Page No.
Date : / /

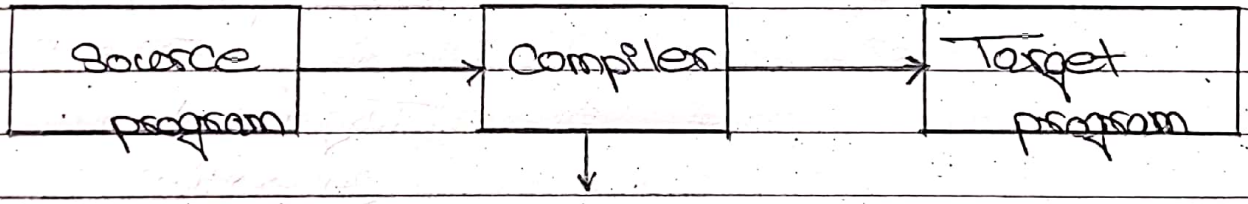
Unit : 1

Review of Compiler Structure and Modern Architecture

* Review of compiler :

A compiler is a program that can read a program in one language (Source language) and translate it into an equivalent program in another language (Target language) as output. The most well known form of a compiler is one that translates a high level language like C into the native assembly language of a machine so, that it can be executed.

An important role of the compiler is to report any errors in the source program that it detects during the Translation process.



* Structure of a Compiler :

The Analysis part breaks up the source program into constituent pieces and impose a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. If the analysis part detects that the source program is either syntactically or semantically wrong, then it must provide informative messages to user, so that users can take corrective action. The analysis part also

collects the information about the source program & store it in a data structure called a symbol table.

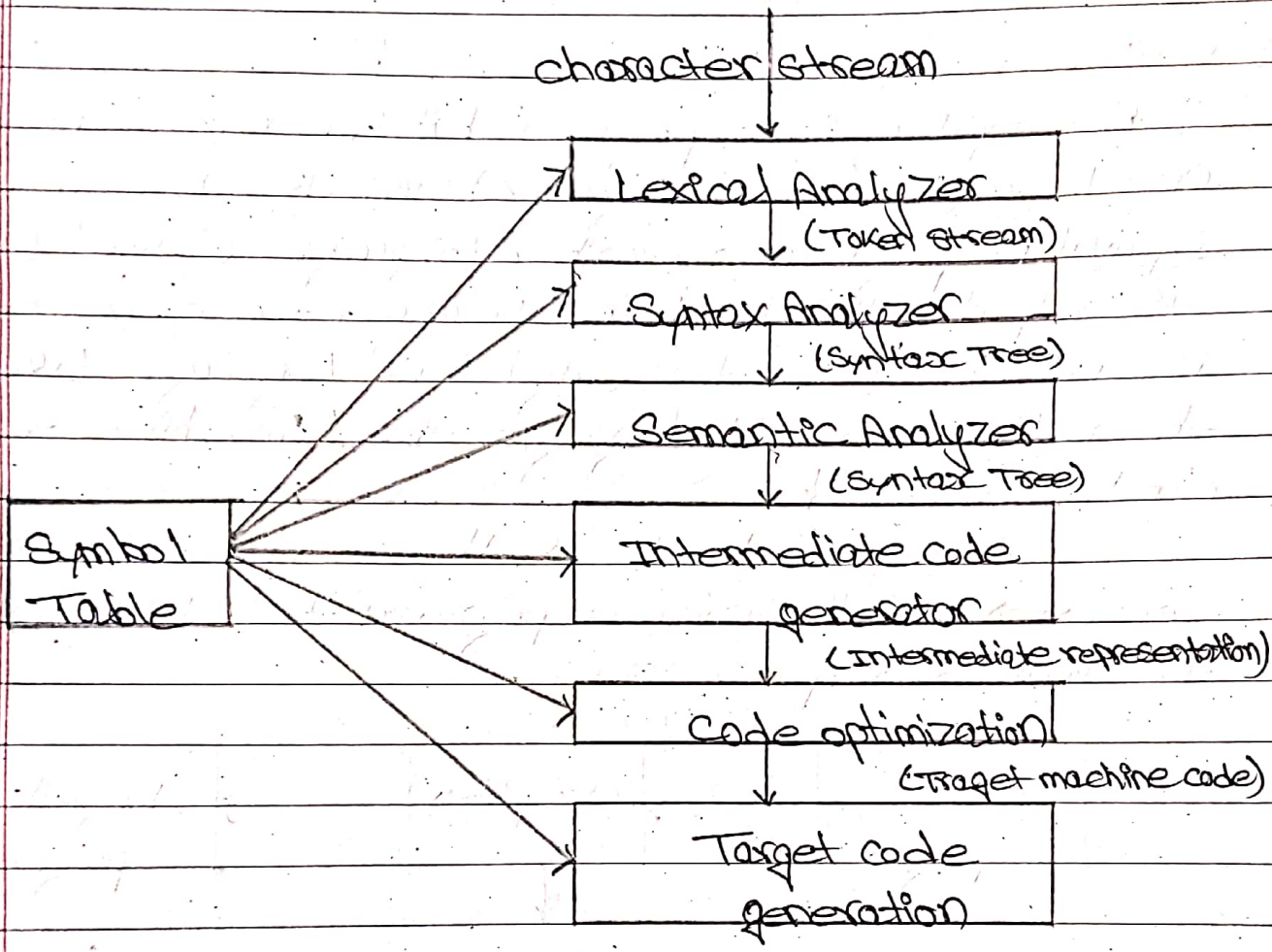


Fig of :- Phases of compiler.

Foreg :-

Position = Initial + rate * 60 %.

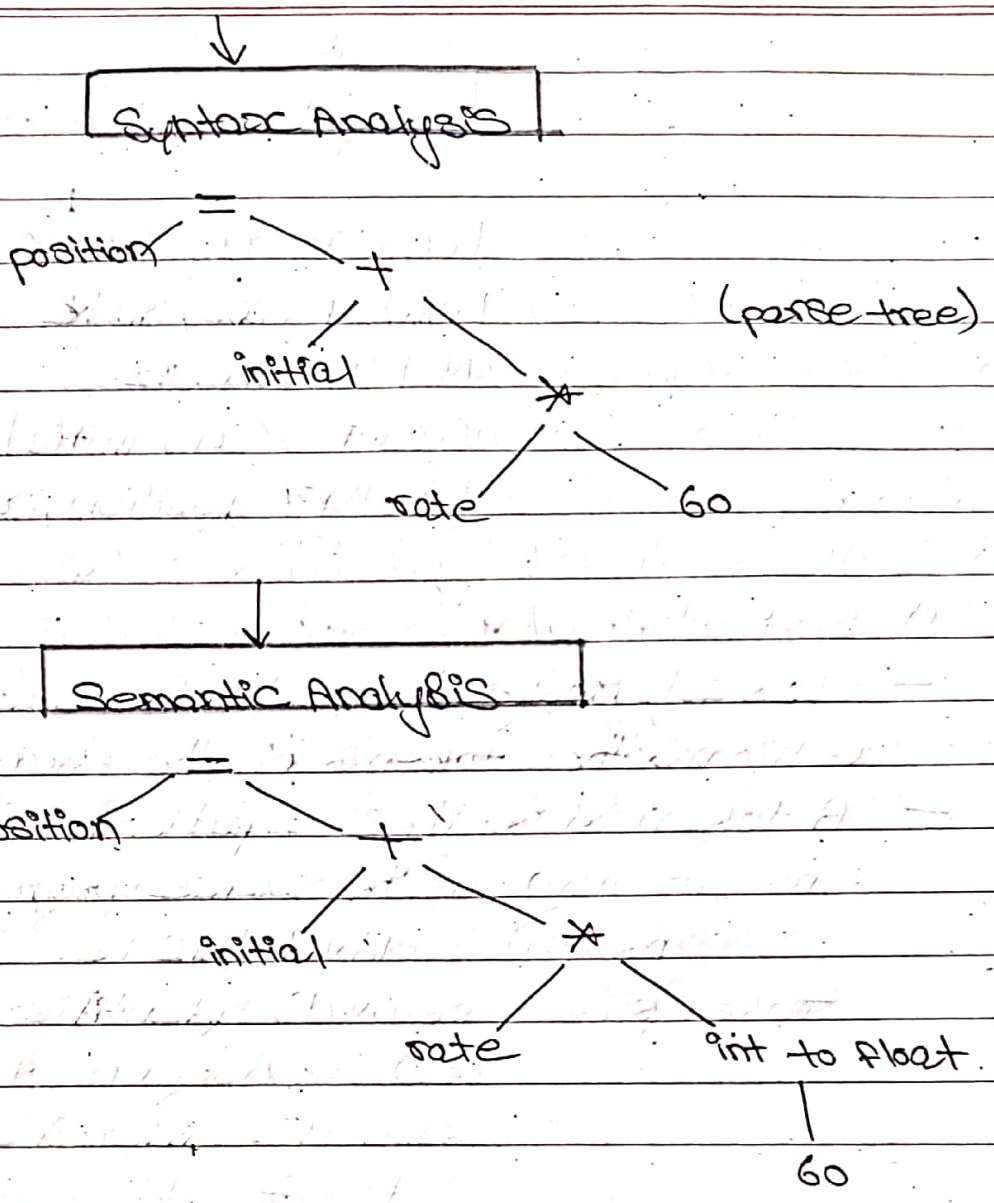
↓

Lexical Analysis

Position = Initial + range * 60

(id1) (Assignment operator) (id2) (plus) (id3) (mul) (integer num)

Page No.
Date : / /



↓

Target code Generation

```
LOAD F r1, #60.0
LOAD F r2, rate
MUL F r1, r2
ADD F r1, initial
LOAD F position, r1
```

1) Lexical Analysis:

- Lexical Analyzer reads the source program and returns the tokens of the source program.
- A token describes a pattern of characters having same meaning in the source program. for eg:-

  newval = oldval + 12 i.e.

  Tokens:- newval = identifier
  (=) = Assignment operator
  oldval = identifier
  + = add operator
  12 = a number.

- Token, patterns, Lexemes
- Patterns is the rule to define the validity of a token
- Valid tokens are lexemes.

  e.g:- int → token → Lexemes

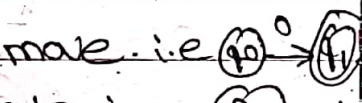
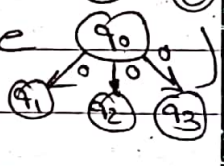
  1a → token → not lexemes.

Page No.
Date : / /

- Puts information about identifier into the symbol table.
- In lexical Analysis regular expression is used to define the patterns of lexeme.
  1) alphabet,
  2) String,
  3) Kleene closure ($A^* \rightarrow \epsilon$)
  4) Positive closure ($A^+ \rightarrow A^* - \{\epsilon\}$)

- A Deterministic Finite state Automaton can be used in the implementation of a lexical Analyzer.

* Recognition of tokens:

- DFA (have single choice of transition move. i.e 
- NFA (have multiple choice of transition move. i.e 

* DFA (Deterministic Finite Automata)

Formally, it can be defined as

$M = (Q, \Sigma, q_0, \delta, F)$ where,

Q = finite set of states,

Σ = finite set of input alphabet,

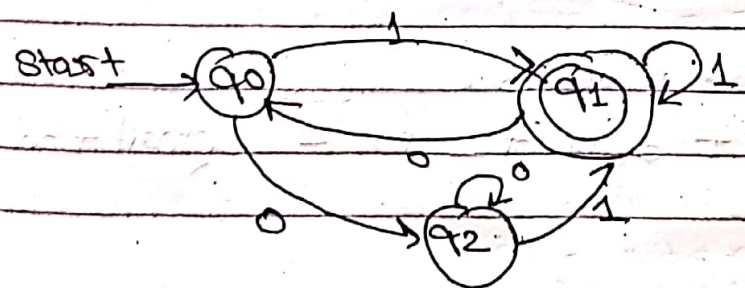
q_0 = Starting symbol

δ = Transition function $\delta : Q \times \Sigma \rightarrow Q$

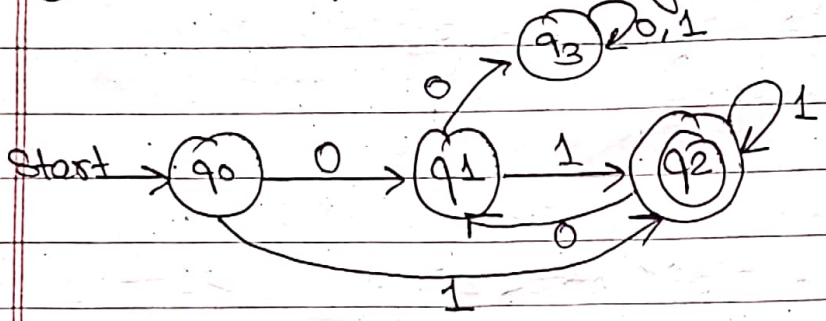
$\{\delta_{NFA} : Q \times \Sigma \Rightarrow 2^{|Q|}\}$

F = finite set of final states.

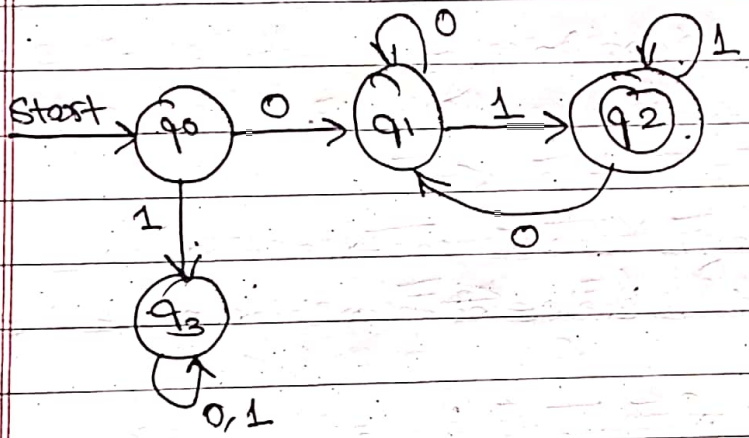
example: $(0+1)^* 1$



(ii) 0 is always followed by 1 :



(iii) $0(0+1)^*1$



* Regular Expression :

A regular expression is basically a shorthand way of showing how a regular language is built from the base set of regular languages.

i.e For each regular expression E, there is a regular language L(E).

For eg :-

Alpha → a-z, A-Z

digit → 0, 1, 2, 3, ... 9

(A) R-E for Gmail or hotmail id :

= [Alpha][Alpha + digit + _ + .]* [gmail + hotmail].com

Page No.
Date : / /

2) Syntax Analysis :

- The ~~SA~~ Second phase of the compiler is Syntax Analysis or parsing.
- The Syntax Analysis (parser) uses the tokens produced by the lexical Analyzer to create a tree-like intermediate representation (parse tree) of the given program.
- It helps to check the grammatical pattern of a language.
- CFG (Context free Grammar) is used for Syntax definition.

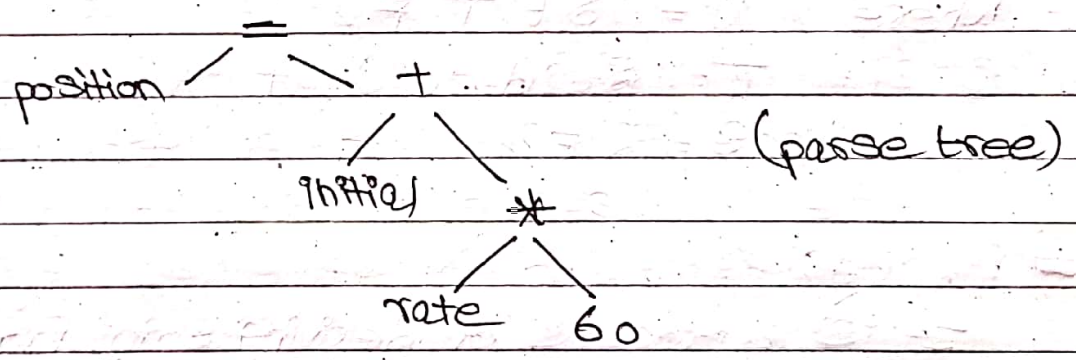
for e.g :-

1) Lexical Analysis :- position = initial + rate * 60

(id 1) (Ass-op) (id 2) (plus) (id 3) (mul) (int-num)

↓

Syntax Analysis



(parse tree)

* Note : ① In parse tree, interior node represents the operations ~~(non-terminals)~~

② The children of the node represent the arguments ~~(terminals)~~ of the operations.

Page No.
Date : / /

* CFG (Context free Grammar):

Formally, it can be defined by (V, T, P, S) where,

V → Set of variables (Non-Terminals)
T → Set of Terminals,
P → Set of production rule,
S → Starting variables,

eg:- Let the Grammar be:

$$E \rightarrow E+T \mid T$$
$$T \rightarrow T*F \mid F$$
$$F \rightarrow (E) \mid id$$

Where, V = {E, T, F}
T = {id, (,), +, *}
S = {E}

* Parsing:

It is a process of deriving string from a given grammar (i.e checking the grammar of the string).

Types: ① Top down parser,
② Bottom up parser.

Page No.
Date : / /

* Left Recursive Grammar:

A grammar having the production of the form:

$$A \rightarrow A\alpha + \beta$$

can be removed by:

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \epsilon$$

① For example:

$$B \rightarrow B\underbrace{ab}_{\alpha} \mid \underbrace{0}_{\beta_1} \mid \underbrace{1}_{\beta_2}$$

now, Removing left recursion:

i.e,

$$B \rightarrow 0B' \mid 1B'$$
$$B' \rightarrow abB' \mid \epsilon$$

Example ②:

$$E \rightarrow E\underbrace{+T}_{\alpha_1} \mid E\underbrace{*T}_{\alpha_2} \mid \underbrace{(E)}_{\beta_1} \mid \underbrace{id}_{\beta_2}$$

Removing left recursion:

$$E \rightarrow (E)E' \mid idE'$$
$$E' \rightarrow +TE' \mid *TE' \mid \epsilon$$

Example ③:

$$E \rightarrow E\underbrace{+T}_{\alpha} \mid \underbrace{T}_{\beta}$$

Removing left recursion

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \epsilon$$

3) Semantic Analysis:

- The Semantic Analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition.
- It checks the semantic meaning of a programming language
- An important part of semantic analysis is Type checking i.e. checking whether each operator has matching operands.
- Implicit / Explicit type conversion.

For example:

① $A \rightarrow B + C$ { A.type == B.type == C.type }

② $S_1 \Rightarrow$ if (boolean) $x = a + b$ else $y = c + d$ { if boolean == true then, S_1.type = x.type else, S_1.type = y.type }

③ if (test) $S_1 = A + B$ else $C - D$ { if test == true S_1.type = A_1.type else, S_1.type = C_1.type }

Page No.
Date : / /

e.g:-

① int x;
char a;
float b;

x = a+b;

- No syntax error
- but Semantically wrong

② const int z;
z = a+b;
- No syntax error
- Semantic error.

* Synthesized and Inherited Attributes:

- If the type of attribute depends upon the child, then it is called synthesized attribute.
  for e.g: c = a+b

- If the type of attribute depends on either parents or siblings, then it is called inherited attributes;
  e.g:- int a, b, c.

* Importance of optimization:

- Using a non-optimizing will usually result a non-efficient code when compare to more sophisticated ~~compilers~~ compilers.

Page No.
Date : / /

① For Example:

```
int a, b, c, d
c = a+b;
d = c+1;
```

Fig 1: C code

```
LDW a, R1
LDW b, R2
ADD R1, R2, R3
STW R3, C
LDW C, R3
ADD R3, 1, R4
STW R4, d
```

Fig 2:- SPARC Code (7-cycle)

#Non-optimized

```
ADD a, b, C
ADD C, 1, d
```

Fig 3:- optimized SPARC code (2 cycle)

#optimized.

- In above Example, the non-optimized SPARC code uses 7 instructions 5 of which are memory access instructions.
- The optimized code is more efficient and uses only 2 instruction in total.
- In terms of run-time improvement, the non-optimized code takes 7 cycles & the optimized code only 2 cycles.

Page No.
Date : / /

* Structure of optimizing Compiler:

A compiler designed to produce fast object code includes optimizer components.

There are mainly two models:

i) Source code is translated to a low level code and all the optimization is done on that form of code, we call this the low level model of optimization.

ii) Source code is translated into intermediate code & optimization is performed, then low level code is produced, and further optimization is done, called mixed model of optimization.

