Projet d'architecture matrielle

ARMAgeddon [1]

Pierre KOEBELIN

16 avril 2018

1. J'aime les jeux de mots

Résumé

L'objectif de ce projet était la réalisation d'un mini-processeur. 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`
- `nop`
- `lsr` : une version simplifiée qui ne décale le nombre en entrée que d'un bit
- `or` avec prédicat
- `and` avec prédicat
- `addi`
- `subi`
- `add` avec prédicat
- `sub` avec prédicat
- `cmpi`
- `cmp` avec prédicat
- `out`
- `in`
- `jeq`
- `jle`
- `jlt`
- `jne`
- `jmp`

Table des matières

1 ALU

Cet ALU permet de faire de nombreuses opérations sur des entiers sur 8 bits, comme l'addition ou la soustraction d'entiers signés et non-signés, ainsi que le complément à 1, l'incrémentation, l'opposé ou encore les ET et OU binaires.

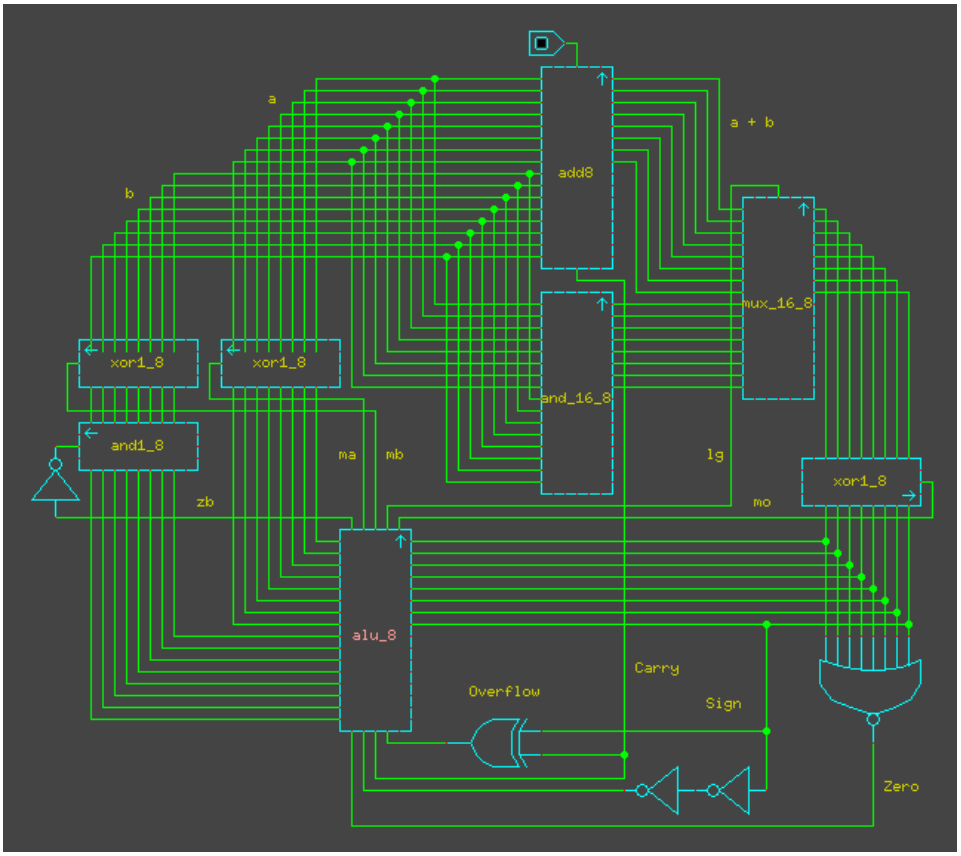

FIGURE 1 – ALU à 5 bits de contrôle et 4 bits de sortie

Bits de contrôle et de sortie Cet ALU prend en entrée deux entiers codés sur 8 bits et est géré par cinq bits de contrôle : `zb`, `ma`, `mb`, `lg` et `mo`. Il propose en sortie le résultat de l'opération sur 8 bits, ainsi que quatre autres bits de sortie : `Zero`, `Sign`, `Carry` et `Overflow`.

1.1 Bits de contrôle

Ces bits de contrôle sont au nombre de cinq et permettent de réaliser différentes opérations sur les entrées `a` et `b`.

