Groupe 584I X31I050

Mini-projet
Réalisation du processeur Irving

Architecture des ordinateurs

LE BADEZET Benoît
PLOUHINEC Glenn

Description du projet

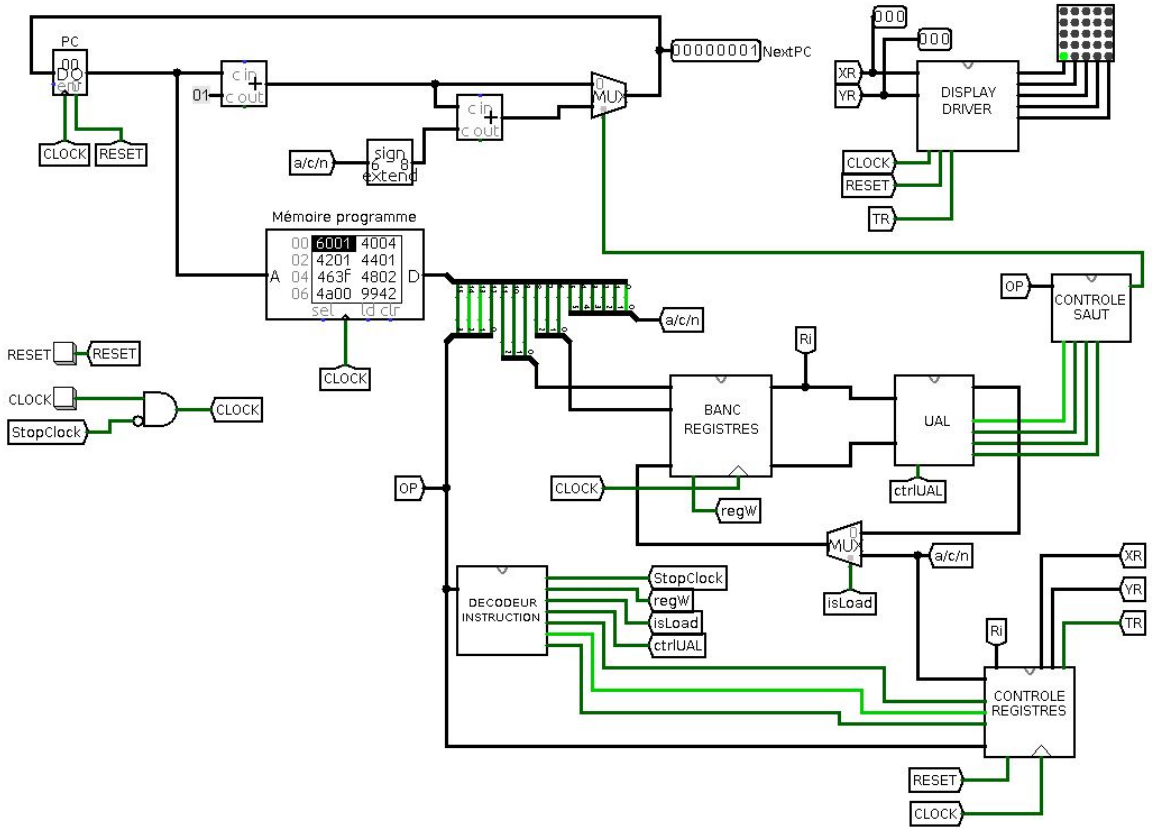
L’objectif de ce projet est de réaliser le circuit électronique du processeur Irving, sur le logiciel *Logisim*. Ce processeur permet alors de simuler les déplacements d’un robot tortue sur une grille de 5x5 cases (pouvant s’agrandir à du 32x32). Cette tortue ne se déplace alors sur la grille que verticalement ou horizontalement, selon l’orientation de cette dernière qui peut être modifiée suivant les quatre directions (haut, bas, droite, gauche). De plus, un dispositif de marquage peut être activé afin que le chemin emprunté par la tortue apparaisse sur la grille.

L’Irving possède 8 registres généraux de 6 bits permettant de mémoriser des informations, en l'occurrence des entiers allant de -32 à 31; un registre PC de 8 bits pointant en mémoire sur la prochaine instructions à exécuter; deux registres XR et YR de 5 bits conservant la position de la tortue sur la grille; un registre OR de 2 bits conserve l’orientation actuelle de la tortue, et un registre TR de 1 bit conserve la position du dispositif de marquage (1 pour “abaissé” et 0 pour “relevé).

Pour les besoin de ce projet, la taille de la grille du pilote d’affichage est restreinte à du 5x5. Nous devons alors réussir à faire se mouvoir cette tortue en utilisant les instructions du langage d’assemblage de l’Irving.

