THEORY OF COMPUTATION AND COMPILERS

Unit - IV

Code Optimization | Data Flow Analysis

Code Optimization

- Consideration for Optimization
- Scope of Optimization
- *Basic blocks and Local Optimization
- Loop Optimization
- Frequency Reduction
- Folding
- DAG Representation

Data Flow Analysis

- Flow Graph
- Data Flow Equation 
- Global Optimization
- Redundant Sub Expression Elimination
- Induction Variable Elements
- Live Variable Analysis 
- Copy Propagation

Dr. R. Madana Mohana
Professor, Artificial Intelligence & Data Science | I/c-Head, Artificial Intelligence & Machine Learning
CHAITANYA BHARATHI INSTITUTE OF TECHNOLOGY
Hyderabad - 500 075, Telangana, INDIA
www.cbit.ac.in

Code Optimization Classification

- By Scope:
  - Local Optimization: within a single basic block.
  - Peephole Optimization : on a window of instructions (usually local)
  - Loop-level Optimization : on one or more loops or loop nests.
  - Global: for an entire procedure
  - Interprocedural: across multiple procedures or whole program.
- By machine information used:
  - Machine-independent versus machine-dependent.
- By effect on program structure:
  - Algebraic transformations (e.g., `x+0`, `x*1`, `3*z*4`, …)
  - Reordering transformations (change the order of 2 computations)
    - Loop transformations: loop-level reordering transformations.

Global Optimization

- Global Optimization: across blocks
  - Common sub-expression elimination
  - Strength reduction
  - Data flow analysis

Data Flow Analysis

- Global Optimizations are based on data flow analysis
- These are algorithms to gather data about a program
- Specify some property that must hold every time an instruction is executed
- The analysis differ in the properties they compute
- Example: constant propagation
  - Computes for each point in the program
  - For each variable used by the program
  - Whether variable has a unique constant value at that point
  - Replace variable reference by a constant value

Data Flow Analysis

- Example: live variable analysis
  - for each point in the program
  - Whether the value of a variable is sure to be overwritten before it is read
  - If yes then there is no need to preserve the value

Data Flow Analysis

Reaching Definition:

- A definition `d` reaches a point `p`
  - If there is a path from `d` to `p`, and
  - `d` is not killed on the path

Available Expression:

- An expression `X op Y` is available at point `p`
  - If every path to `p` evaluates `X op Y`, and
  - After the last such evaluation no assignment is made to `X` or `Y`

Data Flow Equation

- Compiler needs to collect information about the program to do code optimization and code generation. This is called as data flow information, that an optimizing compiler collects by a process known as data flow analysis.
- Data flow information can be collected by setting up and solving systems of equations that relate information at various points in a program.

Data Flow Equation

A Typical Equation has the form:

1. $OUT[S] = GEN[S] \cup (IN[S] - KILL[S])$
2. $IN[S] = \bigcup_{P \text{ of } S} OUT[S]$

// P - predecessor (Incoming) of S

Such equations are called "**Data-flow equations**"

Data Flow Equation

Where

GEN[S]: set of definitions generated, those definitions within block 'S' that reach the end of the block.

KILL[S]: set of definitions killed, that is the set of definitions outside of the block 'S' that define identifiers that also have definitions within block 'S'.

IN[S]: set of input definitions. Consisting of all definitions reaching the point just before the first statement of the block 'S'.

OUT[S]: set of output definitions, that is the set of definitions reaching the point just after the last statement of S.

Data Flow Equation

Use-definition chain (ud-chain)

ud-chain answers the question: Given that identifier '**A**' is used at point '**P**' at what points could the value of '**A**' used at '**P**' have been defined?

Ex:

T_1 = 4 * I

T_2 = addr(A) - 4

$T_2[T_1]$ = B

If A = 3 then the flow graph is shown below:

Data Flow Equation

Use-definition chain (ud-chain)

