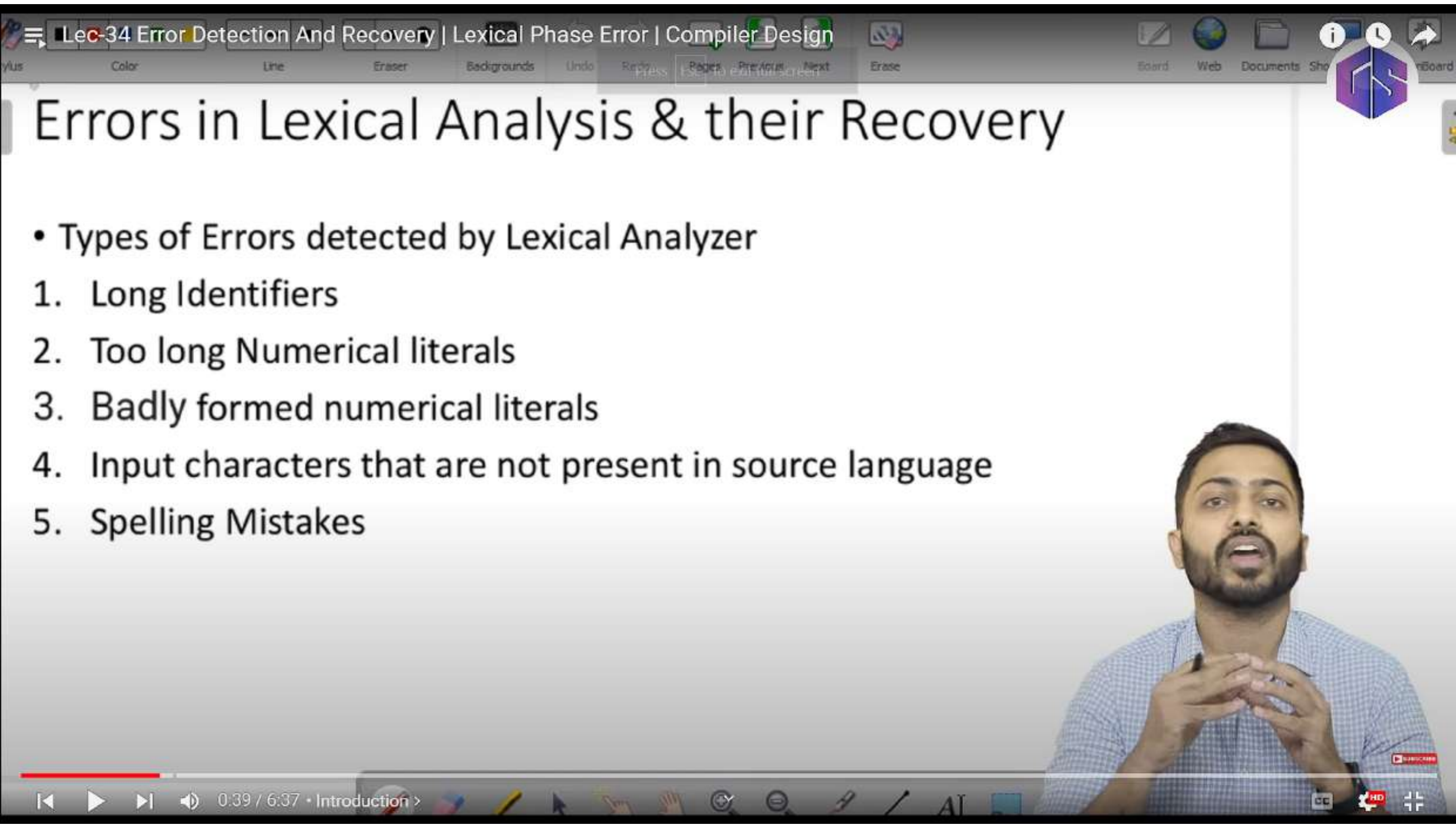
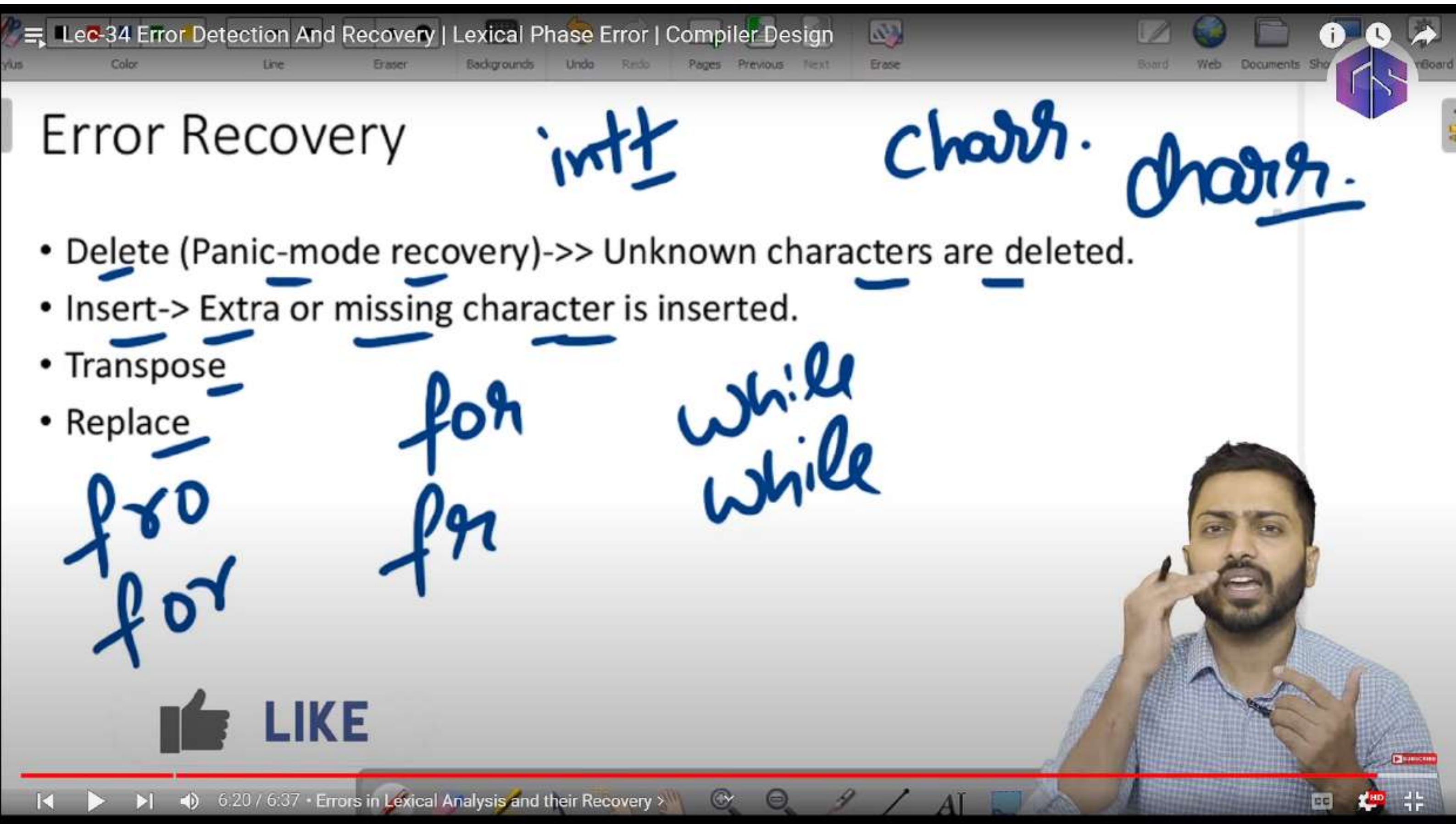
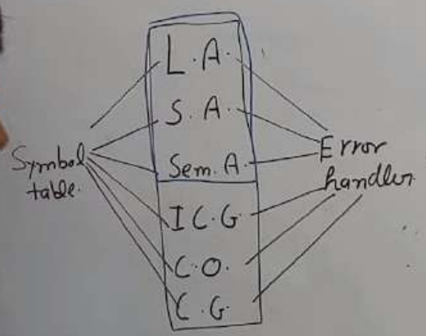

Lec-34 Error Detection And Recovery | Lexical Phase Error | Compiler Design
Errors in Lexical Analysis & their Recovery
• Types of Errors detected by Lexical Analyzer
1. Long Identifiers
2. Too long Numerical literals
3. Badly formed numerical literals
4. Input characters that are not present in source language
5. Spelling Mistakes
0:39 / 6:37 • Introduction ›


