Projet ARMA 2021/2022

Assim Delteil

Pour le 05/01/2022

Sommaire :

I) Description des bits de contrôles 1
II) Tableau des bits de contrôles 2
III) Description de l'ALU 3
IV) Réponse à chaque question 4

I) Description des bits de contrôles

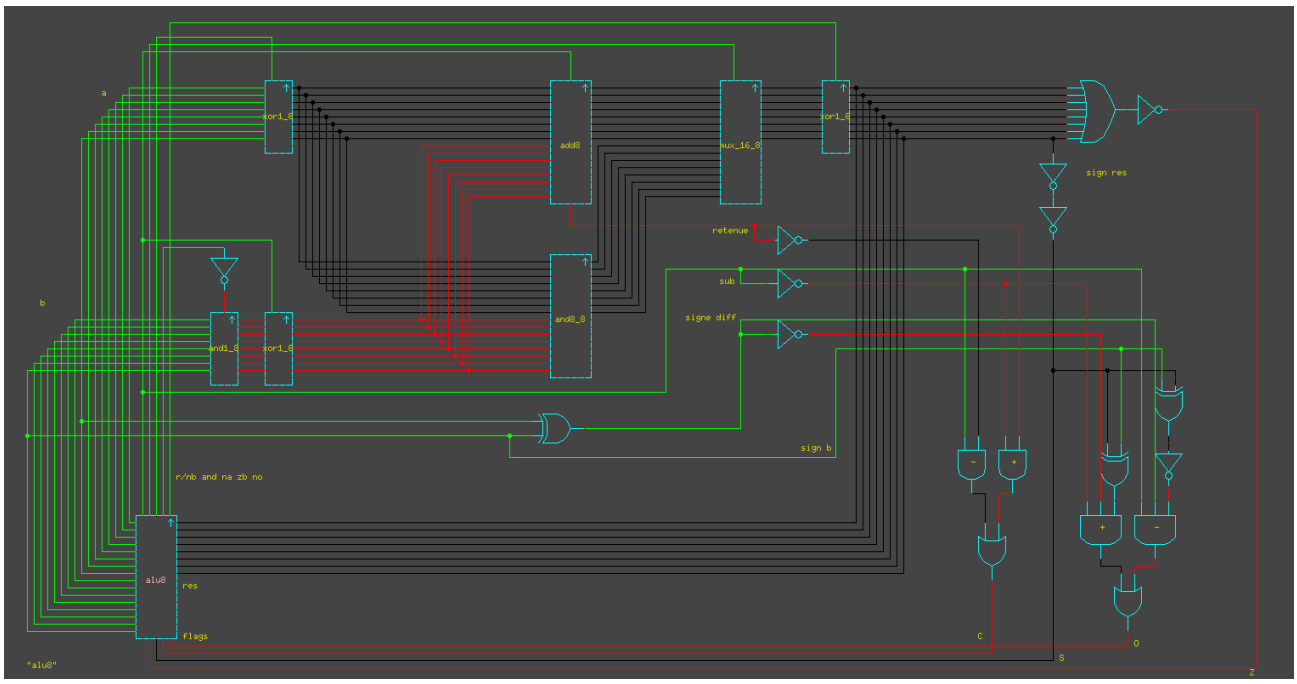
-write_reg : est à 1 quand on écrit dans le registre
-res_imm : le résultat de l'instruction est un entier (opération add, sub, not,…)
-arg2_imm : le 2ème paramètre est un immédiat et non pas un registre (type 2 et 3)
-out et in : l'instruction est out ou in
-jmp_abs : l'instruction est un saut absolu
-jcc : l'instruction est un saut conditionnel (cela n'implique pas forcément un saut)
-jmp : la condition du saut conditionnel est vraie, le saut va être effectué
-src2_is_rd : la 2ème valeur à obtenir du registre est rd et non pas rt
-write_mem : on écrit dans la mémoire (st)
-mem_to_reg : on charge de la mémoire pour la stocker dans un registre (ld)
-not et or : l'instruction est not ou or
-ALU_* : tous les bits de contrôle de l'ALU seront détaillés dans la section dédiée.

II) Tableau des bits de contrôles

														ALU_				
	write_reg	res_imm	arg2_imm	out	in	jmp_abs	jcc	jmp	src2_is_rd	write_mem	mem_to_reg	not	or	r/nb	and	na	zb	n0
nop	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
ldi	1	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
not	1	1	0	0	0	0	0	0	0	1	0	1	0	0	0	0	1	1
lsr	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
or	1	1	0	0	0	0	0	0	0	1	0	0	1	1	1	1	0	1
and	1	1	0	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0
addi	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
subi	1	1	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0
add	1	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
sub	1	1	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0
muli	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
mul	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
st	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
ld	1	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0	0
out	0	0	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0
in	1	0	1	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0
jr	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
jeq	0	0	0	0	0	0	1	?	1	0	0	0	0	1	0	0	0	0
jle	0	0	0	0	0	0	1	?	1	0	0	0	0	1	0	0	0	0
jlt	0	0	0	0	0	0	1	?	1	0	0	0	0	1	0	0	0	0
jne	0	0	0	0	0	0	1	?	1	0	0	0	0	1	0	0	0	0
jmp	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0

III) Description de l'ALU

1) Vue d'ensemble



2) Bits de contrôle

L'ALU possède 5 bits de contrôle : r/nb, and, na, zb, n0. Ceux-ci permettent d'activer, ou non, diverses circuit (xor1_8, and1_8, mux_16_8). Pour les xor et le add, si son bit de contrôle vaut 0, alors ils ne modifieront pas l'entrée.

a) Effet de chaque bit
-r/nb : met c0 du add8 à 1 et b à son complémentaire
-and : le multiplexeur renvoi le résultat du and plutot que du add
-na : met a à son complémentaire
-zb : met b à 0
-n0 : met la sortie du multiplexer à son complémentaire

b) Choisir le comportement de l'ALU
C'est en combinant ces bits de contrôles que l'on obtient de l'ALU les résultats voulus.
-a+b : aucun bit de sont activés
-a-b : r/nb
-and : and
-or : r/nb, and, na, n0
-not: zb, n0
-lsr : non implémenté

3) Flags de l'additionneur

Les flags O, C, S, Z implémenté sont pour l'additionneur, ce sont les mêmes flags que vu en cours (Overflow, Carry, Sign, Zero). Ces flags sont pertinents uniquement dans le cas de a+b ou a-b, leur valeur dans tous les autres cas ne sont pas interprétables.