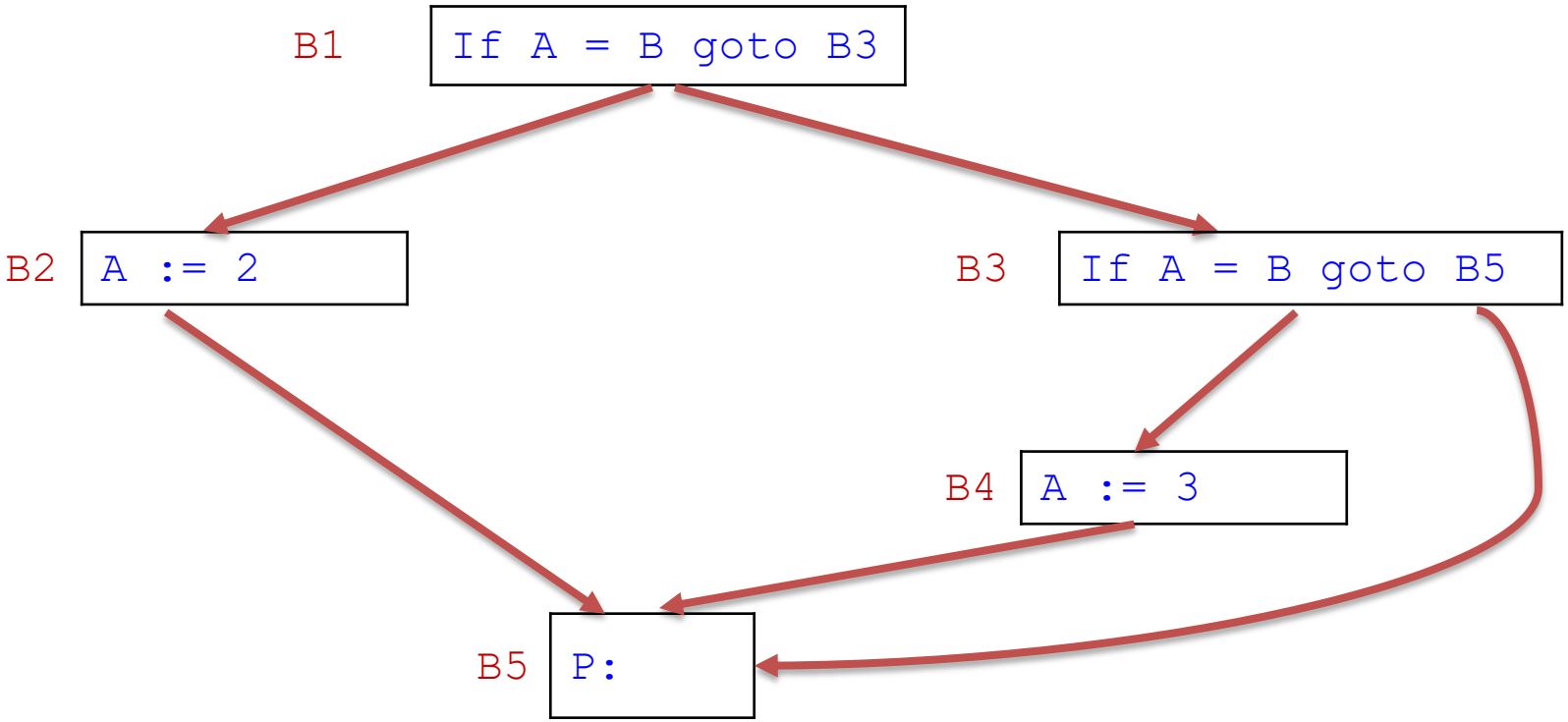

Fig: Flow graph

Solving Data Flow Equations

Block in a Flow graph

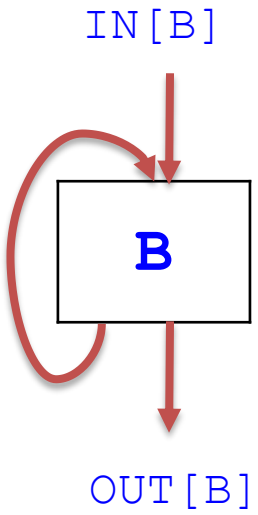

Fig: Flow graph

Solving Data Flow Equations

Ex: For the given flow graph below compute IN, OUT, KILL and GEN values by Solving Data flow equations