Lec-34 Error Detection And Recovery | Lexical Phase Error | Compiler Design
Error Recovery
intt
charr.
charr.
• Delete (Panic-mode recovery)->> Unknown characters are deleted.
• Insert-> Extra or missing character is inserted.
• Transpose
• Replace
for
fr
while
while
fro
for
LIKE
6:20 / 6:37 • Errors in Lexical Analysis and their Recovery >

CD47: Error Detection And Recovery in Compiler Design | Lexical Phase Error and Recovery in Hindi

Error Detection and Recovery



= Error Detection + Error Reporting + Error Recovery

Panic mode

int $\underline{a}$, ~~\$b~~, sum, $\boxed{\$z}$;

Recovery method \ Error	Lexical Phase Error	Syntactic Phase Error	Semantic Phase Error
Panic Mode	✓	✓	×
Phrase level	×	✓	×
Error production	×	✓	×
Global production	×	✓	×
Using Symbol table	×	×	✓

① Lexical Phase Error :-

(i) Exceeding length of identifier — int sum;

(ii) Appearance of illegal character — int a, \$;

(iii) Unmatched string or comment — hello" — */

Ex:

```
void main()
{
  int a, @, $; /*variable declaration */
  a=10;
  printf("%d", a); $
}
```

13:47 / 14:35

CD48: Error Detection And Recovery in Compiler Design|Syntactic Phase Error & Semantic Phase Error

Error Detection and Recovery

Syntactic Phase Error.

(i) missing parenthesis eg. printf("hello" ;

(ii) missing operator $\boxed{a+b\ c}$

(iii) misspelled keyword switch(ch)

(iv) Colon in place of semicolon $\boxed{a=1:}$ $\boxed{a=1;}$

(v) Extra Blank space. /* comment* /

Recovery.

(1) Panic Mode : Similar as in lexical phase error

(2) Phrase level Recovery: when parser encounter an error it perform necessary action on Remaing input and parse rest of input

(3) Error production. Add exta grammar production and make an Augmented grammar and parse the input.

(4) global correction. The parser examine the whole program and tries to find out closest match for it which is error free. due to high space and time complexity it is not implemented practically.

Semantic Phaser error.

(i) Incompatible type of operands.

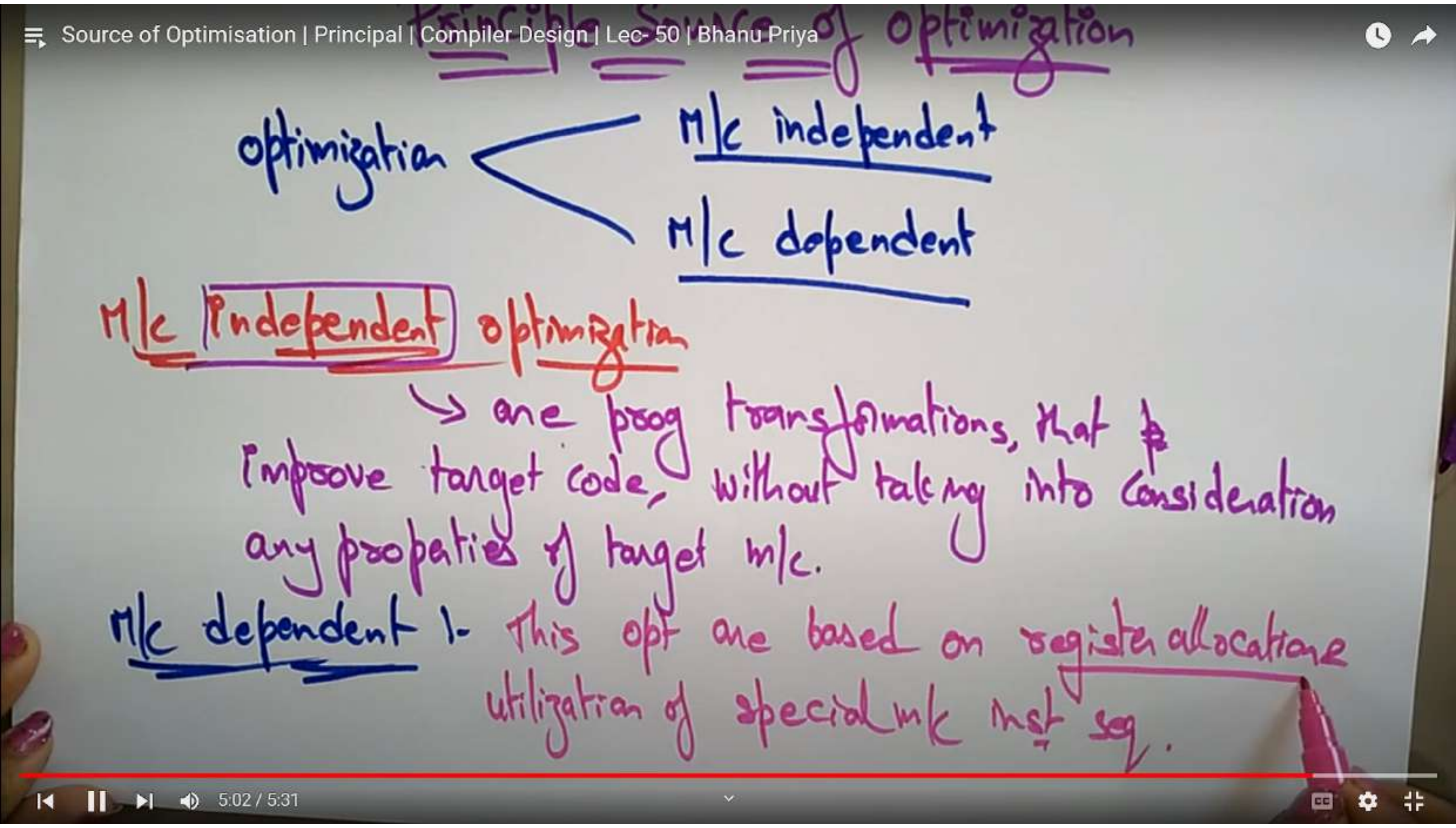
(ii) Undeclared Variable.

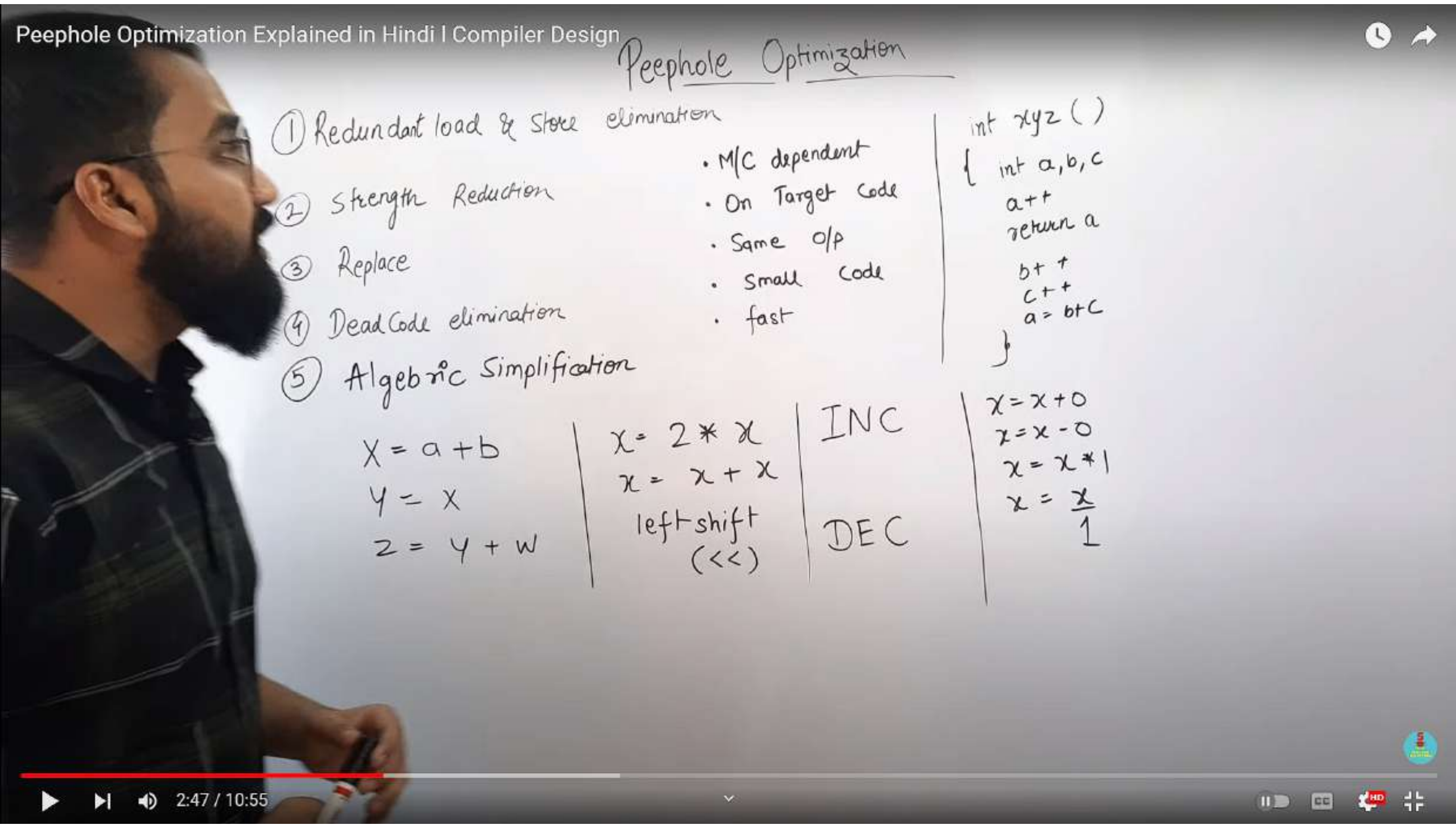
(iii) Not matching actual argument with formal argument.

