Chapitre IV- La prise en compte des processeurs et de leur jeu d'instructions.

Objectifs du chapitre :

- Expliquer la modélisation dans SEP des architectures programmables.
- Présenter le décodeur générique d'instructions.
- Présenter le langage utilisé pour modéliser les jeux d'instructions : SEP-ISDL.

Comme il a été vu dans les chapitres précédents, le concepteur peut rapidement construire les chemins de données de son architecture (cf. Chapitre II). En effet, il suffit simplement d'assembler des composants de bibliothèque puis de les lier par l'intermédiaire de bus de données ou de signaux de contrôle.

Cependant, pour la description d'architectures programmables, il faut être capable de modéliser le décodeur d'instructions, c'est-à-dire le jeu d'instructions et ce, très tôt dans la phase de conception afin de pouvoir évaluer partiellement les blocs construits.

Une description structurelle du décodeur d'instructions au niveau des portes logiques n'est pas envisageable. Un décodeur contient plusieurs dizaines de milliers de portes, une telle approche est très longue et n'est pas incrémentale. Il n'existe pas de lien direct évident entre un mode d'adressage et une porte logique.

Les approches classiques à base de descriptions en VHDL ou en Verilog modélisent en général le décodeur sous la forme d'une machine d'états dont la taille peut devenir très rapidement importante. Cette machine d'états manipule les instructions en langage machine, c'est-à-dire des ensembles de bits. Ces approches, comme la précédente, ne sont pas incrémentales ; en effet, l'ajout de nouvelles instructions ou de modes d'adressage peut complètement remettre en cause la machine d'états ou la structure des instructions exprimées en langage machine.

L'implémentation efficace d'un décodeur d'instructions dont le jeu d'instructions est entièrement maîtrisé n'offre pas de difficultés techniques majeures. C'est une phase très longue et les problèmes viennent souvent du fait que la conception n'est pas atomique. Les modifications successives du jeu d'instructions au cours de la vie d'un processeur conduisent souvent à des décodeurs beaucoup trop complexes ; ils occupent alors une surface de silicium beaucoup plus importante qu'il n'est nécessaire. Ces modifications interviennent, soit pour améliorer les performances et sont alors dues à des oublis, soit dans les générations suivantes d'une famille de processeurs.

Ce phénomène est accentué pour les jeux d'instructions complexes des processeurs de type CISC (Complex Instruction-Set Computer) ou DSP (Digital Signal Processor). Ces modifications entraînent alors la construction de processeurs dont les performances effectives sont sensiblement plus faibles que les performances théoriques estimées.

La réalisation est plus ou moins complexe suivant les architectures, elle est assez simple pour les architectures RISC (Reduced Instruction-Set Computer) mais plus complexe pour les

CISC, les DSP ou les processeurs VLIW (Very Large Instruction Word). Cependant leur fonctionnement est similaire. En effet, il consiste toujours en la transformation d'une instruction spécifique sous forme de langage machine en une composition séquentielle ou concurrente d'actions (données et contrôle) vers les unités fonctionnelles.

Ainsi, il est possible de définir les caractéristiques générales d'un composant de décodage d'instructions. Un décodeur générique d'instructions a été défini dans l'environnement SEP comme un composant élémentaire. Ce composant peut être paramétré en y associant un fichier de description du jeu d'instructions écrit dans le langage SEP-ISDL[1]. Ce langage a été défini afin de prendre en compte efficacement les caractéristiques spécifiques de SEP. Le décodeur générique a été validé en modélisant deux processeurs industriels : le OAK + (processeur de traitement du signal de DSP group) et le ARM7 (processeur RISC de ARM Limited).

Grâce à la définition de ce composant, les modèles des architectures manipulent directement des instructions pseudo-assembleurs. Ces instructions sont du même niveau que des instructions assembleurs, mais leur schéma de représentation est canonique et ne dépend pas d'un assembleur particulier. Ainsi, il est possible de représenter tous les mécanismes d'un langage assembleur. De plus, ces instructions pseudo-assembleurs manipulent des services de haut niveau (services de module : cf. chapitre II), ce qui améliore la lisibilité, la maintenance et les possibilités de réutilisation.

Dans un premier temps, la définition binaire des instructions n'est pas fondamentale afin d'évaluer les performances de l'architecture construite. La méthode proposée permet une simulation cycle par cycle et prend en compte les instructions avec un haut niveau de parallélisme et dont l'exécution est répartie sur plusieurs cycles. Ce chapitre détaille le fonctionnement du décodeur générique (cf. section 1-) et présente les propriétés de SEP-ISDL (cf. section 2-). La section 3- explique comment les mécanismes d'interruption et de pipeline sont pris en compte de façon assez indépendante de la description du jeu d'instructions en SEP-IDSL. Enfin deux exemples illustrent le propos dans la section 4-.

