HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

Reconfigurable & embedded
Digital Systems

Processeur Pipeliné

Départements : TIC

Unité d'enseignement ARO

Auteurs : **Bastian Chollet**
Kevin Ferati

Professeur : **Marina Zapater**
Assistant : **Hänggi Gregory**

Classe : **D**
Salle de labo : **A09**

Date : **05.06.2022**

Table des matières

1 Introduction

Dans ce laboratoire, nous poursuivons l'implémentation de notre processeur MIPS en lui introduisant la notion de parallélisme. Ce laboratoire a en grande partie déjà été réalisé en amont par les enseignants et assistants. Ce rapport se focalisera donc essentiellement sur la réponse aux questions posées ainsi qu'à l'explication des différentes parties réalisées.

Ce rapport sera divisé en deux parties. Une première concernant la mise en place du pipeline ainsi que la détection des aléas de contrôle. La deuxième partie quant à elle se focalisera sur la gestion des aléas de données.

Le forwarding ne sera pas abordé dans ce rapport étant donné des contraintes de temps et un circuit initialement fourni non fonctionnel.

2 Partie 1 – Pipeline & Aléas de contrôle

1.1 Introduction

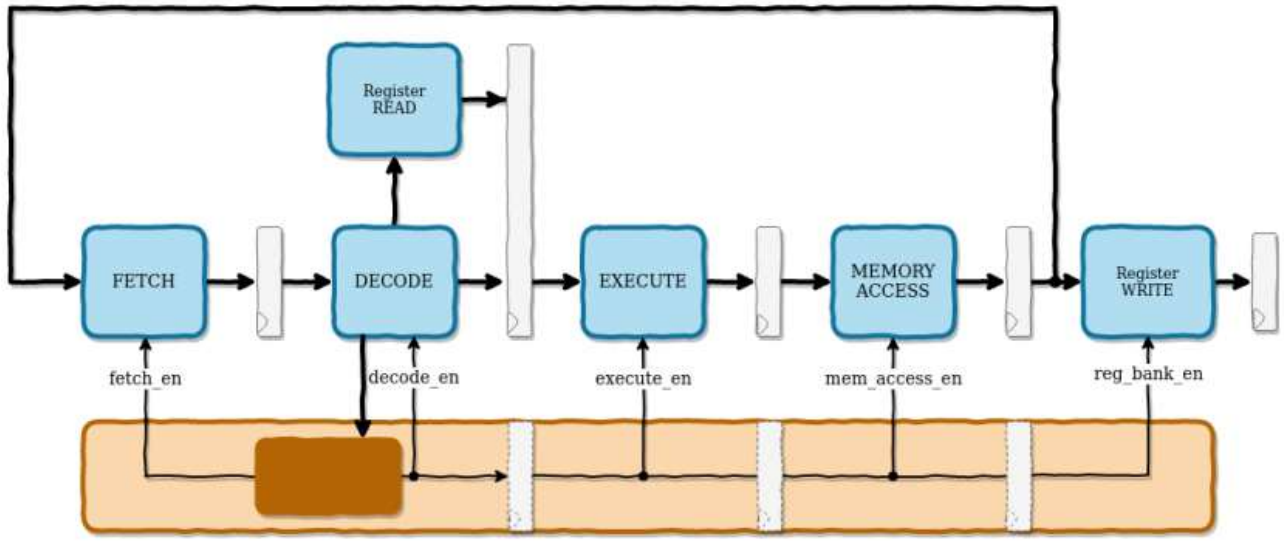
Dans cette partie, nous étudierons et commenterons la mise en place d'un processeur pipeliné capable d'exécuter plusieurs instructions en parallèle. Le processeur fourni est donc capable d'exécuter ces instructions. Cependant, il n'est pas encore doté de la capacité de détection des aléas. Ici, nous tenterons d'implémenter la détection des aléas de contrôle, c'est-à-dire de la partie gérant les détections de branchement.

1.2 Analyse du processeur

1.2.1 Réalisation

Pour des raisons pédagogiques et des contraintes de temps, le processeur fournit pour ce laboratoire a déjà été pipeliné. En effet, il ne suffit pas d'ajouter des registres à chaque bloc du processeur. Il convient également de s'assurer que l'ensemble des signaux de contrôle arrivent au bon moment. Or dans notre processeur non pipeliné, certains signaux étaient directement transmis aux blocs suivants. C'est le cas par exemple de notre execute control bus qui est généré au moment du DECODE de l'instruction, ici, il ne faudra pas directement le transmettre sous peine de corrompre les instructions précédentes.

Le schéma suivant nous a été fourni pour illustrer la mise en place du pipeline :



Où chaque bloc gris représente un registre. Dans les faits, il faut s'imaginer qu'on ne dispose pas d'un registre par bloc, mais d'un registre **par signal** à la fin de chaque bloc. Les informations suivantes nous ont également été fournie sur l'implémentation de notre pipeline :

- Dans le circuit mult_2, les offsets sont incrémentés de 1 au lieu d'être incrémenté de 2.

