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 : **vendredi, 3 juin 2022**

Table des matières

[1 Introduction 3](#_Toc102512870)

[1.1 Méthodologie 3](#_Toc102512871)

[2 Analyse et conception 4](#_Toc102512872)

[2.1 Processeur 4](#_Toc102512873)

[2.2 Architecture de notre processeur 5](#_Toc102512874)

[2.3 Fetch 6](#_Toc102512875)

[2.4 Decode 6](#_Toc102512876)

[2.5 Execute 7](#_Toc102512877)

[2.6 Memory access 7](#_Toc102512878)

[3 Réalisation 8](#_Toc102512879)

[3.1 Fetch 8](#_Toc102512880)

[3.2 Banque de registres 10](#_Toc102512881)

[3.3 Decode 12](#_Toc102512882)

[3.4 Execute 15](#_Toc102512883)

[3.5 Memory access 18](#_Toc102512884)

[3.6 Processeur complet 19](#_Toc102512885)

[4 Simulation 20](#_Toc102512886)

[4.1 Fetch 20](#_Toc102512887)

[4.2 Decode 21](#_Toc102512888)

[4.3 Execute 22](#_Toc102512889)

[5 Validation du circuit global 23](#_Toc102512890)

[5.1 Chronogrammes 24](#_Toc102512891)

[6 Conclusion 25](#_Toc102512892)

[7 Bilan personnel 25](#_Toc102512893)

[8 Table des illustrations 26](#_Toc102512894)

# 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.

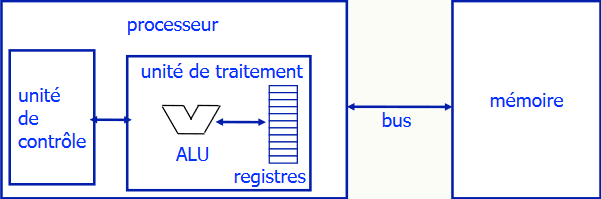


Figure 1 architecture d'un processeur (1)

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.

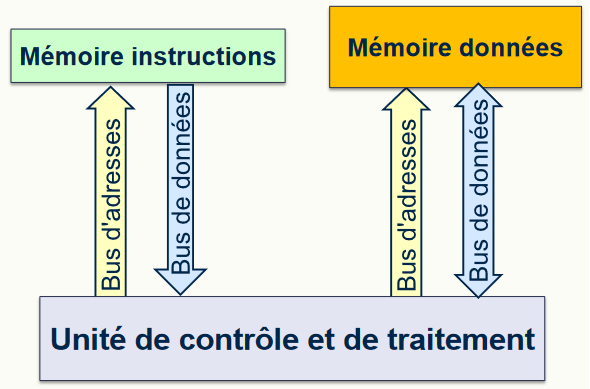
Sur les bus de la mémoire de données sont aussi connectées les entrées/sorties des périphériques potentiellement connectés dans notre système. Cependant, dans un souci de simplicité, pour ce travail pratique, nous n’en tiendront pas compte.

