Rapport de stage - Licence 3 Informatique

Université Rennes 1

Équipe Cairn

Discipline : Informatique

Mise en oeuvre sur FPGA d'un processeur VLIW à l'aide d'outils de Synthèse de Haut-Niveau (Cairn Vex)

Par : **Yohann Uguen**

Sous la direction de Steven Derrien, professeur des Universités

Date de fin de stage : 25 juillet 2014

Remerciements

Merci à Samantha sans qui je ne serais pas là aujourd'hui, ainsi qu'à Finn Bo Jorgensen qui m'a soutenu durant toute cette année. Je remercie également Steven Derrien qui m'a permis d'effectuer ce stage et qui a pris le temps de répondre à mes nombreuses questions. Merci à Simon Rokicki qui m'a beaucoup aidé et qui m'a beaucoup fait rire durant ces deux mois et demi. Et un grand merci à toute l'équipe Cairn qui a su m'accueillir.

Abstract

Dans le monde des systèmes embarqués, il existe un grand nombre d'architectures différentes dont les spécifications peuvent varier d'une application à une autre.
La mise en production d'un processeur spécifique à une tâche donnée est un processus long et coûteux de par son temps de développement et de par sa mise en production à grande échelle.
Dans ce document, nous présenterons le développement d'un processeur spécialisé pour de l'embarqué à l'aide d'outils de synthèse de haut-niveau, réduisant considérablement son temps de production. Puis nous présenterons les outils créés afin de procéder à une exécution sur cette architecture.

Table des matières

Abstract **3**

1 Introduction **5**

2 Contexte **6**

2.1 Les principales familles de processeurs 6

2.2 Vex et HP Vex *Toolchain* 7

2.3 Un example de Vex (le ρ-Vex) 9

3 Cairn Vex **10**

3.1 Les premiers essais 10

3.2 La sémantique du Cairn Vex 11

3.3 Exemples basiques de bonne utilisation des outils de synthèse de haut-niveau . . 11

3.4 Résultats obtenus pour le Cairn Vex 13

4 Chaîne de compilation **14**

4.1 Utilisation du *parseur* Xtext 14

4.2 Chaîne de compilation et d'exécution 14

Conclusion **17**

Annexes **19**

A Utilisation de la chaîne de compilation et d'exécution **19**

Références **20**

1 Introduction

Les outils de conception de systèmes électroniques numériques s'inscrivent comme étant essentiels dans la course aux performances et aux avancées technologiques depuis les années 1980. Ces outils ont connus de grandes améliorations, leurs permettant d'être de plus en plus performants, offrant alors des temps de conception réduits.

Subissant de fortes contraintes en termes de consommation énergétique et de surface de silicium utilisée, le matériel embarqué doit souvent être spécialisé, de manière à satisfaire ces contraintes. Il peut être nécessaire d'avoir recours à des accélérateurs matériel dédiés à un traitement particulier. La mise en place d'un tel matériel est cependant très longue.

Depuis les années 1990, les concepteurs de microarchitectures ont ressentis le besoin de rendre plus accessible, et plus rapide, la conception d'une puce électronique. Plusieurs langages de description en ont émergés, à l'image du **VHDL** (*VHSIC Hardware Description Language*) qui est un langage de description de matériel destiné à représenter le comportement ainsi que l'architecture d'un système électronique numérique. On parle alors de langage de synthèse.

On peut alors simuler le comportement d'une architecture matérielle de manière logicielle. Cependant, la simulation offerte par ce type de langage de synthèse rend le débogage fastidieux. En effet, le comportement d'un composant matériel est plus complexe à analyser que son analogue logiciel.

Depuis 2004, un nouveau type d'outil à fait son apparition, il s'agit des outils de synthèse de haut niveau. Ces derniers prenant alors comme description du matériel un code source de plus haut niveau, tels que du **C** ou du **SystemC**, rendant la programmation d'architectures encore plus accessible et plus rapide [5], à l'image de **Catapult C**, **Vivado**, **LegUp**, **GAUT**, etc. Ces logiciels transforment alors un fichier de description, dans un langage de haut-niveau, en un ensemble de fichiers **VHDL** et/ou **RTL** (*Register-Transfer Level*) [3].

L'architecture produite est cependant moins maîtrisée et peut ainsi être moins optimale en terme d'espace requis sur une puce et en terme de rapidité. Il est donc nécessaire d'adopter une façon d'écrire son code de manière à optimiser la surface de silicium utilisée.

Ces outils sont très performants pour produire des architectures possédant un flot de contrôle (conditions) simple, tels que des algorithmes de multiplications de matrices, des algorithmes de traitement du signal, etc.

