Architecture Matérielle :
Projet — *Conception d'un mini-processeur*

Justin BABONNEAU

3 janvier 2023

Introduction

L'objectif de ce projet était de réaliser un mini-processeur à partir d'un squelette. Pour ce faire, nous devions nous appuyer sur le jeu d'instructions d'un processeur MIPS, que nous avons implémenté dans le logiciel Diglog...

Les instructions qui ont été implémentées sont :

- `nop`
- `ldi`
- `not`
- `lsr`
- `or`
- `and`
- `addi`
- `add`
- `subi`
- `sub`
- `muli`
- `st`
- `ld`
- `out`
- `in`
- `jr`
- `jeq`
- `jle`
- `jlt`
- `jne`
- `jmp`

Table des matières

1 TD 5

1.1 Exercice 1 - Exécution du premier programme

Pour le moment, le processeur fourni ne supporte que les instructions `ldi` et `addi`. Faire appel à toute autre instruction conduit à un comportement non spécifié. Le but de cet exercice est d'ajouter le support des sauts inconditionnels, afin de pouvoir simuler l'exécution du programme suivant :

```
1          ldi r1, 42
2          addi r0, r1, 17
3  end:    jmp end
```

1.1 Donner la valeur de chaque registre à la fin de l'exécution de ce programme.

A la fin de l'exécution de ce programme, on a :

registre	*r0*	*r1*
valeur	59	42

1.2 Traduire à la main le code assembleur en langage machine, sachant que la première instruction sera placée à l'adresse 0000.

On traduit le code assembleur en langage machine (binaire) :

instruction	adresse	op	flags	rd	rs rt / imm5
`ldi r1, 42`	0000	000	01	001	00101010 (imm8)
`addi r0, r1, 17`	0001	010	00	000	001 (rs) 10001 (imm5)
`end: jmp end`	0002	111	00 (imm13(high_1))	000 (imm13(high_2))	00000000 (imm13(low))

1.3 Compiler le code assembleur avec digcomp. Comparer le contenu des fichiers ainsi produits à la réponse de la question précédente.

```
[juauke@localhost digproc]$ ../digcomp/digcomp ../tests/addi/ai.s
end = 2
--------
   0 [x0000]: r1 <- 42
   1 [x0001]: r0 <- r1 + 17
   2 [x0002]: goto end
```

FIGURE 1 – Compilation du programme cité ci-dessus avec Digcomp.

```
1 0000:09
2 0001:40
3 0002:e0
```

FIGURE 2 – `addi.s.hi`

```
1 0000:2a
2 0001:31
3 0002:02
```

FIGURE 3 – `addi.s.lo`

Etant donné que la partie `.hi` contient les 8 premiers bits de la ligne et que la partie `.lo` contient les 8 derniers, on a (en traduisant les résultats hexadécimaux précédents en binaire) :

adresse	`.hi`	`.lo`
0000	000 01 001 (0x09)	00101010 (0x2a)
0001	010 00 000 (0x40)	001 10001 (0x31)
0002	111 00 000 (0xe0)	00000010 (0x02)

Les valeurs obtenues correspondent à celles trouvées précédemment.

1.4 Charger les fichiers obtenus dans Diglog, et déterminer le chemin suivi par les données lors de l'exécution de l'instruction `ldi`.
Note : Pour faire une exécution pas à pas, on utilisera un générateur en guise d'horloge, sur lequel on pourra cliquer afin de passer au front (montant ou descendant) suivant.

Lors de l'instruction `ldi`, on lit sur le front montant la valeur 42 grâce aux imm ($0b00101010$) et sur le front descendant, on écrit dans le registre `r0` cette valeur.

On a maintenant cette horloge :

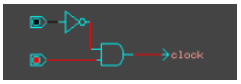

FIGURE 4 – Horloge pas à pas.

1.5 Avancer d'un cycle, et déterminer le chemin suivi par les donnée lors de l'exécution de l'instruction `addi`.

Lors de l'instruction `addi`, on lit sur le front montant la valeur du registre `r0` (c'est-à-dire 42 ou $0b00101010$) et sur le front descendant, on additionne (avec l'ALU en mode addition) cette valeur avec 17 ($0b10001$) et on l'écrit dans le registre `r1`.

1.6 Avancer à nouveau d'un cycle, et identifier comment récupérer l'adresse à laquelle il faudra être après le saut.

Pour récupérer l'adresse à laquelle il faudra être après le saut, il faut que l'on détermine l'écart relatif entre la position actuelle du pointeur courant (PC) et la position à laquelle on veut aller. C'est pourquoi il faut une ALU en mode soustracteur.

1.7 Déterminer le rôle des bits de contrôle *write reg*, *arg2_imm* et *res_imm*, puis compléter le tableau suivant (mettre X si la valeur du bit de contrôle n'a pas d'influence) :

instruction	*do_jmp_abs*	*write_reg*	*arg2_imm*	*res_imm*	*do_sub*
`nop`	0	0	X	X	0
`ldi`	0	1	X	1	0
`addi`	0	1	1	0	0
`subi`	0	1	1	0	1
`add`	0	1	0	1	0
`sub`	0	1	0	1	1
`jmp`	1	0	X	X	0

1.8 Modifier le processeur afin de gérer l'instruction de saut `jmp`. Utiliser pour cela le bit de contrôle *do_jmp_abs (le bit do_jcc servira plus tard pour les sauts conditionnels).*

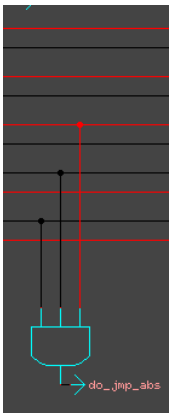

FIGURE 5 – Bit de contrôle *do_jmp_abs*.

J'ai ajouté le bit de contrôle `do_jmp_abs` qui vaut 1 si et seulement si l'instruction courante est `jmp` c'est-à-dire que $\boxed{\text{op} = 111}$

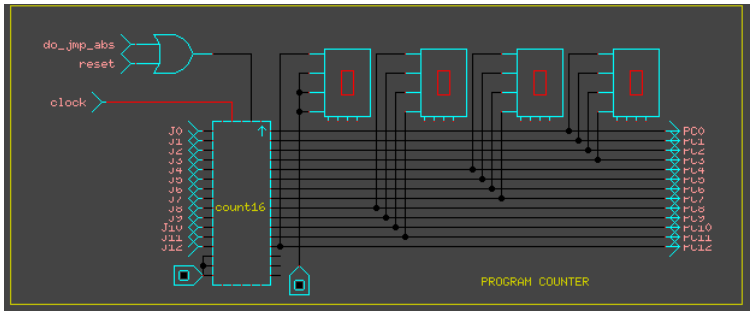

FIGURE 6 – Ajout du bit de contrôle *do_jmp_abs* au compteur de programme.

J'ai aussi ajouté l'arrivée de ce bit de contrôle comme bit de contrôle du compteur de programme.

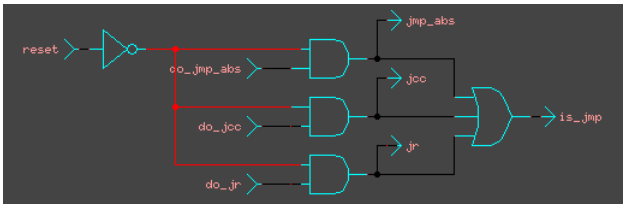

FIGURE 7 – Bit de contrôle *is_jmp* (seule la partie absolue importe pour l'instant).

J'ai également ajouté le bit de contrôle *is_jmp* qui vaut 1 lorsque l'instruction est un saut et que *reset* est à 0.

1.2 Exercice 2 - Gestion complète de l'addition et de la soustraction

2.1 Donner la liste des instructions dont l'exécution nécessite d'utiliser l'ALU en tant que soustracteur.

Les instructions suivantes ont besoin de l'ALU en tant que soustracteur :

- les instructions de soustraction : `subi` et `sub` ;
- les instructions de lecture en RAM : `st` et `ld` ;
- les sauts conditionnels : `jeq`, `jle`, `jlt` et `jne`.

2.2 Modifier le processeur afin d'obtenir la bonne valeur pour le bit de contrôle *do_sub*. Vérifier que le processeur gère désormais aussi l'instruction `subi`.

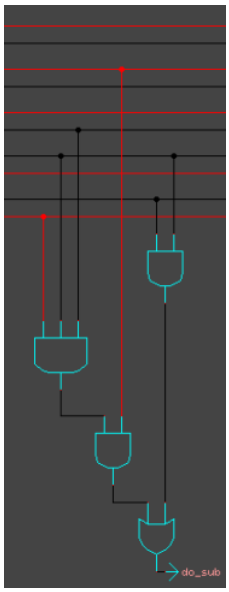

FIGURE 8 – Bit de contrôle *do_sub*

Le processeur gère bien l'instruction `subi`.

2.3 Modifier le banc de registre afin de pouvoir lire les valeurs de deux registres (pas forcément différents) à chaque cycle.

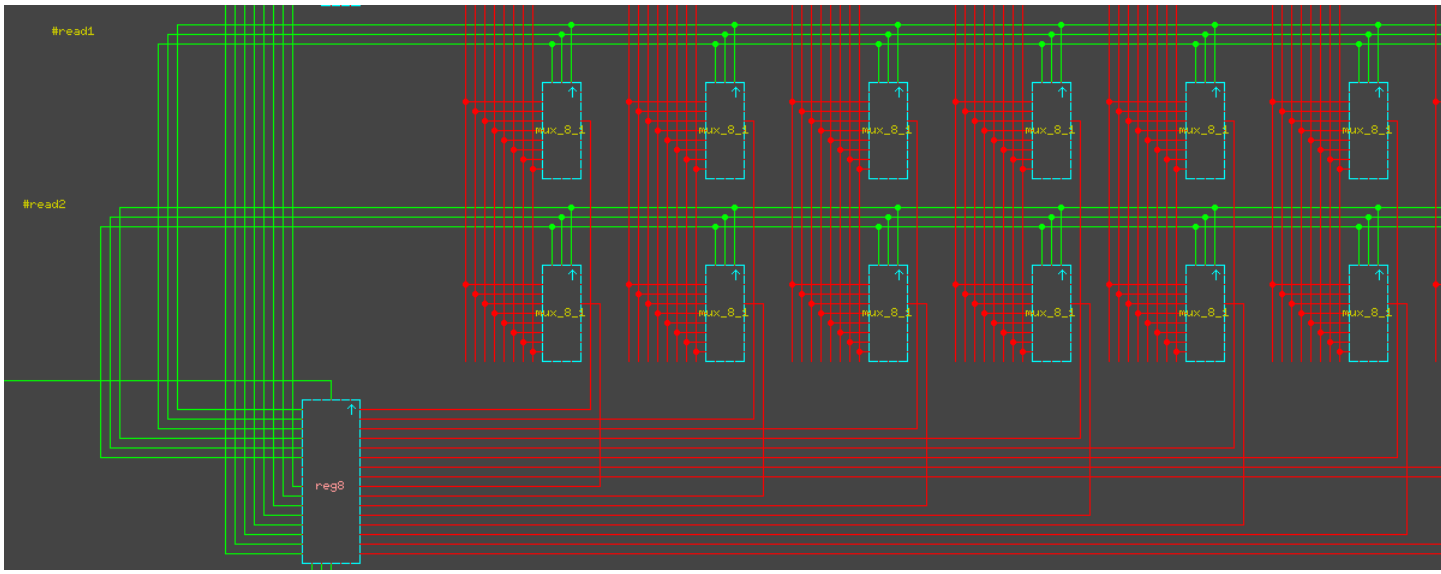

FIGURE 9 – Ajout de la seconde rangée de multiplexeurs sur `reg8` pour lire la valeur d'un deuxième registre

La seconde rangée de multiplexeurs a été ajoutée pour permettre la lecture de la valeur d'un deuxième registre dont l'indice est la nombre binaire formée par les bits 3 à 5 (correspondant à *#read2*) de l'entrée.

