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

Reconfigurable & embedded
Digital Systems

Rapport de rendu

Départements : TIC

Unité d'enseignement ARO

Auteurs : Alexis Martins et Arnaud Monition

Professeur : Marina Zapater
Assistant : Mike Meury

Classe : C
Salle de laboratoire : A09

Date : 06.06.2022

1 Introduction

Ce laboratoire a pour but la réalisation d'un processeur et des différentes parties le composant. Nous allons réaliser ce laboratoire en parallèle du cours d'ARO (Architecture des ordinateurs) durant lequel on aura l'occasion d'acquérir les connaissances requises pour mener à terme ce projet.

L'organisation du processeur a été découpée au préalable en différentes parties clés pour que l'on puisse progresser et comprendre l'intérêt de chaque partie pour le fonctionnement final.

On retrouve dans les parties simulant un processeur, le Fetch, le Decode, l'Execute et le Memory Access. Ces premiers blocs seront réalisés dans la première partie de ce laboratoire. La seconde partie du laboratoire nous permettra de réaliser la partie Pipeline, mémoire cache et mémoire virtuelle.

Ces derniers points sont pour la partie « matérielle », on devra cependant aussi parfois réaliser certains programmes que ça soit pour la validation des blocs effectués ou simplement pour la compréhension de ceux-ci. On retrouvera le détail et les explications de ceux-ci dans le rapport.

2 Analyse et conception

Cette partie du rapport nous permet de définir tout le travail préalable ou secondaire à la réalisation directe du processeur. On y détaille les différentes parties, ainsi que leur rôle au sein du composant total.

On y retrouve aussi une partie traitant de l'organisation de ce rapport, ainsi que des contraintes rencontrées durant ce projet.

2.1 Parties à réaliser

Le composant final à obtenir est un processeur, celui-ci est composé de diverses parties que l'on va réaliser indépendamment pour finalement les assembler au fur et à mesure. Dans un premier temps, on réalisera trois composants principaux du processeur (Fetch, Decode et Execute), puis on réalisera la partie Pipeline et mémoires (cache et virtuelle).

Fetch s'occupera de charger une instruction dans le processeur à partir de la mémoire d'instruction. Il permettra notamment de mettre à jour le PC et de faire le choix entre un saut ou une instruction classique.

Decode permettra de décoder les instructions reçues par la mémoire d'instructions. Ce qui permettra notamment de savoir quelles parties de l'instruction servent à quoi et les envoyer au bon endroit pour pouvoir par la suite effectuer des opérations avec le bloc Execute. On mettra en place pendant cette partie le bloc contenant la banque de registres. Ce bloc permet de stocker l'information transitant dans le processeur, ainsi que les registres spéciaux tels que le Program Counter, le Link Register ou le Stack Pointer.

Execute permet de réaliser des opérations arithmétiques, logiques et plus généralement d'effectuer tous les calculs au sein du processeur. Durant cette partie, on mettra aussi en place le bloc de Memory Access qui permet d'accéder à la mémoire de donnée.

2.2 Schémas

On peut retrouver ci-dessous les différentes parties citées dans la catégorie précédente. On peut observer l'interaction qu'à chaque bloc avec les autres et de quelle manière ceux-ci seront interconnectés les uns aux autres.

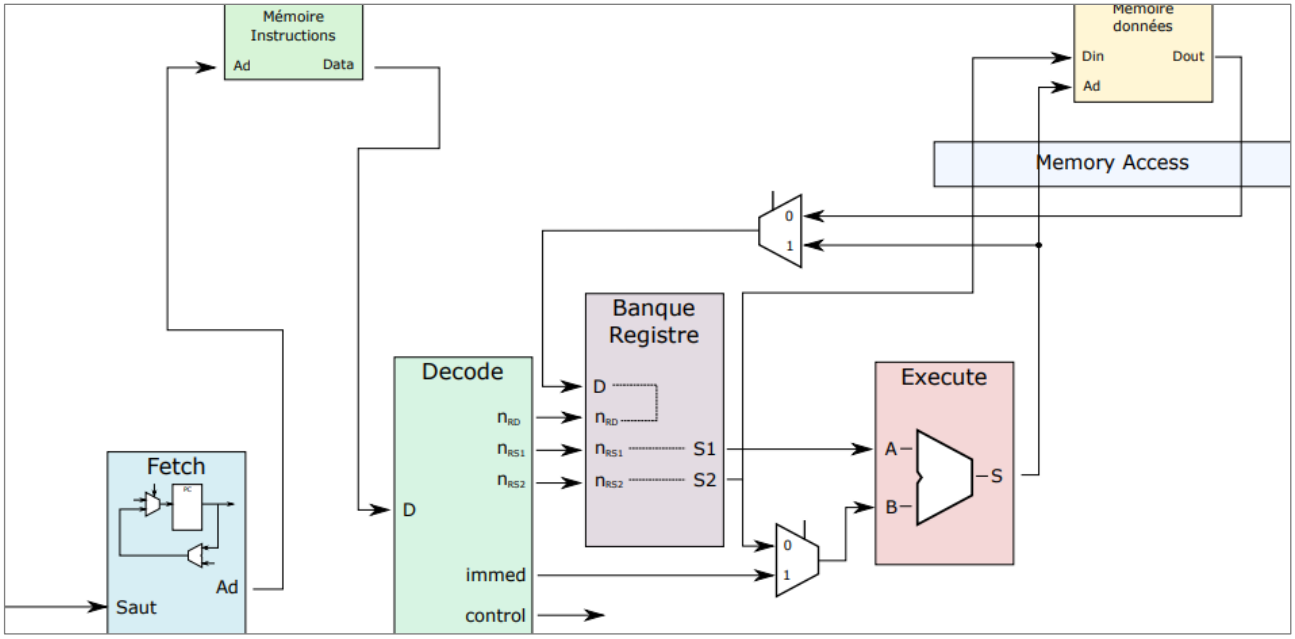

Figure 1 : Schéma processeur

2.3 Contraintes

2.3.1 Utilisation de blocs natifs

Par blocs natifs, on entend tous les blocs qui sont disponibles au sein de Logisim par défaut. Il y a une contrainte qui est de ne pas les utiliser lorsque les entrées ont une structure fixe, par contre on peut totalement les utiliser lorsque les entrées varient. Ceci a notamment posé problème lorsque l'on a dû faire le calcul de l'adresse de saut et utiliser des systèmes contenant des splitters au lieu d'un bloc multiplicateur.

2.4 Organisation du rapport

Nous avons décidé d'organiser le rapport par catégories générales (introduction, réalisation, tests, etc.) et à l'intérieur de celles-ci de détailler la manière dont chaque bloc est réalisé, testé, etc.

3 Réalisation

3.1 Fetch

3.1.1 Introduction

Le « Fetch » traduit de l'anglais « rapporter » va nous permettre de faire le lien entre la mémoire d'instruction et notre CPU. C'est lui qui va aller chercher et charger les opérations que notre processeur va exécuter par la suite.

3.1.2 Implémenter un registre représentant le PC

Nous allons commencer par instancier un registre directement dans le circuit processeur_ARO2. Comme nous sommes qu'au début du laboratoire, nous devons commencer par mettre un registre « simple ». Le PC sera implémenté plus tard comme registre de la banque de registres.

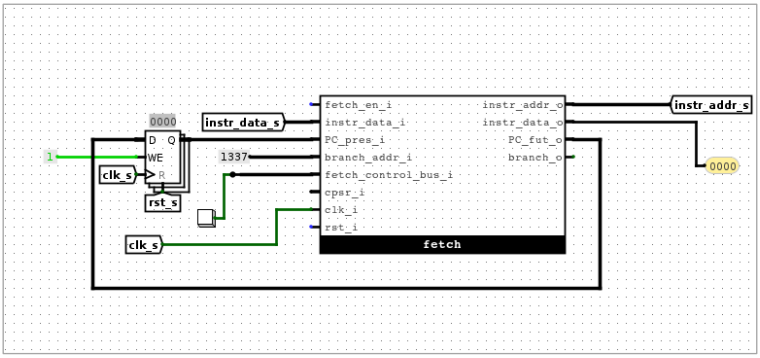

Figure 2 : Implémentation du PC dans le bloc Fetch

3.1.3 Implémenter l'incrémentation du PC dans le bloc FETCH

Au sein du bloc Fetch, on peut créer une partie de circuit permettant d'incrémenter le PC. Il faudra effectuer un incrément de 2 bytes étant donné que nous sommes sur une architecture 16 bits. Il faut donc pour l'instant simplement utiliser un additionneur reliant l'entrée et la sortie du PC.

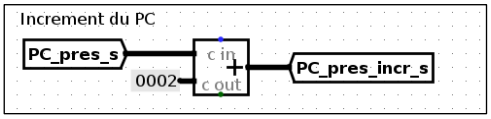

Figure 3 : Incrément du PC

3.1.4 Implémenter les entrées/sorties de la mémoire d'instructions

Ici, nous allons câbler les signaux qui sont connectés à la mémoire d'instructions. Les données sont simplement un câble entre l'entrée et la sortie, tandis que l'adresse est représentée par la valeur courante du PC.

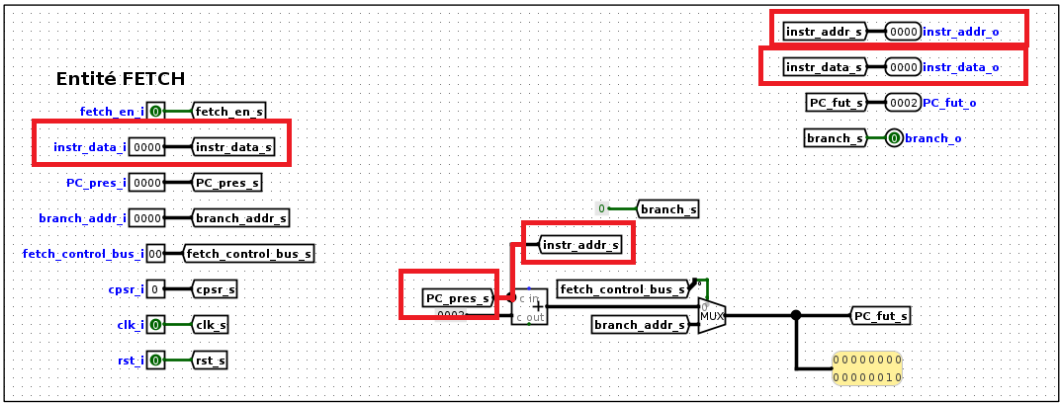

Figure 4 : Entrées/sorties bloc Fetch

3.1.5 Ajout d'un mécanisme de saut dans le bloc Fetch

Pour le mécanisme de saut au sein du bloc Fetch, le système est assez simple. Il faut que la sortie du bloc condition_tester soit activée, ce bloc permet de tester les conditions pour les sauts conditionnels et dans le cas d'un saut inconditionnel elle est activée par défaut. Il faut aussi que le bit 0 du fetch_control_bus soit à 1 pour indiquer que l'instruction courante est un saut.

Pour la partie avec le bloc de test, celui-ci prend en entrée le CPSR qui sera mis en place plus tard, le bit 1 du fetch_control_bus qui permet de déterminer si l'instruction courante est un saut ou non et la condition à tester qui sont les bits 2 à 5 du bus.

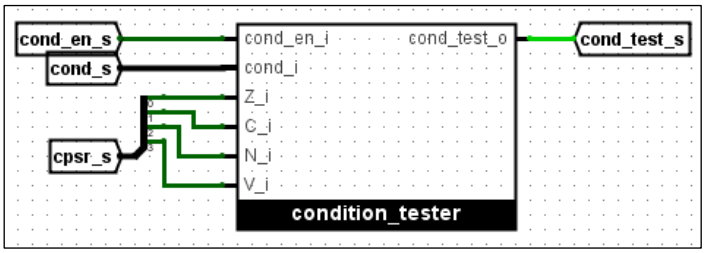

Figure 5 : Bloc condition_tester

Il suffit ensuite de relier ces deux sorties à une porte ET dont la sortie sera l'indicateur pour savoir si l'on incrémente ou si l'on saute. Le second MUX viendra dans la suite du rapport.

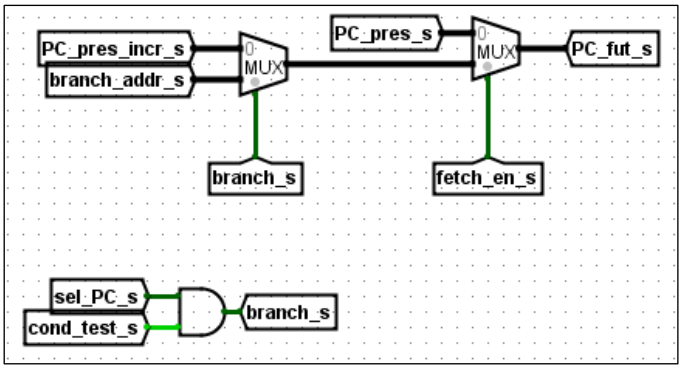

Figure 6 : Gestion du saut

3.1.6 Détection de l'instruction de saut

Cette partie sera dans le futur au sein du circuit decode. Pour l'instant, nous allons réaliser un circuit temporaire, mais suffisant pour la détection du saut. On va donc devoir construire le fetch_control_bus que l'on utilise dans le bloc fetch pour la détection d'un saut. Ce bus de contrôle est organisé de la manière suivante.

Position	Taille	Description
0	1	Flag qui indiquera que l'instruction courant est un saut (conditionnel ou non)
1	1	Flag qui indiquera que l'instruction courante est conditionnelle
5-2	4	Permet de sélectionner la condition que vous voulez tester
6	1	Pas utilisé dans ce laboratoire

Figure 7 : Bus de contrôle du bloc Fetch

Pour le premier bit, on va vérifier que le début de l'instruction corresponde à celle indiquée dans notre manuel. Pour un saut conditionnel, les 4 bits de poids fort sont équivalents à « D » et pour les sauts inconditionnels, les 5 bits de poids fort sont équivalents à « 1C ».

Pour le second bit, on a remarqué que ce qui permettait de différencier les deux types de saut c'est les bits 12 et 13. On a donc décidé que le 12ème bit allait définir le type de saut.

1	1	0	1	Cond		Soffset8	*Conditional branch*
1	1	1	0	0	Offset11		*Unconditional branch*

Figure 8 : Instructions de saut

Les 4 bits permettant de sélectionner la condition à tester sont les bits 8 à 11 de l'instruction. Il a donc suffi de récolter ces bits et de directement les transférer dans le bus. Le dernier bit quant à lui est inutilisé donc nécessite une constante à 0. Le résultat final se situe juste en dessous et ce petit bloc se trouve dans le processeur_ARO2 directement.

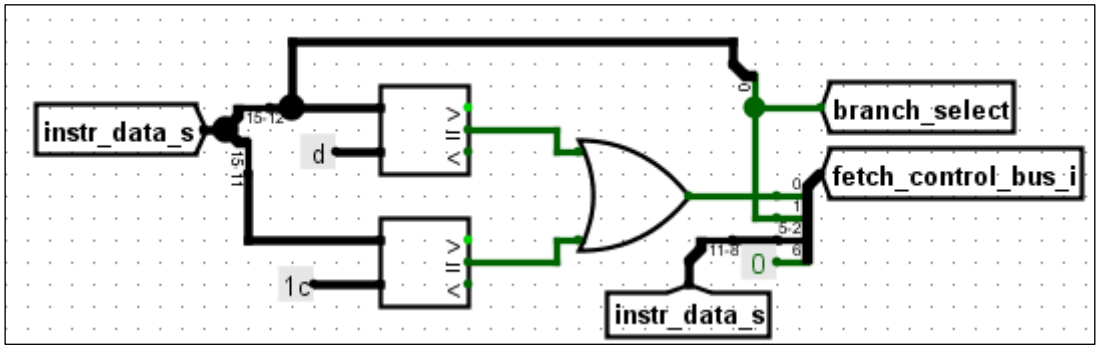

Figure 9 : Construction du bus de contrôle Fetch

3.1.7 Calcul de l'adresse de saut

Cette partie sera aussi plus tard dans un autre bloc du processeur. Mais pour l'instant, on se contente d'appliquer les deux formules vues en cours pour transformer les offsets des instructions en adresses de sautes utilisables.

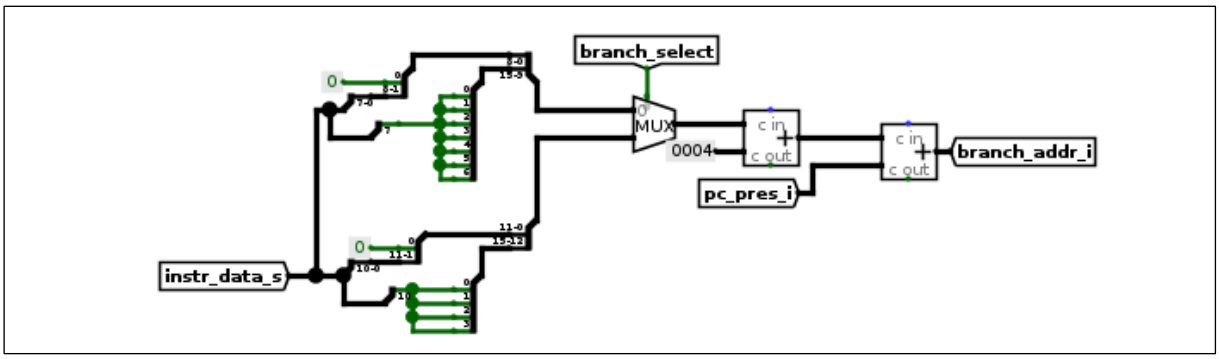

Figure 10 : Calcul de l'adresse de saut

3.1.8 Implémentation de l'enable du fetch

Plus tôt dans le rapport, on a aperçu l'enable du fetch lorsque l'on choisissait de laisser le saut se faire ou non. Le système pour l'enable est assez simple, s'il est activé on passe à l'adresse suivante (saut ou non), sinon on conserve l'adresse courante du PC.