Nous allons donc, à travers ce rapport, essayer de développer un processeur à jeu d'instruction grâce à des outils de synthèse de haut-niveau, un tel développement permettrait alors :

- De mettre en avant les limites des outils de synthèse de haut-niveau actuels.
- De définir des règles d'écritures permettant d'obtenir des bons résultats malgré un flot de contrôle important.
- D'étendre l'utilisation des outils de synthèse de haut-niveau à des architectures plus com-

plexes.

Ce document est organisé comme suit : Dans une première partie, nous mettrons en avant le principe de fonctionnement des différents types de processeurs actuels, nous décrirons ensuite dans une deuxième partie le processus de création du processeur **Cairn Vex**, puis dans une troisième partie nous détaillerons la mise en oeuvre de l'outil de création de binaire exécutable sur ce dernier ainsi que la chaîne de compilation dans son ensemble.

2 Contexte

Dans un premier temps, nous allons décrire précisément le fonctionnement des principaux types de processeurs à jeu d'instruction, en détaillant le type de processeur visé à travers ce rapport. Puis nous décrirons le contexte technique sur lequel nous nous appuierons, et enfin, nous détaillerons une implémentation matérielle de notre processeur, qui nous servira de référence pour valider nos résultats.

2.1 Les principales familles de processeurs

Le monde de l'embarqué est caractérisé par de fortes contraintes sur le coût matériel, sur la consommation énergétique et sur les performances des systèmes utilisés. À cause de ces contraintes, il est souvent nécessaire d'utiliser des architectures très différentes de celles utilisées pour les machines à usage généraliste.

En effet, un processeur généraliste privilégie la simplicité de programmation, entraînant un surcoût énergétique. Prenons l'exemple du processeur super-scalaire à exécution dans le désordre, ce dernier bénéficie d'une réelle simplicité de programmation, car il est capable d'exécuter un jeu d'instruction à sémantique séquentielle tout en faisant apparaître du parallélisme d'instruction. Il permet de plus de réorganiser dynamiquement les instructions afin de ne pas provoquer d'erreurs de dépendances. Une illustration de son fonctionnement est visible sur la figure 1. On y voit que le rôle du compilateur est de produire des instructions séquentielles, le processeur réordonnance alors les instructions afin d'y faire apparaître du parallélisme et d'exécuter plusieurs instructions simultanément.

Les processeurs utilisés dans l'embarqué ne peuvent pas se permettre de contenir du matériel aussi énergivore, c'est pourquoi il est nécessaire d'effectuer cette partie d'ordonnancement des instructions de manière logicielle. Prenons l'exemple de la famille des **VLIW** (*Very Long Instruction Word*), sur laquelle nous baserons notre rapport, qui exécute en parallèle un certain nombre d'instructions, sans se préoccuper des dépendances. Un schéma illustratif de son fonctionnement est décrit dans la figure 2. On y voit alors que le compilateur effectue le travail d'ordonnancement des instructions puis fournit plusieurs instructions à la fois au processeur qui n'a plus qu'à les exécuter.

Un exemple emblématique de VLIW est la famille de processeurs **ST200** développé par

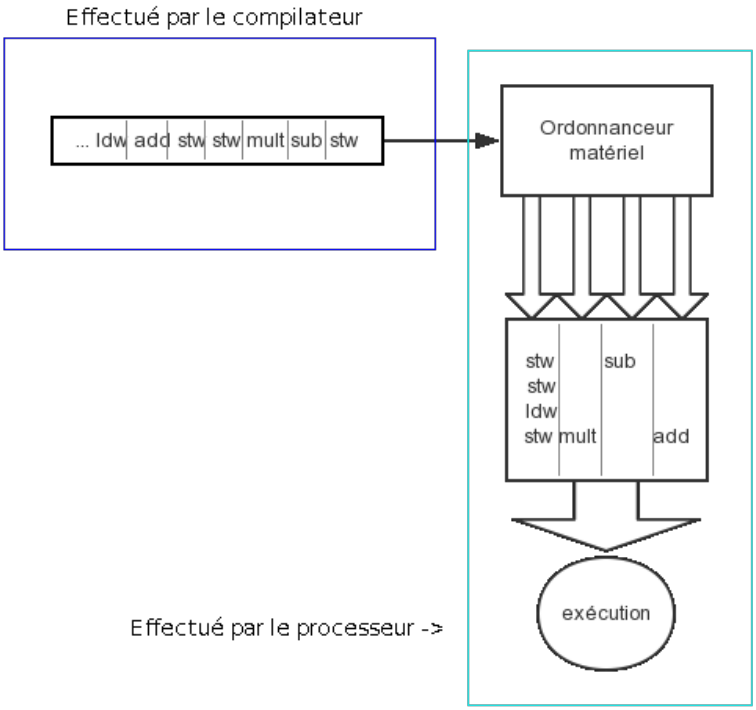

FIGURE 1 : Illustration du fonctionnement d'un processeur superscalaire à exécution dans le désordre

Hewlett-Packard Laboratories et STMicroelectronics, qui s'est écoulé à plus de 70 millions d'exemplaires dans le monde. On le retrouve essentiellement dans des appareils électroménagers.

