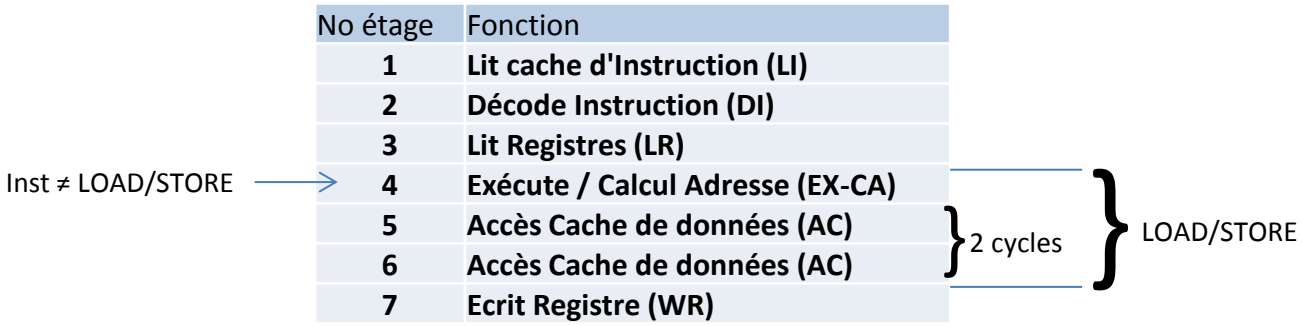
EISTI – ING1

Pipelining

ADO – TD 4

Pipelining 1



- 1. Combien de bulles (NOP) après un LOAD suivi d'un EXE :

L'accès à l'opérande d'un LOAD/STORE - à cause des accès cache - coûte 2 cycles, le résultat du LOAD/STORE est ensuite disponible via le mécanisme de bypass, sans attendre l'écriture registre du résultat.

2 bulles sont insérées car l'instruction LOAD nécessite les 2 cycles suivants. Explication : appelons X l'instruction après le LOAD. L'instruction X reste bloquée à l'étage 3 tant que l'accès cache du LOAD n'est pas terminé, c'est-à-dire tant que le LOAD n'a pas dépassé l'étage 6.

Juste avant que le LOAD rentre dans l'étage 7, le résultat du LOAD est envoyé sur le réseau de *bypass* et l'instruction X va pouvoir rentrer dans l'étage 4. A cet instant précis, les étages 4 et 5 ne contiennent aucune instruction. 2 bulles ont donc été introduites.

PIPELINE 1 : 2 bulles sont introduites dans le pipeline si l'instruction ***immédiatement après*** un LOAD/STORE ***utilise*** comme ***opérande*** le ***résultat*** du LOAD/STORE

Pipelining 1

No étage	Fonction
1	Lit cache d'Instruction (LI)
2	Décode Instruction (DI)
3	Lit Registres (LR)
4	Exécute / Calcul Adresse (EX-CA)
5	Accès Cache de données (AC)
6	Accès Cache de données (AC)
7	Ecrit Registre (WR)

Inst ≠ LOAD/STORE →

} 2 cycles

} LOAD/STORE

	Instruction	Commentaire
1	boucle: R2 = LOAD R1 + 0	// lit l'élément du tableau
2	R3 = R3 ADD R2	// ajoute à la somme
3	R1 = R1 ADD 4	// R1 = R1 + 4
4	R4 = R4 SUB 1	// R4 = R4 - 1
5	BNZ R4, boucle	// BNZ (Branch Not Zero): boucle si R4 diff de 0

Calcule la somme de N élts d'1 tableau

$R4_{init}$ = N

$R1_{init}$ = A_1 *(adr 1er élt du tableau)*

$R3_{init}$ = 0 *(à la fin, il contient le résultat)*

Eléments du tableau sur 4 Octets

- 2. Quel est le débit de cette boucle en instruction par cycle (pour N grand) ?

La boucle itère un grand nombre de fois parce que la valeur intiale de R4 est grande. Donc on peut raisonner comme si la boucle itérait un nombre infini de fois et négliger le temps de remplissage du pipeline. Le corps de la boucle comporte 5 instructions. La 2ème instruction dépend de la 1ère, qui est un LOAD. 2 bulles sont donc générées toutes les 5 instructions.

Le débit d'exécution vaut : $\frac{5}{5+2} = \frac{5}{7} \approx 0.71$ instructions / cycle.

Définition du débit : *débit* = nombre d'instructions / nombre de cycles avec :
nombre de cycles = (nombre d'instructions + nombre de bulles).

