# OPTIMISATION
# à lucarne (“Peephole”)

Petites optimisations plus ou moins locales qui seraient évidentes à un être humain; améliorations à l’intérieur d’un bloc de base ou analyse triviale du flot de contrôle.
peut être effectuée au niveau du code intermédiaire ou du code objet.

- motifs

- code mort: des instructions qui ne pourraient jamais être exécutées

- Puissance des opérateurs: souvent une addition coûte moins qu’une multiplication, et/ou une opération logique coûte encore moins.

- chaînes de `goto`: peuvent être court-circuitées

# Exemples

```
LDT $1, $5, 0
LDT $2, $5, 0

STT, $1, $5, 1
LDT, $1, $5, 1

ADD $2, $1, 0

ADD $1, $1, 1
ADD $1, $1, 5

MULU $1, $2, 32
SLU  $1, $2, 5

ADD $1, $2, $2


   JMP L
L: ...
```

```
   JMP M
   ...
M: JMP B..
```

# LES BOUCLES

Deuxième source très importante d'améliorations:

Définition d'une boucle à partir du graphe des blocs de base: un sous-graphe fortement connexe avec une seule entrée (et une façon de rentrer).

Quels sont les boucles imbriquées??
(évident dans le programme initial; ne l'est plus)
Mais noter les instructions du type `PBZ $X, RA` de MMIX

L'entrée est un bloc de base qui en "domine" tous les autres.

Important parce qu'il y a souvent des optimisations intéressantes et l'effet dynamique peut être très important (parce qu'il y aura plusieurs itérations).

Des instructions du programme de départ qui sont textuellement dans une boucle ne sont forcément pas dans la boucle selon cette définition (si elles sont exécutées après la décision de sortir)

# Les invariantes

Ceci veut dire ici les variables et expressions dont la valeur ne change pas à cause des affectations dans la boucle

Optimisation: calculer les valeurs une seule fois (dans un nouveau bloc le “pré-en-tête” de la boucle). Peut-être affecter ces valeurs à leur destination finale. Si possible laisser ces valeurs dans des registres pendant toute l'exécution de la boucle.

Plus précisément une expression est une invariante si chaque évaluation donnera le même résultat: elle peut dépendre d'autres variables si elles sont des invariantes (et ont déjà reçu leur valeur): algorithme assez simple pour les trouver.

# L'UTILISATION DE REGISTRES

En général, à la fin d'un bloc de base on doit stocker les valeurs dans les registres si (i) les valeurs sont les valeurs de variables, (ii) ces variables ne contiennent pas leurs propres valeurs et (iii) il existe la possibilité d'utiliser cette valeur de la variable sans passer par un bloc de base qui lui en affecte une autre.

A la fin d'une itération d'une boucle on peut très souvent supprimer ces opérations, c'est-à-dire on peut les déplacer à l'extérieur.

# VARIABLES D'INDUCTION (DES BOUCLES)

Souvent des variables sont incrémentées par une constante à chaque itération d'une boucle. s'il y a plusieurs de ces variables la valeur de chacune est donnée par une fonction (linéaire) de la valeur d'une autre.

optimisations:

réduire leur nombre: si `i` est `j+5;` peut-être supprimer `i` et remplacer `A[i]` par `A'[j]` (`A'` un pointer à `A[5]`)
remplacer des multiplications par des additions: peut-être `j:=2*i` est remplacé par `j:=j+2` après initialisation de `j` (cas très commun où une variable d'induction est utilisée pour indiçage dans un tableau).
*exemple d'utilisation d'information sur les invariantes*

`i=exp; while(i>j)` équivaut à `i=exp-j; while(i>0)` si `j` est une invariante (et `i` n'est pas utilisée ailleurs)

# BOUCLES COMPRISES DE PLUSIEURS BLOCS DE BASE

Une suite de blocs qui a plusieurs sorties possibles ne change pas grand-chose.

S'il y a plusieurs retours à la tête ou généralement plusieurs ordres possibles d'exécuter les blocs, il devient plus compliqué de décider:

- quelles sont les variables d'induction: `i:=i+1` est-il exécuté une fois à chaque itération?

- quelles sont les invariantes: Une affectation à `j` est-elle toujours exécutée avant l'utilisation de `j`?

On aborde la question de l'analyse des flots (des données).