2.4 Pour chaque instruction, déterminer le nombre de lectures de registres à effectuer, ainsi que les bits contenant les numéros de registres à lire.

instr	lecture(s)	bits pertinents
nop	0	X
ldi	1	rd
not	1	rd
lsr	2	rd, rs
or	3	rd, rs, rt
and	3	rd, rs, rt
addi	2	rd, rs
add	3	rd, rs, rt
subi	2	rd, rs
sub	3	rd, rs, rt
muli	2	rd, rs
mul	3	rd, rs, rt
st	2	rd, rs
ld	2	rd, rs
out	1	rd
in	1	rd
jr	2	rd, rs
jeq	2	rd, rs
jle	2	rd, rs
jlt	2	rd, rs
jne	2	rd, rs
jmp	0	X

2.5 Ajouter le support des instructions `add` et `sub`, et proposer un code assembleur de test. Vous devrez sûrement modifier le calcul des bits de contrôle introduits dans l'exercice précédent.

Les instructions `add` et `sub` sont désormais supportées.

Les bits de contrôle n'ont pas dû être changés (car on avait déjà prévu les cas correspondant à ces deux instructions).

1.3 Exercice 3 - Entrées/sorties et gestion du saut `jeq`

Afin de pouvoir procéder à de vrais tests, il nous manque deux choses :

- un coté interactif pour accélérer/faciliter les tests, c'est-à-dire la possibilité de saisir des entrées au clavier et d'afficher des résultats sur un écran ;
- notre première instruction de saut conditionnel, afin d'augmenter significativement l'expressivité dans les programmes codés en assembleur.

Pour le premier point, le fichier *io.lgf* fournit déjà un clavier et un écran. Il est possible de récupérer une touche saisie au clavier (fils *kb0* à *kb7*) à condition de positionner le bit de contrôle *in* à 1. De plus, il est possible d'afficher le caractère dont le code ASCII est la valeur RD (fils *RD0* à *RD7*) à condition de positionner le bit de contrôle *out* à 1.

3.1 Faire en sorte que les bits de contrôle *in* et *out* reçoivent la bonne valeur.

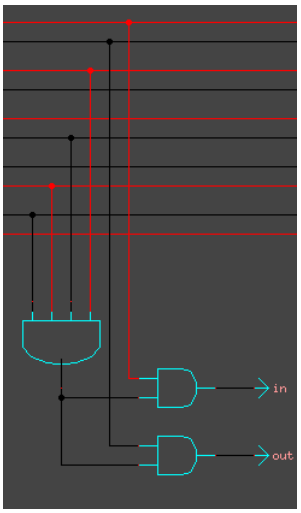

FIGURE 10 – Gestion des bits de contrôle `in` et `out`.

Le bit de contrôle `out` est actif lorsque lorsque l'instruction courante est `out` c'est-à-dire que $\boxed{\text{op} = 100}$ et $\boxed{\text{flags} = 10}$.
Le bit de contrôle `in` est actif lorsque lorsque l'instruction courante est `in` c'est-à-dire que $\boxed{\text{op} = 100}$ et $\boxed{\text{flags} = 11}$.

3.2 Tester le clavier et l'écran. Que se passe-t-il si on récupère les données du clavier alors qu'aucune touche n'a été frappée ? Et si de nombreuses touches ont été frappées depuis la dernière récupération de touche ?

Si l'on récupère les données du clavier alors qu'aucune touche n'a été frappée, on obtient 255 ($0b11111111$). Si de nombreuses touches ont été frappées, la plus ancienne est remplacée par la touche nouvellement pressée. Le *buffer* (tampon) du clavier fonctionne donc comme une file (First In First Out).

3.3 Rappeler comment on peut tester que deux valeurs entières sont égales. Modifier l'ALU afin d'avoir une nouvelle sortie, nécessaire à la réalisation du test.

Pour tester que deux valeurs sont égales, on peut vérifier que leur différence est nulle.
On a ajouté un bit `Zero` qui permet de savoir si la valeur en sortie de l'ALU est nulle.

3.4 Afin de gérer les instructions de saut conditionnel, nous allons utiliser le bit de contrôle *do_jcc*, qui vaudra 1 si et seulement si l'instruction courante est de type saut conditionnel et qu'il convient d'effectuer le saut en question.
Faire en sorte que ce bit de contrôle reçoive la bonne valeur lors de l'exécution d'un `jeq`.

J'ai décidé de renommer le bit de contrôle en *do_jmp_cnd* (car j'utilise *do_jcc* quand je fais effectivement le saut conditionnel [la condition est alors également vérifiée et que l'on est pas en *reset*]).

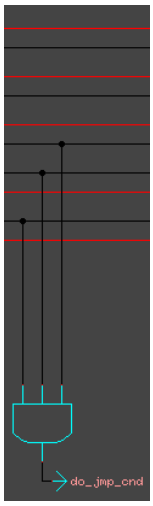

FIGURE 11 – Bit de contrôle *do_jmp_cnd*

Ce bit de contrôle vaut 1 si et seulement si l'instruction courante est un saut conditionnel c'est-à-dire que $\boxed{\text{op} = 110}$.

3.5 Modifier le processeur afin de mettre correctement à jour le pointeur d'instruction PC lors de l'exécution d'une instruction `jeq`.

En plus de l'ajout de la question précédente, on a ajouté un bit de contrôle *do_jcc*, l'arrivée du bit *do_jcc* comme bit de contrôle au compteur de programme.

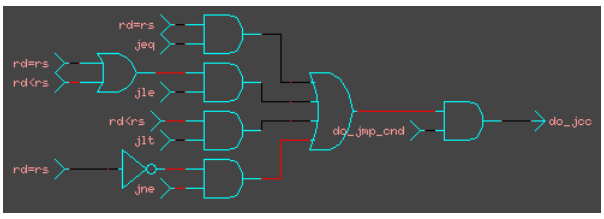

FIGURE 12 – Bit de contrôle *do_jcc*.

Ce bit de contrôle vaut 1 lorsque au moins une (une seule à la fois normalement) des instructions de sauts conditionnels est active et que la condition associée est vérifiée.

3.6 Tester le bon fonctionnement des sauts conditionnels dans le cas d'un saut :

1. en avant (vers une adresse plus grande que la valeur courante dans PC) ;
2. en arrière (vers une adresse plus petite que la valeur courante dans PC).

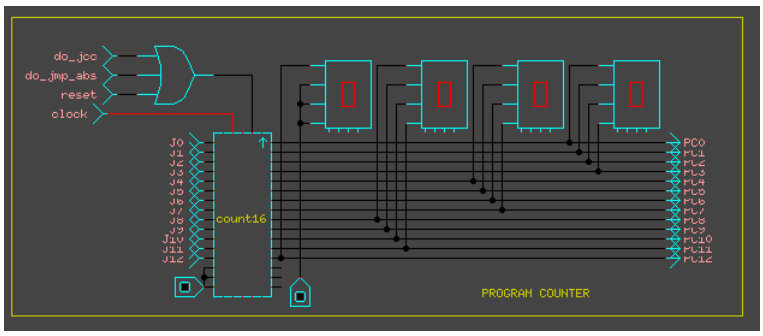

FIGURE 13 – Ajout du bit de contrôle *do_jcc* au compteur de programme.

Le premier code (correspondant à un saut dans le passé) et le second (correspondant à un saut dans le futur) fonctionnent à présent.

3.7 Écrire un code assembleur qui récupère les touches réellement saisies par l'utilisateur, les affichent, et s'arrête dès que l'utilisateur a appuyé sur la touche *Entrée* (*Cr*, de code ASCII 13).

```
1  loop:        in r7 # r7 <- getchar()
2          ldi r5, 13 # r5 <- 13 = <Enter>
3          jeq r7, r5, end # if (r7 == r5) goto end
4          out r7 # putchar() <- r7
5          jmp loop # goto loop
6
7  end:        jmp end
```

FIGURE 14 – `../asm/q3_7/char_read.s`

Le programme `char_read.s` lit les caractères entrés par l'utilisateur et boucle jusqu'à que l'utilisateur appuie sur la touche *Entrée* du clavier.

3.8 Écrire un code assembleur qui récupère un entier $n \in [1, 9]$ et un caractère c, puis qui affiche à l'écran un carré de taille n et composé de caractères c.

```
        ldi r0, 49 # r0 <- 49 = '1'
        ldi r1, 57 # r1 <- 57 = '9'

        ldi r6, 105 # r6 <- 105 = 'i'
        out r6 # putchar() <- r6
        ldi r6, 63 # r6 <- 63 = '?'
        out r6 # putchar() <- r6

loop:       in r7 # r7 <- getchar()
        jlt r7, r0, loop # if (r7 < r0) goto loop (if r7 < '1' )
        jlt r1, r7, loop # if (r1 < r7) goto loop (if r7 > '9')

        out r7 # putchar() <- r7
        subi r6, r7, 24 # r6 <- r7-24 (cannot do subtract strictly more than 31 at once)
        subi r7, r6, 24 # r7 <- r6-24

        ldi r0, 97 # r0 <- 97 = 'a'
        ldi r1, 122 # r1 <- 122 = 'z'

        ldi r6, 13 # r6 <- 13 = <Enter>
        out r6 # putchar() <- r6
        ldi r6, 99 # r6 <- 99 = 'c'
        out r6 # putchar() <- r6
        ldi r6, 63 # r6 <- 63 = '?'
        out r6 # putchar() <- r6

getc:       in r5 # r5 <- getchar()
        jlt r5, r0, getc # if (r5 < r0) = goto loop (if r5 < '1' )
        jlt r1, r5, getc # if (r1 < r5) = goto loop (if r5 > '9')

        out r5 # putchar() <- r5
        ldi r6, 13 # r6 <- 13 = <Enter>
        out r6 # putchar() <- r6

        ldi r3, 0 # r3 <- 0
        add r0, r7, 0 # r0 <- r7
line:       add r1, r7, 0 # r1 <- r7

putc:   out r5 # putchar() <- r5
        subi r2, r1, 1 # r2 <- r1-1
        add r1, r2, 0 # r1 <- r2
        jeq r1, r3, nextl # if (r1 == r3) goto nextl
        jmp putc # goto putc

nextl:      out r6 # putchar() <- r6

        subi r2, r0, 1 # r2 <- r0-1
        add r0, r2, 0 # r0 <- r2
        jeq r0, r3, end # if (r0 == r3) goto end
        jmp line # goto line

end:    jmp end
```

FIGURE 15 – `../asm/q3_8/char_square.s`

Après saisie d'un chiffre non nul n et d'un caractère c par l'utilisateur, le programme `s.s` affiche un carré de c de côté de longueur n.

