UNIVERSITE DE TUNIS EL MANAR

MEMOIRE

Présenté à

L'ECOLE NATIONALE D'INGENIEURS DE TUNIS

En vue de l'obtention du

DIPLOME D'ETUDES APPROFONDIES

En
Systèmes de Communications

Option : Communication Optique et Micro-ondes

Par

Ali HABIBI

Ingénieur en Télécommunications de SUP'COM

Thème :

VERIFICATION FORMELLE DES PROCESSEURS NUMERIQUES DU SIGNAL DE LA FAMILLE ADSP-2100 D'ANALOG DEVICES

Soutenu le 24 mai 2001 devant le jury composé de :

M^me^ Mariem JAIDANE	Présidente
M^me^ Monia TURKI	Membre
M. Adel GHAZEL	Membre
M. Sofiene TAHAR	Membre
M. Samir KALLEL	Invité

Dédicaces

A mes parents

et à MA

AVANT-PROPOS

Le travail de recherche présenté dans ce mémoire a été élaboré dans le cadre de la préparation d'un Diplôme d'Etudes Approfondies en Systèmes de Communications de l'Ecole Nationale d'Ingénieur de Tunis (ENIT) en collaboration avec l'Ecole Supérieure des Communications de Tunis (SUP'COM) et l'Ecole des Polytechniques de Tunisie (EPT). Le sujet de ce travail de recherche concerne la vérification formelle d'un processeur DSP (Digital Signal Processor).

Au terme de ce travail, nous tenons à remercier en particulier:

- Madame Meriam JAIDANE, Professeur à l'ENIT, pour l'honneur qu'elle nous fait en acceptant de présider le jury de notre DEA.

- Madame Monia TURKI, Maître-assistante à l'ENIT, pour avoir bien voulu faire partie du jury et juger notre travail de recherche.

- Monsieur Sofiène TAHAR, Professeur à l'université de Concordia au Canada, pour nous avoir offert l'occasion de travailler sur ce sujet et pour toute l'aide technique et morale qu'il nous a apporté tout au long de ce travail.

- Monsieur Adel GHAZEL, Maître-assistant à SUP'COM, pour son soutien et ses conseils précieux qui ont été d'un appui inestimable dans l'élaboration de ce travail.

Enfin, nous tenons à exprimer notre reconnaissance à toutes les personnes qui ont contribué de prés ou de loin à la réussite de ce travail de DEA.

RESUME

L'objectif de ce travail de recherche a été l'application de la vérification formelle, aux processeurs de la famille ADSP 2100, en utilisant la méthode du prouveur de théorème HOL (Higher Order Logic). Une approche structurée a été mise en œuvre en considérant l'architecture du DSP en tant que sous-unités. Après une spécification détaillée de l'architecture interne et de tout le jeu d'instructions machine une spécification formelle a été établie pour les différentes unités et sous unités du DSP. Les essais de l'approche de vérification formelle proposée dans ce travail ont montré sa capacité à traiter des systèmes complexes. De même les résultats obtenus ont permis de confirmer que le fonctionnement des processeurs testés répond bien aux spécifications du constructeur.

Mots clés : Vérification formelle, HOL (Higher Order Logic), prouveur de théorème, processeur DSP, composants ADSP-2100.

TABLE DES MATIERES

LISTE DES ILLUSTRATIONS

Liste des Figures

Liste des Tableaux

Chapitre 1 : INTRODUCTION GENERALE

De nos jours le monde des télécommunications s'est épanouie de façon spectaculaire pour toucher à toutes les activités de la vie quotidienne. Au départ, c'était le transfert de la voix qui s'est manifesté en premier objectif. Puis, une fois cet objectif dominé, d'autres ambitions sont devenues prédominantes. Mais, dans tout cela, la progression technologique affreuse était l'énergie de base servant pour force motrice simulant l'expansion de ce domaine.

La technologie en terme de semi-conducteurs avait offert à l'homme un vrai paradis d'idées et d'innovations. Les petites créatures à base de silicium étaient capables de réagir différemment comparées aux autres composants découverts ou créés jusque là. L'homme, de son côté, poussé par le désir de la création, était capable d'exploiter ce comportement pour en faire un vrai bâtonnet magique.

Si on se permet d'imaginer qu'un homme mort il y a un siècle s'est renaît aujourd'hui et on lui demande qu'est-ce qui a changé par rapport à son époque, il répondrait tout a changé ou presque. C'est qu'il ne connaissait ni le téléphone, ni la radio, ni la télévision, ni l'ordinateur,… Il ne connaissait, en fait, aucune créature qui contient un composant à base de silicium. Mais, est-ce que sa cervelle supporterait le fait qu'un circuit faisant à peine le bout de son doigt est capable de faire des opérations mathématiques aussi complexes qu'il le croit en quelques fractions de fractions de seconde ?

Malgré qu'il n'est pas facile de définir quel est le facteur le plus important dans la progression du domaine des télécommunications, il est trivial de qualifier la découverte et la création des *microprocesseurs* comme l'un des faits les plus fondamentaux. Ces circuits, qui découlent de la découverte des semi-conducteurs et qui représentent l'une des plus importantes créations humaines, ont détourné l'approche de l'homme dans sa manière de traiter ses problèmes, de concevoir ses solutions et d'implémenter ses systèmes.

Les microprocesseurs ont permis à l'homme de faire des opérations que sa vie tout entière n'était pas suffisante pour les accomplir. Ils sont capables d'illustrer l'intelligence humaine en terme de calcul, de comparaison et de décision de façon parfaite. De façon magique ou presque, ces composants se sont intégrés dans la majorité des productions humaines de ce siècle offrant une variété de fonctionnalités et une très grande souplesse de manipulation.

Ce qui a fait la gloire des microprocesseurs c'est leurs importantes capacités comparées à leurs tailles très réduites. D'un jour au lendemain on entend parler de millions de transistors intégrés sur une même puce faisant à peine quelques portions de pouce. Tout le monde se réjouit de joie en entendant de telles

informations, car voilà qu'on serait en mesure de calculer de façon encore plus rapide, de réduire les temps de conception, de parvenir aux meilleures solutions dans des durées minimes et d'optimiser nos vies.

Mais, la règle du jeu de la vie impose que si l'on gagne d'un côté c'est qu'il y a certainement un prix à payer de l'autre côté. En effet, si on cherche tous à maximiser les performances des microprocesseurs, on ne doit pas oublier que plus ils sont performants, plus leurs structures sont complexes. Il devient nécessaire donc de vérifier à chaque fois le bon fonctionnement d'un processeur avant de l'impliquer dans nos applications. L'objectif n'est plus uniquement de *concevoir* le composant mais plutôt de *vérifier* qu'il fonctionne correctement.

A première vue, on pourrait se contenter de la bonne conception du composant pour être sûr de son bon fonctionnement. Mais, comment peut-on être sûr que la conception est correcte pour un système impliquant des entrées et des sorties *très* variées. Le terme *très* dans ce cas représente des millions de millions de millions de cas possibles !

D'autre part, peut être que le fait que le volume de notre radio varie parfois brusquement ne nous gène pas, mais, est ce que le fonctionnement de la commande d'un avion qu'on est dedans ou encore la modification des chiffres de nos comptes bancaires pourraient être négligées. Pour ces faits et pour d'autres on ne peut pas se permettre d'utiliser un processeur dont on n'a pas vérifié le bon fonctionnement. C'est pourquoi toutes les grandes compagnies de production des processeurs se fanent par le fait qu'elles ont soumis leurs composants à des millions de tests et qu'il fonctionne parfaitement !

Mais, si on prend par exemple l'un des microprocesseurs commercialisés actuellement et on calcule le nombre de cas de tests nécessaires pour son évaluation, on se verrait devant un énorme nombre. Et les tests que l'on entend parler ne font qu'une fraction ne faisant pas la portion de la masse du microprocesseur comparée à celle de la terre. De ce fait, ces méthodes, aussi sophistiquées soient-elles, ne peuvent servir comme critère d'évaluation correcte pour la validité du bon fonctionnement du processeur.

On se demanderait dans ce cas sur l'existence d'une solution pour avoir une évaluation correcte du processeur. En conservant l'œil sur l'énorme nombre de cas possible, on pourrait céder l'affaire avant de s'y lancer. Mais, la solution existe toujours car pour mesurer la température du soleil on n'avait pas besoin de le toucher. Il suffirait parfois d'exploiter l'intelligence humaine pour en retrouver la bonne solution.

Ce sont ce que l'on appelle les méthodes formelles de vérification qui ont servi l'affaire pour résoudre ce problème. Ces méthodes permettent de prouver qu'un processeur fonctionne correctement et quelles que soient ces entrées et ces sorties. Elles permettent d'exploiter des notions mathématiques assez évoluées pour traiter tous les cas possibles et cette fois-ci dans des durées très réduites.

Ce mémoire est une illustration de l'utilisation de l'une des méthodes formelles de vérification, dite *vérification par proveur de théorème*, d'un microprocesseur produit pour des applications dans le domaine des télécommunications. Notre objectif est de prouver le bon fonctionnement ou non de ce composant pour toutes les entrées et les sorties possibles.

On a considéré le cas de la famille de microprocesseur ADSP-2100 de Analog Devices. Ce choix est tributaire de deux facteurs. En premier lieu, cette classe de processeurs est impliquée dans plusieurs domaines de télécommunications. En effet, ils sont utilisés dans des cartes sons, des récepteurs GPS et dans une très grande variété de composant.

D'autre part, notre choix est lié au fait qu'il n'y avait aucune étude de vérification par les méthodes formelles qui s'est intéressée à ce composant auparavant. Donc, c'est l'occasion pour faire preuve de la possibilité de l'utilisation de la méthode de vérification par prouveur de théorèmes (utilisant HOL dans ce cas) dans la vérification de cette classe de processeurs.

Dans ce mémoire nous allons commencer en premier par présenter le problème de vérification. Ainsi on présentera l'état de l'art des différentes méthodes de vérification. Puis, on décrira les principales méthodes formelles de vérification. Ensuite, on entamera le vif du sujet, d'une part, en spécifiant les différentes unités de la famille de processeurs ADSP-2100 et d'autre part, on présentera notre étude (en terme de vérification) de cette famille. Enfin, on terminera ce mémoire par une présentation des commentaires qu'on a pu tirer de cette étude et des perspectives qui en découlent.

Chapitre 2 : CONTEXTE DE L'ETUDE ET POSITION DU PROBLEME

Notre objectif est de vérifier le bon fonctionnement de la famille de microprocesseurs ADSP-2100. Mais, avant d'entamer l'étude proprement dite de ce composant, on débutera notre mémoire par ce chapitre introductif. Celui-ci décrira le contexte de notre étude et présentera une formulation détaillée du problème à qui on cherche la solution. On commencera par un aperçu sur l'importance de la vérification ; puis, on donnera un aperçu sur les méthodes formelles les plus utilisées ; ensuite, on justifiera les choix de l'étude des microprocesseurs de la famille ADSP-2100 ; et enfin, on listera les étapes qu'on a suivies dans notre étude.

2.1. Importance et utilité de la vérification

2.1.1. La progression technologique

Il y a une vingtaine d'années les recherches dans le domaine des circuits intégrés, et en particulier des microprocesseurs, étaient orientées vers la maximisation du nombre de portes par puce. C'était en fait, la voie qui mène vers l'augmentation des performances des composants que ce soit en terme de capacité ou de vitesse d'exécution.

De nos jours, on pourrait qualifier les études précédemment réalisées de réussites. En effet, on parle actuellement de millions de portes intégrées sur la même puce et de rapidité de microprocesseurs dépassant la frontière du 1 GHz. Mais, cela avait accentué l'effet d'un problème supposé jusque là minime qui est le bon fonctionnement de ces composants . C'est que plus nos systèmes sont grands en terme de fonctionnalités et de capacité, plus le nombre d'erreurs qu'ils peuvent engendrer est important [1] .

Si on énumère les résultats de la progression technologique vers ces dernières années on pourrait remarquer que les circuits intégrés sont devenus très complexes. En effet, on parle aujourd'hui de microprocesseurs contenant 5 millions de portes logiques, de circuits intégrés comportant plus que dix millions de portes logiques, de circuits dédiés pour les applications de

télécommunications dépassant le million de portes par puces et de bien une autre variété d'exemples.

2.1.2. La complexité conceptuelle et le bon fonctionnement

L'accroissement rapide du nombre de portes logiques par composant a permis aux concepteurs de ces circuits de produire des systèmes plus variés et plus complexes. Ceci avait imposé aux grandes compagnies de production de ces composants d'accorder une importance à la phase de vérification. La preuve en est dans le budget qu'avait consacré Intel Corporation pour les corrections de ces microprocesseurs Pentium et Pentium Pro et qui avait dépassé les 250 Millions $ à l'année 1994.

D'autre part, les dégâts que peut engendrer une erreur dans un microprocesseur sont d'un effet très considérable surtout lorsque celui-ci est utilisé dans des systèmes assez particuliers tel est le système de commande par exemple. C'est le cas, pour ce qui s'est passé à la fusée Ariane 6 à cause de la mauvaise spécification d'un des modules du microprocesseur de commande.

2.1.3. La vérification : Evaluation et Obligation

Une fois un composant produit, on cherche en premier lieu à vérifier son bon fonctionnement. C'est en effet, la phase d'évaluation qui essaye de déterminer s'il y a des lacunes dans la conception ou des mauvaises manipulations dans l'implémentation. Cette phase cherche à faire preuve de l'accord entre la spécification du composant et son fonctionnement réel. Pour cela, on essaye, dans la majorité des cas, de vérifier certaines configurations des entrées (dites configurations les plus probables) et on vérifie les valeurs des sorties

Mais la vérification n'est pas uniquement une évaluation mais aussi bien une obligation. Car, pour impliquer un composant dans un système assez complexe tel que les systèmes de commande à distance, il est nécessaire que ce composant soit parfaitement conforme à sa spécification. Car, des erreurs de fonctionnement dans de tels systèmes pourraient engendrer des pertes de valeurs exorbitantes.

2.2. Etat de l'art des applications de vérification

2.2.1. La vérification par simulation

C'est la méthode traditionnelle de vérification. Comme son nom l'indique, elle essaye de tester le bon fonctionnement d'un composant en le soumettant à un système réel d'évaluation. Cela revient à présenter un vecteur de valeurs à l'entrée du composant et à comparer le vecteur de sortie à ce que générerait un fonctionnement conforme à la spécification.

Mais, bien que la technique standard de vérification est le test direct (simulation), cette méthode permet d'afficher la présence de bugs mais n'est pas capable de prouver avec certitude leur absence.

> *Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.*
>
> **Edsgar W. Dijkstra**

Cette conclusion découle de plusieurs arguments dont figure principalement l'impossibilité d'énumérer toutes les entrées possibles. Considérant ces deux exemples simples :

Exemple1 :

Si on veut tester une routine de division ayant pour entrées des éléments à 64 bits à virgule flottante, il nous est nécessaire d'énumérer 2^{128} combinaisons. Si on suppose qu'un test prendra 1 μs, on aura besoin de 2^{25} années seulement !

Exemple2 :

Question :Combien demanderait le test d'une mémoire RAM de capacité 256 bits ?

Réponse : 2^{256} d'états d'entrées.

Supposons :

1. qu'on va utiliser toute la matière de notre galaxie pour produire des ordinateurs (10^{17} Kg),
2. que chaque ordinateur fera la taille d'un électron (10^{-30} Kg),
3. et que chaque ordinateur simulera 10^{12} cas par seconde.

Dans ce cas on dépassera, si tout va bien, la frontière de 0.05% des cas possibles dans 10^{10} années !

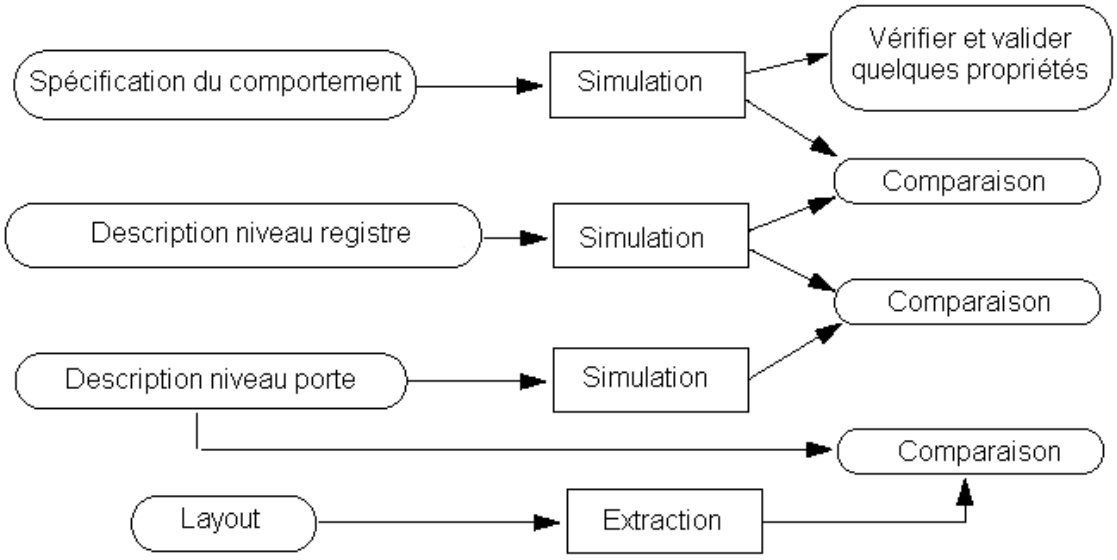


Figure 2.1. La vérification par simulation.

La vérification par simulation présente une simplicité conceptuelle mais possède beaucoup d'inconvénients, dont on cite :

La génération des séquences d'entrée :

La génération des séquences d'entrées nécessaires pour la vérification par simulation est l'une des tâches les plus complexes. Car, vu le nombre très limité des entrées qu'on peut tester comparé au nombre de cas possibles, il devint nécessaire de faire les bons choix. Mais, cela reste un objectif loin d'être facile à joindre.

En effet, on doit passer beaucoup de temps pour énumérer les cas qui pourraient engendrer des erreurs. Cela n'est pas évident surtout lorsqu'on prend en considération que la grande partie des erreurs concerne, dans la majorité des cas, des configurations assez particulières et qui ont échappé à l'œil du concepteur dans la phase conception.

La durée de la phase de vérification :

Pour maximiser la certitude des résultats obtenus par la vérification par simulation, il est nécessaire de considérer un très grand nombre de séquences

d'entrées. Mais, cela est un important inconvénient pour cette méthode, car il impliquerait une longue durée de la phase de test.

La difficulté de suivre le progrès technologique :

Plus les systèmes sont complexes plus le nombre de cas nécessaires pour les tester est important. En effet, plus un microprocesseur est riche en fonctionnalités, plus le nombre de ses combinaisons d'entrées/sorties est grand. Donc, cela implique à la vérification par simulation de tester une plus grande variété de séquences d'entrées. En conséquence, on aura plus de temps pour tester une proportion minime de cas de configurations possibles. Ainsi, cette méthode se verrait, dans son approche classique, incapable de suivre le rythme actuel du progrès technologique.

2.2.2. La vérification formelle

La vérification formelle [1] est un processus qui permet de prouver qu'un système se comporterait en parfait accord avec sa spécification. Cela revient à utiliser des prouveurs mathématiques qui permettent de démontrer que l'implémentation satisfait la spécification [5] . Dans ce cas une considération de tous les cas possibles est implicite.

Ces dernières années, les méthodes formelles ne sont plus uniquement des thèmes de recherche, mais plutôt un concurrent potentiel, si ce n'est pas une solution de remplacement, de la vérification par simulation [4, 11, 12]. En effet, ces méthodes permettent de considérer tous les cas possibles et de prouver si un système fonctionne correctement ou non [2] . De plus, en cas d'un mauvais fonctionnement, elles permettent aussi de déterminer toutes les erreurs possibles et d'énumérer leurs origines.

> *"As designs grow ever more complex, formal verifiers have left the research lab for the production arena."*
> *"Formal methods have already proven themselves, and have a bright future in electronic design automation."*
>
> **[IEEE Spectrum, January 1996]**

Ces points de vue partent d'une correcte énumération des caractéristiques des méthodes formelles. Les avantages des méthodes de vérification formelle comparées à la vérification par simulation sont nombreux. En effet, ces méthodes :

1. considèrent toutes les entrées possibles au système.
2. vérifient la validité des propriétés du système mathématiquement.
3. ne nécessitent pas une spécification des sorties du système prévues.
4. Permettent, pour certains outils, d'identifier les traces des erreurs s'il y a lieu.
5. Traitent, pour certains outils, tous les cas possibles.

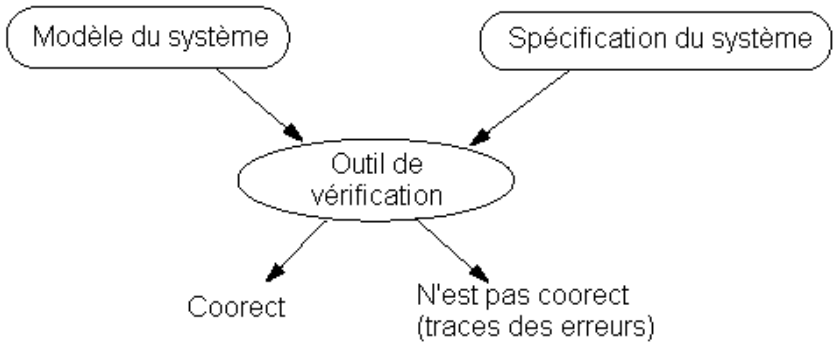


Figure 22. La vérification formelle.

La vérification formelle possède, elle aussi, certains inconvénients [1] . En effet, elle demande un effort supplémentaire pour parvenir à une description complète et simple du système à vérifier. C'est qu'il est nécessaire de définir une spécification, d'une part, tenant en considération tous les détails du système, et d'autre part, assez simple à manipuler dans la phase de vérification.

2.2.3. Comparaison de la simulation et de la vérification formelle

La représentation du système

L'avantage fondamental de la simulation est qu'elle considère un système réel. Ce point le distingue de la vérification formelle qui prend en considération une représentation d'un modèle de ce système.

La vérification

La simulation ne considère qu'une certaine portion des entrées et des configurations possibles du système à vérifier. Par contre, la vérification formelle vérifie le système en considérant toutes les entrées et toues les configurations possibles.

Trace des erreurs

Par simulation, il est possible de déterminer les origines des erreurs. Par contre, par vérification formelle, en dépit de la détermination de l'existence d'erreurs, il est possible, pour certains outils, d'identifier leurs origines.

La complémentarité

Les deux classes de méthodes simulation et vérification formelle sont complémentaires [1] . Il faut, en effet, bénéficier des avantages de ces méthodes. Cela est réalisé en considérant une vérification du système par les méthodes formelles ayant pour objectif d'identifier les erreurs au niveau du système. Ensuite, une fois une erreur identifiée, on procède par la simulation pour vérifier si cette erreur existe. Si c'est le cas, on utilise la vérification formelle de nouveau pour déterminer les traces de l'erreur au niveau de l'implantation.

"Simulation and formal verification have to play together."

[IEEE Spectrum, January 1996]

2.3. Objectifs et motivations de l'étude d'un DSP

Notre choix d'étudier une famille de microprocesseurs DSP est lié à plusieurs facteurs dont figurent principalement : la grande utilisation de cette classe de microprocesseurs dans le domaine des télécommunications [24] , leurs caractéristiques particulières et l'application de la vérification formelle pour leur vérification.

2.3.1. Architecture de la famille ADSP-2100

La famille de microprocesseurs ADSP-2100 [21] se distingue par son architecture interne [22] , Figure 2.3. En effet, chacun des processeurs de cette famille est composé de trois unités indépendantes : l'ALU (unité arithmétique et

logique), le MAC (l'accumulateur/multiplieur) et le Shiffter (barrel shifter) [21, 23, 24]. Chacune de ces différentes unités traite des données de 16 bits .

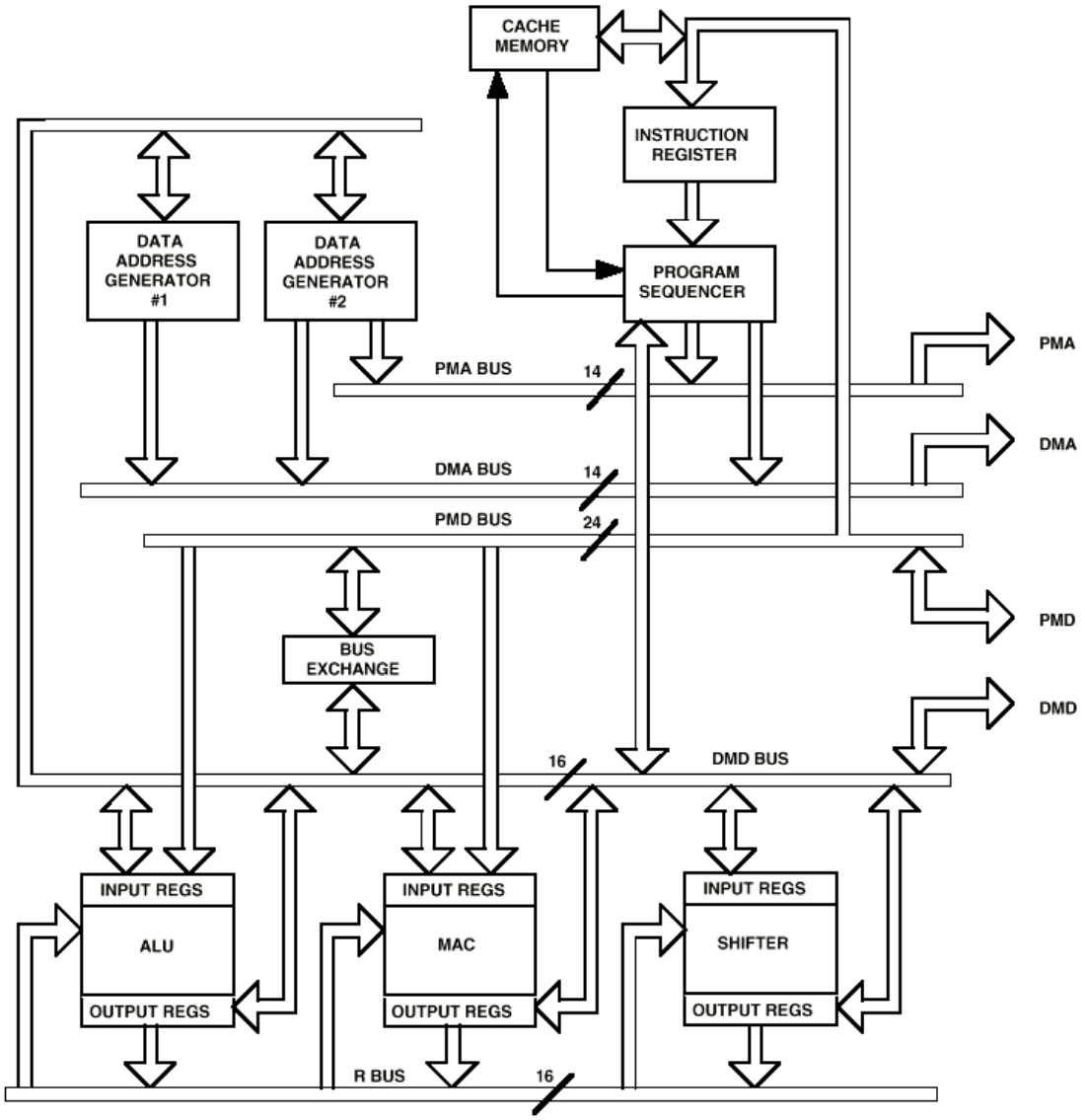


Figure 2.3. Architecture interne de la famille ADSP-2100 [21] .

Un processeur de la famille ADSP-2100 contient deux générateurs d'adresses ce qui permet au séquenceur de programme d'exécuter en même temps deux instructions *fetch* par exemple. D'autres part, ces deux générateurs d'adresses et le séquenceur de programme permettent une exploitation très efficace des différentes unités du processeur en offrant une très grandes variétés de multi-instructions.

L'architecture de base de la famille ADSP-2100 est une architecture *Harvard modifiée* [22] . En effet, les données sont stockées à la fois dans la mémoire programme et dans la mémoire données. Cette propriété est d'une très grande importance car en stockant les données dans la mémoire programme, il est possible de chercher deux opérandes dans le même cycle.

Le transfert de données entre les différentes unités du processeur est assuré par 5 bus qui sont :

1. Bus PMA (Program Memory Address) : bus servant au transfert des adresses à la mémoire programme.
2. Bus PMD (Program Memory Data) : bus servant au transfert des données à/de la mémoire programme.
3. Bus DMA (Data Memory Address) : bus servant au transfert des adresses à la mémoire données.
4. Bus DMD : bus servant au transfert des données à/de la mémoire données.
5. Bus R : bus d'interconnexion entre les différentes unités de calcul du processeur.

Les instructions sont véhiculées de la mémoire programme vers le registre d'instruction à partir de la mémoire externe et ce via le bus PMD. Une instruction est recherchée et chargée dans le registre d'instructions dans un cycle d'horloge ; et est exécutée dans le cycle suivant. Ainsi, le registre d'instruction introduit un seul niveau de pipeline du flux de programme.

Chaque unité contient un ensemble de registres d'entrées et de sorties. Généralement chaque unité prend ses entrées de l'un des registres d'entrée et inscrit le résultat dans l'un des registres de sorties. Néanmoins, il est possible d'accéder directement aux deux bus de données PMD et DMD. Les registres se présentent comme une interface entre les unités du processeur et la mémoire externe introduisant un niveau de pipeline d'ordre 1 au niveau de la sortie.

Les processeurs ADSP-2100 sont dotés d'une mémoire cache interne. Celle-ci est composée de 16 mots (word) et permet d'exécuter à la fois une instruction de recherche (*fetch*) dans la mémoire données et une autre dans la mémoire programme. Cette propriété est d'une très grande importance surtout lorsque les programmes exécutés sont de taille inférieure à 16 mots machines. Car dans ce cas tout l'algorithme sera totalement chargé dans la mémoire cache éliminant ainsi toute forme d'accès à la mémoire externe.

2.3.2. Exemples d'application à base des processeurs ADSP-2100

L'architecture des processeurs ADSP-2100 les a permis d'être utilisés dans plusieurs domaines. En effet, ils ont été exploités dans plusieurs systèmes et circuits de communication comme le décrit bien le Tableau 2.1.

Ce tableau illustre quelques-unes des applications et des systèmes commercialisés et qui intègrent des processeurs de la famille ADSP-2100. On note bien qu'il y a une très grande variété d'utilisation de cette famille ce qui fait preuve de ses capacités et ses performances.

Producteur de l'application	***Processeurs utilisés***	***Applications***		***Description***
Momentum Data Systems, Inc. (http://www.mds.com/products/analog_devices.htm)	ADSP-2189	Hummingbird-89		Carte audio PCMCIA.
	ADSP-2189	Iguana-89		Une carte d'évaluation utilisée pour les applications de débogage.
	ADSP-2189	Hawk-89		Carte audio PCI.
	ADSP-2181	Hawk-81™		Une carte ISA.
	ADSP-218x	MDS-218X-ICE		Un émulateur de circuits pour la famille ADSP-218x.
Optimal Engineering Systems, Inc (http://www.oesincorp.com)	ADSP-21xx	Software	48-Bit Precision Floating-Point Arithmetic Library	Une librairie de calcul arithmétique.
			Motion Control Algorithms	Un algorithme de suivi des mouvements.
	ADSP-21xx	Hardware	MCDAC	Une carte à base d'une architecture de deux processeurs.
			UMCS	Un système de contrôle de suivi de mouvements.
Ensigma Ltd. (http://www.ensigma.com)	ADSP-21xx	Algorithme G.729A. Algorithme Dolby AC-3.		Ce sont des applications qui concernent des codeurs audio, la communication sans fil, et de la téléphonie.
GAO Research & Consulting Ltd. (http://www.gaoresearch.com)	ADSP-21xx	Modem, fax et téléphonie.		V.90; ADSL; Integrated Voice, Fax, et la transmission de la voix sur IP.
Jasmin Infotech Pvt. Ltd	ADSP-218x et ADSP-21065L	MPEG Encode/Decode, MP3 Decode		Des algorithmes de traitement de la parole pour des application multimédia.
EDevice Technology (http://www.edevice.com)	ADSP-218x	Algorithmes : PPP, TCP, IP, SMTP, POP3, HTTP, FTP et Telnet. Produits Hardware : carte de référence IP. Produit software : smartstack.		Ces applications concernent l'intégration du stack IP sur un DSP.
Bayer Digitale (http://www.dsp-bayer.com/english/collab.htm)	ADSP2189L	AD89-PCI		Carte PCI intégrant des fonctionnalités de traitement de signal et d'applications audio.
DSI-Decision Systems Israel Ltd. (http://www.dsi.co.il)	ADSP-2181	DSI/PC104/DSP/COM DSI/2181/sampler		Des ordinateurs à base de circuits DSP.