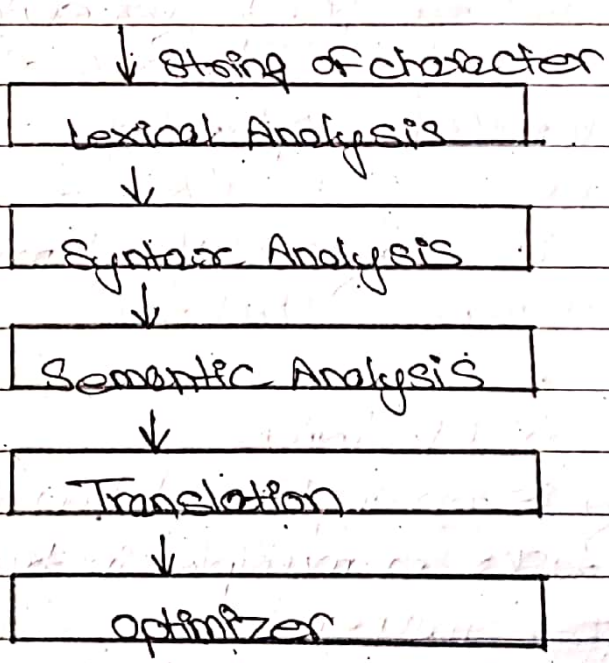


Fig 1:- Low-Level model

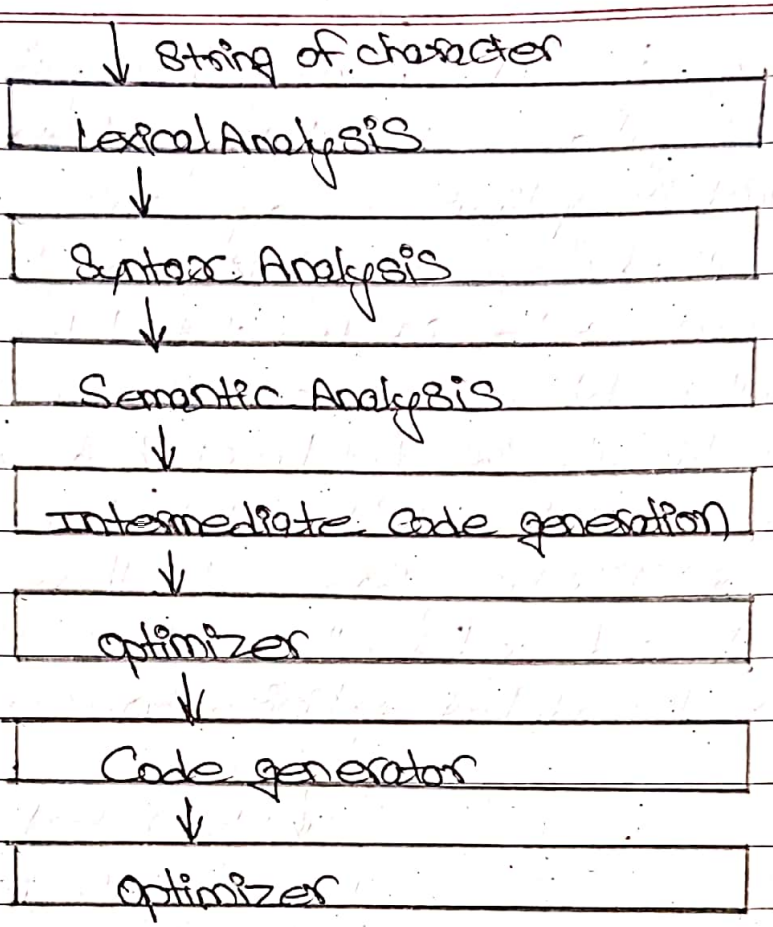


Fig 2:- Mixed model of optimization.

* Instruction Pipelining:

- Pipelining is an implementation technique, where multiple tasks or multiple instructions are performed in overlapped manner,
- It can be implemented when a task is divided into two or more independent sub-tasks.
- Pipelining is the key implementation technique used to make the CPU fast.
- For example: Consider a scenario, where A, B & C each have dirty clothes to be washed. The washing process has three stages: washing, drying &

folding and Each Stages takes 30 minutes to complete.
- unpipelined technique → 4.5 hrs
- Pipelined technique → 2.5 hrs.

* Pipeline Hazards:

- The condition or situation which does not allow the pipeline to operate normally.
- Reduces the performances,
- Types of Hazard: i) Control Hazard
  ii) Data Hazard
  iii) structural Hazard

i) Control Hazard:

- All instructions that change the program counter leads to the control hazards.
- When to make a decision based on the results of one instruction while other are executing.
- Control (Branch, Jump, call / Return) changes the value of PC.
- Eg:-

I1	→ 100
I2	→ 102
I3	→ 104
I4	→ 106
⋮	
I5	

Note: Let, intial PC = 100
next instruction to be executed is I2 which is in 102 M.L but Any instruction that made PC to read 104 (instruction like Branch, Jump, call / Return) Control Hazard occurs

100
PC

I2 → Fetch 102 PC } हुनु पर्ने चाहिँ यो?

I3 → Fetch 104 PC } भयो चाहिँ यो?

- The amount of the datapath that a signal travels through in one cycle is called a stage of the pipeline.
- Consider, the following 5 stage pipeline.

Instruction ↓	Clock cycles → 1	2	3	4	5	6	7	8	9
I1 :	IF	ID	EX	MEM	WB				
I2 :		IF	ID	EX	MEM	WB			
I3 :			IF	ID	EX	MEM	WB		

where, IF : Instruction Fetch
ID : Instruction Decode
EX : Execute
MEM : Memory Access
WB : Write Back.

* For example :- Control Hazard

I1 : BNEZ R1, 200 (BNEZ = Branch if not equal to zero)
if (R1 ≠ 0)
goto 200

Jump 200 Memory Location

	1	2	3	4	5	6	7
I1 :	IF	ID	EX	MEM	WB		
I2 :		IF	ID	EX	MEM	WB ✗	
I3 :			IF	ID	EX	MEM	WB ✗

I2 & I3 will not execute where, I1 Jumps to 200 Memory location resulting control hazard.

Page No.
Date : / /

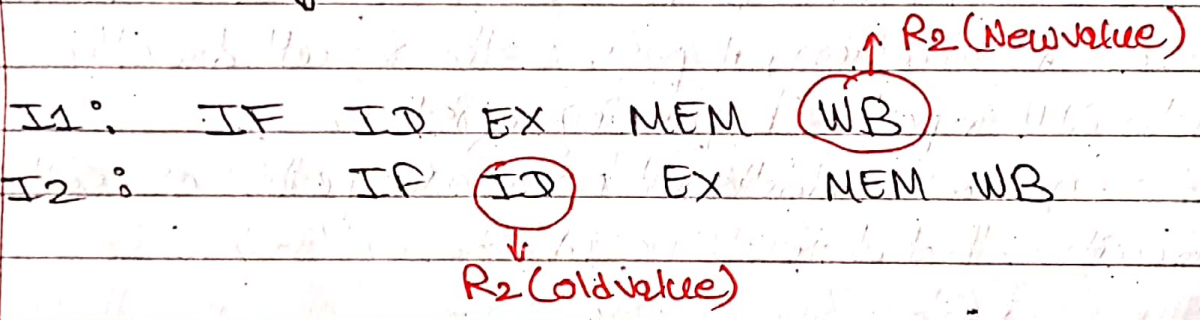
ii) Data Hazard :

- When an instruction depends on the result of a previous instruction; but this result is not yet available.
- Multiple Instruction using same resources.
- Data hazard types : ① RAW (Read After Write)
  ② WAR (Write After Read)
  ③ WAW (Write After Write)

- For example : ① RAW

I1 : R_2 (W) $= R_2 + R_3$

I2 : $R_5 = R_2$ (R) $+ R_4$



We would compute the wrong answer because I_2 reads the old value of R_2.

② WAR :

$a = b + c$ (b → old value)

$b = c + d$ → (b → new value)

③ WAW :

$a = b + c$

$a = c + d$

Page No.
Date : / /

*) Solution to Data Hazards:

- Forwarding or Bypassing,
- Instruction Scheduling.

i) Forwarding or Bypassing:

- use special hardware to detect a conflict and then avoid it by routing the data through the special paths between the pipeline segments.
- example: instead of transferring ALU result into a destination register, the hardware checks the destination, if it is needed as a source in the next instruction, it passes the result directly into ALU input - bypassing register.
- It requires additional hardware paths as well as circuits that detects conflicts.

ii) Instruction Scheduling:

- Compiler attempts to schedule code to fill delay slots.

iii) Structural Hazard:

- If hardware cannot support the combination of instructions that we want to execute in the same clock cycle, Structural Hazard occurs.
- A Structural Hazard occurs when a part of the resources is needed by two or more instructions at a time.
- Solution to Structural Hazard can be resource duplication.

Page No.
Date : / /

- for example:

I1 :	IF	ID	EX	MEM	WB			
I2 :		IF	ID	EX	MEM	WB		
I3 :			IF	ID	EX	MEM	WB	
I4 :				IF	ID	EX	MEM	WB

* Multiple Issue Processor:

- A processor that can sustain more than one instruction per cycle.
- Different hardware and software techniques have been used to utilize ILP (Instruction Level Parallelism)
- CPI = 1 (cycle per instruction = 1) i.e one instruction per cycle.
- If we want to reduce CPI < 1, then we have to look out issuing multiple instruction.
- i.e.; If two instruction per cycle, CPI = $\frac{1}{2}$ = 0.5 < 1
- Issuing N instructions per clock cycle reduces the Ideal - CPI term to 1/N.
- If M is the maximum numbers of instructions that can be issued in one cycle, then the processor is M-wide.

* Types of multiple issue processor:

Two variation : ① Superscalar processor.
② ~~the~~ VLIW processor.

Page No.
Date : / /

① SuperScalar processor:

- It refers to the machine that is designed to improve the performance of the execution of Scalar Instruction.
- Issues Variable number of Instructions per clock.
- i.e. a SuperScalar processor is a CPU that performs a form of parallelism.

e.g:-

I1:	IF	ID	EX	MEM	WB	
I2:		IF	ID	EX	MEM	WB
I3:			IF	ID	EX	MEM WB

② VLIW processor:

- It issues a single very long instruction per clock that contains a large number of real instructions.
- The Compiler or programs control the parallel execution of the instructions.
- This increases compiler complexity but decreases hardware Complexity.

* Processor Parallesim:

- Creating powerful computers simply by connecting many existing smaller ones is called processor parallelism.
- Multiple processors working cooperatively on problems.

Goals:

- Performance, (increases comparison to uniprocessor)
- Cost Efficiency, (build big system with small parts)

- Scalability, (Just add more processors to get more performance)
- Fault tolerance. (one processor fails you still can continue processing).

Types:

i) Synchronous processor parallelism:

- This strategy replicates whole processors, with each processor executing the same program on different portions of the data space.

ii) ASynchronous Processor Parallelism:

- This strategy replicates whole processors, but allows each processor to execute different programs or different parts of the same program.
- on these machines, if synchronization between processors is required, it must be explicitly specified.

#2 Semantic Analysis:

* Syntax Directed Translation:

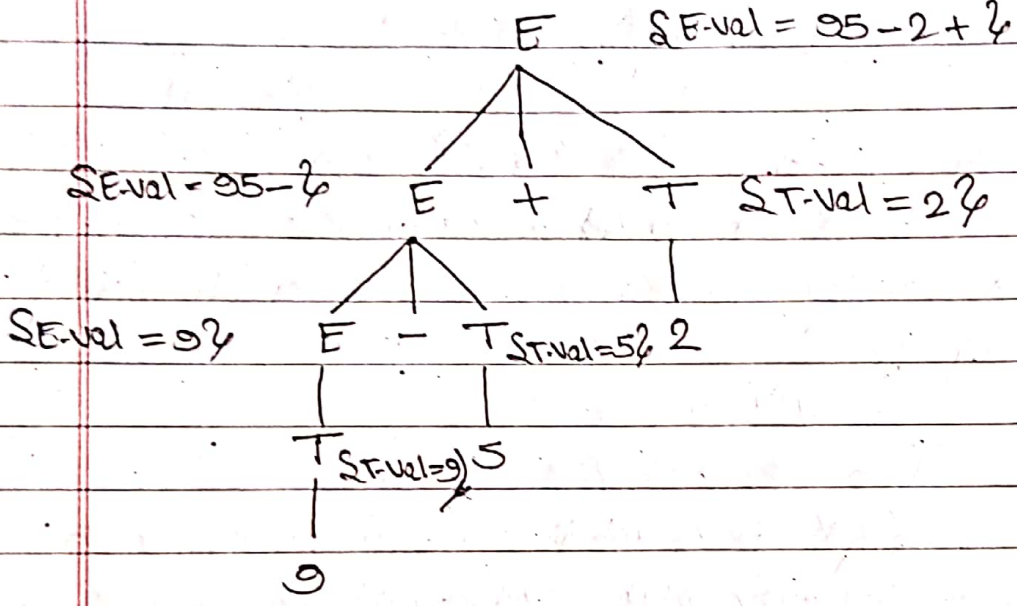
- Defining the semantics action along with production rules.

example:

Production rule:	Semantic Action
$E \rightarrow E + T$	E.val = E.val || T.val || "+"
$E \rightarrow E - T$	E.val = E.val || T.val || "-"
$E \rightarrow T$	E.val = T.val
$T \rightarrow 0$	T.val = "0"
$T \rightarrow 1$	T.val = "1"
:	:
$T \rightarrow 9$	T.val = "9"

Page No.
Date : / /

fore example: 9-5+2



Page No.
Date : / /

Unit : 2
Dependence Analysis and Testing

* Dependence and it's properties :

- For a given Sequential program, a collection of statement-to-statement execution ordering that can be used as a guide to select and apply transformations; that preserve the meaning of the program.
- Each pair of statements in the graphs is called a dependence.
- Dependency represents two kinds of constraints
  1. Data dependence,
  2. Control dependence.

① Data Dependence :

- Ensure that data is produced and consumed in correct order.

  eg:- S1 : PI = 3.14
  S2 : R = 5.0
  S3 : Area = PI * R * R

- S3 cannot be moved before either S1 or S2 without compromising correct results.
- To prevent this we will construct data dependencies from S1 & S2 to S3.
- No execution constraint between S1 & S2 because {S1, S2, S3} or {S2, S1, S3} gives same result.

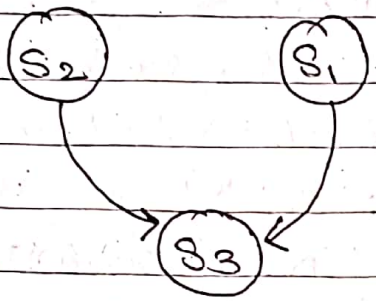


Fig:- Dependency in graph

example ②:

S1: a = 3.14
S2: b = 5.0
S3: C = a*b*b
S4: y = a*C
S5: Z = x

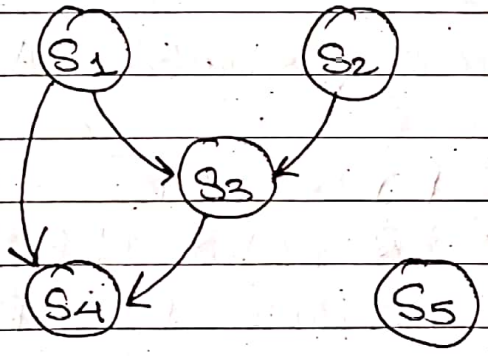


Fig of :- Dependency in graph

② Control Dependence:

- Dependence due to control flow,
- e.g:- S1: IF (T == 0.0) GoTo S3
  S2: A = A/T
  S3: CONTINUE
- here, S2 cannot be executed before S1.

- executing S_2 before S_1 could cause divide by zero.

* Load-Store classification:

i) True dependency:

- The first statement stores into a location & that is later read by the second statement.
- e.g:-

S_1 : x = ...

S_2 : .... = x

- denoted by S_1 & S_2.
- The dependence ensures that the second statement receives the value computed by the first. It is also known as a Flow dependence.

ii) Anti-dependence:

- The first statement reads from a location into which the second statement later stores.
- for e.g: S_1 : .... = x

S_2 : x = ...

- S_2 depends on S_1 is denoted by $S_1 \delta^{-1} S_2$

iii) Output Dependence: (WAW hazard)

- Both statements write into the same location.
- e.g:- S_1 : x = ...

S_2 : x = ...

eg2: S_1 : $x = 1$

S_2 : ....

S_3 : $x = 2$

S_4 : $w = x + y$

- S_2 depends on S_1 is denoted by $S_1 \delta^{o} S_2$

Page No.
Date : / /

* e.g:- Find the dependency between S_1, S_2 and S_3

```
for (i=1; i<=n; i++)
    for (j=1; j<=n; j++)
S1:     x[i,j] = A[i,j] + Y[i,j]
S2:     y[i,j] = C[i,j] + X[i,j]
S3:     X[i,j] = Y[i,j] + D[i,j]
```

S_1, S_2 = True and Anti dependence.
S_2, S_3 = True and Anti dependence
S_1, S_3 = Output dependence.

* Dependence in Loops:

- Let us look at two different loops.

```
DO I = 1, N
  S1 :  A(I+1) = A(I) + B(I)
END DO
```

Fig(1)

```
DO I = 1, N
S1: A(I+2) = A(I) + B(I)
END DO
```

Fig (2)

- In both cases, statement S1 depends on itself
- However, there is a significant difference: In fig(1), depend on previous statement & in fig(2), depend on two previous statements.

Page No.
Date : / . /

* Iteration Number:

For an arbitrary loop in which the loop index I runs from L to U in steps of S, the iteration number i on a specific iteration is equal to the value $\frac{(I-L+1)}{S}$, where I is the value of the index on that iteration.

e.g:- DO I = 0, 10, 2 → (L, U, S)
S1 <Some statement>
END DO

i.e for I = 5, $i = \frac{(I-L+1)}{S}$

$= \frac{(5-0+1)}{2}$

$= 3$

* Iteration Vector:

- Given, a nest of n loops, the iteration vector i of a particular iteration of the innermost loop is a vector of integers that contains the iteration numbers for each of the loops in order of nesting level.
- The iteration vector for $i = \{i_1, i_2, \ldots, i_n\}$ where i_k, $1 \le k \le n$ represents the iteration number for the loop at nesting level k.

e.g:- DO I = 1, 2
DO J = 1, 2
S1 <Some statements>
END DO
END DO

Note: The iteration vector S1 [(2,1)], denotes the instance of S1 executed during the 2nd iteration of I loop & 1st iteration of J loop.

Page No.
Date : / /

* Iteration Space:

- The set of all possible iteration vectors for a statement.
- e.g:
```
DO I = 1, 2
    DO J = 1, 2
        S1 <Some statements>
    END DO
END DO
```
- here, the iteration space for S1 is
  {(1,1), (1,2), (2,1), (2,2)}

- An iteration vector i preceeds another iteration vector j, iff any statement executed on the iteration described by i is executed before any statement on the iteration described by j

* Formal definition of loop dependence:

- There exists a dependence from statement S_1 to statement S_2 in a common nest of loops if and only if there exists two iteration vectors i and j for the nest, such that
  i) $i < j$ or $i = j$ and there is a path from S_1 to S_2 in the body of the loop,
  ii) Statement S_1 accesses memory location M on iteration i and statement S_2 accesses location M on iteration j, and
  iii) one of these accesses is write.

Page No.
Date : / /

* Transformations:

- A Transformation is Safe if the transformed program has the same meaning as the original program.
- Two Computations are equivalent if, on the same inputs, they produce the same outputs in the same order.

* Reordering Transformation:

- A transformation that changes the order of execution of the code, without adding or deleting any execution of any statement.
- A reordering transformation preserves a dependence if it preserves the relative execution order of the source and sink of that dependence.

* Theorem of Dependence:

Statement:

- Any reordering transmission that preserves every dependence in a program preserve the meaning of that program.

Proof (By Contradiction):

- Assume the program be loop free and control free.
- Let $S S_1, S_2, \ldots, S_n$ be the execution order in the original program.
- $i_1, i_2, \ldots, i_n$ be the permutation of the statement.

- Assume that dependence is preserved but the meaning changes.
- i.e Some output statement produces a different result from the corresponding output statement in the original program.
- Let, Sk be the first statement in the new order that produces an incorrect output.
- because the statement is exactly the same as the statement in the original program, it must have found an incorrect value in one of its input memory locations.
- Since all the statements that have executed before Sk produce the same value they did in the original program, there are only three ways that Sk can see incorrect input in a specific location m.