1.4 Exercice 4 - Gestion de la mémoire et des autres sauts conditionnels

4.1 Ajouter le support des instructions `ld` et `st`. Pour cela, il faudra identifier le parcours que les données devront suivre, ajouter des multiplexeurs au besoin, et utiliser au minimum un nouveau bit de contrôle : *write_mem*.
Note : Pour l'instant, on ne traitera pas la partie `imm5` de ces instructions.

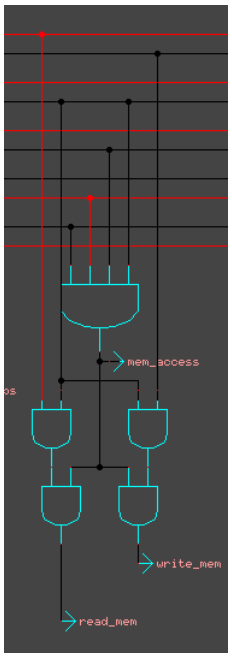

FIGURE 16 – Bits de contrôle liés à la mémoire

On a ajouté les bits de contrôle suivants :

- *mem_access* qui vaut 1 lors des instructions `st` et `ld` (et 0 sinon) c'est-à-dire lorsque op = 100 ;
- *read_mem* qui vaut 1 lors de l'instruction `ld` (et 0 sinon) c'est-à-dire que la première condition est vérifiée et flags = 01 ;
- *write_mem* qui vaut 1 lors de l'instruction `st` (et 0 sinon) c'est-à-dire que la première condition est vérifiée et flags = 00.

On a ajouté les multiplexeurs suivants :

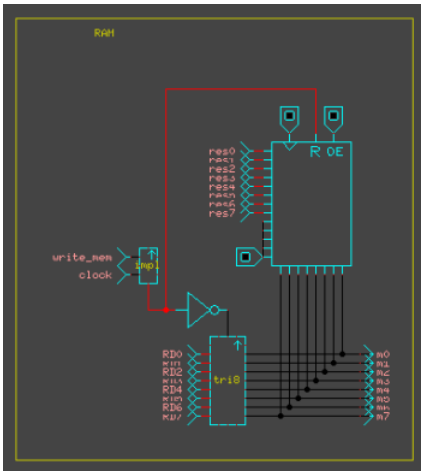

FIGURE 17 – RAM après les modifications.

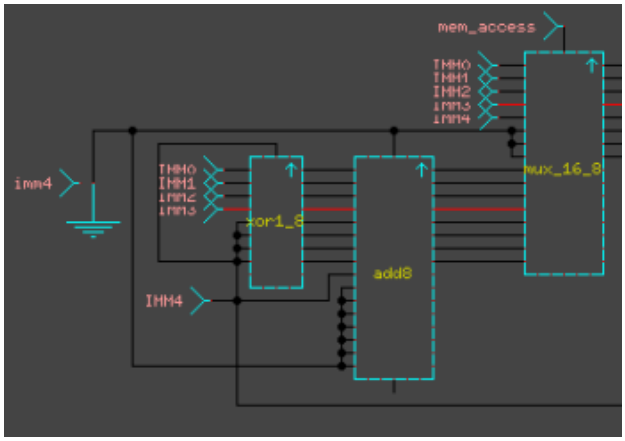

FIGURE 18 – Multiplexeur qui permet de décaler négativement l'adresse en mémoire.

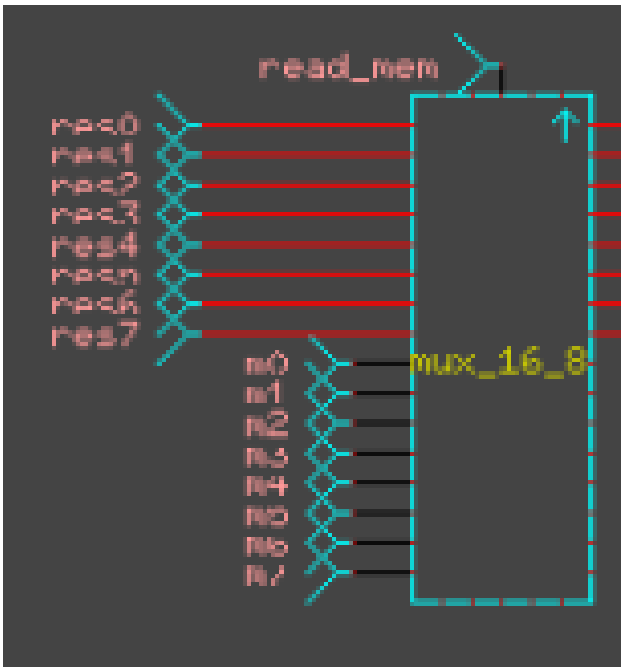

FIGURE 19 – Multiplexeur qui permet de lire en mémoire lors d'un `rd`.

4.2 Écrire un code assembleur qui récupère les touches saisies par l'utilisateur et les stocke en mémoire jusqu'à la saisie de la touche *Entrée*, puis qui réécrit les données ainsi récupérées à l'écran en ordre inverse.
Exemple : La saisie de "123<*Entrée*>" donnera donc lieu à l'affichage de "321".

```
1  begin:       ldi r0, 48 # r0 <- 48 = '0'
2          ldi r1, 57 # r1 <- 57 = '9'
3          ldi r2, 0 # r0 <- 0
4          ldi r3, 1 # r3 <- 1
5
6          ldi r6, 69 # r6 <- 69 = 'E'
7          out r6 # putchar() <- r6
8          ldi r6, 110 # r6 <- 110 = 'n'
9          out r6 # putchar() <- r6
10         ldi r6, 116 # r6 <- 116 = 't'
11         out r6 # putchar() <- r6
12         ldi r6, 105 # r6 <- 105 = 'i'
13         out r6 # putchar() <- r6
14         ldi r6, 101 # r6 <- 101 = 'e'
15         out r6 # putchar() <- r6
16         ldi r6, 114 # r6 <- 114 = 'r'
17         out r6 # putchar() <- r6
18         ldi r6, 32 # r6 <- 32 = ' '
19         out r6 # putchar() <- r6
20         ldi r6, 63 # r6 <- 63 = '?'
21         out r6 # putchar() <- r6
22         ldi r6, 32 # r6 <- 32 = ' '
23         out r6 # putchar() <- r6
24
25 loop:       in r7 # r7 <- getchar()
26         ldi r5, 13 # r5 <- 13 = <Enter>
27         jeq r7, r5, end # if (r7 == r5) goto end
28         jlt r7, r0, loop # if (r7 < r0) goto loop (if r7 < '0')
29         jlt r1, r7, loop # if (r1 < r7) goto loop (if r7 > '9')
30         subi r6, r7, 24 # r6 <- r7-24; convert to decimal by subtracting 48
31         subi r7, r6, 24 # r7 <- r6-24; cannot subtract strictly more than 31 at once
32         mul r5, r2, r3 # r5 <- r2*r3 mod 256
33         add r6, r5, r7 # r6 <- r5+r7; browse the number given
34
35         addi r2, r6, 0 # r2 <- r6
36
37         muli r6, r3, 10 # r6 <- r3*10 mod 256 (view number as number*10^(index_of_digit-1))
38
39         addi r6, r3, 0 # r6 <- r3
40
41         addi r6, r7, 24 # r6 <- r7+24; convert back to ascii by subtracting 48
42         addi r7, r6, 24 # r6 <- r7+24
43         out r7 # putchar() <- r7
44         jmp loop
45
46 end:        jmp end # goto end
```

FIGURE 20 – `reverse_int.s`

Les lignes 6 à 23 affichent 'Entier ? ' à l'écran.

Dans la boucle, on commence par récupérer le caractère entrée par l'utilisateur dans `r7`, si c'est '<Entrée>', on va à la fin. Sinon, on vérifie que ce dernier est un chiffre (entre 0 et 9) : si c'est le cas, on continue et on convertit sa valeur en décimal. Sinon, on boucle. On affiche les chiffres dans l'ordre inverse en lisant de 0 à la taille de la chaîne entrée.

4.3 Modifer l'ALU afin d'avoir 3 nouvelles sorties, correspondant aux drapeaux S (signe), C (carry), et O (overflow).

Remarquons d'abord que *Carry* et *Overflow* sont confondus puisque avoir une retenue revient à dépasser la capacité de l'ALU donc par exemple seuls *Carry* et *Sign* sont à implémenter effectivement.

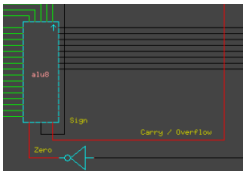

FIGURE 21 – Ajout des drapeaux à l'ALU.

4.4 Démontrer que, pour a et b deux entiers signés sur 8 bits différents, $a \leq b$ est équivalent à $S = O$, où S et O sont obtenus suite au calcul de $b - a$ par l'ALU.

Soient a et b deux entiers signés sur 8 bits. Alors :

$$\exists (a_i)_{0 \leq i \leq 7} \in \{0,1\}^7 \,|\, a = \sum_{i=0}^{7} 2^i a_i$$

$$\exists (b_i)_{0 \leq i \leq 7} \in \{0,1\}^7 \,|\, b = \sum_{i=0}^{7} 2^i b_i$$

Notons S et O les bits de Signe (*Sign*) et de Retenue (*Overflow*) renvoyés par l'ALU après le calcul de $b - a$.

Notons également C_i la retenue dans le calcul de $b - a$ d'ordre $i \in [\![0; 7]\!]$.

Montrons que $a \leq b \iff S = O$

$\boxed{\Longrightarrow}$ Supposons que $a \leq b$.

Comme $a \leq b$, $b - a \geq 0$ d'où $\boxed{S = 0}$.