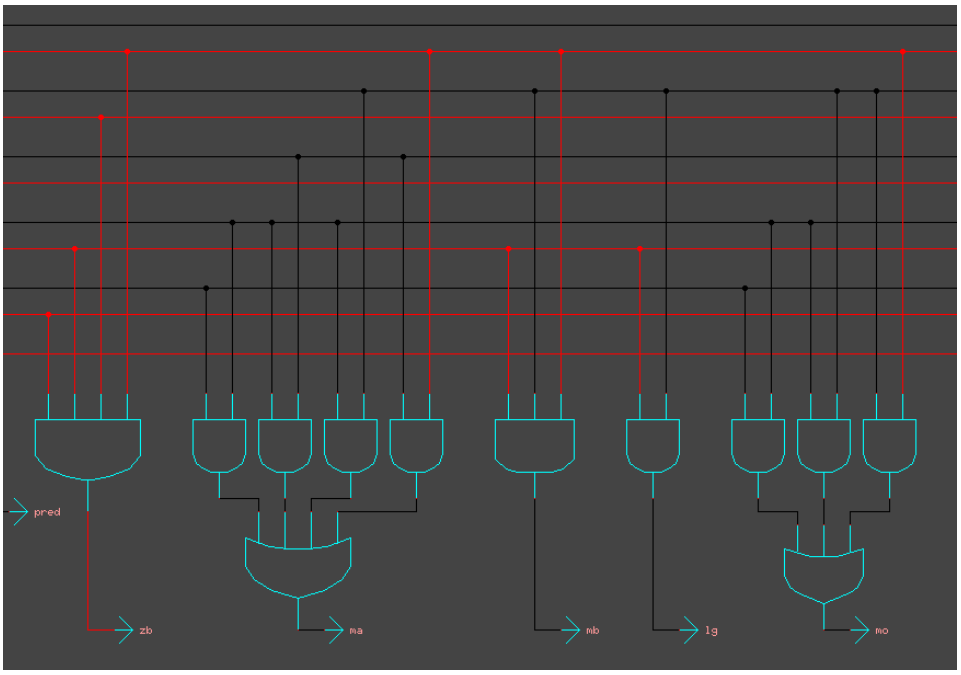

FIGURE 2 – Calcul des bits de contrôle de l'ALU

1.1.1 `zb`

Ce bit de contrôle gère le bloc `and1_8` situé sur les bits de `b`, et permet de mettre `b` à 0 ou non avant de passer dans le bloc `xor1_8`.

1.1.2 `ma`

Ce bit de contrôle gère le bloc `xor1_8` situé sur les bits de `a`, et permet de donner son complément à 1 avant d'être utilisé par `add8` et `and_16_8`.

1.1.3 `mb`

Ce bit de contrôle gère le bloc `xor1_8` situé sur les bits de `b`, et permet de donner son complément à 1 avant d'être utilisé par `add8` et `and_16_8`.

1.1.4 `lg`

Ce bit de contrôle gère le multiplexeur `mux_16_8` permettant de retourner le résultat de `add8` ou celui de `and_16_8`.

1.1.5 `mo`

Ce bit de contrôle gère le bloc `xor1_8` situé après le multiplexeur `mux_16_8`, et permet retourner le résultat de `add8` et `and_16_8` ou son complément à 1.

Voici le tableau répertoriant les valeurs des bits de contrôle pour chaque instruction :

instr	op	flags	zb	ma	mb	lg	mo
	abc	de					
nop	000	00	X	X	X	X	X
ldi	000	01	X	X	X	X	X
	000	10	X	X	X	X	X
	000	11	X	X	X	X	X
not	001	00	1	1	X	X	0
lsr	001	01	0	0	0	0	0
or	001	10	0	1	1	1	1
and	001	11	0	0	0	1	0
addi	010	00	0	0	0	0	0
add	010	01	0	0	0	0	0
subi	010	10	0	1	0	0	1
sub	010	11	0	1	0	0	1
muli	011	00	X	X	X	X	X
mul	011	01	X	X	X	X	X
compi	011	10	0	1	0	0	1
comp	011	11	0	1	0	0	1
st	100	00	0	0	0	0	0
ld	100	01	0	0	0	0	0
out	100	10	X	X	X	X	X
in	100	11	X	X	X	X	X
jr	101	00	X	X	X	X	X
	101	01	X	X	X	X	X
	101	10	X	X	X	X	X
	101	11	X	X	X	X	X
jeq	110	00	0	1	0	0	1
jle	110	01	0	1	0	0	1
jlt	110	10	0	1	0	0	1
jne	110	11	0	1	0	0	1
jmp	111	00	X	X	X	X	X
jmp	111	01	X	X	X	X	X
jmp	111	10	X	X	X	X	X
jmp	111	11	X	X	X	X	X

On peut ainsi obtenir le tableau de Karnaugh suivant :

	de	00	01	11	10
abc					
000		X X X X X	X X X X X	X X X X X	X X X X X
001		1 1 X X 0	0 0 0 0 0	0 0 0 1 0	0 1 1 1 1
011		X X X X X	X X X X X	0 1 0 0 1	0 1 0 0 1
010		0 0 0 0 0	0 0 0 0 0	0 1 0 0 1	0 1 0 0 1
110		0 1 0 0 1	0 1 0 0 1	0 1 0 0 1	0 1 0 0 1
111		X X X X X	X X X X X	X X X X X	X X X X X
101		X X X X X	X X X X X	X X X X X	X X X X X
100		0 0 0 0 0	0 0 0 0 0	0 0 0 0 0	0 0 0 0 0

1.2 Bits de sortie

Cest bits de sortie donnent des informations supplémentaires sur l'opération entre les nombres `a` et `b`.

1.2.1 Zero

Ce bit de sortie est à 1 lorsque le résultat de l'opération est nul. Dans le cas d'une soustraction, il indique donc une égalité.

1.2.2 Sign

Ce bit de sortie est identique au bit de poids le plus fort dans le résultat de l'opération, et indique son signe dans le cas d'un entier signé.

1.2.3 Carry

Ce bit de sortie est la retenue en sortie du bloc `add_8`. Il correspond à un dépassement de capacité en non-signé.

1.2.4 Overflow

Ce bit correspond à un dépassement de capacité en signé.

