Réalisation des processeurs Nono-1 et Nono-2
Projet d'Architecture des ordinateurs

Florian Delavernhe et Thomas Minier
groupe 501A

18 Décembre 2014

Table des matières

1 Introduction **2**

2 Implémentation du Nono-1 **2**
2.1 Unité arithmétique logique 2
2.2 Décodeur d'instruction 2
2.3 Contrôle de saut 3
2.4 Banc de registres 3
2.5 Sélection de registres 3

3 Utilisation du Nono-1 **4**
3.1 Implémentation de l'instruction halt 4
3.2 Fonction pgcd 4
3.3 Fonction pow 4
3.4 Processeur Nono-2 4

4 Conclusion **4**

5 Annexes **4**

1 Introduction

Dans le cadre de ce projet, il nous a été demandé de réaliser un processeur Nono-1 à 16 registres. Il supporte jusqu'à 16 instructions différentes, réparties sur trois formats de données.

Nous avons utilisé le logiciel *Logisim* pour modéliser le circuit et les programmes de tests ont été écrits en assembleur et en binaire.

2 Implémentation du Nono-1

2.1 Unité arithmétique logique

Pour réaliser cette UAL , nous avons décidé d'effectuer chacune des opérations possibles en fonction des deux entrées, puis de diriger les résultats de chacune de ces opérations dans un multiplexeur. En fonction du code sur l'entrée *ctrlUAL*, nous envoyons dans la sortie le résultat correspondant à l'opération souhaitée. Le code reçu par le multiplexeur est détaillé dans la Figure 1. Concernant les différents flags, nous rappelons leur signification et comment nous avons choisi de les implémenter :

- **CF**, ou *Carry Flag*, est armé si une opération arithmétique génère une retenue. Pour l'implémenter, nous identifions d'abord quelles sont les opérations susceptibles de générer une retenue : l'addition et la soustraction. Nous faisons alors un test logique `OU` sur les sorties des retenues de ces deux opérations. Si l'une d'elles est vraie, alors il y a bien une retenue et le *Carry Flag* est armé.
- **ZF**, ou *Zero Flag*, est armé si le résultat d'une opération arithmétique est égal à zéro. Pour l'implémenter, nous faisons simplement un test logique `ET` entre la sortie de l'UAL et une constante fixée à `00000000`, ce qui revient à tester si la sortie est égal à zéro codé sur 8 bits. Si le test vaut vrai, alors le *Zero Flag* est armé.
- **SF**, ou *Sign Flag*, est armé si le résultat d'une opération arithmétique possède un bit de poids fort à 1, signalant ainsi un résultat signé. Pour l'implémenter, nous récupérons le bit de pids fort du résultat avec un splitter, puis nous testons avec un `ET` s'il est égal à 1. Si le test vaut vrai, alors le *Sign Flag* est armé.
- **OF**, ou *Overflow Flag*, est armé si le résultat constitue un nombre dont le codage dépasse 8 bits. Pour l'implémenter, nous faisons un test logique `ET` entre les deux bits de poids forts des entrées. Si le test est vrai, alors le **OF** est armé.

Une fois ces 4 drapeaux identifiés et armés s'il le faut, nous concaténons leurs valeurs au moyen d'un splitter (dans l'ordre **CF - ZF - SF - OF**) puis nous envoyons ce nombre sur 4 bits dans la sortie *flags* de l'UAL.
Le détail du circuit de l'UAL se trouve dans les annexes à la Figure 9.

2.2 Décodeur d'instruction

Pour implémenter le décodeur d'instructions, nous codons les différentes opérations comme indiqué dans la Figure 2.

Nous avons choisi ce codage précis pour nous permettre de réduire le circuit logique. En effet, nous avons fait en sorte que toutes les instructions de saut aient leur 3ème bit égal à 1. Les autres instructions elles ont leur 3ème bit égal à zéro.

