Architecture des ordinateurs

Projet de travaux pratiques

Table des matières

A- Implémentation du nono1

1) Réalisation de l'UAL

L'UAL est l'unité de calcul arithmétique, elle sert pour la majeure partie des instructions de notre processeur. Effectivement, mis-à-part les opérations arithmétiques classiques, elle est aussi nécessaire aux comparaisons au sein des opérations de branchement.

Ce module écrit dans un registre le résultat, et en sortie, affiche les quatre flags (OF, CF, SF, ZF) résultants.

Les instructions demandant un calcul ont toutes le bit de poids fort à 0. Dans les tableaux qui suivent, nous avons donc seulement les 3 bits de poids faible.

OpCode	Opération en sortie
000	halt
001	add
010	sub
011	or
100	and
101	not
110	shl
111	shr

L'ual est placée en annexe.

2) Réalisation du décodeur d'instructions

Le décodeur d'instructions prend en entrée l'OpCode et renvoie la valeur sur 1 bit en sortie correspondant au type de l'opération. Pour l'UAL, on renvoie à la place les 3 bits de poids fort.

Code abcd	regWrite	IsJump	IsUal	isLoad
0000	0	0	1	0
0001	1	0	1	0
0010	1	0	1	0
0011	1	0	1	0

0100	1	0	1	0
0101	1	0	1	0
0110	1	0	1	0
0111	1	0	1	0
1000	0	1	1	0
1001	0	1	1	0
1010	0	1	1	0
1011	0	1	1	0
1100	0	1	1	0
1101	0	1	1	0
1110	0	1	1	0
1111	1	0	0	1

Les équations résultantes sont :
regWrite = (b+c+d)*¬ a
isLoad = (a*b*c*d)
isJMP = a*((c*¬ d)+¬ b+¬ c)
UAL = ¬ a, et individuellement b, c et d.

Le décodeur d'instructions est placé en annexe.

3) Réalisation du module de contrôle de saut

Le module de contrôle de saut prend en entrée l'opCode et les quatre flags issus de l'UAL et retourne en sortie deux bits « nextPCselect »
BCD sont les 3 bits de poids faible de l'opCode, et Saut indique le type de branchement. Notons que load c'est pas un branchement, mais son opCode étant « 1111 », il est traité dans le module de contrôle de saut comme une « instruction suivante »

Ce module renvoie sur 2 bits une valeur indiquant si le registre PC doit :

- Aller à l'instruction suivant
- Faire un saut suivant l'instruction
- Arrêter l'exécution

nextPCselect	q1q0
HALT	11
Suivant	10
Saut	.01
/	.00

B	C	D	Saut
0	0	0	b
0	0	1	beq
0	1	0	bne
0	1	1	bge
1	0	0	ble
1	0	1	bgt
1	1	0	blt
1	1	1	load

Branchement	Etat des flags
Beq	ZF = 1
Bne	ZF = 0
Blt	SF !=OF et ZF= 0
Ble	SF != OF ou ZF = 1
Bgt	SF = OF et ZF = 0
Bge	SF = OF ou ZF = 0

Pour ce module de contrôle de saut, nous avons remarqué que le CarryFlag et le bit de poids fort de l'opCode sont inutiles. Nous les avons donc ignorés pour les tables de vérités, ce qui simplifie largement le résultat. Pour simplifier la lecture, nous avons jugé bon de colorer les cases.

B	C	D		ZF	SF	OF	Saut
0	0	0	**b**	0	0	0	1
0	0	0	no Cond	0	0	1	1
0	0	0		0	1	0	1
0	0	0		0	1	1	1
0	0	0		1	0	0	1
0	0	0		1	0	1	1
0	0	0		1	1	0	1
0	0	0		1	1	1	1
0	0	1	**beq**	0	0	0	0
0	0	1	ZF = 1	0	0	1	0
0	0	1		0	1	0	0
0	0	1		0	1	1	0
0	0	1		1	0	0	1
0	0	1		1	0	1	1
0	0	1		1	1	0	1
0	0	1		1	1	1	1
0	1	0	**bne**	0	0	0	1
0	1	0	ZF = 0	0	0	1	1
0	1	0		0	1	0	1
0	1	0		0	1	1	1
0	1	0		1	0	0	0
0	1	0		1	0	1	0
0	1	0		1	1	0	0
0	1	0		1	1	1	0
0	1	1	**bge**	0	0	0	1
0	1	1	SF = OF	0	0	1	0
0	1	1	or	0	1	0	0
0	1	1	ZF = 1	0	1	1	1
0	1	1		1	0	0	1
0	1	1		1	0	1	1
0	1	1		1	1	0	1
0	1	1		1	1	1	1

