Chapitre X- Le processeur maîtrise.

Objectif du chapitre :

- Illustrer l'utilisation de SEP sur une architecture simple.

Le processeur maîtrise est un processeur simplifié conçu dans un but pédagogique pour être modélisé en VHDL par des étudiants de licence d'informatique. Il doit être validé sur un programme de calcul des termes de la suite de Fibonacci ($u_0 = u_1 = 1$, $u_n=u_{n-1}+u_{n-2}$ pour n >= 2). Cette architecture a été choisie pour sa simplicité et parce qu'elle met en œuvre les composants élémentaires utilisés dans toute architecture programmable.

La section 1- présente l'architecture de ce processeur, puis la section 2- présente l'application. La section 3- montre comment SEP peut être utilisé pour modéliser cette architecture. Enfin la section 4- montre les améliorations que l'on peut apporter à l'architecture à partir des constatations faites lors de la modélisation avec SEP.

1- Présentation de l'architecture.

Le processeur est composé d'un CPU et d'une mémoire comme le montre la Figure 1.

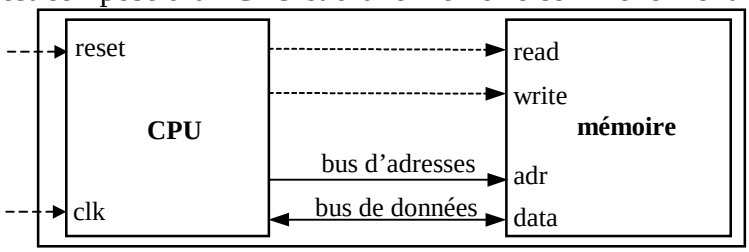

Figure 1 - Structure du processeur « maîtrise »

La Figure 2 montre le schéma bloc du CPU.

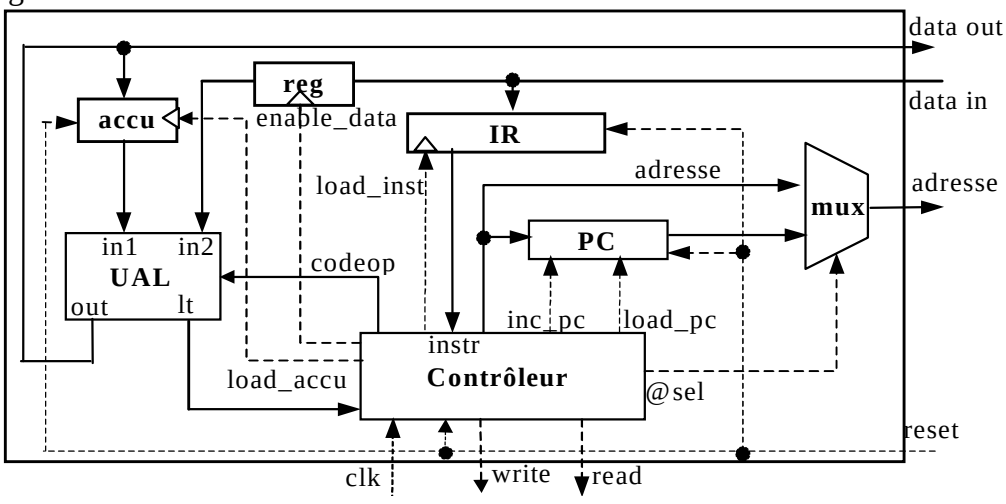

Figure 2 - Structure du CPU

Initialement, l'exercice est prévu pour que l'étudiant écrive le programme en langage machine avec des données et des instructions de 8 bits. Le code opération est codé sur trois

bits qui sont les trois bits de poids fort de l'instruction. Les cinq bits de poids faible constituent l'adresse de la donnée. Les composants *UAL* et *Contrôleur* utilisent le code opération sur 3 bits pour réaliser les actions adaptées.

Le composant *Contrôleur* a un comportement qui correspond à une machine à 6 états cadencée par une horloge. A chaque état, des actions sont effectuées afin d'exécuter l'instruction.

Chacun des composants élémentaires contenus dans cette architecture peut être modélisé aisément aussi bien en VHDL qu'avec SEP.