Pour les instructions de saut, nous allons avoir besoin que l'UAl effectue une soustraction. Pour ce faire, nous relions la sortie du décodeur à un multiplexeur. Si isJUmp est vrai, alors nous renvoyons le code `0 0 1`, qui correspond à l'instruction sub. Sinon, nous renvoyons le code sans son 3ème bit, qui est inutile.

Pour armer ou non les drapeaux *isLoad*, *regWrite* et *isJMP*, nous dressons les tableaux de Karnaugh des Figures 3, 4 et 5.

On en déduit que $isLoad = A\ \overline{B}\ C\ D$, $regWrite = \overline{B}$ et $isJMP = B$. Le détail du circuit se trouve dans la Figure 10.

2.3 Contrôle de saut

Ce module envoyant dans sa sortie un code sur 2 bits indiquant quel type de saut il faut faire (instruction suivante, HALT ou saut), nous commençons par établir le codage de cette sortie, indiqué dans la Figure 6. L'entrée correspond au code `1 0` au niveau du multiplexeur en sortie du module sera neutralisée.

En reprenant notre codage des différentes instructions, nous remarquons qu'il est possible de simplifier une fois encore le circuit. En effet, si le 3ème bit du code est égal à 0, alors $e_0 = 0$. Sinon, $e_0 = 1$. Nous pouvons donc donc déterminer facilement la valeur de e_0 avec un simple test logique `ET` entre le troisième bit du code et une constante fixée à `0`.

Ensuite, nous nous intéressons à e_1. Pour pouvoir implémenter le module, nous devons pouvoir effectuer les tests liés aux différentes instructions de saut conditionnelle. Nous avons donc remarqué les analogies suivantes :

- L'instruction **beq, $t0, $t1, etq** peut se traduire par : si $\$t0 - \$t1 = 0$, alors on saute à *etq*.
- L'instruction **bne, $t0, $t1, etq** peut se traduire par : si $\$t0 - \$t1 \neq 0$, alors on saute à *etq*.
- L'instruction **bge, $t0, $t1, etq** peut se traduire par : si $\$t0 - \$t1 \geq 0$, alors on saute à *etq*.
- L'instruction **ble, $t0, $t1, etq** peut se traduire par : si $\$t0 - \$t1 \leq 0$, alors on saute à *etq*.
- L'instruction **bgt, $t0, $t1, etq** peut se traduire par : si $\$t0 - \$t1 > 0$, alors on saute à *etq*.
- L'instruction **blt, $t0, $t1, etq** peut se traduire par : si $\$t0 - \$t1 < 0$, alors on saute à *etq*.
- L'instruction **b etq** ne nécessite aucune interprétation, on saute à *etq* dès que cette instruction est lue.
- Même chose pour l'instruction **halt**. On arrête le programme dès qu'elle est lue.

Pour les instructions **beq** et **bne**, il nous faudra donc regarder si le drapeau **ZF** est armé ou non. Pour les autres instructions, il nous faudra regarder les drapeaux **ZF** et **SF**. Nous établissons donc une table de vérité, directement exprimée sous la forme d'un tableau de Karnaugh et détaillée dans la Figure 7

On en déduit que $e_1 = \overline{A}\ \overline{C}\ D + \overline{A}\ D\ \overline{ZF}\ \overline{SF} + \overline{A}\ C\ \overline{D}\ ZF + A\ C\ \overline{D}\ \overline{SF} + A\ D\ \overline{ZF}\ SF + \overline{C}\ \overline{ZF}\ \overline{SF} + A$. Le détail du circuit se trouve à la Figure 11.

2.4 Banc de registres

Nous commençons par coder la sélection de registres avec rd, rs et rt, détaillé dans la Figure 8. Ensuite, nous réalisons un sous-circuit **décodeur rd** qui se charge, en fonction du code de l'entrée, de faire sortir 1 sur la sortie correspondant au registre que l'on veut sélectionner. Les autres sorties sont mises à 0.

Grâce à ce sous-circuit, nous pouvons réaliser le module. Nous faisons rentrer *valin* dans l'entrée de chaque registre. Pour savoir s'il faut écrire ou non dans un registre, nous faisons un test logique `ET` entre la sortie de **decodeur rd**, qui correspond au registre courant, et *regWrite*. Le résultat de ce test est alors envoyé dans le verrou du registre courant.