B	C	D		ZF	SF	OF	
1	0	0	**ble**	0	0	0	0
1	0	0	SF!= OF	0	0	1	1
1	0	0	or	0	1	0	1
1	0	0	ZF = 1	0	1	1	0
1	0	0		1	0	0	1
1	0	0		1	0	1	1
1	0	0		1	1	0	1
1	0	0		1	1	1	1

1	0	1	**bgt**	0	0	0	1
1	0	1	SF = OF	0	0	1	0
1	0	1	and	0	1	0	0
1	0	1	ZF = 0	0	1	1	1
1	0	1		1	0	0	0
1	0	1		1	0	1	0
1	0	1		1	1	0	0
1	0	1		1	1	1	0

B	C	D		ZF	SF	OF	
1	1	0	**blt**	0	0	0	0
1	1	0	SF!= OF	0	0	1	1
1	1	0	and	0	1	0	1
1	1	0	ZF = 0	0	1	1	0
1	1	0		1	0	0	0
1	1	0		1	0	1	0
1	1	0		1	1	0	0
1	1	0		1	1	1	0

1	1	1	**load**	0	0	0	0
1	1	1	No jump	0	0	1	0
1	1	1		0	1	0	0
1	1	1		0	1	1	0
1	1	1		1	0	0	0
1	1	1		1	0	1	0
1	1	1		1	1	0	0
1	1	1		1	1	1	0

Voici les équations résultantes de cette table de vérité :

Saut = ¬ B.¬ C.¬ D
+ ¬ B.¬ C.D.¬ **ZF** + B.¬ C.D.¬ **ZF**

+ ┐B.C.D(**ZF** + ┐ **SF**.┐ **OF** + **SF.OF**)
+ B.┐ C.┐ D(**ZF** + **SF**.┐ **OF** + ┐ **SF**.**OF**)
+ B.┐ C.D.┐ **ZF**(┐ **SF**.┐ **OF** + **SF**.**OF**)
+ B.C.┐ D.┐ **ZF**(**SF**.┐ **OF** + ┐ **SF**.**OF**)

Ainsi dans notre module de contrôle de saut, nous supposons que si :

- l'opCode n'est pas « 0000 »
- c'est un branchement « 1XXX » mais les flags ne valident pas la condition du branchement
-

Alors **nextPCselect** aura pour valeur « 10 » soit : instruction suivante. Sinon, si c'est « 0000 », il aura la valeur « 11 » pour indiquer un « Halt ». Dans le dernier cas, c'est un branchement.

Le module de contrôle de saut est placé en annexe.

4) Réalisation du banc de registre

Notre banc de registre contient 16 registres de 8 bits qui sont appelés lors des différentes instructions du processeur. Un interrupteur permet de les réinitialiser, et le bit regWrite permet d'autoriser l'écriture dans le registre souhaité. Afin de simplifier le schéma logique, nous avons eu recours à des tunnels.

Le banc de registre est en annexe

5) Réalisation du sélecteur de registres

Le sélecteur de registre prend 3 champs de quatre bits en entrée, et en sortie indique quels registres correspondants on souhaite utiliser dans le banc de registre , selon la valeur de isJump .

Format\entrée	Bits : 11-8	Bits 7-4	Bits 3-0
F1	rd	rs	rt
F2	rd	Immédiat	
F3	rs	rt	offset
F1 not	rd	rs	/
F3 b	Offset (seul les 4 derniers bits importent)		

6) Implémentation du circuit complet

Nous avons enfin lié entre-eux les différents modules implémentés. Après quelques corrections, nous avons finalement un processeur fonctionnel. Le circuit complet est placé en annexe

B- Utilisation du Nono-1

1) Programmes

Voici le résultat du code C donné dans le sujet. Après implémentation, nous avons un résultat correspondant aux attentes.