IV) Réponses à chaque question

1.1) r1 : 42 et r0 : 59

1.2)

00001 001	00101010
01000 000	001 10001
111 0000	00000010

1.3) On obtient

09 2a
40 31
e0 02

Ce qui correspond à la conversion en hexadécimal du code de la 1.2)

1.4) res_imm=1 donc c'est IMM0-7 qui est transmis au registre. Et comme write_reg=1, ces valeurs sont écrites sur le registre correspondant à la valeur de RD0-2 (rd)

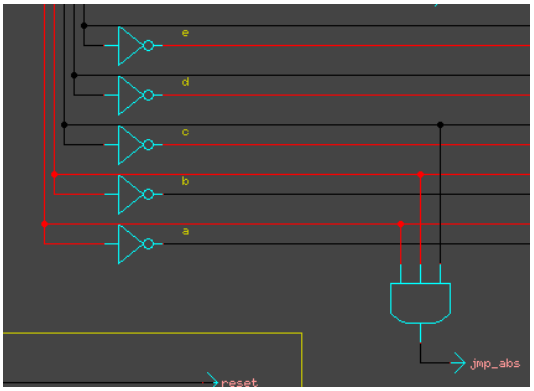
1.5) On récupère la valeur du registre IMM5-7 (rs), la valeur de IMM0-4 (imm5) est récupérée et transmise par le multiplexer car arg2_imm=1. L'ALU additionne rs et imm5 et envoi le résultat au registre (car res_imm=0) qui l'écrit à l'emplacement de RD0-2 (rd) car write_reg=1.

1.6) Pour récupérer le numéro de la prochaine ligne on met à la suite
IMM12, IMM11, RD2, RD1, RD0, IMM7, …, IMM0, ce qui donne IMM13.

1.7)

instruction	jmp_abs	write_abs	arg2_imm	res_imm
ldi	0	0	1	0
addi	0	1	1	0
jmp	1	1	0	0

1.8) jmp_abs est récupèrable comme cela :



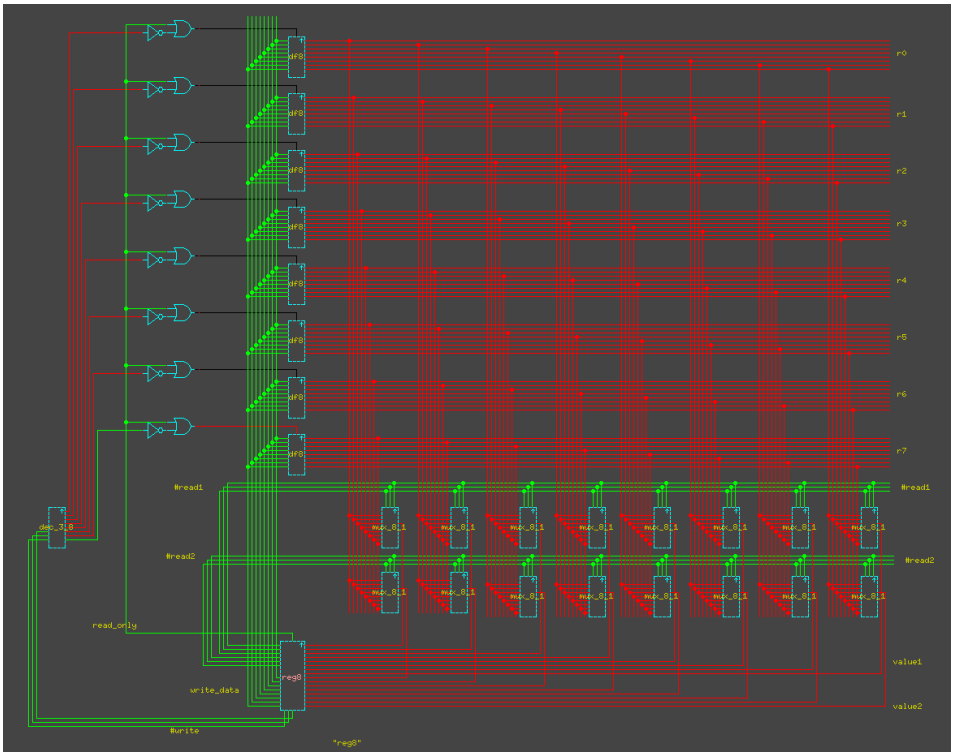
Et mettre les fils IMM12-11, RD2-0, IMM7-0 à gauche des and1_8. (Cette partie-là du circuit ayant évolué, je ne peux pas montrer le résultat)

2.1) La liste des instructions dont l'exécution nécessite d'utiliser l'ALU en tant que soustracteur est : sub, subi, jeq, jle, jlt et jne.