No étage	Fonction
1	Lit cache d'Instruction (LI)
2	Décode Instruction (DI)
3	Lit Registres (LR)
4	Exécute / Calcul Adresse (EX-CA)
5	Accès Cache de données (AC)
6	Accès Cache de données (AC)
7	Ecrit Registre (WR)

Inst ≠ LOAD/STORE →

} 2 cycles

} LOAD/STORE

	Instruction
1	boucle: R2 = LOAD R1 + 0
2	R3 = R3 ADD R2
3	R1 = R1 ADD 4
4	R4 = R4 SUB 1
5	BNZ R4, boucle

		LI	DI	LR	EX-CA	AC	AC	WR	
T1		R2 = LOAD R1 + 0							
T2		R3 = R3 ADD R2	R2 = LOAD R1 + 0						
T3		R1 = R1 ADD 4	R3 = R3 ADD R2	R2 = LOAD R1 + 0					
T4		R4 = R4 SUB 1	R1 = R1 ADD 4	R3 = R3 ADD R2	R2 = LOAD R1 + 0				
T5		R4 = R4 SUB 1	R1 = R1 ADD 4	R3 = R3 ADD R2	NOP	R2 = LOAD R1 + 0			
T6		R4 = R4 SUB 1	R1 = R1 ADD 4	R3 = R3 ADD R2	NOP	NOP	R2 = LOAD R1 + 0		
T7		BNZ R4, boucle	R4 = R4 SUB 1	R1 = R1 ADD 4	R3 = R3 ADD R2	NOP	NOP	R2 = LOAD R1 + 0	
T8	*Inst2-T1*	**R2 = LOAD R1 + 0**	BNZ R4, boucle	R4 = R4 SUB 1	R1 = R1 ADD 4	R3 = R3 ADD R2	NOP	NOP	
T9	*Inst2-T2*	**R3 = R3 ADD R2**	**R2 = LOAD R1 + 0**	BNZ R4, boucle	R4 = R4 SUB 1	R1 = R1 ADD 4	R3 = R3 ADD R2	NOP	
T10	*Inst2-T3*	**R1 = R1 ADD 4**	**R3 = R3 ADD R2**	**R2 = LOAD R1 + 0**	BNZ R4, boucle	R4 = R4 SUB 1	R1 = R1 ADD 4	R3 = R3 ADD R2	
T11	*Inst2-T4*	**R4 = R4 SUB 1**	**R1 = R1 ADD 4**	**R3 = R3 ADD R2**	**R2 = LOAD R1 + 0**	BNZ R4, boucle	R4 = R4 SUB 1	R1 = R1 ADD 4	
T12	*Inst2-T5*	**R4 = R4 SUB 1**	**R1 = R1 ADD 4**	**R3 = R3 ADD R2**	**NOP**	**R2 = LOAD R1 + 0**	BNZ R4, boucle	R4 = R4 SUB 1	
T13	*Inst2-T6*	**R4 = R4 SUB 1**	**R1 = R1 ADD 4**	**R3 = R3 ADD R2**	**NOP**	**NOP**	**R2 = LOAD R1 + 0**	BNZ R4, boucle	
T14	*Inst2-T7*	**BNZ R4, boucle**	**R4 = R4 SUB 1**	**R1 = R1 ADD 4**	**R3 = R3 ADD R2**	**NOP**	**NOP**	**R2 = LOAD R1 + 0**	
T15	*Inst3-T1*	R2 = LOAD R1 + 0	**BNZ R4, boucle**	**R4 = R4 SUB 1**	**R1 = R1 ADD 4**	**R3 = R3 ADD R2**	**NOP**	**NOP**	
T16	*Inst3-T2*	R3 = R3 ADD R2	R2 = LOAD R1 + 0	**BNZ R4, boucle**	**R4 = R4 SUB 1**	**R1 = R1 ADD 4**	**R3 = R3 ADD R2**	**NOP**	
T17	*Inst3-T3*	R1 = R1 ADD 4	R3 = R3 ADD R2	R2 = LOAD R1 + 0	**BNZ R4, boucle**	**R4 = R4 SUB 1**	**R1 = R1 ADD 4**	**R3 = R3 ADD R2**	
T18	*Inst3-T4*	R4 = R4 SUB 1	R1 = R1 ADD 4	R3 = R3 ADD R2	R2 = LOAD R1 + 0	**BNZ R4, boucle**	**R4 = R4 SUB 1**	**R1 = R1 ADD 4**	
T19	*Inst3-T5*	R4 = R4 SUB 1	R1 = R1 ADD 4	R3 = R3 ADD R2	NOP	R2 = LOAD R1 + 0	**BNZ R4, boucle**	**R4 = R4 SUB 1**	
T20	*Inst3-T6*	R4 = R4 SUB 1	R1 = R1 ADD 4	R3 = R3 ADD R2	NOP	NOP	R2 = LOAD R1 + 0	**BNZ R4, boucle**	
T21	*Inst3-T7*	BNZ R4, boucle	R4 = R4 SUB 1	R1 = R1 ADD 4	R3 = R3 ADD R2	NOP	NOP	R2 = LOAD R1 + 0	
T22	*Inst4-T1*	R2 = LOAD R1 + 0	BNZ R4, boucle	R4 = R4 SUB 1	R1 = R1 ADD 4	R3 = R3 ADD R2	NOP	NOP	

