Processeu NONO 1et 2

Cédric BOIS Benjamin SIENTZOFF

18 décembre 2014

Université de Nantes - Licence 3 Informatique
X5I0030 Architecture des Ordinateurs

Table des matières

Introduction

Dans le cadre du cours intitulé *Architecture des ordinateurs*, nous devons recréer un processeur Nono-1. Par la suite, ce processeur sera modifier pour devenir Nono-2. Ce rapport retrace comment nous avons réalisé le premier processeur.

Les circuits électroniques présentés sont produits avec le logiciel *Logisim*. Ces circuits et les différents fichiers permettant notamment de programmer le processeur sont fournis avec la version numérique de ce rapport. Les images RAM peuvent être directement chargées dans la RAM des processeurs Nono. Ces images correspondent aux programmes compilés pour ces architectures et peuvent être exécutés directement dans *Logisim*.

Nous commençons par présenter les différents sous-circuits composants le NONO 1. Puis nous détaillons son utilisation. Enfin, une brève partie explique comment transformer NONO 1 en NONO 2.

1 Opcode des instructions

Nono-1 et Non-2 sont des processeurs utilisant l'assembleur MIPS. Les instructions disponibles sur Nono-1 sont présentés au tableau de la figure 2. On remarque que les instructions reconnues sont relativement restreintes. Ces instructions sont de trois formats différents comme on peut le voir à la figure 1 [1].

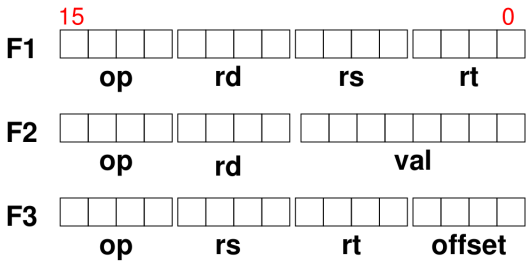


FIGURE 1 – Formats des instructions

Le Format F1 Le format F1 est composé de quatre paquets de bits. Le premier est sur quatre bits, il correspond au code de l'instruction, et c'est le cas pour tous les formats d'instructions. Les trois paquets suivants, sur quatre bits. Ce format est utilisé typiquement pour des opérations faisant intervenir trois registres. Le premier correspond à la destination du résultat et les deux suivants aux registres contenant les opérantes.

Le Format F2 Le format F2 est composé de trois paquets de bits. Le premier est sur quatre bits, il correspond au code de l'instruction. Les 4 bits suivants correspondent à un nom de registre et les 8 derniers à une valeur immédiate. Ce format est typiquement utilisé pour l'instruction *li*.

Le Format F3 Le format F3 peut être divisé en quatre parties. C'est le format utilisé pour les sauts. Les quatre bits correspondent à l'opcode de l'instruction. Le paquet des quatre bits et le suivant constitué des quatre autres bits suivant correspondent à des noms de registres. Enfin les derniers bits (au nombre de quatre) correspondent à un offset. Pour les sauts, cela correspond à l'adresse de l'étiquette où effectuer le saut.

Le choix des opcodes n'a pas était fait au hasard. En effet, en regardant le nombre d'instructions pour les sauts et le nombre d'opérations faisant appellent à l'unité arithmétique et logique du processeur, on s'aperçoit qu'ils peuvent être divisé en deux groupes. On a donc regroupés les opcodes en trois groupes. Le premier correspond aux opcodes qui commence par un 1, ce sont les instructions qui font appel à l'UAL. Le second groupe, les opcodes commencent par un 0, correspondent aux sauts. Enfin le dernier groupe est composé des autres instructions. Citons notamment les opcodes 0000 et 1111. Le tableau des instructions, leur format et les opcodes correspondants est présenté à la figure 2.