Il n'existe que très peu d'informations sur l'implémentation des processeurs de la famille **ST200**, cependant les auteurs du livre *Embedded Computing* [1] sont des chercheurs ayant participé au développement de cette dernière et ont proposé, dans leur livre, un exemple type et malléable de processeur VLIW : Le Vex, que nous étudierons dans la prochaine partie.

2.2 Vex et HP Vex *Toolchain*

Le Vex (*VLIW Example*) est un processeur VLIW *pipeliné* (technologie visant à permettre une plus grande vitesse d'exécution des instructions en isolant chaque étape de cette dernière) décris de manière formelle à travers le livre *Embedded Computing* [1], sans implémentation matérielle de la part de ses concepteurs. Plusieurs spécifications de ce dernier sont laissées libres, telles que :

- **Le nombre de voies** : Le nombre d'instructions pouvant être exécutés en un cycle.
- **Le nombre de registres**
- **La longueur du *pipeline*** : La latence des différentes instructions.
- etc.

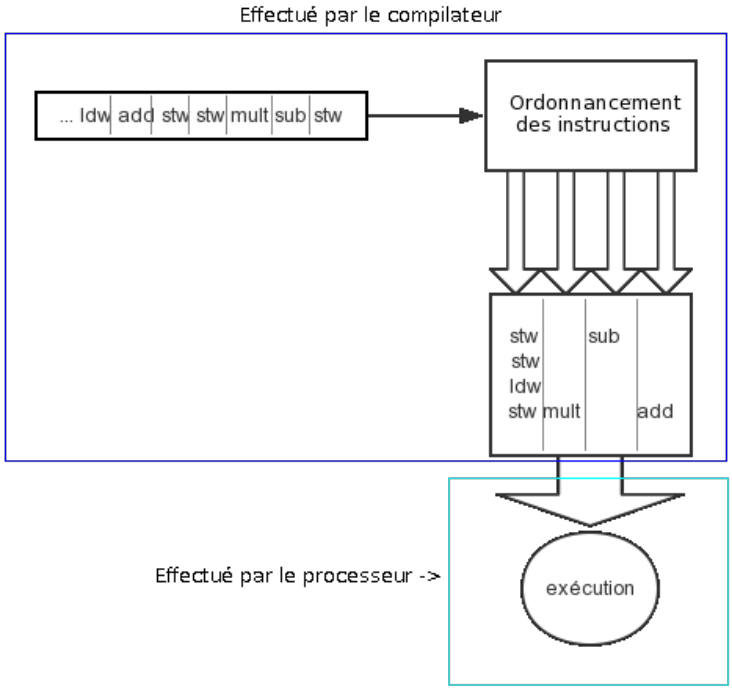

FIGURE 2 : Illustration du fonctionnement d'un processeur VLIW

Cependant, les auteurs ont développés un outil permettant de compiler et de simuler un code source **C** : Il s'agit du *HP Vex Toolchain* [2].

Dans le cadre de ce rapport, nous ne nous intéresserons qu'au compilateur, offrant des possibilités de configurations nécessaires et suffisantes au déroulement du projet. Ce dernier ne reconnaissant pas encore l'intégralité du langage **C**, il est nécessaire de se servir d'un sous-ensemble de ce langage.

Afin d'obtenir un code assembleur visant une architecture ayant la même sémantique que l'architecture vers laquelle nous nous dirigeons, il est nécessaire d'y joindre un fichier de configuration, spécifiant le nombre de voies, de registres, la latence (le nombre de cycles entre la lecture et l'écriture dans la file de registres) de chaque instruction, le nombre de voies effectuant des multiplications, etc.

Le fonctionnement du compilateur est décrit sur la Figure 3, on y voit que celui-ci prend en paramètre un code source **C** ainsi qu'un fichier de configuration, puis produit un fichier contenant du code assembleur.

Il existe un projet *open source* visant à créer une implémentation matérielle du Vex se basant sur le compilateur décris ci-dessus, il s'agit du ρ-Vex.