Comme $a \leq b$, il existe

$$k_0 \in [\![0; 7]\!]$$

tel que :

$$\forall i \in [\![k_0 + 1; 7]\!], a_i = b_i \tag{1}$$

$$a_{k_0} = 0 \tag{2}$$

$$b_{k_0} = 1 \tag{3}$$

Par conséquent, $c := a - b = a + \overline{\text{xor}(b, 255)} + 1$ va vérifier :

$$c_{k_0} = C_{k_0 - 1}$$

$$\forall i \in [\![k_0 + 1; 7]\!], c_i = 1$$

En effet, on a :

$$\begin{aligned} c_{k_0} &= a_{k_0} - b_{k_0} + C_{k_0 - 1} \\ &= a_{k_0} + \overline{b_{k_0}} + C_{k_0 - 1} \\ &\overset{(2) \wedge (3)}{=} 0 + 0 + C_{k_0 - 1} \\ c_{k_0} &= C_{k_0 - 1} \text{ et donc } C_{k_0} = 0 \end{aligned}$$

$$\begin{aligned} \forall i \in [\![k_0 + 1; 7]\!], c_i &= a_i + \overline{b_i} + C_i \\ &\overset{(1)}{=} a_i + \overline{a_i} + 0 \\ c_i &= 1 \end{aligned}$$

Puisque que l'on n'a pas dépassé la limite de 8 bits, on a donc $\boxed{O = 0}$.

Ainsi, on a $S = 0 = O$ et finalement $\boxed{S = O}$.

$\boxed{\Longleftarrow}$ Supposons que $S = O$.

Procédons par une disjonction de cas :

- Supposons que $S = 0$.
  Par conséquent, d'après l'hypothèse, $O = 0$. Donc : $b - a \geq 0$ sans avoir dépassé c'est-à-dire que $\boxed{a \leq b}$.
- Supposons que $S = 1$. D'après l'hypothèse, on a $O = 1$ autrement dit $b - a \leq 0$ et le calcul a dépassé la capacité de l'ALU ce qui revient à dire que le signe est faussé et que $b - a \geq 0$ ou encore $\boxed{a \leq b}$.

4.5 Utiliser le résultat précédent et ce qui a déjà été à la section précédente pour ajouter le support des instructions `jlt`, `jle` et `jne`.

Le support de ces instructions a été ajouté à la question **3.4**.

4.6 Écrire un code assembleur qui effectue à l'aide d'une boucle la multiplication par 10 de la valeur stockée dans *r0*.
Comment peut-on faire ce calcul beaucoup plus efficacement ?

```
1         ldi r0, 4 # r0 <- 4
2         ldi r1, 0 # r1 <- 0
3         ldi r2, 10 # r2 <- 10; max_iter
4         ldi r3, 0 # r3 <- 0
5
6   loop:       add r4, r1, r0 # r4 <- r1+r0
7         add r1, r4, 0 # r1 <- r4
8         add r4, r3, 1 # r4 <- r3+1
9         add r3, r4, 0 # r3 <- r4
10        jeq r3, r2, end # if (r3 == r2) goto end
11        jmp loop # goto loop
12
13                out r4 # putchar() <- r4
14  end:    jmp end # goto end
```

FIGURE 22 – `../asm/q4_6/mul10.s`

Pour réaliser le calcul beaucoup plus efficacement, on peut implémenter la multiplication et utiliser l'instruction "`muli r1, r0, 10`" et récupérer la valeur de `r1`.

4.7 Écrire un code assembleur qui récupère les touches saisies par l'utilisateur et les stocke en mémoire jusqu'à la saisie de la touche *Entrée*, puis qui réécrit à l'écran les données ainsi récupérées mais en majuscules.
Exemple : La saisie de "Abc1<*Entrée*>" donnera donc lieu à l'affichage de "ABC1".

```
1         ldi r0, 0 # r0 <- 0
2         ldi r5, 0 # r5 <- 0
3
4         ldi r3, 13 # r3 <- 13 = <Enter>
5
6   loop:       in r7 # r7 <- getchar()
7         jeq r7, r3, eol # if (r7 == r3) goto eol
8         out r7 # putchar() <- r7
9         add r6, r5, r0 # r6 <- r5+r0
10        st r7, r6, 0 # MEM[r6+0] <- r7
11        addi r1, r0, 1 # r1 <- r0+1
12        add r0, r1, 0 # r0 <- r1
13        jmp loop # goto loop
14
15  eol:      out r3 # putchar() <- r3
16        ldi r3, 0 # r3 <- 0
17
18  loop2:
19        add r6, r5, r3 # r6 <- r5+r3
20        ld r7, r6, 0 # r7 <- MEM[r6+0]
21        subi r6, r7, 16 # r6 <- r7-16 (cannot subtract more than 31 at once)
22        subi r7, r6, 16 # r7 <- r7-16
23        out r7 # putchar() <- r7
24        addi r2, r3, 0 # r2 <- r3
25        addi r3, r2, 1 # r3 <- r2+1
26        jeq r0, r3, end # if (r0 == r3) goto end
27        jmp loop2 # goto loop2
28
29  end:      jmp end # goto end
```

FIGURE 23 – `../asm/q4_7/char_to_caps.s`

1.5 Exercice 5 - Améliorations diverses

Certaines questions de cet exercice impliquent de compléter le code source du compilateur.

5.1 Améliorer l'ALU et modifier le chemin de données de façon à supporter les instructions logiques (catégorie `001`).

J'ai ajouté à l'ALU le support de `mul / muli`, des opérations logiques telles que `not`, `lsr`, `or` et `and`.

Pour cela, j'ai décidé de changer les bits de contrôle en les faisant correspondre aux bits d'opération et de drapeaux de l'instruction et par conséquent, j'ai eu besoin d'un multiplexeur dans les cas où l'opération nécessite une somme ou une addition mais n'en est pas une à proprement parler.

On a donc :

instruction	op	flags
not	001	00
lsr	001	01
or	001	10
and	001	11
add / addi	010	0X
sub / subi	010	1X
mul / muli	011	0X
jeq	110	00
jle	110	01
jlt	110	10
jne	110	11

5.2 Écrire un code qui lit un entier saisi au clavier.

```
1  init:          ldi r0, 49 # r0 <- 49 = '1'
2                   ldi r1, 57 # r1 <- 57 = '9'
3
4  loop:         in r7 # r7 <- getchar()
5
6                   jlt r7, r0, loop # if (r7 < r0) goto loop (if r7 < '1' )
7                   jlt r1, r7, loop # if (r1 < r7) goto loop (if r7 > '9')
8
9                   out r7 # putchar() <- r7
10
11 end:         jmp end
```

FIGURE 24 – `../asm/q5_2/int_read.s`

On lit le caractère écrit par l'utilisateur au clavier dans `r7`, on le renvoie et on arrive à la fin si c'est un entier sinon on boucle jusqu'à que l'utilisateur entre un entier.

5.3 Écrire un code qui effectue la division par 10 de la valeur stockée dans r0.

On commence par diviser par 2 la valeur contenue dans `r0`. Puis, on enlève 5 jusqu'à obtenir un résultat inférieur strictement à 5 (quitte à le faire 0 fois). Et le nombre de soustractions effectuées correspond au quotient de `r0` par 10.

```
                ldi r0, 40 # r0 <- 40
                ldi r1, 0 # r1 <- 0
                ldi r2, 0 # r2 <- 0

                lsr r1, r0, 1 # r1 <- r0 >> 1 = r0/2

loop:         jlt r7, r1, output # if (r7 < r5) goto output
                subi r1, r0, 5 # r1 <- r0-5
                addi r3, r2, 1 # r3 <- r2+1
                addi r2, r3, 0 # r2 <- r3
                jmp loop # goto loop

output:         out r3
end:          jmp end # goto end
```

FIGURE 25 – `../asm/q5_3/div10.s`

5.4 Écrire un code qui lit deux entiers au clavier, puis affiche le résultat de leur multiplication (modulo 256) à l'écran.

```
init:          in r0 # r0 <- getchar()
                ldi r1, 0 # r1 <- 0; iter_index
                  in r2 # r2 <- getchar()
                  # r7 temp reg

int:          subi r7, r0, 24 # r7 <- r0-24
                  subi r0, r7, 24 # r0 <- r7-24; convert r0 to its integer value

                  subi r7, r2, 24 # r7 <- r2-24;
                  subi r2, r7, 24 # r2 <- r7-24; convert r2 to its integer value

loop:         add r7, r1, r0 # r7 <- r1+r0
                add r1, r7, 0 # r1 <- r7
                add r7, r3, 1 # r7 <- r3+1
                add r3, r7, 0 # r3 <- r7
                jeq r3, r2, end # if (r3 == r2) goto end
                jmp loop # goto loop


char:          addi r2, r0, 24
                  addi r0, r2, 24 # convert back r0 to its ascii value

                out r7 # putchar <- r7
end:     jmp end # goto end
```

FIGURE 26 – `../asm/q5_4/read2_mul.s`

5.5 Compléter le support des instructions `ld` et `st` afin de gérer la partie `imm5` de l'instruction.

```
14  #   Suppress old instruction Ld #       -    | Load  of (int * int) (** rd, rs *)
15  #   Suppress old instruction st #       -    | Store of (int * int) (** rs, rd *)
```

FIGURE 27 – `instr 1`

Suppression des anciennes instructions `ld` et `st` du type *instr* de `asm.ml`.

```
18  #   Add new instruction ld      #       +    | Load  of (int * int * int ) (** rd, rs, imm5 *)
19  #   Add new instruction st      #       +    | Store of (int * int * int ) (** rs, rd, imm5 *)
```

FIGURE 28 – `instr 2`

Ajout des nouvelles instructions (supportant les `imm5`) `ld` et `st` au type *instr* de `asm.ml`.

```
40  #   Sup. ld old dump instrction #       -| Load (rd, rs) ->
41  #       "                       #       -       Printf.printf "r%d <- MEM[r%d]\n" rd rs
42  #   Sup. st old dump instrction #       -| Store (rs, rd) ->
43  #       "                       #       -       Printf.printf "MEM[r%d] <- r%d\n" rs rd
```

FIGURE 29 – `dump_instr 1`

Suppression de l'ancien *dump* de `ld` et `st`.

```
48  #   Add ld dump instruction     #       +| Load (rd, rs, im) ->
49  #       "                       #       +       Printf.printf "r%d <- MEM[r%d+%d]\n" rd rs im
50  #   Add st dump instruction     #       +| Store (rd, rs, im) ->
51  #       "                       #       +       Printf.printf "MEM[r%d+%d] <- r%d\n" rs rd im
```

FIGURE 30 – `dump_instr 2`

Ajout du nouveau *dump* de `ld` et `st`.

```
78  #   Old ld asm2bin command      #       -    | Load (rd,rs) ->
79  #       "                       #       -           instr_to_bin_type1b 0b100 0 1 rd rs 0
80  #   Old st asm2bin command      #       -    | Store (rs,rd) ->
81  #       "                       #       -           instr_to_bin_type1b 0b100 0 0 rd rs 0
```

