1. First we have to obtain 3-address code for given statement.
2. The 1st statement in 3-address code is leader statement.
3. The target statement of conditional (or) unconditional loop is also a leader statement.
4. The basic block is identified (or) start with leader statement and end with just before of next leader statement.

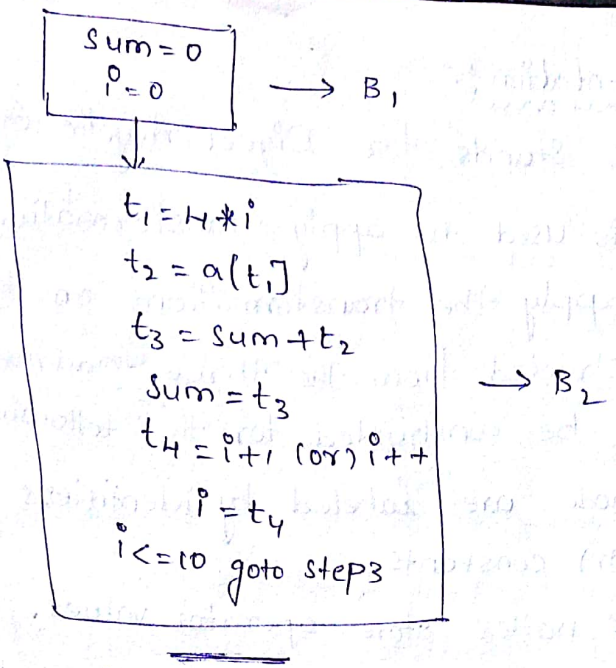
Eg:-1 Consider

```
sum=0
for(i=0; i<=10; i++)
    sum=sum+a[i]
```

Sol The 3-address code for the above code is

1) Sum = 0
2) $i = 0$
3) $t_1 = 4 * i$
4) $t_2 = a[t_1]$
5) $t_3 = sum + t_2$
6) $sum = t_3$
7) $t_4 = i+1$ (or) $i++$
8) $i = t_4$
9) $i <= 10$ goto step 3

Basic block:



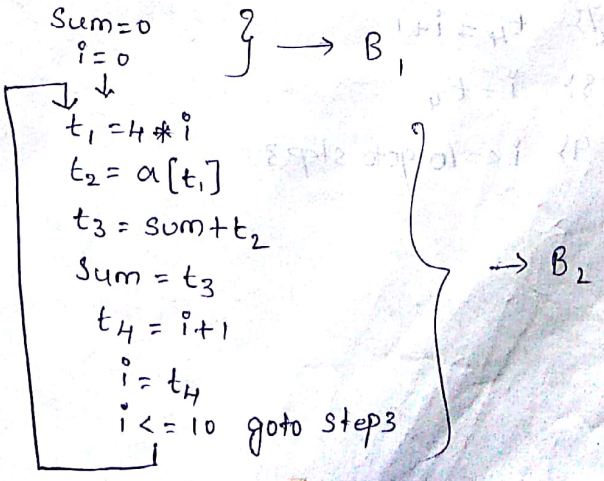
Eg:-2 Construct the basic block of

```
sum = 0
i = 0
do
{
Sum = sum + a[i];
i = i+1;
} while (i <= 10);
```

Sol The 3-address code for above code is

1) Sum = 0
2) i = 0
3) $t_1 = 4 * i$
4) $t_2 = a[t_1]$
5) $t_3 = sum + t_2$
6) $sum = t_3$
7) $t_4 = i + 1$
8) $i = t_4$
9) $i < 10$ goto step 3

Basic block:-



DAG Representation :-

DAG Stands for Direct Acyclic Graph.

The DAG is used to apply transformations on the basic block. To apply the transformations on basic block a DAG is constructed from the three-address statement. A DAG can be constructed for the following.

1. Leaf node are labeled by identifiers (or) variable names (or) constants.
2. Interior nodes store operator values.

The DAG & flow graph are two different pictorial representation each node of the flow graph can be represented by DAG. Because each node of the flow graph is a basic block.