2.2) On met les portes or et and de telle sorte que sub vale 1 quand la partie high de l'instruction vaut 110XX ou 0101X. Et on envoit sub dans le haut de l'alu8. (Cette partie là du circuit ayant évolué, je ne peut pas montrer le résultat)

2.4)
2 lectures de registre :
-or, and, add, sub, mul : rs et rt sont à lire
-st, ld, jr, jeq, jle, jlt, jne : rs et rd sont à lire

2.3) On modifie le banc de registre comme ceci :



2.4)
2 lectures de registre :
-or, and, add, sub, mul : rs et rt sont à lire
-st, ld, jr, jeq, jle, jlt, jne : rs et rd sont à lire

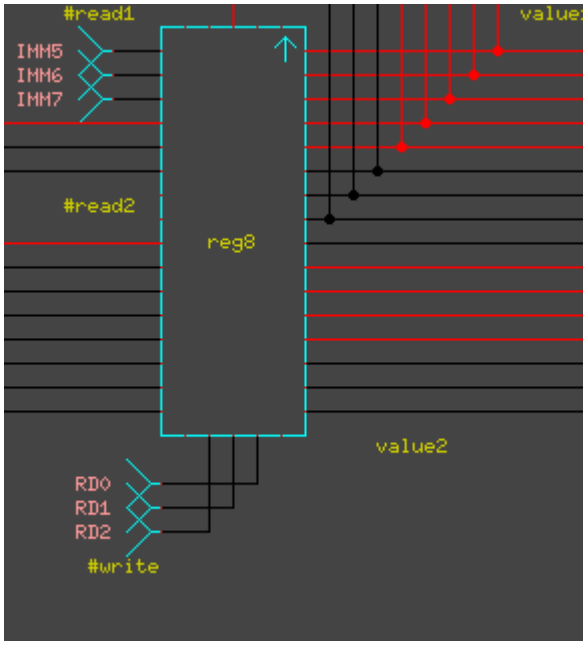
1 lecture de registre :
-not, lsr, addi, subi, muli : rs est à lire
-out : rd est à lire

nop, ldi, in et jmp ne demandent pas de lire de registre.

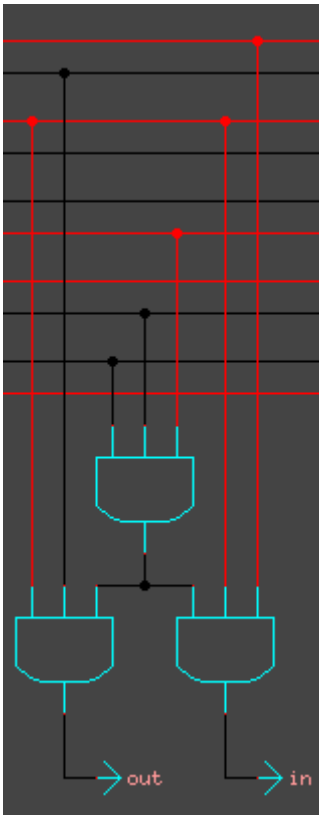
2.5) On rajoute IMM4-2 à gauche de reg8 (en dessous de IMM7-5).
Le code de test est :

```
ldi r0, 31
ldi r1, 20
ldi r2, 10
add r1, r1, r2
sub r0, r0, r1
end: jmp end
```

Et on voit bien 1 écrit dans le registre r0 :



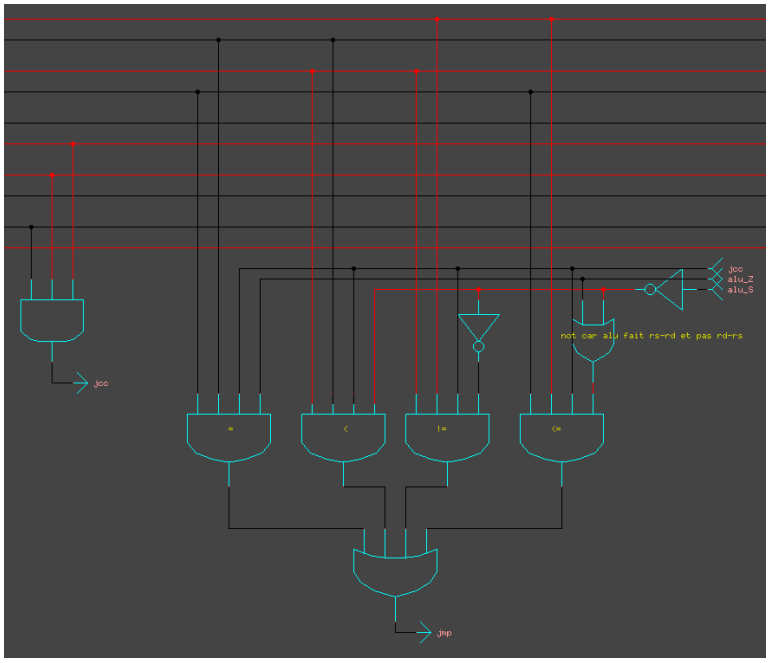
3.1) On récupère in et out comme cela :