2- Le calcul des termes de la suite de Fibonacci.

Vue la simplicité extrême de cette architecture, le programme qui permet de calculer les termes de la suite de Fibonacci est assez long. En particulier, il n'y a aucun registre banalisé dans cette architecture, la mémoire est utilisée pour mémoriser les résultats intermédiaires. En particulier, l'adresse 25 initialisée à 1 est utilisée pour mémoriser la valeur u_{n-1}, l'adresse 26 également initialisée à 1 permet de mémoriser la valeur u_n, l'adresse 27 est libre et permet de mémoriser une valeur temporaire appelée *temp*, l'adresse 28 initialisée à 100 contient une borne supérieure de la dernière valeur de Fibonacci que l'on veut calculer.

Une solution possible écrite en assembleur est alors :

```
boucle:
        load 26          ; accu <= u(n-1)
        store 27         ; temp <= accu
        add 25           ; accu <= accu + u(n-2)
        store 26         ; u(n) <= accu
        load 27          ; accu <= temp
        store 25         ; u(n-1) <= accu
        comp 28          ; accu – limit
        jump boucle lt   ; si un < limit il faut reboucler
        halt             ; sinon on stoppe.
```

Afin de montrer l'intérêt de SEP, nous décidons de modéliser cette architecture afin d'utiliser directement le code assembleur tel qu'il est présenté ci-dessus. Nous n'effectuons pas de traduction en langage machine. La section 3- illustre cette implémentation.

3- L'implémentation dans SEP.

3-1. Le multiplexeur.

multiplexeur

Un multiplexeur est un composant combinatoire avec un service combinatoire appelé *sel*. Ce service sélectionne une de ses n entrées en fonction du nombre décimal présent sur son entrée de sélection *sel*, la valeur de l'entrée sélectionnée est émise sur la sortie. Le multiplexeur qui nous intéresse a 2 entrées.

Voilà le fournisseur de service Java que nous utilisons pour représenter ce service :

```
public class Multiplexeur implements sep.model.ServiceProvider {
    Value sel (Value [] in, int s) { return in[s] ; }
}
```

Le comportement du composant multiplexeur est alors construit à partir de ce service en utilisant le mécanisme décrit au chapitre III.

La valeur de retour du service est émise sur la sortie *out*. Le paramètre *s* est associé au port d'entrée *sel*. Le paramètre *in* est associé au port à valeurs multiples *in* (ici 2 entrées : in0 et in1). Le nombre effectif d'entrées est choisi graphiquement par une valeur personnalisée.

Tous les ports d'entrée du composant sont sensibles sur niveau, c'est-à-dire que le service *sel* est exécuté à chaque modification d'une des valeurs sur au moins un des ports *in* ou *sel*.

Gain par rapport aux approches classiques : Un multiplexeur a des entrées paramétrables, il est possible de changer aussi bien le nombre d'entrées que le type des données manipulées sans changer la spécification du comportement.

3-2. L'accumulateur.

L'accumulateur est un composant séquentiel identique à un registre. Il est muni de deux services séquentiels qui réalisent une mémoire d'un mot de taille quelconque de type *Value*. Le service *reset* permet de mettre à zéro la valeur mémorisée, cette valeur est émise sur la sortie *out*. Le service *load* émet la valeur de l'entrée *in* sur la sortie *out*.

registre Voilà le fournisseur de services *Registre* qui va permettre la construction de cet accumulateur :

```
public class Registre implements sep.model.ServiceProvider{
        protected Value content = LevelValue.Undefined;
        public Value load(Value v) { return content=v; }
        public Value reset() { return content = new IntValue(0) ; }
}
```

Le service *load* est déclenché avec une fonction de commande de type *posedge* sur le port *load*. Le paramètre *v* est associé au port d'entrée *in*. La valeur de retour est associée au port de sortie *out*. Le service *reset* est déclenché avec une fonction de commande de type *posedge* sur le port *reset*. La valeur de retour est associée au port de sortie *out*. Ce service est plus prioritaire que le service *load*, ce qui permet de résoudre les éventuels conflits écriture-écriture sur le port *out*. Rappelons que la priorité est choisie avec l'interface graphique par le mécanisme expliqué au chapitre III.