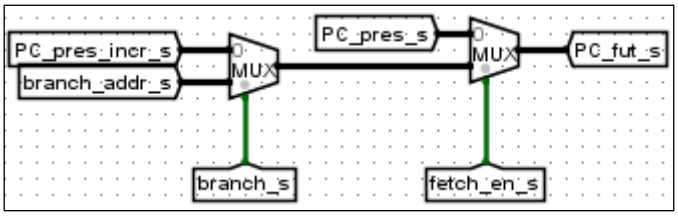

Figure 11 : Enable du Fetch

La partie du bloc fetch est terminée, toutes les validations concernant ce bloc se retrouvent soit dans la partie test et simulation.

3.2 Decode

3.2.1 Introduction

Le bloc decode fait partie de l'unité de contrôle. Il contient la partie « décodage des instructions » et la banque de registre qui est accessible en lecture et écriture. C'est le décode qui va interpréter l'instruction. Il découpe l'instruction en plusieurs parties afin qu'elles puissent être utilisées par d'autres parties du processeur. C'est l'opcode qui déterminera la façon dont la valeur de l'instruction sera traitée.

3.2.2 Banque de registres

La banque de registres est composée de 8 registres. 5 d'entre permettent de stocker les données relatives à l'exécution du programme et les 3 restants sont respectivement le Stack Pointer, le Link Register et le Program Counter.

Adress reg	Fonctionnalité
0	R0
1	R1
2	R2
3	R3
4	R4
5	SP (R5)
6	LR (R6)
7	PC (R7)

Figure 12 : Organisation des registres

3.2.2.1 Instanciation des registres dans la banque

On dispose donc nos registres dans la banque en prenant bien soin de les relier à la même horloge et au même reset.

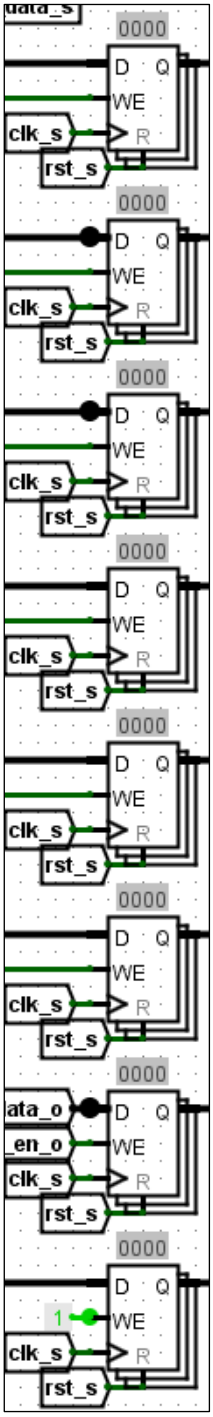

Figure 13 : Implémentation des registres

3.2.2.2 Écriture dans la banque de registre

L'écriture dans la banque de registre est similaire pour tous les registres sauf le LR et le PC qui seront traités indépendamment dans la suite du programme. Pour les autres registres, la manipulation est assez simple, on relie l'entrée reg_write_data_i à chacun d'entre eux. Ils seront par la suite activés par leur entrée « write enable » qui est reliée à un décodeur. Ce décodeur permet d'activer le bon registre en fonction de l'entrée de sélection de celui-ci. Pour la sélection du décodeur, on utilisera donc l'entrée reg_d_sel_i.

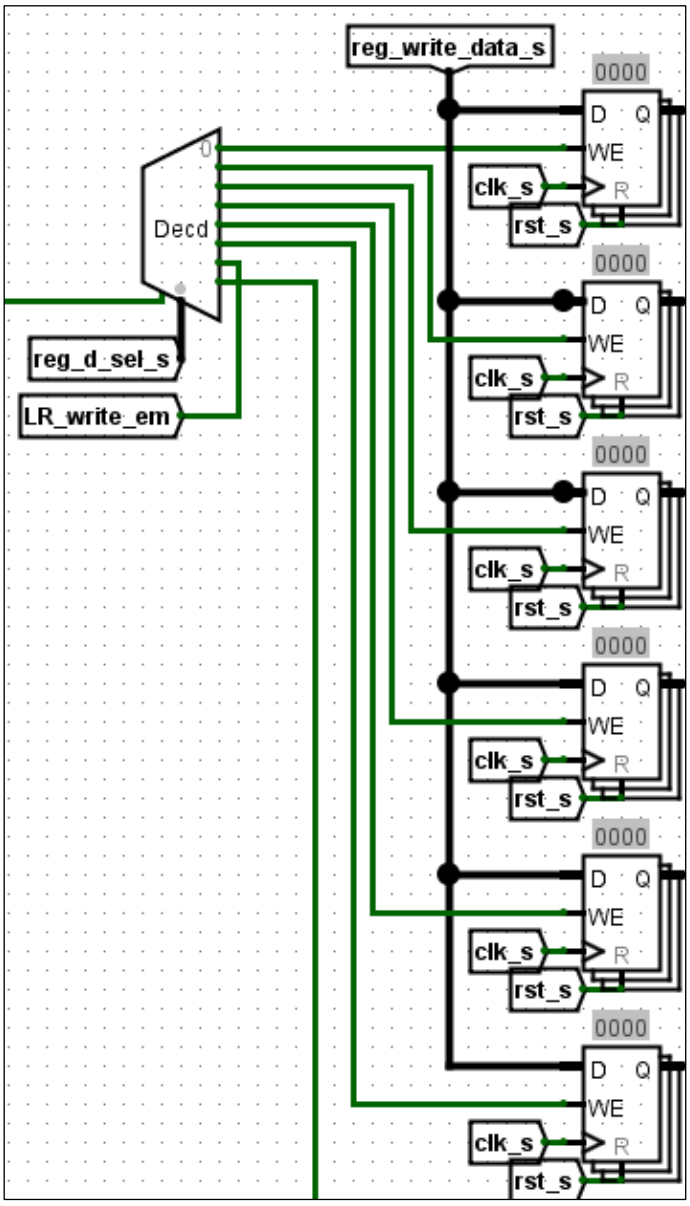

Figure 14 : Écriture dans les registres

3.2.2.3 Lecture de 3 registres dans la banque de registres

Il faut que notre processeur puisse lire en même temps le contenu de 3 registres. On doit donc à l'inverse de l'écriture utiliser 3 MUXs. Chaque registre sera relié au MUX et c'est aux entrées de sélection(reg_n_sel_i, reg_m_sel_i et reg_mem_sel_i) des différentes lectures de sélectionner le bon registre à lire. Dans ce cas, il n'y a pas d'exceptions et tous les registres sont reliés aux MUXs.

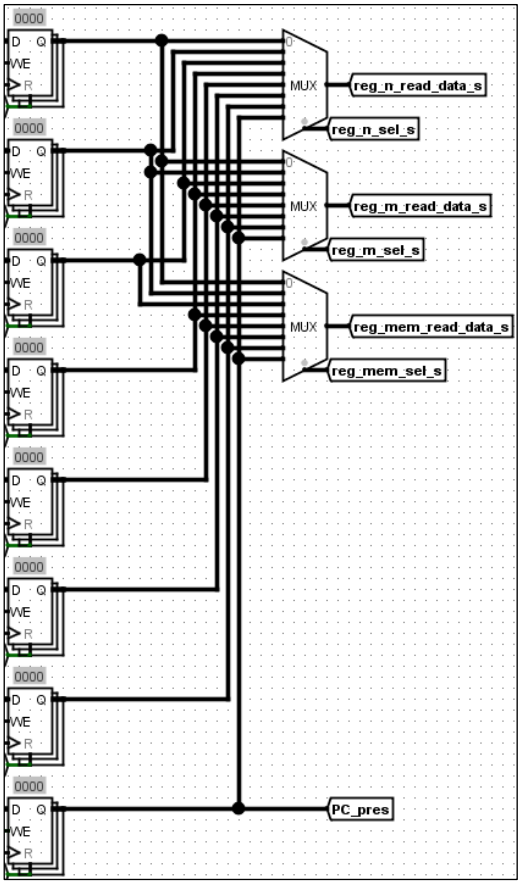

Figure 15 : Lecture des registres

3.2.2.4 Enable de la banque de registres

L'enable permet de définir si on peut écrire ou non dans la banque. C'est-à-dire est-ce que le décodeur doit décoder ou non, donc s'il est actif ou non. Il faut donc connecter à l'entrée enable du décodeur la combinaison du bit 1 du bus de contrôle de la banque de registre et l'entrée enable de la banque de registre. Le premier du bus de contrôle définir si l'écriture est autorisée. Ce qui nous donne le système ci-dessous.

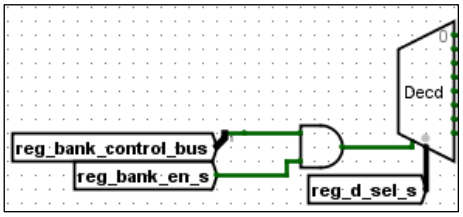

Figure 16 : Enable de la banque de registres

3.2.2.5 Registre PC

Comme dit plus haut, le PC est géré différemment c'est-à-dire qu'il doit respecter le tableau suivant.

write_en_PC	link_en	PC_reg_input+
0	0	PC_fut_i
0	1	PC_fut_i
1	0	register_d_data_i
1	1	PC_fut_i

Figure 17 : Gestion de l'entrée du PC

Il nous faut donc installer un MUX à 4 entrées respectant la disposition ci-dessus. Il aura donc 2 entrées de sélection, la première étant l'entrée link_en_i et la seconde étant la sortie du décodeur pour le PC. Ce MUX sera quant à lui relié directement à l'entrée des données du registre PC. Pour le « write enable », le PC sera toujours actif.

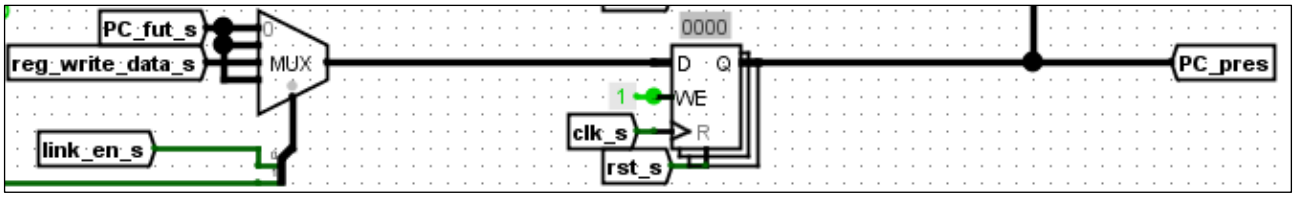

Figure 18 : Implémentation de l'entrée du PC

3.2.2.6 Gestion du LR

Le Link Register n'a pas besoin de grands ajouts, car sa logique est déjà réalisée dans le bloc LR_manager qui nous était fourni au début du laboratoire.

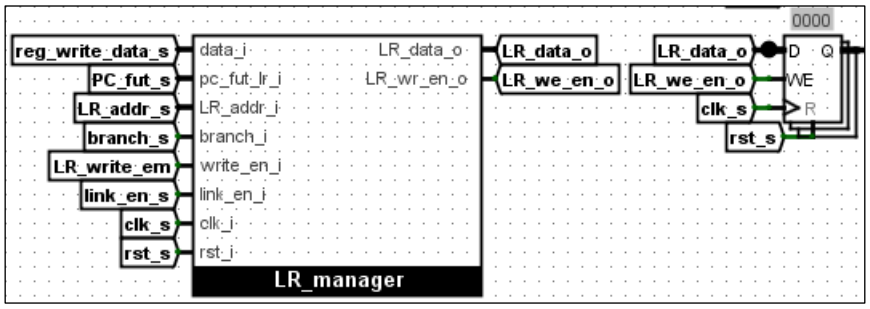

Figure 19 : Mise en place du LR manager

3.2.3 Circuit Decode

3.2.3.1 Adresse des registres

La première étape est d'implémenter la sélection d'adresse des registres N, D et M. Cette sélection se fera via des MUXs, les entrées de sélection de ces MUXs seront directement tirées decode_control_bus_s sortant du bloc main_control_unit déjà disposé sur le circuit Decode. Les bits de sélection devront respecter l'ordre ci-dessous.

Position	Taille	Description
1-0	2	Sélection Rn (source)
2	1	Sélection Rm (source)
3	1	Link enable (Utilisé pour long saut)
4	1	Sélection Rd (destination)
5	1	Pas utilisé

Figure 20 : Bus de contrôle Decode

Ce qui nous donne le résultat suivant dans Logisim. Il faut prendre en compte le nom des tunnels pour bien comprendre la suite.

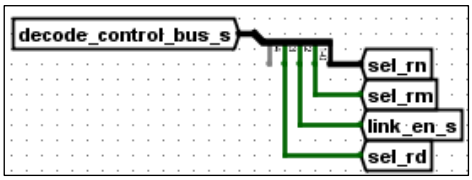

Figure 21 : Construction du bus de contrôle Decode

La sélection des adresses quant à elle ressemble au circuit ci-dessous. Elle correspond aux 3 tableaux qui nous étaient donnés dans la donnée du laboratoire.

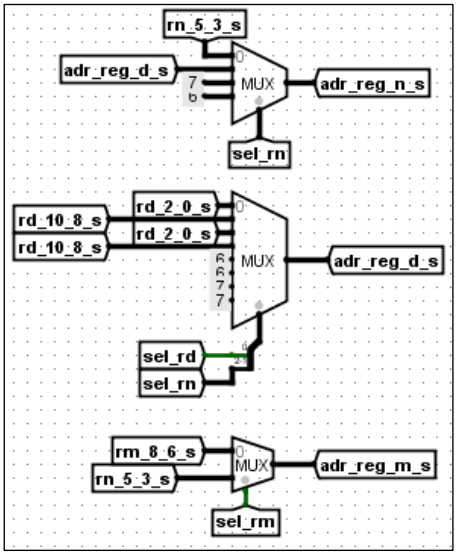

Figure 22 : Sélection d'adresses des registres

3.2.3.2 Circuit opcode_support_unit

Pour cette partie nous avons dû réaliser une partie du circuit qui permet de reconnaître les instructions. On ne va pas faire chaque instruction dans ce rapport, mais il est bien de prendre une des instructions à réaliser comme exemple. Par exemple l'instruction de stockage dans la mémoire de donner « Store Byte » prenant en paramètre 3 registres. Il faut se rendre dans notre manuel et retrouver les détails de l'instruction en question.

L	B	THUMB assembler	ARM equivalent
0	1	STRB Rd, [Rb, Ro]	STRB Rd, [Rb, Ro]

Figure 23 : Instruction type

On y voit que l'instruction a deux paramètres « variables » L et B qu'il faudra prendre en compte lorsque l'on souhaitera déterminer l'opcode représentant l'instruction. Il faut déterminer la suite de bits la représentant.

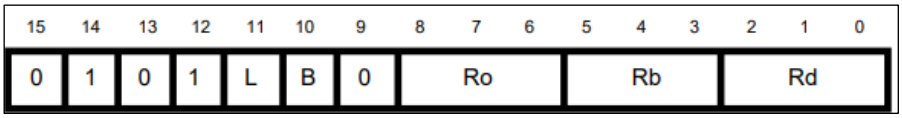

Figure 24 : Construction d'une instruction STRB

Cette suite est donc égale à « 0101010 », 5 bits fixes et deux variables. Ce qui nous donne en hexadécimal « 2A ». Il suffit donc de prendre les bons bits sur Logisim, c'est-à-dire du bit 9 au bit 15 et utiliser un comparateur de valeur pour savoir si ça correspond.

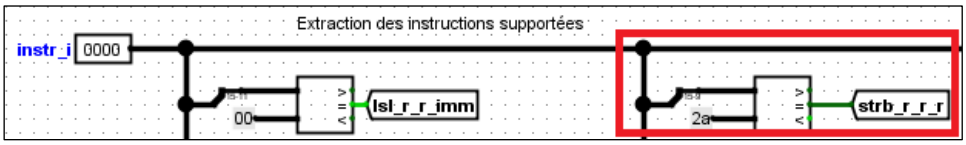

Figure 25 : Détection instruction STRB

Il faut ensuite reproduire cela pour le reste des instructions qui ne sont pas encore supportées.

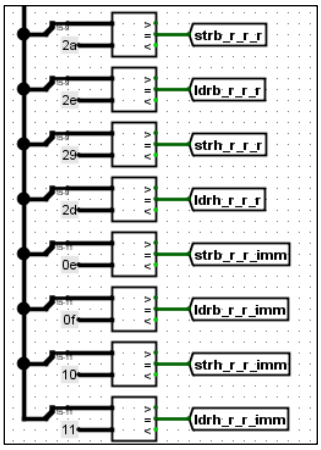

Figure 26 : Vue globale des instructions

3.2.3.3 Circuit fetch_control_unit

Ce circuit permet de déterminer quand est-ce que les sorties utilisées par le bloc fetch doivent être actives en fonction de quelle instruction est exécutée. Il faut donc respecter le tableau donné dans la consigne, il présente cependant des cases à remplir. Les seules cases à 1 de celles inconnues seront le sel_cpsr des deux « add », car c'est une instruction pouvant modifier le CPSR notamment à cause de l'overflow. Le reste des cases gardent la valeur 0.

