

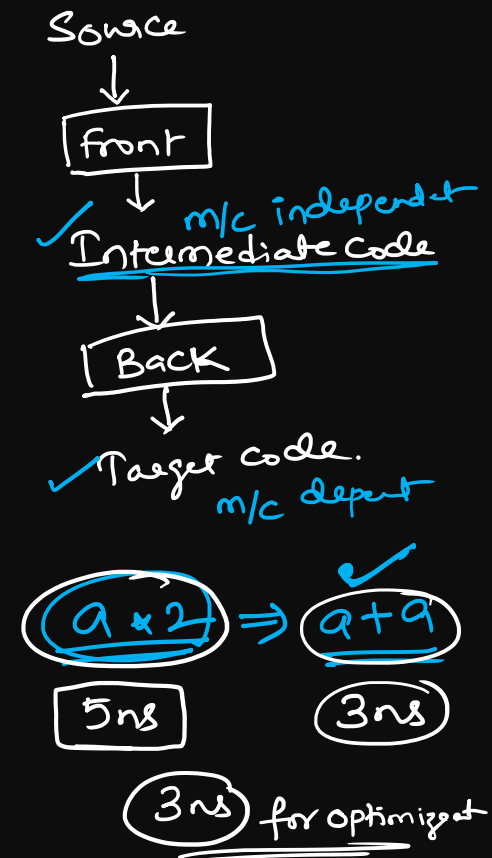
Code Optimization

Code optimization phase is an optional phase in the phases of a compiler, which is either before or after the code generation phase.

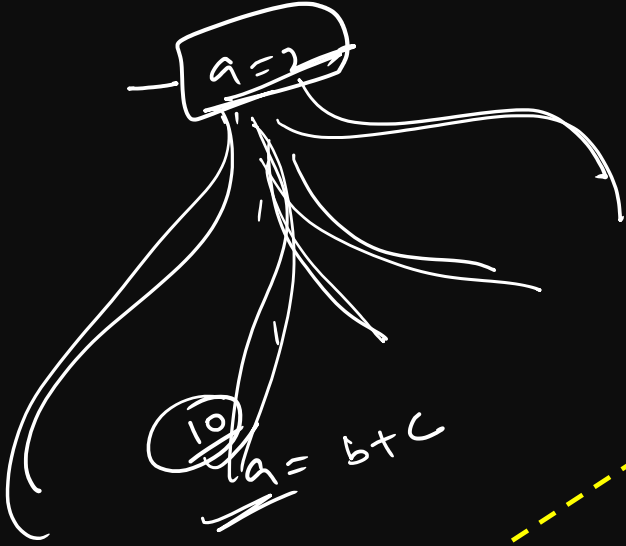
The code optimization is required to produce an efficient target code, this improvement is achieved by program transformations that are traditionally called optimizations.

The transformations provided by an optimizing compiler should have the following properties.

1. The transformation must not change the meaning of the original source program.
2. The Optimizing technique should increase the speed of program by a considerable amount.
3. The transformation technique must justify the effort.



Code Optimization Model



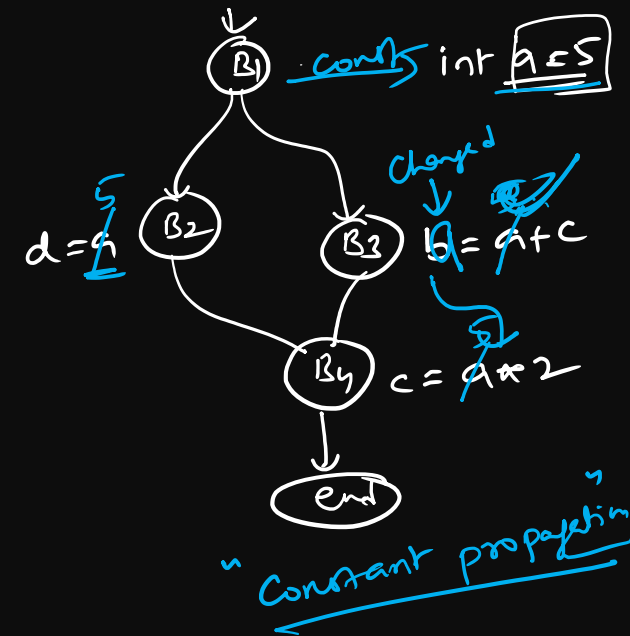
Code Optimizer

Control Flow Analysis

Data Flow Analysis

Transformations

1. **Control flow analysis:** It determines the control flow structure of a program and builds a control flow graph.
2. **Data flow analysis:** It determines the flow of scalar values, that is flow analysis propagates data flow information along a flow graph.
3. **Transformation:** Transformation helps in improving the code without changing the meaning or functionality.



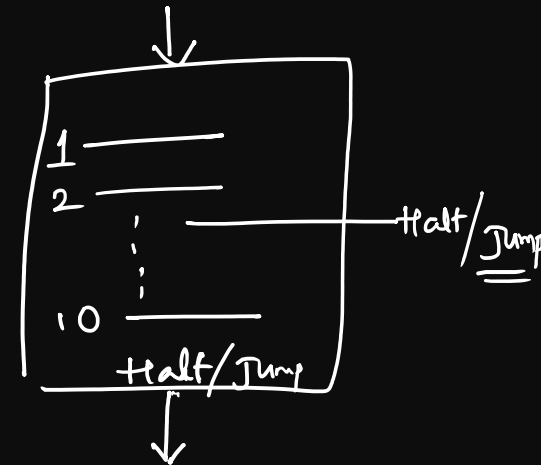
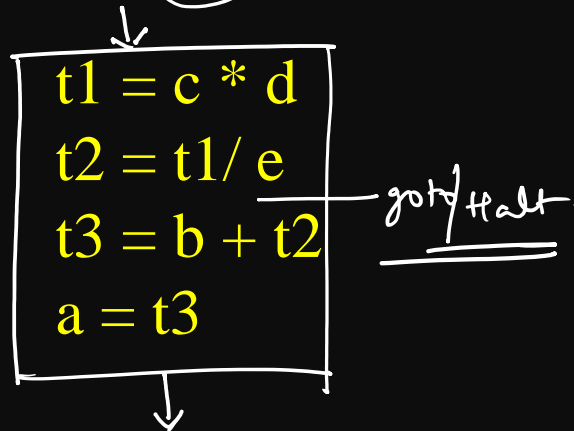
Basic Blocks

A Basic Block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibly of branching except at the end.

