Institut Supérieur
d'Informatique de
Modélisation et de
leurs Applications

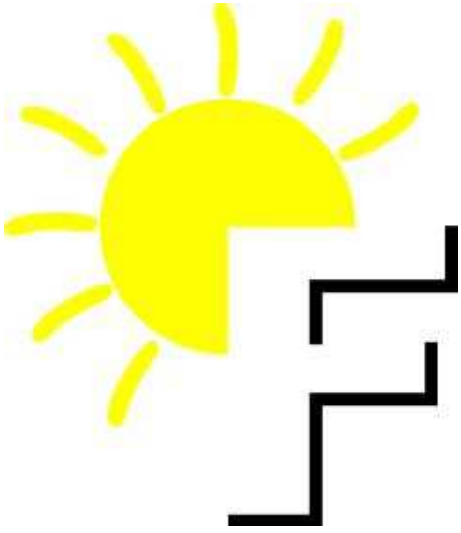
Complexe des Cézeaux
Campus de Clermont-Ferrand
BP 125 - 63173 AUBIERE CEDEX

The Freedom CPU Project
`http://f-cpu.seul.org/`

Rapport de projet de 2ème année

Conception de l'Additionneur de l'Unité de Calcul Flottant pour le projet Freedom-CPU

TOME I.



Présenté par SEMET Gaëtan
Responsable ISIMA : WODEY Pierre
Responsable du projet F-CPU : GUIDON Yann
Année 2003-2004

Institut Supérieur
d'Informatique de
Modélisation et de
leurs Applications

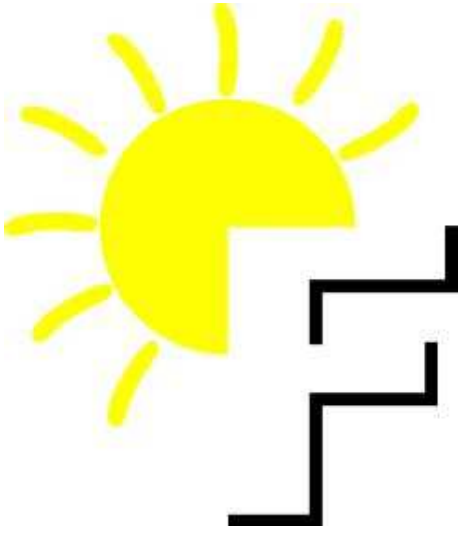
Complexe des Cézeaux
Campus de Clermont-Ferrand
BP 125 - 63173 AUBIERE CEDEX

The Freedom CPU Project
`http://f-cpu.seul.org/`

Rapport de projet de 2ème année

Conception de l'Additionneur de l'Unité de Calcul Flottant pour le projet Freedom-CPU

TOME I.



Présenté par SEMET Gaëtan
Responsable ISIMA : WODEY Pierre
Responsable du projet F-CPU : GUIDON Yann
Année 2003-2004

Ce présent document est placé sous les termes de la GFDL, ou « GNU Free Documentation License », dont le texte peut être trouvé sur sur le site du projet GNU (`http://www.gnu.org`). Une copie de cette licence est inclue dans ce package (fdl.txt).

Résumé

Le projet F-CPU a pour but de développer un microprocesseur RISC, SIMD, superpipeline 64 bits libre. Les sources sont écrits en langage VHDL et sont disponibles sous licence GPL. Ce rapport décrit la conception de l'additionneur de l'unité de calcul en virgule flottante adapté à l'architecture spécifique du F-CPU. Le projet se voulant le plus libre possible, il n'a été utilisé pour le développement que des outils libres ou dans leur version d'évaluation gratuite, tout en veillant à la portabilité du code VHDL d'un compilateur à un autre.
L'unité doit respecter la norme IEEE-754 définissant les opérations sur les nombres à virgule flottante. L'addition en virgule flottante étant une opération essentiellement séquentielle, différentes techniques ont été mises en oeuvre pour réduire le chemin critique, notamment le « double Datapath », Leading-One Prediction, ainsi que des méthodes de calcul rapide de l'arrondi. De plus, l'architecture spécifique du F-CPU impose des contraintes très importantes sur la profondeur de chaque étage du pipeline, qui ont dû être impérativement respectées, ainsi que sur la parallélisation des calculs en utilisant le mode SIMD.
L'étude se limite à la simulation et synthèse de l'unité conçue, qui n'a pu être testée de manière exhaustive.

