Rapport Projet- Circuits et architecture

Salim Kaimoussi – Yassemine Alaoui

1.1.1 Instruction XOR

En algèbre de Boole, faire un XOR avec une valeur dont tous les bits sont à 1 équivaut à faire un NOT (inversion de bits). $A \oplus 1 = \neg A$

$$A \oplus 0 = A$$

Donc, si on fait **A XOR (-1)**, où -1 en binaire (complément à 2 sur 16 bits) est **1111...1111**, on obtient l'inverse de A.

L' instruction XOR en mode immédiat (bit 5 = 1) se lit ainsi : 1001 | DR | SR1 | 1 | Imm5

L'instruction NOT standard du LC-3 fixe les 6 derniers bits à 1. Dans notre nouvelle architecture, cela correspond exactement à l'encodage de l'instruction XOR en mode immédiat avec la valeur -1 (Imm5 = 11111). Comme effectuer un XOR avec -1 (tous les bits à 1) revient à inverser tous les bits, l'opération résultante est mathématiquement identique à un NOT.

1.1.2 Instruction pour le poids de Hamming

Analyse et Choix Architectural :

L'instruction POPCNT (Opcode 1101) doit calculer la somme des 16 bits du registre source (SR). Contrairement aux opérations logiques bit-à-bit, cette opération nécessite une interaction entre tous les bits.

Pour respecter les contraintes de performance (faible profondeur) et d'interdiction du composant "Bit Adder", nous avons opté pour une architecture en arbre de réduction (Arbre d'additionneurs). Cette structure offre une complexité temporelle logarithmique (O(log N)) par rapport à une approche en cascade linéaire.

Le calcul s'effectue en 4 étapes successives purement combinatoires :

1. Niveau 1 : Les 16 bits d'entrée sont séparés et additionnés par paires à l'aide de demi-additionneurs (ou additionneurs simples). Cela produit 8 valeurs de 2 bits.
2. Niveau 2 : Ces 8 valeurs sont additionnées deux à deux pour former 4 valeurs de 3 bits.
3. Niveau 3 : Les résultats sont sommés pour obtenir 2 valeurs de 4 bits.
4. Niveau 4 : Une dernière addition produit le résultat final sur 5 bits (pouvant aller de 0 à 16).

Le résultat de 5 bits est ensuite étendu à gauche par des zéros (Zero-Extension) pour former le mot de 16 bits renvoyé par l'ALU.

1.2.1 Manipulation de chaînes

Analyse et Implémentation de index.asm

Spécification : La fonction localise la première occurrence du caractère stocké dans R2 au sein de la chaîne pointée par R1. L'adresse de l'occurrence est retournée dans R0. Si le caractère est absent, R0 retourne 0.

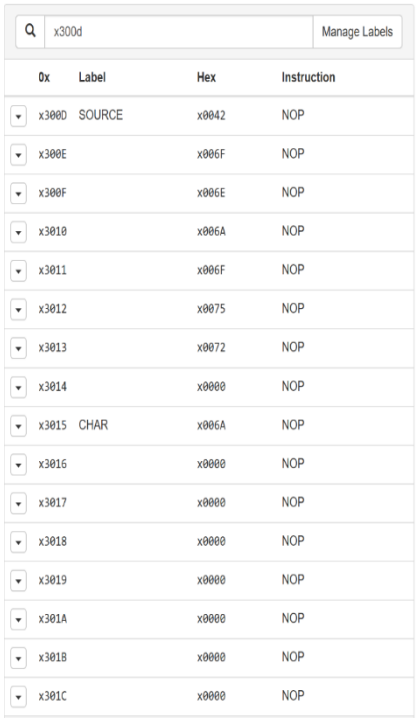
1. **Initialisation :** Le registre de résultat R0 est initialisé à 0 par défaut (AND R0, R0, #0). Cela permet de gérer implicitement le cas d'échec sans branchement complexe en fin de programme.

