



Architecture des microprocesseurs (SN330-SN331)

Cahier d'exercices (TD)

Table des matières

1	Vue d'ensemble d'un ordinateur	3
1.1	Généralités	3
1.2	Premiers pas assembleur : codage d'instructions	4
2	Programmation assembleur 1	5
2.1	Assemblage, etc	5
2.2	Petits programmes en assembleur RISCV	5
3	Programmation assembleur RISCV : pile	6
3.1	Manipulation de pile	6
3.2	Pile et fonctions	8
4	Mémoire cache	9
4.1	Mémoire cache	9
4.2	Somme de deux vecteurs	9
4.3	Somme des éléments d'une matrice	10
4.4	Mémoire cache - petits calculs	11
5	Pipeline	12
5.0.1	Calcul sur des pipelines	12
5.0.2	Calibrage d'un pipeline	12
5.1	Pipeline à 4 étages	12
6	Mémoires	14
6.1	Étude d'une hiérarchie mémoire à trois niveaux	14
6.2	Pagination	14
6.3	Pagination exo 2	16

TD 1

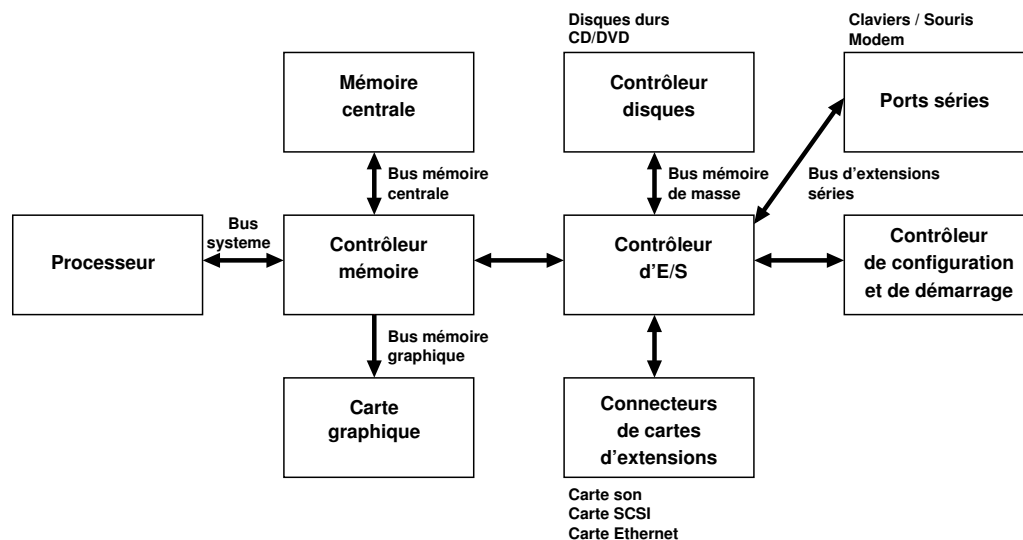
Vue d'ensemble d'un ordinateur

1.1 Généralités

EXERCICE #1 ► Petits calculs autour d'une carte mère

Les tailles des mémoires sont des puissances de deux ou des multiples de telles puissances. On utilise parfois les préfixes SI en leur associant la puissance de 2 la plus proche : par exemple 1024 octets est souvent abusivement noté 1 ko, bien que $1\text{ ko} = 1000\text{ o} \dots$ La norme CEI-60027-2 définit des *préfixes binaires* pour éviter la confusion. Ainsi, 1024 octets = 1 kibi-octet = 1 kio. On utilisera ces préfixes binaires par la suite.

La *carte mère* d'un PC regroupe des fonctionnalités de base d'un ordinateur : processeur, mémoire centrale et gestion des E/S.



Bande passante d'un bus La bande passante d'un bus, aussi appelée débit crête, est la quantité de données pouvant circuler sur ce bus par unité de temps. Sur la carte mère d'un PC, le bus reliant le processeur au contrôleur mémoire est le bus système, souvent appelé *Front Side Bus* (FSB). Supposons ici que le FSB d'un certain ordinateur est capable d'assurer le transfert de 8 octets à la fréquence de 400 MHz.

Un contrôleur mémoire prend à sa charge les échanges entre le processeur et la mémoire centrale, entre le processeur et le contrôleur d'E/S, et entre le processeur et la mémoire vidéo. Le contrôleur mémoire peut en outre mettre en place un accès direct entre le contrôleur d'E/S et la mémoire centrale ou la mémoire vidéo (accès DMA pour *Direct Memory Access*) : le contrôleur d'E/S pourra par exemple transférer directement des données d'un périphérique à la mémoire vidéo sans qu'elles transitent par le processeur.

