Chapitre 8

L'amélioration des performances

8.1 Les mémoires caches

Une mémoire cache est un tampon à accès rapide placé entre la mémoire principale et le processeur. Ce tampon contient les données et les instructions les plus récemment accédées. Quand le cache trouve la donnée dans le cache, on parle de *succès*. Quand le cache ne trouve pas la donnée demandée par l'UC, il se produit *un défaut de cache* ou *échec*, la donnée est alors cherchée dans la mémoire centrale et mise dans le cache.

La mémoire cache ne modifie pas l'aspect logique de la liaison entre le processeur et sa mémoire, on a toujours une architecture de Von Neumann. Physiquement, ce tampon peut être à l'intérieur du processeur ou à côté de celui-ci, sur le circuit imprimé qui le supporte.

8.1.1 La hiérarchie de mémoire

Cette hiérarchie découle d'un principe au niveau du matériel assez simple : *le plus petit est le plus rapide* ; et de la propriété de localité des accès au niveau logiciel. L'idée générale est de conserver à proximité de l'unité centrale les éléments les plus récemment et les plus fréquemment utilisés dans des petites mémoires d'accès rapide, et plus on s'éloigne du processeur, plus ces mémoires sont grandes et lentes. Ainsi de la même façon qu'une mémoire cache est installée entre la mémoire principale et l'unité centrale, le mécanisme de *mémoire virtuelle* fait que certaines données qui servent peu ou n'ont pas servi depuis un certain temps sont déportées sur le disque.

Niveau	1	2	3	4
Nom	Registres	Cache	Mémoire centrale	Disque
Taille typique	< 1 ko	< 512 ko	< 512 Mo	> 1 Go
Temps d'accès (ns)	10	20	100	20 000 000
Débit (Mo/s)	800	200	133	4
Géré par	compilateur	matériel	système d'exploitation	SE et utilisateur
Sauvegarde	cache	mémoire centrale	disque	bande

Une des principales différences entre les registres et les cellules du cache consiste en la gestion de leur allocation. Pour les registres, cette allocation est entièrement réalisée par le compilateur d'une façon statique : le compilateur est conçu pour dédier certains registres à certaines tâches (pointeur de pile, pointeur sur les variables globales, sur le bloc d'exécution courant, etc.), par ailleurs, il peut essayer de prévoir quelles seront les données qui seront utiles fréquemment pour leur allouer des registres dans certaines zones d'un programme (typiquement cette allocation a lieu pour chaque fonction indépendemment des autres fonctions).

Par contre, l'ajustement des données effectivement présentes dans le cache est un phénomène dynamique qui a lieu en permanence. Lorqu'une donnée est lue en mémoire centrale, elle est aussi transférée dans le cache, de façon à ce que de prochains accès à cette même donnée puissent être réalisés beaucoup plus rapidement. L'idée générale consiste à essayer d'avoir une mémoire qui a quasiment la vitesse du processeur et qui contienne les données qui ont la plus forte probabilité d'être

accédées par le processeur. Il faut donc implémenter une politique de remplacement de l'information présente dans le cache. L'état normal du cache est d'être plein et lorsqu'une donnée qui n'est pas présente dans le cache doit être lue ou écrite dans la mémoire centrale, une place doit lui être faite dans le cache. Une information déjà présente dans le cache doit être "sacrifiée". C'est à ce moment que la probabilité d'utilisation future des informations présentes dans le cache doit être évaluée : plusieurs algorithmes existent à ce niveau.

L'unité d'allocation — habituellement appelée un *bloc* ou une *ligne* — dans un cache est habituellement plus grande que l'unité d'allocation dans la mémoire centrale. Certains boîtiers mémoires sont prévus pour fonctionner dans un mode *rafale*. Ce mode permet d'accéder à des cellules successives de la mémoire plus rapidement que si des demandes à des adresses indépendantes sont faites. Ainsi pour charger dans le cache quatre mots successifs avec une mémoire à temps d'accès de 100 nanosecondes, il faut beaucoup moins de 400 nanosecondes.