1. Tiré du sujet du projet rédigé par M. Frédéric GOUALARD

Instruction et paramètres	Format	Opcode
`add` r_d, r_s, r_t	F_1	1000
`sub` r_d, r_s, r_t	F_1	1001
`or` r_d, r_s, r_t	F_1	1010
`and` r_d, r_s, r_t	F_1	1011
`not` r_d, r_s	F_1	1100
`shl` r_d, r_s, r_t	F_1	1101
`shr` r_d, r_s, r_t	F_1	1110
`li` r_d, *val*	F_2	1111
`halt`	F_1	0000
`b` *offset*	F_3	0001
`beq` r_s, r_t, *offset*	F_3	0010
`bne` r_s, r_t,*offset*	F_3	0011
`bge` r_s, r_t, *offset*	F_3	0100
`ble` r_s, r_t, *offset*	F_3	0101
`bgt` r_s, r_t, *offset*	F_3	0110
`blt` r_s, r_t, *offset*	F_3	0111

FIGURE 2 – *Opcode* des différentes instruction du processeur NONO 1

Maintenant que nous avons définit nos opcodes, il est temps de concevoir les circuits électroniques composants le processeur NONO 1. Commençons par l'Unité Arithmétique et Logique.

2 L' unité arithmétique et logique

L'Unité Arithmétique et Logique, abrégé UAL, permet de faire des calculs basiques (additions, divisions, décalages de bits, etc.). Elle effectue les calculs sur huit bits. L'UAL a trois entrées. La première sur trois bits permet de préciser le code de l'opération à utiliser. Les deux autres entrées sur huit bits sont les opérantes de l'opération.
En sortie, sur *ouput* on peut lire le résultat de l'opération. Il y a également quatre drapeaux comme détaillés ci-dessous.

- CF pour *Carrie Flag* est le drapeau levé lorsque que l'opération génère une retenue.
- ZF pour *Zero Flag* est armé lorsque que le résultat comporte uniquement des 0.
- OF pour *Overflow Flag* est un drapeau levé lorsque on dépasse la capacité des nombres représentable par la machine.
- SF pour *Sign Flag* est levé lorsque le résultat a son bit de poids fort à 1, c'est un nombre signé.

Overflow Flag Pour armer le drapeau *OF*, on fait le tableau KARNAUGH de la figure 4 d'où on tire l'équation 1. Ainsi on peut faire facilement le circuit qui contrôle la levée du drapeau.

$$OF = \neg s_1(e_1(\neg e_2.b_1 + e_1.\neg b_1)) + s_1(\neg e_1(\neg e_2.\neg b_1 + e_2.b_1)) \quad (1)$$

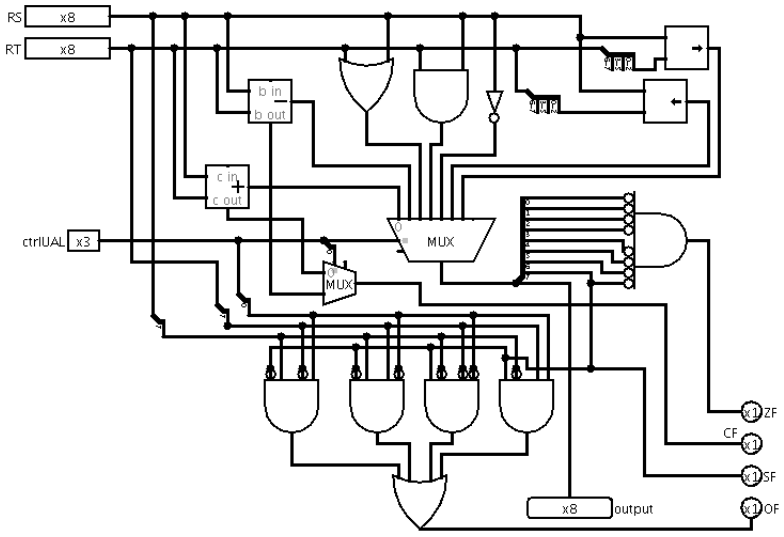


FIGURE 3 – Schéma électronique de l'Unité Arithmétique et Logique

s_1e_1 \ e_2b_1	00	00	11	10
00	0	0	0	0
01	0	1	0	1
11	0	0	0	0
10	1	0	1	0

FIGURE 4 – Tableau de KARNAUGH pour le drapeau OF

Dans l'équation 1, les variables e_1 et e_2 correspondent aux bits de poids fort de *Rs* et *Rt*. Les variables s_1 et b_1 correspondent respectivement au bit de poids fort du résultats de l'UAL et le bit de poids faible de *ctrlUAL*. On choisi ces bits car ils nous renseignent sur la nature du résultat. Par exemple lorsqu'on soustrait un nombre négatif et un nombre positif, si le résultat est positif, c'est qu'un *overflow* est survenu.