Example:

$a = b + c * d / e$

TAC:



Procedure to identify the Basic Blocks

Given a Three Address Code, first identify the leader statement and group the leader statement with the statements up to the next leader.

To identify the leader use the following rules:

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

Handwritten example illustrating leader identification:

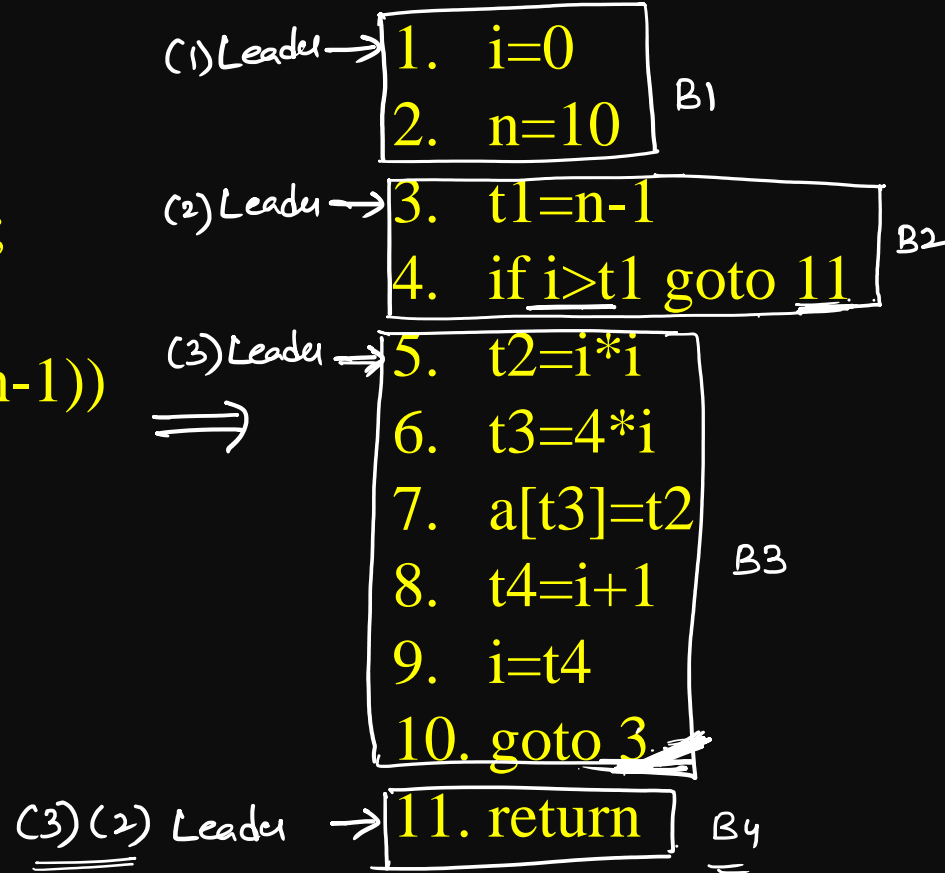
```
1. if a > b goto 4 ← leader
2.
5. goto 10 ← leader.
6
```

Arrows indicate the flow from statement 1 to 4 and from statement 5 to 10, identifying 1, 4, and 5 as leaders.