FIGURE 31 – `op 1`

Suppression des anciennes opérations `ld` et `st` dans *op*.

```
86  #   Add ld asm2bin command      #       +    | Load (rd,rs,v) ->
87  #       "                       #       +           if -16 <= v && v <= 15 then
88  #       "                       #       +                   instr_to_bin_type1b 0b100 0 1 rd rs v
89  #       "                       #       +           else
90  #       "                       #       +               failwith ("Load in memory: Bad value imm5")
91  #   Add st asm2bin command      #       +    | Store (rs,rd,v) ->
92  #       "                       #       +           if -16 <= v && v <= 15 then
93  #       "                       #       +                   instr_to_bin_type1b 0b100 0 0 rd rs v
94  #       "                       #       +           else
95  #       "                       #       +               failwith ("Load in memory: Bad value imm5")
```

FIGURE 32 – `op 2`

Ajout des nouvelles opérations `ld` et `st` dans *op*.

```
118 #   Sup. old declaration ld     #       -    | Load  of (int * int) (** rd, rs *)
119 #   Sup. old declaration st     #       -    | Store of (int * int) (** rs, rd *)
```

FIGURE 33 – `asm.mli 1`

Suppression des anciennes déclarations `ld` et `st` dans le type de *asm.mli.*.

```
122 #   Add declaration ld          #       +    | Load  of (int * int * int ) (** rd, rs, imm5 *)
123 #   Add declaration st          #       +    | Store of (int * int * int ) (** rs, rd, imm5 *)
```

FIGURE 34 – `asm.mli 2`

Ajout des nouvelles déclarations `ld` et `st` dans le type de *asm.mli.*.

```
194 #  Suppress old syntax LD       #       -  | LD REG COMA REG              { Load ($2, $4)  }
```

FIGURE 35 – `parser.mly 1`

Suppression de l'ancienne syntaxe de `ld` dans `parser.mly`.

```
196 #  Suppress old syntax ST       #       -  | ST REG COMA REG              { Store ($4, $2) }
```

FIGURE 36 – `parser.mly 2`

Suppression de l'ancienne syntaxe de `st` dans `parser.mly`.

```
200  #  Add new syntax for LD        #       +  | LD REG COMA REG COMA INT      { Load ($2, $4, $6)  }
```

FIGURE 37 – `parser.mly 3`

Ajout de la nouvelle syntaxe de `ld` dans `parser.mly`.

```
202  #  Add new syntax for ST        #       +  | ST REG COMA REG COMA INT      { Store ($4, $2, $6) }
```

FIGURE 38 – `parser.mly 4`

Ajout de la nouvelle syntaxe de `st` dans `parser.mly`.

5.6 Ajouter le support de l'instruction `jr`.

J'ai ajouté un bit de contrôle *do_jr* et l'arrivée de ce dernier comme bit de contrôle au compteur de programme.

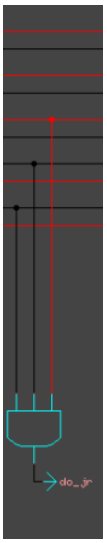

FIGURE 39 – Bit de contrôle *do_jr*.

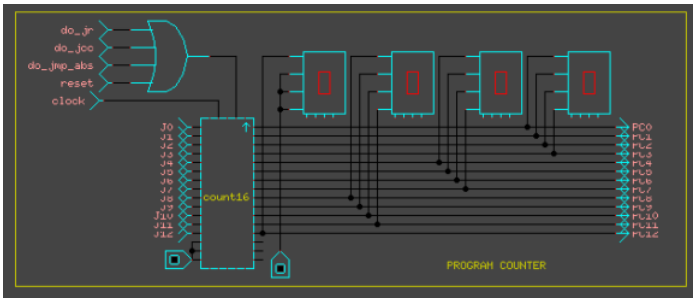

FIGURE 40 – Ajout du bit de contrôle *do_jr* au compteur de programme.

2 ALU (Arithmetic Logic Unit)

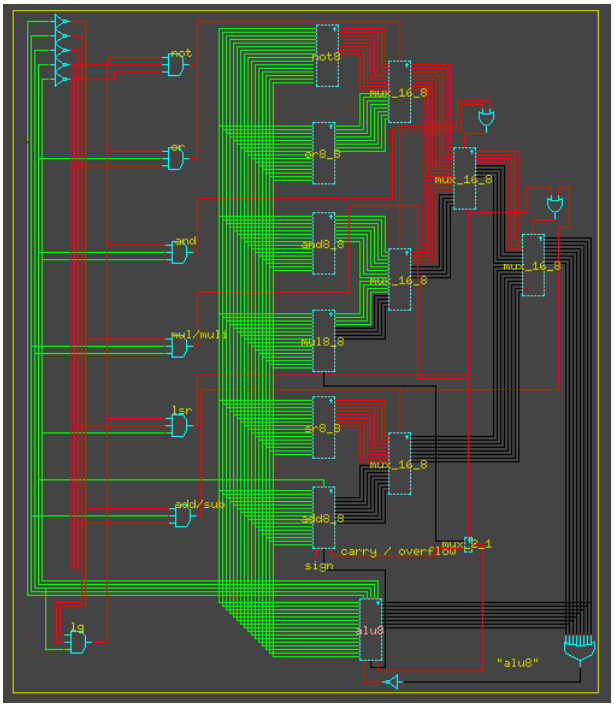

FIGURE 41 – L'unité arithmétique et logique.

2.1 Bits de contrôle

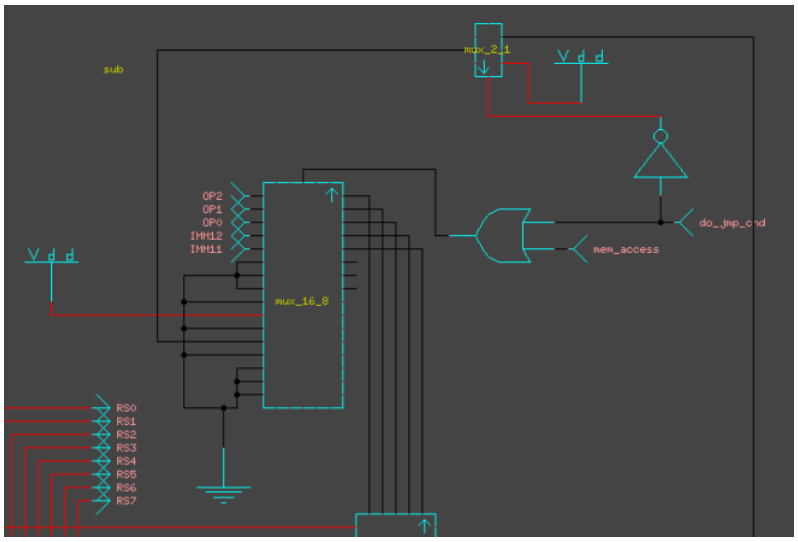

FIGURE 42 – Bits de contrôle de l'ALU

2.1.1 op (3 bits : *op2*, *op1* et *op0*)

Ces 3 bits de contrôle correspondent aux 3 bits d'opération de l'instruction sauf dans le cas des sauts conditionnels (où l'on force une soustraction) ou de l'accès en mémoire (où l'on force une addition ou une soustraction en fonction du décalage).

2.1.2 flags (2 bits : *f1* et *f0*)

Ces 2 bits de contrôle correspondent aux 2 bits de drapeaux de l'instruction sauf dans le cas des sauts conditionnels (où l'on force une soustraction) ou de l'accès en mémoire (où l'on force une addition ou une soustraction en fonction du décalage).

2.2 Bits de sortie

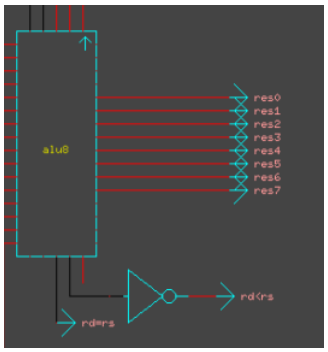

FIGURE 43 – Bits de sortie de l'ALU

2.2.1 Zero

Ce bit de sortie vaut 1 si et seulement si la sortie de l'ALU vaut 0.

2.2.2 Sign

Ce bit de sortie vaut 1 si et seulement la sortie est de signe négatif (ce qui ne peut arriver que lors de soustractions).

2.2.3 Carry / Overflow

Ce bit de sortie vaut 1 si et seulement si il y a eu dépassement de la valeur possible sur 8 bits (ou de manière équivalente, il y a une retenue qui dépasse les 8 bits présents).

3 Bits globaux

3.1 Bits de contrôle

instr	op	flags	*op2*	*op1*	*op0*	*f1*	*f0*
nop	000	00	0	0	0	0	0
ldi	000	01	0	0	0	0	1
	000	10	0	0	0	1	0
	000	11	0	0	0	1	1
not	001	00	0	0	1	0	0
lsr	001	01	0	0	1	0	1
or	001	10	0	0	1	1	0
and	001	11	0	0	1	1	1
addi	010	00	0	1	0	0	0
add	010	01	0	1	0	0	1
subi	010	10	0	1	0	1	0
sub	010	11	0	1	0	1	1
muli	011	00	0	1	1	0	0
mul	011	01	0	1	1	0	1
st	100	00	0	1	0	1	0
ld	100	01	0	1	0	1	0
out	100	10	1	0	0	1	0
in	100	11	1	0	0	1	1
jr	101	00	1	0	1	0	0
	101	01	1	0	1	0	1
	101	10	1	0	1	1	0
	101	11	1	0	1	1	1
jeq	110	00	0	1	0	1	0
jle	110	01	0	1	0	1	0
jlt	110	10	0	1	0	1	0
jne	110	11	0	1	0	1	0
jmp	111	00	1	1	1	0	0
jmp	111	01	1	1	1	0	1
jmp	111	10	1	1	1	1	0
jmp	111	11	1	1	1	1	1

3.1.1 Sauts

instr	*do_jcc*	*do_jmp_abs*	*do_jmp_cnd*	*do_jr*	*is_jmp*	*jcc*
nop	0	0	0	0	0	0
ldi	0	0	0	0	0	0
	0	0	0	0	0	0
	0	0	0	0	0	0
not	0	0	0	0	0	0
lsr	0	0	0	0	0	0
or	0	0	0	0	0	0
and	0	0	0	0	0	0
addi	0	0	0	0	0	0
add	0	0	0	0	0	0
subi	0	0	0	0	0	0
sub	0	0	0	0	0	0
muli	0	0	0	0	0	0
mul	0	0	0	0	0	0
st	0	0	0	0	0	0
ld	0	0	0	0	0	0
out	0	0	0	0	0	0
in	0	0	0	0	0	0
jr	0	0	0	1	1	0
	X	X	X	X	X	X
	X	X	X	X	X	X
	X	X	X	X	X	X
jeq	1	0	1	0	1	1
jle	1	0	1	0	1	1
jlt	1	0	1	0	1	1
jne	1	0	1	0	1	1
jmp	0	1	0	0	1	0
jmp	0	1	0	0	1	0
jmp	0	1	0	0	1	0
jmp	0	1	0	0	1	0

Les bits de contrôle liés aux sauts sont : *do_jcc*, *do_jmp_abs*, *do_jmp_cnd*, *do_jr*, *is_jmp*, *jcc*, *jeq*, *jle*, *jlt*, *jne*, *jmp_abs*, et *jr*.