Gain par rapport aux approches classiques : Le type de la donnée manipulée est quelconque ici.

3-3. Le registre d'instructions et le registre reg.

Ce sont des registres simples identiques à l'accumulateur. Etant donné que l'instruction que nous traitons n'est pas une instruction langage machine, l'instruction lue dans la mémoire est envoyée au décodeur d'instructions (cf. contrôleur). C'est lui qui déterminera le contrôle associé ainsi que l'adresse éventuelle de la donnée concernée ou l'opération arithmétique à effectuer.

3-4. L'unité arithmétique et logique.

UAL L'unité arithmétique et logique est un composant combinatoire constitué de deux services combinatoires. Le premier service *execute* calcule une opération arithmétique ou logique sur les valeurs présentes sur les ports d'entrée *in* et émet le résultat sur le port de sortie *out*. L'opération effectuée est déterminée par le code opération présent sur le port d'entrée *cop*. Le deuxième service *lt* émet une valeur binaire sur le port *lt* qui vaut 1 si le résultat de l'opération est strictement inférieur à zéro, 0 s'il lui est supérieur ou égal.

Voilà le fournisseur de services *UAL* qui va permettre la construction de ce composant :

```
public class UAL implements sep.model.ServiceProvider{
```

```
        protected Value res = LevelValue.Undefined;
        public Value execute(Value[] values, String cop) {
                return res = sep.type.Type.perform(cop, values);
        }
        public boolean lt() {
                return ((LevelValue)sep.type.Type.perform("Lt", res, new IntValue(0))).isHigh();
 }
}
```

La méthode statique *sep.type.Type.perform* implémente le mécanisme présenté au chapitre VI et sélectionne l'opération adaptée en fonction du type des données concernées par l'opération. Le code opération est lu sous la forme d'une chaîne de caractères.

La valeur de retour du service *execute* est émise sur le port *out*. Le paramètre *values* est associé au port d'entrée *in*. C'est un port multiple, le nombre effectif d'entrées sera paramétré avec l'interface graphique. Le paramètre *cop* est associé au port d'entrée *cop*.

La valeur de retour du service *lt* est émise sur le port *lt*.

Gain par rapport aux approches classiques : Le nombre d'entrées est paramétrable, de plus le mécanisme de liaison dynamique présenté au chapitre VI permet d'utiliser ce composant pour additionner des nombres entiers, des nombres réels ou des données de n'importe quel type défini dans l'arbre de sous-typage comme par exemple des WLDD (cf. chapitre VIII).

3-5. Le compteur de programme PC.

PC C'est un module composé d'un additionneur +, d'un multiplexeur *mx* et d'un registre *pc* (cf. Figure 3). Ce module déclare deux services de module ***pc:=pc+1*** et ***pc:=val*** définis respectivement par ***mx.sel*** <= ***0 ; pc.load*** et ***mx.sel*** <= ***1 ; pc.load***. De plus, un service *reset* permet la remise à zéro du registre *pc*.

Le service ***pc:=pc+1*** implémente la commande *inc_pc* et le service ***pc:=val*** implémente la commande ***load_pc*** (cf. Figure 2).

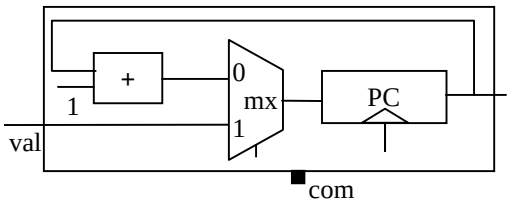

Figure 3 – Le compteur de programme : PC.

Gain par rapport aux approches classiques : L'utilisation de services de modules améliore la lisibilité dans la description du jeu d'instructions (cf. chapitre IV).

3-6. Le contrôleur.

Le contrôleur est un composant séquentiel représenté par une machine d'état. Il déclare deux services. Le service *reset* permet la remise à zéro de l'état. Le service *clk* déclenche le changement d'état. C'est une machine à six états, chaque état effectue une partie des actions nécessaires à l'exécution d'une instruction.