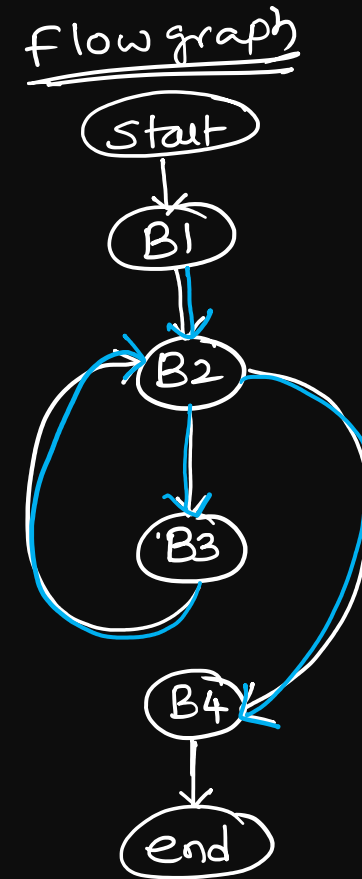
Example

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
}
```

3. t1=n-1
4. if i>t1 goto 115. t2=i*i
6. t3=4*i
7. a[t3]=t2
8. t4=i+1
9. i=t4
10. goto 3

11. return



no of edges = 6
no of nodes = 6
no of blocks = 4
no of controls = 4

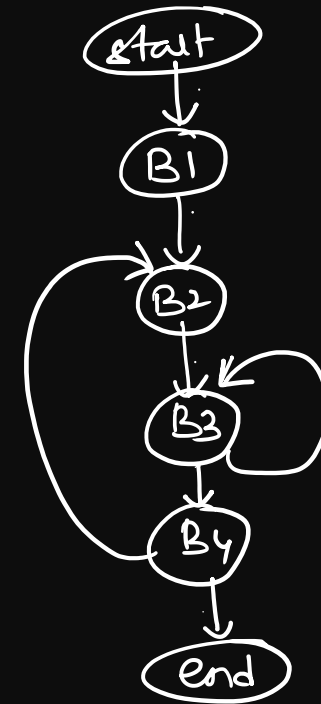
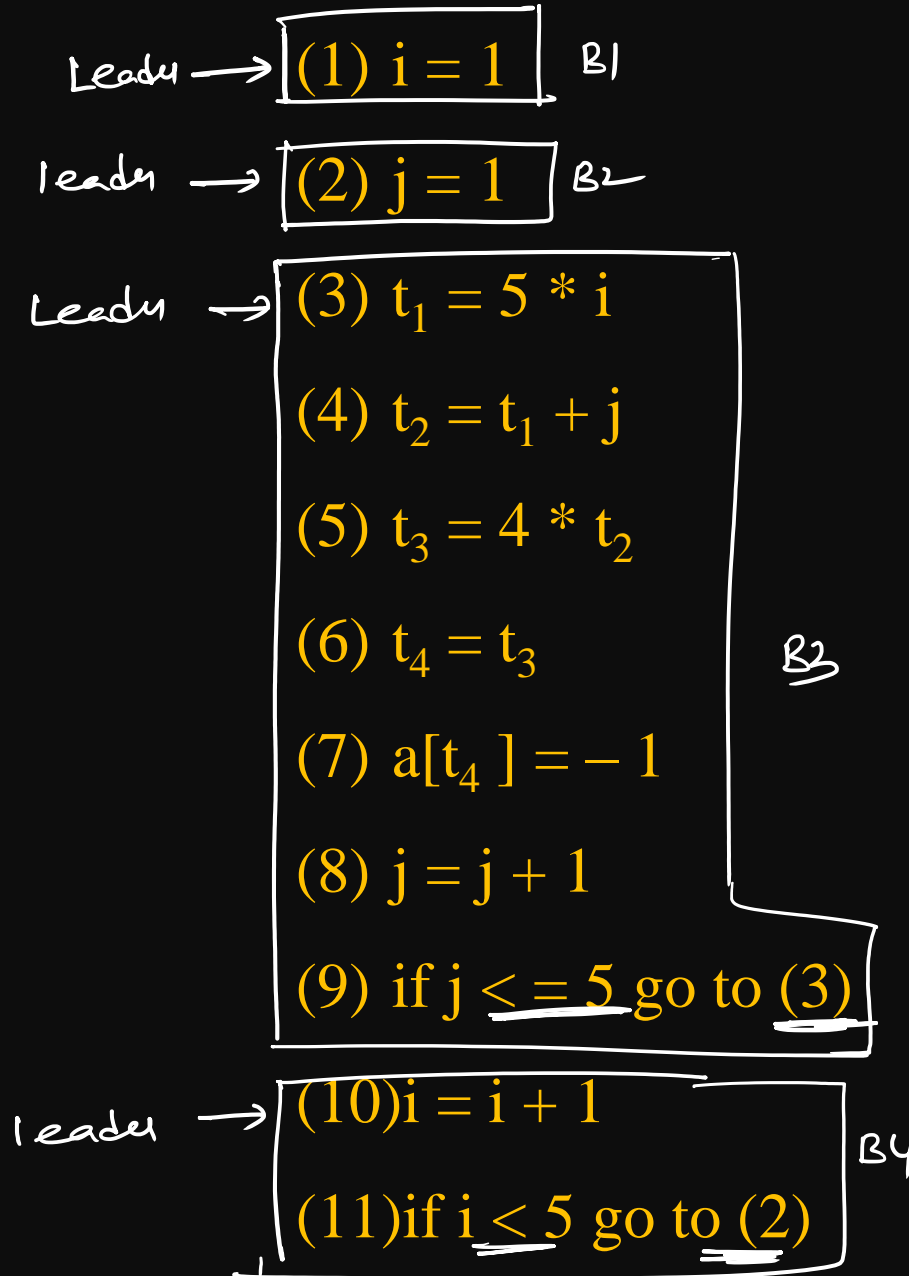
Flow Graph

A graphical representation of three address code is called flow graph.

The nodes in the flow graph represent a single basic block and the edges represent the flow of control.

These flow graphs are useful in performing the control flow and dataflow analysis and to apply local or global optimization.

Q. Consider the intermediate code given below.



no of nodes = 6
edges = 7

The number of nodes and edges in the control-flow graph constructed for the above code, respectively, are

(GATE-15- SET2)

(a) 5 and 7

~~(b) 6 and 7~~

(c) 5 and 5

(d) 7 and 8

Q. Consider the following grammar:

✓ stmt \rightarrow if expr then expr else expr; stmt | ϵ

expr \rightarrow term relop term | term

term \rightarrow id | number

id \rightarrow a | b | c

number \rightarrow [0-9]

where relop is relational operator (e.g., , ...), ϵ refers to the empty statement, and if, then, else are terminals.

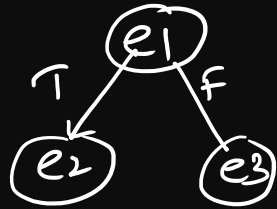
Consider a program P following the above grammar containing ten if terminals.

The number of control flow paths in P is 1024.

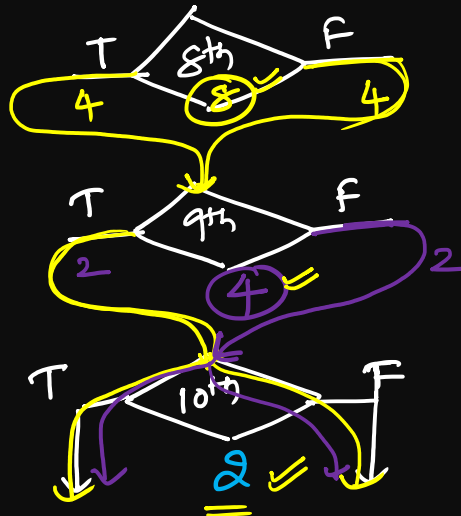
For example, the program

if e_1 then e_2 else e_3 has 2 control flow paths, $e_1 \rightarrow e_2$ and $e_1 \rightarrow e_3$.

(GATE-17-SET1)



2 Controls $e_1 \rightarrow e_2$
 $e_1 \rightarrow e_3$



$$1^{st} = 1024$$

$$2^{nd} = 512$$

$$3^{rd} = 256$$

$$4^{th} = 128$$

$$5^{th} = 64$$

$$6^{th} = 32$$

$$7^{th} = 16$$

$$8^{th} \Rightarrow 8$$

$$9^{th} \Rightarrow 4$$

$$10^{th} \Rightarrow 2$$

Classification of Code Optimization

Code Optimization

