UCS2702 COMPILER DESIGN

Unit 5

Code Optimization

Principal sources of optimization - DAG - Optimization of basic blocks - Global data flow analysis - Introduction to Low level Virtual Machine (LLVM) - Design of LLVM - Core libraries - Developing plugin in LLVM

* Introduction to Code Optimization - Basic Blocks

→ A basic block is a sequence of consecutive TAC statements in which:

(i) Control flow can only enter the block at the first statement

(ii) Control flow can only exit the block from the last statement

For eg - The TAC below forms a basic block

```
t1 := a * a
t2 := a * b
t3 := 2 * t2
t4 := t1 + t3
t5 := b * b
t6 := t4 + t5
```

More on code optimization

→ can be machine dependent or independent

→ can be applied at → source level
→ intermediate code level
→ target program level

→ Principle Sources of Optimization

1. Local Optimization
2. Global Optimization
3. Loop Optimization - Local

* Partition Algorithm for Basic Blocks

Input: A sequence of TAC statements

Output: A list of basic blocks with each TAC statement in exactly one block.

1. Determine the set of leaders

   ① The first statement is the leader

   ② Any statement that is the target of a goto is a leader

   ③ Any statement that immediately follows a goto is a leader

2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

Example1 — Partition into basic blocks

(1) prod = 0 ⟶ Leader
(2) i = 1
(3) t1 = 4 * i ⟶ Leader
(4) t2 = a[t1]
(5) t3 = 4 * i
(6) t4 = b[t3]
(7) t5 = t2 * t4
(8) t6 = prod + t5
(9) prod = t6
(10) t7 = i + 1
(11) i = t7
(12) if (i <= 20) goto (3)

B1 = (1) & (2)

B2 = (3) to (12)

Example: Partition the TAC into basic blocks

1) i = 1 ⟶ Leader
2) j = 1 ⟶ Leader
3) t1 = 10 * i ⟶ Leader
4) t2 = t1 + j
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a[t4] = 0.0
8) j = j + 1
9) if j <= 10 goto (3)
10) i = i + 1 ⟶ Leader
11) if i <= 10 goto (2)
12) i = 1 ⟶ Leader
13) t5 = i - 1 ⟶ Leader
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)

Ans

B1 = (1)

B2 = (2)

B3 = (3) - (9)

B4 = (10), (11)

B5 = (12)

B6 = (13) - (17)

* Transformations on Basic Blocks

→ A code-improving transformation is a code optimization to improve speed or reduce code size.

→ Global transformations are performed across basic blocks

→ Local Transformations are only performed on single basic blocks

→ Transformations must be safe and preserve the meaning of the code. - A local transformation is safe if the transformed basic block is guaranteed to be equivalent to its original form.

* Local Transformations → Algebraic Transformations
→ Structure Preserving Transformations
↓
(i) Common sub expression elimination
(ii) Dead code elimination
(iii) Renaming Temporary Variables
(iv) Interchange of Statements

A. Structure Preserving Transformations

(1) Common Sub-Expression Elimination

eg1

$$\begin{aligned} a &= b+c \\ b &= a-d \\ c &= b+c \\ d &= a-d \end{aligned} \quad \Rightarrow \quad \begin{aligned} a &= b+c \\ b &= a-d \\ c &= b+c \\ d &= b \end{aligned}$$

eg2

$$\begin{aligned} t_1 &= b*c \\ t_2 &= a-t_1 \\ t_3 &= b*c \\ t_4 &= t_2+t_3 \end{aligned} \quad \Rightarrow \quad \begin{aligned} t_1 &= b*c \\ t_2 &= a-t_1 \\ t_4 &= t_2+t_1 \end{aligned}$$

(2) Dead Code Elimination

eg1 assume a is unused

```
b = a+1                    b = a+1
a = b+c        =>
```

eg2

```
if true goto L2
b = x+y
```

} remove unreachable code

(3) Renaming Temporary Variables

→ Temporary variables that are dead at the end of a block can be safely renamed.

eg1

```
t1 = b+c                   t1 = b+c
t2 = a-t1                  t2 = a-t1
t1 = t1 * d        =>      t3 = t1 * d      change t1 to t3
d = t2 + t1                d = t2+t3        so that each var
                                            uniquely represents
                                            one computation
```

(4) Interchange of Statements

→ Independent statements can be reordered

```
t1 = b+c                   t1 = b+c
t2 = a-t1                  t3 = t1 * d
t3 = t1 * d        =>      t2 = a-t1
d = t2+t3                  d = t2+t3
```

B. **Algebraic Transformations**

↳ includes constant folding

→ change arithmetic operations to transform blocks into algebraic equivalent forms

$t_1 = a - a$ ⇒ $t_1 = 0$

$t_2 = b + t_1$ ⇒ $t_2 = b$

$t_3 = 2 * t_2$ ⇒ $t_3 = t_2 << 1$

(Left shift by 1 means mul by 2)