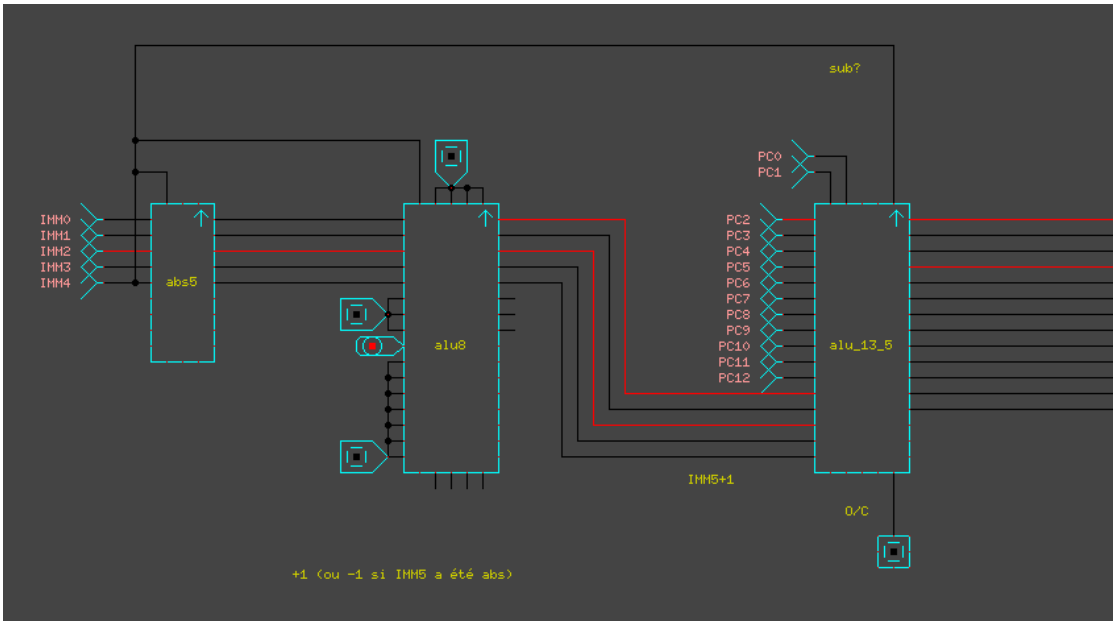
3.2) Si aucune touche n'a été frappée, le clavier renvoie 255. Et si de nombreuses touches ont étés frappées depuis la dernière récupération de touche, elles sont stockées et rendue dans l'ordre par le buffer.

3.3) Deux valeurs sont égales si leur différence est égale à 0. On rajoute le flag Z, en vérifiant que la sortie vaut 0. (Cette partie là du circuit ayant évolué, je ne peut pas montrer le résultat)

3.4) On récupère jcc et jmp comme cela (le traitement de jlt, jle et jne n'est pas à prendre en compte)

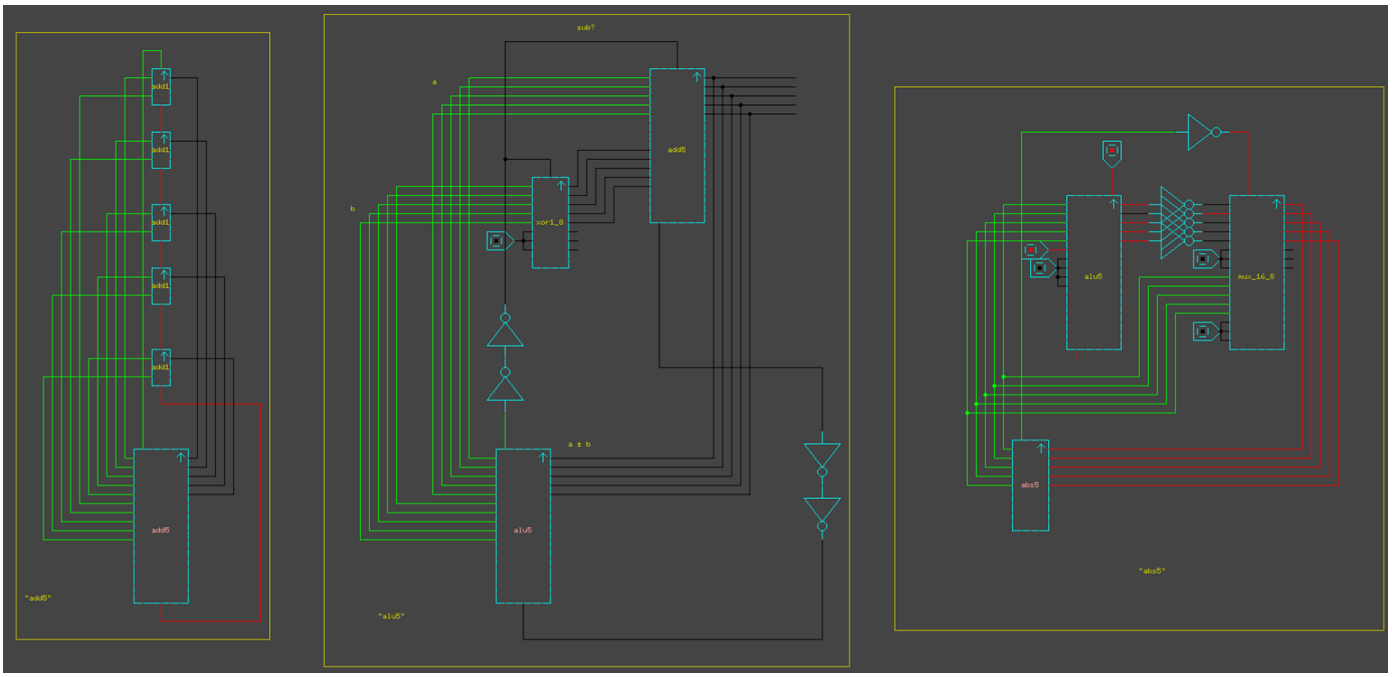


Pour la gestion de PC il fallait additionner PC, un entier sur 13 bits et IMM5, un entier signé sur 5 bits.
Ce circuit permet cela :