- Dans le circuit LR_manager, le signal link_en_i passe dans 3 registres au lieu de 1, pour que le signal link_en_d1_s soit généré au bon timing.
- Le signal branch_i est calculé dans memory_access au lieu de fetch car c'est dans ce bloc que les informations sont disponibles pour le calcul.
- Les signaux passent par tous les blocs même s'ils ne sont pas utilisés dans un bloc. Ceci pour assurer que les informations de contrôle arrive en même temps que les données dans le bloc qui les utilisent.

1.2.2 Questions

1. *Dans le circuit mult_2, les offsets sont incrémentés de 1 au lieu d'être incrémenté de 2 dans le circuit non-pipeliné, pourquoi ?*

   Pour calculer une adresse de saut, nous ajouton +4 à la fin du calcul. Avant, ce calcul s'exécutait en un coup d'horloge quoiqu'il arrive.
   Avec l'ajout du pipeline, nous avons ajouté un registre à la fin du bloc fetch, nous faisant perdre un coup d'horloge. C'est dans le but de compenser cette perte que nous passons l'incrément de l'offset de 2 à 1

2. *Dans le circuit fetch, le signal LR_adr_o vient d'un registre et est connecté au bloc decode au lieu du bloc bank_register, pourquoi ?*

   En procédant ainsi, on retarder l'écriture de l'adresse de LR dans la banque de registre. Ici, l'écriture de LR ne s'effectue qu'après la fin du Write Back. Cette façon de procéder permet d'éviter l'écrasement de la valeur de LR par l'instruction suivante du pipeline.

3. *Dans le circuit decode, le signal adr_reg_d_s est mis dans un registre alors que les signaux adr_reg_n_s, adr_reg_m_s et adr_reg_mem_s sont directement connecté à la sortie, pourquoi ?*

   Les registres opérandes et mémoire sont directement envoyé à l'EXECUTE et sont enregistrés dans ce bloc. Afin que le résultat de l'EXECUTE se retrouve dans le registre de destination souhaité, on retarde son envoi au bloc suivant. Ainsi, le prochain coup d'horloge va exécuter l'opération et on stockera le résulta dans Rd venant d'arriver dans EXECUTE. Si Rd n'est pas retardé, le résultat de l'EXECUTE serait envoyé dans le Rd de l'instruction suivante ce qui n'est pas souhaitable.

4. *Dans le circuit decode, les signaux des bus de contrôle sont connectés aux registres avec une porte MUX contrairement aux autres signaux, pourquoi ?*

   Cela nous permet de bloquer ces signaux de contrôle (lors d'un aléa par exemple) afin d'éviter de propager de mauvaises informations.

5. *Si on voulait ajouter le multiplieur 5x3 pipeliné du laboratoire préparatoire, quelles seraient les conséquences sur le pipeline du processeur ? Comment ça pourrait être fait ?*

   Ce circuit de multiplication étant lui aussi pipeliné sur 3 étages, cela impacterait les performances globales de notre processeur. En effet, plutôt que d'effectuer une multiplication sur un coup d'horloge, ici, il en faudra 3 en plus des autres tick nécessaire à l'exécution complète d'une instruction dans notre processeur. Si nous désirions tout de même utiliser ce

circuit, il faudrait bloquer le pipeline pendant 3 ticks d'horloge supplémentaires le temps que la multiplication s'effectue.

1.3 Test du processeur

Afin de tester le bon fonctionnement de notre processeur pipeliné, nous allons procéder à l'exécution de différents programmes dont nous allons analyser les chronogrammes. Nous répondrons aux questions posées puis nous proposerons des améliorations desdits programmes lorsque ceux-ci nous sont demandé

1.3.1 Premier programme

Soit le programme suivant :

```
@ programme 1
mov r0,#0x3E
mov r1,#3
mov r2,#0xCB
mov r3,#6
nop
@ Partie à analyser
add r4,r0,#2
strh r2,[r0,#4*2]
ldrh r1,[r0,#4*2]
b fin
nop
nop
nop
nop
nop
.org 0x40
fin:
and r1,r3
nop
nop
nop
nop
nop
@ fin de l'analyse
```

Sur la page suivantes, le chronogramme du programme ci-dessus :


Signal Name
Signal Val
PC
14
branch_o
0
decode/imm3_o
0
decode/pipe_imm5_o
0
decode/pipe_offset8_o
16
decode/pipe_offset11_o
16
execute/pipe_adr_reg_d_o
1
execute/pipe_data_out_o
46
memory_access/pipe_adr_reg_d_o
2
memory_access/pipe_mem_out_o
0
bank_registers/R0
3E
bank_registers/R1
3
bank_registers/R2
CB
bank_registers/R3
6
bank_registers/R4
40

1.3.2 Questions

1. *Est-ce que le programme s'exécute correctement ? Est-ce que les registres prennent les bonnes valeurs ?*

   Nous observons effectivement un bon fonctionnement du programme. Nous observons bien le fait que les données passent par chaque étage du pipeline et que les registres se mettent à jours avec les bonnes valeurs.