De nombreux langages permettent la création de simulateurs non spécifiques d'un processeur. Certains de ces langages ne prennent en compte que le jeu d'instructions des processeurs cibles (ISDL **[HHD97]**, nML **[FPF95]**), ils ne permettent qu'une description réduite de l'architecture sous la forme d'une liste d'unités de mémorisation et d'unités fonctionnelles. D'autres ne permettent que la description de l'architecture (MIMOLA **[LeM98]**, ISPS **[Bar81]**), ces langages ne tiennent pas compte du jeu d'instructions ou essayent de l'extraire à partir de l'architecture. Ces méthodes d'extraction ne s'appliquent pas bien pour les jeux d'instructions complexes. Enfin, récemment des langages prennent en compte aussi bien l'architecture que le jeu d'instructions (LISA **[PHZ99]**, EXPRESSION **[Hal&co99]**, TDL **[Kas00]**) cependant leur objectif est de réaliser des compilateurs qui s'adaptent aux processeurs cibles, ils s'orientent vers une description détaillée des contraintes sur le pipeline, en particulier avec des tables de réservations des ressources.

Dans SEP, nous utilisons une description plus précise de l'architecture qui permet une exploration plus approfondie de l'influence des chemins de données sur les performances. Nous modélisons le jeu d'instructions assez précisément afin de permettre une simulation cycle par cycle fiable et la description du pipeline est volontairement plus sommaire. De plus, nous pouvons utiliser efficacement les mécanismes spécifiques de SEP. En particulier, SEP-ISDL est bien adapté aux types définis dans SEP (chaînes de caractères, valeurs multiples) ainsi qu'à l'utilisation des services du module qui n'existent pas dans les autres langages.

[1] SEP-ISDL : SEP Instruction-Set Description Langage : Langage de Description du jeu d'instructions de SEP.

1- Le décodeur générique d'instructions.

Plusieurs solutions sont possibles pour décrire le décodeur d'instructions.

La première concerne le mécanisme de description structurelle de SEP par agencement de portes logiques. Une deuxième solution serait de modéliser le jeu d'instructions par une description en Java de la machine d'états équivalente. Cette approche logicielle de haut niveau permettrait une simulation plus rapide, l'utilisation de la technologie objet permet une description plus rapide et plus efficace. Cependant, cette approche met en œuvre de nombreuses classes. Ces classes se retrouveraient spécialisées pour chacune des architectures modélisées afin de décrire précisément les modes d'adressage et la spécificité de chaque architecture. Les interférences dues à l'utilisation de classes communes peuvent être dangereuses et conduire à un comportement mettant en œuvre des mécanismes de liaison dynamique trop complexes pour être modifiés aisément.

La solution retenue, met en œuvre un décodeur générique. Cette solution logicielle est rapide et élimine les problèmes des deux approches précédentes.

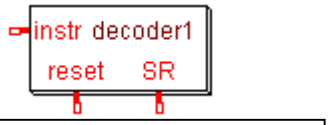

Figure 0a – Le décodeur générique

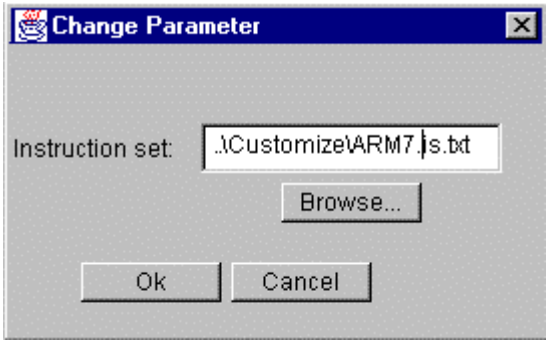

Figure 0b – Personnalisation du décodeur générique avec un fichier SEP-ISDL.