8.1.2 Les caches de données

Pour les caches de données, il faut tenir compte du fait que celles-ci peuvent être écrites, c'est-à-dire que certains emplacement de la mémoire centrale doivent être modifiés. Il existe deux politiques à ce niveau :

- l'*écriture simultanée* (en anglais : write through) : les données modifiées sont copiées dans la mémoire centrale aussitôt qu'elles ont été modifiées ;
- et la *réécriture* (en anglais : copy back) : les données sont copiées depuis le cache dans la mémoire centrale que lorsque la ligne du cache qui les contient doit être "sacrifiée" pour laisser place à d'autres données.

Dans le deuxième cas, un bit de modification stocké dans l'étiquette indique si les données du cache ont été modifiées par rapport à celles de la mémoire, et donc si il y a besoin de réécrire ce bloc en mémoire centrale.

Avec la réécriture, les écritures (en cas de succès) se font à la vitesse du cache, et plusieurs écritures sur le même bloc ne nécessiteront qu'une écriture dans la mémoire centrale. Cependant avec cette méthode, il peut arriver qu'une lecture demandée par le processeur se traduise au niveau de la mémoire cache par une écriture suivie d'une lecture en mémoire centrale. Ceci ne peut pas arriver avec la technique de l'écriture simultanée.

Dans le cas multi-processeurs, on choisit la technique de l'écriture simultanée pour que la mémoire principale ait la valeur la plus récente des données.

8.1.3 Les modèles de cache

Cache à correspondance directe Dans ce type de cache, chaque bloc de mémoire a un unique emplacement possible dans le cache. La correspondance réalise la relation suivante : numéro du bloc dans le cache = numéro du bloc mémoire modulo nombre de blocs dans le cache.

Cache totalement associatif Ici, chaque bloc de mémoire peut être placé n'importe où dans le cache.

Cache associatif par ensemble de blocs Un bloc peut être placé dans un ensemble restreint d'endroits dans le cache. Un *ensemble* est un groupe de deux (ou quatre, ou huit, etc.) blocs dans le cache. Un bloc est d'abord affecté à un ensemble — par exemple, on peut avoir la relation suivante : numéro d'ensemble = numéro du bloc mémoire modulo nombre **d'ensembles** dans le cache —, puis le bloc est placé n'importe où dans l'ensemble.

La figure suivante illustre les positions possibles pour le bloc de mémoire numéro 12 dans un cache à huit positions selon différents modèles de cache. Les positions possibles sont marquées par des X. Bien sûr, les mémoires réelles ont des centaines de milliers de blocs et les caches ont des centaines de blocs. Pour l'associativité par ensemble, nous supposons qu'il y a quatre ensembles de deux blocs chacun.

0	XXXXXXXX
1	XXXXXXXX
2	XXXXXXXX
3	XXXXXXXX
4	XXXXXXXX
5	XXXXXXXX
6	XXXXXXXX
7	XXXXXXXX

Cache totalement associatif.
Le bloc numéro 12 peut aller partout.

0	
1	
2	
3	
4	XXXXXXXX
5	
6	
7	

Cache à correspondance directe.
Le bloc 12 peut aller seulement dans le bloc 4 (12 mod 8).

0	XXXXXXXX
1	XXXXXXXX
2	
3	
4	
5	
6	
7	

Cache associatif par ensemble de blocs.
Le bloc 12 peut aller n'importe où dans l'ensemble 0 (12 mod 4).

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	XXXXXXXX
13	
14	
15	
16	
17	
18	
19	
20	
21	
22	
23	
24	
25	
26	
27	
28	
29	
30	
31	

Dans les caches associatifs, la recherche de la présence doit s'effectuer en parallèle : une recherche en série serait trop lente et annulerait tout le bénéfice que pourrait apporter un cache.

Une entrée dans le cache doit contenir le bloc de mémoire mais aussi certaines informations rangées dans l'*étiquette* :

- bit de validité,
- adresse effective en mémoire centrale,
- bit de modification, pour la *réécriture*,
- indicateurs d'usage, pour l'algorithme de remplacement.

8.1.4 Stratégie de remplacement

Lorqu'on doit libérer un bloc dans le cache, il faut choisir quelles données seront sacrifiées. Il existe principalement deux méthodes :

- le hasard
- *le plus ancien* (en anglais : Least Recently Used, LRU)

8.1.5 Stratégie de chargement

Lorqu'il y a échec sur une écriture, on peut décider d'attribuer ou non un bloc dans le cache au bloc mémoire qui doit être écrit, on parle :

- d'*écriture attribuée*,
- d'*écriture non attribuée*.