Figure 2 Architecture Harvard

## Architecture de notre processeur

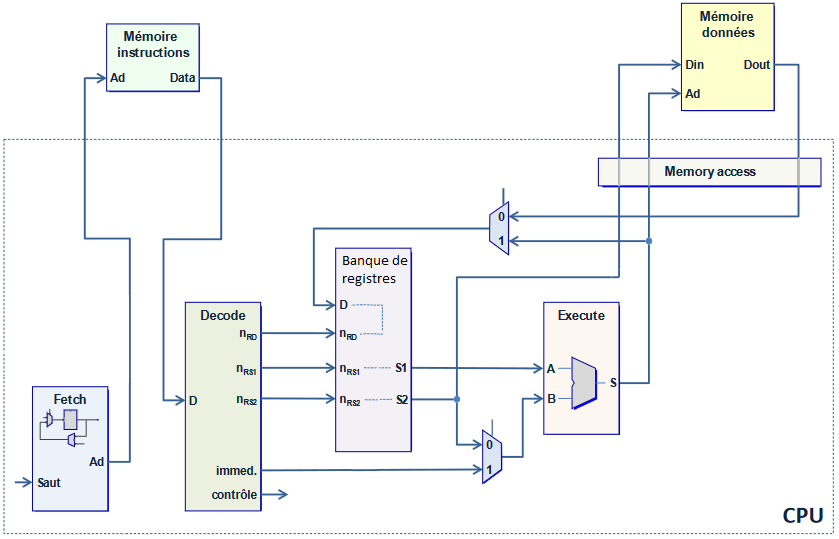


Figure 3 Architecture de notre processeur

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

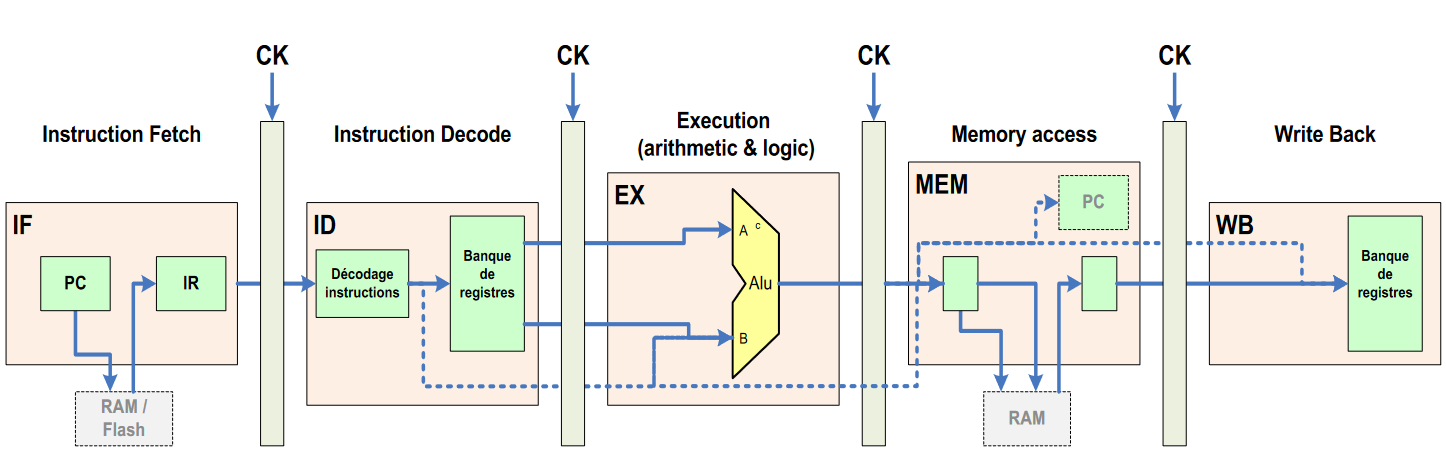
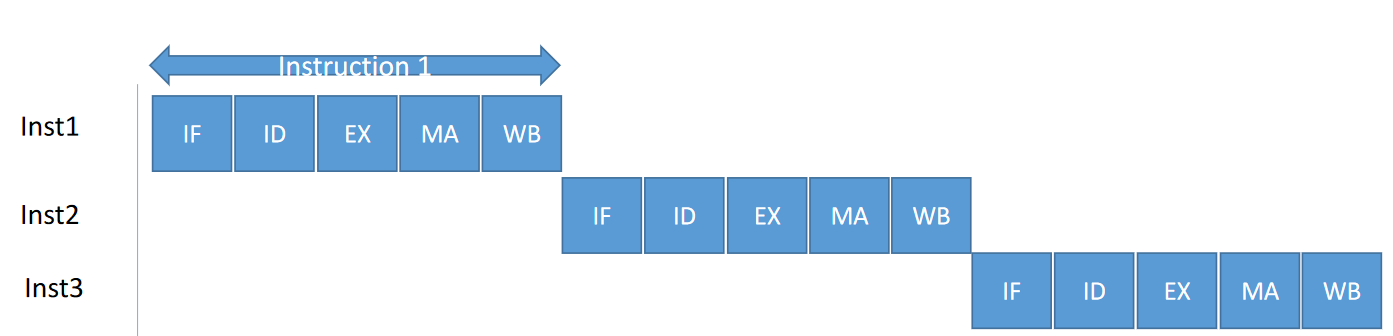
Comme mentionné dans l’introduction, nous devons modifier notre processeur simple pour qu’il fonctionne de manière pipelinée. Le pipeline permet d'accroître la vitesse d'exécution des instructions. Pour cela, le processeur est découpé en 5 étages :  
Fetch, Decode, Execute, Write back et Memory access. 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 Les 5 étages du pipeline