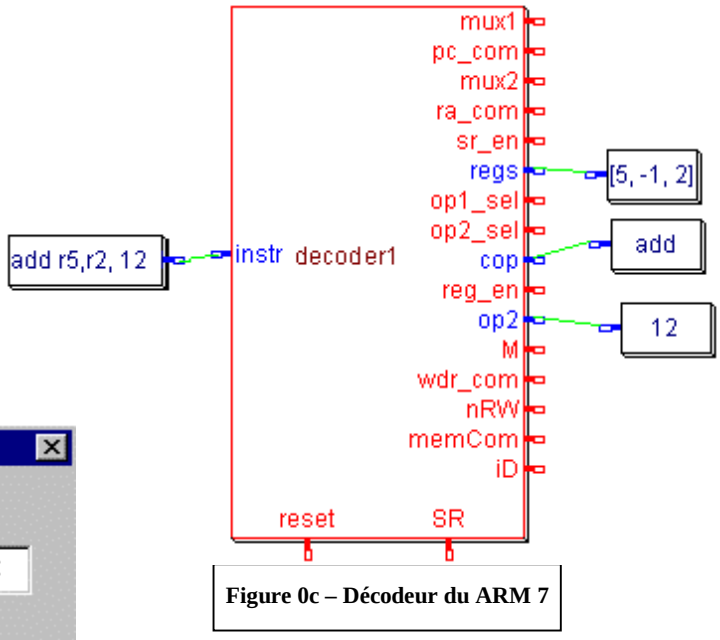

Figure 0c – Décodeur du ARM 7

Un décodeur générique d'instructions est un composant séquentiel SEP constitué de deux services (cf. Figure 0a). Le service *reset* qui permet la mise à zéro de l'état du décodeur ; ce service est déclenché par un signal de commande de type *posedge* sur le port *reset*.

Le service *decode* qui décode une instruction : ce service est commandé par un signal de commande contrôlé par le port *instr*. Le service est exécuté pour chaque occurrence d'événement sur le signal. Notons que le signal de commande n'est pas nécessairement une séquence minimale. Deux événements consécutifs peuvent être évalués avec la même valeur, c'est-à-dire la même instruction à décoder.

Les instructions pseudo-assembleurs qui parviennent au décodeur générique sont décodées en utilisant la description du jeu d'instructions écrite en SEP-ISDL. Pour cela, le concepteur doit paramétrer le décodeur générique avec un fichier SEP-ISDL (cf. Figure 0b). Les sorties du composant sont construites (cf. Figure 0c) à partir de cette description qui est analysée pour déterminer le comportement relatif au décodage de chaque instruction.

Le décodeur générique de SEP possède une entrée *SR*, qui permet à l'environnement de lui fournir l'ensemble des signaux de statut. Ces signaux de statut sont utilisés lors du décodage d'une instruction conditionnelle.

L'utilisation de la notion de décodeur générique, d'un pseudo-assembleur, de la personnalisation d'un composant spécifique à partir d'un composant générique par une description textuelle dans un langage adapté aux mécanismes introduits dans SEP, permet une prise en compte rapide et efficace du jeu d'instructions des architectures programmables modélisées.

2- Le langage SEP-ISDL.

2-1. Présentation du langage.

L'exécution d'une instruction est décomposée en au minimum trois phases (*fetch*, *decode*, *execute*). La phase de *fetch* est réalisée matériellement, elle consiste à déclencher la lecture de l'instruction suivante à partir de la mémoire d'instructions en fonction de l'adresse calculée *PC*. L'instruction ainsi lue est diffusée sur le bus d'instructions vers le port *instr* du décodeur. Le décodeur traduit l'instruction pseudo-assembleur qui lui parvient pour émettre un ensemble d'actions vers les unités extérieures, c'est la phase de *decode*. Enfin, les unités reçoivent les ordres et exécutent le comportement correspondant, c'est la phase *execute*. La séparation entre les différentes phases est effectuée par une description structurelle. Par exemple, une architecture décodant une instruction par cycle peut être construite avec la description structurelle donnée sur la Figure 1.