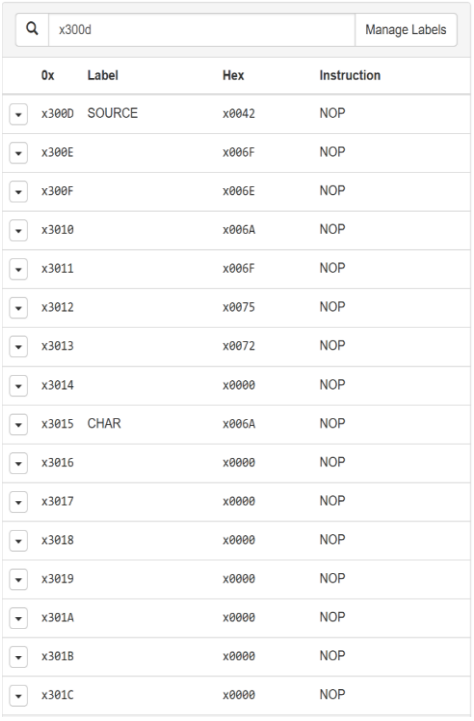
2. **Pré-traitement :** Pour optimiser la boucle de recherche, nous calculons l'opposé du caractère cible en dehors de la boucle. En effectuant le complément à deux de R2 (soit -R2), nous transformons l'opération de comparaison A == B (qui nécessiterait une soustraction à chaque itération) en une addition A + (-B) == 0.
3. **Boucle de parcours :**
    - Chargement du caractère courant pointé par R1 dans R3.
    - **Condition d'arrêt 1 (Fin de chaîne) :** Si R3 est nul (x0000), la chaîne est terminée. Le programme s'arrête (via HALT) et R0 contient toujours 0 (non trouvé).
    - **Comparaison :** On additionne R3 et -R2. Si le résultat est nul (BRz), le caractère est trouvé.
    - **Incrémentation :** Si non trouvé, on incrémente R1 pour avancer dans la mémoire et on boucle.
4. **Succès :** Si le caractère est trouvé, l'adresse courante contenue dans R1 est transférée dans R0 avant de terminer le programme.

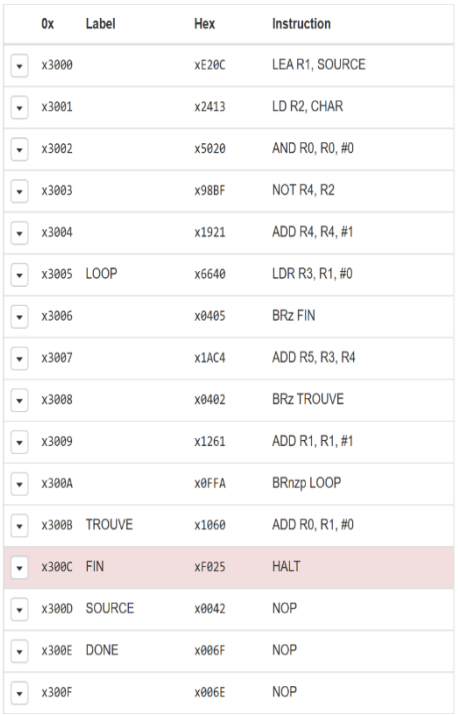
2.3. Validation et Tests

Le programme a été validé avec la chaîne de test "Bonjour" et deux scénarios distincts :

- **Test de succès (Recherche de 'j') :**
    - *Entrée :* R1 pointe sur "Bonjour", R2 contient x006A ('j').
    - *Résultat observé :* Le simulateur s'arrête et le registre R0 contient l'adresse x3010 (adresse mémoire effective de la lettre 'j' dans notre zone de données). Cela confirme que la boucle s'est interrompue au bon moment.
- **Test d'échec (Recherche de 'z') :**
    - *Entrée :* R1 pointe sur "Bonjour", R2 contient x007A ('z').
    - *Résultat observé :* En plaçant un point d'arrêt sur l'instruction HALT, nous avons vérifié que R0 valait bien x0000. Le programme a parcouru toute la chaîne jusqu'au caractère nul sans déclencher la condition de succès.







Analyse et Implémentation de strcpy.asm

Objectif et Spécifications

La fonction strcpy (String Copy) doit copier l'intégralité d'une chaîne source (pointée par R1) vers une zone destination (pointée par R2), y compris le caractère nul de terminaison.

Algorithme et Démarche

La difficulté principale de strcpy réside dans la gestion de l'ordre des opérations pour garantir la copie du caractère nul final.

