Chapitre VII- La validation du contrôle associé aux services.

Objectifs du chapitre :

- Présenter les vérifications statiques qui sont faites lors de la construction des services de modules.
- Montrer comment elles permettent de valider les services de modules et le jeu d'instructions.

Les chapitres II et IV montrent comment l'utilisation de services de modules permet d'améliorer la lisibilité des modèles réalisés dans SEP. Cette amélioration est obtenue par une description structurelle plus claire car moins dense et par une description plus agréable du jeu d'instructions. Cet aspect de la modélisation n'est réellement efficace que s'il est accompagné d'outils de spécification active, c'est-à-dire d'outils qui présentent les différents choix possibles ou restants au fur et à mesure de la construction des modèles et interdisent les constructions incorrectes.

Ce chapitre présente la technique utilisée afin d'empêcher la construction de services de modules qui ne pourraient pas être exécutés. De plus, cette technique permet également de valider le jeu d'instructions construit à partir d'une description SEP-ISDL. En effet, les notions de service de modules et de schéma d'instructions sont similaires.

La section 1- présente l'exemple tiré de la modélisation d'un DSP industriel et qui servira à illustrer nos propos. La section 2- montre comment la technique proposée est appliquée à la description des services de modules. Enfin, la section 3- généralise le procédé à une description SEP-ISDL. Notons que ce chapitre traite de la validation du contrôle dans un service de modules ou dans un schéma d'instructions, il présente la technique qui permet de s'assurer que des ressources nécessaires sont présentes. Le chapitre VIII traite de la validation fonctionnelle des services ainsi constitués.

1- L'exemple d'une unité de calcul.

1-1. Le module de multiplication : MU.

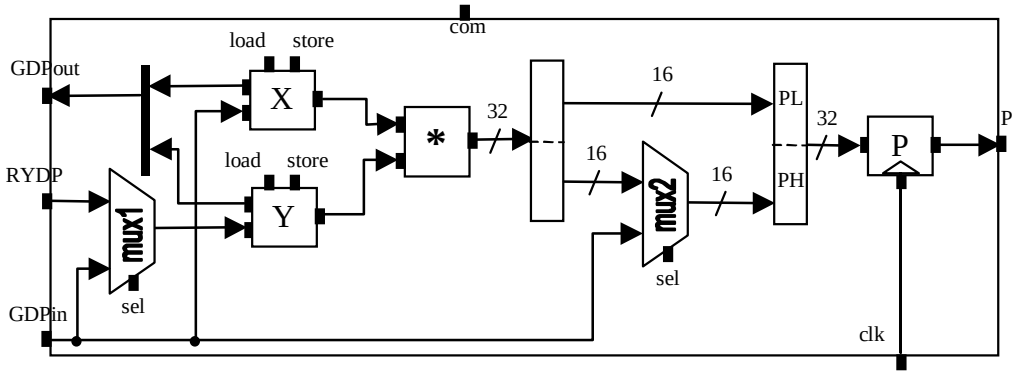

Figure 1 – Le module de multiplication : MU.

La Figure 1 présente une description détaillée des chemins de données du module de multiplication. Cette description est une généralisation de la description faite au chapitre II. En fait, elle constitue un schéma-bloc de au niveau supposé réaliser la fonctionnalité du module de multiplication. Il s'agit principalement d'effectuer la multiplication des valeurs contenues dans les deux registres X et Y. Le résultat sera affecté au registre P synchrone avec l'horloge du processeur.

Un des objectifs de l'utilisation de services de modules est de ne pas surcharger le modèle par la description des chemins de contrôle qui ici n'apporterait pas d'informations supplémentaires ni sur les performances de l'architecture, ni sur sa fonctionnalité.

Les quatre services élémentaires (*loadX, loadY, storeX, storeY*) présentés au chapitre II sont évidemment conservés. En effet, ils permettent l'utilisation des registres X et Y comme des registres de base excepté que le module de multiplication calcule en permanence le produit des valeurs que ces registres contiennent. Remarquons, que le service *storeY* est un peu plus complexe que les autres. En effet, il faut de plus positionner le multiplexeur *mux1* afin qu'il sélectionne son entrée reliée au bus GDP par l'intermédiaire du port GDPin.