Tableau 2.1. Les applications de la famille ADSP-2100.

2.3.3. Approches de vérification

La complexité d'un DSP réside dans le fait que son architecture supporte une très grande variété d'opérations et une répartition des données à la fois entre la mémoire données et la mémoire programme. Cela demande donc plus d'efforts pour pouvoir spécifier de façon complète toutes les caractéristiques d'un DSP dans une formule simple à exploiter.

L'une des particularités d'un DSP est aussi son jeu d'instructions qui autorise une variété de manipulation des différentes unités du microprocesseur à la fois. Il est possible par exemple de faire à la fois un accès pour lecture à la mémoire données, un accès pour écriture à la mémoire programme et une instruction à l'ALU.

Dans les études qui ont été réalisées sur les microprocesseurs DSP, seules la simulation a été utilisée. De ce fait, notre approche d'utiliser la vérification formelle est une première. En effet, on estime que bien que l'architecture de cette classe est différente des autres classes de microprocesseurs, il est possible d'appliquer l'approche formelle pour leur vérification.

Notre conviction que la vérification formelle est capable de traiter le cas des microprocesseurs DSP est liée au fait que l'approche formelle n'impose aucune contrainte sur le système à vérifier. Seulement, on aura besoin d'une représentation à la fois complète et simple du système étudié.

A première vue l'architecture d'un DSP pourrait paraître assez complexe mais une compréhension détaillée de ce composant permettrait de bien manipuler sa vérification. En effet, ce composant est composé de plusieurs blocs qui pourrait être étudié chacun à part. Ainsi, on pourrait décomposer le problème et s'attaquer aux unités une à une.

2.4. La structure du rapport et la démarche de l'étude

Dans ce rapport on propose la démarche décrite dans les paragraphes suivant :

2.4.1. La structure du rapport

Présentation des méthodes formelles

Eu premier lieu, on va commencer par présenter les méthodes formelles. Cela nous permettra de donner un aperçu sur ces méthodes et d'expliquer leurs approches.

Spécification de la famille ADSP 2100

Ce chapitre présenterait la première phase de l'étude de la famille ADSP-2100. Celle-ci concerne la spécification de cette famille. Cette spécification servira dans la phase suivante de notre étude.

Vérification de la famille ADSP 2100

C'est la deuxième phase de notre étude. Elle mettrait en évidence l'utilisation de la vérification formelle pour le cas de la famille ADSP-2100.

2.4.2. La démarche de l'étude

Notre démarche est directement liée à la structure du DSP. En premier lieu, on va commencer par spécifier les différentes unités du processeur. Cela nous permettrait d'alléger les représentations des structures de ces unités. Ainsi, on aura une représentation assez simple de tout le processeur qui servira dans la phase de vérification.

La deuxième phase concernera la vérification des instructions. Notre approche est de considérer les instructions de la famille ADSP-2100 une à une. Cela nous permettrait de s'attaquer sous-ensemble des unités du processeur concerné par l'instruction à vérifier. De cette manière on sera amener à chaque fois à traiter une représentation plus allégée du processeur.

2.5. Conclusion

Dans ce chapitre introductif, on a présenté nos arguments concernant l'étude d'un microprocesseur DSP par une méthode formelle de vérification. Ensuite, on a donné un aperçu sur les différentes méthodes de vérification. Et enfin, on a présenté la structure de ce rapport et la démarche de notre étude.

Chapitre 3 : CONCEPTS ET OUTILS DE LA VERIFICATION FORMELLE

La vérification formelle a débuté il y a une trentaine d'années mais il a fallu une assez longue durée pour qu'elle puisse retrouver sa gloire et son état actuel. La complexité de plus en plus importante des microprocesseurs et l'incapacité de la simulation de traiter toutes les configurations possibles ont joué un rôle fondamental dans l'accroissement de l'importance des méthodes formelles. Dans ce chapitre, nous mettons le point sur l'état de l'art de ces méthodes et en particulier sur la vérification par le prouveur de théorèmes HOL qui sera notre outil pour vérifier la famille de microprocesseurs ADSP-2100.

3.1. La vérification fonctionnelle

3.1.1. Etat de l'art

De point de vue historique, les racines des méthodes formelles, surtout pour ce qui concerne la vérification software, datent des années 60 [1] . En effet, à cette époque là un intérêt particulier a été accordé à cette nouvelle approche. Mais, au départ, le démarrage n'était pas très bien réussi. Plusieurs facteurs ont joué contre l'expansion des ces méthodes.

Au départ, l'utilisation des ces méthodes n'était pas pratique du fait des l'influence de plusieurs facteurs dont on site :

1. Les notations n'étaient pas claires et simples à manipuler donc demandaient un effort supplémentaire lors de l'étude ce qui risque de trop complexifier le problème.
2. Les techniques de vérification étaient loin d'être capable de traiter les cas réels qui concernent des systèmes de tailles très importantes.
3. Les outils utilisés n'étaient pas adaptés aux configurations étudiées ou bien encore trop complexes pour être utilisés.
4. Seules quelques études triviales ont étaient réalisées. Ce qui demande à l'utilisateur un effort supplémentaire pour étudier chaque cas et lui imposer de commencer à chaque fois à zéro.

5. Un nombre très limité de personnes ont le bagage nécessaire pour pouvoir s'investir dans ce sujet.

Mais, malgré tous ces points la vérification formelle était très prometteuse pour le domaine de la *vérification hardware* [2] . Cela découlait en effet de plusieurs facteurs dont on site :

1. La structure des composants hardware est hiérarchique et régulière. C'est que toute entité est composée d'un ensemble de sous-unités.
2. La réutilisation de la même architecture ou d'une architecture similaire est très courant dans le domaine hardware. Donc, il est possible de conserver les mêmes preuves pour des nouveaux systèmes en appliquant quelques modifications minimes.
3. La spécification hardware est plus structurée et commune comparée à la spécification software.
4. Les primitives utilisées sont simples. En effet, représenter une porte *NAND* est beaucoup plus simple que s'attaquer à une boucle *while*, par exemple.
5. Le prix d'erreurs dans le design pour ce qui concerne les composants hardware, et surtout les microprocesseurs, peut coûter des retards pouvant atteindre 6 mois et des pertes dépassant les millions de dollars surtout une fois les masques de production réalisés.

Ces facteurs ont joué en faveur de la vérification formelle et l'ont permis de regagner petit à petit une place plus importante. C'est pourquoi actuellement à cette approche est associée une image prodigieuse. En effet, son succès dans le traitement de plusieurs cas, tel que la vérification de protocoles, lui a permis de s'imposer comme l'une des plus importante et plus prometteuse méthode actuellement et surtout dans l'avenir [4] .

L'un des facteurs qui fait preuve de l'importance de l'approche formelle est le fait qu'elle est actuellement petit-à-petit intégrée dans le domaine industriel. Ce qui a fait preuve de sa capacité de s'attaquer à des systèmes réels. D'autre part, l'intérêt que lui portent des groupes de vérification à l'échelle mondiale, tel que IBM, Motorola, Intel etc., y présente un très important renfort.

3.1.2. L'approche formelle

La vérification formelle touche à plusieurs domaines à la fois : la logique, la conception des systèmes, les notions mathématiques, les applications en télécommunications etc. Mais, elle ne demande que deux principales notions : un langage formel et un outil de calcul déductif.

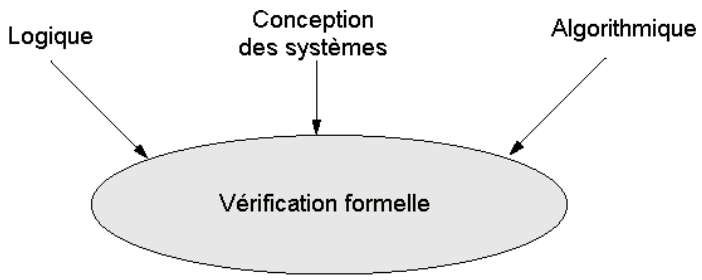


Figure 31. La vérification formelle : un domaine interdisciplinaire.

Le mot formel affecté à cette classe de méthode signifie que celles-ci considèrent des lois formées selon une syntaxe prédéfinie et non pas selon une sémantique. De ce fait la majorité de ces méthodes est basée sur une logique formelle. Une telle logique comprend un langage servant pour définir des phrases syntaxiques décrivant les objets à étudier et les manipulations de leurs propriétés.

Pour le cadre de la vérification hardware, une méthode formelle décrit la spécification ainsi l'implémentation du système à étudier selon la syntaxe d'un langage formel. Puis essaye de faire la preuve de l'implication entre l'implémentation et la spécification. Cela est assuré en utilisant des transformations décrites elles aussi selon la syntaxe du même langage formel.

Les logiques utilisées par les méthodes formelles pourraient être classifiées en trois catégories :

1. La logique du premier ordre : c'est l'algèbre des booléens {0,1}.
2. La logique du second ordre : c'est la logique des prédicats intégrant les quantificateurs d'existence et de généralisation.
3. La logique d'ordre supérieur ordre : rajoute à la logique d'ordre deux la notions de sous-ensembles de quantification sur des fonctions et des prédicats [3] .

L'inconvénient fondamental de l'approche formelle est qu'elle ne garantie pas le fonctionnement du système prouvé correct. Cela signifie que si on fait la preuve qu'un système est correct cela n'impose pas qu'il va dans le monde fonctionner. Ce point de divergence est du à deux principaux facteurs :

1. La conformité entre la spécification et les intentions du concepteur : la spécification doit être à la fois simple et complète ce qui n'est pas une tâche triviale à assurée. Ce qui peut conduire à des approximations ou simplifications du système afin d'alléger son étude.
2. La conformité du comportement de l'implémentation dans le monde réel avec le modèle : Le cadre réel peut dans plusieurs cas rajouter des effets inattendus tel est le cas si les connexions sont assez rapprochées par exemple. C'est phénomène ne sont pas pris en considération dans la

majorité des cas dans le modèle de l'implémentation ce qui présente un des inconvénients des méthodes formelles.

Sous le toit de la vérification formelle se réunissent plusieurs méthodes, mais, on peut considérer qu'il y a deux grandes classes : les méthodes à base des automates à états finis et les méthodes à base des prouveurs de théorèmes. Dans la première classe on va présenter principalement la vérification de modèle. Par contre, dans la deuxième on va s'intéresser à la méthode basée sur l'outil HOL qui nous servira dans l'étude de la famille de processeurs ADSP-2100 sujet de ce projet.

3.1.3. La vérification de modèle

La vérification de modèle est l'une des méthodes qui se classifie sous le cadre des méthodes basées sur les automates à états finis [8, 9]. Elle représente le système étudié sous forme d'automate à états finis. Puis, elle essaye de vérifier si une propriété, décrite dans une formule de logique temporelle, est correcte ou non. Cette phase de vérification est assurée vérificateur de modèle.

Cette classe de méthodes vérifie si une propriété est correcte en assurant une recherche exhaustive des états possibles du système. Mais, cela présente un très grand inconvénient. En effet, cette recherche peut engendrer un phénomène dit *explosion des états*. Cela induit une impossibilité de traiter des systèmes assez complexes tel est le cas des microprocesseurs ADSP-2100 par exemple.

3.1.4. La vérification par prouveur de théorèmes

La vérification par prouveur de théorème fait la preuve qu'une implémentation satisfait une spécification au moyen d'un outil de raisonnement mathématique [5] . Cela est illustré par la Figure 3.2. Dans ce cas, la spécification ainsi que l'implémentation du système sont décrit sous forme de formules logiques et la relation (l'équivalence ou l'implication) entre eux est décrite sous forme d'un théorème qui représente l'objectif à prouver.

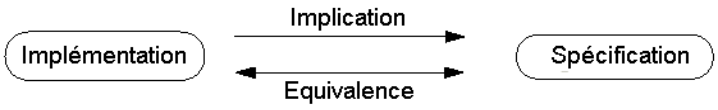


Figure 3.2. La vérification par prouveur de théorème.

Sous le volet de prouveur de théorème sont classifiées plusieurs outils dont on site : Boyer- Moore/ ACL2 (premier ordre logique), HOL (ordre logique supérieur) [14] , PVS (ordre logique supérieur) et Lambda (ordre logique supérieur). Ces systèmes partagent les mêmes avantages ainsi que les mêmes inconvénients. En effet, ils permettent de représentation expressive du système assurant le traitement d'une très grandes variétés de systèmes. Par contre, ils demandant des preuves interactives imposant l'intervention de l'utilisateur pour guider la preuve.

Dans notre projet nous avons opté pour l'outil HOL qui se distingue par sa grande capacité de traiter une très grande variété de systèmes assez complexes. Ce qui le renforce encore est qu'il est utilisé actuellement par plusieurs groupes de recherches ce qui lui permet d'acquérir de plus en plus des bibliothèques renforçant ses capacités et ces performances [11] . Une plus ample description de cet outil est décrite dans le paragraphe suivant.

3.2. L'outil HOL

3.2.1. Le langage ML

ML est un méta-langage fonctionnel [18, 19, 20] qui a était utilisé dans plusieurs universités et particulièrement à Cambridge University, Edinburgh University et INRIA pour implémenter des prouveurs de théorèmes. Le système HOL [13] , en particulier, est écrit dans langage ML. Les preuves de HOL sont des programmes ML et, par conséquent, sont conformes à la syntaxe de ML.

Il existe plusieurs raisons derrière l'utilisation de ML comme langage de rédaction de prouveurs de théorèmes. La première est qu'il possède un système de ***type*** à la fois puissant et flexible. En effet, le type est utilisé lors de la phase de compilation pour vérifier si les définitions sont bien formées. Par exemple, dans l'implémentation du système HOL on définit le type *thm* (théorème) et le compilateur ML utilise ces performances en terme d'identification de type pour vérifier si un argument d'une fonction ou un résultat de retour d'une fonction est supposé être de type *thm* , alors il l'est.

D'autre part, ML intègre la notion de types abstraits qui est utilisée pour grouper ensemble les types de données (*datatype*) et les opérations définis sur eux. Ces types abstraits de données sont écrits de la manière que l'accès direct à leurs constructeurs n'est pas possible de façon directe. Les instances de ces types données pourraient être crées et accédées seulement par l'intermédiaire de fonctions spécifiques définies dans la définition abstraite. HOL exploite cette propriété et définit les théorèmes comme étant un type de données abstrait et

offre un seul mécanisme pour les accéder (qui est définit sous le nom de : *inference rule* ou règle d'inférence).

Enfin, ML supporte et encourage l'utilisation de fonctions comme paramètres d'autres fonctions. Grâce à cette propriété, les commandes ML dans les preuves HOL peuvent être combinées dans des commandes plus complexes et plus puissantes.

3.2.2. Le système HOL

Le prouveur de théorème HOL est un outil qui a été développé à l'université de Cambridge [13] . C'est un système basé sur une logique expressive et générale qui permet d'avoir une formulation correcte et pratique. Ce système est basé sur le prouveur de théorèmes LCF et hérite plusieurs de ces concepts. A ce jour il existe trois versions du système HOL. La première version (HOL88) développée sous ML [14], la deuxième a été (HOL90) développée sous SML [18] et la troisième version (HOL98) qu'on va utiliser tout au long de notre étude.

3.2.2.1. La logique HOL

La logique de HOL est une variété de la logique d'ordre supérieur [14] basée sur une formulation de la théorie simple des types de Church. Elle se présente comme une extension de la logique des prédicats parce que :

1. Les variables peuvent être instanciées par des fonctions, et les arguments des fonctions peuvent être des fonctions.
2. Les fonctions peuvent être représentées par des λ-abstractions.
3. Chaque terme possède un type qui peut être polymorphe (au sens du langage SML).

Les notations utilisées dans la logique du système HOL sont présentées dans le Tableau 3.1.

Terme	***Notation HOL***	***Notation Standard***	***Description***
Vrai	T	T	Vrai
Faux	F	$\bot$	Faux
Négation	~P	$\neg$P	Non P
Disjonction	P \/ Q	P $\vee$ Q	P ou Q
Conjonction	P /\ Q	P $\wedge$ Q	P et Q
Implication	P ==>Q	P $\Rightarrow$ Q	P implique Q
Egalité	P = Q	P = Q	P égale Q
Quantification ($\forall$)	!x. P	$\forall$x. P	Pour tous x : P
Quantification ($\exists$)	?x. P	$\exists$x. P	Il existe x : P
Terme (ε)	x. P	εx. P	Un x tel que P
Conditionnelle	P => Q | R	(P => Q, R)	Si P alors Q sinon R

Tableau 3.1. Les notations de HOL [15] .

3.2.2.2. Le prouveur HOL

Le système HOL est le résultat du codage de la logique décrite précédemment dans le langage fonctionnel ML [15, 16, 17]. Les termes sont représentés comme des objets de ML de type *term*. Le langage ML permet de manipuler ces termes et de contrôler leur bonne formation au moyen d'un algorithme d'inférence de type. Un terme de la logique HOL doit être présenté au système entre guillemets. Si ML lui attribue le terme *term*, alors cela veut dire qu'il est bien formé. Les types de la logique HOL sont codés comme des objets de type *type*, en les faisant précéder de deux points verticales.

La fonction principale du système HOL est de montrer que certains termes de la logique HOL sont des théorèmes. Les théorèmes prouvés dans le système sont des objets d'un autre type du langage ML appelé *thm*. Un théorème est représenté par un ensemble de termes qu'on appelle *hypothèses* et un terme qu'on appelle *conclusion*. Etant donné un ensemble d'hypothèses Γ et une conclusion t, on note le théorème correspondant par Γ |- t. Si Γ est vide alors le théorème était noté tout simplement |- t.

Les théorèmes sont introduits dans le système HOL soit en postulant comme des axiomes, soit en les déduisant des théorèmes existants par des règles d'inférences décrites dans le métalangage ML et qu'on appelle *tactique*. La preuve d'un théorème est une succession de règles d'inférences appliquées aux axiomes ou aux théorèmes déjà prouvés. Les règles d'inférences vérifient que la déduction de la conclusion d'un théorème de ses hypothèses est conforme aux règles logiques de HOL. Le noyau du système HOL est constitué de cinq axiomes et de huit règles d'inférences primitives à partir desquelles toutes les autres règles de la logique sont dérivées.

Le résultat d'une session HOL est un objet appelé *théorie*. Cet objet constitue un ensemble de types, de constantes, d'axiomes, de définitions et de théorèmes. Le système fournit des possibilités d'étendre des théories existantes et de former des hiérarchies de théories. Si des résultats d'autres théories doivent être utilisés dans la théorie en question, on doit alors déclarer ces théories comme parents de la *théorie* qu'on développe. Les théories permettent une structuration des faits.

Un autre concept très utile est celui de *librairie*. Une *librairie* est une collection de théories, de théorèmes, de tactiques et de fonctions ML. Elle n'est pas nécessairement chargée lors du lancement du système HOL, néanmoins elle peut être chargée dynamiquement lors d'une session avec HOL.

3.2.2.3. La construction des preuves

Dans un très grand nombre de cas, et en particulier pour ce qui concerne la vérification hardware, les preuves sont construites par une approche dirigée vers le but. L'idée est de développer la preuve en commençant par le résultat désiré (but) et de le réduire à un certain nombre de sous buts plus simples à résoudre au moyen de programmes ML qu'on appelle des tactiques et dont le concept est dû à *Milner*. Le système gère cette situation au moyen d'une pile (*subgoal stack*) qui peut être manipuler interactivement. Ce genre d'outil est dû à *Paulson*.

Etant donné un but $\mathbf{G}$|-t où $\mathbf{G}$ est une liste d'hypothèses et t un terme à prouver. On commence par initialiser la pile à ce but au moyen d'une fonction spécifique puis on applique des tactiques adéquates pour se ramener à résoudre des sous buts dont la preuve est soit évidente soit plus simple à trouver. Le système permet de nous fournir la preuve désirée en sauvegardant à chaque étape la justification par laquelle on a accomplit la transformation d'un but en un ensemble des sous buts. Cette façon de trouver la preuve réincarne le vieux concept : ***diviser pour régner***.

Le système HOL fournit aussi un autre type de fonctions appelé *conversion* [24]. Ce type de fonction met en correspondance un terme t avec un autre terme u au moyen du théorème |-t = u. Cet outil s'avère très utile dans la définition de stratégies de simplification de termes compliqués. Souvent, la définition de stratégies est basée sur des conversions plus simples qui combinent par des opérateurs définis dans le système HOL.

3.2.3. Un simple exemple

L'objectif de cet exemple est de répondre aux deux questions suivantes :

1. Comment peut-on spécifier un système hardware utilisant une logique d'ordre supérieur ?
2. Comment peut-on vérifier le système ainsi spécifier ?

3.2.3.1. La spécification du hardware

Nous considérons comme exemple d'étude du circuit décrit par la Figure 3.3.

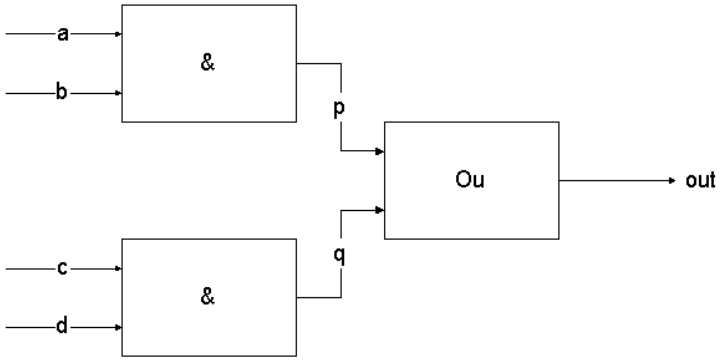


Figure 3.3. Exemple d'étude.

On veut spécifier le comportement interne de circuit ainsi que sa structure de telle façon que les deux peuvent être comparées. Il est possible d'assurer cela au moyen de plusieurs méthodes. Mais, dans ce cas on va présenter le comportement et la structure comme des prédicats et des relations entre les entrées et les sorties.

L'idée de base est que les prédicats forment une contrainte entre les entrées et les sorties. Cette n'est vrai sauf si la relation entre les entrées et les sorties est correcte.

Le terme de logique d'ordre supérieur suivant peut être utilisé pour spécifier la relation entre les entrées (a, b, c et d) et les sorties (out) :

```
system_spec = (`system_spec(a, b, c, d, out) =
(! t : num.
( out t = ((a t) /\ (b t)) \/ ((c t) /\ (d t)) )
)
`);
```

Cette définition pour paraître un peu étrange à première vue. En effet, on pourrait ne pas la considérer pour la définition d'une fonction. Mais, on doit la considérer comme un prédicat lien les entrées aux sorties.

Comme l'illustre la Figure 3.3 le circuit possède quatre entrées, une sortie et deux lignes internes (p et q). L'implémentation utilise deux portes AND et une porte OR. En considérant deux prédicats décrivants les deux portent AND et OR, il devient possible de créer la spécification de l'implémentation. On va par exemple considérer les deux spécifications suivantes pour les deux portes AND et OR :

```
AND = (`AND(a, b, c) =
(! t: num.
(c t = ( (a t) /\ (b t) ) )
)`);

OR = (`OR(a, b, c) =
(! t: num.
(c t = ( (a t) \/ (b t) ) )
)`);
```

Ce qu'on veut est de représenter l'implémentation par un prédicat reliant les entrées aux sorties. On doit prendre en considération les liaisons physiques, pour la sortie de la première AND doit être p. Donc cette porte sera représentée par AND(a,b,p). Le système de son tour sera représenté de la manière suivante :

AND(a,b,p) ∧
AND(c,d,q) ∧
OR(p,q,out)

C'est un prédicat clair qui met sous contrainte les entrées et les sorties selon la répartition des différents composants formant le système. Les signaux p et q sont déclarés comme variables avec une quantification existentielle. La définition de la spécification de l'implémentation est donc :

```
system_imp = (`system_imp(a, b, c, d, out) =
(? p q.
(AND(a,b,p) /\ AND(c,d,q) / \ OR(p,q,out))
)
`);
```

3.2.3.2. La vérification du hardware

La tâche de vérification revient à comparer les deux descriptions du circuit. Dans ce cas on doit identifier ce qu'on veut vérifier est-ce une propriété particulière ou bien une description plus détaillée du comportement prévu du système. Les deux questions auxquelles on s'affronte maintenant sont :

1. Qu'allons e nous comparer ?
2. Quel type de comparaison doit-on faire ?

Concernant la première question, la réponse est qu'on va comparer les deux représentations du système à savoir la spécification et la description de l'implémentation.

La deuxième question possède une réponse plus complexe. Ce qu'on veut dans le cas idéal est de prouver l'équivalence entre les deux représentations du système. Cela pourrait être décrit en HOL par la relation suivante :

```
g `! a b c d out. system_imp(a,b,c,d,out) = system_spec(a,b,c,d,out)`;
```

Pour plusieurs circuits, la preuve de l'équivalence n'est pas toujours possibles à cause des abstractions introduites au circuit pour le spécifier. Dans ces cas on se limite uniquement à l'implication entre l'implémentation et la spécification. Cela pourrait être décrit en langage HOL :

```
g `! a b c d out. system_imp(a,b,c,d,out) ==> system_spec(a,b,c,d,out)`;
```

Cet objectif (*goal* en terme de HOL) doit être simplifié est prouvé en utilisant des tactiques et/ou des conversions de HOL. On doit tout d'abord commencé par éclater l'objectif et cela en remplaçant leurs définitions. Cela peut être obtenu en utilisant la commande suivante :

```
REWRITE_TAC [system_imp, system_spec, AND, OR]
```

Dans un second lieu, on va essayer de simplifier les deux parties de l'implication jusqu'à aboutir à une formule triviale ou bien simple pour prouver au moyen des simplifications existantes dans le système. On peut utiliser la commande HOL suivante pour assurer cela :

```
e(
REPEAT GEN_TAC
THEN REPEAT ( (CONV_TAC LEFT_IMP_EXISTS_CONV)
THEN REPEAT GEN_TAC)
THEN REPEAT ( (CONV_TAC RIGHT_IMP_FORALL_CONV)
THEN REPEAT GEN_TAC)
THEN CONV_TAC (simpLib.SIMP_CONV HOLSimps.hol_ss [])
 );
```

Le système répond par dire que le théorème a été prouvé en utilisant la preuve déclarée précédemment :

```
Initial goal proved.
 |- !a b c d out.
     system_imp (a,b,c,d,out) ==> system_spec (a,b,c,d,out)
: Goalstackpure.goalstack
```

3.3. Conclusion

Dans ce chapitre on a présenté l'état de l'art des méthodes formelles de vérification. Ensuite, on a mis le point sur les méthodes de vérification basées sur le prouveur de théorèmes HOL. Enfin, On a décrit les principales caractéristiques du système HOL et on a conclu par un exemple de vérification mettant en évidence la manipulation d'un tel système.

Chapitre 4 : SPECIFICATION DE LA FAMILLE DE PROCESSEURS ADSP-2100

La vérification d'un microprocesseur dans le cadre général se fait en deux étapes (Figure 4.1). La première s'intéresse à l'écriture de deux spécifications : la spécification de son fonctionnement et la spécification de son implémentation. Alors que la deuxième phase concerne la démonstration de l'implication entre les deux spécifications précédemment définies. Dans ce chapitre, on va s'intéresser à la première phase qui concerne dans notre ces les spécifications fonctionnelles et structurelles de la famille de processeurs ADSP-2100.

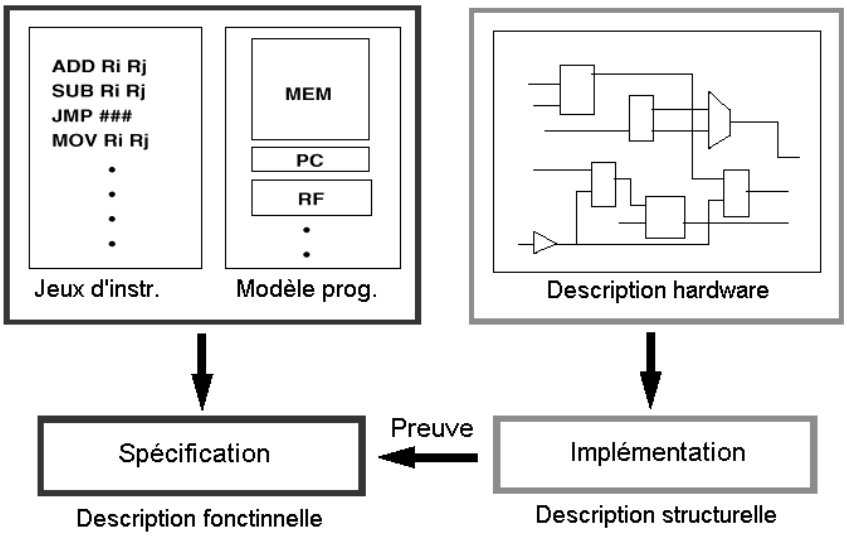


Figure 4.1. Principe de base de la vérification formelle d'un microprocesseur.

4.1. Architecture et caractéristiques

4.1.1. Les unités de base

Les microprocesseurs de la famille ADSP-2100 possèdent la même architecture interne décrite sur la Figure 4.2. Cette architecture est caractérisée, à la différence des autres classes de processeurs, par l'existence de trois unités fonctionnelles indépendantes : une unité arithmétique et logique (ALU), une unité multiplieur/accumulateur (MAC) et une unité de décalage de Barrel (Barrel Shifter). Ces unités traitent des données de 16 bits.

Les deux générateurs d'adresses et le séquenceur de programme permettent de générer les adresses. L'utilisation de deux générateurs d'adresses permet d'assurer deux instructions de recherche des données (*fetch*) à partir de la mémoire externe. Le séquenceur de programme se distingue aussi par sa capacité d'exécuter des boucles internes ce qui augmente l'efficacité de fonctionnement du processeur.

La caractéristique fondamentale de cette classe de processeurs est qu'elle se base sur une architecture Harvard modifiée. C'est à dire que les données sont stockées à la fois dans la mémoire programme et la mémoire données. Cela est d'une grande importance car il permet d'exécuter à la fois deux instructions pendant le même cycle d'horloge.

4.1.2. Les bus

Le processeur ADSP-2100 possède 5 bus internes. Ceux-ci permettent aux différentes unités de communiquer ensemble et de chercher ou de stocker des données dans les mémoires externes.

4.1.2.1. Le bus PMA (Program Memory Adress)

Selon l'instruction actuelle et l'état interne du processeur, le séquenceur de programme génère la nouvelle adresse. Celle-ci est écrite sur le bus PMA qui est composé de 14 bits et qui permet d'adresser jusqu'à 14Ko de mémoire programme et 14Ko de mémoire données externes. La distinction entre les deux mémoires se fait au moyen du pin PMDA.

4.1.2.2. Le bus PMD (Program Memory Data)

Ce bus sert à transférer les instructions de la mémoire externe au registre interne d'instructions. Les instructions sont chargées dans un cycle de fonctionnement du processeur et sont exécutées dans le cycle suivant et de façon simultanée avec le chargement de l'instruction suivante. Toutes les instructions chargées sont aussi inscrites dans la mémoire cache.

4.1.2.3. Le bus DMA (Data Memory Address)

Ce bus est composé de 14 bits et permet l'accès direct à 16 Ko de données. Les adresses des données peuvent parvenir de deux sources : une valeur absolue définie par l'instruction (adressage direct) ou la sortie de l'un des deux générateurs d'adresses (adressage indirect).