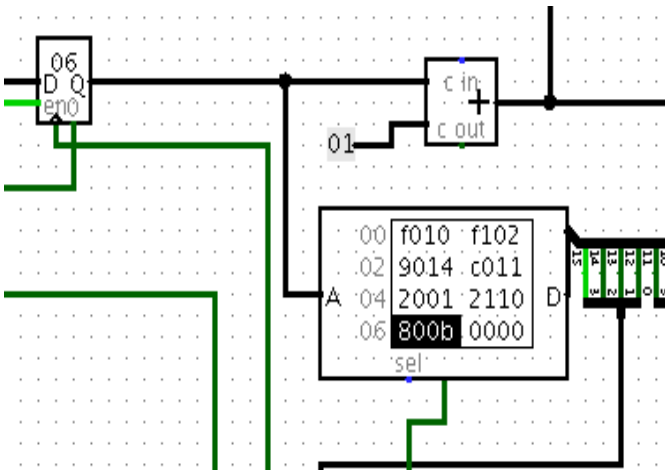
Code C :	Code assembleur :	Binary :	Hexadécimal :

int i, int j ;
while (i!= j) {
if (i > j) {
i - = j ;
} else {
j - = i ;
}
}
	li $0**, i***
	li $1, **j***
	beq $0, $1, 4
	ble $0, $1, 1
	sub $0, $0, $1
	sub $1, $1, $0
	b -5
		1111 0000 0000 **0000***
		1111 0001 0000 **0000***
		1001 0000 0001 0100
		1100 0000 0001 0001
		0010 0000 0000 0001
		0010 0001 0001 0000
		1000 0000 0000 1100
			f00**0***
			f10**0***
			9014
			c011
			2001
			2110
			800b

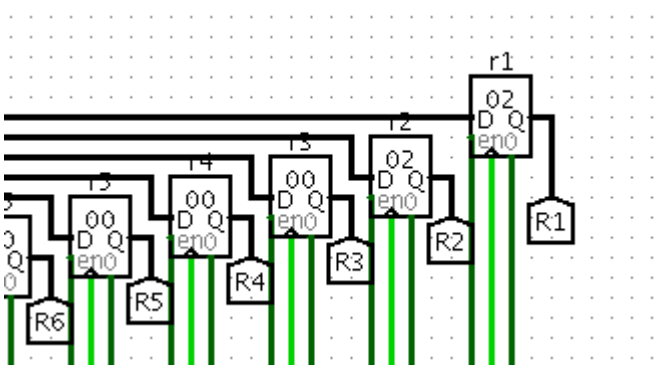
	* :i, j a choisir	* : Nombre a choisir	* : Nombre a définir

Exemple avec i = 16, j = 2
PGCD = 2

Instructions :



Valeurs des registres $0 et $1 :



Voici ensuite un code C que nous avons conçu, fonctionnant avec notre processeur

Code C	Code Assembleur	Binaire	Hexadécimal

int i = 16 ;
int j = 32 ;
int a = 4 ;
int k = j / 4 ;
int l = k * 4 ;
int m = i and j ;
int n = i or j ;
while (j > i) {
  i = i*2 ;
}
if (a ≤ l) {
  m = l;
} else {
  m = a;
}
	li $0, 16
	li $1, 32
	li $2, 2
	shr $3, $1, $2
	shl $4, $3, $2
	and $5, $1, $0
	or $6, $1, $0
	ble $1, $0, 2
	  shl $0, $0, 1
	  b -3
	bgt $2, $4, 1
	  add $5, $4, 0
	  add $5, $2, 0
		1111 0000 0001 0000
		1111 0001 0010 0000
		1111 0010 0000 0100
		0111 0011 0001 0010
		0110 0100 0011 0010
		0100 0101 0001 0000
		0011 0110 0001 0000
		1100 0001 0000 0010
		0110 0000 0000 0001
		1000 0000 0000 1101
		1101 0010 0100 0001
		0001 0101 0100 0000
		0001 0100 0010 0000
			F010
			f120
			f204
			7312
			6432
			4510
			3610
			c102
			6001
			800d
			d241
			1540
			1420

Le code fonctionne correctement, même s'il n'est pas très poussé.