- 1) Quelle est la bande passante du bus FSB considéré exprimée en Go/s? Rappelons que $1\text{ Go} = 10^9\text{ o}$.
- 2) Quelle est la bande passante du bus FSB considérée exprimée en Gio/s? Rappelons que $1\text{ Gio} = 2^{30}\text{ o}$.
- 3) Supposons que le bus mémoire centrale permet le transfert de mots de 32 bits à la fréquence de 266 MHz. Quelle est la bande passante du bus mémoire centrale en Go/s?
- 4) Que penser de la différence de bande passante entre le bus de la mémoire centrale et celle du FSB?

Lecture d'un film Un film est lu à partir d'un disque dur, connecté via le bus IDE au contrôleur de disque. Le film est non-compressé, et constitué d'une succession d'images de 512×384 pixels en 256 couleurs. On suppose que le défilement des images se fait en 24 images par seconde.

- 1) Quels sont les bus utilisés pour le transfert?

- 2) Quel est le débit (en Mo/s) requis pour le transfert du film du disque dur à la mémoire vidéo ?
- 3) Supposons que le bus de la mémoire vidéo a une bande passante identique à celle du bus de la mémoire centrale. Quelle est la part (en pourcentage) de la bande passante du bus de la mémoire vidéo est consommée par la lecture du film ?

EXERCICE #2 ► Taille du bus, volume de mémoire centrale

On suppose ici que le bus entre le processeur et la mémoire centrale comporte 32 fils d'adresse.

1. Si à chaque adresse de la mémoire centrale correspond un octet :
 - (a) Quel est le nombre d'octets adressables ?
 - (b) Quelle est la taille maximale de la mémoire ? Exprimez votre réponse en octet puis en Gio.
 - (c) Combien de fils de données doit comporter le bus ?
2. Si à chaque adresse correspond un mot de 32 bits :
 - (a) Quel est le nombre de mots adressables ?
 - (b) Quelle est la taille maximale de la mémoire ? Exprimez votre réponse en octet puis en Gio.
 - (c) Combien de fils de donnée doit comporter le bus ?

1.2 Premiers pas assembleur : codage d'instructions

EXERCICE #3 ► Jeu d'instruction, codage, exécution d'un programme

On se place sur un processeur hypothétique, qui accède à une mémoire centrale dans laquelle la taille d'une case mémoire est de 2 octets. Ce processeur dispose des registres suivants :

- **IR**, le registre d'instruction et **PC**, le compteur de programme (pas encore vus en cours) ;
- **A** (comme accumulateur), un registre temporaire pour le stockage du résultat des opérations.

Les instructions sont codées comme suit :

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opcode								adresse							

On détaille les instructions suivantes :

mnémonique	opcode		opération réalisée
LOAD	$(0001)_2$	$(1)_H$	charger le mot dont l'adresse est donnée dans A
STORE	$(0010)_2$	$(2)_H$	stocker le mot contenu dans A à l'adresse donnée
ADD	$(0101)_2$	$(5)_H$	ajouter l'entier naturel à l'adresse donnée à A .

- 1) Combien d'instructions, suivant le codage indiqué ci-dessus, peut compter le jeu d'instruction ?
- 2) Quel est le nombre maximal d'adresses auxquelles une telle instruction peut faire référence ?
- 3) On considère le morceau de programme suivant, écrit en « langage d'assemblage » :

LOAD $(130)_H$
 ADD $(131)_H$
 ADD $(132)_H$
 STORE $(133)_H$

Que fait ce programme ?

- 4) Traduisez ce programme en langage machine, et représentez le dans la mémoire centrale en plaçant la première instruction à l'adresse $(100)_H$.
- 5) On suppose que le contenu des cases mémoires $(130)_H$ à $(133)_H$ est initialement le suivant :

$(130)_H$	$(0002)_H$
$(131)_H$	$(0003)_H$
$(132)_H$	$(0001)_H$
$(133)_H$	$(0022)_H$

Représentez le contenu des cases mémoires $(130)_H$ à $(133)_H$, ainsi que le registre **A**, après l'exécution de chacune des instructions considérées.

- 6) Proposez un morceau de programme pour échanger le contenu des cases mémoires $(130)_H$ et $(131)_H$. Écrivez votre proposition en langage d'assemblage.

TD 2

Programmation assembleur 1

Tous les exercices de ce TD sont à faire en utilisant le langage d'assemblage de RISC-V. On se reportera à la mini documentation RISC-V fournie en cours.

2.1 Assemblage, etc

EXERCICE #1 ► Assemblage

Assembler l'instruction RISC-V `mv t1, s1`.

EXERCICE #2 ► Désassemblage

Désassembler l'instruction suivante : `0x00810383`

2.2 Petits programmes en assembleur RISC-V

EXERCICE #3 ► Intervalle de valeurs

Écrire un programme qui initialise le registre t_0 à 1 et l'incrémente jusqu'à ce qu'il soit égal à 8.

EXERCICE #4 ► Somme

Écrire un programme qui calcule la somme des 10 premiers entiers positifs ($0 + \dots + 9$).

