

SE201 : Projet 1

Florian Tarazona, Isaïe Muron, Lucie Molinié

Partie 1 - Jeu d'instruction RISC-V

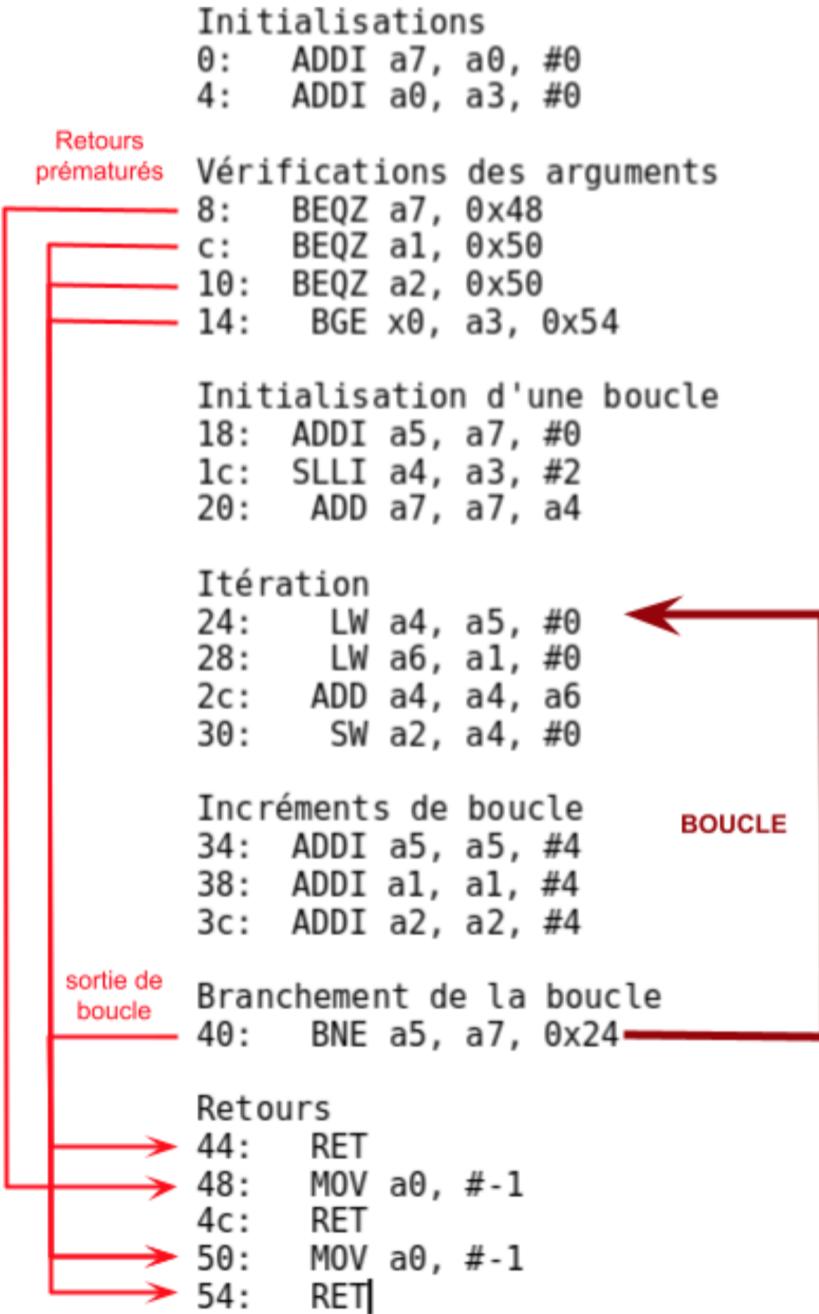
Le programme

Pour traduire ce programme : - On traduit d'abord les instructions hexadécimales en binaires - On identifie le format des instructions et leur mnémonique à l'aide de la documentation *RISC-V* (p.130) et de la dernière page du sujet

On obtient le résultat suivant :

```
0: 0000 0000 0000 0101 0000 1000 1001 0011 ADDI a7, a0, #0 (I)
4: 0000 0000 0000 0110 1000 0101 0001 0011 ADDI a0, a3, #0 (I)
8: 0000 0100 0000 1000 1000 0000 0110 0011 BEQ a7, x0, #64 (SB)
c: 0000 0100 0000 0101 1000 0010 0110 0011 BEQ a1, x0, #68 (SB)
10: 0000 0100 0000 0110 0000 0000 0110 0011 BEQ a2, x0, #64 (SB)
14: 0000 0100 1101 0000 0101 0000 0110 0011 BGE x0, a3, #64 (SB)
18: 0000 0000 0000 1000 1000 0111 1001 0011 ADDI a5, a7, #0 (I)
1c: 0000 0000 0010 0110 1001 0111 0001 0011 SLLI a4, a3, #2 (I)
20: 0000 0000 1110 1000 1000 1000 1011 0011 ADD a7, a7, a4 (R)
24: 0000 0000 0000 0111 1010 0111 0000 0011 LW a4, a5, #0 (I)
28: 0000 0000 0000 0101 1010 1000 0000 0011 LW a6, a1, #0 (I)
2c: 0000 0001 0000 0111 0000 0111 0011 0011 ADD a4, a4, a6 (R)
30: 0000 0000 1110 0110 0010 0000 0010 0011 SW a2, a4, #0 (S)
34: 0000 0000 0100 0111 1000 0111 1001 0011 ADDI a5, a5, #4 (I)
38: 0000 0000 0100 0101 1000 0101 1001 0011 ADDI a1, a1, #4 (I)
3c: 0000 0000 0100 0110 0000 0110 0001 0011 ADDI a2, a2, #4 (I)
40: 1111 1111 0001 0111 1001 0010 1110 0011 BNE a5, a7, #-28 (SB)
44: 0000 0000 0000 0000 1000 0000 0110 0111 JALR x0, x1, #0 (I)
48: 1111 1111 1111 0000 0000 0101 0001 0011 ADDI a0, x0, #-1 (I)
4c: 0000 0000 0000 0000 1000 0000 0110 0111 JALR x0, x1, #0 (I)
50: 1111 1111 1111 0000 0000 0101 0001 0011 ADDI a0, x0, #-1 (I)
54: 0000 0000 0000 0000 1000 0000 0110 0111 JALR x0, x1, #0 (I)
```

On peut organiser le code en plusieurs parties et l'annoter pour mettre en avant les branchements :



Les incrémentations multiples de 4 pour **a1**, **a2** et **a5** suggèrent que ces registres contiennent des adresses. Cela est confirmé par leurs utilisations dans les instructions d'accès mémoire.