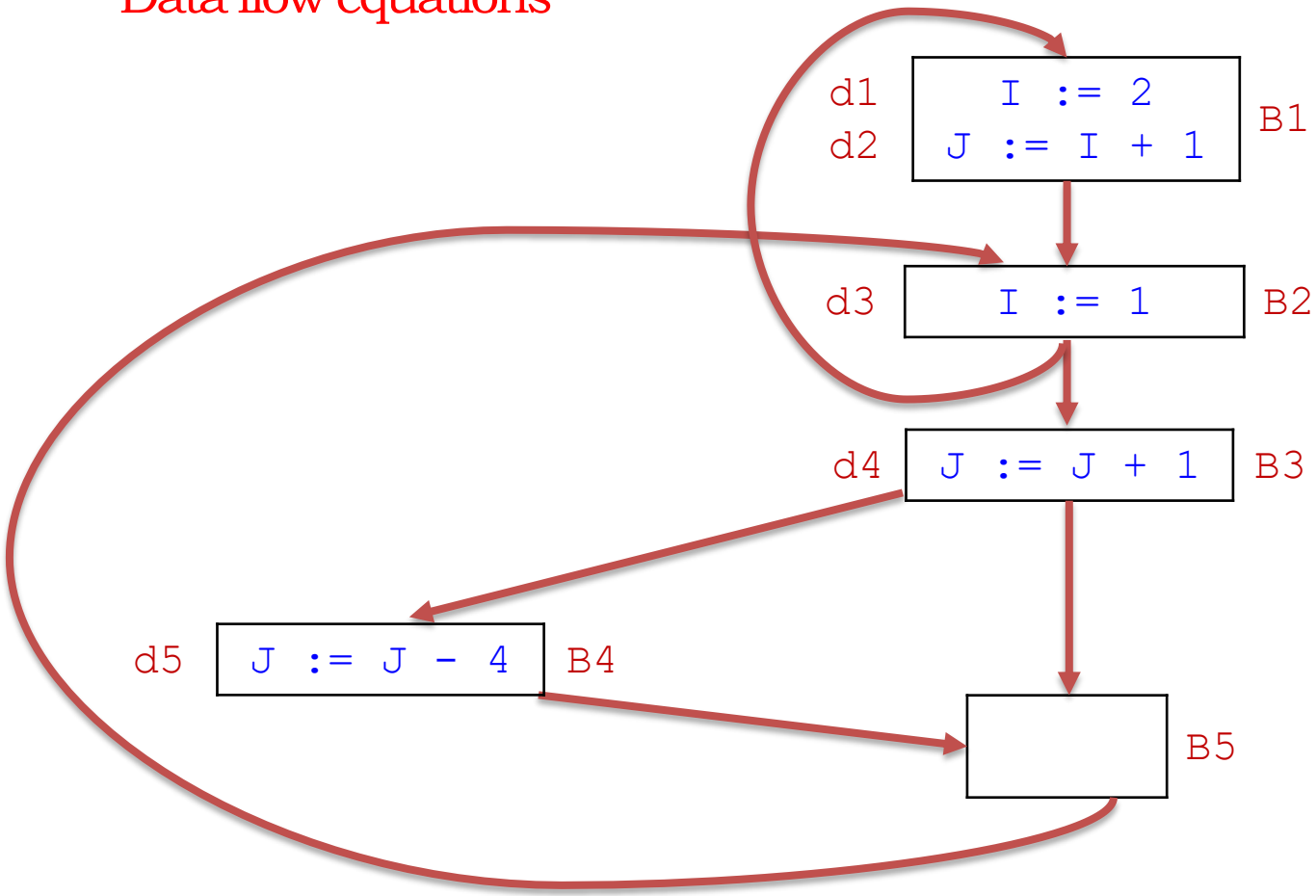

Fig: Flow graph

Solving Data Flow Equations

Ex: For the given flow graph below compute IN, OUT, KILL and GEN values by Solving Data flow equations

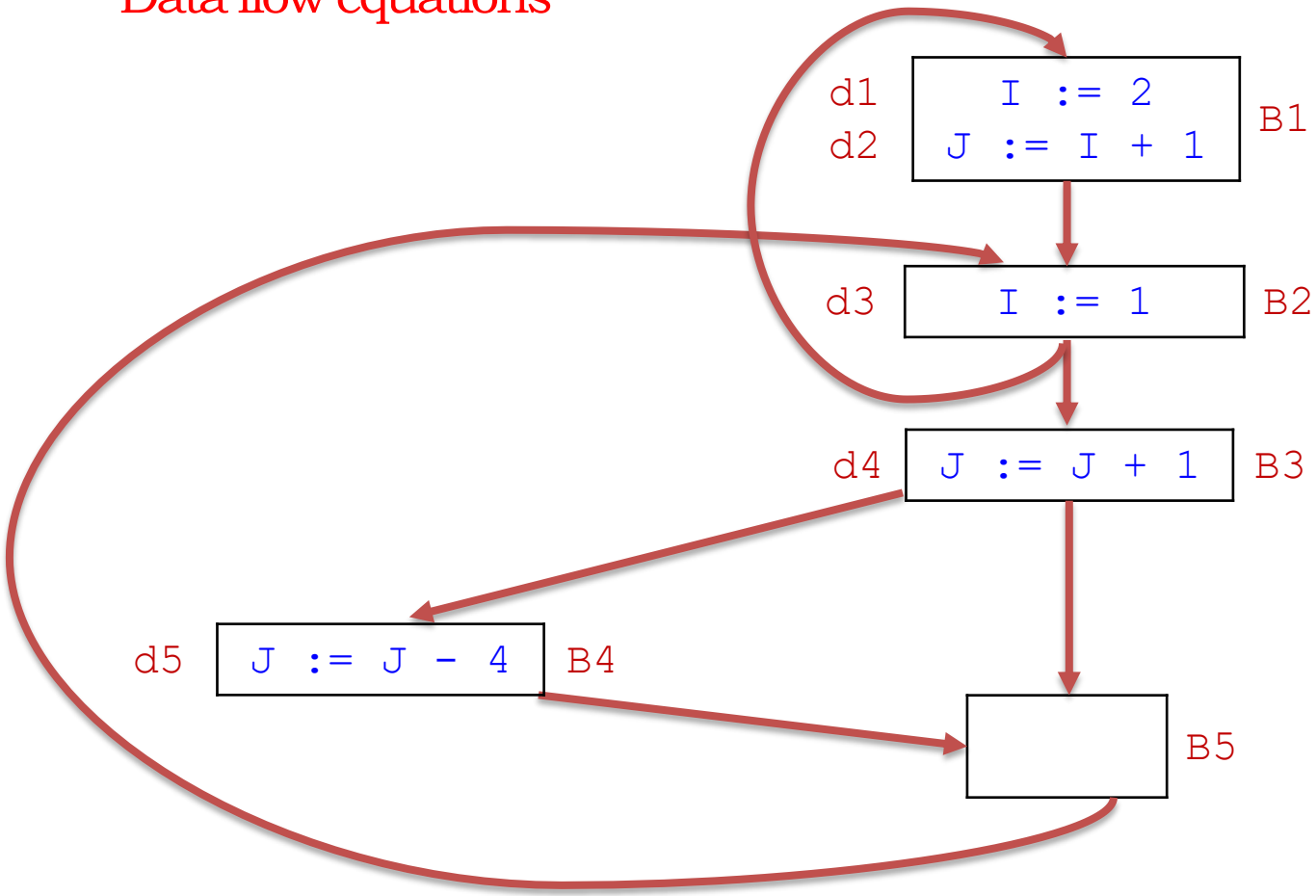

Fig: Flow graph

Computing GEN and KILL:

Block B1: in B1 both I and J are defined, so all the definitions of I and J outside of B1 are killed.

i.e. **KILL[B1] = {d3, d4, d5}**

GEN[B1] = {d1, d2}, since each of d1 and d2 are the last definitions of their respective variables in B1.