Le fichier turtle5x5.circ contient la majeure partie de l’implémentation de ce processeur. Il nous est alors demandé d’implémenter les modules manquants afin de compléter le circuit électronique du processeur, et faire fonctionner un programme permettant de faire se mouvoir la tortue.

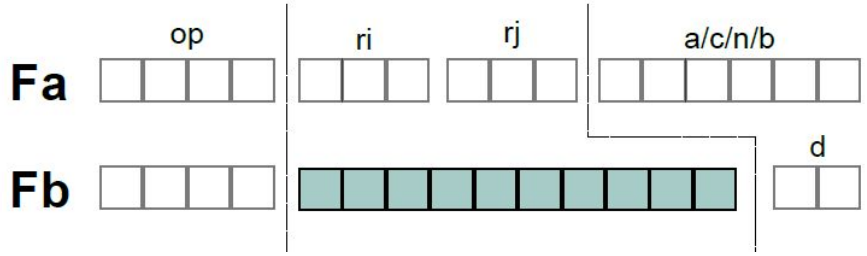
Le Badezet Benoit
Plouhinec Glenn



Le jeu d'instructions de l'Irving est décrit dans la table ci-dessous, cela correspond à toutes les instructions que le processeur peut interpréter.

Instruction	Opcode	Format	Action
`move` c	0000	F_a	Avance la tortue de c cases ($c \in [-32, 31]$)
`move` R_i	0001	F_a	Avance la tortue de R_i cases
`turn` d	0010	F_b	Tourne la tortue dans la direction d ($d \in [0, 3]$)
`turn` R_i	0011	F_a	Tourne la tortue dans la direction R_i
`load` R_i, n	0100	F_a	$R_i \leftarrow n$, avec $n \in [-32, 31]$
`add` R_i, R_j	0101	F_a	$R_i \leftarrow R_i + R_j$
`trace` b	0110	F_a	Définit le statut du marqueur (0 : levé, 1 : baissé)
`trace` R_i	0111	F_a	Définit le statut du marqueur
`beq` R_i, R_j, a	1000	F_a	Ajoute l'offset a à `PC` si $R_i = R_j$ ($a \in [-32, 31]$)
`bne` R_i, R_j, a	1001	F_a	Ajoute l'offset a à `PC` si $R_i \neq R_j$ ($a \in [-32, 31]$)
`bge` R_i, R_j, a	1010	F_a	Ajoute l'offset a à `PC` si $R_i \geqslant R_j$ ($a \in [-32, 31]$)
`bgt` R_i, R_j, a	1011	F_a	Ajoute l'offset a à `PC` si $R_i > R_j$ ($a \in [-32, 31]$)
`halt`	1111	F_a	Arrête définitivement l'exécution du programme

Chaque instruction est codée en binaire, sur 16 bits, suivant les formats F_a ou F_b décrits ci-dessous.



- L'*opcode* d'une instruction est assimilable, pour le processeur, à un identifiant unique, celui-ci est codé sur 4 bits.
- *ri* et *rj* correspondent à l'adresse de registres codée en binaire, sur 3 bits. Ainsi le registre R_0 aura l'adresse 000, le registre R_1 aura l'adresse 001, …, et R_7 111. Ces derniers agissent comme les variables que nous avons l'habitude de manipuler dans d'autres langages de haut niveau, nous pouvons alors effectuer une affectation avec l'instruction ***load*** R_i***, 5;*** (soit $R_i \leftarrow 5$) ou encore comparer deux registres (dont on a déjà affecté une valeur au préalable) avec une instruction de contrôle de saut tel que ***beq*** R_i, R_j***, 3;***
- *a/c/n/b* sont des "immédiats", ce sont des valeurs que nous pouvons directement renseigner dans le programme, comme dans l'exemple de l'instruction *load*.
- Le format d'instruction F_b n'est utilisé que par l'instruction *turn d*, si besoin est, on n'utilise alors que son opcode sur 4 bits, puis 10 zéros d'affilés, et enfin 2 bits indiquent la direction que doit prendre la tortue (entier de 0 à 3).

Réalisation des circuits de l'Irving

Il nous faut alors implémenter les modules suivants:

- L'UAL (l'Unité Arithmétique et Logique) est le module qui permet d'effectuer des calculs
- Le banc de registres, qui correspond à la mémoire du processeur
- Le contrôle de sauts est le module permettant de vérifier qu'une condition est vérifiée ou non, afin de réaliser un branchement, et ainsi, si une condition n'est pas vérifiée, l'exécution du programme se déroule séquentiellement.
- Le décodeur d'instructions, qui prend en entrée l'opcode de l'instruction courante et positionne les différents indicateurs en fonction du chemin de données de cette instruction.

