POLYTECH LYON

IMPLÉMENTATION D'UN CIRCUIT MODÉLISANT UN PROCESSEUR

Abir HANNED, Ashley PADAYODI, Chafae QALLOUJ & Nawal EL KHAL

3e ANNÉE – SPÉCIALITÉ INFORMATIQUE

Responsable de l'UE : Théo Pierron

Pour le 22/12/2024

Composant *Registres* :

Ci-dessous (Figure 1), le circuit que nous avons effectué pour le composant *Registres* du processeur :

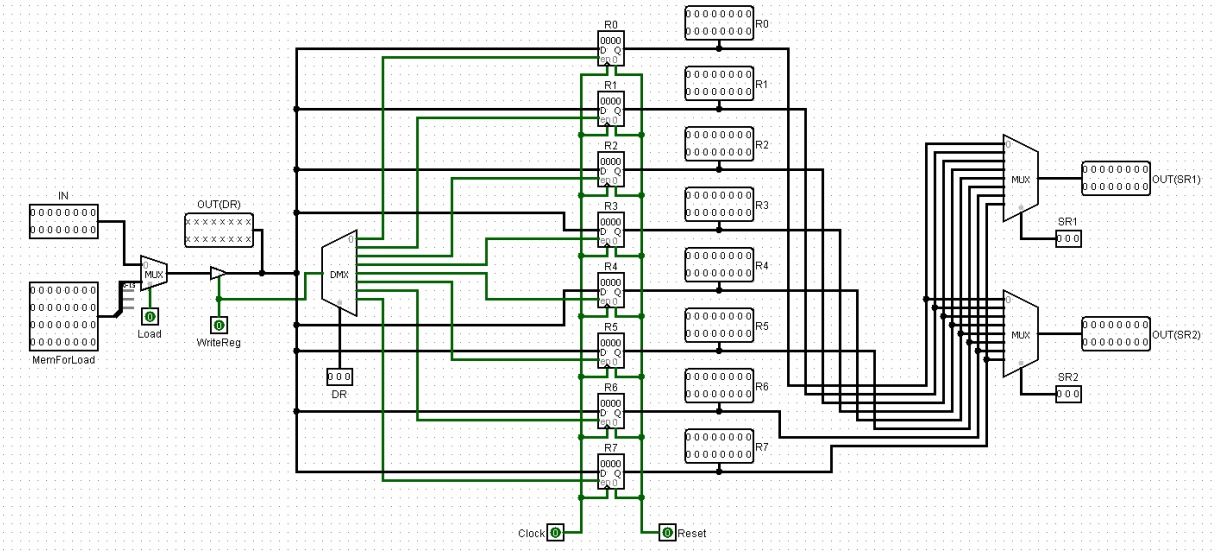

Figure 1 - Implémentation du composant Registres

Entrées :

- Write (1 bit) : active l'écriture dans les registres
- IN (16 bits) : l'information à stocker dans un registre lorsque *Write* est à 1
- SR1, SR2, DR (3 bits chacun) : les adresses des registres source et destination 1
- Load (1 bit)
- MemOUT (32 bits) : l'information à stocker (16 premiers bits de poids faible) dans un registre lorsqu'une instruction *Load* est exécutée
- Clock
- Reset

Sorties :

- Contenu des registres source (16 bits chacun)
- Contenu du registre destination (16 bits) : l'information à stocker dans la *RAM* lorsqu'une instruction *Store* est exécutée

Composant *ALU* :

Ci-dessous (Figure 2), le circuit que nous avons effectué pour le composant *ALU* du processeur :