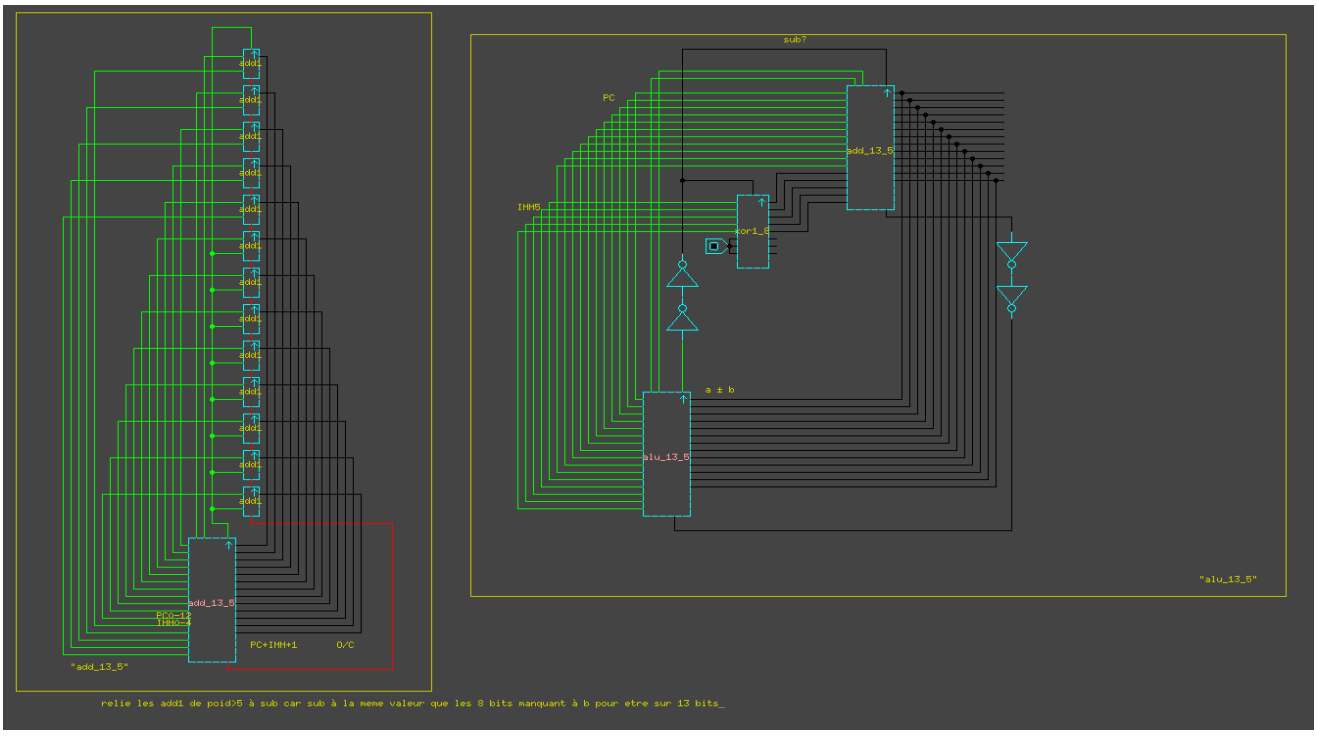


On met IMM5 en valeur absolue (en gardant trace de son signe d'origine), on lui ajoute ou retire 1 en fonction de si IMM5 étais positif ou non.
Ensuite avec un alu13_5 on additionne ou soustrait cela à PC.
On à donc :
-Si IMM5 ≥ 0 : PC ← PC+IMM5+1
-Si IMM5 < 0 : PC ← PC – (IMM5 -1)

J'ai fait cela car mon alu13_5 seul ne me donnais pas le bon résultat. (Alors que pour une addition à bits égaux, que les entiers soit signé ou non n'as aucune importance).
Voici les circuits pour mon abs5 :



J'ai fait l'opération inverse de passage d'entier non-signé à signé : pour représenter -x (avec x>=0) on le passe à son complément à 1 puis lui ajoute 1.
Alors pour avoir la valeur absolue de y on lui retire 1 et on passe à son complémentaire. J'ai défini un alu5 pour cela (on aurais pu utiliser l'alu8) et j'ai utilisé un mupliplexer car si y est positif, il ne faut pas changer sa valeur.
Pour add13_5 :



Le circuit est très similaire à alu8.
Dans cet alu, on fait comme si b étais de la forme 0000 0000 IMM5. Et donc, dans le add13_5, on relie les add1 de poids >5 à sub car si sub=1, alors IMM5 est mis a son complémentaire et vaut donc 1111 1111 Complémentaire(IMM5) et si sub=0, ces bits restent à 0.

3.5) Le code est le suivant :
ldi r1, 13 #13 est la valeur en ASCII pour la touche entrée

loop: in r0 #On récupère une valeur
jeq r0, r1, end #Si c'est entrée, on s'arrête
out r0 #Sinon on l'affiche
jmp loop #On recommence

end: jmp end

Si l'utilisateur ne presse aucune touche, on récupère 255 mais rien ne s'affiche quand on le met sur l'écran donc cela n'as pas d'importance.

3.6) Le code est le suivant :
ldi r1, 13 #ASCII pour la touche entrée
ldi r2, 47 #ASCII pour juste avant 0
ldi r3, 58 #ASCII pour juste après 9
ldi r4, 255 #ASCII pour entrée vide

#getn est la boucle pour obtenir n
getn: in r0
jeq r4, r0, getn #n ne peut pas être le caractère vide
#n doit être en entier, donc entre 48 et 57 compris
jlt r0, r2, getn
jlt r3, r0, getn

#getc est la boucle pour obtenir le caractère
getc: in r5
jeq r4, r5, getc #Le caractère ne peut pas être vide
jeq r4, r1, getc #Le caractère ne peut pas être la touche entrée

ldi r2, 1 #Constante utilisée pour les conditions de sortie de boucle
ldi r4, 48 #Permet d'obtenir la valeure entière de la valeur ASCII de n
ldi r6, 0
sub r0, r0, r4 #Passe de ASCII à entier, i<-n
jeq r0, r6, end
addi r4, r0, 0 #cst<-n, permet de remettre j à n, addi x,x,0 car mov provoque des erreurs

loop_ext:
addi r3, r4, 0 #j<-n

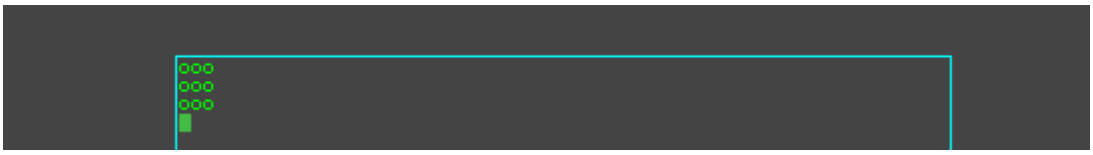
loop_int:

out r5
subi r3, r3, 1 #j-=1
jlt r2, r3, loop_int # Si j>=1

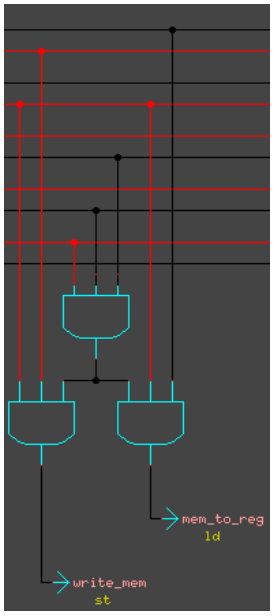
out r1 #out "\n"
subi r0, r0, 1 #i-=1
jlt r2, r0, loop_ext # Si i>=1

end: jmp end

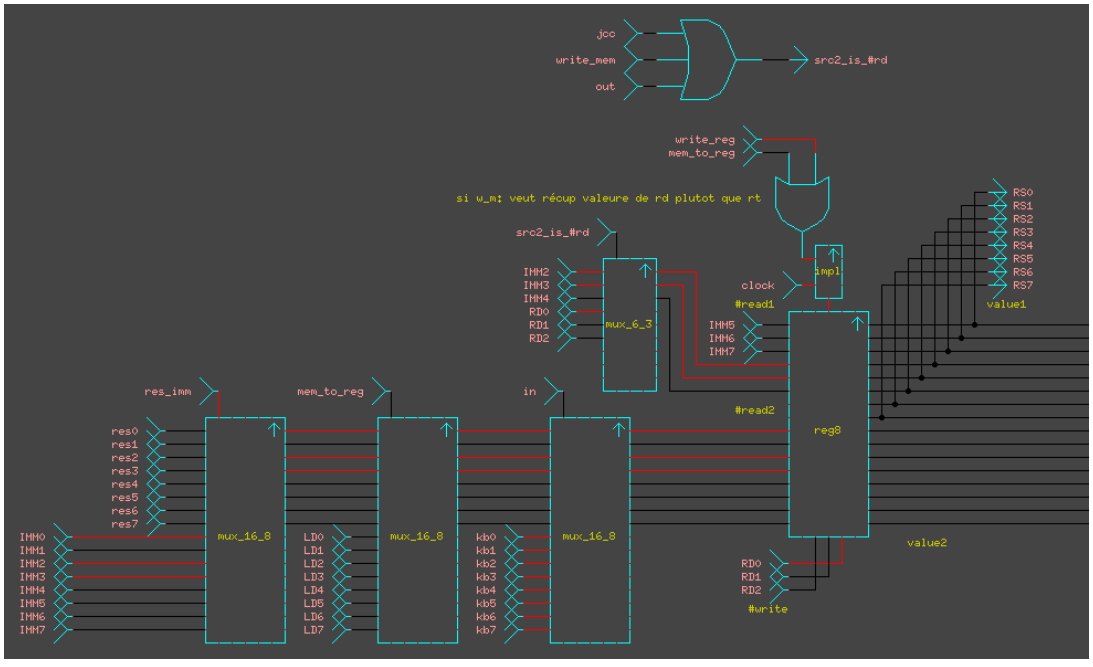
Et le résultat pour n=3 et c=o est :



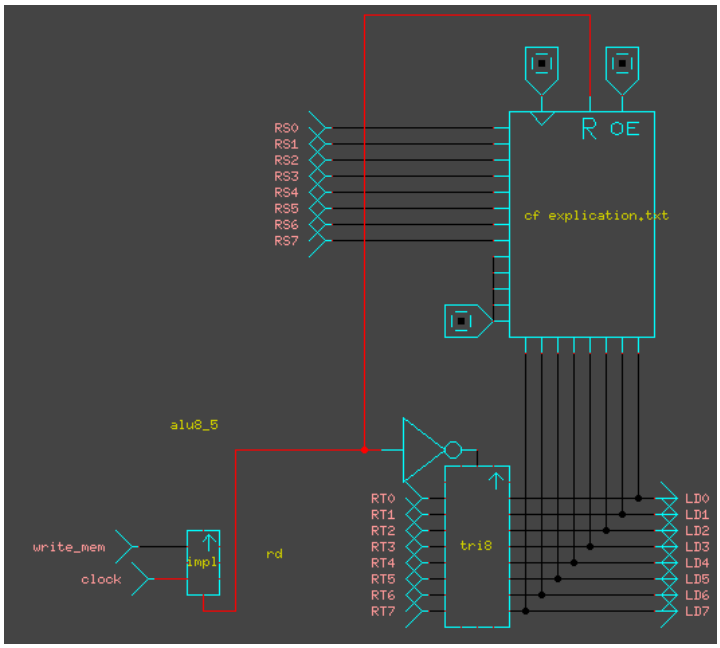
4.1) On récupère deux bits de contrôle (write_mem et mem_to_reg) :



On modifie aussi l'accès au registre comme cela :



On récupère rd plutot que rt quand on veut le stocker. Et on écrit dans le registre quand on récupère une valeur de la RAM.
La RAM donne cela :



4.2) Le code est :
ldi r6, 255 #ASCII pour entrée vide
ldi r7, 13 #ASCII pour la touche entrée

ldi r0, 0

#Boucle d'entrée
get: in r1
jeq r1, r6, get #Si r1=255 (<=> l'utilisateur n'as rien frappé) alors on rerécupère la touche présée
jeq r1, r7, print #Si l'utilisateur appuie sur entrée, on passe à la section d'affichage
st r1, r0
addi r0, r0, 8 # +8 car les caractères ASCII sont sur 8 bits
jmp get

#Boucle d'affichage
print: subi r0, r0, 8 # -8 car les caractères ASCII sont sur 8 bits
ld r1, r0
out r1
ldi r5, 0 #J'ai du le mettre ici car quand je le mettais en dehors de la boucle, j'obtenais r5!=0
jlt r5, r0, print #Si r0 > 0

end: jmp end

Et l'entrée 1 2 3 me donne :

4.3) On ajoute les drapeaux avec les conditions indiquées dans le cours
(Cette partie-là du circuit ayant évolué, je ne peux pas montrer le résultat)

4.4) On démontre que pour a et b deux entiers signés sur 8 bits différents, a ≤ b <=> S = O, où S et O sont obtenus suite au calcul de b − a par l'ALU.