```
graph TD; CO[Code Optimization] --> MD[Machine Dependent]; CO --> MI[Machine Independent];
```

Machine Dependent

- It considers the m/c properties such as instruction format, addressing modes, Register set
- It is performed on Target code

Machine Independent

- It does not consider the m/c properties.
- It depends on program logic
- It is performed on intermediate code.

Scope of Optimization

1. Local Scope: When a local optimization has to be performed its scope is limited to certain specific block of statements
 - Common subexpression
 - dead code
 - Copy propagation
2. Global Scope: When a global optimization has to be performed, its scope is throughout the program. It is performed using dataflow analysis.
 - Code motion
 - strength reduction.

Local Optimization

1. Common sub expression elimination
2. Copy propagation
3. Dead code elimination
4. Constant Propagation
5. Constant folding
6. Strength Reduction

Common sub expression elimination

An expression E is called a common sub expression if it was previously computed, and the values of variables in E have not changed since the previous computation.

In common sub expression elimination we have to avoid recomputing of the expression if it's value is already been computed

Example: $a = b + c$
 $c = a + d$
 $e = b + c$ ← not C.S.E
 $f = a + d$ ← CSE

\Rightarrow

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

$a + d$ is CSE

$\checkmark a = \overset{\text{'E'}}{\underline{b + c}} \checkmark$
.....
 $\checkmark d = \frac{\cancel{b + c}}{\text{E}} \checkmark$
 $d = a$

Common sub expression elimination

Example:

$t1 = 4 * i$

$n = a[t1]$

$t2 = 4 * i$ ← CSE

$t3 = 4 * j$

$t4 = a[t3]$

$a[t2] = t4$

$t5 = 4 * j$ ← CSE

$a[t5] = n$



$t1 = 4 * i$

$n = a[t1]$

$t2 = t1$

$t3 = 4 * j$

$t4 = a[t3]$

$a[t2] = t4$

$t5 = t3$

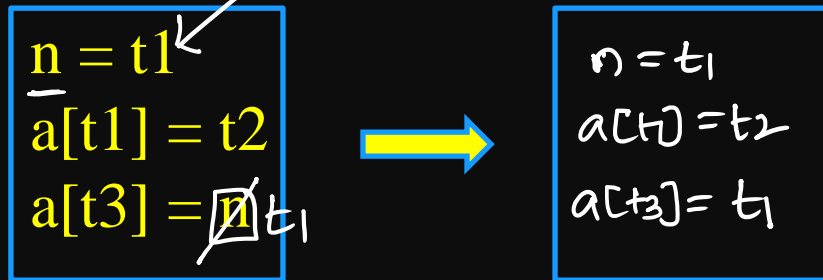
$a[t5] = n$

Copy Propagation

Copy propagation means use of one variable instead of another.

If $x=y$ is a statement, which is called copy statement, then copy propagation is a kind of transformation in which use y for x wherever possible after the copy statement $x=y$.

Example:



$x=y$ Dead
:
 $a = b + \cancel{x} y$

Copy Propagation

Example:

```
t1 = 4*i
n = a[t1]
t2 = 4*i t1
t3 = 4*j
t4 = a[t3]
a[t2] = t4
t5 = 4*j t3
a[t5] = n
```



```
t1 = 4*i
n = a[t1]
t2 = t1 ← copy
t3 = 4*j
t4 = a[t3]
a[t2] = t4
t5 = t3 ← copy
a[t5] = n
```



```
t1 = 4*i
n = a[t1]
t2 = t1 Dead
t3 = 4*j
t4 = a[t3]
a[t1] = t4
t5 = t3 Dead
a[t3] = n
```

After Common Sub
Expression Elimination

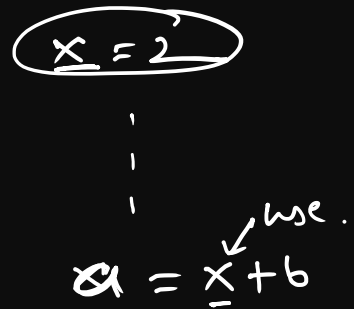
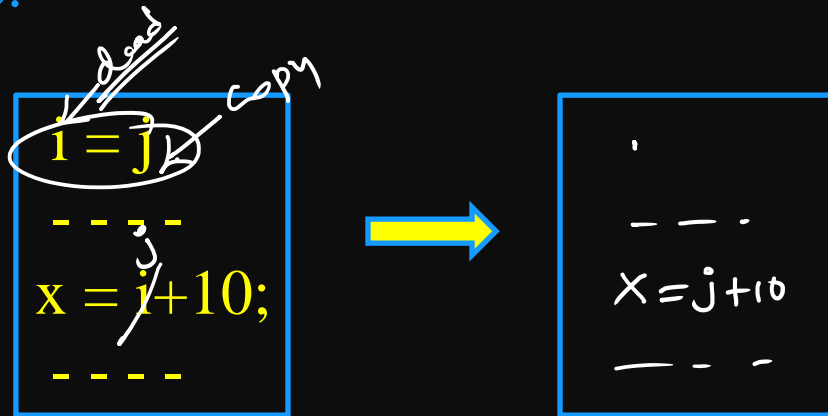
Copy propagation

Dead code Elimination

A variable is said to be live in the program if its value can be used subsequently otherwise it is said to be dead at that point.

The dead code is basically an useless code. An Optimization can be performed by eliminating such a dead code.

Example:



Dead Code Elimination

Example:

