Code Optimization

Code optimization

Machine independent

1. Loop optimization :
   - 1 Code motion or freq. reduction
   - 2 Loop unrolling
   - 3 Loop jamming
2. Folding
3. Redundancy elimination
4. Strength reduction
5. Algebraic simplification

machine dependent

1. Register allocation
2. Use of addressing modes
3. Peephole optimization :
   - 1 Redundant LOAD / STORE
   - 2 Flow of control optimization
   - 3 Use of machine idioms

- **Basic Blocks :**

> A basic block is a sequence of 3-address statements where control enters at the beginning and leaves ONLY at the end WITHOUT ANY JUMPS OR HALTS

- **Steps to construct basic blocks :**

  - 1 Find leaders in the basic block
  - 2 A basic block will start will start from one leader to the next leader but DOES NOT include the next leader

• Steps to find leaders in a basic block :

- 1 The first 3-address instruction in the IC is the leader
- 2 Any instruction that is a target of conditional or unconditional jump is a leader
- 3 Any instruction that immediately follows a conditional or unconditional jump is a leader.

> Have a START & END block compulsorily
> AND INCLUDE THESE NODES AND ITS EDGES
> While counting no. of nodes and edges

Eg. ①

☆ L_1 : f = 1

L_2 : i = 2

☆ L_3 : if (i > n) goto L_8 •

☆ L_4 : t_1 = f * i

L_5 : f = t_1

L_6 : i = i + 1

L_7 : goto L_3 •

☆ L_8 : goto calling function

☆ are leaders

Basic Blocks :

B_1

☆ L_1 : f = 1

L_2 : i = 2

B_2

☆ L_3 : if (i > n) goto L_8 •

B_3

☆ L_4 : t_1 = f * i

L_5 : f = t_1

L_6 : i = i + 1

L_7 : goto L_3 •

B_4

☆ L_8 : goto calling function

Control Flow Graph :

(2)

☆ L_1 : i = 1

☆ L_2 : j = 1

☆ L_3 : t_1 = 5 * i

L_4 : t_2 = t_1 + j

L_5 : t_3 = 4 * t_2

L_6 : t_4 = t_3

L_7 : a[t_4] = -1

L_8 : j = j + 1

L_9 : if (j ≤ 3) goto L_3

☆ L_{10} : i = i + 1

L_{11} : if (i < 5) goto L_2

Basic Blocks :

B_1 ☆ L_1 : i = 1

B_2 ☆ L_2 : j = 1

B_3

☆ L_3 : t_1 = 5 * i

L_4 : t_2 = t_1 + j

L_5 : t_3 = 4 * t_2

L_6 : t_4 = t_3

L_7 : a[t_4] = -1

L_8 : j = j + 1

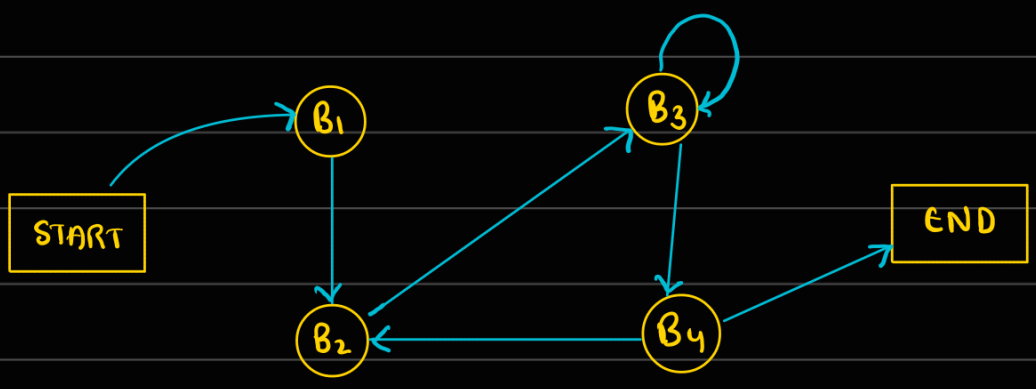
L_9 : if (j ≤ 3) goto L_3

B_4

☆ L_{10} : i = i + 1

L_{11} : if (i < 5) goto L_2

Control Flow Graph :



