Rapport ARO - Processeur

Départements : TIN

Unité d'enseignement ARO

Auteurs : **Timothée Van Hove**   
**Benoit Delay**

Professeur : **Romuald Mosqueron**

Assistant : **Mike Meury**

Classe : **A**

Salle de labo : **A07**

Date : **samedi, 4 juin 2022**

Table des matières

[1 Introduction 3](#_Toc105266757)

[1.1 Méthodologie 3](#_Toc105266758)

[2 Rappels fonctionnement du processeur 4](#_Toc105266759)

[2.1 Architecture de notre processeur 5](#_Toc105266760)

[3 Processeur pipeliné 6](#_Toc105266761)

[3.1 Mécanismes à ajouter 6](#_Toc105266762)

[3.2 Analyse et conception 7](#_Toc105266763)

[3.3 Travail demandé 10](#_Toc105266764)

[4 Processeur pipeliné avec forwarding 19](#_Toc105266765)

[4.1 Mécanismes à ajouter pour les aléas de données 19](#_Toc105266766)

[4.2 Analyse du circuit fourni 19](#_Toc105266767)

[4.3 Mécanismes à ajouter pour le forwarding 23](#_Toc105266768)

[5 Conclusion 29](#_Toc105266769)

[6 Bilan personnel 29](#_Toc105266770)

[7 Table des illustrations 30](#_Toc105266771)

[8 Annexes : réponse aux questions 31](#_Toc105266772)

[8.1 Analyse du processeur - Donnée 31](#_Toc105266773)

[8.2 Test du processeur – Donnée 33](#_Toc105266774)

[8.3 Assembleur : dépendances de données - Donnée 34](#_Toc105266775)

[8.4 Assembleur : aléas de contrôle - Donnée 34](#_Toc105266776)

[8.5 Circuit control\_hazard - Donnée 35](#_Toc105266777)

[8.6 Circuit hazard\_detection - Donnée 37](#_Toc105266778)

[8.7 Test aléas de contrôle - donnée 38](#_Toc105266779)

[8.8 Circuit data\_hazard 39](#_Toc105266780)

[8.9 Commande des signaux dans main\_control\_unit 40](#_Toc105266781)

[8.10 Circuit data\_hazard 41](#_Toc105266782)

[8.11 Circuit Execute 42](#_Toc105266783)

[8.12 Test : pipeline forwarding 42](#_Toc105266784)

# Introduction

Ce travail s’inscrit dans le cadre de la formation Informatique et systèmes de communication de la HEIG-VD à Yverdon-les-Bains. Durant le cours « Architecture des ordinateurs » du 2e semestre, les élèves doivent réaliser un travail pratique portant sur le fonctionnement de la microarchitecture d'un processeur.

L’objectif de ce travail est la compréhension globale du fonctionnement des éléments de base d'un ordinateur, leurs caractéristiques, leurs performances et leurs interactions. En outre, savoir décrire l'organisation et la gestion de la mémoire d'un système à processeur, le fonctionnement et la composition de la microarchitecture d'un processeur.

Ce travail est la deuxième partie du projet global de conception d’un processeur. Dans la première partie, nous avions dû concevoir les parties principales du processeur, à savoir : fetch, decode, banque de registres, execute et memory access. Dans cette partie, nous implémenterons le pipeline et le forwarding.

## Méthodologie

Ce travail consiste à créer les circuits logiques nécessaires au fonctionnement d’un processeur proche du ARM9, ayant une architecture « Harvard ». Pour cela, nous utilisons le logiciel Logisim. Afin de mener à bien ce travail, il faudra mettre en pratique les notions apportées par le cours, en particulier les interactions des parties principales composant notre processeur, à savoir : le pipeline, la gestion des aléas et le forwarding. Pour tester nos circuits, nous allons devoir compiler divers programmes assembleur et donner au processeur les instructions à exécuter.

Ce travail est divisé en 2 étapes successives de réalisation :

1. Pipeline
2. Forwarding

Chaque étape est encadrée par une marche à suivre fournie. De plus, une grande partie des circuits composant le projet Logisim est déjà réalisée et nous est fournie. Chaque étape sera successivement analysée, réalisée et testée individuellement. Finalement, le circuit entier sera également testé pour valider son fonctionnement global.

Dans ce rapport, vous trouverez les explications du fonctionnement de chaque entité de notre circuit et de l’analyse, ainsi que la réalisation et les tests de ces dernières.

# Rappels fonctionnement du processeur

Afin de fonctionner correctement, notre processeur doit accéder aux instructions, les décoder, les exécuter, et lire/écrire les résultats dans la mémoire. Notre processeur devra utiliser le jeu d’instruction THUMB. Il s’agit d’une représentation 16 bits d'un sous-ensemble du jeu d'instructions ARM, qui offre l’avantage de réduire les besoins en mémoire.

Le processeur est composé de deux parties conceptuelles : l’unité de traitement, qui contient l’ensemble des opérateurs arithmétiques et logiques, et l’unité de contrôle qui coordonne les différentes activités du processeur.

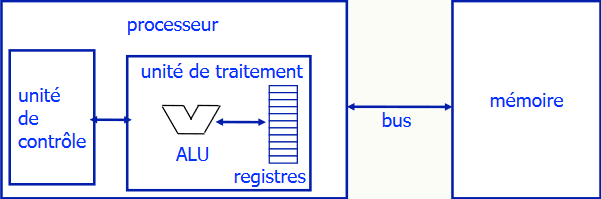


Figure 1 architecture d'un processeur (1)

Comme le processeur a une architecture Harvard, il dispose de 2 mémoires : la mémoire d’instructions, et la mémoire de données. Chaque mémoire dispose d’un bus d’adresses qui transporte les données spécifiant l'adresse d'un emplacement de mémoire, soit pour écrire à cet emplacement, soit pour lire à partir de cet emplacement. Les données d’écriture ou de lecture sont transportées par le bus de données.

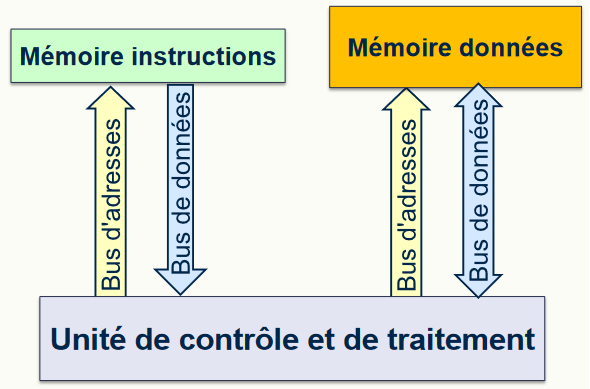
Sur les bus de la mémoire de données sont aussi connectées les entrées/sorties des périphériques potentiellement connectés dans notre système. Cependant, dans un souci de simplicité, pour ce travail pratique, nous n’en tiendront pas compte.

Figure 2 Architecture Harvard

## Architecture de notre processeur

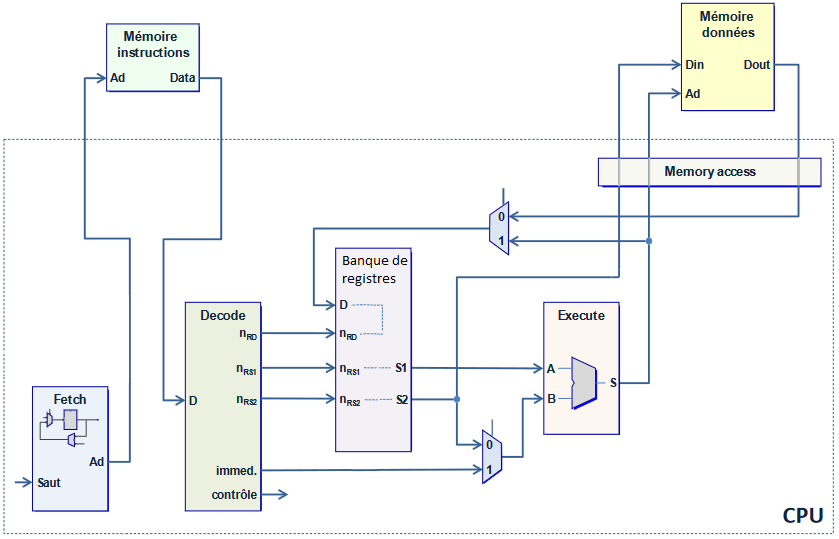


Figure 3 Architecture de notre processeur

La mémoire d’instructions contiendra toutes les instructions que notre processeur devra exécuter. Nous y placerons les instructions brutes sur 16 bits qui ont été compilées en assembleur. Dans la mémoire de données, nous iront, plus tard, placer les données de la pile (stack) qui permet de gérer les sauts et les retours de sauts.

# Processeur pipeliné

## Mécanismes à ajouter

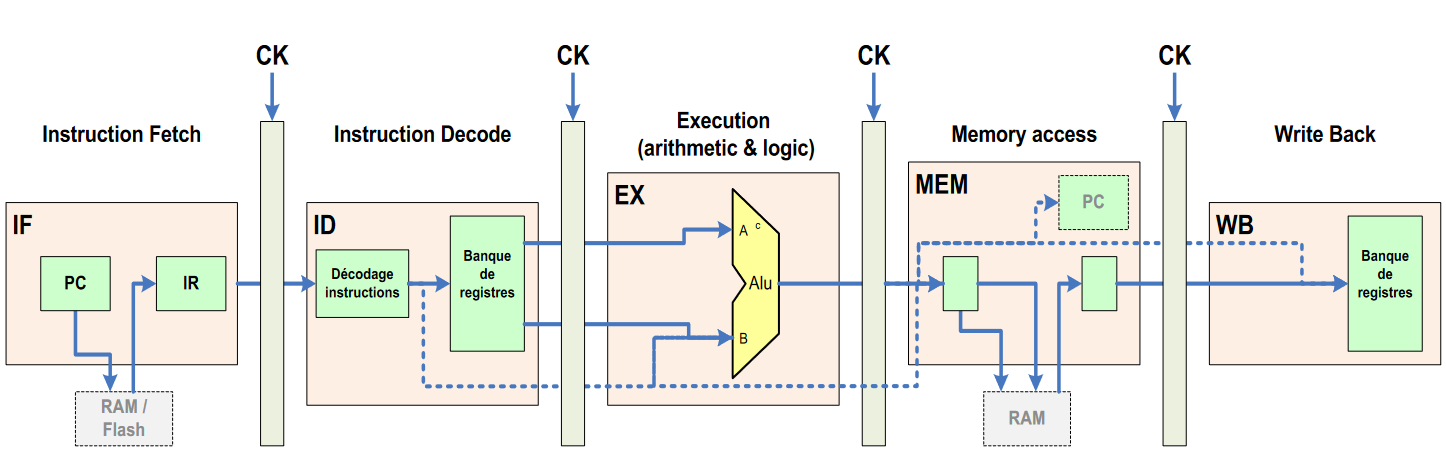
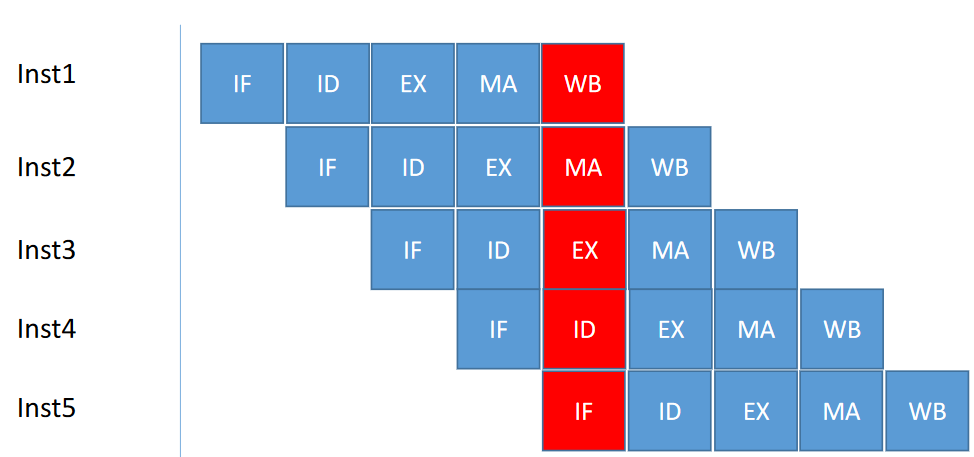
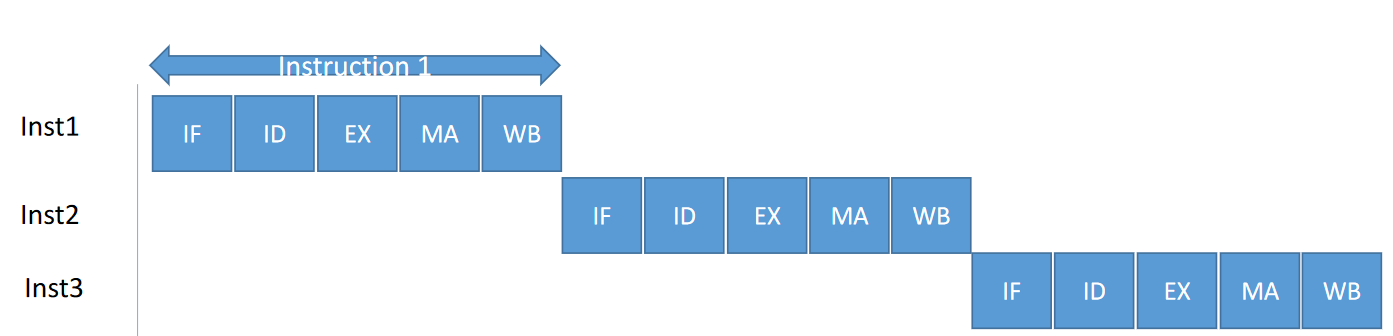
Comme mentionné dans l’introduction, nous devons modifier notre processeur simple pour qu’il fonctionne de manière pipelinée. Le pipeline permet d'accroître la vitesse d'exécution des instructions. Pour cela, le processeur est découpé en 5 étages :  
Fetch, Decode, Execute, Write back et Memory access. Le but est d’exécuter les instructions en cascade à chaque étage du pipeline, à la place d’attendre que l’instruction soit passé dans les 5 étages.