Pipelining 1

No étage	Fonction
1	**Lit cache d'Instruction (LI)**
2	**Décode Instruction (DI)**
3	**Lit Registres (LR)**
4	**Exécute / Calcul Adresse (EX-CA)**
5	**Accès Cache de données (AC)**
6	**Accès Cache de données (AC)**
7	**Ecrit Registre (WR)**

Inst ≠ LOAD/STORE → 4

5, 6 : } 2 cycles

4, 5, 6 : } LOAD/STORE

- 3. Changer l'ordre des instructions pour obtenir un débit de 1 inst/ cycle :

	Instruction
1	**boucle: R2 = LOAD R1 + 0**
2	**R3 = R3 ADD R2**
3	**R1 = R1 ADD 4**
4	**R4 = R4 SUB 1**
5	**BNZ R4, boucle**

	Instruction
1	**boucle: R2 = LOAD R1 + 0**
2	**R1 = R1 ADD 4**
3	**R4 = R4 SUB 1**
4	**R3 = R3 ADD R2**
5	**BNZ R4, boucle**

Pipelining 1

No étage	Fonction
1	Lit cache d'Instruction (LI)
2	Décode Instruction (DI)
3	Lit Registres (LR)
4	Exécute / Calcul Adresse (EX-CA)
5	Accès Cache de données (AC)
6	Accès Cache de données (AC)
7	Ecrit Registre (WR)

Inst ≠ LOAD/STORE →

} 2 cycles

} LOAD/STORE

	Instruction	Commentaire
1	boucle: R2 = LOAD R1 – R4	// lit l'élément du tableau
2	R4 = R4 SUB 4	// ajoute à la somme
3	R3 = R3 ADD R2	// R1 = R1 + 4
4	BNZ R4, boucle	// BNZ (Branch Not Zero): boucle si R4 diff de 0

Calcule la somme de N élts d'1 tableau

$R4_{init}$ = 4N

$R1_{init}$ = A + 4N

$R3_{init}$ = 0 *(à la fin, il contient le résultat)*

Eléments du tableau sur 4 Octets

- 4. Nouveau programme (pour N grand)
  - a. Débit d'exécution ?

a) Pour supprimer les 2 bulles introduites par un LOAD, il faut faire suivre ce LOAD de 2 instructions indépendantes du résultat du LOAD. Ici, le LOAD n'est suivi que d'1 seule instruction indépendante du résultat du LOAD, ce qui introduit 1 *bulle* dans le pipeline.

Le débit d'exécution du programme, comportant 4 instructions et coûtant 5 cycles, est donc de :

$$\frac{4}{4+1} = \frac{4}{5} = 0.8 \text{ instructions / cycle.}$$

Pipelining 1

No étage	Fonction
1	Lit cache d'Instruction (LI)
2	Décode Instruction (DI)
3	Lit Registres (LR)
4	Exécute / Calcul Adresse (EX-CA)
5	Accès Cache de données (AC)
6	Accès Cache de données (AC)
7	Ecrit Registre (WR)

Inst ≠ LOAD/STORE →

} 2 cycles

} LOAD/STORE

	Instruction	Commentaire
1	boucle: R2 = LOAD R1 – R4	// lit l'élément du tableau
2	R4 = R4 SUB 4	// R4 = R4 - 4
3	R3 = R3 ADD R2	// ajoute à la somme
4	BNZ R4, boucle	// BNZ (Branch Not Zero): boucle si R4 diff de 0

Calcule la somme de N élts d'1 tableau
$R4_{init}$ = 4N
$R1_{init}$ = A + 4N
$R3_{init}$ = 0 *(à la fin, il contient le résultat)*
Eléments du tableau sur 4 Octets

- 4. Nouveau programme (pour N grand)
  - b. Ce programme est-il plus performant que celui de 2 ? Et de celui modifié à 3 ?

b) Pour comparer 2 programmes différents effectuant le même travail et n'ayant pas le même nombre d'instructions, il ne faut pas regarder le débit en instructions par cycle mais le temps total pour effectuer le travail.

Ici, on peut se contenter de comparer le nombre de cycles par itération, puisque chaque itération traite un élément du tableau.

Avec le programme de la question **2**, chaque itération coûte 7 cycles par boucle.

Le nouveau programme ne comporte que 4 instructions au lieu de 5. 1 bulle est générée à chaque itération. Chaque itération coûte donc 5 cycles. Le nouveau programme est plus performant que celui de la question **2** et aussi performant que celui de la question **3** qui compte 5 cycles aussi mais sans *bulle*.