2. *Combien de cycles sont nécessaires pour exécuter ce programme ?*

   On constate que du début jusqu'à la fin du programme (jusqu'à la fin du dernier write bacl) **18 cycles** sont nécessaires.

1.3.3 Deuxième programme

Soit le programme suivant :

```
MAIN_START:
MOV r0, #1
MOV r1, #2
MOV r2, #6
STRH r0, [r1, #4]
ADD r4, r2, #1
ADD r3, r2, #4
SUB r4, r1, r0
ADD r0, r0, #5
LSL r2, r2, #1
LSL r2, r2, #1
B PART_2

.org 0x40
PART_2:
MOV r0, #3
MOV r1, #4
MOV r2, #8
b SAUTZERO
ADD r0, r1, r2

.org 0x60
SAUTZERO:
MOV r0, #255
BNE SAUTC
MOV r1, #5

.org 0x80
SAUTC:
MOV r0, #0
BNE NOT_TAKEN
MOV r1, #0
BEQ MAIN_START

.org 0xA0
NOT_TAKEN:
B MAIN_START
MOV r4, #6
```

Focalisons-nous maintenant sur la première partie de ce code (depuis *MAIN_START* à *B PART_ 2*) pour tenter d'analyser les différentes dépendances de données pour chaque instruction. La syntaxe de lecture est la suivante : **N° de ligne responsable de la dépendance (Registre de dépendance).**

	Instructions	Read After Write	Write After Read	Write After Write
L1	MOV r0, #1	-	-	-
L2	MOV r1, #2	-	-	-
L3	MOV r2, #6	-	-	-
L4	STRH r0, [r1, #4]	L2 (R1)	-	L1 (R0)
L5	ADD r4, r2, #1	L3 (R2)	-	-
L6	ADD r3, r2, #4	L3 (R2)	-	-
L7	SUB r4, r1, r0	L2 (R1)	-	L5 (R4)
L8	ADD r0, r0, #5	L1, L4 (R0)	L7 (R0)	L1, L4 (R0)
L9	LSL r2, r2, #1	L3 (R2)	L5, L6 (R2)	L3 (R2)
L10	LSL r2, r2, #1	L3, L9 (R2)	L5, L6, L9 (R2)	L3, L9 (R2)

Il n'est pas nécessaire d'analyser le reste du code pour ce qui concerne les dépendances de données. En effet, un rapide coup d'œil permet de se rendre compte qu'aucun aléa de donnée ne risque d'être déclenché. Nous reviendrons en revanche sur cette partie pour analyser les aléas de contrôles.

Ci-après, le chronogramme de cette première partie de programme :

Signal Name
processeur_ARO2/fetch/PC
processeur_ARO2/decode/imm3
processeur_ARO2/decode/imm5
processeur_ARO2/bank_registers/register_m_data
processeur_ARO2/bank_registers/register_n_data
processeur_ARO2/decode/offset8
processeur_ARO2/decode/offset11
processeur_ARO2/decode/adr_reg_d
processeur_ARO2/execute/adr_reg_d
processeur_ARO2/execute/data_out
processeur_ARO2/memory_access/adr_reg_d
processeur_ARO2/memory_access/mem_out
processeur_ARO2/bank_registers/R0
processeur_ARO2/bank_registers/R1
processeur_ARO2/bank_registers/R2
processeur_ARO2/bank_registers/R3
processeur_ARO2/bank_registers/R4

Nous constatons plusieurs problèmes avec ce chronogramme. Le plus flagrant étant le résultat des opérations arithmétiques qui sont erronés. Par exemple, le premier ADD s'exécute alors que la valeur de R2 n'est pas encore enregistrée. Ainsi, au lieu d'enregistré la valeur 3, R4 enregistre la valeur 1 car l'opération exécuté à ce moment-là est : 0 + 1.
Pour remédier à ces problèmes, on peut manuellement indiquer des instructions de pause (NOP) le temps que les aléas de données se résolvent. Voici le code modifié pour que son exécution produise le résultat attendu :

```
MAIN_START
MOV r0, #1
MOV r1, #2
MOV r2, #6
NOP
NOP
STRH r0, [r1, #4]
ADD r4, r2, #1
ADD r3, r2, #4
SUB r4, r1, r0
ADD r0, r0, #5
LSL r2, r2, #1
NOP
NOP
NOP
LSL r2, r2, #1
```

Et ci-après le chronogramme prouvant que ce programme est correct :