Figure 4 Exécution des instructions d'un processeur non pipeliné

Figure 5 Principe d'exécution en cascade du processeur pipeliné

Figure 6 Les 5 étages du pipeline

## Analyse et conception

Pour réaliser cette première partie, un circuit nous a été fourni. Il contient déjà toutes les modifications nécessaires du pipeline, à l’exception des circuits de détection des aléas. Il nous faudra, en aussi analyser les circuits qui ont été ajoutés pour comprendre le fonctionnement global de notre processeur

### Mémorisation de l’état des étages

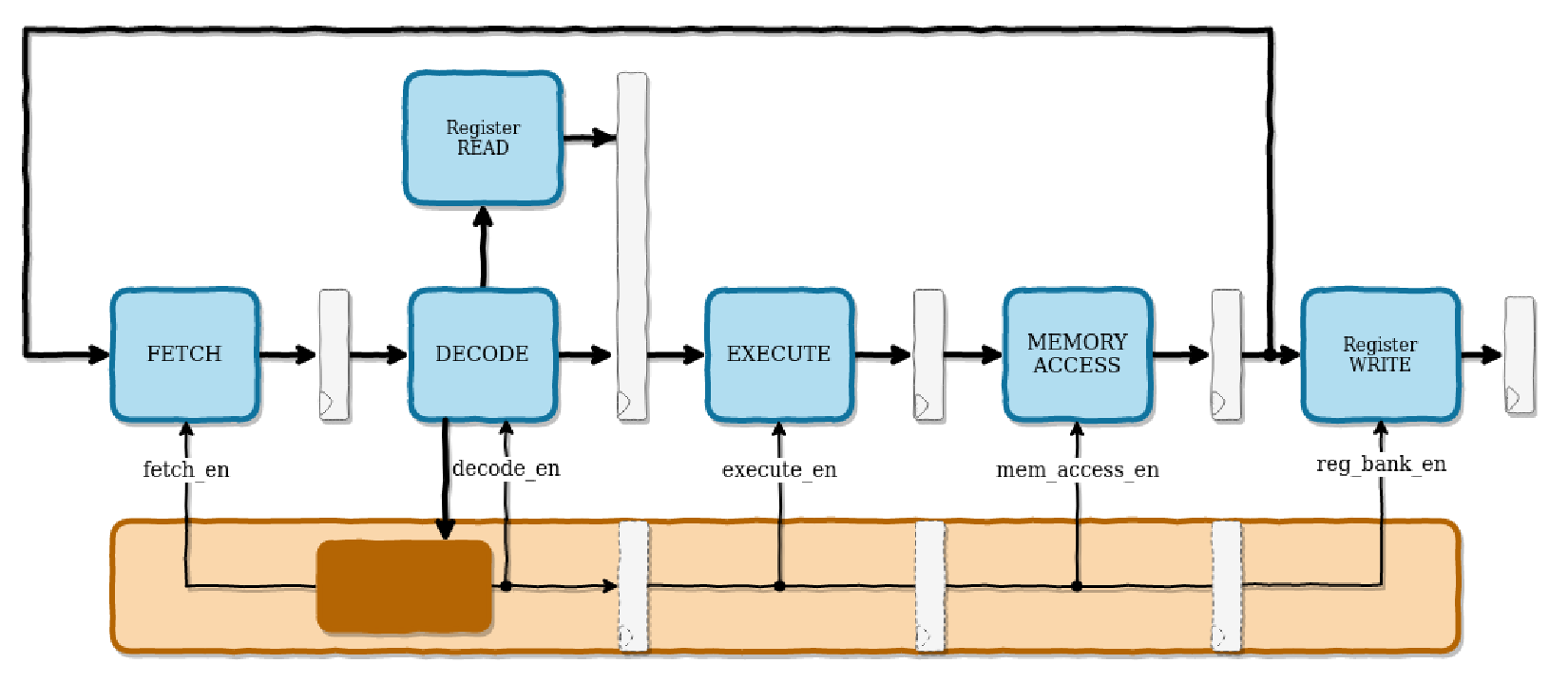
Pour pouvoir implémenter le forwarding, nous devons stocker la valeur de sortie de chaque étage du processeur. C’est grâce à cette mémorisation des états que nous pouvons utiliser le mécanisme d’exécution en cascade.

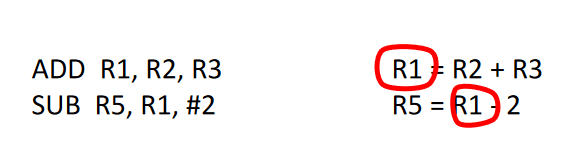
Figure 7 Diagramme conceptuel du processeur avec pipeline

Dans notre cas, il faut également s’assurer que tous les signaux de contrôle arrivent au bon moment à chaque étage. Le schéma ci-dessus est un croquis du processeur pipeliné. Les registres sont en gris. Sur le schéma, il y a un grand registre entre les stages du pipeline, or dans Logisim, il y a un registre par signal. Register READ et Register WRITE sont implémentés dans Logisim dans bank\_register.

### Dépendances et aléas de données

La mise en place d’in processeur pipeliné pose toutefois des problèmes de dépendances si deux instructions successives doivent avoir accès aux mêmes registres. Il existe 3 types de dépendances. RAW, WAR et WAW. Dans notre architecture, uniquement les dépendances de type RAW posent problème.

Une dépendance RAW (Read after write) existe si deux instructions successives doivent avoir accès aux mêmes registres. Plus exactement, si l’instruction n°2 doit lire un registre que l’instruction n°1 doit écrire. En effet, lorsque la première instruction est exécutée, la suivante doit attendre que l’étage Write Back ait écrit dans le registre N, avant que l’instruction n° 2 aille lire ce même registre, autrement l’instruction n°2 ira lire dans un registre qui contient une valeur non mise à jour. Nous apollons ça un aléa de données.

**Exemple :**Lors de l’exécution du programme suivant :

Nous voyons que l’instruction n°2 doit récupérer la valeur se trouvant dans le registre R1 et la soustraire avec la valeur 2. Le problème, c’est que le registre R1 est modifié par l’instruction précédente. C’est à l’étage décode que la lecture s’effectue dans les registres, et à l’étage WriteBack que s’effectue l’écriture.

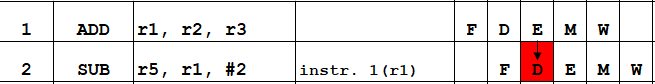
Si aucun mécanisme de détection d’aléa n’est mis en place, l’instruction 1 sera exécutée par l’étage Execute et l’instruction 2 sera exécutée par l’étage Fetch.

Figure 8 Aléa de données (1)

Il faut donc attendre que l’instruction précédente ait écrit le résultat de l’addition de R2 + R3 dans R1 avant que l’instruction n°2 aille lire dedans. En autres termes, attendre que l’instruction 1 ait terminé le write back pour que l’instruction 2 puisse effectuer le décode.

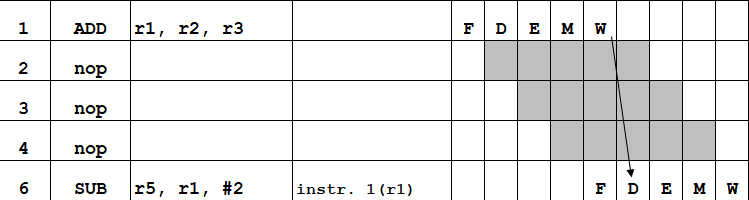
Pour réaliser cette « attente » nous pouvons utiliser 2 méthodes différentes : La résolution **software** et la détection **hardware**. La première méthode consiste simplement à ajouter, dans le code assembleur des instructions « nop » (no operation). Ces instructions ne font rien, mais permettent d’attendre 1 coup d’horlge du processeur.

Figure 9 Résolution d'aléas de données software

La deuxième solution consiste à implémenter une détection hardware d’un aléa de données. Le processeur arrêtera donc l’étage du processeur posant un probéme de dépendance pendant N coups d’horloge.

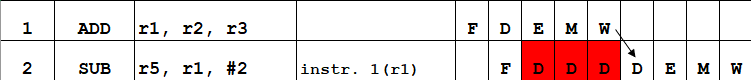


Figure 10 Résolution d'aléas de données hardware

### Aléas de contrôle

Les aléas de contrôle se produisent lors de sauts (Branch). En effet, pour que l’étage fetch sache à quelle adresse chercher une instruction suivant un saut, il recevoir l’adresse où chercher l’instruction. L’adresse de saut est calculée à l’étage Execute du pipeline.

L’étage Fetch de l’instruction suivant un saut doit donc attendre que l’instruction de saut ait précédemment calculé l’adresse de saut.

**Exemple :**

Lors de l’exécution du programme suivant :

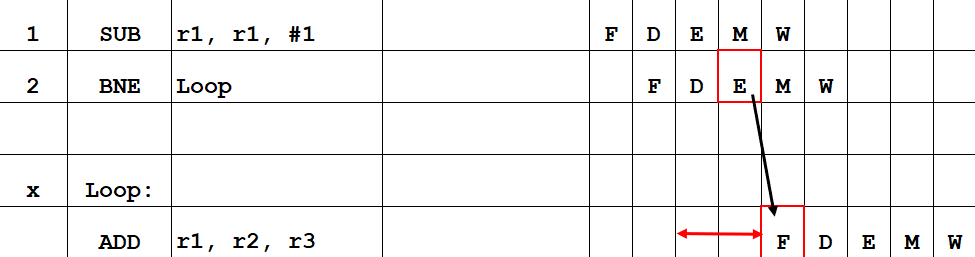


Figure 11 Exemple d'un aléa de contrôle

Nous partons du principe que la condition du saut est prise. L’étage Fetch de l’instruction suivante doit attendre que l’étage execute de l’instruction BNE ait calculé l’adresse de saut, pour pouvoir fetcher l’instruction ADD dans la mémoire de données. Nous voyons que cet arrêt coûte 2 cycles d’horloge, et qu’il faut 3 coups d’horloge à partir du fetch jusqu’au prochain fetch.

Pour résoudre ces aléas de contrôle, nous devons implémenter un circuit de gestion dans le processeur. Le processeur arrêtera donc les étages problématiques de l’instruction suivant un saut durant 2 cycles.

## Travail demandé

Dans ce chapitre, nous allons reprendre le travail demandé de la donnée du laboratoire, pour expliquer notre démarche.

### Test du processeur avec un programme

Il nous a été demandé d’analyser l’exécution du programme ci-dessous. Avant de prendre un chronogramme, nos allons réaliser un diagramme des aléas.

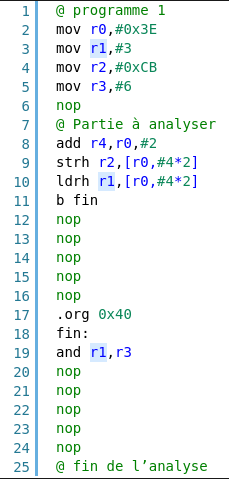
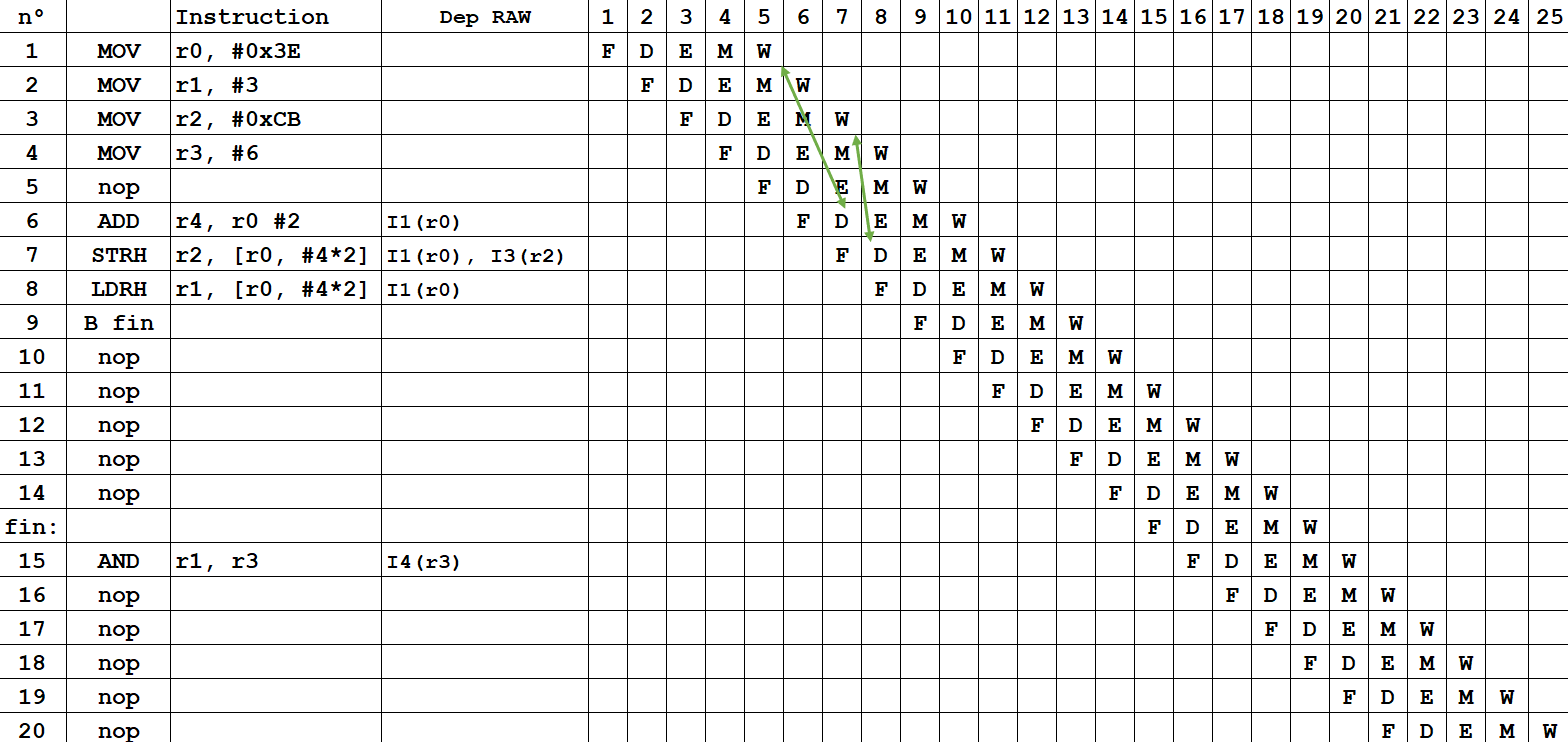