8.2 Les architectures RISC

Dans les années 1970, l'état de la technologie a conduit à certains choix dans l'architecture des processeurs, choix résumés par quelques règles :

1. la technologie mémoire utilisée pour les microprogrammes croît très vite, il ne coûte rien ou presque d'utiliser de très gros microprogrammes ;
2. comme les micro-instructions sont beaucoup plus rapides que les instructions machines normales, le transfert de fonctions logicielles au niveau du microcode accélère le processeur et rend les fonctions plus fiables ;
3. puisque la vitesse d'exécution est proportionnelle à la taille du programme, les techniques architecturales qui diminuent la taille des programmes accélèrent les processeurs ;
4. les registres sont démodés et rendent difficiles la réalisation des compilateurs. Les piles ou les architectures mémoire à mémoire sont des modèles d'exécution supérieurs.

Ainsi le choix CISC s'oriente vers un grand nombre d'instructions complexes (c'est-à-dire qui font beaucoup de choses) avec des modes d'adressages larges et orthogonaux (au type d'opération) en laissant les données en mémoire (c'est-à-dire en utilisant peu les registres). Le grand nombre d'instructions combinés avec la multiplicité des modes d'adressage amène à avoir un codage des instructions sur un nombre variable de mots, ce qui fait qu'il faut commencer à décoder une instruction pour savoir combien de mots elle occupe.

En fait, comme nous allons le détailler, ces instructions complexes se sont avérées peu utilisées par les compilateurs. L'idée RISC consiste à régulariser le fonctionnement et à simplifier le jeu d'instructions de façon à accélérer les différentes étapes de l'exécution des instructions.

8.2.1 Critères de performance

8.2.1.1 Exécution d'un programme simple

La durée d'exécution T_e d'un programme s'exprime de la manière suivante :

$$T_e = N_i \times N_c \times T_c$$

où

N_i représente le nombre d'instructions machine exécutées par le programme ;

N_c représente le nombre de cycles d'horloge nécessaires en moyenne pour exécuter une instruction ; et,

T_c représente le temps de cycle de l'horloge de base du processeur.

N_i est fonction des instructions machines disponibles et de l'usage qu'en fait le compilateur.

N_c dépend de la complexité des instructions. Une question à se poser est sur le mode de calcul de la moyenne (quelle *mesure* (au sens mathématique) est utilisée ?) par rapport aux modes d'adressages, aux instructions effectivement utilisées par le compilateur.

T_c est fonction de la technologie utilisée qui détermine le temps d'exécution des étapes élémentaires et de la décomposition des instructions en ces étapes élémentaires.

8.2.1.2 Les étapes d'exécution d'une instruction

Nous donnons ici un aperçu des différentes actions à effectuer pour l'exécution des instructions.
- Lecture de l'instruction (*fetch*)
- Mise à jour du compteur de programme
- Décodage de l'instruction
- Lecture des opérandes
- Lecture des opérandes dans l'UAL
- Rangement des opérandes

Chaque architecture de processeurs choisit de regrouper ces actions en quatre ou cinq *phases*.

8.2.2 Propriétés dynamiques des programmes

En fait, le choix des instrcutions utiles (réellement utilisées) ne peut se faire que sur une étude statistique des programmes réels.

8.2.2.1 Les instructions des langages de haut niveau

Ce tableau présente les occurences dynamiques relative des "instructions" des langages de haut niveau :

Langage Application	Pascal Scientifique	Fortran Etudiant	Pascal Système	C Système	SAL Système
Affectation	74%	67%	45%	38%	42%
Boucle	4%	3%	5%	3%	4%
Appel de procédure	1%	3%	15%	12%	12%
Si	20%	11%	29%	43%	36%
Aller à	2%	9%		3%	
Autres		7%	6%	1%	6%

8.2.2.2 Les instructions machines

La traduction des programmes en langage machine nous oblige à pondérer ces pourcentages. Deux critères peuvent être utilisés, les instructions machines engendrées par chacune des instructions du langage de haut niveau, ou le nombre d'accès mémoire à réaliser à l'exécution :