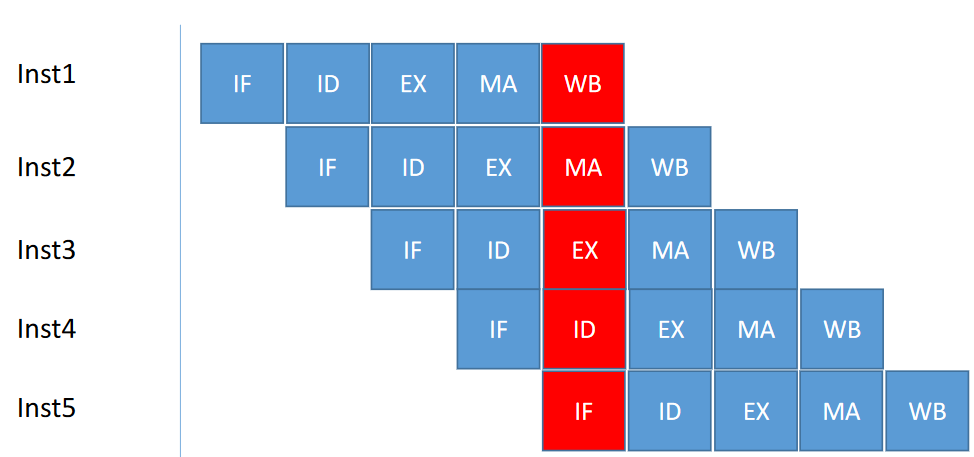


Figure 6 Principe d'exécution en cascade du processeur pipeliné

## 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

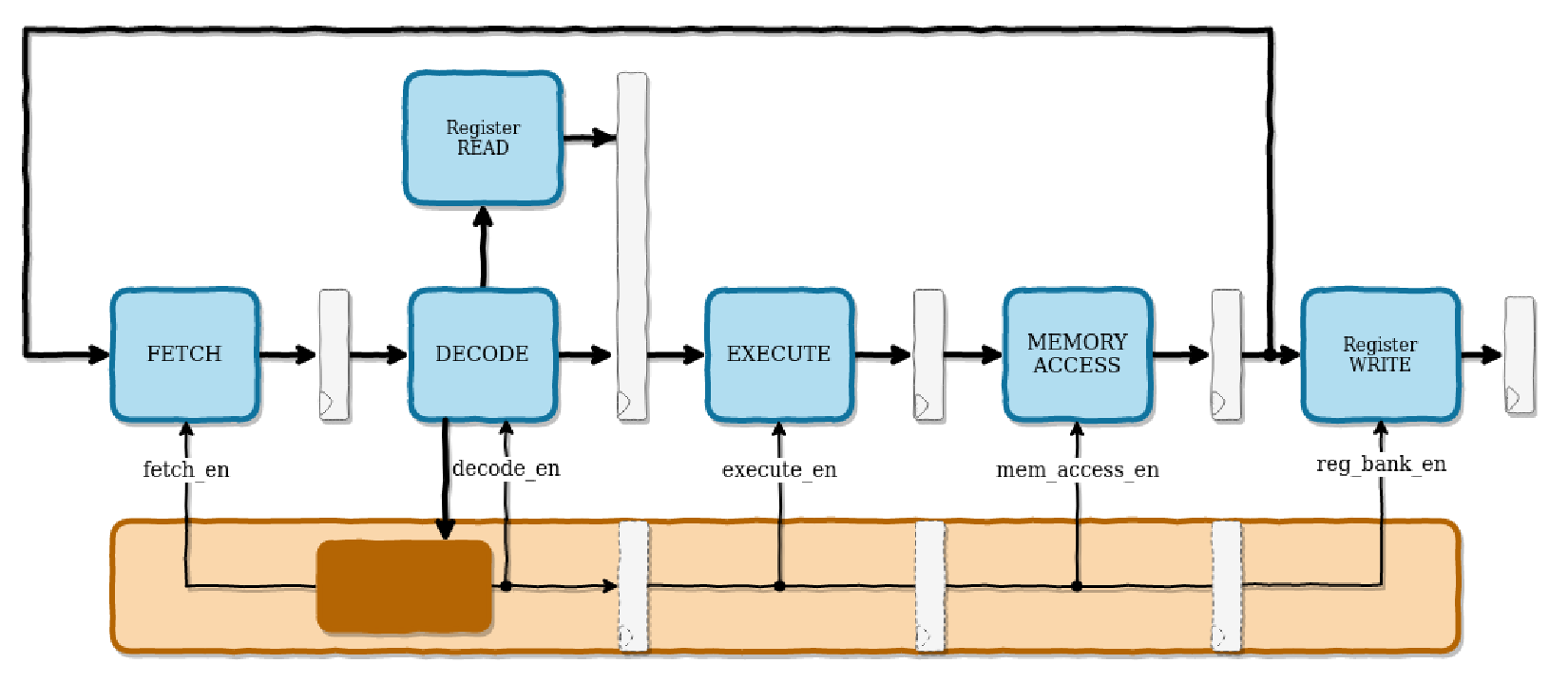
Pour pouvoir implémenter le forwarding, nous devons stocker la valeur de sortie de chaque étage du processeur. C’est grâce à cette mémorisation des états que nous pouvons utiliser le mécanisme d’exécution en cascade.

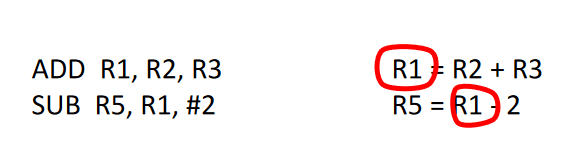
Figure 7 Diagramme conceptuel du processeur avec pipeline

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.

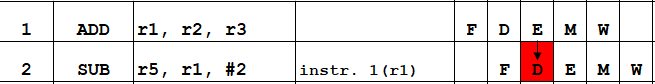
Si aucun mécanisme de détection d’aléa n’est mis en place, l’instruction 1 sera exécutée par l’étage Execute et l’instruction 2 sera exécutée par l’étage Fetch.

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.