1 - L'UAL

L'UAL prend en entrée deux valeurs A et B sur 6 bits, et un bit de sélection SEL. Le but est alors de réaliser l'addition de A+B, ou bien la soustraction A-B, selon la valeur du bit de sélection : si ce dernier vaut 0, on effectue alors l'addition A+B, et s'il vaut 1, on réalise alors la soustraction A-B. On se servira alors du multiplexeur de Logisim afin d'effectuer le choix de l'opération à faire en fonction du bit de SEL.

Pour effectuer une addition binaire, on utilise le module "Adder" présent dans Logisim. L'addition se fait alors aisément en connectant A et B aux entrées du module "Adder", et en récupérant le résultat à la sortie de ce dernier.

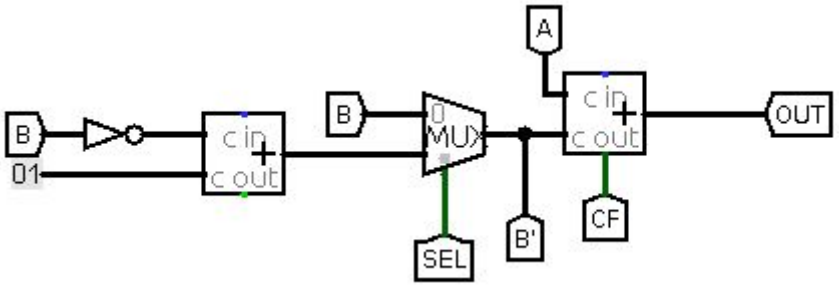
Pour effectuer la soustraction de deux nombres binaires, nous pouvions alors simplement utiliser le module "Substractor" de Logisim, mais il nous était alors demandé d'effectuer l'opération en utilisant intelligemment l'opération de complément à deux. En effet, il est possible d'effectuer l'opération A-B en additionnant le complément à deux de B, ce qui donnerai A-B = A+(cà2(B)). Pour rappel, le complément à deux d'un nombre binaire consiste à inverser tous les bits, et ajouter 1 au bit de poids faible (ou "lsb", le bit le plus à droite).

Prenons par exemple A = 010011, et B = 0010010, avec son complément à deux B' = 110110, nous obtenons les opérations :

A-B :	010011	A+B' :	010011
	- 001010		+ 110110
	1		1 1 11
	001001		001001

Le résultat est identique, ce qui nous permet donc d'implémenter la soustraction en utilisant l'additionneur de Logisim, et en calculant le complément à deux de B qui est obtenu en additionnant $\overline{B}$ avec une constante qui vaut 00 0001 (01 en hexadécimal).

Enfin, afin de simplifier un peu le circuit, nous plaçons notre multiplexeur de façon à ce qu'il choisisse de faire entrer dans notre additionneur, soit B, soit B', et cela en fonction du bit de SEL. De cette façon, nous avons alors fini d'implémenter la partie "calculs" de l'UAL.



Viennent maintenant les flags (ou drapeaux) : CF, SF, ZF, OF. Ceux-ci sont des indicateurs qui nous apportent des informations sur le résultat d'une opération arithmétique :

- Le Carry Flag (CF) indique qu'une retenue a été générée sur le bit de poids fort ("msb", le plus à gauche), générant ainsi un dépassement de capacité.
- Le Sign Flag (SF) est positionné si le résultat de l'opération est négatif (donc si le msb vaut 1).
- Le Zero Flag (ZF) est positionné si le résultat de l'opération est nul.
- L'Overflow Flag (OF) indique un changement anormal de signe, utilisé uniquement si le msb du résultat diffère du msb des deux opérandes.

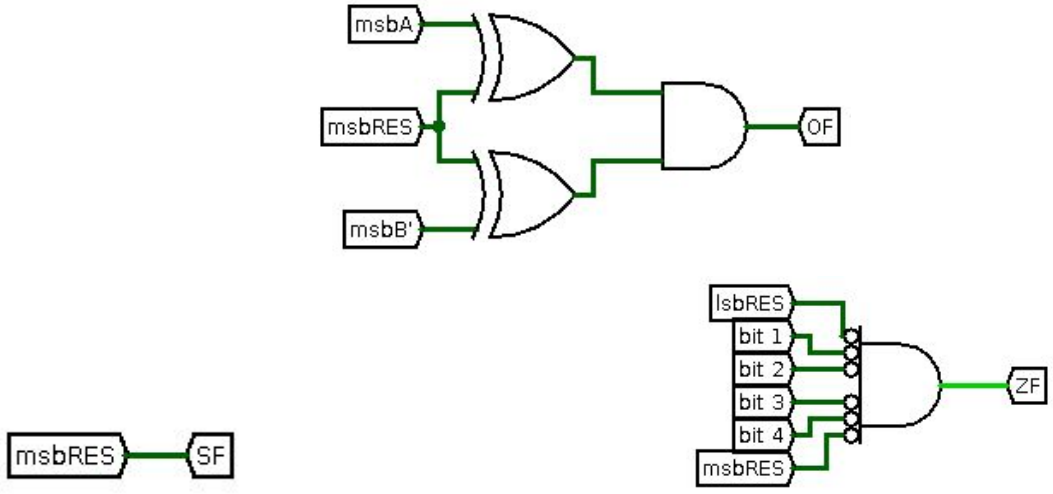
Notre UAL doit alors - en plus de nous transmettre la valeur du résultat sur sa sortie - nous indiquer la valeur de chacun de ces flags, ce qui nous donne alors un total de 5 sorties.