De plus, le concepteur désire rajouter le service *ph := GDP* qui permet d'affecter la partie haute du registre P avec la valeur présente sur le bus GDP – il suffit pour cela de sélectionner correctement le multiplexeur *mux2* – et le service *p := X * Y* qui charge les deux registres X et Y et sélectionne les multiplexeurs *mux1* et *mux2*.

1-2. Le module arithmétique et logique : ALB.

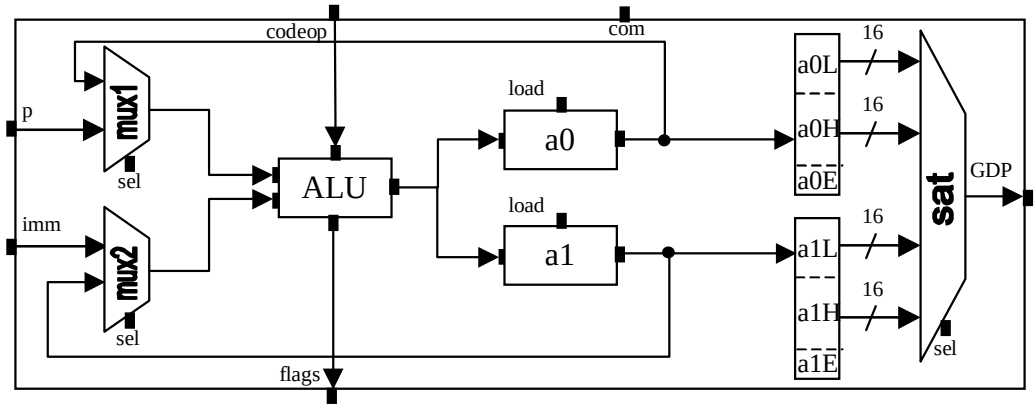

Figure 2 – Le module arithmétique et logique : ALB.

La Figure 2 présente les chemins de données du module arithmétique et logique. Ce module contient une unité arithmétique et logique *ALU* capable d'effectuer suivant son code opération (*codeop*) des calculs arithmétiques ou logiques. Les résultats calculés peuvent être mémorisés (service séquentiel *load*) dans les deux accumulateurs a0 et a1. Les opérandes sont choisies grâce à deux multiplexeurs *mux1* et *mux2*. Les deux accumulateurs peuvent manipuler des données de taille quelconque, dans le cas qui nous intéresse ils ne manipuleront que des données sur 36 bits. Leurs valeurs peuvent être décomposées en données de 16 bits (a0H, a0L, a1H, a1L) et émises sur le bus GDP grâce à l'unité *sat* qui possède un service séquentiel *sel*.

Ce module propose les services *write_a0* et *write_a1* qui permettent la mémorisation dans les accumulateurs, les services *GDP := a0H, GDP := a0L, GDP := a1H, GDP := a1L* qui permettent par une sélection de l'unité *sat* d'écrire sur le bus GDP la valeur des registres 16 bits a0H, a0L, a1H, a1L, et les services *read_a0, read_a1, read_p, read_imm* qui permettent de sélectionner les multiplexeurs mux1 et mux2 afin de choisir les opérandes de l'ALU.

1-3. L'unité de calcul : CU.

La Figure 3 présente l'unité de calcul constituée du module de multiplication (MU) et du module arithmétique et logique (ALB). Le composant combinatoire *sign* est un composant d'extension de signe. Il transforme les données signées en un équivalent sur 36 bits.

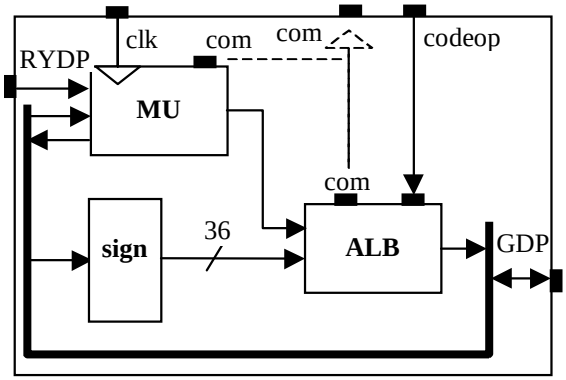

Figure 3 – L'unité de calcul : CU.