EXERCICE #5 ► Étoiles

Dessiner des carrés et des étoiles, avec n un paramètre stocké quelque part en mémoire. Par exemple, pour $n=3$, cela donne :

```
***          *
***          * *
***          * * *
```

On fera l'hypothèse de l'existence d'une fonction `print_char`. Le caractère '*' a pour code ascii 42.

EXERCICE #6 ► Nombre de bits non-nuls

On désire écrire un programme qui compte le nombre de bits non nuls d'un entier 64 bits n chargé à partir de la mémoire.

- Que calcule l'expression $n \& 4$?
- Implémenter le pseudo-code suivant en RISC-V.

```
c=0
while(t2 >= 0) {
    if(n & t2 == 1) c++;
    t2=t2*2;
}
```

EXERCICE #7 ► Parcours d'un tableau

Écrire une routine qui ajoute 1 à chacune des cases d'un tableau d'entiers 64 bits dont on connaît l'adresse de début (dans a_0) et la taille (dans a_1).

TD 3

Programmation assembleur RISCv : pile

Un lien intéressant <https://riscv-programming.org/book/riscv-book.html>

3.1 Manipulation de pile

EXERCICE #1 ► Renversement de chaîne

Où la structure de données pile rencontre une pile d'exécution.

Le but de cet exercice est d'écrire en langage d'assemblage RISCv une routine permettant de renverser une chaîne de caractères terminée par le caractère '`\0`' (dont le code ASCII est l'entier 0), *via* la pile. Par exemple, le mot miroir de la chaîne `saper` est la chaîne `repas`. La pile utilisée sera la pile d'exécution manipulée à l'aide du registre `sp`. Le principe est le suivant :

- initialisation : on empile un '`\0`', qui permettra de détecter qu'il faudra arrêter de dépiler;
- boucle d'empilement : tant que l'on n'a pas rencontré le '`\0`' de la chaîne initiale (à renverser), on empile un à un ses caractères.
- boucle de dépilement : tant que l'on n'a pas rencontré le '`\0`' dans la pile, on dépile un à un les caractères de la pile, et on les range dans la chaîne de destination (renversée).
- on n'oublie pas d'ajouter un '`\0`' final à la chaîne de destination.

On n'oubliera pas, au début de la routine, d'empiler l'adresse, et de la dépiler à la fin de la routine. *Affichage* : on considère qu'on dispose d'une routine `print_char` qui affiche à l'écran le caractère dont le code ascii est contenu dans le registre `a0`.

1) Remplir le code du *main* ci-dessous :

```
1 # renversement d'une chaîne de caractères en RISCv
2     .text
3     .globl main
4 main: # Programme principal
5     # stockage de l'adresse de retour sur la pile.
6
7
8
9     # gestion des paramètres d'appel de rev
10
11
12
13     # appel
14
15
16     # impression de la chaîne avec println_string
17
18
19
20     # On rend la main au système (attention à l'état de la pile)
21
22
23
24     ret    # fin du programme
25
```

```

26 # Ci-dessous les données.
27     .section .data
28 machaine:
29     .string "saper"
30 chaineresu:
31     .space 64

```

2) Donner un pseudo-code pour la boucle d'empilement telle qu'expliquée ci-dessus. Vous utiliserez t2 comme pointeur (ie qui *stocke des adresses*) pour parcourir la chaîne initiale, en supposant que l'adresse de son premier caractère est contenue dans a0. Utiliser t1 pour le stockage temporaire des caractères.

3) Compléter maintenant le code de la routine :

```

1 # routine rev, qui renverse une chaine de caractères
2 #paramètres formels:
3 #     a0 contient l'adresse de la chaine
4 #     a1 contient l'adresse de la chaîne de sortie
5 # a0, a1, doivent être préservés.
6     # Gestion de la pile pour l'appel de routine
7
8
9     # Maintenant on code l'algo
10    ## préparation de pile: on "empile '\0'", puis les caractères
11    ## t1 contient des caractères
12    ## t2 contient des @de caractères.
13    # empilement de 0 :
14
15
16    ...      # t2 récupère adresse du premier caractère
17 loop1:    # compléter la boucle de stockage sur la pile
18    ...
19
20
21
22
23
24    addi t2, t2, 1 # +1 sur l'adresse car caractères
25    j loop1
26 endloop1:
27    # maintenant on dépile (avec les mêmes t1, t2)
28    .... # t2 <-adresse init de la chaîne resultat
29 loop2:
30    ...
31
32
33
34
35    addi t2, t2, 1
36    j loop2
37 endloop2:
38    sd zero, 0(t2) # fin de chaîne
39    # on a dépilé autant de fois que d'éléments, ouf!
40    # Fin de l'algo
41    ## Gestion de la pile pour le retour de routine
42    ...
43
44

```

```

45
46         ret

```

EXERCICE #2 ► Bonus (pas simple) : calculette à pile

Écrire une fonction assembleur qui évalue une expression préfixée :

* + 1 2 4

en utilisant une pile.