Signal Name
Si...
processeur_ARO2/fetch/PC_pres_i
40
processeur_ARO2/decode/pipe_imm3_o
0
processeur_ARO2/decode/pipe_imm5_o
0
processeur_ARO2/bank_registers/pipe_register_m_read_data_o
6
processeur_ARO2/bank_registers/pipe_register_n_read_data_o
6
processeur_ARO2/decode/pipe_offset8_o
0
processeur_ARO2/decode/pipe_offset11_o
0
processeur_ARO2/decode/pipe_adr_reg_d_o
0
processeur_ARO2/execute/pipe_adr_reg_d_o
0
processeur_ARO2/execute/pipe_data_out_o
6
processeur_ARO2/memory_access/pipe_adr_reg_d_o
0
processeur_ARO2/memory_access/pipe_mem_out_o
6
processeur_ARO2/bank_registers/R0
6
processeur_ARO2/bank_registers/R1
2
processeur_ARO2/bank_registers/R2
18
processeur_ARO2/bank_registers/R3
A
processeur_ARO2/bank_registers/R4
1

1.3.4 Questions – Dépendance des données

1. *Quelles dépendances posent des problèmes d'aléas ?*

   On distingue 3 types de dépendances pouvant causer des aléas :
   1) **Les dépendances structurelles**, lorsque deux blocs accèdent aux mêmes ressources (par exemple lorsque le FETCH d'une instruction accède à la mémoire en même temps qu'une instruction MEMORY ACCESS d'une autre)
   2) **Les dépendances de données,** lorsque des instructions tentent d'accéder aux mêmes registres. C'est typiquement le cas dans cette moitié de programme avec les dépendances de type Read After Write.
   3) **Les dépendances de contrôles**, lorsqu'une instruction peut ou non s'exécuter / s'interrompre car un saut dans le programme a été effectué

2. *Combien de cycles sont nécessaires pour résoudre un aléa de donnée ?*

   Seules les dépendances de type Read After Write vont provoquer un retard d'exécution sur le pipeline. Ainsi, le nombre de cycle va dépendre d'où se situe la dépendance. Par exemple, si l'instruction 10 et dépendante de l'instruction 1, aucun cycle supplémentaire de retard ne sera créé. En revanche, dans le pire des cas, l'aléa se produira comme ceci :

L1	F	D	E	M	W				
L2		F	D	D	D	D	E	M	W

   Rajoutant un total de **3 cycle supplémentaire dans le pire des cas** par rapport à un exécution sans aléa.

3. *Quelle est l'IPC ? Le throughput si la clock vaut 4KHz ? La latence ?*

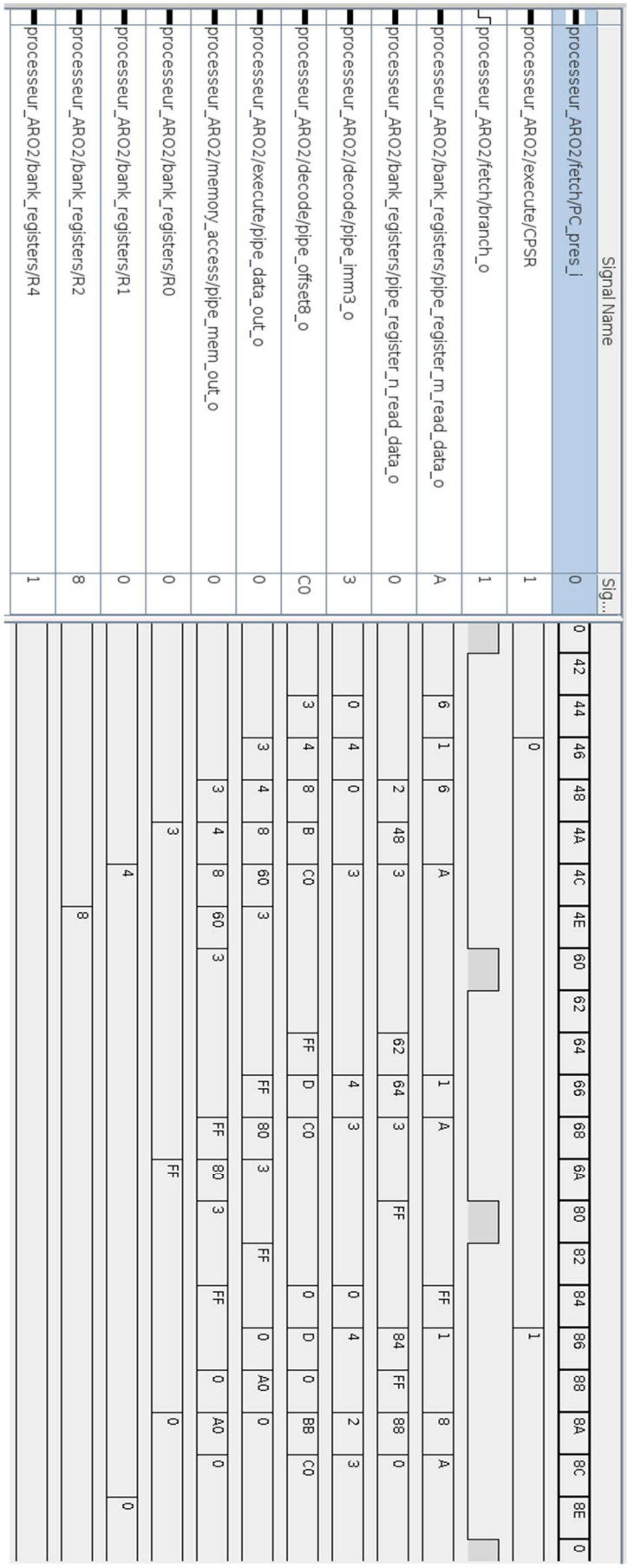
   (Pour les calculs, nous considérons uniquement cette première moitié de programme corrigé testée).