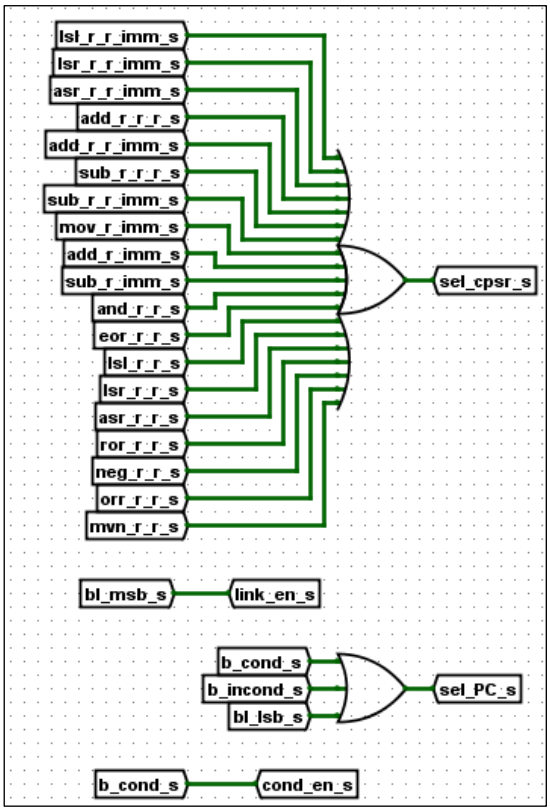

Figure 27 : fetch_control_unit

3.2.3.4 Circuit decode_control_unit

Il faut ici, aussi se fier au tableau donné dans la consigne. Pour remplir les parties manquantes dans celui-ci, il faut se rendre dans son manuel pour vérifier quels sont les registres à utiliser pour les différentes opérations.

On pourrait prendre pour exemple la ligne manquante « add_r_r_imms_s », lorsque l'on se rend dans le manuel pour observer la composition de l'instruction.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	I	Op	Rn/Offset3			Rs			Rd		

Figure 28 : Construction d'une instruction add

On remarque que les bits 0 à 2 sont pour le registre de destination, les 3 suivants pour un des registres en entrée et les 3 suivants pour un offset. Si on se rappelle de la sélection des registres faite précédemment on observe que les bits de sélection doivent être à 0 pour correspondre à la sélection des registres voulus. Exemple ci-dessous avec le registre de destination qui prend les bits 0 à 2 quand les bits de sélection sont à 0.

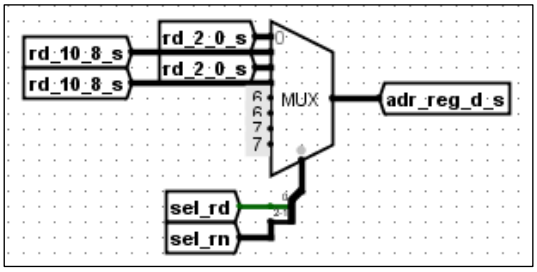

Figure 29 : Sélection de l'adresse de registre

Comme on peut le voir ci-dessous, les deux cases inconnues « ror » et « mov » sont toutes les deux une fois à 1, respectivement pour sel_rm et sel_rd.

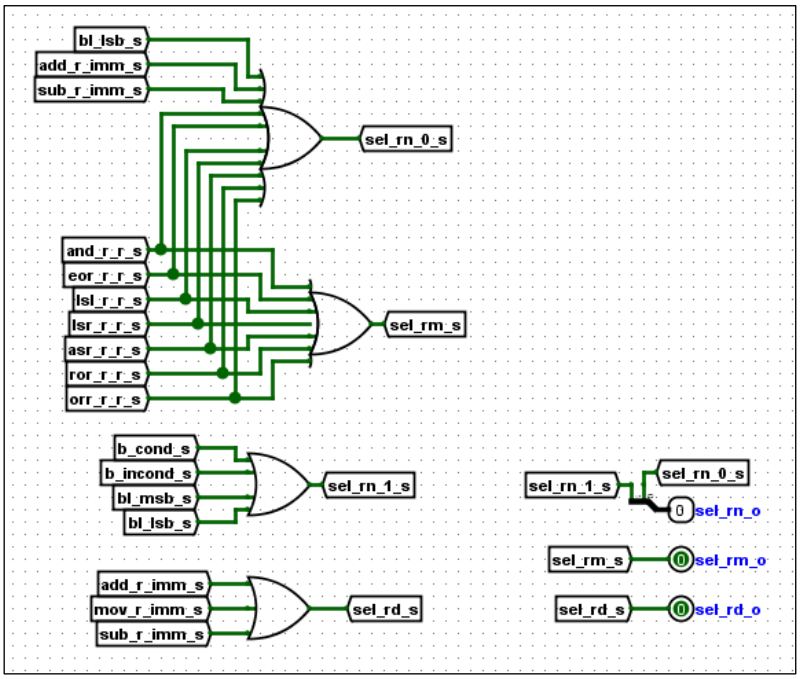

Figure 30 : decode_control_unit

3.2.3.5 Circuit reg_bank_control_unit

Pour ce cas, ce sont les instructions activant une écriture dans la banque des registres qui passeront la sortie à 1. Des cases inconnues, il n'y a que lsl_r_r_imm_s et eor_r_r_s qui écrivent dans la banque de registre. strh_r_r_r_s écrit bien quelque part, mais c'est dans la mémoire de donnée.

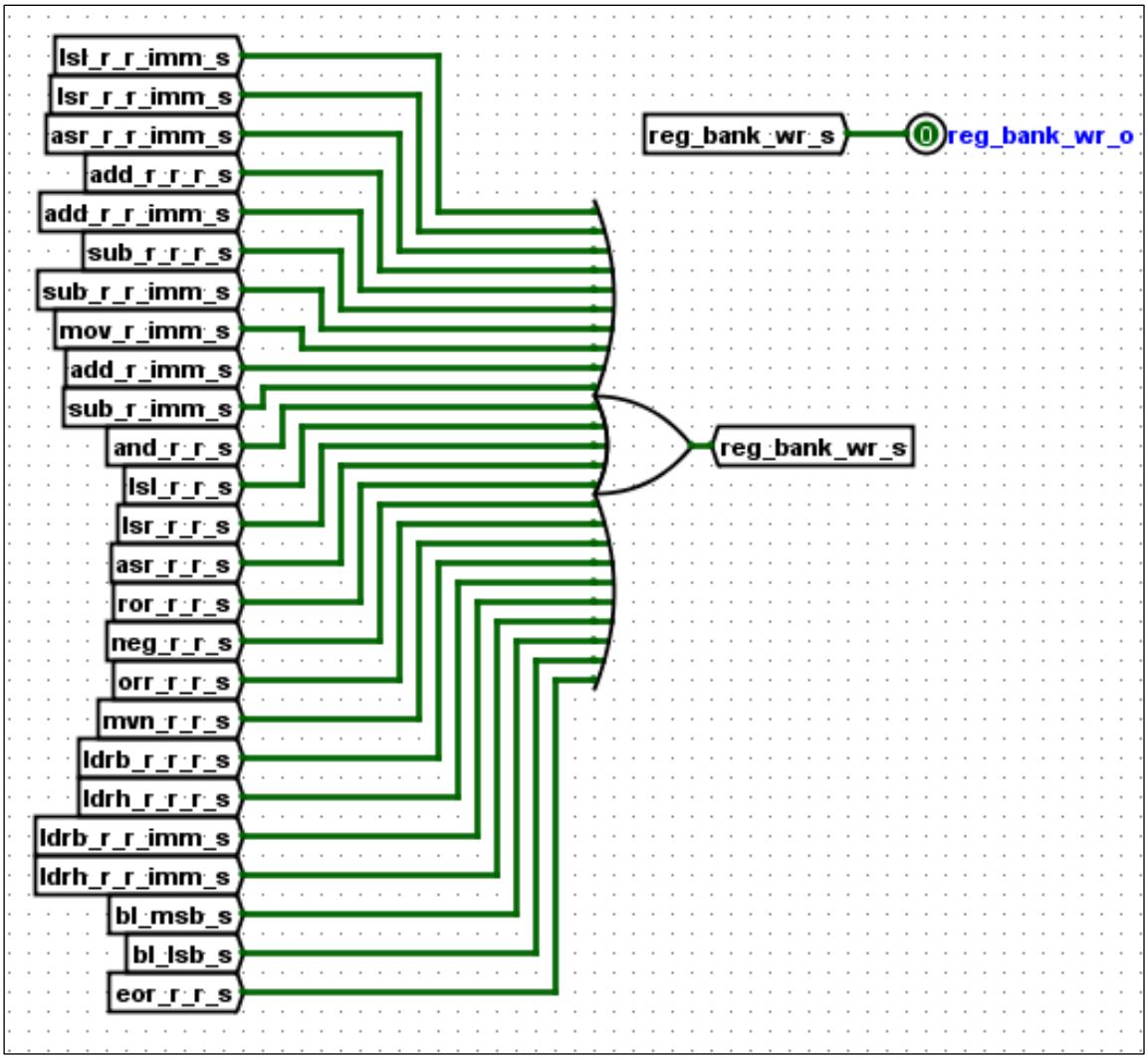

Figure 31 : reg_bank_control_unit

3.2.3.6 Circuit execute_control_unit

Le tableau définissant ce circuit est déjà complet dans la donnée, il suffit de l'implémenter dans Logisim.

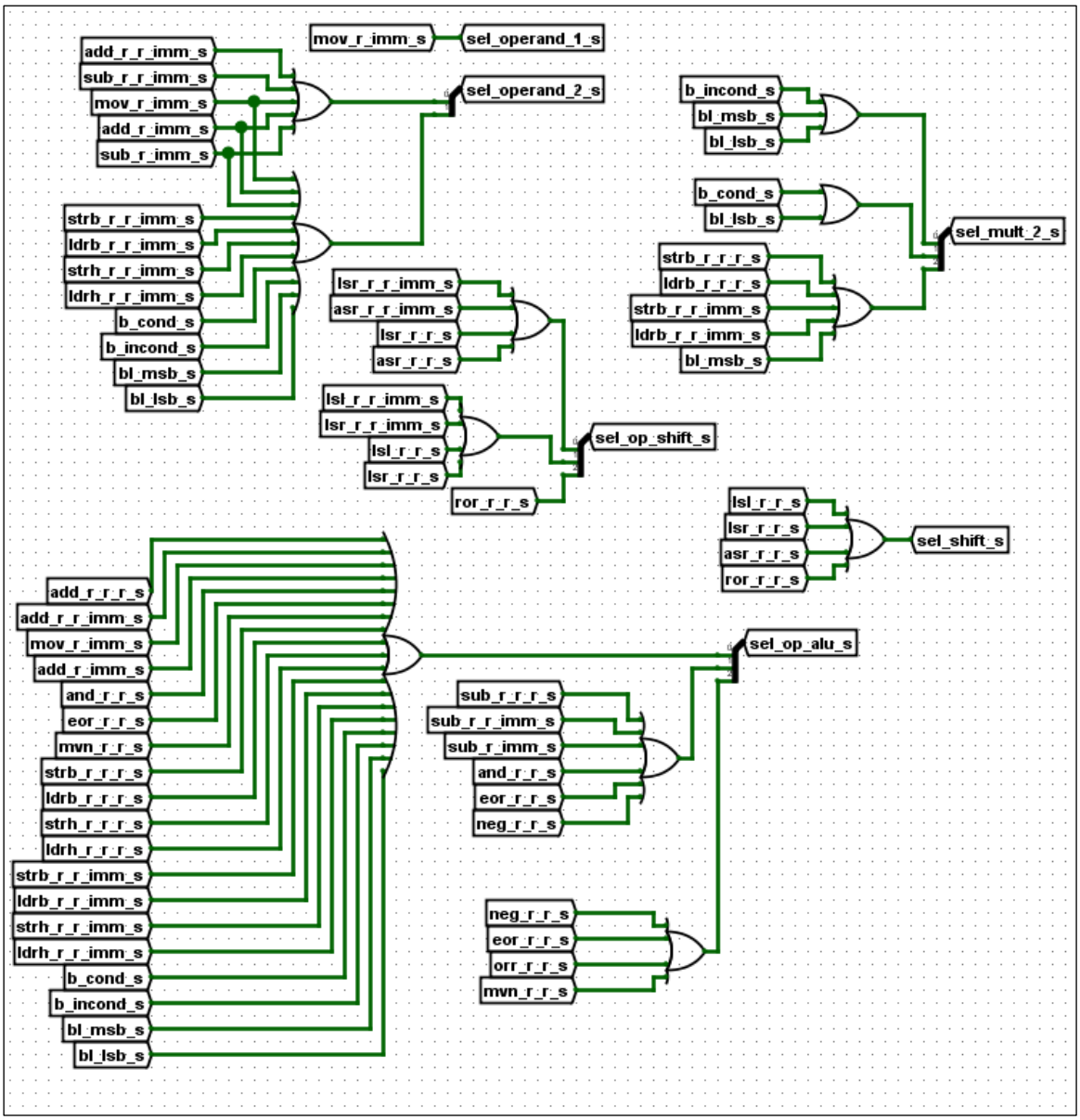

Figure 32 : execute_control_unit

3.2.3.7 Circuit memory_access_control_unit

Pour les cases inconnues sur le tableau, seules les instructions ldrh et strb accèdent à la mémoire. L'instruction strb est celle qui écrit dans la mémoire (store) et ldrh est celle qui lit (load). L'accès par Byte est pour strb (dont le « b » représente Byte).

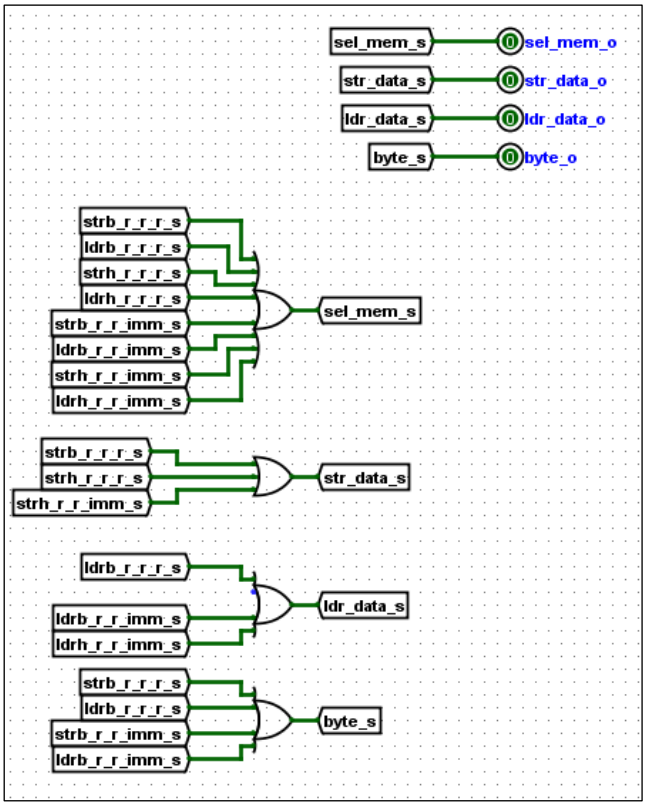

Figure 33 : memory_access_control_unit

3.3 Execute et Memory Access

3.3.1 Introduction

Le bloc Execute est composé du bloc ALU, du bloc SHIFT et du registre CPSR. Il permet d'effectuer toutes les opérations nécessaires au sein du processeur. Le bloc Memory Access permet quant à lui d'accéder à la mémoire de données en lecture et écriture.

3.3.2 Bloc Execute

Notre bloc Execute sera composé d'un bloc SHIFT et d'un bloc ALU que l'on devra réaliser, ainsi que d'un bloc MULT_2 et d'un bloc ZCNV_UNIT qui sont tous les deux déjà existants. Il faudra simplement les relier entre eux comme on le verra par la suite.

3.3.2.1 Bloc Shift

Le premier bloc à réaliser est le bloc SHIFT, celui-ci permettra de faire les différentes sortes de shifts afin de pouvoir exécuter les opérations supportées. Le bloc comprend les entrées suivantes :

Nom de l'entrée	But
operand_i	Valeur que l'on va shifter
sel_shift_data_i	Détermine de combien on shift
sel_op_shift_i	Permets de sélectionner le shift

On utilisera pour décaler, le bloc « Décalage » de Logisim en paramétrant leur option « Type de décalage » afin de définir le shift à accomplir. On devra relier tous les shifts à un MUX en respectant le tableau suivant :

sel_op_shift_s	shift_data_out_s
0	operand_s (bypass)
1	shift arithmétique (ASR)
2	shift logique à gauche (LSL)
3	shift logique à droite (LSR)
4	shift rotatif à droite (ROR)
5	0x0000 (pas assigné)
6	0x0000 (pas assigné)
7	0x0000 (pas assigné)

Figure 34 : Opérations bloc Shift

Le résultat final que l'on a réalisé sur Logisim correspond au circuit ci-dessous.

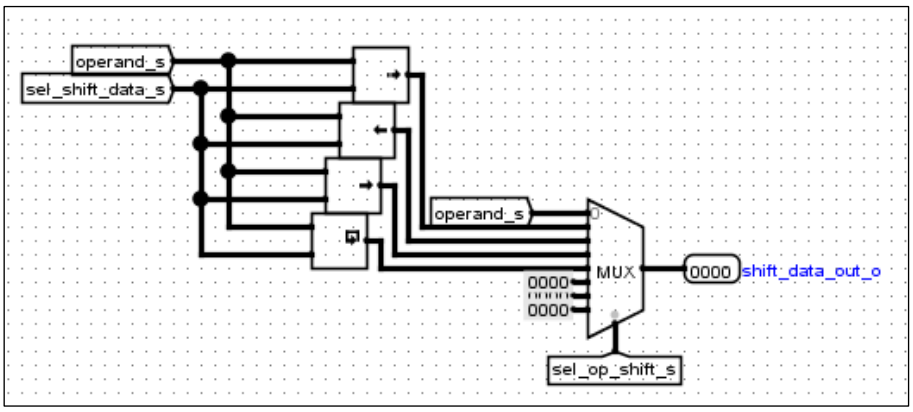

Figure 35 : Implémentation du bloc Shift

3.3.2.2 Bloc Alu

Le second bloc que l'on doit réaliser est un bloc ALU. Il nous permettra de réaliser toutes les opérations arithmétiques de notre processeur. Il faudra en plus de fournir le résultat de l'opération, indiquer si l'opération à provoquer un « carry » ou un « overflow ». Notre bloc est composé des entrées suivantes :

Nom de l'entrée	But
operand_1_i	Première valeur de l'opération
operand_2_i	Seconde valeur de l'opération
sel_op_alu_i	Sélection de l'opération à effectuer

La première étape dans ce bloc est d'implémenter les opérations arithmétiques. Celles-ci sont décrites dans le tableau ci-dessous.

sel_op_alu_s	data_out_s
0	operand_1_s (bypass)
1	operand_1_s + operand_2_s
2	operand_1_s - operand_2_s
3	operand_1_s AND operand_2_s
4	operand_1_s OR operand_2_s
5	NOT operand_1_s
6	NEG operand_1_s (=> *-1)
7	operand_1_s XOR operand_2_s

Figure 36 : Construction du bloc Alu

Logisim implémente nativement tous les blocs pour réaliser ces opérations. On va donc profiter de pouvoir les utiliser comme ci-dessous.