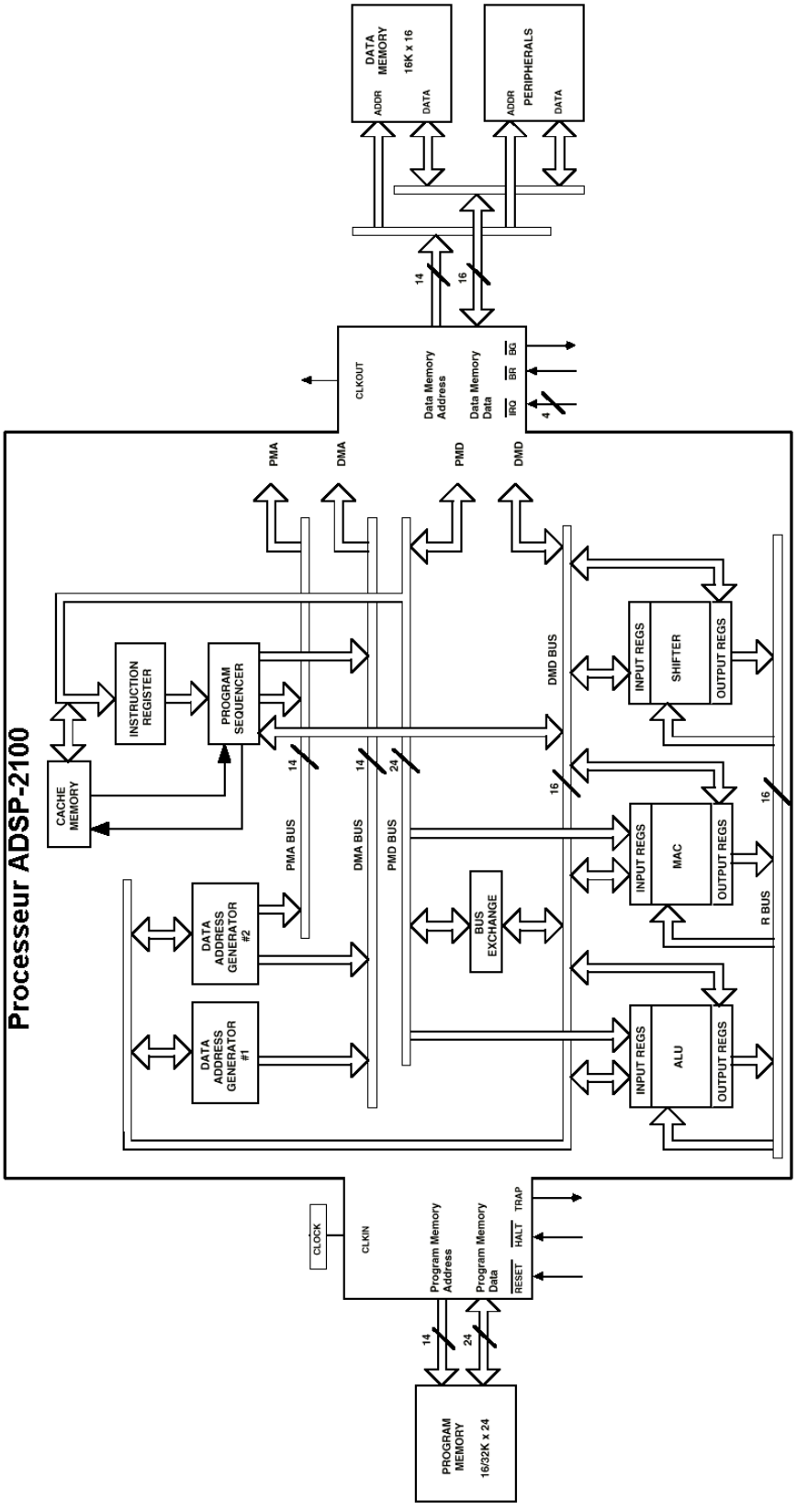


Figure 4.2. Architecture des processeurs de la famille ADSP-2100. Le bus DMD (Data Memory Data) [23] .

4.1.2.4. Le bus DMD

Le bus DMD, composé de 16 bits, permet d'accéder au contenu de n'importe quel registre et permet de transférer ce contenu à n'importe quel autre registre ou à n'importe quel mémoire de données externe. Ce transfert est assuré en un seul cycle d'horloge.

4.1.2.5. Le bus R

Les unités du processeur sont organisées de façon à assurer un fonctionnement parallèle. Pour éliminer toute forme de délais excessifs de pipeline, un bus interne R assure le transfert de données entre les différents registres des trois unités de calcul du processeur. Cela permet un transfert direct et dans le même cycle d'horloge.

4.1.3. La mémoire

L'architecture de la famille de processeurs ADSP-2100 leur autorise de stocker les données dans la mémoire données et les données et les instructions dans la mémoire programme. Chaque processeur possède soit une mémoire RAM et/ou une mémoire ROM.

4.2. Le jeu d'instruction de la famille ADSP-2100

Le jeu d'instructions de la famille ADSP-2100 se distingue par deux propriétés fondamentales : la classification des différentes instructions et la multiplicité d'instructions par commande [21] . Les instructions sont classées selon les unités concernées. En d'autres termes, les instructions concernant l'ALU ont le même format qui se diffère par rapport à celui concernant les instructions du séquenceur de programme. D'autre part, la majorité des instructions permettent à la fois un ou deux accès à la mémoire (en lecture ou en écriture) et une exécution d'une opération (ADD par exemple).

4.2.1. Format des instructions

Le format général d'une instruction est composé comme l'indique la Figure 4.3, de deux parties principales : l'identifiant de l'instruction et les paramètres de l'instruction. Sur la Figure 4.3, est illustré le format général de l'instruction

exécutant une opération de l'ALU et deux lectures une de la mémoire programme et l'autre de la mémoire programme.

FORMAT GENERAL:

Identifiant de l'opcode	Paramètres de l'opcode

EXEMPLE : OPCODE DU TYPE 1

23	22	21 20	19 18	17 16 15 14 13	12 11	10 9 8	7 6	5 4	3 2	1 0
1	1	PD	DD	AMF	Yop	Xop	PMI	PMM	DMI	DMM

Figure 4.3. Format général des instructions de la famille ADSP-2100 [21] .

4.2.2. Classification des instructions

Les différentes instructions de la famille ADSP-2100 pourraient être classifiées en trois sous-classes :

1. Les instructions MAC/ALU avec accès à la mémoire : elles concernent des opérations assurées par les unités ALU et MAC ainsi que des accès lecture aux mémoires données et/ou programme.

2. Les instructions d'accès à la mémoire : elles permettent l'accès aux mémoires données et programme pour lecture et/ou écriture.

3. Les instructions ALU/MAC avec transfert de données inter-registres : elles permettent d'exécuter des opérations ALU ou MAC ainsi qu'un transfert de données entre les registres des différentes unités.

4. Les instructions d'accès direct aux registres : elles permettent de lire directement le contenu de l'un des registres des trois unités ALU, MAC ou Shifter.

5. Les instructions ALU/MAC conditionnelles : elles permettent d'exécuter des instructions ALU/MAC selon des conditions définies sur les bits de sorties des différentes unités.

6. Les instructions de décalage : elles concernent les opérations relatives à l'unité Shifter.

7. Les instructions de contrôle du mode : elles permettent d'agir sur le mode de fonctionnement du processeur tel est le cas pour changer le banc de registre actif par exemple.

8. Les instructions du séquenceur de programme : elles concernent les opérations de JUMP, JUMP conditionnel et la boucle DO UNTIL.

9. Les instructions de division : elles concernent les deux opérations DIVS et DIVQ relatives à la division.

10. Les instructions de commande : elles concernent les opérations NOP (No Operation) et Idle.

4.2.3. Spécification du jeu d'instructions

Dans notre spécification des instructions nous avons considéré chaque classe à part. En effet, vu la similarité des instructions relatives à chaque classes, cela nous permettra d'élaborer des preuves similaires pour les membres de chaque classe dans la phase de vérification.

La spécification que nous avons considérée est une représentation du jeu d'instructions selon le langage HOL. Le décodage des instructions revient à les classifier en premier lieu et puis à extraire les champs qu'elles comportent. Nous n'avons opté pas pour la représentation de l'ensemble du jeu d'instructions du premier coup. Plutôt, nous avons considéré les instructions au départ une à une. Cela possède l'avantage de simplifier les tâches de vérification.

Une fois les preuves de l'ensemble des instructions d'une classe réalisées, on a passé à la représentation de toute la classe étudiée. De cette manière le jeu d'instruction considéré est une classe d'instruction. Une fois toutes les preuves des différentes classes terminées, on passe à la spécification de toutes les instructions. Ainsi, la preuve générale comportant toutes les instructions est devenue soluble et beaucoup moins complexe.

4.3. Hypothèses et Approximations

Dans notre étude de la famille de processeurs ADSP-2100, nous avons considéré certaines hypothèses. Celles-ci ont pour objectif d'éliminer certains problèmes de manipulation des preuves sans pour autant toucher à la validité des résultats auxquels nous allons parvenir.

4.3.1. Le temps d'accès à la mémoire

Nous avons considéré que le temps d'exécution d'une instruction demande un cycle d'horloge. Cela pourrait paraître à première vue en contradiction avec le fonctionnement normal du processeur car on n'a pas tenu compte du temps d'accès aux mémoires externes. Mais, comme, pour le cas de la famille ADSP-2100, toutes les instructions sont chargées dans la mémoire cache avant d'être exécutées, on pourrait supposer que le temps effectif d'exécution d'une instruction revient à un cycle d'horloge.

4.3.2. Le transfert de données dans les bus

Nous considérons que le transfert de données dans les bus se fait de façon instantanée. Cela n'est pas en contradiction avec la réalité. Car, Le transfert de données par les différents bus du processeur se fait de façon très rapide comparé à la valeur de la période du cycle d'horloge. Donc, le fait de le considérer comme nul ne touche en rien la représentation du fonctionnement global du système réel.

4.3.3. L'accès aux registres

Les registres sont accédés au début du cycle pour des raisons d'écriture et à sa fin pour des raisons de lecture. Pour cela, on a considéré que la valeur à un instant *t* d'un registre donné n'est autre que celle présente à son entrée à l'instant *t-1* (c'est à dire à la fin du cycle précédent). Cette représentation ne fausse pas le contenu du registre et permet des manipulations plus souples au niveau des preuves des différentes instructions.

4.4. La spécification formelle

Pour notre cas d'étude la spécification formelle concerne deux volets : la spécification du hardware et la spécification du fonctionnement du processeur. Notre approche est de décortiquer les preuves le plus que possible. De ce fait, nous avons spécifier toutes les unités hardware du processeur en premier lieu. D'un autre côté nous avons spécifié le fonctionnement de ces unités. Notre première tâche dans la phase vérification revient à prouver l'implication :

représentation hardware ⇒ représentation software pour toutes les unités.

Cette première phase nous permettra de manipuler des représentations plus simples des unités dans la phase vérification des instructions. En effet, dans cette phase on aura à manipuler une représentation plus organisée et souple à décortiquer du système sans toucher en rien la validité de nos preuves. De plus amples détails sur notre approche seront présentés dans le chapitre 5 qui présentera la démarche de la vérification des différentes instructions du processeur ADSP-2100.

4.4.1. Les types de données

Dans notre manipulation du processeur ADSP-2100, nous avons manipulé plusieurs types de données. Ceux-ci sont classifiés dans le Tableau 4.1.

4.4.2. Les multiplexeurs

Dans chaque unité du processeur sont intégrés plusieurs multiplexeurs qui permettent la sélection entre les entrées disponibles et les sorties possibles. En parcourant toutes les unités, on s'aperçoit que ces multiplexeurs ne sont pas tous de la même classe. En effet, ils se distinguent par le nombre d'entrées et par le type de ces entrées. De ce fait, on a considéré 7 représentations des différents multiplexeurs. Néanmoins, pour chaque classe on a considéré que le transfert entre l'entrée et la sortie est ponctuel dans le temps. Cela ne contredit pas la réalité vu la rapidité du transfert entre l'entrée et la sortie.

Type	***Signification***	***Commentaires***
Bool	Type booléen : Vrai ou Faux.	Ce type concerne les bits de commandes et de sortie des différentes unités du processeur.
Word5	Mot machine composé de 5 bits.	Ce type concerne le paramètre exposant relatif à l'unité Shifter.
Word8	Mot machine composé de 8 bits.	Ce type concerne le paramètre de décalage du Shifter ainsi que le troisième registre de la MAC.
Word14	Mot machine composé de 14 bits.	Ce type concerne les bus PMA et DMA.
Word16	Mot machine composé de 16 bits.	Ce type concerne la majorité des registres d'entrée ou de sortie des différentes unités du processeur ainsi que les bus PMD et R.
Word24	Mot machine composé de 24 bits.	Ce type concerne le bus PMD qui permet l'accès à la mémoire programme.
Word32	Mot machine composé de 32 bits.	Ce type concerne la sortie du bloc ShifterArray de l'unité Shifter.
Word40	Mot machine composé de 40 bits.	Ce type concerne la première entrée du bloc ADD/SUBSTRACT de l'unité MAC.
Num	Un entier naturel.	Ce type est utilisé pour représenter le paramètre temps (t).
OpCode	Une liste composée de bool.	Représentation du code d'une instruction.

Tableau 4.1. Les types de données utilisés pour la spécification du processeur ADSP-2100.

Le Tableau 4.2 illustre les différentes classes qu'on a utilisé pour spécifier les différentes unités du processeur ADSP-2100.

Classe	*Entrées*		*Sortie*
	Nombre	*Type*	*Type*
1	2	(word16,word16)	Word16
2	2	(word16,word8)	Word16
3	2	(word16,word8)	Word8
4	2	(word8,word16)	Word8
5	2	(word8,word16)	Word5
6	2	(word14,word16)	Word14
7	3	(word8,word8,Word8)	Word8

Tableau 4.2. Les classes des multiplexeurs considérées dans la spécification du processeur ADSP-2100.

La spécification suivante, écrite ne langage ML, illustre le cas du multiplexeur de la classe 4. C'est à dire le cas ou on a deux entrées, la première de type Word8 et la deuxième de type Word 8, et une sortie de type Word8. La sélection entre les deux entrées se fait selon la valeur du paramètre booléen *input_select* qui lorsqu'il est Vrai (T : True) affecte la première entrée à la sortie et lorsqu'il est Faux (F : False) affecte la deuxième entrée à la sortie :

```
MUX_8_16_8 = Define(`
MUX_8_16_8
(FromWord16ToWord8 :word16 -> word8)
(input1 :num -> word8, input2 :num -> word16, output :num -> word8, input_select :num ->
bool) =
( !t :num.
(
  ( input_select t = T ) /\ ( output t = input1 t ) )
\/ ( ( input_select t = F ) / \ ( output t = ( FromWord16ToWord8 (input2 t) ) )
)
)
`);
```

La définition précédente indique que quelque soit l'instant *t* la sortie du multiplexeur *output* correspond à

- l'entrée une *input1* si le paramètre de sélection *input_select* est à la valeur *T*.
- l'entrée une *input2* si le paramètre de sélection *input_select* est à la valeur *F*.

Ainsi le système HOL répond en indiquant qu'il a accepté la définition du multiplexeur et qu'il l'avait enregistré sous l nom *MUX_8_16_def*. Cette définition est stockée sous forme d'un nouveau théorème du type *Thm.thm*.

4.4.3. Les registres

Les différentes unités du processeur regroupent un très grand nombre de registres qui se distinguent par leurs tailles ainsi que par leurs structures. En effet, les tailles des registres varient selon les unités et selon les fonctionnalités auxquelles ils servent. D'autres part, plusieurs registres sont regroupés ensemble pour former un seul banc. De plus, tous les registres sont dupliqués pour permettre l'utilisation d'un second banc directement sans modifier l'état existant en cas d'interruption.

Le Tableau 4.3 illustre les différentes classes de registres que nous avons définies lors de la spécification du hardware des différentes unités du processeur.

Classe	*Nombre de sous-registres*	*Taille (en nombre de bits)*	*Exemple*
1	1	16	Registres AF et AR de l'ALU.
2	2	2x16	Le registre AX qui est composé des deux sous-registres AX0 et AX1.
3	1	8	Registre MR2 de l'unité MAC.
4	1	5	Registre SB de l'unité Shifter.
5	4	4x14	Regitre I du générateur d'adresses qui est composé de 4 sous-registres chacun de taille 14.

Tableau 4.3. Les classes de registres considérés dans la spécification du processeur ADSP-2100.

Le code suivant illustre la spécification du registre AF de l'ALU et qui n'est autre qu'une représentation générique de la classe 1 des registres. On a considéré l'entrée du registre *AF_input* et sa sortie *AF_output* et le paramètre *banc_select*. Ce dernier paramètre booléen permet de sélectionner entre les deux bancs de registres disponibles.

```
ALU_AF_reg = (`ALU_AF_reg(AF_input, AF_output, banc_select) =
!t .
  (? AF_output_banc0 AF_output_banc1 .
    (
     (banc_select t = T) =>   ( if (t = 0) then F else (AF_output_banc0 t = AF_input (t-1)) )
                        |   ( if (t = 0) then F else (AF_output_banc1 t = AF_input (t-1)) )
    )
     / \
    (
     (banc_select t = T) =>   (AF_output t = AF_output_banc0 t)
                        |   (AF_output t = AF_output_banc1 t)
    )
  )
`);
```

La définition précédente indique que quelque soit l'instant *t* la sortie du registre *AF_output* correspond à son entrée à l'instant (*t-1*). Nous avons considéré aussi

le paramètre *banc_select* qui spécifie le banc de registre utilisé parmi les deux bancs disponibles.

4.4.4. Les bus de données

Dans notre représentation des bus de données, nous avons considéré que la sortie du bus correspond à son entrée au même instant. Cette approximation ne contredit pas le fonctionnement normal du processeur, car la lecture des données se fait au début du cycle et leur écriture se fait à sa fin.

Nous avons représenté chaque bus à part de façon à avoir plus de souplesse de clarté dans la manipulation des preuves. Ainsi, on a considéré 5 représentations des bus du processeur : DMD, DMA, PMD, PMA et R. A titre d'exemple, on présente la spécification suivante qui concerne le bus PMA :

```
PMABusImp = (`PMABusImp(PMABusInput :num -> word14,
                        PMABusOutput :num -> word14, t:num) =
 (
   (PMABusOutput t) = (PMABusInput t)
 )
`);
```

La définition du bus *PMA* indique que sa sortie à l'instant *t* correspond à son entrée au même instant *t*.

4.4.5. L'accès à la mémoire

La famille de processeurs ADSP-2100 contient trois types de mémoire : mémoire *BOOT*, mémoire *programme* et mémoire *données*. Les deux mémoires *programmes* et *données* sont accessibles en lecture et en écriture alors que la mémoire *BOOT* n'est accessible qu'en lecture seulement. Pour représenter les différents modes d'accès à ces mémoires, nous avons considéré 4 fonctions qui selon leurs paramètres d'entrées accèdent à l'une des trois mémoires disponibles. Le Tableau 4.4 décrit ces 4 foncions :

Fonction	***Paramètres***	***Accès***
fetch_data	(BMS_low, PMS_low, DMS_low) = (F,T,T)	Cherche des données de la mémoire boot.
	(BMS_low, PMS_low, DMS_low) = (T,F,T)	Cherche des données de la mémoire programme.
	(BMS_low, PMS_low, DMS_low) = (T,F,T)	Cherche des données de la mémoire données.
fetch_address	(BMS_low, PMS_low, DMS_low) = (F,T,T)	Cherche des adresses de la mémoire boot.
	(BMS_low, PMS_low, DMS_low) = (T,F,T)	Cherche des adresses de la mémoire programme.
	(BMS_low, PMS_low, DMS_low) = (T,T,F)	Cherche des adresses de la mémoire données.
Store_data	(PMS_low, DMS_low) = (F,T)	Charge des données dans la mémoire programme.
	(PMS_low, DMS_low) = (T,F)	Charge des données dans la mémoire données.
Store_address	(PMS_low, DMS_low) = (F,T)	Charge des adresses dans la mémoire programme.
	(PMS_low, DMS_low) = (T,F)	Charge des adresses dans la mémoire données.

Tableau 4.4. Les fonctions d'accès à la mémoire.

Les fonctions d'accès à la mémoire prennent aussi en considération les paramètres lecture et d'écriture *RD_LOW* (READ) et *WR_LOW* (WRITE). Cela est illustré pour le cas de la fonction *storedata* :

```
!program_mem_data data_mem_data Index_register Modify_register PMS_low
      DMS_low RD_low WR_low data_to_write t.
     store_data program_mem_data data_mem_data
       (Index_register,Modify_register,PMS_low,DMS_low,RD_low,WR_low,
        data_to_write,t) =
     ((RD_low t,WR_low t) = (T,F)) / \
     (((PMS_low t,DMS_low t) = (F,T)) / \
      (data_to_write t =
       program_mem_data (Index_register (t - 1)) (Modify_register (t - 1))
         (t - 1)) \ /
      ((PMS_low t,DMS_low t) = (T,F)) / \
      (data_to_write t =
       data_mem_data (Index_register (t - 1)) (Modify_register (t - 1))
         (t - 1)))
```

L'accès à la mémoire se fait dans ce cas selon les valeurs des deux paramètres *Index_Register* et *Modify_Register*. Ceux-ci sont définis selon le type de l'instruction.

4.4.6. L'unité ALU

Pour chacune des unités du processeur, nous avons défini une spécification du fonctionnement de l'unité et une spécification du hardware de l'unité. Cela est réalisé dans l'objectif de manipuler dans la phase de vérification des instructions des spécifications assez simples mais qui comprennent toutes les caractéristiques du hardware.

4.4.6.1. Spécification hardware de l'ALU

L'ALU, voir Figure 4.4, est l'unité responsable des opérations arithmétiques et logiques. Elle est composée d'un ensemble de registre d'entrée, d'un ensemble de registre de sortie, de plusieurs multiplexeurs et d'une unité de base assurant les opérations arithmétiques et logiques.

Pour spécifier le hardware de l'ALU nous avons considéré les représentations des registres et des multiplexeurs qu'on a définis dans les paragraphes précédents. L'unité de base de l'ALU (ALU Unit) est représentée par un ensemble de fonctions qui prennent comme entrée les deux paramètres X et Y et génère comme sortie la valeur R ainsi que les différents bits d'état (flags de l'ALU).

Paramètre	***Type***	***Signification***
DMDBUSoutput	Num -> Word16	La sortie du bus DMD.
AX0input	Num -> Word16	L'entrée du registre AX0.
AX1input	Num -> Word16	L'entrée du registre AX1.
AY0input	Num -> Word16	L'entrée du registre AY0.
AY1input	Num -> Word16	L'entrée du registre AY1.
ARoutput	Num -> Word16	La sortie du registre AR.
AFoutput	Num -> Word16	La sortie du registre AF.
AZ	Num -> bool	Vrai si le résultat de l'ALU est nulle.
AN	Num -> bool	Vrai si le résultat de l'ALU est négatif.
AC	Num -> bool	Bit de résidu des opérations de l'ALU.
ALUMUXXinput	Num -> bool	Le paramètre de sélection entre les deux entrées du multiplexeur servant l'entrée pour le registre AX.
ALUMUXYinput	Num -> bool	Le paramètre de sélection entre les deux entrées du multiplexeur servant l'entrée pour le registre AY.
ALUMUXARinput	Num -> bool	Le paramètre de sélection entre les deux entrées du multiplexeur servant l'entrée pour le registre AR.
Axinputselect	Num -> bool	Le paramètre de sélection entre les deux sous-registres AX0 et AX1.
Ayinputselect	Num -> bool	Le paramètre de sélection entre les deux sous-registres AY0 et AY1.
FunctionCode	Num -> Num	La fonction à assurer par l'ALU.

Tableau 4.5. Paramètres d'entrée/sortie de l'ALU.

Nous avons noté que tous les types de données transférés dans l'ALU sont du type Word16, pour cela nous avons considéré un type abstrait générique pour la représentation des différents paramètres. Cela possède l'avantage d'assurer des preuves indépendantes des tailles des registres ou des bus. C'est à dire que notre représentation du système reste valable pour n'importe quel autre circuit de même architecture mais ayant des tailles de registres différentes de la valeur 16.

Les paramètres d'entrée/sortie qu'on a considéré sont définis dans le Tableau 4.5.

4.4.6.2. Spécification du fonctionnement de l'ALU

Dans la spécification du fonctionnement de l'ALU nous avons essayé de représenter cette unité d'une manière simple et optimisée. La représentation qu'on avait utilisée se compose de trois parties : la détermination des entrées, la sélection de la fonction appropriée et en fin le stockage du résultat.

La détermination des entrées

Les entrées de l'unité de base de l'ALU dépendent des multiplexeurs à l'entrée des registres AX et AY et des deux entrées X et Y. Dans notre représentation nous avons sélectionné la valeur de l'entrée de cette unité de base en fonction directement des commandes de ces multiplexeurs. Ces commandes sont pour notre cas : *ALUMUXXinput*, *ALUMUXYinput*, *Axinputselect* et *Ayinputselect*.

La sélection de la fonction appropriée

La fonction à assurer par l'ALU provient de l'instruction par l'intermédiaire du séquenceur de programme. Le paramètre *FunctionCode* identifie l'opération actuelle. Les opérations possibles sont définies dans le Tableau 4.6.

Le stockage des données

Le résultat de l'opération de l'ALU, une fois calculé, est stocké dans le registre AF uniquement ou bien à la fois dans les deux registres AF et AR. Ce résultat peut être soit stocké dans la mémoire données ou programme soit encore utilisé directement par les autres unités par l'intermédiaire du bus R.

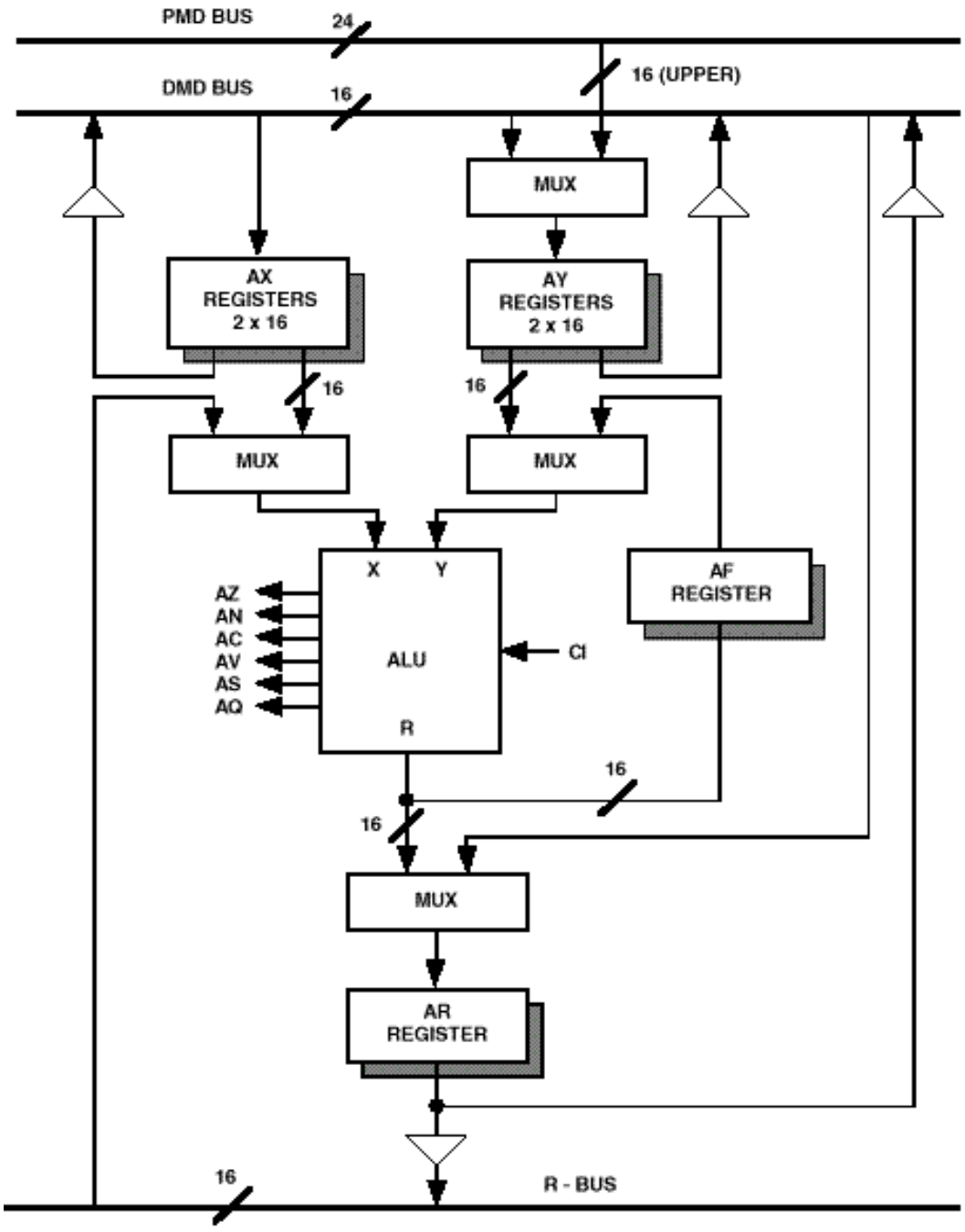


Figure 44. Schéma de l'ALU [21] .

Fonction	*Type*	*Signification*
AddOneToWord16	Word16 -> word16	Incrémentation d'un Word16.
AddWord16WithCI	Word16 -> word16 -> word16 -> word16	Addition de deux éléments de type Word16 avec considération du bit du résidu.
AddWord16	Word16 -> word16 -> word16	Addition de deux éléments de type Word16.
NotWord16	Word16 -> word16	La négation d'un Word16.
MinusWord16	Word16 -> word16	L'opposé d'un Word16.
SubWord16WithCI	Word16 -> word16 -> word16 -> word16	Soustraction de deux éléments de type Word16 avec considération du bit du résidu.
SubWord16	Word16 -> word16 -> word16	Soustraction de deux éléments de type Word16
SubOneFromWord16	Word16 -> word16	Décrémentation d'un Word16.
AndWord16	Word16 -> word16 -> word16	ET logique entre deux éléments de type Word16.
OrWord16	Word16 -> word16 -> word16	OU logique entre deux éléments de type Word16.
XorWord16	Word16 -> word16 -> word16	XOR logique entre deux éléments de type Word16.
AbsWord16	Word16 -> word16	Valeur absolue d'un élément de type Word16.

Tableau 4.6. Les fonctions de l'ALU.

La spécification fonctionnelle de l'ALU telle qu'écrite en langage ML dans l'environnement HOL est décrite par le code ci-dessous :

```
ALU_spec = (`ALU_spec(DMD_BUS_output, AX0_input, AX1_input, AY0_input, AY1_input,
AR_output, AF_input, AF_output, AZ, AN, AC, AV, AS, ALU_MUX_Xinput, ALU_MUX_Yinput,
ALU_MUX_ARinput, AX_input_select, AY_input_select ) =
 (
  ! t.
    (? X AY_output.
      ( (ALU_MUX_Xinput = T) => ( X t = AR_output t)
                          | ( (AX_input_select = T) => (X t = AX0_input (t-1) )
                                                  | (X t = AX1_input (t-1) )
                            )
      )
     /\ ( (AY_input_select = T) => (AY_output t = AY0_input (t-1))
                          | (AY_output t = AY1_input (t-1))
      )
     /\ (? Y R.
          ( (ALU_MUX_Yinput = T)  =>  ( Y t = AY_output t) | (Y t = AF_output t) )
          /\ ALU_unit(X, Y, R, AZ, AN, AC, AV, AS)
          /\ ((ALU_MUX_ARinput = T)  =>  (AR_output t = R (t-1))
                               |  (AR_output t = DMD_BUS_output (t-1))
      )
          /\ ( AF_output t = R ( t-1) )
      )
   )
 )
`);
```

4.4.7. L'unité MAC

4.4.7.1. Spécification du hardware de l'unité MAC

L'unité MAC, voir Figure 4.5, permet les opérations de multiplication, multiplication avec addition cumulative, multiplication avec soustraction cumulative, saturation et mise à zéro. Elle est composée d'un ensemble de registres d'entrée, d'un ensemble de registres de sortie, de plusieurs multiplexeurs et de deux unités de base la première (Multiplier Unit) assurant les opérations de multiplication et la deuxième (Add/Substract Unit) assurant les opérations d'addition et de soustraction.