Le Carry Flag est généré automatiquement par le module "Adder" de Logisim, dont nous nous servons alors.

Le Sign Flag est alors simple à implémenter : il suffit de vérifier que le bit de poids fort du résultat vaut 1.

Le Zero Flag vérifie avec une porte logique ET à 6 entrées, que chacun des bits du résultat vaut 0.

L'Overflow Flag est alors géré avec deux portes logiques OU exclusifs (XOR) afin de vérifier que le msb du résultat diffère des msb des deux opérandes. On fait alors attention à regarder le msb de A, et la sortie du multiplexeur nous redonne alors soit la valeur de B, soit de son complément à deux, nous regardons alors le msb de la valeur en sortie du multiplexeur que nous avons nommé B' dans nos tunnels, afin de simplifier le circuit.



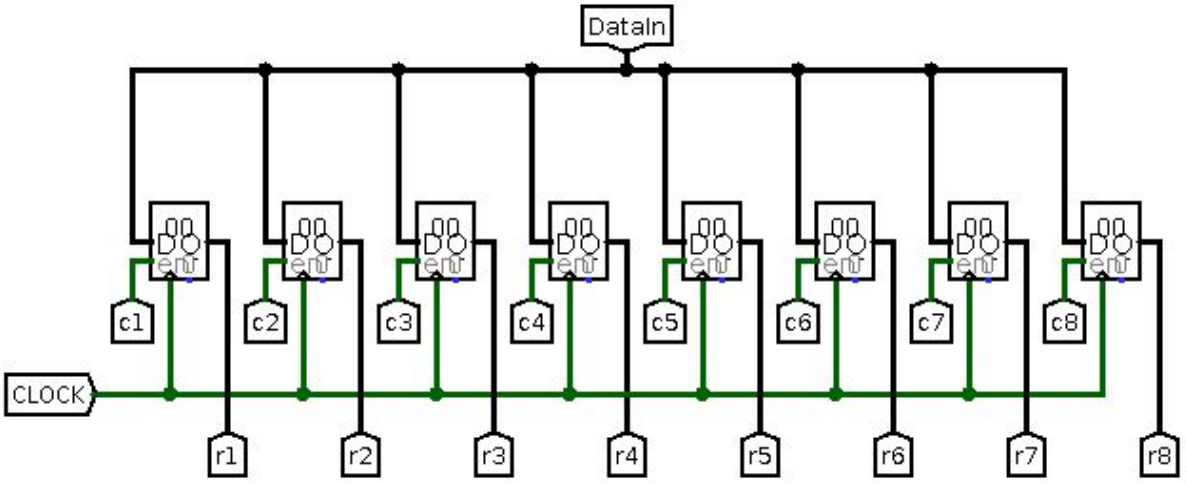
L'opération d'addition et de soustraction étant gérées correctement, et les flags associés étant implémentés, notre UAL est alors terminée.

2 - Le banc de registres

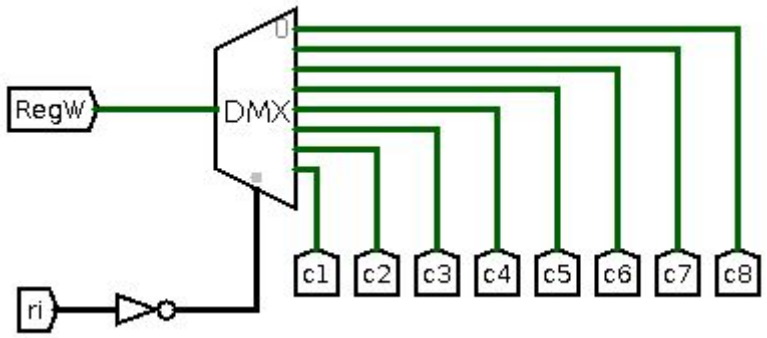
Le banc de registres de l'Irving est formé de 8 registres de 6 bits chacun. Ce module reçoit en entrée les valeurs ri et rj, sur 3 bits, qui correspondent alors à l'adresse d'un des 8 registres. Ainsi, si ri vaut 010, nous devons gérer la valeur contenue dans le registre R_2. L'entrée DataIn, sur 6 bits, représente une information que nous devons sauvegarder dans nos registres. Le bit d'entrée RegW correspond à "RegisterWriter", ce qui signifie que quand ce bit est à 1, on autorise l'écriture dans un registre. Le bit CLOCK sert à synchroniser nos registres sur les fronts d'horloge.