```
t1 = 4*i
n = a[t1]
t2 = 4*i
t3 = 4*j
t4 = a[t3]
a[t2] = t4
t5 = 4*j
a[t5] = n
```



```
t1 = 4*i
n = a[t1]
t2 = t1
t3 = 4*j
t4 = a[t3]
a[t2] = t4
t5 = t3
a[t5] = n
```



```
t1 = 4*i
n = a[t1]
t2 = t1
t3 = 4*j
t4 = a[t3]
a[t1] = t4
t5 = t3
a[t3] = n
```



```
t1 = 4*i
n = a[t1]
t3 = 4*j
t4 = a[t3]
a[t1] = t4
a[t3] = n
```

After Common Sub
Expression Elimination ✓

After ✓
Copy Propagation

Dead code ✓

Constant Propagation

It can be defined as the process of replacing the constant value of variables in the expression.

Constants assigned to a variable can be propagated through the flow graph and can be replaced when the variable is used.

Const int a = 2;
- -
- -
p = ~~a~~ + b
 ↑
 2

Example:

const int a = 30;
b = 20 - a / 2;
c = b * (30 / a + 2) - a;



Const int a = 30;
b = 20 - 30 / 2 b = 5 ✓
c = b * (30 / 30 + 2) - 30

Constant Folding

It is a technique in which the constant expressions are calculated during compilation.

Replacing expressions consisting of constants with their final value at compilation time rather than doing the calculation at run time.

Example:

```
const int a = 30;
```

```
b = 20 - a / 2;
```

```
c = b * ( 30 / a + 2 ) - a;
```



```
const int a = 30;
```

```
b = 20 - 30 / 2;
```

```
c = b * ( 30 / 30 + 2 ) - 30;
```

constant propagation



```
const int a = 30;
```

```
b = 5;
```

```
c = b * 3 - 30;
```

constant folding.

Strength Reduction

Strength reduction is used to replace the expensive operation by the cheaper one .

Example: $a \times 2 \Rightarrow a + a$

$$a / 2 \Rightarrow a \times 0.5$$

$$a \times 4 \Rightarrow a \times 2^2 \Rightarrow \underline{\underline{a \ll 2}}$$

Loop Optimization

Loop optimization is a technique in which code optimization is performed on inner loops.

1. Code Motion
2. Induction variables and strength reduction
3. Loop Unrolling
4. Loop Jamming

Code Motion

Code motion is a technique which moves the code outside the loop.

If there is some expression in the loop whose result remains unchanged even after executing the loop several times, then such an expression should be placed just before the loop.

Ex:-

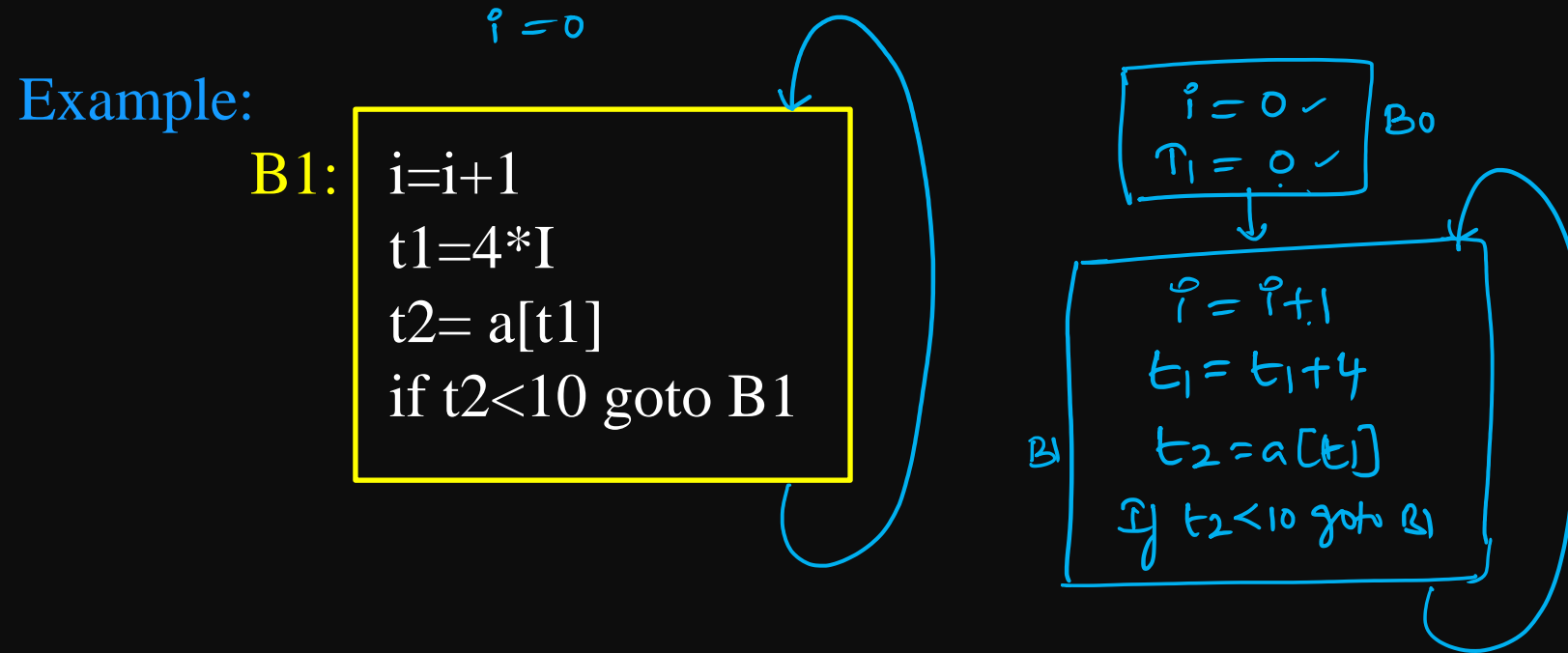
```
sum = 0
while (i <= max-1)
{
    sum = sum + a[i]
    i++
}
```

\Rightarrow