Case i:

A statement Sm that originally stored its result into M before it was read by Sk, now, stores into M after it is read by Sk (i.e, Sk reads old value).

i.e., reordering fail to preserve true dependence (Sm δ Sk)

Case ii:

A statement Sm that originally stored into M after it was read by Sk, now writes M before Sk reads it (i.e. Sk reads new value).

i.e., reordering fail to preserve anti-dependence ($S_m \delta^{-1} S_K$)

Case III:

Two ~~depende~~ statements that both wrote into M before it is read by S_K, causing the wrong value to be left in M.

i.e, reordering failed to preserve output dependence

Hence, it exhausts the way that S_K can get the wrong value, the result is proved by contradiction.

* Distance Vector:

Suppose that there is a dependence from statement S_1 on iteration i of a loop to statement S_2 on iteration j; then the distance vector d(i,j) is defined as the vector of length n such that $d(i,j)_K = j_K - i_K$

* Direction Vector:

Suppose that there is a dependence from statement S_1 on iteration i to statement S_2 on iteration j, then the direction vector D(i,j) is defined as,

$$D(i,j)_K = \begin{cases} \text{"<"} & \text{if } d(i,j)_K > 0 \\ \text{"="} & \text{if } d(i,j)_K = 0 \\ \text{">"} & \text{if } d(i,j)_K < 0 \end{cases}$$

Page No.
Date : / /

for eg:-

```
for (i=1; i<=100; i++)
{
  for (j=1; j<=100; j++)
  {
    for (k=1; k<=100; k++)
    {
      A[i+1, j, k] = A[i, j, k+1] + B
    }
  }
}
```

A[i+1, j, k] : Sink
A[i, j, k+1] : source

Let, the iteration vector be ⟨3,8,5⟩

① distance vector = ⟨3+1, 8, 5⟩ − ⟨3, 8, 5+1⟩
(sink − source) = ⟨4, 8, 5⟩ − ⟨3, 8, 6⟩
= ⟨1, 0, −1⟩

② direction vector = ⟨<, =, >⟩

Note: For correct meaning & dependence, first place is always <.

* Eg-2: Compute distance and direction vector at iteration ⟨1,1,1⟩ for following program.

```
DO I = 1, N
  DO J = 1, M
    DO K = 1, L
      A(I+1, J, K) = A(I, J, K)
    END DO
  END DO
END DO
```

Page No.
Date : / /

① Distance vector = SINK − SOURCE
= (1+1, 1, 1) − (1, 1, 1)
= (2, 1, 1) − (1, 1, 1)
= (1, 0, 0)

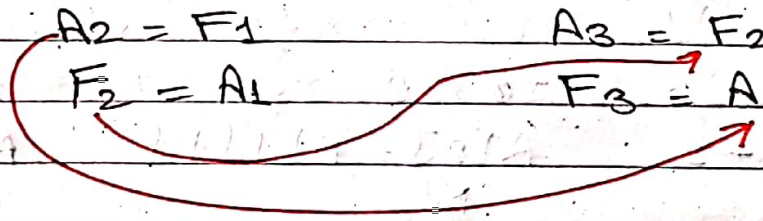
② Direction vector = (<, =, =)

* Loop Carried Dependence:

- Dependence exists across iterations i.e. if the loop is removed, the dependence no longer exists.

e.g:-
```
DO I = 1, N
  S1 : A(I+1) = F(I)
  S2 : F(I+1) = A(I)
END DO
```

i.e. I = 1 → A2 = F1, F2 = A1
I2 = 2 → A3 = F2, F3 = A2



* Loop independent Dependence:

- Dependence exists within an iteration
- i.e. if the loop is removed, the dependence still exists.

e.g:-
```
DO I = 1, 10
  S1 : A(I) = ....
  S2 : .... = A(I)
END DO
```

Page No.
Date : / /

* Find the type of dependence in following loop:

eg:-

```
DO I = 1,9
S1:  A(I) = ---
S2:  --- = A(10-I)
END DO.
```

On the fifth iteration of loop, statement S1 stores into A(5) while statement S_2 fetches from A(5). Therefore, the dependence is loop-independent.

eg:-

```
for (i = 1; i<n; i++)
{
S1 : A[i] = A[i-1] + 1;  (S1 is loop carried depe
S2 : B[i] = A[i];  (S2 is loop independent)
}
```

i.e, for S1:

I = 1 : A[1] = A[0] + 1

I = 2 : A[2] = A[1] + 1

I = 3 : A[3] = A[2] + 1

i.e. loop-carried dependent.

For S_2:

B[i] = A[i]

i = 1

B[1] = A[1]

i.e, the value of B[i] is directly dependent on value of A[1]. Therefore, it is loop independent.

Page No.
Date : / /

* Parallelization:

— It is valid to convert a sequential loop to a parallel loop if the loop carries no dependence.

— for e.g:-

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

→

$A_1 = A_1 + A_0$
$A_2 = A_2 + A_1$
$A_3 = A_3 + A_2$
pallel ✗
dependence ✓

eg:2 :

```
A[0] = 1
for (i=1; i<N; i=i+2)
{
    A[i] = A[i] + A[i-1];
}
```

→

A[1] = A[1] + A[0]
A[3] = A[3] + A[2]
A[5] = A[5] + A[4]
parallel ✓
dependence ✗

* Dependence Testing:

— It is a method used to determine whether dependences exist between two subscripted references to the same array in a loop nest.

General problem:

```
Do i1 = L1, U1
  Do i2 = L2, U2
    :
      Do in = Ln, Un
S1 :      A(f1(i1,---,in), ---, fm(i1,---,in)) = ---
S2 :      --- = A(g1(i1,---,in), ---, gm(i1,---,in))
      END DO
  END DO.
```

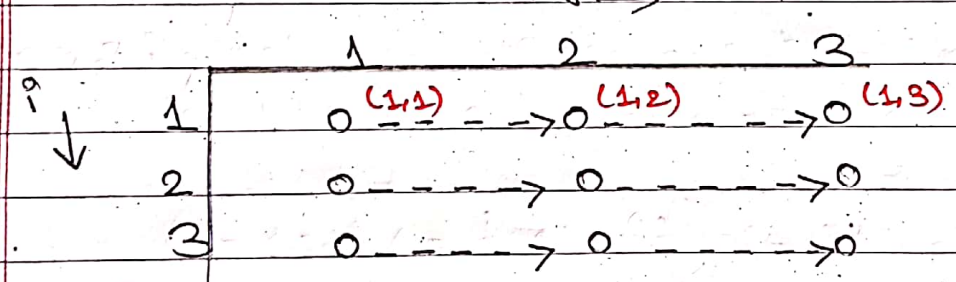
Page No.
Date : / /

here, Dependence exists if
$f_i(\alpha) = g_i(\beta)$ where, α & β are iteration vectors.

* Iteration Space Traversal Graph (ISTG):

- shows graphically, the order of traversal in the iteration space.
- A node represents a point in the iteration space.
- A directed edge indicates the next point that will be encountered after the current point is traversed.
- for e.g:

```
for (i=1; i<n; i++)
    for (j=1; j<4; j++)
S1 :    A[i][j] = A[i][j-1] + 1
```



* GCD Test:

- We can use linear diophantine equation and GCD test to find the dependency on loop.
- Diophantine equation is a polynomial equation, usually involving two or more unknowns, such that the only solutions of interest are the integer ones.
- e.g:- $Ax + By = C$, where A, B, C are integers.
- If GCD (A, B) divides C, then dependency exists

and in that case the loop is not parallelizable.

eg:- Solving diophantine equation.

$$70x + 112y = 168$$

Step 1:

GCD (70, 112) = ?

$$70 \overline{)\,112\,(1}$$
$$-70$$
$$42 \overline{)\,70\,(1}$$
$$-42$$
$$28 \overline{)\,42\,(1}$$
$$-28$$
$$14 \overline{)\,28\,(2}$$
$$-28$$
$$\times$$

Step 2:

Find the value of x & y

Reverse-method:

$$14 = 42 - 28$$
$$= 42 - (70 - 42) \quad (\because 42 = 70 + 42)$$
$$= 2(42) - 70$$
$$= 2(112 - 70) - 70$$
$$14 = 2(112) - 3(70)$$

Multiply both side by 12:

$$168 = 24(112) - 36(70)$$

$\therefore x = 24$

$y = -36.$

Page No.
Date : / /

* eg:- Find the dependency in following loop:

```
DO I = i, N
A(4i) = ....
        x
.... = A(2i+2)
         y
END DO.
```

Sol

$4x = 2y + 2$

or, $4x - 2y = 2$

$GCD(4, -2) = 2$

* For which iteration dependency exists: $x = 2, y = \frac{4*2-2}{2}$

$= 3$

$(x = 2, y = 3)$ dependency exist

Since, 2 divides 2, there exists dependency in the loop, hence the loop is not parallelizable.

* eg 2: Find whether the dependency will exists in following loop or not & find the iteration vector.

```
for (i=1; i<=100; i++)
{
  A(3*i + 20) = ...
  .... = A[2*i]
}
```

Sol: $3x + 20 = 2y$

$3x - 2y = -20$

$GCD(3, -2) = 1$

Let, $x = 2$ then $y = \frac{3 \times 2 + 20}{2} = 13$

iteration vector = $(x = 2, y = 13)$ where dependency exists.

Page No.
Date : / /

* eg:- Find the dependence in following cases:

Case 1:

```
DO I = 1, N
   DO J = 1, N
      A(2I + 2J) = ...
      ... = A(4I - 6J + 3)
   END DO
END DO
```

Sol:- $2a + 2b = 4c - 6d + 3$

$2a + 2b - 4c + 6d = 3$

GCD(2, 2, -4, 6) = 2 & 3 is not divisible by 2 so, No dependency exists.

Case 2: ($S_1 \rightarrow S_1$, $S_2 \rightarrow S_2$)

```
DO I = 1, 3
   DO J = 1, 3
      DO K = 1, 3
S1 :  A(I+1, 2J+2, K+1) = B(I, 2J, K)
S2 :  C(I, J, K) = A(I, 1, 9-K)
   END DO
END DO
END DO.
```

Sol:- For $S_1 \rightarrow S_1$

$a + 1 + 2b + 2 + c + 1 = d + 2e + f$

$a + 1 + 2b + 2 + c + 1 - d - 2e - f = 0$

$a + 2b + c - d - 2e - f + 4 = 0$

$a + 2b + c - d - 2e - f = -4$

Page No.
Date : / /

* Subscript categories:

① ZIV (zero index integer):

- If it contains no loop index variable.

for eg:-
```
DO J = 1, 100
   Si:   A(e1) = A(e2) + B(J)
END DO
```
where, e_1 & e_2 are constant.

② SIV (Single index variable)

- If it contains only one index variable.

```
DO I = 1, N
   A(4i = ....
     .... = A(2i+2)
END DO
```

③ MIV (Multiple index variable)

- If it contains multiple loop index variables.

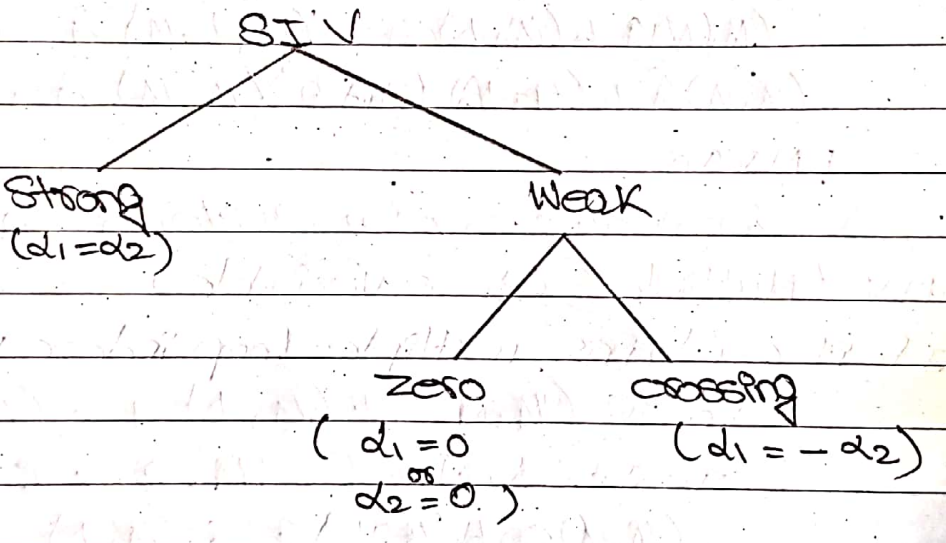
```
DO I = 1, N
   DO J = 1, N
      A(2I+2J) = ....
        .... = A(4I - 6J + 3)
   END DO
END DO
```

* ZIV Test:

- ZIV subscripts contains no references to any loop induction variables.
- They do not vary within any loop.

eg:- DO I = 1, N
A(e1) = A(e2) + B(I)
END DO

here, if $(e_1 - e_2) \neq 0$, then there is no dependence



* Assume the following statements

S1 : A(---, $a_1 I + c_1$, ---) = ---

S2 : --- = A(---, $a_2 I + c_2$, ---)

Case i:

($a_1 = = a_2$ i.e Strong SIV Test)

Case ii:

($a_1 = = -a_2$) i.e Weak Crossing SIV Test.

Page No.
Date : / /

① Strong SIV Test:

— $a = \frac{c_2 - c_1}{\alpha}$

— If $a \in Z$ and $|a| \leq U - L$ then dependency exists.

for eg:

```
DO I = 1, N
  S1 : A(I + 2*N) = A(I+N) + C
END DO
```

i.e. Let ($\alpha_1 = 1$ & $\alpha_2 = 1$) & compare above eqn with

$$A(\ldots, \alpha_1 I + C_1, \ldots) = \ldots$$
$$\ldots = A(\ldots, \alpha_2 I + C_2, \ldots)$$

we get,

$$C_1 = 2N \quad \& \quad C_2 = N$$

then,

$$a = \frac{C_2 - C_1}{\alpha} = \frac{N - 2N}{1} = -N \in Z \quad \text{True}$$

Also,

$$|a| \leq U - L \quad (\text{where, } U = N \text{ \& } L = 1)$$
$$|a| \leq N - 1$$
$$N \leq N - 1 \quad \text{False,}$$

For dependency to exists both $a \in Z$ & $|a| \leq U - L$ needs to be satisfied but in our case $|a| \leq U - L$ doesn't satisfied so, No dependency exists.

Page No.
Date : / /

* Weak Zero SIV Test:

- $a = \frac{c_2 - c_1}{\alpha}$, if a is integer then depence exists.

e.g:
```
Do I = 1, N
    A(I, N) = A(1, N) + A(N, N)
END DO
```

i.e.

~~I = 1, A(1, N) = A(1, N) + A(N, N)~~
I = 2, A(2, N) = A(1, N) + A(N, N)
⋮
I = N-1, A(N-1, N) = A(1, N) + A(N, N)
~~I = N, A(N, N) = A(1, N) + A(N, N)~~

Dependency exists so, we need to remove dependency by:

```
A(1, N) = A(1, N) + A(N, N)
Do I = 2, N
    A(1, N) = A(1, N) + A(N, N)
END DO
A(N, N) = A(1, N) + A(N, N)
```

This program will run iteration I = 1 & I = N outside the loop which removes the dependency so, parallelization is possible.

Page No.
Date : / /

* Weak crossing SIV test:

- $a = \frac{c_2 - c_1}{\alpha}$ if $2a$ is integer then dependence exist.

- e.g:-
```
DO I = 1, N
   S1 : A(I) = A(N-I+1) + C
END DO
```

i.e.

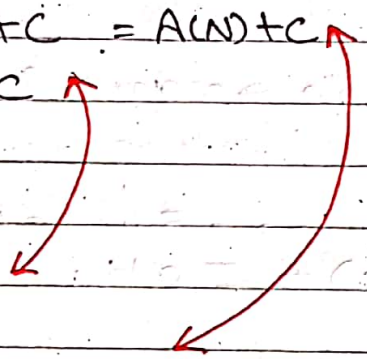
I_1 : $A(1) = A(N-1+1) + C = A(N) + C$

I_2 : $A(2) = A(N-1) + C$

⋮

$I = N-1$: $A(N-1) = A_2 + C$

$I = N$: $A(N) = A_1 + C$



Dependency exists, so we need to remove dependency by:

```
DO I = 1, (N+1)/2
   A(I) = A(N-I+1) + C
END DO
DO I = (N+1)/2, N
   A(I) = A(N-I+1) + C
END DO
```

This program will divide the programs in two half which removes the dependency so, parallelization is possible.

Page No.
Date : / /

Unit : 3

Preliminary Transformations

(This chapter permits methods of perliminary Transformations to make them fit for dependence Testing)

* **Loop overheads:**

① Initialize counter
② Increment / decrement } Software
③ Test and branch
④ FSM (Finite state Machine)
⑤ Condition check } Hardware
⑥ Extra cycles for R/W Memory

* **Loop Induction variables:**

- An Induction variable is a variable that gets increased or decreased by a fixed amount on every iteration of a loop, or is a linear function of another induction variable.
- The loop index is trivially an induction variables
- for e.g:-

```
      J = 0
      DO I = 1, N
        J = J + 1
        K = 2 * J
        M = 3 * K
        A = B + C
      END DO
```

Note: I, J, K, M are Induction variable i.e there values changes linearly. where, I is loop Induction variable.

↑ value doesn't changes so, it isn't induction variable.

Page No.
Date : / /

* Induction Variable optimization:

```
for (i=0; i<N; i++)
  {
    j = 4*i+3;
    y = f(j);
  }
```

⇒ i = 0; j = 3;
i = 1; j = 7; (+4)
i = 2; j = 11
⋮
i = N; j = 4*N+3;

Dead optimization

```
for (i=0; i<N; i++)
  {
    for (j=3; j<4*N+3; j=j+4)
      {
        y = f(j);
      }
  }
```

* Preliminary Transformation:

- Most dependence test requires subscript expression to be linear function of loop induction variables.
- But programs are not typically written with dependence testing in mind.
- A number of transformations can be applied prior to dependence testing with the goal of making testing more accurate. Known as preliminary transformation.

Advantages:

- Higher dependence test accuracy,
- Easier Implementation of dependence test.

e.g:
```
INC = 2
KI = 0                                  [∵ KI = 0]
DO I = 1, 100
    DO J = 1, 100
        KI = KI + INC                   [2]   [∵ KI = 2]
        U(KI) = U(KI) + W(J)            <- 2, 4, 6 ... 10
    END DO
    S(I) = U(KI)
END DO
```

- Only two subscripts W(J) and S(I) are linear functions of the loop induction variables.
- The expression U(KI) cannot be tested in the form written because KI varies within the loop.
- INC is invariant within the inner loop.
- Assignment to KI within J loop increment its value by a constant amount on each loop iteration.
- KI ⇒ Auxiliary induction variable.

i.e. The above code must be transformed to make testing more accurate.

* Induction Variable Substitution:

- Transforms every reference to an auxiliary induction variable into a direct function of Loop index.

Step 1: Replace references to auxiliary induction variables with function of loop index.

```
INC = 2
KI = 0
DO I = 1, 100
   DO J = 1, 100
      KI = KI + INC
      U(KI + J*INC) = U(KI + J*INC) + W(I)
END DO
KI = KI + 100*INC
S(I) = U(KI)
END DO
```

Step 2: Remove all references to KI

```
INC = 2
KI = 0
DO I = 1, 100
   DO J = 1, 100
U(KI + (I-1)*100*INC + J*INC) =
   U(KI + (I-1)*100*INC + J*INC) + W(I)
END DO
KI = KI + 100*INC
S(I) = U(KI + I*(100*INC))
END DO
KI = KI + 100*100*INC
```

Page No.
Date : / /

Step 3: Substitute the constants:

```
INC = 2
KI = 0
DO I = 1, 100
  DO J = 1, 100
  U(I*200 + J*2 - 200) =
  U(I*200 + J*2 - 200) + W(J)
END DO
  S(I) = U(I*200)
END DO
  KI = 20000
```

Step 4: Remove all unused code

```
DO I = 1, 100
  DO J = 1, 100
  U(I*200 + J*2 - 200) =
  U(I*200 + J*2 - 200) + W(J)
  END DO
S(I) = U(I*200)
END DO.
```

Page No.
Date : / /

* Loop Normalization:

- To make the dependence testing process as simple as possible, many compilers normalize all loops to run from a lower bound of 1 to some upper bound with a step of 1. i.e.

~~(for (i=1;~~

for (i = 1; i < N; i++) ✓

For (i = 3; i < N; i = i + 2) ✗

- Replace references to the original loop, induction variables with linear function of the new induction variables.
- This transformation is called loop Normalization.