Pipelining 1

	Instruction
1	boucle: R2 = LOAD R1 – R4
2	R4 = R4 SUB 4
3	R3 = R3 ADD R2
4	BNZ R4, boucle

- 5. Pipeline 2 et le programme du 4
  - La performance du prgm de 4 sur le P est-elle sup ou inf à celle sur P1 ?

No étage	Fonction *(Pipeline1)*
1	Lit cache d'Instruction (LI)
2	Décode Instruction (DI)
3	Lit Registres (LR)
4	Exécute / Calcul Adresse (EX-CA)
5	Accès Cache de données (AC)
6	Accès Cache de données (AC)
7	Ecrit Registre (WR)

No étage	Fonction *(Pipeline2)*
1	Lit cache d'Instruction (LI)
2	Décode Instruction (DI)
3	Lit Registres (LR)
4	Calcul Adresse (CA)
5	Accès Cache de données (AC)
4	Exécute / Accès Cache (EX-AC)
7	Ecrit Registre (WR)

. **Pipeline initial (P1) :** 2 *bulles* sont introduites si le résultat de l'exécution d'une instruction LOAD/STORE est utilisé comme opérande dans l'instruction suivante.

. **Nouveau Pipeline (P2) :** 2 *bulles* sont introduites avec le calcul d'adresse pour chaque opérande d'une instruction LOAD/STORE dépendant de l'instruction précédente.

Si les 2 instructions précédant un LOAD/STORE sont indépendantes du calcul d'adresse de ou des opérandes de l'instruction LOAD/STORE, aucune *bulle* n'est introduite.

Pipelining 1

	Instruction
1	boucle: R2 = LOAD R1 – R4
2	R4 = R4 SUB 4
3	R3 = R3 ADD R2
4	BNZ R4, boucle

- 5. Pipeline 2 et le programme du 4
  - La performance du prgm de 4 sur le P est-elle sup ou inf à celle sur P1 ?

No étage	Fonction *(Pipeline1)*
1	Lit cache d'Instruction (LI)
2	Décode Instruction (DI)
3	Lit Registres (LR)
4	Exécute / Calcul Adresse (EX-CA)
5	Accès Cache de données (AC)
6	Accès Cache de données (AC)
7	Ecrit Registre (WR)

No étage	Fonction *(Pipeline2)*
1	Lit cache d'Instruction (LI)
2	Décode Instruction (DI)
3	Lit Registres (LR)
4	Calcul Adresse (CA)
5	Accès Cache de données (AC)
4	Exécute / Accès Cache (EX-AC)
7	Ecrit Registre (WR)

Pipeline P1 : 2 bulles sont introduites dans le pipeline si l'instruction *immédiatement après* un LOAD/STORE *utilise* comme *opérande* le *résultat* du LOAD/STORE.

Pipeline P2 : 2 bulles sont insérées dans le pipeline si l'instruction *immédiatement avant* un LOAD/STORE *met à jour* un *opérande* du LOAD/STORE.

Dans le cas du programme de la question **4**, le calcul d'adresse du LOAD dépend de R1 et de R4. Seul R4 est mis à jour dans la boucle par l'instruction SUB. Comme il y a 2 instructions intercalées entre le SUB et le LOAD, aucune *bulle* n'est introduite ici. Chaque itération coûte 4 cycles sur le nouveau pipeline (P2), au lieu de 5 cycles sur l'ancien pipeline (P1).

Pipelining 2

No étage	Fonction *(Pipeline1)*
1	Lit cache d'Instruction (LI)
2	Décode Instruction (DI)
3	Lit Registres (LR)
4	Exécute / Calcul Adresse (EX-CA)
5	Accès Cache de données (AC)
6	Accès Cache de données (AC)
7	Ecrit Registre (WR)

Soit le pipeline d'instructions (**P1**) représenté ci-dessous, ayant un débit théorique maximum égal à 1 instruction/cycle :

Un cycle est le temps nécessaire au traitement d'un étage du pipeline.

On suppose que toutes les instructions, sauf les instructions LOAD et STORE, sont exécutées à l'étage 4. Pour les instructions LOAD/STORE, le calcul d'adresse du/des opérandes du LOAD/STORE se fait à l'étage 4 et l'accès au cache de données est pipeliné sur 2 cycles (étages 5 et 6). Pour les instructions qui ne sont pas des LOAD/STORE, les étages 5 et 6 se comportent comme des étages vides.

On suppose qu'il y a un mécanisme de *bypass* permettant de transmettre le résultat d'une instruction aux instructions suivantes sans attendre l'écriture registre. Si un opérande source d'une instruction n'est pas disponible au moment où celle-ci va rentrer dans l'étage 4, les étages 1 à 3 sont bloqués jusqu'à ce que l'opérande indisponible soit accessible via le mécanisme de *bypass*, ce qui conduit à l'insertion de *bulles* dans le pipeline.

2 bulles sont insérées dans le pipeline P1 si l'instruction immédiatement après un LOAD/STORE utilise comme opérande le résultat du LOAD/STORE.