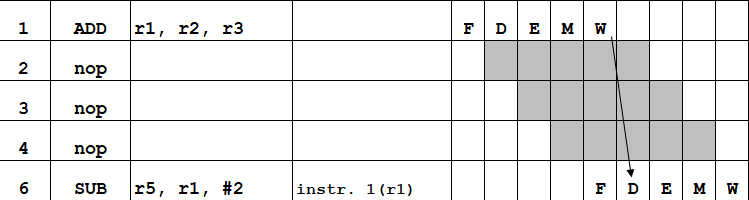
Pour réaliser cette « attente » nous pouvons utiliser 2 méthodes différentes : La résolution **software** et la détection **hardware**. La première méthode consiste simplement à ajouter, dans le code assembleur des instructions « nop » (no operation). Ces instructions ne font rien, mais permettent d’attendre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

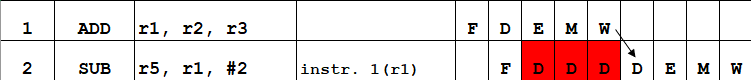


Figure 10 Résolution d'aléas de données hardware

### 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 :

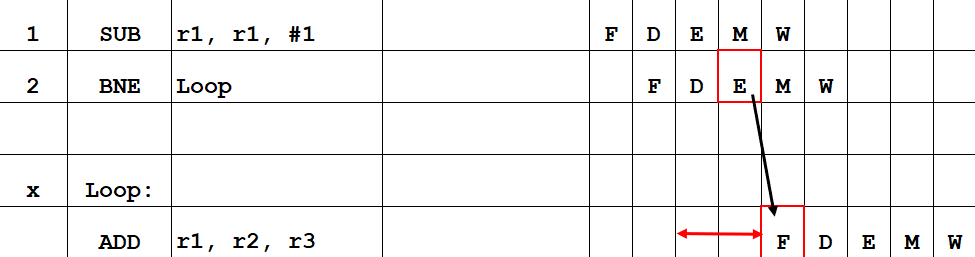


Figure 11 Exemple d'un aléa de contrôle

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.

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.

## Analyse des modifications apportées au processeur

## 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.

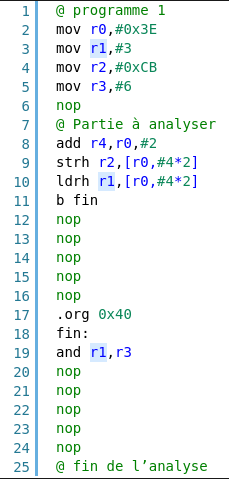
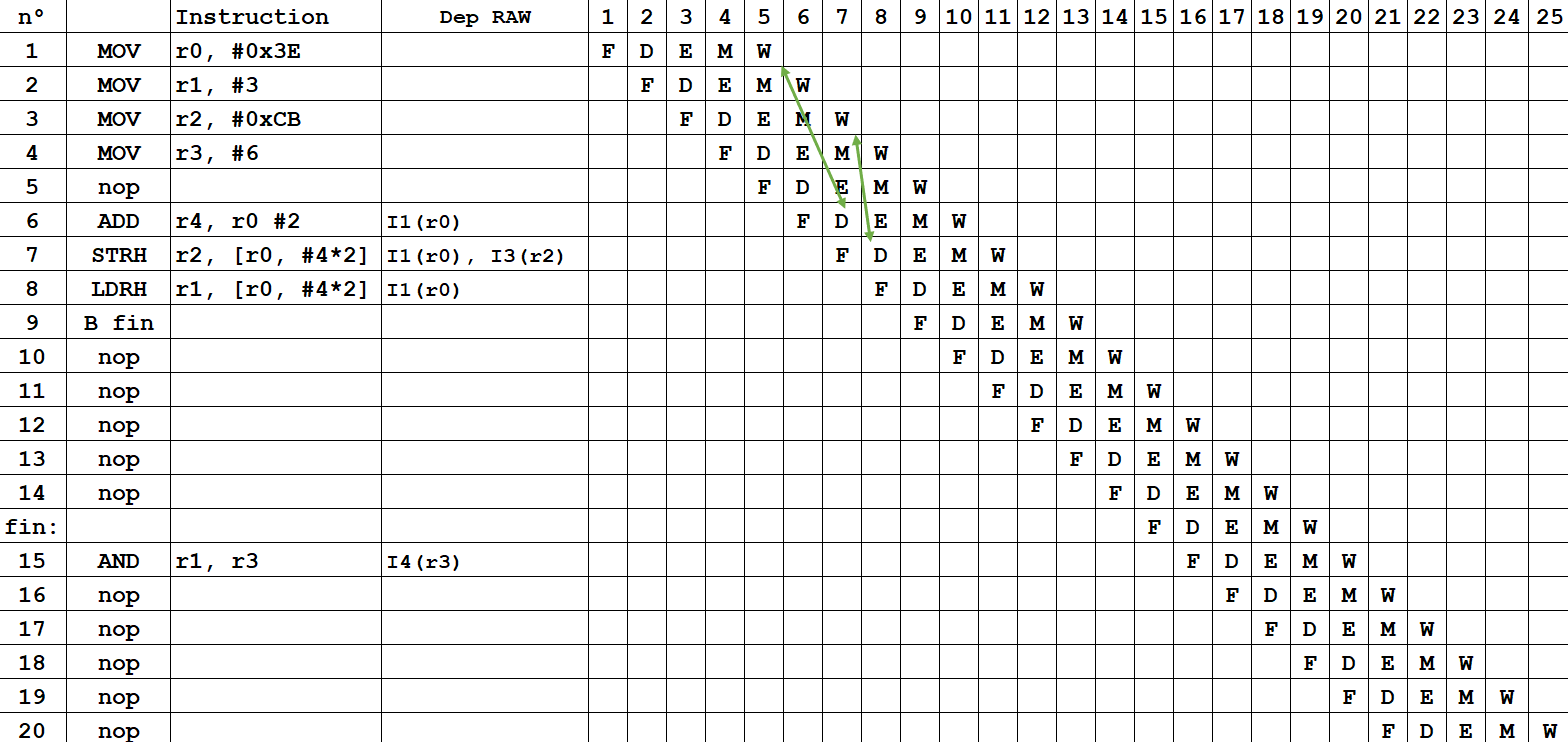
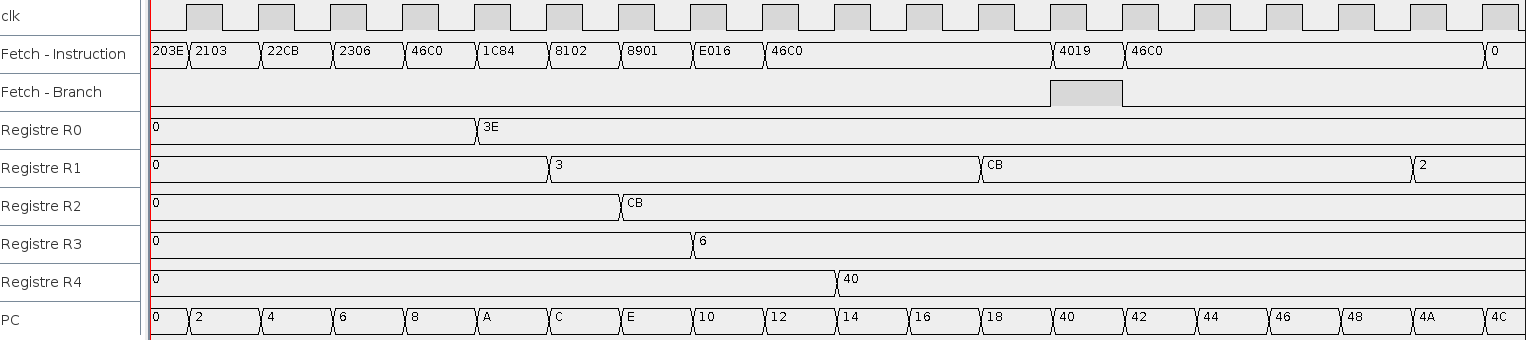