* Algorithm:

Procedure normalized loop (La) // Lo is the loop to be normalized.

- Let i be a unique compiler-generated loop Induction variable.

S1: Replace the loop header for Lo

Do I = L, U, S

with adjusted loop header

Do $\overset{\circ}{I}$ = 1, (U-L+S)/S

S2: replace each reference to I within the loop by

i * S - S + L;

END Normalized loop

Page No.
Date : / /

eg:- Do I = 1, M
  Do J = I, N → (L=I, U=N, S=1)
  A(J, I) = A(J, I-1) + 5
  END DO → (J*S-S+L)
END DO [J*1-1+I → J-1+I]

⇓ Normalized

Let J be the loop to be normalized i.e.

Do I = 1, M
  Do J = 1, (N-I+1) [∵ i = 1, (U-L+S)/S]
  A(J-1+I, I) = A(J-1+I, I-1) + 5
  END DO [∵ J = j*S-S+L]
END Do.

Also, we need to check the meaning of the program before & After Normalization i.e.

① Before Normalization:

I = 1, J = 1, 2, 3
A(1,1), A(2,1), A(3,1)

I = 2, J = 2, 3
A(2,2), A(3,2)

② After Normalization:

I = 1, J = 1, 2, 3
A(1,1), A(2,1), A(3,1)
I = 2, J = 1, 2
A(2,2), A(3,2).

Page No.
Date : / /

e.g: DO I = 7 (L), max (U), 3 (S)
A(I) = B(I) + 5
END DO.

$i * 3 - 3 + 7 \Rightarrow 3i + 4$

⇓ Normalized.

DO I = 1, (max − 7 + 3)/3 [∵ i=1, (U−L+S)/S]
A(3i+4) = B(3i+4) + 5 [∵ i = i*S − S + L]
END DO.

① Before Normalization:

I = 7
A7 = B7 + 5
A10 = B10 + 5
A13 = B13 + 5

② After Normalization:

I = 1,
A(3×1+4) = B(3×1+4) + 5
⇒ A7 = B7 + 5
I = 2,
A(3×2+4) = B(3×2+4) + 5
⇒ A10 = B10 + 5

Page No.
Date : / /

Data Flow Analysis:

It is a Analysis that determines the information regarding to the definition and use of data in a program.

for eg:-

d1 : x = a+b

* denotes the definition of x using (a,b).

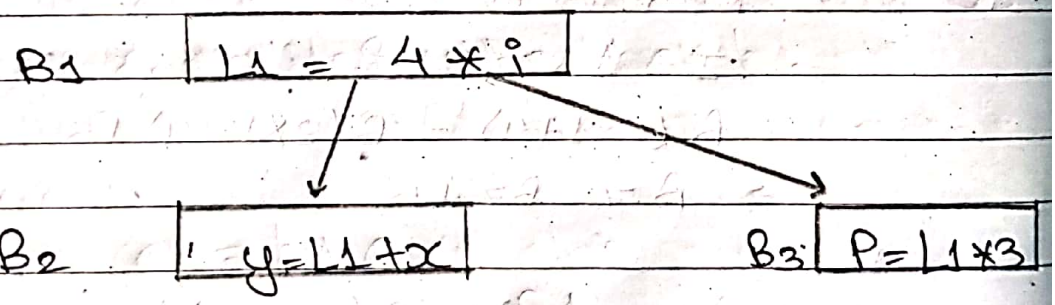
Data Flow properties:

① Available expression
② Reaching Definition

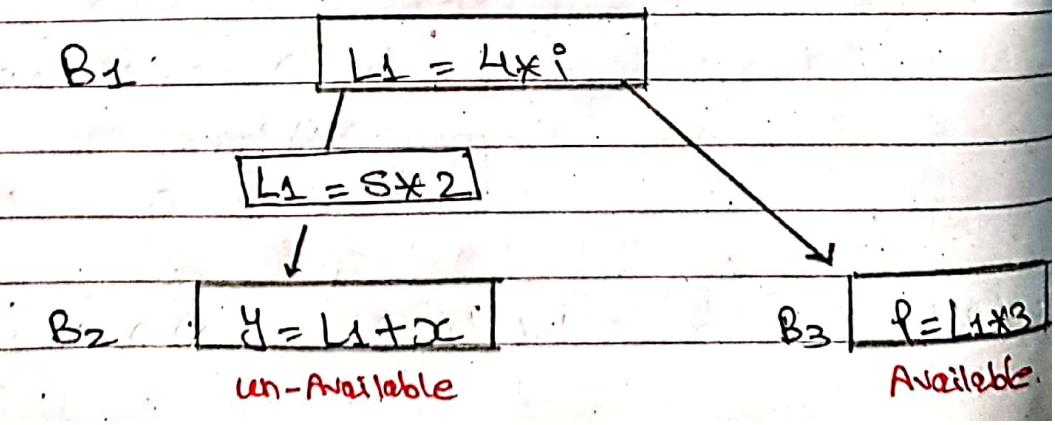
Available expression:

An expression "a+b" is said to be available at program point "x", if none of its operands gets undefined before their use.

for eg:

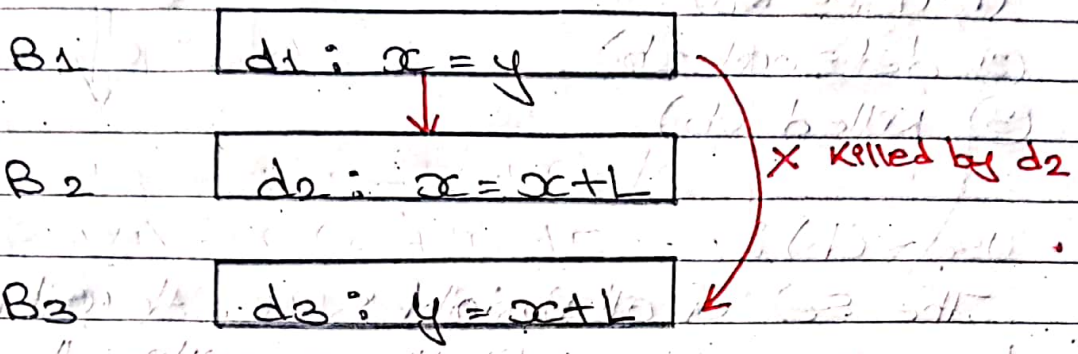


L1 is Available expression for both B2 & B3

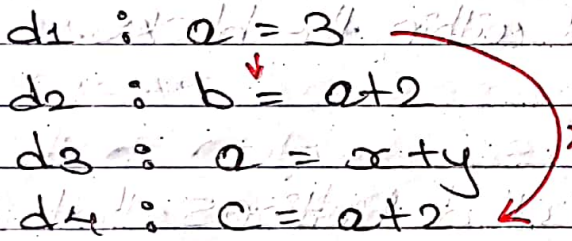


② Reaching Definition:

- A definition D is reaching to a point x if D is not killed or redefined before that point.
- For e.g:-



- d1 is reaching definition for B2 but not for B3, it is killed by d2 in B2.
- For e.g:-



- In above Example, d1 is reaching definition for d2 but not for d4, it is killed by d3.

* Definition-use graph:

- It is a graph that contains an edge from each definition point in the program to every possible use of the variable at run time.
- Provides the map of variables usage,
- Heavily used by the transformations.

Page No.
Date : / /

* Definition-use graph sets (or definition-use chain)

- Basic block computation produces the following sets.

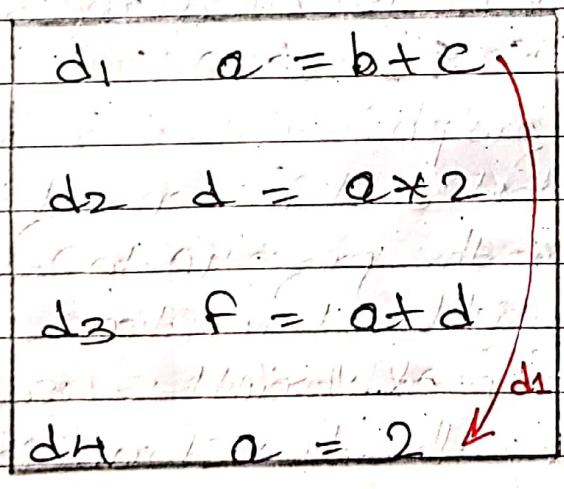
① Uses (b)
② defsout (b)
③ Killed (b)

- **Uses (b):**
  The set of all variables used within block b that have no prior definitions within the block.

- **defsout (b):**
  The set of all definitions within blocks b that are not killed within the blocks.

- **Killed (b):**
  The set of all definitions that defines variables killed by other definitions within block b.

- for eg:-

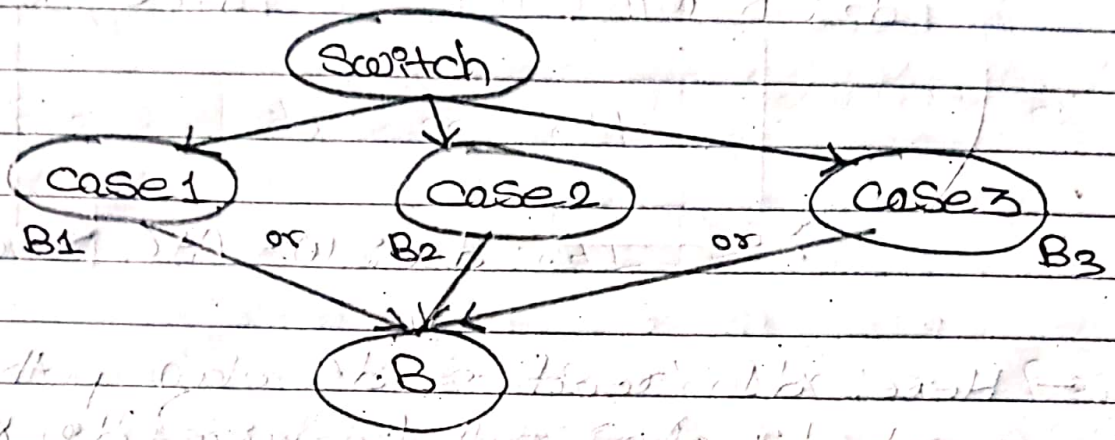


Uses (B1) : {b, c}, defsout(B1) = {d, f}, Killed(B1) = {a}

Page No.
Date : / /

* Blocks:

- Basic block is a maximal group of statements such that one statement in the group is executed if & only if every statements is executed.
- e.g:-

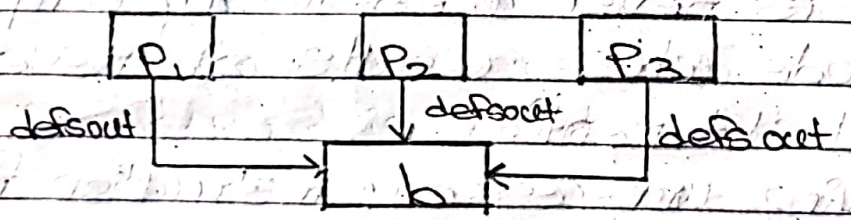


* Reaches (b):

- Let, us assume a graph with one basic block b, and some number of predecessors, each of which can reach b, through some form of control flow.

$$p(b) = \{P_1, P_2, P_3\}$$

i.e,

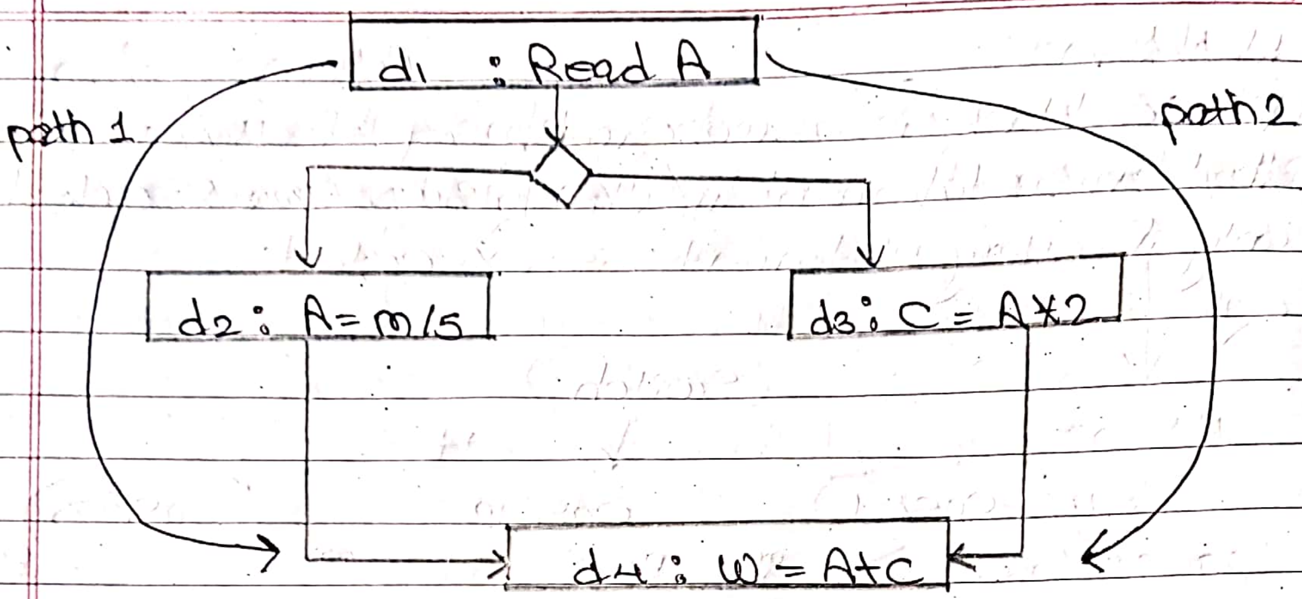


- In this graph, reaches (b) along any one predecessor p is the set of all definitions that reach p (reaches (p)) and not killed inside p ($\neg$ killed (p)) plus those definitions in P that reach the exit of P (defsout(p))

i.e.

$$reaches(b) = \bigcup_{P \in p(b)} (defsout(p) \cup (reaches(p) \cap \neg killed(p))$$

P_1 P_2 P_3



- Here, d1 reaches d4 along path 2 but doesnot reach d4 along path 1. Since, d2 Kelled the definition of d1.

* Dead Code Elimination:

- Dead code is a Section in the Sosrce code of the program which is executed but whose resceft is never used in any computations.
- i.e. Code whose rescelts are never used in any 'Usefeel statement'.
- Removing unnecessary instruction from the program.
- For eg:-

c = a*b
t = a → Dead code
d = a*b + 4c

→ Dead code Elimination →

c = a*b
d = a*b + 4c

Page No.
Date : / /

* Algorithm for Dead Code Elimination:

```
Procedure eliminate DeadCode (P);
- let worklist = § absolutely useful statement §;
while worklist ≠ ϕ do begin
        x = an arbitrary element of worklist;
        mark x useful;
        worklist = worklist − §x§;
    for all (y, x) ∈ defuse do
    if y is not marked useful then
            worklist = worklist ∪ §y§;
end
    delete every statement that is not marked
    useful;
end eliminate dead code.
```

* Constant Propagation:

- Initially, discover values that are constant on all possible execution, & propagate values i.e replace all variables that have constant values at runtime with those constant values.
- It is process of substituting the constant values in the place of all variables that holds them.

Steps:

① Use of variables replaced by a constant value called propagation.

② expressions evaluated at compile time ~~need not be~~ don't need to evaluated at execution time. i.e. If such expression are inside loops, a single evaluation at compile time can save many evaluations at execution time.

Page No.
Date : / /

- eg:- x = 14
y = 7 - x/2
return y(28/x+2)

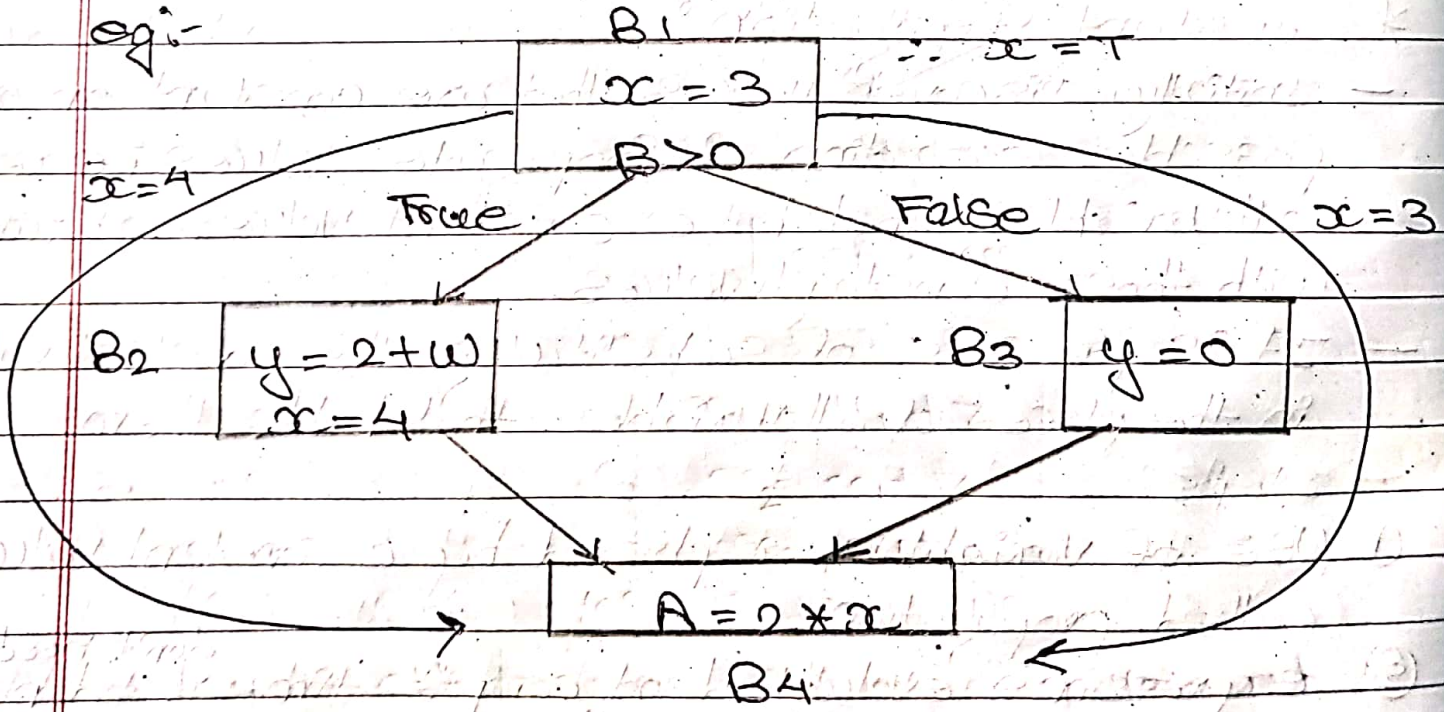
x = 14
y = 7 - 14/2
return y(28/14+2)

x = 14
y = 0
return 0;

* Symbols used:

① ⊥ (Bottom) → This Statement never executes,
② C → Constant.
③ T (Top) → not a constant (i.e x can be any value)

eg:-



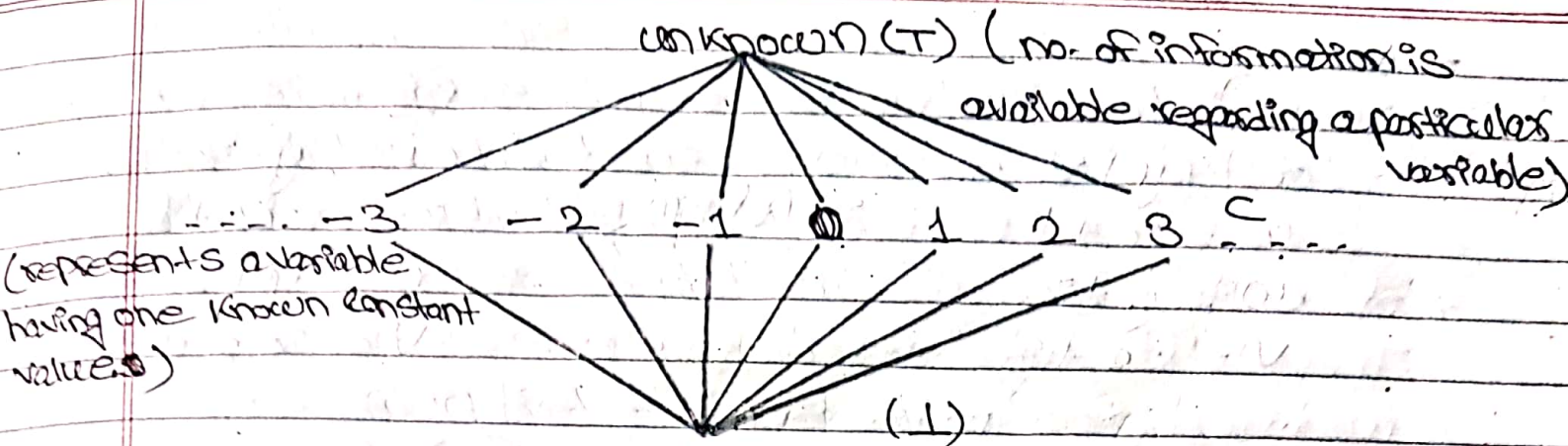


Fig of :- Lattice for constant propagation.