Solving Data Flow Equations

Ex: For the given flow graph below compute IN, OUT, KILL and GEN values by Solving Data flow equations

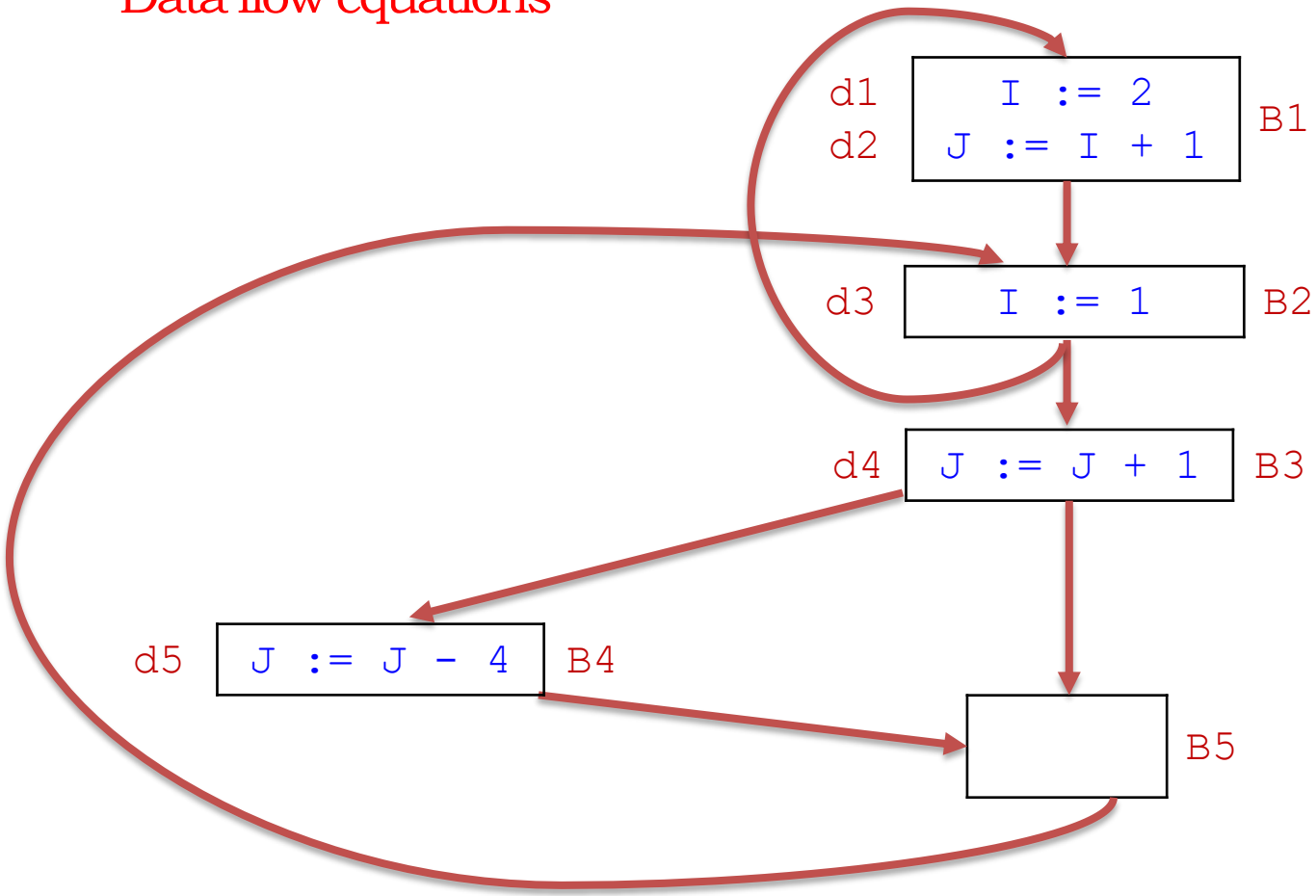

Fig: Flow graph

Computing GEN and KILL:

Block B2: in B2 I is defined, so all the definitions of I outside of B2 are killed.

i.e. **KILL[B2] = {d1}**

GEN[B2] = {d3}, since d3 is the last definition of their respective variables in B2.

Solving Data Flow Equations

Ex: For the given flow graph below compute IN, OUT, KILL and GEN values by Solving Data flow equations