3.2 Pile et fonctions**EXERCICE #3 ► Affichage et récursivité : countdown!**

Affichage : on considère qu'on dispose d'une routine `print_char` qui affiche à l'écran le caractère dont le code ascii est contenu dans le registre `a0`.

Il s'agit d'écrire les sous-routines `affn0` et `aff0n` de manière à ce qu'elles affichent respectivement les chiffres décimaux de n à 0 et de 0 à n (On supposera toujours $n \geq 0$). Le programme devra donc afficher :

```

76543210
01234567

```

Les deux routines doivent être récursives, et doivent utiliser la pile mise en place dans le programme pour gérer la récursivité. On sauvegardera le registre `ra` "comme avant", et on se posera la question de sauvegarder ou pas la valeur du paramètre d'appel `a0`.

On demande d'écrire le pseudo-code de ces routines, avant de les traduire en langage d'assemblage.

EXERCICE #4 ► Bonus : (pas si simple) une fonction doublement récursive

Implémenter en assembleur la fonction récursive suivante :

```

int fib_recursive(int a) {
    if (a==0) return 0;
    if (a==1) return 1;
    return fib_recursive(a-1) + fib_recursive(a-2);
}

```

TD 4

Mémoire cache

4.1 Mémoire cache

Un ordinateur dispose d'une mémoire centrale de 64 kio : les adresses sont codées sur 16 bits, et la taille de chaque case mémoire est de 1 o. Entre le processeur et la mémoire centrale est placée une petite mémoire cache directe, composée de 8 entrées pouvant chacune contenir 32 o. On décompose les adresses en trois champs :

INDICATEUR								ENTREE			OCTET				
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

On précise que : la concaténation des champs INDICATEUR et ENTREE donne en binaire l'indice d'une ligne de cache ; le champ ENTREE donne en binaire l'indice d'une entrée dans la mémoire cache ; le champ OCTET donne l'indice d'un octet dans une ligne de cache (ou un entrée dans la mémoire cache).

- 1) Pourquoi le champ OCTET comporte-t-il 5 bits, et le champ ENTREE 3 bits?
- 2) En combien de lignes de cache se décompose la mémoire centrale?
- 3) Complétez le tableau suivant ; notez les numéros de lignes en indice de la lettre ℓ , et les numéros d'entrées en indice de la lettre e .

adresse (hexa.)	adresse (binaire)															ligne (décimal)	entrée (décimal)
0BBF _H																	
0BC0 _H																	
0BC1 _H																	
05AE _H																	
05BF _H																	
05C0 _H																	

- 4) Le processeur effectue successivement des accès aux adresses 0BBF_H, 0BC0_H, 0BC1_H, 05AE_H, 05BF_H, 05C0_H (à celles-là seulement). On suppose le cache initialement vide. Indiquez quelle est la ligne contenue dans chacune des entrées du cache après chaque accès, et signalez chaque défaut de cache (par une croix dans la dernière colonne).

adresse	entrées								défaut
	e_0	e_1	e_2	e_3	e_4	e_5	e_6	e_7	
0BBF _H									
0BC0 _H									
0BC1 _H									
05AE _H									
05BF _H									
05C0 _H									

4.2 Somme de deux vecteurs

On considère le programme C suivant :

```
int main(void) {
    int A[16], B[16], C[16];
    int i;
```

```

for(i=0; i<16; i++) C[i] = A[i] + B[i];
...
}

```

On suppose que l'on compile et que l'on exécute ce programme sur un ordinateur qui ne dispose que d'un seul niveau de cache de données direct : ce cache comporte 4 entrées de 32 octets (on ne se préoccupe pas du chargement des instructions, ni de la variable *i* dans cet exercice). On note les lignes de cache $\ell_0, \ell_1, \ell_2, \dots$, et les entrées du cache e_0, e_1, e_2, e_3 . Les entiers de type `int` sont d'une taille de 32 bits. Les tableaux sont stockés de manière contiguë en mémoire, dans l'ordre de leur déclaration.

- 1) Combien d'entiers de type `int` peut contenir une ligne de cache ?
- 2) Combien de lignes de caches consécutives occupe chacun des tableaux A, B et C ?
- 3) En supposant que $A[0]$ est aligné sur le début de la ligne de cache ℓ_0 , indiquez sur un schéma à quelles lignes de cache appartiennent les éléments de A, B et C. Indiquez également sur ce même schéma dans quelles entrées du cache seront rangés ces éléments lorsqu'ils seront accédés.
- 4) Dans un tableau de la forme indiquée ci-dessous, indiquez quelle ligne de cache contient chaque entrée du cache à la fin de chaque itération de la boucle `for(i=0; i<16; i++) C[i] = A[i] + B[i];` (Vous placerez un « ? » quand on ne peut pas savoir). Indiquez également le nombre de *cache miss* qui ont eu lieu au cours de l'itération.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
e_0																
e_1																
e_2																
e_3																
cache miss																