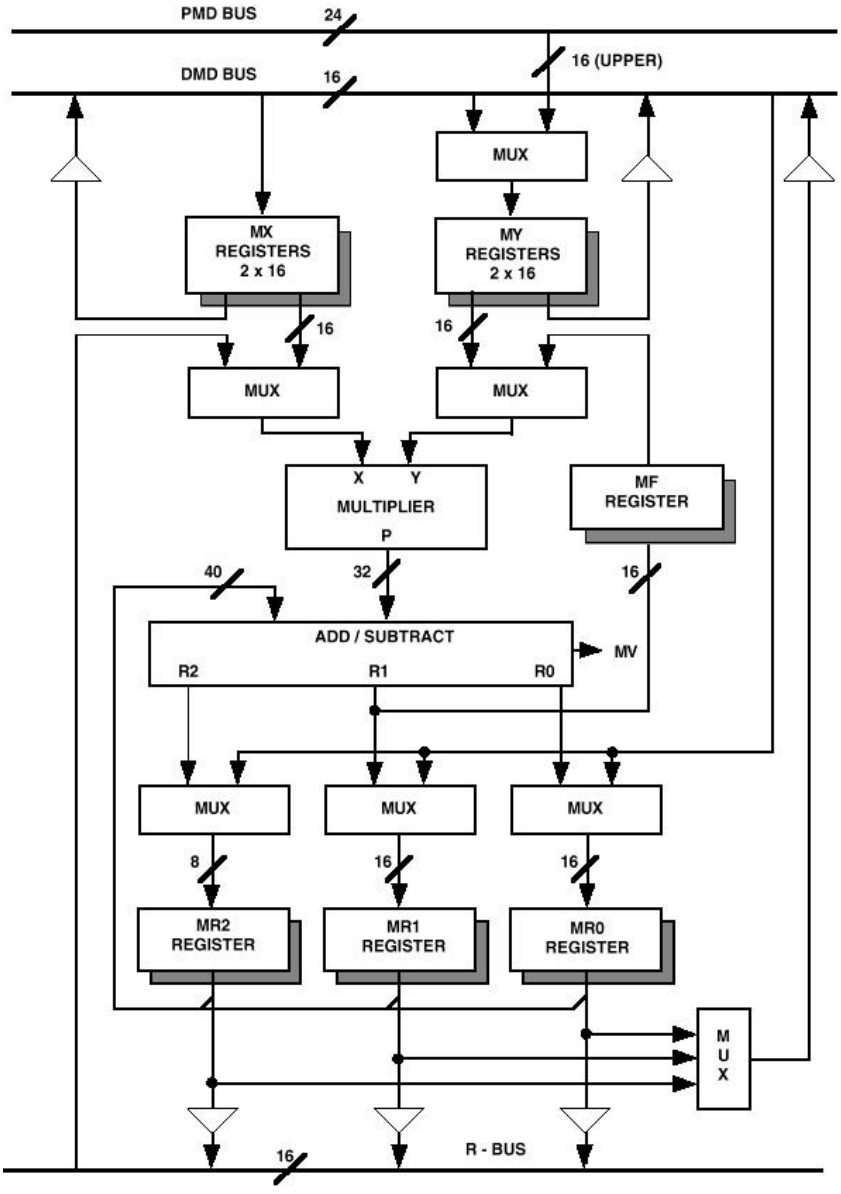


Figure 45. Schéma de l'unité MAC [21] .

Pour spécifier le hardware de l'unité MAC nous avons considéré les représentations des registres et des multiplexeurs qu'on a définis dans les paragraphes précédents. La première unité de base de l'unité MAC (Multiplier Unit) est représentée par un ensemble de fonctions qui prennent comme entrée les deux paramètres X et Y et génère comme sortie la valeur P. La deuxième fonction de base (Add/Sustract Unit) est, elle aussi, représentée par un ensemble de fonctions. Celles-ci prennent leurs entrées de la valeur P (sortie de la première unité) ainsi que la valeur de sortie du résultat de toute l'opération (sortie des registres MR0, MR1 et MR2).

Les paramètres d'entrée/sortie qu'on a considéré sont définis dans le Tableau 4.7.

4.4.7.2. Spécification du fonctionnement de l'unité MAC

Dans cette spécification du fonctionnement de l'unité MAC nous avons essayé de représenter cette unité d'une manière simple et optimisée. La représentation qu'on avait utilisée se compose de quatre parties : la détermination des entrées de l'unité Multiplier, la sélection de la fonction appropriée assurée par l'unité Multiplier, la sélection de la fonction appropriée assurée par l'unité Add/Substract et en fin le stockage du résultat.

La détermination des entrées

Les entrées de l'unité de base de l'unité MAC dépendent du multiplexeur à l'entrée du registre MY et des deux multiplexeurs aux entrées X et Y respectivement. Dans notre représentation nous avons sélectionné la valeur de l'entrée de cette unité de base en fonction directement des commandes de ces multiplexeurs. Ces commandes sont pour notre cas : *MACMYInputMUX*, *MACMXOutputMUX* et *MACMXOutputMUX*.

La sélection de la fonction appropriée pour l'unité Multiplier

La fonction à assurer par l'unité Multiplier de la MAC provient de l'instruction par l'intermédiaire du séquenceur de programme. Le paramètre *FunctionCode* identifie l'opération actuelle. Les opérations qui correspondent à l'unité Multiplier sont définies dans le Tableau 4.8.

Paramètre	***Type***	***Signification***
DMDBUSoutput	Num-> Word16	La sortie du bus DMD.
PMDBusOutput	Num-> Word16	La sortie du bus PMD.
MX0input	Num-> Word16	L'entrée du registre MX0.
MX1input	Num-> Word16	L'entrée du registre MX1.
MY0input	Num-> Word16	L'entrée du registre MY0.
MY1input	Num-> Word16	L'entrée du registre MY1.
MR0input	Num-> Word16	La sortie du registre MR0.
MR1input	Num-> Word16	La sortie du registre MR1.
MR2input	Num-> Word16	La sortie du registre MR2.
Mfoutput	Num-> Word16	La sortie du registre MF.
RbusInput	Num-> Word16	Entrée du bus R.
MACMYInputMUX	Num-> bool	Le paramètre de sélection entre les deux entrées du registre MY.
MACMXOutputMUX	Num-> bool	Le paramètre de sélection entre les deux entrées du multiplexeur servant la valeur d'entrée X de l'unité Multiplier.
MACMYOutputMUX	Num-> bool	Le paramètre de sélection entre les deux entrées du multiplexeur servant la valeur d'entrée Y de l'unité Multiplier.
MACMR0InputMUX	Num-> bool	Le paramètre de sélection de l'entrée du registre MR0 (les deux valeurs possibles sont DMDBusOutput et la première sortie de l'unité ADD/Substract).
MACMR1InputMUX	Num-> bool	Le paramètre de sélection de l'entrée du registre MR1 (les deux valeurs possibles sont DMDBusOutput et la deuxième sortie de l'unité ADD/Substract).
MACMR2InputMUX	Num-> bool	Le paramètre de sélection de l'entrée du registre MR2 (les deux valeurs possibles sont DMDBusOutput et la troisième sortie de l'unité ADD/Substract).
MACRBusInputSelect0	Num-> bool	Premier paramètre de sélection entre les trois sorties des registres MR0, MR1 et MR2 pour être véhiculer par le bus R.
MACRBusInputSelect1	Num-> bool	Premier paramètre de sélection entre les trois sorties des registres MR0, MR1 et MR2 pour être véhiculer par le bus R.
MACMRoutputMUX0	Num-> bool	Premier paramètre de sélection entre les trois sorties des registres MR0, MR1 et MR2 pour être véhiculer par le bus DMD.
MACMRoutputMUX1	Num-> bool	Premier paramètre de sélection entre les trois sorties des registres MR0, MR1 et MR2 pour être véhiculer par le bus DMD.
MACMXRegSelect	Num-> bool	Le paramètre de sélection entre les deux sous-registres MX0 et MX1.
MACMYRegSelect	Num-> bool	Le paramètre de sélection entre les deux sous-registres MY0 et MY1.
FunctionCode	Num-> Num	La fonction à assurer par l'unité MAC.

Tableau 4.7. Paramètres d'entrée/sortie de l'unité MAC.

Fonction	*Type*	*Signification*
MAC_NO_OP	Word16 -> word16 -> word32	Aucune opération.
Multi_Word_16	Word16 -> word16 -> word32	Multiplication de deux éléments de type word16.
Multi_Word_16_SS	Word16 -> word16 -> word32	Multiplication de deux éléments signés (Signed) de type word16.
Multi_Word_16_SU	Word16 -> word16 -> word32	Multiplication de deux éléments le premier non-signé (Unsigned) et le deuxième signé (Signed).
Multi_Word_16_US	Word16 -> word16 -> word32	Multiplication de deux éléments le premier signé (Signed) et le deuxième non-signé (Unsigned).
Multi_Word_16_UU	Word16 -> word16 -> word32	Multiplication de deux éléments non-signés (Unsigned) de type word16.

Tableau 4.8. Les fonctions de l'unité Multiplier de la MAC.

La sélection de la fonction appropriée pour l'unité Add/Substract

La fonction à assurer par l'unité Multiplier de la Add/Substract provient de l'instruction par l'intermédiaire du séquenceur de programme. Le paramètre *FunctionCode* identifie l'opération actuelle. Les opérations qui correspondent à l'unité Add/Substract sont définies dans le Tableau 4.9.

Fonction	*Type*	*Signification*
ADD_Word_40_Word_32	word40 -> word32-> word16 # word16 # word8	Rajoute la valeur de retour de type word40 à une valeur de type word32 et génère trois valeurs : word16, word16 et word8.
SUB_Word_40_Word_32	word40 -> word32-> word16 # word16 # word8	Soustrait la valeur de retour de type word40 à une valeur de type word32 et génère trois valeurs : word16, word16 et word8.

Tableau 4.9. Les fonctions de l'unité Add/Substract de la MAC.

Le stockage des données

Le résultat de l'opération de l'unité MAC, une fois calculé, est stocké dans les trois registres MR0, MR1 et MR2. Ce résultat peut être soit stocké dans la mémoire données ou programme soit encore utilisé directement par les autres unités par l'intermédiaire du bus R. Mais, vu que les bus DMD et R sont chacun

de taille 16 bits et que le résultat de l'unité MAC est de taille 40 bits, uniquement le contenu de l'un des registres MR0 ou MR1 ou MR2 est lu dans un seul cycle d'horloge.

Les paramètres qui spécifient lequel des registres MR0 ou MR1 ou encore MR2 sera écrit sur le bus DMD sont : *MACMRoutputMUX0*, *MACMRoutputMUX0*. Les paramètres qui spécifient lequel de ces registres sera écrit sur le bus R pour être utilisé par les autres unités du processeur sont : *MACRBusInputSelect0*, *MACRBusInputSelect0*.

La spécification fonctionnelle du MAC telle qu'écrite en langage ML dans l'environnement HOL est décrite par le code ci-dessous :

```
val MAC_Spec = (`    MAC_Spec
(DMDBusOutput    :num -> word16, PMDBusOutput :num -> word24, MX0Input      :num ->
word16, MX1Input     :num -> word16,  MY0Input      :num -> word16, MY1Input     :num ->
word16, MFInput       :num -> word16, MR0Input      :num -> word16, MR1Input     :num ->
word16, MR2Input       :num -> word8,  MV :num -> bool, RBusInput      :num -> word16,
MACMXRegSelect :num -> bool, MACMYRegSelect :num -> bool, functionCode :num -> num,
t :num, X :num -> word16, Y :num -> word16, R :num -> word40, P :num -> word32) =
  ( (
            (*--- X input selection ---*)
       (( (MACMXOutputMUX t = T)
          /\ (X t = RBusInput (t-1)) )
        \/ ( ((MACMXRegSelect t, MACMXOutputMUX t) = (T,F))
          /\ (X t = MX0Input (t-1)) )
          \/ ( ((MACMXRegSelect t, MACMXOutputMUX t) = (F,F))
          /\ (X t = MX1Input (t-1)) ) )
       /\ (*--- Y input selection ---*)
       ( ( (MACMYOutputMUX t = F)
          /\ (Y t = MFInput (t-1)) )
        \/ ( ((MACMYRegSelect t, MACMYOutputMUX t) = (T,T))
          /\ (Y t = MY0Input (t-1)) )
        \/ ( ((MACMYRegSelect t, MACMYOutputMUX t) = (F,T))
          /\ (Y t = MY1Input (t-1)) ) )
            (*--- Multiplier Unit ---*)
        /\  (Multiplier_unit (X,Y,P,functionCode,t))
            (*--- R output from multiplier unit ---*)
        /\  (R t = (ConstructWord40FromRRegisters
                  (MR0Input (t-1), MR1Input (t-1), MR2Input (t-1)) ) )
            (*--- AddSbstruct Unit---*)
        /\  (ADD_Substruct_unit (R,P,R0,R1,R2,MV,functionCode,t))
          (*--- MR0 load value selection---*)
        /\ ( ( (MACMR0InputMUX t = T) /\ (MR0Input t = R0 t) )
           \/  ( (MACMR0InputMUX t = F) /\ (MR0Input t = DMDBusOutput t) ) )
     \/  ( (MACMR2InputMUX t = F) /\ (MR2Input t = (FromWord16ToWord8 (DMDBusOutput
t)) ) ) )
            (*--- DMD Bus load value selection---*)
        /\ ( ( ((MACMRoutputMUX0 t, MACMRoutputMUX1 t) = (F,F))
            /\ (DMDBusOutput t = (FromWord8ToWord16 (MR2Input (t-1))) )
          ) \/ (  ((MACMRoutputMUX0 t, MACMRoutputMUX1 t) = (F,T))
            /\ (DMDBusOutput t = MR1Input (t-1)) )
            \/ ( ((MACMRoutputMUX0 t, MACMRoutputMUX1 t) = (T,F))
            /\ (DMDBusOutput t = MR0Input (t-1)) ) )
             (*--- R Bus load value selection ---*)
```

```
))`);
```

4.4.8. L'unité Shifter

4.4.8.1. Spécification du hardware de l'unité Shifter

L'unité Shifter, voir Figure 4.6, traite d'un élément de 16 bits pour générer un élément de 32 bits. Les opérations assurées par cette unité sont : le décalage arithmétique, le décalage logique et la normalisation. Ces fonctions combinées ensemble permettent une variété de manipulation du format numérique de l'entrée ; cela correspond en particulier pour représenter les éléments à virgule flottante.

L'unité Shifter est composée d'un ensemble de registres d'entrée, d'un ensemble de registres de sortie, de plusieurs multiplexeurs et de deux unités de base la première (Shifter Array) assurant les opérations de décalage et la deuxième (OR/PASS Unit) assurant l'opération OR entre la sortie du Shifter Array et la sortie des deux registres SR0 et SR1.

Pour spécifier le hardware de l'unité Shifter nous avons considéré les représentations des registres et des multiplexeurs qu'on a définis dans les paragraphes précédents. La première unité de base de l'unité Shifter (Shifter Array) est représentée par un ensemble de fonctions qui prennent comme entrée les deux paramètres I (élément à décalée), X (bit de remplissage), R (permet de définir le point de référence pour le décalage) et C (le nombre de bits à décaler) et génère comme sortie la valeur O. La deuxième fonction de base (OR/PAASS Unit) est représentée par une seule fonction OR logique qui prend comme entrée la sortie de la première unité ainsi que la valeur de sortie des registres de résultat SR0 et SR2.

Les paramètres d'entrée/sortie qu'on a considéré sont définis dans le Tableau 4.10.

4.4.8.2. Spécification du fonctionnement de l'unité Shifter

Dans cette spécification du fonctionnement de l'unité Shifter nous avons essayé de représenter cette unité d'une manière simple et optimisée. La représentation qu'on avait utilisée se compose de quatre parties : la détermination des entrées de l'unité Shifter Array, la sélection de la fonction appropriée assurée par l'unité Shifter Array, la sélection de la fonction appropriée assurée par l'unité OR/PASS et en fin le stockage du résultat.

La détermination des entrées

L'unité de base (Shifter Array) de l'unité Shifter possède quatre entrées : **I**, **X**, **R** et **C**.

I : l'élément à décaler dépend du multiplexeur à l'entrée du registre SE.

X : le bit de remplissage est spécifié par le séquenceur de programme.
R : le point de référence pour le décalage de l'élément **I**.
C : le nombre de bits à décaler. Ce paramètre correspond au contenu du registre SE, à la négation du registre SE ou encore à une valeur spécifiée directement par l'instruction.

La sélection de la fonction appropriée

La fonction à assurer par les unités Shifter Array et OR/PASS provient de l'instruction par l'intermédiaire du séquenceur de programme. Le paramètre *FunctionCode* identifie l'opération actuelle. Nous avons représenté ces deux unités sous forme d'une seule. Les opérations qui correspondent à l'unité Shifter sont définies dans le Tableau 4.11.

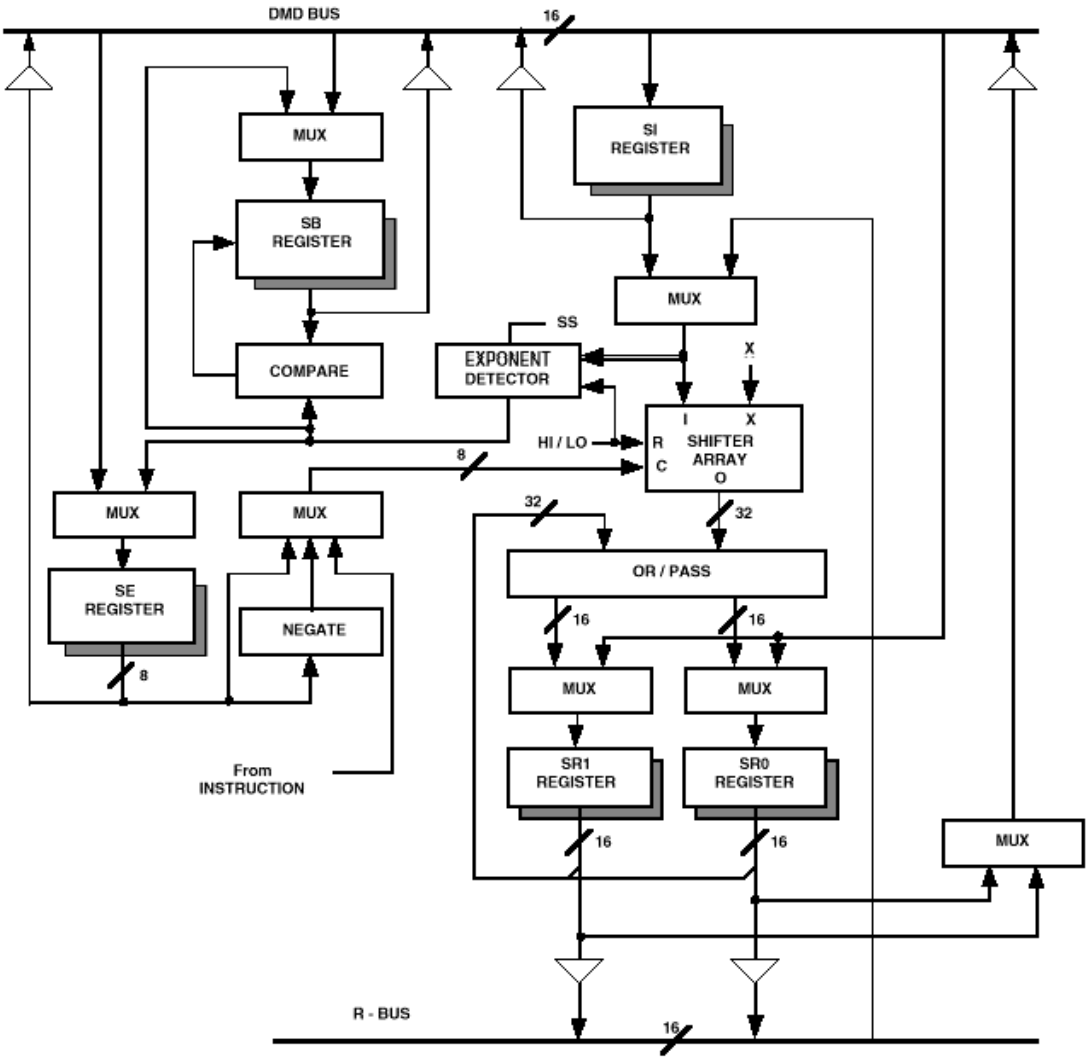


Figure 4.6. Schéma de l'unité Shifter [21] .

Paramètre	***Type***	***Signification***
DMDBUSOutput	Num-> Word16	La sortie du bus DMD.
DMDBusInput	Num-> Word16	L'entrée du bus DMD.
SIInput	Num-> Word16	L'entrée du registre SI.
SR0Input	Num-> Word16	L'entrée du registre SR0.
SR1Input	Num-> Word16	L'entrée du registre SR1.
SEInput	Num-> Word8	L'entrée du registre SE.
SBInput	Num-> Word5	La sortie du registre SB.
RbusInput	Num-> Word16	La sortie du bus R.
C_ValueFromInstruction	Num-> Word8	La valeur du nombre de bits à décaler au cas où elle est spécifiée directement par l'instruction.
ShifterIinputMUX	Num-> bool	Le paramètre de sélection de l'entrée I de l'unité Shifter Array parmi les deux paramètres : SIInput et la sortie du bus DMD.
ShifterCinputMUX0	Num-> bool	Le premier paramètre de sélection de l'entrée C de l'unité Shifter Array parmi les trois paramètres : la sortie du registre SE, la négation de la sortie du registre SE et une valeur spécifiée par l'instruction.
ShifterCinputMUX1	Num-> bool	Le deuxième paramètre de sélection de l'entrée C de l'unité Shifter Array parmi les trois paramètres : la sortie du registre SE, la négation de la sortie du registre SE et une valeur spécifiée par l'instruction.
ShifterSBinputMUX	Num-> bool	Le paramètre de sélection de l'entrée du registre SB (les valeurs possibles sont : la sortie de l'unité Exponent Detector et la sortie du bus DMD).
ShifterSEinputMUX	Num-> bool	Le paramètre de sélection de l'entrée du registre SE (les valeurs possibles sont : la sortie de l'unité Exponent Detector et la sortie du bus DMD).
ShifterSR0inputMUX	Num-> bool	Le paramètre de sélection de l'entrée du registre SR0 (les deux valeurs possibles sont DMDBusOutput et la première sortie de l'unité OR/PASS).
ShifterSR1inputMUX	Num-> bool	Le paramètre de sélection de l'entrée du registre SR1 (les deux valeurs possibles sont DMDBusOutput et la deuxième sortie de l'unité OR/PASS).
ShifterDMDinputMUX	Num-> bool	Le paramètre de sélection entre les trois sorties des registres SR0 et SR1 pour être véhiculer par le bus DMD.
ShifterRBusInputSelect	Num-> bool	Le paramètre de sélection entre les trois sorties des registres SR0 et SR1 pour être véhiculer par le bus R.
ExponentDetectorOutput	Num-> word8	Paramètre de sortie de l'unité Exponenet Detector.
CompareUnitOutput	Num-> word5	Paramètre de sortie de l'unité Compare Unit.
FunctionCode	Num-> Num	La fonction à assurer par l'unité Shifter.

Tableau 4.10. Paramètres d'entrée/sortie de l'unité Shifter.

Fonction	***Type***	***Signification***
SHIFTER_LSHIFT_HI	word16 # bool # word8 # word32 -> word16 # word16	Décalage à gauche avec considération du paramètre C au niveau HI.
SHIFTER_LSHIFT_HI_OR	word16 # bool # word8 # word32 -> word16 # word16	Décalage à gauche avec considération du paramètre C au niveau HI et application du OR logique au résultat obtenu.
SHIFTER_LSHIFT_LO	word16 # bool # word8 # word32 -> word16 # word16	Décalage à gauche avec considération du paramètre C au niveau LO.
SHIFTER_LSHIFT_LO_OR	word16 # bool # word8 # word32 -> word16 # word16	Décalage à gauche avec considération du paramètre C au niveau HI et application du OR logique au résultat obtenu.
SHIFTER_ASHIFT_HI	word16 # bool # word8 # word32 -> word16 # word16	Décalage à droite avec considération du paramètre C au niveau HI.
SHIFTER_ASHIFT_HI_OR	word16 # bool # word8 # word32 -> word16 # word16	Décalage à droite avec considération du paramètre C au niveau HI et application du OR logique au résultat obtenu.
SHIFTER_ASHIFT_LO_OR	word16 # bool # word8 # word32 -> word16 # word16	Décalage à droite avec considération du paramètre C au niveau LO et application du OR logique au résultat obtenu.
SHIFTER_NORM_HI	word16 # bool # word8 # word32 -> word16 # word16	Normalisation à gauche avec considération du paramètre C au niveau HI.
SHIFTER_NORM_HI_OR	word16 # bool # word8 # word32 -> word16 # word16	Normalisation à gauche avec considération du paramètre C au niveau HI et application du OR logique au résultat obtenu.
SHIFTER_NORM_LO	word16 # bool # word8 # word32 -> word16 # word16	Normalisation à gauche avec considération du paramètre C au niveau LO.
SHIFTER_NORM_LO_OR	word16 # bool # word8 # word32 -> word16 # word16	Normalisation à gauche avec considération du paramètre C au niveau LO et application du OR logique au résultat obtenu.
SHIFTER_EXP_HI	word16 # bool # word8 # word32 -> word16 # word16	Détection de l'exposant du paramètre d'entrée I en considération du paramètre C à la valeur HI.
SHIFTER_EXP_LO	word16 # bool # word8 # word32 -> word16 # word16	Détection de l'exposant du paramètre d'entrée I en considération du paramètre C à la valeur LO.
SHIFTER_Derive_Block_Control	word16 # bool # word8 # word32 -> word16 # word16	Cherche le plus grand exposant des éléments d'une liste de paramètres.

Tableau 4.11. Les fonctions de l'unité Multiplier de la MAC.

Le stockage des données

Le résultat de l'opération de l'unité Shifter, une fois calculé, est stocké dans les deux registres SR0 et SR1. Ce résultat peut être soit stocké dans la mémoire données ou programme soit encore être utilisé directement par les autres unités par l'intermédiaire du bus R. Mais, vu que les bus DMD et R sont chacun de taille 16 bits et que le résultat de l'unité Shifter est de taille 32 bits, uniquement le contenu de l'un des registres SR0 ou SR1 est stocké au cours d'un cycle d'horloge.

Le paramètre qui spécifie lequel des registres SR0 ou SR1 sera écrit sur le bus DMD est : *ShifterDMDinputMUX*. Le paramètre qui spécifie lequel des registres SR0 ou SR1 sera écrit sur le bus R pour être utilisé par les autres unités du processeur est : *ShifterRBusInputSelect*.

La spécification fonctionnelle du Shifter telle qu'écrite en langage ML dans l'environnement HOL est décrite par le code ci-dessous :

```
val Shifter_spec = (`Shifter_spec
 (DMDBusOutput :num -> word16, DMDBusInput :num -> word16,SIInput :num -> word16,
SR0Input :num -> word16, SR1Input :num -> word16, SEInput :num -> word8, RBusInput
:num -> word16, C_ValueFromInstruction :num -> word8, ShifterRBusInputSelect :num ->
bool, ShifterI :num -> word16, ShifterC :num -> word8, ShifterR :num -> word8, ShifterX
:num -> bool, ShifterS0 :num -> word16, ShifterS1 :num -> word16,
ExponentDetectorOutput :num -> word8, CompareUnitOutput :num -> word5, functionCode
:num -> num, SROutput :num -> word32, SBOutput :num -> word5 ) =

 (
                (*** I Shifter Array unit selection ***)
   ( (SIInput t = DMDBusOutput t)
     /\ ( ((ShifterIinputMUX t = T) /\ (ShifterI t = SIInput (t-1)))
        \/ ((ShifterIinputMUX t = F) /\ (ShifterI t = RBusInput (t-1))) ) )
  /\ (*** SE Register Output ***)
   ( ((ShifterSEinputMUX t = T) /\ (SEInput t = (FromWord16ToWord8(DMDBusOutput t)) ))
     \/ ((ShifterSEinputMUX t = F) /\ (SEInput t = ExponentDetectorOutput t) ) )
/\ (*** Shifter unit: Array unit and ADD/OR unit ***)
   ( (SROutput t = (FromWord16Word16ToWord32((SR0Input (t-1)),(SR1Input (t-1)))) )
     /\ Shifter_unit (ShifterI,ShifterX,ShifterC,SROutput,ShifterS0,ShifterS1,functionCode,t) )
/\ (*** SR0 Register Input ***)
   ( ( ( (ShifterSR0inputMUX t = T) /\ ( SR0Input t = (DMDBusOutput t) ) )
       \/ ( (ShifterSR0inputMUX t = F) /\ ( SR0Input t = (ShifterS0 t) ) ) ) )
  /\ (*** SR1 Register Input ***)
   ( ( ( (ShifterSR1inputMUX t = T) /\ ( SR1Input t = (DMDBusOutput t) ) )
       \/ ( (ShifterSR1inputMUX t = F) /\ ( SR1Input t = (ShifterS1 t) ) ) ) )
  /\ (*** DMD BUS Input Select ***)
   ( ( (ShifterDMDinputMUX t = T) /\ (DMDBusInput t = SR0Input (t-1)) )
     \/ ( (ShifterDMDinputMUX t = F) /\ (DMDBusInput t = SR1Input (t-1)) ) )
  /\ (*** R BUS Input Select ***)
   ( ( (ShifterRBusInputSelect t = T) /\ (RBusInput t = SR0Input (t-1)) )
     \/ ( (ShifterRBusInputSelect t = F) /\ (RBusInput t = SR1Input (t-1)) ) )
 )
`);
```

4.4.9. Le séquenceur de programmes

Nous avons considéré une représentation du fonctionnement du séquenceur de programme. En effet, la documentation disponible ne nous a pas permis de représenter le hardware du séquenceur de programme. Mais, nous avons pourtant écrit une spécification du fonctionnement de cette unité afin de pouvoir l'utiliser dans la vérification du jeu d'instructions.

Notre représentation du séquenceur de programme le considère comme l'unité responsable du décodage des instructions. Autrement dit, pour chaque instruction le séquenceur de programme identifie la fonction à réaliser ainsi que les paramètres qui entre en jeu lors de sa réalisation. Le schéma décrit le procédé de fonctionnement du séquenceur de programme.

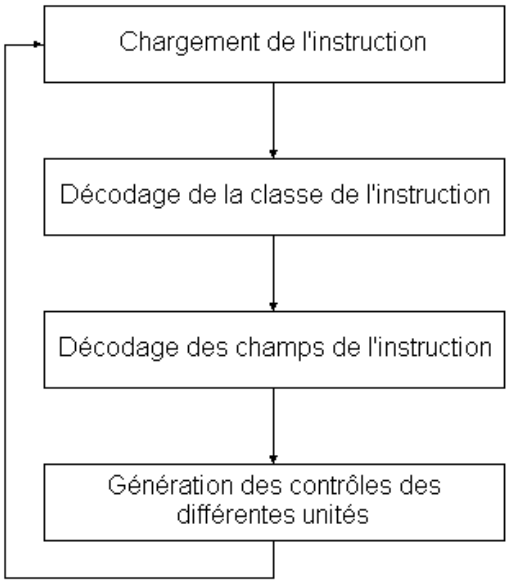


Figure 4.7. Description du fonctionnement du séquenceur de programmes.