Le corps de la boucle charge deux valeurs, les additionne puis stocke le résultats dans une autre partie de la mémoire : **la fonction est un additionneur vectoriel**. Donnons finalement l'usage de la fonction :

```
int add_vector(sc1_array, sc2_array, dst_array, size);
```

- **sc1_array**, **sc2_array** sont les deux vecteurs à additionner.
- **dst_array** est le vecteur dans lequel on met le résultat.

- `size` est la taille des vecteurs

La fonction renvoie `-1` si l'un des vecteurs `dst`, `sc1`, `sc2` a une adresse nulle, `size` sinon.

Les Branch Delay Slots

Dans tous les processeurs dont l'architecture est basée sur un pipeline, les instructions de branchement impliquent de casser le pipeline.

En effet, prenons l'exemple d'un processeur RISC-V, basé sur un pipeline à 5 étages comme vu en cours, qui exécute l'instruction

```
bge a0, a1, #20
```

Supposons que $a_0 < a_1$. Le processeur ne pourra s'en rendre compte qu'à l'étage d'**EXÉCUTION** du pipeline. À ce moment, l'instruction suivante dans la mémoire sera déjà dans l'étage de **DÉCODAGE**. Au coup d'horloge suivant, cette dernière, alors qu'elle vient d'être décodée, sera tuée par le processeur. Cela fait perdre au processeur un coup d'horloge.

Dans des architectures assez anciennes comme *MIPS*, la technique des branch delay slots était employée. Elle consiste simplement à exécuter l'instruction suivant une instruction de branchement. **C'était au programmeur de prêter attention à ce qu'il faisait.** Il pouvait toutefois profiter de cette instruction afin de prévoir par exemple une instruction pour aller chercher une donnée. Il pouvait concevoir des optimisations très fines, mais le code devenait moins lisible et les compilateurs plus difficiles à optimiser.

RISC-V, au contraire, prend le parti de ne pas nécessairement exécuter une instruction suivant un branchement. Cela permet au programmeur de gérer son code de manière plus naturelle. Toutefois, il est possible d'implémenter au sein du processeur un système de prédition de branchement qui permettra de tenter de charger directement la bonne instruction.

Partie 2 - Outils de compilation RISC-V

Voici un programme d'addition vectorielle en C :

```
1 int add_vector(const int *v1, const int *v2, int *vtot, const int size) {
2     if(!v1)          { return -1; }
3     if(!v2)          { return -1; }
4     if(!vtot)        { return -1; }
5     for(int i = 0; i < size; i++)
6         vtot[i] = v1[i] + v2[i];
7     return size;
8 }
```

Compilation avec -O0

Une première compilation avec **gcc sans optimisation** donne le code assembleur suivant :

```
1 add_vector-00.o:      file format elf32-littleriscv
2
3
4 Disassembly of section .text:
5
6 00000000 <add_vector>:
7   0: fd010113          addi    sp,sp,-48
8   4: 02812623          sw      s0,44(sp)
9   8: 03010413          addi    s0,sp,48
10  c: fca42e23          sw      a0,-36($0)
11 10: fcb42c23          sw      a1,-40($0)
12 14: fcc42a23          sw      a2,-44($0)
13 18: fcd42823          sw      a3,-48($0)
14 1c: fdc42783          lw      a5,-36($0)
15 20: 00078a63          beqz   a5,34 <.L2>
16 24: fd842783          lw      a5,-40($0)
17 28: 00078663          beqz   a5,34 <.L2>
18 2c: fd442783          lw      a5,-44($0)
19 30: 00079663          bnez   a5,3c <.L3>
20
21 00000034 <.L2>:
22 34: fff00793          li      a5,-1
23 38: 0680006f          j       a0 <.L4>
24
25 0000003c <.L3>:
26 3c: fe042623          sw      zero,-20($0)
27 40: 0500006f          j       90 <.L5>
28
29 00000044 <.L6>:
30 44: fec42783          lw      a5,-20($0)
31 48: 00279793          slli   a5,a5,0x2
32 4c: fdc42703          lw      a4,-36($0)
33 50: 00f707b3          add    a5,a4,a5
34 54: 0007a683          lw      a3,0(a5)
35 58: fec42783          lw      a5,-20($0)
36 5c: 00279793          slli   a5,a5,0x2
37 60: fd842703          lw      a4,-40($0)
38 64: 00f707b3          add    a5,a4,a5
39 68: 0007a703          lw      a4,0(a5)
40 6c: fec42783          lw      a5,-20($0)
41 70: 00279793          slli   a5,a5,0x2
42 74: fd442603          lw      a2,-44($0)
43 78: 00f607b3          add    a5,a2,a5
44 7c: 00e68733          add    a4,a3,a4
45 80: 00e7a023          sw      a4,0(a5)
46 84: fec42783          lw      a5,-20($0)
47 88: 00178793          addi   a5,a5,1
48 8c: fef42623          sw      a5,-20($0)
```

```

49
50 00000090 <.L5>:
51 90: fec42703           lw      a4,-20($0)
52 94: fd042783           lw      a5,-48($0)
53 98:faf746e3            blt    a4,a5,44 <.L6>
54
55 0000009c <.LBE2>:
56 9c: fd042783           lw      a5,-48($0)
57
58 000000a0 <.L4>:
59 a0: 00078513            mv     a0,a5
60 a4: 02c12403            lw      $0,44($p)
61 a8: 03010113            addi   sp,sp,48
62 ac: 00008067            ret

```

Nous pouvons encore distinguer plusieurs parties :

- l'**initialisation de la stack**
- les vérifications et retours prématurés : **0x20, 0x28, 0x30, .L2, .L3**
- le retour de la fonction : **.L4**

Nous pouvons constater plusieurs points qui diffèrent de la première version présentée :

- L'utilisation d'instructions d'accès mémoire est systématique pour la lecture et l'écriture des variables. On peut par exemple reconnaître la suite d'instructions implémentant l'accès en lecture $v[i]$. *Pour l'accès en écriture, il suffit de remplacer le dernier LW par SW.*

