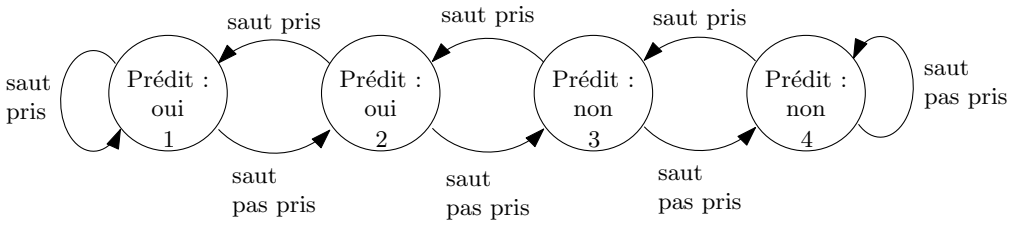
Architecture des ordinateurs
L3 Informatique 2020-2021 Binôme :
TP 5 - pipelines & prédicteurs de branchement

- Chaque binôme rendra une feuille d'énoncé complétée et la déposera sur *e-learning* avec les programmes écrits pour répondre aux questions.
- Toutes les mesures de temps doivent être faites au moyen de la fonction `print_timing` écrite au TP 2.
- Ce TP a pour objectif de comprendre la manière dont les prédicteurs de branchement affectent l'efficacité d'un programme.

Exercice 1. ★ Fonctionnement d'un prédicteur de branchement On s'intéresse dans cet exercice au comportement d'un processeur dont le pipeline utilise un prédicteur de branchement à 4 états.



a. On considère le programme suivant et sa traduction en assembleur :

```
1  int compte(int n, int* tab){
2    int s;
3
4    for (int i = 0; i < n; ++i){
5        if (tab[i]<50){
6            s += 1;
7        }
8    }
9    return s;
10 }
```

```
1  compte: mov     rdx, rsi
2          lea     ecx, [rdi-1]
3          lea     rsi, [rsi+4+rcx*4]
4  .L1:    cmp     DWORD PTR [rdx], 50
5          jge     .L2
6          inc     eax
7  .L2:    add     rdx, 4
8          cmp     rdx, rsi
9          jne     .L1
10         ret
```

Indiquer les numéros de ligne des instructions assembleur de saut conditionnel et, pour chacune d'entre elle, le numéro de ligne de l'instruction C correspondante.

b. On suppose que le processeur consacre un prédicteur de branchement 4 états uniquement à l'instruction assembleur de la ligne 5 (`jge .L2`). On appelle la fonction sur un tableau de 20 cases dont les valeurs provoquent le résultats suivants :

indice dans `tab`	0	1	2	3	4	5	6	7	8	9
saut pris ?	oui	oui	oui	non	non	non	non	non	oui	oui
indice dans `tab`	10	11	12	13	14	15	16	17	18	19
saut pris ?	oui	oui	non	non	oui	oui	non	oui	non	non

(a) On suppose que le prédicteur est initialement dans l'état 1. Indiquer les indices du tableau pour lesquels le prédicteur de branchement fait une erreur de prédiction.

(b) Même question en supposant que le prédicteur est initialement dans l'état 4.

Exercice 2. ★ (Un saut conditionnel) Dans cet exercice, nous allons examiner l'influence du pipeline sur l'exécution d'une boucle comportant un saut conditionnel.

a. Écrire une fonction (en C) qui permet de compter tous les nombres plus petits qu'un certain seuil (donné en paramètre) dans un tableau d'entiers.

b. On veut tester cette fonction sur un grand tableau (de taille N) d'entiers aléatoires uniformes entre 0 et 1000, avec comme seuil 500. À votre avis, quelle valeur devrait renvoyer la fonction (approximativement) ? Et avec 200 comme valeur de seuil ?

c. Écrire un programme qui teste successivement cette fonction sur un grand tableau d'entiers aléatoires uniformes entre 0 et 1000, pour différentes valeurs du seuil (0, 50, 100, ...) et mesurer le temps d'execution de la fonction pour chaque valeur de seuil. Qu'observe-t-on et comment peut-on l'expliquer ?

d. Confirmer vos soupçons en utilisant la commande `perf`. Pour apprendre à l'utiliser : `https://perf.wiki.kernel.org/index.php/Tutorial`.

Noter les résultats (intéressants) obtenus avec `perf`.

e. **(Optionnel)** Recommencez cette expérience en compilant avec l'option `-O1`. Qu'observe-t-on ? Appelez votre chargé(e) de TP et proposez lui une explication.