Peephole Optimization

- Perform on a small set of instructions (peephole / window)

- **Redundant LOAD / STORE optimization :**

Eg. `a = b + c`
`d = a + e`

⇉

```
LOAD R0, b
ADD R0, c
STORE a, R0
LOAD R0, a
ADD R0, e
STORE d, R0
```

= ADD R0, R0, c meaning, add the contents of R0 and c and store it back in R0

Is redundant & can be removed

- **Flow of Control Optimization :**

  - 1. Dead code elimination :

Eg.
```
int i = 0;
if (i == 1) { ... }
else <statements>
```
⇒
```
int i = 0;
<statements>
```

  - 2. Avoid jumps on jumps :

Eg.
```
L1 : jump L2
L2 : jump L3
L3 : jump L4
L4 : x = a + b * c
```
⇒
```
L1 : jump L4
L2 : jump L4
L3 : jump L4
L4 : x = a + b * c
```

- **Use of machine idioms :**

Eg. i = i+1 :

```
LOAD R0, i
ADD R0, #1      ⇒ INC i
STORE i, R0
```

[constants are represented using a hash symbol (here)]

Machine Independent Optimization

- **Loop Optimizations :**

  - First, detect loops (in IC). To do so, we must use control flow analysis using program flow graph (PFG) or control flow graph (CFG)

- **Code motion (freq. reduction) :**

  - A statement / expr. that can be moved outside the loop body without affecting the semantic of the program
  - Moving from high freq. region (inside loop body) to low freq. regio (outside loop body) is called freq. reduction / code motion.

Eg.

```
i=0                          i=0 ; t = a/b
while (i<1000) {             while (i < 1000) {
  x = (a/b) + i      ⇒         x = t + i
  i++                          i++
}                            }
```

• 2 Loop Unrolling :

- Reducing the no. of times comparisons are made in the loop.

Eg.
```
for (i=0 ; i<10 ; i++) {
    print("a")
}
```
⇒
```
for (i=0 ; i<10 ; i+=2) {
    print("a")
    print("a")
}
```

[we could also have i<5 & i++ instead]

• 3 Loop Jamming :

- Combine the bodies of 2 loops :

Eg.
```
for (i=0 ; i<10 ; i++) {
    a = i+5
}
for (i=0 ; i<10 ; i++) {
    b = i+10
}
```
⇒
```
for (i=0 ; i<10 ; i++) {
    a = i+5
    b = i+10
}
```

• Constant Folding :

- Replacing an expr. that can be computed at COMPILE TIME by its value

Eg. ① $C = 2 \times 3.14 \times r \Rightarrow C = 6.28 \times r$

② $2 + 5 + a \Rightarrow 7 + a$

- **Strength Reduction :**

  - Replacing an expensive operator by a cheaper one.

  Eg. ① $x^2 \Rightarrow x * x$

  ② $2 * x \Rightarrow x + x$

  ③ $x/2 \Rightarrow x * 0.5$ OR $x >> 1$

  ④ $x = y * 2 \Rightarrow x = y << 1$

- **Algebraic Simplification :**

  Eg. ① $x + 0$ OR $0 + x$ OR $x - 0 \Rightarrow x$

  ② $x / 1$ OR $x * 1$ OR $1 * x \Rightarrow x$

- **Common Sub-Expression Elimination (CSEE) :**

  - Done using a DAG
  - Are of 2 types : •1 Local CSEE : • CSEE done within a basic blocks
    •2 Global CSEE : • CSEE done across many basic blocks

If we were to perform CSEE on say basic block B_1 and do it across say B_4, B_5 and B_7, the value of the CSEE MUST NOT CHANGE ALONG ANY PATHS FROM START NODE TO DESTINATION NODE

i.e. value of CSE in B_1, B_4, B_5 & B_7 can be eliminated $\Leftrightarrow$ the value of CSE DOES NOT CHANGE for all paths from B_1 to B_4, all paths from B_1 to B_5 and all paths from B_1 to B_7 RESPECTIVELY (obviously, because if the value of CSE did change along some path, it would not be a CSE)