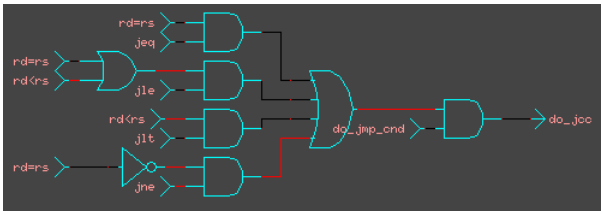

FIGURE 44 – Bit de contrôle *do_jcc*.

do_jcc vaut 1 si et seulement si l'instruction courante est un saut conditionnel et que la condition de saut est vérifiée.

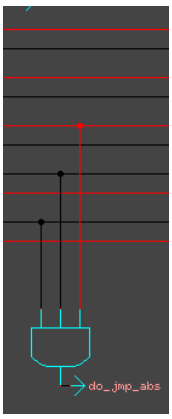

FIGURE 45 – Bit de contrôle *do_jmp_abs*.

do_jmp_abs vaut 1 si et seulement si l'instruction courante est un saut absolu.

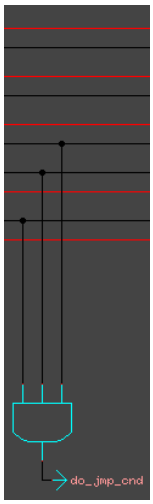

FIGURE 46 – Bit de contrôle *do_jmp_cnd*.

do_jmp_cnd vaut 1 si et seulement si l'instruction courante est un saut conditionnel.

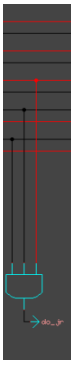

FIGURE 47 – Bit de contrôle *do_jr*.

do_jr vaut 1 si et seulement si l'instruction courante est `jr`.

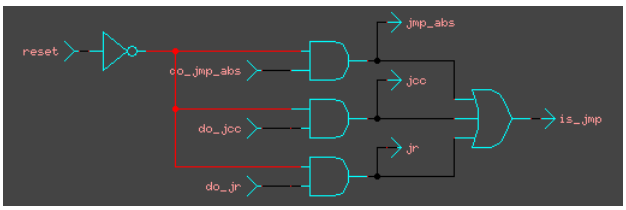

FIGURE 48 – Bit de contrôle *is_jmp*.

is_jmp vaut 1 si et seulement si le processeur lit une instruction de saut (`jr`, `jmp` ou tout saut conditionnel).

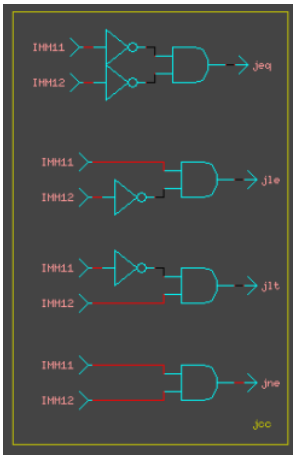

FIGURE 49 – Bits de contrôles liés aux instructions spécifiques des sauts conditionnels.

jeq*, *jle*, *jlt* et *jne valent 1 respectivement quand l'instruction éponyme est l'instruction courante.

3.1.2 Utilisation de la RAM

Les bits de contrôle liés à l'utilisation de la RAM sont : *mem_access*, *write_mem* et *read_mem*.

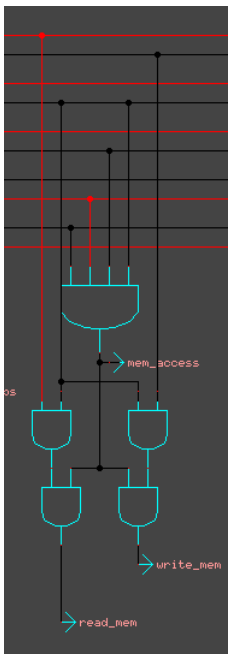

FIGURE 50 – Bits de contrôles liés à l'utilisation de la mémoire.

mem_access vaut 1 si et seulement si l'instruction courante est `ld` ou `st`.

write_mem vaut 1 si et seulement si l'instruction courante est `st`.

read_mem vaut 1 si et seulement si l'instruction courante est `ld`.

3.1.3 Autre

instr	*src2_is_rd*	*res_imm*	*arg2_imm*
nop	X	X	X
ldi	0	1	0
	X	X	X
	X	X	X
not	0	0	X
lsr	0	0	1
or	0	0	0
and	0	0	0
addi	0	0	1
add	0	0	0
subi	0	0	1
sub	0	0	0
muli	0	0	1
mul	0	0	0
st	1	0	1
ld	1	0	1
out	1	0	X
in	0	0	X
jr	1	0	0
	X	X	X
	X	X	X
	X	X	X
jeq	1	1	0
jle	1	1	0
jlt	1	1	0
jne	1	1	0
jmp	0	1	1
jmp	0	1	1
jmp	0	1	1
jmp	0	1	1

Les autres bits de contrôles sont : *src2_is_rd*, *res_imm* et *arg2_imm*.

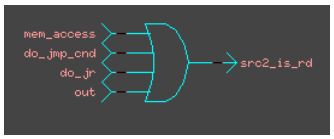

FIGURE 51 – Bit de contrôle *src2_is_rd*.

src2_is_rd vaut 1 si et seulement si l'instruction utilise le registre `rd` pour y réaliser des opérations. C'est le cas pour les sauts conditionnels ou de registres, les accès en mémoire ou l'écriture sur l'écran.

res_imm vaut 1 si et seulement si l'instruction a pour résultat l'immédiat donné en argument. C'est le cas pour `ldi`.

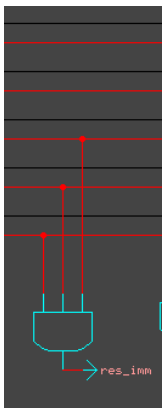

FIGURE 52 – Bit de contrôle *res_ imm*.

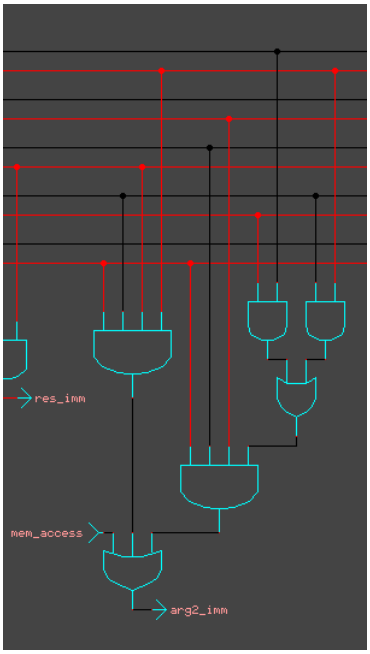

FIGURE 53 – Bit de contrôle *arg2_ imm*.

arg2_imm vaut 1 si et seulement si le deuxième argument de l'instruction est un immédiat. C'est le cas pour toutes les opérations dont le nom se termine par i ou `lsr`.

3.2 Bits de statut

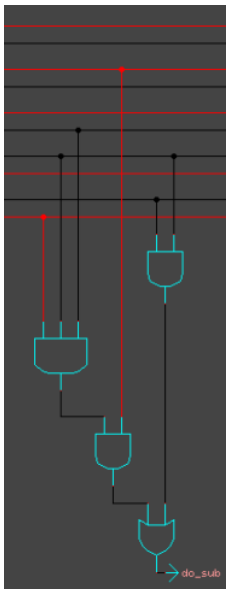

FIGURE 54 – Bit de contrôle *do_sub*.

do_sub vaut 1 si et seulement si l'instruction courante demande l'utilisation de l'ALU en mode soustracteur.

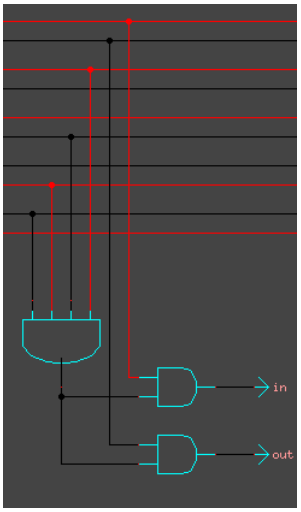

FIGURE 55 – Bits de contrôle d'entrée-sortie.

in vaut 1 si et seulement si l'instruction courante est `in`.

out vaut 1 si et seulement si l'instruction courante est `out`.

rd*=*rs* et *rd*<*rs valent respectivement 1 lorsque la valeur contenue dans `rd` est égale (respectivement strictement inférieure) à la valeur contenue dans `rs`.

4 Changements apportés au compilateur

4.1 asm.ml / asm.mli

4.1.1 asm.mli

4.1.2 asm.ml

```
5  #                              #        type instr =
6  #                              #             Nop
7  #                              #           | Ldi   of (int * int)   (** rd, imm8 *)
8  #   Add instruction not        #       +   | Not   of (int * int)   (** rd, rs *)
9  #   Add instruction lsr        #       +   | Lsr   of (int * int * int)        (** rd, rs, imm5 *)
10 #   Add instruction or         #       +   | Or    of (int * int * int)        (** rd, rs, rt *)
11 #   Add instruction and        #       +   | And   of (int * int * int)        (** rd, rs, rt *)
12 #                              #           | Add   of (int * int * int * bool) (** rd, rs, rt, sub? *)
13 #                              #           | Addi  of (int * int * int * bool) (** rd, rs, imm5, sub? *)
14 #   Suppress old instruction Ld #       -   | Load  of (int * int) (** rd, rs *)
15 #   Suppress old instruction st #       -   | Store of (int * int) (** rs, rd *)
16 #   Add instruction mul        #       +   | Mul   of (int * int * int) (** rd, rs, rt *)
17 #   Add instruction muli       #       +   | Muli  of (int * int * int) (** rd, rs, imm5 *)
18 #   Add new instruction ld     #       +   | Load  of (int * int * int ) (** rd, rs, imm5 *)
19 #   Add new instruction st     #       +   | Store of (int * int * int ) (** rs, rd, imm5 *)
20 #                              #           | In    of int          (** rd *)
21 #                              #           | Out   of int          (** rs *)
22 #   Add instruction jr         #       +   | Jr    of (int * int) (** rd, rs *)
23 #                              #           | CJmp  of (int * int * label * cond) (** rd, rs, addr, cond *)
24 #                              #           | Jmp   of label (** label name *)
25 #                              #
```

FIGURE 56 – `Type instr`

Ajout dans l'implémentation du type *instr* (correspondant aux instructions) des instructions `not`, `lsr`, `or`, `and` et `jr`.