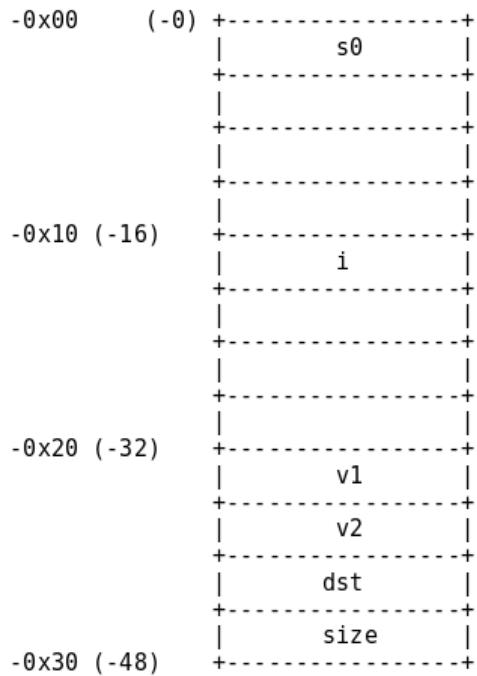
```

lw    a5, -20($0)      //Chargement de i dans a5
                      //depuis la stack
slli a5, a5, 0x2       //Multiplication de i pour être
                      //compatible avec une adresse
lw    a4, -36($0)       //Chargement de v dans a4
                      //depuis la stack
add  a5, a4, a5        //Calcul de l'adresse v + i
lw    a3, 0(a5)         //Chargement de la valeur v[i]

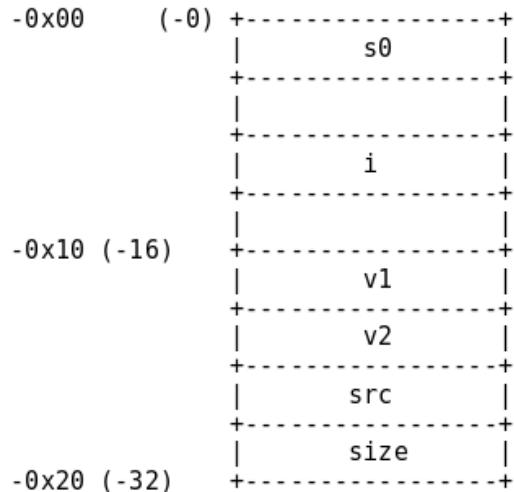
```

- L'utilisation de la pile, alors que la première version se contentait de travailler avec les registres de travail a0 - a7.

Une remarque surprenante est que la stack est initialisée avec une taille nettement supérieure aux besoins de la fonction : 12 mots-mémoire, mais seulement 6 utilisés. On peut représenter la stack ainsi :



Pour des raisons d'optimisation, gcc essaie d'aligner les éléments de la stack, comme l'indique ce thread sur GitHub. On peut modifier cela en utilisant l'argument `-mpreferred-stack-boundary=3`. On obtient alors :



- La gestion des branchements est également différente : on constate la présence d'un préambule **.L2** menant directement à **.L4** après avoir mis la valeur de retour à -1.

Compilation avec -O3

```
1 add_vector-03.o:      file format elf32-littleriscv
2
3
4 Disassembly of section .text:
5
6 00000000 <add_vector>:
7   0:  00050793          mv    a5,a0
8   4:  04050063          beqz a0,44 <.L8>
9   8:  02058e63          beqz a1,44 <.L8>
10  c:  02060c63          beqz a2,44 <.L8>
11
12 00000010 <.LBB2>:
13 10:  00269893          slli  a7,a3,0x2
14 14:  011508b3          add   a7,a0,a7
15 18:  02d05263          blez a3,3c <.L5>
16
17 0000001c <.L4>:
18 1c:  0007a703          lw    a4,0(a5)
19 20:  0005a803          lw    a6,0(a1)
20 24:  00478793          addi  a5,a5,4
21 28:  00458593          addi  a1,a1,4
22 2c:  01070733          add   a4,a4,a6
23 30:  00e62023          sw    a4,0(a2)
24 34:  00460613          addi  a2,a2,4
25 38:  ff1792e3          bne   a5,a7,1c <.L4>
26
27 0000003c <.L5>:
28 3c:  00068513          mv    a0,a3
29
30 00000040 <.LVL3>:
31 40:  00008067          ret
32
33 00000044 <.L8>:
34 44:  fff00513         li    a0,-1
35
36 00000048 <.LVL5>:
37 48:  00008067          ret
```

On retrouve un code bien plus proche de la première version. On constate cependant quelques différences :

- Certaines instructions ne sont pas présentes, en particulier au début de la fonction ou à la fin. La structure est conservée néanmoins.
- Les *incrément d'adresses* sont effectués **de manière plus rapprochée** de l'*instruction d'accès mémoire* qui utilise l'adresse.

On peut essayer d'expliquer cela comme une optimisation pour éviter de se retrouver confronté à une obligation de **stall** les instructions suivant directement les accès mémoire. En effet, dans le premier code, l'instruction **0x2C** devait être retardée car elle utilisait **a6** directement après sa lecture depuis la mémoire.

On avait un potentiel problème également avec l'instruction **0x30** qui utilisait **a4** directement après y avoir stocké un résultat de l'ALU. On peut éviter de retarder l'instruction en utilisant une technique de **data-forwarding**, en permettant à l'étage d'**EXÉCUTION** du processeurs de prendre en entrée la valeur calculée au coup d'horloge précédent.

Ici, les adresses peuvent être incrémentées sans attendre, et ces instructions permettent d'attendre la ressource en faisant quelque chose d'utille.

Partie 3 - Architecture RISC-V

Flot d'exécution

L'exécution pas à pas de la fonction avec les paramètres (0x200, 0x200, 0x200, 0x2) donne :