2) Nono-2

Si l'on veut ajouter des fonctions, il faut :

- Sauvegarder l'adresse de retour de la fonction : **avoir un registre $ra**, on peut réserver un de nos registres déjà existant pour $ra, le dernier par exemple.

Comment le réserver au retour de fonction ?
Par convention avoir **deux sauts absolus** pour le retour de fonction et pour l'appel :

- Une instruction jal, qui saute à l'adresse de la fonction dans le code et qui enregistre dans le registre $ra, la ligne qui suit l'appel de cette fonction
- Une instruction jr, qui saute à l'adresse sauvegardée dans un registres

- L'ajout des sauts absolus implique un **branchement supplémentaire sur le multiplexeur** dirigé par le contrôle de saut, ça tombe bien, il en manquait un. On y passe directement le numéro de la ligne où l'on veut sauter qui est contenu dans le registre $ra pour jr ou bien dans un immédiat pour jal.

- L'ajout des deux instructions implique un **bit supplémentaire sur l'opcode** puisqu'il est déjà rempli. Et donc l'ajout d'un bit sur les instructions en général. Il faut ensuite modifier le décodeur d'instructions pour qu'il puisse reconnaître les deux nouvelles instructions. Il faut également modifier le contrôleur de saut pour qu'il puisse indiquer à quel moment il faut faire un saut absolu.

-il faut aussi ajouter un circuit qui choisit un immédiat ou le contenu de $ra selon le type de saut absolu (jal ou jr).

Par contre pour les fonctions imbriqué nous allons avoir un problème : le registre $ra sera écrasé. Il faudrait ajouter une pile d'adresses de retour. Mais nous ne voyons pas bien comment faire, nous allons donc nous contenter de fonctions non imbriquées pour l'instant.
Nous avons fait une demi-implémentation du Nono-2 disponible dans l'archive déposée sur Madoc. Ne souhaitant pas encombrer l'archive avec des captures d'écrans supplémentaires, nous ne l'avons pas ajouté ici.