	Occurence dynamique		Pondérée en instructions machine		Pondérée en accès mémoire	
	Pascal	C	Pascal	C	Pascal	C
Affectation	45%	38%	13%	13%	14%	15%
Boucle	5%	3%	42%	32%	33%	26%
Appel de procédure	15%	12%	31%	33%	44%	45%
Si	29%	43%	11%	21%	7%	13%
Aller à		3%				
Autres	6%	1%	3%	1%	2%	1%

8.2.2.3 Occurences des opérandes

Nature	Pascal	C	Moyenne
Constante entière	16%	23%	20%
Variable scalaire	58%	53%	55%
Tableau/structure	26%	24%	25%

8.2.2.4 Nombre de paramètres par procédure

L'importance des accès mémoire pour les appels de procédures conduit à étudier le nombre de paramètres à passer aux procédures appelées et leur niveau d'imbrication. En effet, si ces deux caractéristiques peuvent être bornées par un nombre suffisamment petit, alors en implantant suffisamment

de registres dans le processeur tous les paramètres des procédures pourraient être passés dans ces registres.

Une première étude montrait que 98% des procédures appelées lors de l'exécution des programmes considérés utilisaient moins de six paramètres, et que 92% utilisaient moins de six variables locales. D'autres travaux confirment ces résultats :

Pourcentage de procédures exécutées avec ...	Compilateurs, interpréteurs et éditeurs	Petits programmes non numériques
plus de 3 arguments	0 à 7%	0 à 5%
plus de 5 arguments	0 à 3%	0
plus de 8 arguments et variables locales	1 à 20%	0 à 6%
plus de 12 arguments et variables locales	1 à 6%	0 à 3%

8.2.2.5 Profondeur des appels

En ce qui concerne la profondeur des appels, il est très rare qu'il y ait de grandes variations par une suite ininterrompue d'appels suivie de la suite inverse des retours. Le plus souvent, les appels se stabilisent dans une fenêtre de largeur limitée, de l'ordre de 5 appels imbriqués, même s'il arrive qu'une telle fenêtre change de profondeur.

8.2.2.6 L'utilisation effective des architectures CISC

Les opérandes

Sur un VAX :

Opérandes	TeX	Spice	gcc
Immédiats	18%	8%	19%
Registre	57%	53%	51%
Mémoire	25%	39%	30%

Sur un 8086 :

Opérandes	Lotus	Assembleur	TurboC
Immédiats	5%	7%	11%
Registre	52%	55%	46%
Mémoire	43%	37%	43%

Bien que les architectures CISC comportent peu de registres, et ne favorisent donc pas leur utilisation, on constate qu'une grande majorité des opérandes sont situés dans ceux-ci.

Les modes d'adressage

Sur un VAX :

Mode d'adressage	TeX	Spice	gcc
Auto-incrément	1%	3%	4%
Indirect via mémoire	2%	7%	2%
Indexé avec facteur d'échelle	0%	20%	9%
Indirect via registre	41%	4%	18%
Indirect via registre + déplacement	55%	66%	66%

Sur un 8086 :

Mode d'adressage	Lotus	Assembleur	TurboC
Indirect via registre	15%	12%	9%
Direct	34%	36%	18%
Indirect via registre + déplacement	51%	52%	73%

Dans cette table, le modes "base + index" ou "base + index + déplacement", qui ont une fréquence d'utilisation très faible n'apparaissent pas.

En fait, le mode "indirect via registre + déplacement" correspond à l'accès aux variables locales ou aux paramètres des procédures, et le mode "indirect via registre" correspond à l'accès de données

via pointeurs ou adresse calculée. Ces utilisations correspondent à la quasi totalité des accès qui sont effectivement faits en mémoire.

Les instructions Dans les deux tableaux qui suivent, il est indiqué entre parenthèses le nombre d'instruction concernées par le titre. De plus, les instructions qui ont une fréquence d'utilisation inférieure à 1,5% n'ont pas été mentionnées.