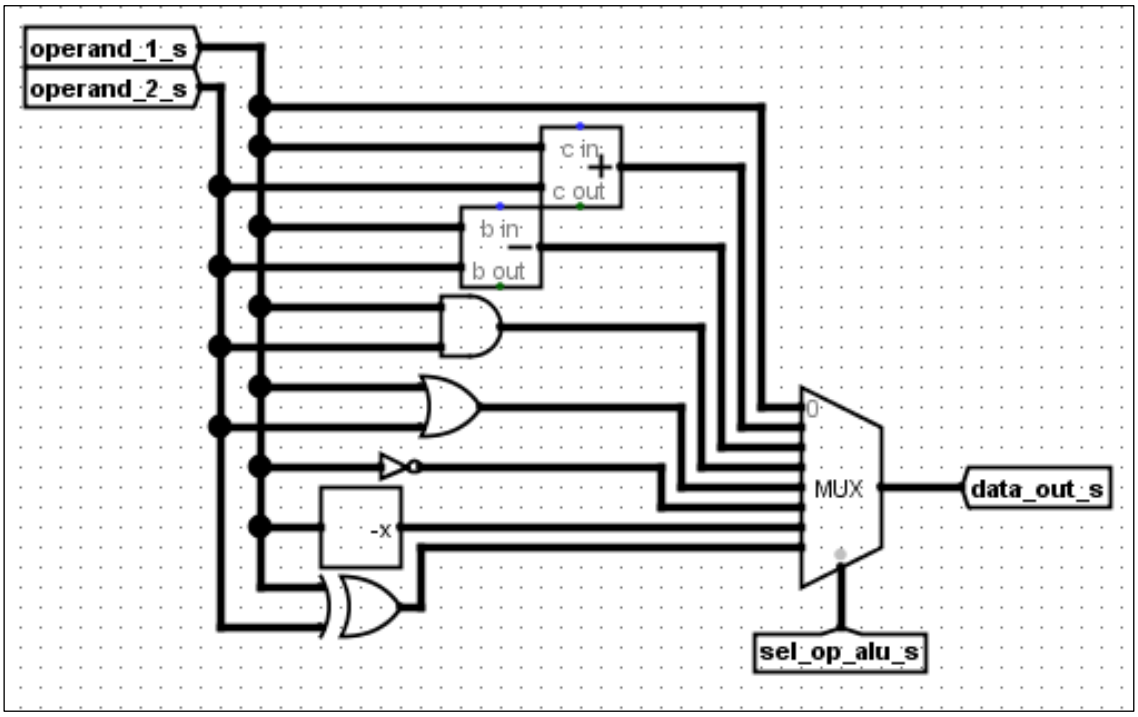

Figure 37 : Implémentation du bloc Alu

Il faut aussi pouvoir gérer le carry et l'overflow qui seront deux informations utilisées par le CPSR. Pour le carry c'est assez simple étant donné que le bloc additionneur et soustracteur proposent tous deux la gestion du carry. Pour l'overflow, la méthode la plus simple est de mettre en place le même système que celui vu en SYL.

Opération	A	B	Résultat indiquant un overflow
A + B	≥ 0	≥ 0	< 0
A + B	< 0	< 0	≥ 0
A - B	≥ 0	< 0	< 0
A - B	< 0	≥ 0	≥ 0

Figure 38 : Conditions d'Overflow

Ces deux étapes permettent de faire évoluer notre circuit de la manière suivante.

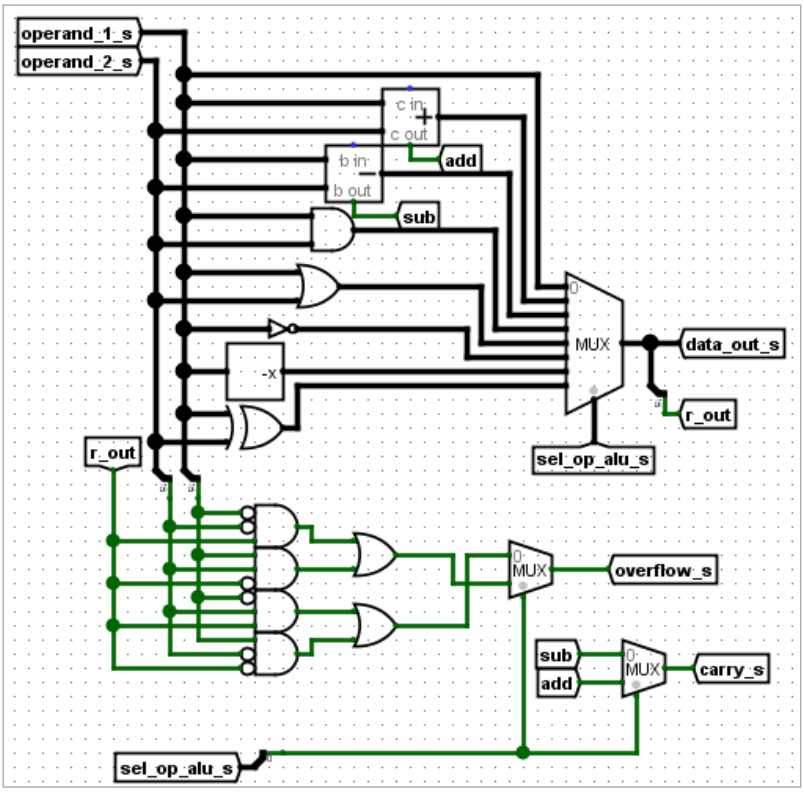

Figure 39 : Implémentation du système d'Overflow

3.3.2.3 Réalisation de la sélection de l'opérande 2

Cette partie est simplement composée d'un MUX qui correspond au tableau ci-dessous. On utilisera les bits 6 et 7 de l'entrée « execute_control_bus_i » comme indiqué dans la construction du bus en question.

sel_operand_2_s	operand_2_s
0	reg_data_m_s
1	ext16_unsigned(imm3_s)
2	mult_2_out_s
3	ext16_unsigned(imm8_s)

Figure 40 : Construction de l'operand_2

La réalisation du circuit nous donne la partie suivante que l'on connecte à l'entrée « operand_2_i » du bloc ALU.

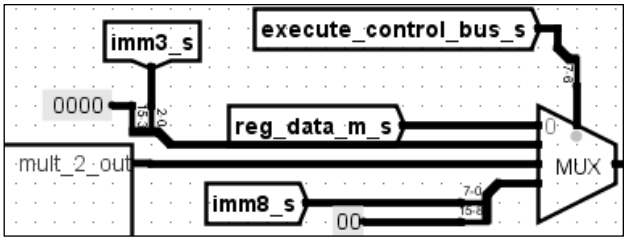

Figure 41 : Implémentation de l'operande_2

3.3.2.4 Réalisation de la sélection de l'opérande 1

La sélection de cet opérande est similaire au précédent, sauf que le bit de sélection pour le MUX est le bit 8 et que les entrées du MUX doivent respecter le tableau suivant.

sel_operand_1_s	operand_1_s
0	reg_data_n_s
1	0x0000

Figure 42 : Construction de l'operand_1

L'application au sein du circuit donne le bloc ci-dessous que l'on connecte à l'entrée « operand_i » de notre bloc SHIFT.

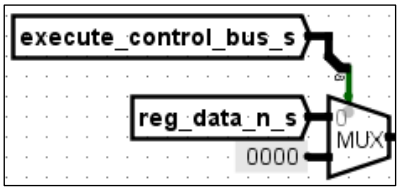

Figure 43 : Implémentation de l'operand_1

3.3.2.5 Réalisation de la sélection de la valeur pour le shift

Cette partie est aussi composée d'un MUX et permettra de donner au bloc SHIFT le décalage à réaliser. Le tableau correspond au MUX sera le suivant.

sel_shift_i	sel_shift_data_s
0	imm5_i(3 :0)
1	reg_data_m_i(3 :0)

Figure 44 : Construction de la valeur pour le shift

Le circuit quant à lui sera représenté de la manière suivante.

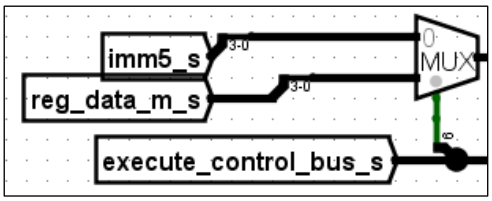

Figure 45 : Implémentation de la sélection de la valeur pour shift

3.3.2.6 Implémentation du CPSR

La dernière partie que l'on doit implémenter nous même au sein de ce bloc execute est la gestion du CPSR. Pour ce faire, nous avions à disposition un bloc ZCNV_UNIT déjà fourni et fonctionnel. Il suffit de réaliser les bonnes connexions.

On va donc connecter notre bloc ALU dont les sorties correspondent aux entrées du bloc ZCNV_UNIT. Puis les sorties de ce bloc seront simplement insérées dans un registre 4 bits qui représentera le registre CPSR.

Ce registre sera d'ailleurs relié au bit 10 du bus de contrôle du bloc execute qui correspond au flag qui met à jour le CPSR. Donc lorsque ce bit sera actif, le registre se mettra à jour au prochain coup d'horloge.

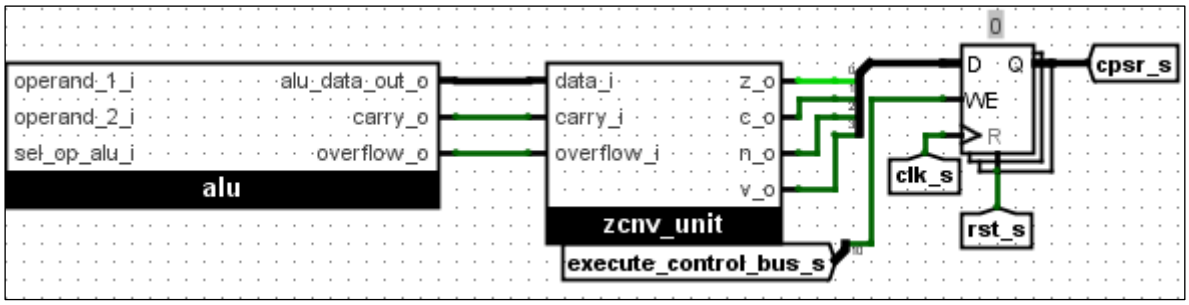

Figure 46 : Implémentation du registre CPSR

3.3.2.7 Vue d'ensemble

Pour les entrées du bloc, elles ont toutes été indiquées précédemment. Pour les sorties, la sortie « cpsr_o » sera reliée à la sortie du registre CPSR et la sortie « data_out_o » est reliée à la sortie « alu_data_out_o » du bloc ALU.

Au niveau des connexions entre les blocs, le bloc ZCNV_UNIT est relié au bloc ALU comme vu ci-dessus. Le bloc ALU quant à lui reçoit dans son entrée « operand_1_i », la sortie du bloc SHIFT et dans sa seconde entrée d'opérande « operand_2_i » le MUX qui prend lui-même le bloc MULT_ 2 dans une de ses entrées. La vue d'ensemble est disponible dans les deux captures ci-dessous.

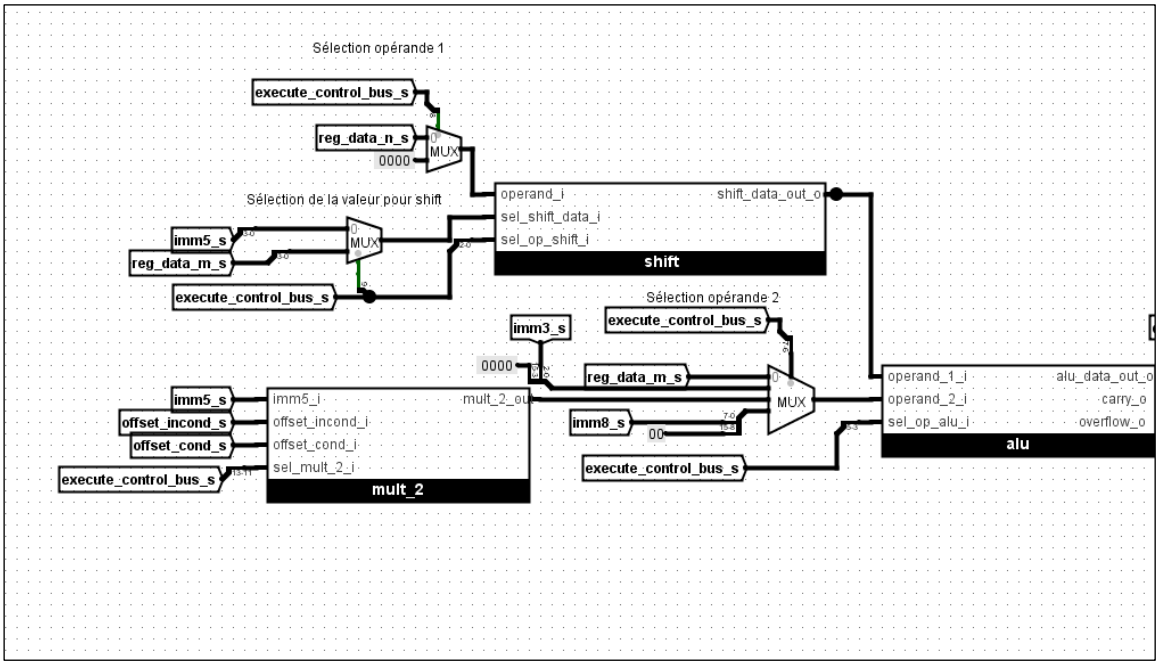

Figure 47 : Vue d'ensemble du bloc Execute (1/2)

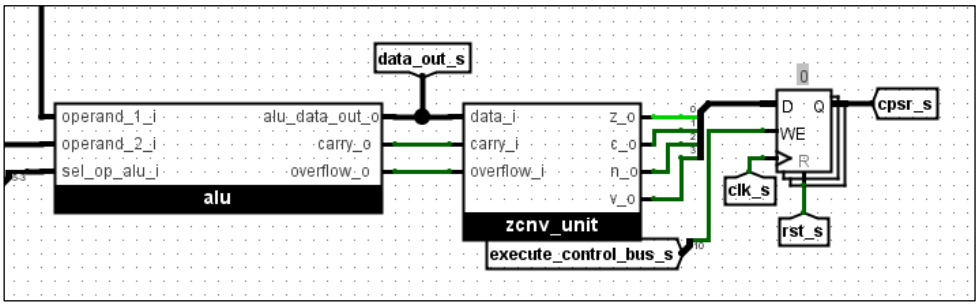

Figure 48 : Vue d'ensemble du bloc Execute (2/2)

3.4 Pipeline Partie 1

3.4.1 Introduction

Le principe du pipeline est un concept introduit dans le but d'augmenter la vitesse de traitement d'un micro-processeur (RISC).

Le pipeline est l'équivalent informatique du travail en chaîne. L'instruction demande un certain nombre d'étapes élémentaires, et chaque unité travaille à partir du résultat d'une autre étape précédemment réalisé. Ainsi le processus complète du pipeline pour un produit nécessite de passer par toutes les unités en succession, ce ceci représente la latence de notre pipeline.
Analyse et test du processeur

3.4.1.1 Analyse du processeur

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

Dans le circuit non-pipeliné, les opérations se réalisaient en une seule étape. Dans le circuit pipeliné, il y a un registre supplémentaire dans le bloc fetch, on perd alors un coup de clock. Il faut alors le compenser en passant l'incrément de l'offset de 2 à 1. Ainsi, on obtient $PC + (offset + 1) * 2 + 2 = PC + (offset + 2) * 2$. L'addition manquante étant réalisée directement dans le bloc fetch.

3.4.1.1.2 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 ?

Comme dit dans la consigne tous les signaux passent par tous les blocs même s'ils n'y sont pas utilisés, afin de s'assurer que les informations de contrôle arrivent au même temps que les données

3.4.1.1.3 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 mémoires sont directement envoyés à l'execute et sont enregistrés dans ce bloc. Cependant, dans le decode on doit pouvoir décoder l'instruction. C'est pourquoi il faut sauvegarder quelque part l'instruction précédente pour ensutie envoyer les données venant du bloc execute dans les instructions suivantes.

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

S'il y'a une dépendance qui provoque un aléa et donc qu'il faut bloquer le pipeline. Le système hazard_detection permet de désactiver les étages nécessaires et donc leur bus de contrôle sera à 0 pour qu'ils n'effectuent aucune opération. Cela permet notamment de bloquer les signaux et de ne pas propager la mauvaise information.

3.4.1.1.5 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 ?

Ça rajouterait une étape dans le circuit du Pipeline. Il faudrait aller faire des modifications dans la manière dont son relier les blocs.