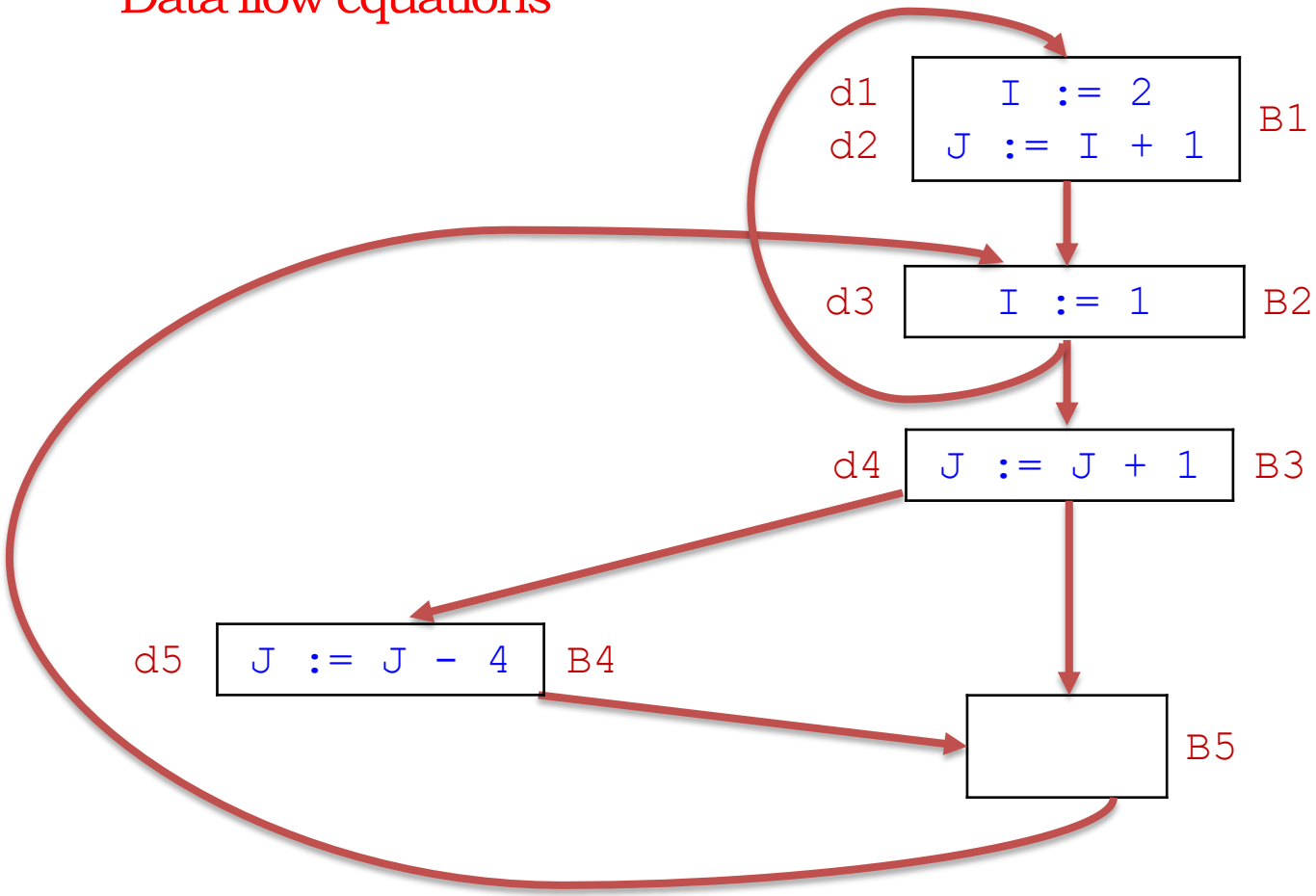

Fig: Flow graph

Computing GEN and KILL:

Block B3: in B3 J is defined, so all the definitions of J outside of B3 are killed.

i.e. **KILL[B3] = {d2, d5}**

GEN[B3] = {d4}, since d4 is the last definition of their respective variables in B3.

Solving Data Flow Equations

Ex: For the given flow graph below compute IN, OUT, KILL and GEN values by Solving Data flow equations