* Meet operations:

1) Any constant $\leq$ Unknown
2) NAC $\leq$ Any constant
3) Unknown $\wedge x = x$
4) NAC $\wedge x$ = NAC
5) NAC $\wedge$ Unknown = NAC
6) $x \wedge x = x$ by $x \wedge y$ = NAC

if $x \neq y$

* SSA (Single Static Assignment) Form:

- A program is in SSA form, if each use of variable is reached by exactly one definition.
- Optimization become simpler if each variable has only one definition.
- i.e. every variable has single definition.

Page No.
Date : / /

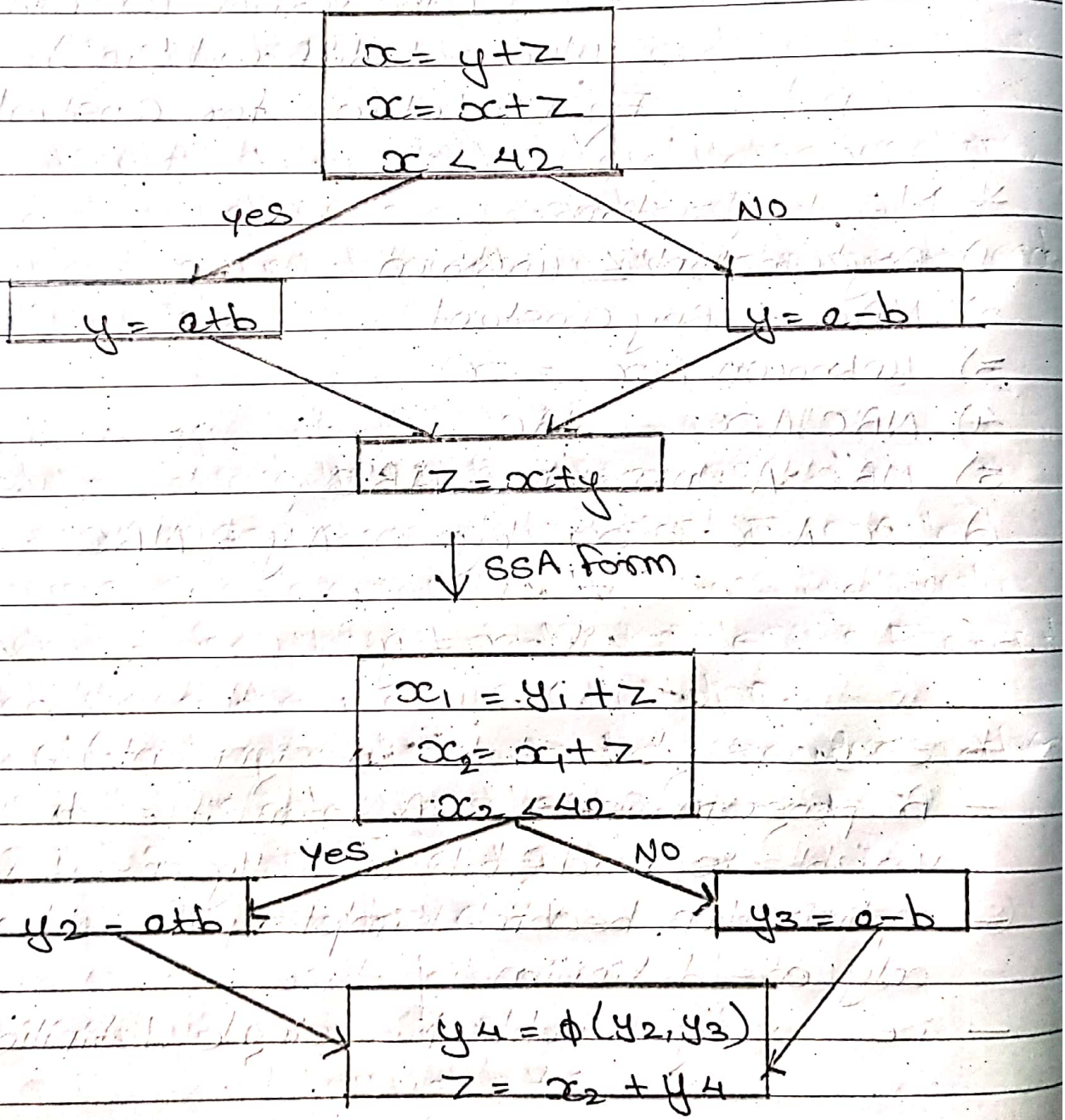
- for eg:

$x = y + z$
$x = x + 1$
$w = y + z$
$v = x + 3$

SSA form →

$x_1 = y + z$
$x_2 = x_1 + 1$
$w = y + z$
$v = x_2 + 3$

eg:-



Page No.
Date : / /

The ϕ Function:

It is a Merge function

ϕ merges multiple definitions along multiple control paths into a single definition.

At a basic block with P predecessors, there are P arguments to the ϕ functions,

i.e, $x_{new} = \phi(x_1, x_2, \ldots, x_p)$

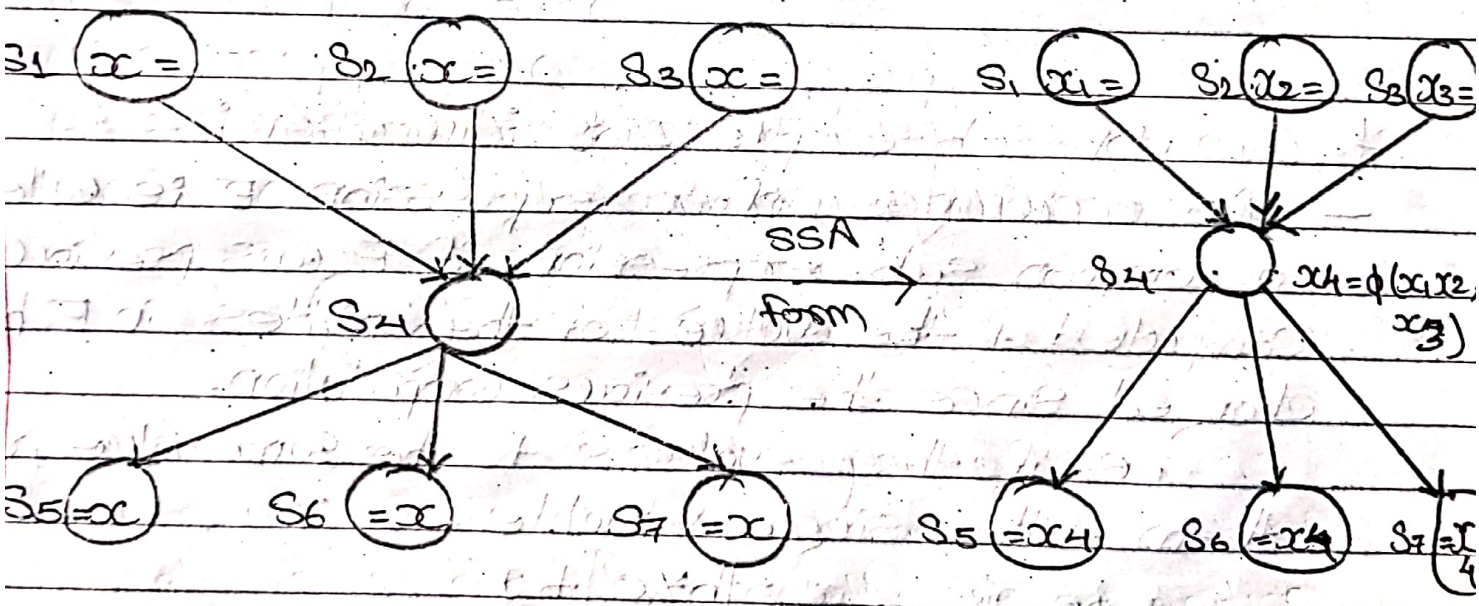
ϕ function has the semantics of copying the correct definition depending on which path they are reached by the execution flow.

SSA (Single Static Assignment):

reduces the number of definition-use edges.

improves performance of algorithm.

for eg:-



Statements S1, S2, & S3 all define the variable x. These definitions all reach the uses in statements S5, S6, S7 by passing through Statement S4. There are nine edges: (S1, S5), (S1, S6), (S1, S7), (S2, S5), (S2, S6),

(S_2, S_7), (S_3, S_5), (S_3, S_6), (S_3, S_7)

— To reduce this number of operations is to put a special pseudo operation in the node for statement S_4: $x = x$. Because this definition kills the values of x that are created in statements S_1, S_2 & S_3. The total number of definition-use edges in the modified program is six:
(S_1, S_4), (S_2, S_4), (S_3, S_4), (S_4, S_5), (S_4, S_6) & (S_4, S_7)

* Construction of SSA typically proceeds in two major points

i) Identification of the points where merge functions, called ϕ-functions are needed &

ii) Variable renaming to create a unique name for each definition point.

* Common Sub-expression Elimination:

— An occurance of an expression E is called a common sub-expression, if E was previously computed & the values of the variables in E haven't changed since the previous computation.

i.e. if they evaluate to the same value, replace them with a single variable.

For eg:- $a = b*c + g$

$d = b*c * k$

↓

$temp = b*c$

Page No.
Date : / /

a = temp + g
d = temp * K

eg:-

```
t6 = 4*i                    t6 = 4*i
x = a[t6]                   x = a[t6]
t7 = 4*j  -> temp           t7 = temp
t8 = 4*j        ------>     t8 = temp
t9 = a[t8]                  t9 = a[temp]
a[t7] = t9                  a[temp] = t9
t10 = 4*j -> temp           t10 = temp
a[t10] = x                  a[temp] = x
goto B2                     goto B2
```

* Forward Substitution:

- It is the reverse of common sub expression Elimination.
- It may seem that common sub-expression reduces the ALU operations.
- But still it might not necessary Advantageous, since it causes a register to be occupied for a long time to hold an expressions value.
- For eg:-

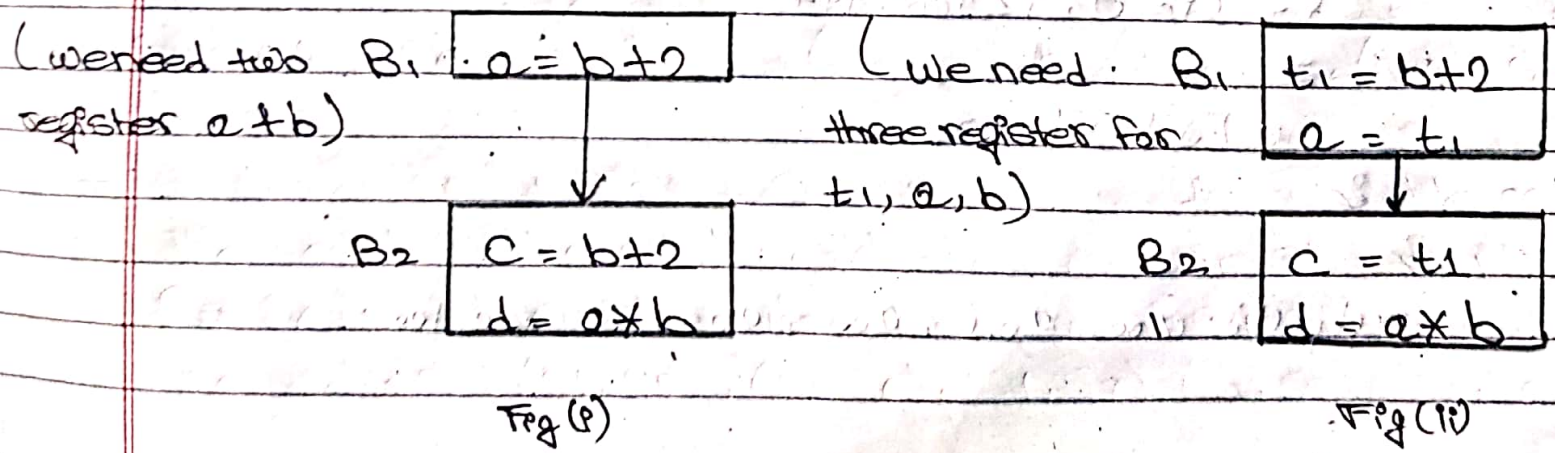


Fig (i) Fig (ii)

Page No.
Date : / /

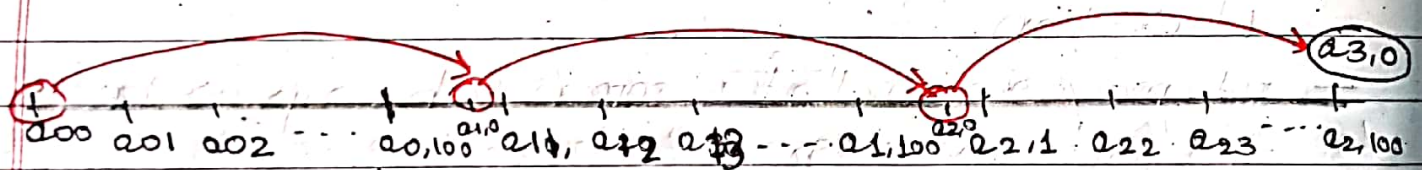
Unit 4:
Loop optimization

* Loop optimization (Interchange):

- The process of switching the nesting order of two loops. (exchange inner loop with outer loop).
- for eg:

```
for (i = 0; i < 100; i++)
  {
    for (j = 0; j < 100; j++)
      {
        a[j][i] = 0;
      }
  }
```

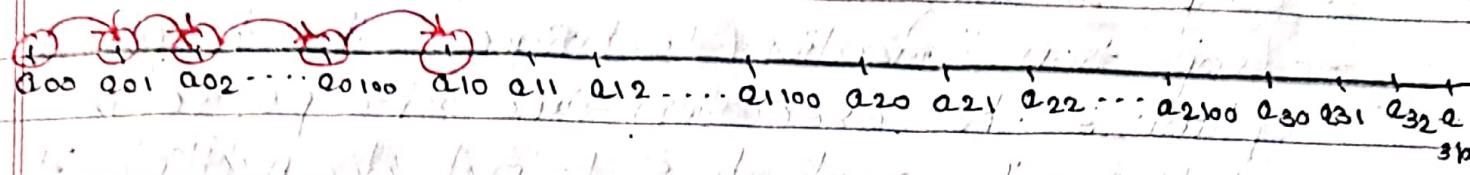
output: $a_{00}, a_{10}, a_{20}, \ldots$



↓ Loop Interchange

```
for (j = 0; j < 100; j++)
  {
    for (i = 0; i < 100; i++)
      {
        a[j][i] = 0;
      }
  }
```

output: $a_{00}, a_{0,1}, a_{02}, a_{03}, \ldots a_{0,100}$



- but not all loop interchange are safe.

for eg:-
```
DO J = 1, M
   DO I = 1, N
      A(I, J+1) = A(I+1, J) + B
   END DO
END DO
```

Result:

J=1, I=1 $A_{12} = A_{21} + B$

I=2 $A_{22} = A_{31} + B$ (True dependence)

J=2, I=1 $A_{13} = A_{22} + B$

I=2 $A_{23} = A_{32} + B$

Loop Interchange.

```
DO I = 1, N
   DO J = 1, M
      A(I, J+1) = A(I+1, J) + B
   END DO
END DO
```

✗ Loop interchange is not possible

Result:

I=1, J=1 $A_{12} = A_{21} + B$

J=2 $A_{13} = A_{22} + B$ (Anti-dependence)

I=2, J=1 $A_{22} = A_{31} + B$

I=2, J=2 $A_{23} = A_{32} + B$

Page No.
Date : / /

* Loop unrolling:

- Loop transformation techniques that helps to optimize the execution time of a program.
- It basically replicates a loop body multiple times, adjusting loop terminal code.
- e.g:-

```
Sum = 0                                   runs 100th times
for (i = 0; i < 100; i++)
{
    Sum = Sum + 10
}
```

Loop unrolling →

```
Sum = 0                                   runs 95th time
for (i = 0; i < 100; i+=5)
{
    p1  Sum = Sum + 10;
    p2  Sum = Sum + 10;
    p3  Sum = Sum + 10;
    p4  Sum = Sum + 10;
    p5  Sum = Sum + 10;
}
```

processors: p1, p2, p3, p4, p5

* Loop invariant: (change नहुने variable)

- A boolean statement that is true at the start of the loop and at the end of each iteration.

i.e.

for e.g:-

```
for (i = 0; i < 100; i++)
{
    x = i + (a/b);
    y = y + 2;
}
```

→

```
n = a/b
for (i = 0; i < 100; i++)
{
    x = i + n;
    y = y + 2;
}
```

Page No.
Date : / /

* Loop Fusion :

- Adjacent Loops are merged to a single loop.
- for e.g :-

```
for (i=0; i<100; i++)          #same loop
{
    x[i] = 1;
}
for (i=0; i<100; i++)          #same loop
{
    y[i] = 2;
}
```

+ Fusion →

```
for(i=0; i<100; i++)
{
    x[i] = 1;
    y[i] = 2;
}
```

Loop Fusion

* Fusion safety :

- A loop ~~independence~~ independent dependence between Statement S1 & S2 in loop L1 & L2 respectively is fusion preventing if fusing L1 & L2 causes the dependence to be carried by the combined loop.
- for e.g :-

```
Do I = 1, N
    S1 : A(I) = B(I) + C
END DO
Do I = 1, N
    S2 : D(I) = A(I+1) + E
END DO
```

↓ Loop Fusion

```
Do I = 1, N
    S1 : A(I) = B(I) + C
    S2 : D(I) = A(I+1) + E
END DO
```

Result:

I_1 : $A_1 = B_1 + C$

$D_1 = A_2 + E$ (Loop independent dependence)

I_2 : $A_2 = B_2 + C$

$D_2 = A_3 + E$

* Loop Vectorization:

- Loop vectorization helps to transform procedural loops by assigning a processing unit to each pair of operands.
- Programs spent most of their time within such loops.
- Therefore, Vectorization can significantly accelerate them, especially over large datasets.
- for eg:

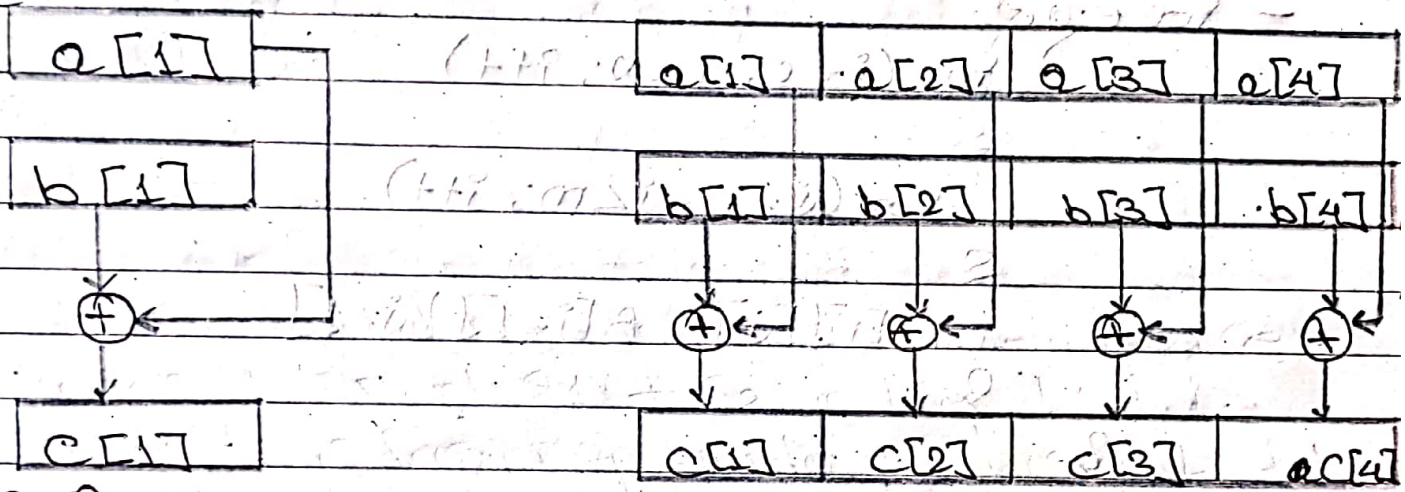
```
for (i = 0; i < 1024; i++)
{
  c[i] = a[i] + b[i];
}
```

Vectorized form

```
for (i = 0; i < 1024; i = i + 4)
{
  c[i:i+3] = a[i:i+3] + b[i:i+3];
}
```

c_1, c_2, c_3, c_4 a_1, a_2, a_3, a_4 b_1, b_2, b_3, b_4

tere, C[i:i+3] sepresents the four array elements [i], c[i+1], c[i+2], c[i+3].
i.e, vector processor can perform four operations for a single vector instruction.
Since, the four vector operations complete in roughly the same time as one scalar instruction, the vector approach can run upto four times faster than the original code.
i.e.