→ Also eliminate :

$x = x + 0$ ⇒ x

$x = x * 1$ ⇒ x

→ Modify $x = 4 ** 2$ ⇒ $x = 4 * 4$

* DAG and Loop Optimization

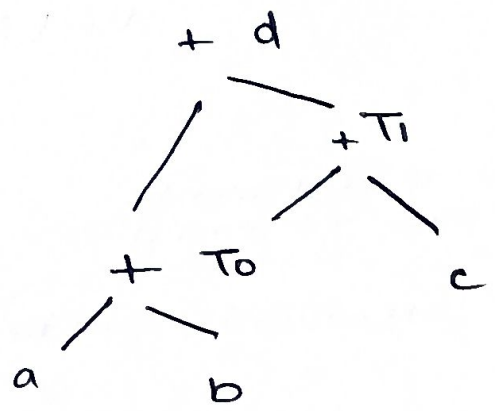
* DAG for Basic Blocks

eg1 $T_0 = a + b$

$T_1 = T_0 + c$

$d = T_0 + T_1$

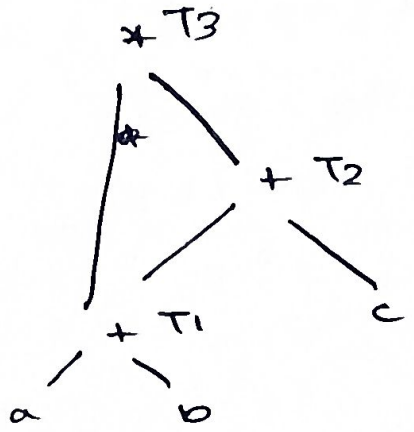
→



eg2 $T_1 = a + b$

$T_2 = T_1 + c$

$T_3 = T_1 * T_2$



eg3 Simplify the given TAC using a DAG

(1) $t1 = 4 * i$

$t2 = a[t1]$

$t3 = 4 * i$

$t4 = b[t3]$

$t5 = t2 * t4$

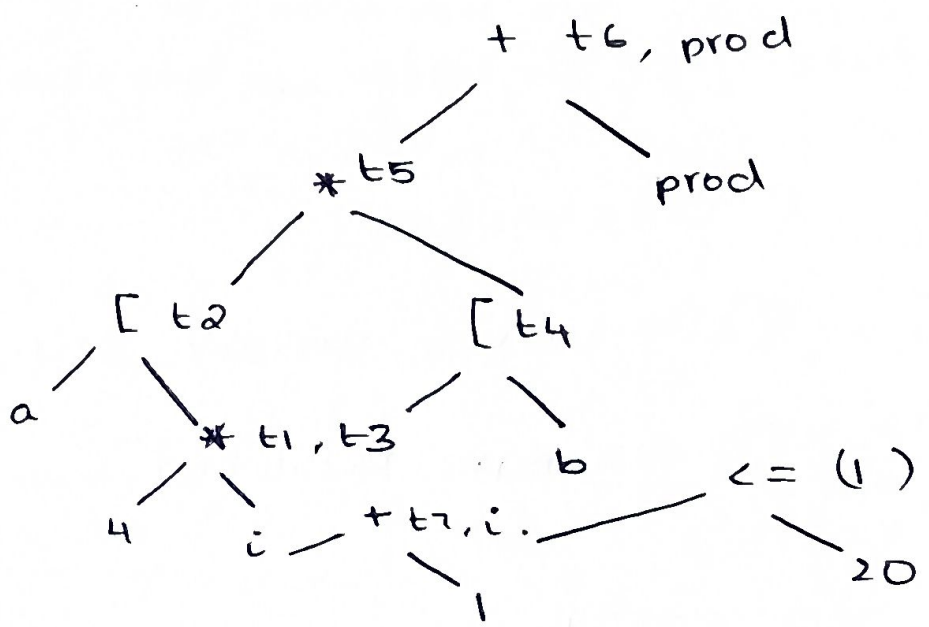
$t6 = prod + t5$

$prod = t6$

$t7 = i + 1$

$i = t7$

if $i <= 20$ goto (1)



replacements to make

replace

t6 with prod

t3 with t1

t7 with i

eliminate

t6 = prod → } no need to

i = t7 → } double-assign, use i & prod

t3 = 4 * i → redundant

$t1 = 4 * i$

$t2 = a[t1]$

$t4 = b[t1]$

$t5 = t2 * t4$

$prod = prod + 5$

$i = i + 1$

if $i <= 20$ goto (1)

* Loop Optimization

→ The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside the loop.

→ 3 techniques for loop optimization are:

① Code motion

② Induction variable elimination

③ Strength Reduction

A. Code Motion

→ This transformation takes an expression that yields the same result independent of the no. of times a loop is executed, and places the expression before the loop.