Figure 12 Programme à analyser Test du processeur

Figure 13 Table des aléas Test du processeur

Dans ce tableau des aléas, nous pouvons voir qu’il y existe des dépendances pour les instructions 6, 7, 8 et 15. Cependant, elles sont suffisamment espacées en termes de coups d’horloge pour ne pas générer d’aléas (voir flèches vertes).

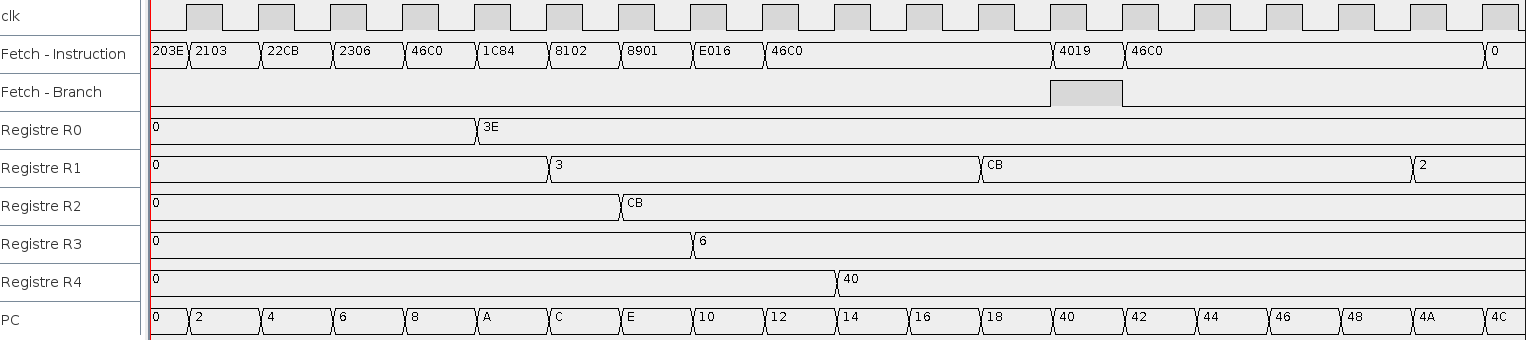
* Les valeurs sont bien déplacées dans les registres durant les 4 premiers coups d’horloge.

Figure 14: Chronogramme Test du processeur

* L’addition de 0x3ce + 2 est juste. Nous voyons que le résultat (0x40) est bien stocké dans le registre 4.
* Les instructions LDRH et STRH fonctionnent car nous voyons que la valeur 3 dans r2 stockée à l’adresse [r0, #4\*2] (PC = 16) est bien récupérée dans le registre r1 au coup d’horloge suivant (PC = 18).
* Le saut s’execute correctement, car on passe de PC = 18 à PC = 40 au moment ou la détection de saut est à 1 dans le fetch
* Finalement, l’instruction AND stocke la bonne valeur dans r1.

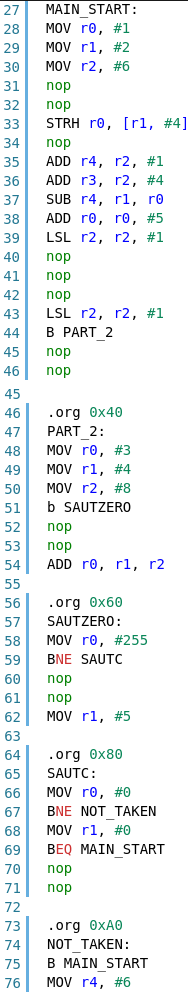
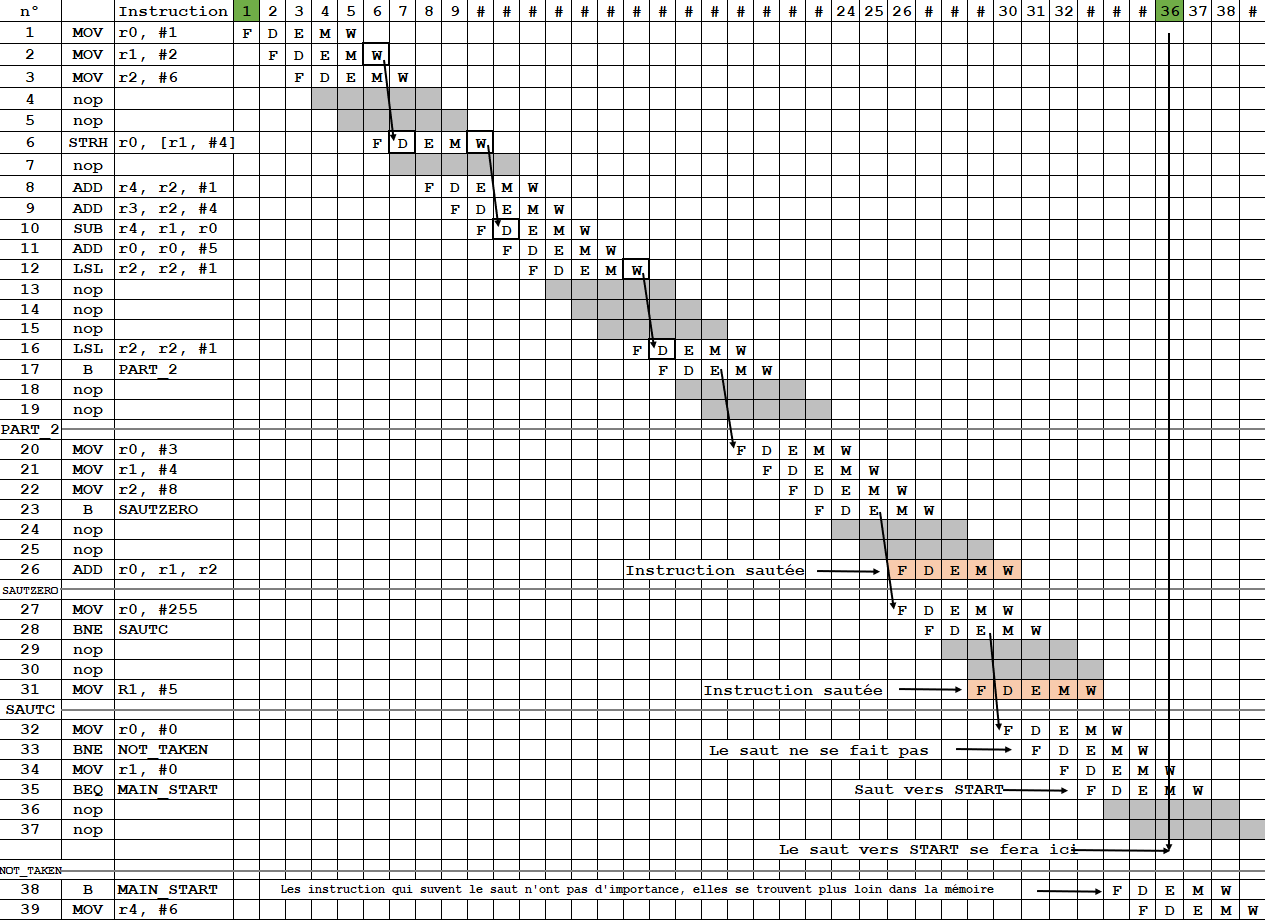
Nous avons 9 instructions (sans nop) pour un total de 36 coups d’horloge

**Calcul de l’IPC :**

### Assembleur : dépendances de données et Assembleur : aléas de contrôle

Un autre programme nous a été donné. Comme fait précédemment, nous allons l’analyser pour savoir ou placer nos nop dedans :

Figure 15 Programme et table des aléas dépendances de données et de contrôle

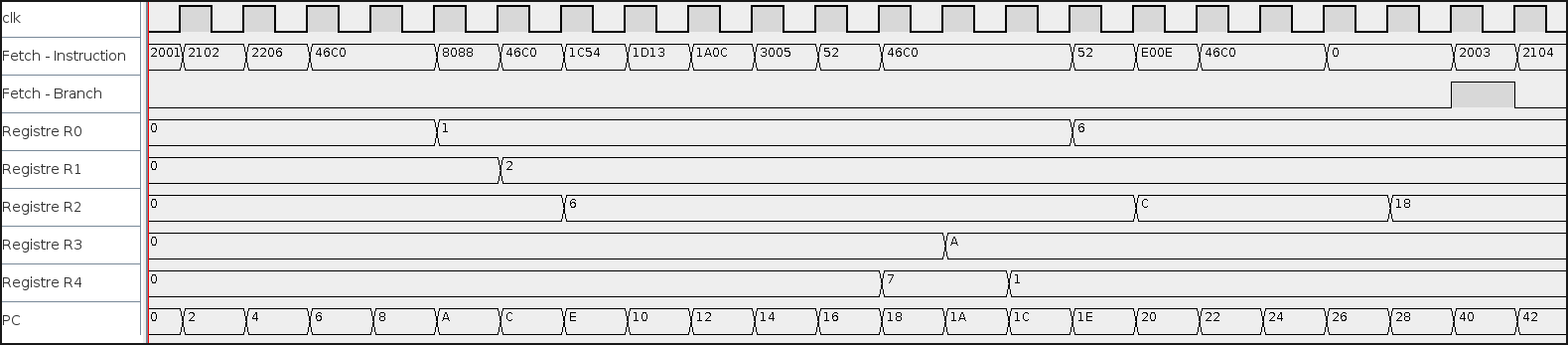


**Explication de notre démarche jusqu’à PART\_2  :**

* Il y a un aléa de données entre le MOV (I3) et le STRH(I6), nous avons donc ajouté 2 nop.
* Il y a un aléa de données entre le STRH (I6) et le ADD (I8), nous avons donc ajouté 1 nop.
* Il y a un aléa de données entre les 2 instructions LSL (I12 et I13). Nous avons donc décidé d’ajouter 2 nop pour pallier l’aléa.

**Chronogramme pour vérifier notre théorie :**

Figure 16 Chronogramme aléas de données



3x nop

3x nop

B PART\_2

12

14

16

18

20

6

8

10

2

4

22

* La flèche bleue nous montre qu’à partir du fetch de l’instruction, il nous faut 5 coups d’horloge pour avoir la valeur dans un registre
* Nous pouvons voir qu’au 12e coup d’horloge nous avons 0x7 dans R4, ce qui correspond au WB de ADD r4, r2, #1 avec r2 = 6
* Au 14e Nous avons 0xa dans r3, ce qui correspond au WB de ADD r3, r2, #4, avec r2 = 6
* Au 15e nous avons 0x1 and r4 ce qui correspond au WB de SUB r4, r1, r0 avec r1 = 2 et r0 = 1
* Au 16e nous avons 0x6 dans r0, ce qui correspond au WB de ADD r0, r0, #5 avec r0 = 1
* Au 17e nous avons 0xc dans r2 ce qui correspond au WB de LSL r2, r2, #1 avec r2 = 6
* Au 21e nous avons 0x18 dans r2 ce qui correspond au WB de LSL r2, r2, #1 avec r2 = c

Nous pouvons en conclure que notre programme fonctionne

**Explication de notre démarche depuis PART\_2 :**

Nous avons placé nos nop après chaque saut, car des instructions pourraient se trouver aux adresses qui suivent les sauts dans la mémoire de données, et nous ne voulons pas que le processeur traite quoi que ce soit durant la durée de traitement de nos sauts.

