X5I0030
Réalisation des processeurs Nono-1 et Nono-2

Corentin Chédotal

2 Décembre 2016

Table des matières

1 Introduction

Dans le cadre de l'Unité d'Enseignement **X5I0030 "Architecture des ordinateurs"** nous avons été amené à réaliser un processeur (le *Nono-1*) en utilisant le logiciel libre *Logisim*. Le cahier des charges du *Nono-1* était conséquent et donné dans le sujet. Ainsi on notera que le processeur devait posséder 16 registres de 8 bits, une mémoire permettant de stocker des instruction et justement un jeu de 16 instructions. Le but du projet était donc la réalisation de tous les sous-circuits nécessaire au bon fonctionnement du processeur, à son "assemblage" sous *Logisim* et aussi à son exécution. En effet il fallait aussi tester le bon fonctionnement de notre implémentation en exécutant un programme donné (effectuant le pgcd de deux entiers) et en codant et exécutant un programme de notre choix.

2 Implémentation du *Nono-1*

Le processeur *Nono-1* est composé de plusieurs sous-circuits. Ceux-ci ont été réalisés chacun de leur coté en profitant de la fonction de sous-circuits de *Logisim*. Puis ils ont été assemblés. Sera explicité les circuits "fait main".

La mémoire programme n'a pas été implémentée manuellement mais fait usage de l'objet RAM de *Logisim*. Le *Sign Extender* fait usage de la porte *Bit Extender* de *Logisim* réglée pour adapter son extension en fonction du signe. Enfin le *Program Counter* a été réalisé en faisant appel à un *Counter* préimplémenté dans le logiciel et qui a été réglé afin de ne pas revenir au début en cas d'*overflow*.

2.1 L'Unité Arithmétique et Logique (UAL)

2.1.1 Présentation

L'UAL est le coeur du processeur et est responsable de tous les calculs de celui-ci. C'est la première pièce du projet a avoir été implémentée.

2.1.2 Entrées

L'UAL reçoit trois entrées :
- `Rs` (8 bits) : Le premier registre que l'UAL sera amenée à lire pour faire ses opérations
- `Rt` (8 bits) : Le deuxième registre que l'UAL sera amenée à lire pour faire ses opérations, il est ignoré lorsque l'instruction `not` est reçue (opcode 0100)
- `ctrUAL` (3 bits) : Les trois derniers bits de l'opcode sont transmis à l'UAL afin de choisir quelle opération effectuer, le premier bit de l'opcode n'est pas transmis car il est dispensable

2.1.3 Sorties

L'UAL possède deux sorties :
- `res` (8 bits) : Le résultat de l'opération effectuée
- `flags` (4 bits) : Les *flags* retournés par l'UAL, ils sont testés quelque soit l'opération effectivement effectuée par l'UAL, ils sont codés dans cet ordre : CF,ZF,SF,OF [1]

2.1.4 Fonctionnement

En réalité l'UAL effectue tous les calculs en même temps et c'est `ctrUAL` qui par le biais d'un multiplexeur choisi quel résultat va effectivement être envoyé dans `res`. En ce qui concerne les *flags* ils sont testés indépendamment de l'opération effectuée, chacun de leur coté.

1. Respectivement *Carry Flag*, *Zero Flag*, *Sign Flag* et *Overflow Flag*

2.2 Le décodeur d'instructions

2.2.1 Présentation

Le décodeur d'instructions est le circuit responsable du contrôle de l'opcode des mots de la mémoire qui sont envoyés au processeur. Il libère alors les registres s'il faut écrire dessus, appelle un saut si besoin et transmet à l'UAL la partie de l'opcode la concernant.

2.2.2 Entrée

Le décodeur d'instruction ne reçoit qu'une seule entrée. Il s'agit de l'opcode (sur 4 bits) extrait de l'instruction sortant de la mémoire.