Fig①: Original code

Fig②: Vectorization code

Loop Vectorization attempts to rewrite the loop in order to execute its body using vector instructions Such instructions are commonly referred as SIMD (Single Instruction Multiple Data) where identical operations are performed simultaneously by the hardware.

Eg:-
```
for (i=1; i<=N; i++)
S
  A[i] = 2*B[i]+1;
}
```

vectorized form →

```
for(i=1; i<=N; i=i+4)
S
  A[i:i+3] = 2*B[i:i+3] +
             [1,1,1,1];
}
```

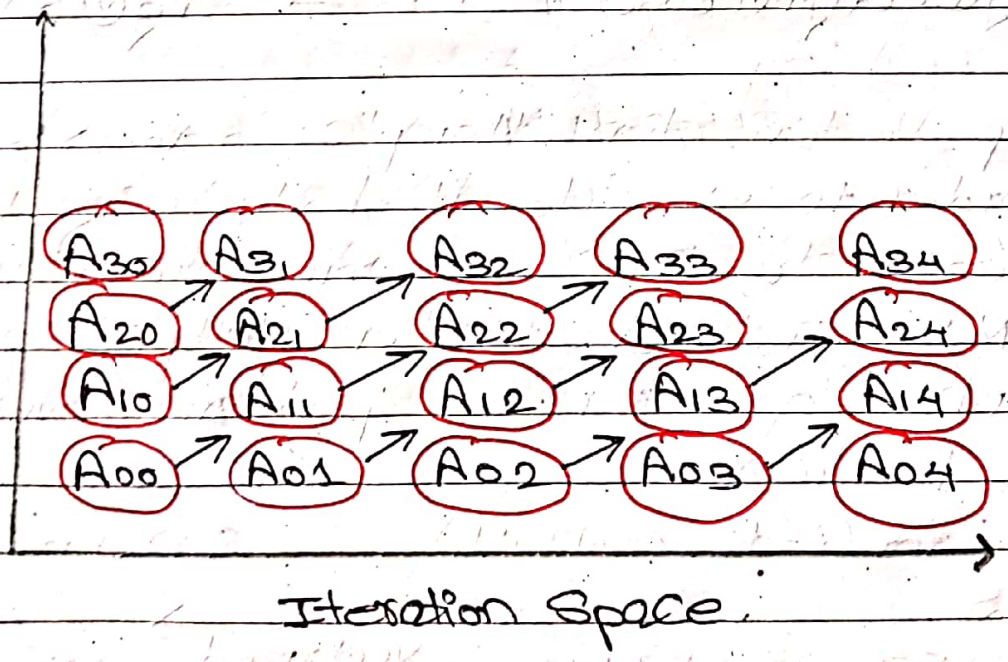
Fig: Loop vectorization scheme of length 4.

Page No.
Date : / /

* Loop Skewing:

- It is a transformation that reshapes an iteration space to make it possible to express to existing parallelism with conventional parallel loops.
- It changes the shape of the iteration space without changing the dependencies.
- It looks like geometrical transformation which can help to expose 4 canonnical parallelism.
- For e.g:-

```
for (i = 0; i < N; i++)
{
  for (j = 0; j < m; j++)
  {
    A[i][j] = A[i-1][j-1]
  }
}
```



Iteration Space.

Page No.
Date : / /

- The dimension for S1 → [0, N] x [0, M]
- The given function can be modeled as:

$$(i, j) = (i'-1, j'-1)$$

$$\Rightarrow p(i, j) = p(i'-1, j'-1)$$

- i.e, $p = i - j$ satisfies the space position ~~conditio~~ constraint.
- now, [-M, N] be the minimum & maximum value that P can take,
- $-M \leq i_{min} - j_{max} = 0 - M$

$$N = i_{max} - j_{min} = N - 0$$

- now, the naive code is generated as:

```
for (p = -M; p <= N; p++)
{
 for (i = 0; i <= N; i++)
 {
  for (j = 0; j <= M; j++)
  {
   if (p == i-j)
   {
    A[i][j] = A[i-1][j-1]
   }
  }
 }
}
```

→ In this loop, there is no data dependency, that's two iterations are completely disjoint.

→ This means, to execute the statement S1, if we are in correct processor.

- now, we have to eliminate the empty iterations.

Page No.
Date : / /

① Tightening the bound for inner most loop (j)

— Since, $p = i - j \Rightarrow j = i - p$

```
for (p = -M; p <= N; p++)
{
  if (i-p >= 0 && i-p <= M)
  {
    ~~A[i][j]~~
    A[i][i-p] = A[i-1][i-p-1]
  }
}
```

② Now, there is still some empty iterations in i loop.

— here,

$i - p \geq 0 \Rightarrow i \geq p$

$i - p \leq M \Rightarrow i \leq M + p$

now,

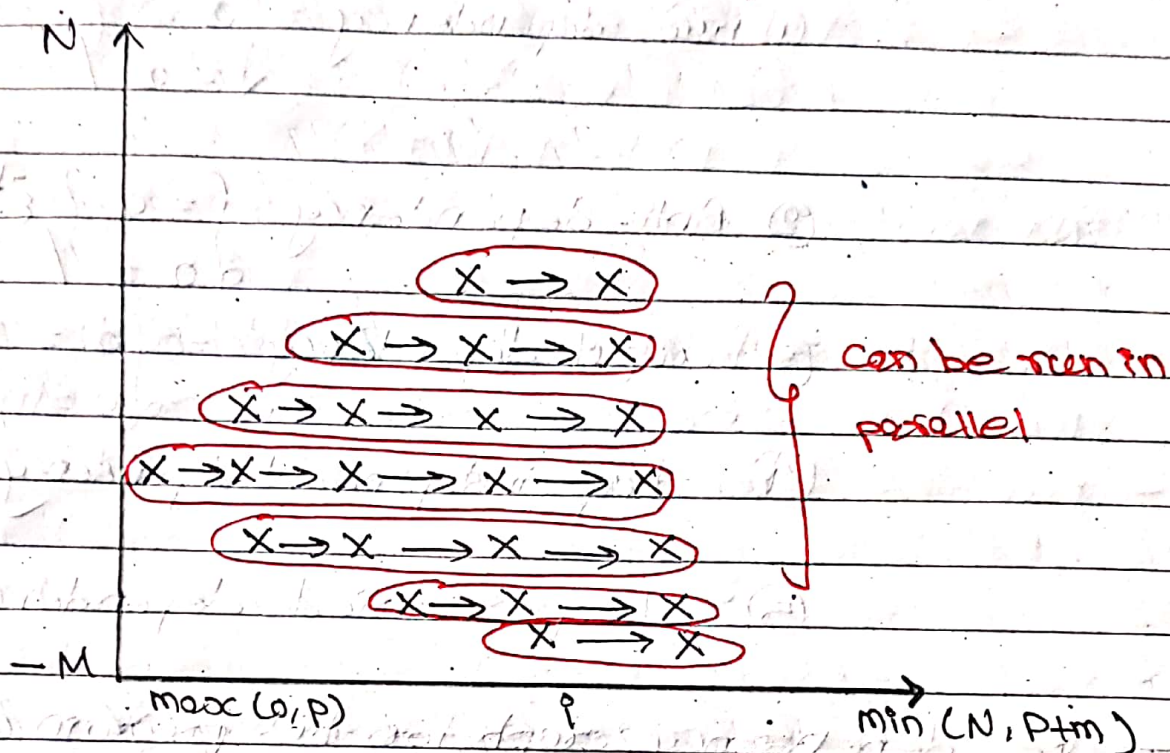
```
for (p = -M; p <= N; p++)
{
  for (i = max(0, p); i <= min(N, p+M); i++)
  {
    A[i][i-p] = A[i-1][i-p-1]
  }
}
```

Previous lower bound

Previous upper bound

no if condition

- how the iteration space looks like:



- Now the outerloop can be parallelized.

* Scalar Expansion:

- The program frequently uses scalar temporaries in computations involving arrays.

e.g:-

```
DO I = 1, N
S1 : T = A(I)
S2 : A(I) = B(I)
S3 : B(I) = T
END DO.
```

Swap the contents of two arrays

Page No.
Date : / /

Note: Important notations :

① True dependency = δ°

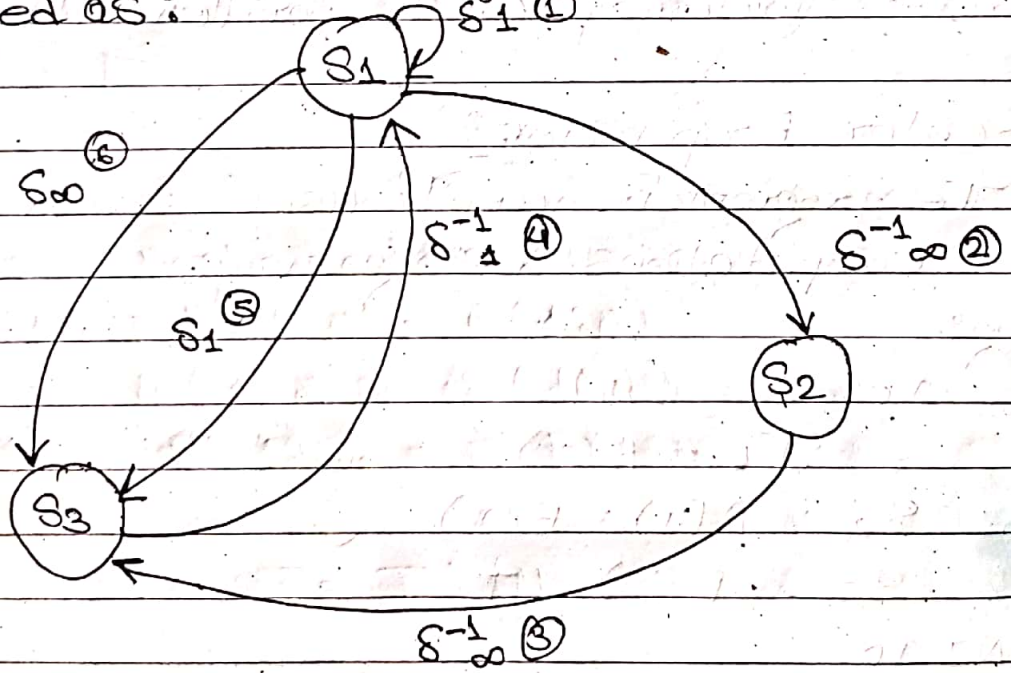
② Anti-dependency = δ^{-1}

③ output dependency = δ°

④ Loop independent dependency = ∞

⑤ Loop carried dependency = 1

- The dependency graph for this program can be explained as :



case i: In first iteration T is defined and in second iteration it is again defined, so there is loop carried dependency (output dependency)

Page No.
Date : / /

Case ii: Loop independent dependency due to A(I) also, Anti-dependency.

Case iii: Loop Independent dependency due to B(I) also, (Anti-dependency.

Case iv: T is used in S3 & again defined in next iteration, so loop carried dependency (Anti-dependency)

Case v: S1 of first iteration defined T & S3 of second iteration is using it, so loop carried dependency (True dependency)

Case vi: Loop independent dependency due to T. (True dependency).

- All loop carried dependencies are due to scalar variable T
- So, use vector variable instead of scalar variable.
- i.e., If each iterations has a seperate locations to use as a temporary, these dependency disappears
- So, Scalar transformation is the process of transforming the programs scalar references with the references of a compiler generated temporary array that has a seperate location for each loop iterations.
- now, the code looks like:

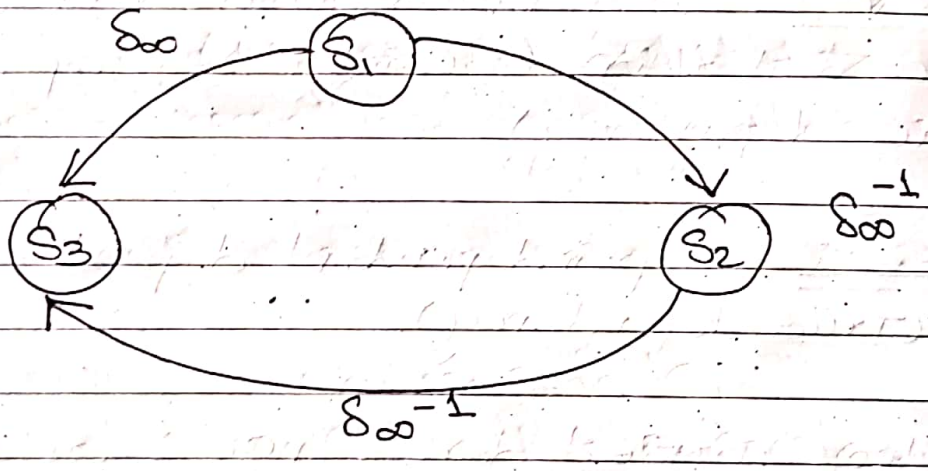
Page No.
Date : / /

```
DO I = 1, N
S1 : T$(I) = A(I)
S2 : A(I) = B(I)
S3 : B(I) = T$(I)
END DO
```

} Replacing T by T$

T = T$(N) : here T is original variable which may have some use after this program.

- now the dependency graphs looks like:



- and can be vectorized as

S1 : T$(1:N) = A(1:N)
S2 : A(1:N) = B(1:N)
S3 : B(1:N) = T$(1:N)

i.e, T = T$(N)

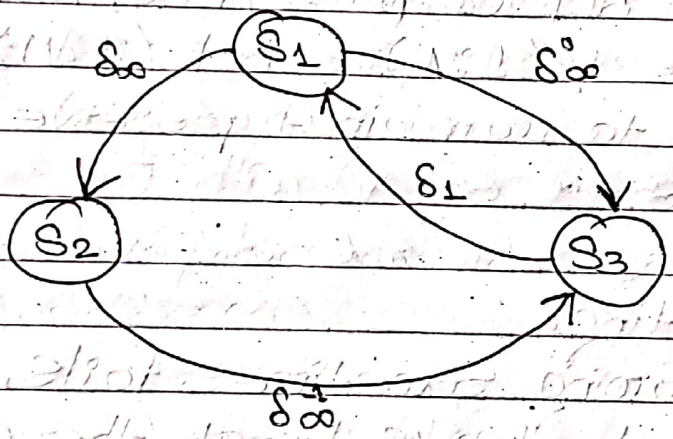
Page No.
Date : / /

* Array Renaming:

- Array locations are sometimes reused, i.e (reading unnecessary anti-dependencies and output dependencies
- e.g:-

```
    Do I = 1, N
S1 :  A(I) = A(I-1) + X
S2 :  Y(I) = A(I) + Z
S3 :  A(I) = B(I) + C
```

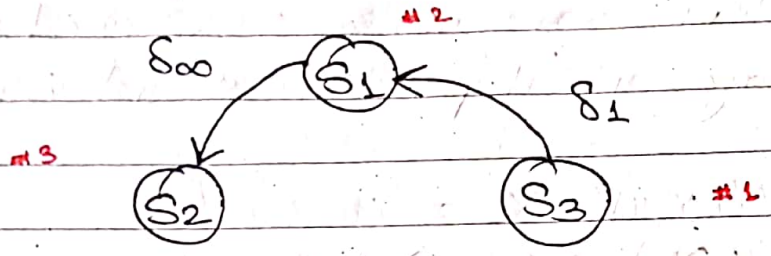
- The dependence graph will be:



- here, there is cycle in dependence graph, so it cannot be vectorized.
- Since A(I) is redefined in S3, it has no relation with A(I) of S2, so break this using array renaming.
- now, after array renaming it becomes

```
    Do I = 1, N
S1 :  A$(I) = A(I-1) + X
S2 :  Y(I) = A$(I) + Z
S3 :  A(I) = B(I) + C
END DO
```

- now, the dependence graph will be,



- Since, there is no cycle, so we can vectorize this as,

S_3 : $A(1:N) = B(1:N) + C$

S_1 : $A\$(1:N) = A(0, N-1) + X$

S_2 : $Y(1:N) = A\$(1:N) + Z$

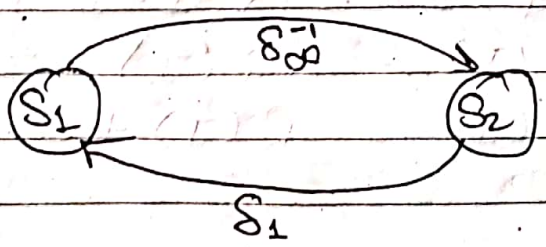
- Reordering to maintain & preserve their related dependences.

* Node Splitting:

- Array renaming sometimes fails,
- It may not always break the cycle.
- eg:

```
DO I = 1, N
S1 : A(I) = x(I+1) + x(I)
S2 : x(I+1) = B(I) + 32
END DO.
```

- The dependence graph looks like:



- which creates a cycle, so cannot be vectorized.

Page No.
Date : / /

- This cycle cannot be broken using array renaming because we cannot break the iterations in different parts.
- In such case node splitting can be used.
- The main target of node splitting is also to break the cycle.
- i.e. breaking anti-dependence.
- makes copy of node from which anti-dependence emanates. (source of Anti-dependence).
- here,

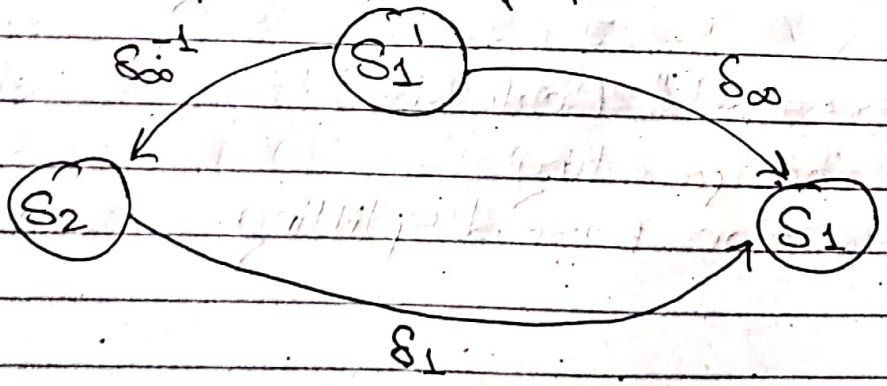
$$A(I) = X(I+1) + X(I)$$

is the part that is the source of anti-dependence.

- So, split the node S1
- now,

```
      DO I = 1, N
S1' : X$(I) = X(I+1)
S1  : A(I) = X$(I) + X(I)
S2  : X(I+1) = B(I) + B2
END DO.
```

- now, the dependence graph will be,



- here, there is no cycle, even though one dependence has increased
- This can be vectorized as:

S_1' : X$ (1:N) = X (2:N+1)
S_2 : X(2:N+1) = B(1:N) + 32
S_1 : A(1:N) = X$ (1:N) + X(1:N)

* Node-Splitting Algorithm:

1) Takes a loop independent anti-dependence D.
2) Add new assignment x : T$ = Source(D).
3) Insert x before Source(D)
4) Replace Source(D) with T$
5) make changes in the dependence graph.

* Index-Set Splitting:

- It sub-divides the loop into different iteration ranges.
- Tries partial parallelization.
- i.e. Iterations not involving in dependence can be parallelized.
- Common Index-Set splitting transformations are:
  i) Threshold Analysis
  ii) Loop peeling
  iii) Section Based Splitting.