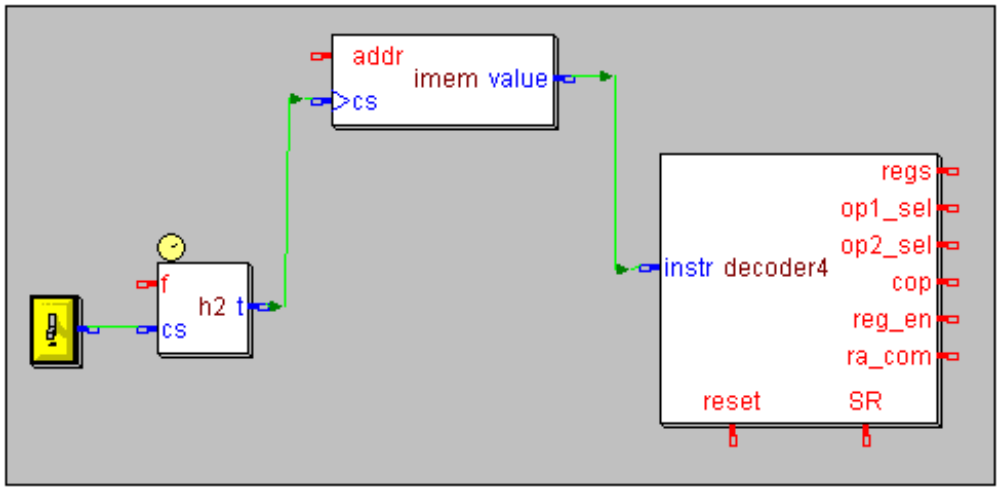

Figure 1 – Configuration structurelle des phases fetch/decode pour une architecture qui traite une instruction par cycle (h2 : horloge, imem : mémoire d'instructions, decoder3 : décodeur d'instructions).

C'est la modélisation de la phase *decode* qui est l'objet de ce chapitre. Des phases supplémentaires peuvent être modélisées (cf. section 3-1).

Le langage SEP-ISDL est basé sur la définition de schémas d'instructions. Pour chaque schéma un ensemble d'actions est défini, une action correspond à l'exécution de services des composants de l'architecture qui appartiennent à la même configuration.

Chaque instruction qui parvient au port *instr* du décodeur est décodée, c'est-à-dire que le décodeur d'instructions identifie le schéma auquel elle est associée. Les actions de ce schéma sont alors exécutées.

Les schémas d'instructions doivent être les plus généraux possibles afin de permettre une description dense du jeu d'instructions. Pour cela, certaines unités lexicales – auxquelles il est possible d'associer un comportement similaire – sont regroupées sous un identificateur unique appelé *macro-identificateur*. En particulier, ces *macro-identificateurs* permettent de modéliser les différents modes d'adressage du jeu d'instructions.

2-2. La déclaration de macro-identificateurs.

macro-identificateur

Un macro-identificateur est un nom suivi de la liste ordonnée des unités lexicales qu'il représente. L'ordre permet d'associer un indice entier à chaque unité lexicale pour la caractériser. Par exemple, on peut définir de la façon suivante le macro-identificateur *rN* constitué des unités lexicales *r0,r1,r2,r3,r4,r5* : **<rN r0|r1|r2|r3|r4|r5**.

Lorsque le macro-identificateur *rN* est défini, le concepteur peut l'utiliser dans les schémas d'instructions pour représenter une ou l'autre des unités lexicales. S'il n'avait pas été défini, l'identificateur *rN* ne représenterait que la chaîne de caractères *"rN"*. Notons que, pour éviter les ambiguïtés, une unité lexicale ne peut être utilisée que par la définition explicite d'un seul macro-identificateur.

Un macro-identificateur qui représente les entiers est prédéfini, il porte le nom *_NUM_*. Les macro-identificateurs peuvent être utilisés dans la définition de nouveaux macro-identificateurs définis par la suite (aucune définition récursive n'est autorisée). Ainsi, la déclaration **<imm #_NUM_** définit le macro-identificateur *imm* comme une chaîne de caractères qui commence par le caractère # suivi par un nombre entier.

Les nombres entiers peuvent être représentés sous forme décimale, ou hexadécimale, le codage est alors précédé de la chaîne *0x*.

Voici la grammaire[2] de déclaration des macro-identificateurs :

```
macro_definitions ::= (‘<’ macro_definition ‘\n’ ( macro_action_definition )* )*
macro_definition ::= macro_nom ( element ) + ( ‘|’ ( element ) + )*
element ::= identificateur | ‘_NUM_’

macro_action_definition ::= macro_action_nom ‘:’ ‘\n’ ( actions ‘\n’ )*
actions ::= port_nom valeur type
          |   port_nom valeur_multiple type_multiple
valeur ::= string | ‘#this’ | ‘$this’
valeur_multiple ::= valeur ( ‘,’ valeur ) +
identificateur ::= string | ‘_’ macro_nom ’_’
```

Un macro-identificateur peut définir des macro-actions, c'est-à-dire des actions qui lui sont spécifiques. Pour la construction il peut utiliser les mots clés *$this* et *#this*. *$this* représente la

[2] Les lexèmes terminaux sont représentés entre quotes et en gras. La production (X)* signifie que X est présent un nombre de fois indéterminé ou absent. Le symbole **‘|’** traduit les différentes productions possibles. *String* représente une chaîne de caractères quelconque. Le **‘\n’** désigne les sauts de ligne obligatoires. Le symbole ‘ø’ représente la production vide.

valeur de l'unité lexicale réelle appartenant au macro-identificateur courant et pour lequel la macro-action a été invoquée. *#this* représente l'indice de l'unité lexicale dans la définition du macro-identificateur.