<= :

-Si S=O=1 : alors O=1 => le résultat de l'ALU est de même signe que a ET les signes de a et b sont opposé.
Comme S=1 => résultat de l'ALU est négatif, on a : a négatif et b positif
Donc a ≤ b

-Si S=O=0: alors O=0 => le résultat de l'ALU est exactement b-a est ce résultat est positif
Donc a ≤ b

=> :

On à a ≤ b, étudions le flag S du résultat de b-a en fonction de O
-Si O=1 : Comme les signes de a et b sont opposé.
Et que a ≤ b, alors on a négatif et b positif
Et comme le résultat de l'ALU est de même signe que a, on a S=1
Donc S=O

-Si O=0 : alors le résultat de l'ALU est de signe opposé à a OU les signes de a et b sont les mêmes.

-Si les signes de a et b sont les mêmes, alors comme a ≤ b et O=0 (pas d'overflow) on à b-a ≥ 0
Donc S=0

-Si le résultat de l'ALU est de signe opposé à a :

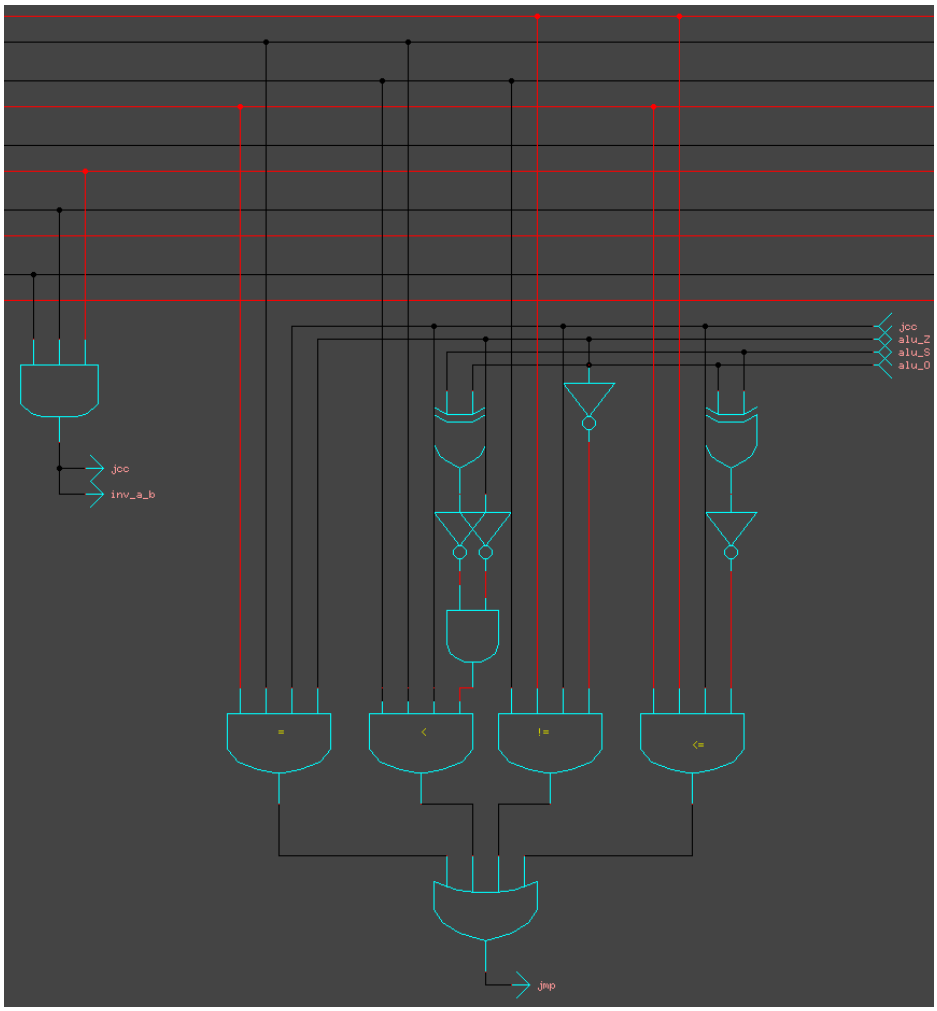
-Si a ≥ 0 alors comme a ≤ b, on a b-a ≥ 0 donc le résultat de l'ALU à le même signe que a.
Contradiction !

Donc a<0
Donc le résultat de l'ALU est positif
Donc S=0

Donc S=O

Ainsi, a ≤ b <=> S = O, où S et O sont obtenus suite au calcul de b − a par l'ALU.

4.5) On ajoute cela au niveau du bit de contrôle jmp :



Sachant qu'on a : a<b quand a≤b et a!=b

4.6) Le code est :
ldi r0, 2 #Valeur de départ
ldi r4, 20 #Valeur attendue

ldi r1, 9 #Compteur
ldi r3, 0 #Constante

ldi r2, 0 #Valeur courrante
loop: add r2, r2, r0
subi r1, r1, 1 #i-=1
jlt r3, r1, loop #Si i>0

toto: jeq r0, r4, toto #Boucle sur cette ligne si la valeur est bien celle attendue
end: jmp end

Le circuit boucle bien à la ligne 8 (ligne 8 une fois compilée)
On pourrait faire ce calcul bien plus efficacement et décalant à droite r0 (x2) et en l'additionnant à lui-même que 5 fois.

