

# Exploration ALM

## Architecture Logicielle et Matérielle

### INFO3

## Réalisation et modification d'un processeur jouet

**Durée** : 3 heures

**Support de cours associé** : Processeur

Nous allons réaliser un processeur "jouet" très simple.  
Il ressemble beaucoup au **premier processeur** que le physicien et mathématicien von Neumann a mis au point en 1945.

## 1 Rappel sur le fonctionnement d'un processeur

La figure suivante rappelle les principaux composants du processeur et ses liaisons avec la mémoire.

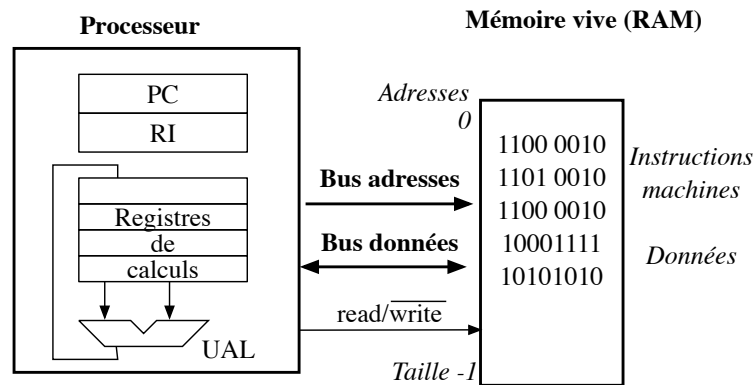


Figure 1 – Processeur et mémoire

- Le processeur est relié à une mémoire (la RAM) qui permet de stocker les programmes et les données de ces programmes.

Les deux bus adresses et données permettent cet accès.

Pour réaliser une lecture (resp. écriture) d'un mot en mémoire à l'adresse X :

- le processeur délivre sur le bus d'adresse l'adresse X ,
- active le signal de lecture/écriture (*read/write* sur la figure) pour une lecture (resp. écriture)
- lit (resp. écrit) la valeur lue (resp. à écrire) sur le bus données.

Les bus données est donc bi-directionnel.

- Il exécute sans arrêt des instructions qui se trouve dans la mémoire. Pour cela, il lit en mémoire chaque instruction qu'il stocke dans un registre particulier (RI : Registre Instruction) et l'exécute avant de passer à la suivante.

Le registre PC contient à tout moment l'adresse de l'instruction en cours d'exécution (ou la suivante).

- Il possède entre autres des instructions (*load* et *store*) permettant de lire ou écrire en mémoire à une adresse particulière.
- Le processeur contient un certains nombres de registres de calcul (mémoire rapide interne). Il peut faire des calculs à partir de ces registres, en général le résultat du calcul doit être stocké dans un des registres de calcul.

## 2 Spécifications du processeur

Il faut commencer par définir les caractéristiques du processeur :

- la taille des bus donnée et adresse

- le jeu d’instruction : nombre de registres de calculs, calcul, mode d’adressage (valeur immédiate, adresse, déplacement...)
- la forme des instructions : le codage binaire et l’organisation (champs spécifiques...)

## 2.1 Taille des bus et forme des instructions

Les bus adresses et données sont sur  $n$  bits. Nous choisissons de coder les instructions sur 2 mots de  $n$  bits.

Le premier mot permet de différencier les instructions. Le deuxième peut contenir une valeur immédiate ou une adresse.

Par la suite  $Mem[X]$  désigne le mot se trouvant en mémoire à l’adresse  $X$ .

## 2.2 Le jeu d’instruction

Voici les 4 instructions que notre processeur sait exécuter. Ce jeu d’instruction n’est pas réaliste, il ne permet pas d’écrire n’importe quel algorithme. Il est cependant représentatif des différents ”types” d’instructions nécessaires. Nous le modifierons par la suite.