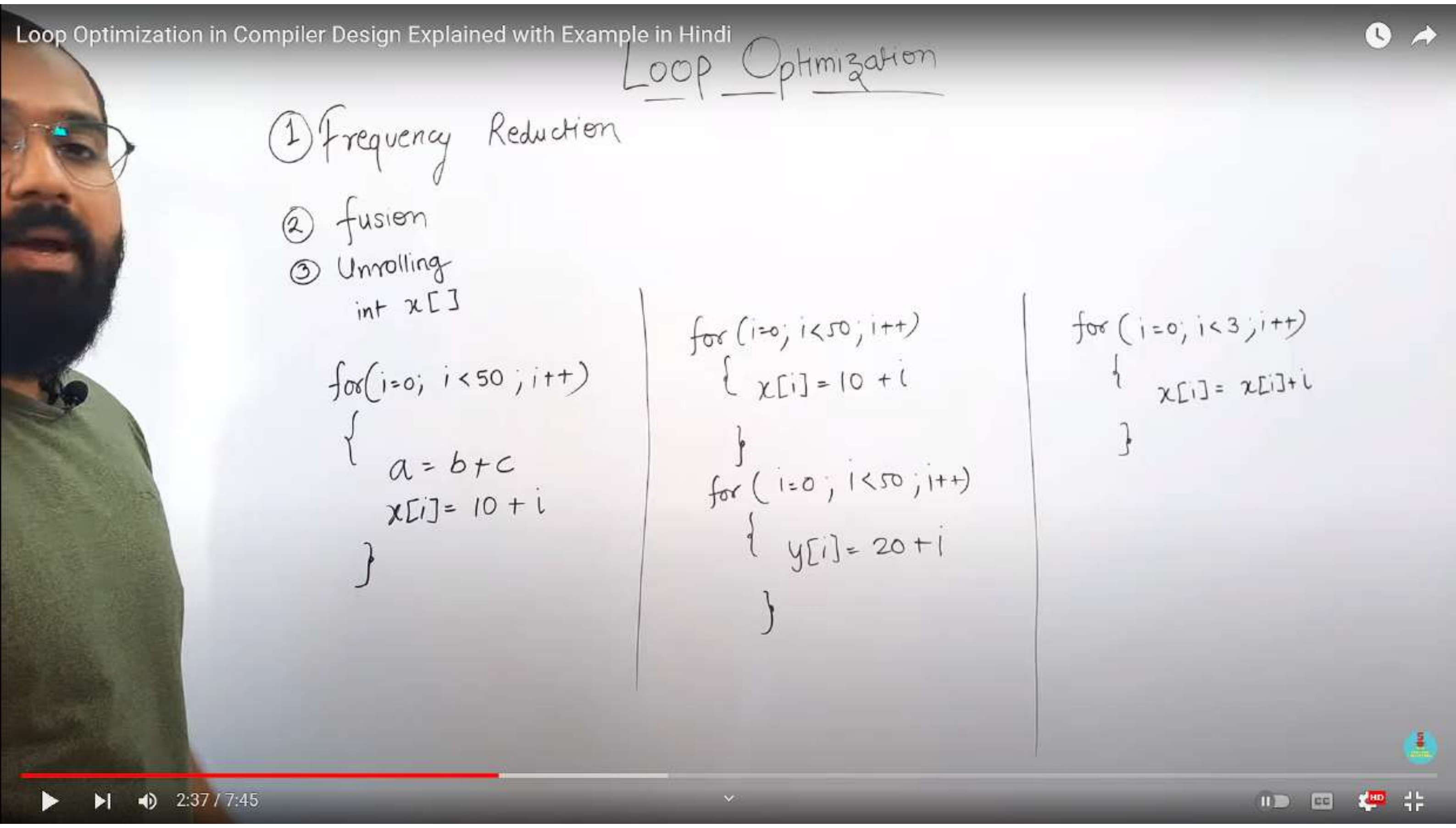
e.g. int a[10], b;

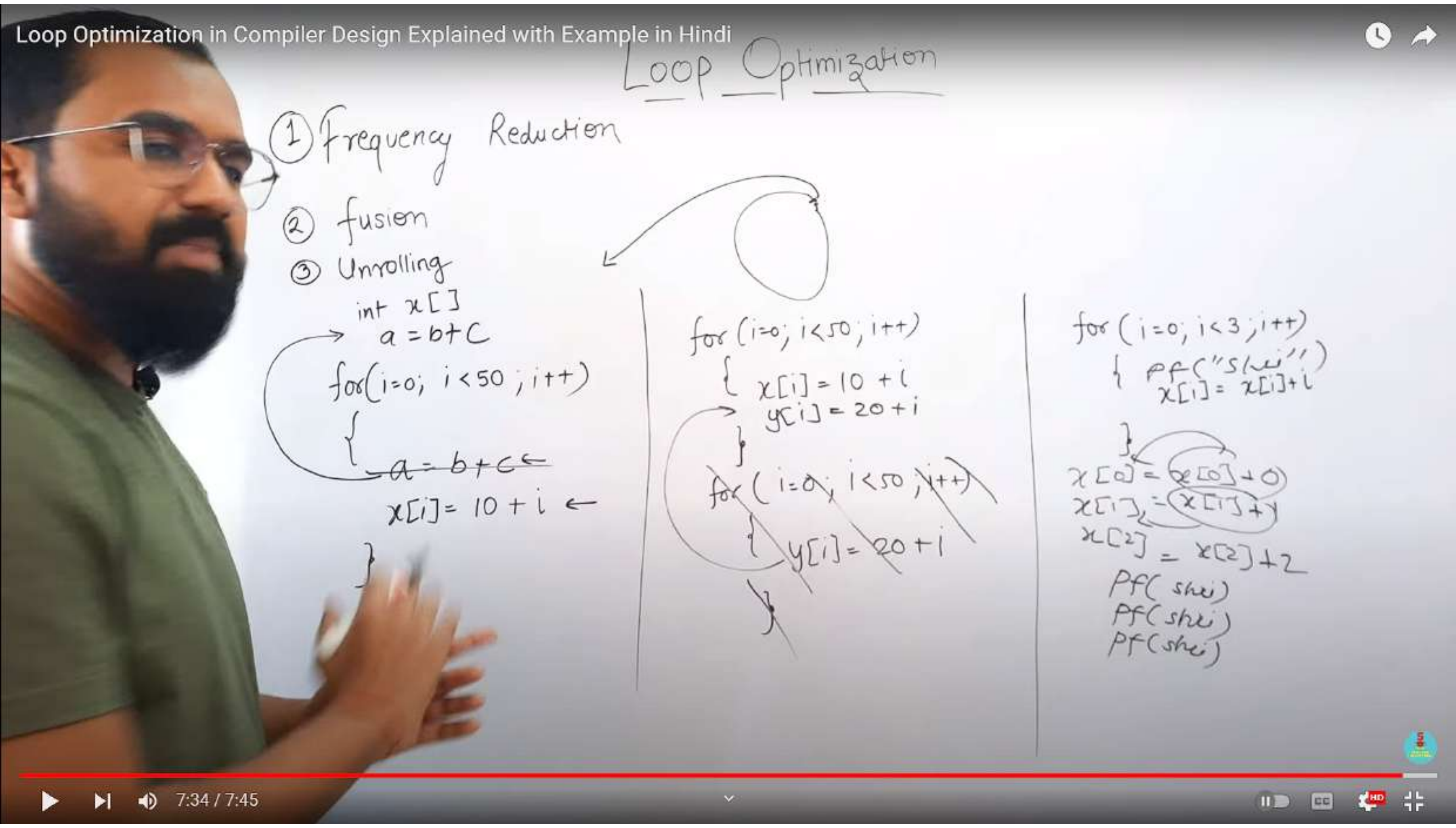
a = b;