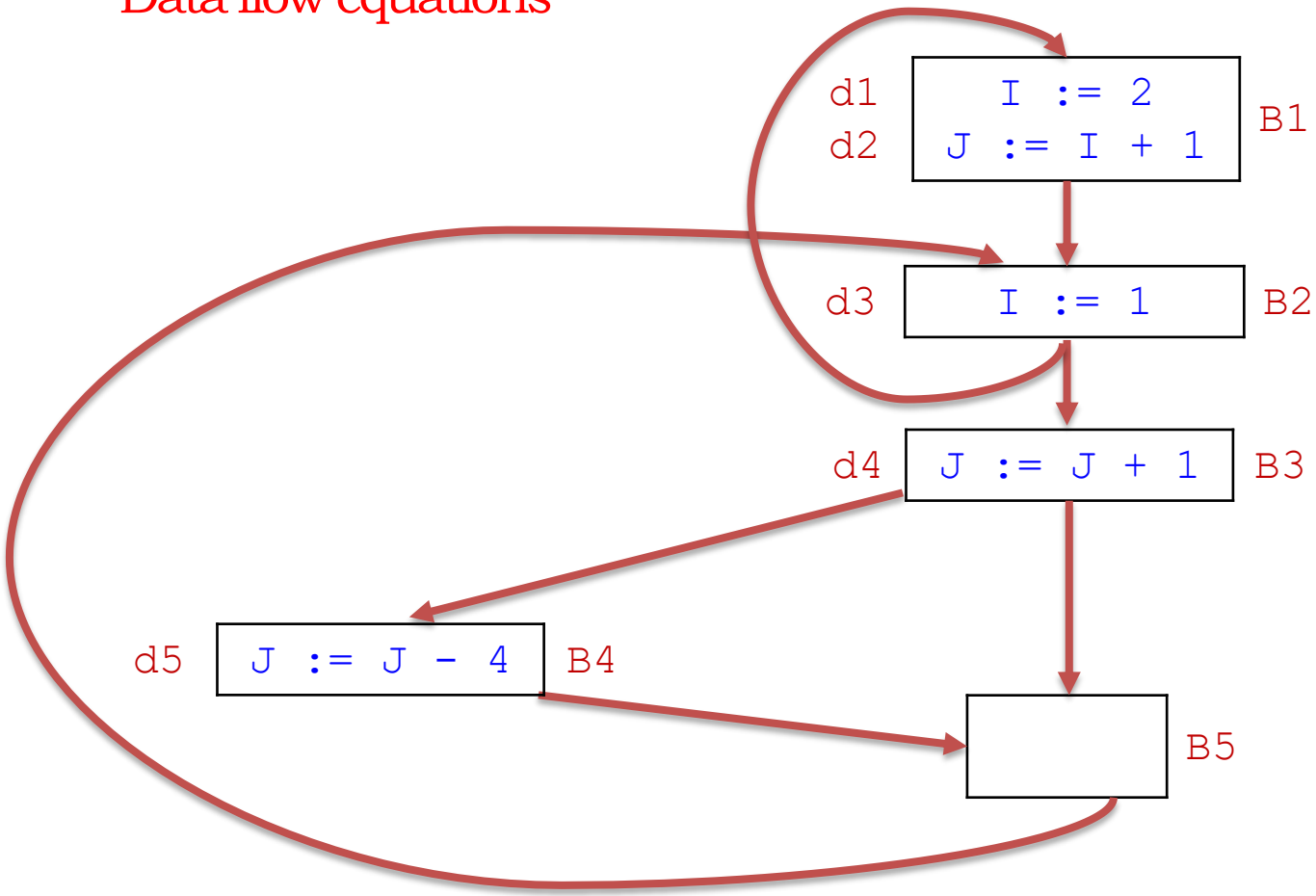

Fig: Flow graph

Computing GEN and KILL:

Block B4: in B4 J is defined, so all the definitions of J outside of B4 are killed.

i.e. **KILL[B4] = {d2, d4}**

GEN[B4] = {d5}, since d5 is the last definition of their respective variables in B4.

Solving Data Flow Equations

Ex: For the given flow graph below compute IN, OUT, KILL and GEN values by Solving Data flow equations

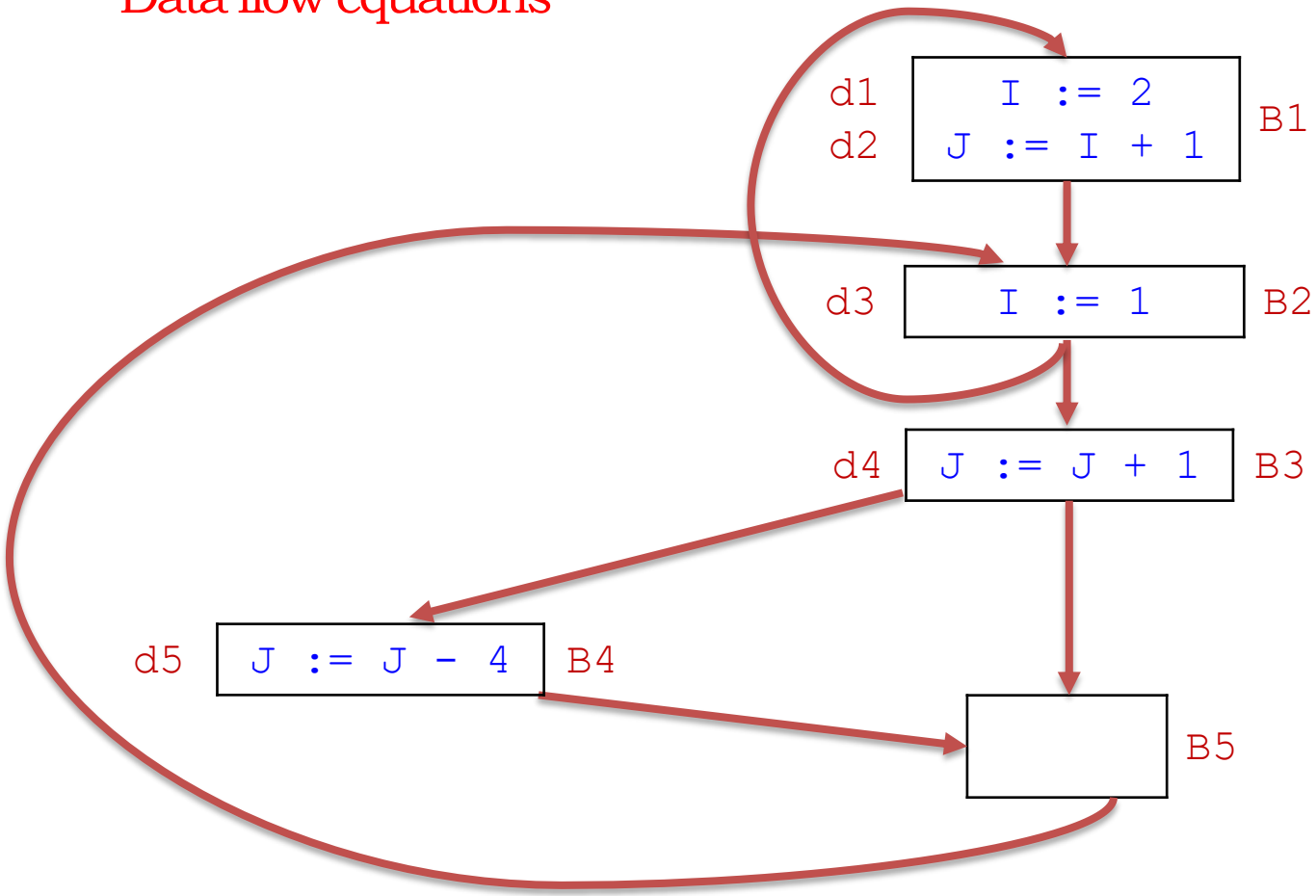

Fig: Flow graph

Computing GEN and KILL:

Block B5: in B5 nothing is defined, i.e. **KILL[B5] = Φ**

GEN[B5] = Φ, since no definitions are in B5.