For eg.

```
while (i<=n-2)
        {
          }
```

Evaluation of n-2 is loop independent. Place it outside

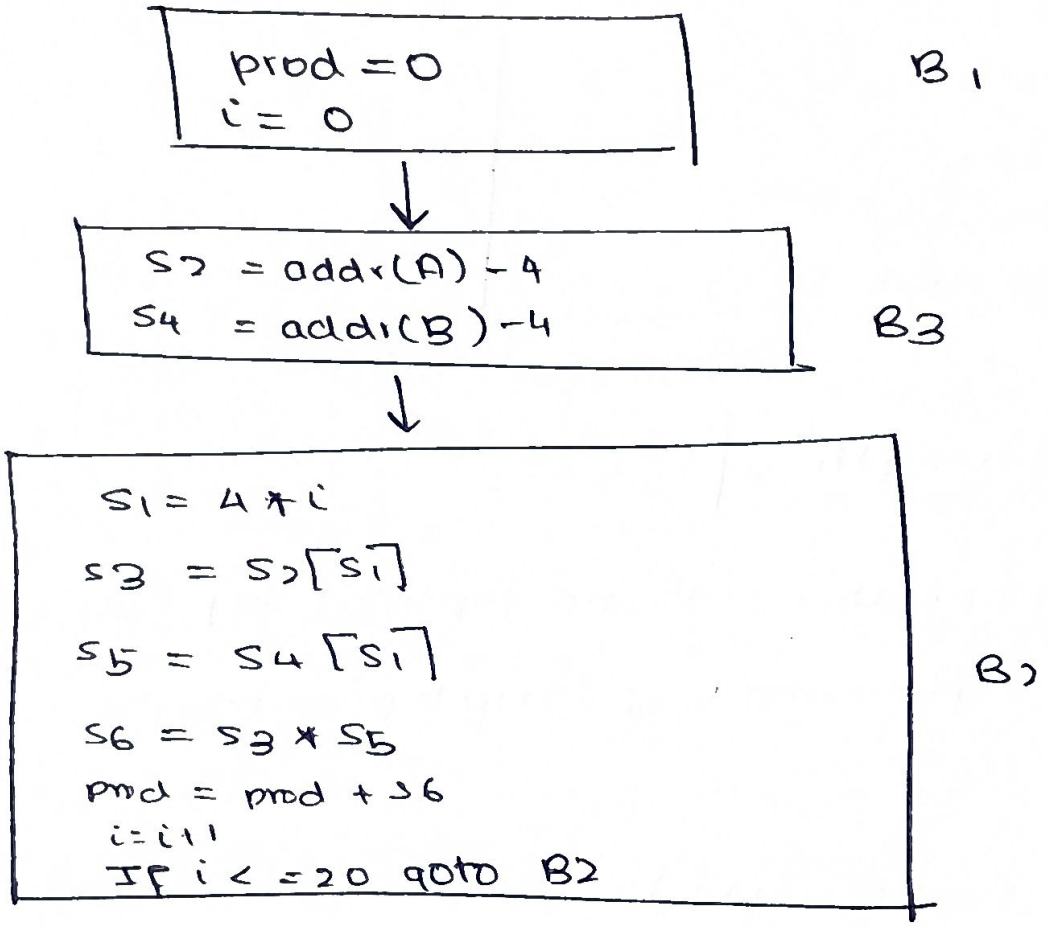
```
t = n-2
while ( i <= t) {

}
```

B. Induction Variable Elimination

→ An induction variable is one that changes in a regular patten (eg. increments / decrements in each iteration)

→ Observe patterns where the induction var is used and eliminate it entirely, by replacing its use with an alternate computation

Consider the following example:



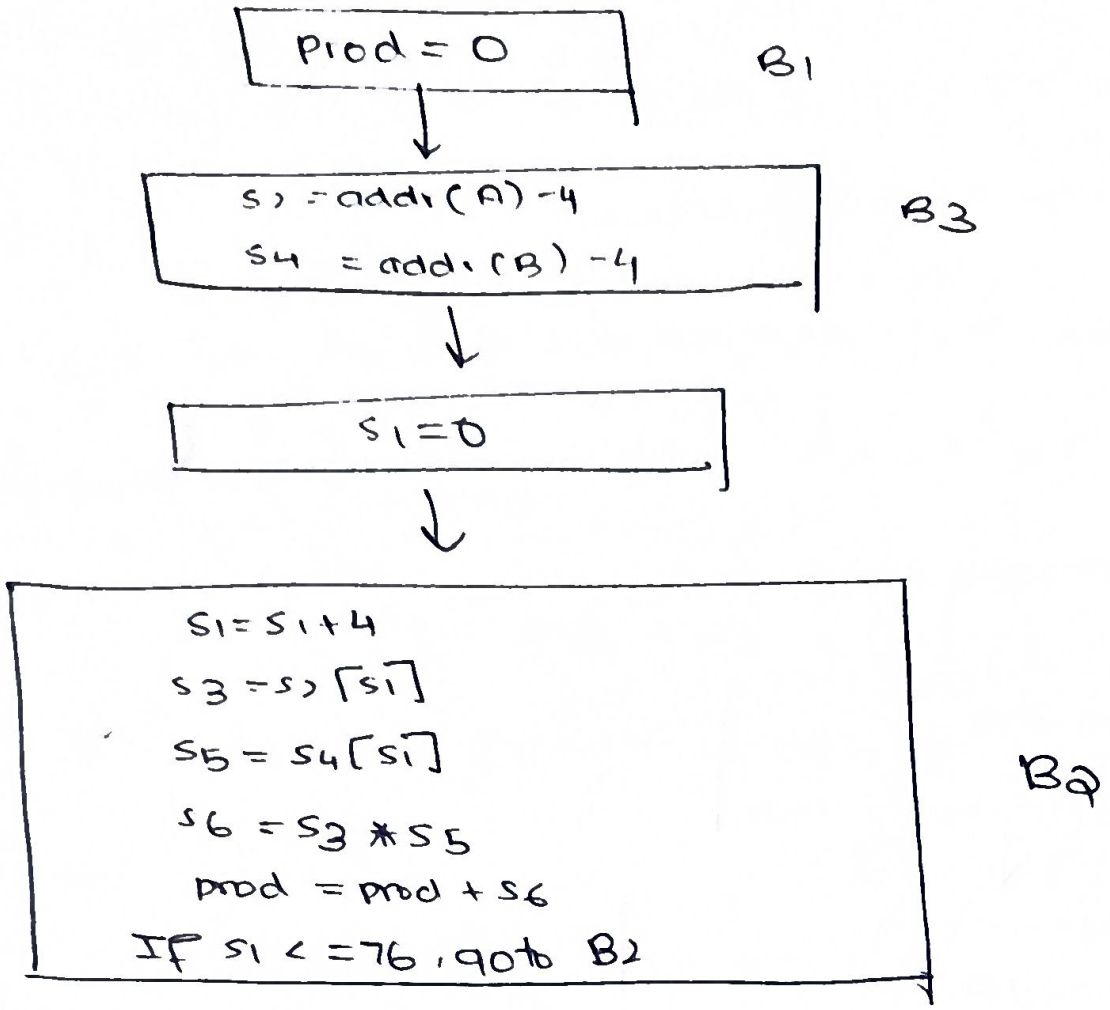
In this code, i takes the value 1, 2 20

⇒ s1 would take the values 4, 8, ... 80

⇒ Replace 4*i and i = i+1 with s1 = s1 + 4

In the if block, i<=20 can be replaced with s1 <= 76 (after calculating value)

∴ The new code block would be:



c. **Reduction in Strength**

→ Higher strength operators can be replaced by lower strength operators - eg. replacement of multiplication by repeated addition

For eg

```
For (i=1, i<=50, i++)
{
    c = i*7
}
```

replace as

```
t=7
For (i=1, i<=50, i++)
{
    c=t
    t = t+7
}
```

* Peephole Optimization

→ Peephole optimization is a technique for improving code by examining small, localized sections of target code and replacing inefficient or redundant instructions with more efficient ones.

→ A small moving 'peephole' is used to analyze and optimize sequences of instructions

→ Limited to short code sequences rather than the entire program

Techniques include → Eliminate redundant loads & store
→ Eliminate unreachable code
→ Flow of control optimization
→ simplify algebraic expressions
→ Strength Reduction

A. Eliminate redundant loads & stores

consider:

(1) mov R0, a
(2) mov a, R0

(2) can be removed if it does not have a label

B. Eliminate unreachable code

Consider the following

```
#define debug 0
if (debug) {
        print( )
}       goto L2
        print( )
```

The print statements are inaccessible

C. **Flow of Control Optimization**

Consider :

```
Goto L1
  :
L1 : Goto L2
```

If under all cases, L2 follows L1, remove one goto

make it goto L2

D. **Simplify algebraic expressions**

$x := x + 0 \Rightarrow x$

$x := x \times 1 \Rightarrow x$

E. **Strength Reduction**

(i) $X * 2 \rightarrow X << 1$

(ii) $X / 2 \rightarrow X >> 1$

(iii) ADD #1, R $\rightarrow$ INCR

(iv) $X \wedge 2 \rightarrow X * X$

Example - Identify the code optimization techniques and optimize the code given below.

