RAPPORT DE PROJET : ARCHITECTURES DES MICROPROCESSEURS.

REALISE PAR :

MARYAM AIT DAOUD

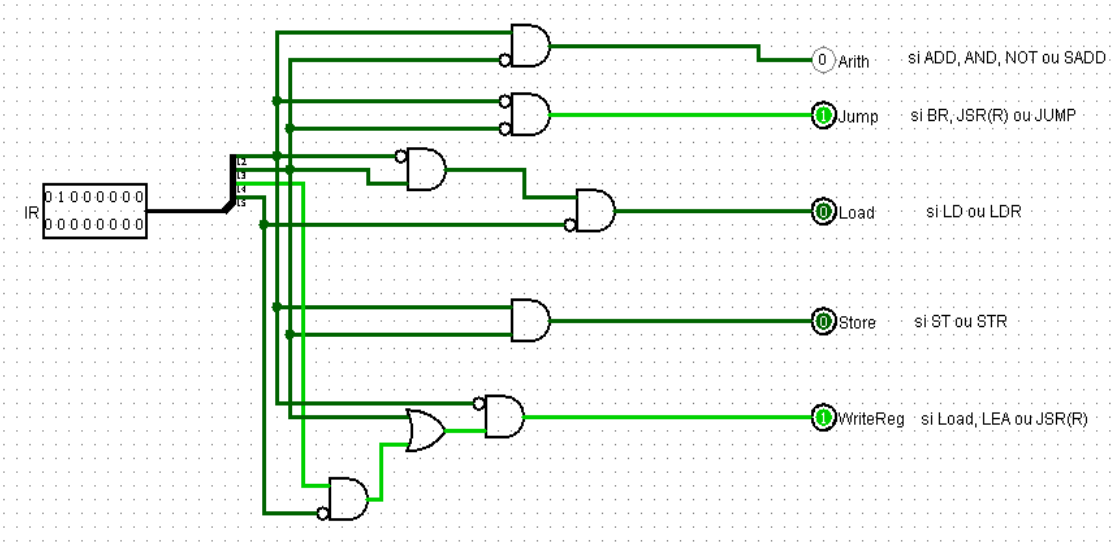
et

Arnold NGNOUBA NDIE TCHEGOU

Instructions LC-3 à implémenter

DecodeIR

Pour câbler les instructions Arth, Jump, Load, Store et WriteReg, nous avons utilisé la table de Karnaugh regroupant toutes les instructions du LC-3. Nous avons également utilisé les bits 14 et 15 afin de différencier les instructions LD et LDR (qui permettent de charger le contenu de la mémoire dans un registre) de l'instruction LEA (qui permet de charger une adresse dans un registre). Nous avons également pris en compte le fait que l'instruction JSR(R) écrit l'adresse de retour d'une fonction dans le registre R7.



ALU

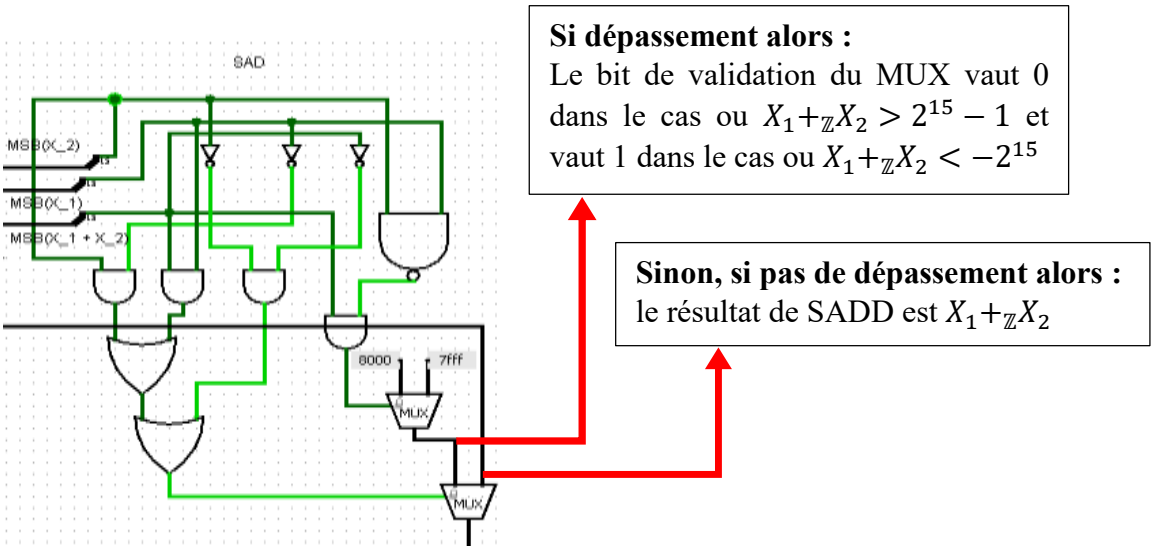
Dans le circuit ALU, nous avons ajouté le câblage de l'instruction **SADD** (addition saturée). Le tableau de vérité suivant présente notre démarche pour la réalisation de **SADD** :