- *LOAD #Vi* : chargement par une valeur immédiate (contenu dans l’instruction) du seul registre de calcul du processeur appelé ACC (comme accumulateur) :  $ACC=Vi$
- *STORE [Adresse]* : stockage en mémoire à l’adresse donnée dans l’instruction du contenu du registre ACC :  $Mem[Adresse]=ACC$ . A remarquer que dans le processeur ARM ce type d’instruction n’est pas possible il faut d’abord mettre l’adresse dans un registre.
- *ADD [Adresse]* : addition du contenu de la mémoire à l’adresse spécifiée au registre ACC :  $ACC=Mem[Adresse]+ACC$
- *JUMP Adresse* : Saut à l’adresse spécifiée dans l’instruction, la prochaine instruction exécutée est à cette adresse :  $PC=Adresse$

Voici un exemple de programme écrit dans ce jeu d’instructions assembleur :

```

LOAD #3          -- ACC=3
Etiqu1: ADD [ 8 ] -- ACC=ACC+Mem[ 8 ]
```

```

STORE [8]      -- Mem[8]=ACC
JUMP Etiqu1    -- Saut à l'étiquette Etiqu1

```

## 2.3 Codage binaire des instructions

Il nous reste à organiser et fixer le codage binaire de ces instructions.  
 Nous fixons  $n$  à 4. Les instructions sont toutes codées sur 2 mots.

Nous choisissons d'utiliser les codes suivants :

- *LOAD*  $\#Vi$  : premier mot= 0000, deuxième mot contient la valeur immédiate  $Vi$
- *STORE*  $[Adresse]$  : premier mot= 0001, deuxième mot contient l'adresse  $Adresse$
- *ADD*  $[Adresse]$  : premier mot= 0010, deuxième mot contient l'adresse  $Adresse$
- *JUMP*  $Adresse$  : premier mot= 0011, deuxième mot contient l'adresse  $Adresse$

Voici en binaire le programme précédemment écrit. Le programme est supposé chargé à l'adresse 0.

| <i>Adresse</i> | <i>Contenu mémoire</i> |
|----------------|------------------------|
| 0              | 0000                   |
| 1              | 0011                   |
| 2              | 0010                   |
| 3              | 1000                   |
| 4              | 0001                   |
| 5              | 1000                   |
| 6              | 0011                   |
| 7              | 0010                   |

Vérifiez que cela correspond bien au programme précédent. Quelle est l'adresse correspondant à l'étiquette *Etiqu1* ?

## 3 Réalisation du processeur

Il suffit de décrire l'algorithme que déroule le processeur lorsqu'il exécute les instructions stockées en mémoire. Nous pourrons ensuite utiliser la méthode de construction d'un circuit à base d'une architecture *partie contrôle/partie Opérative*.

### 3.1 L'algorithme du processeur

```
PC=0;           -- la premiere instruction se trouve a l'adresse 0
tant que vrai faire  -- boucle infini
debut
    RI1=MEM[PC]; --le registre RI1 contient le 1er mot de l'instruction
    PC=PC+1;
    RI2=MEM[PC]; --le registre RI2 contient le 2eme mot de l'instruction
    PC=PC+1;

    Suivant RI1:
        0: ACC=RI2;           --LOAD
        1: MEM[RI2]=ACC;      --STORE
        2: ACC= MEM[RI2]+ACC; --ADD
        3: PC=RI2;           --JUMP
    fin
```

Les registres RI1 et RI2 permettent de stocker l'instruction en cours d'exécution.

### 3.2 La partie opérative du processeur

Nous avons besoin de 4 registres *RI1*, *RI2*, *ACC*, *PC*.

Pour les échanges avec la mémoire, nous avons un bus donnée permettant de récupérer ou envoyer une donnée, et un bus adresse permettant de spécifier l'adresse du mot à charger/stocker. Il faut pouvoir écrire *RI2* et *PC* sur le bus adresse. Il faut pouvoir amener (lors d'une lecture en mémoire) le bus donnée :

- sur un des bus opérande de l'UAL pour  $ACC=ACC+[Adresse]$
- à l'entrée de *RI1* et *RI2*

Voici une partie opérative permettant de faire les calculs apparaissant dans l'algorithme :

