

SE201 : Projet 1

Lucie Molinié, Isaïe Muron, Florian Tarazona

Partie 1 - Jeu d'instruction RISC-V

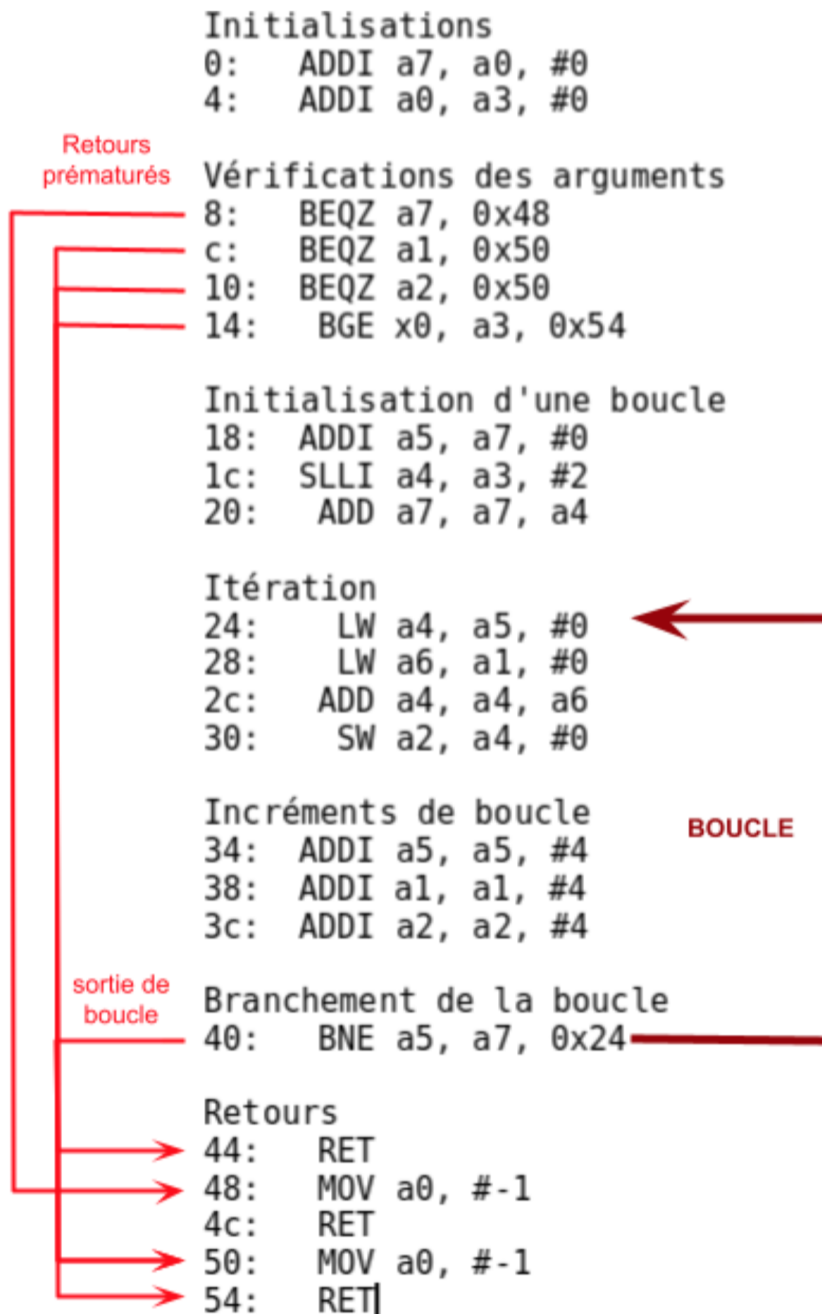
Le programme

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 incréments 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 `a0 < a1`. 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 prédiction 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-O0.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(s0)
11 10:   fcb42c23          sw     a1,-40(s0)
12 14:   fcc42a23          sw     a2,-44(s0)
13 18:   fcd42823          sw     a3,-48(s0)
14 1c:   fdc42783          lw     a5,-36(s0)
15 20:   00078a63          beqz   a5,34 <.L2>
16 24:   fd842783          lw     a5,-40(s0)
17 28:   00078663          beqz   a5,34 <.L2>
18 2c:   fd442783          lw     a5,-44(s0)
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(s0)
27 40:   0500006f          j      90 <.L5>
28
29 00000044 <.L6>:
30 44:   fec42783          lw     a5,-20(s0)
31 48:   00279793          slli   a5,a5,0x2
32 4c:   fdc42703          lw     a4,-36(s0)
33 50:   00f707b3          add     a5,a4,a5
34 54:   0007a683          lw     a3,0(a5)
35 58:   fec42783          lw     a5,-20(s0)
36 5c:   00279793          slli   a5,a5,0x2
37 60:   fd842703          lw     a4,-40(s0)
38 64:   00f707b3          add     a5,a4,a5
39 68:   0007a703          lw     a4,0(a5)
40 6c:   fec42783          lw     a5,-20(s0)
41 70:   00279793          slli   a5,a5,0x2
42 74:   fd442603          lw     a2,-44(s0)
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(s0)
47 88:   00178793          addi    a5,a5,1
48 8c:   fef42623          sw     a5,-20(s0)
```

```

49
50 00000090 <.L5>:
51 90: fec42703          lw      a4,-20(s0)
52 94: fd042783          lw      a5,-48(s0)
53 98: faf746e3          blt     a4,a5,44 <.L6>
54
55 0000009c <.LBE2>:
56 9c: fd042783          lw      a5,-48(s0)
57
58 000000a0 <.L4>:
59 a0: 00078513          mv      a0,a5
60 a4: 02c12403          lw      s0,44(sp)
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(s0)      //Chargement de i dans a5
                        //depuis la stack
slli    a5, a5, 0x2      //Multiplication de i pour être
                        //compatible avec une adresse
lw      a4, -36(s0)      //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 :

-0x00	(-0)	+-----+
		s0
		+-----+
		+-----+
		+-----+
-0x10	(-16)	+-----+
		i
		+-----+
		+-----+
		+-----+
-0x20	(-32)	+-----+
		v1
		+-----+
		v2
		+-----+
		dst
		+-----+
		size
-0x30	(-48)	+-----+

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 :

-0x00	(-0)	+-----+
		s0
		+-----+
		+-----+
		i
		+-----+
		+-----+
-0x10	(-16)	+-----+
		v1
		+-----+
		v2
		+-----+
		src
		+-----+
		size
-0x20	(-32)	+-----+

- 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 *incréments 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'utile.

Partie 3 - Architecture RISC-V

Flot d'exécution

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

Exec Diagram

Exec Diagram

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 :

NO HAZARD

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

HAZARD 1

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 :

HAZARD 2

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 :

HAZARD 3

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

Bruh

Bruh2