Mots clés : F-CPU, Addition en virgule flottante, SIMD, superpipeline, LOP, double Datapath, mode d'arrondi IEEE

Abstract

The F-CPU project aims at designing a Free, SIMD, superpipelined 64 bit RISC microprocessor, as free as linux is. VHDL source codes are under GPL license. This report describes the design of the Floating Point Adder adapted for the F-CPU specific architecture. Sources are as portable as possible to be compiled, simulated and synthetised on many plateforms and tool suites. Only GPL tools or free evaluation versions of commercial tools have been used.
This unit must comply with IEEE-754 standard defining Floating Point Operations. The floating point addition is mainly a serial operation, that is why many methods were used to reduce the critical path: Double Datapath, Leading One Prediction, Fast Rounding. Moreover, F-CPU's specific architecture defines heavy constraints on pipeline depth, SIMD modes, ...
This study is limited to simulation and synthesis of the code, which were not heavily tested.

Keywords: F-CPU, FPU, floating point adder, SIMD, superpipeline, LOP, double datapath, IEEE rounding Modes

Remerciements

J'aimerais exprimer ma gratitude pour tous les acteurs du projet qui répondent à chaque message de la Mailing List. J'aimerais remercier particulièrement Yann Guidon pour ses conseils et la relecture du rapport, Nicolas Boulay et spécialement Michael Riepe pour leurs conseils techniques très pointues, ainsi que M. Pierre Wodey.

Table des matières

Table des figures

Liste des tableaux

Glossaire

Chemin Critique	Le chemin le plus long qu'un signal doit parcourir dans une unité.
CISC	Complex Instruction Set Computer, ordinateur à jeu d'instruction complexe. S'oppose au RISC. Exemple : architecture x86.
CPU	Acronyme de Central Processing Unit, Unité centrale de traitement d'un ordinateur (microprocesseur central). Exemple : Intel Pentium, IBM PowerPC
FPGA	Acronyme de Field-Programmable Gate Array, Réseau logique programmable. Ces puces sont de plus en plus utilisées pour implémenter de manière peu coûteuse un microprocesseur ou une unité de calcul. L'intérêt principale est que la logique interne est reprogrammable autant de fois que l'on souhaite. Avec, il serait possible de reprogrammer son microprocesseur. Plusieurs compagnies se partagent ce marché, comme Actel, Altera, Atmel, Xilinx.
FPU	Acronyme de Floating Point Unit, Unité de Calcul en Virgule Flottante. Unité du CPU spécialisée dans le traitement des opérations en virgule flottante
F-CPU	Projet Freedom-CPU
GPL	Acronyme de General Public License. C'est une licence de type « Copyleft », que l'auteur d'un programme utilise pour autoriser toute autre personne à utiliser, examiner, recopier et modifier ce programme et ses sources, à condition de préserver le copyright et la licence. C'est une alternative au « domaine public » qui garantit la liberté et la paternité du logiciel. La GPL est pour l'instant utilisée par le projet F-CPU pour protéger le code source VHDL en attendant l'arrivée d'une licence plus adaptée à la conception matérielle.
Latence	la latence caractérise le nombre de cycles qu'une opération doit utiliser pour rendre son résultat. Plus il est petit, plus l'opération est rapide. De plus, sur une architecture pipeline, la latence correspond au nombre d'étages que l'on doit traverser pour avoir le résultat, car chaque étage fait un cycle.