$\mathbf{MSB\_X_1}$	$\mathbf{MSB\_X_2}$	$\mathbf{MSB\_X_1 +_{\mathbb{Z}} X_2}$	$\mathbf{X_1 +_{\mathbb{Z}} X_2}$	$\mathbf{2^{15} - 1}$	$\mathbf{-2^{15}}$
0	0	0	1	0	0
0	0	1	0	1	0
0	1	0	1	0	0
0	1	1	1	0	0
1	0	0	1	0	0
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0

Si X_1 et X_2 sont négatifs (bit de poids fort égale à 1) et la somme $\boldsymbol{X_1 +_{\mathbb{Z}} X_2}$ est positif (bit de poids fort égale à 0) alors il y'a un dépassement. Dans ce cas le résultat de l'addition saturée est -2^{15}.

Si X_1 et X_2 sont positifs (bit de poids fort égale à 0) et la somme $\boldsymbol{X_1 +_{\mathbb{Z}} X_2}$est négatif (bit de poids fort égale à 1) alors il y'a un dépassement. Dans ce cas le résultat de l'addition saturée est $2^{15} - 1$.

- Pour pouvoir réaliser le câblage de SADD nous avons utilisé deux multiplexeurs 2 vers 1 en cascades.

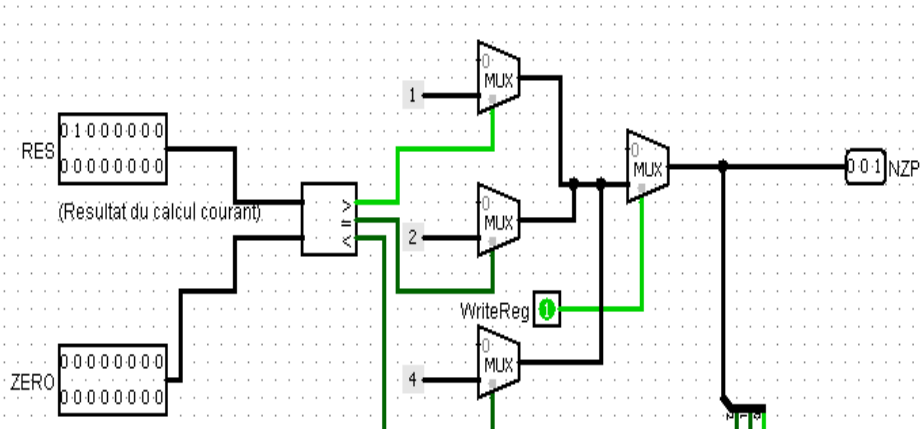


Si dépassement alors :
Le bit de validation du MUX vaut 0 dans le cas ou $X_1 +_{\mathbb{Z}} X_2 > 2^{15} - 1$ et vaut 1 dans le cas ou $X_1 +_{\mathbb{Z}} X_2 < -2^{15}$

Sinon, si pas de dépassement alors :
le résultat de SADD est $X_1 +_{\mathbb{Z}} X_2$