3 Le contrôleur de saut

Le second circuit électronique composant le processeur est le contrôleur de sauts. Son rôle est de mettre à jour le pointeur d'instructions *PC* (pour *Program Counter*) en fonction du résultat de l'opération que vient d'effectuer l'UAL pour le cycle suivant. Le saut est déterminé en fonction des indicateurs que le circuit a en entrée. C'est-à-dire *SF* et *ZF*. Pour ce là, on fait une table de vérité présente à la figure 6. Une fois la table remplie, on en tire l'équation suivante :

$$\begin{aligned} C_1 = & \neg C_1.O_1.\neg O_0.\neg ZF \\ & + \neg O_2.O_1.O_0.ZF \\ & + O_2.\neg O_1.O_0.\neg ZF.SF \\ & + O_2.O_1.\neg ZF.\neg SF \\ & + O_2.O_1.O_0.(ZF + \neg ZF.SF) \\ & + O_2.O_1.O_0.(ZF + \neg ZF.\neg SF) \end{aligned} \tag{2}$$

On traduit alors l'équation en circuit et on obtient le circuit de la figure 5.

Dans l'équation 2, O_i correspond au i-ème bit de l'opcode. La variable C_1 nous renseigne lorsqu'il faut faire un saut. Enfin, *ZF* et *SF* correspondent aux drapeaux levés par l'UAL.

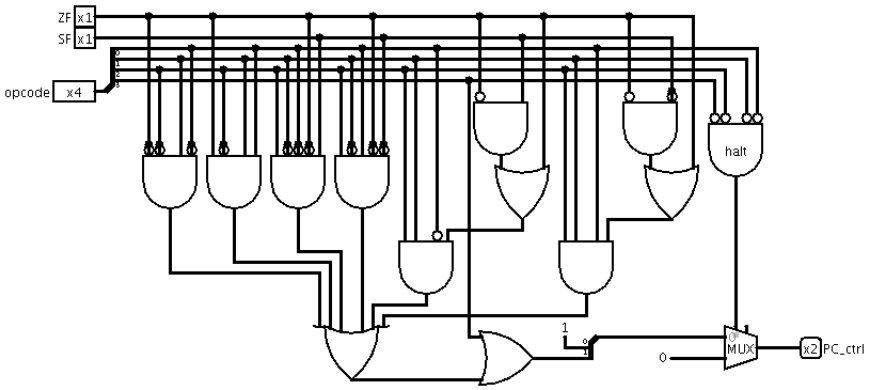


FIGURE 5 – Schéma électronique du contrôleur de sauts

Le multiplexeur qui contrôle *PC* réponds au codage suivant : À la fin du programme, pour l'instruction *halt*, le code du multiplexeur est 00, pour aller à la prochaine instruction on envoie 11. Enfin, lors d'un saut on y envoie 01.

O_2	O_1	O_0	ZF	SF	C_1
0	0	1	×	×	0
0	1	0	0	×	1
0	1	0	0	×	0
0	1	1	1	×	0
0	1	1	1	×	1
1	0	0	0	0	0
1	0	0	0	1	1
1	0	0	1	0	0
1	0	0	1	1	0
1	0	1	0	0	1
1	0	1	0	1	0
1	0	1	1	0	0
1	0	1	1	1	0
1	1	0	0	0	0
1	1	0	0	1	1
1	1	0	1	0	1
1	1	0	1	1	1
1	1	1	0	0	1
1	1	1	0	1	0
1	1	1	1	0	1
1	1	1	1	1	1

FIGURE 6 – Table de vérité du contrôleur de sauts

Penchons-nous maintenant sur le décodeur d'instructions.

4 Le décodeur d'instructions

Le décodeur d'instructions permet de déterminer quel opération va effectuer le processeur pour le cycle à venir. C'est lui qui lit les opcodes déterminés précédemment.