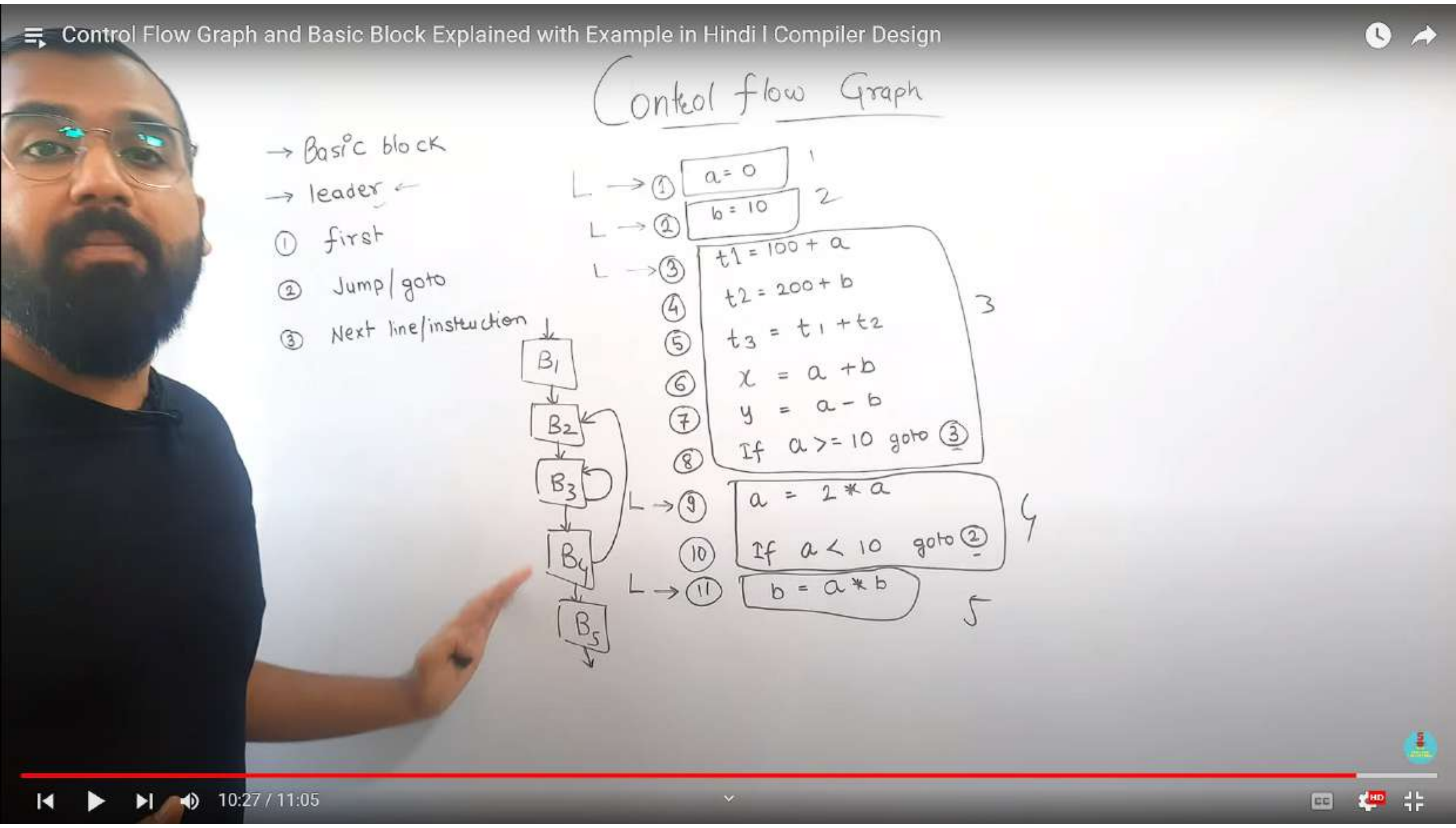
4:35 / 14:01

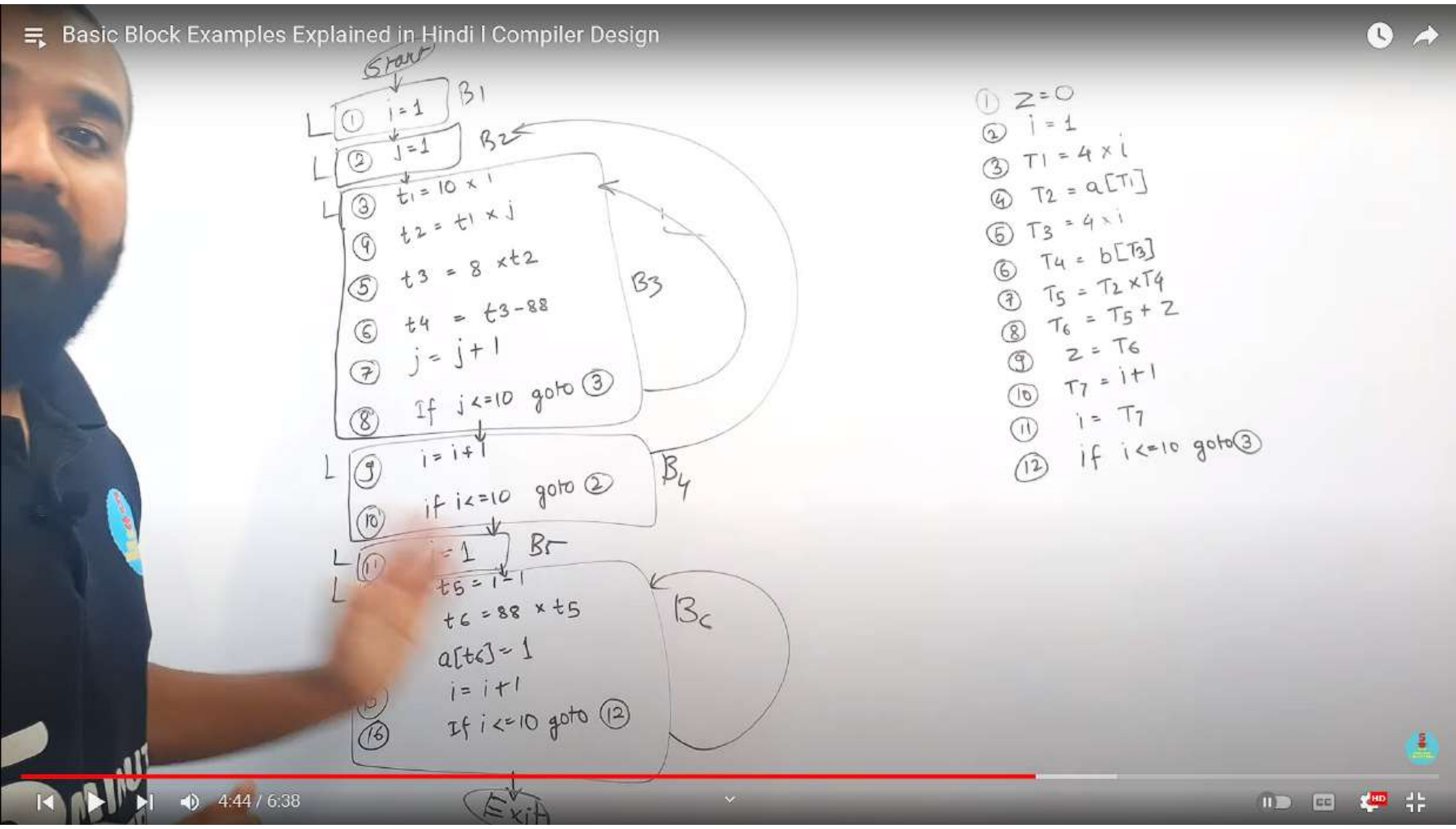

Source of Optimisation | Principal | Compiler Design | Lec- 50 | Bhanu Priya
Principle Source of optimization
optimization
M/c independent
M/c dependent
M/c independent optimization
→ are prog transformations, that improve target code, without taking into consideration any properties of target m/c.
M/c dependent :- This opt are based on register allocation & utilization of special m/c inst seq.
5:02 / 5:31