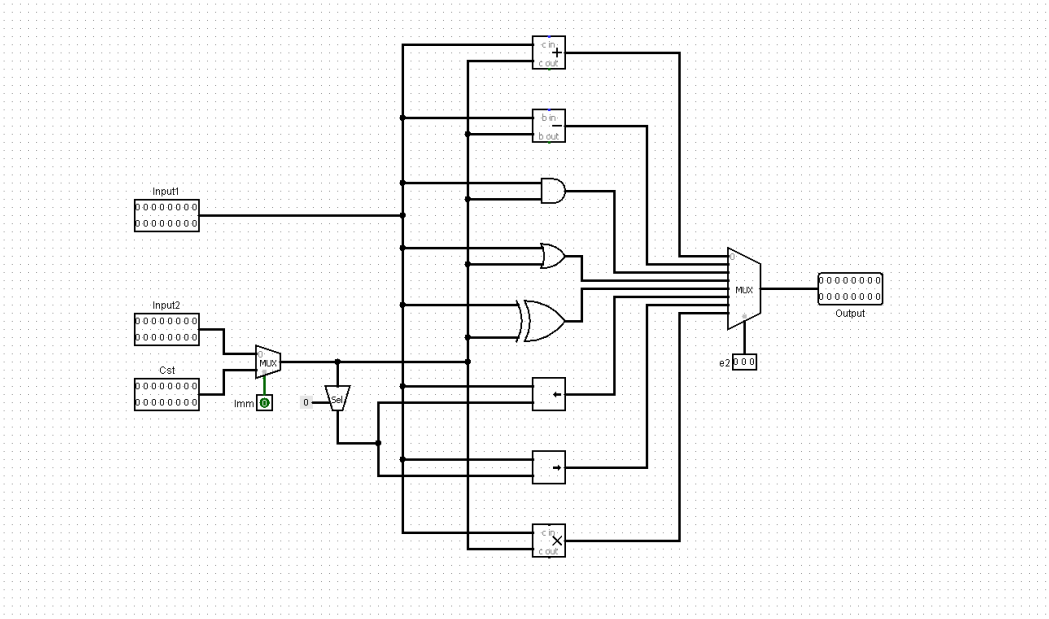

Figure 2 - Implémentation du composant ALU

Entrées :

- SR1, SR2 (3 bits chacun) : les adresses des registres contenant deux opérandes
- Cst (16 bits) : la valeur de la constante en cas d'opération immédiate
- Imm (1 bit) : permet de prendre pour deuxième opérande la constante
- GetOp (3 bits) : le code de chaque opération arithmétique

Sortie :

- Outpout (16 bits) : le résultat de l'opération effectuée dans l'*ALU*

Modification de GetCst :

Nous avons fait le choix de n'encoder les constantes des opérations immédiates que sur les 16 derniers bits d'IR (Figure 3) pour rester cohérent avec notre encodage des opérations arithmétiques (voir Annexe 1 : Choix des encodages).

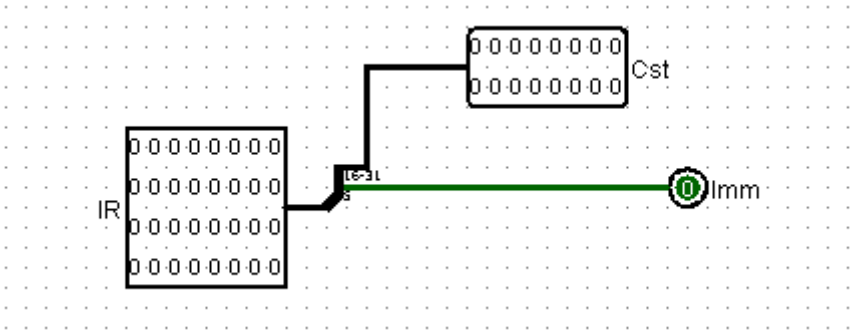

Figure 3 - Modification du composant GetCst

Composant *Décode IR* :

Ci-dessous (Figure 4), le circuit que nous avons effectué pour le composant *Decode IR* du processeur :