Modification dans l'implémentation du type *instr* des instructions `ld` et `st` pour gérer la partie `imm5` de ces instructions.

```
26  #                                #       @@ -48,20 +55,29 @@
27  #                                #        let dump_instr = fun i -> match i with
28  #                                #        | Nop -> Printf.printf ".\n"
29  #                                #        | Ldi (r,v) -> Printf.printf "r%d <- %d\n" r v
30  #   Add not dump instruction     #       +| Not (rd,rs) -> Printf.printf "r%d <- not(r%d)\n" rd rs
31  #   Add lsr dump instruction     #       +| Lsr (rd,rs,s) -> Printf.printf "r%d <- r%d >> %d\n" rd rs s
32  #   Add or dump instruction      #       +| Or  (rd,rs,rt) -> Printf.printf "r%d <- r%d || r%d\n" rd rs rt
33  #   Add and dump instruction     #       +| And (rd,rs,rt) -> Printf.printf "r%d <- r%d && r%d\n" rd rs rt
34  #                                #        | Add (rd,rs,rt,b) ->
35  #                                #               let c = if b then '-' else '+' in
36  #                                #               Printf.printf "r%d <- r%d %c r%d\n" rd rs c rt
37  #                                #        | Addi (rd,rs,v,b) ->
38  #                                #               let c = if b then '-' else '+' in
39  #                                #               Printf.printf "r%d <- r%d %c %d\n" rd rs c v
40  #   Sup. ld old dump instrction  #       -| Load (rd, rs) ->
41  #       "                        #       -       Printf.printf "r%d <- MEM[r%d]\n" rd rs
42  #   Sup. st old dump instrction  #       -| Store (rs, rd) ->
43  #       "                        #       -       Printf.printf "MEM[r%d] <- r%d\n" rs rd
44  #   Add mul dump instruction     #       +| Mul (rd,rs,rt) ->
45  #       "                        #       +       Printf.printf "r%d <- r%d x r%d mod 256\n" rd rs rt
46  #   Add muli dump instruction    #       +| Muli (rd,rs,v) ->
47  #       "                        #       +       Printf.printf "r%d <- r%d x %d mod 256\n" rd rs v
48  #   Add ld dump instruction      #       +| Load (rd, rs, im) ->
49  #       "                        #       +       Printf.printf "r%d <- MEM[r%d+%d]\n" rd rs im
50  #   Add st dump instruction      #       +| Store (rd, rs, im) ->
51  #       "                        #       +       Printf.printf "MEM[r%d+%d] <- r%d\n" rs rd im
52  #                                #        | In rd ->
53  #                                #               Printf.printf "r%d <- getchar()\n" rd
54  #                                #        | Out rs ->
55  #                                #               Printf.printf "putchar(r%d)\n" rs
56  #   Add jr dump instruction      #       +| Jr (rd,rs) -> Printf.printf "goto r%d + r%d x 256\n" rs rd
```

FIGURE 57 – `Fonction dump_instr`

Ajout dans la fonction de *dump* à la compilation *dump_ instr* des instructions `not`, `lsr`, `or`, `and`, `mul`, `muli` et `jr`.

Modification dans la fonction de *dump* à la compilation *dump_ instr* des instructions `ld` et `st` pour gérer la partie `imm5` de ces instructions.

```
57  #                                #         | CJmp (rd,rs,lbl,cd) ->
58  #                                #                 let op = match cd with
59  #                                #                         | LT -> "<"
60  #                                #       @@ -137,20 +153,40 @@
61  #                                #                         instr_to_bin_type2 0b000 0 0 0 0
62  #                                #             | Ldi (r,v) ->
63  #                                #                         instr_to_bin_type2 0b000 0 1 r v
64  #   Add not dump instruction     #       +     | Not (rd,rs) ->
65  #       "                        #       +             instr_to_bin_type1b 0b001 0 0 rd rs 0
66  #   Add lsr dump instruction     #       +     | Lsr (rd,rs,s) ->
67  #       "                        #       +             instr_to_bin_type1b 0b001 0 1 rd rs s
68  #   Add or dump instruction      #       +     | Or  (rd,rs,rt) ->
69  #       "                        #       +             instr_to_bin_type1 0b001 1 0 rd rs rt
70  #   Add and dump instruction     #       +     | And (rd,rs,rt) ->
71  #       "                        #       +             instr_to_bin_type1 0b001 1 1 rd rs rt
72  #                                #             | Add (rd,rs,rt,b) ->
73  #                                #                     let f1 = if b then 1 else 0 in
74  #                                #                     instr_to_bin_type1 0b010 f1 1 rd rs rt
75  #                                #             | Addi (rd,rs,v,b) ->
76  #                                #                     let f1 = if b then 1 else 0 in
77  #                                #                     instr_to_bin_type1b 0b010 f1 0 rd rs v
78  #   Old ld asm2bin command       #       -     | Load (rd,rs) ->
79  #       "                        #       -             instr_to_bin_type1b 0b100 0 1 rd rs 0
80  #   Old st asm2bin command       #       -     | Store (rs,rd) ->
81  #       "                        #       -             instr_to_bin_type1b 0b100 0 0 rd rs 0
82  #   Add mul asm2bin command      #       +     | Mul (rd,rs,rt) ->
83  #       "                        #       +             instr_to_bin_type1 0b011 0 1 rd rs rt
84  #   Add muli asm2bin command     #       +     | Muli (rd,rs,v) ->
85  #       "                        #       +             instr_to_bin_type1b 0b011 0 0 rd rs v
86  #   Add ld asm2bin command       #       +     | Load (rd,rs,v) ->
87  #       "                        #       +             if -16 <= v && v <= 15 then
88  #       "                        #       +                     instr_to_bin_type1b 0b100 0 1 rd rs v
89  #       "                        #       +             else
90  #       "                        #       +                 failwith ("Load in memory: Bad value imm5")
91  #   Add st asm2bin command       #       +     | Store (rs,rd,v) ->
92  #       "                        #       +             if -16 <= v && v <= 15 then
93  #       "                        #       +                     instr_to_bin_type1b 0b100 0 0 rd rs v
94  #       "                        #       +             else
95  #       "                        #       +                 failwith ("Load in memory: Bad value imm5")
96  #                                #             | In rd ->
97  #                                #                     instr_to_bin_type2 0b100 1 1 rd 0
98  #                                #             | Out rs ->
99  #                                #                     instr_to_bin_type2 0b100 1 0 rs 0
100 #   add jr asm2bin command       #       +     | Jr (rd,rs) ->
101 #       "                        #       +             instr_to_bin_type1b 0b111 0 0 rd rs 0
102 #                                #             | CJmp (rd,rs,lbl,cd) ->
103 #                                #                     let (f1,f2) = match cd with
104 #                                #                                 | EQ -> (0,0)
```

FIGURE 58 – `../patch/digcomp_jb.patch.dbu`

Ajout dans *op* des instructions `not`, `lsr`, `or`, `and`, `mul`, `muli` et `jr`.
Modification dans *op* des instructions `ld` et `st` pour gérer la partie `imm5` de ces instructions.

```
#==============================#       diff -urN digcomp_orig/asm.mli digcomp_new/asm.mli
# In asm.mli                   #       --- digcomp_orig/asm.mli        2018-03-05 15:26:51.000000000 +0100
#    -------                   #       +++ digcomp_new/asm.mli        2023-01-03 08:01:41.617126449 +0100
#                              #       @@ -35,12 +35,19 @@
#                              #        type instr =
#                              #            Nop
#                              #          | Ldi   of (int * int)   (** rd, imm8 *)
#   Add declaration not        #       +  | Not   of (int * int)   (** rd, rs *)
#   Add declaration lsr        #       +  | Lsr   of (int * int * int)        (** rd, rs, uimm5 *)
#   Add declaration or         #       +  | Or    of (int * int * int)        (** rd, rs, rt *)
#   Add declaration and        #       +  | And   of (int * int * int)        (** rd, rs, rt *)
#                              #          | Add   of (int * int * int * bool) (** rd, rs, rt, sub? *)
#                              #          | Addi  of (int * int * int * bool) (** rd, rs, uimm5, sub? *)
#   Sup. old declaration ld    #       -  | Load  of (int * int) (** rd, rs *)
#   Sup. old declaration st    #       -  | Store of (int * int) (** rs, rd *)
#   Add declaration mul        #       +  | Mul   of (int * int * int) (** rd, rs, rt *)
#   Add declaration muli       #       +  | Muli  of (int * int * int) (** rd, rs, uimm5 *)
#   Add declaration ld         #       +  | Load  of (int * int * int ) (** rd, rs, imm5 *)
#   Add declaration st         #       +  | Store of (int * int * int ) (** rs, rd, imm5 *)
#                              #          | In    of int         (** rd *)
#                              #          | Out   of int         (** rs *)
#   Add declaration jr         #       +  | Jr    of (int * int)   (** rd, rs *)
#                              #          | CJmp  of (int * int * label * cond) (** rs, rt, addr, cond *)
#                              #          | Jmp   of label (** addr *)
#                              #
```

FIGURE 59 – `../patch/digcomp_jb.patch.dbu`

Ajout dans la déclaration du type *instr* des instructions `not`, `lsr`, `or`, `and`, `mul`, `muli` et `jr`.

Modification dans la déclaration du type *instr* des instructions `ld` et `st` pour gérer la partie `imm5` de ces instructions.

```
#                              #       @@ -102,7 +102,7 @@
#                              #                   let _ = Printf.printf "--------\n" in
#                              #                   let _ = List.iteri
#                              #                               (fun idx instr ->
#   Sup old line diag digcomp  #       -                            let _ = Printf.printf "%4d: " idx in
#   Add hexa on diag digcomp   #       +                            let _ = Printf.printf "%4d [x%04x]: " idx idx in
#                              #                                    dump_instr instr
#                              #                               )
#                              #                               code
```

FIGURE 60 – `../patch/digcomp_jb.patch.dbu`

Ajout du numéro en ligne en hexadécimal dans le dump.

```
145 #                               #       @@ -8,14 +8,21 @@
146 #                               #            [ "nop",  NOP;
147 #                               #              "ldi",  MOV;
148 #                               #              "mov",  MOV;
149 #   Link word not to token NOT  #       +      "not",  NOT;
150 #   Link word lsr to token LSR  #       +      "lsr",  LSR;
151 #   Link word or to token OR    #       +      "or",   OR;
152 #   Link word and to token AND  #       +      "and",  AND;
153 #                               #              "add",  ADD;
154 #                               #              "addi", ADDI;
155 #                               #              "sub",  SUB;
156 #                               #              "subi", SUBI;
157 #   Link word mul to token MUL  #       +      "mul",  MUL;
158 #   Link word muli to token MULI#       +      "muli", MULI;
159 #                               #              "ld",   LD;
160 #                               #              "st",   ST;
161 #                               #              "in",   IN;
162 #                               #              "out",  OUT;
163 #   Link word jr to token JR    #       +      "jr",   JR;
164 #                               #              "jlt",  JLT;
165 #                               #              "jle",  JLE;
166 #                               #              "je",   JEQ;
```