* Nous avons remarqué qu’il y avait un aléa de contrôle après le premier saut B PART\_2 (I17).
* Ensuite, nous voyons que l’instruction B SAUTZERO est suivie d’une instruction ADD. Pour éviter d’exécuter cette instruction nous avons placé 2 nop
* Après l’instruction BNE SAUTC, nous avons aussi placé 2 nop pour éviter d’exécuter l’instruction qui la suit
* Le saut BNE NOT\_TAKEN n’est jamais pris, car sa condition est toujours fausse. Nous devons tout de même ajouter 2 nop
* C’est donc le saut suivant (BEQ MAIN\_START) qui sera pris. Sa condition étant toujours vraie, on sautera toujours vers MAIN\_START.

Dans ce programme théorique, nous avons 21 instructions traitées (sans nop) pour un total de 36 cycles.

**Calcul du temps de cycle avec une fréquence de 4kHz :**

**Calcul de l’IPC théorique** :

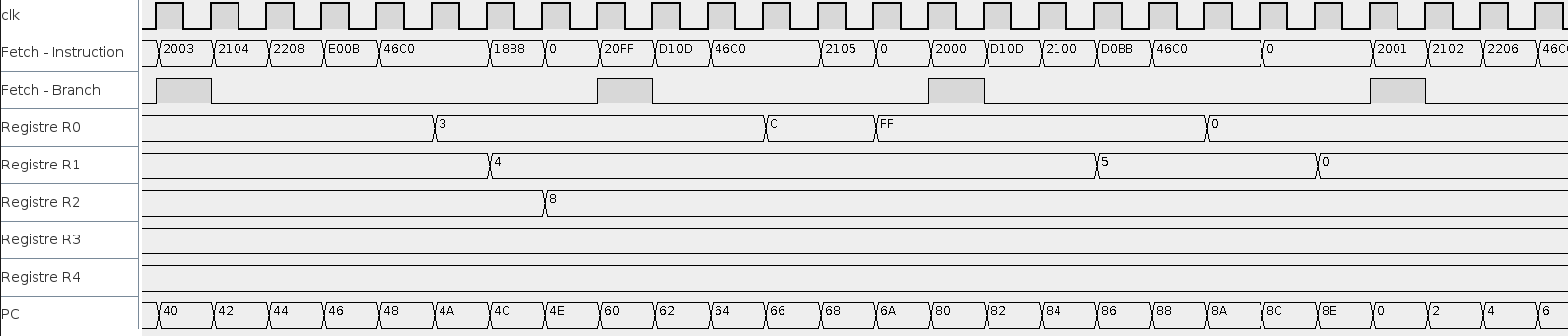
**Calcul de latence :**

**Calcul du débit pour une horloge de 4kHz :**

Pour être sûr de notre prédiction, nous allons réaliser un chronogramme. Nous prédisons qu’il faudra 38 coups d’horloge pour exécuter notre programme.

En voyant ce chronogramme, nous nous rendons compte qu’il faut 4 coups d’horloge depuis l’execute de l’instrution de saut jusqu’au saut effectif, et non 2 coups comme nous l’avions appris en théorie. Cela vient probablement du fait que l’adresse du saut, passe par les étages Memory access et Bank registers, ce qui demande au signal de faire 2 coups d’horloge supplémentaire avant d’arriver au fetch. C’est dommage, il suffirant de faire passer ce signal directemeent depuis l’execute vers le fetch pour gagner 2 coups d’horloge (pour ARO l’année prochaine !).

Figure 17 Chronogramme aléas de contrôle



22

24

28

30

34

36

38

40

26

42

32

B SAUTZERO

BNE SAUTC

BEQ START

BNE NOT\_TAKEN

F

D

E

44

Cela implique que nous finissons notre programme en 44 coups d’horloge à la place des 36 prédits dans notre tableau des aléas. Etant donnée que nos saut prennent 2 coups d’horloge de plus ce résultat est logique : 4 sauts x 2 cycles supplémentaires + 36 cycles prédits = 44 cycles au final. Maintenant, nous savons qu’il faut prévoir 4 coups d’horloge pour éviter un aléa de contrôle. Dans ce le vrai programme, nous avons 21 instructions effectuées pour un total de 44 cycles.

**Calcul de l’IPC réel :**

**Calcul du débit réel :**

### Circuits pour la détection des aléas

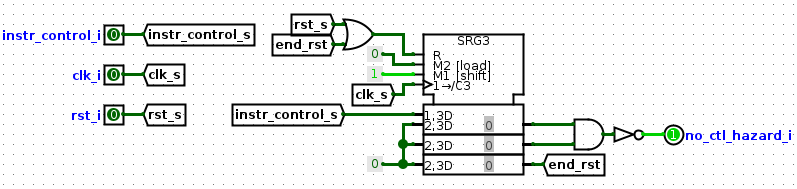
Nous avons vu comment pallier aux aléas de contrôle en ajoutant des nop dans notre programme assembleur. Il s’agit maintenant d’implémenter l’arrêt de pipeline hardware. Cela évite au programmeur la tâche de connaitre précisément le fonctionnement du processeur. Pour cela nous avons dû compléter 1 circuit. Il se trouve dans hazard\_detection, dans main\_control\_unit, dans le bloc décode.

Figure 18 circuit control hazard

Nous avons décidé de gérer les aléas de contrôle avec un registre à décalage. A chaque fois qu’une instruction de contrôle est détectée, nous mettons le signal no\_ctl\_hazard à 0 pendant 2 cycles.

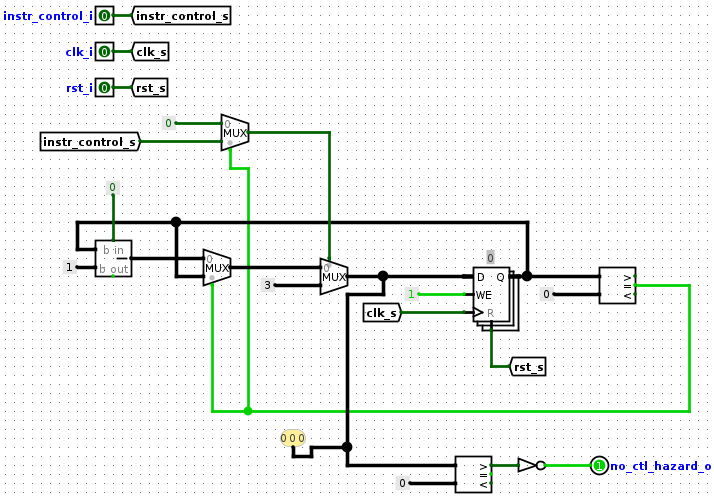
Cela était dans notre hypothèse qu’il ne fallait que 2 coups d’horloge pour pallier à un aléa de contrôle. Or, comme vu précédemment, il nous faut en réalité 4 coups d’horloge. La solution proposée qui nous a été fournie est différente.

Figure 19 Control hazard proposé

Nous voyons que ce circuit joue le rôle d’un décompteur. A chaque fois qu’une instruction de saut est détectée, nous décomptons de 3 à 0. Lors du décompte la sortie no\_ctrl\_hazard est à 0, ce qui arrête les différents étages du pipeline durant 4 cycles (voir suite).

### Circuit d’arrêt de pipeline lors d’aléas de contrôle

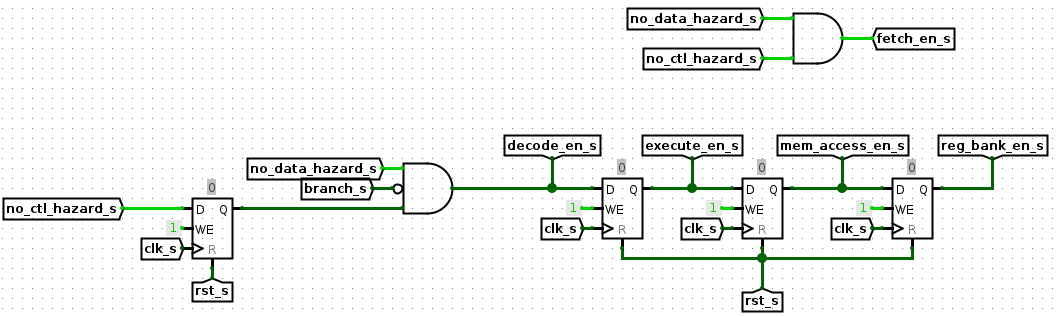
Maintenant que nous avons un signal qui nous permet d’arrêter notre pipeline pendant 4 cycles, il nous faut un circuit d’arrêt en cascade. La solution qui nous a été fournie est la suivante :

Figure 20 Circuit d'arrêt de pipeline lors d'aléas de contrôle

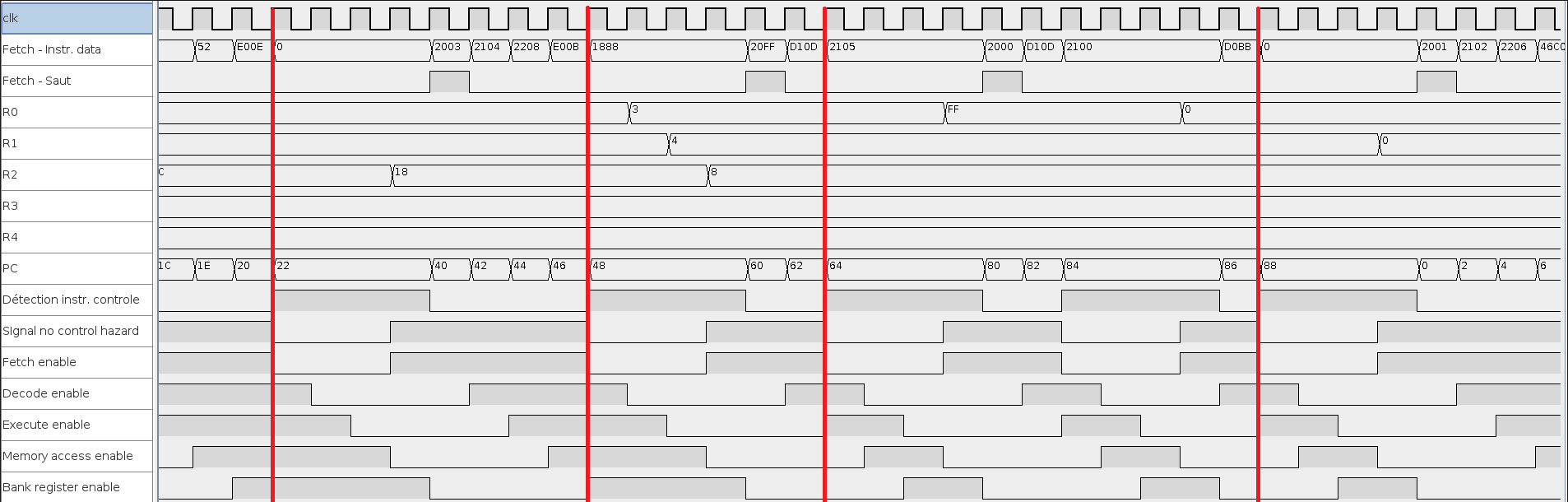
Nous voyons dans ce circuit, que tant que le signal no\_ctrl\_hazard est à 1, tous les enable des étages du pipeline sont à 1. Lorsque le signal passe à 0, les enable de chaque circuit va passer successivement à 0, pendant 4 cycles en commençant par le fetch (lié à la porte AND en haut), puis un cycle plus tard, le décode, etc.

Cette désactivation en cascade permet au traitement de la première instruction de s’effectuer jusqu’à la fin.

### Test aléas de contrôle avec la solution proposée

Comme précédemment, nous allons, maintenant analyser le programme, mais cette fois-ci sans nop, avec un chronogramme pour vérifier qu’il fonctionne comme prévu.

Figure 21Chronogramme de vérification du circuit fourni pour les aléas de contrôle



En rouge sont les instructions de saut fetchées. Nous voyons que lorsqu’une instruction de saut est prise, les étages du pipeline vont s’arrêter en cascade pendant 4 cycles.

Ce qui est plus étrange, c’est que si un saut n’est pas pris, en l’occurrence le saut BNE NOT\_TAKEN (instr. D10D), le signal no\_ctrl\_hazard passe quand même à 0. C’est étrange, car si un saut n’est pas pris, nous n’avons aucune raison d’arrêter les étages du pipeline, car nous n’avons pas besoin de calculer l’adresse du saut. Pour pallier à ce problème (p.ex pour le cours de l’année prochaine), il faudrait ajouter une condition à l’entrée du bloc control\_hazard, qui prendrait le résultat du CPSR de l’execute, et en fonction du type de saut conditionnel utilisé, indiquer si le saut sera pris ou pas.

Au final, nous pouvons constater que l’implémentation du contrôle des aléas de contrôle fonctionne, mais n’est pas optimal.

# Processeur pipeliné avec forwarding

Dans cette partie, nous mettrons en place le mécanisme de forwarding, ainsi que la gestion des aléas de données. Nous n’avons pas dû compléter de circuits, mais uniquement les analyser pour vérifier qu’ils fonctionnement.

## Mécanismes à ajouter pour les aléas de données

### Gestion des aléas de données

Dans le chapitre précédent, nous avons su gérer les aléas de contrôle en ajoutant un mécanisme d’arrêt de pipeline dans notre processeur. Dans cette partie, nous allons utiliser la même logique pour gérer les aléas de données.