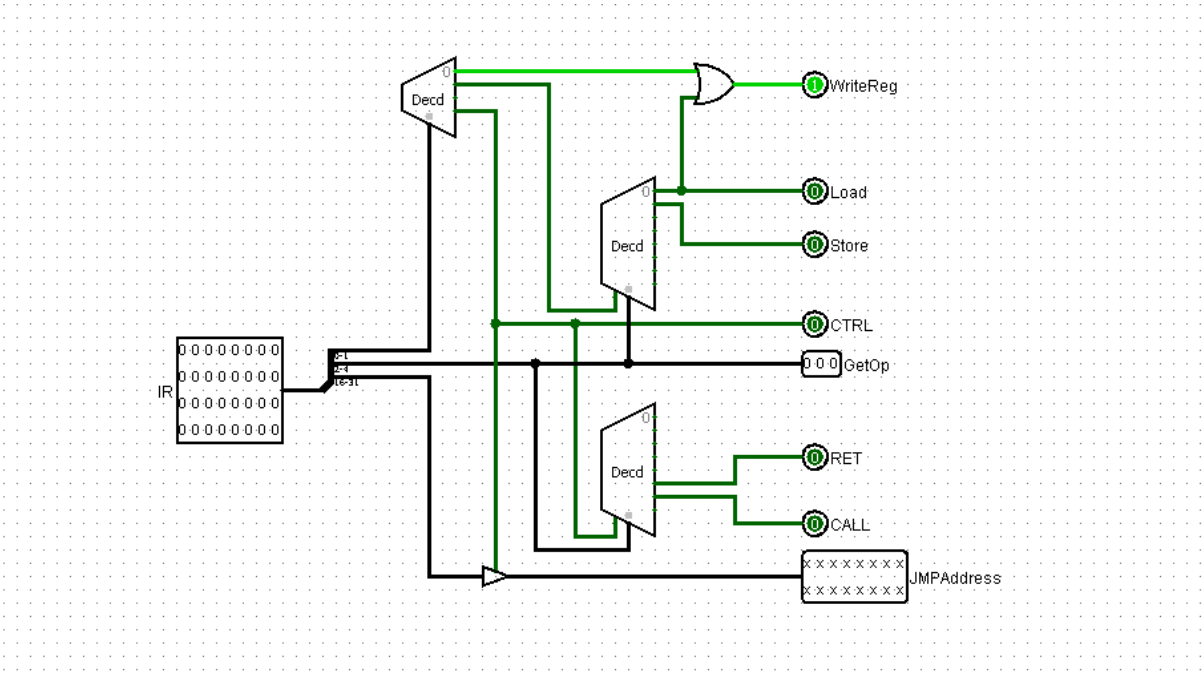

Figure 4 - Implémentation du composant Décode IR

Entrée :

- Contenu de RegIR (32 bits) : la conversion en binaire du code écrit en assembleur

Sorties :

- GetOp (3 bits) : le code de chaque opération, valeur récupérée aux bits 2 à 4 de l'instruction assembleur convertie en binaire (voir Annexe 1 : Choix des encodages)
- Load (1 bit) : s'allume lorsque dans *IR*, on détecte un appel d'opération de type *MÉMOIRE* (01) et d'*op code* '000' (voir Annexe 1 : Choix des encodages)
- Store (1 bit) : s'allume lorsque dans *IR*, on détecte un appel d'opération de type *MÉMOIRE* (01) et d'*op code* '001' (voir Annexe 1 : Choix des encodages)
- CTRL (1 bit) : s'allume lorsque dans *IR*, on détecte un appel d'opération de type *CONTRÔLE* (11) (voir Annexe 1 : Choix des encodages)
- RET (1 bit) : s'allume lorsque dans *IR*, on détecte un appel d'opération de type *CONTRÔLE* (11) et d'*op code* '101'(voir Annexe 1 : Choix des encodages)
- CALL (1 bit) : s'allume lorsque dans *IR*, on détecte un appel d'opération de type *CONTRÔLE* (11) et d'*op code* '110' (voir Annexe 1 : Choix des encodages)
- JMPAdress (16 bits) : l'adresse de saut lors des instructions de contrôle, valeur récupérée aux bits 16 à 31 de l'instruction assembleur convertie en binaire (voir Annexe 1 : Choix des encodages)

RegPc permet d'accéder à l'adresse de l'instruction souhaitée (dans la pile d'instruction de la mémoire).

GetAddr permet quant à lui d'avoir en sortie soit *PC* (l'adresse de l'instruction à executer) durant la phase *Fetch* et l'adresse du registre source pendant la phase *Exec*.