2.2.3 Sorties

Le décodeur d'instruction possède quatre sorties qui sont des bits de contrôles utilisés par les autres sous-circuits :

— `ctrUAL` (3 bits) : Les trois derniers bits de l'opcode sont envoyés à l'UAL quand l'instruction est un calcul, si l'instruction est un saut le décodeur d'instructions enverra les derniers bits de l'opcode de la soustraction à l'UAL (001)

— `isJMP` (1 bit) : Bit de contrôle envoyé à destination du sélecteur de registre pour lui indiquer que l'instruction est un saut

— `isLoad` (1 bit) : Bit de contrôle envoyé à destination du multiplexeur décidant si l'on charge un immédiat ou un résultat de l'UAL

— `regWrite` (1 bit) : Bit de contrôle envoyé au banc de registre afin de lui indiquer qu'il sera nécessaire d'écrire dans un registre d'après l'instruction reçue

2.2.4 Fonctionnement

Le fonctionnement du décodeur d'instructions suit la table de vérité qui suit, à l'exception des cas de saut, auquel cas `ctrUAL` se voit assigné la valeur 001, celle de la soustraction afin que le contrôleur de saut puisse faire usage des *flags* de l'UAL afin de faire ses comparaisons dans le cas de sauts conditionnel.

Partie 1									
2. Décodeur d'instructions									
Table de vérité									
e3	e2	e1	e0	isJMP	regWrite	isLoad	ctrUAL2	ctrUAL1	ctrUAL0
0	0	0	0	0	1	0	0	0	0
0	0	0	1	0	1	0	0	0	1
0	0	1	0	0	1	0	0	1	0
0	0	1	1	0	1	0	0	1	1
0	1	0	0	0	1	0	1	0	0
0	1	0	1	0	1	0	1	0	1
0	1	1	0	0	1	0	1	1	0
0	1	1	1	0	1	1	1	1	1
1	0	0	0	0	0	0	x	x	x
1	0	0	1	1	0	0	x	x	x
1	0	1	0	1	0	0	x	x	x
1	0	1	1	1	0	0	x	x	x
1	1	0	0	1	0	0	x	x	x
1	1	0	1	1	0	0	x	x	x
1	1	1	0	1	0	0	x	x	x
1	1	1	1	1	0	0	x	x	x

2.3 Le contrôleur de saut

2.3.1 Présentation

Le contrôleur de saut est responsable de la gestion du flux de lecture des différentes adresses mémoire. C'est lui qui va autoriser les sauts ou les arrêts du programme chargé en mémoire.

2.3.2 Entrées

Afin de faire son travail le contrôleur de saut a deux entrées :
— `flags` (4 bits) : Les *flags* de l'UAL, ils sont transmis au contrôleur de saut afin de s'en servir pour faire les tests requis par les sauts conditionnels
— `opcode` (4 bits) : L'opcode de l'instruction en cours est envoyée au contrôleur de saut afin de savoir s'il s'agit d'un saut, d'un arrêt ou d'un simple calcul et d'agir en conséquence

2.3.3 Sortie

Le contrôleur de saut n'a qu'une seule sortie. Elle est envoyée au multiplexeur gérant la prochaine adresse lue par le processeur. Elle est codée sur 2 bits ainsi :
— 00 : Comportement normal, la prochaine adresse lue est la suivante
— 01 : Arrêt immédiat du programme (cf paragraphe 3.1)
— 10 : Saut, le multiplexeur laissera passer la nouvelle adresse à lire

2.3.4 Fonctionnement

Le contrôleur de saut se sert de l'opcode afin de savoir s'il s'agit d'un saut, d'un arret ou d'un simple calcul. dans le premier cas il utilise aussi l'opcode afin de déterminer si'il s'agit d'un saut conditionnel. Auquel cas le contrôleur de saut examine le résultat du test équivalent effectué par le biais des *flags* transmis lors d'une soustraction. Dans le cas de l'arrêt et des instructions "classiques" (calculs, chargement d'un immédiat...) l'opcode seul est utilisé.

2.4 Le banc de registres

2.4.1 Présentation

Le banc de registre est le lieu de stockage temporaire du processeur. Il est composé dans le cas du *Nono-1* de 16 registres de 8 bits.