Pour le détail sur l'utilisation des macro-actions, se reporter aux sections 2-4 et 3-.

2-3. Les schémas d'instructions.

schéma d'instructions

Un schéma d'instructions est composé d'un identificateur d'instruction, d'une liste de paramètres séparés par des virgules et d'un indicateur pour indiquer qu'une instruction est, ou n'est pas, une instruction conditionnelle. Voici la grammaire pour la déclaration d'un schéma d'instructions :

```
schema_declaration ::= '>' schema_nom '(' parametre ( ',' parametre )* ')' condition '\n'
schema_nom ::= identificateur
parametre ::= 'bit' | ( identificateur ) +
condition ::= true | '?' | ø
```

instruction conditionnelle

Les instructions conditionnelles sont des instructions qui ne sont exécutées que lorsque certains indicateurs (flags) sont positionnés. Le jeu d'instructions peut prévoir que la condition soit insérée dans chaque instruction, elle peut alors être prise en compte de façon matérielle. SEP-ISDL modélise ces instructions par un champ supplémentaire à la fin du schéma d'instructions, ce champ peut dépendre de n'importe quel indicateur. Les indicateurs autorisés sont définis dans la déclaration du registre de statut.

registre de statut

La valeur du registre de statut est lue sur l'entrée *SR* du décodeur générique. C'est une liste de bits, sans sémantique prédéfinie, chaque bit est associé à un nom d'indicateur dans la définition du macro-identificateur *SR*. Par exemple, la déclaration suivante déclare l'existence potentielle de sept indicateurs (*Carry*, *Overflow*, *Less than*, *Greater than*, *Less or equal*, *Greater or equal*, *Sign*). Ces indicateurs correspondent aux sept bits de poids faible de la valeur lue sur le port *SR*, l'identificateur correspondant pourra être utilisé dans une instruction conditionnelle.

Exemple de registre de statut: <SR Cy|Ov|Lt|Gt|Le|Ge|S

Prenons l'exemple du schéma d'instructions **>mov (ind, rN*, ?) true**. Il déclare l'instruction conditionnelle *mov*, c'est la présence de l'indicateur de condition *true* qui le montre. Cela veut dire que les actions correspondantes ne sont exécutées que si la condition éventuellement[3] explicitée dans l'instruction (pas dans le schéma) est évaluée à vrai.

Cette instruction a trois paramètres. Le premier correspond à une des unités lexicales représentée par le macro-identificateur *ind* : **<ind (_rN_)**. Le deuxième correspond à une des unités lexicales représentée par le macro-identificateur *rN** : **<rN* _rN_|cfgi|cfgj|sp**. Le troisième peut être n'importe quelle unité lexicale non vide.

Lorsque ce schéma est défini, l'instruction **mov (r1),r5,1** est acceptée et exécutée sans condition car elle est représentée par ce schéma. En effet, l'opération correspond au nom du schéma et chacun des arguments de cette instruction – *(r1), r5, 1* – correspond respectivement à un paramètre du schéma – *ind*, *rN**, *1* –. De même, l'instruction **mov (r1),r5,1 Gt** est acceptée, mais n'est exécutée que si l'indicateur *Gt* est présent dans le registre de statut, c'est-à-dire si le 4^{ième} bit de poids faible est positionné.

L'exécution d'une instruction, consiste en l'exécution des actions associées au schéma d'instructions qui correspond.

[3] En effet, si aucune condition n'est explicitée dans l'instruction, l'instruction est exécutée inconditionnellement.

Le paramètre 'bit' représente un argument dont la valeur peut être '0' ou '1'. Il est utilisé pour les actions conditionnelles (cf. section 2-4).

2-4. Les actions.

Les actions associées à un schéma d'instructions peuvent être de deux types. Les actions simples consistent à émettre une donnée typée sur un port de sortie du décodeur. Les chemins de données de l'architecture diffuseront cette donnée vers le composant à qui elle est destinée. Elles sont donc composées du nom du port sur lequel la donnée doit être émise, d'une expression dont la valeur est la donnée à émettre, du type[4] de la donnée à émettre.