Les sorties R_i et R_j, sur 6 bits, contiennent les valeurs contenues dans les registres auxquels on a accédé grâce aux valeurs de ri et rj.

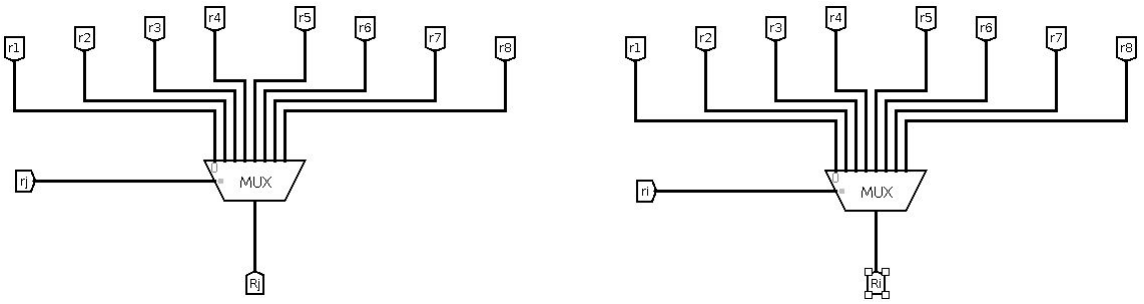
Pour commencer à implémenter ce module, nous utilisons les registres de Logisim en modifiant leur capacité de mémoire (6 bits de stockage). Nous disposons donc les 8 registres auxquels nous relions l'entrée d'horloge au bit de CLOCK, et la valeur de DataIn est alors reliée à chacun des registre (sans les splitter), afin que nous puissions par la suite choisir dans quel registre nous voulons stocker l'information contenue dans DataIn.



La suite du travail consiste à "choisir" le registre dans lequel nous voulons écrire/stocker l'information, et cela en fonction de la valeur de ri. La notion "d'écriture" dans un registre étant alors gérée par le bit RegW, nous disposons donc de RegW et ri pour écrire dans un registre en particulier, comme dit précédemment, si ri vaut 000, on écrira dans le registre R_0, soit le premier de nos huit registres. C'est ce qui est alors effectué en utilisant un démultiplexeur, lorsque RegW est à 1, on regarde la valeur de ri pour envoyer un signal au i-ème registre, en lui disant alors de mémoriser la valeur de DataIn courante, en se branchant sur l'entrée "enable" du registre. Le démultiplexeur nous permet alors de gérer 8 signaux différents à sa sortie, chacun permettant alors de gérer un i-ème registre. Notons que la porte logique "NON" a été placée devant ri seulement dans le but d'avoir un résultat visuel plus adéquat avec nos choix de numéros de registres. Ainsi, le multiplexeur nous sert ainsi d'alternative là où d'autres groupes de travail ont utilisé un décodeur.



Enfin, lorsque les informations sont stockées dans les registres, une fois qu'on a écrit dans ces derniers, il nous faut alors récupérer l'information d'un registre en particulier, et ce, encore une fois, grâce à la valeur de ri et rj. Ainsi, lorsqu'on voudra accéder à la valeur du i-ème registre, on se servira de la valeur de ri (ou rj, cela dépend alors du rôle de l'instruction…) pour retransmettre la valeur du registre dans la sortie R_i (ou R_j).

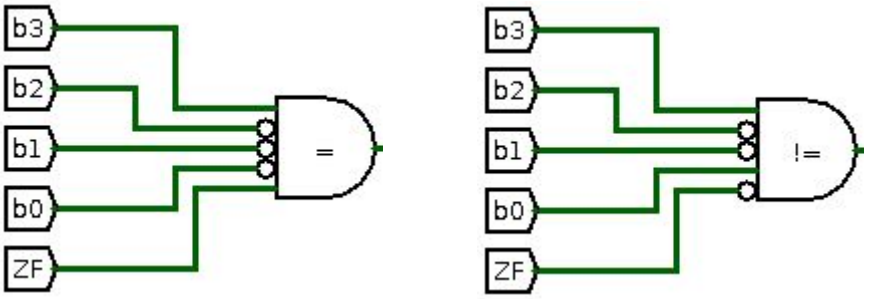


Pour être plus précis au niveau de l'écriture, on remarque que seules les instructions *load* R_i *, a;* et *add* R_i, R_j *; (*soit $R_i \leftarrow R_i + R_j$) permettent d'écrire dans un registre, et que donc seule "l'adresse" ri, ou plutôt la valeur de ri (conformément au format de l'instruction F_a) est utile pour écrire dans un registre. Cependant, pour accéder aux registres, il existe plusieurs instructions qui nécessitent d'accéder à deux registres en particulier, et notamment les sauts d'instructions (beq, bne, bge, bgt).