Peephole Optimization Explained in Hindi I Compiler Design
Peephole Optimization
1 Redundant load & Store elimination
2 Strength Reduction
3 Replace
4 Dead Code elimination
5 Algebric Simplification
• M/C dependent
• On Target Code
• Same O/P
• Small Code
• fast
int xyz ( )
{ int a, b, c
a++
return a
b++
c++
a = b+c
}
X = a + b
Y = X
Z = Y + W
X = 2 * X
X = X + X
left shift
(<<)
INC
DEC
X = X + 0
X = X - 0
X = X * 1
X = X / 1
2:47 / 10:55

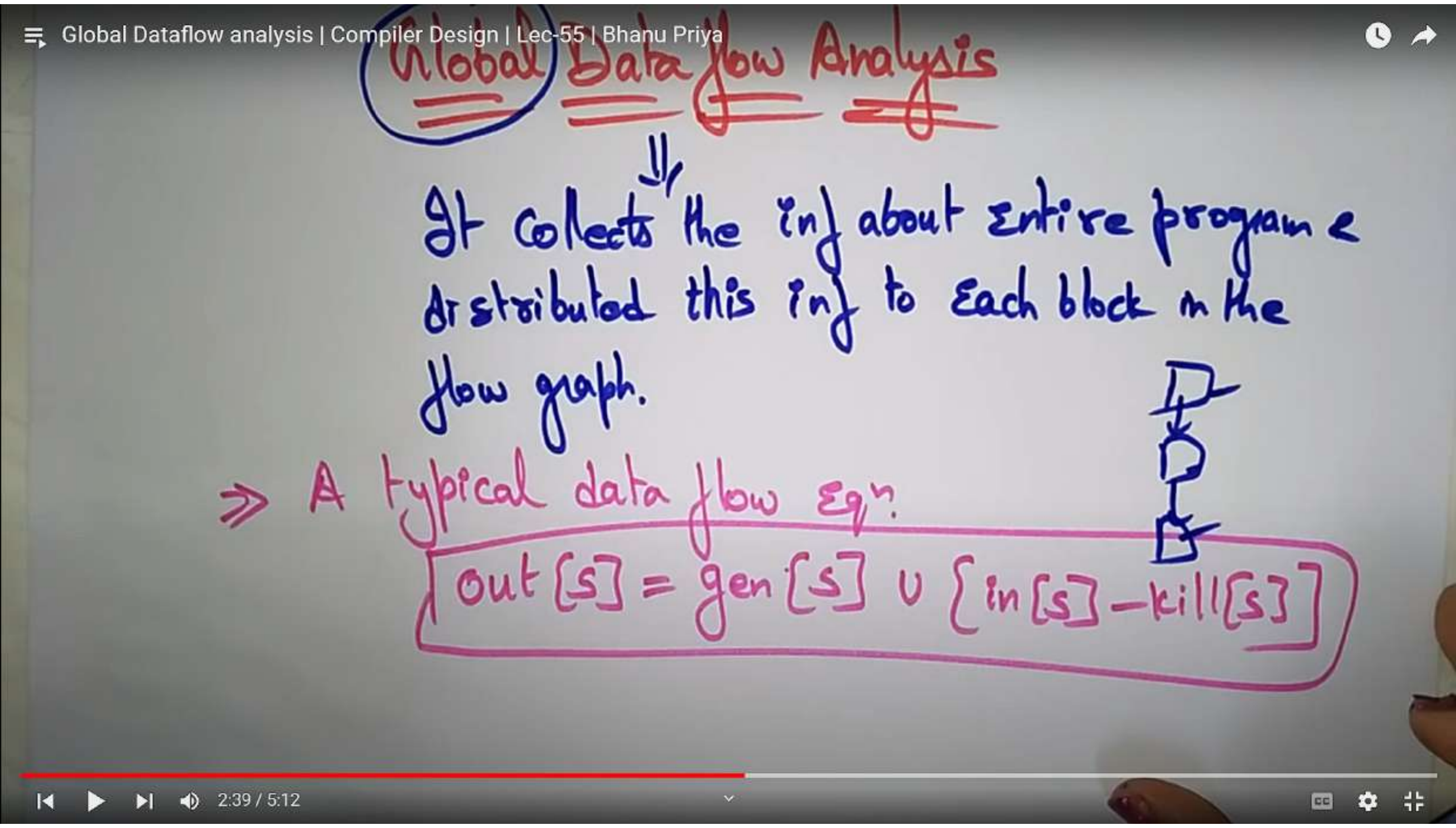

Loop Optimization in Compiler Design Explained with Example in Hindi
Loop Optimization
1 Frequency Reduction
2 fusion
3 Unrolling
int x[ ]
for(i=0; i<50 ; i++)
{
a = b+c
x[i]= 10 + i
}
for (i=0; i<50; i++)
{ x[i] = 10 + i
}
for (i=0 ; i<50 ; i++)
{ y[i] = 20 + i
}
for (i=0; i<3 ; i++)
{
x[i] = x[i]+i
}
2:37 / 7:45