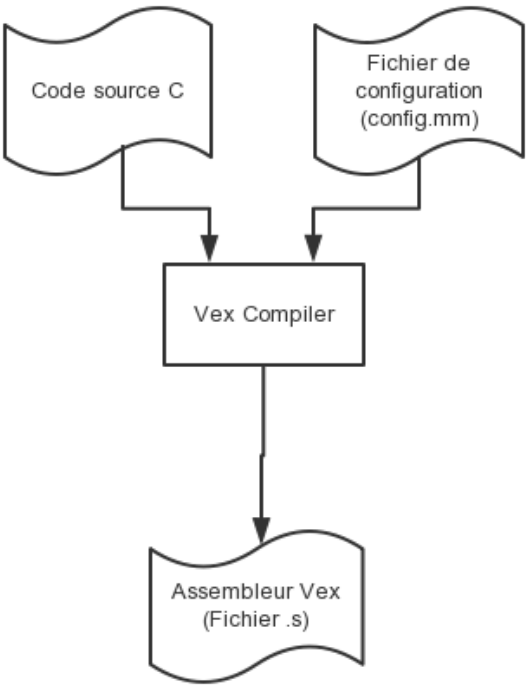

FIGURE 3 : Illustration du fonctionnement du compilateur Vex

2.3 Un example de Vex (le ρ-Vex)

Le ρ-Vex est un VLIW reconfigurable et extensible, originellement développé par *Thijs van As*, un étudiant de l'université de *Delft University of Technology* aux Pays-Bas [7]. Celui-ci a été écrit en **VHDL** et a nécessité plus de six mois de développement. La surface nécessaire à son implémentation sur un **FPGA** était de l'ordre de 16 000 LUTs (*look-up table*), l'unité standard de mesure d'espace utilisé sur une carte de ce type.

Le projet ayant été repris par d'autres étudiants, il s'est avéré que le ρ-Vex n'était pas complet, et ne fonctionnait pas. Après une seconde phase de développement, et l'officialisation du ρ-Vex 2 suivant les directives du Vex, l'espace nécessaire pour une implémentation sur FPGA était de 23 253 LUTs pour un Vex à quatre voies [8].

Dans cette section, nous avons présenté les différentes ressources dont nous disposons afin de tenter de mettre en oeuvre un Vex à quatre voies à l'aide d'outils de synthèse de haut-niveau. Nous allons maintenant présenter les différentes étapes de la création du **Cairn Vex**, un processeur **VLIW** basé sur le Vex, écrit en **C**.

3 Cairn Vex

Dans cette partie, nous allons dans un premier temps présenter nos tous premiers essais de génération d'architecture possédant un flot de contrôle conséquent grâce à des outils de synthèse de haut-niveau, puis nous détaillerons l'architecture du **Cairn Vex** et mettrons en avant certaines règles d'écritures permettant aux outils de synthèse de haut-niveau de mieux comprendre ces flots de contrôles complexes. Ensuite nous montrerons les résultats obtenus lors de ce stage.

3.1 Les premiers essais

Le développement du **Cairn Vex** s'étant fait de manière incrémentale, nous allons dans un premier temps présenter les résultats obtenus grâce aux outils de synthèse de haut-niveau pour un processeur possédant une voie sans *pipeline* (isolation de chaque étape de l'exécution d'une instruction). Il s'agissait d'un processeur RISC (*Reduced Instruction Set Computer*) possédant un jeu d'instructions simple (additions, soustractions, décalages, etc.), pour un total de dix-huit opérations différentes.

L'outil de synthèse de haut niveau que nous avons utilisé est **Catapult C Synthesis** de *Mentor Graphics* [4]. Dans la suite ce rapport, toute référence à des outils de synthèse de haut-niveau sera directement liée à cet outil.

Les résultats obtenus pour cette architecture étaient très décevants avec une surface utilisée d'environ 60 000 LUTs. Rappelons que la surface de référence était de 23 253 LUTs pour un processeur à quatre voies, qui était *pipeliné*. Nous avons alors fait plusieurs observations sur le comportement de **Catapult** et avons commencés à modifier l'écriture de notre code. En effet, l'écriture d'un code pour les outils de synthèse de haut-niveau est différente d'une écriture d'un code classique. En programmation, on va favoriser la rapidité du code et éviter de faire des opérations inutiles. En programmation architecturale, destinée à des outils de synthèse de haut-niveau, il faut comprendre que chaque variable déclarée sera transformée en fil, bus ou registre et que sa valeur sera calculée à chaque cycle malgré un flot de contrôle important.

Après réécriture du code, nous n'avons pas réussi à descendre en dessous de 40 000 LUTs.

Nous avons alors implémenté un *pipeline* à cinq étages sur notre processeur en espérant que **Catapult** reconnaisse la structure, nous en avons profité pour y ajouter des instructions plus coûteuses, comme des multiplications et des accès mémoire. Nous avons alors obtenus une surface d'environ 20 000 LUTs. Puis en adoptant les règles d'écritures décrites précédemment, nous avons réussis à obtenir une surface totale d'environ 18 000 LUTs.

Nous avons donc ajouté des voies, que nous avons spécialisées afin d'obtenir un Vex sémantiquement correct.

3.2 La sémantique du Cairn Vex

Le processeur Vex n'étant pas clairement définit, nous avons fait certains choix d'implémentation. Le **Cairn Vex** dispose donc de quatre voies, chaque voie exécute une instruction de 32 bits, soit à chaque cycle, le processeur reçoit une nouvelle instruction de 128 bits. Nous avons de plus implémenté un *pipeline* à cinq étages, chaque étage s'exécutant en un cycle :

- *Fetch* (F) : Séparation de l'instruction de 128 bits en quatre instructions de 32 bits.
- *Decode* (D) : Sélection de l'opération à effectuer, accès à la file de registres et préparation des opérandes pour l'étage suivant.
- *Execution* (Ex) : Exécution des opérations.
- *Memory* (M) : Accès mémoire si nécessaire.
- *Write Back* (WB) : Écriture des résultats sur la file de registres si nécessaire.