Solving Data Flow Equations

Ex: For the given flow graph below compute IN, OUT, KILL and GEN values by Solving Data flow equations

The entire list of GEN and KILL values are shown below:

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

Solving Data Flow Equations

Ex: For the given flow graph below compute IN, OUT, KILL and GEN values by Solving Data flow equations

Computing **OUT** and **IN** values:

1. OUT[S] = GEN[S] U (IN[S] – KILL[S])
2. IN[S] = U OUT[S]

// P – predecessor (Incoming) of S

Here **S = {B1, B2, B3, B4, B5}**

Solving Data Flow Equations

Ex: For the given flow graph below compute IN, OUT, KILL and GEN values by Solving Data flow equations

Computing **OUT** and **IN** values:

Block B	IN [B]	Calculation	OUT [B]	Calculation
B1	00100	New IN[B1] = ∪ OUT[P of B1] = OUT[B2] = GEN[B2] = 00100	11000	**OUT[B1]** = GEN[B1] ∪ (IN[B1] – KILL[B1]) = **11000** + [00100 – 00111] [// Minus (**–**): **¬ ∧** 00100 – 00111 = **x = 00100** **y = ¬ of** 00111 = **11000** **x ∧ y = 00100 ∧ 11000 = 00000]** = 11000 + 00000 = 11000

Solving Data Flow Equations

Ex: For the given flow graph below compute IN, OUT, KILL and GEN values by Solving Data flow equations

Computing **OUT** and **IN** values:

Block B	IN [B]	Calculation	OUT [B]	Calculation
B2	11000	New IN[B2] = U OUT[P of B2] = OUT[B1] + OUT[B5] = OUT[B1] + GEN[B5] = 11000 + 00000 = 11000	01100	**OUT[B2]** = GEN[B2] U (IN[B2] – KILL[B2]) = 00100 + [11000 – 10000] = 00100 + 01000 = 01100
B3	01100	New IN[B3] = U OUT[P of B3] = OUT[B2] = 01100	00110	**OUT[B3]** = GEN[B3] U (IN[B3] – KILL[B3]) = 00010 + [01100 – 01001] = 00010 + 00100 = 00110

Solving Data Flow Equations

Ex: For the given flow graph below compute IN, OUT, KILL and GEN values by Solving Data flow equations

Computing **OUT** and **IN** values:

Block B	IN [B]	Calculation	OUT [B]	Calculation
B4	00110	New IN[B4] = U OUT[P of B4] = OUT[B3] = 00110	00101	**OUT[B4]** = GEN[B4] U (IN[B4] – KILL[B4]) = 00001 + [00110 – 01010] = 00001 + 00100 = 00101
B5	00111	New IN[B5] = U OUT[P of B5] = OUT[B3] + OUT[B4] = 00110 + 00101 = 00111	00111	**OUT[B5]** = GEN[B5] U (IN[B5] – KILL[B5]) = 00000 + [00111 – 00000] = 00000 + 00111 = 00111

Solving Data Flow Equations

Ex: For the given flow graph below compute IN, OUT, KILL and GEN values by Solving Data flow equations

Computing **OUT** and **IN** values:

	Initial		Pass1		Pass2		Pass3	
Block B	IN [B]	OUT [B]	IN [B]	OUT [B]	IN [B]	OUT [B]	IN [B]	OUT [B]
B1	00000	= GEN[B1] = 11000	00100	11000	01100	11000	01111	11000
B2	00000	= GEN[B2] = 00100	11000	01100	11111	01111	11111	01111
B3	00000	= GEN[B3] = 00010	01100	00110	01111	00110	01111	00110
B4	00000	= GEN[B4] = 00001	00110	00101	00110	00101	00110	00101
B5	00000	= GEN[B5] = 00000	00111	00111	00111	00111	00111	00111

Next Pass i.e. Pass4 contains the same values as in Pass3

Live Variable Analysis

Live Variable Analysis

- The variable **x** is live at the point **P**, if the value of **x** at **P** could be used along some path in the flow graph, starting at **P**; otherwise, **x** is dead at **P**.
- Sets of variables constitute the domain of data-flow values.
- Backward flow problem, with confluence operator **U**.
- **IN[B]** is the set of variables live at the beginning of **B**.
- **OUT[B]** is the set of variables live just after **B**.
- **DEF[B]** is the set of variables definitely assigned values in **B**, prior to any use of that variable in **B**.
- **USE[B]** is the set of variables whose values may be used in **B** prior to any definition of the variable.

Live Variable Analysis

Live Variable Analysis

OUT[B] = ∪ IN[S] // S is a successor (out edges) of B

IN[B] = USE[B] ∪ (OUT[B] - DEF[B])

IN[B] = Φ , for all B (initialization only)

Live Variable Analysis

Ex: Compute **USE**, **DEF**, **IN** and **OUT** values of blocks for the given flow graph