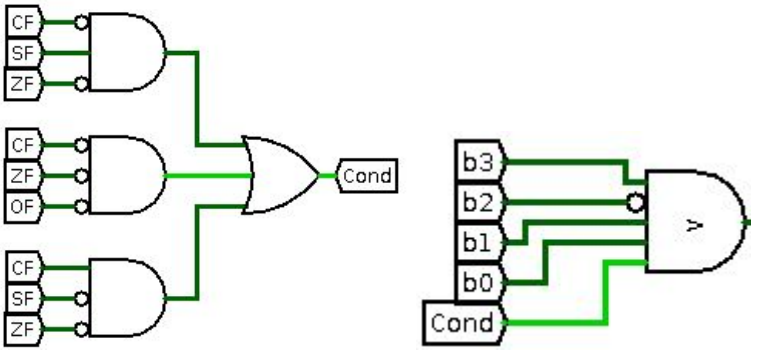
3 - Le contrôle de sauts

Le module de contrôle de sauts détermine la valeur du signal de sortie nextPCsel en fonction de l'opcode de l'instruction, dont nous disposons en entrée, et de la valeur des entrées ZF, SF, CF, et OF. Il nous faut donc vérifier que l'opcode reçu corresponde à celui d'une instruction parmi beq, bne, bge, et bgt, et que cette même instruction vérifie la condition lui correspondant. On va donc vérifier que pour une instruction *beq*, on ait bien l'opcode 1000, et que la condition $R_i = R_j$ est bien respectée. Pour *bne*, on vérifie que l'opcode soit 1000, et la condition $R_i \neq R_j$ est respectée. Pour *bge*, l'opcode doit être 1010 et la condition $R_i \geq R_j$. Pour *bgt*, l'opcode est 1011 et la condition $R_i > R_j$.

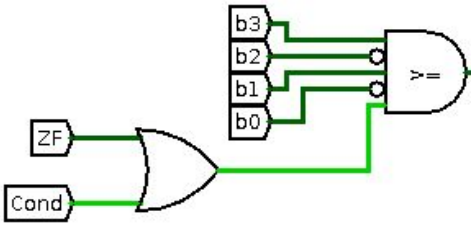
Les conditions des instructions peuvent alors être vérifiées grâce aux entrées ZF, SF, CF, et OF fournies par l'UAL suite à une opération arithmétique sur R_i et R_j. Ainsi, en effectuant l'opération $R_i - R_j$, si le résultat vaut 0, on est assuré que $R_i = R_j$, on obtient donc l'information avec le bit ZF à 1. Et sur la même logique, on a donc pour $R_i \neq R_j$, $R_i - R_j \neq 0$ donc ZF doit être à 0 : soit $\overline{ZF}$.



La plus grosse difficulté d'implémentation de ce module reste le fait de déterminer sous quelles conditions est vérifié $R_i > R_j$. Une première approche naïve consisterait à considérer nos registres 6 bits comme non signés (≥ 0), on aurait donc simplement $\overline{SF}$ puisque le résultat de l'opération $R_i - R_j$ serait toujours positif. Seulement, puisqu'on considère ici les nombre signés en complément à deux, il faut aussi prendre en compte les cas où l'on soustrait des nombres négatifs. Viennent alors une longue série de tests pour tenter de déterminer quels flags nous permettent de dire si $R_i > R_j$. En effectuant des tests au niveau de l'UAL et en relevant de nombreux cas où $R_i > R_j$, nous obtenons la table de vérité ci-dessous. Après simplifications par Karnaugh, on obtient $\overline{CF}.\overline{ZF}.\overline{OF} + \overline{CF}.SF.\overline{ZF} + CF.\overline{SF}.\overline{ZF}$



Pour ce qui est de déterminer $R_i \geq R_j$, il suffit simplement d'associer les résultats trouvés pour $R_i > R_j$ et $R_i = R_j$ pour trouver sa table de vérité.