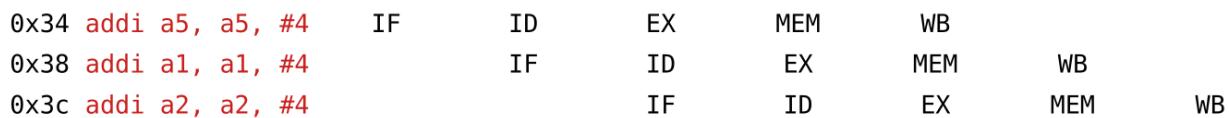
PC	Instruction	a0	a1	a2	a3	a4	a5	a6	a7	Explication
INIT		0x200	0x200	0x200	0x2	0x0	0x0	0x0	0x0	
0x00	addi a7, a0, #0	0x200	0x200	0x200	0x2	0x0	0x0	0x0	0x200	copie la valeur de a0 dans a7
0x04	addi a0, a3, #0	0x2	0x200	0x200	0x2	0x0	0x0	0x0	0x200	copie la valeur de a3 dans a0
0x08	beqz a7, 0x48	0x2	0x200	0x200	0x2	0x0	0x0	0x0	0x200	vérifie que le 1er argument (adresse du 1er vecteur) n'est pas null
0x0c	beqz a1, 0x50	0x2	0x200	0x200	0x2	0x0	0x0	0x0	0x200	vérifie que le 2e argument (adresse du 2e vecteur) n'est pas null
0x10	beqz a2, 0x50	0x2	0x200	0x200	0x2	0x0	0x0	0x0	0x200	vérifie que le 3e argument (adresse du vecteur de destination) n'est pas null
0x14	bge x0, a3, 0x54	0x2	0x200	0x200	0x2	0x0	0x0	0x0	0x200	vérification sur la taille du vecteur ? non nulle ?
0x18	addi a5, a7, #0	0x2	0x200	0x200	0x2	0x0	0x200	0x0	0x200	copie la valeur de a7 dans a5
0x1c	slli a4, a3, #2	0x2	0x200	0x200	0x2	0x8	0x200	0x0	0x200	effectue un shift left (2bits) sur la valeur de a3, met le résultat dans a4
0x20	add a7, a7, a4	0x2	0x200	0x200	0x2	0x8	0x200	0x0	0x208	met a7+a4 dans a7
0x24	lw a4, a5, #0	0x2	0x200	0x200	0x2	0x61	0x200	0x0	0x208	charge a4 avec la valeur située à l'adresse [a5] (0x61 pour la 1e itération)
0x28	lw a6, a1, #0	0x2	0x200	0x200	0x2	0x61	0x200	0x61	0x208	charge a6 avec la valeur située à l'adresse [a1] (0x61 pour la 1e itération)
0x2c	add a4, a4, a6	0x2	0x200	0x200	0x2	0xc2	0x200	0x61	0x208	met a6+a4 dans a4
0x30	sw a2, a4, #0	0x2	0x200	0x200	0x2	0xc2	0x200	0x61	0x208	écrit a4 dans la case mémoire d'adresse a2
0x34	addi a5, a5, #4	0x2	0x200	0x200	0x2	0xc2	0x204	0x61	0x208	incrément la valeur de a5 (adresse de l'élément suivant du 1er vecteur)

0x38	addi a1, a1, #4	0x2	0x204	0x200	0x2	0xc2	0x204	0x61	0x208	incrément la valeur de a1 (adresse de l'élément suivant du 2e vecteur)
0x3c	addi a2, a2, #4	0x2	0x204	0x204	0x2	0xc2	0x204	0x61	0x208	incrément la valeur de a2 (adresse de l'élément suivant du vecteur de destination)
0x40	bne a5, a7, 0x24	0x2	0x204	0x204	0x2	0xc2	0x204	0x61	0x208	si a5 != a7 (l'adresse à laquelle on regarde est différente de la 1e adresse hors vecteur), on saute en 0x24 (nouvelle itération)
0x24'	lw a4, a5, #0	0x2	0x204	0x204	0x2	0x20	0x204	0x61	0x208	charge a4 avec la valeur située à l'adresse [a5] (0x20 pour la 2e itération)
0x28'	lw a6, a1, #0	0x2	0x204	0x204	0x2	0x20	0x204	0x20	0x208	charge a6 avec la valeur située à l'adresse [a1] (0x20 pour la 2e itération)
0x2c'	add a4, a4, a6	0x2	0x204	0x204	0x2	0x40	0x204	0x20	0x208	met a6+a4 dans a4
0x30'	sw a2, a4, #0	0x2	0x204	0x204	0x2	0x40	0x204	0x20	0x208	écrit a4 dans la case mémoire d'adresse a2
0x34'	addi a5, a5, #4	0x2	0x204	0x204	0x2	0x40	0x208	0x20	0x208	incrément la valeur de a5 (adresse de l'élément suivant du 1er vecteur)
0x38'	addi a1, a1, #4	0x2	0x204	0x204	0x2	0x40	0x208	0x20	0x208	incrément la valeur de a1 (adresse de l'élément suivant du 2e vecteur)
0x3c'	addi a2, a2, #4	0x2	0x204	0x208	0x2	0x40	0x208	0x20	0x208	incrément la valeur de a2 (adresse de l'élément suivant du vecteur de destination)
0x40	bne a5, a7, 0x24	0x2	0x204	0x208	0x2	0x40	0x208	0x20	0x208	on a a5 == a7 , on ne saute pas en 0x24
0x44	ret	0x2	0x204	0x208	0x2	0x40	0x208	0x20	0x208	La fonction retourne a0 , c'est à dire la taille des vecteurs.

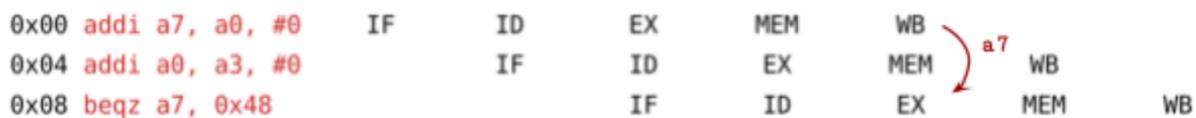
Pipelining

Afin d'illustrer les problématiques d'**aléas**, nous traçons partiellement les diagrammes de pipeline.

En temps normal (sans aléa), le diagramme est simplement le suivant :

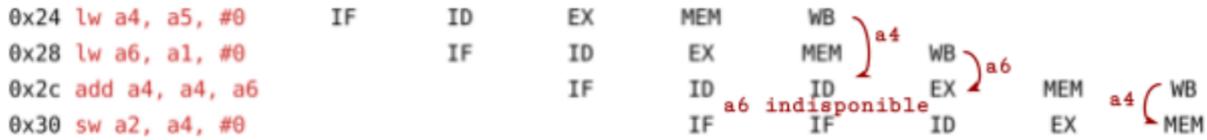


Le *premier* aléa ayant lieu dans l'exécution du programme se produit à la première vérification (instruction 0x08) :



L'instruction **beqz a7, 0x48** a besoin de la valeur de **a7**, mais celle-ci n'a pas encore été réécrite par l'instruction 0x00 qui en est encore à l'étape WRITEBACK. Cette étape peut simplement **forwarder** la valeur de **a7** à l'étape d'**EXÉCUTION**.