4.5. Conclusion

Dans ce chapitre nous avons spécifié les différentes unités du microprocesseur ADSP-2100. Nous avons défini pour chaque unité une représentation du hardware et une représentation du fonctionnement. Notre objectif à travers cette approche est de commencer dans la phase vérification par définir une représentation plus simple et plus compacte des différentes unités du processeur. Ensuite, on procédera à la vérification du jeu d'instructions du processeur en utilisant ces représentations simplifiées.

Chapitre 5 : VERIFICATION FORMELLE DU PROCESSEUR ADSP-2100

La vérification de la famille de processeurs ADSP-2100 est décomposée en deux étapes. La première consiste en la simplification de la représentation des différentes unités du processeur. Cela revient à la démonstration de la preuve que l'implémentation de ces unités implique leurs spécifications. Cette première phase est nécessaire pour simplifier la tâche de la deuxième étape qui concerne la vérification du jeu d'instruction du processeur.

5.1. Plan de vérification

5.1.1. Vérification des unités

Dans notre étude on a commencé par vérifier la relation suivante :

Spécification du hardware de l'unité $\Rightarrow$ *Spécification du fonctionnement de l'unité*

Ces deux spécifications ont été décrites dans le chapitre précédent. Notre objectif est de manipuler des représentations simplifiées des unités du processeur. Ainsi la vérification des instructions se retrouvera allégée.

Notre approche dans la simplification des unités du processeur est structurée. En d'autres termes, nous ne sommes pas attaqués directement au problème mais on l'avait décortiqué. Le schéma suivi est décrit par le diagramme suivant :

Preuve Principale

Spécification du hardware de l'unité $\Rightarrow$ *Spécification du fonctionnement de l'unité*

Sous Preuves

Etape 1 : *Spécification hardware de l'unité* $\Rightarrow$ $1^{ère}$ *Spécification intermédiaire de l'unité*

Etape 2 : $1^{ère}$ *Spécification intermédiaire de l'unité* $\Rightarrow$ $2^{ème}$ *Spécification intermédiaire de l'unité*

. .

. .

. .

Etape n : $(n-1)^{ème}$ *Spécification intermédiaire de l'unité* $\Rightarrow$ $n^{ème}$ *Spécification intermédiaire de l'unité*

Etape finale : $n^{ème}$ *Spécification intermédiaire de l'unité* $\Rightarrow$ *Spécification du fonctionnement de l'unité*

Les spécifications intermédiaires sont des représentations hybrides de l'unité. En d'autres termes, certaines des composantes hardware de l'unité sont remplacées par leurs spécifications fonctionnelles. Ainsi la spécification obtenue est composée à la fois de partie fonctionnelle et d'une autre hardware. Ce procédé permet de simplifier les preuves. L'idée derrière ce procédé est que chaque unité du processeur est composée de sous groupes d'unités indépendantes et qui pourraient être spécifiée chacune à part.

5.1.2. Classification des instructions

Le jeu d'instructions de la famille de processeurs ADSP-2100 est composé de 31 instructions [21] . Ces instructions pourraient être classifiées en des sous-groupes qui se distinguent par l'unité concernée par l'instruction. La classification que nous avons choisi est la suivante :

Instructions d'accès à la mémoire

Cette classe regroupe les instructions d'accès à la mémoire sans assurer aucune autre fonction supplémentaire. Les opcocdes concernés sont :

Type de l'opcode	***Fonction***
2	Ecriture immédiate de données dans la mémoire.
3	Ecriture immédiate d'une adresse dans la mémoire.

Instructions de manipulation des registres

Se sont les instructions concernant la manipulation des registres données (registres de l'ALU, le MAC et le Shifter) et les registres non-données (registres d'état, d'adresses, etc.).

Type de l'opcode	***Fonction***
6	Chargement immédiat d'un registre données.
7	Chargement immédiat d'un registre non-données.
17	Transfert de données inter registres.

Instructions de l'ALU

Ce groupe d'instructions concerne les opérations de l'ALU ainsi que l'accès aux mémoires programme et données. Les opcodes qui sont affectées à ce groupe sont :

Type de l'opcode	***Fonction***
1	Fonctions ALU avec accès en lecture de la mémoire.
4	Fonctions ALU avec accès lecture/écriture mémoire données.
5	Fonctions ALU avec accès lecture/écriture mémoire programme.
8	Fonctions ALU avec transfert de données inter registres.
9	Fonctions ALU conditionnelles.

Instruction du MAC

Ce groupe d'instructions concerne les opérations du MAC ainsi que l'accès aux mémoires programme et données. Les opcodes qui sont affectées à ce groupe sont :

Type de l'opcode	***Fonction***
1	Fonctions MAC avec accès en lecture de la mémoire.
4	Fonctions MAC avec accès lecture/écriture mémoire données.
5	Fonctions MAC avec accès lecture/écriture mémoire programme.
8	Fonctions MAC avec transfert de données inter registres.
9	Fonctions MAC conditionnelles.

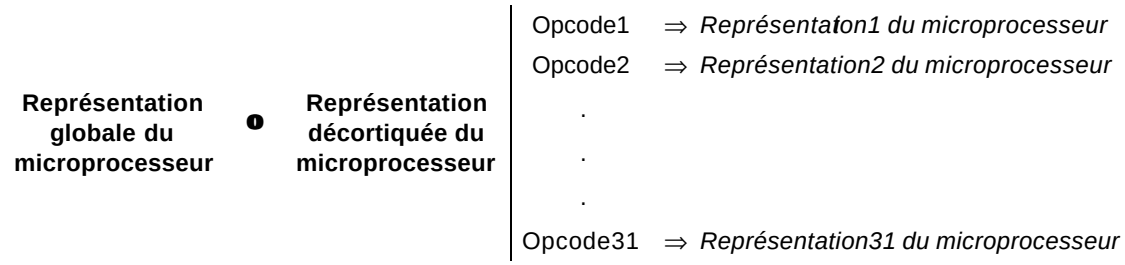
Instruction du Shifter

Ce groupe d'instructions concerne les opérations du Shifter ainsi que l'accès aux mémoires programme et données et le transfert de données inter registres. Les opcodes qui sont affectées à ce groupe sont :

Type de l'opcode	***Fonction***
12	Décalage (Shift) avec écriture/lecture de la mémoire données.
13	Décalage (Shift) avec écriture/lecture de la mémoire programme.
14	Décalage (Shift) avec transfert de données interne inter registres.
15	Décalage immédiat (Immediate Shift).
16	Décalage conditionnel.

5.1.3. Représentation du système

Notre manipulation des preuves nous a appris qu'il faut prendre en considération la connaissance approfondie du système lors de sa spécification. De ce fait, notre représentation du microprocesseur est *hiérarchique*. En d'autres termes, on n'avait pas considéré une seule représentation du microprocesseur, mais plutôt un ensemble de représentations dont chacune correspond à une classe d'instructions.



Représentation globale du microprocesseur

Cette représentation englobe tous les paramètres du microprocesseur. Elle considère toutes les entrées et les sorties. C'est la spécification complète de tout le microprocesseur.

Représentation décortiquée du microprocesseur

Cette représentation prend en considération la connaissance du fonctionnement du microprocesseur. En effet, au lieu de considérer toutes les unités lors de la vérification d'une instruction qui concerne l'ALU, il suffit de traiter les entrées de l'ALU uniquement. Car si une instruction concerne l'ALU elle n'agira que sur cette unité et par conséquent la représentation du MAC n'est pas donc nécessaire.

On se voit ainsi face à une nouvelle représentation du système sous forme d'un ensemble de sous-systèmes. Pour chaque instruction on considère une spécification à part du hardware et du fonctionnement. Dans cette spécification on n'a considéré que les paramètres qui sont affectés par l'instruction étudiée. Ainsi, on allège les preuves des différentes instructions.

5.2. Représentation des unités

Au lieu de considérer dans la spécification hardware du microprocesseur des spécifications hardware de ces différentes unités, on a plutôt considérer des spécifications du fonctionnement de ces unités. De ce fait, avant d'entamer la phase de preuve des instructions, on a abordé en premier lieu, pour chacune des unités du processeur, la preuve de la relation :

Spécification du hardware de l'unité $\Rightarrow$ *Spécification du fonctionnement de l'unité*

5.2.1. Représentation l'ALU

Les deux spécifications hardware et du fonctionnement de l'ALU ont été définies dans le chapitre précédent. Notre objectif dans ce cas est de faire preuve de la relation : *Spécification hardware de l'ALU* $\Rightarrow$ *Spécification du fonctionnement de l'ALU.*

Pour assurer cet objectif, on a décomposé la preuve en utilisant la preuve par étape de l'implication entre le hardware et le fonctionnement de l'unité. Cela est assuré en commençant en premier lieu par des sous-systèmes de l'ALU et en prenant en considération petit à petit les différents autres composants jusqu'à décrire toute l'unité.

Le Tableau 5.1 décrit les étapes qu'on a suivi pour faire la preuve de l'implication entre la spécification du hardware de l'ALU et la spécification de son fonctionnement. Dans l'étape 1, on a considéré un sous-système de l'ALU. Celui-ci correspond à l'unité de base de l'ALU ainsi qu'à la sélection de la première entrée Y. Puis, dans la deuxième étape, une fois cette preuve assurée, on a considéré la représentation du fonctionnement du sous-système précédent et on lui a rajouté les registres AY et AR. Dans la troisième étape, on a rajouté le registre AX et le multiplexeur à l'entrée X. Enfin, on a considéré toute l'ALU.

Etape	**Preuve**	
	Spécification hardware	***Spécification du fonctionnement***
1	Le multiplexeur à la sortie du registre AY. L'unité de base de l'ALU.	$\Rightarrow$ Sous-système 1 de l'ALU.
2	Sous-système 1 de l'ALU. Le multiplexeur à l'entrée du registre AY. Le registre AY. Le registre AF.	$\Rightarrow$ Sous-système 2 de l'ALU.
3	Sous-système 2 de l'ALU. Le multiplexeur à l'entrée du registre AX. Le registre AX.	$\Rightarrow$ Sous-système 3 de l'ALU.
4	Sous-système 3 de l'ALU. Le multiplexeur à l'entrée du registre AR. Le registre AR.	$\Rightarrow$ Toute l'ALU.

Tableau 5.1. Etapes de preuve de l'ALU.

Notre compréhension du principe de fonctionnement de l'ALU nous a permis de décortiquer l'implication entre son hardware et son fonctionnement. Ce qui nous a permis de manipuler des preuves assez simples. D'autre part, les durées des preuves étaient très courtes (ne dépassant pas 5s) pour chacune. C'est que notre choix des tactiques et de l'ordre de leurs applications était primordial pour garantir une résolution la plus directe que possible du système. Un exemple de preuve qu'on a utilisé est le suivant :

```
g `
    ! DMD_BUS_output AX0_input AX1_input AY0_input AY1_input
     AR_output AF_output
     AZ AN AC AV AS
     ALU_MUX_Xinput ALU_MUX_Yinput ALU_MUX_ARinput
     AX_input_select AY_input_select .
    ALU_second_sub_imp(
                DMD_BUS_output,
                AX0_input, AX1_input, AY0_input, AY1_input,
                AR_output, AF_output,
                AZ, AN, AC, AV, AS,
                ALU_MUX_Xinput, ALU_MUX_Yinput, ALU_MUX_ARinput,
                AX_input_select, AY_input_select
               )
    ==>
        ALU_second_sub_spec(
                    DMD_BUS_output,
                    AX0_input, AX1_input, AY0_input, AY1_input,
                    AR_output, AF_input, AF_output,
                    AZ, AN, AC, AV, AS,
                    ALU_MUX_Xinput, ALU_MUX_Yinput, ALU_MUX_ARinput,
                    AX_input_select, AY_input_select
                   )
`;
```

Déclaration de l'objectif

Notre objectif dans ce cas est de faire preuve de la relation suivante :

$$ALU\_second\_sub\_imp \Rightarrow ALU\_second\_sub\_spec$$

Où :

ALU_second_sub_imp : désigne l'implémentation du second sous-système de l'ALU.

ALU_second_sub_spec : désigne la spécification du second sous-système de l'ALU.

La preuve

La preuve qu'on a considérée se fait en quatre étapes :

1. La réécriture des spécifications des différents composants des deux systèmes ALU_second_sub_imp et ALU_second_sub_spec.

   REWRITE_TAC[ALU_second_sub_imp, ALU_second_sub_spec,

```
ALU_first_sub_spec, ALU_AX_reg, ALU_AY_reg, ALU_MUX]
```

2. La considération de toutes les valeurs possibles des paramètres booléens qui sont dans ce cas : ALU_MUX_Xinput, ALU_MUX_ARinput, ALU_MUX_Xinput, AX_input_select, AY_input_select et banc_select.

```
MAP_EVERY BOOL_CASES_TAC [`ALU_MUX_Xinput : bool``,
                          ``ALU_MUX_Yinput : bool`` ,
                          ``ALU_MUX_ARinput : bool``,
                          ``AX_input_select : bool`` ,
                          ``AY_input_select : bool``,
                          ``banc_select : bool``]
```

3. Décomposition de l'objectif en un ensemble de sous-objectifs (subgoals).

```
STRIP_TAC
```

4. Utilisation d'un algorithme de preuve pour démontrer les sous-objectifs obtenus.

```
bossLib.PROVE_TAC []
```

La preuve complète est :

```
e (
  REWRITE_TAC[ALU_second_sub_imp, ALU_second_sub_spec,
        ALU_first_sub_spec, ALU_AX_reg, ALU_AY_reg, ALU_MUX]
  THEN
     REPEAT GEN_TAC
  THEN
     MAP_EVERY BOOL_CASES_TAC [``ALU_MUX_Xinput : bool``,
                        ``ALU_MUX_Yinput : bool`` ,
                        ``ALU_MUX_ARinput : bool``,
                        ``AX_input_select : bool`` ,
                        ``AY_input_select : bool``,
                        ``banc_select : bool``]
  THEN
     REPEAT GEN_TAC
  THEN
     STRIP_TAC
  THEN
     ASM_REWRITE_TAC[ALU_second_sub_imp]
  THEN
     STRIP_TAC
  THEN
     bossLib.RW_TAC bossLib.arith_ss []
  THEN
     bossLib.PROVE_TAC []
);
```

La réponse du système

La réponse du système à la preuve précédente est la suivante :

```
Initial goal proved.
|- !DMD_BUS_output AX0_input AX1_input AY0_input AY1_input AR_output
    AF_output AZ AN AC AV AS ALU_MUX_Xinput ALU_MUX_Yinput
    ALU_MUX_ARinput AX_input_select AY_input_select.
   ALU_second_sub_imp
     (DMD_BUS_output,AX0_input,AX1_input,AY0_input,AY1_input,AR_output,
      AF_output,AZ,AN,AC,AV,AS,ALU_MUX_Xinput,ALU_MUX_Yinput,
      ALU_MUX_ARinput,AX_input_select,AY_input_select) ==>
      ALU_second_sub_spec
     (DMD_BUS_output,AX0_input,AX1_input,AY0_input,AY1_input,AR_output,
      AF_input,AF_output,AZ,AN,AC,AV,AS,ALU_MUX_Xinput,
      ALU_MUX_Yinput, ALU_MUX_ARinput,AX_input_select,AY_input_select)
: Goalstackpure.goalstack
```

Ainsi l'objectif est atteint et le système le déclare sous forme d'un nouveau théorème.

5.2.2. Représentation du MAC

Comme c'est le cas pour l'ALU, Les deux spécifications hardware et du fonctionnement du MAC ont été définies dans le chapitre précédent. L'objectif dans ce cas est de faire preuve de la relation :

Spécification hardware du MAC $\Rightarrow$ *Spécification du fonctionnement du MAC.*

La décomposition en sous-systèmes du MAC est décrite par le Tableau 5.2. Dans l'étape 1, on a considéré un sous-système du MAC qui correspond aux deux unités de base ADD/SUBSTRUCT et MULTIPLIER ainsi qu'aux deux multiplexeurs aux sorties des registres MX et MY. Dans la deuxième étape, on a considéré la représentation du fonctionnement du sous-système précédent et on lui a rajouté les registres MX, MY et MF. Dans la troisième étape, on a rajouté les trois multiplexeurs à l'entrée des registres MR0, MR1 et MR2. Dans la quatrième phase, on a rajouté les trois registres MR0, MR1 et MR2. Et enfin, dans la dernière phase, on a considéré toute l'unité MAC.

De même que pour l'ALU, nous avons utilisé notre connaissance du fonctionnement interne du MAC pour simplifier la manipulation de la preuve de l'implication entre les différents sous-systèmes considérés et leurs spécifications fonctionnelles. Les durées des preuves de chacune des implications décrites dans le Tableau 5.2 n'avaient pas dépassé 5 s. Cela est du à la prise en considération des particularités de chacune des unités lors de l'écriture des spécifications et lors de la sélection des tactiques (utilisées dans les preuves).

Etape	Preuve		
	Spécification hardware	⇒	***Spécification fonctionnelle***
1	Le multiplexeur à la sortie du registre MX. Le multiplexeur à la sortie du registre MY. L'unité MULTIPLIER. L'unité de ADD/SUNSTRACT.	⇒	Sous-système 1 du MAC.
2	Sous-système 1 du MAC. Le multiplexeur à l'entrée du registre MY. Les registres MX, MY et MF.	⇒	Sous-système 2 du MAC.
3	Sous-système 2 du MAC. Les multiplexeurs aux entrées des registres : MR0, MR1 et MR2.	⇒	Sous-système 3 du MAC.
4	Sous-système 3 du MAC. Les registres : MR0, MR1 et MR2.	⇒	Sous-système 4 du MAC.
5	Sous-système 4 du Shifter. Le multiplexeur à d'entrée du bus DMD.	⇒	Toute l'unité MAC.

Tableau 5.2. Etapes de preuve du MAC.

Un exemple de preuve qu'on a utilisé est le suivant :

Déclaration de l'objectif

Notre objectif dans ce cas est de faire preuve de la relation suivante :

$$MACSubSystemImplementation4 \Rightarrow MACImplementation.$$

Où :

MACSubSystemImplementation4 : désigne l'implémentation du quatrième sous-système du MAC.

MACImplementation : désigne la spécification du MAC (mais qui sera utilisée comme implémentation du MAC dans la preuve des instructions instruction du microprocesseur).

```
g `
 !
      (DMDBusOutput    :num -> word16) (PMDBusOutput  :num -> word24)
      (MX0Input        :num -> word16) (MX1Input      :num -> word16)
      (MY0Input        :num -> word16) (MY1Input      :num -> word16)
      (MFInput         :num -> word16)
      (MR0Input        :num -> word16) (MR1Input      :num -> word16)
      (MR2Input        :num -> word8)  (MV :num -> bool)
      (RBusInput       :num -> word16) (MACMYInputMUX   :num -> bool)
      (functionCode :num -> num) (t :num) (X :num -> word16) (Y :num -> word16)
      (R :num -> word40) (P :num -> word32)
      (R0 :num -> word16) (R1 :num -> word16) (R2 :num -> word8)
      (MXOutput   :num -> word16) (RBusOutput :num -> word16)
      (MYOutput   :num -> word16) (MFOutput   :num -> word16)
      (banc_select : num -> bool)
      (MR0Output :num -> word16) (MR1Output :num -> word16)
      (MR2Output :num -> word8).

MACSubSystemImplementation4
         (DMDBusOutput,PMDBusOutput,MX0Input,MX1Input,MY0Input,MY1Input,
          MFInput,MR0Input,MR1Input,MR2Input,MV,RBusInput,MACMYInputMUX,
          MACMYRegSelect, functionCode,t,X,Y,R,P,R0,R1,R2,
          RBusOutput,MFOutput,banc_select, MR0Output,MR1Output,MR2Output)
    ==>
       MACImplementation
(DMDBusOutput,PMDBusOutput,MX0Input,MX1Input,MY0Input,MY1Input,
          MFInput,MR0Input,MR1Input,MR2Input,MV,R BusInput,MACMYInputMUX,
          MACMYRegSelect, functionCode,t,X,Y,R,P,R0,R1,R2,
          RBusOutput,MFOutput,banc_select, MR0Output,MR1Output,MR2Output)
`;
```

La preuve

La preuve qu'on a considérée consiste en trois étapes :

1. La réécriture des spécifications des différents composants des deux systèmes MACImplementation et MACSubSystemImplementation4 :

   ```
   REWRITE_TAC [MACSubSystemImplementation4, MACImplementation]
   ```

2. La considération de toutes les valeurs possibles des paramètres booléens qui sont dans ce cas : MACMRoutputMUX0 et MACMRoutputMUX0 :

   ```
   MAP_EVERY BOOL_CASES_TAC [
                        ``(MACMRoutputMUX0 (t:num))  : bool``,
                        ``(MACMRoutputMUX1 (t:num))  : bool``
   ```

3. Terminaison de la preuve utilisant les conversions :

   ```
   CONV_TAC (simpLib.SIMP_CONV HOLSimps.hol_ss []).
   ```

La preuve complète est :

```
e(
    REWRITE_TAC [MACSubSystemImplementation4, MACImplementation]
  THEN
    REPEAT GEN_TAC
  THEN
    CONV_TAC (simpLib.SIMP_CONV HOLSimps.hol_ss [])
  THEN
    MAP_EVERY BOOL_CASES_TAC [
                    ``(MACMRoutputMUX0 (t:num)) : bool``,
                    ``(MACMRoutputMUX1 (t:num)) : bool``
                   ]
  THEN
    CONV_TAC (simpLib.SIMP_CONV HOLSimps.hol_ss [])
);
```

5.2.3. Représentation du Shifter

Pour traiter le Shifter nous avons suivi la même approche comme pour les cas des deux unités ALU et MAC. Notre dans ce cas est de faire preuve de l'implication suivante :

Spécification hardware du Shifter ⇒ Spécification du fonctionnement du Shifter.

La décomposition en sous-systèmes du Shifter est décrite par le Tableau 5.3. Dans l'étape 1, on a considéré un sous-système du Shifter composé des deux unités de base SHIFTER/ARRAY et OR/PASS ainsi qu'au registre SI et au multiplexeur se trouvant à la sortie de ce registre. Dans la deuxième étape, on a considéré la représentation du fonctionnement du sous-système précédent et on lui a rajouté les deux registres de sortie SR1 et SR2 ainsi que les multiplexeurs qui se trouvent à leurs entrées. Dans la troisième étape, on a rajouté les composants formant le module qui génère la valeur du décalage à effectuer (cela correspond aux éléments suivants : registre SE, le multiplexeur à l'entrée de SE, l'unité NEGATE et le multiplexeur à la sortie de l'unité NEGATE). Dans la quatrième phase, on a rajouté le module responsable de la détermination de l'exposé (cela correspond aux éléments suivants : registre SB, le multiplexeur à l'entrée de SB, l'unité COMPARE et l'unité EXPONENT DETECTOR). Et enfin, dans la dernière phase, on a considéré toute l'unité Shifter.

De même que pour le cas des unités ALU et MAC, nous nous sommes référés à notre connaissance du fonctionnement de l'unité Shifter pour simplifier la manipulation de la preuve de l'implication entre les différents sous-systèmes considérés et leurs spécifications fonctionnelles. Les durées des preuves de chacune des implications décrites dans le Tableau 5.3 n'avaient pas dépassé 5 s. Cela est du à deux facteurs qui sont :

- la prise en considération des particularités de chacune des unités lors de l'écriture des spécifications
- la sélection des tactiques (utilisées dans les preuves) les plus adaptées au cas de l'unité Shifter et l'ordre de leurs applications.

Etape	Preuve		
	Spécification hardware	⇒	***Spécification fonctionnelle***
1	Le registre SI. Le multiplexeur à la sortie du registre SI. L'unité SHIFTER/ARRAY. L'unité OR/PASS.	⇒	Sous-système 1 du Shifter.
2	Sous-système 1 du Shifter. Les multiplexeurs aux entrées des registres SR1 et SR2. Les registres SR1 et SR2.	⇒	Sous-système 2 du Shifter.
3	Sous-système 2 du Shifter. Le multiplexeur à l'entrée du registre SE. Le registre SE. L'unité NEGATE. Le multiplexeur à la sortie de l'unité NEGATE. Le multiplexeur à l'entrée C de l'unité SHIFTER/ARRAY.	⇒	Sous-système 3 du Shifter.
4	Sous-système 3 du Shifter. Le multiplexeur à l'entrée du registre SB. Le registre SB. L'unité COMPARE. L'unité EXPONEN DETECTOR.	⇒	Sous-sys tème 4 du Shifter.
5	Sous-système 4 du Shifter. Le multiplexeur à d'entrée du bus DMD.	⇒	Toute l'unité Shifter.

Tableau 5.3. Etapes de preuve du Shifter.

5.3. Vérification des instructions

Comme on l'avait indiqué précédemment, on va considérer une spécification à part pour chaque instruction du microprocesseur. De ce fait, pour chacune des instructions qui suivent, on va définir à chaque fois une représentation du processeur en considérant uniquement les paramètres qui rentrent en jeux dans l'instruction étudiée.

5.3.1. Instructions d'accès à la mémoire

5.3.1.1. Ecriture immédiate de données dans la mémoire

Cette instruction permet une écriture immédiate des données dans la mémoire. La sélection entre les deux générateurs d'adresses est spécifiée par le paramètre booléen **G**. L'adresse d'écriture est définie par les deux paramètres **I** et **M**, qui spécifient respectivement le registre *Index* et le registre *Modify* du générateur d'adresse sélectionné.

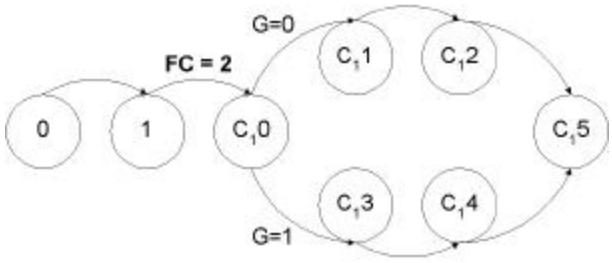


Figure 5.1. Représentation du processeur : instruction Ecriture immédiate de données dans la mémoire.

Etat	*Description*
0	Chargement de l'instruction.
1	Décodage de la fonction à réaliser. La valeur de cette fonction est stockée dans le paramètre **FC** (FunctionCode).
$C_1$0	Décodage du paramètre booléen **G** qui spécifie lequel des générateurs de registres sera utilisé pour définir l'adresse mémoire de stockage des données.
$C_1$1	Génération des contrôles de lecture du générateur1.
$C_1$2	Lecture des registres **I** et **M** sélectionnés du générateur 1.
$C_1$3	Génération des contrôles de lecture du générateur2.
$C_1$4	Lecture des registres **I** et **M** sélectionnés du générateur 2.
$C_1$5	Stockage des données selon les valeurs des deux registres **I** et **M**.

Tableau 5.4. Etapes de l'instruction Ecriture immédiate de données dans la mémoire.

La représentation du microprocesseur dans ce cas est définie par la structure décrite sur la Figure 5.1. Les étapes d'exécution de l'instruction sont définies dans le Tableau 5.4. Les deux premiers états concernent la lecture de l'instruction et le décodage de la fonction en cours. Les états C_10, C_11, C_12, C_13, C_14 et C_15 sont spécifiques à la classe 1.

Notre représentation du microprocesseur au cours de l'exécution de cette instruction prend en considération les registres et les composants concernés uniquement. Dans la représentation hardware du processeur, dans ce cas, chacun des états définie l'opération à réaliser. Pour le cas de l'état 1, par exemple, c'est le séquenceur de programme qui spécifie la fonction à exécuter selon la valeur du paramètre **FC** (FunctionCode).

Les deux spécifications hardware et fonctionnelle du processeur sont définies dans le cadre de cette instruction par :

Spécification Hardware

Séquenceur de programme (ce qui correspond aux états : 1, C_10 et C_13) avec :

> **paramètres d'entrée** : contrôle de sélection du générateur d'adresse **G**, contrôle des registres **I** et **M** qui seront utilisés pour accéder à la mémoire.
> **paramètres de sortie** : paramètres de contrôle d'accès à la mémoire.

Générateur d'adresse (ce qui correspond aux états : C_12 ou C_14 selon le générateur d'adresse à utiliser) avec :

> **paramètres d'entrée** : contrôle des registres **I** et **M** qui seront utilisés pour accéder à la mémoire.
> **paramètres de sortie** : contenus des registres **I** et **M** qui seront utilisés pour accéder à la mémoire.

Accès à la mémoire (ce qui correspond à l'état C_12) avec :

> **paramètres d'entrée** : paramètre de contrôle d'accès à la mémoire (spécifiés par le séquenceur de programme), données à écrire dans la mémoire (spécifiées directement par la mémoire).

Spécification du fonctionnement

De point de vue fonctionnel, l'objectif est de stocker les données indiquées par l'instruction dans l'adresse indiquée par les deux registres **I** et **M** dans la mémoire données ou programme (selon le paramètre **G**).

La vérification

Dans cette phase, on s'est intéressé à la démonstration de l'implication entre la spécification hardware du processeur et sa spécification fonctionnelle. En utilisant notre connaissance du processeur et comme on avait procédé dans la vérification des unités ALU, MAC et Shifter, on a décortiqué la preuve principale. En effet, on a commencé par le décodage du générateur d'adresse qui sera

utilisé, en suite, on s'est intéressé à la lecture des deux registres d'adressage **I** et **M**, et enfin, on a considéré le stockage des données dans la mémoire.

5.3.1.2. Ecriture immédiate d'une adresse dans la mémoire

Spécification Hardware

Séquenceur de programme (ce qui correspond aux états : 1, C_16, C_17 et C_18 sur la Figure 5.2) avec :

paramètres d'entrée : la classe de registres définie par le paramètre **RGP**, le registre (le paramètre **REG**) de la classe spécifiée précédemment et qui servira pour la lecture ou l'écriture de la mémoire, le paramètre de contrôle du sens d'accès à la mémoire (**D**).
paramètres de sortie : le registre qui servira pour la lecture de (ou encore l'écriture dans) la mémoire, les paramètres de contrôle d'accès à la mémoire.

Accès à la mémoire (ce qui correspond à l'état C_12) avec :

paramètres d'entrée : paramètre de contrôle d'accès à la mémoire (spécifiés par le séquenceur de programme), le registre qui servira pour la lecture de (ou encore l'écriture dans) la mémoire.

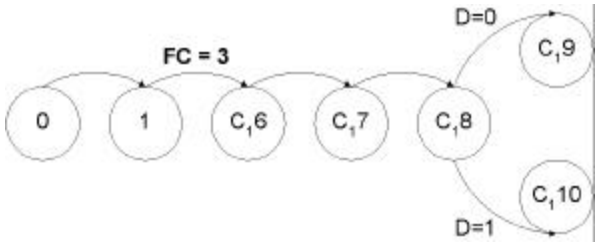


Figure 52. Représentation du processeur : instruction Ecriture immédiate d'une adresse dans la mémoire.

Spécification du fonctionnement

Selon la valeur du paramètre **D**, cette instruction permet :

L'écriture du contenu du registre défini par le paramètre **REG** de la classe **RGP** dans l'adresse **ADDR** définie par l'instruction,

ou bien :

Le chargement du registre défini par le paramètre **REG** de la classe **RGP** par le contenu de l'adresse **ADDR** définie par l'instruction.

La vérification

Dans le processus de vérification de cette instruction, on a considéré deux étapes consécutives qui sont :

1. Le décodage du registre qui servira pour la lecture de (ou encore l'écriture dans) la mémoire. Dans cette étape on a prouvé que les valeurs des registres d'entrées des unités sont identiques pour les deux représentation hardware et du fonctionnement.
2. Le stockage ou bien la lecture des données de la mémoire. Cela revient à faire preuve que la mémoire est chargée par la même données en considérant la représentation hardware ou du fonctionnement.

Etat	***Description***
C_16	Spécification de la classe de registres définie par le paramètre **RGP**.
C_17	Spécification du registre définie par le paramètre **REG**.
C_18	Décodage du paramètre booléen **D** qui spécifie le sens d'accès à la mémoire : lecture ou écriture.
C_19	Stockage des données, spécifiées par l'adresse mémoire définie par l'instruction, dans le registre sélectionné.
C_110	Ecriture du contenu du registre sélectionné dans l'adresse mémoire définie par l'instruction.