2 Bits de contrôle globaux

instr	op	flags	write_reg	arg2_imm	src2_is_rd	res_imm
nop	000	00	0	X	X	X
ldi	000	01	1	X	X	1
	000	10	0	X	X	X
	000	11	0	X	X	X
not	001	00	1	X	X	0
lsr	001	01	1	1	X	0
or	001	10	1	0	0	0
and	001	11	1	0	0	0
addi	010	00	1	1	X	0
add	010	01	1	0	0	0
subi	010	10	1	1	X	0
sub	010	11	1	0	0	0
muli	011	00	1	1	X	0
mul	011	01	1	0	0	0
compi	011	10	0	1	X	X
comp	011	11	0	0	0	X
st	100	00	0	1	1	X
ld	100	01	1	1	X	0
out	100	10	0	X	1	X
in	100	11	1	X	X	0
jr	101	00	0	1	1	X
	101	01	0	X	X	X
	101	10	0	X	X	X
	101	11	0	X	X	X
jeq	110	00	0	1	1	X
jle	110	01	0	1	1	X
jlt	110	10	0	1	1	X
jne	110	11	0	1	1	X
jmp	111	00	0	X	X	X
jmp	111	01	0	X	X	X
jmp	111	10	0	X	X	X
jmp	111	11	0	X	X	X

3 Autres bits de contrôle

Ces bits de contrôle sont spécifiques à certaines instructions, et doivent seulement être mis à 1 dans ces cas là.