Sur un VAX :

Instructions	TeX	Spice	gcc	Cobolx	Moyenne
Contrôle (5)	30%	18%	30%	25%	26%
Arithmétique et logique (10)	33%	23%	40%	24%	30%
Transfert de données (4)	28%	15%	19%	4%	16%
Virgule flottante (5)	0%	23%	0%	0%	6%
Décimal et caractères (5)	1%	0%	0%	38%	10%
Total	92%	79%	88%	88%	87%

Sur un 8086 :

Instructions	Lotus	Assembleur	TurboC	Moyenne
Contrôle (5)	21%	20%	32%	24%
Arithmétique et logique (10)	23%	24%	26%	25%
Transfert de données (4)	49%	46%	30%	42%
Total	93%	90%	88%	90%

8.2.3 La philosophie RISC

L'idée générale consiste donc à n'implanter sur le silicium du processeur que les instructions et les modes d'adressage statistiquement utiles. De ce fait, la complexité de la partie contrôle de la machine est largement diminuée et peut de nouveau être cablée puisqu'elle devient la méthode la plus rapide et la moins coûteuse en surface, et donc en prix de développement et de fabrication. Cette diminution de la partie contrôle peut alors être réutilisée pour augmenter le nombre de registres. La simplification des modes d'adressages a aussi des influences sur la complexité des compilateurs. Réciproquement, pour qu'un compilateur puissent optimiser efficacement, les caractéristiques suivantes sont utiles :

- format d'instructions à trois adresses,
- grand nombre de registres,
- jeu d'instructions symétrique.

Une autre idée consiste à accélérer au maximum l'exécution des instructions. Ainsi dans la formule $T_e = N_i \times N_c \times T_c$ on essaye de diminuer le facteur N_c, la limite à atteindre étant 1. Cela conduit aux choix suivants :

- instructions de longueurs fixes,
- codage simple et homogène des instructions,
- accès mémoire avec uniquement deux instructions **load** et **store**,
- modes d'adressage simples,
- branchements retardés.

8.3 Le pipeline d'instructions

Si les différentes phases de l'exécution d'une instruction peuvent être rendues indépendantes, il est possible de mettre autant d'éléments dans le processeur qu'il y a de phases et que chacun de ces éléments travaille à la chaîne sur les instructions successives.

Prenons un exemple avec quatre phases :

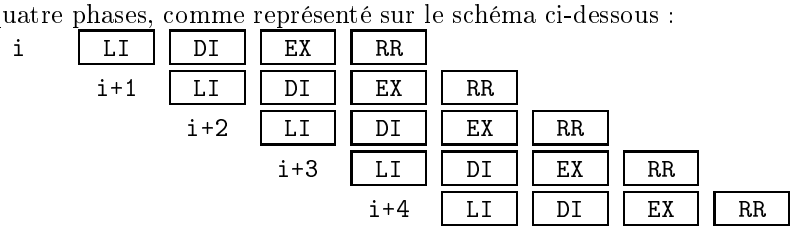
LI lecture de l'instruction et mise à jour du compteur de programme,

DI décodage de l'instruction (et lecture des opérandes, ou calcul des adresses mémoire, ou calcul de l'adresse de branchement),

EX exécution de l'instruction dans l'unité arithmétique et logique (ou accès mémoire),

RR rangement du résultat.

Ainsi avec quatre unités d'exécution, chacune prenant en charge une des phases, à chaque instant, quatre instructions sont en cours d'exécution et chacune de ces quatre instructions est dans une des quatre phases, comme représenté sur le schéma ci-dessous :



Dans notre exemple de pipeline, il y a quatre cycles d'horloge entre le début et la fin de l'exécution d'une instruction. Cependant, à chaque cycle d'horloge une instruction est terminée

Cependant, il n'est pas possible de toujours assurer un fonctionnement optimal d'un pipeline. En effet, certaines interdépendances entre des instructions successives empêchent parfois d'utiliser à certains moments tous les étages du pipeline. Chaque fois qu'un (ou plusieurs) étage du pipeline sont inutilisés pendant un (ou plusieurs) cycle, on parle de *bulle*. Nous verrons dans la suite les cas les plus courants de tels problèmes.

Par ailleurs, ce n'est pas nécessairement une bonne idée d'augmenter le nombre de phases (dans le but de les simplifier de façon à ce que chaque phase puisse être réalisée en *un* cycle). En effet, plus il y a de phases, et donc d'étages dans le pipeline, plus les conséquences des situations qui perturbent le fonctionnement idéal du pipeline (sauts, branchements, interruptions, etc.) seront importantes.