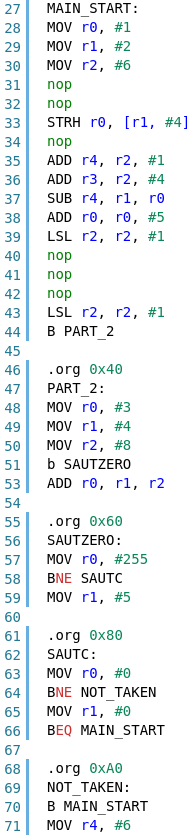
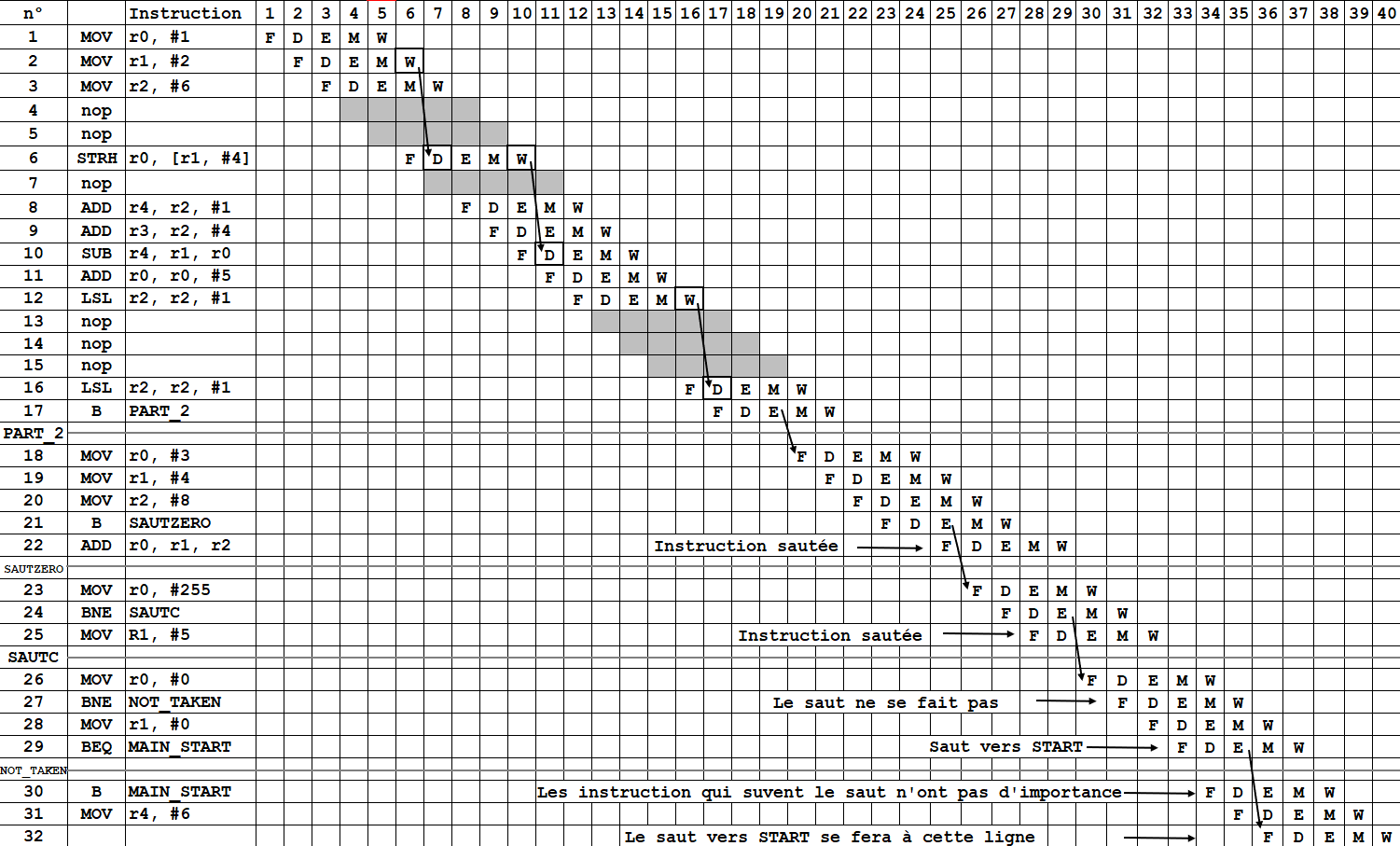
Figure 12 Programme à analyser (1)