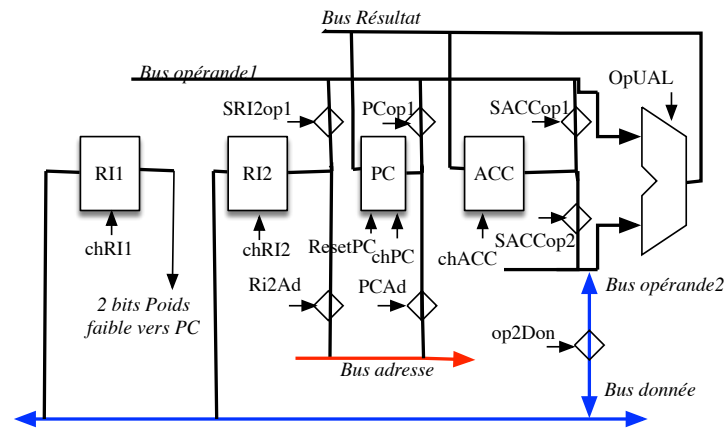


Figure 2 – Partie opérative du processeur

En Lustre il n'est pas possible de décrire les portes 3 états nous les remplaçons par des multiplexeurs. Il n'est pas possible non plus d'avoir un bus en lecture et en écriture à la fois, nous modifions la partie opérative en conséquence, avec deux bus de données.

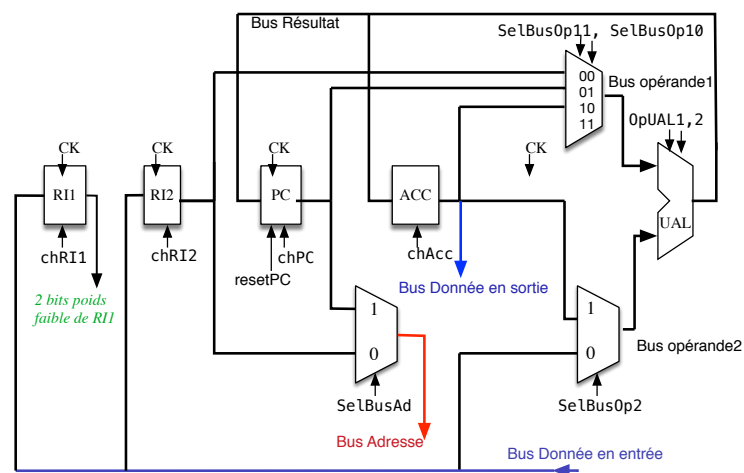


Figure 3 – Partie opérative du processeur avec multiplexeurs

Vérifiez que cette PO est décrite correctement dans le fichier *procPO.lus*.

### 3.3 La partie contrôle du processeur

Nous pouvons remarquer qu'il est possible grâce à la Partie opérative précédemment décrite de charger depuis l'extérieur RI1 et RI2 et d'incrémenter PC en même temps.

Voici l'automate de la partie contrôle :

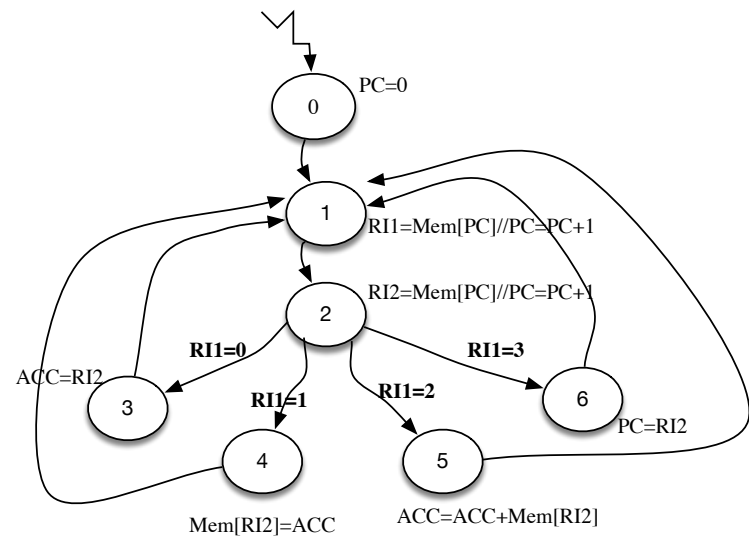


Figure 4 – Partie contrôle du processeur

- Donnez la valeur des signaux de commande de la PO *SelBusOp11*, *SelBusOp10*, *SelBusOp2*, *SelBusAd*, *chRI1*, *chRI2*, *chPC*, *chACC*, *OpUAL1*, *OpUAL0*, *resetPC* de la PO pour chacun des états de cet automate.
- Les signaux de commande de l'UAL *OpUAL1* et *OpUAL0* sont codés comme suit :

| OpUAL1 | OpUAL0 | résultat        | OPÉRATION EFFECTUÉE                            |
|--------|--------|-----------------|------------------------------------------------|
| 0      | 0      | $T = op1 + op2$ | Addition                                       |
| 0      | 1      | $T = op1 + 1$   | Incrémentation                                 |
| 1      | 0      | $T = op1$       | Opérande 1                                     |
| 1      | 1      | $T = op1 - op2$ | Soustraction non utilisé dans un premier temps |