Pour que chaque phase puisse être exécutée en un cycle, certaines contraintes apparaissent, nous les évoquons ci-dessous :

Lecture de l'instruction On sait que les temps d'accès à la mémoire sont longs par rapport aux temps de cycle des processeurs actuels — qui ont des horloges entre 20 et 100 MHz. Pour que la lecture de l'instruction puisse malgré tout être effectuée en *un* cycle, l'instruction doit être dans une mémoire cache, ou dans une tampon (une file) d'instructions chargé "à l'avance".

Décodage de l'instruction Le décodage de l'instruction doit être réalisé en un cycle, pour cela le format des instructions doit être très régulier pour que cette contrainte n'entraîne pas une grande surface de silicium pour implanter le décodage. Par exemple, les bits qui indiquent quels sont les registres qui doivent être utilisés doivent être positionnés au même endroit pour toutes les instructions.

Exécution ou accès mémoire Comme pour la lecture de l'instruction, pour que l'accès mémoire puisse être réalisé en un cycle, la donnée doit être dans une mémoire cache.

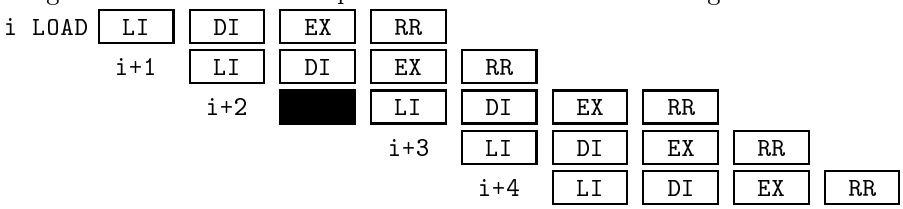
Par ailleurs, l'unité arithmétique et logique doit exécuter toutes les opérations qu'elle connaît en un cycle. Sur les premiers processeurs RISC, il n'y avait ni multiplication ni division pour cette raison. Ce problème existe toujours pour les calculs flottants et sa résolution conduit à avoir une unité spécialisée pour ceux-ci et à gérer des dépendances lorsqu'ils ne peuvent être effectués en un cycle.

Au niveau de l'architecture interne de la machine, on s'aperçoit que certaines unités doivent être implantées plusieurs fois. Typiquement, dans une architecture simple contrôlée par micro-programme, seule l'unité arithmétique et logique est en mesure de réaliser des additions. Dans une architecture RISC, on trouvera plusieurs additionneurs, un pour incrémenter le compteur ordinal, un pour calculer les adresses des opérandes en mémoire, et un dans l'unité arithmétique et logique. De cette façon, les unités peuvent effectuer leurs additions en parallèle.

8.3.1 Les conflits de ressources

Si une ressource (par exemple la mémoire cache) ne peut gérer qu'un seul accès à la fois, il faut interdire son utilisation simultané par plusieurs unités. Ainsi la phase `EX` d'une instruction **load** au

rang `i` est en conflit avec la phase `LI` de l'instruction de rang `i+2`.



8.3.2 Les dépendances de données

Il arrive que l'instruction de rang `i` ait besoin du résultat de l'instruction de rang `i-1`. Par exemple, la séquence $R_5 \leftarrow R_1 - R_2$ puis $R_4 \leftarrow R_5 + R_3$ doit être effectuée de la façon suivante :

8.3.3 Le problème des branchements

L'instruction qui suit une instruction de branchement ne peut pas être immédiatement chargée puisqu'il faut connaître son adresse, et que cette adresse ne sera connue qu'à l'issue de la deuxième phase de l'exécution de l'instruction de branchement :

On s'aperçoit aussi que cette adresse de branchement doit être calculée le plus tôt possible de façon à ne pas introduire une bulle plus grosse dans le pipeline.

Certains processeurs introduisent la notion de branchement retardé pour éviter l'introduction d'une bulle sur les branchements. Cela consiste à exécuter systématiquement l'instruction qui suit immédiatement le branchement. Un compilateur peut toujours y installer une instruction **NOP** ou une instruction plus utile s'il optimise plus en réordonnant les instructions.