2.4.2 Entrées

Le banc de registres a sept entrées :
— `clk` (1 bit) : Les registres ont besoin de l'horloge afin de changer d'état
— `regWrite` (1 bit) : Bit de contrôle du décodeur d'instructions autorisant ou non l'écriture dans les registres
— `RESET` (1 bit) : Bit de contrôle permettant la remise à zéro des registres
— `Rd` (4 bits) : Le numéro du registre à accéder en lecture
— `Rs` (4 bits) : Le numéro de l'un des registre à accéder en écriture
— `Rt` (4 bits) : Même chose que plus haut
— `valin` (8 bits) : La valeur a écrire dans le registre selectionné par `Rd`

2.4.3 Sorties

Le banc de registres a deux sorties :
— `Rs` (8 bits) : La valeur stockée dans le registre sélectionné par l'entrée du même nom est transmis à l'UAL
— `Rt` (8 bits) : Même chose que plus haut

2.4.4 Fonctionnement

En réalité `valin` est bombardé dans tous les registres. Ce qui permet de sélectionner vraiment où elle va être sotckée est en réalité `regWrite`. En effet `Rd` va choisir par un démultiplexeur quel registre va effectivement être *enabled* ou déverrouillé pour permettre l'écriture. Pour la lecture de `Rs` et `Rt` c'est le même procédé mais inversé, utilisant un multiplexeur à la place.

2.5 Le sélecteur de registres

2.5.1 Présentation

Le sélecteur de registres est chargé de sélectionner dans l'instruction les bits qui correspondent aux indices de registres. En effet cela dépend du style d'instruction. Un bit de contrôle est ainsi employé afin de savoir de quel format s'agit-il et d'agir en conséquence.

2.5.2 Entrées

Le sélecteur de registres possède quatre entrées :
— `isJMP` (1 bit) : Bit de contrôle permettant d'indiquer si l'instruction reçue est de format F_1 ou F_3 (dans le cas d'un saut)
— `rd/rs` (4 bits) : La valeur des bits de l'instruction qui peuvent correspondre ou à l'indice de `Rd` ou à celui de `Rs`
— `rs/rt` (4 bits) : La valeur des bits de l'instruction qui peuvent correspondre ou à l'indice de `Rs` ou à celui de `Rt`
— `rt` (4 bits) : La valeur des bits de l'instruction qui peuvent correspondre à l'indice de `Rt`, est ignoré dans le cas du format F_3

2.5.3 Sorties

Le sélecteur de registre a trois sorties :
— `rd` (4 bits) : L'indice réel du registre `Rd` (le cas échéant, est flottant dans le cas d'une instruction de format F_3)
— `rs` (4 bits) : L'indice réel du registre `Rs`
— `rt` (4 bits) : L'indice réel du registre `Rt`

2.5.4 Fonctionnement

Suivant le bit de `isJMP` le démultiplexeur et les multiplexeurs sont actionnés de façon a sélectionner les bons bits dans l'instruction d'origine en fonction du format de celle-ci.

2.6 Assemblage final

L'assemblage final est fait suivant la Figure 2. du sujet. L'arrangement de la dite figure a été reproduit le plus fidèlement possible y compris au niveau des visuels des sous-circuits.
La fonctionnalité permettant de *RESET* le processeur est un bouton car cela semblait être la méthode la plus simple d'implémentation et d'utilisation par l'Utilisateur.

3 Utilisation du *Nono-1*

3.1 L'instruction `halt`

D'après la Figure 2. du sujet la commande `halt` est implémentée de la façon suivante. Le contrôleur de saut reçoit l'opcode signifiant la terminaison du programme. Il l'interprète et va envoyer au multiplexeur gérant les sauts du *Program Counter* (PC) une valeur particulière aux bits de selection du multiplexeur (dans notre implémentation

01). Cette sélection résulte en l'envoi par le multiplexeur de la constante `0xFF` comme instruction suivante pour l'exécution du programme. Cela implique donc deux contraintes. Tout d'abord la fameuse instruction localisée en `0xFF` ne doit pas contenir d'instruction réelle. De plus celle-ci étant la dernière instruction que le PC peut stocker il faut aussi s'assurer contre un *Overflow* de celui-ci, surtout si cela causerai son retour à l'instruction `0x00` qui elle peut être une instruction viable.

3.2 Le pgcd

Afin de tester le processeur implémenté nous devions traduire en assembleur *Nono-1* puis en code machine le programme calculant le pgcd tel que donné dans le sujet. C'est chose faite ci-dessous.
Voilà le programme calculant le pgcd traduit en assembleur *Nono-1* :

```
1        #while
2            beq $a0, $a1, 3
3        #if
4            ble $a0, $a1, 2
5        #then
6            sub $a0, $a0, $a1
7            b 1
8        #else
9            sub $a1, $a1, $a0
10       #endif
11           b -6
12       #endwhile
13           halt
```