Enfin, nous utilisons des multiplexeurs pour déterminer les sorties du module. Les sorties de chaque registre sont envoyées dans chacun des multiplexeurs et les valeurs de *rs* et *rt* servent à sélectionner les bonnes valeurs qui seront envoyées dans les sorties. Le détail du circuit se trouve à la Figure 12.

2.5 Sélection de registres

Pour ce module, nous avons simplement à vérifier la valeur de *isJMP*. Si elle vaut 1, alors on a les sorties suivantes :

- $rd = \{\text{rd/rs}\}$
- $rs = \{\text{rs/rt}\}$
- $rt = \{\text{rt}\}$

Sinon, alors on a les sorties suivantes :

- $rd = \{\text{rd/rs}\}$
- $rs = \{\text{rd/rs}\}$
- $rt = \{\text{rs/rt}\}$

Le détail du circuit se trouve à la Figure 13.

3 Utilisation du Nono-1

3.1 Implémentation de l'instruction halt

D'après la figure de l'énoncé, l'instruction *halt* est implémentée par une constante qui déclenche un saut à la fin du programme.

Cela impose une contrainte sur le code machine. En effet, étant donné que la constante *halt* vaut `0xFF`, on ne peut pas avoir d'instructions au delà de cette adresse. Sinon, l'instruction halt représenterait un simple saut et non la fin du programme.

3.2 Fonction pgcd

Le code assembleur et binaire se trouve dans les fichiers **bin/pgcd.asm** et **bin/pgcd.bin**.

3.3 Fonction pow

Nous avons décidé de coder la fonction puissance, qui renvoie 2^i, où i est précisé dans le programme. Le code assembleur et binaire se trouve dans les fichiers **bin/pow.asm** et **bin/pow.bin**.

3.4 Processeur Nono-2

Nous n'avons pas eu le temps de traiter la question, par manque de temps.

4 Conclusion

En conclusion, ce projet aura été assez intéressant. Il nous aura permis de comprendre le fonctionnement d'un processeur et comment le code assembleur est interprété et utilisé par le circuit.

5 Annexes

Instruction	**code**
add	`0 0 0`
sub	`0 0 1`
or	`0 1 0`
and	`0 1 1`
not	`1 0 0`
right shift	`1 0 1`
left shift	`1 1 0`

FIGURE 1 – Codage de ctrlUAL

Instruction	A B C D
add	0 0 0 0
sub	0 0 0 1
or	0 0 1 0
and	0 0 1 1
halt	0 1 0 0
b	0 1 0 1
beq	0 1 1 0
bne	0 1 1 1
not	1 0 0 0
shl	1 0 0 1
shr	1 0 1 0
li	1 0 1 1
bge	1 1 0 0
ble	1 1 0 1
bgt	1 1 1 0
blt	1 1 1 1

FIGURE 2 – Codage des instructions sur 4 bits

$AB\backslash CD$	0 0	0 1	1 1	1 0
0 0	0	0	0	0
0 1	0	0	0	0
1 1	0	0	0	0
1 0	0	0	1	0

FIGURE 3 – Tableau de Karnaugh de isLoad

$AB\backslash CD$	0 0	0 1	1 1	1 0
0 0	1	1	1	1
0 1	0	0	0	0
1 1	0	0	0	0
1 0	1	1	1	1

FIGURE 4 – Tableau de Karnaugh de regWrite

$AB\backslash CD$	0 0	0 1	1 1	1 0
0 0	0	0	0	0
0 1	1	1	1	1
1 1	1	1	1	1
1 0	0	0	0	0

FIGURE 5 – Tableau de Karnaugh de isJMP

Type de saut	e_1e_0
instruction suivante	0 0
HALT	0 1
saut	1 1

FIGURE 6 – Codage de la sortie du contrôle de saut

Instruction	$ACD\backslash ZFSF$	0 0	0 1	1 1	1 0
b	0 0 1	1	1	1	1
bne	0 1 1	1	1	0	0
beq	0 1 0	0	0	1	1
bgt	1 1 0	1	0	0	0
halt	1 1 1	0	1	0	0
ble	1 0 1	0	1	1	1
bge	1 0 0	1	0	1	1