Boucle de copie :

- Lecture du caractère source (LDR R3, R1, #0).
- **Écriture immédiate** (STR R3, R2, #0) : Nous écrivons le caractère dans la destination *avant* de vérifier sa valeur. C'est un point crucial : si nous testions la fin de chaîne avant l'écriture, le \0 final ne serait pas copié, laissant la chaîne destination potentiellement non terminée (risque de lecture mémoire invalide par la suite).
- **Test de fin :** Après l'écriture, nous vérifions si R3 est nul (BRz). Si oui, la copie est terminée et on sort de la boucle.
- **Mise à jour des pointeurs :** Si la chaîne continue, R1 et R2 sont incrémentés simultanément.

3.3. Validation et Tests

Nous avons défini une zone mémoire réservée (.BLKW #20) pour la destination.

- **Jeu de données :** Chaîne source "CopieMoi".
- **Vérification mémoire :** Après exécution, l'inspection de la mémoire à l'adresse de destination montre une réplique exacte des valeurs ASCII de la source (x43, x6F, x70...), suivie impérativement de la valeur x0000.

x300A | Manage Labels

0x	Label	Hex	Instruction
x300A		x006F	NOP
x300B	TROUVE	x0070	NOP
x300C		x0069	NOP
x300D		x0065	NOP
x300E	DONE	x004D	NOP
x300F		x006F	NOP
x3010		x0069	NOP
x3011		x0000	NOP
x3012	DEST	x0043	NOP
x3013		x006F	NOP
x3014		x0070	NOP
x3015	CHAR	x0069	NOP
x3016		x0065	NOP
x3017		x004D	NOP
x3018		x006F	NOP

x0043 | Manage Labels

0x	Label	Hex	Instruction
x0043		xFD00	.FILL xFD00
x0044		xFD00	.FILL xFD00
x0045		xFD00	.FILL xFD00
x0046		xFD00	.FILL xFD00
x0047		xFD00	.FILL xFD00
x0048		xFD00	.FILL xFD00
x0049		xFD00	.FILL xFD00
x004A		xFD00	.FILL xFD00
x004B		xFD00	.FILL xFD00
x004C		xFD00	.FILL xFD00
x004D		xFD00	.FILL xFD00
x004E		xFD00	.FILL xFD00
x004F		xFD00	.FILL xFD00
x0050		xFD00	.FILL xFD00
x0051		xFD00	.FILL xFD00

Analyse et Implémentation de strncpy.asm

Objectif et Spécifications

La fonction strncpy est une version sécurisée de la copie de chaîne. Elle prend un argument supplémentaire dans R0 : le nombre maximum de caractères à copier. La copie s'arrête si le caractère nul est rencontré OU si le compteur R0 atteint zéro.

Algorithme et Démarche

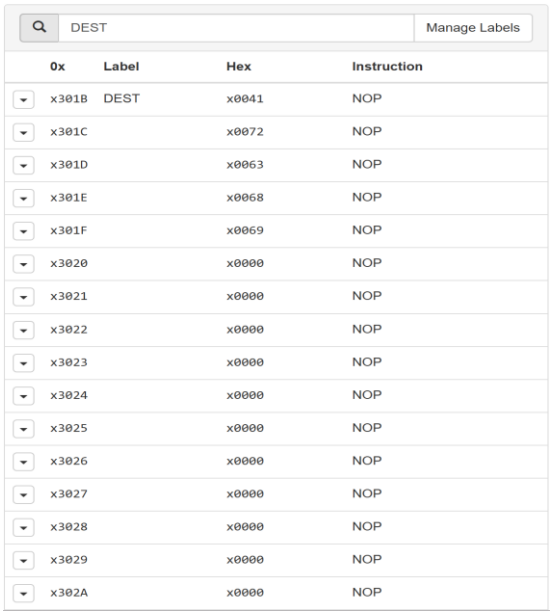
Cet algorithme nécessite une double condition de sortie à chaque itération :

1. **Vérification du quota (R0) :**
    - Au début de chaque tour de boucle, on teste si R0 <= 0 (ADD R0, R0, #0 suivi de BRnz). Si le quota est atteint, on arrête immédiatement, même si la chaîne n'est pas finie.
2. **Copie et vérification du contenu :**
    - Si le quota n'est pas atteint, on charge (LDR) et on copie (STR) le caractère courant.
    - On teste ensuite si ce caractère est le caractère nul (BRz). Si oui, on arrête prématurément (comportement standard simplifié pour ce projet).
3. **Mise à jour :**
    - On incrémente les pointeurs source et destination.
    - On **décrémente** le compteur R0 (ADD R0, R0, #-1) pour comptabiliser la copie effectuée.