Figure 13 Table des aléas (1)

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.
* 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.

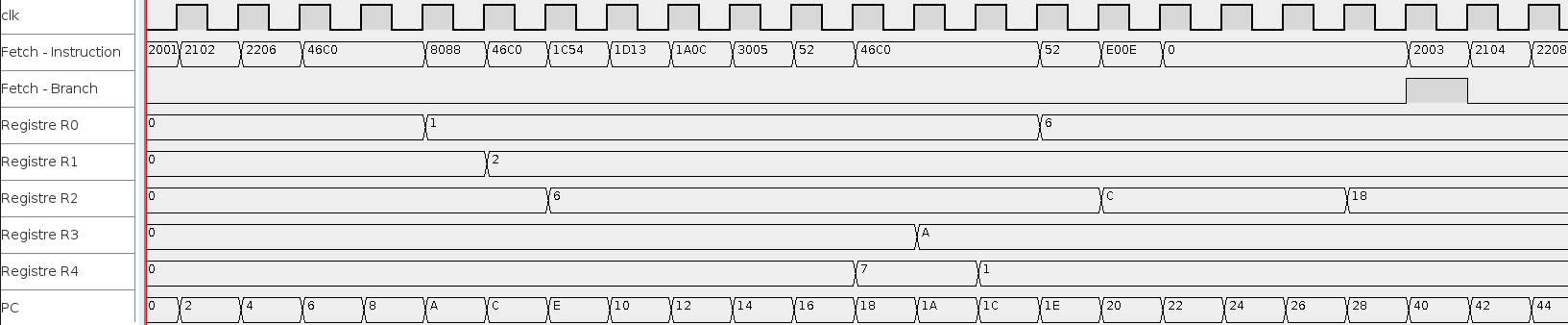
### Assembleur : dépendances de données et Assembleur : aléas de contrôle