De plus on définit les services *mulacc_a0* et *mulacc_a1*. Ces services effectuent en parallèle une multiplication sur les données présentes sur les bus GDP et RYDP avec le module de multiplication et une opération entre p et respectivement a0 ou a1, le résultat est accumulé dans l'accumulateur de départ, c'est-à-dire a0 ou a1. L'opération effectuée dépend du code opération présent sur l'ALU au moment de l'opération.

2- Une validation partielle des services de modules.

Comme il est précisé dans le chapitre V, chaque service séquentiel ne peut être invoqué qu'une seule fois par cycle de simulation *(règle 1)*. Les services combinatoires (ici, les multiplexeurs) peuvent quant à eux être invoqués plusieurs fois par cycle, a priori une fois par micro-instant. Il semble cependant sain de n'autoriser leur invocation volontaire, c'est-à-dire par leur signal de contrôle (ici, *sel*), qu'une seule fois par instant *(règle 2)*. En effet, les micro-instants n'apparaissent pas à ce niveau de description. Les instants apparaissent explicitement avec l'action *sleep* qui permet au concepteur de spécifier que les actions suivantes ne seront exécutées qu'à l'instant suivant.

La validation consiste alors à s'assurer que pour un service donné, les services combinatoires ne sont invoqués qu'une fois par instant, que les services séquentiels ne sont invoqués qu'une fois par cycle et que les connecteurs ont la capacité de recevoir l'information. En fait, aussi bien les signaux de contrôle que les bus de données sont des composants combinatoires avec plusieurs services de lecture et un seul service d'écriture. En appliquant la règle précédente, il est donc impossible d'écrire sur un signal de contrôle ou sur un bus de données plusieurs valeurs dans un même instant.

Cette validation se fait à la construction ; lors de la construction d'un service de modules, les services élémentaires disponibles et utilisables sont présentés, seuls sont présentés les services élémentaires qui respectent la règle précédemment énoncée.

Le procédé de spécification active permet alors de s'assurer que la construction de chacun des services présentés à la section 1- est correcte.

2-1. Le module de multiplication.

Les services *loadX*, *storeX*, *storeY* n'utilisent qu'un seul service qui en fait se réduit à un service *load* ou *store* d'un registre *load/store* (respectivement *X.com=load*, *X.com=store*,

Y.com=store), c'est-à-dire à un service *load* d'un registre élémentaire (cf. chapitre II – Figure 7).

Pour sa part, le service *loadY* est l'exécution séquentielle d'un service combinatoire et d'un service séquentiel : *loadY = (mux1.sel<=1 ; Y.com=load)* . Le symbole ';' est l'équivalent de l'instruction *sleep* de SEP-ISDL. Pour comprendre l'utilisation d'une composition séquentielle dans ce cas plutôt que d'une composition parallèle, il faut rappeler que les services séquentiels introduisent un délai (instant) lors de leur exécution, c'est-à-dire qu'ils utilisent les valeurs présentes sur leurs entrées à l'instant précédent. L'utilisation de la séquence ici permet de prendre en compte la valeur effective de la sortie du multiplexeur *mux1* après l'exécution de l'action : *mux1.sel <= 1*.

Le service *ph := GDP* consiste simplement en l'exécution d'un service combinatoire représenté par l'action *mux2.sel <= 1*.

Le service *p := X * Y* défini par *((mux1.sel <= 0 || mux2.sel <= 0) ; (Y.com=store || storeX))*, ne provoque pas non plus de conflit. Lors du premier instant les deux services combinatoires des multiplexeurs sont exécutés ; puis, lors de l'instant suivant, les deux services séquentiels de chargement des registres sont exécutés à leur tour.

En revanche, notons que l'intérêt de cette technique de spécification active réside dans le fait que la construction du service *loadY || loadX* est interdite à cause du conflit sur l'utilisation du service d'écriture du bus connecté au port *GDPout*. Alors que la construction des services *storeX || storeY* ou *storeX || loadY* [1] est autorisée.

Le service *p := X*Y || ph := GDP* n'est pas non plus autorisé à cause du conflit sur l'utilisation du service combinatoire de sélection du multiplexeur *mux2*.