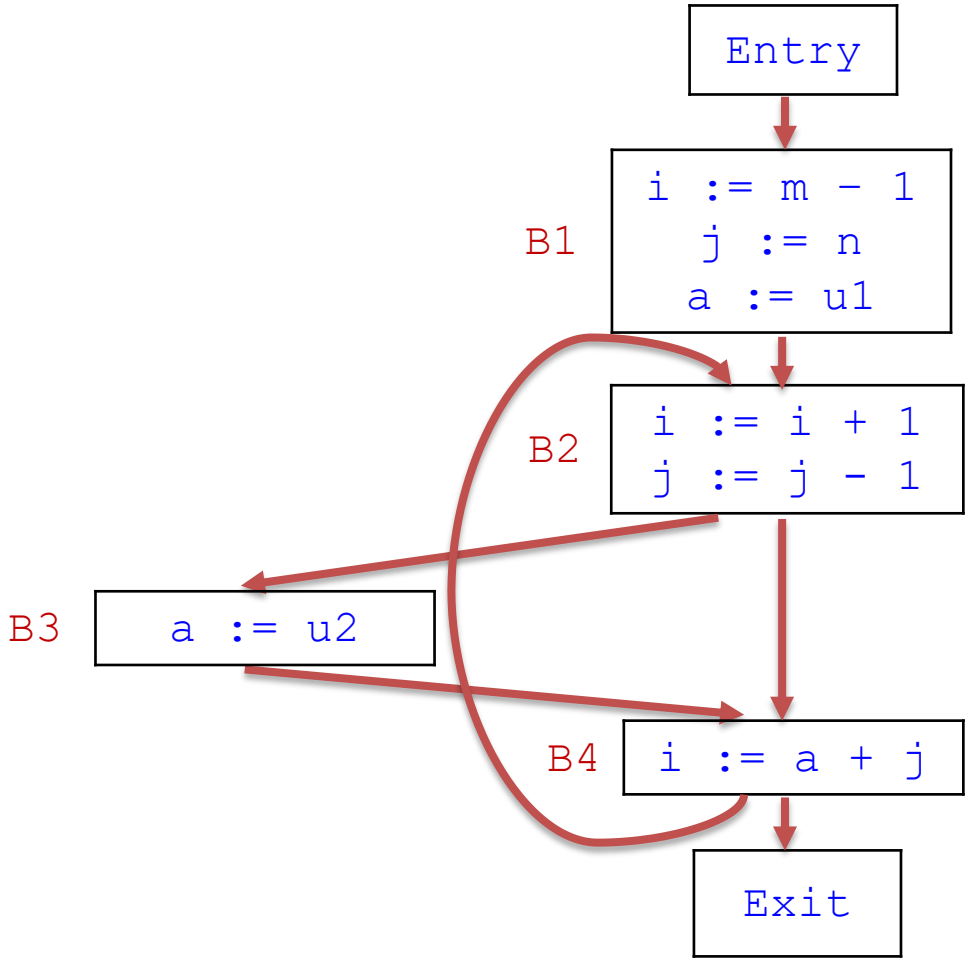

Fig: Flow graph

OUT[B] = ∪ IN[S] // S is successor of B

IN[B] = USE[B] ∪ (OUT[B] – DEF[B])

USE[B]: Set of variables used by S (a read of a variables value)

DEF[B]: Set of variables defined by S (an assignment of a value to a variable)

IN[S]: Variables live on entry to S

OUT[S]: Variables live on exit from S

IN[S] ⊇ USE[S]

Live Variable Analysis

Ex: Compute **USE**, **DEF**, **IN** and **OUT** values of blocks for the given flow graph

Computing **USE** and **DEF** values:

Block B	USE [B]	DEF[B]
B1	{m, n, u_1}	{i, j, a}
B2	{i, j}	{i, j}
B3	{u_2}	{a}
B4	{a, j}	{i}

Live Variable Analysis

Ex: Compute **USE**, **DEF**, **IN** and **OUT** values of blocks for the given flow graph

Computing **IN** and **OUT** values:

For B1:

OUT[B1] = ∪ IN[S] // S is successor(out edges) of B1

= IN[B2]

B1 → B2 → B4

B1 → B2 → B3 → B4

= (USE + DEF) of B2, B3, B4

= {i, j} ∪ {a, u2} ∪ {i, a, j} = {i, j, a, u2}

IN[B1] = USE[B1] ∪ (OUT[B1] - DEF[B1])

= {m, n, u1} ∪ ({i, j, a, u2} - {i, j, a})

= {m, n, u1} ∪ {a2} = {m, n, u1, u2}

Live Variable Analysis

Ex: Compute **USE**, **DEF**, **IN** and **OUT** values of blocks for the given flow graph

Computing **IN** and **OUT** values:

For B2:

OUT[B2] = ∪ IN[S] // S is successor(out edges) of B2

= IN[B3] ∪ IN[B4]

B2 → B4

B2 → B3 → B4

= (USE + DEF) of B3, B4

= {a, u2} ∪ {i, a, j} = {i, j, a, u2}

IN[B2] = USE[B2] ∪ (OUT[B2] - DEF[B2])

= {i, j} ∪ ({i, j, a, u2} - {i, j})

= {i,j} ∪ {a, u2} = {i,j, a, u2}

// same for B3 & B4

Live Variable Analysis

Ex: Compute **USE**, **DEF**, **IN** and **OUT** values of blocks for the given flow graph

Final values of **USE**, **DEF**, **OUT** and **IN**:

Block B	USE [B]	DEF[B]	OUT[B]	IN[B]
B1	{m, n, u_1}	{i, j, a}	{i, j, a, u2}	{m, n, u1, u2}
B2	{i, j}	{i, j}	{i, j, a, u2}	{i,j, a, u2}
B3	{u_2}	{a}	{a, j, u2}	{j, u2}
B4	{a, j}	{i}	{a, i, j, u2}	{a, j, u2}