Adresse	interprétation	programme normal		avec branchement retardé		avec branchement retardé optimisé	
100	$R_1 \leftarrow a$	`LOAD`	`R1,a`	`LOAD`	`R1,a`	`LOAD`	`R1,a`
101	$R_1 \leftarrow R_1 + 1$	`ADD`	`R1,R1,1`	`ADD`	`R1,R1,1`	`JUMP`	`105`
102		`JUMP`	`105`	`JUMP`	`106`	`ADD`	`R1,R1,1`
103	$R_1 \leftarrow R_1 + R_2$	`ADD`	`R1,R1,R2`	`NOP`		`ADD`	`R1,R1,R2`
104	$R_3 \leftarrow R_3 - R_2$	`SUB`	`R3,R3,R2`	`ADD`	`R1,R1,R2`	`SUB`	`R3,R3,R2`
105	$z \leftarrow R_1$	`STORE`	`z,R1`	`SUB`	`R3,R3,R2`	`STORE`	`z,R1`
106				`STORE`	`z,R1`		

Le cas du branchement conditionnel est plus complexe puisqu'il va être plus difficile pour le compilateur de trouver une instruction utile à mettre juste après le branchement et qui doive être exécutée dans les deux cas. Il est aussi possible de "parier" sur le cas qui va être choisi, le plus simple est de parier que le branchement ne sera pas fait et que c'est donc (comme d'habitude) l'instruction suivante qui sera à exécuter. Par contre, si le processeur perd son pari une bulle apparaît :

Certains processeurs mettent en œuvre des techniques assez complexes pour essayer de ne pas introduire de bulles dans le cas des branchements conditionnels.

8.4 Exercices

Exercice 8.1 (Exécution des instructions) *Un ordinateur est composé (entre autre) d'un processeur, de mémoire vive à temps d'accès de 70 nanosecondes, d'un disque magnétique à temps de positionnement moyen de 30 millisecondes et temps de latence de 16.6 millisecondes. Son horloge de base est à 10 mégahertz.*

Le processeur est remplacé par un nouveau modèle pouvant fonctionner à 20 mégahertz, et l'horloge modifiée en conséquence.

Question 8.1.1 *Peut-on dire approximativement quel est le gain de performances obtenues ?*

Exercice 8.2 (Langages) ***Question 8.2.1*** *Quelles sont les différences entre le langage machine et le langage d'assemblage ?*

Question 8.2.2 *Citez quelques langages "de haut niveau".*

Question 8.2.3 *Quels sont les avantages des langages de haut niveau par rapport au langage machine et au langage d'assemblage ?*

Exercice 8.3 (Mémoire morte) ***Question 8.3.1*** *Y'a-t-il toujours de la mémoire morte dans une machine avec un processeur ? Pourquoi ?*

Question 8.3.2 *Fait-elle partie de l'espace d'adressage du processeur ?*

Exercice 8.4 (Représentation des nombres) ***Question 8.4.1*** *Les deux expressions booléennes suivantes sont-elles équivalentes ? Elles sont exprimées en langage Pascal et m et n sont deux variables entières signées du type* `integer`*.*

- $m < n$
- $(m + 1) <= n$

Justifiez.

Exercice 8.5 (Pagination) *On considère la déclaration suivante :*

```
#define N 4096
float tab[N][N];
int i,j;
```

et les deux séquences d'instructions suivantes d'initialisation du tableau `tab` *:*

```
for (i=0; i<N; ++i)
  for (j=0; j<N; ++j)
    tab[i][j] = 0.;
```

```
for (j=0; j<N; ++j)
  for (i=0; i<N; ++i)
    tab[i][j] = 0.;
```

On utilise un ordinateur avec 32 méga-octets de mémoire centrale, un processeur à 50 Mips, un disque à 10 ms de temps de positionnement et 8.3 ms de temps de latence, le système gère la mémoire par pagination et les pages ont une taille de 4 kilo-octets.

Question 8.5.1 *Ces deux séquences d'instructions sont-elles équivalentes ?*

Question 8.5.2 *Et si on remplace* `#define N 4096` *par* `#define N 9216` *?*

Bradley's Bromide : If computers get too powerful, we can organize them into a committee – that will do them in.

Brook's Law : Adding manpower to a late software project makes it later