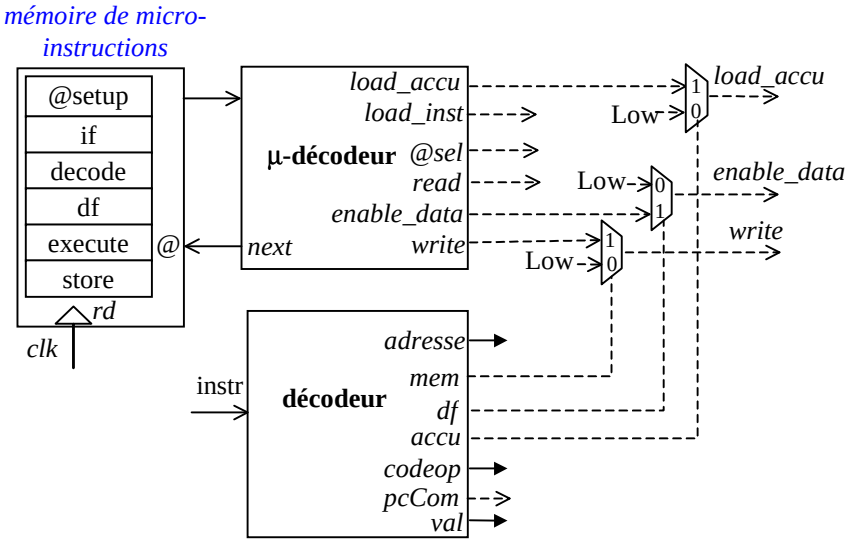

Figure 4 – Détail du contrôleur.

Les six états sont :

- @ instruction setup : Mettre l'adresse de la prochaine instruction sur le bus d'adresse.
- instruction fetch (if) : Lire l'instruction dans la mémoire.
- decode : Décoder l'instruction (code opération de l'opération arithmétique à effectuer, adresse de la donnée, exécution conditionnelle, saut).
- data fetch (df) : Lire la donnée en mémoire.
- execute : Exécuter l'opération arithmétique s'il y a lieu.
- store : Mémoriser le résultat dans le support adapté (accu, mémoire, aucun).

Ces six étapes sont à effectuer quelle que soit l'instruction. La composition structurelle de la Figure 4 présente une modélisation possible. Cette configuration utilise une mémoire de six micro-instructions (@setup, if, decode, df, execute, store), un décodeur de micro-instructions et un décodeur d'instructions. Le décodeur d'instructions positionne l'adresse des données, le code opération et inhibe éventuellement le contrôle généré par le décodeur de micro-instructions en positionnant correctement les multiplexeurs.

Chaque micro-instruction correspond à une des étapes de l'exécution d'une instruction. Une micro-instruction est lue à chaque cycle de l'horloge *clk*.

Le µ-décodeur est un composant décodeur générique présenté au chapitre IV. Son code SEP-ISDL est alors :

```
<@setup
        @sel            0       Int     // le pc est mis sur le bus d'adresse
        read            Low     Level
        write           Low     Level
        load_accu       Low     Level
        enable_data     Low     Level
        load_inst       Low     Level
        next            1       Int     // prochaine µ-instruction = if
<if
        read            High    Level   // lecture de l'instruction en mémoire
        next            2       Int     // prochaine µ-instruction = decode
<decode
        load_inst       High    Level   // instruction vers le décodeur.
        read            Low     Level
        @sel            1       Int     // L'adresse de la donnée est
                                        // envoyée sur le bus d'adresse.
```

```
        next            3       Int             // prochaine µ-instruction = df
<df
        read            High    Level           // Lecture de la donnée en mémoire.
        next            4       Int             // prochaine µ-instruction = execute
<execute
        enable_data     High    Level           // La donnée est envoyée vers l'UAL.
        next            5       Int             // prochaine µ-instruction = store
<store
                write           High    Level           // Ecriture en mémoire.
                load_accu       High    Level           // Chargement de l'accu.
                next            0       Int             // prochaine µ-instruction = @setup.
```

Le décodeur est aussi réalisé avec un composant décodeur générique dont le code SEP-ISDL est décrit et commenté ci-dessous.