3) Conclusion

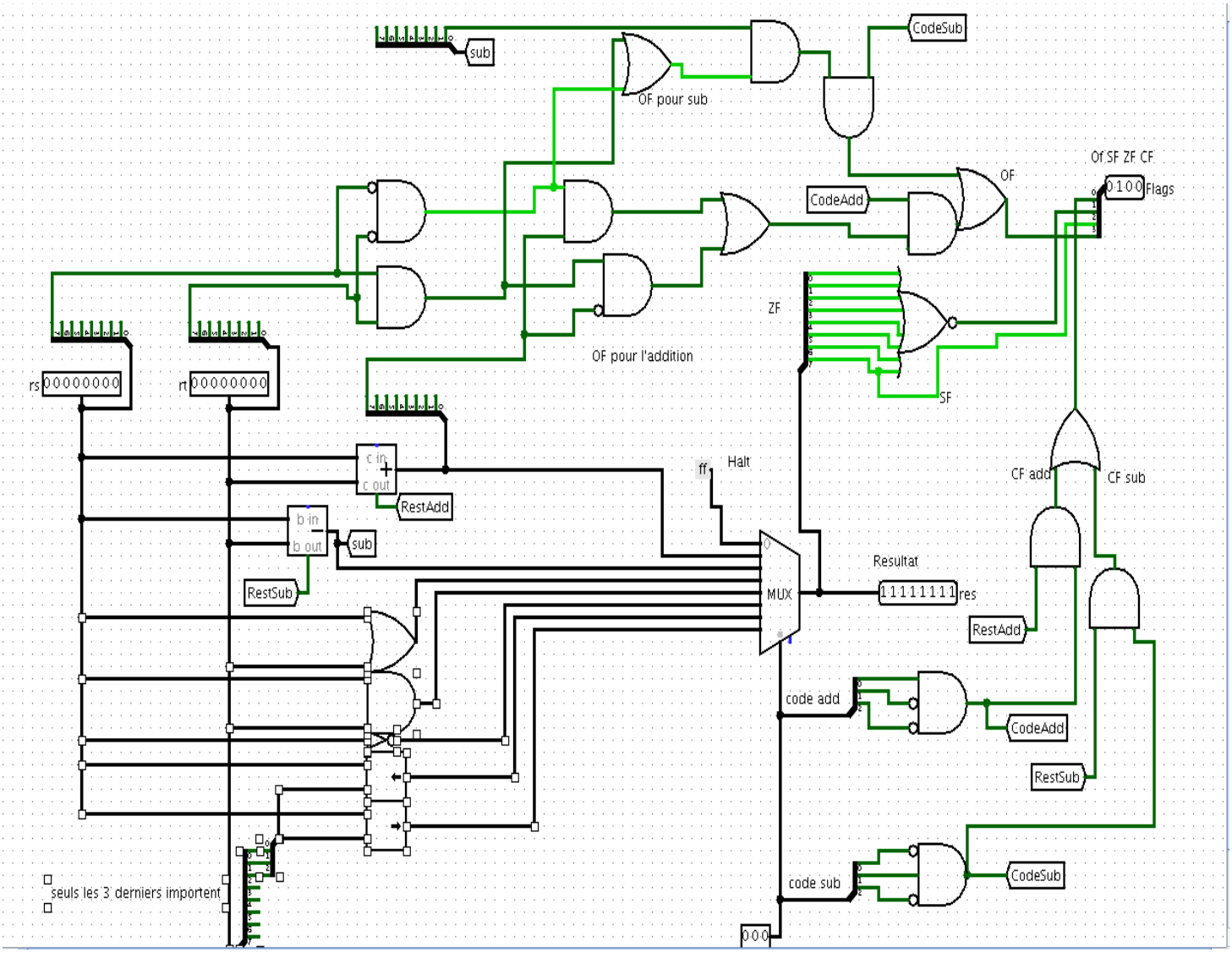
Pour conclure ce projet, nous avons apprécié bâtir ce processeur et avons vraiment trouvé ça très instructif. Monter de toute pièce cette partie centrale de l'ordinateur nous a permis de comprendre avec plus de facilité son fonctionnement, et notamment comment l'ordinateur interprète le code assembleur pour exécuter les programmes.

Cependant ce projet fût tout de même relativement long à faire, les erreurs étant difficiles à déceler.