Gestion des sauts Lorsque le décodeur de sauts lit une instruction de type b, il se contente d'appeler l'Unité Arithmétique et Logique pour faire une soustraction. Il arme également la sortie *isJMP*.

Le schéma électronique du décodeur d'instructions est présenté à la figure 4. La partie qui décode le signal *ctrlUAL* correspond au tableau de KARNAUGH de la figure 8 duquel on extrait l'équation 3.

$$ctrlUAL = b_3.\neg b_2 + b_3.b_2\neg b_1 + b_3.b_1\neg b_0 \tag{3}$$

Cette équation nous permet alors de réaliser le circuit électronique décodant *ctrlAUL*. Les variables b_3, b_2, b_1 et b_0 correspondent aux bits de l'opcode.

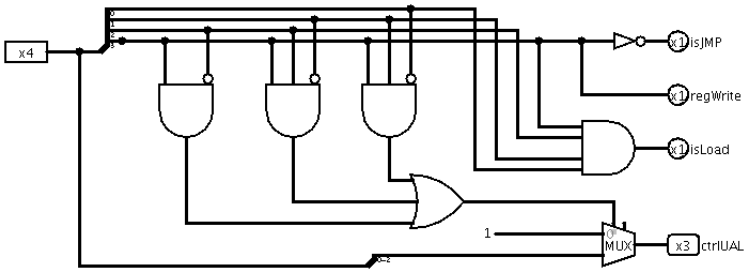


FIGURE 7 – Schéma électronique pour le décodeur d'instructions

b_3b_2 \ b_1b_0	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	1	1	0	1
10	1	1	1	1

FIGURE 8 – Tableau de KARNAUGH pour le décodage d'instructions.

5 La sélection des registres

Le sélecteur de registres permet de déterminer le registre dans lequel on va lire ou écrire un octet. Son circuit électronique est présenté à la figure 9. Son fonctionnement est relativement simple, on ne s'attardera donc pas dessus.

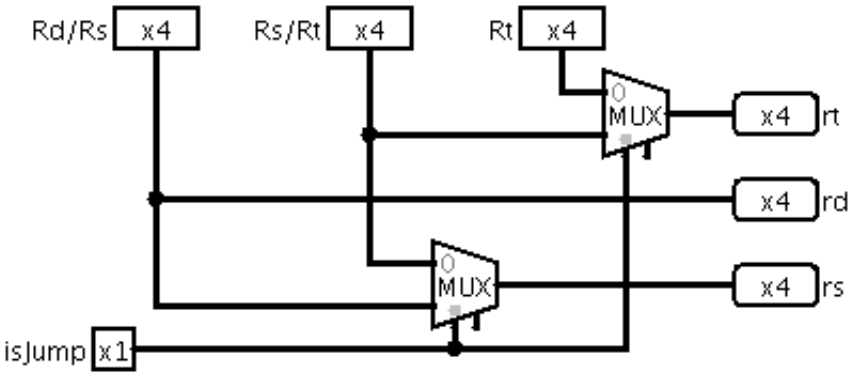


FIGURE 9 – Schéma électronique pour la sélection de registres

Maintenant que nous sommes en capacité de choisir un registre et l'opération qu'on souhaite y faire, réalisons le banc de registres.

6 Le banc de registres

Le banc de registres constitue la mémoire du processeur. Il possède 16 registres

d'un octet. Un tel circuit est composé de bascules D en cascade. Il y en a une pour chaque registre. Il est à la figure 6.
L'écriture d'un registre est possible uniquement lorsque *regWrite* est à vrai.

Sélecteur de registres Attention, le sélecteur à la figure 11, n'est pas le même que le sélecteur de registres présenté en amont. Celui-ci est spécifique au banc de registres. C'est le circuit *BC selec Reg* sur le schéma à la figure 6.

Erratum Sur la figure 6, le numéro des tunnels reliés aux registres ne sont pas bon, il faudrait les décrémenter de un. r_1 devient alors r_0, r_2 devient r_1, et cetera.

Trouvez à la figure 12 le schéma global du processeur Nono 1 utilisant les circuits présentés dans ce rapport.