Eg:- Consider

```
Sum = 0
for (i=0; i<=10; i++)
    Sum = sum + a[i]
```

Sol 3-address code is

1) Sum = 0
2) i = 0
3) $t_1 = 4 * i$
4) $t_2 = a[t_1]$
5) $t_3 = Sum + t_2$
6) sum = t_3
7) $t_4 = i + 1$
8) $i = t_4$
9) i <= 10 goto step 3

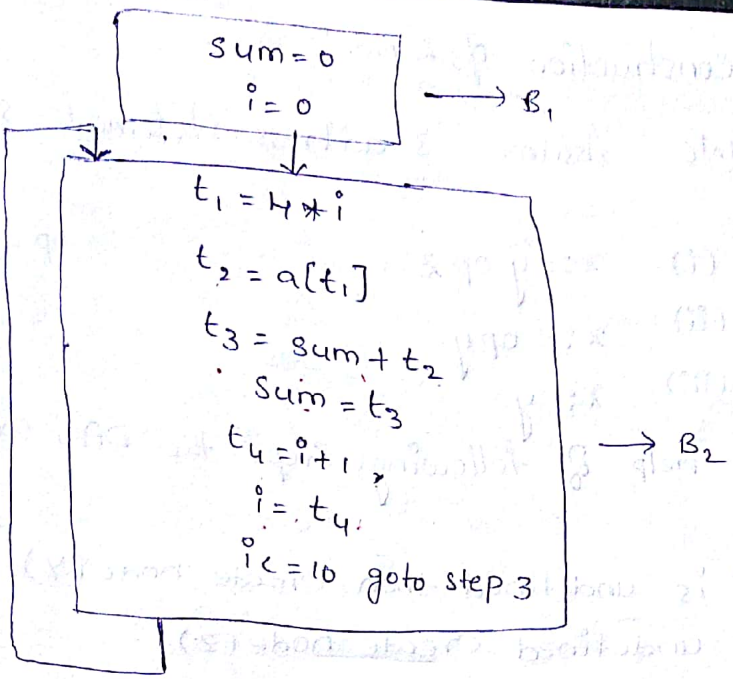
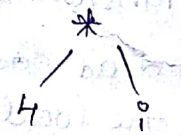


fig: Basic block.

Now Let us consider block B_2 for construction of DAG by numbering it.

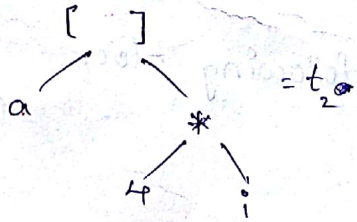
1) $t_1 = 4 * i$
2) $t_2 = a[t_1]$
3) $t_3 = sum + t_2$
4) $sum = t_3$
5) $t_4 = i + 1$
6) $i = t_4$
7) $i <= 10$ goto Step 3

1)



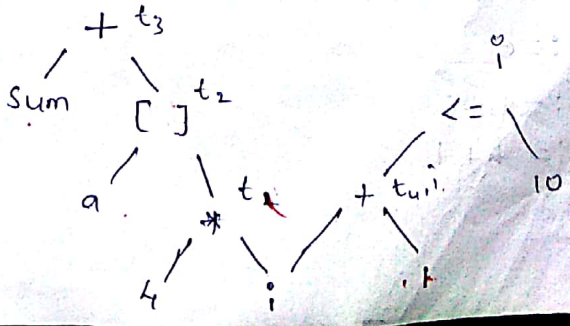
$= t_1$

2)



$= t_2$

Similarly



Algorithm for Construction of DAG :-

We Assume 3-address statement should follow

op - operator

Case (i) $x := y \ op \ z$

Case (ii) $x := op \ y$

Case (iii) $x := y$

With the help of following steps the DAG can be Constructed.

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

Step 2: For the case(i) create a node (op) whose left child is node (y) and node (z). will be the right child. and check for any common sub expression. for the case (ii) determine whether is a node labeled op, such node will have child node(y). In case (iii) node (n) will be node(y)

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

DAG Based local optimization :-

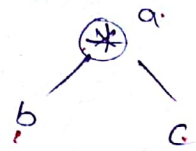
As mentioned earlier DAG can be constructed for a block and a certain transformation such as common sub expression elimination dead code elimination can be applied for performing the local optimization.

