* Code optimization -
  - faster Execution
  - Space Saving
  - Efficiency ↑, performance ↑

Machine Independent

1) Loop optimization
   (a) Code motion or frequency reduction
   (b) Loop Unrolling
   (c) Loop Jamming
   (d) Loop fusion
2) folding
3) Redundancy elimination
4) Strength Reduction
5) Algebric simplification

Machine Dependent

1) Register Allocation
2) Use of Addressing modes
3) Peephole optimization
   a) Redundant Load/store
   b) flow of Control optimization
   c) Use of machine idioms

⇒ To Apply Loop optimization, we must first detect Loops
↓
We use Control flow analysis using Program flow graph.
↓
To find PFG, We need to find Basic Blocks.

- Basic Blocks- (We need to find leaders in the Three Address Code)
  ↳ A basic block will start from one leader to the next leader but not including next leader.

finding leaders-

1.) The first three Address ~~Code~~ Instrⁿ in the intermediate Code is a leader.
2) Any instrⁿ that is the target of Conditional and unConditional jump is a leader.
3) Any instrⁿ that immediately follows a Conditional or unConditional jump is a leader.

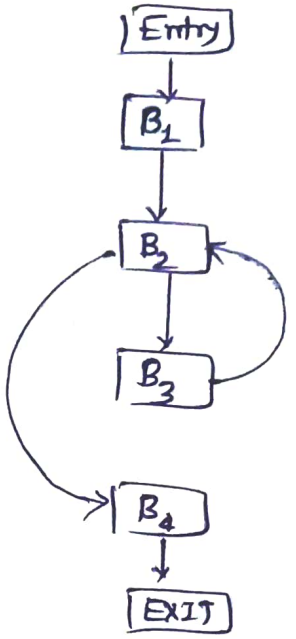
Exa-

B_1
* 1) $f = 1$
2) $i = 2$

B_2
* 3) if $(i > n)$ goto 8

B_3
* 4) $t_1 = f * i$
5) $f = t_1$
5) $i = i+1$
6) goto 3

B_4
* 8) goto Calling fn / Exit



cycle detected
↓
Loop ✓

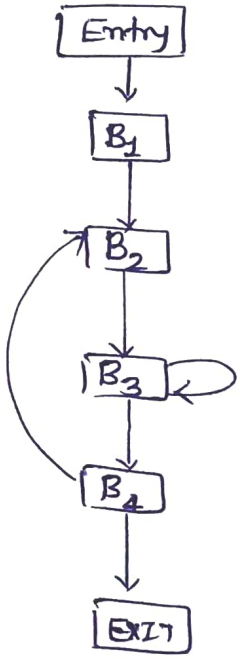
Exa-

B_1
* 1) $i = 1$

B_2
* 2) $j = 1$

B_3
* 3) $t_1 = 5 * i$
4) $t_2 = t_1 + j$
5) $t_3 = 4 * t_2$
6) $t_4 = t_3$
7) $a[t_4] = -1$
8) $j = j+1$
9) if $(j <= 5)$ goto 3

B_4
* 10) $i = i+1$
11) if $(i < 6)$ goto 2



- Loop optimization -

a) Code motion (frequency reduction)

A statement or expression which can be moved outside the loop body without affecting the semantic of the program.

Eg-

```
i=0
While (i<1000)
{
   A = (a/b) + i
   i++
}
```

⟹

```
i = 0
t = a/b
While (i<1000)
{
  A = t + i
  i++
}
```

b) Loop Unrolling -

Reducing the No. of times comparisons are made in the loop

Eg- ~~for~~ for (i=0, i<10; i++)

```
{
  Printf ("Hi")
}
```

⇓

```
for (i=0; i<5; i++)
{
   Printf ("Hi").
   Printf ("Hi");
}
```

c) Loop Jamming -

↳ Combines the body of two loops.

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

⟹

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

d) Removal of induction variables -

if the value of any variable in any loop gets changed every time, then such a variable is known as an induction variable.

```
for(i=0 ; i<n ; i++)
{
   x = 4 * i;
   a[x] = 0;
}
```

⟶

```
x = 0
for (i=0; i<n ; i++)
{
   a[x] = 0;
   x = x+4;
}
```

Multiplication is Costly

Compared to addition

- **Constant folding :-**

  ↳ Replacing an expression that can be computed at compile time by its value.

  Exa- $C = 2 * 3.14 * r \Rightarrow C = 6.28 * r$

  $2 + 3 + B + C \longrightarrow 5 + B + C$

- **Redundancy Elimination -**

  $a = b + c$

  $e = d + b + c \longrightarrow e = d + a$

- **Strength Reduction -**

  ↳ Replacing a expensive operator by cheaper one.