```
sum = 0 ,
t1 = max-1;
while (i <= t1)
{
    sum = sum + a[i] ;
    i++
}
```

Induction variables and reduction in strength

A variable x is called an induction variable of loop, if the value of variable gets changed every time, it is either incremented or decremented by some constant.



$$\left\{ \begin{array}{l} i = 0, 1, 2, 3, 4, \dots \\ t1 = 0, 4, 8, 12, 16, \dots \end{array} \right\} \text{Induction Variables}$$

$$t1 = 4 \times i \quad \text{--- ①}$$

$$t1 = 4 \times (i + 1)$$

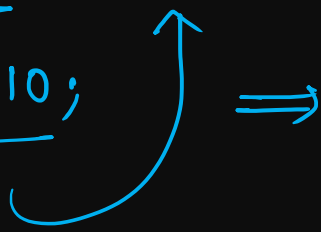
$$t1 = \underline{4 \times i} + 4$$

$$\underline{t1 = t1 + 4}$$

Loop Unrolling

In this method we duplicate the body of the loop several times, in order to decrease the number of times loop condition is tested and number of jumps.

```
for (i=1; i<=100; i++)  
{  
    A[i] = 10;  
}
```



Condition Tested = 100 times ✓
no of Jumps 100 times

i = 1, 2, 3, - - -

```
for (i=1; i<=100; i++)  
{  
    A[i] = 10;  
    i++;  
    A[i] = 10;  
}
```

i = 1, 3, 5, - - -

Condition Tested = 50 times ✓
no of Jumps = 50 times.

Note: this approach requires that, the number of iterations to be known at compile time.

Loop Jamming / Loop Combining ? Loop Fusion

In this method several loops are merged to one loop.

It is another technique which attempts to reduce loop overhead when two adjacent loops would iterate the same number of times, their bodies can be combined as long as they make no reference to each others data.

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