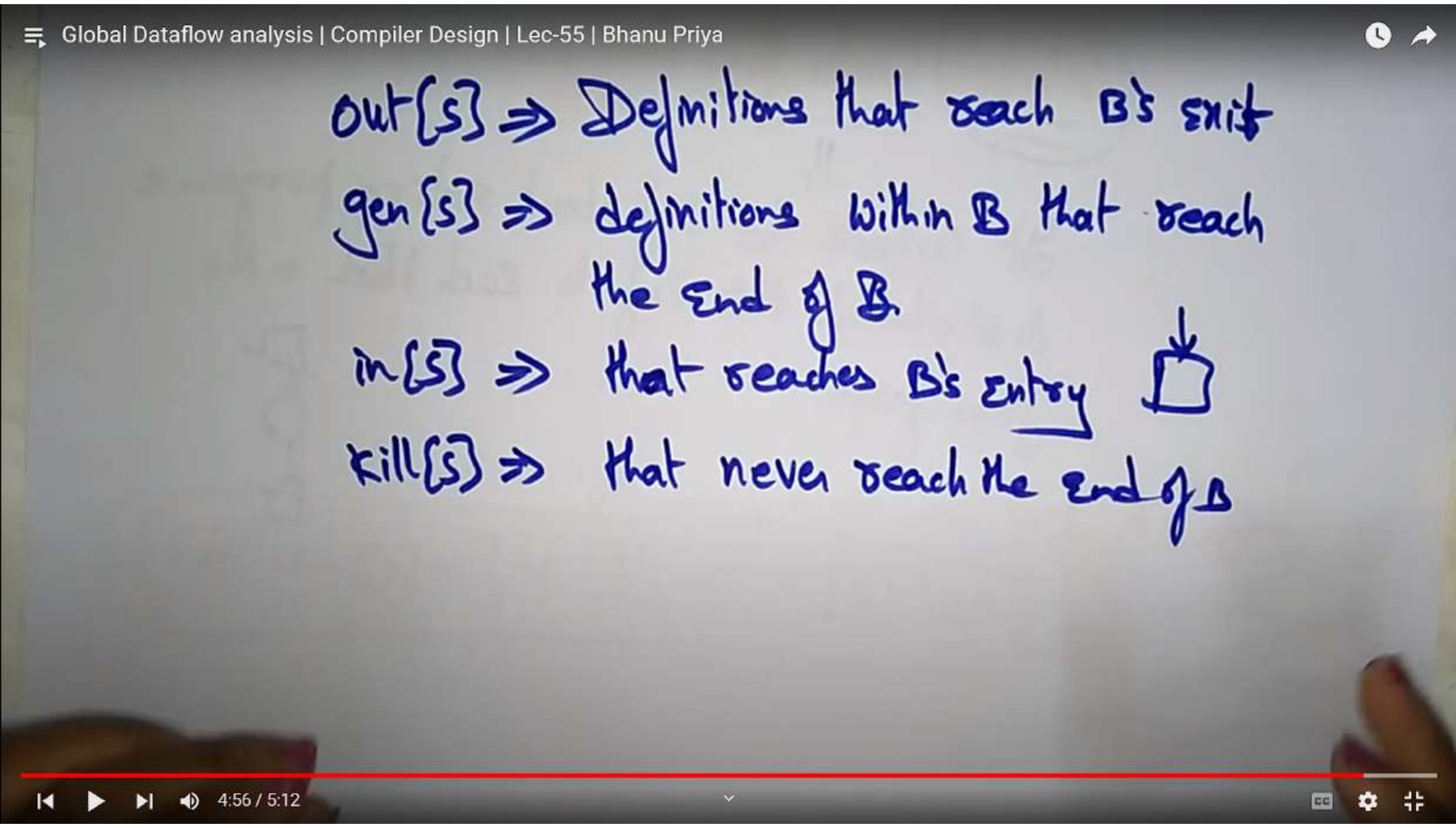

Loop Optimization in Compiler Design Explained with Example in Hindi
Loop Optimization
1 Frequency Reduction
2 fusion
3 Unrolling
int x[ ]
a = b+C
for(i=0; i<50 ; i++)
{
a = b+c
x[i]= 10 + i
}
for (i=0; i<50; i++)
{ x[i] = 10 + i
y[i] = 20 + i
}
for (i=0; i<50; i++)
{ y[i] = 20+i
}
for (i=0; i<3; i++)
{ pf("shei")
x[i] = x[i]+i
}
x[0] = x[0] + 0
x[1] = x[1] + 1
x[2] = x[2] + 2
pf(shei)
pf(shei)
pf(shei)
7:34 / 7:45

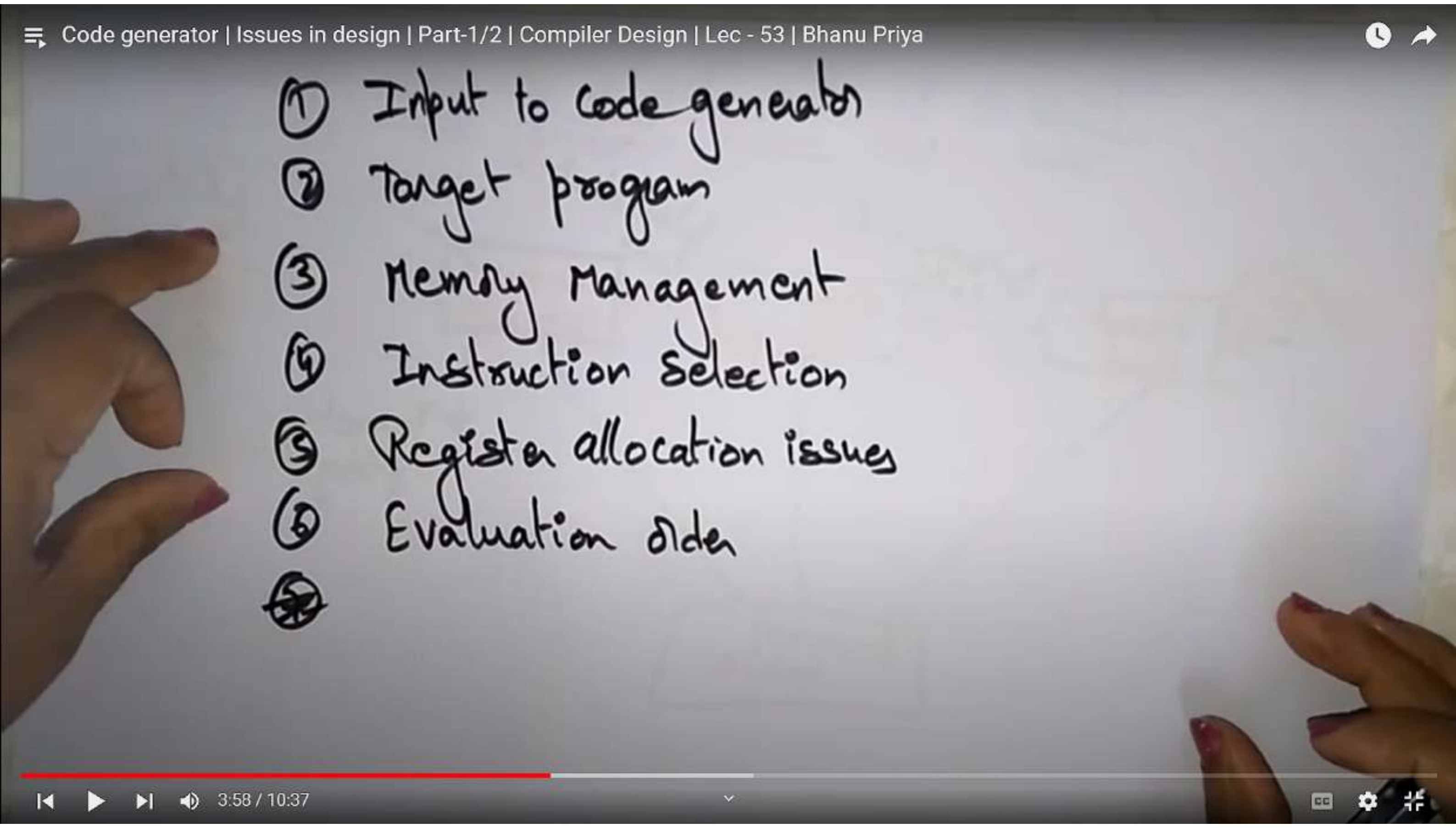

Control Flow Graph and Basic Block Explained with Example in Hindi I Compiler Design
Control flow Graph
→ Basic block
→ leader ←
① first
② Jump/goto
③ Next line/instruction
B1
B2
B3
B4
B5
L → ① a = 0
L → ② b = 10
L → ③ t1 = 100 + a
④ t2 = 200 + b
⑤ t3 = t1 + t2
⑥ x = a + b
⑦ y = a − b
⑧ If a >= 10 goto ③
L → ⑨ a = 2 * a
⑩ If a < 10 goto ②
L → ⑪ b = a * b
1
2
3
4
5
10:27 / 11:05