Voici la grammaire associée :

```
instruction_definition ::= ( action ( commentaire ) * '\n' ) *
action ::= action_simple | macro_action

action_simple ::= port_nom expression type
                | port_nom expression_multiple type_multiple
                | 'sleep'
port_nom ::= string
expression ::= expression_simple | expression_conditionnelle
expression_simple ::= string | '#' integer | '$' integer
expression_conditionnelle ::= '#' integer '?' expression_simple ':' expression_simple
expression_multiple ::= expression ( ',' expresion ) +
type ::= 'Int' | 'Level' | 'String' | 'CodeOp' | '?'
type_multiple ::= 'Multiple' | 'MultString'

macro_action ::= integer '.' macro_action_nom
commentaire ::= string
```

On peut remarquer les expressions simples préfixées par '#' ou par '$'. L'identificateur entier qui suit (***integer***) correspond au numéro du paramètre concerné par l'expression. Ce paramètre est alors obligatoirement le nom d'un macro-identificateur. Le numéro 0 correspond à l'identificateur d'instruction. Le préfixe '#' représente une valeur entière qui correspond à l'indice dans la définition du macro-identificateur correspondant de l'unité lexicale passée en argument. Le préfixe '$' représente la valeur textuelle de l'argument correspondant dans l'instruction pseudo-assembleur.

Le résultat de l'évaluation d'une expression conditionnelle dépend de la valeur de l'argument de type 'bit'. S'il vaut 1, la première expression est évaluée, sinon la deuxième expression est évaluée.

Les types correspondent aux types de base définis dans SEP :

Type SEP-ISDL	Type SEP.	Valeurs correspondantes.
Int	sep.type.IntValue	Les valeurs entières
Level	sep.type.LevelValue	HIGH, LOW, ⊥
String	sep.type.Value	Chaîne de caractères
CodeOp	sep.type.Value	Chaîne de caractères qui correspond à un code opération accepté par l'ALU.
Multiple	sep.type.Multiple	Ensemble de plusieurs valeurs.
MultString	sep.type.Value	Chaîne de caractères formée par la concaténation des multiples valeurs qui la composent.
?	sep.type.Value	Aucun type n'est spécifié, le type de la donnée peut changer.

[4] L'information sur le type est utilisée pour valider la composition de composants (cf. chapitre VI).

Les valeurs multiples permettent de diffuser simultanément plusieurs valeurs sur un même bus. Le rassemblement de différentes valeurs peut améliorer la compréhension du modèle dans certains cas.

Les appels de macro-actions correspondent à l'appel des actions déclarées avec un macro-identificateur. L'entier désigne le numéro du paramètre concerné, la notion pointée montre l'appel de l'action définie spécifiquement pour ce service. Les macro-actions permettent une modélisation plus efficace des modes d'adressage. Le comportement d'un ensemble d'actions dans lequel il y a des macro-actions est déterminé en remplaçant l'appel de la macro-action par les actions qui la constituent. Le this est alors remplacé par le numéro de l'argument, c'est-à-dire le numéro utilisé lors de l'appel.

L'action *sleep* est utilisée pour modéliser le parallélisme dans les phases *decode* et *execute* (cf. 3-1).

Pour des exemples qui illustrent l'utilisation d'actions et de schémas d'instructions, le lecteur peut se reporter directement à la section 4-, il devra alors admettre l'utilisation du mot clé *sleep*.

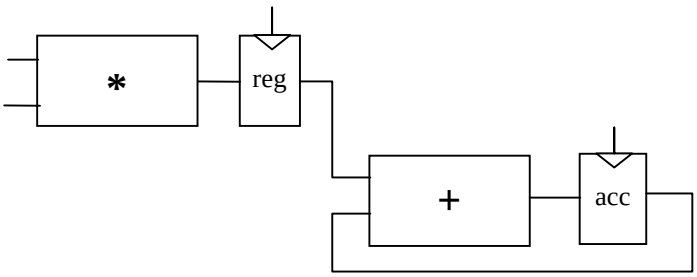

Figure 2 – schéma de multiplication-accumulation

3- Compléments.

3-1. La description du pipeline et du parallélisme dans les instructions.

Les architectures de type DSP ont pour spécificité de traiter quasiment toutes leurs instructions en un cycle d'horloge. Cependant, elles possèdent aussi des instructions avec des modes d'adressage complexes et un haut niveau de parallélisme. C'est pourquoi, ces architectures utilisent le mécanisme de pipeline, ce qui augmente le taux d'utilisation des unités fonctionnelles. Dans la partie d'exécution de l'instruction, ce pipeline est matérialisé par la présence de registres synchrones – c'est-à-dire commandés par l'horloge du système – entre les différents étages. C'est ainsi que par exemple une instruction multiplication-accumulation (MAC) peut être réalisée en moyenne en un cycle (cf. Figure 2).