1. $A = 2 * (22.0 / 7.0) * r$

$\rightarrow$ constant folding

$A = 6.28 * r$

2. $c = a * b$

$x = a$

$d = a^x * b + 4$

$\rightarrow$ Redundant code elimination

$c = a * b$

~~$x = a$~~

$d = a * b + 4$

3. a = 200

```
while (a>0) {
        b = x+4,
        if (a%. b = =0) {
                printf (" %.d", a),
        }
}
```

→ Loop optimization - code motion

→ move b = x+4 outside loop

4. c = a * b

x = a

d = a * b + 4

→ common subexpression elimination

d = c + 4

5. i = 1

```
while(i<100) {
        y = i * 4
}
```

→ Induction variable elimination

```
y = 4
while (y < 400) {
        y = y * 4
}
```

* Global Flow Data Analysis

→ A technique to optimize code by analyze the flow of data across the entire program.

→ Specifically it:

(i) Tracks where variables are defined and used throughout the program

(ii) Identifies relationships between definitions and the corresponding uses (called Use-Definition (Ud) chains)

(iii) Helps eliminate redundant or unnecessary computations

→ For Ud-chaining Reaching definitions should be found

→ Then data-flow equations have to be determined iteratively

Reaching Definition → A definition of a variable is said to 'reach' a point in the program if there exists a path from the definition to that point, such that the value assigned to the variable is still valid, and not redefined along the way.

Algorithm

1. Assign a number to each equation
2. Compute each of the following for each block 'B':
   A. GEN[B] - set of generated definitions within basic block B that reach the end of the block.

B. KILL(B) - set of definitions outside of B that have a definition within B

C. Use bit vectors for representation

3. Compute IN[B] for all blocks B - all definitions reaching the point just before the first statement of Block B

4. Compute OUT[B] for all blocks B - the set of definitions reaching the point just after the last statement of B

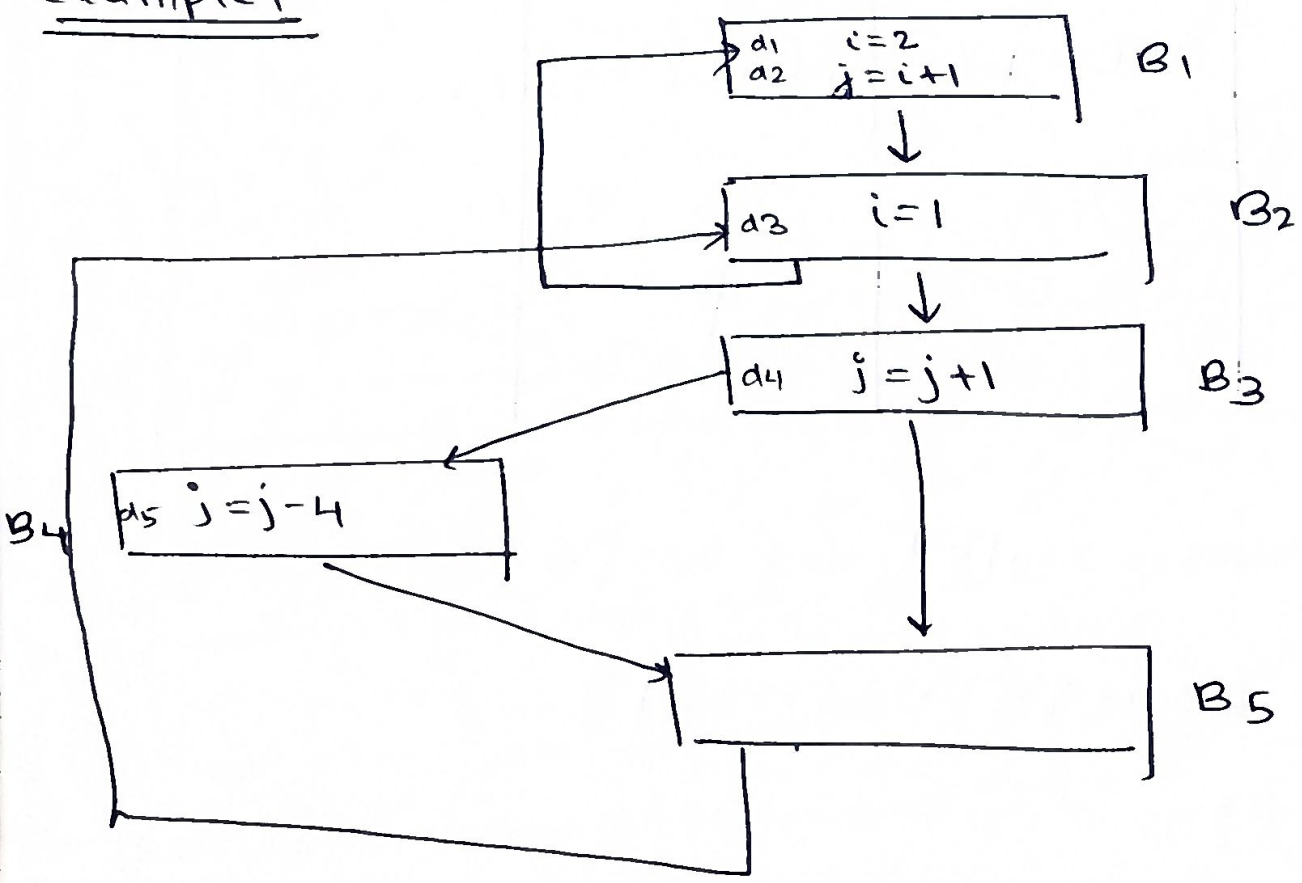
5. Repeat 3,4,B iteratively until convergence.