On supposera un prédicteur de branchement parfait. On supposera également que toutes les lectures d'instructions font des *hits* dans le cache d'instructions et que tous les LOAD/STORE font des *hits* dans le cache de données (cela signifie qu'il n'y a que des *cache-hits* et pas de *cache-miss*, c'est-à-dire que les accès cache sont toujours tels que le cache respectivement d'instructions / données contient les cibles, instructions / données).

Pipelining 2

No étage	Fonction *(Pipeline1)*
1	Lit cache d'Instruction (LI)
2	Décode Instruction (DI)
3	Lit Registres (LR)
4	Exécute / Calcul Adresse (EX-CA)
5	Accès Cache de données (AC)
6	Accès Cache de données (AC)
7	Ecrit Registre (WR)

On considère le programme ci-dessous écrit dans un assembleur donné, qui calcule la somme des N éléments d'un tableau contenant des entiers. Le registre R4 est initialisé avec N, le registre R1 est initialisé avec l'adresse A du 1er élément du tableau et le registre R3 contenant en fin d'exécution le résultat de la somme est initialisé à 0. Les éléments du tableau sont stockés sur 2 octets. Enfin, le calcul de la somme est interrompu si celle-ci égale une valeur seuil donnée. Le registre R6 est initialisé avec ce seuil :

```
1:      boucle:  R2 = LOAD R1+0        // lit element du tableau
2:               R4 = R4 SUB 1         // R4 = R4 - 1
3:               R3 = R3 ADD R2        // ajoute a la somme
4:               R7 = R3 SUB R6        // R7 = R3 - R6
5:               BZ R7, fin            // BZ (Branch Zero) : saut à l'étiquette fin si R7 = 0
6:               R1 = R1 ADD 2         // R1 = R1 + 2
7:               BNZ R4, boucle        // BNZ (Branch Not Zero) : boucle si R4 ≠ 0
8:      fin:     …                     // fin de traitement
```

1. En supposant N grand et en partant d'une configuration où le calcul de la somme n'est pas interrompu, quel est le débit d'exécution de la boucle de programme en *instructions par cycle* ?

2. On modifie ainsi le programme précédent, le registre R5 étant initialisé à 0 :

```
1:      boucle:  R2 = LOAD R1+0        // lit element du tableau
2:               R3 = R3 ADD R2        // ajoute a la somme
3:               R5 = R5 ADD R2        // recalcul et sauvegarde du resultat dans R5
4:               R7 = R3 SUB R6        // R7 = R3 - R6
5:               BZ R7, fin            // BZ (Branch Zero) : saut à l'étiquette fin si R7 = 0
6:               R1 = R1 ADD 2         // R1 = R1 + 2
7:               R4 = R4 SUB 1         // R4 = R4 - 1
8:               BNZ R4, boucle        // BNZ (Branch Not Zero) : boucle si R4 ≠ 0
9:      fin:     …                     // fin de traitement
```

Dans les mêmes hypothèses que précédemment, indiquer le débit d'exécution de la boucle de programme en *instructions par cycle* ?

Pipelining 2

No étage	Fonction *(Pipeline1)*
1	Lit cache d'Instruction (LI)
2	Décode Instruction (DI)
3	Lit Registres (LR)
4	Exécute / Calcul Adresse (EX-CA)
5	Accès Cache de données (AC)
6	Accès Cache de données (AC)
7	Ecrit Registre (WR)

Pipeline P1

1. En négligeant le temps de remplissage du pipeline, lorsqu'un LOAD/STORE est immédiatement suivi d'une instruction utilisant le résultat du LOAD/STORE, 2 bulles sont insérées dans le pipeline.

Dans le programme considéré, 1 bulle est donc introduite du fait de l'instruction LOAD car 1 instruction indépendante du résultat du LOAD est intercalée entre le LOAD et l'instruction 3 dépendante du résultat du LOAD (1 bulle sur les 2 insérées par le LOAD a pu être éliminée).

Le corps de la boucle comporte 7 instructions, le débit d'exécution vaut : $\frac{7}{7+1} = \frac{7}{8} = 0.875$ *instructions / cycle.*

2. L'instruction 1 (LOAD) est suivie immédiatement d'une instruction utilisant le résultat du LOAD, 2 bulles sont insérées dans le pipeline. L'instruction 3 n'introduit pas de bulles supplémentaires car les 2 bulles précédentes sont éliminées après l'instruction 2.

En négligeant le temps de remplissage du pipeline : le corps de la boucle comporte 8 instructions, 2 bulles sont donc générées toutes les 8 instructions.