- e.g: for (i=0; i<N; i++)
  {
  Ai = Bi + CX → Not involve in dependence.
  Xi = Yi + D } True dependence
  Zi = Xi + E
  }

↓ Index-Set splitting

Ai = Bi + CX → A[1:N] = B(1:N) + CX

for (i=0; i<N; i++)
{
Xi = Yi + D
Zi = Xi + E
}

i) Threshold Analysis:

- Threshold is the number of iterations between the source and sink of the dependence.
- i.e., threshold = distance between source & sink.
- Threshold = 5 means, if source is in 1st iteration then sink will be on 6th iterations.

e.g:

```
DO I = 1, 20
  A(I+20) = A(I) + B
END DO
```

→

I = 1, A21 = A1 + B
I = 2, A22 = A2 + B
I = 3, A23 = A3 + B
:
I = 20, A40 = A20 + B
I = 21, A41 = A21 + B

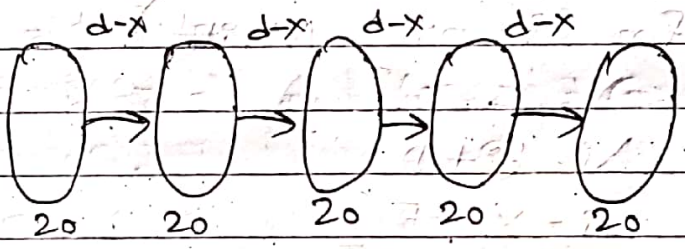
dependence occurs after 20 iterations.

- i.e., the value defined in 1st iteration is used in 21st iteration, 2nd in 22nd iteration, 3rd in 23rd iteration & so on.
- but here the maximum value of I is 20, so sink is never executed here,
- So, this loop has no dependencies.
- all iterations are independent and can be vectorized as

$$A(21:40) = A(1:20) + B$$

i.e,

I = 5



- now, let us consider another example:

e.g:- DO I = 1, 100

A(I+20) = A(I) + B

END DO

Threshold = 20

- now, we have dependence among iterations.
- So, we can divide 100 iterations into chunk of 20 iterations each (called Strip mining)

```
DO I = 1, 100, 20
    DO J = I, I + 19
        A(J+20) = A(J) + B   → Parallel ✓
    END DO
END DO
```

- now, inner loop can be vectorized but outer loop has to execute sequentially.
- So, it can be vectorized as

```
DO I = 1, 100, 20
A(I+20 : I+39) = A(I : I+19) + B
END DO
END DO
```

- eg:-

```
DO I = 1, 100
A(100-I) = A(I) + B
END DO
```

→ I = 1, A99 = A1 + B
I = 2, A98 = A2 + B
⋮
I = 49 A51 = A49 + B
I = 50 A50 = A50 + B
I = 51 A49 = A51 + B

```
Threshold = 50
DO I = 1, 100
A(99:199) = A(1:N) + B
END DO
```

ii) Loop peeling:

- used when source of dependence is in a single iteration,
- eg:-

```
DO I = 1, N
A(I) = A(I) + A(1)
END DO
```

A1 = A1 + A1
A2 = A2 + A1
A3 = A3 + A1

- In every iterations from 2 to N, we are using A1, calculated in 1st iteration.
- So, peel off that operation out of the loop.

```
A(1) = A(1) + A(1)
DO I = 2, N
A(I) = A(I) + A(1)
END DO.
```

A2 = A2 + A1
A3 = A3 + A1

Page No.
Date : / /

- now, can be vectorized as?

$$A(1) = A(1) + A(1)$$
$$A(2:N) = A(2:N) + A(1)$$

iii) Section based splitting:

- Peel off a group of statement,
- a variation of loop peeling.

eg:

```
Do I = 1, N
  Do J = 1, N/2
  S1 : B(J, I) = A(J, I) + C
END DO
```

S_{00}

```
Do J = 1, N
  S2 : A(J, I+1) = B(J, I) + D
END DO
```

divide into two
1 - N/2
N/2+1 - N

```
END Do.
```

- here, S_1 has dependence to S_2 because of B, so outerloop cannot be vectorized because of B
- Since, inner loops have different upperbounds i.e, N, N/2, all statements of second innerloop are not dependent.
- i.e, only N/2 statements of second innerloop are dependent.
- So, partition second inner loop that uses result of S1 & that does not.

```
DO I = 1, N
  DO J = 1, N/2
  S1 : B(J,I) = A(J, I) + C
END DO
  DO J = 1, N/2
    S2 : A(J, I+1) = B(J, I) + D
  END DO
  DO J = N/2 + 1, N
    S3 : A(J, I+1) + B(J, I) + D
  END DO
END DO
```

- now, these is three inner loops & S_3 is dependence free.
- This permits the vectorization of S_3 in two dimensions, while the remainder will vectorize only in J-Dimension.
- now, we get

```
      DO I = 1, N
      DO J = N/2 + 1, N
        S3 : A(J, I+1) = B(J, I) + D
      END DO
    END DO
      DO I = 1, N
        DO J = 1, N/2
          S1 : B(J, I) = A(J, I) + C
        END DO
      DO J = 1, N/2
      S2 : A(J, I+1) = B(J, I) + D
    END DO
END DO
```

- now, we can vectorized as:

S_3: A(N/2+1:N, 2:N+1) = B(N/2+1:N, 1:N)+D

```
DO I = 1, N
  S1 : B(1:N/2, I) = A(1:N/2, I) + C
  S2 : A(1:N/2, I+1) = B(1:N/2, I) + D
END DO.
```

* Alignment:

- Many loop carried dependencies are due to array alignment issues.
- If we can align all references, then dependences would go away, & parallelism is possible.

eg:-

```
DO I = 2, N
  S1 : A(I) = B(I) + C(I)
  S2 : B(I) = A(I-1) * 2.0
```

δ_1

There is loop carried dependence, so we cannot parallelize.

- The two statements can be aligned to compute and use the values in the same iteration by adding an extra iteration and adjusting the indices of one of the statement to produce.

Page No.
Date : / /

```
DO I = 1, N
  S1 : IF (I > 1) A(I) = B(I) + C(I)
  S2 : IF (I < N) D(I+1) = A(I) * 2.0
END DO
```

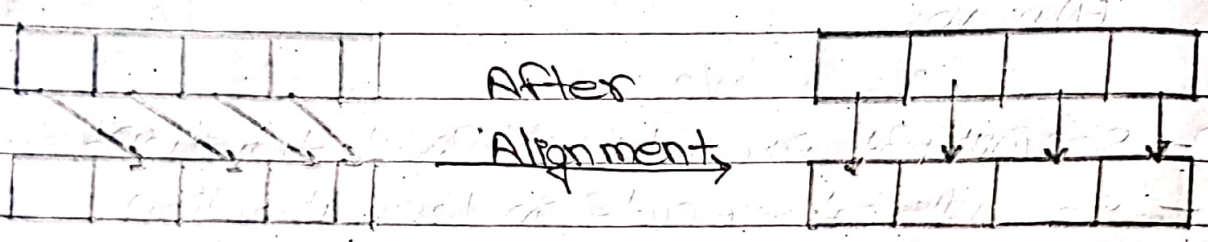
- S1 will not execute in first iteration,
- S2 will not execute in last iteration.

previous Code	Aligned Code
I = 2	I = 1
A2 = B2 + C2	D2 = A1 * 2.0
D2 = A1 * 2.0	
I = 3	I = 2
A3 = B3 + C3	A2 = B2 + C2
D3 = A2 * 2.0	D3 = A2 * 2.0
I = 4	I = 3
A4 = B4 + C4	A3 = B3 + C3
D4 = A3 * 2.0	D4 = A3 * 2.0
:	:
I = N	I = N
An = Bn + Cn,	An = Bn + Cn
Dn = An-1 * 2.0	

- So, Comparing these two, we can see that meaning is same but dependency has changed from loop carried dependency to loop independent

dependency.

- now, we can parallize this loop.
- This can be visualized as.



* Code Replication:

- An Alignment conflict occurs if two or more dependences emanating from the same source and entering the same sink.

e.g:-

```
Do I = 1, N
  A(I+1) = B(I) + C
  X(I)   = A(I+1) + A(I)
```

δ_1^{-1}

$\delta\infty$

- After Alignment we get,

```
Do I = 0, N
IF (I != 0) A(I+1) = B(I) + C
IF (I != N) X(I+1) = A(I+2) + A(I+1)
END Do
```

Page No.
Date : / /

previous Code

I=1,
$A_2 = B_1 + C$
$X_1 = A_2 + A_1$

I=2
$A_3 = B_2 + C$
$X_2 = A_3 + A_2$

I=3
$A_4 = B_3 + C$
$X_3 = A_4 + A_3$

Aligned code

I=0,
$X_1 = A_2 + A_1$

I=1,
$A_2 = B_1 + C$
$X_2 = A_3 + A_2$

I=2,
$A_3 = B_2 + C$
$X_3 = A_4 + A_3$

- here, the original loop carried dependence has been eliminated but it has transformed the original loop independence dependent to loop carried dependence.
- i.e, the loop still cannot be run in parallel.
- In such case, we use code replication.
- It involves redundantly replications of some computations.

```
Do I = 1, N
    A(I+1) = B(I) + C
    IF (I == 1)
        t = A(I)
    ELSE
        t = B(I-1) + C
```

Page No.
Date : / /

END IF
X(I) = A(I+1) + t

END DO

Original code

I=1
A2 = B1 + C
X1 = A2 + A1

I=2
A3 = B2 + C
X2 = A3 + A2

Replicated Code

I=1
A2 = A.B1 + C
t = A1
So X1 = A2 + t

I = 2,
A3 = B2 + C
So t = B1 + C
X2 = A3 + t

— i.e., there is no loop carried dependence, so now it can be parallelize.

* Parallel Code Generation and its Problem:

Page No.
Date : / /

Unit 5 : Control Dependence

* Control Dependence :

```
Do 100 I = 1, N
S1 :  IF (A(I-1) > 0.0) GoTo 100
S2 :  A(I) = A(I) + B(I) * C
100 : CONTINUE
```

- here, S2 has not any data dependence, so it can be executed in parallel.
- It can be vectorized as :

```
S2 : A(1:N) = A(1:N) + B(1:N) * C
Do 100 I = 1, N
  S1 : IF (A(I-1) > 0.0) GoTo 100
100 : CONTINUE
```

- But this vectorized code is not equivalent to original code, why?
- In every iteration S2 is not going to be executed as it depends on S1, called Control dependence.

* IF Conversion :

- IF Conversion Eliminate Control dependencies by converting them to data dependencies:
- The idea is to convert statements affected by branches to conditionally execute statements.

- for eg:-

```
DO 100 I = 1, N
S1 :  IF (A(I-1) > 0.0) GOTO 100
S2 :  A(I) = A(I) + B(I) * C
100 CONTINUE
```

- can be converted as

```
DO I = 1, N
  IF (A(I-1) ≤ 0.0)
    A(I) = A(I) + B(I) * C
END DO
```

now, it can be vectorized as:

```
WHERE (A(0:N-1) ≤ 0.0)
    A(1:N) = A(1:N) + B(1:N) * C
```

e.g.2 :

```
DO 100 I = 1, N
S1 : IF (A(I-1) > 0.0) GOTO 100
S2 : A(I) = A(I) + B(I) * C
S3 : B(I) = B(I) + A(I)
100 CONTINUE
```

↓ can be converted as

```
DO I = 1, N
S1 : IF (A(I-1) ≤ 0.0) A(I) = A(I) + B(I) * C
S2 : IF (A(I-1) ≤ 0.0) B(I) = B(I) + A(I)
END DO
```

- It can be vectorized as:

Page No.
Date : / /

```
DO I=1,N
S1 : IF (A(I-1) .LE. 0.0) A(I) = A(I)+B(I)*C
END DO.

S2 : WHERE (A(0:N-1) .LE. 0.0) B(1:N) = B(1:N)+
                                        A(1:N)
```

* Branch classification:

- Control dependencies arises because of branch instruction, because it alters the flow.

① Forward Branch:

- It transfers control to a target that occurs lexically after the branch but at the same level of nesting.

② Backward Branch:

- It transfers control to a statement occurring lexically before the branch but at the same level of nesting.

③ Exit branch:

- It terminates one or more loops by transforming control to a target outside a loop nest.

e.g:- break statement.

Page No.
Date : / /

* Branch Removal (Forward Branches):

Basic Idea:

i) Make pass through the program
ii) Maintain a boolean expression "cc" that represent the condition that must be true for the current expression to be executed.
iii) On encountering a branch, conjoin the controlling expression into "cc".
iv) On encountering a target of branch, its controlling expression is disjoined into "cc".

- e.g:- Removing forward Branches:

```
      DO I = 1, N                          M1
C1    IF (A(I) > 10) GO TO 60
20    A(I) = A(I) + 10
C2    IF (B(I) > 10) GO TO 80              M2
40    B(I) = B(I) + 10
60    A(I) = B(I) + A(I)
80    B(I) = A(I) - 5
      END DO
```

- St. 20 is executed if C1 is false,
- St. 40 is executed if both C1 & C2 are false
- St. 60 can be directly reached by C1, C2 both are false or C1 is true
- St. 80 can reached by falling from 60 or C1 false & C2 is true.
- how, inserting appropriate guards.

Page No.
Date : / /

```
DO I = 1, N
   M1 = A(I) > 10

20 : IF (!M1) A(I) = A(I) + 10
     IF (!M1) M2 = B(I) > 10

40 : IF (!M1 AND !M2) B(I) = B(I) + 10

60 : IF ((!M1 AND !M2) OR (M1)) A(I) = B(I) + A(I)

80 : IF ((!M1 AND !M2) OR (M1) OR (!M1 AND M2))
        B(I) = A(I) - 5
```

- We can simplify it as:

```
DO I = 1, N
   M1 = A(I) > 10

20 : IF (!M1) A(I) = A(I) + 10
     IF (!M1) M2 = B(I) > 10

40 : IF (!M1 AND !M2) B(I) = B(I) + 10

60 : IF (M1 OR !M2) A(I) = B(I) + A(I)

80 : B(I) = A(I) - 5

END DO.
```

- It can be vectorized as: DO I = 1, N
  M1 (1:N) = A (1:N) > 10

20: WHERE (.M1) A (1:N) = A (1:N) + 10
  WHERE (.M1.M2) M2 = B (1:N) > 10

40: WHERE (.M1 AND .M2)
  B (1:N) = B (1:N) + 10

60: WHERE (M1 OR .M2)
  A (1:N) = B (1:N) + A (1:N)

80: B (1:N) = A (1:N) - 5

END DO

Note: i) Backward branch } refer to Bishnu
  ii) Exit branch } Gautam notes.

* Iterative Dependences:

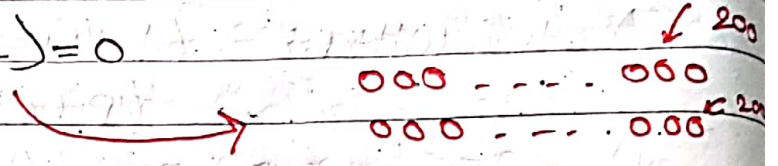
- Iterative statements can also create control dependence.

e.g:-

```
20:   DO I = 1, 100          → irange1()
40:     L = 2 * I
60:     DO J = 1, L          → irange2()
80:       A (I, J) = 0
        END DO
      END DO
```

- If the assumption is made that iterative statements do not carry any control dependences, the example could be incorrectly vectorized as,

```
20 :  DO I = 1, 100
40 :  L = 2 * I
      END DO
      A (1:100, 1:L) = 0
```



- The original example zeroes out a triangular section of the result array.
- The vectorized example zeroes out a rectangular section.
- i.e. because dependencies are missing.
- Their dependencies must capture the notion that a DO statement controls the number of times a particular statement is executed.
- To model this, we will introduce the notion that each statement in a loop has an implicit iteration range input
- e.g:

  A (I, J) (irange) denotes the statement is controlled by iteration range "irange"

- The iteration range variable for a given loop will be assigned at a point corresponding to the point where the iteration range would be evaluated in the original program.

- e.g: 20: irange 1 = (1, 100)
DO I = irange 1
40: L = 2 * I (irange (1))
60: irange 2 = (1, L) (irange (1))
DO I = irange 12
80: A(I, J) = 0 (irange (2))
END DO
END DO

- now, substituting constants

```
DO I = 1, 100
  L = 2 * I (1, 100)
  DO J = 1, L (1, 100)
  A(I, J) = 0 (1, L) (1, 100)
END DO
END DO
```

It can be vectorized as

```
DO I = 1, 100
  L = 2 * I
  A(I, 1:L) = 0
END DO
```

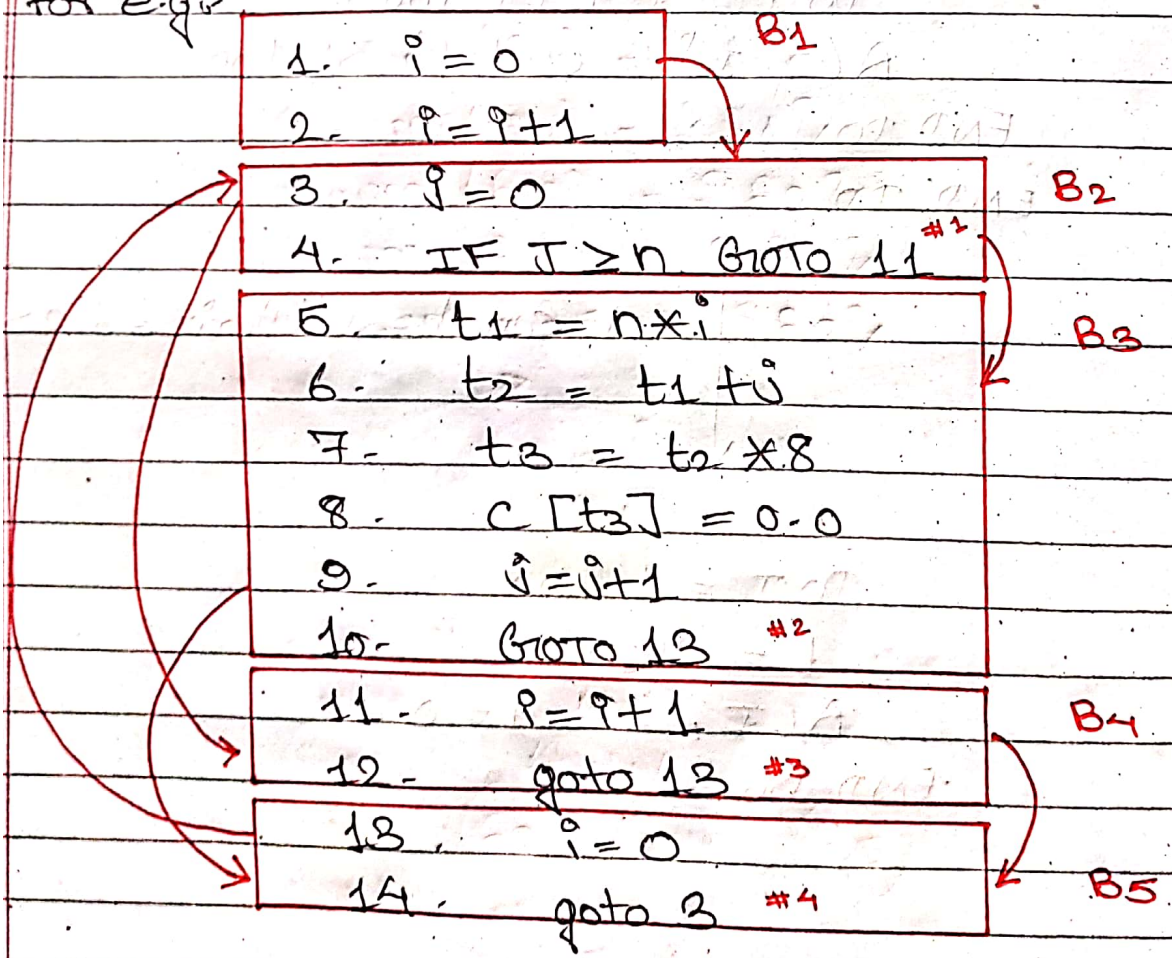
Page No.
Date : / /

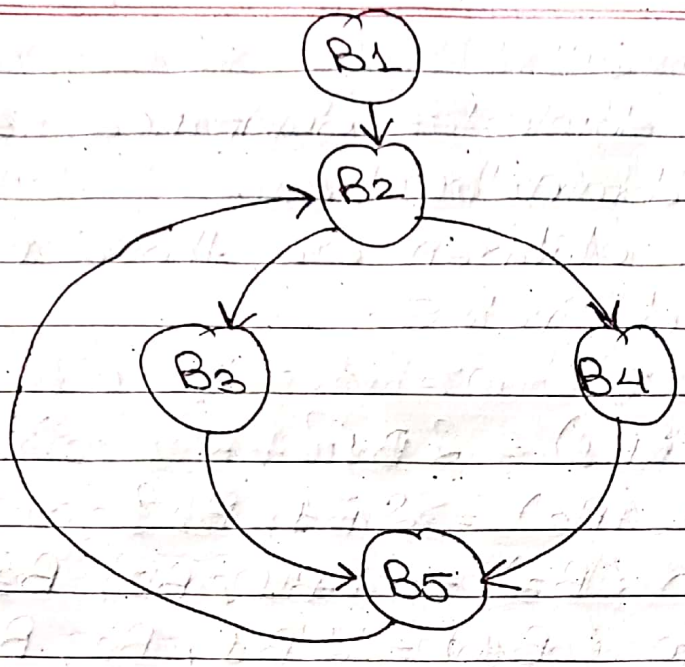
CFG (Control Flow Graph):

It is a directed graph whose nodes are the basic blocks of a program.