3.1 Instructions de saut

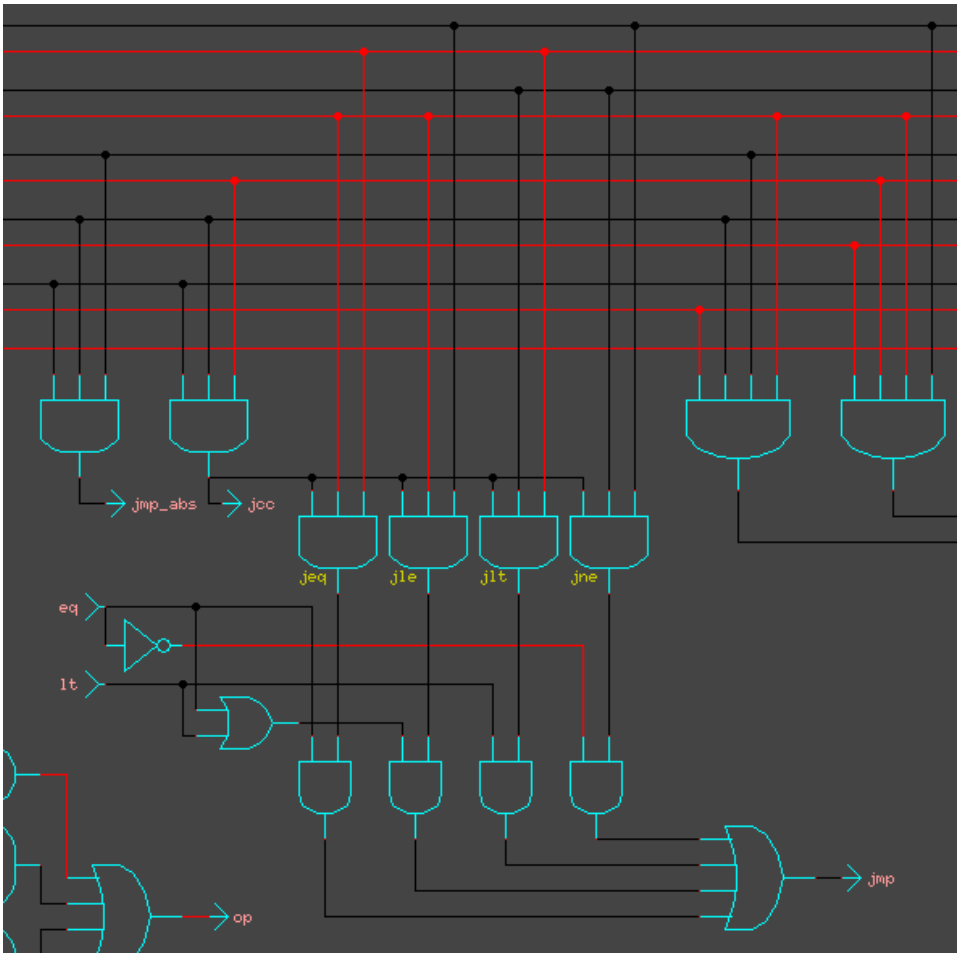

FIGURE 3 – Calcul des bits de sauts absolus ou conditionnels

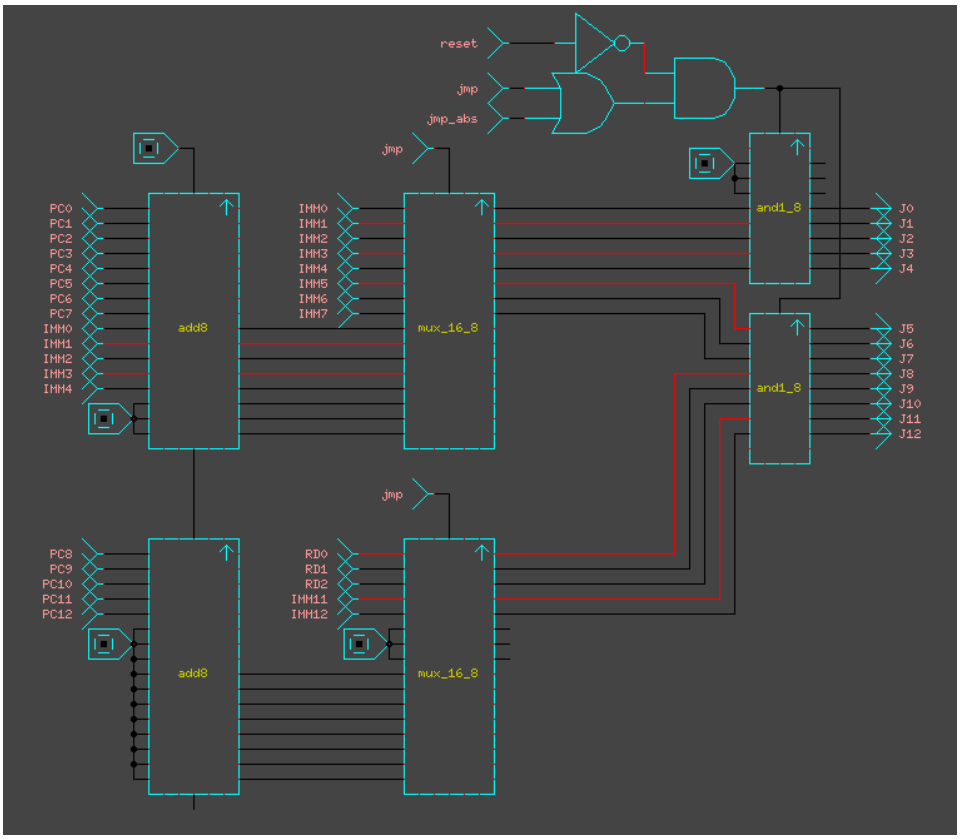

FIGURE 4 – Gestion des sauts absolus ou conditionnels

3.2 Instructions `in` et `out`

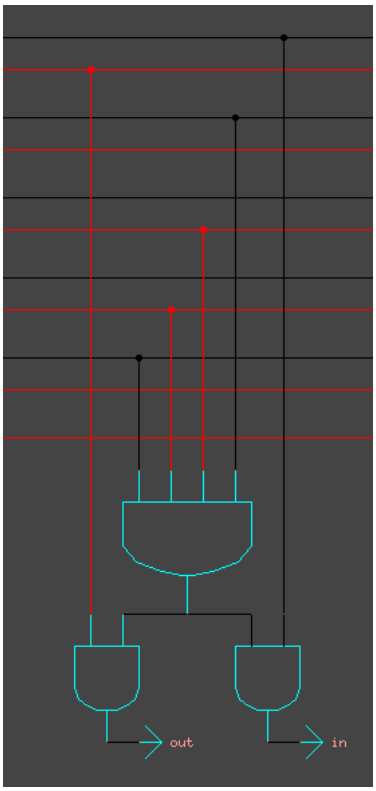

FIGURE 5 – Calcul des bits `in` et `out`

3.3 Instruction `lsr`

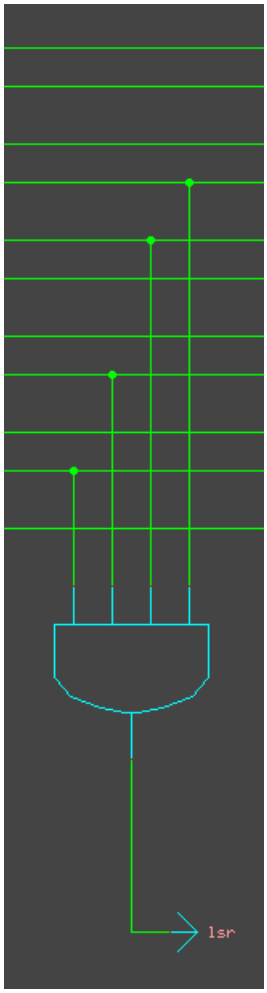

FIGURE 6 – Calcul d bit `lsr`

4 Modifications apportées au processeur

4.1 Introduction du registre `flags`

Pour enregistrer le résultat d'une comparaison (instructions `cmpi` et `cmp`), on utilise un registre trois bits, dans lequel on écrit uniquement lorsqu'une de ces instructions est utilisée. Au début, les trois bits sont à `0`.