3.4.1.2 Assembleur : dépendances de données

```
MAIN_START:
MOV r0, #1
MOV r1, #2
MOV r2, #6
NOP
NOP
STRH r0, [r1, #4] @WAW r0 || RAW r1
ADD r4, r2, #1    @RAW r2
ADD r3, r2, #4    @RAW r2
SUB r4, r1, r0    @WAW R4 || RAW r1
ADD r0, r0, #5    @WAW r0 || RAW r0
LSL r2, r2, #1    @WAW r2 || RAW r2
NOP
NOP
NOP
LSL r2, r2, #1    @WAW r2 || RAW r2
B PART_2
```

Figure 49 : Code avec dépendances de données

Voici le chronogramme du code ci-dessus. Nous pouvons constater que les NOP sont pris en compte. C'est la valeur avec le code d'instruction46C0. On peut aussi constater que le STRH sort bien la valeur 1. La valeur sortant égale à 0 est due aux NOP.

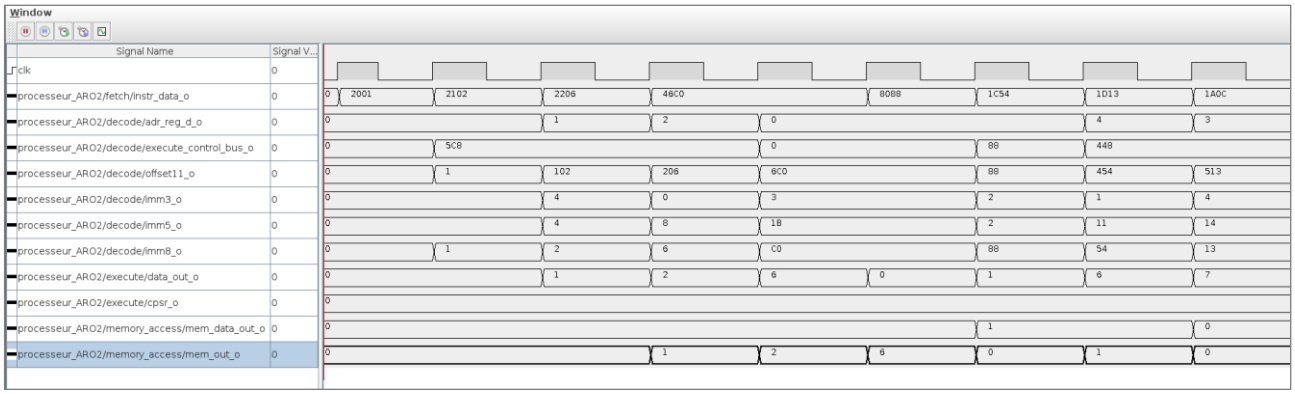

Figure 50 : Chronogramme avec dépendances de données

3.4.1.2.1 Quelles dépendances posent des problèmes d'aléas ?

C'est r1 : STRH r0, [r1, #4]

C'est r2 : LSL r2, r2, #1

3.4.1.2.2 Combien de cycles sont nécessaires pour résoudre un aléa de donnée ?

Il faut 3 cycles pour résoudre un aléa de donnée.

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

10 instructions et 19 cycles pour toutes les réaliser.

IPC = 0.52 --> 10 instructions / 19 cycles

Débit = 0.52 instr/cycle * 4000 cycles/sec = 2080 instr/cycle

Latence = 1/Débit = 1/2080 = 0,00048077 sec = 0,480 ms

3.4.1.3 Assembleur : aléas de contrôle

Sur les deux chronogrammes suivants, nous avons l'exécution du même code, mais en premier sans les NOP, et en suite avec. On peut immédiatement remarquer que notre mem_out n'est pas la même. Les NOP ont permis de corriger l'erreur du E qui vient s'intercaler dans la sortie de notre programme.

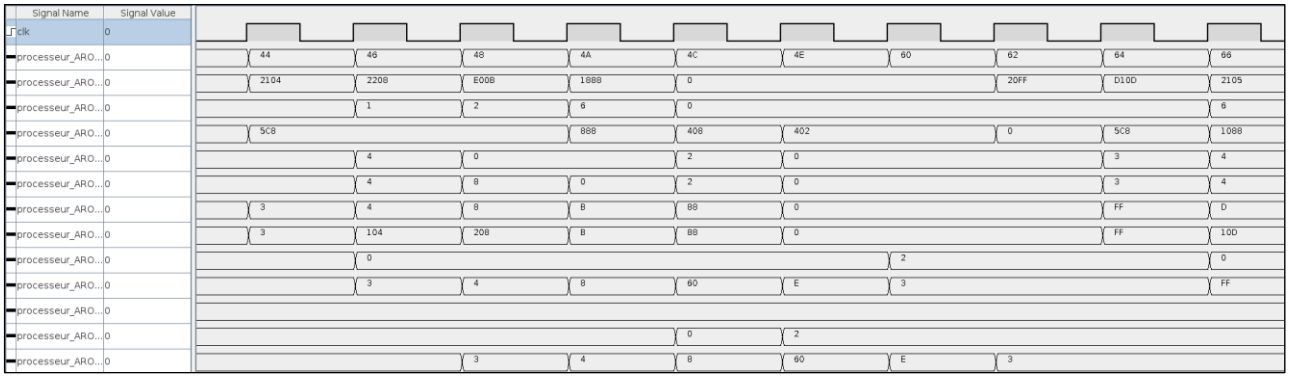

Figure 51 : Chronogramme sans les NOP

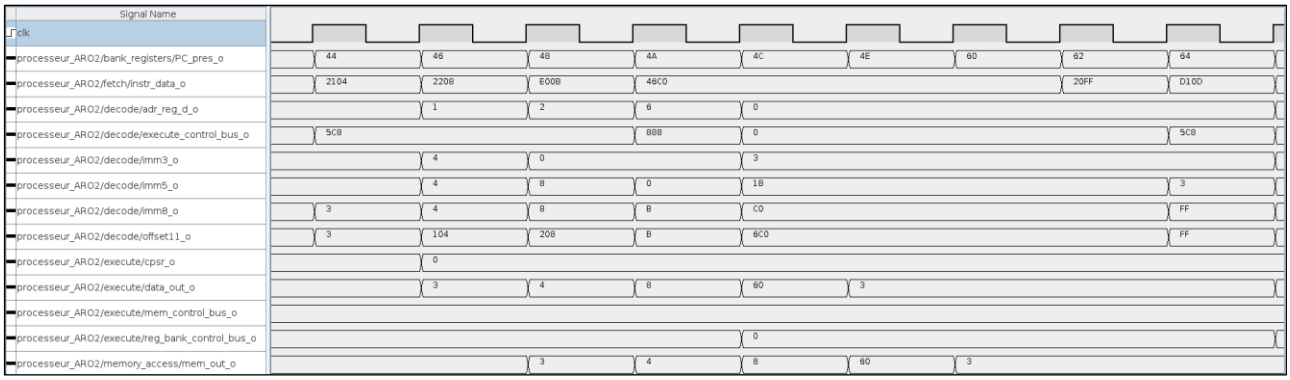

Figure 52 : Chronogramme sans les NOP

3.4.1.3.1 Combien de cycles sont nécessaires pour résoudre un aléa de contrôle ?

Il faut exactement 4 cycles (NOP) pour éviter un aléa de contrôle comme le montre le premier chronogramme sans les NOP

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

IPC = 0.52 --> 10 instructions / 19 cycles

Débit = 0.52 instr/cycle * 4000 cycles/sec = 2080 instr/cycle

Latence = 1/Débit = 1/2080 = 0,00048077 sec = 0,480 ms

3.4.2 Aléas de contrôle

3.4.2.1 Circuit control_hazard

Il y a deux conditions pour que les registres sont remis à zéro :

- rst_s est à 1
- Aucune instruction de contrôle en plus d'un aléa de contrôle

Dans le cas ou une autre instruction de contrôle intervient avant la fin de l'aléa, le reset sert à prolonger l'output.

Le XNOR final, sort un aléa de contrôle si une instruction de contrôle :

- Est en cours actuellement
- A eu lieu 3 étapes auparavant.

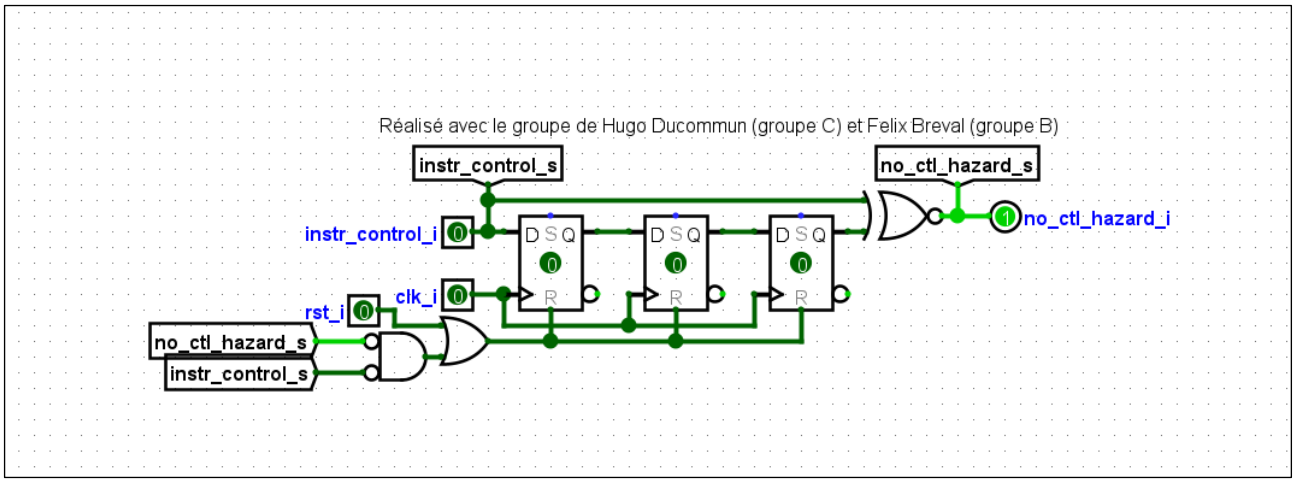

Figure 53 : Circuit control_hazard

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

Il va falloir 4 cycles pour que le pipeline bloque dans le cas d'un aléa de contrôle.

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

Pour s'assurer que le programme n'exécute pas les instructions suivantes. Typiquement dans le cas d'un saut qui devrait être réalisé.

3.4.2.1.3 Quels sont les conditions pour qu'un aléa de contrôle ait lieu ?

Lors de l'exécution d'une instruction de branchement conditionnel, on dit que le branchement est pris si la condition est vérifiée et que le programme se poursuit effectivement à la nouvelle adresse.

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

L'aléa de donnée est prioritaire, car il faut que l'on ait les bonnes valeurs dans les registres avant de faire un calcul d'adresses.

3.4.2.2 Circuit hazard_detection

Nous avons réalisé cette partie dans le bloc main_control_unit, et passons directement instr_control_s comment entrée du bloc hazard_detection. L'entrée instr_control_s est active lorsque l'instruction courante est un saut qu'il soit conditionnel, in conditionnel ou non.

On a remarqué que ce même système se retrouve dans le fetch_control_unit fourni.

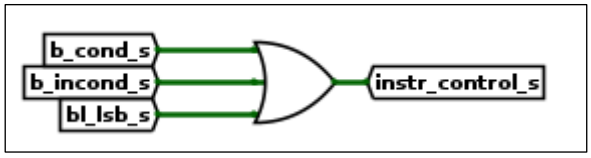

Figure 54 : Connections pour le signal instr_control_s

3.4.2.2.1 Quelles instructions génèrent un aléa de contrôle ?

Ce sont les instructions de branchement

3.4.2.2.2 Comment les aléas de contrôle influencent les différents enables ?

On remarque que le fetch est directement impacté, car il est branché au signal directement. Tandis que le reste des signaux ont des registres avant, donc ils sont décalés de certains coup de clock.

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

Le pipeline s'interrompt et le programme se poursuivra à la nouvelle adresse donc à la nouvelle instruction.

3.4.2.2.4 Pourquoi branch_i est dans les entrées du circuit hazard_detection ?

Cela nous permet d'identifier si l'instruction précédente était un saut et s'il faut bloquer le pipeline pour que celui-ci puisse avoir le temps de s'exécuter.

3.4.2.2.5 Pourquoi instr_control_i du bloc control_hazard dépend de no_data_hazard_s?

Car on ne va pas prendre en compte les aléas de données

3.4.2.3 Test aléas de contrôle

```
MAIN_START:
MOV r0, #1
MOV r1, #2
MOV r2, #6
B PART_2

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

.org 0x80
SAUTC:
MOV r0, #0
BEQ MAIN_START
```

Figure 55 : Code contenant des aléas de contrôle

On peut constater sur le chronogramme suivant que lorsque nous arrivons à l'instruction de saut conditionnel le programme interrompt la lecture des instructions suivantes durant 4 cycles afin de pouvoir calculer l'adresse de saut. On remarque aussi qu'il pré-fetch la prochaine instruction mais celle-ci ne sera jamais décodée / exécutée.

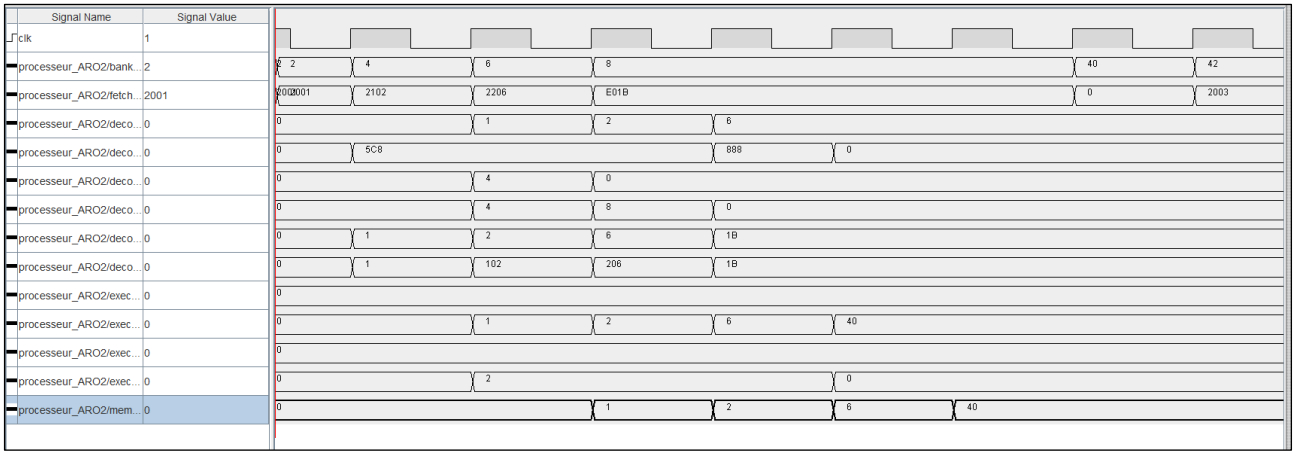

Figure 56 : Chronogramme aléas de contrôle

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

Oui, elles le sont. On arrête le pipeline au fetch de la prochaine instruction, mais celle-ci n'est pas traitée si un saut est effectué.

3.4.2.3.2 Quel est l'IPC pour votre programme ?

IPC = 0.47 = 10/21

3.4.2.3.3 Pourquoi l'instruction BL génère en même temps un aléa de contrôle et un aléa de donnée ?

BL est composé de deux sous-instructions avec le msb et le lsb. msb écrit dans le LR et lsb ira lire dans ce même registre, ce qui génère un aléa de données. Pour l'aléa de contrôle c'est simplement que BL est un saut.

3.5 Pipeline Partie 2

3.5.1 Aléas de donnée

3.5.1.1 Circuit data_hazard

Attention, nous allons détailler pour chaque partie uniquement une des parties du système. Il faut donc réappliquer l'analyse sur l'ensemble pour tout comprendre.

Avec le circuit suivant, nous pouvons vérifier si le registre en lecture à l'instruction présente a été utilisé comme registre d'écriture lors des 3 instructions précédentes. Ce qui permet de vérifier si un aléa doit être générer suite à une dépendance RAW.

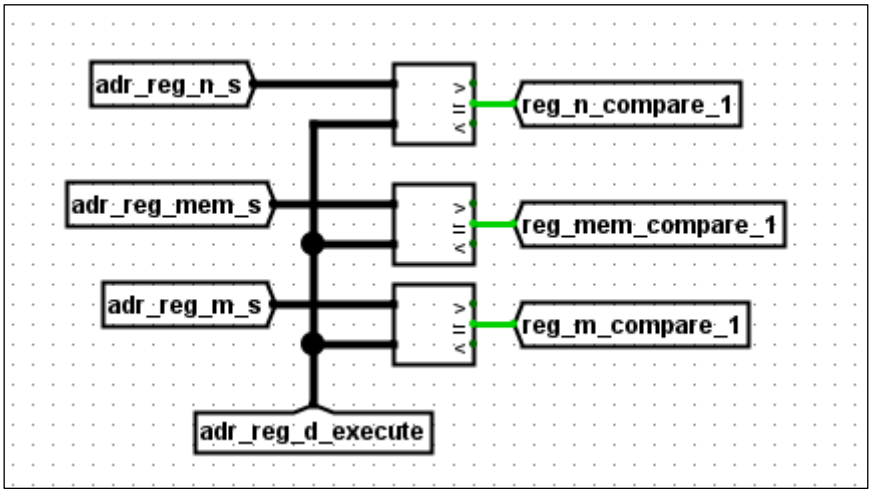

Figure 57 : Comparaison du registre de destination utilisé

Ici, nous vérifier que différentes conditions signalant un aléa de donnée soient activées. Ces conditions sont notamment composées du signal calculé précédemment nous indiquant si un registre génère une dépendance RAW, du signal indiquant si le bloc exécute est activé, si la banque de registre à précédemment été utilisé et si un registre de lecture est utilisé par l'instruction courante.