7 Nono-1

PGCD On commence par utiliser le processeur en implémentant la fonction qui calcule le plus grand diviseur commun de deux nombres. Pour ce faire, nous traduisons d'abord le code C écrit dans le sujet (figure 13), puis, on traduit le code MIPS (figure 14) en hexadécimale (figure 15), en se basant sur les opcodes définis plus tôt et le format des instructions.

Ce programme nous donne bien le résultat attendu dans les registres : 3.

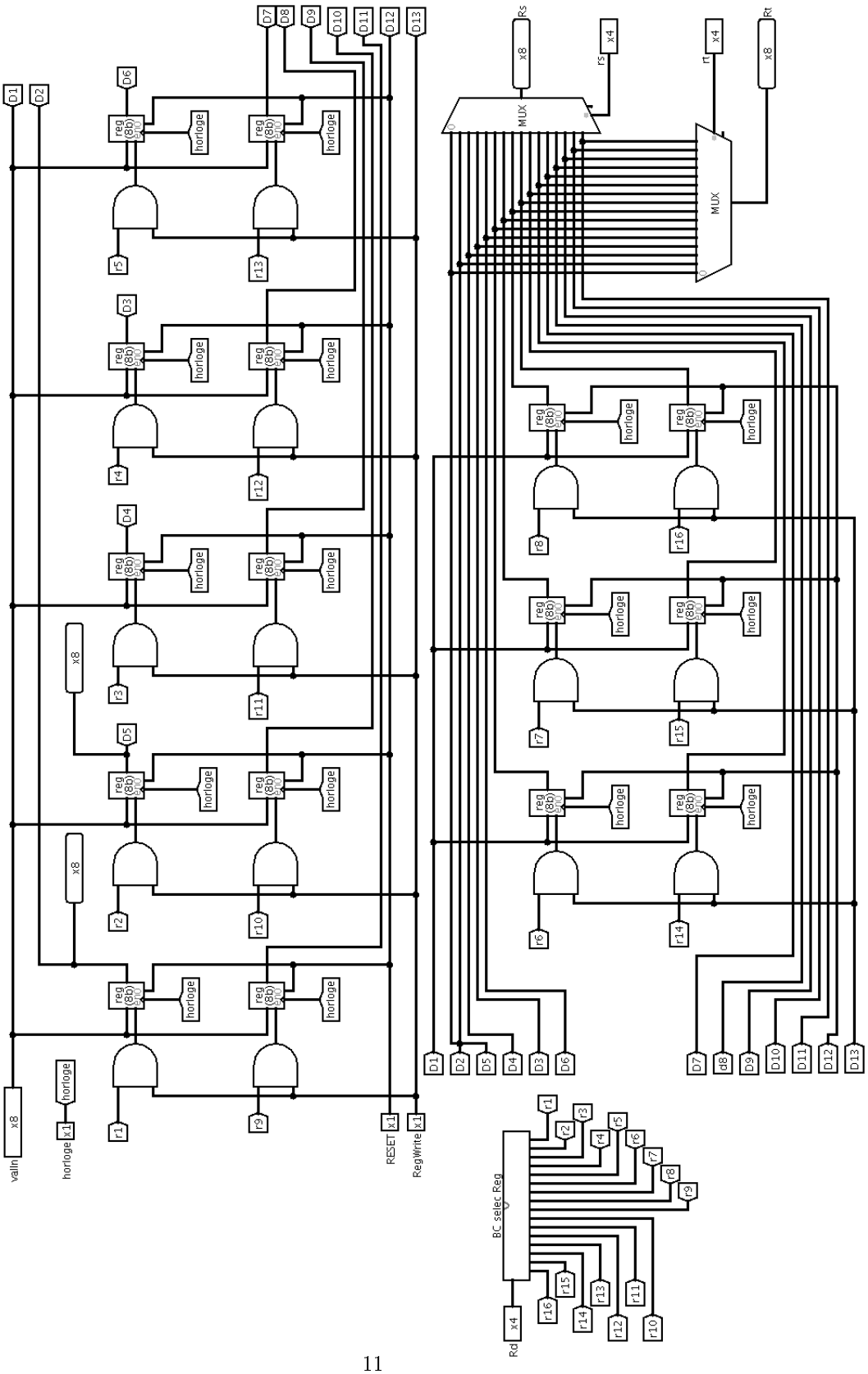
NTZ Pour aller plus loin nous avons codé un second programme. Nous avons choisi d'implémenter la fonction *ntz* présente sur l'examen de 2012/2013 de cette même matière.
À la figure 16 on a le code C de la fonction, à la figure 17 son code MIPS et à la figure 17 le code hexadécimal du programme

Le programme *ntz* stock son résultat dans le second registre.