Nous allons réutiliser la même logique de désactivation en cascade que pour les aléas de contrôle. Pour détecter une dépendance de type RAW qui générerai un aléa, il suffit de mémoriser les 3 dernières instructions qui ont été fetchées. Si l’instruction en cours doit lire un registre qui a été écrit par l’une des 3 dernières instructions, nous devrons arrêter le pipeline.

Pour savoir combien de cycles nous devons arrêter le pipeline, il suffit de savoir quelle instruction précédente a écrit le registre que nous devons lire :

1. Instruction -1 => arrêt de 3 cycles
2. Instruction -2 => arrêt de 2 cycles
3. Instruction -3 => arrêt de 1 cycle

Le challenge de ce circuit sera de savoir quelle instruction écrit dans quel registre. Pour chaque instruction nous avons 4 signaux à comparer : l’adresse du registre de destination, l’adresse du registre n et l’adresse du registre m et l’adresse du registre m.

## Analyse du circuit fourni

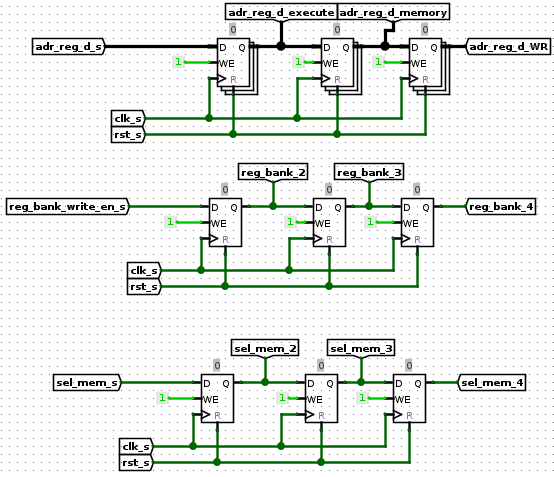
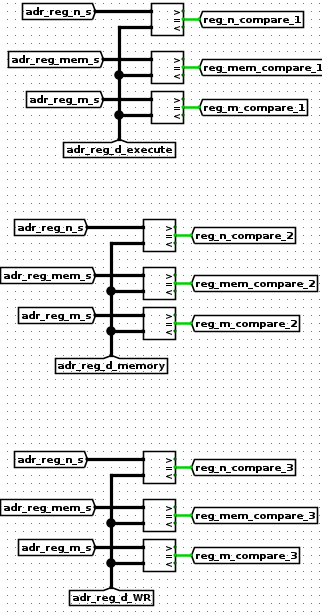
Cette partie du circuit est dédiée à la mémorisation des registres d’écriture des 3 dernières instructions.

Figure 22 Partie du circuit data\_hazard (1)

Dans cette partie, le circuit compare l’adresse de des registres de lecture avec les adresses des registres d’écriture des 3 dernières instructions. Cela sert à identifier un aléa de type RAW

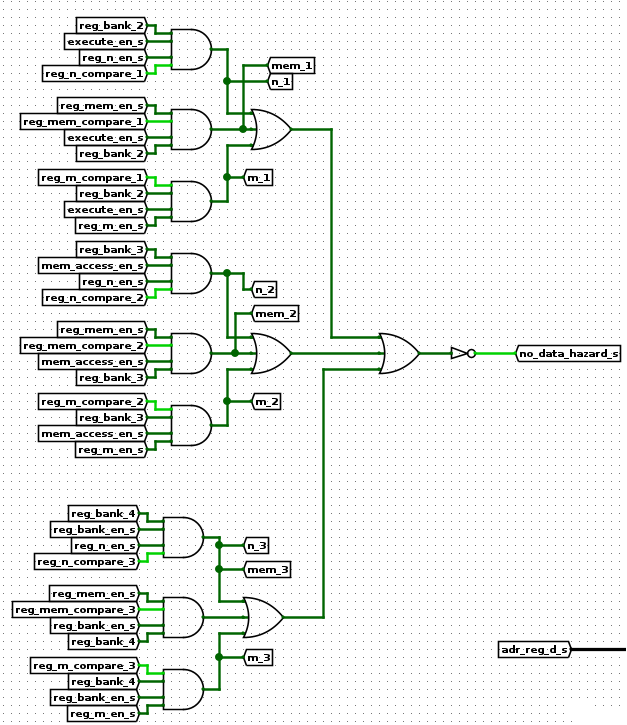
Dans cette partie du circuit, nous identifions si le signal de détection d’un aléa doit être activé. Ce signal dépend de plusieurs paramètres :

Figure 23 Partie du circuit data\_hazard(2)

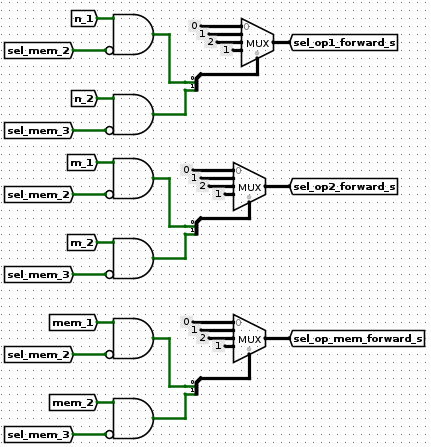
Pour que le signal soit activé, il faut que nous détections qu’une instruction veuille lire dans un registre qui a été écrit récemment (Figure 23). Il faut également vérifier que l’étage du pipeline associé soit activé et que la banque de registre soit activée.

Figure 24 Partie du circuit data\_hazard (3)

Cette partie nous sert pour le forwarding. Nous l’analyserons plus tard.

Figure 25 Partie du circuit data\_hazard (4)

### Test : aléas de donnée

Pour tester notre circuit, nous allons à nouveau créer un code avec des aléas, un tableau des aléas et un chronogramme

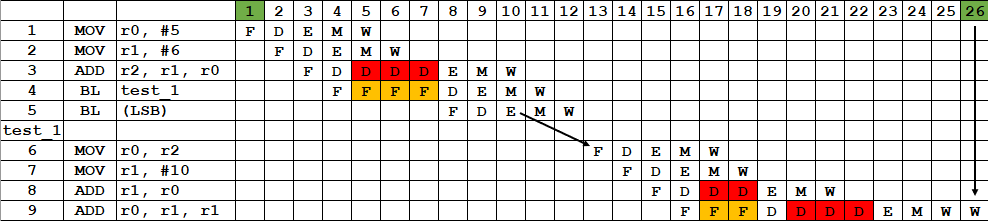
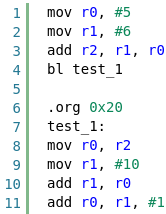


Figure 26 Programme et tableau des aléas pour le test des aléas de données

Dans ce programme théorique, nous avons 9 instructions traitées (le BL compte pour 2 instructions), pour un total de 26 coups d’horloge.

**Calcul du temps de cycle avec une fréquence de 4kHz :**

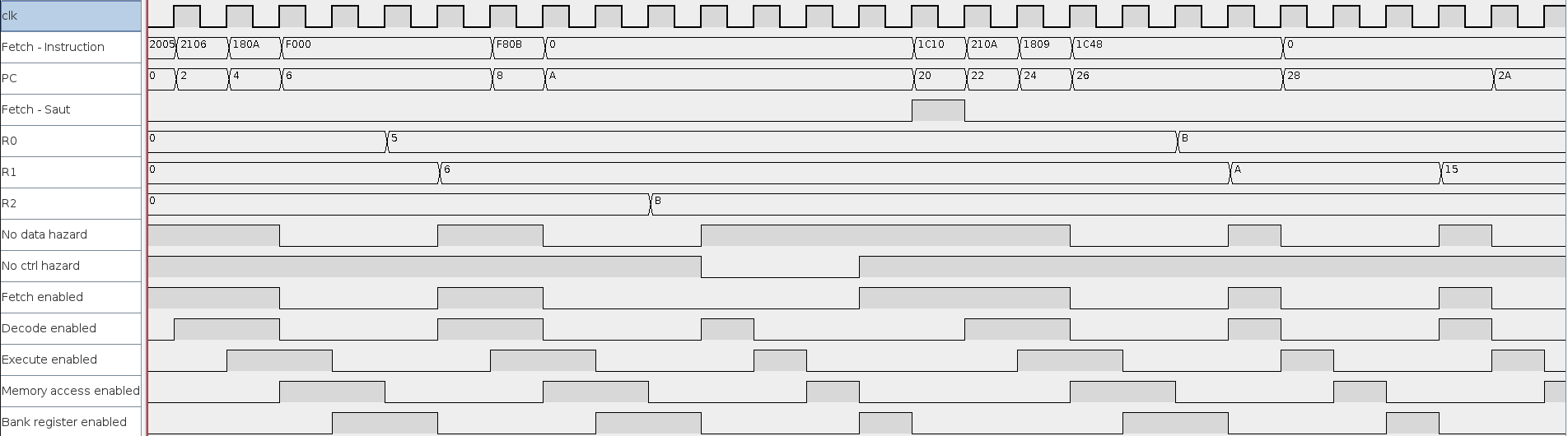
**Calcul de l’IPC théorique** :

**Calcul de latence :**

**Calcul du débit pour une horloge de 4kHz :**

Dans ce chronogramme, nous sommes à nouveau surpris de voir que nous n’obtenons pas le résultat prédit => 27 coups d’horloge.

Figure 27 Chronogramme pour le test des aléas de données



2

4

6

8

10

12

14

16

18

20

22

24

26

* Au niveau de la détection des aléas, nous voyons que ça ne fonctionne pas comme prévu. Lors de l’instruction add r1, r0 (Instr.1809), nous ne devrions avoir un arrêt que de 2 coups d’horloge. Cependant nous avons un arrêt de 3 coups d’horloge, c’est étrange.
* Si non les autres aléas sont gérés avec 3 coups d’horloge comme prévu
* A noter que l’instruction BL est composée de 2 instructions et qu’elle génère 1 aléa de données et 1 aléa de contrôle 0=> 8 coups d’horloge
* C’est étrange que lors d’une détection d’aléa de données, l’étage fetch et decode se fasse en même temps

**Calcul de l’IPC théorique** :

**Calcul du débit pour une horloge de 4kHz :**

## Mécanismes à ajouter pour le forwarding

Le forwarding est une méthode permettant de perdre moins de temps de cycles lors d’un aléa de données de type RAW. Pour éviter de devoir attendre que l’étage Write Back ait fini d’écrire le résultat de l’instruction précédente dans le registre D, nous stockons directement le résultat pour qu’il soit disponible immédiatement lors du traitement de l’instruction suivante.

Ce mécanisme ne permet malheureusement pas de de traiter les aléas pour les valeurs qui proviennent d’une instruction mémoire (LDR, STR). Cependant, pour toutes les autres instructions, cela ne coute pas de temps de cycle supplémentaire lors d’un aléa.

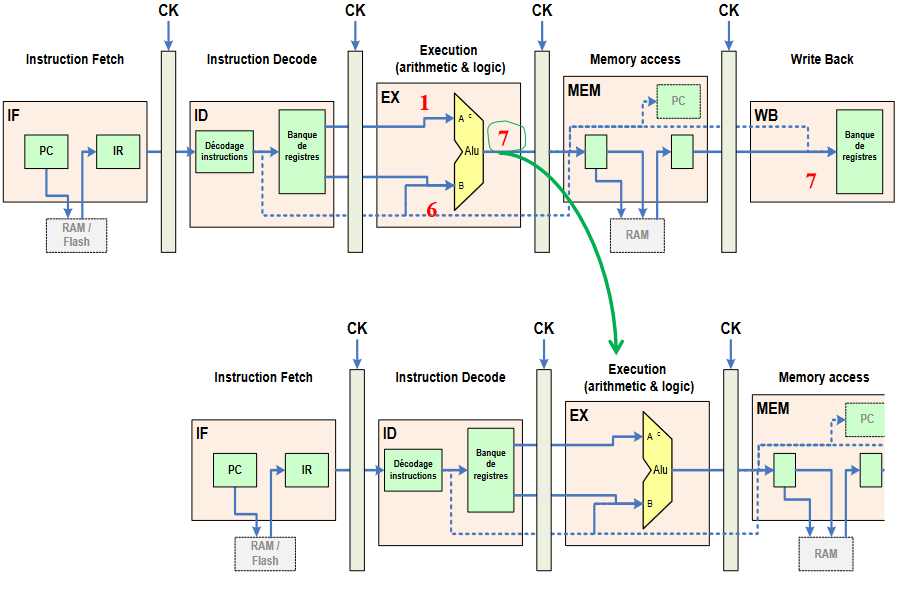


Figure 28 Représentation du principe de forwarding

### Pipeline Forwarding : Circuit data\_hazard

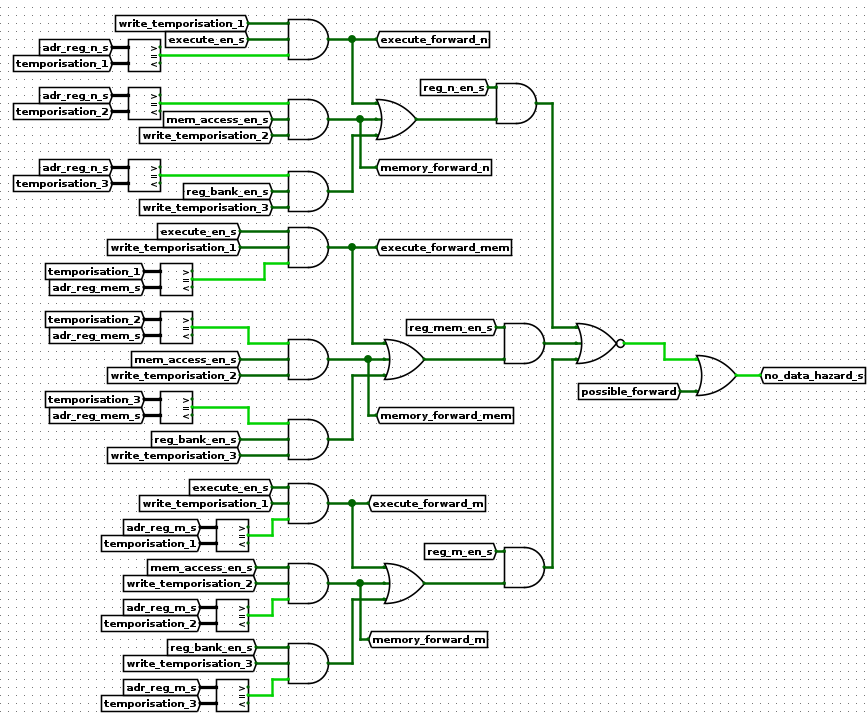
Pour implémenter le forwarding, nous devons effectuer quelques modifications dans le circuit data\_hazard.