Un *deuxième* aléa se présente à l'instruction 0x2c :



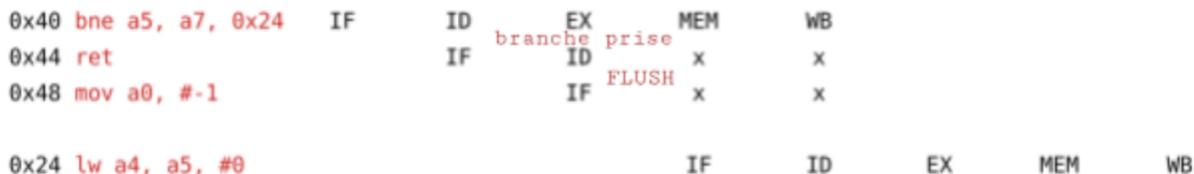
À l'étape d'EXÉCUTION, l'instruction 0x2c (add a4, a4, a6) requiert les valeurs de a4 et a6. a4 est mis à jour par l'instruction 0x24 (lw a4, a5, #0), qui a fini de lire la mémoire. L'étape WRITEBACK peut alors **forwarder** la valeur. Cependant, l'instruction 0x28 (lw a6, a1, #0) n'a pas fini de récupérer la valeur de a6 dans la mémoire.

Dans ce cas, aucun **forwarding** n'est envisageable : la pipeline est **stalled**, mise en attente jusqu'à ce que WRITEBACK puisse **forwarder** la valeur de a6.

L'instruction 0x30 (sw a2, a4, #0) requiert durant la phase d'ACCÈS MÉMOIRE la valeur de a4, qui vient d'être modifiée par l'instruction 0x2c. L'étape de WRITEBACK **forward** la valeur de a4.

Imaginons le cas où l'instruction 0x2c avait effectué un calcul sur a2. Dans ce cas, l'instruction 0x30 aurait eu besoin de cette nouvelle valeur durant l'étape d'EXÉCUTION qui se charge du calcul de l'adresse. L'étape d'ACCÈS MÉMOIRE aurait alors **forwardé** la valeur de a2 à l'étape d'EXÉCUTION.

Enfin, le cas d'un branchement pris est visible à l'instruction 0x40. C'est le *troisième* aléa :



À l'étape d'EXÉCUTION l'instruction 0x40 (bne a5, a7, 0x24) détermine qu'il faut effectuer un branchement sur l'instruction 0x24. Les deux instructions suivantes, qui viennent d'être chargées de la mémoire, sont **flushées** et l'instruction 0x24 est **fetched** (l'adresse a été calculée lors de l'étape d'EXÉCUTION par l'instruction de branchement).

Partie 4 - Processor Design

Instruction Set Architecture

Instruction Set

Dans un premier temps, nous imaginons un set d'instruction codées sur **16 bits**. La difficulté est alors de **compacter** les informations. L'instruction de **branchement conditionnelle beqz** devant avoir un immédiat de **10 bits** et un numéro de registre codé sur **4 bits**, il ne reste que **2 bits** afin de définir un **opcode** pour différencier l'instruction.

En considérant un **opcode de 2 bits**, on ne pourrait alors définir que **4 instructions**.

Pour pallier à ce problème, on définit, pour d'autres types d'instructions, **1 à 2 bits** dits "de fonction" permettant de différencier plusieurs instructions appartenant à une même catégorie.

Cependant **deux instructions de même opcode n'ont pas toujours le même format**. Ci-dessous est décrit le jeu d'instructions que nous avons imaginé :

15	12 11	8 7	4	3	2	1	0	
	rs2	rs1	rd	11	10	add	A	
	rs2	rs1	rd	10	10	nand		
	rs2	rs1	rd	01	10	sub		
	imm[7:0]		rd	00	10	mov	L	
0	imm[6:0]		rd	00	00	sht	S	
	imm[6:3]	rs1	rd	imm[2] 0	01	str	M	
	imm[6:3]	rs1	rd	imm[2] 1	01	ldr		
	imm[8:5]	rs1	imm[4:0 9]			11	bnez	B
	0000	rs1	0000	10	00	jmp	J	
	imm[8:5]	0000	imm[4:1 9]		1	00	call	C

Nous détaillons les instructions.

1 - Instructions arithmétiques - format A

Le jeu d'instructions dispose de **3 instructions artihmétiques** :

rs2	rs1	rd	11	10	add
rs2	rs1	rd	01	10	sub
rs2	rs1	rd	10	10	nand

add rd, rs1, rs2 effectue la somme de **rs1** et **rs2**, puis la stocke dans **rd**. Les opérandes sont signées.

$$rd \leftarrow rs1 + rs2$$

sub rd, rs1, rs2 effectue la différence entre **rs1** et **rs2**, puis la stocke dans **rd**. Les opérandes sont signées.

$$rd \leftarrow rs1 - rs2$$

nand rd, rs1, rs2 effectue l'opération logique and bit à bit entre **rs1** et **rs2**, puis stocke la négation de ce résultat dans **rd**.

$$rd \leftarrow \sim(rs1 \& rs2)$$

Le choix d'avoir implémenter **nand** permet de calculer toute opération logique par universalité de **nand**.

2 - Instructions de chargement - format L

La compacité du jeu d'instructions ne nous permet pas d'avoir des opérations arithmétiques avec opérandes immédiates. Afin de pallier à cela, nous implémentons une instruction **mov** :

imm[7:0]	rd	00	00	mov
----------	----	----	----	-----

mov rd, imm charge **imm** dans le registre **rd**. L'immédiat est un octet. Il est chargé dans l'octet de poids faible.

3 - Instruction de décalage - format S

L'instruction de chargement d'immédiat ne permet de stocker qu'un octet. Afin de remplir un registre avec une valeur immédiate de 32 bits, nous implémentons une instruction de décalage :

0	imm[6:0]	rd	00	00	sht
---	----------	----	----	----	-----

sht rd, imm effectue un décalage logique de la valeur du registre **rd**. L'immédiat est signé, de sorte que le décalage puisse être fait vers la gauche ou la droite.