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

**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 :**



3x nop

3x nop

B PART\_2

12

14

16

18

20

6

8

10

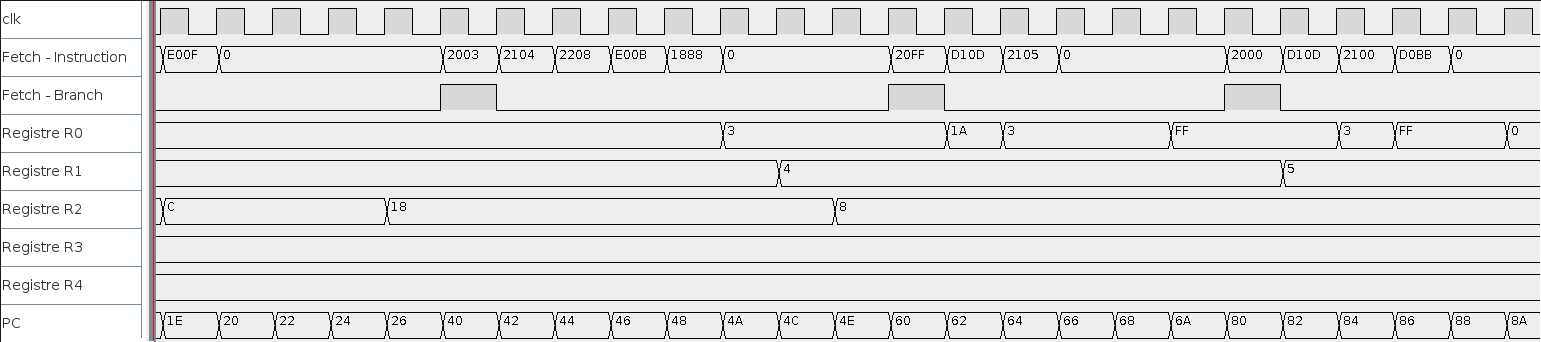
* Nous pouvons voir qu’au 12e coup d’horloge nous avons 0x7 dans R4, ce qui correspond au WB de ADD r4, r2, #1 avec r2 = 6
* Au 13e Nous avons 0xa dans r3, ce qui correspond à ADD r3, r2, #4, avec r2 = 6
* Au 14e nous avons 0x1 and r4 ce qui correspond à SUB r4, r1, r0 avec r1 = 2 et r0 = 1
* Au 15e nous avons 0x6 dans r0, ce qui correspond à ADD r0, r0, #5 avec r0 = 1
* Au 16e nous avons 0xc dans r2 ce qui correspond à LSL r2, r2, #1 avec r2 = 6
* Au 20e nous avons 0x18 dans r2 ce qui correspond à LSL r2, r2, #1 avec r2 = c

Nous pouvons en conclure que notre programme fonctionne

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

* Nous avons remarqué qu’il y avait un aléa de contrôle après le premier saut B PART\_2 (I17). Cependant ce n’est pas grave, car après ce saut il n’y a pas d’autres instructions, l’instruction suivant étant à 0x40 dans la mémoire de données.
* Ensuite, nous voyons que l’instruction B SAUTZERO est suivie d’une instruction ADD. S’agissant d’un saut inconditionnel, il est inutile d’ajouter de nop, car le ADD sera toujours sauté.
* L’instruction 25 BNE SAUTC ne pose non plus pas de problème, car la condition est toujours vraie pour le sut. Nous sauterons donc toujours l’instruction d’après (MOV) SAUTC se trouvant plus loin dans la mémoire d’instruction ce n’est pas grave d’exécuter les 2 instructions suivant ce saut
* Le saut BNE NOT\_TAKEN n’est jamais pris, car sa condition est toujours fausse.
* C’est donc le saut suivant (BEQ MAIN\_START) qui sera pris. Sa condition étant toujours vraie, on sautera toujours vers MAIN\_START, donc inutile de se préoccuper des instructions suivantes.

Au final, en théorie, nous n’avons pas besoin d’ajouter de nop. Entre B\_PART\_2 et BEQ MAIN\_START. Pour en être sûr, nous allons réaliser un chronogramme.



### Assembleur : dépendances de données

### Assembleur : aléas de contrôle

### Circuit control\_hazard

### Circuit hazard\_detection

### Test aléas de contrôle

# Processeur pipeliné avec forwarding

## Analyse

Dans cette partie, nous n’avons pas besoin de réaliser, ni de compléter des circuits, car le projet fonctionnel nous a été fourni. Il nous faudra donc l’analyser afin de comprendre les modifications qui ont été apportées.

## Analyse du circuit fourni

### Aléas de donnée : Circuit data\_hazard

### Aléas de données : Circuit hazard\_detection

### Test : aléas de donnée

### Pipeline Forwarding : Circuit data\_hazard

### Pipeline Forwarding : Circuit execute

### Test : pipeline forwarding

## Tests et validation

# Conclusion

De notre point de vue, toutes les fonctionnalités de ce processeur sont fonctionnelles, notamment la gestion de la pile. Nous n’avons pas eu de grande difficulté technique pour la réalisation de ce laboratoire, hormis la gestion du temps pour la rédaction du présent rapport. Nous sommes conscients que les tests et la validation des blocs auraient être plus approfondis, mais nous ne disposions pas d’assez de temps pour les réaliser.