Expensive	Cheaper
x^2	$x * x$
$2 * x$	$x + x$
$x/2$	$x * 0.5$
$B = A * 2$	$B = (A << 1)$

- **Algebric simplification -**

  $x + 0 = 0 + x = x$, $x - 0 = x$

  $x * 1 = 1 * x = x$, $x/1 = x$

* **peephole optimization -**

↳ Perform on a small set of instruction. (peephole or window)

(A) **Redundant Load / Store Elimination -**

$a = b + c$

$d = a + e$

```
Load R0, b
ADD R0, C
STORE a, R0   | X Redundant
Load R0, a    |
ADD R0, e
STORE d, R0
```

(B) flow of Control optimization-

(i) Avoid Jumps on jumps

L_1 : Jump ~~L_2~~ → L_4

L_2 : Jmp L_3

L_3 : Jmp L_4

L_4 : $x = a+b*c$

(ii) Elimination of dead Code-

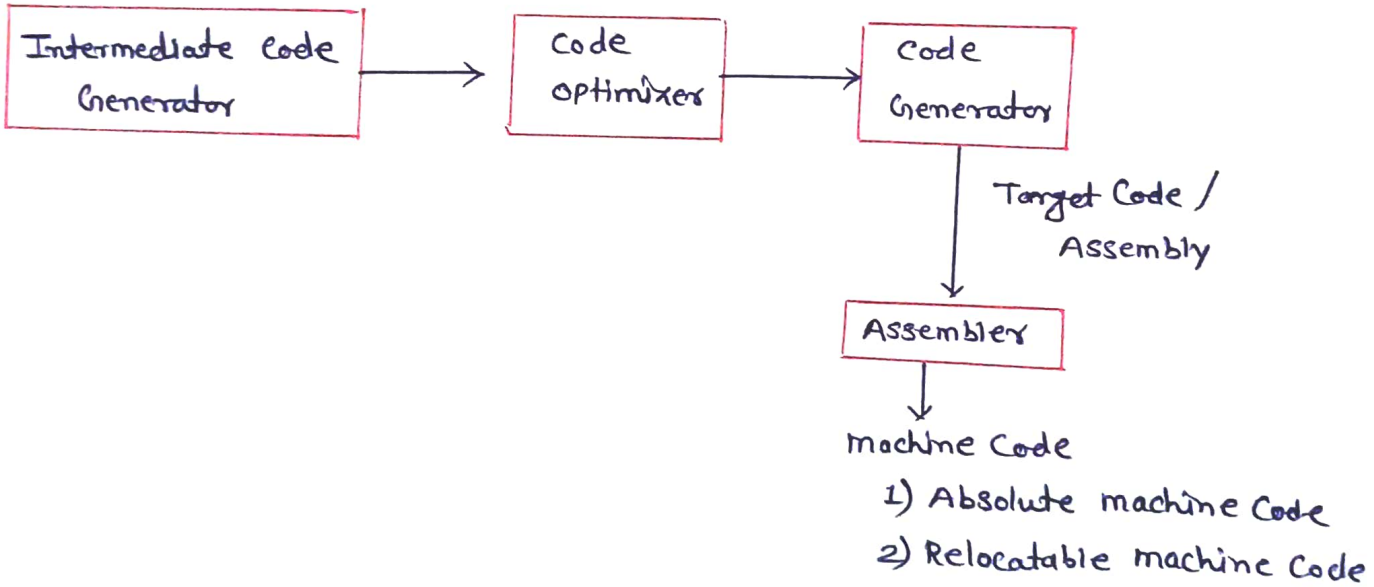
```
int i = 0
dead Code { if (i == 1) {
                printf(" NULL");
            }
```

(C) Use of m/c Idioms

```
i = i+1     LOAD R0, i
            ADD R0, #1
            STORE i, R0
```

→ INC i (for Some systems)

* Code Generation-



• Issues in Code Generation-

1) Instruction selection : choosing the best assembly Instruction (use INC instead of ADD x,1)
↳ faster, Cheaper

2) Register Allocation & Assignments

3) Instruction reordering → for better CPU utilization

$$x = a+b-[(c+d)-e]$$

$$t_1 = a+b$$

$$t_2 = c+d$$

$$t_3 = t_2-e$$

$$t_4 = t_1-t_3$$

Statement	L	Instrn selection	Register Descriptor	Address Descriptor
$t_1 = a+b$	R_0	mov R_0, a ADD R_0, b	R_0 holds t_1	t_1 is in R_0
$t_2 = c+d$	R_1	mov R_1, c ADD R_1, d	R_1 holds t_2	t_2 is in R_1
$t_3 = t_2-e$	R_1	SUB R_1, e	R_1 holds t_3	t_3 is in R_1
$t_4 = t_1-t_3$	R_0	SUB R_1, R_0 mov x, R_0	R_0 holds x	x is in R_0 and memory

* Instruction Cost-

mode	form	meaning	Address Cost
Absolute	m	Actual memory Address is used	1
Register	R	Data is inside a CPU Register	0
Indexed	c(R)	Address Reg(R) + c (c: Constant)	1
Indirect Reg	*R	R holds the address of data	0
Indirect Indexed	*c(R)	Content(c + Content(R))	1
Literal	#c	Actual Number	1

Total Cost = 1 + Cost for source + Cost for destination

↳ for the Instruction itself.

mov R_0 R_1

Cost = 1+0+0 = 1

Assembly Code – B = A[i]

mov R_1, i ($R_1 \leftarrow i$)

MUL R_1, #4

mov R_2, A(R_1)

mov B, R_2

x = *p

mov R_1, p

mov R_2, [R_1]

mov x, R_2

if x<y goto L

Load R_1, x

Load R_2, y

cmp R_1, R_2

BLTZ L

* Sethi – Ulman Algorithm