La plupart des instructions s'exécutent en cinq cycles, passant successivement dans tous les étages du *pipeline*. Cependant, malgré les possibilités de configuration de l'assembleur Vex, il est spécifié que l'instruction **MOV** s'exécute en un cycle de latence (nombre d'étages du *pipeline* entre le *Decode* et le *Write Back*). De plus, les instructions de branchement (**CALL**, **GOTO**, **BR**, **BRF**, etc.) s'exécutent en 2 cycles.

Le **Cairn Vex** possède une unique voie effectuant des accès mémoires (voie 2), une unique voie effectuant des multiplications (voie 4) et une unique voie effectuant des branchements (voie 1). De plus, chaque voie est capable d'effectuer toutes les opérations de base (**ADD**, **AND**, **SH2ADD**, etc).

Le jeu d'instruction implémenté est basé sur celui du ρ-Vex, auquel on a retiré/ajouté certaines instructions. Les instructions ayant été ajoutés ne sont pas visibles lors de la production de code assembleur, cependant elles sont nécessaires lors de la production de binaire exécutable par le **Cairn Vex** afin d'en assurer sa cohérence avec la sémantique du Vex.

Le **Cairn Vex** dispose de 64 registres de 32 bits, 8 registres de 1 bit et d'un registre de lien de 32 bits.

Afin de réaliser un tel processeur à l'aide d'outils de synthèse de haut-niveau, sans occuper une surface trop importante de la puce, nous avons adopté des règles d'écritures spécifiques à la conception d'architectures possédant un flot de contrôle important pour des outils de synthèse de haut-niveau. Nous allons à présent mettre en avant certaines de ces techniques.

3.3 Exemples basiques de bonne utilisation des outils de synthèse de haut-niveau

La synthèse de haut-niveau est un processus très complexe, demandant un temps de compilation très long, et qui n'est pas toujours optimale. Il est donc nécessaire d'adapter son code source **C**,

de manière à mettre en avant les optimisations que l'outil n'aurait pas trouvé de lui-même.

Afin de limiter le nombre de composants nécessaires au bon fonctionnement du circuit, il est judicieux de ne pas dupliquer les opérations. Un exemple simple est le suivant :

```
Exemple 1 :

if(bool){
        result = a*b;
}
else{
        result = c*d;
}
```

```
Exemple 2 :

int operand1, operand2;
if(bool){
        operand1 = a;
        operand2 = b;
}
else{
        operand1 = c;
        operand2 = d;
}
result = operand1 * operand2;
```

Dans l'exemple 1, le circuit produit par **Catapult** comportera deux multiplicateurs, qui sont très coûteux, organisés de la manière décrite sur la figure 4, qui montre que a et b vont être multipliés, de même que c et d, puis qu'un multiplexeur va sélectionner le résultat dont il a besoin. Alors que l'exemple 2 va permettre à **Catapult** de réutiliser l'opérateur de la multiplication dans les deux cas en produisant deux multiplexeurs, nettement moins coûteux. Le circuit produit est décrit sur la figure 5, qui montre qu'en utilisant deux multiplexeurs, on peut sélectionner les opérandes à multiplier dans un premier temps, puis effectuer la multiplication dans un second temps.

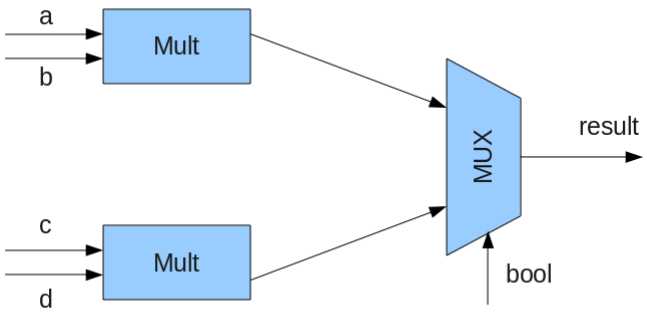

FIGURE 4 : Utilisation coûteuse des outils de synthèse de haut-niveau

Il est important de contrôler le nombre de cycles requis par chaque opération, car **Catapult** ne comprendra pas toujours que le circuit que vous décrivez est combinatoire. En utilisant le code source de l'exemple 3, **Catapult** va créer un circuit qui, dans un premier cycle, va transférer le contenu du registre 3 (*REG[3]*) dans le registre *result* puis, dans un second cycle, va transférer le contenu du registre 4 (*REG[4]*) dans le registre *result* si nécessaire. Alors que dans l'exemple