4.3 Somme des éléments d'une matrice

On considère le programme C suivant :

```

#define n 16

int main(void) {
    int i, j;
    double A[n][n];      /* A va permettre de stocker une matrice de doubles */
    double s;

    for(i=0; i<n; i++) for(j=0; j<n; j++) A[i][j] = i+j;  /* On initialise A */
    flush_cache();      /* fonction qui a pour effet de vider le cache ! */

    /* On calcule la somme des éléments de A */
    s = 0.0;
    for(j=0; j<n; j++)
        for(i=0; i<n; i++) s += A[i][j];

    /* On affiche le résultat */
    printf("%f\n", s);

    return(0);
}

```

On suppose que l'on compile et que l'on exécute ce programme sur un ordinateur qui ne dispose que d'un seul niveau de cache de données direct : ce cache comporte 8 entrées de 64 octets. L'ordinateur dispose d'un cache d'instruction séparé : on ne se préoccupe pas du chargement des instructions dans cet exercice.

- 1) Comment est stockée la matrice A en mémoire centrale ? Représentez la situation dans le cas où $n=5$.
- 2) On suppose que $n=16$, comme indiqué dans le programme. Combien d'octets sont occupés par une ligne de la matrice A ?
- 3) En supposant que l'élément $A[0][0]$ de la matrice est stockée sur les 8 premiers octets de la ligne de cache 0, indiquez à quelle ligne appartient chacun des éléments de la matrice. Indiquez également dans quelle entrée du cache sera stocké chaque élément de la matrice.

- 4) On fixe un certain j , $0 \leq j \leq 7$, et on considère la boucle `for(i=0; i<n; i++) s += A[i][j]` : combien de *cache-misses* sont provoqués par les accès en lecture à la matrice A? Que se passe-t-il si on fixe j tel que $8 \leq j \leq 15$?
- 5) Au total, combien de *cache-misses* sont provoqués par les opérations d'accès en lecture à la matrice A dans la double boucle de sommation? En déduire le taux de *cache-miss*, c'est-à-dire le pourcentage de *cache-misses* qui se sont produits sur le nombre total d'accès à la matrice.
- 6) Comment transformer le programme de manière à ramener le nombre de *cache-misses* à 32? Quel est désormais le taux de *cache-miss*?
- 7) On compile et on exécute le programme initial sur un autre ordinateur, dont la micro-architecture comporte un cache associatif à 8 entrées de 64 octets et 2 voies. Combien de *cache-misses* provoquent les accès en lecture à la matrice A?

4.4 Mémoire cache - petits calculs

On considère, dans cet exercice¹, que les mots mémoire sont de taille un octet. Soit un cache direct de 4ko (= 2^{12} octets) dont les lignes font 128 (= 2^7) octets. Soient l'adresses hexadécimales de deux données :

— $x A23847EF$

— $x 7245E824$

- 1) Sur combien de bits les adresses sont-elles codées? Donnez le nombre de bits codant les champs INDICATEUR, ENTREE, et OFFSET (selon la terminologie du cours sur les caches directs).
- 2) Pour chacune de ces adresses, donnez (en hexadécimal) le numéro de la ligne du cache où on peut la trouver ainsi que l'étiquette (INDICATEUR) de cette ligne et l'offset (OCTET) de la donnée correspondante dans le cache. *Certains calculs fastidieux peuvent être évités...*
- 3) Si la donnée correspondant à l'adresse $(A23847EF)_{16}$ considérée est effectivement dans le cache, donnez (en hexadécimal) les adresses qui seront stockées dans la même ligne du cache que celle-ci. *On pourra se contenter d'une réponse où les calculs ne sont pas effectués jusqu'au bout.*

1. Source : <http://www.liafa.jussieu.fr/~amicheli/Ens/Archi/td11.pdf>

TD 5

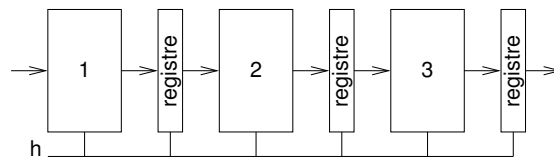
Pipeline

EXERCICE #1 ► Calculs autour de l'exécution pipelinée

On suppose que le chemin de données d'un processeur peut être divisé pour former un pipeline à k étages : chaque étage est un circuit logique, qui consomme les résultats de l'étage précédent, et produit des résultats à destination de l'étage suivant. Après chaque étage i est placé un registre pour stocker les résultats intermédiaires produits à l'étage i à un certain cycle, et pour qu'ils puissent être utilisés par l'étage $i + 1$ au cycle suivant de l'horloge h .