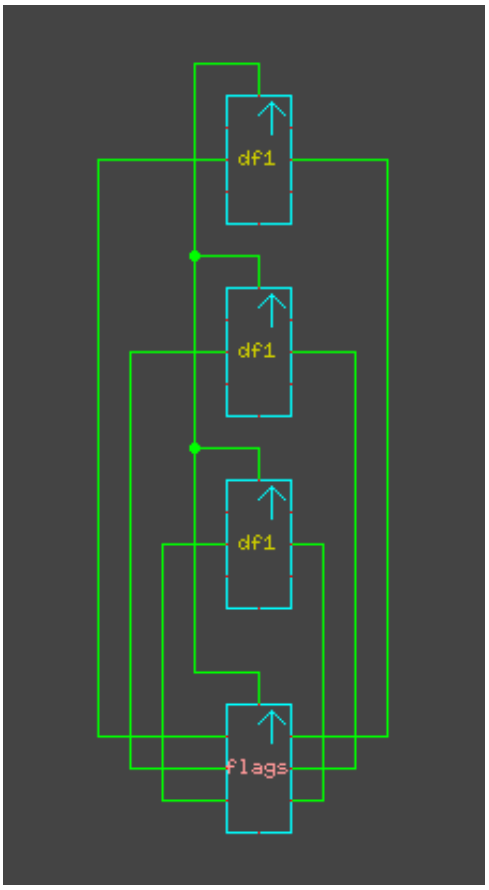

FIGURE 7 – Registre `flags`

4.2 Gestion des prédicats

Les instructions de type `1a` possèdent 2 bits actuellement inutilisés. Il est donc possible d'y attribuer des valeurs pour compléter ces instructions.
Le but ici est d'autoriser leur exécution en fonction de la dernière comparaison entre deux valeurs, dont le résultat est stocké dans le registre `flags`.

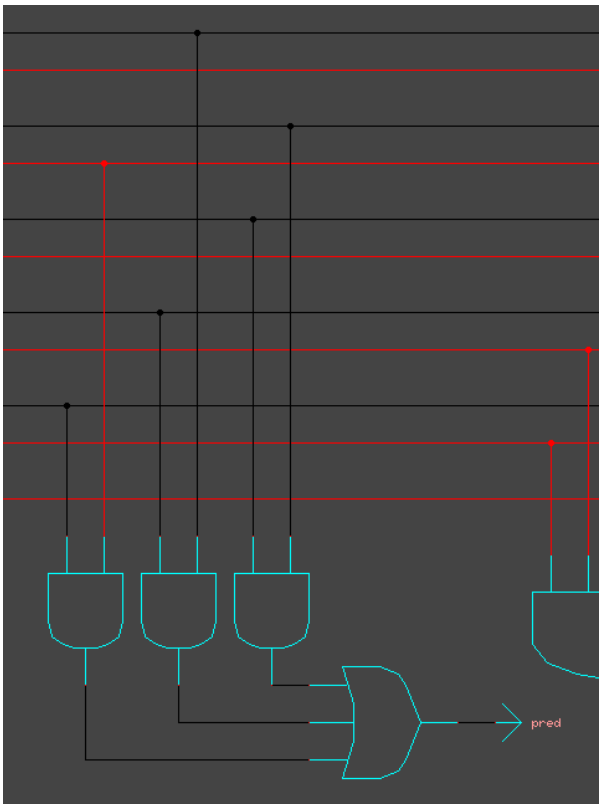

FIGURE 8 – Calcul du bit `pred`

La sortie `pred` permet de savoir si les deux bits de poids plus faibles sont un prédicat, afin de savoir si l'instruction doit être réalisée ou non. Dans le cas contraire, il ne s'agit tout simplement pas d'une instruction de type `1a`.

4.3 Implémentation des instructions `cmpi` et `cmp`

Spécificité de `cmpi` L'instruction `cmpi` est la seule à être de type `2` tout en nécessitant une lecture du registre `#rd`. C'est pourquoi un multiplexeur a été ajouté afin de lire le registre `#rd` plutôt que `#rs`, et de pouvoir le comparer avec `IMM8` via l'ALU.
Il suffit alors d'avoir deux bits `cmp` et `cmpi` pour gérer chaque cas (le bit `cmpi` étant inclu dans `cmp`).

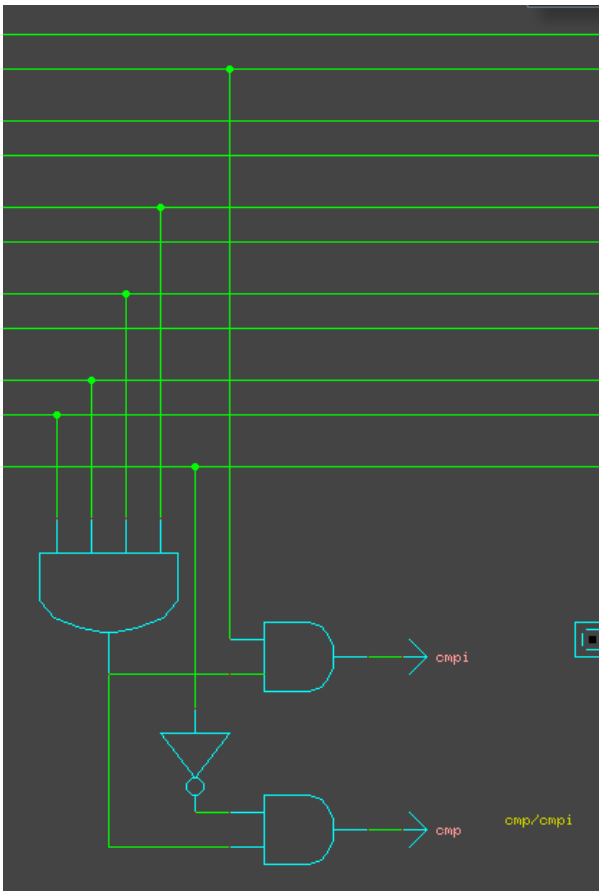

FIGURE 9 – Calcul des bits `cmp` et `cmpi`

4.4 Comparaison et calcul du bit `op`

À partir de la dernière comparaison dont le résultat est enregistré dans le registre `flags`, et le prédicat de l'instruction, on calcule le bit `op`, qui autorise ou non l'écriture dans le banc de registre ou la mémoire. L'écran n'est pas concerné, l'instruction correspondant étant de type `2`.