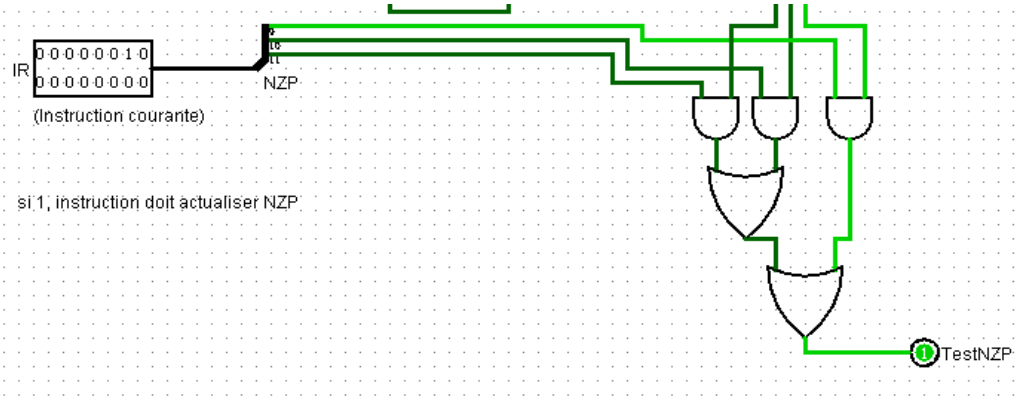
NZP

- Pour réaliser le câblage de la valeur actuelle du registre d'état NZP, nous avons choisi de mettre en parallèle trois MUX 2 vers 1. Les entrées de validation de chacun des trois MUX correspondent chacune à une des trois sorties du comparateur (qui valent 1 si la sortie est sélectionnée).



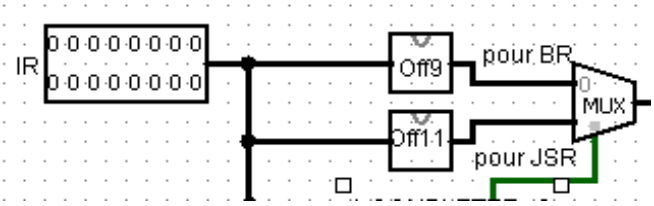
- Les trois bits de numéros 11 à 9 déterminent respectivement si l'indicateur N, Z et P est pris en compte dans la condition. La condition $TestNZP$ est donnée par la formule suivante :

$$TestNZP = (b_{11} \wedge N) \vee (b_{10} \wedge Z) \vee (b_9 \wedge P)$$

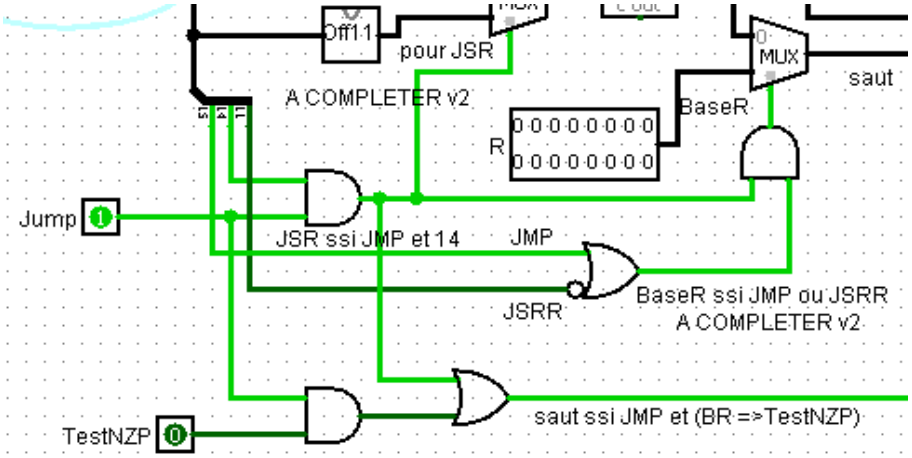


RegPC

- Tout d'abord nous avons rajouté l'offset 11 nécessaire pour l'instruction JSR.