2-2. Le module arithmétique et logique.

De même, les services du module arithmétique et logique sont acceptés car ils ne font intervenir qu'un seul service élémentaire. Dans ce cas, l'utilisation de services de modules, permet de renommer les services élémentaires avec des noms plus explicites qui seront utilisés dans une description SEP-ISDL et la rendront plus lisible.

Toutes les constructions de services où l'introduction du parallélisme engendre des problèmes sont rejetées.

2-3. L'unité de calcul.

Les services *mulacc_a0* et *mulacc_a1* se traduisent respectivement par *(p := x*y || read_p || read_a0) ; write_a0* et *(p := x*y || read_p || read_a1) ; write_a1*.

La décomposition du service *mulacc_a1* en services élémentaires donne l'expression terminale suivante : *(((MU.mux1.sel<=0 || MU.mux2.sel <= 0) ; (MU.Y.com=store || MU.X.com=store)) || ALB.mux1.sel=1 || ALB.mux2.sel=1) ; ALB.a1.load.* Etant donné que la notion d'instant est globale dans un système, il est facile de montrer que cette séquence peut être réorganisée de la façon suivante :
(MU.mux1.sel<=0 || MU.mux2.sel <= 0 || ALB.mux1.sel=1 || ALB.mux2.sel=1) ; (MU.Y.com=store || MU.X.com=store) ; ALB.a1.load qui est acceptable puisque ne contenant aucun conflit.

En revanche, la décomposition du service mulacc_a0 donne la séquence : *(MU.mux1.sel<=0 || MU.mux2.sel <= 0 || ALB.mux1.sel=1 || ALB.mux1.sel=0) ; (MU.Y.com=store || MU.X.com=store) ; ALB.a0.load.* Cette séquence contient le conflit

[1] Le lecteur peut remarquer au passage que du fait de la spécification, les services *storeX || loadY* et *storeY || loadX* n'ont pas un comportement symétrique.

évident *ALB.mux1.sel=1 || ALB.mux1.sel=0* qui provient du fait qu'il manque un chemin de donnée entre le registre a0 et le multiplexeur *mux2*.

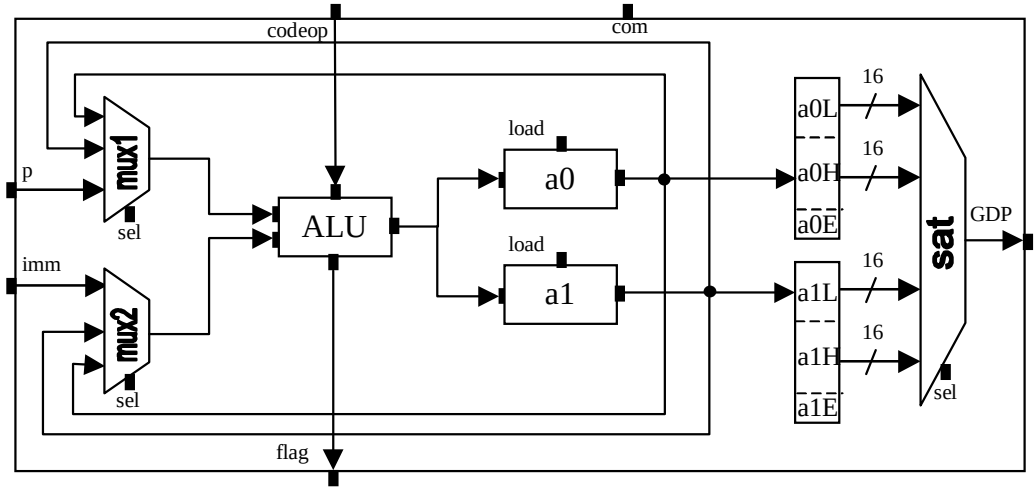

Figure 4 – Le module arithmétique et logique corrigé.

Ce problème est immédiatement résolu par l'ajout du chemin de données. Le module obtenu est présenté par la Figure 4. Il s'agit aussi de remplacer les services *read_a0, read_a1, read_p* et *read_im* par des services *op1 := a0, op1 := a1, op1 := p, op2 := a0, op2 := a1, op2 := imm,* qui permettent de choisir les opérandes de l'unité arithmétique et logique d'une façon qui permet la réalisation d'une opération de type *mulacc*.