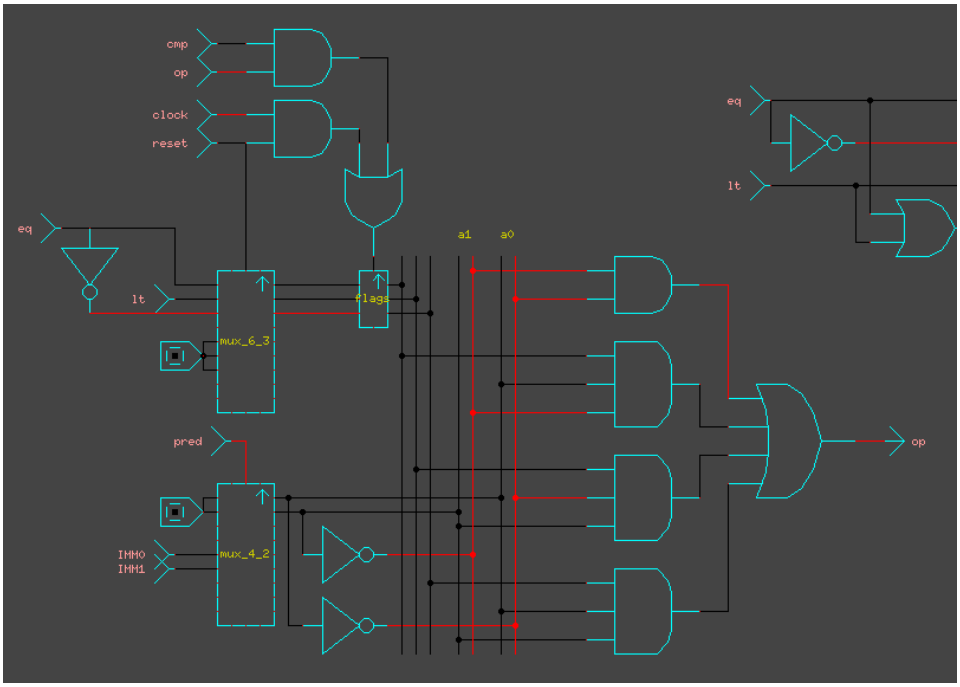

FIGURE 10 – Comparaison et calcul du bit `op`

4.5 Utilisation du bit `op`

Le bit `op` vient en complément des bits de contrôle `write_reg` et `write_mem` ; il est indispensable pour pouvoir écrire dans le banc de registre ou dans la mémoire. Cela permet d'ignorer les instructions dont le prédicat ne correspond pas au dernier résultat des instructions `cmpi` et `cmp`.

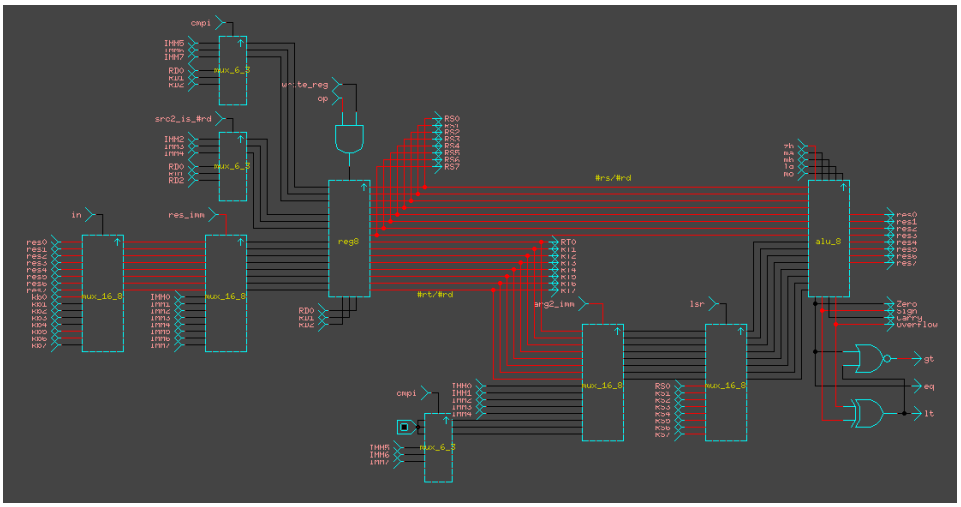

FIGURE 11 – Utilisation du bit `op` sur le banc de registres

.