En parallèle, nous avons utilisé un programme Python pour traduire un code assembleur en un code binaire (Voir Annexe 2 : Utilisation de l'assembleur), compréhensible par un processeur virtuel, tel que celui utilisé dans Logisim. Le but de ce programme est de simuler l'exécution d'un programme écrit en assembleur en le convertissant dans un format binaire qui peut être chargé dans la mémoire du processeur et exécuté dans un simulateur.

Afin de gérer les opérations de contrôle, nous avons ajouté un sous-circuit *CondChecker* permettant de tester les conditions recherchées. Ci-après, l'explication de son fonctionnement :

Composant *CondChecker* :

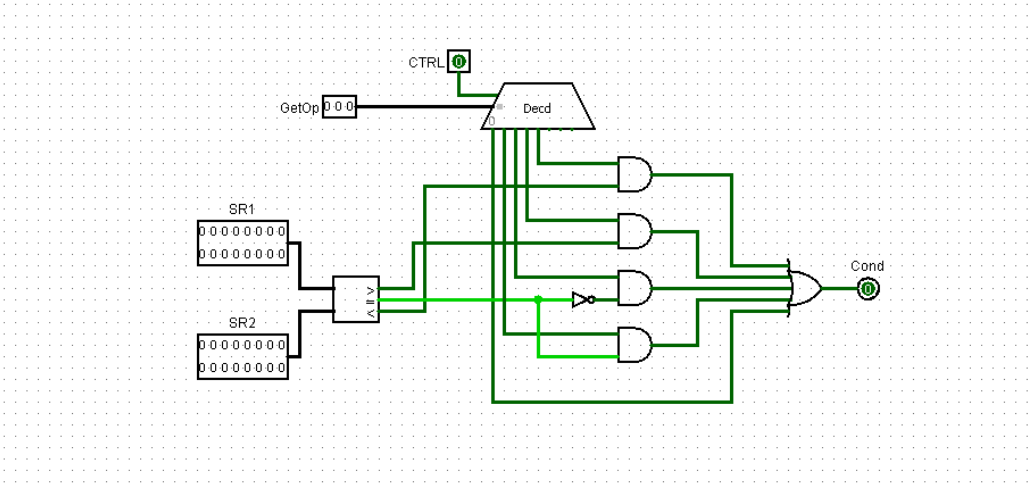

Figure 5 - Implémentation du composant CondChecker

Entrées :

- SR1 et SR2 (16 bits chacun) : valeurs comparées pour les opérations de contrôle
- GetOp (3bits) : le code de chaque opération de contrôle (voir Annexe 1 : Choix des encodages)
- CTRL (1 bit) : permet d'activer des comparaisons pour les opérations de contrôle

Sortie :

- Cond (1 bit) : est allumé lorsque les conditions sont vraies

Composant *GetAddr* :

Ci-dessous (Figure 6), le circuit que nous avons effectué pour le composant *GetAddr* du processeur :

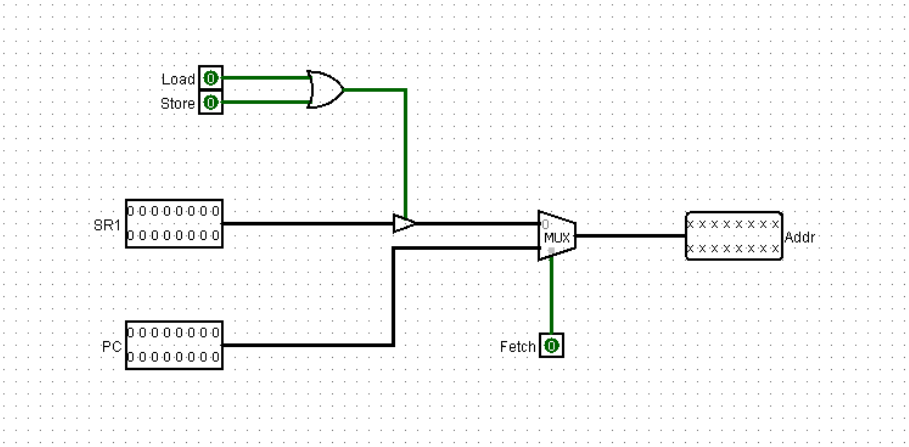

Figure 6 - Implémentation du composant GetAddr

Entrées :