Figure 5 – Opérations de l'Unité Arithmétique

- Quel codage est utilisé pour les états, compact ou un parmi n ?
- Où retrouve t on les compte rendus de la PO composés des deux bits de poids faible du registre *RI1* ?
- Vérifiez vos résultats sur les signaux de commande en analysant le code Lustre fourni dans le fichier *procPC.lus*.

### 3.4 Le processeur et la mémoire

Habituellement le processeur est relié à la mémoire par un bus adresse, un bus donnée (permettant soit de lire, soit d'écrire une valeur dans un mot mémoire) et deux signaux permettant au processeur de décider si il veut :

- accéder à la mémoire ou pas : Chip Select ( $CS$ ),
- lire ou écrire en mémoire : ( $read/\overline{write}$ ).

La figure suivante représente ces connexions :

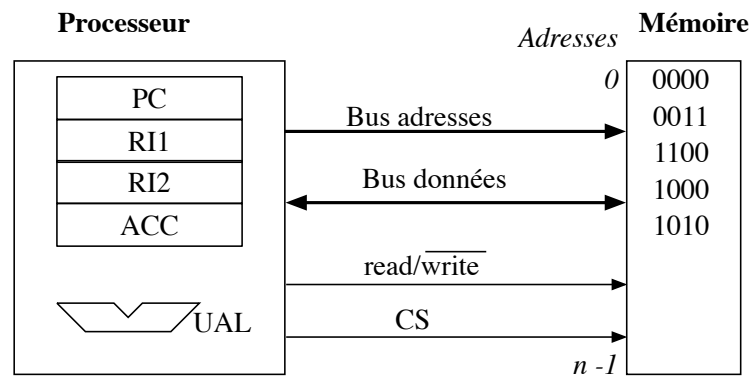


Figure 6 – Connexion du processeur et de la mémoire

Il nous faut adapter ces connexions à notre environnement de simulation.

- Afin de pouvoir simuler notre processeur lors de l'exécution d'un programme, il va falloir tout d'abord mettre ce programme en mémoire. Nous rajoutons donc le moyen de remplir cette mémoire depuis l'extérieur.
- Nous remplaçons le bus de donnée par deux bus, un pour l'écriture, un pour la lecture.
- On suppose que l'on accède toujours à la mémoire, plus besoin de signal CS.

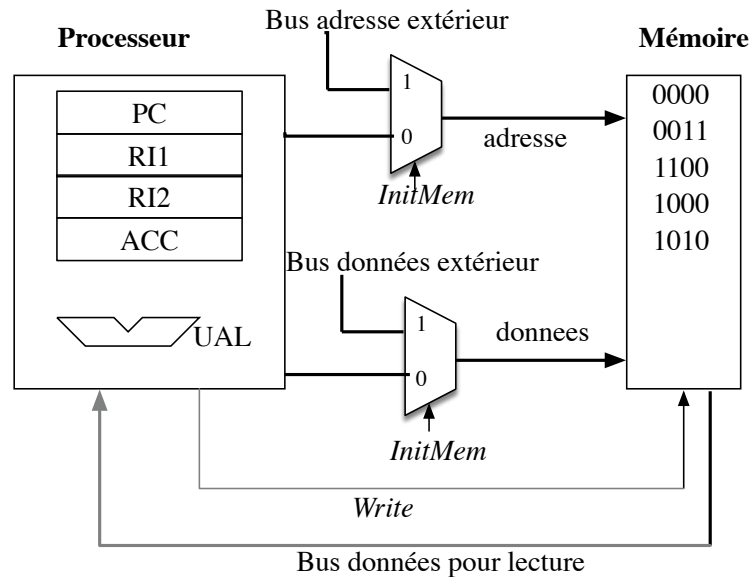


Figure 7 – Connexion du processeur et de la mémoire permettant un accès à la mémoire depuis l'extérieur

- Le signal *write* permet de décider de l'écriture en mémoire du contenu du bus donnée (*entree*).
- Le signal *InitMem* permet de sélectionner les entrées de la mémoire soit de puis l'exterieur, soit depuis le processeur.
- Le bus donnée en lecture contient toujours le mot contenu dans la mémoire à l'adresse donnée sur le bus *adresse*.