The data flow equations to compute IN and OUT are:

$$OUT[B] = IN[B] - (KILL[B] \cup GEN[B])$$

$$IN[B] = \cup\ OUT[P], \quad P \text{ is a predecessor}$$

Example1



Ans Step1 : Assign numbers to the definitions

$d_1 \rightarrow i = 2$

$d_2 \rightarrow j = i + 1$

$d_3 \rightarrow i = 1$

$d_4 \rightarrow j = j + 1$

$d_5 \rightarrow j = j - 4$

write predecessor

$P(B_1) = B_2$

$P(B_2) = B_1, B_5$

$P(B_3) = B_2$

$P(B_4) = B_3$

$P(B_5) = B_3, B_4$

Step2 Compute GEN and KILL for each block

	GEN[B]	Bit Vector	KILL[B]	Bit Vector
B_1	$\{d_1, d_2\}$	11000	$\{d_3, d_4, d_5\}$	00111
B_2	$\{d_3\}$	00100	$\{d_1\}$	10000
B_3	$\{d_4\}$	00010	$\{d_2, d_5\}$	01001
B_4	$\{d_5\}$	00001	$\{d_2, d_4\}$	01010
B_5	ϕ	00000	ϕ	00000

Step3 : Iteratively compute IN[B] and OUT[B]

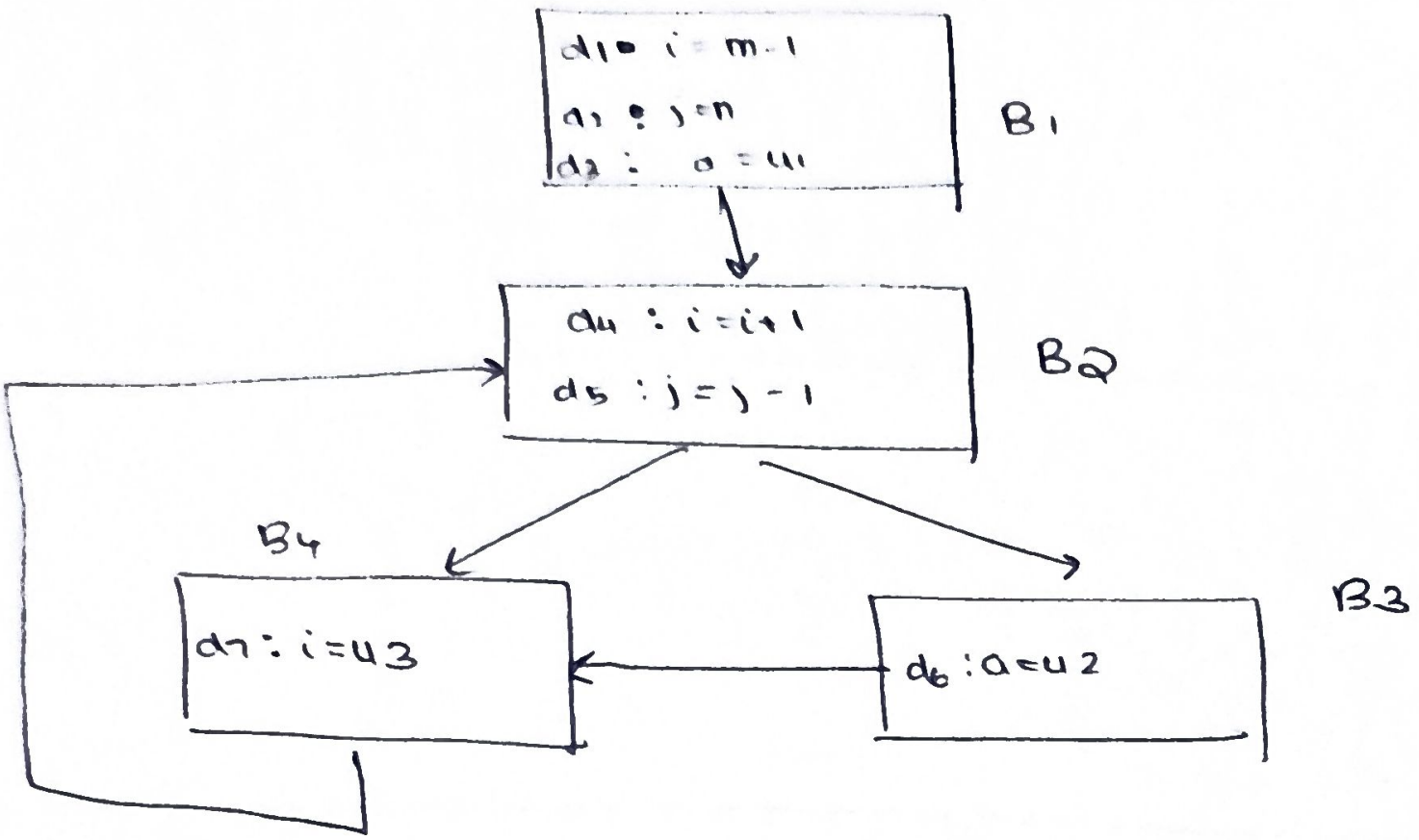
$OUT[B] = IN[B] - \{KILL[B] \cup GEN[B]\}$