```
for (j=0; j<10; j++)  
{  
    B[j] = 5;  
}
```

⇒

```
for (i=0; i<10; i++)  
{  
    A[i] = 10;  
    B[i] = 5;  
}
```

Data Flow Analysis

In Dataflow analysis the analysis is made on the flow of data. That is it determines the information regarding the definition and use of the data in the program. Using this kind of analysis optimization can be done.

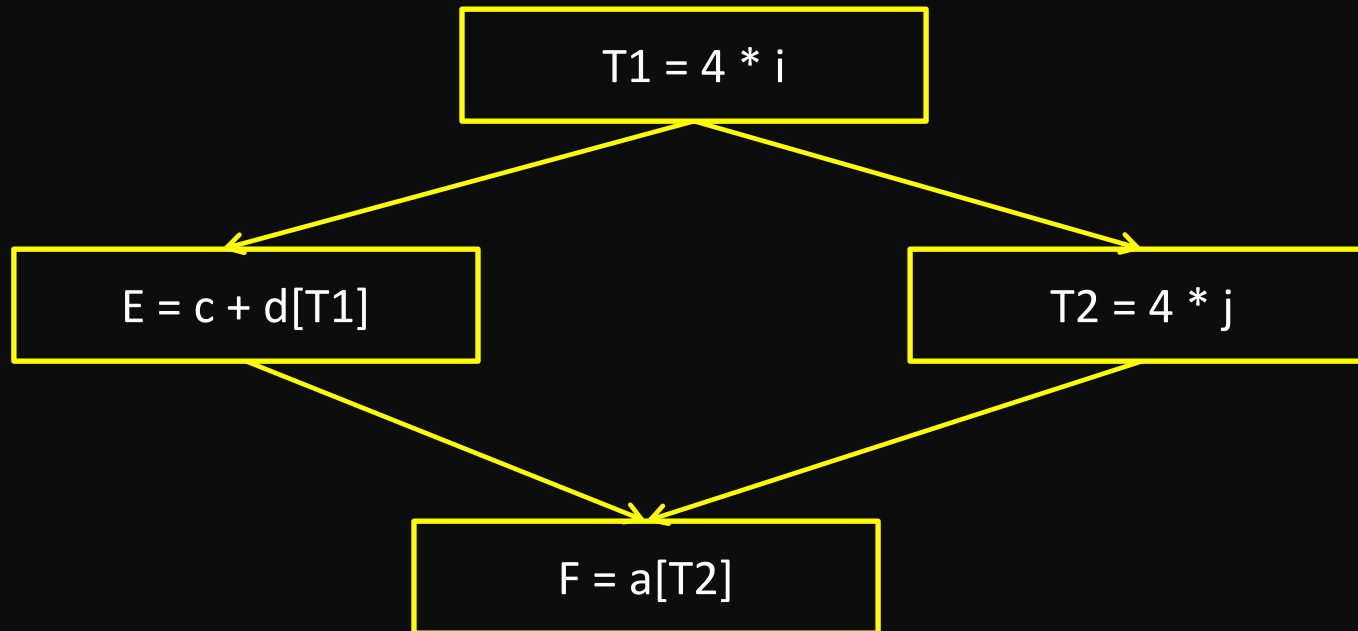
The Dataflow analysis is basically a process in which the values are computed using dataflow properties. The dataflow property represents the certain information regarding usefulness of the data items for the purpose of optimization.

These dataflow properties are,

1. Available expressions
2. Reaching definitions
3. Live variables
4. Busy Expressions

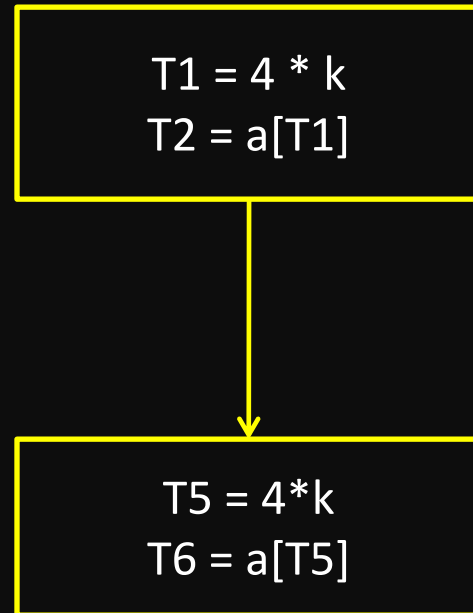
Available Expression

An expression $x+y$ is available at point P if every path from the node to P evaluates $x+y$ and after the last such evaluation prior to reaching P , there are no subsequent assignments to X or Y .



Available Expression

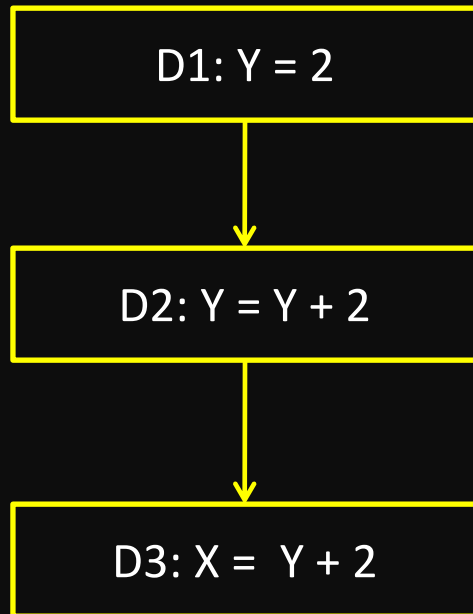
Advantage: The use of available expression is to eliminate Common Sub Expressions.



Reaching Definition

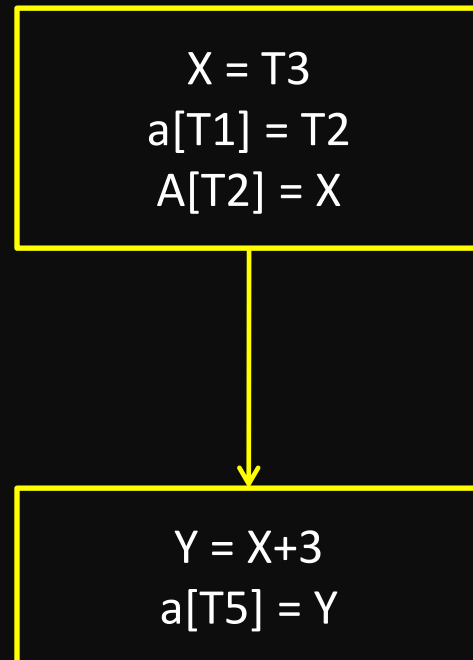
A definition 'D' reaches at point P if there is a path from D to P along which D is not killed.

A definition 'D' of variable x is killed when there is a redefinition of x.



Reaching Definition

Advantage: The Reaching definitions are used in constant and variable propagation.



Live Variables

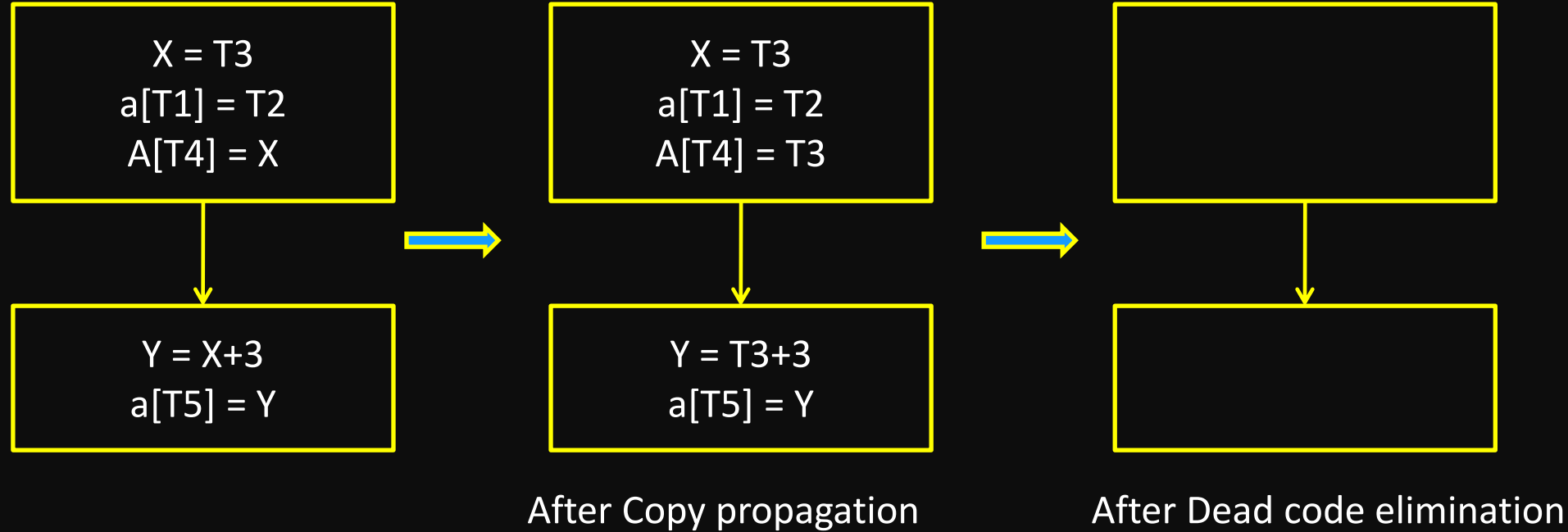
A Variable 'X' is live at some point 'P' if there is some path from 'P' to the exit, along which the value of 'X' is used before it is redefined, otherwise it is said to be dead at that point.

Live Variables

Advantage: Live variables are useful for dead code elimination.

If variable X is not live after an assignment $X = \dots$

Then the assignment is redefined and can be deleted as dead code.



Live Variables

Advantage: Live variables are useful in register allocation.

If variable X is live in basic block B , it is potential candidate for register allocation.

Busy Expression

An expression 'E' is said to be a busy expression along some path $P_i - - - P_j$ if and only if an evaluation of E exists along some path $P_i - - - P_j$ and no definition of any operand exists before its evaluation along the path.