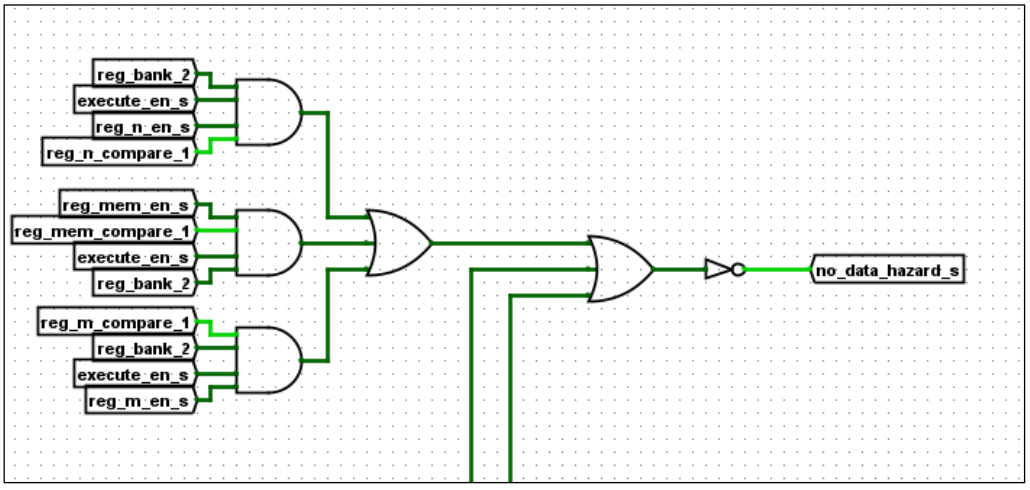

Figure 58 : Vérification du no_data_hazard_s

Le circuit du dessus permet d'enregistrer le registre de destination utilisé durant les trois derniers coups d'horloge. On utilise cette information pour détecter si l'instruction courante génère une dépendance RAW.

La partie du bas permet de mémoriser si la banque de registre à été utilisé durant les trois dernières instructions.

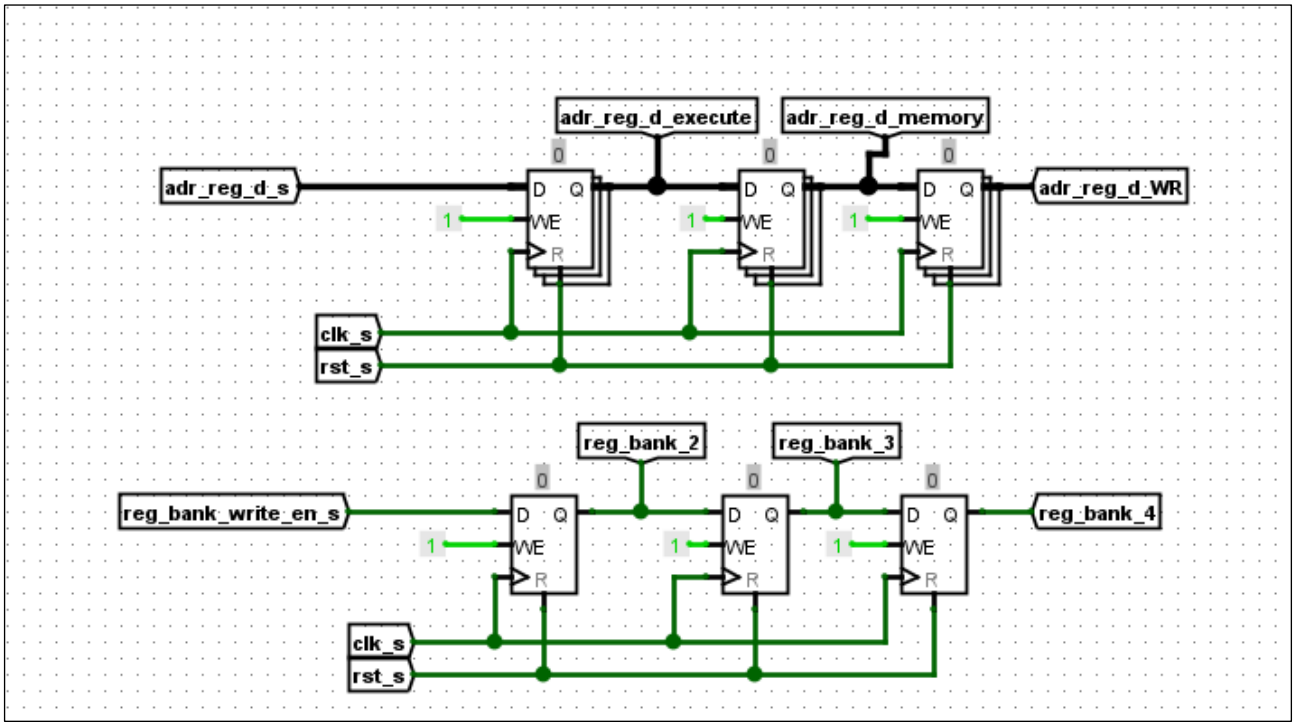

Figure 59 : Propagation de l'information via les registres par étapes

3.5.1.1.1 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 ?

On compare les registres devant être accédés par l'instruction avec les registres accédés aux différentes étapes du Pipeline précédemment. On stocke notamment l'adresse en sortie d'execute, du memory access et lors du write-back.

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

Oui, car si on n'a pas encore écrit le résultat en mémoire alors on peut avoir un problème lorsque l'on récupère l'information.

3.5.1.1.3 Quelles informations doivent être mémorisées pour chaque instruction ?

Le registre de destination est utilisé si le signal no_data_hazard_s est à 0.

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

La détection arrête le Decode de l'instruction qui pose problème, ça propagera l'arrêt du pipeline pour les enables suivants.

3.5.1.2 Circuit hazard_detection

Afin d'ajouter les signaux **reg_n_en_s**, **reg_m_en_s**, **reg_mem_en_s**, il faut nous rendre dans le circuit main_control_unit. Nous allons relier comme expliquer ci-dessous les trois signaux.

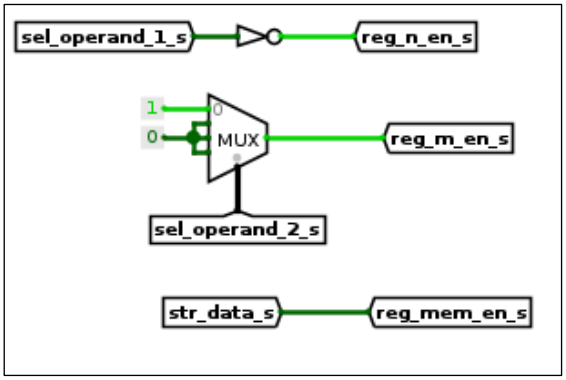

Figure 60 : Modification du circuit main_control_unit

Comme on le remarque dans le tableau ci-dessous pour que le signal indiquant que le registre N est utilisé il faut que la valeur de sel_operand_1_s soit 0. C'est le même principe pour la valeur de M ainsi que l'oprand_2

sel_operand_1_s	operand_1_s
0	reg_data_n_s
1	0x0000

Figure 61 : Valeurs de N

sel_operand_2_s	operand_2_s
0	reg_data_m_s
1	ext16_unsigned(imm3_s)
2	mult_2_out_s
3	ext16_unsigned(imm8_s)

Figure 62 : Valeurs de M

Le str_data_o indique directement que le signal reg_mem_en_s est activé. On retrouve l'explication dans la capture ci-dessous tirée de la donnée du laboratoire execute.

str_data_o	Flag qui indique qu'on veut stocker une donnée en mémoire

Figure 63 ; Valeur str_data_o

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

Se référer au laboratoire execute où on nous indique quand est-ce que les différents registres doivent être utilisés.

N	Lorsque sel_operand_1_s est désactivé, alors on utilise le registre N.
M	Lorsque sel_operand_2_s est désactivé, alors on utilise le registre M.
Mem	Str_data_s indique que l'on va écrire dans la mémoire nous permet de déterminer si le registre Mem est utilisé.

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

Le registre de destination est utilisé si le signal no_data_hazard_s est à 0.

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

La détection arrête le Decode de l'instruction qui pose problème, ça propagera l'arrêt du pipeline pour les enables suivants.

3.5.2 Pipeline Forwarding

Ces questions ont été répondues avec la version non fonctionnelle du pipeline forwarding.

3.5.2.1 Circuit data_hazard

3.5.2.1.1 A quoi sert le signal sel_mem_i ?

Flag qui indique que le bloc memory access a été utilisé

3.5.2.1.2 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 ?

Oui, c'est possible, mais ça serait inutile à mon avis. Si on peut déjà récupérer les données avant qu'elles soient écrites alors on n'a pas besoin dès les récupérer après.

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

On veut lire un registre où l'information est encore dans le Pipeline (Execute ou Memory Access) et qu'elle n'a pas été écrite dans les registres finaux. Il est utile lorsqu'une instruction veut accéder à une information qui a été modifiée dans les instructions précédentes et que celle-ci n'est pas encore écrite dans son registre de destination.

3.5.2.1.4 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 ?

Cela permet de diminuer la charge des aléas, étant donné qu'il ne faut plus attendre autant de temps pour pouvoir traiter une instruction dépendante. On pourrait même dire que ça les élimine totalement dans les cas extrêmes où il n'y a pas de LOAD.

3.5.2.2 Circuit execute

3.5.2.2.1 Pourquoi doit-on faire ça ?

Pour pouvoir ré-utiliser les valeurs qui sont dans le pipeline aux stages Execute et Memory Access si besoin.

3.5.2.2.2 Pourquoi doit-on faire ça pour le signal reg_mem_data_s ?

Dans le cas de LAOD qui vont lire des informations dans la mémoire de données et que l'on souhaite accéder à cette information, mais qu'elle n'est pas encore dans son registre final.

3.5.2.2.3 Que devrait-on faire si on avait un data forwarding venant du WRITE_BACK ?

Rajouter le même mécanisme de sélection comme pour les deux autres parties (MUX).

4 Test et simulation

4.1 Fetch

4.1.1 Écrire et compiler un programme simple qui utilise les instructions supportées

Ici, nous allons compiler un programme avec les fonctions suivantes (*mov*, *add* et *and*) afin de vérifier le bon fonctionnement de notre système.

```
mov r0, #5
mov r1, #7
mov r2, #9

add r0, r1, r2
and r0, r1
```

Figure 64 : Programme de test Fetch

Les quelques lignes suivantes donnent les codes d'instruction suivants dans la mémoire d'instruction.

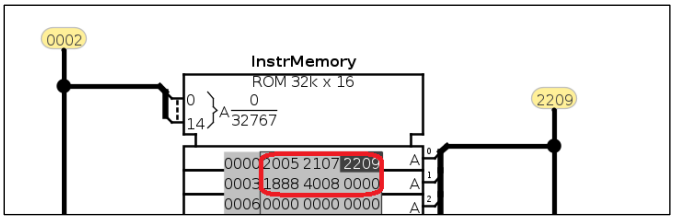

Figure 65 : Aperçu mémoire

En ajoutant une sonde sur notre bloque fetch dans le processeur_ARO2 nous pouvons vérifier qu'à chaque période, il passe de valeur en valeur. Pour exemple, ici nous somme au 3ème Tick, nous devons donc avoir la valeur « 2009 » qui doit résulter de la sortie instr_data_o du bloc fetch.

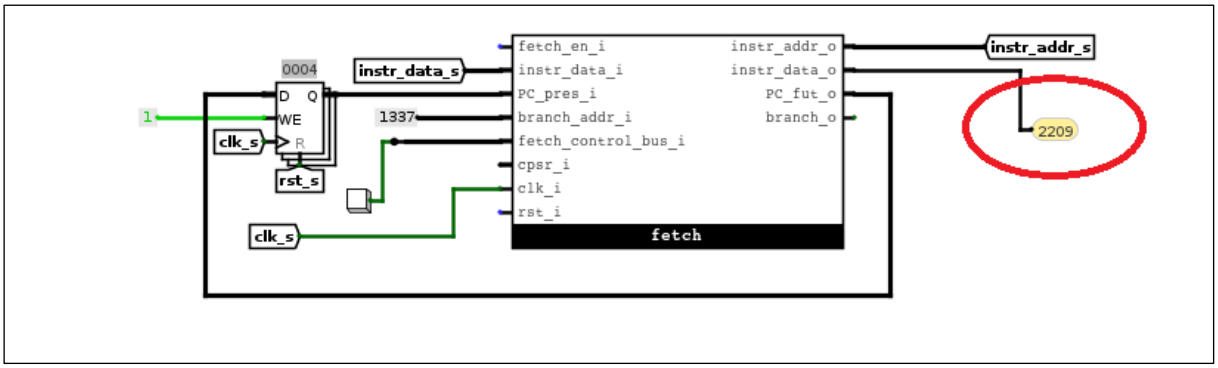

Figure 66 : Résultat dans le circuit

Voici le résultat du chronogramme correspondant à nos instructions. Nous pouvons facilement comprendre qu'à chaque coup d'horloge, la valeur du processeur s'incrémente selon les valeurs chargées dans la mémoire d'instruction. On peut en déduire que la séquence se passe dans le bon ordre et selon les fonctions voulues.

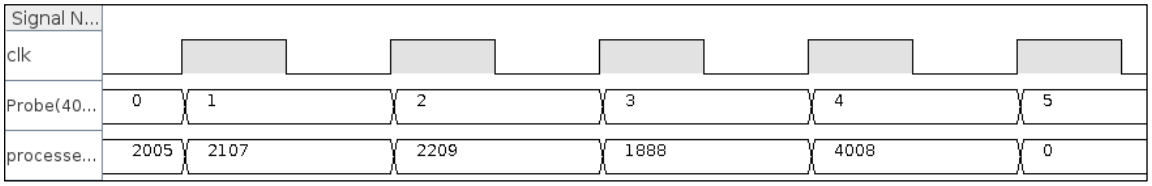

Figure 67 : Chronogramme du programme Fetch

4.1.2 Écrire un programme avec des sauts inconditionnels

On va faire le test de nos sauts inconditionnels avec le programme suivant.

```
ADR_debut_prog :
mov r0, #5
mov r1, #2
B ADR_saut_1

.org 0x40
ADR_saut_1 :
mov r2, #6
add r0, r1, r2
B ADR_saut_2

.org 0x60
ADR_saut_2 :
mov r0, #4
B ADR_debut_prog
```

Figure 68 : Programme de test pour les sauts inconditionnels

Le résultat dans la mémoire d'instruction est représenté comme tel.

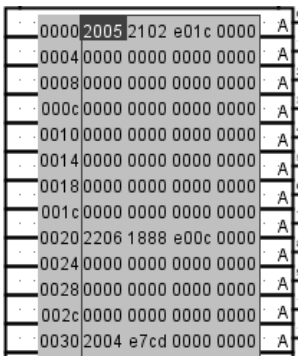

Figure 69 : Aperçu mémoire

On peut voir grâce au chronogramme ci-dessous que le saut s'effectue bien lorsque l'on atteint celui-ci.

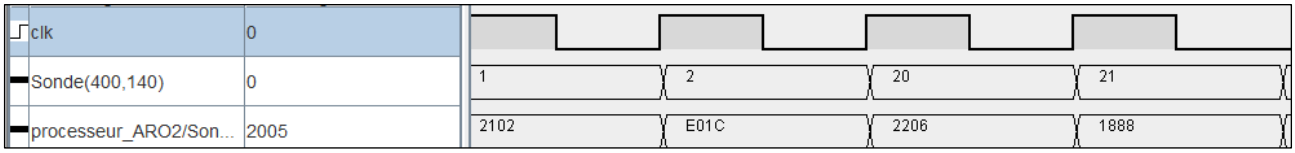

Figure 70 : Chronogramme programme de sauts inconditionnels

4.1.3 Programme à rendre pour la validation

Comme vu ci-dessus toutes les fonctionnalités de notre bloc Fetch fonctionnent, il ne manque plus qu'à tester les sauts conditionnels. On a donc produit le code suivant pour vérifier que les sauts conditionnels fonctionnent.

```
ADR_debut_prog :
mov r0, #5
mov r1, #2
add r2, r0, r1
add r3, r2, #4
B ADR_saut_1

.org 0x40
ADR_saut_1 :
mov r2, #6
sub r3, r2, #1
cmp r3, #5
BEQ ADR_saut_2

.org 0x60
ADR_saut_2 :
mov r0, #4
mov r1, #128
B ADR_debut_prog
```

Figure 71 : programme de test pour la validation du bloc Fetch

C'est surtout les instructions du milieu qui vont nous intéresser pour vérifier que tout fonctionne. On obtient d'ailleurs les instructions suivantes dans la mémoire d'instruction.

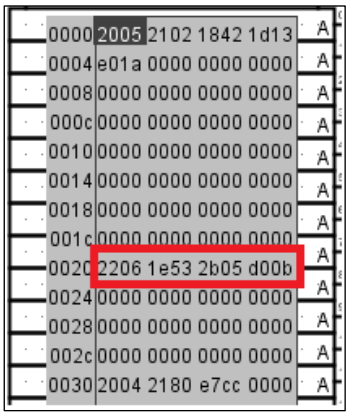

Figure 72 : Aperçu mémoire des sauts

On va réaliser deux chronogrammes le premier quand le CPSR sera à 1 et c'est à ce moment-là que le saut aura lieu. Et un second où le CPSR sera à 0 et on pourra apercevoir à ce moment-là que le saut ne s'effectue pas et qu'on continue simplement à incrémenter le PC.