Eg:- Construct the DAG for the following block.

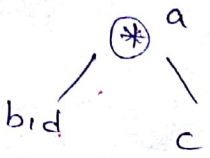
```
a := b*c
d := b
e := d*c
b := e
f := b+c
g := f+d
```

Sol

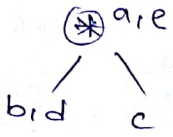
Step 1:



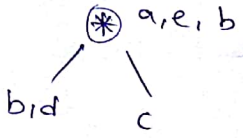
Step 2:



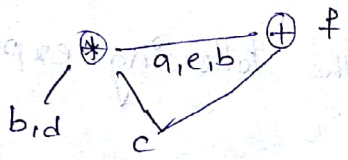
Step 3:



Step 4:



Step 5:



Step 6:

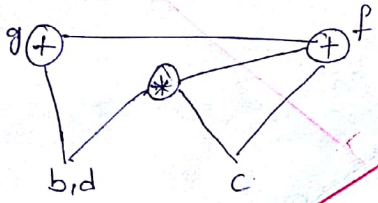


fig: construction of DAG

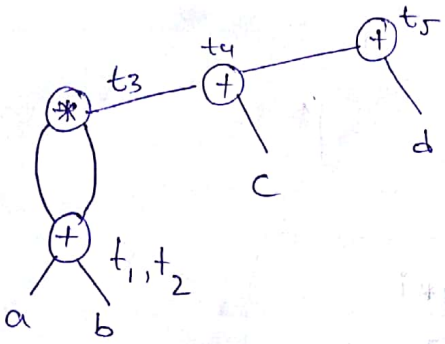
The optimized code can be generated by traversing the DAG. The local optimization on the above block can be done.

1) A common sub expression e:=d*c which is actually b*c since d:=b is eliminated.

2) Dead code for b:=e is eliminated. Therefore the optimized code of basic block is

a:= b*c
d:= b
f:= b+c
g:= f+d

The optimized code of DAG representation is



2) Write the 3-address code & basic block for the given code.

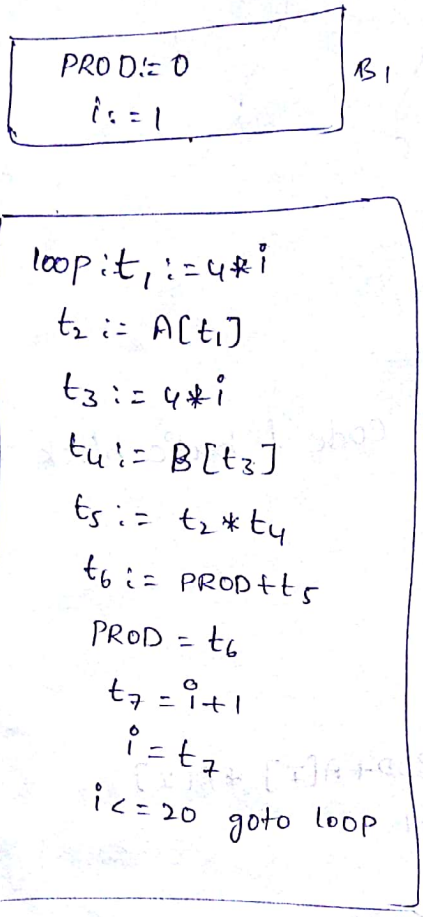
```
begin
PROD:=0
I:=1
do
  begin
  PROD:= PROD+A[I] *B[I]:
    I:= I+1
    END
  While I<=20
      END
```

Sol

3- address code

1) PROD:=0
2) i:=1.
3) loop: $t_1 := 4 * i$
4) $t_2 := A[t_1]$
5) $t_3 := 4 * i$
6) $t_4 := B[t_3]$
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) $i <= 20$ goto loop (2)

Basic Block:



Applications of DAG :-

1) We define common sub expression.
2) eliminate the common sub expression.
3) Eliminate the dead code.