Le stockage dans un registre d'un immédiat de 32 bit doit se faire octet par octet, en partant de l'octet de poids fort :

```
mov r0, byte0; sht r0, #24
mov r0, byte1; sht r0, #16
mov r0, byte2; sht r0, #8
mov r0, byte3
```

4 - Instructions d'interaction mémoire - format M

Afin d'accéder à la mémoire de données, le jeu d'instructions dispose d'instructions de load et store :

imm[6:3]	rs1	rd	imm[2]	1	01	ldr
imm[6:3]	rs1	rd	imm[2]	0	01	str

ldr rd, rs1, imm charge la valeur située à l'adresse **rs1 + imm** dans le registre **rd**. La valeur de **rs1** est non signée, celle de l'immédiat est signée.

```
rd ← data_memory[rs1 + imm]
```

str rd, rs1, imm charge la valeur située à l'adresse **rs1 + imm** dans le registre **rd**. La valeur de **rs1** est non signée, celle de l'immédiat est signée.

```
data_memory[rs1 + imm] ← rd
```

*La valeur de l'immédiat respecte l'alignement sur 32 bits** des valeurs en mémoire. De la même manière, les deux bits de poids faible de rs1 sont ignorés.*

5 - Instruction de branchement - format B

Nous fournissons une instruction de branchement conditionnel :

imm[8:5]	rs1	imm[4:0 9]	11	bnez
----------	-----	------------	----	------

bnez rs1, imm teste si la valeur du registre **rs1** est non nulle. Si elle l'est, elle effectue alors un saut à l'adresse **pc + imm*2**. La valeur de **rs1** et de l'immédiat sont signées.

```
if rs1 != 0
    pc ← pc + imm * 2
```

L'immédiat n'est pas aligné sur 16 bits (1 instruction). Il représente le nombre d'instructions dont on se décale par rapport à l'instruction en cours.

Si la branche est prise, l'instruction suivant le branchement est exécutée. Si cette instruction est un saut (J), un branchement (B) ou un appel (C), ses effets sont ignorés.

6 - Instruction de saut inconditionnel - format J

Le jeu d'instruction permet également d'effectuer un saut inconditionnel :

0000	rs1	0000	10	00	jmp
------	-----	------	----	----	-----

jmp rs1 charge la valeur de **rs1** dans **pc**.

```
pc ← rs1
```

Le bit de poids faible de rs1 est ignoré afin de respecter l'alignement sur 16 bits.

L'instruction suivant le saut est exécutée. Si cette instruction est un saut (J), un branchement (B) ou un appel (C), ses effets sont ignorés.

7 - Instruction d'appel de fonction (format C)

Nous fournissons une instruction simplifiant l'appel de fonctions :

imm[8:5]	0000	imm[4:1 9]	1	00	call
----------	------	------------	---	----	------

call imm enregistre la valeur de **pc** dans **lr** puis charge la valeur de l'immédiat dans **pc**. L'immédiat est signé.

```
lr ← pc
pc ← imm
```

La valeur de l'immédiat est alignée sur 16 bits.

8 - Instruction nop

Le jeu d'instructions ne propose pas d'instruction **nop** spécifique. Néanmoins nous pouvons utiliser l'instruction de décalage afin de trouver une instruction qui n'effectue aucun changement sur les registres.

Il s'agit de l'instruction nulle **0x0000** qui n'est autre que **sht r0, #0**. Cette instruction décale la valeur de **r0** de 0 bits, i.e. n'effectue aucun changement.

Les registres

Le processeur dispose de 16 registres, dont certains ont des fonctions particulières.

Un programmeur **ne devrait pas écrire directement dans lr (r14) ou pc (r15)**. Il devrait pour cela utiliser les instructions de branchement.

Lors d'un **appel de fonction**, la fonction appelante donne ses arguments à la fonction appelée dans les registres **r0 - r3**. La fonction appelée est à même de modifier tout registre de travail (**r0 - r12**). Elle doit donc sauvegarder les registres qu'elle utilise dans la stack grâce à **sp**, en particulier le **link register lr**.

La **fonction appelante** exécute enfin l'instruction **call**. Cette instruction effectue un saut inconditionnel vers l'adresse contenue dans le registre d'opérande. Elle sauvegarde également l'instruction suivante de la fonction appelante dans le **link register**.

La **fonction appelée** s'exécute et se termine par la libération de la stack qu'elle a prise puis une instruction **jmp** sur le **link register**. La fonction appelante restore ses registres depuis la stack.

Voici un tableau récapitulatif :

Registre(s)	Utilisation
r0	return value
r0 - r3	function arguments
r0 - r12	general purpose registers
r13 (sp)	stack pointer
r14 (lr)	link register
r15 (pc)	program counter

Application

Nous pouvons à l'aide de nos nouvelles instructions réécrire la fonction d'addition vectorielle. Nous rappelons la définition de ses arguments :

- **r0** : Adresse du premier vecteur
- **r1** : Adresse du deuxième vecteur
- **r2** : Adresse du vecteur somme
- **r3** : Taille du vecteur