Voici la table de vérité permettant de modéliser le comportement des instructions à traiter, en fonction des flags CF, SF, ZF et OF :

CF	SF	ZF	OF	beq (Ri = Rj)	bne (Ri ≠ Rj)	bgt (Ri > Rj)	bge (Ri ≥ Rj)
0	0	0	0	0	1	1	1
0	0	0	1	0	1	0	0
0	0	1	0	1	0	0	1
0	0	1	1	1	0	0	1
0	1	0	0	0	1	1	1
0	1	0	1	0	1	1	1
0	1	1	0	1	0	0	1
0	1	1	1	1	0	0	1
1	0	0	0	0	1	1	1
1	0	0	1	0	1	1	1
1	0	1	0	1	0	0	1
1	0	1	1	1	0	0	1
1	1	0	0	0	1	0	0
1	1	0	1	0	1	0	0
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	1

Notons que d'autres simplifications auraient pu être faites, notamment au niveau des opcodes, étant donné que nous n'en utilisons que les deux derniers bits pour faire coïncider les conditions.

4 - Le décodeur d'instructions

Pour implémenter le décodeur d'instructions, nous devons identifier quels indicateurs doivent être activés en fonction de l'opcode des instructions, il s'agit donc d'identifier le rôle de chacun de ces indicateurs, qui sont les sorties de notre module.

- StopClock est à 1 lorsque l'instruction halt est reçue.
- regW, comme vu dans la partie du banc de registres, seules les instructions load et add doivent activer cette sortie.
- isLoad ne sert que pour faire l'affectation d'une valeur à un registre, soit l'instruction load.
- ctrlUAL correspond aux sauts d'instructions, soit beq, bne, bge te bgt.
- XRen correspond à la position de la tortue sur la grille (via les registres XR et YR), soit les deux instructions move.
- TRen sert au dispositif de marquage, donc les deux instructions trace.
- ORen sert à l'orientation de la tortue, donc les instructions turn.

On obtient donc la table de vérité suivante :

	StopClock	regW	isLoad	ctrlUAL	XRen	TRen	ORen
move c	0	0	0	0	1	0	0
move Ri	0	0	0	0	1	0	0
turn d	0	0	0	0	0	0	1
turn Ri	0	0	0	0	0	0	1
load Ri, n	0	1	1	0	0	0	0
add Ri, Rj	0	1	0	0	0	0	0
trace b	0	0	0	0	0	1	0
trace Ri	0	0	0	0	0	1	0
beq Ri, Rj, a	0	0	0	1	0	0	0
bne Ri, Rj, a	0	0	0	1	0	0	0
bge Ri, Rj, a	0	0	0	1	0	0	0
bgt Ri, Rj, a	0	0	0	1	0	0	0
halt	1	0	0	0	0	0	0

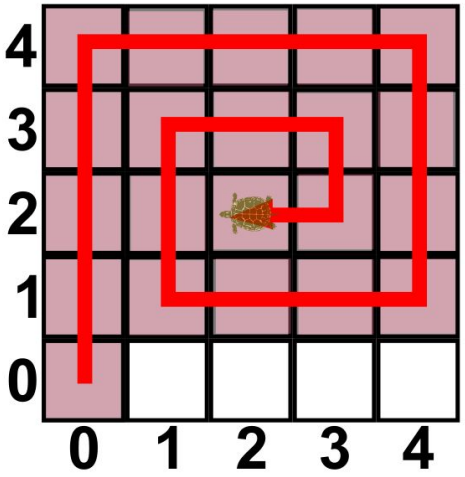
Des simplifications au niveau des opcodes ont été effectuées dans le circuit, par exemple pour l'indicateur ctrlUAL, les instructions beq, bne, bge et bgt ont toutes leur opcode commençant par 10xx, on contrôle alors les deux premiers bits pour gérer cet indicateur.

Ainsi, en notant les 4 bits de l'opcode xxxx avec respectivement "bit4", "bit3", "bit2", bit1"nous avons :

- StopClock = $bit4.bit3.bit2.bit1$
- regW = $\overline{bit4}.bit3.\overline{bit2}$
- isLoad = $\overline{bit4}.bit3.\overline{bit2}.\overline{bit1}$
- ctrlUAL = $bit4.\overline{bit3}$
- XREN = $\overline{bit4}.\overline{bit3}.\overline{bit2}$
- TREN = $\overline{bit4}.bit3.bit2$
- OREN = $\overline{bit4}.\overline{bit3}.bit2$

Utilisation de l’Irving

Dans cette partie, on utilise alors le processeur construit pour exécuter un programme de dessin. On va alors créer un code que l’Irving peut interpréter en utilisant le jeu d’instructions à notre disposition. L’objectif est donc de faire parcourir un tracé en colimaçon à la tortue, en utilisant une structure de boucle.