Avec ce schéma, une multiplication et une accumulation peuvent être exécutées à chaque cycle. Mais il est clair qu'il n'est pas possible de réaliser la multiplication de deux données et d'accumuler le résultat avec un accumulateur en un cycle, il en faudra deux.

Cependant, certaines instructions nécessitent le haut niveau de parallélisme qui n'est pas atteint ici. Par exemple, en un cycle, une instruction doit lire deux valeurs dans un banc de registres, effectuer une opération sur ces données et mémoriser le résultat dans un banc de registres. Ce mécanisme peut être réalisé matériellement par l'utilisation d'une horloge à deux phases. Pendant la première phase, les valeurs sont lues et l'opération est effectuée, puis pendant la deuxième phase, le résultat de l'opération est mémorisé dans le banc de registres. L'ensemble des opérations est réalisé effectivement en un cycle. Plusieurs implémentations

matérielles sont possibles pour implémenter cette spécification. SEP-ISDL permet de modéliser un tel comportement sans préciser la réalisation matérielle choisie, il introduit pour cela l'action *sleep*. Cette action informe le système du changement de phase. Les actions qui suivent l'action *sleep* seront exécutées à une date logique strictement antérieure par rapport à la date à laquelle les actions qui précèdent le *sleep* l'ont été. Cependant la date physique des événements produits sera identique pour chaque action. Enfin si l'utilisateur de SEP veut modéliser le délai réel entre les phases, il peut préciser un temps physique : *sleep n*, n est alors exprimé en nanosecondes. Pour la réalisation de ce mécanisme en simulation, le lecteur se reportera au chapitre V.

3-2. La prise en compte des interruptions.

Le langage SEP-ISDL ne met en œuvre aucun mécanisme qui permette la définition d'entrées supplémentaires. Ceci est dû au fait que, dans les diverses architectures que nous avons modélisées, l'utilisation d'entrées supplémentaires n'a jamais était nécessaire. En particulier, le mécanisme présenté ici permet de prendre en compte les interruptions matérielles.

Cette section explique comment les interruptions matérielles sont prises en compte dans les modèles construits avec SEP.

La gestion des interruptions est presque indépendante du décodeur générique. En fait, on utilise un composant extrêmement simple appelé *Interrupt* que l'on intercale sur le bus d'instructions entre la sortie de la mémoire d'instructions et l'entrée *instr* du décodeur générique. Ce composant prend en entrée une instruction qui provient du bus d'instructions ainsi que les signaux d'interruption. Le composant peut être paramétré afin de prendre en compte autant de niveaux d'interruption que nécessaires. Ce composant transmet simplement son instruction d'entrée lorsqu'aucun signal d'interruption n'est levé. Dans le cas contraire, il transmet une instruction adaptée au niveau de l'interruption déclenchée. Cette instruction qui parvient au décodeur générique sera associée à un schéma d'instructions à définir. Les actions correspondantes permettent de changer le contexte, commuter les registres ou empiler le PC. L'instruction de fin d'interruption permet de restaurer le contexte.

Ce composant prend en compte l'imbrication des niveaux d'interruption. Le lecteur peut se reporter au chapitre XI pour la présentation complète de ce mécanisme.

4- Exemples d'utilisation.

Cette section présente deux exemples extraits d'un fichier de définition des processeurs industriels modélisés (cf. chapitre XI). Le premier exemple ne contient que des macro-indentificateurs sans macro-actions, il illustre l'utilisation des actions simples. Le second contient des macros-actions et illustre leur utilisation pour la modélisation d'instructions avec des modes d'adressage multiples.

4-1. Premier exemple.

Tout d'abord les macro-identificateurs *aX*, *rN*, *ind* et *mod* sont définis de la façon suivante :

```
<aX a0|a1
<rN r0|r1|r2|r3|r4|r5
<ind ( _rN_ )
<mod +1|-1|+s|no
```

Puis un schéma d'instructions et ses actions associées sont définis :

```
>mov (ind,aX,mod) true
DAUcom      readX,#1,$3        Multiple
rW          HIGH               Level        Lecture
CUcom       op2:=gdp           String
CUop        Mov                CodeOp
sleep
CUcom       write_,$2          MultString
selPc       0                  Int
```

Plusieurs instructions pseudo-assembleur peuvent correspondre à ce schéma ; en particulier, c'est le cas de l'instruction '*mov (r5),a0,+s*'. Le code instruction correspond à *mov*, l'argument *(r5)* correspond au paramètre *ind*, l'argument *a0* correspond au paramètre *aX* et l'argument *+s* correspond au paramètre *mod*. Aucune condition n'est précisée, l'instruction est donc exécutée inconditionnellement.

