LEGE Hugo
LEVY Lucas

Projet – Circuits et Architecture

1.1 Instructions LC-3 à implémenter

1.1.1 Instruction XOR

Dans cette partie, la porte XOR est autorisée, nous l'avons utilisée pour modifier l'ALU. Nous avons remplacé la porte NOT par la porte XOR. Il suffit de prendre une constante remplie de 1 et faire le XOR avec l'input, ceci nous affichera le NOT. Donc on n'a pas besoin de changer l'opcode.

1.1.2 Instruction pour le poids de Hamming

Dans cette partie, nous avons créé un circuit POPCNT, dans lequel nous avons utilisé un splitter pour séparer les bits de l'entrée de 16bit pour pouvoir les additionner parallèlement. Ceci nous fait 15 portes binaires(8 portes à 1 bit, 4 portes à 2 bit, 2 portes à 3 bit et 1 porte à 4 bit). Nous augmentons les bits de chaque additionneur FA car nous prenons en compte les retenues. Le résultat maximum du poids de hamming pour une entrée à 16 bit se tiens sur 5 bit donc nous avons utilisé une constante vide pour pouvoir combler et éviter l'erreur.

LEA

L'instruction LEA se comporte de la sorte : nous allons écrire dans le premier paramètre, donc dans un registre, l'adresse mémoire du deuxième paramètre, qui peut être un label, un immédiat ou une adresse.

Pour écrire dans un registre, il faut WriteReg = 1, et cela écrit la valeur dans RegIn.

Pour WriteReg, on part de decodeIR, on sélectionne correctement l'opcode de l'instruction LEA, c'est-à-dire 1110, branché pour activer WriteReg avec une combinaison de AND, NOT et XOR.

Pour RegIn, le branchement devait respecter la condition donnée. Ainsi en branchant avec un multiplexeur, IR étant en 1110, on met en sortie PC+SEXT(Offset9), simplement avec un additionneur. Par ailleurs, il a fallu faire attention au fait que LEA prend l'adresse PC+1 (PC après le fetch) , ce qui nous provoquait des erreurs dans les tests.

Nous avons choisi de ne pas prendre GetAddr car cela utilise beaucoup plus de portes logiques (un additionneur, deux multiplexeurs) ainsi que des entrées pas utilisés (baseR par exemple), ceci-dit nous n'aurions pas eu l'erreur du +1, car il l'implémente déjà dedans.

Finalement pour tester cela nous avons mis dans un fichier « test.mem » les instructions en hexadécimal suivantes, E001 (LEA R0, #1), F025 (HALT) et 0001 (label). La sortie est donc l'adresse de « 0001 » dans R0 c'est-à-dire x0002.

NZP

Nous prenons les bit 9,10,11 en argument du nzp. Nous faisons attendre les arguments dans un verrou qui laisse passer si le writeReg est à 1. Puis nous comparons le résultat actuel (RES) avec les valeurs de nzp pour que la condition soit valide (testNZP =1) nous avons 3 conditions : N, le bit de point fort soit égal à 1. Z tous les bits doivent être nul et P c'est non(Z) et non(N).

1.2 PROGRAMMATION LC-3 ÉTENDU

1.2.1 Manipulation de chaînes

La fonction index a pour but de nous renvoyer la première apparition d'un caractère, pour cela nous avons un algorithme brut force qui compare caractère par caractère par une soustraction, les caractères représentés en hexa, si on fait la soustraction de deux caractères identiques alors leur différence est nulle. On loop tant que les flags n et p sont activés, lorsqu'on a 0 on saute sur un label qui met dans R0 l'adresse voulue.

1.2.2 Renversement de bit

La fonction « reverse » a pour but de renversé une suite de bits donnée. Intuitivement nous nous sommes représentés les int en tant que tableau de 0 et de 1, ce qui nous a induit en erreur. En effet, la difficulté de l'exercice, selon nous, était d'interpréter les entiers passés en paramètres en tant que suite de bits. Ainsi un nouveau système de pensé c'est ouvert, l'introduction de masques pour résoudre le problème.

Voici l'idée, on va utiliser deux masques et utiliser une boucle. Le premier masque m1, va servir de comparateur avec l'instruction AND dans le but de déterminer si le bit est 0 ou 1. Initialisé à 1 (=x0001 en hexa), le multiplié par deux permet de faire un décalage à gauche et passer au bit suivant.

Dans le cas où on a 0, nous n'allons sauter aux paramètres de la boucle. Dans le cas où on a un 1, le deuxième masque m2 entre en jeu. Il s'agit d'un tableau avec les 16 valeurs possibles des puissances de 2. Il est décroissant dans le but de faciliter la lecture des indices dans les boucles.

Ainsi on ajoute les puissances de deux dans la variable résultat, comme le tableau est décroissant on a inversé l'ordre.

1.2.3 Bit Scan

Pour implémenter la sous-routines ctz efficacement nous devions utiliser XOR et POPCNT et ainsi faire l'opération proposé. POPCNT n'étant pas défini dans le simulateur LC-3, pour tester nous avons décidé de faire des sous-routines équivalentes. Donc pour XOR nous sommes partis de la forme algébrique : XOR = a*not(B)+not(a)*b. Pour POPCNT, l'idée étant de compter les uns, nous nous sommes inspirés des codes précédant et l'utilisation d'un masque pour comparer bit à bit.