FIGURE 7 – Tableau de Karnaugh du contrôle de saut

Registre	A	B	C	D
r0	0	0	0	0
r1	0	0	0	1
r2	0	0	1	0
r3	0	0	1	1
r4	0	1	0	0
r5	0	1	0	1
r6	0	1	1	0
r7	0	1	1	1
r8	1	0	0	0
r9	1	0	0	1
r10	1	0	1	0
r11	1	0	1	1
r12	1	1	0	0
r13	1	1	0	1
r14	1	1	1	0
r15	1	1	1	1

FIGURE 8 – Codage de la sélection de registres sur 4 bits

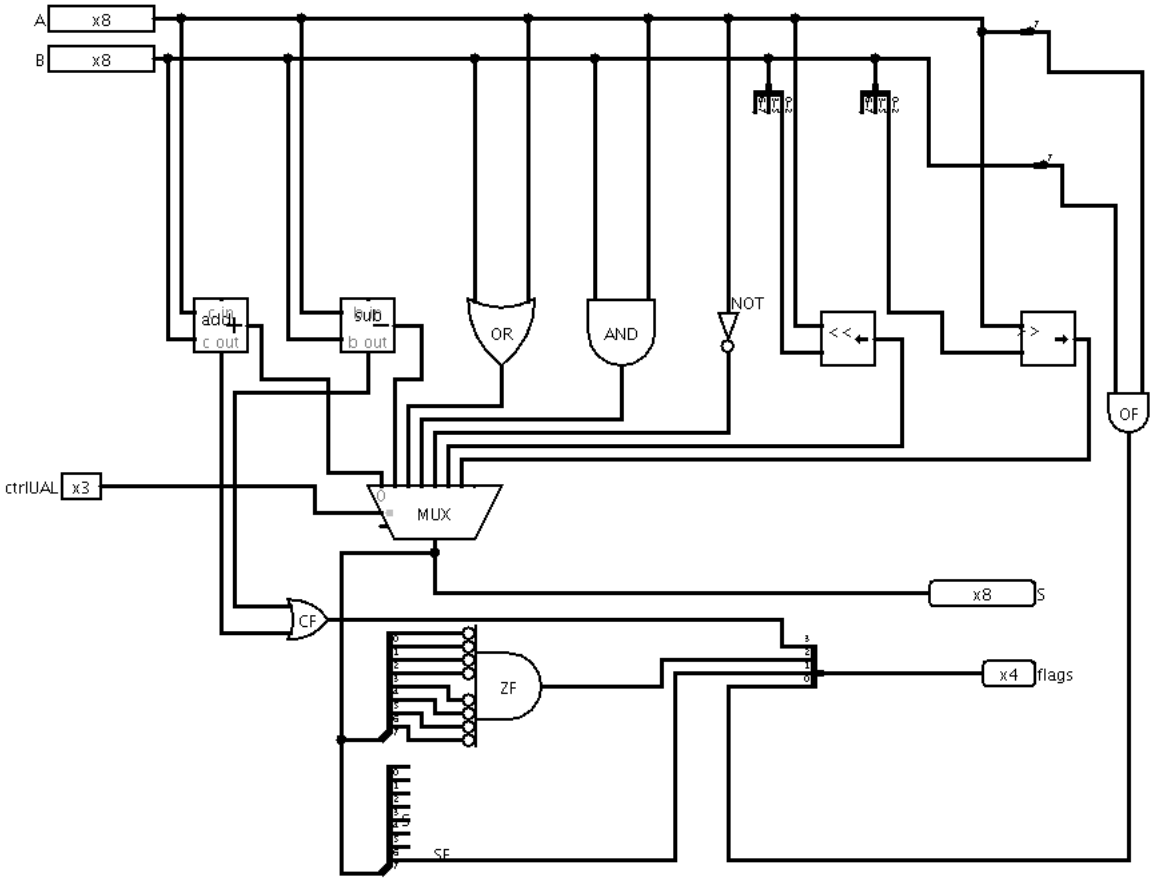

FIGURE 9 – Circuit logique de l'UAL

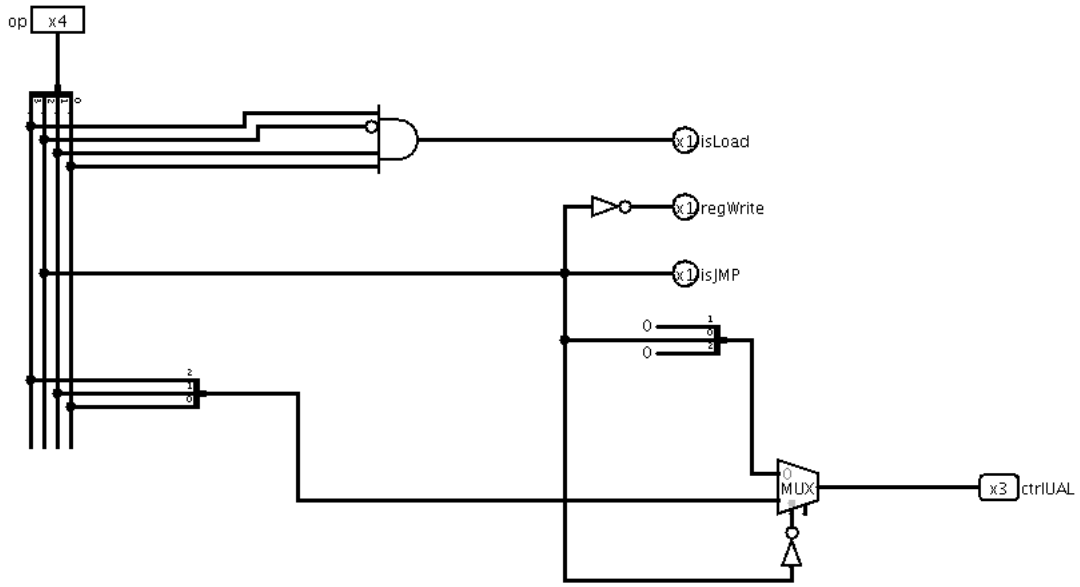