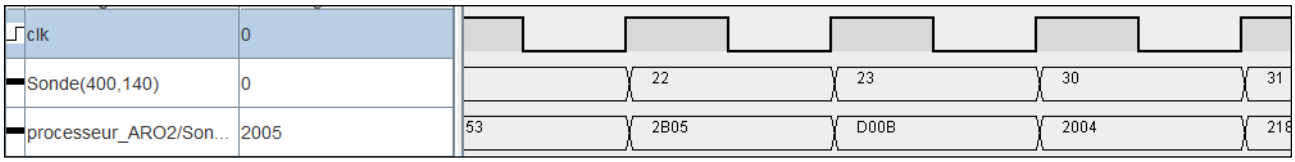

Figure 73 : Chronogramme avec saut qui s'exécute

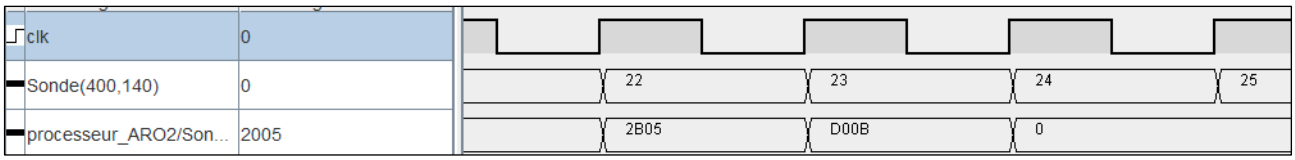

Figure 74 : Chronogramme sans le saut qui s'exécute

4.2 Decode

Pour la phase de test du circuit Decode, on va vérifier que les instructions soient correctement décodées. C'est-à-dire que l'on envoie bien les bonnes informations aux bons registres. Pour le test, on utilisera le programme ci-dessous qui est plutôt complet.

```
debut:
mov r0, #5
mov r1, #2
mov r2, #4

add r0, r1, r2
sub r3, r0, r2

strh r3, [r3, r1]
ldrh r1,[r3,r1]

BEQ ADR_saut_cond_vide
B debut

ADR_saut_cond_vide:
```

Figure 75 : Programme de test Decode

La première partie du programme contient des déplacements de valeurs, ce qui nous permet de vérifier que les instructions mélangeant registres et valeurs immédiates fonctionnent. Le plus important est de vérifier que c'est le bon registre de destination et la bonne valeur.

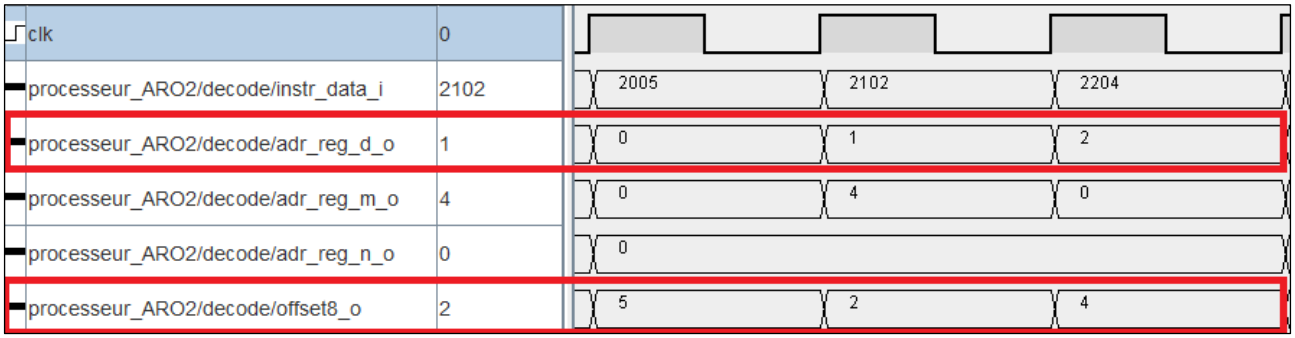

Figure 76 : Chronogramme décodage MOV

Pour la seconde partie, celle-ci s'occupe de vérifier que les opérations avec de multiples registres sont correctement décodées comme on peut le voir ci-dessous.

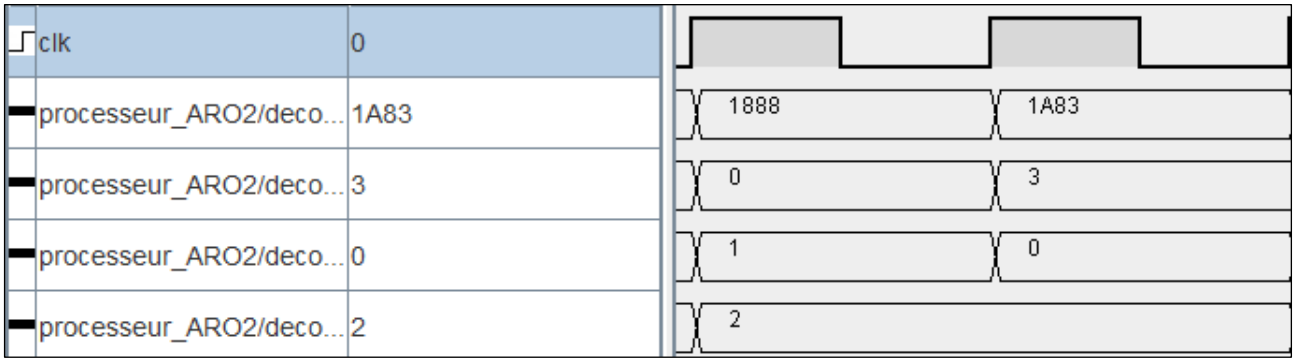

Figure 77 : Chronogramme décodage ADD/SUB

La troisième partie se charge de l'accès à la mémoire de donnée. Comme on peut le voir, ce sont bien les bons registres de sélectionner. Le bus de contrôle de la mémoire indique bien les bonnes valeurs, c'est-à-dire 3 lorsque l'on effectue un STRH (écriture mémoire) pour dire que l'on accède à la mémoire et que l'on écrite et il affiche 1 quand on fait un LDRH ce qui indique que l'on accède simplement à la mémoire.

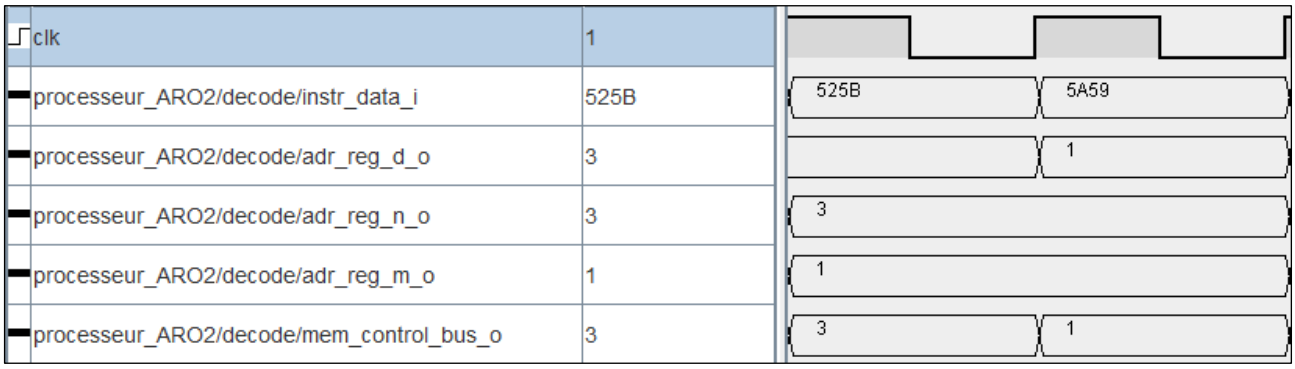

Figure 78 : Chronogramme décodage accès mémoire

La dernière partie permet de vérifier que les informations relatives aux sauts sont correctement sélectionnées. Le plus important est de vérifier que le bus de contrôle fetch ait bien les bonnes informations. Dans ce cas, on remarque que dans le cas du premier saut (conditionnel) les 2 bits indiquant que l'instruction est un saut et qu'il est conditionnel sont activés. Pour le second saut (inconditionnel), on remarque que le seul le premier bit indiquant que c'est un saut est activé.

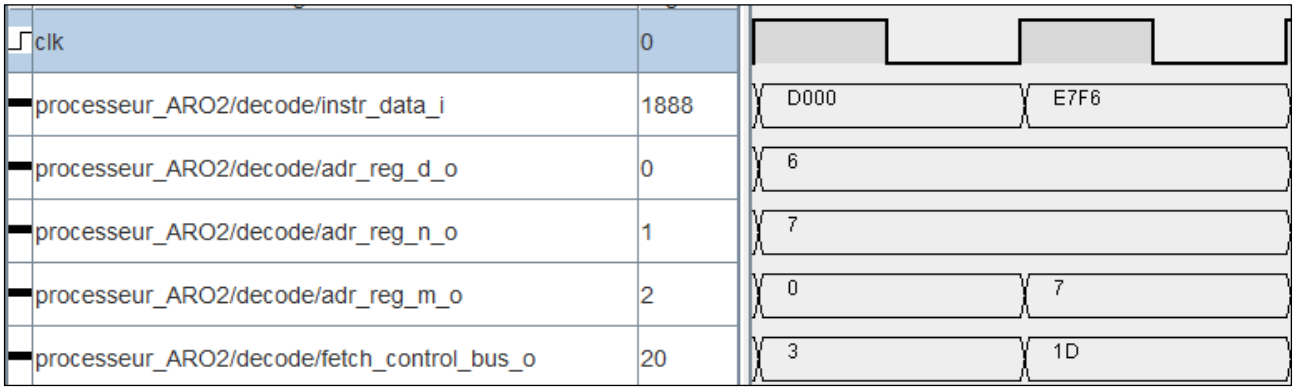

Figure 79 : Chronogramme décodage instruction de saut

Avec ces quelques instructions, on peut déterminer que le bloc Decode est totalement fonctionnel et que les instructions seront correctement envoyées au bloc Execute.

4.3 Execute et Memory Access

4.3.1.1 Execute

On va valider l'Execute indépendamment du Memory Access, puis on groupera les deux ensembles. Donc pour valider le Execute étant donné que les opérations réalisées sont plutôt simples. Nous avons mis en place un petit programme de quelques instructions où on peut voir les fonctionnalités du bloc se faire.

```
mov r0, #0
mov r1, #1
mov r2, #1


ROR r2, r1

add r2, r2, r2

neg r1, r1
```

Figure 80 : Programme de test Execute

Dans ce petit programme, on va essentiellement tester quelques opérations du bloc Execute tout en testant bien que la partie du CPSR fonctionne bien. On réalise donc dans un premier temps, un shift rotatif à droite pour avoir la valeur 0x8000 et cela nous permet de tester le bit qui indique un résultat négatif. Ensuite, on peut additionner cette valeur avec elle-même pour tester le carry, l'overflow et le zéro.

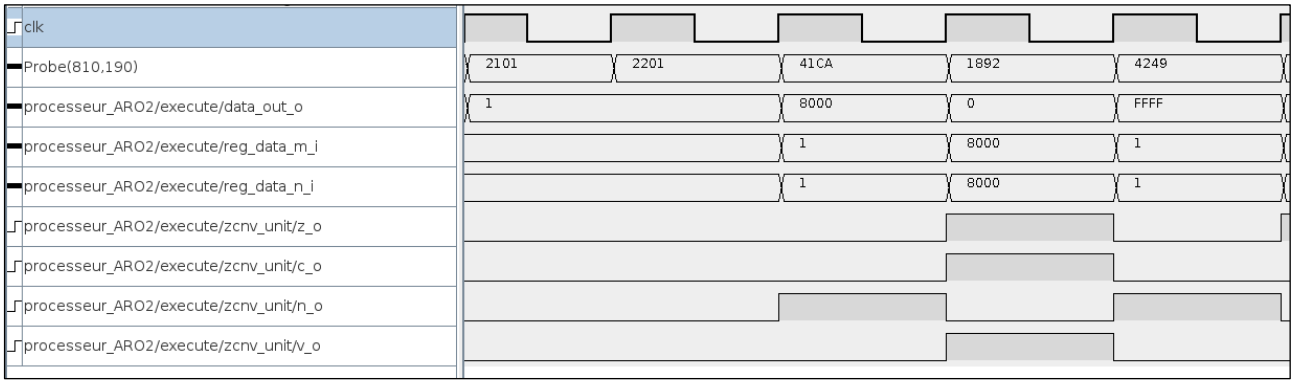

Figure 81 : Chronogramme Execute

4.3.1.2 Memory Access

Pour cette partie, on va tester 4 instructions principales qui sont listées dans le programme ci-dessous.

```
mov r0, #1
mov r1, #2
mov r2, #3
mov r3, #31

strb r3, [r0, r2]
ldrb r4, [r0, r2]

strh r2, [r3, #10]
ldrh r4, [r3, #10]
```

Figure 82 : Programme de test Memory Access

La première des instructions touchant à la mémoire permet de stocker la valeur des bits 0 à 7 (un Byte) du troisième registre dans la mémoire de données à l'adresse fournie par l'addition des deux autres registres.

La seconde instruction quant à elle va lire la donnée à cette même case de la mémoire de données et la stocker dans le quatrième registre au prochain coup d'horloge.

On peut accessoirement voir les 4 derniers paramètres évolués en fonction de si l'action à effectuer est une lecture ou une écriture.

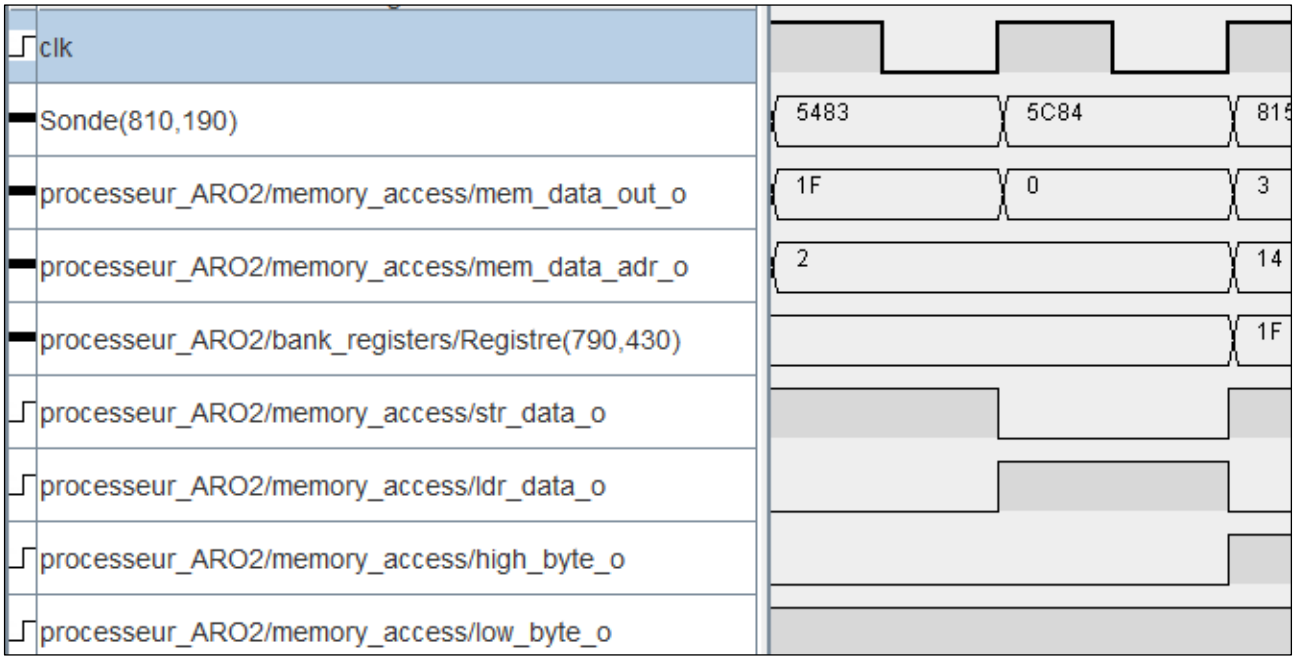

Figure 83 : Chronogramme accès mémoire par Byte

La deuxième paire d'instructions est assez similaire à celle-ci dessus. Elle permet juste de démontrer que les instructions « halfword » fonctionnent aussi. Mais le principe est le même.

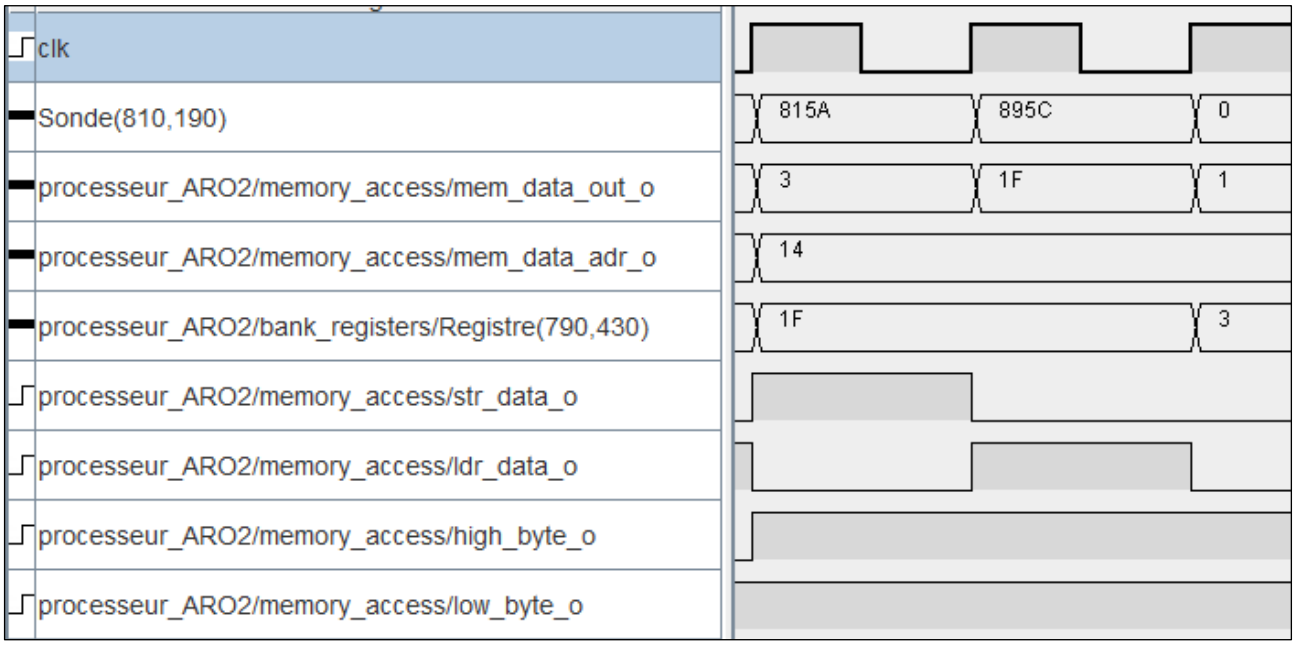

Figure 84 : Chronogramme accès mémoire par "Halfword"

Les quelques instructions réalisées ci-dessus remplissent la mémoire de données « Low » de la manière suivante. On retrouve bien nos deux valeurs stockées aux adresses calculées.