Dominators:

A node "d" of a flow graph G, dominates a node "n" if every path in G from the initial node to "n" goes through "d".
It is denoted as d dom n or d >> n.
every node dominates itself.
The initial node dominates all nodes in G.
for e.g:

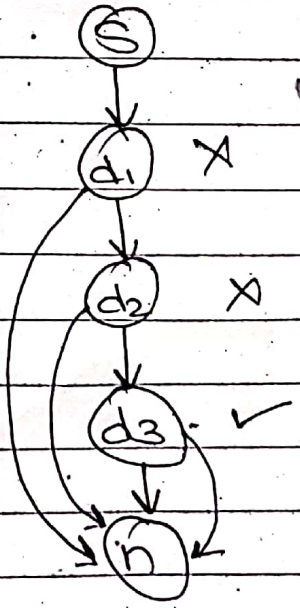




CFG1

* Immediate dominator:

- node d immediately dominates n iff
  1. d dom n
  2. d ≠ n
  3. ∄ m ∴ (m ≠ n) ∧ (m ≠ d) ∧ (d dom n) ∧ (m dom n)
- i.e, closed dominator.



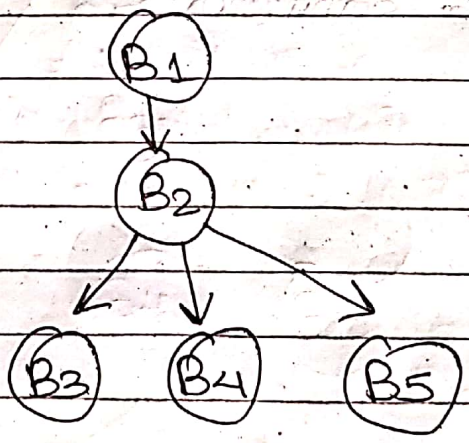
Page No.
Date : / /

* Dominator Tree:

- Immediate ~~dominator~~ dominance relationship form tree called dominator tree.
- Each node's children are those nodes, it immediately dominates.
- e.g:-

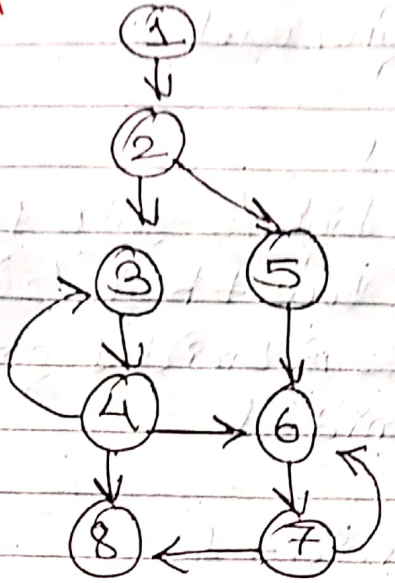
dom (B1) = {B1}
dom (B2) = {B1, B2}
dom (B3) = {B1, B2, B3}
dom (B4) = {B1, B2, B4}
dom (B5) = {B1, B2, B5}

i. dom (B1) = {ϕ}
i dom (B2) = {B1}
i dom (B3) = {B2}
i dom (B4) = {B2}
i dom (B5) = {B2}



Page No.
Date: / /

* Create Dominator tree from following control flow graph:



Sol

dom(1) = {1}
dom(2) = {1,2}
dom(3) = {1,2,3}
dom(5) = {1,2,5}
dom(4) = {1,2,3,4}
dom(6) = {1,2,6}
dom(7) = {1,2,6,7}
dom(8) = {1,2,8}

i dom(1) = {0}
i dom(2) = {1}
i dom(3) = {2}
i dom(4) = {3}
i dom(5) = {2}
i dom(6) = {2}
i dom(7) = {6}
i dom(8) = {2}

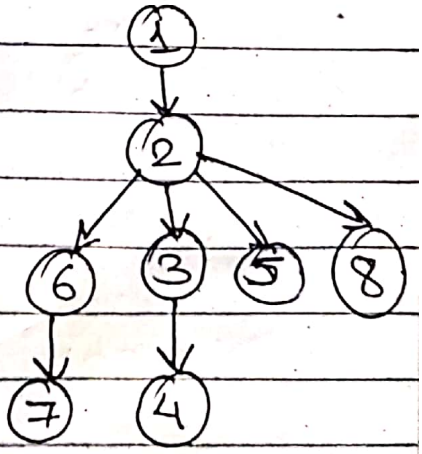
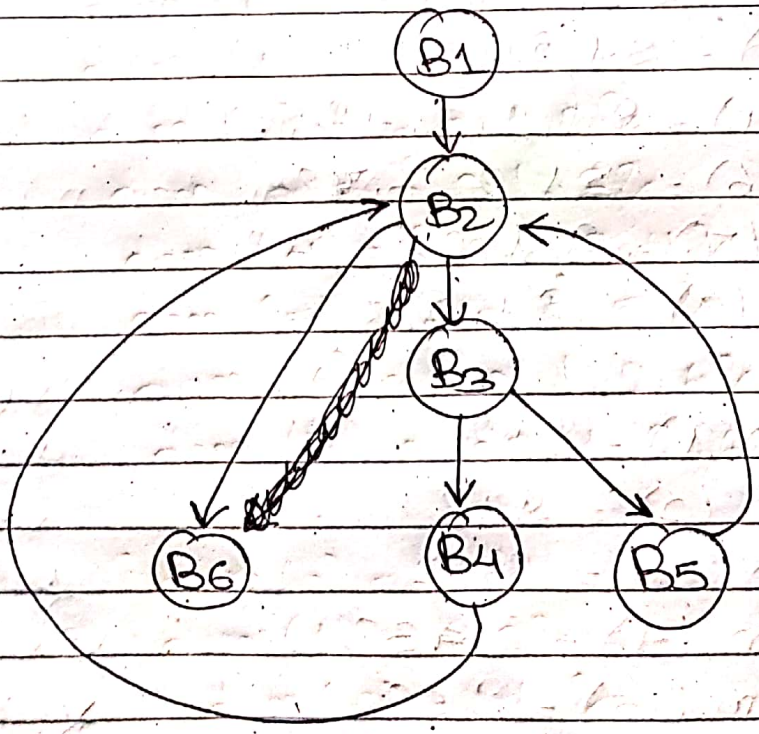


Fig: Dominant tree

Page No.
Date : / /

Create a Dominant Tree from following program:

```
1.  int low, high, mid                    B1
2.  low = 0
3.  high = n-1
4.  IF low > high /2 goto 11              B2
5.  mid = (low + high /2)                 B3
6.  IF x > A[mid] goto 9
7.  high = mid - 1                        B4
8.  goto 4
9.  low = mid + 1                         B5
10. goto 4
11. return mid                            B6
12. END
```



CFG

dom (B1) = {B1}
dom (B2) = {B1, B2}
dom (B3) = {B1, B2, B3}
dom (B4) = {B1, B2, B3, B4}
dom (B5) = {B1, B2, B3, B5}
dom (B6) = {B1, B2, B6}

idom (B1) = {ϕ}
idom (B2) = {B1}
idom (B3) = {B2}
idom (B4) = {B3}
idom (B5) = {B3}
idom (B6) = {B2}

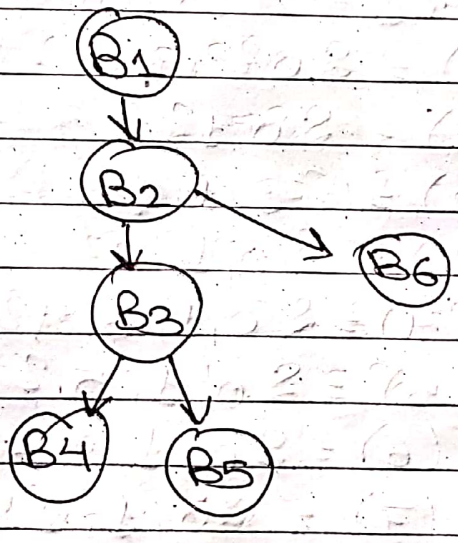
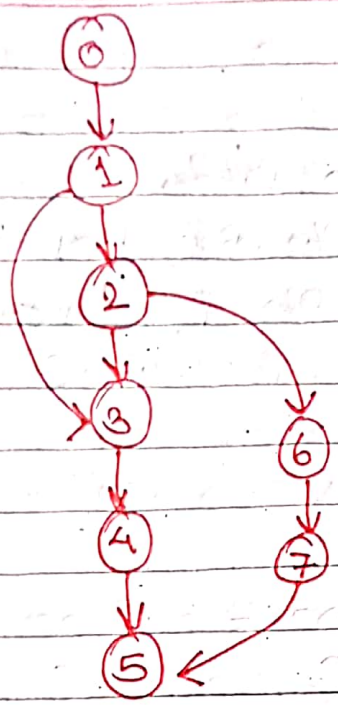


Fig: Dominant Tree



CFG

Create a Dominant Tree from given CFG:

= Sol

dom (0) = {0}
dom (1) = {0,1}
dom (2) = {0,1,2}
dom (3) = {0,1,3}
dom (4) = {0,1,3,4}
dom (5) = {0,1,5}
dom (6) = {0,1,2,6}
dom (7) = {0,1,2,6,7}

now, i dom (0) = {Φ}
i dom (1) = {0}
i dom (2) = {1}
i dom (3) = {1}
i dom (4) = {3}
i dom (5) = {1}
i dom (6) = {2}
i dom (7) = {6}

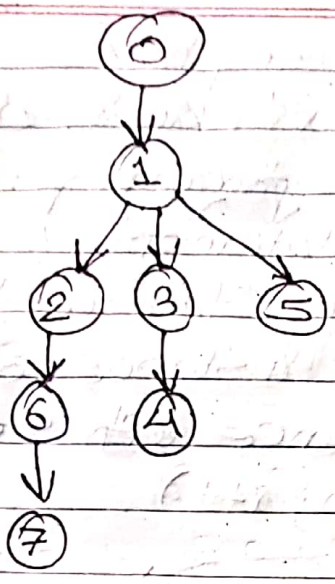


fig:- Dominant Tree

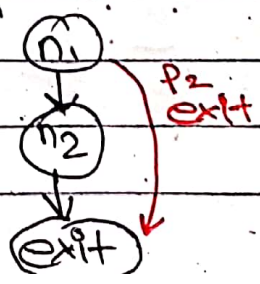
* Control Dependence:

- If two statements are dependent to each other based on a control.
- For e.g:-

```
n1  while (power != 0)
    {
    S1 :  z = z + x;
    S2 :  power = power + 1;
    }
```

This will execute only if power != 0

- Let C be a CDG (Control dependence Graph) with nodes n1 & n2, n1 being predicate (i.e. control).
- node n2 is control dependent on n1 if
  1. There is at least one path from n1 to program exit that includes n2.
  2. and at least one path from n1 to perform exit that excludes n2.



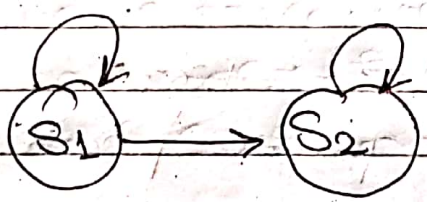
Page No.
Date : / /

* Program dependence Graph:

- It describes different kinds of dependence among statements in a program p.
- ① node → assignment statement.
- ② edge → edge from statement S_1 to statement S_2 iff some instance of S_1 shares a data dependence with some instance of S_2
- e.g:-

```
for (i=0; i<n; i++)
{
    x[i] = x[i] + x[i-1]; //S1
}
for (i=0; i<n-10; i++)
{
    x[i] = x[i] + y[x[i]] //S2
}
```

i.e.



Page No.
Date: / /

Unit - 6
Interprocedural Analysis and optimization

* Interprocedure Analysis:
- It is the process of gathering information about the whole program instead of a single procedure
- i.e. Modifying more than one procedures.
- e.g:- Side-effect Analysis, Alias analysis.

* Side-effect Analysis:
- Modification and reference side effect.
- ① MOD(S) → Set of variables that may be modified as a side effect of call at S.

② REF(S) → Set of variables that may be referenced as a side effect of call at S.

```
DO I = 1, N
  S1 : CALL P
  S2 : X(I) = X(I) + Y(I)
END DO.
```

- can Vectorize if,
  1. P neither modifies X nor uses X
  2. P doesnot modify Y.

Page No.
Date : / /

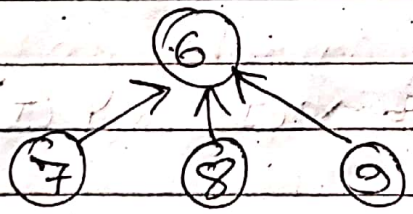
* Alias Analysis:

- It describes a situation in which a data location can be accessed through different symbolic names (called aliases).
- It is a technique used to determine if a storage location may be accessed in more than one way.
- e.g:-

```
int *p;
int x = 4;
p = &x;
print ("%d", *p);
```

* Call Graph:

- It is a directed graph in which each node is call site (i.e. a procedure) & each edge connects two nodes (f, g) indicating that procedure f calls procedure g.
- e.g:-



procedure 7, 8, 9 call the procedure 6.

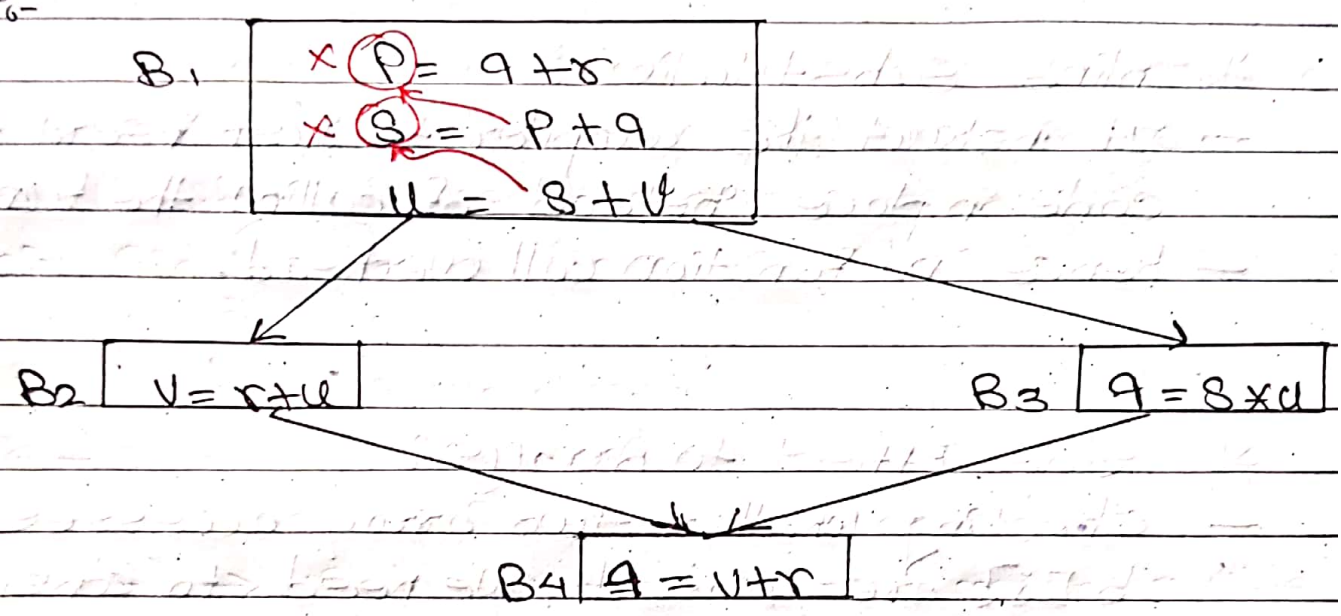
* Live & Use Analysis:

① USE(s) → set of variables that are used in s before any assignment (i.e. variable in RHS but not in LHS of prior statement of s)

Page No
Date : / /

② KILL (S) → Set of variables that are assigned a value in S (i.e. variables in LHS)

- eg:-



node	USE	KILL
B1	q, r, v	P, S, u
B2	r, u	v
B3	s, u	q
B4	v, r	q

* Constant Propagation and alias analysis:

```
2
  int x, y;
  int *p;
  p = &x;
  x = 5;
  *p = 42;
  y = x + 1;
2
```

x can be either 42 or 5.

- If P might point to x, then we have two aliases that reach the last statement, so x is not a constant.

* Inline Substitution:

- It instruct the compiler to just insert the function code in place, instead of calling the function.
- hence, no function call overhead.

* Side Effect to Arrays:

- Checking whether two array accesses a[i] and b[i] are aliased, we need to consider not only the points to set of base variables, but also the values of i & j.

a[i] & b[i]	i = UNDEF	j = C_2	j = NAC
P = UNDEF	Not Aliased	Not Aliased	Not Alia
P = C_1	Not Aliased	Aliased if $C_1 = C_2$	Aliased
P = NAC	Not Aliased	Aliased	Aliased

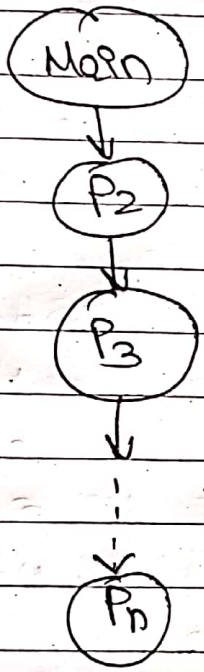
* Procedure cloning:

- It is an interprocedural transformation where the compiler creates specialized copies of procedure bodies.

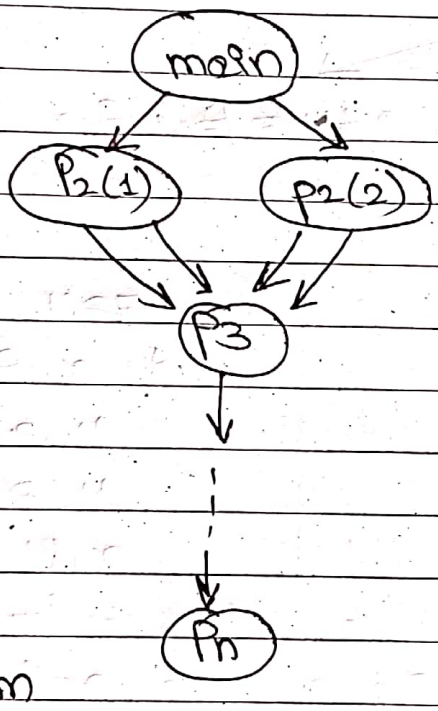
Page No.
Date : / /

— e.g:-

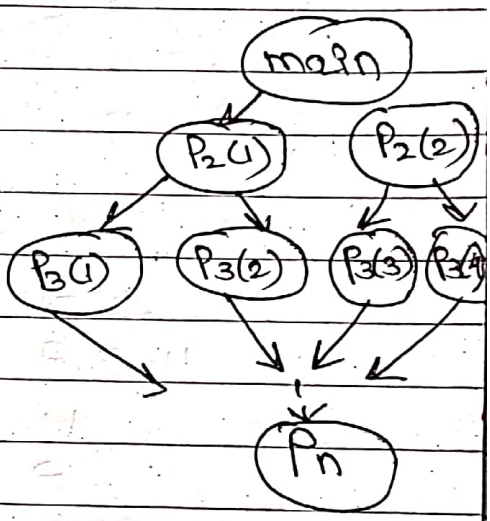
```
Subroutine P2 (input)
call p3 (3 * input)
call p3 (4 * input)

Subroutine P3 (input)
call p4 (5 * input)
call p4 (6 * input)
```



Fig(1): Initial program



Fig(2): After cloning P2



Fig(3): After cloning P3