Figure 29 Forwarding : modifications de data\_hazard(1)

Sur l’image précédente, nous pouvons voir qu’il a été ajouté 2 types de signaux distincts :

1. : Les signaux de forwarding depuis le bloc execute
2. Les signaux de forwarding depuis le bloc memory access

Dans le premier cas, ces signaux seront activés si les adresses des registres de lecture de la banque de registres sont les mêmes que les adresses d’un registre d’écriture de l’une des 3 dernières instructions.

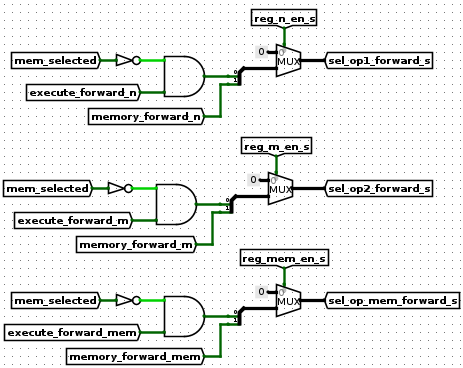
Dans le deuxième cas, les signaux seront activés si les adresses des registres de lecture en mémoire sont les mêmes que les adresses d’un registre d’écriture de l’une des 3 dernières instructions.

Figure 30 Forwarding : modifications de data\_hazard(2)

Ensuite, nous avons la sélection des opérandes à utiliser dans l’execute pour le forward. En fonction du registre de lecture, et de la nature du registre de lecture (en mémoire au dans la banque de registres), nous sélectionnons le signal désiré.

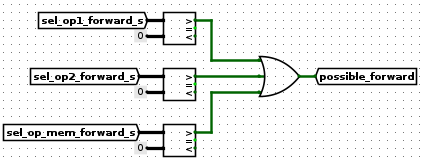
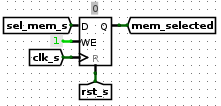
Ce circuit permet simplement de désactiver la sortie no\_data\_hazard, si l’une des opérandes à forwarder est à 0.

Figure 31 : Forwarding : modifications de data\_hazard(4)

Figure 32 Forwarding : modifications de data\_hazard(3)

Cette mémorisation permet de retarder d’un coup d’horloge l’inforation qu’une instruction d’accès à la mémoire de données est décodée. Typiquement les instructions LDRH et STRH.

### Pipeline Forwarding : Circuit execute

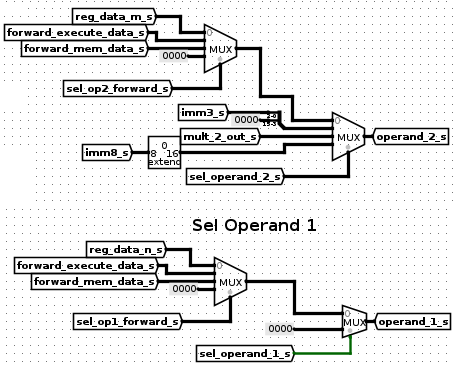
Pour implémenter le forwarding, nous devons effectuer quelques modifications dans le circuit execute. Pour la gestion des aléas de données, il faut prendre la valeur du dernier résultat de l’execute pour pouvoir effectuer l’instruction courante. Pour faire cela, nous devons sélectionner les signaux sel\_op\_forward avec les deux signaux sel\_operand. Lors d’un forwarding, les signaux sel\_op\_forward doivent prendre valeur forward\_execute\_data\_s ou forward\_mem\_data\_s dépendant de si le forwarding doit être utilisé dans l’étage execute ou dans l’étage memory access.

Figure 33 Forwarding : modifications du bloc execute (1)

Nous pouvons constater sur l’image précédente, qu’un multiplexeur a été ajouté pour la sélection de l’opérande 1 et 2. Cela permet de donner la valeur qui a été mémorisée à l’opérande 1 et/ou l’opérande 2.

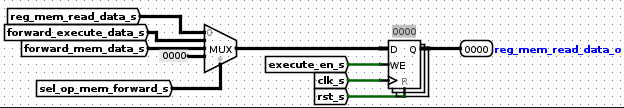


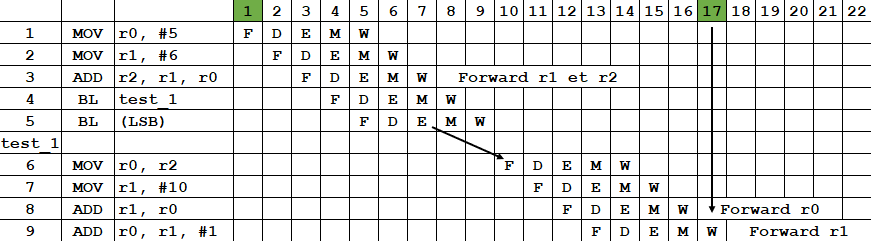
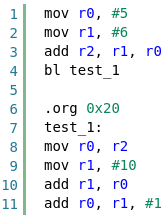
Figure 34 Forwarding : modifications du bloc execute (2)

Finalement, la même chose a été fait pour les signaux qui doivent aller au blocs Memory access.

### Test : pipeline forwarding

Nous allons réutiliser le même programme que celui pour le test des aléas, sans forwarding. Nous prédisons 17 coups d’horloge pour ce programme avec le forwarding. Nous prenons en compte le fait qu’il faille 4 coups d’horloge pour un saut.

Figure 35 Test du Forwarding : Programme et tableau des aléas



Dans ce programme théorique, nous avons 9 instructions traitées (le BL compte pour 2 instructions), pour un total de 17 coups d’horloge.

**Calcul du temps de cycle avec une fréquence de 4kHz :**

**Calcul de l’IPC théorique** :

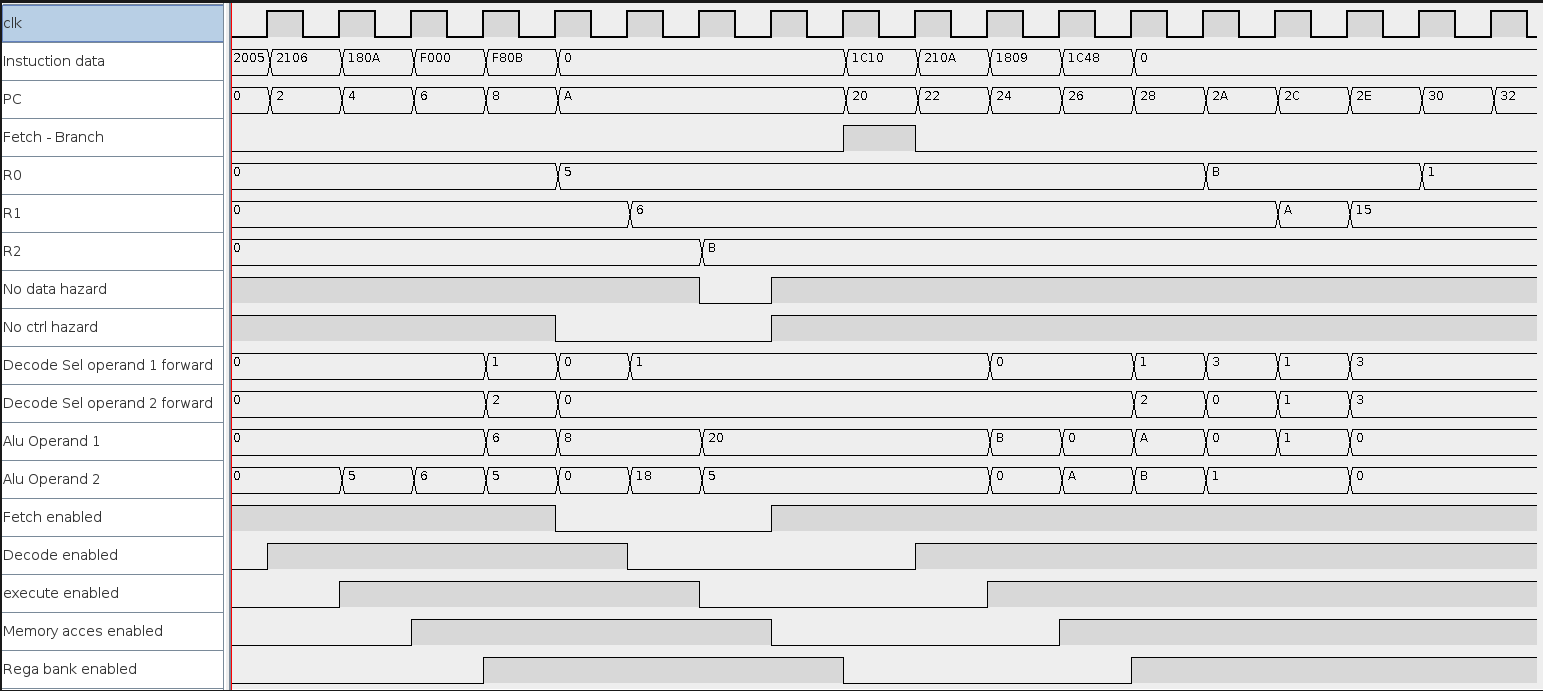
**Calcul de latence :**

**Calcul du débit pour une horloge de 4kHz :**

Soit une amélioration de 800 coups d’horloge comparé au processeur sans forwarding.

Nous pouvons observer que notre théorie est juste, car nous obtenons le même nombre de coups d’horloge dans ce chronogramme.

Figure 36 Test forwarding : Chronogramme



2

4

6

8

10

12

14

18

16

Nous pouvons observer que le BL prend 4 cycles pour s’effectuer, car l’aléa de données est pis en charge par le forwarding.

Cependant, ce qui est étrange c’est la valeur dans R0 après le write back de la dernière instruction (cycle 18). Normalement nous devrions avoir 22 et non pas 1. Il s’agit probablement d’une erreur de sélection de l’opérande dans le bloc execute, mais nous n’avons pas réussi à trouver l’origine de cette erreur.

# Conclusion

De notre point de vue, toutes les fonctionnalités de ce processeur ne sont pas fonctionnelles. En effet, nous avons pu relever un problème de gestion d’aléa de contrôle au point 3.3.5, le saut perd 4 coups d’horloge au lien de 2 et un saut conditionnel qui n’est pas pris génère aussi un aléa de contrôle. Au point 4.2.1 nous avons remarqué qu’un aléa de contrôle sensé se résoudre en 2 cycles, en prend 3. Finalement au point 4.3.3 la valeur dans un registre de destination n’est pas correcte. Cependant, la majorité des ajouts du processeur fonctionnent bien et sommes ravis d’avoir pu analyser ces différentes étapes de construction.

# Bilan personnel

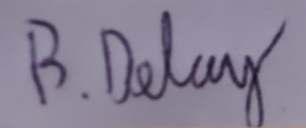
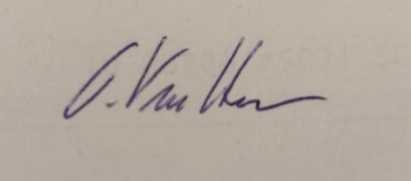
Cette deuxième partie de travail était bien différente de la première. Nous n’avons presque pas dû compléter de circuits, mais plutôt analyser les circuits qui nous ont été fournis. Ce travail d’analyse est moins évident qu’un travail de construction, car il est plus difficile de comprendre le fonctionnement global des mécanismes.

Nous trouvons dommage que nous apprenions, en théorie des choses différentes que ce que nous devons mettre en pratique, car cela nous pousse à la confusion. Voici quelques exemples :

* Nous apprenons qu’n aléa de contrôle fait perdre 2 coups d’horloge, alors que dans le circuit fourni il en fait perdre 4
* En théorie, Le Write back s’effectue dans un bloc dédié mais en pratique, elle se fait dans la banque de registres
* En théorie, nous apprenons que la gestion du forwarding s’effectue dans le bloc Execute. Dans notre cas, elle se fait dans le bloc decode
* Nous pouvons voir que la gestion des aléas, et en général la gestion de tous les signaux de notre processeur se fait dans le bloc décode, et pour nous ce n’est pas très intuitif.

Nous avons trouvé ce travail très intéressant, car il nous a vraiment plongé au cœur du fonctionnement d’un processeur. Grâce à ce travail nous arrivons mieux à comprendre les mécanismes de programmation bas niveau, de pile, d’IPC et de gestion de la mémoire.