4.7) Le code est :

```
ldi r6, 255 #ASCII pour entrée vide
ldi r7, 13 #ASCII pour la touche entrée

ldi r0, 0

#Boucle d'entrée
get: in r1
jeq r1, r6, get #Si r1=255 (<=> l'utilisateur n'as rien frappé) alors on rerécupère la touche présée
jeq r1, r7, print #Si l'utilisateur appuie sur entrée, on passe à la section d'affichage
st r1, r0
addi r0, r0, 8 # +8 car les caractères ASCII sont sur 8 bits
jmp get

ldi r2, 0 #Vas augmenter de 8 en 8 jusqu'à atteindre r0
jeq r0, r2, end #Si r0 = 0 (<=> aucune valeur stockée) on stop
ldi r6, 97 #ASCII pour "a"
ldi r7, 122 #ASCII pour "z"
ldi r5, 32 #32 est la différence entre min et maj en ASCII

#Boucle d'affichage
print: ld r1, r2
#Si r1 n'est pas une minuscule, pas besoin de le passer en majuscule
jlt r1, r6, pas_min
jlt r7, r1, pas_min
sub r1, r1, r5 #Passe de min à maj
pas_min: out r1
addi r2, r2, 8
jlt  r2, r0, print #Si r2<r0, alors on print le caractère suivant

ldi r0, 13
out r0

end: jmp end
```

Mais je n’obtient pas le résultat voulu à l’écran.

5.1) L’alu obtenu est celui décrit dans la section III), pour plus de détails, s’y référer.

5.2)Le code est :

```
ldi r2, 48 #ASCII pour 0
ldi r3, 57 #ASCII pour 9
ldi r4, 255 #ASCII pour entrée vide

getn: in r0
jeq r4, r0, getn #Si entrée vide, recommence
#Si pas en entier, recommence
jlt r0, r2, getn
jlt r3, r0, getn

end: jmp end
```

Et r0 contient l'entier tapée

5.3) Le code est :

```
ldi r0, 42 #Valeur de départ
ldi r7, 4 #Valeur attendue
ldi r1, 10 #Constante
ldi r2, 0 #Compteur

loop: jlt r0, r1, end_loop #Si valeur_courante<10
sub r0, r0, r1  #valeur_courante-=10
addi r2, r2, 1 #Compteur +=1
jmp loop

end_loop: jeq r2, r7, end_loop #Si la valeure obtenue est la valeur attendue, on boucle sur cette ligne
end: jmp end
```

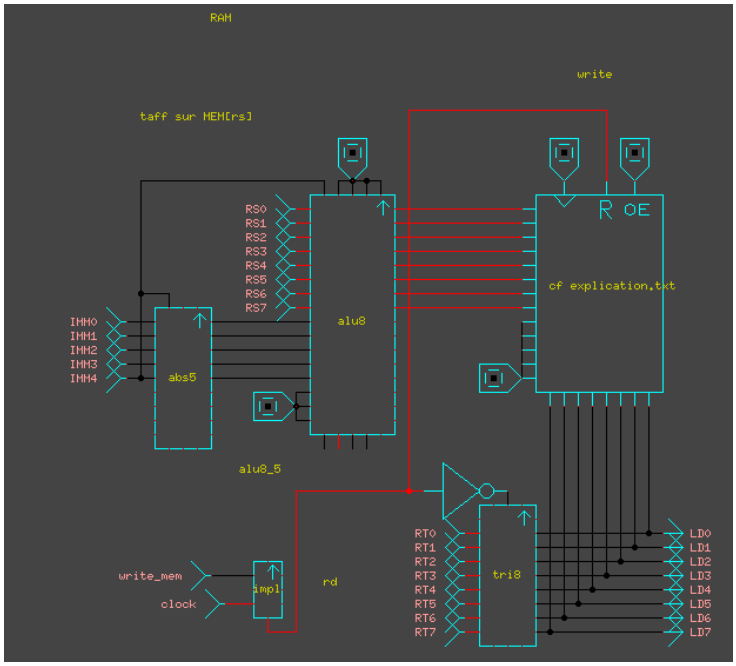
Et le circuit boucle bien sur la ligne 8 (la ligne de end_loop une fois compilée)

5.4) Le code est :

```
ldi r5, 48 #ASCII pour 0
ldi r6, 57 #ASCII pour 9
ldi r7, 255 #ASCII pour entrée vide

#On récupère n1
getn1: in r0
jeq r7, r0, getn1
jlt r6, r0, getn1
jlt r0, r5, getn1

#On récupère n2
getn2: in r1
jeq r7, r1, getn2
jlt r6, r1, getn2
jlt r1, r5, getn2

ldi r2, 0 #Valeur courrante
ldi r4, 0 #Constante
#Pour passer de ASCII à entier
ldi r3, 48
sub r1, r1, r3
sub r0, r0, r3

#On ajoute r1 dans r2, r0 fois
loop: jle r0, r4, print #Si r0 <= 0, on affiche le résultat
add r2, r2, r1 #valeur_courante += r1
subi r0, r0, 1 # r0 -= 1
jmp loop

print: add r2, r2, r3
out r2
end: jmp end
```

Et permet de multiplier 2 entiers à 1 chiffre et de renvoyer le résultat si celui ci est inférieur à 9.

5.5) On ajoute tout simplement ce circuit à la RAM :



La problématique est la même que pour la gestion de jeq, comment additionner un entier signé sur 5 bits avec un entier sur 8 bits ?
Ma solution est de passer par la valeur absolue. Le circuit est très similaire à celui présenté à la question 3.4)