```
i = 0  
T1 = max - 1  
i = i + 1  
If T1 <= 10 goto B1
```

Advantage: Busy Expressions are useful in performing code movement optimization.

Conclusion

By performing Data flow analysis we get available expressions, reaching definitions, live variables and busy expressions and these are useful for applying various global common sub expression elimination, copy propagation, Dead code elimination and code movement optimizations.

Previous GATE Question's

Q. Which one of the following is **FALSE**?

(GATE – 14 – SET1)

- (a) A basic block is a sequence of instructions where control enters the sequence at the beginning and exists at the end.
- (b) Available expression analysis can be used for common sub-expression elimination.
- (c) Live variable analysis can be used for dead code elimination.
- (d) $x = 4 * 5 \Rightarrow x = 20$ is an example of common sub-expression elimination

Previous GATE Question's

Q. Consider the following C code segment.

```
for (i = 0; i < n; i++)  
{  
    for (j = 0; j < n; j++)  
    {  
        if (i % 2)  
        {  
            X += (4 * j + 5 * i);  
            Y += (7 + 4 * j);  
        }  
    }  
}
```


Which one of the following is **FALSE**?

(GATE - 06)

- (a) The code contains loop-invariant computation.
- (b) There is scope of common sub expression elimination in this code.
- (c) There is scope of strength reduction in this code.
- (d) There is scope of dead code elimination in this code.

Previous GATE Question's

(a) The code contains loop-invariant computation.

Q. Consider the following C code segment.

```
for (i = 0; i < n; i++)  
{  
    for (j = 0; j < n; j++)  
    {  
        if (i % 2)  
        {  
            X += (4 * j + 5 * i);  
            Y += (7 + 4 * j);  
        }  
    }  
}
```

Previous GATE Question's

(b) There is scope of common sub expression elimination in this code.

Q. Consider the following C code segment.

```
for (i = 0; i < n; i++)  
{  
    for (j = 0; j < n; j++)  
    {  
        if (i % 2)  
        {  
            X += (4 * j + 5 * i);  
            Y += (7 + 4 * j);  
        }  
    }  
}
```

Q. Consider the following C code segment.

```
for (i = 0; i < n; i++)  
{  
    for (j = 0; j < n; j++)  
    {  
        if (i % 2)  
        {  
            X += (4 * j + 5 * i);  
            Y += (7 + 4 * j);  
        }  
    }  
}
```

Q. Consider the following C code segment.

```
for (i = 0; i < n; i++)  
{  
    for (j = 0; j < n; j++)  
    {  
        if (i % 2)  
        {  
            X += (4 * j + 5 * i);  
            Y += (7 + 4 * j);  
        }  
    }  
}
```

Previous GATE Question's

Q. Consider the following ANSI C code segment:

```
z = x + 3 + y → f1 + y → f2;
```

```
for (i = 0; i < 200; i = i + 2 )
```

```
{
```

```
    if (z > i)
```

```
    {
```

```
        p = p + x + 3;
```

```
        q = q + y → f1;
```

```
    }
```

```
    else
```

```
    {
```

```
        p = p + y → f2;
```

```
        q = q + x + 3;
```

```
    }
```

```
}
```

Previous GATE Question's

Assume that the variable y points to a struct (allocated on the heap) containing two fields $f1$ and $f2$ and the local variables x , y , z , p , q and i are allotted registers. Common sub-expression elimination (CSE) optimization is applied on the code. The number of addition and dereference operations (of the form $y \rightarrow f1$ or $y \rightarrow f2$) in the optimized code, respectively, are:

(GATE - 21-Set2)

- (a) 403 and 102
- (b) 303 and 102
- (c) 203 and 2
- (d) 303 and 2

Previous GATE Question's

Q. For a statement S in a program, in the context of liveness analysis, the following sets are defined.

USE (S): the set of variables used in S

IN (S): the set of variables that are live at the entry of S

OUT (S): the set of variables that are live at the exit of S

Consider a basic block that consists of two statements

S_1 followed by S_2 .

Which one of the following statements is correct? **(GATE - 21-Set2)**

(a) $OUT(S_1) = IN(S_1) \cup USE(S_1)$

(b) $OUT(S_1) = IN(S_2)$

(c) $OUT(S_1) = USE(S_1) \cup IN(S_2)$

(d) $OUT(S_1) = IN(S_2) \cup OUT(S_2)$