5.0.1 Calcul sur des pipelines

Le temps de stabilisation d'un registre est de 20 ps. Dans le cas de 3 étages, la situation peut être représentée comme suit (h est le signal d'horloge).



Dans les trois questions suivantes, le temps de traitement total d'une instruction, hors passage par les registres, reste de 300 ps.

- 1) On suppose que le pipeline comporte 3 étages et qu'ils ont tous la même durée de traitement de 100 ps. Quelle est la durée minimale du cycle d'horloge? Quel est le débit maximal de ce pipeline, exprimé en Gop/s (Giga opérations par seconde)? Quelle est dans ce cas la latence, c'est-à-dire la durée d'exécution d'une instruction?
- 2) On suppose toujours que le pipeline comporte 3 étages, mais maintenant la durée de traitement est variable : 50 ps pour l'étage 1, 150 ps pour l'étage 2, 100 ps pour l'étage 3. Répondez aux mêmes questions que précédemment.
- 3) On suppose maintenant que l'on double le nombre d'étages et qu'ils ont tous la même durée de traitement. Que deviennent la latence et le débit du pipeline?
- 4) Quel débit ne dépassera-t-on jamais, même en continuant d'augmenter la profondeur du pipeline?

5.0.2 Calibrage d'un pipeline

On considère un chemin de données non-pipeliné qui possède une latence de 10 ns. On suppose que l'on est capable de diviser ce chemin de données en k étage, en répartissant équitablement la latence des circuits logiques entre chaque étage. Le temps de stabilisation d'un registre est de 500 ps.

- 1) Quel est le temps de cycle d'une version pipelinée du processeur avec un pipeline à k étage? Faites le calcul pour $k = 2, 4, 8$ et 16.
- 2) Combien d'étages de pipeline sont requis pour atteindre un temps de cycle de 2 ns? Combien en faut-il pour atteindre la fréquence de 1 GHz?

5.1 Pipeline à 4 étages

On considère un pipeline à 4 étages, comme celui du cours. Les étages du pipeline sont FE (*fetch*, chargement), DE (*decode*, décodage), EX (*execute*, exécution) et SR (*store register*, stockage du résultat en registre). On rappelle que la phase EX d'une instruction ne peut être effectuée avant que ses opérandes n'aient été effectivement calculées et stockées (phase SR terminée) ; si ce n'est pas le cas on parlera ici de *défaut de pipeline*.

- 1) En l'absence de défaut, si une instruction entre dans le pipeline au cycle i , à quel cycle se finit son exécution ?
- 2) En ignorant les défauts de pipeline, représentez le traitement des instructions dans le pipeline en remplissant avec FE, DE, EX ou SR dans les cases du tableau suivant. Vous ajouterez entre parenthèses les registres lus dans la phase EX et ceux écrits dans la phase SR :

Instruction	Cycle 1	2	3	4	5	6	7	8	9	10	11
ADD R4, R1, R2											
NOT R4, R4											
ADD R3, R3, R4											
ADD R4, R5, R1											
ADD R5, R5, R2											

- 3) Entourez dans le tableau précédent les défauts de pipeline et expliquez-les rapidement.
- 4) Résoudre les défauts de cache en rajoutant des attentes de pipeline (un *stall*, que vous noterez S). On considère qu'un stall bloque une instruction à un étage précis du pipeline, ainsi que toutes les instructions qui la suivent dans le pipeline.
- 5) Proposez (en expliquant) un ré-ordonnancement des instructions minimisant le temps total d'exécution. Combien de cycle(s) avez-vous gagné ?

EXERCICE #2 ► Boucles et exécution pipelinée

On se place sur un processeur dont le chemin de données est implanté à l'aide d'un pipeline à 3 étages :

- chargement de l'instruction pointée par le compteur de programme et décodage ;
- exécution de l'instruction ;
- mise à jour des registres ou de la mémoire d'après le résultat de l'exécution.

On suppose que la phase d'exécution d'une instruction ne peut pas commencer avant que la phase de mise à jour d'une instruction dont elle dépend soit terminée. Le chemin de données ne gère pas seul les problèmes de dépendances dans les programmes : on peut néanmoins les éviter en utilisant judicieusement l'instruction NOP, qui ne modifie pas l'état du processeur. On considère la portion de programme ci-dessous, donnée dans le langage d'assemblage du LC3 :

```

        AND R0, R0, 0
        ADD R1, R0, 2
loop:   ADD R0, R0, R1
        ADD R1, R1, -1
        BRp loop
        ADD R0, R0, 2

```

On souhaite l'adapter, pour permettre son exécution sur le processeur pipeliné décrit.

- 1) Mettre en évidence les dépendances qui existent dans ce programme. Quels conflits vont se produire si l'on tente d'exécuter en l'état le programme ?
- 2) Donnez un ordonnancement de l'exécution du code dans le pipeline en insérant au fur et à mesure les instructions NOP nécessaires. En déduire une version modifiée du programme qui pourra être exécutée sans problème.
- 3) En oubliant la seconde instruction du programme, donnez la latence de la boucle en fonction de R1.