HALT ! D'après le fonctionnement général de Nono 1, l'instruction *halt* donne l'adresse mémoire de la dernière instruction. Le problème est que le processeur ne s'arrête pas lorsqu'il a atteint cette instruction. et il va boucler sur cette instruction. Ainsi, si cette dernière est utilisée, elle va être exécutée à l'infini.

8 Nono-2

Un processeur Nono 2 est un processeur Nono 1 implémentant des routines. La gestion des routines suppose que le processeur est capable de gérer plus choses.



11

FIGURE 10 – Schéma électronique pour le banc de registres

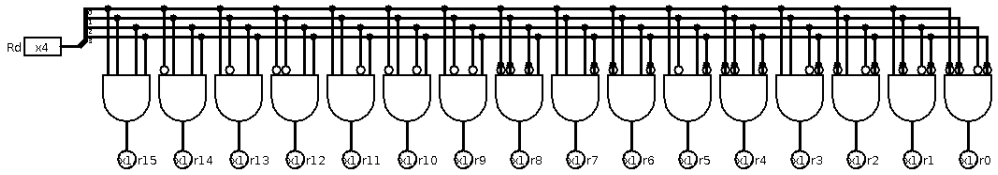


FIGURE 11 – Schéma électronique pour le sélecteur du banc de registres

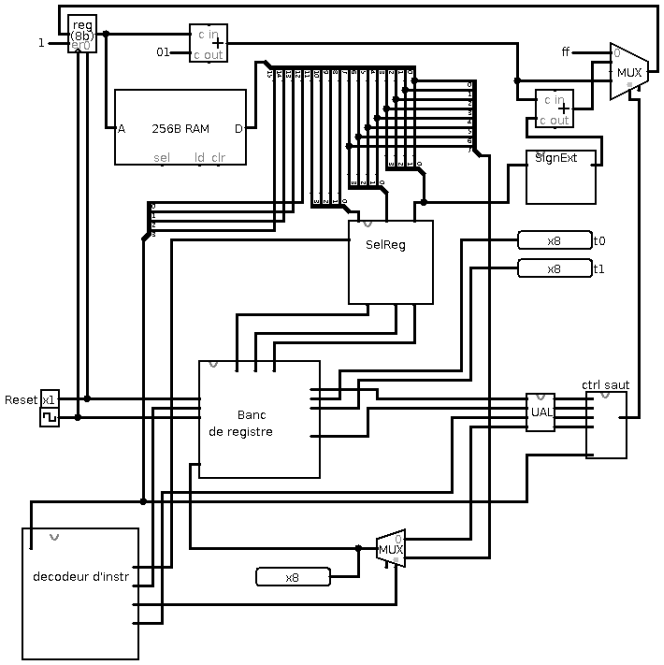


FIGURE 12 – Schéma de Nono 1

```
1  int i = 27;
2  int j = 24;
3  while (i != j)
4  {
5          if (i > j)
6          {
7                  i -= j;
8          }
9          else
10         {
11                 j -= i;
12         }
13 }
```

FIGURE 13 – Code C de la fonction *pgcd* tirée du sujet

```
1           .data
2           .text
3           .globl __start
4
5  __start:
6           li $t1, 27      #i
7           li $t0, 24      #j
8
9  while :
10          beq $t1, $t0, end_while
11 if:
12          ble $t1, $t0, else
13 then:
14          sub $t1, $t1, $t0
15          b end_if
16 else:
17          sub $t0, $t0, $t1
18 end_if:
19          b while
20 end_while:
21
22 # fin du programme
23 halt
```