Validation et Tests

Pour valider la robustesse de cette fonction, nous avons testé le cas de la "troncature" :

- **Jeu de données :** Source "Architecture" (longueur 12), mais avec une demande de copie limitée à R0 = 5.
- **Résultat observé :**
    - Dans la zone destination, seuls les caractères "Archi" sont présents.
    - L'adresse mémoire suivant le 'i' final ne contient pas le reste du mot ("tecture").
    - Cela prouve que la condition sur le compteur R0 a pris le pas sur la fin de chaîne, validant ainsi la logique de strncpy.



1.2.4 Produit saturé

Prédiction de débordement

Sous-routine CLZ (Count Leading Zeros)

Algorithme : Balayage par Décalage Contrairement à CTZ qui bénéficie d'une astuce arithmétique (utilisant POPCNT), la fonction CLZ a été implémentée par une approche logicielle itérative. L'algorithme fonctionne ainsi :

1. On initialise un compteur à 0.
2. On boucle 16 fois maximum.
3. À chaque itération, on teste le bit de poids fort (MSB) du registre :
    - Si le nombre est négatif (MSB = 1), on arrête : le premier "1" est trouvé.
    - Si le nombre est positif ou nul (MSB = 0), on incrémente le compteur de zéros et on effectue un décalage à gauche logique (ADD R1, R1, R1).

Le cœur de la boucle repose sur l'instruction de branchement BRn (Branch on Negative) qui détecte le bit de signe (MSB).

2. Sous-routine PREDICT_OVF (Prédiction de Débordement)

Théorie Mathématique : Pour qu'un produit A x B tienne sur 16 bits sans débordement, la somme des bits significatifs des deux opérandes doit être inférieure ou égale à 16. Si $N_{bits}(x)$ est le nombre de bits utiles d'un entier, on sait que $N_{bits}(x) \approx 16 - CLZ(x)$. La condition de non-débordement est :

$$(16 - \text{CLZ(A)}) + (16 - \text{CLZ(B)}) \leq 16$$

$$32 - (\text{CLZ(A)} + \text{CLZ(B)}) \leq 16$$

$$\text{CLZ(A)} + \text{CLZ(B)} \geq 16$$

Implémentation : Notre fonction calcule la valeur critique V = 15 - (CLZ(A) + CLZ(B)). Le résultat dans **R0** suit la logique suivante :

- **Si CLZ(A) + CLZ(B) ≥ 16**(Total de zéros > 15) : V est négatif. **Pas de débordement.**
- **Si CLZ(A) + CLZ(B) = 15 :** V vaut 0. **Risque limite.**
- **Si CLZ(A) + CLZ(B) ⟨ 15 :** V est positif. Débordement certain.

Produit saturé 16 bits

1.2.4 Produit Saturé (Multiplication non signée)

Objectif : Cette sous-routine réalise la multiplication de deux entiers non signés (16 bits) stockés dans R1 et R2. Le résultat est stocké dans R0. La particularité est la gestion de la saturation : si le résultat mathématique excède la capacité du registre ($> 2^{16} - 1$), la valeur renvoyée est plafonnée à 0xFFFF au lieu de subir un débordement (wrap-around).

Algorithme : Shift-and-Add (Décalage et Addition)

nous utilisons l'algorithme classique de multiplication binaire ("Shift-and-Add").

Le principe est d'itérer 16 fois (pour chaque bit du registre) :

1. On décale le résultat courant vers la gauche (multiplication par 2).
2. On inspecte le bit de poids fort du multiplicateur (R1).
3. Si ce bit est à 1, on additionne le multiplicande (R2) au résultat courant.

Gestion de la Saturation (Prédictive) :

Le défi principal est de détecter le débordement avant qu'il ne corrompe le résultat. Deux contrôles sont effectués à chaque itération :

1. **Lors du décalage :** Si le résultat courant (R0) a déjà son bit de poids fort à 1, un décalage à gauche provoquera une perte d'information. On sature immédiatement.