- PC (16 bits) : contient les adresses mémoire de chaque instruction
- Fetch (1 bit) : active la phase *Fetch*, c'est-à-dire la phase de récupération d'une instruction en mémoire puis de son chargement dans le *registre IR*
- Load (1 bit)
- Store (1 bit)
- SR1 (16 bits) : contient l'adresse nécessaire pour les instructions de mémoire (voir Annexe 1 : Choix des encodages) :
  LOAD : DR ← MEM[SR1] et STORE : MEM[SR1] ← SR

Sortie :

- Adress (16 bits) : contient l'adresse nécessaire pour les opérations de contrôle losqu'elle sont appelées, sinon Address = PC

Composant *RegPC* :

Ci-dessous (Figure 7), le circuit que nous avons effectué pour le composant *RegPC* du processeur :

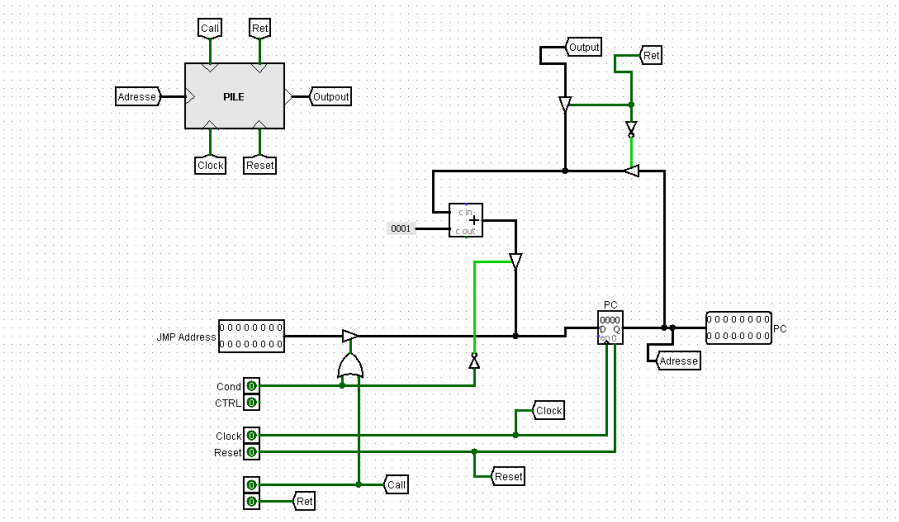

Figure 7 - Implémentation du composant RegPC

Entrées :

- Ret (1 bit) : Lorsqu'il est activé, la valeur en sortie de pile est récupérée dans *PC*, dans le cas contraire, *PC* fonctionne comme expliqué au point (*)
- JMPAdress (16 bits) : l'adresse de saut lors des instructions de contrôle
- Call (1 bit)
- Cond (1 bit)
- Exec (1 bit)
- Reset (1 bit)

Sortie :

- (*) PC (16 bits) : lorsque les conditions de comparaison sont vraies ou qu'un CALL est appelé, *PC* prend la valeur de *JMPAdress* puis, à chaque cycle *Exec*, est incrémenté de 1
- Adresse (16 bits) : prend la valeur de *PC* pour l'empiler dans la pile

Afin de gérer les instructions *CALL* et *RET*, nous avons créé un composant *Pile* contenant une mémoire sur laquelle on peut empiler ou dépiler des informations. Ci-après, l'explication de son fonctionnement :

Composant *Pile* :