Nous utilisons les *branch delay slots* en plaçant une instruction à la suite de chaque instruction de branchement. Cette instruction sera toujours exécutée et permet de réduire l'impact qu'ont les sauts d'une exécution "normale". En effet, comme nous ne disposons que d'une instruction de branchement conditionnelle, une exécution avec des paramètres valides produira parfois des sauts.

```
define_constants:  
0x00      sht r4, #32      //r4 ← 0  
0x02      nand r5, r4, r4 //r5 ← -1  
0x04      sht r6, #32  
0x06      mov r6, #4        //r6 ← 4  
  
check_addresses:  
//On intercale le transfert de r3  
0x08      sht r12, #32  
0x0a      mov r12, #40 <invalid_ret>  
0x0c      bnez r0, #3  
0x0e      add r7, r4, r0
```

```

0x10      jmp r12 <invalid_ret>
0x12      bnez r1, #3
0x14      add r0, r4, r3
0x16      jmp r12 <invalid_ret>
0x18      bnez r2, #3
0x1a      mov r8, 0x80          //On commence à créer un masque 0x80000000
0x1c      jmp r12 <invalid_ret>

check_size:
0x1e      bnez r3, #3
0x20      mov r12, #0x3e
0x22      jmp r12 <ret>
0x24      sht r8, #24
0x26      nand r9, r3, r8
0x28      nand r9, r9, r9
0x2a      bnez r9, #14 <ret>

loop:
//Nous n'avons plus besoin de r8 et r9, nous nous en
//servons pour récupérer les valeurs de la mémoire
0x2c      ldr r8, r7, #0
0x2e      add r7, r7, r6      // r7 (v1) += 4
0x30      ldr r9, r1, #0
0x32      add r1, r1, r6      //r1 (v2) += 4
0x34      add r8, r8, r9
0x36      str r8, r2, #0
0x38      add r3, r3, r5      //Décrément du compteur
0x3a      bnez r3, #-7 <loop>
0x3c      add r2, r2, r6      //r2 (vtot) += 4

ret:
0x3e      jmp lr

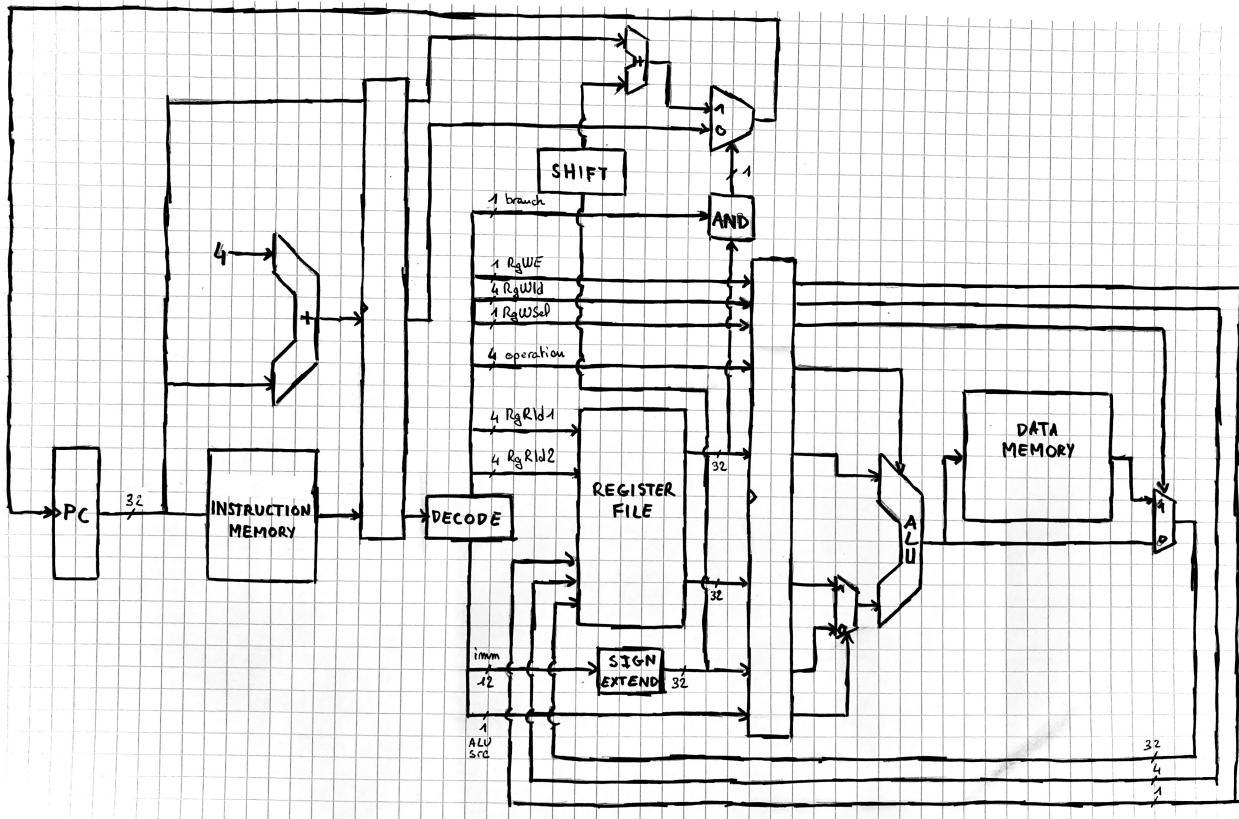
invalid_ret:
0x40      add r0, r4, r5
0x42      jmp lr

```

Pipelining

Processor diagram

En utilisant pour référence le processeur abordé en classe et en l'adaptant à notre jeu d'instructions, on obtient le diagramme suivant :



Comme indiqué dans l'énoncé, le processeur fait usage d'un pipeline à 3 étages :

- Le premier étage, **Instruction Fetch**, est inchangé par rapport au cours
- Le deuxième étage, **Instruction Decode**, inclut désormais l'exécution des instructions modifiant le PC (`bnez`, `jmp`, `call`). Pour effectuer ce changement, le bloc de calcul a été ramené avant la bascule D associée (ID/EX). Cela est possible car notre seule instruction de saut conditionnel teste si un registre donné est non-nul. Il n'y a donc pas besoin d'attendre un calcul de l'ALU pour obtenir notre condition, on peut directement faire la vérification à la sortie du bloc **REGISTER FILES**. On peut noter le rôle important du bloc opératoire **SHIFT**, assurant l'alignement de l'adresse à laquelle on saute.
- Le troisième étage, **Execute**, comprend désormais toutes les étapes des accès mémoire (calcul d'adresse, lecture, écriture). Il n'y a pas de bascule D sur les signaux `RgWE`, `RgWId`, `RgWSel` : l'écriture des registres se fait au début de EX, tandis que sa lecture à la fin de l'étape ID.

Les signaux décodés sont mis à jour selon l'instruction reçue. Un signal restant inchangé d'une instruction précédente implique qu'il n'est pas utilisé ou qu'il sera ignoré par l'instruction courante. Voici un tableau explicatif des signaux :

Nom du signal	Taille en bits	Description
branch	1	Indique si l'instruction est un saut (1 pour <code>bnez</code> , <code>jmp</code> , 0 sinon)
RgWE	1	Indique l'accès en mode écriture à REGISTER FILE
RgWId	4	Sélectionne le registre dans lequel écrire

Nom du signal	Taille en bits	Description
RgWSel	1	Sélectionne la source de la donnée à écrire en mémoire (1 pour ldr , 0 sinon)
operation	4	Sélectionne la bonne opération au niveau de l'ALU
RgRId1	4	Premier registre lié à l'opération
RgRId2	4	Second registre lié à l'opération
imm	12	Immédiat lié à l'opération
ALUsrc	1	Sélectionne le deuxième argument fourni à l'ALU

call instruction

Rappel sur l'instruction **call** :

imm[8:5]	0000	imm[4:1 9]	1	00	call
----------	------	------------	---	----	------

On utilise un immédiat pour donner sa nouvelle valeur au registre **pc**, l'ancienne étant stockée dans le registre **lr**. **Notre implémentation de call requiert de pouvoir écrire pc et lr dans le même cycle.** Cela est dû à la valeur mise dans **pc** qui n'est pas relative à l'ancienne. Autrement dit, il faudrait faire une écriture classique plutôt que de passer par le bloc de l'étage ID.