Le débit d'exécution vaut : $\frac{8}{8+2} = \frac{8}{10} = \frac{4}{5} = 0.8$ *instructions / cycle.*

EISTI – ING1

Pipelining

ADO – TP 4

Pipelining 1

No étage	Fonction *(Pipeline1)*
1	Lit cache d'Instruction (LI)
2	Décode Instruction (DI)
3	Lit Registres FP (LR)
4	Exécute FP / Accès Cache de données (EX-AC)
5	Exécute FP / Accès Cache de données (EX-AC)
6	Exécute FP / Accès Cache de données (EX-AC)
7	Ecrit Registre FP (WR)

Le débit théorique maximum de ce pipeline est de 1 instruction/cycle.

On suppose un prédicteur de branchements parfait et les instructions déjà dans les caches.

Les instructions en virgule flottante (FP) sont pipelinées sur 3 cycles.

Une instruction FP dépendant de l'instruction immédiatement précédente doit attendre 2 cycles avant de rentrer dans le 1er étage d'exécution
→ 2 bulles sont insérées si une instruction utilise en opérande le résultat de l'instruction immédiatement avant.

On veut calculer l'expression : $ay^3 + by^2 + cy + d$. On propose 2 méthodes pour implémenter ce calcul :

- méthode 1 (de Horner) : $d + y(c + y(b + ya))$ opérations : 3 × ; 3 +

- méthode 2 : $(d + cy) + ((y \times y) \times (ay + b))$ opérations : 4 × ; 3 +

On remarquera que la méthode 2 demande 1 multiplication de plus que la méthode 1 (de Horner).

On supposera que y,a,b,c,d se trouvent déjà dans les registres flottants respectifs F0, F1, F2, F3, F4 et que les autres registres sont libres. On veut obtenir le résultat dans le registre F5.

Pipelining 1

No étage	Fonction *(Pipeline1)*
1	Lit cache d'Instruction (LI)
2	Décode Instruction (DI)
3	Lit Registres FP (LR)
4	Exécute FP / Accès Cache de données (EX-AC)
5	Exécute FP / Accès Cache de données (EX-AC)
6	Exécute FP / Accès Cache de données (EX-AC)
7	Ecrit Registre FP (WR)

a) Comprendre les pseudo-codes assembleur proposés pour les 2 méthodes :

Pseudo-codes assembleur

Rappel : Initialisation :

F0 = y
F1 = a
F2 = b
F3 = c
F4 = d

Méthode 1 (de Horner) : d + y(c + y(b + ya))

Le programme suivant implémente la méthode 1 (de Horner) :

```
1 :   F5 = F1 FMUL F0      // F5 = ay
2 :   F5 = F5 FADD F2      // F5 = ay + b
3 :   F5 = F5 FMUL F0      // F5 = (ay + b)y
4 :   F5 = F5 FADD F3      // F5 = (ay + b)y + c
5 :   F5 = F5 FMUL F0      // F5 = [(ay + b)y + c]y
6 :   F5 = F5 FADD F4      // F5 = [(ay + b)y + c]y + d
```

Méthode 2 : (d + cy) + ((y × y) × (ay + b))

La méthode 2 peut être implémentée avec le programme suivant :

```
1 :   F5 = F0 FMUL F1      // F5 = ya
2 :   F6 = F0 FMUL F3      // F6 = yc
3 :   F7 = F0 FMUL F0      // F7 = y^2
4 :   F5 = F5 FADD F2      // F5 = ya + b
5 :   F6 = F6 FADD F4      // F6 = yc + d
6 :   F5 = F5 FMUL F7      // F5 = (ya + b)y^2
7 :   F5 = F5 FADD F6      // F5 = (ya + b)y^2 + yc + d
```

b) Quelle méthode est la plus performante sur le pipeline considéré ?

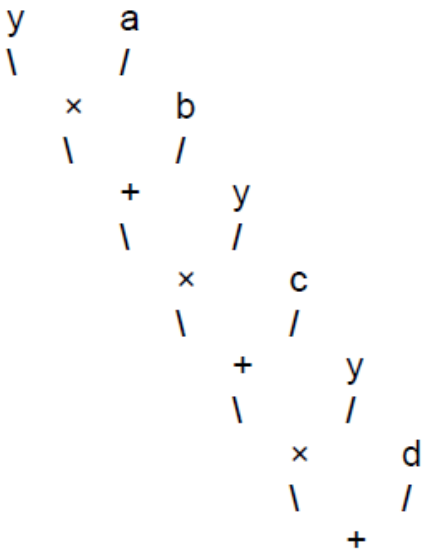
Pipelining 1

No étage	Fonction *(Pipeline1)*
1	Lit cache d'Instruction (LI)
2	Décode Instruction (DI)
3	Lit Registres FP (LR)
4	Exécute FP / Accès Cache de données (EX-AC)
5	Exécute FP / Accès Cache de données (EX-AC)
6	Exécute FP / Accès Cache de données (EX-AC)
7	Ecrit Registre FP (WR)