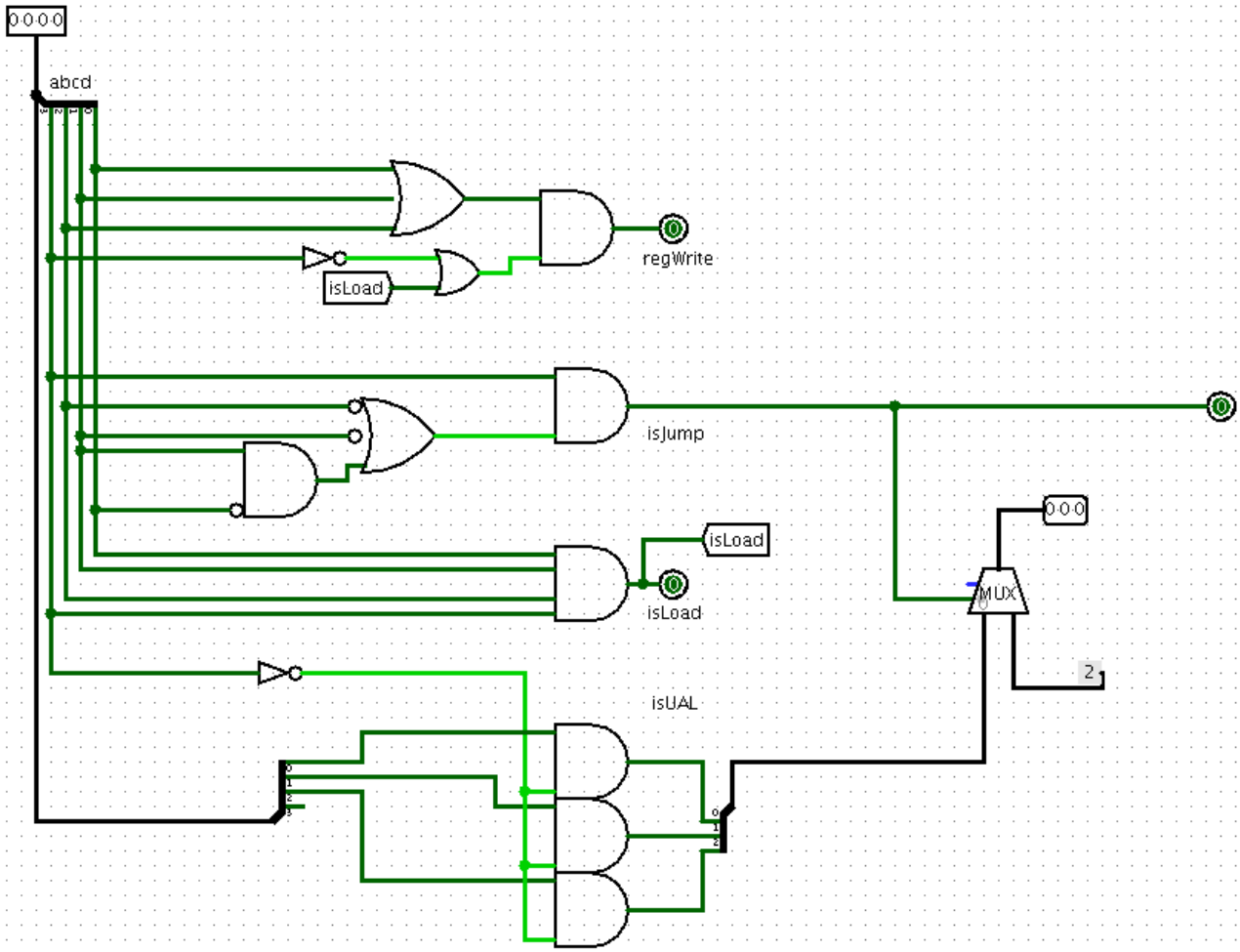
Afin d'aller plus loin, nous aurions aussi pu écrire un bref compilateur dans un langage de haut niveau, qui aurait pu sans trop de difficultés générer le code hexadécimal à ajouter ensuite à notre processeur.