Yverdon-les-Bains, le 04.06.2022



Benoît Delay Timothée Van Hove

# Table des illustrations

[Figure 1 architecture d'un processeur (1) 4](#_Toc105265849)

[Figure 2 Architecture Harvard 4](file:///C:\Users\ROG\Documents\HEIG\ARO\HEIG_ARO_LaboPipelinePartie2_BDY_TVE\Rapport%20ARO%20Partie%202.docx#_Toc105265850)

[Figure 3 Architecture de notre processeur 5](#_Toc105265851)

[Figure 4 Exécution des instructions d'un processeur non pipeliné 6](file:///C:\Users\ROG\Documents\HEIG\ARO\HEIG_ARO_LaboPipelinePartie2_BDY_TVE\Rapport%20ARO%20Partie%202.docx#_Toc105265852)

[Figure 5 Principe d'exécution en cascade du processeur pipeliné 6](file:///C:\Users\ROG\Documents\HEIG\ARO\HEIG_ARO_LaboPipelinePartie2_BDY_TVE\Rapport%20ARO%20Partie%202.docx#_Toc105265853)

[Figure 6 Les 5 étages du pipeline 6](file:///C:\Users\ROG\Documents\HEIG\ARO\HEIG_ARO_LaboPipelinePartie2_BDY_TVE\Rapport%20ARO%20Partie%202.docx#_Toc105265854)

[Figure 7 Diagramme conceptuel du processeur avec pipeline 7](file:///C:\Users\ROG\Documents\HEIG\ARO\HEIG_ARO_LaboPipelinePartie2_BDY_TVE\Rapport%20ARO%20Partie%202.docx#_Toc105265855)

[Figure 8 Aléa de données (1) 8](file:///C:\Users\ROG\Documents\HEIG\ARO\HEIG_ARO_LaboPipelinePartie2_BDY_TVE\Rapport%20ARO%20Partie%202.docx#_Toc105265856)

[Figure 9 Résolution d'aléas de données software 8](file:///C:\Users\ROG\Documents\HEIG\ARO\HEIG_ARO_LaboPipelinePartie2_BDY_TVE\Rapport%20ARO%20Partie%202.docx#_Toc105265857)

[Figure 10 Résolution d'aléas de données hardware 8](#_Toc105265858)

[Figure 11 Exemple d'un aléa de contrôle 9](#_Toc105265859)

[Figure 12 Programme à analyser Test du processeur 10](file:///C:\Users\ROG\Documents\HEIG\ARO\HEIG_ARO_LaboPipelinePartie2_BDY_TVE\Rapport%20ARO%20Partie%202.docx#_Toc105265860)

[Figure 13 Table des aléas Test du processeur 10](file:///C:\Users\ROG\Documents\HEIG\ARO\HEIG_ARO_LaboPipelinePartie2_BDY_TVE\Rapport%20ARO%20Partie%202.docx#_Toc105265861)

[Figure 14: Chronogramme Test du processeur 11](file:///C:\Users\ROG\Documents\HEIG\ARO\HEIG_ARO_LaboPipelinePartie2_BDY_TVE\Rapport%20ARO%20Partie%202.docx#_Toc105265862)

[Figure 15 Programme et table des aléas dépendances de données et de contrôle 12](file:///C:\Users\ROG\Documents\HEIG\ARO\HEIG_ARO_LaboPipelinePartie2_BDY_TVE\Rapport%20ARO%20Partie%202.docx#_Toc105265863)

[Figure 16 Chronogramme aléas de données 13](file:///C:\Users\ROG\Documents\HEIG\ARO\HEIG_ARO_LaboPipelinePartie2_BDY_TVE\Rapport%20ARO%20Partie%202.docx#_Toc105265864)

[Figure 17 Chronogramme aléas de contrôle 15](file:///C:\Users\ROG\Documents\HEIG\ARO\HEIG_ARO_LaboPipelinePartie2_BDY_TVE\Rapport%20ARO%20Partie%202.docx#_Toc105265865)

[Figure 18 circuit control hazard 16](file:///C:\Users\ROG\Documents\HEIG\ARO\HEIG_ARO_LaboPipelinePartie2_BDY_TVE\Rapport%20ARO%20Partie%202.docx#_Toc105265866)

[Figure 19 Control hazard proposé 16](file:///C:\Users\ROG\Documents\HEIG\ARO\HEIG_ARO_LaboPipelinePartie2_BDY_TVE\Rapport%20ARO%20Partie%202.docx#_Toc105265867)

[Figure 20 Circuit d'arrêt de pipeline lors d'aléas de contrôle 17](file:///C:\Users\ROG\Documents\HEIG\ARO\HEIG_ARO_LaboPipelinePartie2_BDY_TVE\Rapport%20ARO%20Partie%202.docx#_Toc105265868)

[Figure 21Chronogramme de vérification du circuit fourni pour les aléas de contrôle 18](file:///C:\Users\ROG\Documents\HEIG\ARO\HEIG_ARO_LaboPipelinePartie2_BDY_TVE\Rapport%20ARO%20Partie%202.docx#_Toc105265869)

[Figure 22 Partie du circuit data\_hazard (1) 19](file:///C:\Users\ROG\Documents\HEIG\ARO\HEIG_ARO_LaboPipelinePartie2_BDY_TVE\Rapport%20ARO%20Partie%202.docx#_Toc105265870)

[Figure 23 Partie du circuit data\_hazard(2) 20](file:///C:\Users\ROG\Documents\HEIG\ARO\HEIG_ARO_LaboPipelinePartie2_BDY_TVE\Rapport%20ARO%20Partie%202.docx#_Toc105265871)

[Figure 24 Partie du circuit data\_hazard (3) 20](file:///C:\Users\ROG\Documents\HEIG\ARO\HEIG_ARO_LaboPipelinePartie2_BDY_TVE\Rapport%20ARO%20Partie%202.docx#_Toc105265872)

[Figure 25 Partie du circuit data\_hazard (4) 20](file:///C:\Users\ROG\Documents\HEIG\ARO\HEIG_ARO_LaboPipelinePartie2_BDY_TVE\Rapport%20ARO%20Partie%202.docx#_Toc105265873)

[Figure 26 Programme et tableau des aléas pour le test des aléas de données 21](file:///C:\Users\ROG\Documents\HEIG\ARO\HEIG_ARO_LaboPipelinePartie2_BDY_TVE\Rapport%20ARO%20Partie%202.docx#_Toc105265874)

[Figure 27 Chronogramme pour le test des aléas de données 22](file:///C:\Users\ROG\Documents\HEIG\ARO\HEIG_ARO_LaboPipelinePartie2_BDY_TVE\Rapport%20ARO%20Partie%202.docx#_Toc105265875)

[Figure 28 Représentation du principe de forwarding 23](file:///C:\Users\ROG\Documents\HEIG\ARO\HEIG_ARO_LaboPipelinePartie2_BDY_TVE\Rapport%20ARO%20Partie%202.docx#_Toc105265876)

[Figure 29 Forwarding : modifications de data\_hazard(1) 24](file:///C:\Users\ROG\Documents\HEIG\ARO\HEIG_ARO_LaboPipelinePartie2_BDY_TVE\Rapport%20ARO%20Partie%202.docx#_Toc105265877)

[Figure 30 Forwarding : modifications de data\_hazard(2) 25](file:///C:\Users\ROG\Documents\HEIG\ARO\HEIG_ARO_LaboPipelinePartie2_BDY_TVE\Rapport%20ARO%20Partie%202.docx#_Toc105265878)

[Figure 31 : Forwarding : modifications de data\_hazard(4) 25](file:///C:\Users\ROG\Documents\HEIG\ARO\HEIG_ARO_LaboPipelinePartie2_BDY_TVE\Rapport%20ARO%20Partie%202.docx#_Toc105265879)

[Figure 32 Forwarding : modifications de data\_hazard(3) 25](file:///C:\Users\ROG\Documents\HEIG\ARO\HEIG_ARO_LaboPipelinePartie2_BDY_TVE\Rapport%20ARO%20Partie%202.docx#_Toc105265880)

[Figure 33 Forwarding : modifications du bloc execute (1) 26](file:///C:\Users\ROG\Documents\HEIG\ARO\HEIG_ARO_LaboPipelinePartie2_BDY_TVE\Rapport%20ARO%20Partie%202.docx#_Toc105265881)

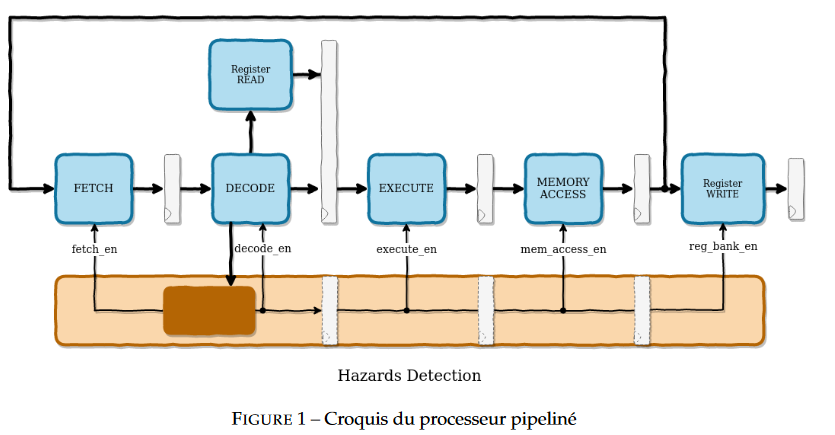
[Figure 34 Forwarding : modifications du bloc execute (2) 26](#_Toc105265882)

[Figure 35 Test du Forwarding : Programme et tableau des aléas 27](file:///C:\Users\ROG\Documents\HEIG\ARO\HEIG_ARO_LaboPipelinePartie2_BDY_TVE\Rapport%20ARO%20Partie%202.docx#_Toc105265883)

[Figure 36 Test forwarding : Chronogramme 28](file:///C:\Users\ROG\Documents\HEIG\ARO\HEIG_ARO_LaboPipelinePartie2_BDY_TVE\Rapport%20ARO%20Partie%202.docx#_Toc105265884)

# Annexes : réponse aux questions

## Analyse du processeur - Donnée

Le processeur qui vous a été fourni a été pipeliné à partir du processeur que vous aviez implémenté dans les laboratoires précédents. Certains changements ont dû être opérés pour pouvoir supporter le pipeline. Pour pipeliner un processeur, il ne suffit pas d’ajouter des registres entre chaque bloc. Il faut, par exemple, s’assurer que tous les signaux de contrôle arrivent au bon moment. Le signal execute\_control\_bus est généré au moment où l’instruction est décodée, mais il est utilisé au moment où l’instruction est exécutée. Il faut donc que le signal de contrôle arrive au même moment que l’instruction dans le bloc execute. Le schéma ci-dessus est un croquis du processeur pipeliné. Les registres sont en gris. Sur le schéma, il y a un grand registre entre les stages du pipeline, or dans Logisim, il y a un registre par signal. Register READ et Register WRITE sont implémentés dans Logisim dans bank\_register.

Les changements qui ont été faits pour le processeur pipeliné :

* 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.

### Réponse aux 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 ?**

On a mis un registre dans la Bank Register il nous faut 2x plus de temps pour sortir donc on doit faire + 1 pour temporiser vu qu’on ne part pas plus 4 parce qu’on part une instruction après

1. **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 ?**

Pour pouvoir temporiser un coup de clock vu qu’on est en parallèle. Cela nous permet de ne pas a attendre de devoir attendre que le registre se met a jour lors de l’etape write back et donc de sauvegarder des temps de cycle

1. **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 ?**

Car adr\_reg\_d est utilisé pour écrire dans un registre. Il faut donc que ce signal passe par les étages suivants du pipeline.

En revanche adr\_reg\_n, adr\_reg\_m et adr\_reg\_mem sont utilisé pour lire des registres dans ce même bloc Decode, donc dans ce même étage du pipeline.

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

On utilise un MUX pour pouvoir temporiser l’arrêt du processeur et pas que cela gêne le bon fonctionnement si on a un saut conditionnel.

1. **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 ?**

Vu qu’on a besoin de 3 étages de pipeline pour la multiplication et que pour le processeur on a besoin de 5 étages, si l’on ajoute le multiplicateur pipeliné on additionne les étages ce qui nous donnerait 8 étages de pipeline.

## Test du processeur – Donnée

Compilez et testez 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 |

Relevez le chronogramme de l’exécution du code ci-dessus depuis le début du traitement de l’instruction **add r4, r0, #2** jusqu’à la fin du traitement de l’instruction **and r1, r3** Vous devez vous inspirer de l’exemple donné en cours. Votre chronogramme doit comprendre les signaux suivants : clock, PC, sortie du registre de chacun des 5 étages du pipeline.

### Réponse aux questions

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

Répondu dans le rapport

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

Répondu dans le rapport

## Assembleur : dépendances de données - Donnée

Dans le programme main.S qui vous a été fourni, indiquez en commentaire pour la première partie (depuis MAIN\_START jusqu’à B PART\_2), les dépendances de données pour chaque instruction. Relevez le chronogramme de l’exécution du code.

Ajoutez le nombre minimum d’instructions NOP pour résoudre les aléas de donnée. Relevez le chronogramme de l’exécution du code.