Pour réaliser ce dessin, on implémente en premier lieu un programme simple sans structure de boucle afin de bien cerner l’utilisation des instructions dans ce processeur.

```
trace 1;        0110 000 000 000001 (6001)
move 4;         0000 000 000 000100 (0004)
turn 1;         0010 0000000000 01  (2001)
move 4;         0000 000 000 000100 (0004)
turn 2;         0010 0000000000 10  (2002)
move 3;         0000 000 000 000011 (0003)
turn 3;         0010 0000000000 11  (2003)
move 3;         0000 000 000 000011 (0003)
turn 0;         0010 0000000000 00  (2000)
move 2;         0000 000 000 000010 (0002)
turn 1;         0010 0000000000 01  (2001)
move 2;         0000 000 000 000010 (0002)
turn 2;         0010 0000000000 10  (2002)
move 1;         0000 000 000 000001 (0001)
turn 3;         0010 0000000000 11  (2003)
move 1;         0000 000 000 000001 (0001)
halt;           1111 000 000 000000 (f000)
```

On traduit donc ces instructions en binaire suivant le format F_a ou F_b, on associe donc les quatre premiers bits de l’opcode à l’instruction, les valeurs ri et rj sont inutilisées, et on traduit les valeurs numériques en binaire pour obtenir l’équivalent de ce code en binaire. Puis nous devons traduire chacune de ces instructions en hexadécimal afin de stocker le programme dans la mémoire de l’Irving.

Suite à ça, il nous faut alors implémenter un programme utilisant une structure de boucle, on commence donc par faire un algorithme en pseudo-code afin de le traduire par la suite en langage d'assembleur Irving.

```
mouvements ← 4;
tourne ← 1;
i ← 1;
j ← -1;
ctrl ← 2;
cond ← 0;
Tant que mouvements ≠ cond faire                    // TQ le nombre de mouvements ≠ 0
        Si ctrl = cond Alors                        // si ctrl a été assez décrémenté
                mouvements ← mouvements + j;        // mouvements--;
                ctrl ← 2;                           // on réinitialise ctrl
        Sinon
                avancer de mouvements cases;
                tourner dans la direction tourne;
                tourne ← tourne + i;                // tourne++; avec tourne ∈ [0,3]
                ctrl ← ctrl + j;                    // ctrl--;
        Fin si
Fin Tant que
exit;
```

Pour traduire ce programme en assembleur Irving, la méthode reste la même que précédemment, à la différence qu'il nous faut maintenant gérer les contrôles de sauts. Ainsi, lorsqu'on utilise les instructions de sauts (en l'occurrence bne pour dire que mouvements $\neq$ cond), en comparant les deux registres R_i et R_j, on doit également ajouter l'offset "a" à PC, cela signifie que lorsque la condition est vérifiée, nous allons "sauter" les a instructions qui suivent notre condition. Donc en utilisant l'instruction ***bne R4, R5, 2;*** on dit au processeur de "sauter" 2 instructions plus loin si cette condition est vérifiée. Pour la condition de boucle, on utilisera ***bne R0, R5, -8;*** ce qui signifie que l'on revient 8 instructions en arrière dans le code, sans oublier de traduire cette valeur négative par son complément à deux.

Voici le programme en assembleur avec la structure de boucle, et sa traduction en binaire et hexadécimal :

```
trace 1;                                0110 000 000 000001     6001
load R0, 4;                             0100 000 000 000100     4004
load R1, 1;                             0100 001 000 000001     4201
load R2, 1;                             0100 010 000 000001     4401
load R3, -1;                            0100 011 000 111111     463F
load R4, 2;                             0100 100 000 000010     4802
load R5, 0;                             0100 101 000 000000     4A00
        bne R4, R5, 2;                  1001 100 101 000010     9942
                add R0, R3;             0101 000 011 000000     50C0
                load R4, 2;             0100 100 000 000010     4802
        move R0;                        0001 000 000 000000     1000
        turn R1;                        0011 001 000 000000     3200
        add R1,R2;                      0101 001 010 000000     5280
        add R4,R3;                      0101 100 011 000000     58C0
bne R0, R5, -8;                         1001 000 101 111000     9178
halt;                                   1111 000 000 000000     F000
```

Remarquons que notre instruction jouant le rôle de boucle peut être remplacée par bgt R0, R5, -8; ce qui correspond à “Tant que mouvements > 0 faire…”, on peut alors remplacer l’avant dernière instruction “9178” par “b178” pour obtenir un résultat équivalent au niveau du dessin sur le pilote d’affichage.

Conclusion

Ce mini-projet nous a permis de mieux comprendre le fonctionnement d’un processeur, ainsi que d’approfondir nos connaissances du cours d’Architecture des Ordinateurs, notamment au niveau des circuits combinatoires et du langage assembleur.

Nous avons eu l’occasion de travailler avec d’autres binômes afin de mieux appréhender les problèmes rencontrés, nous avons pu comparer nos résultats et échanger quelques programmes afin de mettre à l’épreuve nos implémentations. Certains doutes subsistent malgré tout au niveau des flags, et du module de contrôle de saut, et notamment l’instruction bgt dont nous avons observés de nombreuses implémentations différentes.