Annexe

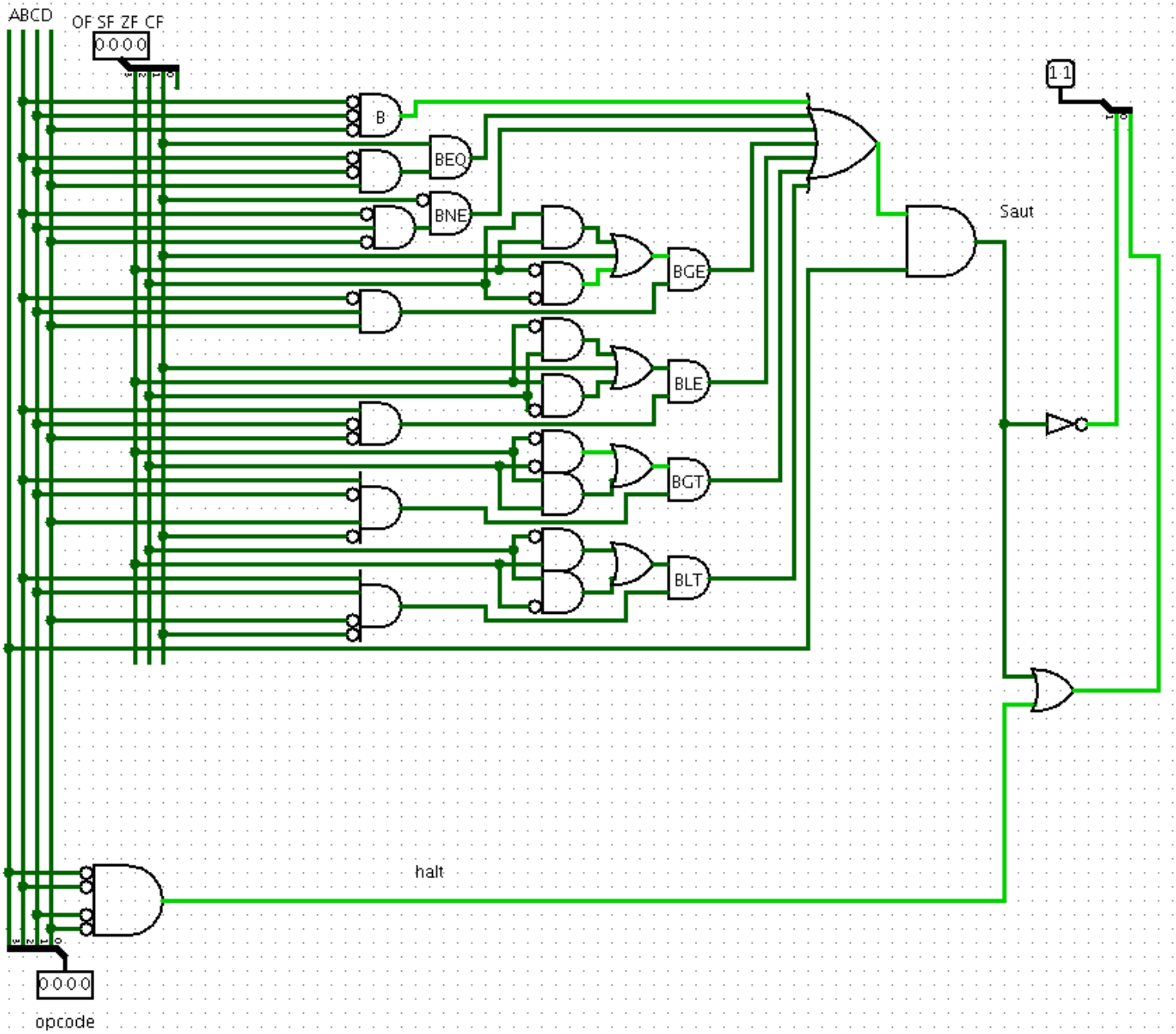
1) UAL


sub
OF pour sub
CodeSub
Of SF ZF CF
OF
CodeAdd
0100
Flags
ZF
OF pour l'addition
SF
rs
00000000
rt
00000000
c in
c out
RestAdd
b in
b out
sub
RestSub
ff
Halt
CF add
CF sub
Resultat
MUX
11111111
res
RestAdd
code add
CodeAdd
RestSub
code sub
CodeSub
000
seuls les 3 derniers importent