TD 6

Mémoires

6.1 Étude d'une hiérarchie mémoire à trois niveaux

Cet exercice traite de l'étude d'une hiérarchie mémoire incluant plusieurs niveaux de cache. Les temps d'accès des mémoires sont exprimés en nombre de cycles d'horloge processeur. Le temps d'accès en cas d'échec correspond au temps qu'il faut pour rechercher la donnée dans le niveau de cache. Le temps d'accès en cas de succès est le temps total pour accéder à la donnée en cas de succès (temps de recherche de la donnée plus temps de lecture/écriture de la donnée).

no	Niveau	temps d'accès - succès	temps d'accès - échec	taux de succès	taille
1	cache L1	3	1	80%	128 Kio
2	cache L2	5	2	90%	512 Kio
3	cache L3	12	4	95%	2 Mio
4	Mémoire	40			1 Gio

- 1) Exprimez le temps d'accès moyen $\overline{t_i}$ au niveau i , en fonction :
 - du temps d'accès au niveau i en cas de succès $t_{s,i}$, et en cas d'échec $t_{e,i}$,
 - du temps d'accès moyen $\overline{t_{i+1}}$ au niveau $i + 1$,
 - du taux de succès T_i au niveau i .
- 2) À partir des performances du tableau ci-dessus, calculer le temps d'accès moyen global pour les 4 hiérarchies mémoires suivantes :
 - mémoire centrale seule;
 - mémoire centrale avec cache L1;
 - mémoire centrale avec cache L1 et L2;
 - mémoire centrale avec cache L1, L2 et L3.

Conclure sur l'apport de chaque niveau de cache dans cette hiérarchie.

6.2 Pagination

De façon similaire à la mémoire des processeurs INTEL PENTIUM, la table 6.1 représente une mémoire paginée :

- Chaque adresse est codée sur 9 bits.
- Une adresse de 9 bits correspond à 1 mot mémoire de 32 bits
- Chaque page contient 8 mots mémoires (1 ligne du tableau 6.1).
- Il y a 2 niveaux d'indirection (table de répertoires de page et table de pages).
- Une adresse logique est composée de 9 bits, de gauche à droite : 3 pour le répertoire de pages, 3 pour la page et 3 pour le décalage dans la page.
- Dans les tables de pages, pour chaque page, 7 bits sont utilisés pour détailler les propriétés de la page :
 1. pour signaler l'existence de la page
 2. pour signaler la présence de la page en mémoire
 3. pour signaler le droit d'écriture
 4. pour signaler le droit d'exécution
 5. pour signaler le « copy-on-write »
 6. pour le bit d'accès
 7. pour le dirty bit

Décalage dans la page								
mem	0	1	2	3	4	5	6	7
00	70	48	22	142	32	126	215	128
01	1100000.11	0000000.13	0000000.07	0000000.02	0000000.21	1100000.33	0000000.23	0000000.11
02	107	63	71	241	171	91	93	223
03	29	167	27	139	154	87	59	79
04	33	54	197	180	13	112	71	168
05	130	15	233	70	8	206	76	139
06	178	227	4	159	116	23	58	200
07	204	28	123	120	61	173	51	11
10	185	63	13	130	24	18	117	87
11	1101010.05	1101000.06	1100110.25	1110110.37	0010100.57	0010010.77	0001100.66	0010000.74
12	157	115	237	19	159	219	30	187
13	0000000.16	0010100.37	0001111.67	0000000.37	0000000.03	0000000.34	0000000.34	1110000.20
14	197	248	206	145	47	28	197	249
15	129	126	122	137	205	70	220	127
16	1101010.05	1101000.06	1100110.25	1110110.37	0011000.46	1110000.14	0001010.06	1110000.27
17	1100000.16	0000000.07	0000000.25	0000000.20	0000000.25	1100000.13	0000000.07	0000000.06
20	122	17	153	217	242	151	150	209
21	205	201	173	237	183	208	31	78
22	147	148	236	235	243	118	195	249
23	145	42	235	41	148	181	30	83
24	193	36	2	240	167	207	147	192
25	69	69	216	223	163	41	58	141
26	41	93	176	16	99	10	178	207
27	223	135	55	26	205	211	248	24
30	193	26	8	3	42	191	181	232
31	165	162	122	104	67	191	78	123
32	219	55	208	116	148	253	55	143
33	0001010.57	0010100.32	1110010.14	0011000.31	0000000.63	0001111.31	0001010.06	1110011.23
34	191	142	215	171	241	121	112	180
35	1	94	177	206	225	148	86	2
36	111	5	242	120	32	44	143	201
37	0	0	0	0	0	0	0	0

FIGURE 6.1 – Memoire paginée, 1

— Dans les tables de répertoire de pages, seul le premier bit d'information est utilisé (celui de l'existence du répertoire correspondant).