$IN[B] = \cup\ OUT[P]$

→ remove definitions from IN that are in KILL

Block	IN	OUT initially	IN	OUT	IN	OUT
B_1	ϕ	same as GEN $\{d_1, d_2\}$	OUT[B_2] $= \{d_3\}$	IN[B_1] = (NULL[B] ∪ GEN[B]) $\{d_3\} - \{d_3, d_4, d_5\} + \{d_1, d_2\}$ $\boxed{\{d_1, d_2\}}$	$\{d_2, d_3\}$	$\{d_2, d_3\} - \{d_3, d_4, d_5\} + \{d_1, d_2\}$ $\boxed{\{d_1, d_2\}}$
B_2	ϕ	$\{d_3\}$	OUT[B_1] $= \{d_1, d_2\}$	$\{d_1, d_2\} - \{d_1\} + \{d_3\} = \boxed{\{d_2, d_3\}}$	$\{d_1, d_2, d_3, d_4, d_5\}$	$\{d_1, d_2, d_3, d_4, d_5\} - \{d_1\} + \{d_3\}$ $\boxed{\{d_2, d_3, d_4, d_5\}}$
B_3	ϕ	$\{d_4\}$	OUT[B_2] $= \{d_2, d_3\}$	$\{d_2, d_3\} - \{d_2, d_5\} + \{d_4\}$ $\{d_3\} + \{d_4\} = \boxed{\{d_3, d_4\}}$	$\{d_2, d_3\}$ *	$\{d_2, d_3\} - \{d_2, d_5\} + \{d_4\}$ $\boxed{\{d_3, d_4\}}$
B_4	ϕ	$\{d_5\}$	OUT[B_3] $= \{d_3, d_4\}$	$\{d_3, d_4\} - \{d_2, d_4\} + \{d_5\} = \boxed{\{d_3, d_5\}}$	$\{d_3, d_4\}$	$\{d_3, d_4\} - \{d_2, d_4\} + \{d_5\}$ $\boxed{\{d_3, d_5\}}$
B_5	ϕ	$\{\phi\}$	$\{d_3, d_5\}$ d_4	$\{d_3, d_5\} - \{\phi\} + \{\phi\}$ $\boxed{\{d_3, d_4, d_5\}}$	$\{d_3, d_4, d_5\}$	$\boxed{\{d_3, d_4, d_5\}}$ → if you calc again, you'll see it doesn't change

Example : Analyze the data flow for the following example.



Ans Step 1 Compute GEN & KILL

	GEN [B]	Bit Vector	Kill [B]	Bit Vecto
B_1	{d1, d2, d3}	1110000	{d4, d5, d6, d7}	0000111
B_2	{d4, d5}	0001100	{d1, d2, d7}	1100001
B_3	{d6}	0000010	{d3}	0010000
B_4	{d7}	0000001	{d1, d4}	1001000

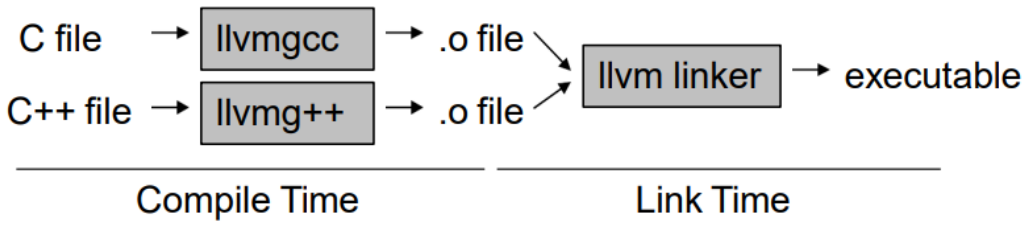
1. What is LLVM?

- LLVM stands for **Low-Level Virtual Machine**. It is a **compiler infrastructure** that provides reusable libraries to build compilers.
- Key Features:
    - **Implemented in C++** for flexibility and speed.
    - Supports **multiple front-ends** (e.g., C, C++, ADA) and **back-ends** (e.g., ARM, x86, MIPS).
    - Produces an **intermediate representation (IR)** that is language and target independent.
    - Open-source, first released in 2003, and widely adopted for various compiler-related tasks.