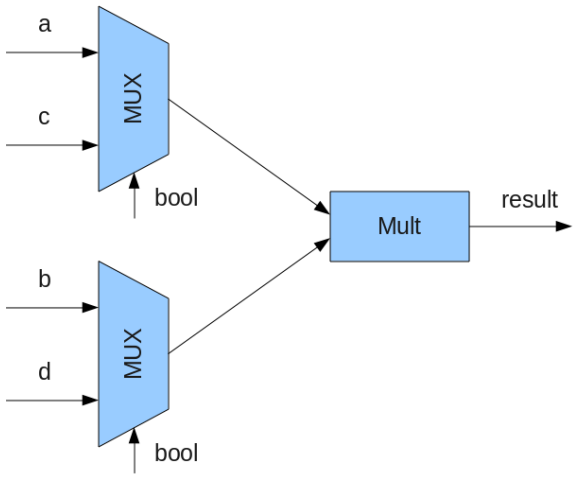

FIGURE 5 : Utilisation judicieuse des outils de synthèse de haut-niveau

4, le circuit réalisé va, en un seul cycle, transférer le contenu du bon registre dans le registre *result*.

```
Exemple 3 :

int result = REG[3];
if (bool){
    result = REG[4];
}
```

```
Exemple 4 :

if (bool){
    result = REG[4];
}
else{
    result = REG[3];
}
```

Nous avons donc démontré, à travers quelques exemples simples et illustratifs, l'importance d'écrire un code source plus adapté aux outils de synthèse de haut-niveau afin d'en obtenir de meilleurs résultats.

3.4 Résultats obtenus pour le Cairn Vex

En adoptant les règles d'écritures décrites dans la sous-section précédente, nous avons pu optimiser l'espace requis par le **Cairn Vex** sur un **FPGA**, ainsi que son nombre de cycle par opération. Après compilation sous **Catapult** puis sur le logiciel **Quartus II** développé par *Altera*, qui sert à connecter le circuit réalisé avec différents composants ainsi qu'à le charger sur la carte, nous avons obtenu un résultat de 28 563 LUTs. Chiffre tout à fait satisfaisant car, rappelons-le, le ρ-Vex utilisait 23 253 LUTs. Nous utilisons donc 122% de la surface de référence, pour un temps de développement inférieur à 30%.

Nous avons donc prouvé l'efficacité des outils de synthèse de haut niveau actuels pour des

circuits possédant un flot de contrôle important, à travers un temps de développement réduit et des performances similaires à un développement classique en **VHDL**.

Cependant, à ce stade du projet, le **Cairn Vex** exécute des instructions de 128 bits écrites directement en binaire dans sa mémoire d'instruction. Il est donc judicieux de réaliser un assembleur qui nous permettrait d'exécuter et d'effectuer des tests beaucoup plus facilement qu'en encodant les instructions à la main.

4 Chaîne de compilation

Dans cette partie, nous allons présenter le processus de production de code binaire exécutable par le **Cairn Vex**. Dans un premier temps, nous allons présenter les outils que nous avons utilisé ainsi que la mise en oeuvre de notre assembleur, puis dans un second temps, nous présenterons notre travail final ainsi sa chaîne de compilation et d'exécution.

4.1 Utilisation du *parseur* Xtext

Afin de transformer le code assembleur produit par le *HP Vex toolchain* et de le transformer en un fichier d'initialisation mémoire, nous avons eu besoin d'un outil nous permettant de *parser* (reconnaître la grammaire) du code assembleur contenu dans le fichier '*.s*'. Nous avons donc utilisé le *parseur* **Xtext**, nous permettant d'écrire une grammaire reconnaissant le langage. De plus, afin de permettre une meilleure réutilisation de cette grammaire, nous avons suivi le modèle de **GeCoS** (Generic Compiler Suite), qui est une infrastructure de compilation développée par l'équipe **Cairn** du laboratoire **IRISA**. Le *parseur* lit alors le texte passé en paramètre et le transforme en une représentation intermédiaire définie par la grammaire. Nous pouvons ensuite manipuler cette représentation intermédiaire à travers des objets **Java**.

Malgré une automatisation de chaque étape menant d'un code source **C** à une simulation, l'exécution de chaque étape, l'une après l'autre reste encore fastidieuse. Nous avons alors ressentis le besoin de d'automatiser l'ensemble de la chaîne de compilation.

4.2 Chaîne de compilation et d'exécution

Afin de permettre une utilisation plus facile et plus intuitive de l'outil mis en place à travers ce rapport, nous avons écrit un *script* **Shell** permettant d'effectuer l'ensemble de la chaîne de compilation. Il s'agit du fichier *cairn_vex*.