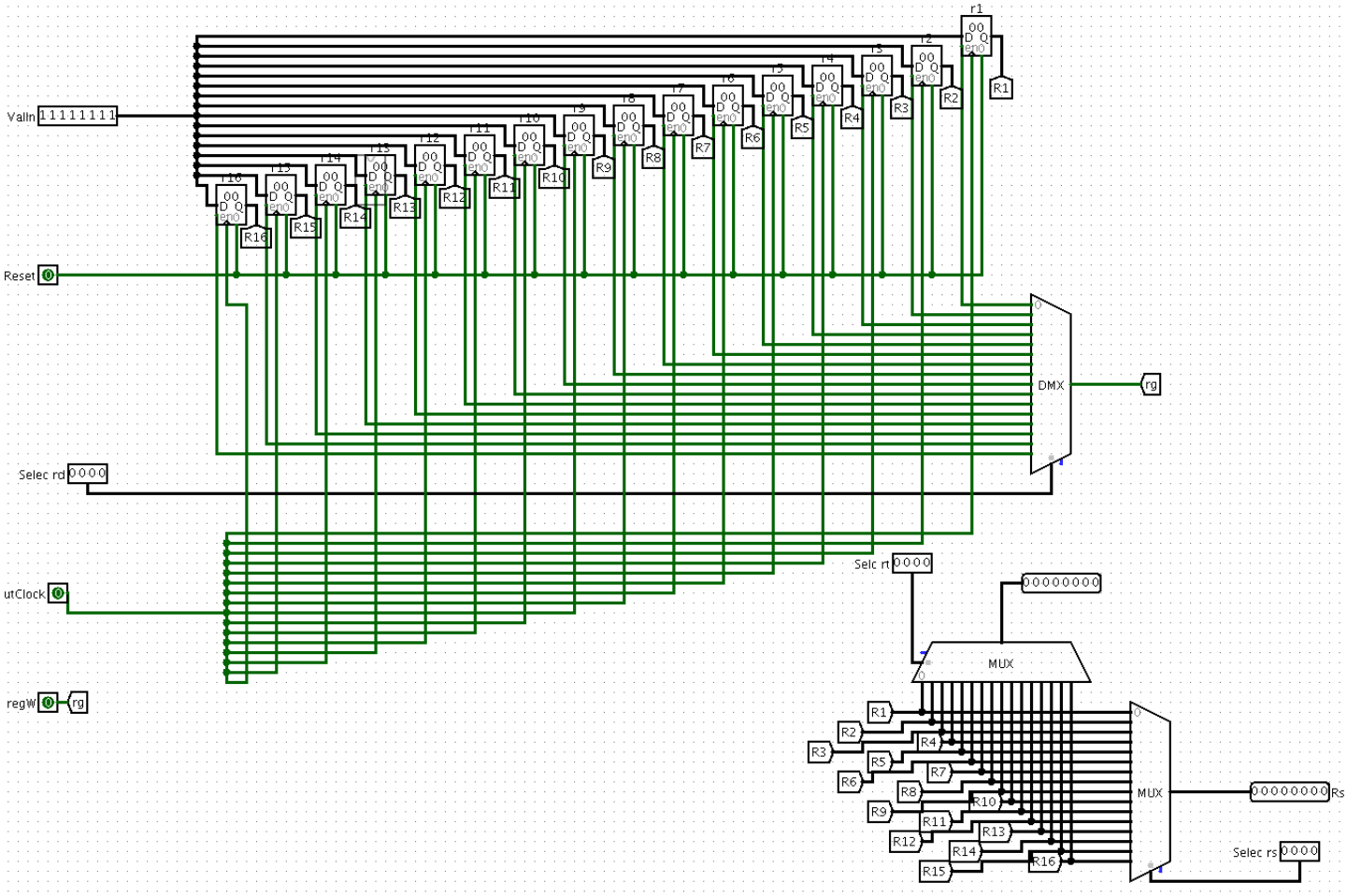
2) Décodeur d'instructions



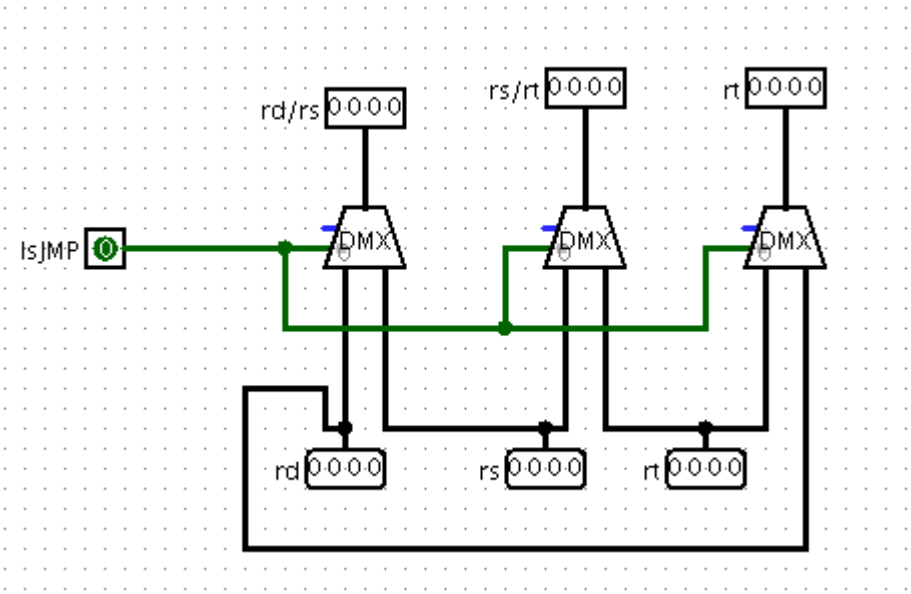
3) Contrôle de saut



4) Banc de registres



5) Sélecteur de registres



6) Circuit complet