Illustration:

Think of LLVM as a toolkit to:

- Transform programming languages into an optimized intermediate form (LLVM IR).
- Convert this optimized form into executable machine code for specific architectures.



---

2. LLVM Compiler System

The system is divided into:

- **LLVM Compiler Infrastructure**:
    - Reusable components for **building compilers**.
    - Reduces time/cost by avoiding building everything from scratch.
    - Builds static compilers, Just-In-Time (JIT) compilers, and optimizers.
- **LLVM Compiler Framework**:
    - Creates end-to-end compilers (e.g., compilers for C and C++).
    - Supports both native (machine code) and intermediate code generation.

---

3. Three Primary Components

1. **LLVM Virtual Instruction Set**:
    - Centralized **Intermediate Representation (IR)** that is independent of programming languages and hardware.
    - Represents code in a **persistent, target-independent format**.
2. **Well-Integrated Libraries**:
    - Include libraries for **code generation**, optimization, debugging, profiling, garbage collection, etc.
3. **Tools Built from Libraries**:
    - Assemblers, debuggers, linkers, code generators, etc., simplify compiler development.

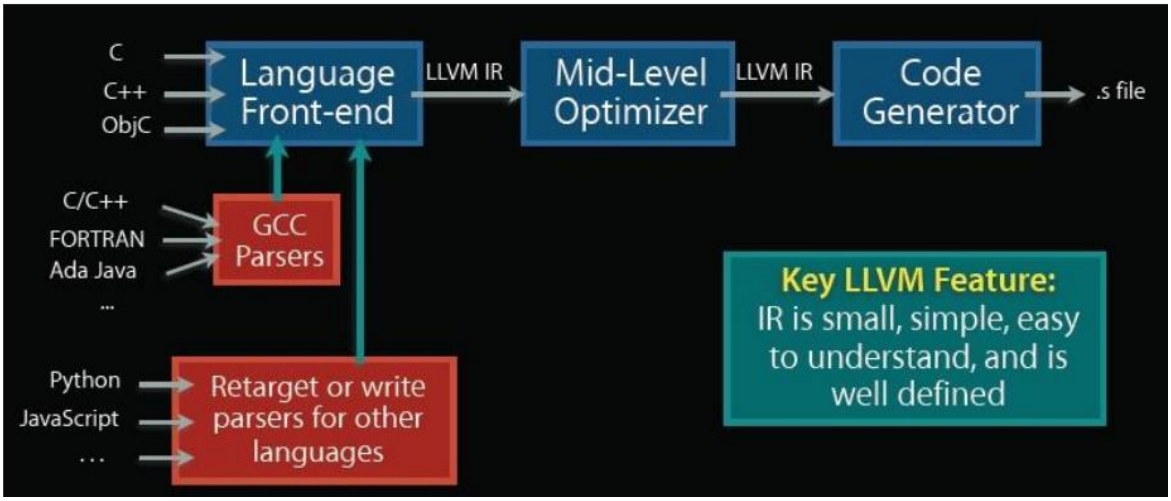
---

4. LLVM C/C++ Compiler

- LLVM provides **clang** (C/C++ front-end compiler) that:
    - Uses LLVM IR as an intermediate step.
    - Translates source code into **.o files** containing IR or machine code.
    - Supports GCC makefiles and uses GCC parsers (e.g., gcc 3.4).

Compile Time Workflow:

1. C/C++ source code → LLVM front-end (e.g., llvmgcc).
2. IR passes through multiple **optimizers**.
3. Resulting optimized IR is linked by llvm-linker into:
    - Executable machine code (native) or bytecode (for JIT execution).



---

5. Compiler Organization with LLVM IR

- The **compiler pipeline**:
    - Language-specific **front-end** converts source code to LLVM IR.
    - **Mid-level optimizers** improve LLVM IR using optimizations like inlining, dead code elimination, etc.
    - **Code generators** create target-specific machine code.

---

6. Compile-Time Events

- Key Steps:
    - **Front-End Parsing**:
        - Converts source code into Abstract Syntax Tree (AST) or LLVM IR.
        - Example: C files use cc1, while C++ files use cc1plus.
    - **Optimization**:
        - Runs **analysis and transformation passes** to optimize code.
        - Example passes: Dead Argument Elimination, Loop Invariant Code Motion (LICM), and Memory Promotion.
    - **IR Verification**:
        - Verifies correctness and validity of LLVM IR before code generation.

---

7. Link-Time Events

- During linking:
    - The **LLVM linker** combines .o files and applies final optimizations.
    - **Native code generators** (e.g., llc) convert LLVM IR to machine code.
    - Optionally, the code can remain in LLVM IR or bytecode format for further JIT compilation.