Il permet d'effectuer une simulation à partir d'un fichier **C**, de produire les fichiers d'initialisation de mémoire pour l'outil **Quartus II**, de s'arrêter après la phase de compilation et de produire un fichier contenant le code assembleur du fichier **C** et enfin, d'effectuer une simulation à partir d'un fichier assembleur à la place d'un fichier **C**.

L'ensemble de la chaîne de compilation et d'exécution, pour une simulation, est décrite dans

la figure 6. En effet, dans un premier temps, le fichier **C** (passé en paramètre lors de l'appel) va être passé en argument du compilateur *HP Vex Toolchain* ainsi que le fichier de configuration spécifiant l'architecture ciblée. Le fichier contenant le code assembleur produit va être passé en argument de l'assembleur **Cairn Vex**. Le fichier d'initialisation mémoire va alors être introduit dans le code source **C** du simulateur, que l'on va compiler grâce au compilateur **G++**. Enfin, le *script* lance l'exécution du fichier exécutable produit.

Dans cette deuxième partie, nous avons mis en avant les outils développés afin de compiler et d'exécuter du code **C** sur le **Cairn Vex**, que ce soit en simulation ou sur un **FPGA**.

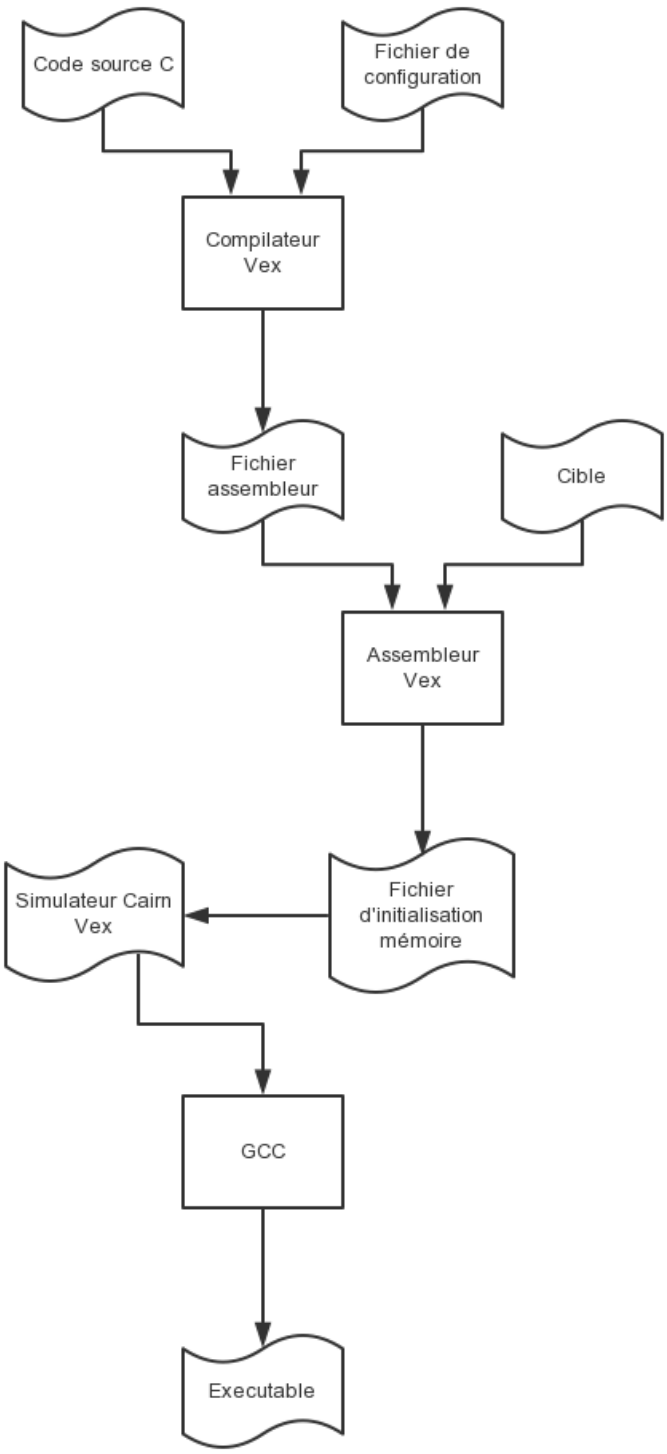

FIGURE 6 : Chaîne de compilation et d'exécution du **Cairn Vex**

Conclusion

Le temps de mise en production d'une architecture spécialisée est un point critique dans le domaine des systèmes embarqués. Comme nous l'avons montré dans la partie 1, les outils de synthèse de haut-niveau actuels sont de plus en plus performants, et permettent d'obtenir des résultats presque similaires à un développement classique, quand bien même l'architecture ciblée possède un flot de contrôle important, nécessitant une simple réécriture du code.