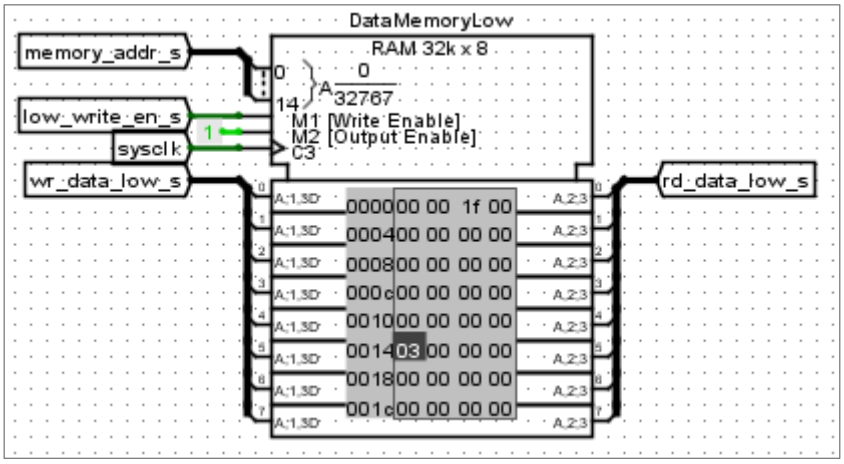

Figure 85 : Aperçu de la mémoire de données

4.3.1.3 Pile

Pour réaliser la pile, il nous faut les quelques lignes présentées ci-dessous. On stocke dans la mémoire de données à l'adresse courante du SP la valeur du LR. On peut ensuite incrémenter notre SP et finalement on réalise notre saut. Ce principe est à répéter autant de fois que l'on souhaite faire des sauts imbriqués. Lorsque l'on arrive au dernier saut, il ne faut plus que déplacer le contenu du LR dans le SP pour revenir au saut précédent. Puis pour dépiler, il ne faut plus que faire les opérations inverses c'est-à-dire décrémenter et lire la valeur contenue dans le mémoire de données.

```
00000000 <debut>:
   0:   2101        movs    r1, #1
   2:   2202        movs    r2, #2
   4:   2303        movs    r3, #3
   6:   2404        movs    r4, #4
   8:   2500        movs    r5, #0
   a:   f000 f801   bl  10 <saut_1>
   e:   e030        b.n 72 <saut_incond>

00000010 <saut_1>:
  10:   802e        strh    r6, [r5, #0]
  12:   3502        adds    r5, #2
  14:   f000 f802   bl  1c <saut_2>
  18:   3d02        subs    r5, #2
  1a:   882f        ldrh    r7, [r5, #0]
```

Figure 86 : Programme de la pile avec saut

Comme on peut le voir sur le chronogramme représentant l'écriture ci-dessous, on écrit dans la mémoire de données l'adresse de retour après le saut.

Figure 87 : Chronogramme écriture des données dans la pile

Ici, on retrouve le chronogramme lors de l'opération de lecture de la mémoire de données. On peut y voir la lecture retour du premier saut. On repose donc la valeur que l'on avait stockée dans le chronogramme ci-dessus dans le PC.

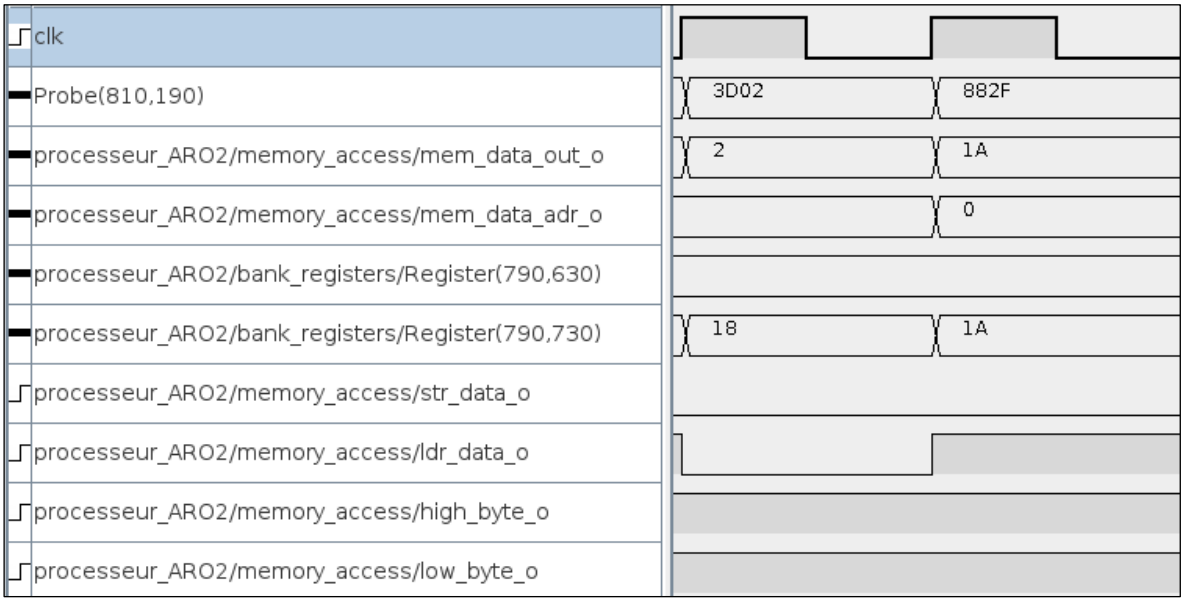

Figure 88 : Chronogramme lecture des données depuis la pile

On voit ci-dessous un extrait de la mémoire de données du programme fourni plutôt.

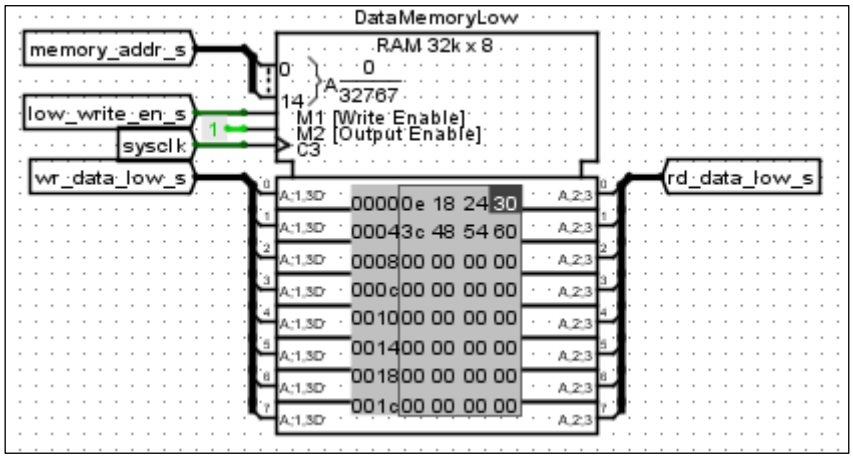

Figure 89 : Aperçu de la mémoire de données

4.4 Pipeline

4.4.1 Analyse et test du processeur

4.4.1.1 Test du processeur

Relevé du chronogramme :

Dans le chronogramme ci-dessous, nous pouvons observer l'exécution du programme compilé. L'instruction 1c84 correspond au add r4, r0, #2. Afin de lire correctement chaque exécution, nous devons décalé de 1 sur la droite à chaque fois que l'on passe d'un bloque à l'autre. De ce fait, en partant de l'instruction fetch, si nous décalons de 4 (fetch, decode, execute et memory access) on arrive sur la valeur 40, qui correspond bien à r0 + 2 soit 0x3E + 2.

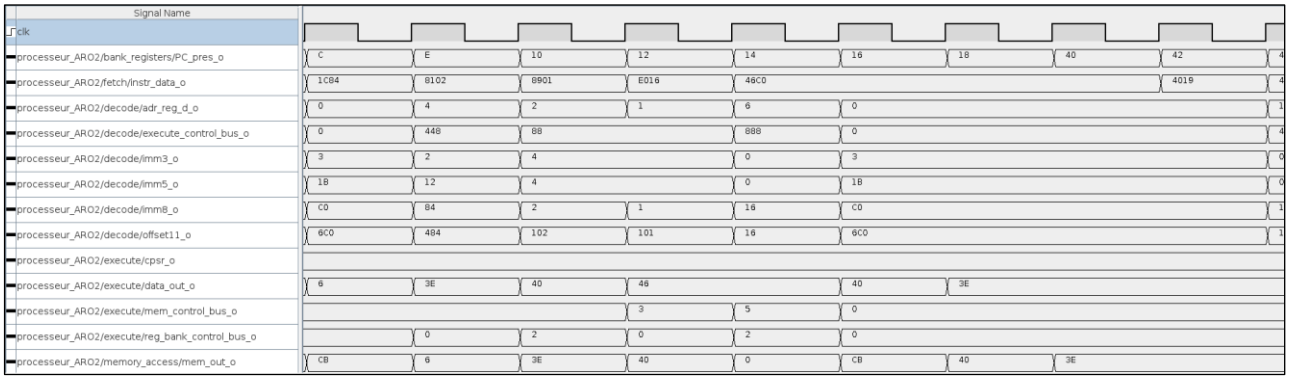

Figure 90 : Chronogramme code à tester

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

Oui, le programme s'exécute parfaitement

Combien de cycles sont nécessaires pour exécuter ce programme ?

Afin de pouvoir exécuter correctement ce programme, il est nécessaire d'avoir 18 coups d'horloge.

4.4.1.2 Test aléas de donnée

Le code ci-dessous nous permets de tester les aléas de donnée. On retrouver une dépendance RAW qui génère un aléa de donnée lors de l'addition. On essaye d'accéder au registre R1 R0 alors que ceux-ci ne sont pas encore écrit dans la banque des registres. On peut constater dans le chronogramme un arrêt de pipeline de 3 cycles ce qui est correct.

```
MAIN_START:
MOV r0, #1
MOV r1, #2
MOV r2, #6
ADD r2, r1, r0
MOV r3, #14
B PART_2

.org 0x40
PART_2:
MOV r0, #3
MOV r1, #4
MOV r2, #8
BL SAUTC

.org 0xF0
SAUTC:
MOV r0, #0
BEQ MAIN_START
```

Figure 91 : Code de test avec des aléas de donnée

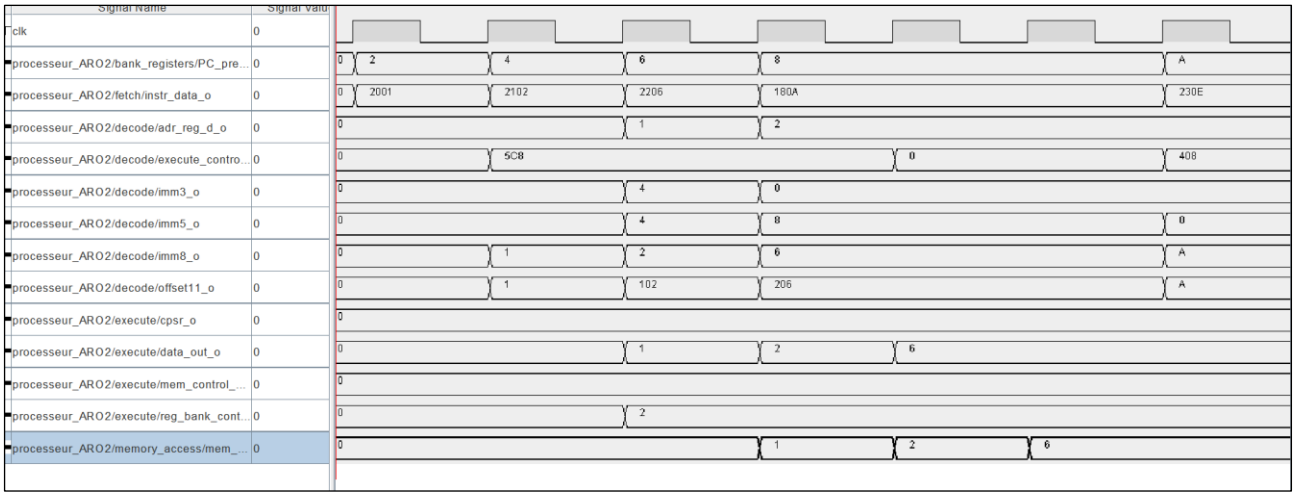

Figure 92 : Chronogramme de test d'aléas de donnée

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

Oui toutes les valeurs sont correctes

4.4.1.2.2 Pourquoi l'instruction BL génère un aléa de contrôle et un aléa de donnée ?

BL est composé de deux sous-instructions avec le msb et le lsb. Msb écrit dans le LR et lsb ira lire dans ce même registre, ce qui génère un aléa de données. Pour l'aléa de contrôle c'est simplement que BL est un saut.

4.4.1.2.3 Combien de cycles sont nécessaires pour résoudre les aléas de l'instruction BL?

8 coups de clock sont nécessaires pour exécuter un BL.

4.4.1.2.4 Quel est l'IPC pour votre programme ?

IPC = 11/31 = 0.41 instruction(s)/cycle(s)

4.4.1.3 Test : pipeline forwarding

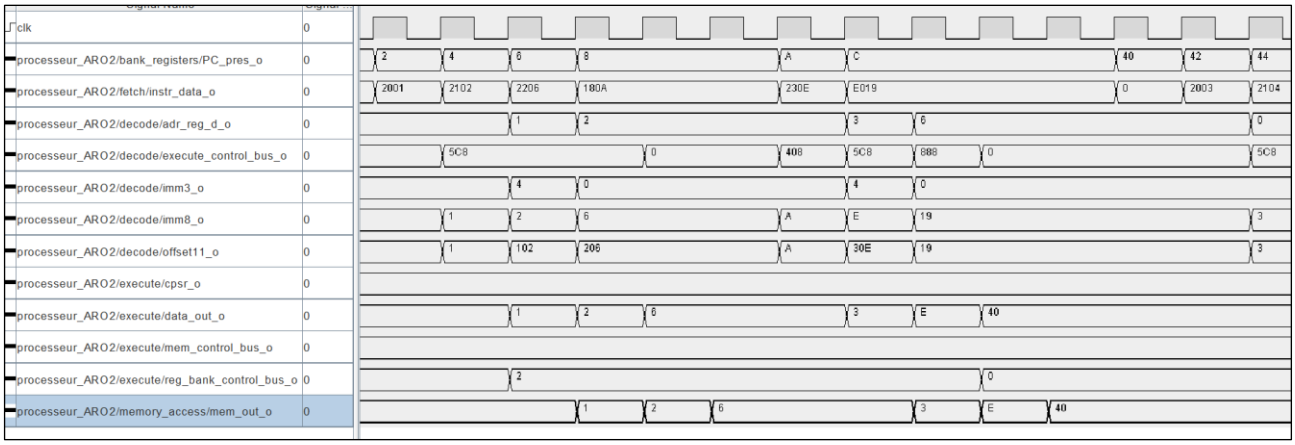

Figure 93 : Chronogramme test du pipeline forwarding

4.4.1.3.1 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 ?

Non, le processeur fournit ne fonctionne pas correctement. Le forwarding ne s'exécute pas comme il le devrait.

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

IPC = 13/31 = 0.41 instr/cycle

Débit = 0.41 * 4000 = 1677 inst/secs

Latence = 1/Débit = 1/1793 = 0.59 ms

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

Il faudra 8 cycles pour que l'instruction BL soit complétée.

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

A priori, on pourrait empiler plusieurs mêmes architectures pour qu'elles fonctionnent en parallèles.

5 Conclusion

5.1 Conclusion générale

Toutes les fonctionnalités demandées ont pu être réalisées et documentées. Nous n'avons pas eu de difficultés spécifiques durant la réalisation de ce laboratoire

5.2 Conclusion personnelle

5.2.1 Alexis

Ce que j'ai apprécié durant le laboratoire c'est le complément que celui-ci apporte à la théorie. Les concepts théoriques n'étaient pas toujours évidents à comprendre en classe, mais le fait de pouvoir les essayer et les pratiquer en laboratoire a réellement servi de renforcement. J'ai aussi bien aimé le fait de faire un objet « complexe » comme un processeur, mais dont la réalisation a été assez simple par rapport à la complexité attendue.

5.2.2 Arnaud

Pour ma part ce labo n'est pas seulement un labo d'ARO. Je rejoins totalement Alexis sur le fait que le labo nous a permis de renforcer et de comprendre la matière, mais c'est aussi la première fois que nous avons un projet à mener à deux sur le long terme. On apprend donc aussi à faire en fonction de la donnée, de ce que nous comprenons, mais aussi du binôme avec qui nous sommes. Faire ce rapport, prouve plus que tout que sans échange et sans réflexion commune, il est impossible d'être les deux en accord avec le rapport et le labo.

Date : 06.06.2022

Noms des étudiants : Arnaud Monition et Alexis Martins

6 Annexes

6.1 Table des illustrations