### Réponse aux questions

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

Répondu dans le rapport

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

Répondu dans le rapport

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

Répondu dans le rapport

## Assembleur : aléas de contrôle - Donnée

Dans la deuxième partie (depuis l’instruction B PART\_2) du programme main.S qui vous a été fourni, ajoutez le nombre minimum d’instructions NOP pour résoudre les aléas de contrôle. Relevez le chronogramme de l’exécution du code.

### Réponse aux questions

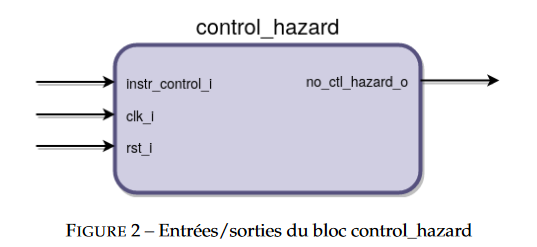
1. **Combien de cycles sont nécessaires pour résoudre un aléa de contrôle ?**

Répondu dans le rapport

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

Répondu dans le rapport

## Circuit control\_hazard - Donnée

Ce circuit permet de détecter si le pipeline doit être bloqué à cause d’un aléa de contrôle. Vous devez compléter le circuit de ce bloc afin de générer un signal no\_ctl\_hazard\_o qui indique qu’il n’y pas d’aléa de contrôle. Lorsqu’une instruction qui génère un aléa de contrôle est détectée (entrée instr\_control\_i), la sortie o\_ctl\_hazard\_o doit être mise à 0 pendant un certain nombre de coups de clock. Puis le signal doit de nouveau être à 1 pendant au minimum 1 coup de clock pour laisser la prochaine instruction être fetch-ée.

Description des différentes entrées/sorties du bloc :

|  |  |
| --- | --- |
| **Nom I/O** | **Description** |
| instr\_control\_i | Indique que l’instruction en cours de décodage est une instruction de contrôle |
| clk\_i | Clock du système |
| rst\_i | Reset asynchrone du système |
| no\_ctl\_hazard\_o | Indique s’il n’y a pas d’aléas de contrôle pour cette instruction |

Comme précédemment, répondez aux questions ci-dessous puis transposez vos réponses sur Logisim.

### Réponse aux questions

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

Répondu dans le rapport

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

Le CPSR est dans le bloc Execute. Le calcul de l’adresse du saut se fait aussi dans le bloc Execute. Donc dans le cas d’un saut, le bloc fetch ne peut pas savoir :

1. Quelle sera l’adresse du saut
2. Est-ce que le saut sera effectué (si saut conditionnel)

Donc le blocage pipeline permet de laisser les coups d’horloge nécessaires à la résolution du saut.

1. **Quels sont les conditions pour qu’un aléa de contrôle ait lieu ?**

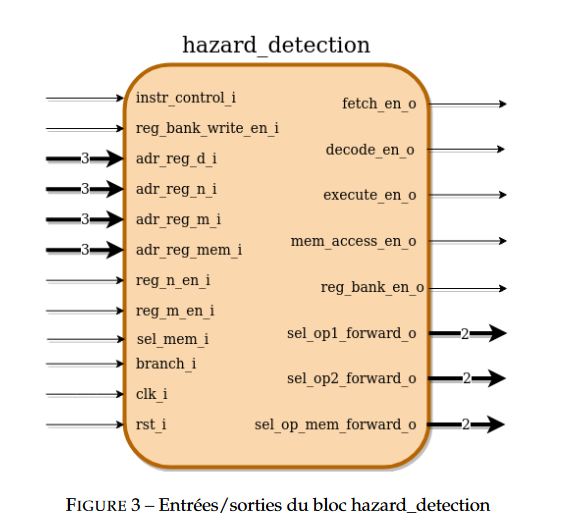
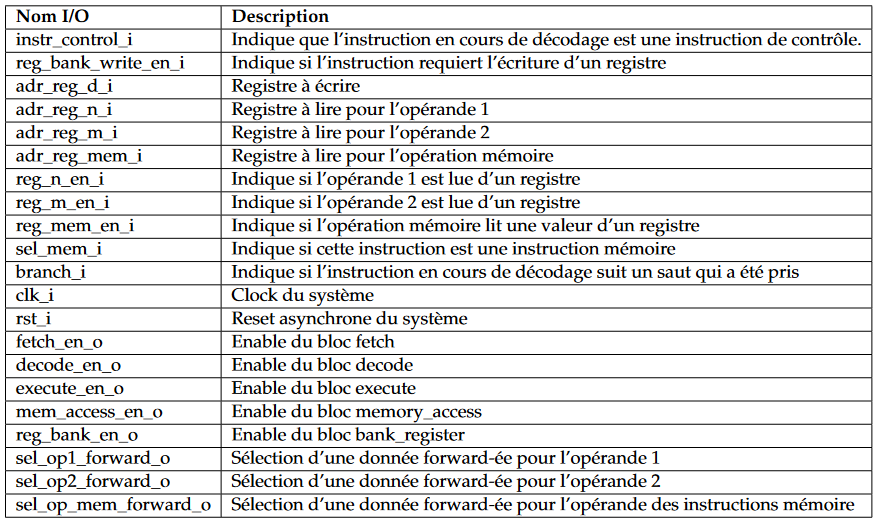
Il faut qu’il y ait un saut qui soit pris et des instructions à la suite du saut.

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

Comme nous devons avoir les bonnes valeurs dans les registres pour calculer l’adresse, l’aléa de donnée doit être résolu en premier. Ensuite, on peut résoudre l’aléa de contrôle.

## Circuit hazard\_detection - Donnée

Ce circuit est instancié dans main\_control\_unit qui est lui dans le bloc decode. La plupart des connections de ce bloc sont déjà effectuées. Dans le circuit main\_control\_unit vous devez ajouter les connections pour le signal instr\_control\_s. Ce signal indique qu’une instruction va générer un aléa de contrôle. Les signaux du bloc hazard\_detection sont décrits ici.

Description des différents signaux du bloc :

### Réponse aux questions

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

B, BEQ, BL sont les instructions qui génèrent des aléas.

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

Les aléas de contrôle sont résolus quand il n’y a pas des aléas de données.

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

S’il y a un saut la prochaine instruction est fetch jusqu’à ce que le pc a été mis a jour puis les instructions fetch jusque-là sont abandonnées.

1. **Pourquoi branch\_i est dans les entrées du circuit hazard\_detection ?**

Pour savoir si on doit sauter ou non est comme ça on sait si on doit avoir un saut.

1. **Pourquoi instr\_control\_i du bloc control\_hazard dépend de no\_data\_hazard\_s ?**

Comme on l’a vu au point 2, on évalue les aléas de contrôle s’il n’y a pas eu de d’alea de donnée. Il nous faut donc vérifier qu’il n’y a pas de contrôle de donnée préalable.

## Test aléas de contrôle - donnée

Ecrivez un programme qui contient des aléas de contrôle. Tester votre programme en faisant un chronogramme. Eviter pour le moment l’instruction BL car elle génère un aléa de donnée et un aléa de contrôle.

### Réponse aux questions

1. **Est-ce que les valeurs dans les registres sont mises à jour correctement et au bon moment ?**

Répondu dans le rapport

1. **Quel est l’IPC pour votre programme ?**

Répondu dans le rapport

1. **Pourquoi l’instruction BL génère en même temps un aléa de contrôle et un aléa de donnée ?**

L’instruction BL est composée de 2 instructions : bl\_msb et bl\_lsb. bl\_msb calcule une partie de l’adresse du saut et l’écrit dans le LR, puis bl\_lsb lit le LR et calcule d’adresse du saut complète. Il y a donc un aléa de données, car il faut que bl\_msb ait fini d’écrire avant que bl\_lsb vienne lire dans le LR Il y a aussi un aléa de contrôle, car il faut attendre que la première partie de l’adresse du saut soit calculée via le bloc execute.

## Circuit data\_hazard

### Questions

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 ?

Il suffit de mémoriser pour les 3 derniers cycles l’adresse du registre de destination. Si pour l’instruction en cours, l’adresse des registres n, m ou mem en lecture est le même que l’un des registres de destination mémorisé, on sait qu’il y aura un aléa de type RAW.

1. Est-ce que ça pose un problème si une instruction dépend du résultat d’une instruction qui est au stage WRITE\_BACK ?

Techniquement, sur Logisim ça fonctionne, le WRITE BACK et la lecture du registre sont faites de manière instantanée, mais en réalité, il faudrait ajouter un coup de clock, à cause du délai de propagation.

1. Quelles informations doivent être mémorisées pour chaque instruction ?
2. L’adresse du registre de destination
3. L’information de si pour cette instruction on veut écrire dans la banque de registre
4. L’information de si pour cette instruction on veut écrire dans la mémoire de données
5. Quelles informations permettent de savoir si le registre D est utilisé ?

Il suffit de savoir si une écriture est demandée dans la banque de registres, donc on doit connaître le signal bank\_wr\_s qui vient du reg\_bank\_cotrol\_unit dans le main\_ctrl\_unit dans le bloc decode.

## Commande des signaux dans main\_control\_unit

### Questions

1. Quelles informations permettent de savoir si le registre N, M ou mem sont utilisés ?
   1. Pour N : Si le bloc Execute a sélectionné l’opérande 1, on sait que la lecture se fait sur le registre N
   2. De même, si le bus de sélection de l’opérande 2 est à 0, on sait qu’on va lire le registre M
   3. Finalement, si on sait que l’instruction est une instruction de type STR, STRH ou STRB (signal str\_data du bloc memory\_access\_control\_unit dans le bloc decode), on sait que le registre MEM sera lu

### Questions Commande des signaux dans hazard\_detection

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

Il suffit de savoir si une écriture est demandée dans la banque de registres, donc on doit connaître le signal bank\_wr\_s qui vient du reg\_bank\_cotrol\_unit dans le main\_ctrl\_unit dans le bloc decode.

1. Une détection d’aléa de donnée va influencer quel(s) enable(s) ? A quel moment ? Pourquoi ?

Lors de la détection d’un aléa de donnée, et si aucun aléa de contrôle n’est détecté, chaque enable des blocs sera désactivé en cascade : DECODE => EXECUTE => M\_ACCESS => WRITE\_BACK (REG\_BANK). La désactivation en cascade permet de terminer de traiter l’instruction en cours.

### Questions

1. Est-ce que les valeurs dans les registres sont mises à jour correctement et au bon moment ?

Répondu dans le rapport

1. Pourquoi l’instruction BL génère un aléa de contrôle et un aléa de donnée ?

L’instruction BL est composée de 2 instructions : bl\_msb et bl\_lsb.

bl\_msb calcule une partie de l’adresse du saut et l’écrit dans le LR, puis bl\_lsb lit le LR et calcule d’adresse du saut complète.

Il y a donc un aléa de données, car il faut que bl\_msb ait fini d’écrire avant que bl\_lsb vienne lire dans le LR

Il y a aussi un aléa de contrôle, car il faut attendre que la première partie de l’adresse du saut soit calculée via le bloc execute.

1. Combien de cycles sont nécessaires pour résoudre les aléas de l’instruction BL ?

Répondu dans le rapport

1. Quel est l’IPC pour votre programme ?

Répondu dans le rapport

## Circuit data\_hazard

1. A quoi sert le signal sel\_mem\_i ?

Sel\_mem\_i nous informe qu’une instruction d’accès à la mémoire de données est décodée. Typiquement les instructions LDRH et STRH

1. 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 ?

Si on veut garder le fonctionnement du forwarding, il faudrait mémoriser les données dans le WRITE\_BACK (dans notre cas Reg Bank). Il faudrait repasser la donnée mémorisée dans le RegBank au bloc execute en cas d’aléa de type RAW, pour que l’execute calcule le résultat de l’instruction précédente.

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

Pour que le forwarding ait lieu, il faut que le résultat écrit dans un registre de la banque de données soit disponible immédiatement à l’étage de l’execute lors de l’instruction suivante. Pour que le Forwarding soit utile, il faut que le programme à exécuter comporte des aléas de type RAW. Dans ce cas, nous pouvons gagner des coups de clock.

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

En cas d’aléas de données, sauf dans le cas d’une instruction de type LDR, le forwarding permet de ne pas arrêter les blocks du pipeline, car le bloc execute a accès directement aux données nécessaires pour faire le calcul de la prochaine instruction. On économise.

Il n’y a pas de conséquences pour la gestion des aléas de contrôle, car de toutes manière, le calcul du saut doit se faire dans le bloc execute. Nous perdons donc 2 cycles avec ou sans forwarding.

## Circuit Execute

1. Pourquoi doit-on faire ça ?

Pour la gestion des aléas de données, il faut prendre la valeur du dernier résultat de l’execute pour pouvoir effectuer l’instruction courante

1. Pourquoi doit-on faire ça pour le signal reg\_mem\_data\_s ?

On doit stocker l’adresse de la mémoire de donnée pour les instructions LRDH et STRH pour aller la chercher au cycle d’après dans le cas d’un aléa de donnée.

1. Que devrait-on faire si on avait un data forwarding venant du WRITE\_BACK ?

Il faudrait ajouter un registre supplémentaire dans le WRITE\_BACK permettant de fournir la valeur calculée de l’instruction précédent au bloc execute

## Test : pipeline forwarding

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 ?

Répondu dans le rapport

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

Répondu dans le rapport

1. Combien de cycles sont nécessaires pour que l’instruction BL soit complétée ?

Répondu dans le rapport

1. Avez-vous d’autres idées d’optimisation de ce processeur ?

Résoudre les problèmes que nous avons relevé (se référer à la conclusion de notre rapport).