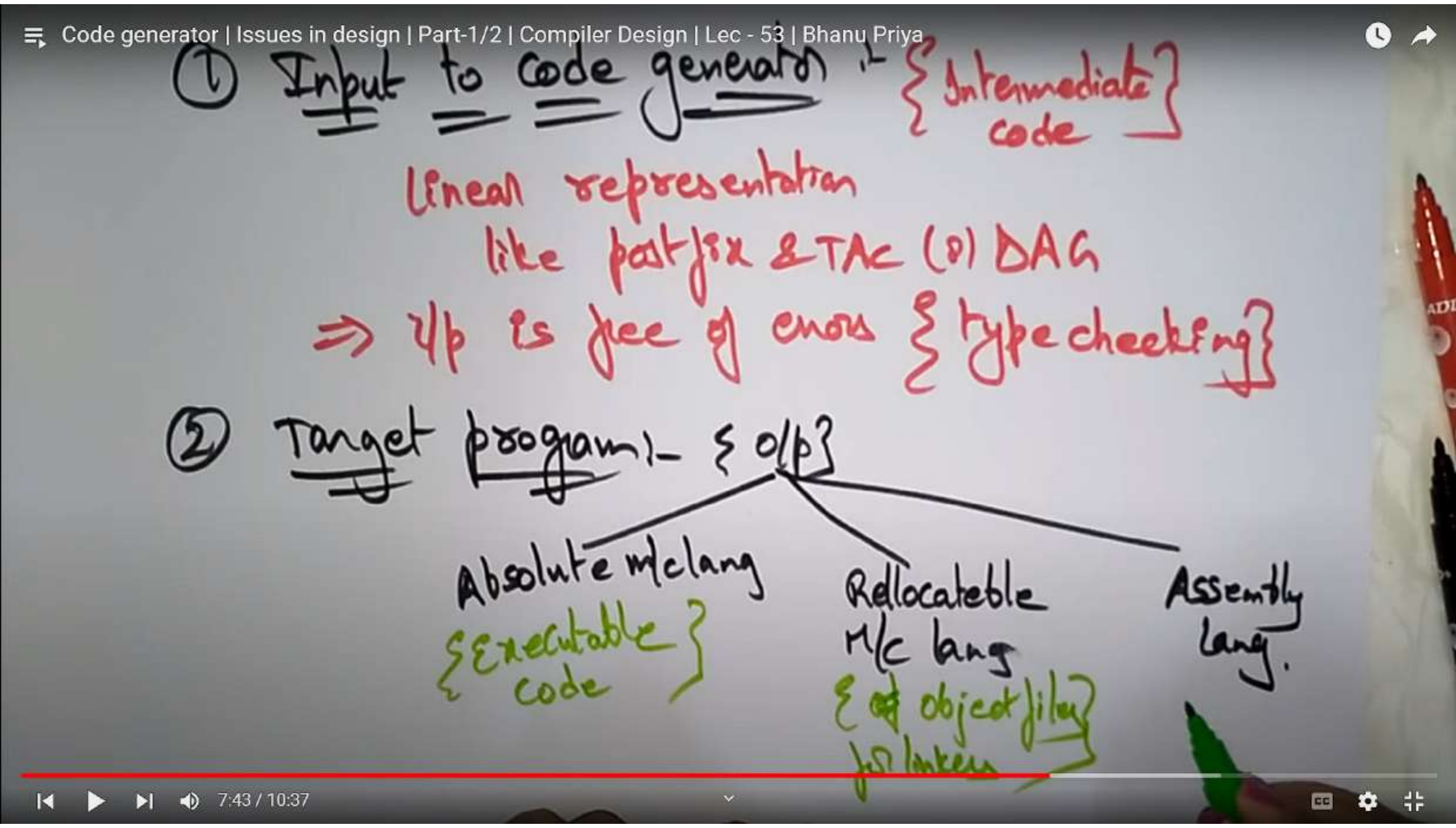


Basic Block Examples Explained in Hindi I Compiler Design
Start
B1
i = 1
B2
j = 1
t1 = 10 x i
t2 = t1 x j
t3 = 8 x t2
t4 = t3 - 88
j = j + 1
If j <= 10 goto 3
B3
i = i + 1
if i <= 10 goto 2
B4
i = 1
B5
t5 = i - 1
t6 = 88 x t5
a[t6] = 1
i = i + 1
If i <= 10 goto 12
B6
Exit
z = 0
i = 1
T1 = 4 x i
T2 = a[T1]
T3 = 4 x i
T4 = b[T3]
T5 = T2 x T4
T6 = T5 + Z
Z = T6
T7 = i + 1
i = T7
if i <= 10 goto 3
4:44 / 6:38

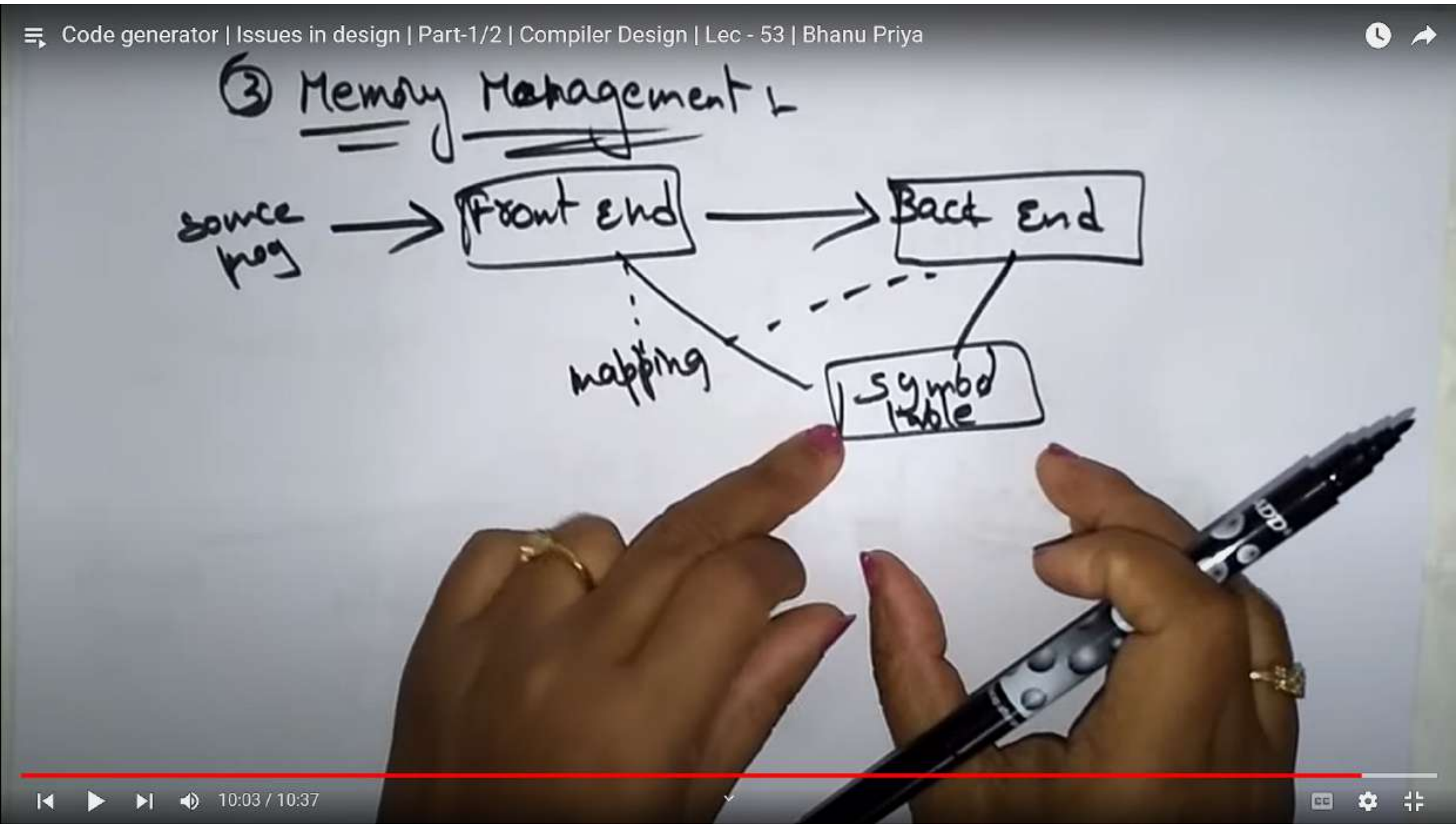

Global Dataflow analysis | Compiler Design | Lec-55 | Bhanu Priya
Global Data flow Analysis
It collects the inf about Entire programe
distributed this inf to Each block in the
flow graph.
⇒ A typical data flow Eqn:
out[S] = gen[S] ∪ {in[S] − kill[S]}
2:39 / 5:12