Voici un tableau qui explique l'exécution des actions correspondantes :

Actions	Conséquences	Explications
DAUcom **readX,#1,$3** *Multiple*	La valeur multiple *[readX,5,+s]* est émise sur le port *DAUop*.	*readX* est une chaîne de caractères. *5* est l'indice de l'unité lexicale r5 dans la définition du macro-identificateur *ind*. *+s* est le 3ième argument effectif.
RW **HIGH** *Level* Lecture	La value *High* est émise sur le port *rW*.	*Lecture* est un commentaire
CUcom **op2:=gdp** *String*	La chaîne de caractères *op2:=gdp* est émise sur le port *CUcom*.	
CUop **Mov** *CodeOp*	Le code opération *Mov* est émis sur le port *CUop*.	
sleep	Séparation logique entre l'exécution des actions (cf. 3-1).	
CUcom **write_,$2** *MultString*	La chaîne de caractère *write_a0* est émise sur le port *CUcom*.	a0 est le deuxième argument.
selPc **0** *Int*	La valeur entière 0 est émise sur le port selPc.	

4-2. Deuxième exemple.

On veut modéliser les modes d'adressage de l'instruction *mov* d'un processeur de type DSP. Les macros-identificateurs *rN*, *aXX*, *imm*, *aX*, *mov_src* et *mov_dst* sont définis de la façon suivante :

```
<rN r0|r1|r2|r3|r4|r5
read :
   DAUcom read,#this,NO Multiple        // Demande au gestionnaire d'adresse de lire
                                        // le registre vers le bus général.
write :
   DAUcom write,#this Multiple          // Demande l'écriture du registre à partir de la valeur lue sur
                                        // le bus général.

<aXX a0H|a1H|a0L|a1L
read :
   CUcom read,$this MultString      // Demande la lecture du registre appropriée
   CUout 2 Int                      // Met la valeur lue sur le bus général.

<imm #_NUM
read :
   GDP #this Int   // Met la donnée immédiate sur le bus général.
```

```
<aX a0|a1
write:
   CUcom   op1:=gdp  String                  // La valeur du bus général (DP) est envoyée vers l'ALU.
                                             // op :=gdp est un service de l'unité CU.
   CUop    Mov    CodeOp            // Met la première opérande de l'ALU sur sa sortie.
   CUcom   write_,$2      MultString         // Demande l'écriture de la sortie de l'ALU dans le registre.

<mov_src  _aXX_ | _imm_ | _rN_
<mov_dst  _rN_ | _aX_
```

Il est alors possible de définir le schéma d'instructions suivant :

```
>mov (mov_src,mov_dst) true
1.read                                   Lecture de la source
sleep
2.write                                  Ecriture de la destination
```

Ce schéma d'instructions permet de représenter six modes d'adressage pour l'instruction *mov*. Ainsi les instructions suivantes correspondent à ce schéma : *mov a0H, r5* ; *mov #12, a0* ; *mov r3, r0*.

En conclusion, le décodeur générique permet à l'utilisateur de SEP de prendre en compte rapidement le jeu d'instructions des architectures programmables qu'il modélise. Une telle approche permet une simulation mixte du jeu d'instructions et de l'architecture. Cette méthode permet de modéliser les mécanismes complexes, le haut-niveau de parallélisme dans les instructions et les interruptions matérielles. Le langage proposé permet d'utiliser la spécificité de SEP et en particulier l'utilisation des services de modules. Ce langage nous a permis de décrire le jeu d'instructions d'un DSP et d'un processeur RISC industriels en moins de deux semaines. Le lecteur peut se reporter au chapitre XI qui détaille la modélisation de ces architectures. Le code complet de description du jeu d'instructions de ces processeurs ainsi que la grammaire complète de SEP-ISDL sont disponibles en annexe.

Cette description des instructions sous forme d'ensemble d'actions permet une validation partielle des schémas d'instructions et offre une aide à la découverte du parallélisme inhérent à l'architecture et aux modes d'adressage potentiels. En effet, les schémas d'instructions peuvent être assimilés à des services de module, on bénéficie alors des mécanismes de validation présentés aux chapitres VII et VIII.

SEP permettrait aussi une description du décodeur d'instructions avec un langage réactif synchrone bien adapté à la description du contrôle. Ce dernier aspect est abordé dans le chapitre IX.

Une fois les performances souhaitées obtenues, un outil devrait être capable de générer une description sous la forme d'une machine d'états qui correspond au décodeur, ce n'est actuellement le cas dans SEP que si le concepteur utilise un langage synchrone pour la description du décodeur.