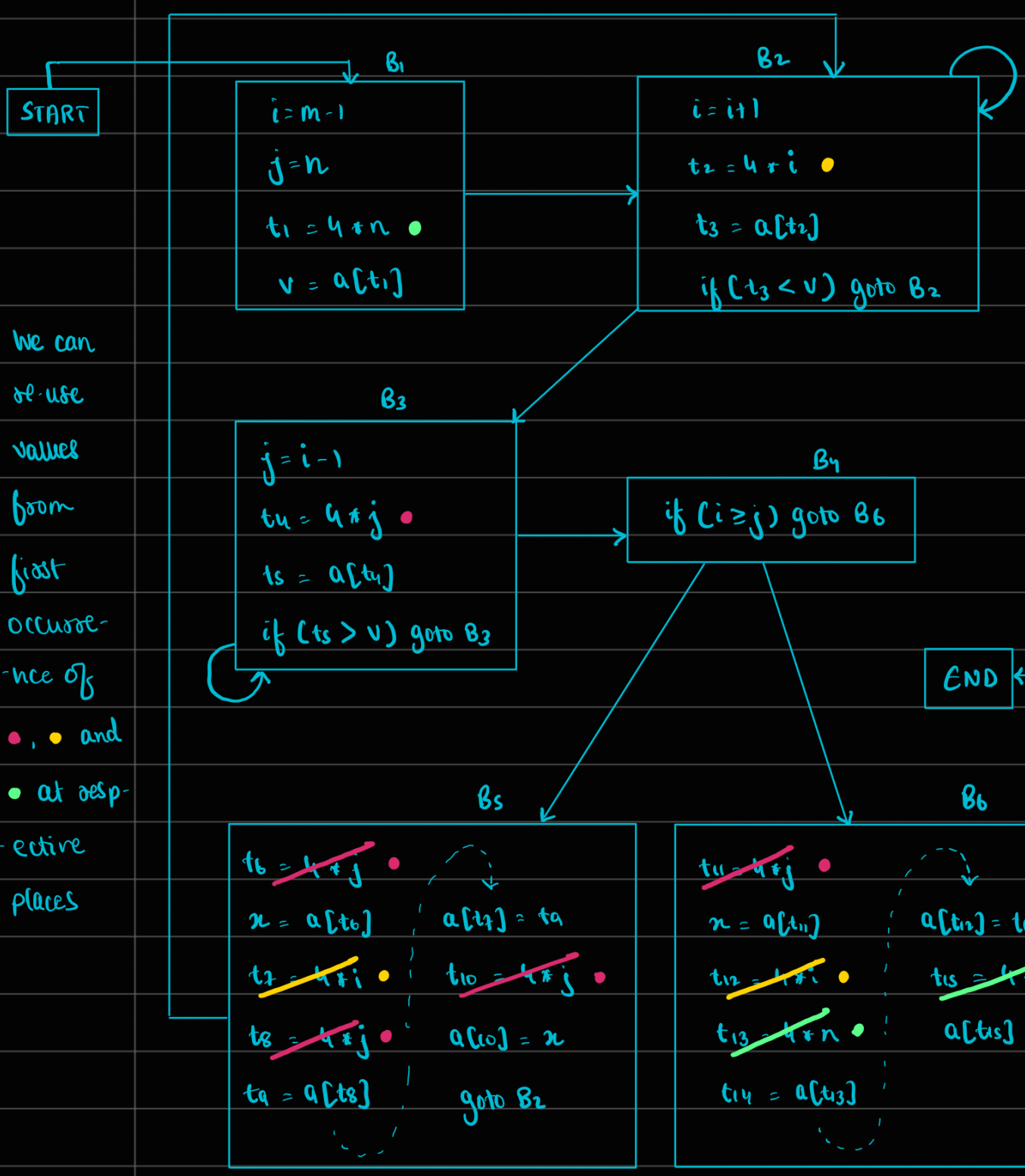
Eg. ① Basic Block :

$a = x+y+z$
$r = p+q$
$b = x+y+r$

$\Rightarrow$

$t = x+y$
$a = t+z$
$r = p+q$
$b = t+r$

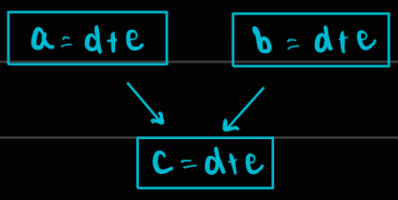
② Control flow Graph for quicksort :