FIGURE 61 – `../patch/digcomp_jb.patch.dbu`

Ajout dans les reconnaissances des instructions en token de `lexer.mll` des instructions `not`, `lsr`, `or`, `and`, `mul`, `muli` et `jr`.

```
#                               #       @@ -4,7 +4,7 @@
#                               #
#                               #
#                               #        %token <int> INT
#  Add new tokens               #       -%token NOP MOV ADD ADDI SUB SUBI JMP LD ST IN OUT JLE JLT JEQ JNE
#   NOT LSR OR AND MUL & MULI   #       +%token NOP MOV NOT LSR OR AND ADD ADDI SUB SUBI MUL MULI JMP LD ST IN OUT JR JLE JLT J
#                               #        %token COMA COLON LPAR RPAR
#                               #        %token <int> REG
#                               #        %token <string> LABEL
#                               #       @@ -27,6 +27,10 @@
#                               #          | NOP                          { Nop }
#                               #          | MOV REG COMA INT             { Ldi ($2,$4) }
#                               #          | MOV REG COMA REG             { Addi ($2,$4,0,false)  }
#  Add syntax for NOT           #       +  | NOT REG COMA REG                 { Not ($2,$4) }
#  Add syntax for LSR           #       +  | LSR REG COMA REG COMA INT          { Lsr ($2,$4,$6) }
#  Add syntax for OR            #       +  | OR  REG COMA REG COMA REG          { Or  ($2,$4,$6) }
#  Add syntax for AND           #       +  | AND REG COMA REG COMA REG          { And ($2,$4,$6) }
#                               #          | ADD REG COMA REG COMA REG     { Add  ($2,$4,$6,false) }
#                               #          | ADDI REG COMA REG COMA INT    { assert (0<=$6 && $6<32); Addi ($2,$4,$6,false) }
#                               #          | ADD REG COMA REG COMA INT     { assert (0<=$6 && $6<32); Addi ($2,$4,$6,false) }
#                               #       @@ -34,12 +38,15 @@
#                               #          | SUB REG COMA REG COMA REG     { Add  ($2,$4,$6,true)  }
#                               #          | SUB REG COMA REG COMA INT     { assert (0<=$6 && $6<32); Addi ($2,$4,$6,true)  }
#                               #          | SUBI REG COMA REG COMA INT    { assert (0<=$6 && $6<32); Addi ($2,$4,$6,true)  }
#  Suppress old syntax LD       #       -  | LD REG COMA REG               { Load ($2, $4)  }
#  Suppress old syntax MOV      #       -  | MOV REG COMA LPAR REG RPAR    { Load ($2, $5)  }
#  Suppress old syntax ST       #       -  | ST REG COMA REG               { Store ($4, $2) }
#  Suppress old syntax MOV      #       -  | MOV LPAR REG RPAR COMA REG    { Store ($3, $6) }
#  Add syntax for MUL           #       +  | MUL REG COMA REG COMA REG     { Mul  ($2,$4,$6) }
#  Add syntax for MULI          #       +  | MULI REG COMA REG COMA INT    { assert (0<=$6 && $6<32); Muli ($2,$4,$6) }
#  Add new syntax for LD        #       +  | LD REG COMA REG COMA INT      { Load ($2, $4, $6)  }
#  Add new syntax for MOV       #       +  | MOV REG COMA LPAR REG RPAR    { Load ($2, $5, 0)  }
#  Add new syntax for ST        #       +  | ST REG COMA REG COMA INT      { Store ($4, $2, $6) }
#  Add new syntax for MOV       #       +  | MOV LPAR REG RPAR COMA REG    { Store ($3, $6, 0) }
#                               #          | IN REG                       { In $2  }
#                               #          | OUT REG                      { Out $2 }
#  Add syntax for JR            #       +  | JR REG COMA REG               { Jr  ($2,$4) }
#                               #          | cjump REG COMA REG COMA LABEL { CJmp ($2,$4,$6,$1) }
#                               #          | JMP LABEL                    { Jmp $2 }
#                               #        ;
```

FIGURE 62 – `../patch/digcomp_jb.patch.dbu`

Ajout des syntaxes *instr* des instructions `not`, `lsr`, `or`, `and`, `mul`, `muli` et `jr`.

Modification des syntaxes des instructions `ld`, `st` et `mov` pour gérer la partie `imm5` de ces instructions.

5 Codes de tests

Dans cette partie, un X signifie que l'on ne connaît pas la valeur.

5.1 Vérification des opérations logiques (`not`, `lsr`, `or` et `and`)

```
1  init:   mov r0, 16 # r0 <- 16
2          mov r1, 48 # r1 <- 48
3          mov r2, 0 # r0 <- 0
4
5  lg:     not r7, r2 # r7 <- 255
6          lsr r6, r0, 1 # r6 <- 8
7          or r5, r0, r1 # r5 <- 48
8          and r4, r0, r1 # r4 <- 16
9
10 end:    jmp end
```

FIGURE 63 – `../tests/lg.s`

Ce code teste les instructions `not`, `lsr`, `or` et `and`.
État des registres en fin d'exécution :

registre	r0	r1	r2	r3	r4	r5	r6	r7
valeur	16	48	0	X	16	48	8	255

5.2 Vérification des opérations arithmétiques (`add / addi`, `sub / subi`, `mul / muli`)

FIGURE 64 – `../tests/arith.s`

Le programme `arith.s` teste les fonctions `add / addi`, `sub / subi` et `mul / muli`.
État des registres en fin d'exécution :

registre	r0	r1	r2	r3	r4	r5	r6	r7
valeur	0	1	2	X	20	2	1	-1

5.3 Vérification de l'accès en lecture et écriture à la RAM (`ld, st`)

```
        ldi r7, 53 # r7 <- 53
        ldi r0, 66 # r0 <- 66
        ldi r1, 76 # r1 <- 76
        ldi r2, 86 # r2 <- 86

        st r0, r7, 0 # MEM[r7+0] <- r0=66
        st r1, r7, 1 # MEM[r7+1] <- r1=76
        st r2, r7, -4 # MEM[r7-4] <- r2=86

        ld r3, r7, 0 # r3 <- MEM[r7+0] = 66
        ld r3, r7, 1 # r3 <- MEM[r7+1] = 76
        ld r3, r7, 2 # r3 <- MEM[r7+2] = 86

        ldi r1, 256 # r1 <- 256

        st r1, r7, 1 # MEM[r7+1] <- r1 = 256

        ld r3, r7, 0 # r3 <- MEM[r7+0] = 66
        ld r3, r7, 1 # r3 <- MEM[r7+1] = 256
        ld r3, r7, 2 # r3 <- MEM[r7+2] = 86
```

FIGURE 65 – `../tests/mem.s`

Le programme `mem.s` teste l'accès en lecture (`ld`) et écriture (`st`) à la mémoire RAM.
État des registres en fin d'exécution :

registre	r0	r1	r2	r3	r4	r5	r6	r7
valeur	0	1	2	X	20	2	1	-1=255

5.4 Vérification de l'entrée-sortie (`in, out`)

```
        ldi r7, -1 # r7 <- 255
loop:   in r0 # r0 <- getchar()
        jeq r0, r7, loop # if (r0 == r7), goto loop
        out r0 # putchar() <- r0
end:    jmp end

```

FIGURE 66 – `../tests/io.s`

Le programme `io.s` teste les entrées-sorties sur le processeur à l'aide du clavier et de l'écran présents dans `io.lgf`.
État des registres en fin d'exécution :

registre	r0	r1	r2	r3	r4	r5	r6	r7
valeur	getchar()	X	X	X	X	X	X	255

5.5 Vérification du saut en registre (`jr`)

Le programme `jr.s` fait un saut en registre (`jr`) vers l'adresse 1 (donc la deuxième instruction).
État des registres en fin d'exécution :

registre	r0	r1	r2	r3	r4	r5	r6	r7
valeur	r0	1	X	X	X	X	X	X

```
        mov r0, 0 # r0 <- 0
        mov r1, 1 # r1 <- 1

        jr r0, r1 # goto (256r0 + r1 = 256*0 + 1 = 1)

end: jmp end
```

FIGURE 67 – ../tests/jr.s

```
        ldi r1, 42 # r1 <- 42
        out r1 # putchar() <- r1=42

        subi r0, r1, 10 # r0 <- 42-10=32
        out r0 # putchar() <- r0=32

        jeq r0, r1, j1 # if (r0 == r1) goto j1
        jlt r0, r1, j1 # if (r0 < r1) goto j1
        jle r0, r1, j1 # if (r0 <= r1) goto j1

        add r2, r0, r1 # r2 <- 32+42=74
        out r2 # putchar() <- r2=74
        jeq r2, r1, end # if (r2 == r1) goto end

j1:         sub r1, r0, r0 # r1 <- r0-r0 = 0

        ldi r7, 106 # r7 <- 106 ; register r7 used for msg; 106 = 'j'
        out r7 # putchar() <- r7
        ldi r7, 49 # r7 <- 49 ; 49 = '1'
        out r7 # putchar() <- r7
        # msg should be 'j1'

        out r1 # putchar() <- r1=0

end:        jmp end
```

FIGURE 68 – ../tests/jcc.s

5.6 Vérification des sauts conditionnels (jeq, jle, jlt et jne)

Le programme jcc.s teste jeq, jle et jlt.
État des registres en fin d'exécution :

registre	r0	r1	r2	r3	r4	r5	r6	r7
valeur	42	0	74	X	X	X	X	49

5.7 Vérification du saut dans le passé

```
        ldi r7, -1 # r7 <- -1
loop:       in r0 # r0 <- getchar()
        jeq r0, r7, loop # if (r0 == r7) goto loop (-1)
        out r0 # putchar() <- r0
end: jmp end
```

FIGURE 69 – ../tests/pj.s

Le programme pj.s teste les sauts conditionnels (ici jeq) dans le passé.
État des registres en fin d'exécution :

registre	r0	r1	r2	r3	r4	r5	r6	r7
valeur	getchar()	X	X	X	X	X	X	-1=255

```
1         ldi r7, -1 # r7 <- -1
2  loop:       in r0 # r0 <- getchar()
3         jeq r0, r7, goto # if (r0 == r7) goto 'goto' (+2)
4         jmp end
5  goto:       jmp loop # goto loop (-3)
6  end:       jmp end
```

FIGURE 70 – `../tests/fj.s`

5.8 Vérification du saut dans le futur

Le programme `fj.s` teste les sauts conditionnels (ici `jeq`) dans le futur.
État des registres en fin d'exécution :

registre	r0	r1	r2	r3	r4	r5	r6	r7
valeur	getchar()	X	X	X	X	X	X	-1=255

Conclusion

Toutes les questions ont été réalisées mais le processeur peut encore être amélioré avec d'autres types d'instructions comme par exemple la gestion de prédicats.