- **Lors de l'addition :** Avant d'ajouter R2 à R0, nous vérifions s'il y a assez de "place" restante. La place disponible est approximée par **NOT R0 (0xFFFF - R0).** Si R2 > NOT R0, alors l'addition provoquera un débordement.

Analyse du Circuit de Décodage d'Instruction (DecodeIR)

Rôle et Fonctionnement Général Le module DecodeIR agit comme le "cerveau analytique" de l'unité de contrôle. Sa fonction est de traduire le code binaire de l'instruction en signaux de commande électriques compréhensibles par le reste du matériel.

Comme le montre le schéma, son fonctionnement se décompose en trois étapes :

- **Extraction de l'Opcode :** Un *splitter* isole les bits 12 à 15 du registre d'instruction (IR), qui définissent la nature de l'opération (ADD, AND, LDR, etc.).
- **Décodage 4-vers-16 :** Ces 4 bits entrent dans un décodeur qui active une ligne unique parmi 16. Chaque ligne correspond donc à un opcode précis (ex: la ligne 1 pour ADD, la ligne 9 pour NOT/XOR).
- **Génération des Signaux (Regroupement) :** C'est ici que réside l'intelligence du circuit. Des portes logiques OU regroupent les différentes instructions qui partagent les mêmes besoins matériels.

Analyse du Circuit de Décodage d'Instruction (NZP)

- **Détermination des Flags (Analyse du Résultat) :** Le circuit scrute le bus de données RES. Le flag **N** est directement tiré du bit de signe (MSB), **Z** est activé si tous les bits sont nuls, et **P** est déduit logiquement (si ni N, ni Z).
- **B. Mémorisation (Registre d'État) :** Ces trois flags sont stockés dans un registre dédié. Ce dernier ne se met à jour que lorsque le signal de contrôle **WriteReg** est actif, figeant ainsi l'état du processeur après une opération d'écriture (ADD, AND, etc.) jusqu'à la suivante.
- **C. Décision de Branchement (Comparaison) :** Le circuit compare les flags stockés avec les conditions demandées par l'instruction de saut (bits **11, 10, 9** de l'IR).
    - Des portes **ET** vérifient la correspondance entre la condition voulue et l'état réel.
    - Une porte **OU** finale génère le signal **TestNZP**. Si ce signal vaut 1, la condition est validée et le saut est autorisé.

Analyse du Circuit de Gestion du Compteur Ordinal (RegPC)

Rôle et Fonctionnement : Le module RegPC est le "chef d'orchestre" du flux d'exécution. Il détermine à chaque cycle d'horloge l'adresse mémoire de la prochaine instruction. Son architecture repose sur une cascade de multiplexeurs qui sélectionnent la nouvelle valeur du PC parmi trois sources possibles :

1. **Exécution Séquentielle (PC + 1) :** C'est le mode par défaut. La sortie du PC traverse un incrémenteur. Si aucune instruction de saut n'est active, le processeur passe simplement à la ligne de code suivante.
2. **Saut Relatif (PC + Offset)** : Utilisé pour les branchements (BR) et les appels de sous-routines (JSR). Le circuit additionne un décalage (Offset de 9 ou 11 bits) à la valeur du PC incrémenté pour atteindre une étiquette cible.
3. **Saut Absolu (BaseR) :.** Utilisé pour les instructions JMP ou JSRR. L'adresse n'est pas calculée mais provient directement d'un registre général (ex: R7), permettant des sauts dynamiques.

Analyse du Circuit de Sélection de Donnée (WriteVal)

Rôle Fonctionnel : Le circuit WriteVal agit comme un aiguillage final. Sa mission est de sélectionner une seule source de données parmi quatre possibles (ALU, Mémoire, Adresse calculée, ou PC) pour l'envoyer vers le registre de destination (RegIN).

Choix Standard (ALU vs Mémoire) : Le premier étage sélectionne soit le résultat du calcul (ALURes), soit la donnée lue en mémoire (MemOUT), piloté par le signal Load.

Priorité LEA (Adresse) : Si l'instruction est un LEA (détecté par les portes logiques sur l'IR), un second multiplexeur force le passage de l'adresse calculée (PC + Offset9), écrasant le choix précédent.

Priorité JSR (Retour Sous-routine) : En dernier recours, si l'instruction est un JSR, le multiplexeur final sélectionne directement la valeur du **PC**. Cela permet de sauvegarder l'adresse de retour dans R7.