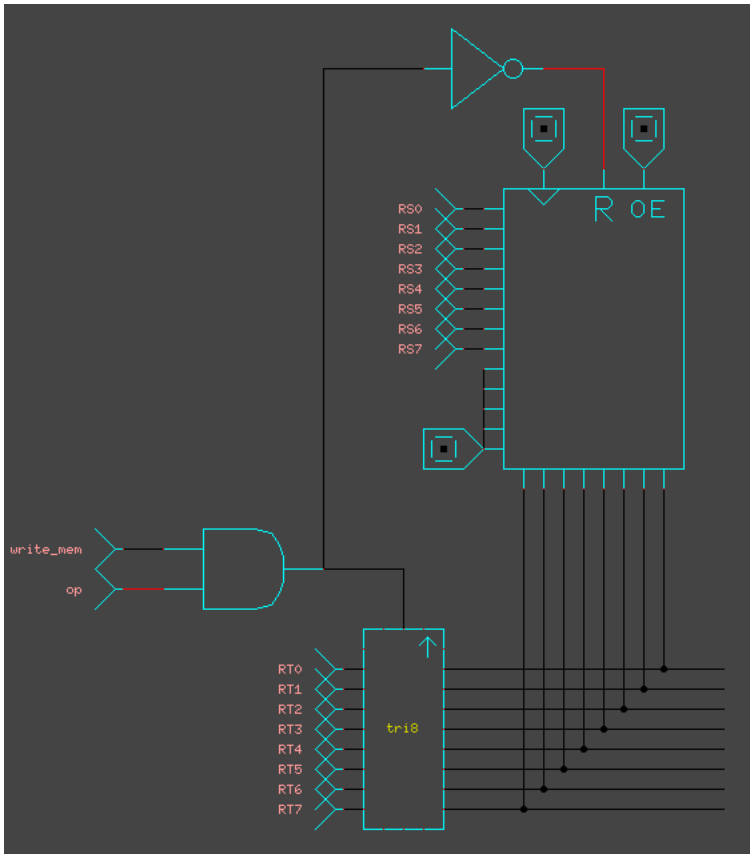

FIGURE 12 – Utilisation du bit `op` sur la RAM

5 Modifications apportées au compilateur

5.1 Changement des instructions `ld` et `st` en type `1a`

5.1.1 *asm.ml*

`instr_to_bin` On gère le cas où `i` correspond à `Load` ou `Store`, au lieu d'appeler la fonction `instr_to_bin_type1b`, on appelle la fonction `instr_to_bin_type1`.

```
    let instr_to_bin = fun i caddr assoc ->
        match i with
        | Load (rd,rs) ->
-           instr_to_bin_type1b 0b100 0 1 rd rs 0
+           instr_to_bin_type1 0b100 0 1 rd rs 0
        | Store (rs,rd) ->
-           instr_to_bin_type1b 0b100 0 0 rd rs 0
+           instr_to_bin_type1 0b100 0 0 rd rs 0
```

5.2 Ajout des instructions `cmpi` et `cmp`

5.2.1 *lexer.mll*

`keyword_table` On ajoute les lignes correspondant aux instructions `cmpi` et `cmp` dans la table de hashage `keyword_table`.

```
    let keyword_table = Hashtbl.create 53
    let _ =
        List.iter (fun (kwd, tok) -> Hashtbl.add keyword_table kwd tok)
        [
+           "cmpi", CMPI;
+           "cmp",  CMP;
        ]
```

5.2.2 *parser.mly*

`code` On ajoute les lignes correspondant aux instructions `cmpi` et `cmp` dans `code`.

```
    code:
+       | CMPI REG COMA INT { assert (-128<=$4 && $4<=127); Cmpi ($2, $4) }
+       | CMP REG COMA REG  { Cmp ($2, $4) }
    ;
```

5.2.3 *asm.ml* et *asm.mli*

`type instr` On commence par ajouter `Cmpi` et `Cmp` au type `instr`.

```
    type instr =
+       | Cmpi  of (int * int)          (** rd, uimm8       *)
+       | Cmp   of (int * int * int)    (** rs, rt, pred    *)
```

`dump_instr` On ajoute les cas correspondant aux instructions `cmpi` et `cmp` pour afficher leur résultat sur sortie standard.

```
    let dump_instr = fun i -> match i with
+   | Cmpi (rd, v) ->
+       Printf.printf "flags <- cmp(r%d,%d)\n" rd v
+   | Cmp (rs, rt) ->
+       Printf.printf "flags <- cmp(r%d,r%d) (%d)\n" rs rt
```

`instr_to_bin` On ajoute les cas correspondant aux instructions `cmpi` et `cmp`. On appelle alors respectivement les fonctions `instr_to_bin_type2` et `instr_to_bin_type1` qui les transformera en compte compris par *Diglog*.

```
    let instr_to_bin = fun i caddr assoc ->
        match i with
+       | Cmpi (rd,v) ->
+           instr_to_bin_type2 0b011 1 0 rd v
+       | Cmp (rs,rt) ->
+           instr_to_bin_type1 0b011 1 1 0 rs rt
```