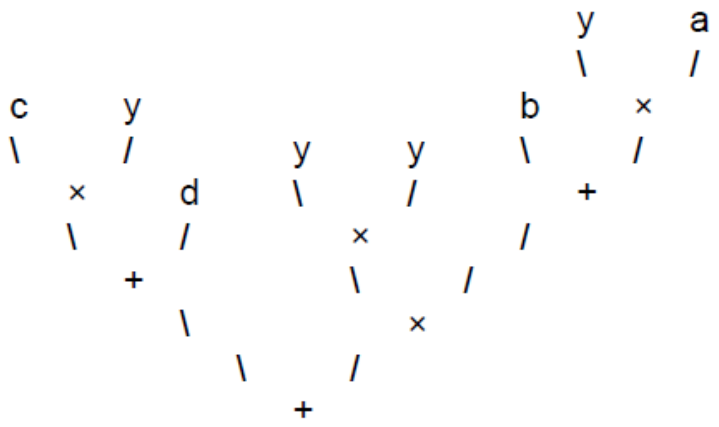
1. Pipelining (1)

Une remarque importante est que le compilateur doit effectuer le calcul exactement comme spécifié par le programme car l'addition et la multiplication en virgule flottante ne sont pas des opérations associatives.

On peut représenter le calcul par un arbre de dépendance, comme représenté ci-dessous :



Méthode 1 (de Horner) : d + y(c + y(b + ya))



Méthode 2 : (d + cy) + ((y × y) × (ay + b))

Pipelining 1

No étage	Fonction *(Pipeline1)*
1	Lit cache d'Instruction (LI)
2	Décode Instruction (DI)
3	Lit Registres FP (LR)
4	Exécute FP / Accès Cache de données (EX-AC)
5	Exécute FP / Accès Cache de données (EX-AC)
6	Exécute FP / Accès Cache de données (EX-AC)
7	Ecrit Registre FP (WR)

a) Pseudo-codes assembleur

Rappel : Initialisation :

F0 = y
F1 = a
F2 = b
F3 = c
F4 = d

Méthode 1 (de Horner) : d + y(c + y(b + ya))

Le programme suivant implémente la méthode 1 (de Horner) :

```
1 :   F5 = F1 FMUL F0     // F5 = ay
2 :   F5 = F5 FADD F2     // F5 = ay + b
3 :   F5 = F5 FMUL F0     // F5 = (ay + b)y
4 :   F5 = F5 FADD F3     // F5 = (ay + b)y + c
5 :   F5 = F5 FMUL F0     // F5 = [(ay + b)y + c]y
6 :   F5 = F5 FADD F4     // F5 = [(ay + b)y + c]y + d
```

Méthode 2 : (d + cy) + ((y × y) × (ay + b))

La méthode 2 peut être implémentée avec le programme suivant :

```
1 :   F5 = F0 FMUL F1     // F5 = ya
2 :   F6 = F0 FMUL F3     // F6 = yc
3 :   F7 = F0 FMUL F0     // F7 = y^2
4 :   F5 = F5 FADD F2     // F5 = ya + b
5 :   F6 = F6 FADD F4     // F6 = yc + d
6 :   F5 = F5 FMUL F7     // F5 = (ya + b)y^2
7 :   F5 = F5 FADD F6     // F5 = (ya + b)y^2 + yc + d
```

Pipelining 1

No étage	Fonction *(Pipeline1)*
1	Lit cache d'Instruction (LI)
2	Décode Instruction (DI)
3	Lit Registres FP (LR)
4	Exécute FP / Accès Cache de données (EX-AC)
5	Exécute FP / Accès Cache de données (EX-AC)
6	Exécute FP / Accès Cache de données (EX-AC)
7	Ecrit Registre FP (WR)

Méthode 1 (de Horner) : d + y(c + y(b + ya))

Le programme suivant implémente la méthode 1 (de Horner) :

```
1 :    F5 = F1 FMUL F0      // F5 = ay
2 :    F5 = F5 FADD F2      // F5 = ay + b
3 :    F5 = F5 FMUL F0      // F5 = (ay + b)y
4 :    F5 = F5 FADD F3      // F5 = (ay + b)y + c
5 :    F5 = F5 FMUL F0      // F5 = [(ay + b)y + c]y
6 :    F5 = F5 FADD F4      // F5 = [(ay + b)y + c]y + d
```

Méthode 1 (de Horner)
Chacune des 5 instructions (2 à 6) dépend de l'instruction précédente. Ce programme va occuper l'étage du décodage (Décode Instruction) pendant :
Nb de Cycles = 6 + (5 x 2) = 16 cycles
(= 6 inst + 10 (5x2) bulles = 16 cycles

Méthode 2 : (d + cy) + ((y × y) × (ay + b))