FIGURE 14 – Code MIPS du programme *pgcd*

```
1 0xf11b # li $t0 24
2 0xf018 # li $t1 27
3 0x2105 # bed $t1 $t0 5
4 0x5102 # ble $t1 $t0 2
5 0x9110 # sub $t1 $t1 $t0
6 0x1001 # b 1
7 0x9001 # sub $t0 $t0 $t1
8 0x100a # b -6
9 0x0000 # halt
```

FIGURE 15 – Code hexadécimal du programme *pgcd* compilé pour NONO 1

```c
int ntz( unsigned int x )
{
        int n = 0;
        x = ~x & (x-1);
        while (x != 0)
        {
                n = n + 1;
                x = x >> 1;
        }
        return n;
}
```

FIGURE 16 – Code C de la fonction *ntz* tirée du sujet de l'examen de 2012-2013

```
        .data
        .text
        .globl __start

__start:
# initialisation des variables
        li $t0 32                   # x = 32
        li $t1 0                    # n = 0
        li $t3 1                    # pour -1
        li $t4 0                    # pour zero

        sub $t2 $t0 $t3             # x - 1
        not $t0 $t0                 # not x
        and $t0 $t0 $t2             # and(x-1)

while:
        beq $t0, $t4, end_while
        add $t1, $t1, $t3           # n = n + 1
        shr $t0, $t0, 1             # x = x >> 1
        b while
end_while:
        # fin du programme
        halt
```

FIGURE 17 – Code MIPS de la fonction *ntz*

```
1  0xf020  # li $t2 32
2  0xf100  # li $t1 0
3  0xf301  # li $t3 1
4  0xf400  # li $t4 0
5  0x9203  # sub $t2 $t0 $t3
6  0xc000  # not $t0 $t0
7  0xb002  # and $t0 $t0 $t2
8  0x2043  # beq $t0 $t4 3
9  0x8113  # add $t1 $t1 $t3
10 0xe003  # shr $t0 $t0 1
11 0x100c  # b -4
12 0x0000  # halt
```

FIGURE 18 – Code hexadécimal de la fonction *ntz*

Sauts longs Un appel à une fonction est traduit grossièrement en assembleur par un saut dans le code du programme. Les sauts longs (les instructions de type `j étiquette`) ne sont pas du même type que les sauts effectués par un `b`. Pour cela, il faut être capable d'incrémenter le pointeur d'instruction *PC*. Ce qui est déjà possible avec NONO 1.

Paramètres de fonctions et pile Lorsqu'on appelle une fonction, en général, c'est pour effectuer un calcul. Calcul qu'on souhaite effectuer sur des paramètres spécifiques. Il faut donc être capable de passer ces paramètres à une fonction mais également récupérer son résultat. Lorsqu'on utilise une fonction *simple* (prenant en paramètres des scalaires), il suffit d'initialiser des registres spécifiques pour que la fonction puisse récupérer les valeurs données. Puis la fonction modifie certains registres lorsqu'elle renvoie un résultat. Or, NONO 1 ne permet pas cela car il n'a pas de registres dédiés.
D'autre part, lorsque la fonction prend en paramètre des types plus complexes (un tableau par exemple) ou qu'elle appelle elle-même une fonction, il faut alors gérer les cadres de pile des fonctions. Ce qui n'est pas possible actuellement.

Tous ces éléments suggèrent également l'utilisation d'instructions que le processeur ne possède pas. Par exemple, les instructions de saut, les chargements en mémoire (gestion de la pile), etc. Pour cela, il faudrait, par exemple, étendre le jeu d'instructions actuel. Cela entraînerai alors une modification importante du processeur, notamment car la taille des opcodes ne serait plus la même.
Une seconde solution envisageable et réalisable, serait de sacrifier des instructions du jeu actuel au profil de nouvelles. Par exemple, une comparaison de type *inférieur ou égale* revient à faire une comparaison du type *supérieur*. On peut alors remplacer l'instruction de cette comparaison par une autre.

Malheureusement, par manque de temps, la réalisation du processeur NONO 2 reste à ce jour inachevée.

Conclusion

La réalisation d'un processeur n'est pas trivial. Un processeur est un composant relativement complexe. Mais la division de ce dernier par unités spécialisées permet de simplifier sa conception.

En réalisant nous même un processeur, on a put prendre conscience à quel point un langage d'assemblage est lié à une architecture physique.

Table des figures