# Bilan personnel

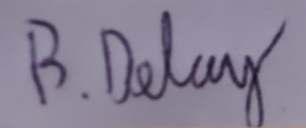
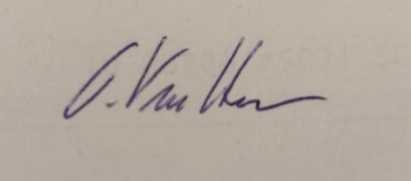
Cette deuxième partie de travail était bien différente de la première. Nous n’avons presque pas dû compléter de circuits, mais plutôt analyser les circuits qui nous ont été fournis. Ce travail d’analyse est moins évident qu’un travail de construction, car il est plus difficile de comprendre le fonctionnement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

* En théorie, la gestion du 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 la banque de registre
* 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 03.05.2022



Benoît Delay Timothée Van Hove

# Table des illustrations

[Figure 1 architecture d'un processeur (1) 4](#_Toc102513472)

[Figure 2 Architecture Harvard 4](file:///C:\Users\ROG\Documents\HEIG\ARO\Labo%20ARO.docx#_Toc102513473)

[Figure 3 Architecture de notre processeur 5](#_Toc102513474)

[Figure 4 Logigramme du bloc Fetch 6](file:///C:\Users\ROG\Documents\HEIG\ARO\Labo%20ARO.docx#_Toc102513475)

[Figure 5 Exemple d'instruction THUMB 7](file:///C:\Users\ROG\Documents\HEIG\ARO\Labo%20ARO.docx#_Toc102513476)

[Figure 6 Notre simulation mini decodeur 8](file:///C:\Users\ROG\Documents\HEIG\ARO\Labo%20ARO.docx#_Toc102513477)

[Figure 7 Program counter 9](file:///C:\Users\ROG\Documents\HEIG\ARO\Labo%20ARO.docx#_Toc102513478)

[Figure 8 Conception de notre bloc fetch 9](file:///C:\Users\ROG\Documents\HEIG\ARO\Labo%20ARO.docx#_Toc102513479)

[Figure 9 Registres de la banque de registres 10](file:///C:\Users\ROG\Documents\HEIG\ARO\Labo%20ARO.docx#_Toc102513480)

[Figure 10 LR manager 10](file:///C:\Users\ROG\Documents\HEIG\ARO\Labo%20ARO.docx#_Toc102513481)

[Figure 11 Décodage des registres de destination 11](file:///C:\Users\ROG\Documents\HEIG\ARO\Labo%20ARO.docx#_Toc102513482)

[Figure 12 Sélection des registres de lecture 11](file:///C:\Users\ROG\Documents\HEIG\ARO\Labo%20ARO.docx#_Toc102513483)

[Figure 13 Visualisation du bloc décode 12](file:///C:\Users\ROG\Documents\HEIG\ARO\Labo%20ARO.docx#_Toc102513484)

[Figure 14 Sélection des registres d, n et m 13](file:///C:\Users\ROG\Documents\HEIG\ARO\Labo%20ARO.docx#_Toc102513485)

[Figure 15 Extrait du fetch control unit 13](file:///C:\Users\ROG\Documents\HEIG\ARO\Labo%20ARO.docx#_Toc102513486)

[Figure 16 Extrait de la table de verité fetch control unit 13](file:///C:\Users\ROG\Documents\HEIG\ARO\Labo%20ARO.docx#_Toc102513487)

[Figure 17 Extrait du bloc execute control unit 14](file:///C:\Users\ROG\Documents\HEIG\ARO\Labo%20ARO.docx#_Toc102513488)

[Figure 18 Extrait de la table de vérité execute control unit 14](file:///C:\Users\ROG\Documents\HEIG\ARO\Labo%20ARO.docx#_Toc102513489)

[Figure 19 Bloc shift 15](file:///C:\Users\ROG\Documents\HEIG\ARO\Labo%20ARO.docx#_Toc102513490)

[Figure 20 Bloc Alu 16](file:///C:\Users\ROG\Documents\HEIG\ARO\Labo%20ARO.docx#_Toc102513491)

[Figure 21 Bloc Decode 17](file:///C:\Users\ROG\Documents\HEIG\ARO\Labo%20ARO.docx#_Toc102513492)

[Figure 22 Bloc mémory access 18](file:///C:\Users\ROG\Documents\HEIG\ARO\Labo%20ARO.docx#_Toc102513493)

[Figure 23 Bloc processeur complet 19](file:///C:\Users\ROG\Documents\HEIG\ARO\Labo%20ARO.docx#_Toc102513494)

[Figure 24 Chronogramme fetch 20](file:///C:\Users\ROG\Documents\HEIG\ARO\Labo%20ARO.docx#_Toc102513495)

[Figure 25 programme fetch 20](file:///C:\Users\ROG\Documents\HEIG\ARO\Labo%20ARO.docx#_Toc102513496)

[Figure 26 Programme decode 21](file:///C:\Users\ROG\Documents\HEIG\ARO\Labo%20ARO.docx#_Toc102513497)

[Figure 27 chronogramme decode 21](file:///C:\Users\ROG\Documents\HEIG\ARO\Labo%20ARO.docx#_Toc102513498)

[Figure 28 Programme et chronogramme execute 22](file:///C:\Users\ROG\Documents\HEIG\ARO\Labo%20ARO.docx#_Toc102513499)

[Figure 29 Programme de vérification final 23](file:///C:\Users\ROG\Documents\HEIG\ARO\Labo%20ARO.docx#_Toc102513500)

[Figure 30 Chronogramme du programme de vérification final 24](file:///C:\Users\ROG\Documents\HEIG\ARO\Labo%20ARO.docx#_Toc102513501)