Pour des raisons de claretés, le schéma n'a pas été refait, mais deux solutions sont possibles pour palier à ce problème :

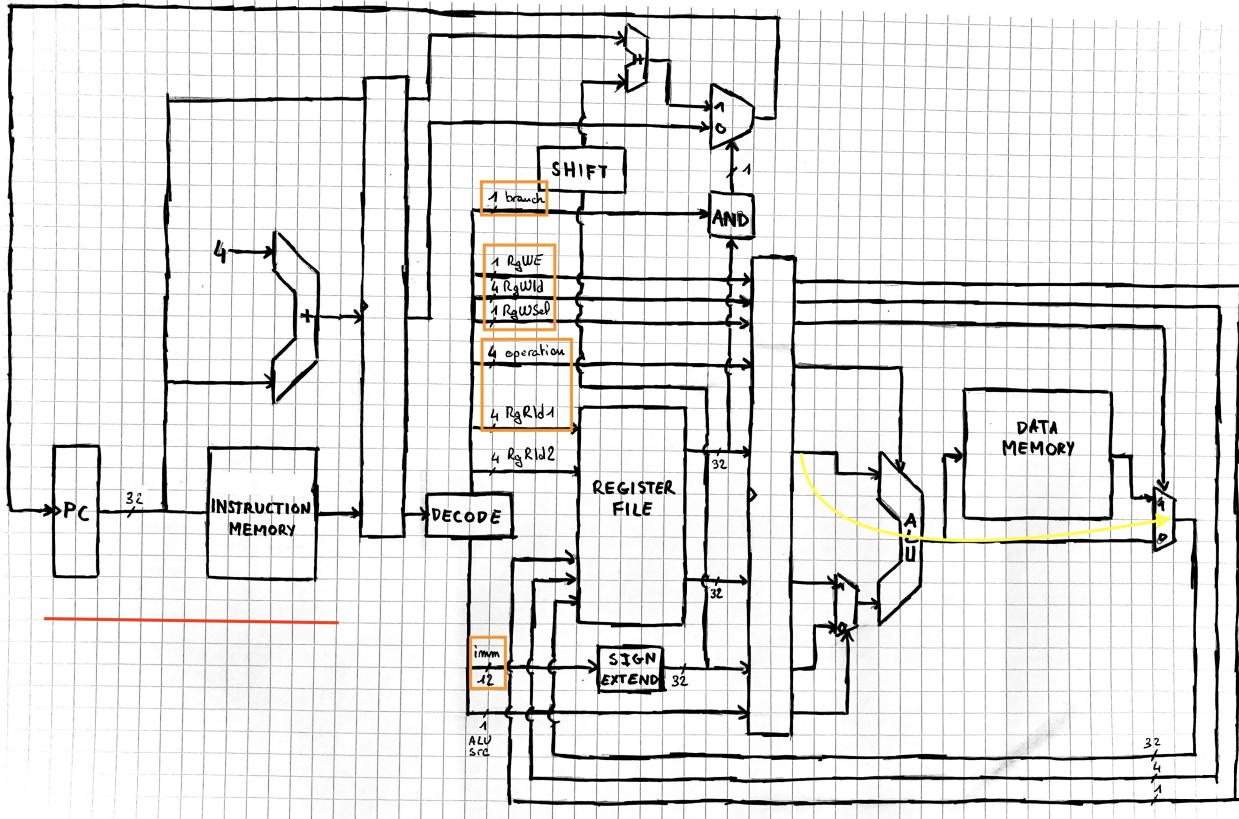
- Ajouter un canal dédié pour l'écriture de **pc**. Cette solution est assez lourde au niveau matériel, puisque cela implique l'utilisation de nouveaux signaux et introduit la problématique d'accès mémoire concurrent.
- Ajouter un multiplexeur avant le bloc de saut à l'étage ID permettant de choisir la nouvelle valeur de **pc** parmi les suivantes :
 - **pc+4**
 - **pc+imm*2**
 - **imm**

Le signal contrôlant ce multiplexeur peut être branch, qui est désormais écrit sur deux bits plutôt qu'un.

La deuxième solution est préférable. C'est avec celle-ci que je vais décrire l'exécution d'une instruction **call**. Posons pour le nouveau signal **branch** :

Nouvelle valeur de pc	Valeur de branch
pc+4	2'b00
pc+imm*2	2'b01
imm	2'b11

De cette manière, le bit de poids faible joue un rôle identique à l'ancien signal **branch** (de 1 bit) tandis que le bit de poids fort détermine si l'on effectue un décalage relatif ou non.



Il n'y a rien de notable à l'étage IF du pipeline pour l'exécution de l'instruction `call`.

À l'étage ID, Le signal d'entrée est décodé pour mettre à jour tous les signaux sortant sauf `RgRId2` et `ALUsrc` qui ne sont pas utiles. Voici les valeurs prises par les signaux modifiés :

Signal	Taille en bits	Valeur
branch (bis)	2	2'b11, comme posé plus haut
RgWE	1	1'b1, on effectue une écriture dans REGISTER FILE
RgWId	4	4'1110, on écrit la valeur de pc dans lr
RgWSel	1	1'b0, on veut utiliser ALU_out, pas une donnée mémoire
operation	4	4'bXXXX valeur résultant en ALU_out = rs1
RgRId1	4	4'1111, le registre dont on veut la valeur est pc/r15
imm	12	12'b000 [imm], avec [imm] l'immédiat sur 9 bits fourni dans l'instruction

À l'étage EX, le processeur se comporte comme si l'on voulait copier la valeur d'un registre dans un autre (pc dans lr en l'occurrence).

Hazards, flushing logic

On peut être confronté à des *data hazards* en cas d'utilisation immédiate du résultat d'une instruction. Dans le cas de deux instructions de saut successives, l'implémentation du branch delay slot peut éventuellement mener à un *control hazard*. Il ne devrait pas se produire de *structural hazard* de par :

- L'implémentation matérielle de `call` choisie. La seconde mentionnée aurait pu mener à des accès simultanés aux registres.
- L'utilisation conventionnelle des registres (i.e. ne pas directement interagir avec `pc/lr`).

Notre processeur ne nécessite pas de logique pour flush les instructions. Ceci est le résultat de deux facteurs :

- L'exécution des sauts se fait à l'étape ID du pipeline. Ceci implique qu'une seule instruction aura vu son traitement débuter lors du saut. **Il y a donc potentiellement une instruction à flush.**
- Notre architecture implémente **un branch delay slot**. Autrement dit, une unique instruction suivant un saut est exécutée plutôt que d'être *flush*. Il n'y a donc pas besoin de *flush* d'instruction.