- Le bit 11 permet de distinguer l'instruction JSRR ($b_{11} = 0$) de l'instruction JSR ($b_{11} = 1$). Le bit 15 permet également de distinguer l'instruction JMP de JSRR et JSR. D'après la réalisation du circuit, si $b_{14} \wedge jump = 1$ alors on fait un saut soit par JMP, JSRR ou JSR. Nous utilisons donc un MUX 2 vers 1 pour choisi soit le registre R ou offset11 selon le cas :



WriteVal

- Nous avons utilisé un MUX 4 vers 1 pour la réalisation de JSR, LEA et Load. Le tableau de vérité suivant présente notre démarche.

S_1	S_0	ALURes	MemOUT	PC+SEXT (offset9)	PC
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

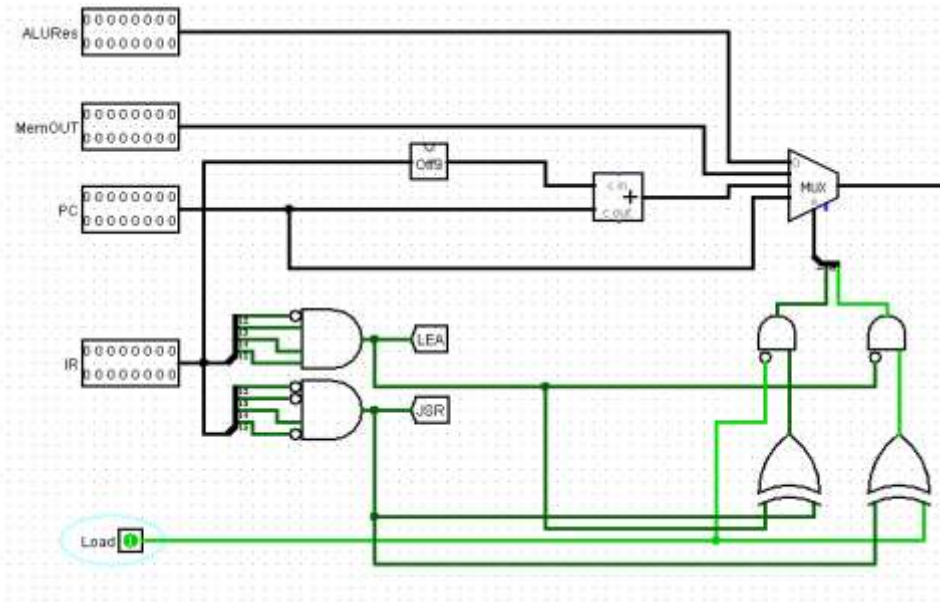
JSR

LEA

$Load$

$\overline{Load}.\overline{LEA}.\overline{JSR}$

$$S_0 = \overline{LEA} \wedge (Load \oplus JSR)$$
$$S_1 = \overline{Load} \wedge (LEA \oplus JSR)$$



Programmes .mem

- **Code hexadécimal de la routine strlen**
  v2.0 raw
  001d e21c 4801 0e19 2dfb 1dbe 7581 7380 907f 1021 6440 0402 1261 0ffc 1001 6380 6581 1da2
  c1c0 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0048 0065 006c 006c 006f 0000

Programmation en LC-3 étendu

Pour l'implémentation des programmes assembleurs, nous avons suivi une approche méthodique. Tout d'abord, nous avons écrit les fonctions en langage C, puis nous les avons traduites en langage assembleur. Cette démarche a été appliquée aux cinq premières fonctions de manipulation de chaînes de caractères. Une fois cette étape initiale franchie, nous avons acquis suffisamment d'autonomie pour passer à la programmation directe en assembleur.

Pour la réalisation de la version avancée, qui impliquait l'écriture de chaque fonction comme une sous-routine ne modifiant pas les registres, à l'exception de celui contenant le résultat, nous avons introduit l'utilisation d'une pile. De manière spécifique, nous avons opté pour une approche ***callee-save***, où la sous-routine est responsable de la sauvegarde et de la restauration du contenu des registres.