Pour implémenter dans le circuit Logisim, il faut implémenter POPCNT et XOR avec les opcode et arguments associés.

Clz a pour objectif de compter les zéros jusqu'au premier bit égal à un. Pour cela l'idée, inspiré des codes précédant était d'introduire un masque m décroissant (c'est-à-dire de x8000, puis x4000 etc ...) , puis de faire une comparaison AND entre le masque et notre nombre bit par bit. La condition pour s'arrêter est de trouver un bit de valeur 1, qui résulte avec le AND d'un 1, qui fait sauter sur le HALT et termine le programme. Sinon on incrémente un compteur et on passe au bit suivant.

1.2.4 Produit Saturé

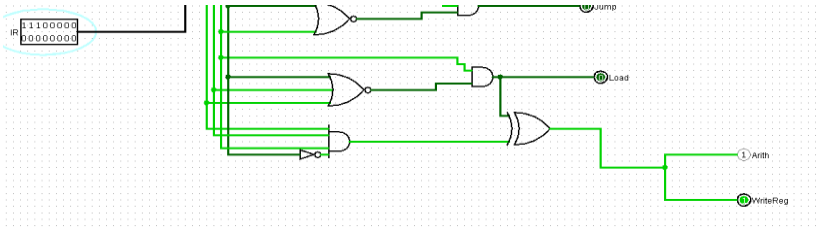
Notre produit saturé fonctionne en deux parties. Dans un premier temps il teste si le produit des entrées ne fait pas de dépassement . Pour cela nous avons utilisé la fonction clz utilisé précédemment. En effet, en partant d'exemples rapides nous avons remarqué que si la somme des premiers zéros des arguments était inférieure au nombre de bit alors le produit est possible, c'est-à-dire clz(a) + clz(b) < 16. Nous avons de plus traiter les autres cas tel que R0 renvoie 2^16 -1 comme nous l'indiquait l'énoncé. Nous l'avons cependant modifié rapidement de sorte que R6 soit utilisé comme paramètre.

Ensuite, pour faire une multiplication, nous avons calqué la méthode comme un calcul à la main, c'est-à-dire on fait la somme des produits entre a et 1 en faisant attention aux décalages. Par exemple a=0010 et b = 0110, nous faisons 0000 + 0100.+ 10.. + 0... les points étant les décalages par colonnes. Pour cela une nouvelle fois la méthode des masques avec le AND, la différence cette fois était qu'il fallait multiplié par deux pour faire un shift a gauche pour les décalages et avoir le bon résultat.

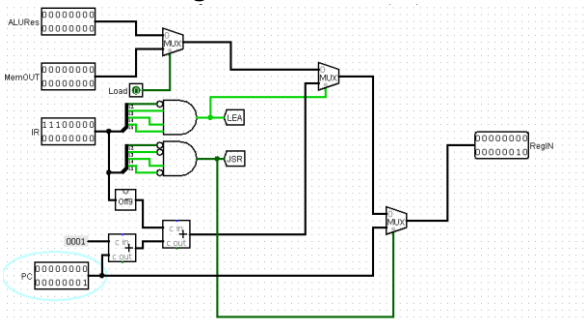
Annexe

LEA :

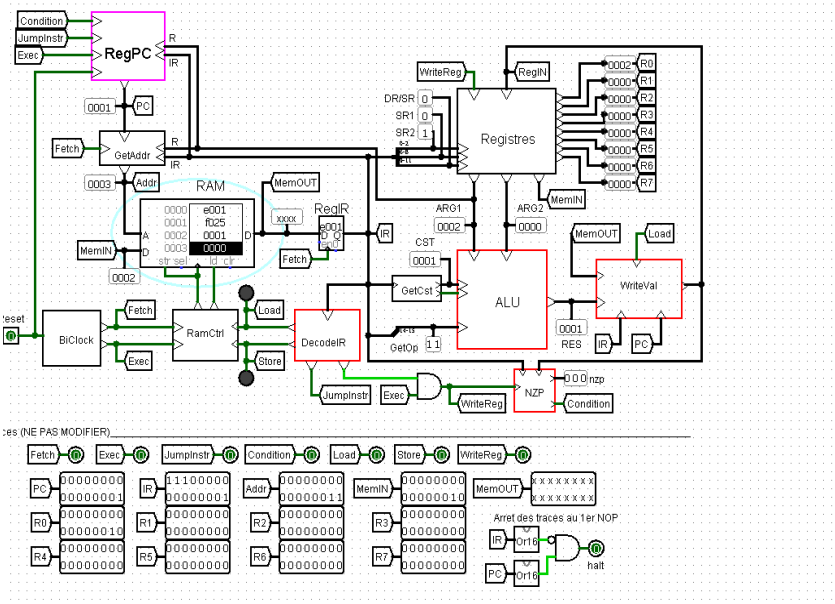
Activer le WriteReg



Valeur dans RegIn :



Test :



Valeur à regarder : R0 et instructions hexa dans la RAM