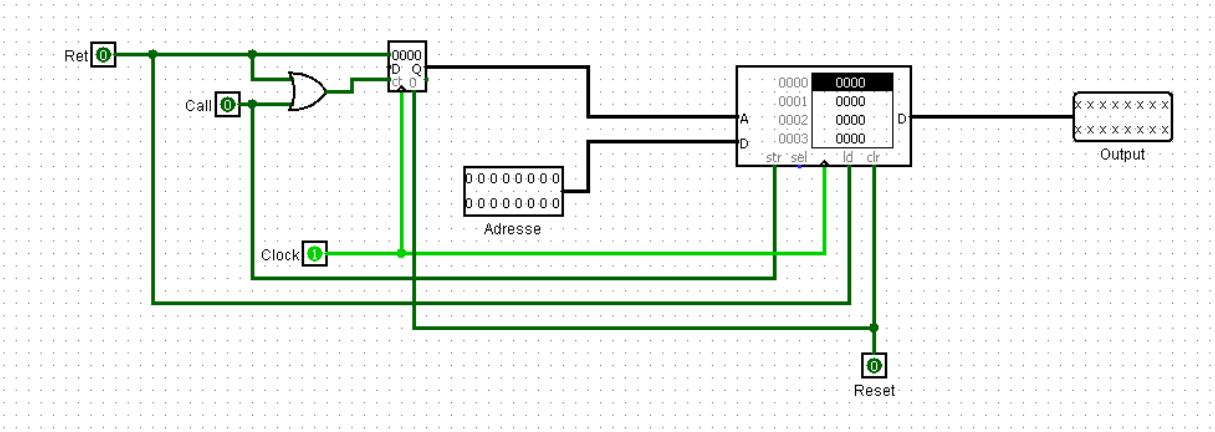

Figure 8 - Implémentation du composant Pile

Entrées :

- Ret (1 bit)
- Call (1 bit)
- Adresse (16 bits) : valeur à empiler dans la pile lorsque l'instruction *CALL* est appelée
- Clock (1bit)
- Reset (1 bit)

Sortie :

- Output (16 bits) : valeur dépilée la pile lorsque l'instruction *RET* est appelée

ANNEXE 1 : CHOIX DES ENCODAGES

Opérations arithmétiques :

UAL : 00

ADD : 000 – SUB : 001 – AND : 010 – OR : 011 – XOR : 100 – SL : 101 – SR : 110 – MUL : 11

NOT(IMM) : 0 DR SR1 SR2 Non utilisés → 0

Opérations arithmétiques immédiates :

UAL : 00

ADD : 000 – SUB : 001 – AND : 010 – OR : 011 – XOR : 100 – SL : 101 – SR : 110 – MUL : 11

IMM : 1 DR SR1 Non utilisés → 0 Cst

Opérations d'accès à la mémoire :

MEM : 01 LD: 000 – STR : 001 Non utilisé → 0 DR/SR SR1 Non utilisés → 0

Opérations de contrôle :

- JUMP et CALL

CTRL : 11 JMP: 000 – CALL : 110 Non utilisé → 0 Adresse

- JEQU, JNEQ, JSUP et JINF

CTRL : 11 JEQU : 001 – JNEQ : 010 – JSUP : 011 – JINF : 100

Non utilisé → 0 DR/SR SR1 Non utilisés → 0

- RET

CTRL : 11 RET : 101 Non utilisé → 0

ANNEXE 2 : UTILISATION DE L'ASSEMBLEUR

Nom du fichier : Assembleur.py

Ce programme prend en entrée un fichier appelé assembly.txt contenant un code assembleur simplifié et le convertit en format binaire. Le code binaire généré est ensuite converti en hexadécimal et écrit dans un fichier de sortie nommé binaryLogisim.txt, qui peut être utilisé par Logisim pour simuler l'exécution d'un circuit de processeur.

Règle d'écriture du langage assembleur :

- Pour insérer un commentaire, le précéder du caractère « ; »

Une fois le programme exécuté, il sera affiché un message confirmant le succès de son exécution. Il sera ainsi possible de retrouver dans son répertoire le fichier de sortie binaryLogisim.txt.

Pour charger ces valeurs dans la RAM du processeur virtuel, il suffit de faire un clique droit dessus, puis sélectionner « Load Image » comme montré ci-contre et ouvrir binaryLogisim.txt :