La méthode 2 peut être implémentée avec le programme suivant :

```
1 :    F5 = F0 FMUL F1      // F5 = ya
2 :    F6 = F0 FMUL F3      // F6 = yc
3 :    F7 = F0 FMUL F0      // F7 = y^2
4 :    F5 = F5 FADD F2      // F5 = ya + b
5 :    F6 = F6 FADD F4      // F6 = yc + d
6 :    F5 = F5 FMUL F7      // F5 = (ya + b)y^2
7 :    F5 = F5 FADD F6      // F5 = (ya + b)y^2 + yc + d
```

Méthode 2
Les instructions 1, 2, 3 sont indépendantes
L'instruction 4 dépend de l'instruction 1 mais la pénalité de 2 cycles est couverte par 2 et 3
De même la dépendance entre les instructions 2 et 5 est couverte par 3 et 4
La dépendance entre les instructions 4 et 6 n'est que partiellement couverte par la 5, donc on paye 1 cycle de pénalité
Quant à la dépendance entre les instructions 6 et 7, elle n'est pas couverte, on paye donc 2 cycles dde pénalité.
Au total, le programme occupe l'étage de décodage (Décode Instruction) pendant :
Nb de Cycles = 7 + (2 + 1) = 10 cycles
(= 7 inst + 3 (2+1) bulles = 10 cycles

La méthode 2 est plus performante que la méthode 1

Pipelining 2

2. Pipelining (2)

1. Pourquoi le pipelining améliore t-il la performance ?

2. Quelles sont les limites de l'amélioration de performance apportée par le pipelining ?

3. Calcul du temps d'exécution d'une séquence d'instructions :

Exemple : **Rappel :** latence = durée d'exécution, temps de traitement, d'une instruction

Supposons qu'un processeur non pipeliné possedant un temps de cycle (temps de traitement d'1 instruction) de 25 ns soit divisé en 5 étages de pipeline de latences respectives de 5, 7, 3, 6 et 4 ns. La latence de latch du pipeline (registre de sortie de chaque étage du pipeline) est de 1 ns. Le pipeline est supposé sans délais de branchement (prédiction de branchement parfaite, sans aléas).

3.1. Quelle est le temps de cycle (temps de traitement d'1 étage du pipeline) du processeur pipeliné ?

3.2. Quelle est la latence totale du pipeline (temps de traitement de tous les étages du pipeline) ?

Pipelining 2

1.

Dans un processeur non pipeliné, chaque instruction doit entiérement être exécutée avant que l'exécution de la suivante ne puisse commencer.

Dans un processeur pipeliné, l'exécution de l'instruction est décomposée en plusieurs étapes correspondant aux étages du pipeline.

Dès que le premier étage du pipeline est franchi par une instruction, l'instruction suivante peut entamer son exécution en y accédant a son tour pendant que l'instruction précédente continue son exécution. Ceci augmente la vitesse d'exécution.

Le pipelining devise le chemin de données d'un processeur en étages séparés par des latches (registres).

Dans un processeur non pipeliné, une instruction doit être capable de parcourir la totalité du chemin de données en un seul cycle d'horloge.

Dans un processeur pipeliné, l'instruction doit simplement pouvoir passer à l'étage suivant à chaque cycle, ce qui permet d'avoir un cycle d'horloge beaucoup plus court.

Dans un processeur pipeliné à *n* étages, le débit théorique maximal est multiplié par *n* par rapport à celui d'un processeur non pipeliné, mais le temps calcul d'une instruction est identique pour les 2 architectures.

Pipelining 2

2.

Il existe 2 limites principales.

La 1ère tient au fait que, à mesure que le nombre d'étages du pipeline augmente, la fraction de temps que représente la latence (temps de traitement) du latch de chaque étage devient plus importante - en valeur relative.

La 2nde limite provient des dépendances des données et des délais de branchement.

Les instructions qui dépendent des résultats d'autres instructions doivent attendre que ces derniéres aient terminé leur exécution et provoquent dans ce cas des blocages dans le pipeline (*bulles*).

Par ailleurs, les instructions qui suivent les branchements doivent attendre que ces branchements se terminent pour pouvoir être insérées dans le pipeline. Ces différents retards conduisent le pipeline à exécuter moins d'1 instruction par cycle en moyenne, ce qui implique qu'une plus grande partie du temps processeur sera consacré à attendre les déblocages du pipeline.

3.1. L'étage le plus long posséde une latence de 7 ns. En ajoutant le délai de 1 ns pour le latch de pipeline (latch de chaque étage de pipeline), on obtient un temps de cycle de 8 ns (si on a des étages de différentes latences, on prend toujours la latence de l'étage le plus long).

3.2. Puisqu'il y a 5 étages, la latence totale du pipeline est de : 8 ns * 5 étages = 40 ns.

Note :

Le temps d'exécution *réel* (en nombre de cycles) d'une séquence d'instructions d'un pipeline sans délais de branchement ni dépendance de données est :

nombre d'étages + nombre d'instructions – 1.