   IPC = 15 / 19 = 0.789
   Latence = 4.75 ms
   Débit = 3'157.89 instructions / s

Tentons maintenant la deuxième partie du programme. Il va s'agir ici de gérer les aléas de contrôle également via l'ajout d'instruction NOP. La particularité ici et qu'il faut s'assurer que lorsque la décision d'un saut est prise, il faut impérativement que les instructions suivant le saut soient « annulées » ou du moins, qu'elles ne soient pas traitées tout en s'assurant que les instructions précédents le saut soient bien arrivée aux termes de leur Write Back. Nous devons également prêter attention à ce que la valeur du CPSR soit correct au moment de la décision du saut. Ainsi, la deuxième partie du programme nécessite l'ajout d'instructions NOP.

```
@Première partie pas répétée ici…
B PART2
NOP
NOP
NOP
NOP

.org 0x40
PART_2:
MOV r0, #3
MOV r1, #4
MOV r2, #8
b SAUTZERO
NOP
NOP
NOP
NOP
ADD r0, r1, r2

.org 0x60
SAUTZERO:
MOV r0, #255
BNE SAUTC
NOP
NOP
NOP
NOP
MOV r1, #5

.org 0x80
SAUTC:
MOV r0, #0
BNE NOT_TAKEN
MOV r1, #0
BEQ MAIN_START
NOP
NOP
NOP
NOP

.org 0xA0
NOT_TAKEN:
B MAIN_START
@Pas besoin d'ajouter de NOP, on atterri jamais ici
MOV r4, #6
```

Et voici le chronogramme du programme fonctionnel (seul la 2e moitié du programme est représentée):


Signal Name
Sig...
processeur_ARO2/fetch/PC_pres_i
0
processeur_ARO2/execute/CPSR
1
processeur_ARO2/fetch/branch_o
1
processeur_ARO2/bank_registers/pipe_register_m_read_data_o
A
processeur_ARO2/bank_registers/pipe_register_n_read_data_o
0
processeur_ARO2/decode/pipe_imm3_o
3
processeur_ARO2/decode/pipe_offset8_o
C0
processeur_ARO2/execute/pipe_data_out_o
0
processeur_ARO2/memory_access/pipe_mem_out_o
0
processeur_ARO2/bank_registers/R0
0
processeur_ARO2/bank_registers/R1
0
processeur_ARO2/bank_registers/R2
8
processeur_ARO2/bank_registers/R4
1

1.3.5 Questions – Aléas de contrôle

1. *Combien de cycles sont nécessaires pour résoudre un aléa de contrôle*

   3 cycles supplémentaire sont nécessaires pour résoudre un aléa de contrôle.

2. *Quelle est l'IPC ? Le throughput si la clock vaut 4KHZ ? La latence ?*

   (Pour les calculs, nous considérons uniquement cette deuxième moitié de programme corrigé testée).

   IPC = 19 / 23 = 0.826
   Latence = 5.75 ms
   Débit = 3'304.348 instructions / s

1.4 Aléas de contrôle

Nous avons vu dans la section précédente comment procéder manuellement à l'arrêt du pipeline lorsque nous rencontrons un aléa de contrôle. Toutefois, dans un programme plus complexe, il est indispensable de pouvoir automatiser la gestion de ces aléas. En effet, lors d'une instruction de saut conditionnel, l'aléas peut être provoqué ou non, et nous ne pourrons pas toujours garantir le cas d'exécution, rendant impossible l'écriture d'instructions NOP en dur.

1.4.1 Questions – Circuit control_hazard

1. *Combien de cycles le pipeline doit être bloqué dans le cas d'un aléa de contrôle ?*

   Comme nous l'avons vu précédemment, 4 cycles suffisent pour éviter que les instructions suivants le saut soient fetchées

2. *Pourquoi faut-il bloquer le pipeline lorsqu'il y a un aléa de contrôle ?*

   Fetcher les instructions suivant un saut peuvent modifier les valeurs des registres de la banque de mémoire, modifier une valeur stockée en mémoire. Si de telles données sont modifiés, cela corrompt le programme en écrasant des données qui auraient dû être conservées.

3. *Quels sont les conditions pour qu'un aléa de contrôle ait lieu ?*

   Il y a aléa de contrôle à chaque fois qu'un saut, conditionnel ou non, sera exécuté par le programme. Les sauts conditionnels ne respectant pas la condition pour être exécuté ne génère pas d'aléa de contrôle.