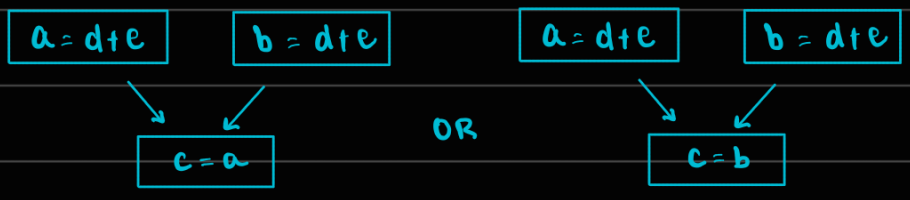
We can re-use values from first occurrence of •, • and • at respective places

- **Copy Propagation:**

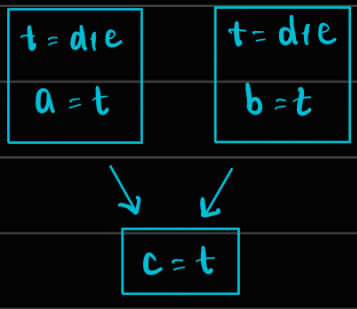
Eg.



INCORRECT :



CORRECT :



Reason: Since control may reach c = d+e either after the assignment to a or after the assignment to b, it would BE INCORRECT to replace c with either a or b.

Liveness Analysis

- Purpose : Assigning multiple variables to single register without changing the program behaviour i.e. for register allocation

x is a live statement s: if and only if :

- There is a statement S_j using x (reading)
- There is a path from S_i to S_j and (reachability)
- There is no new definition of x BEFORE S_j INCLUDING S_j and AT OR AFTER THE CURRENT STATEMENT

If there are multiple paths, for a variable to be alive, it MUST BE ALIVE IN ATLEAST 1 OF THE PATHS from S_i to S_j

Conversely, for a variable to be dead, IT MUST BE DEAD IN ALL PATHS FROM S_i to S_j (obviously)

(Statements can also mean blocks here)

Small thing to note:

In a compiler, this line-by-line liveness analysis is not performed instead it does liveness analysis on a basic block basis:

- variables that are live while coming into the basic block are called LIVE IN
- variables that are live while going out of the basic block are called LIVE OUT

Eg. ① L_1 : $x = a+b$

L_2 : $y = d+c$

L_3 : $x = x+b$

L_4 : $a = b+d$

L_5 : $x = a+b+c$

While evaluating a L_i, we consider everything

	a	b	c	d	x	y
L_1	L	L	L	L	D	D
L_2	D	L	L	L	L	D
L_3	D	L	L	L	L	D
L_4	D	L	L	L	D	D
L_5	L	L	L	D	D	D

L : Live

D : Dead

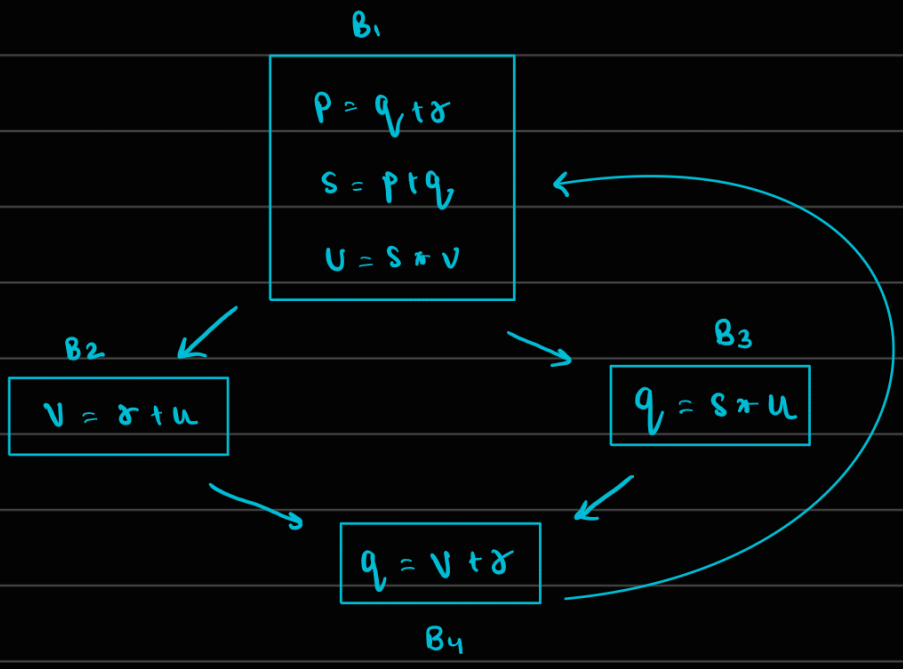
Explanation:

(L_1, d) : Dead since L_4 initializes x before L_3 (where it is read)

(L_2, a) : Dead since L_4 initializes a before L_5 (where it is read)

(L_2, x) : Live. We look at lines before L_3 (we don't consider L_3 even though there is an assignment due to •3). We see L_4 defines x BUT WE ARE CURRENTLY AT L_2. SO, WE DO NOT CONSIDER THE LINES BEFORE THE CURRENT STATEMENT. So, we do not consider L_1 as an initialization (obviously since we are past that)

② The no. of common live variables in basic block B_2 and B_3 are ? (GATE 2015)



	p	q	r	s	u	v
B_2	D	D	L	D	L	D
B_3	D	D	L	L	L	L

$\Rightarrow$ Answer = 2

Explanation :

- For B_2 row :

  L_1 : First statement in B_2
  L_2 : " " in B_4
  L_3 : " " in B_1
  L_4 : 2nd " " in B_1
  L_5 : 3rd " " in B_1

- For B_3 row :

  L_1 : First statement in B_3
  L_2 : " " in B_4
  L_3 : " " in B_1
  L_4 : 2nd " " in B_1
  L_5 : 3rd " " in B_1

Now do normal liveness analysis

PTO

③

L_1 : $u = 0$

L_2 : $v = u + 1$

L_3 : $w = w + v$

L_4 : $u = v * 3$

L_5 : if $(u < 10)$ goto L_1

L_6 : return w

	u	v	w
B_1	D	D	L
B_2	L	D	L
B_3	D	L	L
B_4	D	L	L
B_5	L	D	L
B_6	D	D	L

Explanation:

- (B_1, w) : The statement $w = w + 1$ is actually executed as temp = w+1 then w = temp. So in cases like these, the variable is USED FIRST and only then DEFINED LATER
- (B_4, w) : Here both the paths : $B_4\ B_5 \to B_6$ and $B_4 \to B_5 \to B_1 \to B_2 \to B_3$ has a live w (although only 1 path was enough)

④ Assuming all operations take their operands from registers. what is the minimum no. of registers needed to execute this program without spilling? (GATE 2010)

L_1: $a = 1$ L_2: $b = 10$ L_3: $c = 20$ L_4: $d = a + b$ L_5: $e = c + d$ L_6: $f = c + e$ L_7: $b = c + e$ L_8: $e = b + f$ L_9: $a = 5 + e$ L_{10}: return (d + f)

A) 2 B) 3 C) 4 D) 6

Solution: No spilling means NO VARIABLES ARE STORED IN MEMORY.
All the variables are stored in registers ONLY

L_1: a = 1 → R_0

L_2: b = 10 → R_1

L_3: c = 20 → R_2

L_4: d = a(R_0) + b(R_1) → R_0 (since a is dead at L_4)

L_5: e = c(R_2) + d(R_1) → R_0 (since d is dead at L_5) ☆

L_6: f = c(R_2) + e(R_0) → R_1 (since b is dead at L_6)

L_7: b = c(R_2) + e(R_1) → R_0 (since e is dead at L_7)

L_8: e = b(R_0) + f(R_1) → R_1 (since b is dead at L_8)

L_9: d = s(R_2) + e(R_1) → R_1 (since e is dead at L_9)

↳ (since c is dead at L_9)

L_{10}: return (d(R_1) + f(R_2)) → ANY [since we are not storing anything]

☆ we could have also used R_1 since b is also dead at L_4