Exercice 3. ★ (Deux sauts conditionnels imbriqués) On s'intéresse maintenant aux sauts conditionnels imbriqués. On considère un tableau d'un million d'entiers dont les valeurs sont comprises entre 1 et 100. On veut compter le nombre d'entrées de ce tableau comprises entre 25 et 50 (bornes inclues).

a. Écrivez deux fonctions qui réalisent ce compte par deux imbrications de `if` différentes :

```
if (v>24)
  if (v<51)
    j++;
```

et

```
if (v<51)
  if (v>24)
    j++;
```

b. Compilez votre programme avec les options `-O0 -falign-functions` et mesurez les temps d'exécution de ces deux fonctions. La différence de temps vous paraît-elle significative ? Si oui, en faveur de quel ordre ?

c. Supposons que chacune des 100 valeurs possibles est répétée le même nombre de fois dans le tableau (soit 10000 répétitions chacune). Calculer, pour chacun des quatre `if` (deux par fonction) le pourcentage de fois où la condition est vraie.

d. En utilisant ces calculs, proposez une explication précise de la différence de vitesse observée.

e. Vérifiez votre explication au moyen de l'outil `perf` et notez les résultats obtenus :

Exercice 4. ★★ (Bulles de pipeline) Dans cet exercice, on s'intéresse à la manière dont le compilateur `gcc` optimise le code pour réduire les bulles dans le pipeline. On prendra comme exemple le code C suivant :

```
1  #include <stdio.h>
2  int main() {
3    int i, j=0, k=0, l=0, res=0;
4    for (i=1; i<10; i++){
5      j+=i*i*i*i;
6      k+=j*j*j*j;
7      l+=j*j*k*k;
8      res+=j/k;
9      res+=l;
10   }
11   printf("%d\n",res); return 0;
12 }
```

a. Compiler ce code en `-O1` et `-O2` et indiquer côte à côte les traductions en assembleur de la boucle (lignes 5 à 9).

b. Relier ensemble les instructions qui ont été déplacées entre `-O1` et `-O2` et expliquer pourquoi cela réduit le nombre de bulles du pipeline.

Exercice 5. ★ (Astuce pour power coders) Un site web [1] propose l'astuce de programmation suivante : quand un programme comporte un `if` dont la condition est un `et` logique entre deux conditions, il vaut mieux le séparer en deux `if` imbriqués. Ce site dit qu'il faudrait remplacer

```
1       if (X > 5 && X < 95) //branch is taken 90% of the time
2         do_something();
```

par le code

```
1       if(X > 5) //branch is taken 95% of the time
2         if(X < 95) //branch is taken 95% of the time
3           do_something();
```

a. Testez expérimentalement cette affirmation en exécutant chacun de ces blocs d'instructions sur un grand nombre de données aléatoires entre 0 et 100. La différence vous semble-t-elle significative ? Si oui, est-ce en faveur de la condition double ou des deux conditions simples ?

b. Examiner la traduction en assembleur de chacun de ces deux blocs d'instructions. Quelles différences identifiez-vous ?

c. Commenter l'utilité de cette "astuce pour power coders".

Exercice 6. (Recherche dichotomique biaisée) Un test "biaisé" répété un grand nombre de fois est en général exécuté plus rapidement qu'un test "équilibré" (comme constaté à l'exercice 3). La recherche dichotomique standard coupe, à chaque itération, l'intervalle considéré en deux parties égales. Pour des données réparties uniformément, cela ne produit que des tests équilibrés. Il est possible de biaiser ces tests en coupant l'intervalle non pas à la moitié, mais à une autre proportion (par exemple 3/8 et 5/8).

a. Écrivez une fonction qui initialise un tableau de grande taille avec des valeurs aléatoires, puis trie ce tableau (avec `qsort`, par exemple).

b. Écrivez deux fonctions de recherche dichotomique dans ce tableau. La première sera la dichotomie standard (découpe des intervalles au milieu), la seconde sera biaisée par un paramètre `P` (découpe des intervalles en proportion `P`). La valeur de `P` sera définie comme une constante (`#define`).

c. Déterminez, au moyen de l'outil `perf`, une valeur de `P` pour laquelle le nombre moyen d'erreurs de prédictions est significativement plus faible pour la recherche biaisée que pour la recherche standard.

d. Déterminez, par des mesures de vitesse, une valeur de `P` pour laquelle la recherche biaisée est plus efficace que la recherche standard.

1. `http://www.futurechips.org/tips-for-power-coders/quick-post-trick-improve-branch-prediction.html`