Tableau 5.5. Etapes de l'instruction Ecriture immédiate d'une adresse dans la mémoire.

5.3.2. Instructions de manipulation des registres

5.3.2.1. Chargement immédiat d'un registre données

Spécification Hardware

Séquenceur de programme (ce qui correspond aux états : 1, C_20 sur la Figure 5.3) avec :

paramètres d'entrée : le code du registre (le paramètre **REG**) destination des données.
paramètres de sortie : paramètre de contrôle de l'unité contenant le registre concerné (ALU, MAC ou Shifter).

Les unités ALU, MAC et Shifter (ce qui correspond aux états : 1, C_21 sur la Figure 5.3) avec :

paramètres d'entrée : les contrôles de l'unité, la valeur **DATA** des données à stocker dans le registre spécifié.

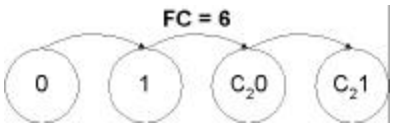


Figure 53. Représentation du processeur : instruction Chargement immédiat d'un registre données.

Spécification du fonctionnement

Cette instruction permet de charger le registre défini par le paramètre **REG** par une valeur spécifiée par l'instruction.

La vérification

Dans le processus de vérification de cette instruction, on a considéré deux étapes consécutives qui sont :

1. Le décodage du registre qui servira pour le stockage des données définies par l'instruction.
2. Le stockage des données dans le registre.

Etat	*Description*
C_20	Spécification du registre défini par le paramètre **REG**.
C_21	Stockage des données, lues directement par l'instruction, dans le registre sélectionné.

Tableau 5.6. Etapes de l'instruction Chargement immédiat d'un registre données.

5.3.2.2. Chargement immédiat d'un registre non-données

Spécification Hardware

Séquenceur de programme (ce qui correspond aux états : 1, C_22, C_23 sur la Figure 5.4) avec :

paramètres d'entrée : le code de la classe de registre concernée par l'instruction (paramètre **RGP**), le code du registre (le paramètre **REG**) destination des données.

paramètres de sortie : paramètre de contrôle de l'unité contenant le registre concerné (ALU, MAC, Shifter, DAG1, DAG2 ou encore le ProgramSequencer).

Les unités ALU, MAC, Shifter, DAG1, DAG2 ou ProgramSequencer (ce qui correspond aux états : 1, $C_2$4 sur la Figure 5.4) avec :

paramètres d'entrée : les contrôles de l'unité, la valeur **DATA** des données à stocker dans le registre spécifié.

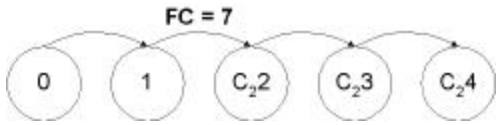


Figure 5.4. Représentation du processeur : instruction Chargement immédiat d'un registre non-données.

Spécification du fonctionnement

Cette instruction permet de charger le registre de la classe **RGP** et défini par le paramètre **REG** par une valeur spécifiée par l'instruction.

La vérification

Dans le processus de vérification de cette instruction, on a considéré deux étapes consécutives qui sont :

1. Le décodage du registre qui servira pour le stockage des données définies par l'instruction.
2. Le stockage des données dans le registre.

La preuve de cette instruction est similaire à la précédente de point de vue structurelle. Mais, elle est plus complexe car elle intègre plus d'unités que la précédente.

Etat	*Description*
$C_2$2	Spécification de la classe de registres définie par le paramètre **RGP**.
$C_2$3	Spécification du registre définie par le paramètre **REG**.
$C_2$4	Stockage des données, lues directement par l'instruction, dans le registre sélectionné.

Tableau 5.7. Etapes de l'instruction Chargement immédiat d'un registre non-données.

5.3.2.3. Transfert de données inter registres

Spécification Hardware

Séquenceur de programme (ce qui correspond aux états : 1, $C_2$5, $C_2$6, $C_2$7, $C_2$8 sur la Figure 5.5) avec :

paramètres d'entrée : le code de la classe de registre contenant le registre destination (paramètre **DST RGP**), le code du registre destination (le paramètre **Dest REG**) destination, le code de la classe de registre contenant le registre source (paramètre **SRC RGP**), le code du registre destination (le paramètre **Source REG**) source.
paramètres de sortie : paramètre de contrôle de l'unité contenant le registre source (ALU, MAC, Shifter, DAG1, DAG2 ou encore le ProgramSequencer), paramètre de contrôle de l'unité contenant le registre destination (ALU, MAC, Shifter, DAG1, DAG2 ou encore le ProgramSequencer).

Les unités ALU, MAC, Shifter, DAG1, DAG2 ou ProgramSequencer (ce qui correspond aux états : 1, $C_2$9 sur la Figure 5.5) avec :

paramètres d'entrée : les contrôles de l'unité indiquant le registre source, les contrôles de l'unité indiquant le registre destination.

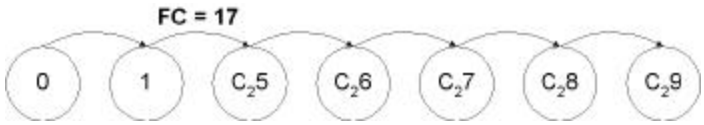


Figure 5.5. Représentation du processeur : instruction Transfert de données inter registres.

Spécification du fonctionnement

Cette instruction permet de charger un registre destination de la classe **DST RGP** et défini par le paramètre **Dest REG** par le contenu d'un registre source de la classe **SRC RGP** et défini par le paramètre **Source REG**.

La vérification

Dans le processus de vérification de cette instruction, on a considéré trois étapes consécutives qui sont :

1. Le décodage du registre source.
2. Le décodage du registre destination
3. Le transfert de données entre les deux registres source et destination.

Etat	*Description*
$C_2$5	Spécification de la classe de registres destination définie par le paramètre **DST RGP**.
$C_2$6	Spécification du registre destination définie par le paramètre **Dest REG**.
$C_2$7	Spécification de la classe de registres source définie par le paramètre **SRC RGP**.
$C_2$8	Spécification du registre destination définie par le paramètre **Source REG**.
$C_2$9	Transfert des données entre les deux registres source et destination.

Tableau 5.8. Etapes de l'instruction Transfert de données inter registres.

5.3.3. Instructions de l'ALU

5.3.3.1. ALU avec accès en lecture de la mémoire

Spécification Hardware

Séquenceur de programme (ce qui correspond aux états : 1, $C_3$0, $C_3$1, $C_3$2, $C_3$3, $C_3$6 et $C_3$7 sur la Figure 5.6) avec :

paramètres d'entrée : le code de sélection du registre de lecture de la mémoire programme (paramètre **PD**), le code de sélection du registre de lecture de la mémoire données (paramètre **DD**), les codes des sélection des registres **I** et **M** d'accès à la mémoire programme (paramètres **PMI** et **PMM**), les codes de sélection des registres **I** et **M** d'accès à la mémoire données (contrôles **DMI** et **DMM**), les codes de contrôle de l'entrée **X** de l'ALU (paramètre **Xop**), les codes de contrôle de l'entrée **Y** de l'ALU (paramètre **Yop**) et le code de la fonction à exécuter par l'ALU (paramètre **AMF**).
paramètres de sortie : paramètre de contrôle de l'accès à la mémoire, paramètre de contrôle de l'unité ALU.

Accès à la mémoire (ce qui correspond à l'état $C_8$4 et $C_8$5 sur la Figure 5.6) avec :

paramètres d'entrée : paramètres de contrôle d'accès à la mémoire programme en lecture et paramètres de contrôle d'accès à la mémoire données en lecture.

L'unité ALU (ce qui correspond aux états : $C_3$8, $C_2$9 sur la Figure 5.6) avec :

paramètres d'entrée : les contrôles de l'unité indiquants les valeurs des entrées **X** et **Y**, le code de la fonction à exécuter par l'ALU (paramètre **AMF**).

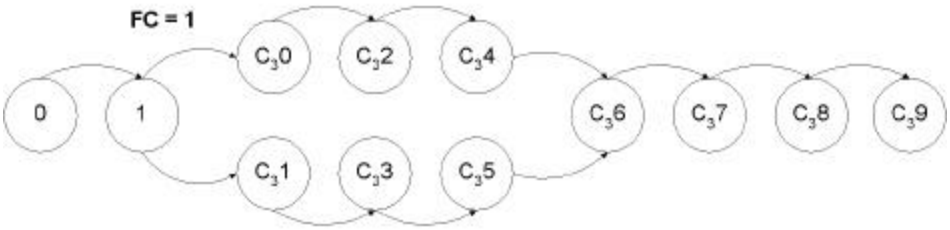


Figure 5.6. Représentation du processeur : instruction ALU avec accès en lecture de la mémoire.

Etat	*Description*
$C_3$0	Sélection des codes des registres destination pour la lecture des données de la mémoire programme (contrôle **PD**).
$C_3$1	Sélection des codes des registres destination pour la lecture des données de la mémoire données (contrôle **DD**).
$C_3$2	Sélection des registres **I** et **M** d'accès à la mémoire programme (contrôles **PMI** et **PMM**).
$C_3$3	Sélection des registres **I** et **M** d'accès à la mémoire données (contrôles **DMI** et **DMM**).
$C_3$4	Lecture des données de la mémoire programme.
$C_3$5	Lecture des données de la mémoire données.
$C_3$6	Génération des codes de contrôle de l'entrée **X** de l'ALU (paramètre **Xop**).
$C_3$7	Génération des codes de contrôle de l'entrée **Y** de l'ALU (paramètre **Yop**).
$C_3$8	Décodage de la fonction à exécuter par l'ALU (paramètre **AMF**).
$C_3$9	Génération des codes de contrôle de l'ALU.

Tableau 5.9. Etapes de l'instruction ALU avec accès en lecture de la mémoire.

Spécification du fonctionnement

Cette instruction permet d'exécuter trois tâches qui sont :

- charger un premier registre défini par le code **PD** par des données de la mémoire programme,
- charger un second registre défini par le code **DD** par des données de la mémoire données,
- exécuter une fonction de l'ALU définie par le code **AMF**.

La vérification

Dans le processus de vérification de cette instruction, on a considéré quatre étapes consécutives qui sont :

1. Le décodage du registre destination des données de la mémoire programme.
2. Le décodage du registre destination des données de la mémoire données.
3. Le transfert des données de la mémoire aux registres destinations.
4. La fonction réalisée par l'ALU.

5.3.3.2. ALU avec accès lecture/écriture mémoire données

Spécification Hardware

Séquenceur de programme (ce qui correspond aux états : 1, $C_3$10, $C_3$11, $C_3$12, $C_3$6, $C_3$7 et $C_3$16 sur la Figure 5.7) avec :

paramètres d'entrée : les codes des registres destination pour la lecture des données de la mémoire (contrôle **DREG**), le code du générateur d'adresse nécessaire pour l'accès à la mémoire (contrôle **G**), les codes de sélection des registres **I** et **M** d'accès à la mémoire données (contrôles **I** et **M**), les codes de contrôle de l'entrée **X** de l'ALU (paramètre **Xop**), les codes de contrôle de l'entrée **Y** de l'ALU (paramètre **Yop**), le code de la fonction à exécuter par l'ALU (paramètre **AMF**) et code de contrôle du registre de stockage des données (paramètre **Z**).
paramètres de sortie : paramètres de contrôle du générateur d'adresse qui va être utilisé pour adresser la mémoire, paramètres de contrôle de l'accès à la mémoire, paramètres de contrôle de l'unité ALU.

Générateur d'adresse (ce qui correspond à l'état $C_3$13 sur la Figure 5.7) avec :

paramètres d'entrée : contrôle des registres **I** et **M** qui seront utilisés pour accéder à la mémoire.
paramètres de sortie : contenus des registres **I** et **M** qui seront utilisés pour accéder à la mémoire.

Accès à la mémoire (ce qui correspond à l'état $C_3$14 et $C_3$15 sur la Figure 5.7) avec :

paramètres d'entrée : paramètres de contrôle d'accès à la mémoire données.

L'unité ALU (ce qui correspond aux états : $C_3$8, $C_2$9 sur la Figure 5.7) avec :

paramètres d'entrée : les contrôles de l'unité indiquants les valeurs des entrées **X** et **Y**, le code de la fonction à exécuter par l'ALU (paramètre **AMF**).

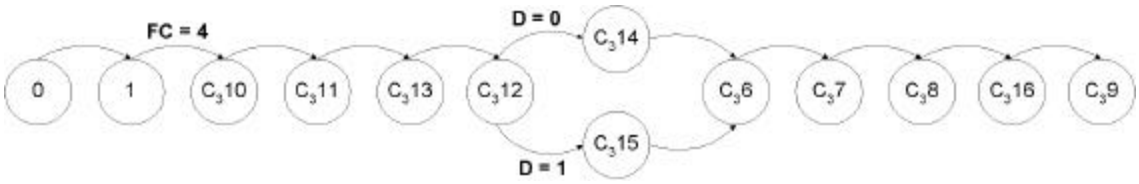


Figure 5.7. Représentation du processeur : instruction ALU avec accès lecture/écriture mémoire données.

Etat	*Description*
$C_3$10	Sélection des codes des registres destination pour la lecture des données de la mémoire (contrôle **DREG**).
$C_3$11	Sélection du générateur d'adresse nécessaire pour l'accès à la mémoire (contrôle **G**).
$C_3$12	Sélection du sens d'accès à la mémoire : lecture ou écriture (contrôle **D**).
$C_3$13	Sélection des registres **I** et **M** d'accès à la mémoire (contrôles **I** et **M**).
$C_3$14	Stockage des données dans le registre sélectionné, de la mémoire données, selon l'adresse spécifiée par les registres **I** et **M**.
$C_3$15	Ecriture des données, contenues dans le registre sélectionné, dans la mémoire données, selon l'adresse spécifiée par les registres **I** et **M**.
$C_3$6	Génération des codes de contrôle de l'entrée **X** de l'ALU (paramètre **Xop**).
$C_3$7	Génération des codes de contrôle de l'entrée **Y** de l'ALU (paramètre **Yop**).
$C_3$8	Décodage de la fonction à exécuter par l'ALU (paramètre **AMF**).
$C_3$16	Génération du code de contrôle du registre de stockage des données (paramètre **Z**).
$C_3$9	Génération des codes de contrôle de l'ALU.

Tableau 5.10. Etapes de l'instruction ALU avec accès lecture/écriture mémoire données.

Spécification du fonctionnement

Cette instruction permet d'exécuter deux tâches qui sont :

- Selon la valeur du paramètre **D**, cette instruction permet :

  L'écriture du contenu du registre défini par le paramètre **DREG** dans l'adresse de la mémoire données définie par les registres **I** et **M** du générateur d'adresse (1 ou 2 selon la valeur du paramètre **G**),

ou bien :

Le chargement du registre défini par le paramètre **DREG** par le contenu de l'adresse de la mémoire données définie par les registres **I** et **M** du générateur d'adresse (1 ou 2 selon la valeur du paramètre **G**).

- exécuter une fonction de l'ALU définie par le code **AMF**.

La vérification

Dans le processus de vérification de cette instruction, on a considéré trois étapes consécutives qui sont :

1. Le décodage du registre à utiliser pour la lecture ou l'écriture des données.
2. Le décodage des registres qui serviront pour adresser la mémoire données.
3. Le transfert des données de la mémoire données au registre destination.
4. La fonction réalisée par l'ALU.

5.3.3.3. ALU avec accès lecture/écriture mémoire programme

Spécification Hardware

Séquenceur de programme (ce qui correspond aux états : 1, $C_3$10, $C_3$12, $C_8$6, $C_3$7 et $C_3$16 sur la Figure 5.8) avec :

paramètres d'entrée : les codes des registres destination pour la lecture des données de la mémoire (contrôle **DREG**), les codes de sélection des registres **I** et **M** d'accès à la mémoire données (contrôles **I** et **M**), les codes de contrôle de l'entrée **X** de l'ALU (paramètre **Xop**), les codes de contrôle de l'entrée **Y** de l'ALU (paramètre **Yop**), le code de la fonction à exécuter par l'ALU (paramètre **AMF**) et code de contrôle du registre de stockage des données (paramètre **Z**).
paramètres de sortie : paramètres de contrôle du générateur d'adresse qui va être utilisé pour adresser la mémoire, paramètres de contrôle de l'accès à la mémoire, paramètres de contrôle de l'unité ALU.

Générateur d'adresse (ce qui correspond à l'état $C_3$13 sur la Figure 5.8) avec :

paramètres d'entrée : contrôle des registres **I** et **M** qui seront utilisés pour accéder à la mémoire.
paramètres de sortie : contenus des registres **I** et **M** qui seront utilisés pour accéder à la mémoire.

Accès à la mémoire (ce qui correspond à l'état $C_8$17 et $C_3$18 sur la Figure 5.8) avec :

paramètres d'entrée : paramètres de contrôle d'accès à la mémoire données.

L'unité ALU (ce qui correspond aux états : $C_3 8$, $C_2 9$ sur la Figure 5.8) avec :

paramètres d'entrée : les contrôles de l'unité indiquants les valeurs des entrées **X** et **Y**, le code de la fonction à exécuter par l'ALU (paramètre **AMF**).

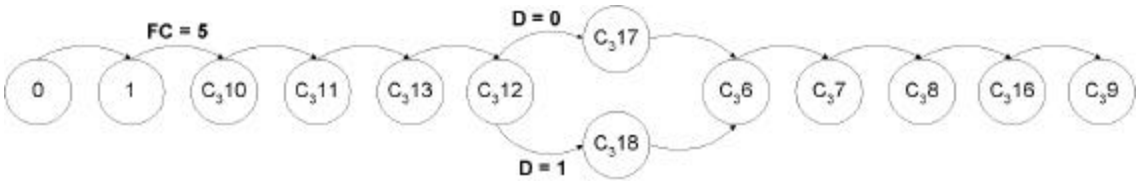


Figure 58. Représentation du processeur : instruction de ALU avec accès lecture/écriture mémoire programme.

Spécification du fonctionnement

Cette instruction permet d'exécuter deux tâches qui sont :

- Selon la valeur du paramètre **D**, cette instruction permet :

  L'écriture du contenu du registre défini par le paramètre **DREG** dans l'adresse de la mémoire programme définie par les registres **I** et **M** du générateur d'adresse 1,

  ou bien :

  Le chargement du registre défini par le paramètre **DREG** par le contenu de l'adresse de la mémoire programme définie par les registres **I** et **M** du générateur d'adresse 1.

- exécuter une fonction de l'ALU définie par le code **AMF**.

La vérification

Dans le processus de vérification de cette instruction, on a considéré trois étapes consécutives qui sont :

1. Le décodage du registre à utiliser pour la lecture ou l'écriture des données.
2. Le décodage des registres qui serviront pour adresser la mémoire programme.
3. Le transfert des données de la mémoire programme au registre destination.
4. La fonction réalisée par l'ALU.

La preuve de cette instruction est similaire à la preuve de l'instruction précédente. La seule différence concerne le type de mémoire adresser.

Etat	*Description*
$C_3$10	Sélection des codes des registres destination pour la lecture des données de la mémoire (contrôle **DREG**).
$C_3$12	Sélection du sens d'accès à la mémoire : lecture ou écriture (contrôle **D**).
$C_3$13	Sélection des registres **I** et **M** d'accès à la mémoire (contrôles **I** et **M**).
$C_3$17	Stockage des données dans le registre sélectionné, de la mémoire programme, selon l'adresse spécifiée par les registres **I** et **M**.
$C_3$18	Ecriture des données, contenues dans le registre sélectionné, dans la mémoire programme, selon l'adresse spécifiée par les registres **I** et **M**.
$C_3$6	Génération des codes de contrôle de l'entrée **X** de l'ALU (paramètre **Xop**).
$C_3$7	Génération des codes de contrôle de l'entrée **Y** de l'ALU (paramètre **Yop**).
$C_3$8	Décodage de la fonction à exécuter par l'ALU (paramètre **AMF**).
$C_3$16	Génération du code de contrôle du registre de stockage des données (paramètre **Z**).
$C_3$9	Génération des codes de contrôle de l'ALU.

Tableau 5.11. Etapes de l'instruction de ALU avec accès lecture/écriture mémoire programme.

5.3.3.4. ALU avec transfert de données inter registres

Spécification Hardware

Séquenceur de programme (ce qui correspond aux états : 1, $C_3$19, $C_3$20, $C_3$21, $C_3$6, $C_3$7 et $C_3$8 sur la Figure 5.9) avec :

paramètres d'entrée : le code du registre destination (contrôle **Dest REG**), le code du registre source (contrôle **Source REG**), les codes de contrôle de l'entrée **X** de l'ALU (paramètre **Xop**), les codes de contrôle de l'entrée **Y** de l'ALU (paramètre **Yop**), le code de la fonction à exécuter par l'ALU (paramètre **AMF**) et code de contrôle du registre de stockage des données (paramètre **Z**).

paramètres de sortie : paramètres de contrôle des unités concernées par les registres source et destination, paramètres de contrôle de l'unité ALU.

Les unités ALU, MAC, Shifter, DAG1, DAG2 ou ProgramSequencer (ce qui correspond aux états : 1, $C_2$9 sur la Figure 5.9) avec :

paramètres d'entrée : les contrôles de l'unité indiquant le registre source, les contrôles de l'unité indiquant le registre destination.

L'unité ALU (ce qui correspond aux états : $C_3$8, $C_2$9 sur la Figure 5.9) avec :

paramètres d'entrée : les contrôles de l'unité indiquants les valeurs des entrées **X** et **Y**, le code de la fonction à exécuter par l'ALU (paramètre **AMF**).

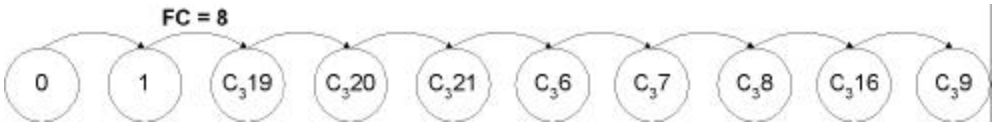


Figure 59. Représentation du processeur : instruction de ALU avec transfert de données inter registres.

Spécification du fonctionnement

Cette instruction permet d'exécuter deux tâches qui sont :

- Affectation du contenu du registre source dans le registre destination.
- Exécuter une fonction de l'ALU définie par le code **AMF**.

Etat	*Description*
$C_3$19	Spécification du registre destination définie par le paramètre **Dest REG**.
$C_3$20	Spécification du registre source définie par le paramètre **Source DREG**.
$C_4$21	Affectation du contenu du registre source dans le registre destination.
$C_3$6	Génération des codes de contrôle de l'entrée **X** de l'ALU (paramètre **Xop**).
$C_3$7	Génération des codes de contrôle de l'entrée **Y** de l'ALU (paramètre **Yop**).
$C_3$8	Décodage de la fonction à exécuter par l'ALU (paramètre **AMF**).
$C_3$16	Génération du code de contrôle du registre de stockage des données (paramètre **Z**).
$C_3$9	Génération des codes de contrôle de l'ALU.

Tableau 5.12. Etapes de l'instruction de ALU avec transfert de données inter registres.

La vérification

Dans le processus de vérification de cette instruction, on a considéré quatre étapes consécutives qui sont :

1. Le décodage du registre source.
2. Le décodage du registre destination.
3. Le transfert de données entre les deux registres source et destination.

4. La fonction réalisée par l'ALU.

5.3.3.5. ALU conditionnelle

Spécification Hardware

Séquenceur de programme (ce qui correspond aux états : 1, $C_3$22, $C_3$6, $C_3$7 et $C_3$8 sur la Figure 5.10) avec :

paramètres d'entrée : le code d'identification de la condition à vérifier (paramètre **COND**), les codes de contrôle de l'entrée **X** de l'ALU (paramètre **Xop**), les codes de contrôle de l'entrée **Y** de l'ALU (paramètre **Yop**), le code de la fonction à exécuter par l'ALU (paramètre **AMF**) et code de contrôle du registre de stockage des données (paramètre **Z**).

paramètres de sortie : paramètre d'identification du paramètre de la condition, paramètres de contrôle de l'unité ALU.

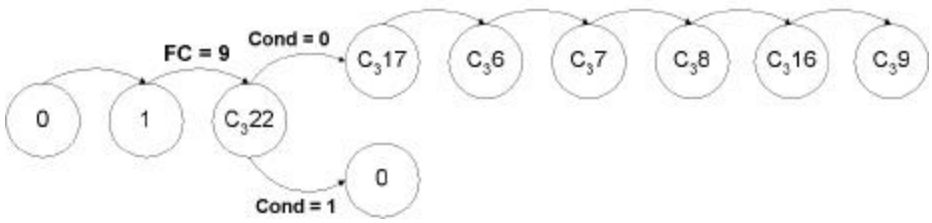


Figure 510. Représentation du processeur : instruction ALU conditionnelle.

Etat	*Description*
$C_3$22	Vérification de la condition d'exécution de l'instruction spécifiée par le paramètre **COND**.
$C_3$6	Génération des codes de contrôle de l'entrée **X** de l'ALU (paramètre **Xop**).
$C_3$7	Génération des codes de contrôle de l'entrée **Y** de l'ALU (paramètre **Yop**).
$C_3$8	Décodage de la fonction à exécuter par l'ALU (paramètre **AMF**).
$C_3$16	Génération du code de contrôle du registre de stockage des données (paramètre **Z**).
$C_3$9	Génération des codes de contrôle de l'ALU.

Tableau 5.13. Etapes de l'instruction ALU conditionnelle.

L'unité ALU (ce qui correspond aux états : $C_3$8, $C_2$9 sur la Figure 5.10) avec :

paramètres d'entrée : les contrôles de l'unité indiquants les valeurs des entrées **X** et **Y**, le code de la fonction à exécuter par l'ALU (paramètre **AMF**).

Spécification du fonctionnement

Exécuter une fonction de l'ALU définie par le code **AMF** si la condition spécifiée par le paramètre COND est valable.

La vérification

Dans le processus de vérification de cette instruction, on a considéré deux étapes consécutives qui sont :

1. Vérification de la condition.
2. La fonction réalisée par l'ALU.

5.3.4. Instructions du MAC

Les instructions du MAC sont similaires à celles de l'ALU. La seule différence est les paramètres d'entrée de cette unité. De ce fait, à par la partie qui concerne les paramètres de contrôle des entrées du MAC, les étapes des preuves de toutes les instructions du MAC sont identiques à celles de l'ALU.

Dans notre présentation des preuves des instructions du MAC, nous allons nous limiter à présenter les étapes de progression des instructions et à quelques commentaires concernant les preuves. Cela ne veut en cas dire que notre traitement des instructions du MAC n'était pas nécessaire vue la similarité avec l'ALU. Car, lors de la réalisation des preuves au moyen de HOL, il y a une très grande différence entre les deux cas. En effet, les fonctions MAC agissent sur des données de tailles variées 5bits, 8bits, 16 bits, 32 bits et 40 bits alors que les fonctions de l'ALU ne considèrent que des données de taille 16 bits. De plus, le résultat des opérations de l'ALU est directement dirigé vers un registre sortie (*AR* ou *AF*) de 16 bits de longueur. Alors que le résultat de la MAC est chargé dans trois différents registres (*MR0*, *MR1* et *MR2*) qui compose les 40 bits du résultat.

Malgré que les instructions de l'ALU sont similaires à celles du MAC, la preuve de chaque classe de ces instructions est très différentes de l'autre. Cela est du au fait que les entrées de l'ALU sont différentes de celles du MAC. De plus, on peut qualifier les preuves des instructions du MAC de plus complexes par rapport à celles de l'ALU, car les registres et les types de données du MAC sont très variés. Cela induit un nombre assez important d'entrées/sorties et des traitements spécifiques pour certains cas, ce qui complexifie les preuves.

5.3.4.1. MAC avec accès en lecture de la mémoire

Les états de progression de l'instruction MAC avec accès en lecture de la mémoire sont décrit par la Figure 5.11. La fonction de chaque état de cette figure est décrite dans le Tableau 5.14.

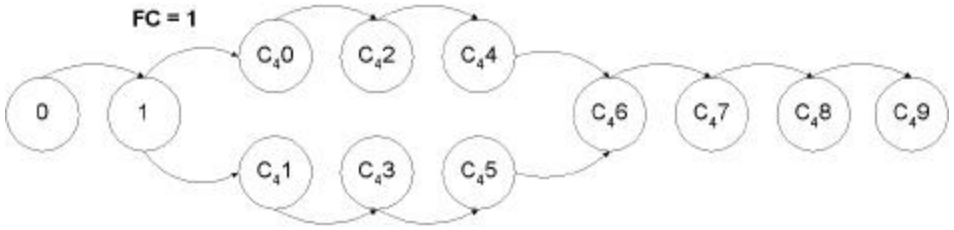


Figure 5.11. Représentation du processeur : instruction MAC avec accès en lecture de la mémoire.

Etat	*Description*
$C_4$0	Sélection des codes des registres destination pour la lecture des données de la mémoire programme (contrôle **PD**).
$C_4$1	Sélection des codes des registres destination pour la lecture des données de la mémoire données (contrôle **DD**).
$C_4$2	Sélection des registres **I** et **M** d'accès à la mémoire programme (contrôles **PMI** et **PMM**).
$C_4$3	Sélection des registres **I** et **M** d'accès à la mémoire données (contrôles **DMI** et **DMM**).
$C_4$4	Lecture des données de la mémoire programme.
$C_4$5	Lecture des données de la mémoire données.
$C_4$6	Génération des codes de contrôle de l'entrée **X** du MAC (paramètre **Xop**).
$C_4$7	Génération des codes de contrôle de l'entrée **Y** du MAC (paramètre **Yop**).
$C_4$8	Décodage de la fonction à exécuter par le MAC (paramètre **AMF**).
$C_4$9	Génération des codes de contrôle de le MAC.

Tableau 5.14. Etapes de l'instruction MAC avec accès en lecture de la mémoire.

Dans le processus de vérification de cette instruction, on a considéré quatre étapes consécutives qui sont :

1. Le décodage du registre destination des données de la mémoire programme.

2. Le décodage du registre destination des données de la mémoire données.
3. Le transfert des données de la mémoire aux registres destinations.
4. La fonction réalisée par le MAC.

5.3.4.2. MAC avec accès lecture/écriture mémoire données

Les états de progression de l'instruction MAC avec accès lecture/écriture mémoire données sont décrit par la Figure 5.12. La fonction de chaque état de cette figure est décrite dans le Tableau 5.15.

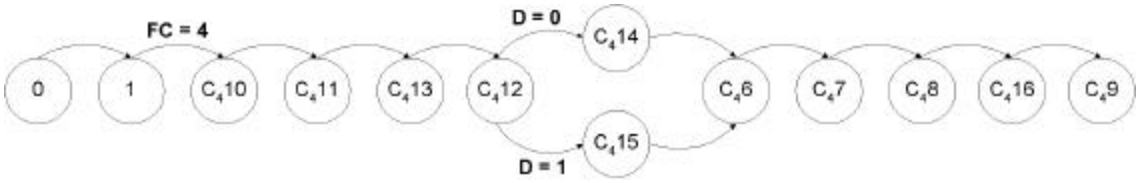


Figure 5.12. Représentation du processeur : instruction MAC avec accès lecture/écriture mémoire données.

Etat	*Description*
$C_4$10	Sélection des codes des registres destination pour la lecture des données de la mémoire (contrôle **DREG**).
$C_4$11	Sélection du générateur d'adresse nécessaire pour l'accès à la mémoire (contrôle **G**).
$C_4$12	Sélection du sens d'accès à la mémoire : lecture ou écriture (contrôle **D**).
$C_4$13	Sélection des registres **I** et **M** d'accès à la mémoire (contrôles **I** et **M**).
$C_4$14	Stockage des données dans le registre sélectionné, de la mémoire données, selon l'adresse spécifiée par les registres **I** et **M**.
$C_4$15	Ecriture des données, contenues dans le registre sélectionné, dans la mémoire données, selon l'adresse spécifiée par les registres **I** et **M**.
$C_4$6	Génération des codes de contrôle de l'entrée **X** de Le MAC (paramètre **Xop**).
$C_4$7	Génération des codes de contrôle de l'entrée **Y** de Le MAC (paramètre **Yop**).
$C_4$8	Décodage de la fonction à exécuter par Le MAC (paramètre **AMF**).
$C_4$16	Génération du code de contrôle du registre de stockage des données (paramètre **Z**).
$C_4$9	Génération des codes de contrôle de Le MAC.