FIGURE 10 – Circuit logique du décodeur d'instructions

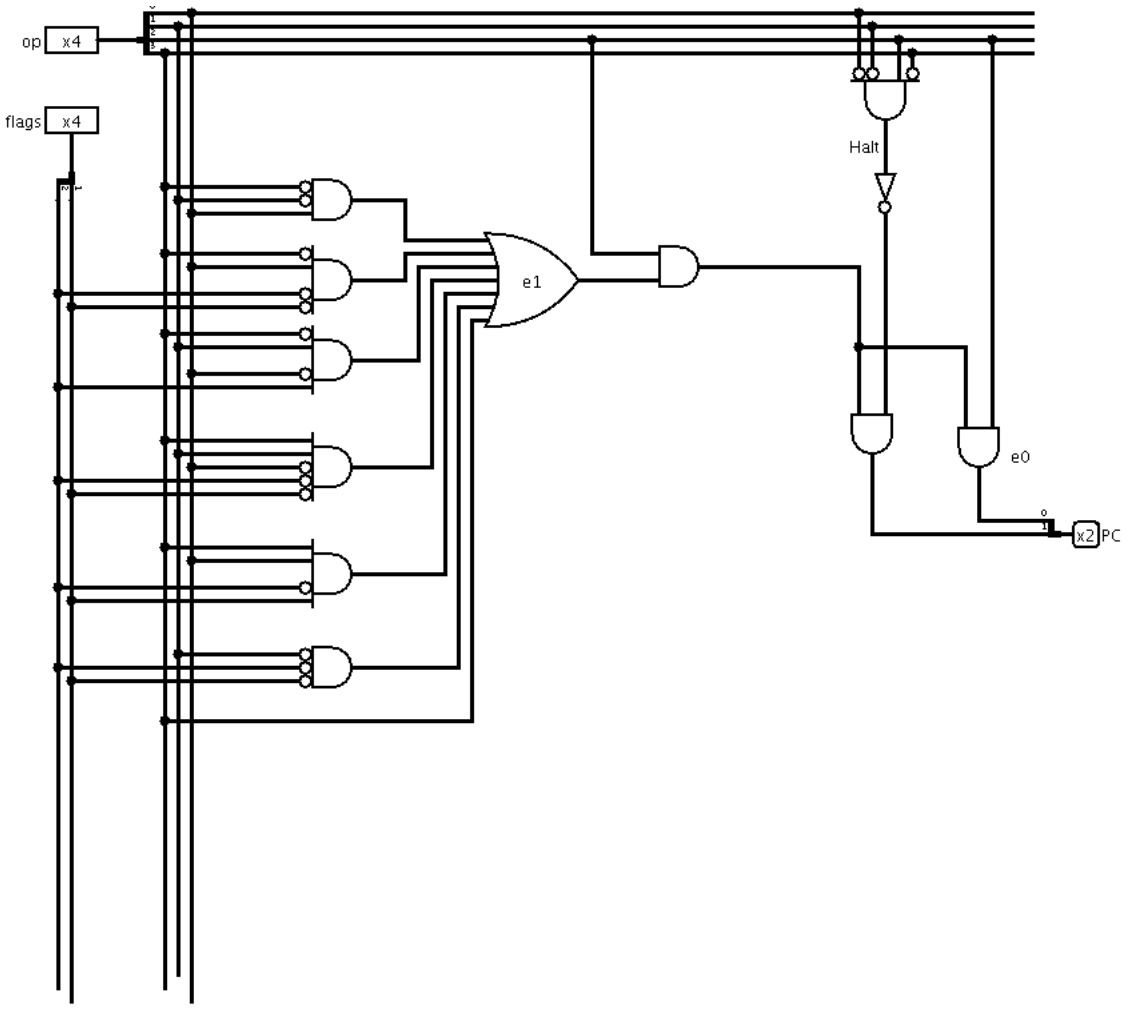

FIGURE 11 – Circuit logique du contrôle de saut

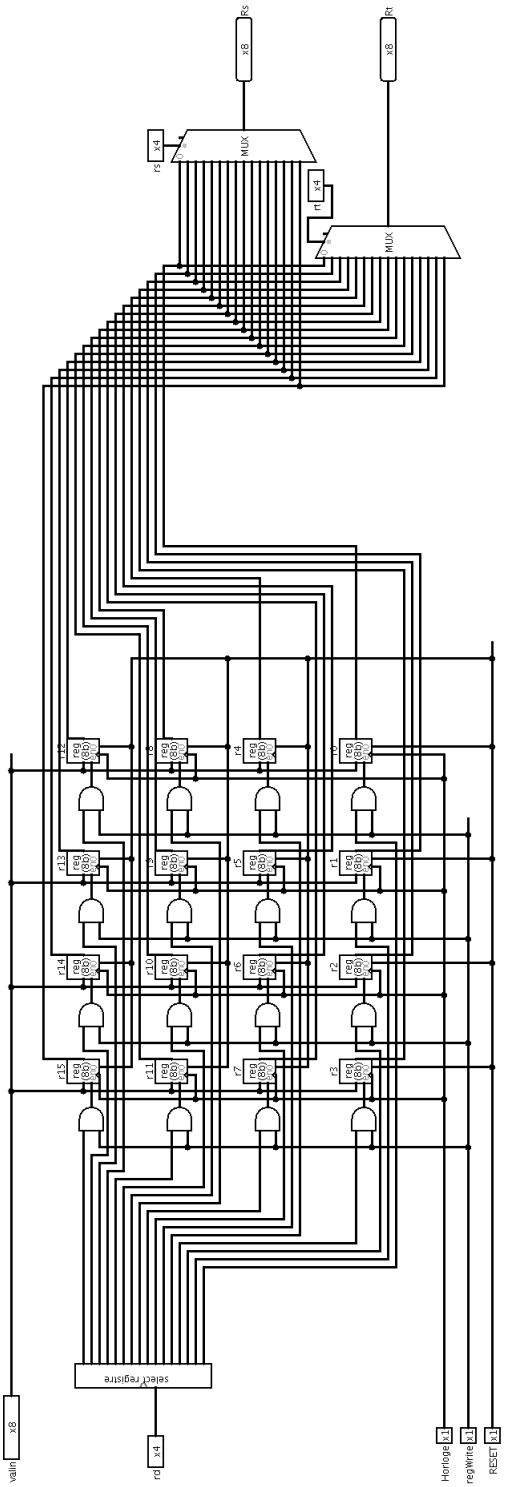