Le service *mulacc_a0* est alors défini par : *(p := x*y || op1 := p || op2 := a0) ; write_a0*. Evidemment cette description est valide.

Il est vrai que le concepteur d'une architecture capable d'effectuer un *mulacc* sur les deux accumulateurs n'aurait sûrement pas oublié ce chemin de données. Rappelons tout de même, pour justifier l'intérêt de la méthode et de l'exemple, que le module arithmétique et logique peut provenir de la description d'une architecture précédente qui n'avait pas les mêmes contraintes, un chemin de données de 36 bits en trop coûte cher. De plus, SEP est destiné à servir pour la modélisation rapide avec des modifications interactives des chemins de données. Cette méthode est donc très utile pour s'assurer que les services obtenus restent valides au cours des modifications successives des chemins de données.

3- Généralisation à une description SEP-ISDL.

Lors d'une description du jeu d'instructions en SEP-ISDL ce mécanisme de spécification active va permettre de découvrir les modes d'adressage valides. De plus, seuls des schémas d'instructions valides peuvent ainsi être construits.

En effet, lors de la construction d'un schéma d'instructions, les services offerts par les différents composants de l'architecture peuvent être sélectionnés. Au fur et à mesure de la construction des services ou des schémas d'instructions, seuls les services qui ne violent aucune des deux règles énoncées à la section 2- peuvent être sélectionnés. Intuitivement, les ressources disponibles dans une architecture sont en nombre fini et ne peuvent être utilisées plusieurs fois simultanément, l'utilisation de services mobilise ces ressources, l'outil s'assure lors de la construction de schémas d'instructions que l'utilisation concurrente des services telle qu'elle est réalisée est possible en fonction des ressources disponibles.

Pour illustrer ce point, analysons l'exemple de l'architecture Harvard de la Figure 5. Cette architecture met en œuvre un décodeur d'instructions qui décode les instructions en provenance de la mémoire d'instructions non représentée sur ce schéma, deux mémoires de données *X* et *Y* qui permettent de générer deux données par cycle, un générateur d'adresses *DAU* et l'unité de calcul *CU* décrite dans les sections précédentes.

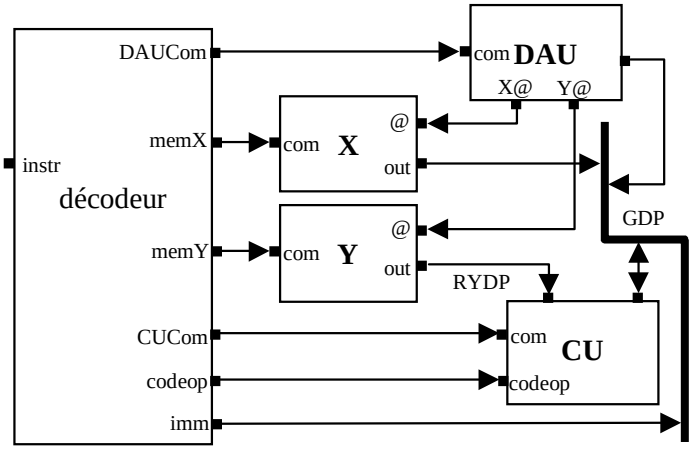

Figure 5 – Configuration simplifiée d'un processeur de traitement du signal.

Le générateur d'adresses est un composant dont le comportement est complètement décrit en Java, son fonctionnement est détaillé au chapitre XI. Principalement ce générateur d'adresses contient six registres r0, ..., r5. Il possède trois services séquentiels *readX(ri)*, *readY(rj)*, *read_rn*. Le service *read_rn* permet de lire la valeur du registre *rn* sur le bus GDP. Les services *readX(ri)* et *readY(rj)* permettent une lecture de la valeur des registres *ri* ou *rj* vers les bus d'adresse X@ ou Y@ afin de générer une adresse pour les mémoires de données X ou Y.

L'architecture a été construite afin de permettre l'exécution en un cycle d'une instruction *mac* qui réalise une multiplication à partir de données lues par un adressage indirect[2] dans une mémoire, une accumulation du résultat précédent de la multiplication avec une valeur contenue dans un accumulateur, et finalement écrit le résultat dans cet accumulateur.