Le seul mode d'adressage est un adressage par adresse. On peut aussi définir un macro-identificateur pour les opérations arithmétiques.

```
>adr _NUM_
>cop    add | sub | xor | and
```

La comparaison est l'opération la plus simple, aucun résultat n'est mémorisé, il suffit d'effectuer une soustraction entre l'accumulateur et la donnée chargée en mémoire.

```
<cmp adr
        df      1               Int             // Il faut charger une donnée.
        codeop  sub             Codeop
        accu    0               Int             // Pas d'accumulation
        mem     0               Int             // Pas d'écriture dans la mémoire
        pcCom   pc :=pc+1       String
```

L'instruction de saut est une instruction conditionnelle, si la condition est vraie :

```
<jump   adr  true
        accu    0               Int             // Pas d'accumulation
        mem     0               Int             // Pas d'écriture dans la mémoire
        df      0               Int             // Pas de donnée à lire
        val     $1              Int
        pcCom pc :=val          String          // Saut
```

Si la condition est fausse alors il faut passer à l'instruction suivante :

```
<jump   adr false
        df      0               Int             // Pas de donnée à lire
        accu    0               Int             // Pas d'accumulation
        mem     0               Int             // Pas d'écriture dans la mémoire
        pcCom pc :=pc+1String
<halt
        df      0               Int             // Pas de donnée à lire
        accu    0               Int             // Pas d'accumulation
        mem     0               Int             // Pas d'écriture dans la mémoire
```

L'instruction *load* permet de charger dans l'accumulateur une donnée de la mémoire, on utilise l'opération *mov2* de l'UAL qui recopie la deuxième opérande sur la sortie :

```
<load adr
        codeop mov2             Codeop          // ual.out <= ual.in2.
        df      1               Int             // Il faut charger une donnée.
        accu    1               Int             // Il faut accumuler le résultat.
        mem     0               Int             // Pas d'écriture dans la mémoire
        pcCom   pc :=pc+1       String
```

L'instruction *store* permet de mémoriser en mémoire la donnée contenue dans l'accumulateur, nous utilisons l'opération *mov1* de l'UAL qui recopie la première opérande sur la sortie :

```
<store adr
        codeop mov1          Codeop      // ual.out <= ual.in1.
        df     0             Int         // Pas de donnée à lire
        accu   0             Int         // Pas d'accumulation
        mem    1             Int         // Ecriture dans la mémoire
        pcCom  pc :=pc+1     String
```

Il y a quatre opérations arithmétiques, l'**addition** *add*, la **soustraction** *sub*, le **et logique** *and* et le **ou exclusif** *xor*. L'opération est effectuée avec une donnée lue en mémoire et l'accumulateur. Le résultat est mémorisé dans l'accumulateur.

```
<cop adr
        codeop  $0           Codeop      // On positionne le code opération adapté.
                df     1             Int         // Il faut charger une donnée.
                accu   1             Int         // Il faut accumuler le résultat.
                mem    0             Int         // Pas d'écriture dans la mémoire
                pcCom  pc :=pc+1     String
```

4- Commentaires et améliorations

Ce processeur n'est pas un processeur pipeline. Il utilise plusieurs cycles pour l'exécution d'une seule instruction, l'exécution des instructions ne peut pas être entrelacée. La construction de cette architecture sous SEP met en évidence le goulot d'étranglement qu'elle contient. En effet, on souhaiterait maintenant effectuer les six étapes en un seul cycle du processeur. Cette opération est impossible car certaines instructions nécessitent deux accès à la mémoire : la lecture de l'instruction et la lecture ou l'écriture d'une donnée.