FIGURE 12 – Circuit logique du banc de registres

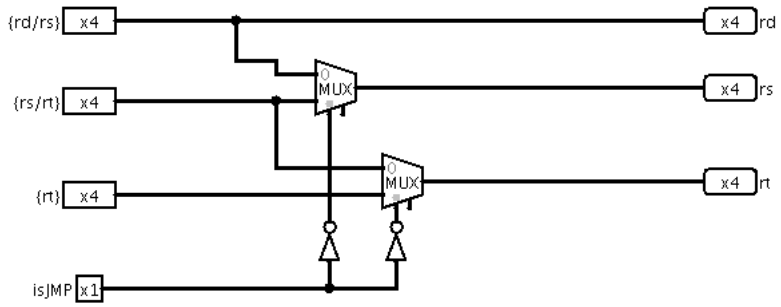

FIGURE 13 – Circuit logique de la sélection de registres

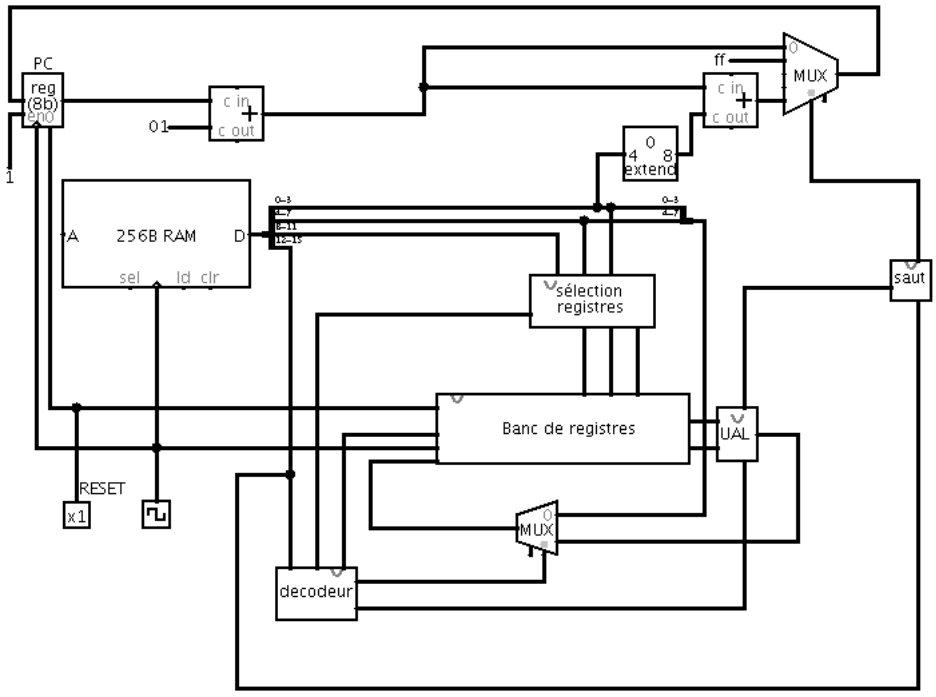

FIGURE 14 – Circuit du processeur Nono-1