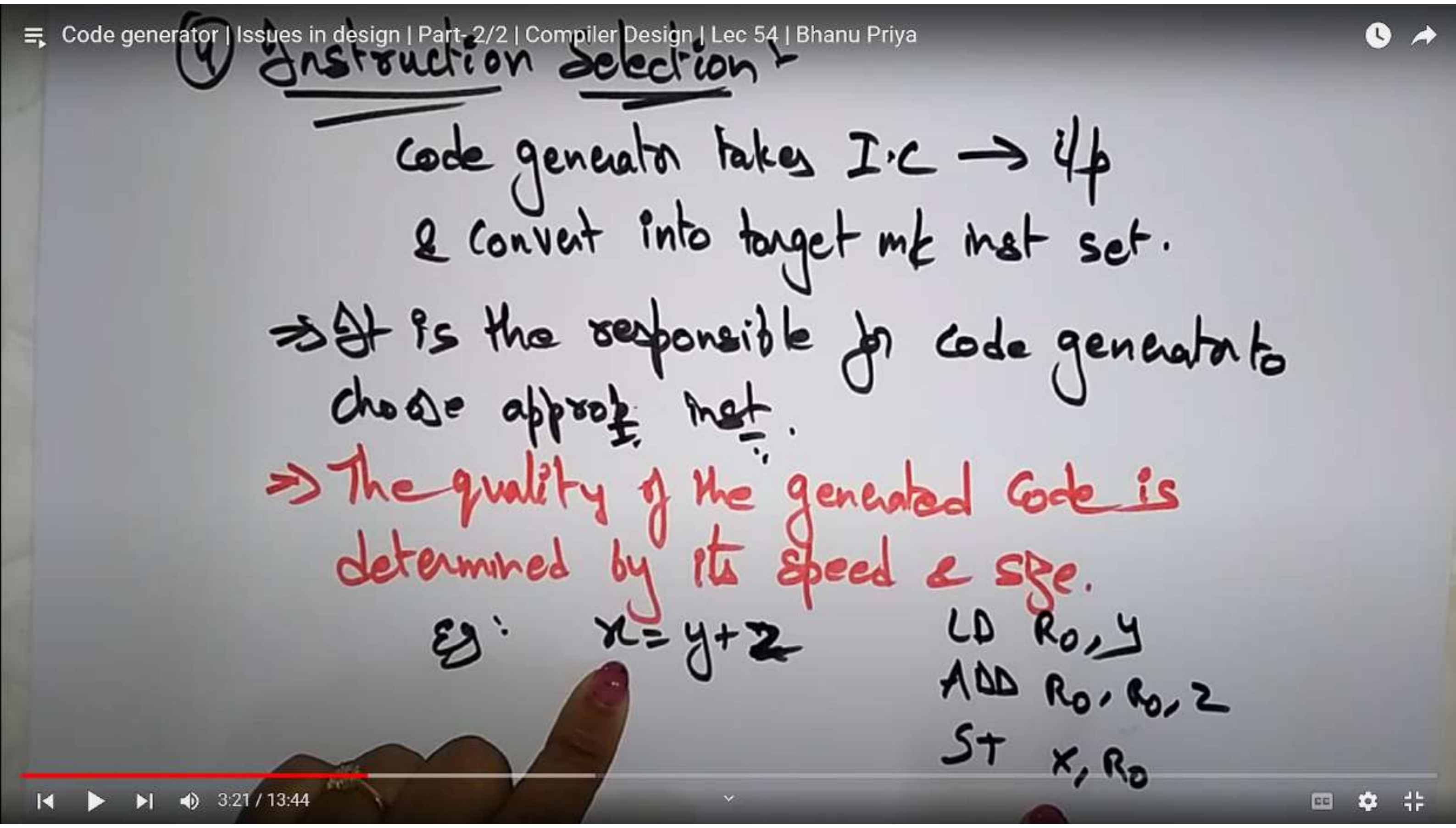

Global Dataflow analysis | Compiler Design | Lec-55 | Bhanu Priya
out[S] ⇒ Definitions that reach B's exit
gen[S] ⇒ definitions within B that reach the end of B.
in[S] ⇒ that reaches B's entry
kill[S] ⇒ that never reach the end of B
4:56 / 5:12

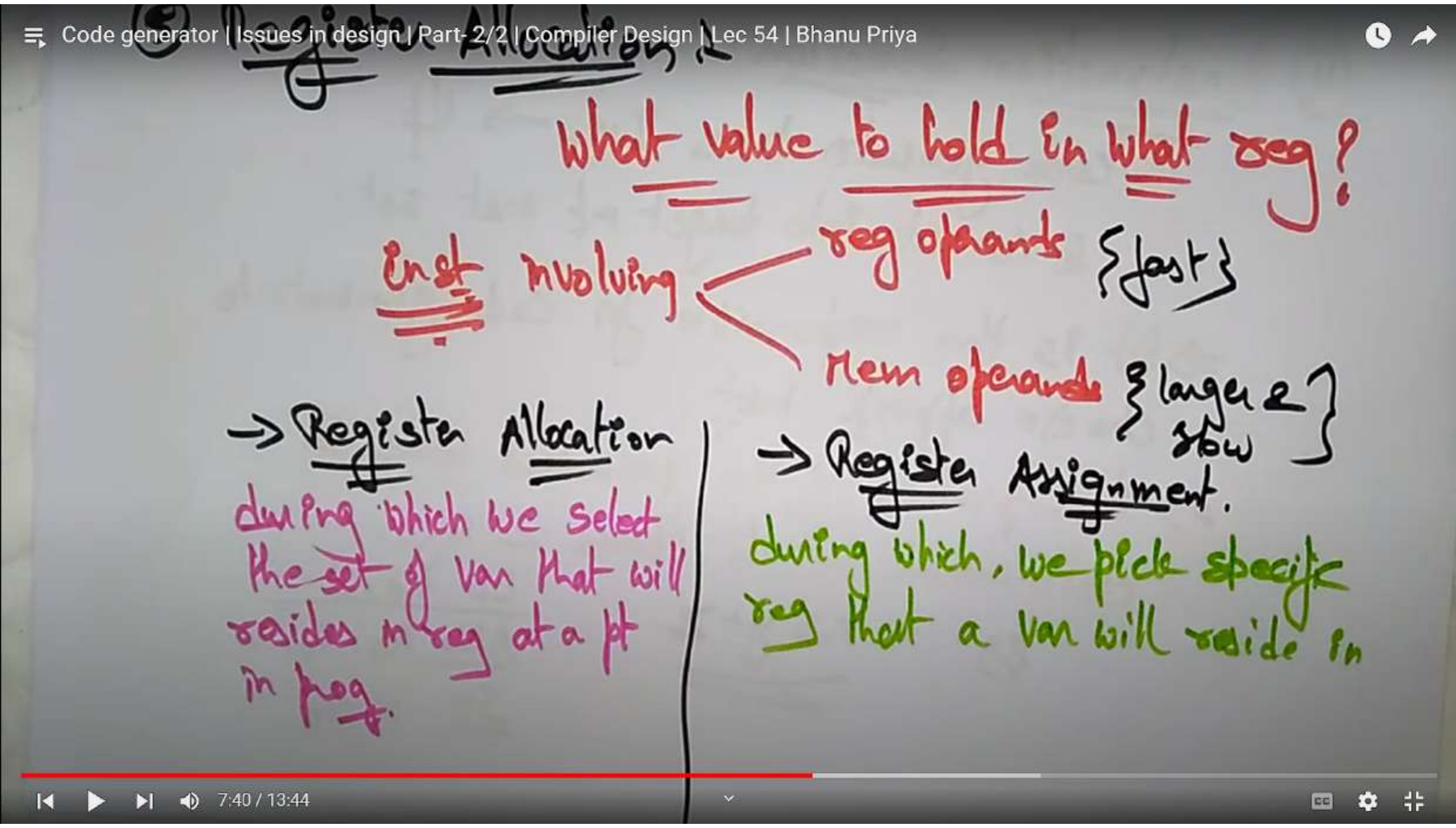

Code generator | Issues in design | Part-1/2 | Compiler Design | Lec - 53 | Bhanu Priya
① Input to code generator
② Target program
③ Memory Management
④ Instruction Selection
⑤ Register allocation issues
⑥ Evaluation order
3:58 / 10:37



Code generator | Issues in design | Part-1/2 | Compiler Design | Lec - 53 | Bhanu Priya
① Input to code generator :- {Intermediate code}
Linear representation
like postfix & TAC (or) DAG
⇒ I/p is free of errors {type checking}
② Target program :- {o/p}
Absolute m/c lang
{Executable code}
Rellocateble M/c lang
{object file for linkers}
Assembly lang.
7:43 / 10:37


Code generator | Issues in design | Part-1/2 | Compiler Design | Lec - 53 | Bhanu Priya
③ Memory Management
source prog
Front end
Back end
mapping
symbol table
10:03 / 10:37


Code generator | Issues in design | Part- 2/2 | Compiler Design | Lec 54 | Bhanu Priya
(4) Instruction Selection:
code generator takes I.C → i/p
& convert into target m/c inst set.
⇒ It is the responsible for code generator to
choose approp. inst.
⇒ The quality of the generated code is
determined by its speed & size.
Eg: x = y + z
LD R0, y
ADD R0, R0, z
ST x, R0
3:21 / 13:44


Code generator | Issues in design | Part- 2/2 | Compiler Design | Lec 54 | Bhanu Priya
Register Allocation :
what value to hold in what reg?
inst involving
reg operands {fast}
mem operands {larger & slow}
→ Register Allocation
during which we select the set of var that will resides in reg at a pt in prog.
→ Register Assignment.
during which, we pick specific reg that a var will reside in
7:40 / 13:44

Code generator | Issues in design | Part- 2/2 | Compiler Design | Lec 54 | Bhanu Priya

6 Evaluation order :

⇒ The order in which computations are performed can affect the efficiency of the target code.

⇒ When inst are independent then Evaluation order can be changed.

9:12 / 13:44