5.3 Gestion des prédicats

5.3.1 *parser.mly*

`code` Pour chaque ligne traîtant une instruction de type `1a`, on la modifie pour que la fonction qu'elle appelle prenne un paramètre supplémentaire de type `int`, qu'on initialise à 0. De plus, pour chacune de ces lignes, on en crée une nouvelle qui nous permettra d'indiquer un prédicat pouvant aller de 0 à 3 (entier), et qui sera ajouté par le compilateur dans les instructions concernées.

```
    code:
+       | ADD REG COMA REG COMA REG COMA INT
                { assert (0<=$8 && $8<=3); Add ($2,$4,$6,false,$8)  }
-       | ADD REG COMA REG COMA REG      { Add  ($2,$4,$6,false)     }
+       | ADD REG COMA REG COMA REG      { Add  ($2,$4,$6,false,0)   }
+       | SUB REG COMA REG COMA REG COMA INT
                { assert (0<=$8 && $8<=3); Add ($2,$4,$6,true,$8)   }
-       | SUB REG COMA REG COMA REG      { Add  ($2,$4,$6,true)      }
+       | SUB REG COMA REG COMA REG      { Add  ($2,$4,$6,true,0)    }
+       | CMP REG COMA REG COMA INT
                { assert (0<=$6 && $6<=3); Cmp ($2, $4, $6)         }
-       | CMP REG COMA REG               { Cmp ($2, $4)             }
+       | CMP REG COMA REG               { Cmp ($2, $4, 0)          }
+       | LD REG COMA REG COMA INT
                { assert (0<=$6 && $6<=3); Load ($2, $4, $6)        }
-       | LD REG COMA REG                { Load ($2, $4)            }
+       | LD REG COMA REG                { Load ($2, $4, 0)         }
+       | MOV REG COMA LPAR REG RPAR COMA INT
                { assert (0<=$8 && $8<=3); Load ($2, $5, $8)        }
-       | MOV REG COMA LPAR REG RPAR     { Load ($2, $5)            }
+       | MOV REG COMA LPAR REG RPAR     { Load ($2, $5, 0)         }
+       | ST REG COMA REG COMA INT       { Store ($4, $2, $6)       }
-       | ST REG COMA REG                { Store ($4, $2)           }
+       | ST REG COMA REG                { Store ($4, $2, 0)        }
+       | MOV LPAR REG RPAR COMA REG COMA INT
                { assert (0<=$8 && $8<=3); Store ($3, $6, $8)       }
-       | MOV LPAR REG RPAR COMA REG     { Store ($3, $6)           }
+       | MOV LPAR REG RPAR COMA REG     { Store ($3, $6, 0)        }
```

```
    ;
```

5.3.2 *asm.ml*

`instr_to_bin_type1` On modifie la fonction pour qu'elle puisse prendre en paramètre un prédicat `p`, qu'elle ajoutera ensuite au code de l'instruction.

```
-   let instr_to_bin_type1 = fun c f1 f2 r1 r2 r3 ->
+   let instr_to_bin_type1 = fun c f1 f2 r1 r2 r3 p ->
        let hi = (c lsl 5) + (f1 lsl 4) + (f2 lsl 3) + r1 in
-       let lo = (r2 lsl 5) + (r3 lsl 2) in
+       let lo = (r2 lsl 5) + (r3 lsl 2) + p in
        (Printf.sprintf "%02x" hi, Printf.sprintf "%02x" lo)
```

`instr_to_bin` Les instructions gérées par la fonction prennent maintenant en compte les prédicats

```
    let instr_to_bin = fun i caddr assoc ->
        match i with
-       | Add (rd,rs,rt,b) ->
-           let f1 = if b then 1 else 0 in
-           instr_to_bin_type1 0b010 f1 1 rd rs rt
+       | Add (rd,rs,rt,b,p) ->
+           let f1 = if b then 1 else 0 in
+           instr_to_bin_type1 0b010 f1 1 rd rs rt p
-       | Cmp (rs,rt) ->
-           instr_to_bin_type1 0b011 1 1 0 rs rt
+       | Cmp (rs,rt,p) ->
+           instr_to_bin_type1 0b011 1 1 0 rs rt p
-       | Load (rd,rs) ->
-           instr_to_bin_type1 0b100 0 1 rd rs 0
+       | Load (rd,rs,p) ->
+           instr_to_bin_type1 0b100 0 1 rd rs 0 p
-       | Store (rs,rd) ->
-           instr_to_bin_type1 0b100 0 0 rd rs 0
+       | Store (rs,rd,p) ->
+           instr_to_bin_type1 0b100 0 0 rd rs 0 p
```

6 Codes de tests

6.1 Vérification du fonctionnement des prédicats

```
    ldi r1, 42
    subi r0, r1, 17
    add r2, r0, r1, 0
    sub r1, r0, r0, 2
end:    jmp end
```

../asm/1.asm

6.2 Vérification de la gestion des prédicats

```
    ldi r1, 42
    subi r0, r1, 17
    cmp r0, r1
    cmp r1, r0, 1
    add r2, r0, r1, 0
    sub r1, r0, r0, 2
end:    jmp end
```

../asm/2.asm

6.3 Vérification des instructions `in` et `out`

```
    nop
    in r0
    out r0
    in r1
    out r1
    out r0
    addi r0, r1, 2
    out r0
end:    jmp end
```

../asm/3.asm

6.4 Vérification des instructions de sauts conditionnels

```
    ldi r1, 42
    subi r0, r1, 17
    jeq r0, r1, j1
    jlt r0, r1, j1
    jle r0, r1, j1
    cmpi r1, 42
    add r2, r0, r1, 1
    jeq r2, r1, end
j1: sub r1, r0, r0, 2
    cmp r0, r0, 0
end:    jmp end
```

../asm/4.asm