Dans ce rapport, nous avons proposé une architecture de type **VLIW**, réalisé dans un temps réduit sous contrainte de ressources, en utilisant des outils de synthèse de haut-niveau. Puis nous avons mis en place un outil permettant d'automatiser une simulation ou de créer des fichiers d'initialisation mémoire pour porter le processeur sur un **FPGA**.

Il reste cependant des pistes de développement possibles pour notre **Cairn Vex**. Certains points présentés dans ce rapport peuvent être améliorés :

- La création d'un outil permettant de transformer un code **C** donné en un code **C** compréhensible par le compilateur *HP Vex Toolchain*. Cela offrirait une plus grande souplesse dans la rédaction de code pour une exécution sur le **Cairn Vex**.
- Permettre la génération d'une architecture spécialisée. Grâce aux outils de synthèse de haut-niveau, il aurait été possible de spécifier le nombre de voies souhaités ainsi que les spécifications de ces dernières afin d'en générer une architecture. Seule la phase de génération de binaire exécutable par la machine aurait été fastidieuse.
- L'implémentation de plusieurs *Clusters* afin d'augmenter les nombre de voies présentes sur la carte sans augmenter de manière déraisonnable la taille de la file de registres.
- L'ajout d'un cache L2 et d'un accès à une mémoire RAM (*Random Acces Memory*) non *on-chip*.

La suite de ce travail consistera à développer un simulateur matériel de multi-coeur **VLIW**. En effet, étant donné que le **Cairn Vex** est fonctionnel, il s'agirait d'en démultiplier sa file de registres et ses mémoires, sans démultiplier les voies du processeur. Ce dernier exécuterait alors, à chaque cycle, une instruction d'un coeur différent. Cela permettrait d'obtenir des résultats de simulation de multi-coeur **VLIW** sous de fortes contraintes de ressources, tout en se servant des outils de synthèse de haut niveau. L'approche multi-coeur semble donc être une continuation logique à ce projet.

Annexes

A Utilisation de la chaîne de compilation et d'exécution

Le fichier *cairn_vex* offre plusieurs options, disponibles en effectuant la commande :

```
cairn_vex --help
```

Il permet :

- D'effectuer une simulation à partir d'un fichier **C** grâce à l'option *-s*
- De produire les fichiers d'initialisation de mémoire pour l'outil **Quartus II** grâce à l'option *-fpga*
- De s'arrêter après la phase de compilation et de produire un fichier nommé *cairn_vex_asm.s* contenant le code assembleur pour le **Cairn Vex** correspondant au fichier **C** passé en paramètre avec l'option *-a*
- De produire dans un fichier *nios_init.c*, du code permettant de d'initialiser la mémoire du **Cairn Vex** piloté par un **NIOS** (processeur configurable disponible avec l'outil **Quartus II**)
- D'effectuer une simulation à partir d'un fichier assembleur à la place d'un fichier **C** grâce à l'option *-b*

Il n'est possible d'utiliser qu'une seule option à la fois.

Prenons l'exemple d'une simulation à partir d'un fichier **C**, le *script* va alors être appelé avec la commande :

```
cairn_vex -s yourFile.c
```

Dans un premier temps, le fichier *yourFile.c* va être passé en argument du compilateur *HP Vex Toolchain* ainsi que le fichier de configuration spécifiant l'architecture ciblée. Le fichier *.s* produit va être passé en argument de l'assembleur **Cairn Vex** auquel on va spécifier l'option *-s*. Le fichier d'initialisation mémoire va alors être introduit dans le code source **C** du simulateur, que l'on va compiler grâce au compilateur **G++**. Enfin, le *script* lance l'exécution du fichier exécutable produit.

L'ensemble des fichiers sont disponibles en téléchargement [6].

Références

[1] Joseph A Fisher, Paolo Faraboschi, and Clifford Young. *Embedded computing : a VLIW approach to architecture, compilers and tools.* Elsevier, 2005.

[2] Joseph A Fisher, Paolo Faraboschi, and Clifford Young. Hp vex toolchain, 2005. http ://www.hpl.hp.com/downloads/vex/.

[3] Gary Smith Grant Martin. High-level synthesis : Past, present, and future. *IEEE Design and Test of Computers*, 26(4) :18–25, 2009.

[4] Mentor Graphics. Catapult c synthesis, 2011. http ://calypto.com/en/products/catapult/overview/.

[5] Electronic Engineering Times. Behavioral synthesis crossroad. *www.eetimes.com*, 2004.

[6] Yohann Uguen, Steven Derrien, and Simon Rockiki. Cairn vex. *https ://www.dropbox.com/l/EORrS7Qtt0QzmTtCP43ewn ?*

[7] Thijs van As. ρ-vex : A reconfigurable and extensible vliw processor. *code.google.com/p/r-vex/.*

[8] S. Wong, T. van As, and G. Brown. ρ-vex : A reconfigurable and extensible softcore vliw processor. In *Proc. International Conference on Field-Programmable Technology*, Taipei, Taiwan, December 2008.