4. *Que se passe-t-il si une instruction génère un aléa de contrôle et un aléa de donnée ?*

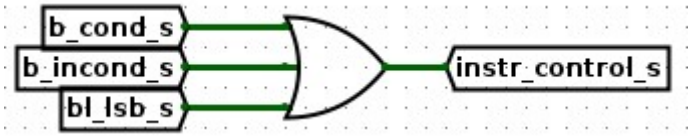
   Dans le cas des aléas de contrôle, il faut être vigilent à la situation ou l'instruction générant un aléa de contrôle a une dépendance de type Read After Write. En effet, lors d'un saut conditionnel, la valeur du CPSR est lue, et dans le cas d'une dépendance RAW, cette valeur

peut ne pas encore avoir été enregistré avec la bonne valeur ce qui peut fausser la décision de saut.

1.4.2 Questions - hazard_detection

Dans cette partie de ce laboratoire, il nous a été demandé de fournir la logique venant alimenter l'entrée **instr_control_i.** Ce circuit va notamment être responsable de détecter les différents aléas de contrôle mais également les aléas de donnée. Les aléas de données feront partie de la deuxième partie de ce rapport.

L'implémentation de cette partie consistait simplement en une porte OR à laquelle était reliée les différentes instructions responsables d'un aléa de contrôle. Voici l'implémentation réalisée dans le circuit main_control_unit (là où se situe le circuit hazard_detection) :



1. *Quelles instructions génèrent un aléa de contrôle ?*

   Les instructions **de sauts inconditionnels, de saut conditionnel, et de saut long avec lien.**

2. *Comment les aléas de contrôle influencent les différents enables ?*

   La détection d'un aléa de contrôle va désactiver l'input no_ctl_hazard_s pendant un compte à rebours de 3 ticks d'horloges. Pendant ces 3 ticks, les différents blocs vont être désactivés au fur et à mesure puit réactivé également au fur et à mesure.

3. *Que se passe-t-il dans le pipeline si un saut est pris ? Quelle est la prochaine instruction exécutée ?*

   Dès la détection d'un aléa de contrôle, le FETCH sera instantanément désactivé. Puis, ce sont les blocs suivants qui chacun leur tour va être désactivé de façon que seul l'instruction du saut soit exécutée sur les 5 étage du pipeline.
   Dès que le saut a effectué sont WRITE BACK, et que l'aléa de contrôle n'est plus détecté, le FETCH est réactivé et les autres blocs au fur et à mesure. Ainsi, la première nouvelle instruction fetchée sera celle post saut.

4. *Pourquoi branch_i est dans les entrées du circuit hazard_detection ?*

   Cette entrée est nécessaire dans le cas où l'instruction de saut est un long saut avec lien.

5. *Pourquoi instr_control_i du bloc control_hazard dépend de no_data_hazard_s ?*

   Car comme nous l'avons déjà évoqué, si un saut a une dépendance de type Read After Write, il est impératif que l'on s'assure que cet aléa de donnée puisse être traité avant l'aléa de contrôle dans le cas où la dépendance RAW affecterait le CPSR et donc la décision de saut.

2 Partie 2 – Sans forwarding

2.1 Aléas de données

2.1.1 Questions sur data_hazard

1. *Comment savoir si une instruction est dépendante d'une instruction qui est pour le moment dans le stage EXECUTE ? dans le stage MEMORY_ACCESS ? Dans le stage WRITE_BACK ?*

Le comportement est dans le même principe dans les trois cas : pour chacun des registres de lecture (n, m, mem) on les compare simplement à l'état des derniers cycles pour voir si ce sont les mêmes.

2. *Est-ce que ça pose un problème si une instruction dépend du résultat d'une instruction qui est au stage WRITE_BACK ?*

Non car elle est placée dans des registres nous pouvons y accéder directement.

3. *Quelles informations doivent être mémorisées pour chaque instruction ?*

S'il y a une écriture dans la banque de registre dans les derniers coups d'horloge et les l'adresse de destination (adr_reg_d_s)

4. *Quelles informations permettent de savoir si le registre D est utilisé ?*

De decode_en_i qui vient de hazard_detection.

2.1.2 Questions sur hazard_dection

1. *Quelles informations permettent de savoir si le registre N, M ou mem sont utilisés ?*

Des entrées reg_[n, m, mem]_en_i, qui sont gérés dans le bloc main_control_unit à partir des sélecteurs d'opérandes, ceux-ci faits dans le composant execute_control_unit.

2. *Quelles informations permettent de savoir si le registre D est utilisé ?*

De l'entrée reg_bank_write_en_i, celle-ci gérée dans le main_control_unit par le composant reg_bank_control_unit qui va tout simplement activer le signal si l'instruction courant écrit dans la banque de registre.