Tableau 5.15. Etapes de l'instruction MAC avec accès lecture/écriture mémoire données.

Dans le processus de vérification de cette instruction, on a considéré quatre étapes consécutives qui sont :

1. Le décodage du registre à utiliser pour la lecture ou l'écriture des données.
2. Le décodage des registres qui serviront pour adresser la mémoire données.
3. Le transfert des données de la mémoire données au registre destination.
4. La fonction réalisée par le MAC.

5.3.4.3. MAC avec accès lecture/écriture mémoire programme

Les états de progression de MAC avec accès lecture/écriture mémoire programme sont décrit par la Figure 5.13. La fonction de chaque état de cette figure est décrite dans le Tableau 5.16.

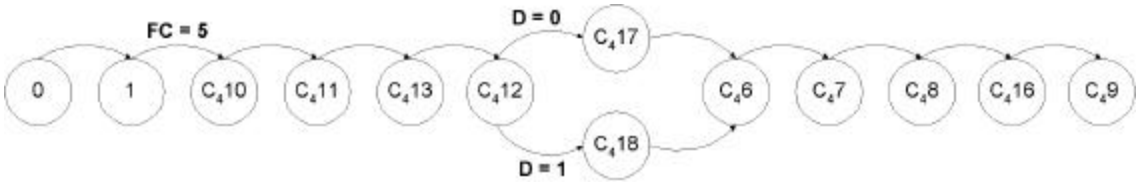


Figure 5.13. Représentation du processeur : instruction MAC avec accès lecture/écriture mémoire programme.

Dans le processus de vérification de cette instruction, on a considéré trois étapes consécutives qui sont :

1. Le décodage du registre à utiliser pour la lecture ou l'écriture des données.
2. Le décodage des registres qui serviront pour adresser la mémoire programme.
3. Le transfert des données de la mémoire programme au registre destination.
4. La fonction réalisée par le MAC.

La preuve de cette instruction est similaire à la preuve de l'instruction précédente. La seule différence concerne le type de mémoire adresser.

Etat	*Description*
$C_4$10	Sélection des codes des registres destination pour la lecture des données de la mémoire (contrôle **DREG**).
$C_4$12	Sélection du sens d'accès à la mémoire : lecture ou écriture (contrôle **D**).
$C_4$13	Sélection des registres **I** et **M** d'accès à la mémoire (contrôles **I** et **M**).
$C_4$17	Stockage des données dans le registre sélectionné, de la mémoire programme, selon l'adresse spécifiée par les registres **I** et **M**.
$C_4$18	Ecriture des données, contenues dans le registre sélectionné, dans la mémoire programme, selon l'adresse spécifiée par les registres **I** et **M**.
$C_4$6	Génération des codes de contrôle de l'entrée **X** de Le MAC (paramètre **Xop**).
$C_4$7	Génération des codes de contrôle de l'entrée **Y** de Le MAC (paramètre **Yop**).
$C_4$8	Décodage de la fonction à exécuter par Le MAC (paramètre **AMF**).
$C_4$16	Génération du code de contrôle du registre de stockage des données (paramètre **Z**).
$C_4$9	Génération des codes de contrôle de Le MAC.

Tableau 5.16. Etapes de l'instruction MAC avec accès lecture/écriture mémoire programme.

5.3.4.4. MAC avec transfert de données inter registres

Les états de progression de l'instruction MAC avec transfert de données inter registres sont décrit par la Figure 5.14. La fonction de chaque état de cette figure est décrite dans le Tableau 5.17.

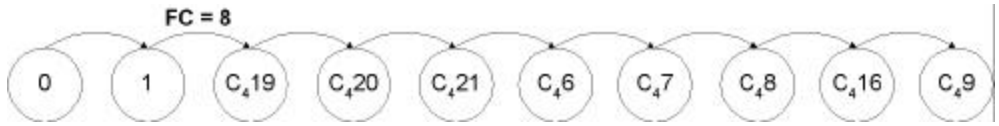


Figure 5.14. Représentation du processeur : instruction MAC avec transfert de données inter registres.

Etat	*Description*
$C_4$19	Spécification du registre destination définie par le paramètre **Dest REG**.
$C_4$20	Spécification du registre source définie par le paramètre **Source DREG**.
$C_4$21	Affectation du contenu du registre source dans le registre destination.
$C_4$6	Génération des codes de contrôle de l'entrée **X** de Le MAC (paramètre **Xop**).
$C_4$7	Génération des codes de contrôle de l'entrée **Y** de Le MAC (paramètre **Yop**).
$C_4$8	Décodage de la fonction à exécuter par Le MAC (paramètre **AMF**).
$C_4$16	Génération du code de contrôle du registre de stockage des données (paramètre **Z**).
$C_4$9	Génération des codes de contrôle de Le MAC.

Tableau 5.17. Etapes de l'instruction MAC avec transfert de données inter registres.

Dans le processus de vérification de cette instruction, on a considéré quatre étapes consécutives qui sont :

1. Le décodage du registre source.
2. Le décodage du registre destination.
3. Le transfert de données entre les deux registres source et destination.
4. La fonction réalisée par le MAC.

5.3.4.5. MAC conditionnelle

Les états de progression de l'instruction MAC conditionnelle sont décrit par la Figure 5.15. La fonction de chaque état de cette figure est décrite dans le Tableau 5.18.

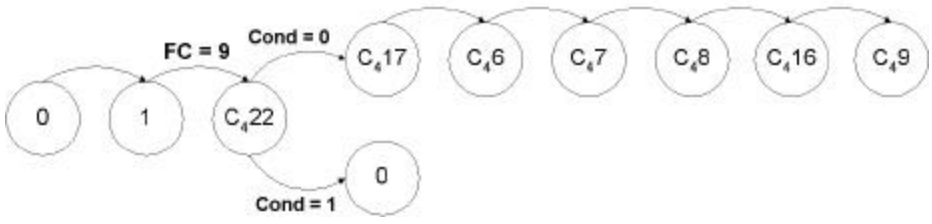


Figure 5.15. Représentation du processeur : instruction MAC conditionnelle.

Etat	*Description*
$C_4$22	Vérification de la condition d'exécution de l'instruction spécifiée par le paramètre **COND**.
$C_4$6	Génération des codes de contrôle de l'entrée **X** de Le MAC (paramètre **Xop**).
$C_4$7	Génération des codes de contrôle de l'entrée **Y** de Le MAC (paramètre **Yop**).
$C_4$8	Décodage de la fonction à exécuter par le MAC (paramètre **AMF**).
$C_4$16	Génération du code de contrôle du registre de stockage des données (paramètre **Z**).
$C_4$9	Génération des codes de contrôle du MAC.

Tableau 5.18. Etapes de l'instruction MAC conditionnelle.

Dans le processus de vérification de cette instruction, on a considéré deux étapes consécutives qui sont :

1. Vérification de la condition.
2. La fonction réalisée par le MAC.

5.3.5. Instruction du Shifter

5.3.5.1. Shift avec écriture/lecture de la mémoire données

Spécification Hardware

Séquenceur de programme (ce qui correspond aux états : 1, $C_5$0, $C_5$1, $C_5$2, $C_5$3 et $C_5$6 sur la Figure 5.16) avec :

paramètres d'entrée : le code de sélection du registre de lecture/écriture de la mémoire données (paramètre **REG**), les codes de sélection des registres **I** et **M** d'accès à la mémoire (paramètres **I** et **M**), le code du sens d'accès à la mémoire (contrôle **D**), les codes de contrôle de l'entrée **X** de l'unité Shifter et le code de la fonction à exécuter par l'unité Shifter (paramètre **SF**).
paramètres de sortie : paramètre de contrôle de l'accès à la mémoire, paramètre de contrôle de l'unité Shifter.

Accès à la mémoire (ce qui correspond à l'état $C_5$4 et $C_5$5 sur la Figure 5.16) avec :

paramètres d'entrée : paramètres de contrôle d'accès en lecture ou en écriture à la mémoire données.

L'unité Shifter (ce qui correspond aux états : $C_5$7, $C_5$8 sur la Figure 5.16) avec :

paramètres d'entrée : les contrôles de l'unité indiquants les valeurs de l'entrées **X** et le code de la fonction à exécuter (paramètre **SF**).

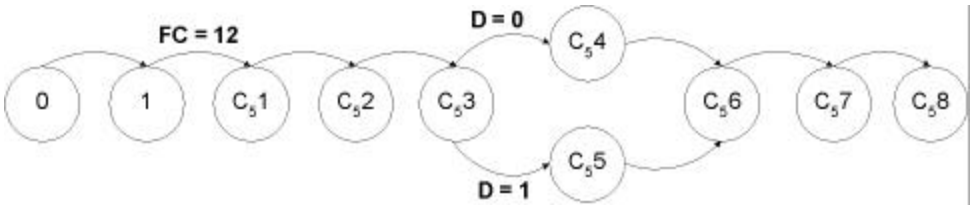


Figure 5.16. Représentation du processeur : instruction Shift avec écriture/lecture de la mémoire données.

Etat	*Description*
$C_5$0	Sélection du générateur d'adresse nécessaire pour l'accès à la mémoire (contrôle **G**).
$C_5$1	Spécification du registre pour le stockage ou la lecture des données (selon le paramètre **D**) défini par le paramètre **REG**.
$C_5$2	Sélection des registres **I** et **M** d'accès à la mémoire (contrôles **I** et **M**).
$C_5$3	Sélection du sens d'accès à la mémoire : lecture ou écriture (contrôle **D**).
$C_5$4	Stockage des données dans le registre sélectionné, de la mémoire données, selon l'adresse spécifiée par les registres **I** et **M**.
$C_5$5	Ecriture des données, contenues dans le registre sélectionné, dans la mémoire données, selon l'adresse spécifiée par les registres **I** et **M**.
$C_5$6	Génération des codes de contrôle de l'entrée **X** du Shifter (paramètre **Xop**).
$C_5$7	Décodage de la fonction à exécuter par le Shifter (paramètre **SF**).
$C_5$8	Génération des codes de contrôle de Le Shifter.

Tableau 5.19. Etapes de l'instruction Shift avec écriture/lecture de la mémoire données.

Spécification du fonctionnement

Cette instruction permet d'exécuter deux tâches qui sont :

- Selon la valeur du paramètre **D**, cette instruction permet :

  L'écriture du contenu du registre défini par le paramètre **REG** dans l'adresse de la mémoire données définie par les registres **I** et **M** du générateur d'adresse (1 ou 2 selon la valeur du paramètre **G**),

ou bien :

Le chargement du registre défini par le paramètre **REG** par le contenu de l'adresse de la mémoire données définie par les registres **I** et **M** du générateur d'adresse (1 ou 2 selon la valeur du paramètre **G**).

- exécuter une fonction de l'unité Shifter définie par le code **SF**.

La vérification

Dans le processus de vérification de cette instruction, on a considéré quatre étapes consécutives qui sont :

1. Le décodage du registre à utiliser pour la lecture ou l'écriture des données.
2. Le décodage des registres qui serviront pour adresser la mémoire données.
3. Le transfert des données de la mémoire données au registre destination.
4. La fonction réalisée par l'unité Shifter.

5.3.5.2. Shift avec écriture/lecture de la mémoire programme

Spécification Hardware

Séquenceur de programme (ce qui correspond aux états : 1, $C_5$1, $C_5$2, $C_5$3 et $C_5$6 sur la Figure 5.17) avec :

paramètres d'entrée : les codes des registres destination pour la lecture des données de la mémoire (contrôle **DREG**), les codes de sélection des registres **I** et **M** d'accès à la mémoire données (contrôles **I** et **M**), les codes de contrôle de l'entrée **X** de l'unité Shifter (paramètre **X**), le code de la fonction à exécuter par l'unité Shifter (paramètre **SF**).
paramètres de sortie : paramètres de contrôle du générateur d'adresse qui va être utilisé pour adresser la mémoire, paramètres de contrôle de l'accès à la mémoire, paramètres de contrôle de l'unité Shifter.

Générateur d'adresse avec :

paramètres d'entrée : contrôle des registres **I** et **M** qui seront utilisés pour accéder à la mémoire.
paramètres de sortie : contenus des registres **I** et **M** qui seront utilisés pour accéder à la mémoire.

Accès à la mémoire (ce qui correspond à l'état $C_5$9 et $C_5$10 sur la Figure 5.17) avec :

paramètres d'entrée : paramètres de contrôle d'accès à la mémoire données.

L'unité Shifter (ce qui correspond aux états : $C_5$7, $C_5$8 sur la Figure 5.17) avec :

paramètres d'entrée : les contrôles de l'unité indiquants les valeurs de l'entrée **X** et le code de la fonction à exécuter par l'unité Shifter (paramètre **SF**).

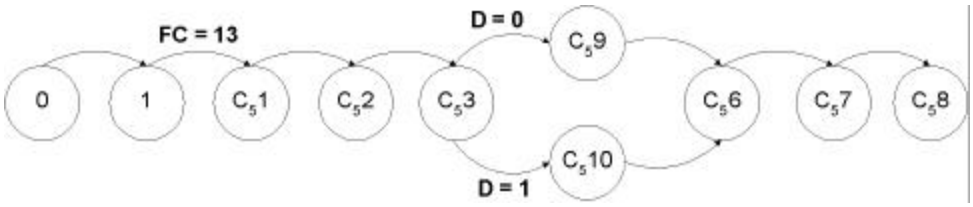


Figure 5.17. Représentation du processeur : instruction Shift avec écriture/lecture de la mémoire programme.

Etat	*Description*
$C_5$1	Spécification du registre pour le stockage ou la lecture des données (selon le paramètre **D**) défini par le paramètre **REG**.
$C_5$2	Sélection des registres **I** et **M** d'accès à la mémoire (contrôles **I** et **M**).
$C_5$3	Sélection du sens d'accès à la mémoire : lecture ou écriture (contrôle **D**).
$C_5$9	Stockage des données dans le registre sélectionné, de la mémoire programme, selon l'adresse spécifiée par les registres **I** et **M**.
$C_5$10	Ecriture des données, contenues dans le registre sélectionné, dans la mémoire programme, selon l'adresse spécifiée par les registres **I** et **M**.
$C_5$6	Génération des codes de contrôle de l'entrée **X** de le Shifter (paramètre **Xop**).
$C_5$7	Décodage de la fonction à exécuter par le Shifter (paramètre **SF**).
$C_5$8	Génération des codes de contrôle du Shifter.

Tableau 5.20. Etapes de l'instruction Shift avec écriture/lecture de la mémoire programme.

Spécification du fonctionnement

Cette instruction permet d'exécuter deux tâches qui sont :

- Selon la valeur du paramètre **D**, cette instruction permet :

  L'écriture du contenu du registre défini par le paramètre **DREG** dans l'adresse de la mémoire programme définie par les registres **I** et **M** du générateur d'adresse 1,

ou bien :

Le chargement du registre défini par le paramètre **DREG** par le contenu de l'adresse de la mémoire programme définie par les registres **I** et **M** du générateur d'adresse 1.

- exécuter une fonction de l'unité Shifter définie par le code **SF**.

La vérification

Dans le processus de vérification de cette instruction, on a considéré trois étapes consécutives qui sont :

1. Le décodage du registre à utiliser pour la lecture ou l'écriture des données.
2. Le décodage des registres qui serviront pour adresser la mémoire programme.
3. Le transfert des données de la mémoire programme au registre destination.
4. La fonction réalisée par l'unité Shifter.

5.3.5.3. Shift avec transfert de données interne inter registres

Spécification Hardware

Séquenceur de programme (ce qui correspond aux états : $C_5$11, $C_5$12, $C_5$13 et $C_5$6 sur la Figure 5.18) avec :

paramètres d'entrée : le code du registre destination (contrôle **Dest REG**), le code du registre source (contrôle **Source REG**), les codes de contrôle de l'entrée **X** de l'unité Shifter (paramètre **Xop**) et le code de la fonction à exécuter par l'unité Shifter (paramètre **SF**).
paramètres de sortie : paramètres de contrôle des unités concernées par les registres source et destination, paramètres de contrôle de l'unité Shifter.

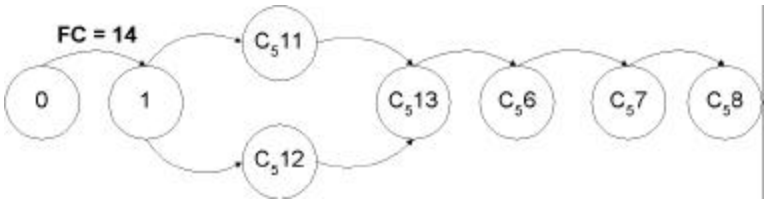


Figure 518. Représentation du processeur : instruction Shift avec transfert de données interne inter registres.

Les unités ALU, MAC, Shifter, DAG1, DAG2 ou ProgramSequencer avec :

paramètres d'entrée : les contrôles de l'unité indiquant le registre source, les contrôles de l'unité indiquant le registre destination.

L'unité Shifter (ce qui correspond aux états : $C_5$7, $C_5$8 sur la Figure 5.18) avec :

paramètres d'entrée : les contrôles de l'unité indiquants les valeurs des entrées **X** et **Y**, le code de la fonction à exécuter par l'ALU (paramètre **AMF**).

Spécification du fonctionnement

Cette instruction permet d'exécuter deux tâches qui sont :

- Affectation du contenu du registre source dans le registre destination.
- Exécuter une fonction de l'unité Shifter définie par le code **SF**.

Etat	*Description*
$C_4$11	Spécification du registre destination définie par le paramètre **Dest REG**.
$C_4$12	Spécification du registre source définie par le paramètre **Source DREG**.
$C_4$13	Affectation du contenu du registre source dans le registre destination.
$C_5$6	Génération des codes de contrôle de l'entrée **X** du Shifter (paramètre **Xop**).
$C_5$7	Décodage de la fonction à exécuter par le Shifter (paramètre **SF**).
$C_5$8	Génération des codes de contrôle du Shifter.

Tableau 5.21. Etapes de l'instruction Shift avec transfert de données interne inter registres.

5.3.5.4. Shift immédiat

Spécification Hardware

Séquenceur de programme (ce qui correspond aux états : 1 et $C_5$6 sur la Figure 5.19) avec :

paramètres d'entrée : les codes de contrôle de l'entrée **X** de l'unité Shifter (paramètre **Xop**), le code de la fonction à exécuter par l'unité Shifter (paramètre **SF**).
paramètres de sortie : paramètres de contrôle de l'unité Shifter.

L'unité Shifter (ce qui correspond aux états : $C_5$7, $C_5$8 et $C_5$14 sur la Figure 5.19) avec :

paramètres d'entrée : les contrôles de l'unité indiquants les valeurs de l'entrées **X**, le code de la fonction à exécuter par l'unité Shifter (paramètre **SF**).

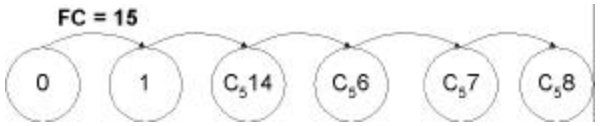


Figure 5.19. Représentation du processeur : instruction Shift immédiat.

Spécification du fonctionnement

Exécuter une fonction de l'unité Shifter définie par le code **SF** en utilisant directement le paramètre **exponent**.

La vérification

On a établi la preuve de cette instruction en une seule étape qui traite l'exécution de l'unité Shifter.

Etat	*Description*
$C_4$14	Affectation du paramètre **exponent** lu directement de l'instruction.
$C_5$6	Génération des codes de contrôle de l'entrée **X** du Shifter (paramètre **Xop**).
$C_5$7	Décodage de la fonction à exécuter par le Shifter (paramètre **SF**).
$C_5$8	Génération des codes de contrôle du Shifter.

Tableau 5.22. Etapes de l'instruction Shift immédiat.

5.3.5.5. Shift Conditionnel

Spécification Hardware

Séquenceur de programme (ce qui correspond aux états : 1, $C_5$15 et $C_5$6 sur la Figure 5.20) avec :

paramètres d'entrée : le code d'identification de la condition à vérifier (paramètre **COND**), les codes de contrôle de l'entrée **X** de l'unité Shifter (paramètre **Xop**), le code de la fonction à exécuter par l'unité Shifter (paramètre **SF**).
paramètres de sortie : paramètre d'identification du paramètre de la condition, paramètres de contrôle de l'unité Shifter.

L'unité Shifter (ce qui correspond aux états : $C_5$7, $C_5$8 sur la Figure 5.20) avec :

paramètres d'entrée : les contrôles de l'unité indiquants les valeurs de l'entrées **X**, le code de la fonction à exécuter par l'unité Shifter (paramètre **SF**).

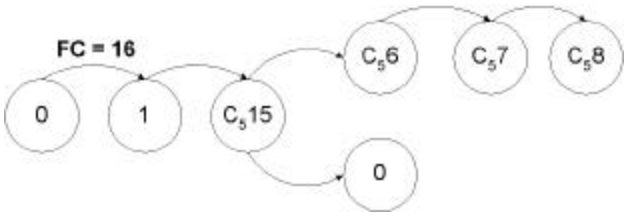


Figure 520. Représentation du processeur : instruction Shift Conditionnel.

Spécification du fonctionnement

Exécuter une fonction de l'unité Shifter définie par le code **SF** si la condition spécifiée par le paramètre COND est valable.

La vérification

Dans le processus de vérification de cette instruction, on a considéré deux étapes consécutives qui sont :

1. Vérification de la condition.
2. La fonction réalisée par l'unité Shifter.

Etat	***Description***
$C_5$15	Vérification de la condition d'exécution de l'instruction spécifiée par le paramètre **COND**.
$C_5$6	Génération des codes de contrôle de l'entrée **X** du Shifter (paramètre **Xop**).
$C_5$7	Décodage de la fonction à exécuter par le Shifter (paramètre **SF**).
$C_5$8	Génération des codes de contrôle de le Shifter.

Tableau 5.23. Etapes de l'instruction Shift Conditionnel.

5.4. Résultats et commentaires

5.4.1. Résultats des preuves

Dans la majorité des cas il est plus facile de retrouver un contre-exemple pour un théorème que de le démonter. De ce fait, les preuves qu'on a établies sont assez importantes dans le sens où elles indiquent que le hardware de la famille de processeurs ADSP-2100 est en parfait accord avec son implémentation (en considérant notre niveau d'abstraction).

Dans ce rapport, on n'a pas intégré toutes les preuves en HOL car cela demandera des centaines de pages pour chaque instruction. De ce fait, on s'est

limité à quelques exemples. Néanmoins, tous les détails qui concernent la considération du système et les étapes des preuves ont été décrites.

Lors du traitement des preuves, nous avons parfois rencontré quelques cas de divergence. Néanmoins, en vérifiant les spécifications du fonctionnement et surtout hardware, on s'est rendu compte qu'il y avait certaines erreurs dans notre représentation du système. Mais, par correction de ces spécifications, ces preuves aboutissent.

5.4.2. Vérification de tout le jeu d'instructions

Après avoir vérifier les différentes instructions des unités ALU, MAC et Shifter, la vérification du jeu d'instructions est devenue une tâche assez simple. En effet, notre approche avait pour objectif de départ la simplification de la représentation du processeur dans la phase finale de vérification. Il suffit, en effet, de considérer la représentation de tout le processeur (Figure 5.21) pour comprendre que la tâche de vérification revient uniquement à l'invocation des preuves précédemment réalisées.

La preuve dans ce cas est triviale. En effet, elle revient à considérer pour chaque valeur d'opcode la preuve précédemment réalisée. En effet, on a stocké les résultats de ces preuves sous forme de théorèmes que nous sommes capables d'invoquer dans n'importe quelle autre preuve.

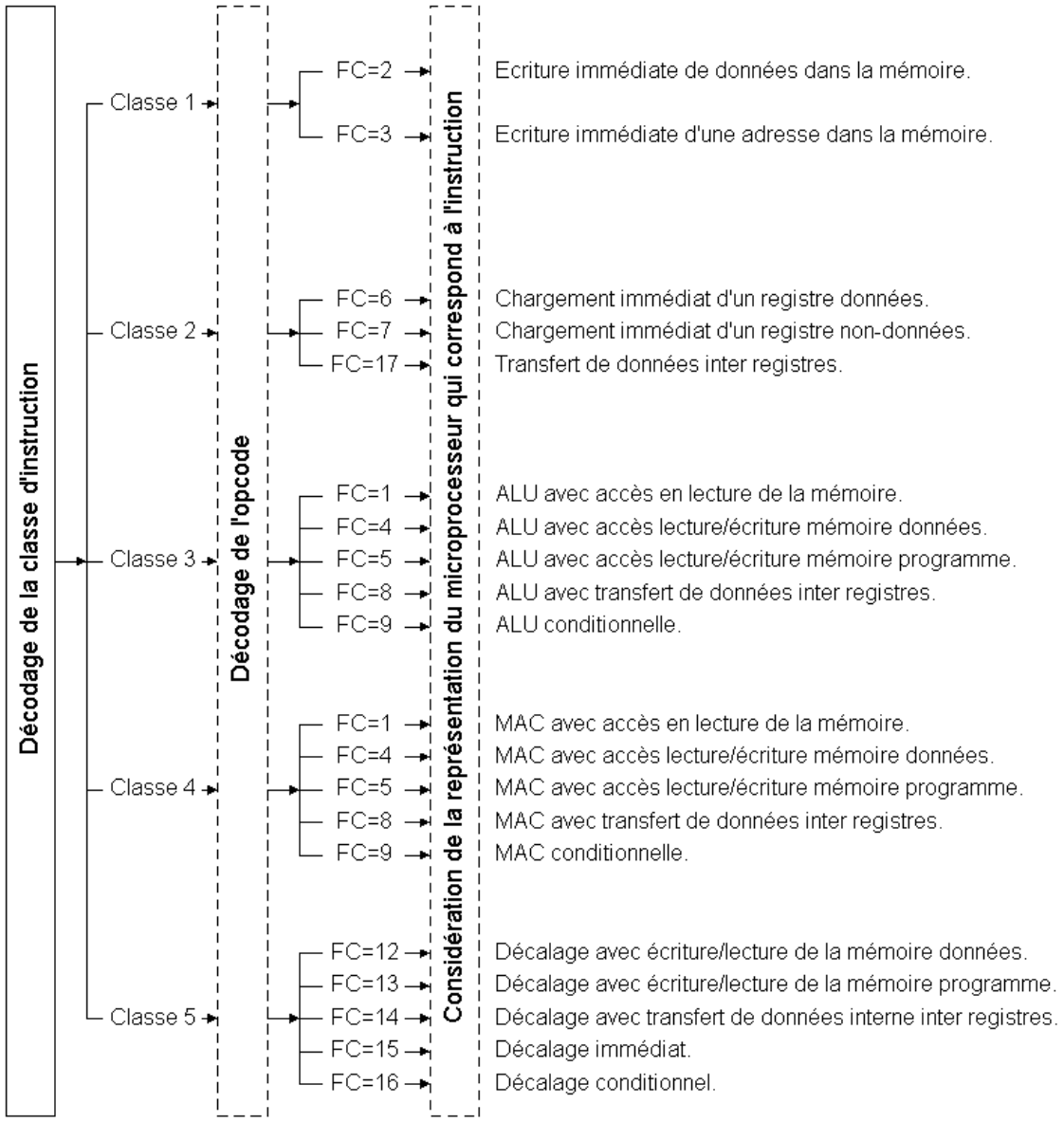


Figure 5.21. La représentation de tout le processeur.

5.4.3. Commentaires

5.4.3.1. La représentation des unités

Nous n'avons commencé directement par la preuve des instructions. Mais plutôt, on a commencé par la simplification des spécifications des preuves. Nous avons nommé cette étape *Vérification des Unités* car on avait vérifier que la structure hardware de chacune des unités du processeur implique sa spécification fonctionnelle. Cette dernière décrit le fonctionnement de l'unité en faisant

abstraction à sa structure hardware. Ainsi, lors du traitement des preuves, cela simplifiera énormément notre tâche.

On pourrait considérer aussi que la preuve par étapes des unités de gaspillage de temps car on doit définir à chaque fois une nouvelle spécification du système. Mais si on commence directement la preuve on risque de se retrouver face à un problème assez complexe qui demande un temps d'exécution assez long et une structure de preuve assez complexe. Donc, si on perd de point de vu écriture des spécifications des unités, on gagne au plan simplicité et temps d'exécution des preuves.

5.4.3.2. Les durées des preuves

Dans toutes les preuves qu'on a réalisées, on a décortiqué encore les étapes qu'on a décrit dans les paragraphes précédents. Cela nous a permis de manipuler des problèmes assez simples à résoudre ce qui a minimiser les durées des preuves qui n'avaient dépassé en aucun cas les dizaines de secondes.

Nous étions capables de réaliser ces décompositions des preuves grâce à deux facteurs qui sont : la progression par étape des instructions dans le temps et notre compréhension des détails du fonctionnement interne du processeur. Ces facteurs ont marqué notre approche qui s'est distinguée par la manipulation de preuves assez simples et ne demandant pas beaucoup de traitements. C'est, en effet, l'avantage fondamental qui nous a permis de traiter tout le jeu d'instructions du microprocesseur malgré sa complexité.

5.5. Conclusion

Dans ce chapitre on a présenté les preuves, d'une part, des spécifications des différentes unités fonctionnelles de la famille de processeurs 2100 et, d'autre part, les preuves d'une grande partie du jeu d'instructions de cette famille de processeurs. A notre niveau d'abstraction, on a fait preuve que quelles que soient les entrées et les sorties du processeur, son hardware fonctionne parfaitement en accord avec ce qui est prévu par son jeu d'instructions.

Chapitre 6 : PERSPECTIVES ET CONCLUSION GENERALE

Notre vérification de la famille de processeurs ADSP-2100 nous a permis de mettre en évidence la capacité de la méthode de vérification par le prouveur de théorèmes HOL à traiter des cas réels. Mais, en dépit de ce résultat, notre étude a ouvert le volets sur plusieurs autres perspectives qui sont présentées dans ce chapitre.

6.1. Perspectives

6.1.1. Considération des temps d'accès à la mémoire

Dans notre vérification du jeu d'instructions de la famille de processeurs ADSP-2100, nous avons considéré le temps d'accès à la mémoire. Mais, cela ne signifie pas que nos résultats sont incorrects. En effet, en traitant une seule instruction à la fois, il n'est pas nécessaire d'introduire le temps d'accès à la mémoire comme paramètre.

Dans des études plus poussées, nous opterons pour le cas des séquences d'instructions. Dans ce cas, le temps d'accès à la mémoire s'impose comme paramètre indispensable dans la vérification. En effet, la considération de œ paramètre, soulève plusieurs problèmes tel que le déroulement des séquences des instructions et les conflits de ressources qui pourraient apparaître.

6.1.2. Automatisation de l'outil HOL

En utilisant HOL, on avait dans plusieurs cas répéter des preuves de la même structure. De ce fait, nous estimons qu'il est possible d'automatiser certains preuves. On peut par exemple considérer des structures pré-définies de preuves et qu'on peut appliquer directement pour certaines configurations. Ainsi, la tâche de vérification de certains systèmes reviendrait à un choix de la meilleure

structure parmi un ensemble de départ sans s'engager dans la recherche au niveau tactique.

6.1.2.1. Génération automatique des spécifications

Dans notre manipulation de la vérification de la famille de processeurs ADSP-2100, nous avions à chaque fois besoin d'écrire des spécifications de certains systèmes. Cela présente une très grande perte de temps car on possède le schéma descriptif du système qu'on désire spécifier. On pourrait se demander dans ce cas sur la possibilité de générer automatiquement une spécification d'un système [25] .

Nous croyons qu'il est possible de générer une spécification d'un système à partir de son schéma. Néanmoins, pour garantir cet objectif, il est nécessaire d'écrire une variété de bibliothèques capable de : générer la spécification d'un ensemble de composants et d'optimiser cette spécification.