On considère 2 processus, la table des répertoires de pages du processus 1 est à l'adresse 0001 et celle du processus 2 à l'adresse 0017.

- 1) Quel est la capacité d'adressage?
- 2) Pour chacun des deux processus, que contient la case d'adresse 0b10111001.
- 3) Que se passe-t-il si le processus 1 essaye de lire la valeur de la case 0b101101101

- 4) Même question si le processus 2 tente d'exécuter le code de l'adresse 0b000001011
- 5) Que remarque-t-on pour l'adresse 0b101010000 du processus 1 et l'adresse 0b000101000 du processus 2? Dans quelle cas cela peut-il arriver?
- 6) Mêmes questions pour l'adresse 0b000011111 des 2 processus. Que se passe-t'il si l'un des processus écrit à cette adresse.

6.3 Pagination exo 2

De façon similaire à la mémoire des processeurs INTEL PENTIUM, la table 6.1 représente une mémoire paginée :

- Une adresse correspond à 1 mot mémoire de 32 bits.
- Chaque page contient 8 mots mémoires (1 ligne).
- Il y a 2 niveaux d'indirection (table de répertoires de page et table de pages).
- Une adresse logique est composée de 9 bits, de gauche à droite, 3 pour le répertoire de pages, 3 pour la page et 3 pour le décalage dans la page.
- Dans les tables de pages, pour chaque page, 7 bits sont utilisés pour détailler les propriétés de la page :
 1. pour signaler l'existence de la page
 2. pour signaler la présence de la page en mémoire
 3. pour signaler le droit d'écriture
 4. pour signaler le droit d'exécution
 5. pour signaler le « copy-on-write »
 6. pour le bit d'accès
 7. pour le dirty bit
- Dans les tables de répertoire de page, seul le premier bit d'information est utilisé (celui de l'existence du répertoire correspondant).

On considère 2 processus, la table des répertoires de pages du processus 1 est à l'adresse (octale) 0o06 et celle du processus 2 à l'adresse 0o07.

- 1) Pour les 2 processus, que contiennent les cases d'adresse 000001010? Est-ce du code ou une donnée?
- 2) Pour le processus 1 que contient la case d'adresse binaire 010000111? Est du code ou une donnée?
- 3) Mêmes questions pour la case d'adresse binaire 000011010 et le processus 2.
- 4) Que remarquez-vous avec les questions précédentes? Comment cela peut-il arriver?
- 5) Pour les deux processus, que contiennent les cases d'adresse 101100011?
- 6) Que remarquez-vous? Comment cela peut-il arriver?
- 7) Que se passe-t-il si le processus 1 essaye de modifier la case d'adresse 101101101?

mem	0	1	2	3	4	5	6	7
00	18	16	198	129	32	169	15	193
01	1110011.11	1101000.14	0100110.10	0001000.22	0010000.56	0010010.00	0000000.33	0011111.22
02	1001000.56	0000000.23	0011111.06	1101000.27	1111010.00	0000110.34	0000000.24	0010100.27
03	0000000.33	0001000.32	1101000.27	0011010.11	1100100.32	1101100.13	0101010.66	0010011.42
04	0001000.00	1110100.12	0010000.12	1101000.14	0011000.04	0010100.47	0110100.00	0110000.07
05	0011000.00	1010010.32	0011100.55	0000000.22	1100101.32	0010010.12	0001110.29	0011111.23
06	1100000.01	0000000.00	1100000.02	0000000.00	0000000.00	1100000.03	0000000.00	0000000.00
07	1100000.04	0000000.00	0000000.00	0000000.00	0000000.03	1100000.05	0000000.00	0000000.00
10	71	191	237	217	54	177	179	190
11	91	50	143	169	217	181	237	234
12	60	186	45	98	110	108	81	28
13	14	55	249	80	75	122	107	225
14	128	171	165	3	220	157	53	169
15	230	187	51	244	186	152	115	237
16	158	160	89	97	27	14	217	0
17	27	200	233	213	239	217	195	10
20	198	244	77	179	135	241	143	45
21	10	168	52	44	6	36	21	57
22	236	168	4	203	151	9	223	146
23	196	254	193	140	136	153	95	13
24	118	158	169	227	111	197	24	86
25	217	10	226	121	84	180	2	223
26	64	255	138	80	18	231	16	90
27	121	22	181	58	188	243	171	228
30	220	34	198	197	152	83	178	188
31	86	104	21	130	23	223	224	82
32	201	72	27	246	6	155	168	68
33	162	252	2	178	37	223	247	4
34	221	16	73	168	217	129	40	154
35	81	185	183	240	225	209	154	227
36	228	41	195	153	212	85	22	226
37	117	113	59	76	147	73	74	129

TABLE 6.1 – Contenu mémoire 2