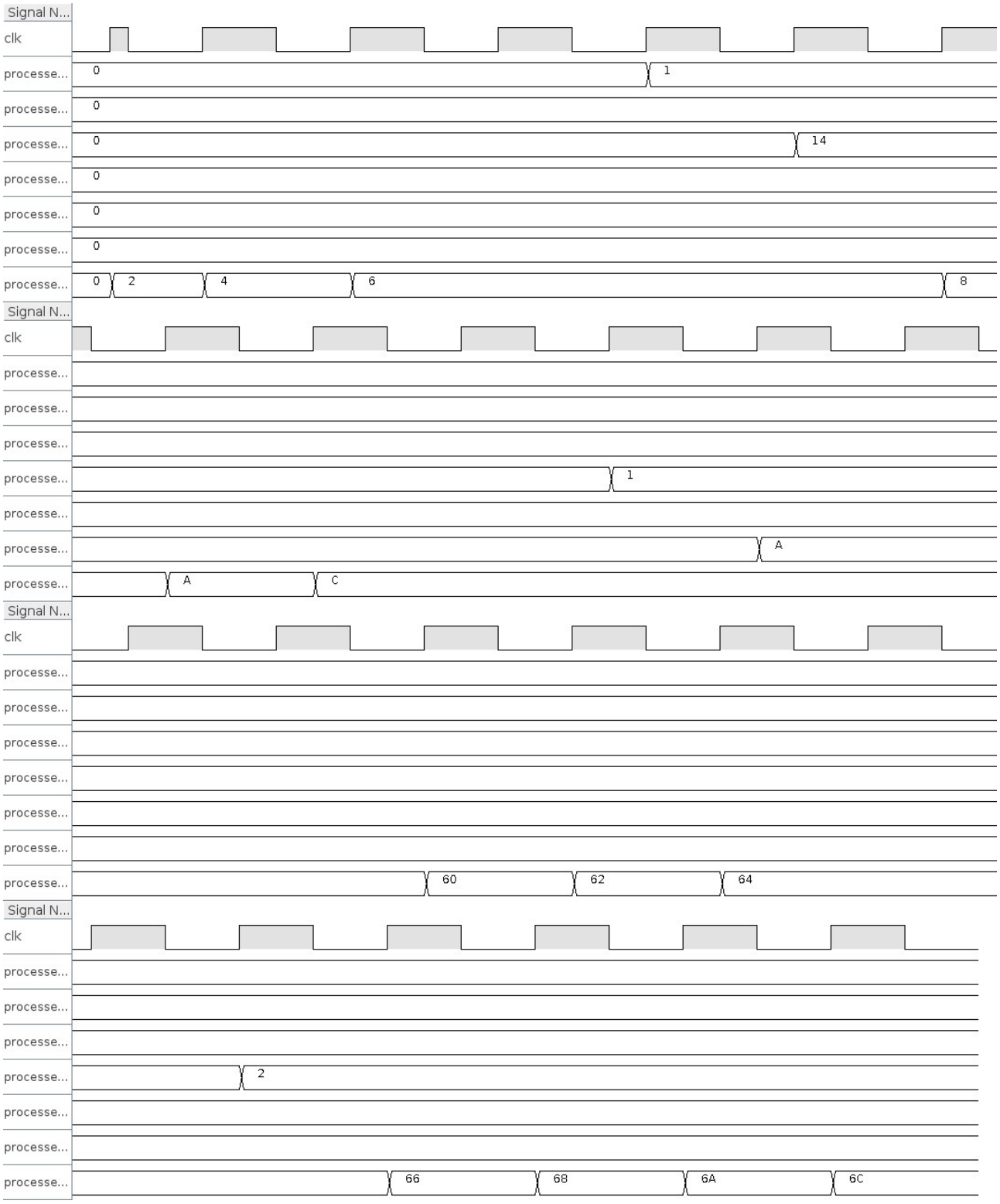
3. *Une détection d'aléa de donnée va influencer quel(s) enable(s) ? A quel moment ? Pourquoi ?*

La partie influencée sera le fetch. En effet, si un aléa est détecté, le fetch est désactivé ce qui va faire que les prochains coups d'horloge ne vont pas récupérer d'instruction et donc bloquer le CPU. Qui plus est, les composants decode et suivants vont être bloqués ou non, en fonction de l'état des aléas.

2.2 Tests

Programme exécuté :

```
START :
        MOV r0, #1
        MOV r2, #20

        STRH r0, [r2]
        LDRH r3, [r2]

        BL FONCTION
        MOV r4, #2


FONCTION:
        Mov r3, #2
        NOP
        NOP
        NOP
```



Les registres vont de R0 à R4, LR, PC.

2.2.1 Questions

1. *Est-ce que les valeurs sont mis à jours correctement dans les bons registres ?*

Oui, les registres sont correctement à jour.

2. *Pourquoi l'instruction BL génère un aléa de contrôle et un aléa de donnée?*

L'aléa de contrôle est inhérent au BL : l'adresse de saut étant calculé en execute, il faut attendre le calcul pour que cela se fasse. Mais, en attendant, les instructions en dessous de BL continuent d'être traités dans le pipe.

Quant aux données, ce calcul se fasant par rapport au PC qui est un registre, il devra y aaccéder => aléa de données.

3. *Combien de cycles sont nécessaires pour résoudre les aléas de l'instruction BL?*

Il en faut 3, comme explicité sur le chronogramme.