Vérifions alors que l'architecture proposée permet d'effectuer cette opération. Pour cela, nous réalisons la description partielle SEP-ISDL suivante :

```
<aX a0 | a1
<rIind (r0)|(r1)|(r2)|(r3)
:read
      DAUCom        readX,#this     Multiple
      memX          read            String
:write
      DAUCom        readX,#this     Multiple
      memX          write           String

<rJind (r4)|(r5)
:read
      DAUCom        readY,#this     Multiple
      memY          read            String
:write
      DAUCom        readY,#this     Multiple
      memY          write           String
```

[2] L'adressage indirect consiste à lire en mémoire une donnée dont l'adresse est contenue dans un registre (i.e. r0). Cet référence se note généralement (r0).

```
>mac (rIind, rJind, aX)
1.read                                          // Lecture du premier registre indirect
2.read                                          // Lecture du deuxième registre indirect
codeop      Add     String
sleep                                           // Attendre que les lectures soient effectives.
CUCom       mulacc_, #3     MultString          // Invoquer le service mulacc sur l'accumulateur.
```

Le schéma d'instruction décrit mac (rIind, rJind, aX) énonce que l'opération se fait, en lisant simultanément, la valeur du premier registre par un adressage indirect, la valeur du deuxième registre par un adressage indirect et en positionnant le code opération addition sur l'unité arithmétique et logique ; il faut ensuite attendre l'instant suivant pour que les opérations de lecture soient effectives, le service *mulacc_a0* ou *mulacc_a1* de l'unité de calcul est alors invoqué.

Les services *mulacc* de l'unité de calcul ont été acceptés (cf. section 2-3). Le premier adressage indirect utilise le service *readX(ri)* de l'unité DAU donc le bus X@, ainsi que le service *read* de la mémoire X, donc le bus GDP. Le deuxième adressage indirect utilise le service *readY(rj)* de l'unité DAU donc le bus Y@, ainsi que le service *read* de la mémoire Y, donc le bus RYDP. Aucun conflit de ressources n'apparaît donc ce service est accepté.

Le schéma d'instruction *mac* avec deux adressages indirects est autorisé, en effet il met en œuvre les deux bus indépendants RYDP et GDP. En revanche, des conflits apparaissent lors de l'utilisation des modes d'adressage immédiat *imm*, c'est-à-dire lorsque la donnée à multiplier est donnée dans l'instruction, elle est alors directement émise sur le bus GDP. Le mode d'adressage direct *rN* où la donnée à multiplier est contenue dans un registre pose aussi des problèmes à cause du partage du GDP dans les chemins de données. Le tableau suivant montre les modes d'adressage pour l'instruction *mac* dont la construction est autorisée.

Les modes d'adressage supplémentaires nécessitent la définition de macro-identificateurs supplémentaires :

```
<imm _NUM_
:read
    imm             #this           ?
<rN r0|r1|r2|r3|r4|r5
:read
    DAUCom          read_,#this     Multiple
<aXX a0H|a0L|a1H|a1L
:read
    CUCom           read_,#this     MultString
```

Valeur 1	Valeur 2	Résultat	Explication
rIind	rJind	Accepté	Voir ci-dessus.
rIind	imm	Refusé	Les deux valeurs sont générées sur le bus GDP.
imm	rN	Refusé	
rN	imm	Refusé	
aX	imm	Refusé	
imm	rJind	Accepté	La première valeur est générée sur le bus GDP, la deuxième valeur sur le bus RYDP.
rN	rJind	Accepté	
aX	rJind	Accepté	

Pour conclure, ce chapitre illustre sur un exemple concret l'application d'une technique de spécification active afin de valider la construction des services de module et des schémas d'instructions et de s'assurer qu'ils restent valides malgré les différentes modifications des chemins de données.

Attention, dans ce chapitre, il ne s'agit pas de vérifier que les chemins de données sont corrects mais seulement de s'assurer que la composition de services concurrents telle qu'elle est réalisée dans les services de modules et dans les descriptions SEP-ISDL est cohérente en fonction des ressources disponibles.

La validation fonctionnelle des services est traitée dans le chapitre VIII.