Ce qui en code machine donne ceci [2] :

```
1        1010 0100 0101 0011
2        1101 0100 0101 0010
3        0001 0100 0100 0101
4        1001 0000 0000 0001
5        0001 0101 0101 0100
6        1001 0000 0000 1010
7        1000 0000 0000 0000
```

3.3 Le programme au choix : minmax

En plus de réaliser la traduction du programme pour le pgcd donné en C/C++ nous devions chercher et choisir un autre programme à écrire en c/C++, traduire en assembleur puis en code machine. C'est une fonction `minMax()` qui suivant un argument donne le minimum ou le maximum entre deux nombres.
Le code en C/C++ est le suivant :

```
1        int minMax(int arg, int a, int b) {
2            if (arg == 0) {
3                if (a <= b) {
4                    return(a);
5                } else {
6                    return(b);
7                }
8            } else {
```

2. En ce qui concerne le numéro des registres ont utilise le même codage que le MIPS.

```
9                   if (a >= b) {
10                      return(a);
11                  } else {
12                      return(b);
13                  }
14              }
15          }
```

Ce qui une fois traduit en assembleur *Nono-1* donne[3] :

```
1           #if1
2               bne $a0, $zero, 6
3           #then1
4           #if1a
5               bgt $a1, $a2, 3
6           #then1a
7               sub $a0, $a0, $a0
8               add $a0, $a0, $a1
9               b 2
10          #else1a
11              sub $a0, $a0, $a0
12              add $a0, $a0, $a2
13          #endif1a
14              b 6
15          #else1
16          #if1b
17              blt $a1, $a2, 3
18          #then1b
19              sub $a0, $a0, $a0
20              add $a0, $a0, $a1
21              b 2
22          #else1b
23              sub $a0, $a0, $a0
24              add $a0, $a0, $a2
25          #endif1b
26          #endif1
27              halt
```

Et en code machine on obtient donc ces suites de bits :

```
1               1011 0100 0000 0110
2               1110 0101 0110 0011
3               0001 0100 0100 0100
4               0000 0100 0100 0101
5               1001 0000 0000 0010
6               0001 0100 0100 0100
7               0000 0100 0100 0101
8               1001 0000 0000 0110
9               1111 0101 0110 0011
10              0001 0100 0100 0100
11              0000 0100 0100 0101
12              1001 0000 0000 0010
```

3. Le résultat du programme sera toujours donné dans le registre $a0.

```
13              0001  0100  0100  0100
14              0000  0100  0100  0101
15              1000  0000  0000  0000
```

De plus est mis à disposition dans l'archive mise en ligne sur *Madoc* un fichier hexadecimal [4] montrant le programme en fonctionnement pour *Nono-1*. Trois instructions auront été ajoutés afin de charger dans les registres les valeurs de test.

3.4 Le processeur *Nono-2*

Par manque de temps il ne m'a pas été possible de me pencher plus en détail sur le fonctionnement du processeur *Nono-2*.

4 Conclusion

Ce projet fut un projet qui m'a particulièrement plu. En effet il y a cela deux mois j'avais déjà mis les mains dans l'assembleur grâce au jeu *Shenzen I/O* et c'était très intéressant et enrichissant de rentrer en détail dans le fonctionnement bas niveau des programmes. De plus ayant toujours aimé l'électronique mais ne pouvant m'y mettre vraiment car étant trop occupé j'étais heureux de découvrir *Logisim* et d'avoir une matière dans laquelle nous ferions de l'électronique (bien que simulée). Ainsi ce projet fut la culmination de ces intérets.

Malheureusement rien n'est jamais parfait. En effet bien que je voulais m'investir au maximum dans ce projet je ne pus vraiment étant donné les nombreux autres projets à faire. Des choix de priorité durent être faits et étant seul sur ce projet là je ne risquais finalement que "de couler" la note de moi-même et pas d'un éventuel collègue si je ne parvenais pas à bout de ce projet.
D'ailleurs le fait d'être seul à travailler fut aussi la cause de grandes difficultés car j'ai eu et ai encore du mal avec les tableau de Karnaugh et j'ai pataugé beaucoup avant d'être débloqué par un autre groupe, que je remercie chaudement.

Enfin je dirais que ce projet fût très enrichissant et je me remettrai peut-être dessus durant mon temps libre pour le finir et pourquoi pas à terme pour réaliser physiquement *Nono-1*.

4. Il s'agit du fichier `minMax`