4. *Calcul de l'IPC*

Le programme a 10 instructions, réalisés sur 20 cycles => IPC = 10/20 = 0.5

3 Partie 2 – Avec forwarding

3.1 Circuit data_hazard

1. *A quoi sert le signal sel_mem_i ?*

A savoir si la mémoire est accédée (famille d'instruction STR / LDR).

2. *Est-il possible / utile de faire un data forwarding depuis le stage WRITE_BACK ? (l'écriture dans le registre dans la banque de registres). Comment pourrait-il être ajouté au circuit ?*

Non car on modifie justement le reg voulu à ce stade.

3. *Quelles sont les conditions pour que le forwarding puisse avoir lieu ? Quelles sont les conditions pour que le forwarding soit utile ?*

Possibilité :
- Le composant traité doit être passé, ceci il y a un cicle ;
- Un changement sur les registres ou la lecture doit être présent.

Utilité :
- Si les instructions en dessous de celle-ci réutilise le résultat du calcul de l'ALU. Dans ce cas le forwarding est utile.

4. *Quelles sont les conséquences du forwarding sur la gestion des aléas de données et de contrôle ?*

Celles-ci sont mieux gérées, grâce aux registres.

3.2 Circuit execute

3.2.1 Questions

1. *Pourquoi doit-on faire ça ?*

Afin de pouvoir correctement effectuer le forwarding et sélectionner les valeurs « forwardées » dans ce cas.

2. *Pourquoi doit-on faire ça pour le signal reg_mem_data_s ?*

C'est pour avoir directement accès à la valeur qui a été écrite en mémoire et ne pas attendre.

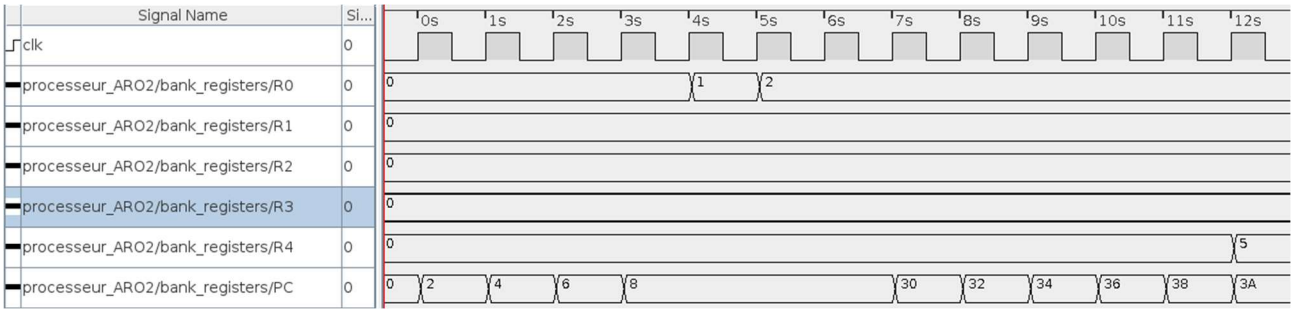
3. *Que devrait-on faire si on avait un data forwarding venant du WRITE_BACK ?*

Dans ce cas, il faudrait ajouter une entrée à execute et remplacer les valeurs 0000 du MUX de la sélection des opérandes par cette entrée là. Il faudrait aussi modifier les sel_opx_forward_i en amont pour prendre ceci en compte.

3.3 Tests du forwarding

En admettant le programme suivant :

```
START :
        MOV r0, #1
        ADD r0, r0
        BL FONCTION

.org 0x30
FONCTION:
        MOV r4, #5
```



3.3.1 Questions

1. Est-ce que votre processeur fonctionne correctement? Est-ce que les timings sont respectés ? Est-ce que les registres contiennent les bonnes valeurs si on regarde étape par étape l'exécution des instructions

Oui, le registre r0 passe de 1 à 2, et ensuite r4 prend 5, indiquant que le saut est passé.

2. Quel est l'IPC de votre programme ? et le throughput si on considère une clock à 4KHz ?

4 instructions ont été réalisées en 12 cycles : nous avons un IPC de 1/3. Le throughput est de 4'000 cycles par secondes.

3. Combien de cycles sont nécessaires pour que l'instruction BL soit complétée ?

6 cycles sont nécessaires, le fetch ayant commencé à la clock 2s.

4. Avez-vous d'autres idées d'optimisation de ce processeur ?

Nous pourrions ajouter un forwarding au memory access et augmenter le nombre de parties (étages) du processeur. Si nous augmentons le nombre d'étages, nous pourrions aussi effectuer d'autres améliorations, comme la prédiction de branchement.

4 Conclusion

Fort de ces différentes analyses et preuves, nous constatons que l'implémentation du pipeline est très complexe et peut très facilement déboucher sur plusieurs bugs. Cependant, il peut grandement aider les performances. Malgré les circuits fournis, il nous a été complexe d'analyser celui-ci et de le comprendre et tester. Cependant, nous avons beaucoup appris.s