Regardez le contenu du fichier *ordinateur.lus*, il va nous permettre de charger un programme et de lancer son exécution.

Compilez le noeud *ordi* à l'aide de *lv6*. Dans un premier temps on fixe la taille des bus à 4, nous avons donc 16 mots mémoire. (*lv6 -ec -node ordi -o ordi.ec ordinateur.lus* )  
Puis à l'aide de Luciole lancez une simulation.

- Les valeurs sur le bus donnée et le bus adresse peuvent être données en décimal.
- Le contenu de la mémoire et des registres sont donnés à droite.
- L'état courant de la partie contrôle est aussi affiché.
- Il est possible de changer l'organisation de l'affichage de luciole en modifiant le fichier *ordi.iop* (que l'on peut générer à l'aide du menu *files* de Luciole).

Dans un premier temps charger le programme suivant dans la mémoire. (*InitMem=1*)

| <i>Adresse</i> | <i>Contenu mémoire</i> |
|----------------|------------------------|
| 0              | 0000                   |
| 1              | 0011                   |
| 2              | 0010                   |
| 3              | 1000                   |
| 4              | 0001                   |
| 5              | 1000                   |
| 6              | 0011                   |
| 7              | 0010                   |

Puis observez son exécution, *InitMem*=0 puis *reset*=1 puis *reset*=0.

- Quelles sont les instructions assembleur de ce programme ?
- Combien de *step* de simulation (période d’horloge) faut il pour exécuter une instruction ?
- Le processeur fonctionne t-il comme convenu ?

## 4 Amélioration du processeur

### 4.1 Ajout d’une instruction de calcul

Modifiez le processeur afin qu’il interprète la nouvelle instruction  
*SUB* [*Adresse*] :  $ACC = ACC - Mem[Adresse]$

Pour cela :

- Choisir un code pour cette instruction.
- L’UAL fourni est déjà capable de faire une soustraction, retrouvez dans les commentaires du fichier *UALproc.lus* le code correspondant à la soustraction.
- Modifier l’automate de la PC (fichier *procPC.lus*) pour que le processeur puisse exécuter cette nouvelle instruction.
- Modifier la partie opérative (fichier *procPO.lus*) afin de renvoyer les comptes rendus nécessaires à cette nouvelle instruction.

## 4.2 Mémorisation des flags (Z, N,V et C) lors d'une instruction de calcul

Ajouter quatre bascules permettant de mémoriser les quatre flags sortant de l'UAL lors d'une instruction de calcul (ADD ou SUB).

Attention le processeur ne doit pas changer ces flags par ailleurs (incrémentation de PC...)

## 4.3 Ajout d'une instruction de branchement conditionnel

On veut ajouter une nouvelle instruction *BHI etiquette* (branchement si strictement supérieur en base 2) qui se branche à l'adresse correspondant à l'étiquette si  $C.\overline{Z}$  vaut 1. L'adresse de branchement est dans le deuxième mot de l'instruction comme pour le *Jump*. Le flag C est le complément de l'emprunt de la soustraction (comme dans le ARM).

Modifiez le processeur afin qu'il interprète cette nouvelle instruction.

- Choisir un code pour cette nouvelle instruction.
- Retournez de la PO à la PC les deux flags mémorisés C et Z.
- Modifiez l'automate de la PC.
- Testez vos modifications en simulant l'exécution du programme suivant.

```
Etiqu1: LOAD #3          -- ACC=3
        STORE [8]       -- Mem[8]=ACC
        LOAD #5         -- ACC=5
Etiqu2: SUB [8]          -- ACC=ACC-Mem[8]
        BHI Etiqu2      -- Saut à l'étiquette Etiqu1
                           -- si supérieur (C.Zbarre)
        JUMP Etiqu1     -- ACC=5
```