6.1.2.2. Réutilisation des preuves

Dans ce projet, on a considéré le cas de la famille de processeurs ADSP-2100. Néanmoins, il est possible de réutiliser les preuves qu'on a établies pour d'autres processeurs DSP. Cela n'est pas directement, mais, moyennant quelques modifications il devient possible d'exploiter les résultats obtenus pour d'autres cas.

L'une des perspectives de ce travail est de permettre l'exploitation des résultats obtenus dans d'autres preuves. Nous estimons qu'il est possible de traiter le cas d'autres processeurs DSP dans des durées assez courtes en exploitant les résultats de ce travail. Il est encore possible de présenter des règles générales qui peuvent être exploiter pour traiter les preuves similaires à celles qu'on a élaboré.

D'autre part, on estime que l'élaboration d'une bibliothèque de règles de spécification et de preuves permettra une exploitation plus optimisée de HOL. En effet, partant du fait que les processeurs DSP, par exemple, possèdent des architectures similaires, il serait plus rentable d'utiliser la méthode de spécification d'un cas de départ que de commencer le traitement à chaque fois à zéro. D'autre part, en examinant un exemple réel, il est certain qu'on aura plus de chance d'aboutir au résultat dans des durées minimes.

6.1.2.3. Interface graphique et analyseur lexico-syntaxique

La manipulation de l'outil HOL présente un inconvénient assez particulier qui concerne son interface graphique. En effet, il est nécessaire de rédiger toute la définition ou toute la tactique puis attendre la réponse du système. Alors qu'il est plus important de traiter les définitions au fur et à mesure que les définitions sont écrites.

D'autre part, la réponse du système aux erreurs est très grossière. En effet, le système répond qu'il y a une faute syntaxique sans spécifier la ligne qui a causé le problème ou le conflit qui s'est présenté. Ainsi, en traitant des spécifications de grande taille, tel est le cas des processeurs DSP, une très grande partie du temps de traitement est gaspillée à la recherche des fautes de frappes ou d'une parenthèse manquante.

Pour optimiser l'outil HOL, il est nécessaire de lui rajouter une interface graphique plus optimisée et d'utiliser un analyseur lexico-syntaxique robuste et fonctionnel en temps réel. De cette manière, il devient possible de limiter le temps de la recherche des erreurs syntaxiques et d'optimiser le temps d'analyse des spécifications.

6.1.3. Méthodes de vérification hybrides

L'objectif principal des méthodes formelles est d'aider les ingénieurs à concevoir des systèmes de plus en plus complexe. Ces méthodes interfèrent donc avec plusieurs domaines qui touchent aux télécommunications et à l'informatique. Leurs fondements sont mathématiques et leurs applications sont à la fois hardware et software.

Plusieurs avancements sur plusieurs volets ont marqué ces dernières années. Comme la technologie progresse, il devient de plus en plus possible d'aborder des systèmes de plus en plus complexes. Mais, la progression dans n'importe quel domaine demande le développement de nouveaux outils et surtout l'intégration des différentes méthodes existantes [1] .

6.1.3.1. Les directions futures

Il n'existe aucune méthode qui peut servir pour tous les cas. Pourtant il est nécessaire de traiter toutes les situations possibles. De ce fait, ces méthodes doivent satisfaire plusieurs arguments dont figurent les suivants:

- *L'apprentissage rapide :* Il est nécessaire que ces méthodes doivent être simples à apprendre et à manipuler pour en bénéficier les plus vite que possible.

- *L'accroissement du gain selon l'effort dissiper* : il est nécessaire que la manipulation de ces méthodes permette l'accroissement des connaissances en terme de rédactions des spécifications ou de manipulation des preuves.
- *La réutilisation* : Il est nécessaire de réutiliser les résultats obtenus pour un cas pour en traiter plusieurs autres afin d'amortir l'effort dissipé au départ.
- *La possibilité de l'inter-intégration* : une méthode est d'autant plus robuste qu'il est possible de l'intégrer avec d'autre ou d'exploiter ces résultats avec les outils et les langages de programmations connus.
- *L'efficacité* : Une méthode aussi complète soit-elle ne peut être robuste sans être efficace. En effet, si on ne peut pas accepter de passer la majorité du temps de traitement à la recherche des erreurs syntaxiques.
- *La détection orientée des erreurs* : Pour simplifier la tâche de l'utilisateur, il est nécessaire de définir les erreurs avec précisions et d'identifier leurs causes.
- *L'évolution* : une méthode ne peut pas survivre si elle n'est pas évolutive. Cela pourrait être assuré en offrant un certain degré de liberté de manipulation à l'utilisateur [26] .

Pour assurer les points précédemment définis, au lieu de manipuler un seul outil, il faudrait former des "méta-outils" qui eux-mêmes produisent un outil particulier pour chaque cas de figure. Ces méta-outils, tel est le cas des générateurs de compilateurs, permettent une production automatiques de prouveurs de modèles ou de vérificateurs de preuves.

6.1.3.2. L'intégration des méthodes :

Il est évident qu'il n'existe aucune méthode actuellement qui est capable d'analyser tous les cas des systèmes réels. Une approche plus pratique est de d'utiliser une combinaison de méthodes. Mais, dans de telles combinaisons, il est nécessaire de prendre en considération les facteurs suivant :

- La détermination un style adéquat de l'utilisation de méthodes différentes.
- La détermination de la logique qui conduit à l'utilisation de méthodes différentes.

Il est possible de combiner les deux méthodes de vérification par prouveur de théorèmes et la simulation. Cela revient à commencer par traiter le système par la méthode formelle (proveur de théorème dans ce cas). Ensuite, à chaque fois qu'un désaccord entre la spécification et le hardware est détecté, un test est validé. Le résultat de ce test impliquera l'étape qui suivra que se soit la

modification des spécifications ou du hardware. Cela est illustré sur la Figure 6.1.

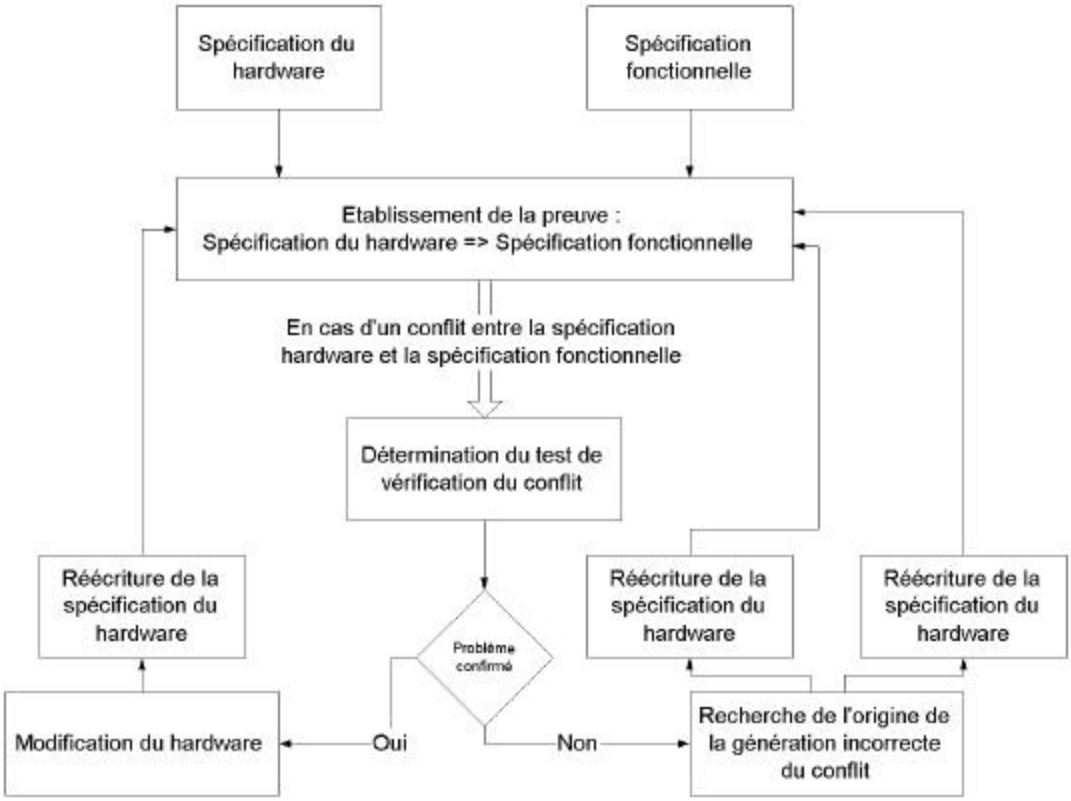


Figure 6.1. Méthode hybride "Prouveur de théorèmes & Simulation".

6.2. Conclusion Générale

Nous venons de présenter dans ce travail les étapes de vérification par une méthode formelle de vérification d'un processeur DSP. Cette méthode est dite par proveur de théorèmes HOL (Higher Order Logic) qui considère des représentations hardware et fonctionnelle du processeur et essaye de réaliser la preuve indiquant que le hardware répond au fonctionnement pour toutes les instructions du processeur.

Nous avons décomposé notre démarche d'étude en deux partie. Au premier lieu, nous avons commencé par *alléger* la représentation du hardware des différentes unités du processeur. En effet, au lieu de considérer directement la description

hardware de ces unités, on a utilisé l'outil HOL pour arriver à des représentations plus proches de la spécification fonctionnelle du processeur. Cette première étape est d'une grande importance car elle nous a permis de minimiser le gap entre la description hardware du processeur et sa spécification fonctionnelle.

Une fois la description hardware du processeur simplifiée, on s'est concentré sur le vif du sujet qui n'est autre que la vérification du jeux d'instructions du processeur. On a considéré pour cela les instructions une à une tout en exploitant les particularité de la classe de processeurs DSP (Digital Signal Processor) objet de l'étude.

Le grand apport de notre projet a été la mise en évidence de la capacité de la méthode de vérification par le proveur de théorème HOL a traiter le cas des processeurs DSP. Ce résultat est d'une importance, car il ouvre le volet à cette méthode pour vérifier des composants complexes et ayant des architectures variées.

Néanmoins, dans notre étude on s'est rendu compte qu'il y a plusieurs volets qui restent à améliorer ou à explorer. En effet, et à titre d'exemple, dans l'étude du processeur ADSP 2100, l'outil HOL offre uniquement l'assistance aux preuves qu'on a dû élaborer seuls. Imposant ainsi le refait de tout le travail pour étudier d'autres processeurs. De ce fait, on estime qu'il est nécessaire d'explorer le voie de l'automatisation de l'outil HOL, car sinon, cet outil risque d'être délaissé vue la complexité de son utilisation.

D'autre part, on s'est convenu qu'il est encore primordial de marier les méthodes formelles à la simulation pour réaliser une étude complète et réaliste du processeur. En effet, la représentation formelle, malgré qu'elle prend en considération tous les cas possibles, reste une représentation théorique du sujet qui risque d'être parfois en divergence avec le cas réel.

Notre projet est un premier pas sur deux voies la première est l'application des méthodes formelles pour les architectures DSP et la deuxième est la l'évaluation des performances de la méthode de vérification par le prouveur de théorème HOL. En effet, le résultat auquel on a abouti est qu'il est possible d'appliquer cette méthode de vérification à une très grande variété de processeurs et même au cas des micro-systèmes. Néanmoins, il faudrait pour pouvoir exploiter cette méthode dans le domaine industrielle, l'automatiser et la marier avec la simulation.

La vérification est une obligation plutôt qu'une nécessité car pour avoir des systèmes robustes, il est nécessaire de garantir leurs fonctionnements pour toutes les configurations possibles. Cet objectif peut être satisfait par la méthode formelle de vérification par le prouveur de théorème HOL qui se voit lancer pour l'une des principales méthodes futurs de vérification. L'utilisation, l'amélioration et l'adaptation aux configurations réelles de cette méthodes restent des voies très ouvertes qui nécessitent beaucoup d'efforts dans les prochaines années…

BIBLIOGRAPHIE

[1] E. M. Clarke and J. M. Wing. *Formal Methods: State of the Art and Future Directions*. *ACM Computing Surveys* , December 1996.

[2] T. Kropf. *Introduction to Formal Hardware Verification* , Springer Verlag, 1999.

[3] Melham, T.F., *Higher Order Logic and Hardware Verification*, Cambridge Tacts in Theoritical Computer Science 31, Cambridge University Press, 1993..

[4] C.H. Pygott. *Verification of VIPERS's ALU*. Technical report, Divisional Memo (Draft), the Royal Signals and Radar Establishment, 1991.

[5] V. Stavridou, T. F. Melham and R. T. Boute. : *Theorem Provers in Circuit Design*. Proceedings of the IFIP WG10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice, and Experience, Nijmegen, The Netherlands, 22-24 June 1992.

[6] K. Keutzer. *The Need for Formal Verification in Hardware Design and What Formal Verification has not Done for Me Lately*. Proc. HOL Theorem Proving System and its Application , Miami, Florida, USA, 1991.

[7] G. Birtwistle, B. Graham, and S.- K. Chin: *new_ theory 'HOL'; An Introduction to Hardware Verification in Higher Order Logic,* August 1994.

[8] Clarke, E.M. and Emerson, E. A. *Synthesis of synchronization skeletons for branching time temporal logic. In Logic of Programs*: Workshop, Yorktown Heights, NY, May 1981, Volume 131 of Lecture Notes in Computer Science (1981). Springer-Verlag.

[9] Vardi, M. Y. and Walper, P*. An automata-theoretic approach to automatic program verification*. In Proc. Of Logic in Computer Science (1986).

[10] Clarke, E.M. and Kurshan, R. *Computer-Aided Verification*. IEEE Spectrum 33, 6, 61-67.

[11] S. Tahar and R. Kumar: *A Practical Methodology for the Formal Verification of RISC Processors*; Formal Methods in Systems Design, Vol. 13, No. 2, September 1998, Kluwer Academic Publishers, pp. 159- 225.

[12] S. Tahar and R. Kumar: *Formal Specification and Verification Techniques for RISC- Pipeline Conflicts*; The Computer Journal, Vol. 38, No. 2, July 1995, Oxford University Press, pp.111- 120.

[13] Gordon, M. and T. Melham, *Introduction to HOL: A theorem Proving Environment for Higher-Order Logic*, Cambridge University Press: Cambridge, UK, 1993.

[14] P. Andrews. *An Introduction to Higher Order Logic: To Truth through Proof*. Academic Press, New York, 1986.

[15] The HOL System. Description. Technical Report, Cambridge Research Center, SRI International under contract to DSTO Australia, Cambridge, England, 1999.

[16] The HOL System. Reference. Technical Report, Cambridge Research Center, SRI International under contract to DSTO Australia, Cambridge, England, 1999.

[17] The HOL System. Tutorial. Technical Report, Cambridge Research Center, SRI International under contract to DSTO Australia, Cambridge, England, 1999.

[18] R. Milner and. M. Tofte. *The definition of Standard ML*. The MIT Press, Cambridge, Massachusetts and London, England, 1991.

[19] S. Romanenko and P. Sestoft. *Moscow ML Owners Manual, version 1.44*, August 1999.

[20] S. Romanenko and P. Sestoft. *Moscow ML Language Overview, version 1.44*, August 1999.

[21] The Applications Engineering Staff of Analog Devices, DSP Division. *DIGITAL SIGNAL PROCESSING APPLICATIONS USING THE ADSP-2100 FAMILY*. PRENTICE HALL, Englewood Cliffs, NJ 07632, 1996.

[22] Higgins, Richard J. *Digital Signal Processing in VLSI*. Englewood Cliffs, NJ: Prentice-Hall, 1990.

[23] The Applications Engineering Staff of Analog Devices, DSP Division. *Data sheet of the ADSP-2100 Family DSP - Microcomputers - ADSP-21xx*, 1996.

[24] Steven W. Smith , The Scientist and Engineer's Guide to Digital Signal Processing, Second Edition. California Technical Publishing, 1999.

[25] Curzon, P. *Correct Hardware Design and Verification Methods*. Eds. Paolo E Cammurati and Hans Evenking, LNCS 987, pp 56-70, Springer-Verlag, 1995.

[26] Curzon, P. *Tracking Design Changes with Formal Verification*. IEEE Proceedings 7th International Workshop on Higher Order Logic Theorem Proving and its Applications, Lecture Notes in Computer Science 859, Eds. T.F. Melham and J. Camilleri, Springer-Verlag, 1994.

ANNEXE : LE JEU D'INSTRUCTIONS DE LA FAMILLE DE PROCESSEURS ADSP-2100

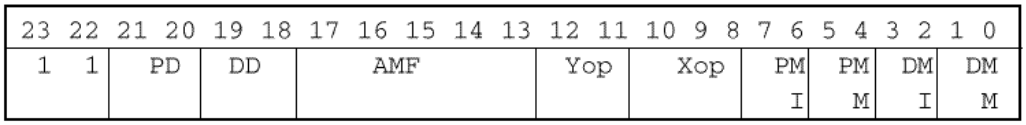
TABLE DES MATIERES

Dans ce qui va suivre on va présenter le jeux d'instructions de la famille de microprocesseur ADSP-2100. On va spécifier toutes les ***opcodes*** ainsi que la signification des champs qui en figurent.

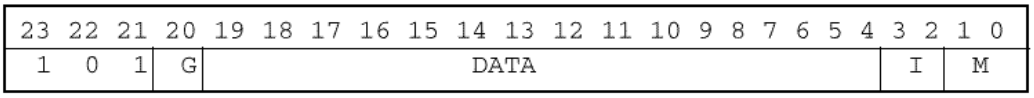
I).Liste des opcodes

Les différentes instructions sont codées sur 24 bits. En totalité il y a 31 type d'instructions pour la famille ADSP-2100 qui sont décrits par la liste suivante :

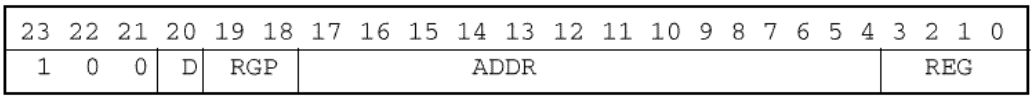
Type1 : ALU/MAC avec lecture de données et de la mémoire programme.



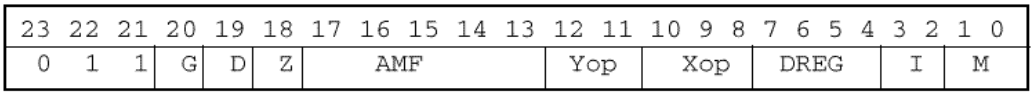
Type2 : Ecriture dans la mémoire donnée (données immédiate).



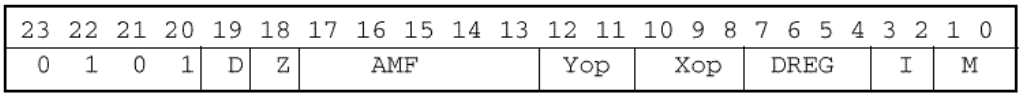
Type3 : Lecture/écriture de la mémoire données (adresse immédiate).



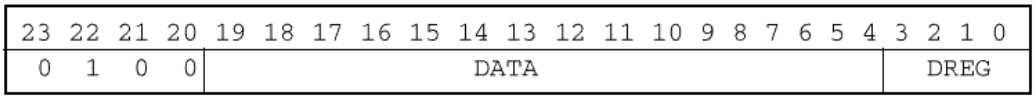
Type4 : ALU/MAC avec lecture/écriture da la mémoire données.



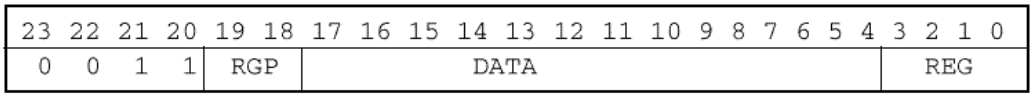
Type5 : ALU/MAC avec lecture/écriture da la mémoire programme.



Type6 : Charger les données du registre immédiatement.



Type7 : Charger les informations (non-données) du registre immédiatement.



Type8 : ALU/MAC avec transfert de registre interne.

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	Z	AMF					Yop		Xop			Dest DREG				Source DREG			

Generate ALU Status (NONE = <ALU>) *(ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)*

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	1	0	AMF					Yop		Xop			1	0	1	0	1	0	1	0

ALU codes only

Type9 : ALU/MAC conditionnel.

xop * yop

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	Z	AMF					Yop		Xop			0	0	0	0	COND			

xop * xop *(ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)*

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	Z	AMF					0	0	Xop			0	0	0	1	COND			

xop AND/OR/XOR constant *(ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)*

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	Z	AMF					YY		Xop			CC		BO		COND			

BO, CC, and YY specify the constant according the table shown at the end of this appendix.

PASS constant (constant ≠ 0,1,–1) *(ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)*

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	Z	AMF					YY		Xop			CC		BO		COND			

Type10 : Jump conditionnel.

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	S	ADDR														COND			

Type11 : Do Until.

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	1	ADDR														TERM			

Type12 : Shift avec lecture/écriture de la mémoire données.

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	1	G	D	SF					Xop			DREG			I	M		

Type13 : Shift avec lecture/écriture de la mémoire programme.

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	1	D	SF				Xop			DREG				I		M	

Type14 : Shift avec transfert interne de registres.

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	0	0	SF				Xop			Dest DREG				Source DREG			

Type15 : Shift immédiat.

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	SF				Xop			exponent							

Type16 : Shift conditionnel.

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	0	0	SF				Xop			0	0	0	0	COND			

Type17 : Transfert de données interne.

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	1	0	0	0	0	DST RGP		SRC RGP		Dest REG				Source REG			

Type18 : Mode de contrôle.

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	0	0	TI		MM		AS		OL		BR		SR		GM		0	0

Mode Control codes:

SR: Secondary register bank
BR: Bit-reverse mode
OL: ALU overflow latch mode
AS: AR register saturate mode
MM: Alternate Multiplier placement mode
GM: GO Mode; enable means execute internal code if possible
TI: Timer enable

11 = Enable Mode
10 = Disable Mode
0 1 = no change
0 0 = no change

Type19 : Jump conditionnel.

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0	I		0	S	COND			

Type20 : Return conditionnel.

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	T	COND			

Type21 : Modifies le registre d'adresse.

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	G	I		M	

Type22 : Réservé.

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	COND			

Type23 : DIVQ.

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	1	0	0	0	1	0	Xop			0	0	0	0	0	0	0	0

Type24 : DIVS.

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	1	0	0	0	0	Yop		Xop			0	0	0	0	0	0	0	0

Type25 : Saturation de MR.

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Type26 : Contrôle de stack.

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	PP	LP	CP	SPP	

Type27 : Call ou Jump lorsque le flag est activé.

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	1	Address												Addr		FIC	S

12 LSBs 2 MSBs

Type28 : Modifier le flag de sortie.

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	0	0	0	0	FO		FO		FO		FO		COND			

FL2 FL1 FL0 FLAG_OUT

Type29 : I/O espace mémoire lecture/écriture.

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1	D	ADDR											DREG			

Type30 : Aucune opération (nop).

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Type31 : En état de pause (Idle).

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Type31' : . En état de pause n (Idle(n)).

23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	DV			

II).Signification des champs des opcodes

AMF (ALU/MAC Functions code) : Codage des fonctions de l'ALU/MAC

0 0 0 0 0 No operation

MAC Function codes

0 0 0 0 1	X * Y	(RND)
0 0 0 1 0	MR + X * Y	(RND)
0 0 0 1 1	MR – X * Y	(RND)
0 0 1 0 0	X * Y	(SS) Clear when y = 0
0 0 1 0 1	X * Y	(SU)
0 0 1 1 0	X * Y	(US)
0 0 1 1 1	X * Y	(UU)
0 1 0 0 0	MR + X * Y	(SS)
0 1 0 0 1	MR + X * Y	(SU)
0 1 0 1 0	MR + X * Y	(US)
0 1 0 1 1	MR + X * Y	(UU)
0 1 1 0 0	MR – X * Y	(SS)
0 1 1 0 1	MR – X * Y	(SU)
0 1 1 1 0	MR – X * Y	(US)
0 1 1 1 1	MR – X * Y	(UU)

1 0 0 0 0	Y	Clear when y = 0
1 0 0 0 1	Y + 1	PASS 1 when y = 0
1 0 0 1 0	X + Y + C	
1 0 0 1 1	X + Y	X when y = 0
1 0 1 0 0	NOT Y	
1 0 1 0 1	– Y	
1 0 1 1 0	X – Y + C – 1	X + C – 1 when y = 0
1 0 1 1 1	X – Y	
1 1 0 0 0	Y – 1	PASS –1 when y = 0
1 1 0 0 1	Y – X	– X when y = 0
1 1 0 1 0	Y – X + C – 1	–X + C – 1 when y = 0
1 1 0 1 1	NOT X	
1 1 1 0 0	X AND Y	
1 1 1 0 1	X OR Y	
1 1 1 1 0	X XOR Y	
1 1 1 1 1	ABS X	

COND : Les états des codes de conditions

0 0 0 0	Equal	EQ
0 0 0 1	Not equal	NE
0 0 1 0	Greater than	GT
0 0 1 1	Less than or equal	LE
0 1 0 0	Less than	LT
0 1 0 1	Greater than or equal	GE
0 1 1 0	ALU Overflow	AV
0 1 1 1	NOT ALU Overflow	NOT AV
1 0 0 0	ALU Carry	AC
1 0 0 1	Not ALU Carry	NOT AC
1 0 1 0	X input sign negative	NEG
1 0 1 1	X input sign positive	POS
1 1 0 0	MAC Overflow	MV
1 1 0 1	Not MAC Overflow	NOT MV
1 1 1 0	Not counter expired	NOT CE
1 1 1 1	Always true	

CP (counter stack Pop Codes) :

0	No change
1	Pop

D (memory access Directions codes) :

0	Read
1	Write

DD (Double Data Fetch Data Memory Destination codes) :

0 0	AX0
0 1	AX1
1 0	MX0
1 1	MX1

DREG (Data Register codes) :

0 0 0 0	AX0
0 0 0 1	AX1
0 0 1 0	MX0
0 0 1 1	MX1
0 1 0 0	AY0
0 1 0 1	AY1
0 1 1 0	MY0
0 1 1 1	MY1
1 0 0 0	SI
1 0 0 1	SE
1 0 1 0	AR
1 0 1 1	MR0
1 1 0 0	MR1
1 1 0 1	MR2
1 1 1 0	SR0
1 1 1 1	SR1

DV (Divisor codes for Slow Idle instruction « IDLE (n) ») :

0 0 0 0	Normal Idle instruction (Divisor=0)
0 0 0 1	Divisor=16
0 0 1 0	Divisor=32
0 1 0 0	Divisor=64
1 0 0 0	Divisor=128

FIC (FI condition code):

1	latched FI is 1	" FLAG_IN "
0	latched FI is 0	" NOT FLAG_IN "

FO (Control codes for Flag Output Pins (FO, FL0, FL1, FL2)):

0 0	No change
0 1	Toggle
1 0	Reset
1 1	Set

G (Data Address Generator codes) :

0	DAG1
1	DAG2

I (Index Register codes) :

G =	0	1
0 0	I0	I4
0 1	I1	I5
1 0	I2	I6
1 1	I3	I7

LP (Loop Stack Pop codes) :

0	No Change
1	Pop

M (Modify Register codes) :

G =	0	1
0 0	M0	M4
0 1	M1	M5
1 0	M2	M6
1 1	M3	M7

PD (Dual Data Fetch Program Memory Destination codes) :

0 0	AY0
0 1	AY1
1 0	MY0
1 1	MY1

PP (PC Stack Pop codes) :

0	No Change
1	Pop

REG (Register codes) :

RGP =	00	01	10	11
0 0 0 0	AX0	I0	I4	ASTAT
0 0 0 1	AX1	I1	I5	MSTAT
0 0 1 0	MX0	I2	I6	SSTAT (read only)
0 0 1 1	MX1	I3	I7	IMASK
0 1 0 0	AY0	M0	M4	ICNTL
0 1 0 1	AY1	M1	M5	CNTR
0 1 1 0	MY0	M2	M6	SB
0 1 1 1	MY1	M3	M7	PX
1 0 0 0	SI	L0	L4	RX0
1 0 0 1	SE	L1	L5	TX0
1 0 1 0	AR	L2	L6	RX1
1 0 1 1	MR0	L3	L7	TX1
1 1 0 0	MR1	–	–	IFC (write only)
1 1 0 1	MR2	–	–	OWRCNTR (write only)
1 1 1 0	SR0	–	–	–
1 1 1 1	SR1	–	–	–

S (Jump/Call codes) :

0	Jump
1	Call

SF (Shifter Function codes) :

0 0 0 0	LSHIFT	(HI)
0 0 0 1	LSHIFT	(HI, OR)
0 0 1 0	LSHIFT	(LO)
0 0 1 1	LSHIFT	(LO, OR)
0 1 0 0	ASHIFT	(HI)
0 1 0 1	ASHIFT	(HI, OR)
0 1 1 0	ASHIFT	(LO)
0 1 1 1	ASHIFT	(LO, OR)
1 0 0 0	NORM	(HI)
1 0 0 1	NORM	(HI, OR)
1 0 1 0	NORM	(LO)
1 0 1 1	NORM	(LO, OR)
1 1 0 0	EXP	(HI)
1 1 0 1	EXP	(HIX)
1 1 1 0	EXP	(LO)
1 1 1 1	Derive Block Exponent	

SPP (Status Stack Push/Pop codes) :

0 0	No change
0 1	No change
1 0	Push
1 1	Pop

T (Return Type codes) :

0	Return from Subroutine
1	Return from Interrupt

TERM (Termination codes for DO UNTIL) :

0 0 0 0	Not equal	NE
0 0 0 1	Equal	EQ
0 0 1 0	Less than or equal	LE
0 0 1 1	Greater than	GT
0 1 0 0	Greater than or equal	GE
0 1 0 1	Less than	LT
0 1 1 0	NOT ALU Overflow	NOT AV
0 1 1 1	ALU Overflow	AV
1 0 0 0	Not ALU Carry	NOT AC
1 0 0 1	ALU Carry	AC
1 0 1 0	X input sign positive	POS
1 0 1 1	X input sign negative	NEG
1 1 0 0	Not MAC Overflow	NOT MV
1 1 0 1	MAC Overflow	MV
1 1 1 0	Counter expired	CE
1 1 1 1	Always	FOREVER

X (X Operand codes) :

0 0 0	X0 (SI for shifter)
0 0 1	X1 (invalid for shifter)
0 1 0	AR
0 1 1	MR0
1 0 0	MR1
1 0 1	MR2
1 1 0	SR0
1 1 1	SR1

Y (Y Operand codes)

0 0	Y0
0 1	Y1
1 0	F (feedback register)
1 1	zero

YY : voir YY, CC, BO.

Z (ALU/MAC Result Register codes) :

0	Result register
1	Feedback register

YY, CC, BO ALU / MAC Constant codes (Type 9)

(ADSP-217x, ADSP-218x, ADSP-21msp58/59 only)

Constant (hex)	YY	CC	BO	Bit #
0001	00	00	01	bit 0
0002	00	01	01	bit 1
0004	00	10	01	bit 2
0008	00	11	01	bit 3
0010	01	00	01	bit 4
0020	01	01	01	bit 5
0040	01	10	01	bit 6
0080	01	11	01	bit 7
0100	10	00	01	bit 8
0200	10	01	01	bit 9
0400	10	10	01	bit 10
0800	10	11	01	bit 11
1000	11	00	01	bit 12
2000	11	01	01	bit 13
4000	11	10	01	bit 14
8000	11	11	01	bit 15
FFFE	00	00	11	! bit 0
FFFD	00	01	11	! bit 1
FFBF	01	10	11	! bit 6
FF7F	01	11	11	! bit 7
FEFF	10	00	11	! bit 8
FDFF	10	01	11	! bit 9
FBFF	10	10	11	! bit 10
F7FF	10	11	11	! bit 11
EFFF	11	00	11	! bit 12
DFFF	11	01	11	! bit 13
BFFF	11	10	11	! bit 14
7FFF	11	11	11	! bit 15