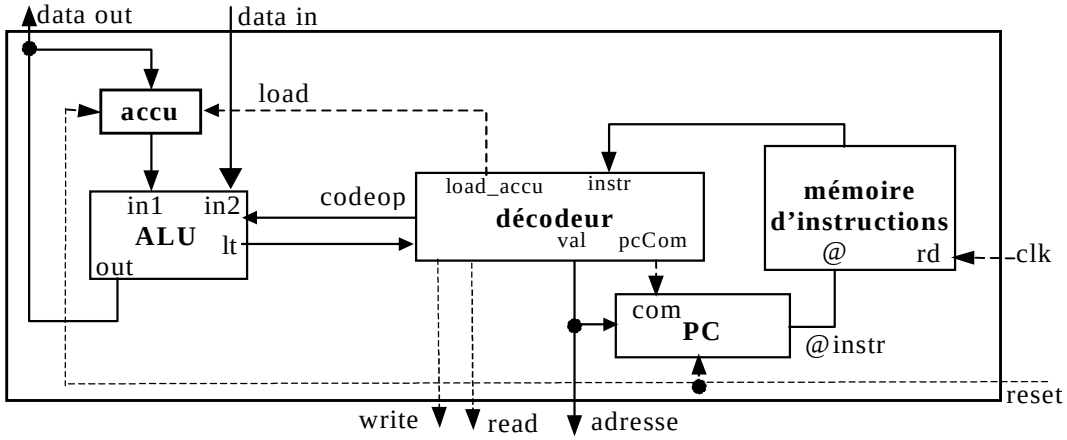

Figure 5 - Structure du CPU modifiée.

En fait, nous avons besoin de deux services de lecture. Ceci implique alors la création de chemins de contrôle disjoints ainsi que de chemins de données séparés pour les instructions et les données. Avec cette modification nous obtenons l'architecture présentée à la Figure 5 qui conduit à une description plus simple dans SEP.

A chaque cycle une instruction est lue et envoyée vers le décodeur d'instructions. Celui-ci génère les commandes pour lire ou écrire une donnée, effectuer une opération arithmétique, charger l'accumulateur ou modifier le pc. Les composants décrits sont identiques à ceux présentés dans la section précédente.

Les trois étapes initiales (*@setup*, *if* et *decode*) sont réalisées par la configuration structurelle sans intervention du décodeur d'instructions. Le PC fournit en permanence

l'adresse de l'instruction suivante, l'horloge déclenche la lecture de l'instruction qui est alors immédiatement délivrée au décodeur d'instructions.

Seul le comportement des étapes *df*, *execute* et *store* doit être décrit dans le comportement du décodeur. Le code SEP-ISDL pour réaliser ce comportement devient alors :

Une macro-action est définie pour le macro-identificateur adr. Elle positionne l'adresse sur le bus et provoque une lecture dans la mémoire.

```
>adr _NUM_
: read
                val             $this       Int         // On met l'adresse sur le bus
                read            Low         Level
                sleep
                read            High        Level       // Lecture de la donnée
>cop    add | sub | xor | and

        <cmp adr
                1.read                                  // positionner l'adresse et lire la donnée.
                codeop          sub         Codeop
                pcCom           pc :=pc+1   String
        <jump adr  true
                val             $1          Int
                pcCom           pc :=val    String
        <jump adr false
                pcCom           pc :=pc+1   String

        <halt                                           // Rien à faire.

        <load adr
                1.read
                codeop          mov2        Codeop      // ual.out <= ual.in2.
                load_accu       Low         Level
                sleep
                load_accu       High        Level
                pcCom           pc :=pc+1   String
        <store adr
                1.read
                codeop          mov1        Codeop      // ual.out <= ual.in1.
                write           Low         Level
                sleep
                write           High        Level       // Ecriture dans la mémoire.
                pcCom           pc :=pc+1   String
<cop adr
                1.read
                codeop          $0          Codeop      // On positionne le code opération adapté.
                load_accu       Low         Level
                sleep
                load_accu       High        Level
                pcCom           pc :=pc+1   String
```

Cette deuxième solution propose une modélisation de plus haut niveau et donc moins dépendante des choix d'implémentation.

Gain par rapport aux approches classiques : Cette approche permet de mettre en évidence une architecture plus simple et plus efficace. De plus, la construction de la deuxième solution dans SEP est beaucoup plus rapide que la construction d'un programme VHDL. Enfin, tous les composants utilisés pour la construction de cette architecture sont réutilisables en l'état pour la définition d'une nouvelle architecture.

Notons pour conclure qu'il est possible d'écrire un modèle presque identique à une description VHDL. Le contrôleur serait alors écrit complètement en Java comme la description d'une machine à 6 états. Cette description ne présente pas beaucoup d'intérêt ici puisque SEP n'apporte rien pour ce type de description.