IP	Acronyme de Intellectual Property, Propriété Intellectuelle. Dans le droit anglo-saxon, elle regroupe le secret industriel, le droit des marques, le Copyright et le droit des brevets. Dans le domaine de la microélectronique, un « coeur IP » (ou « IP core ») est souvent une fonction électronique vendue sous forme logicielle, déjà compilée ou sous forme de fichiers sources. Exemple : Le synthétiseur Synopsys est livré avec toute une gamme de « IP » où l'on retrouve des unités de calcul entièrement conçues et optimisées que l'on peut directement inclure dans sa conception.
Mailing List	adresse email où l'on peut s'inscrire pour recevoir tous les messages que tous les membres lui envoient. C'est le lieu principal des échanges du projet F-CPU, cela fonctionne comme une salle où chacun parle librement, par email interposé
RISC	Acronyme de Reduced Instruction Set Computer, ordinateur à jeu d'instruction réduit. S'oppose généralement au CISC. D'autres définitions existent : Relegate the Important Stuff to the Compiler, ou Reduced Instruction Set Complexity. RISC est une méthodologie de conception des processeurs, qui favorise la simplicité de l'ensemble du système pour augmenter la vitesse de traitement. Exemple : MIPS, SPARC
SIMD	Acronyme de Single Instruction Multiple Data, Une seule instruction, plusieurs données. Méthode pour paralléliser des calculs en donnant au microprocesseur une seule instruction où les opérandes de taille n sont en fait un assemblage d'opérandes de taille n/m. Il y a donc m opérations à effectuer en parallèle pour un mode donné, l'intérêt étant que l'on peut avoir plusieurs modes possibles (donc plusieurs m_i).

Introduction

Le projet F-CPU est constitué d'une équipe internationale de passionnés en informatique et électronique. Ils ont pour but la conception d'un coeur de microprocesseur de type RISC, 64 bits hautement pipeliné. Les membres sont totalement bénévoles, et le produit de leurs travaux est distribué sous licence « copyleft », de la même manière que le célèbre système d'exploitation libre GNU/Linux [1]. Cela s'oppose à la forte tradition de l'industrie du semiconducteur : tout est protégé par des brevets dans le but d'empêcher les concurrents de développer des produits similaires. C'est pour cela que l'architecture développée doit s'appuyer sur des techniques suffisamment anciennes (mais toujours aussi efficace[2]), ou développer de nouveaux concepts.
Le projet en est toujours à sa phase initiale, bien qu'il existe depuis août 1998, et se déroule essentiellement (voir exclusivement) par échange de mail sur le réseau Internet entre les protagonistes. Le développement est donc très distribué, et chacun apporte sa vision du projet, ce qui ne manque pas d'alimenter de longues chaînes d'email. Car c'est aussi une caractéristique importante du projet : chacun peut librement apporter sa pierre à l'édifice. Mais les connaissances et compétences demandées pour participer sont assez poussées, c'est pourquoi le projet ne passionne pas autant que Linux lorsqu'il a été révélé en 1991.
Le coeur actuel est appelé « FC0 » (pour « F-CPU Core 0 ») et est développé entièrement avec le langage de description matériel VHDL'93, qui est un standard de l'industrie. La conception doit constamment respecter ce standard pour être le plus portable possible d'un compilateur à un autre, d'un simulateur ou un synthétiseur à un autre.
En Octobre 2003, Yann Guidon, actuellement « coordinateur » du projet, est venu présenter le F-CPU à l'ISIMA et a ainsi proposé à l'école de participer, dans le cadre d'un projet de deuxième année (ainsi qu'un projet de troisième année), à la réalisation de cette architecture. Il m'a ainsi été demandé de concevoir l'unité de calcul flottant respectant la norme IEEE-754. Mais devant l'ampleur de la tâche, le sujet a rapidement été limité à la conception de l'additionneur flottant spécifiquement adapté au F-CPU. En effet, comme nous le verrons par la suite, l'architecture définit de nombreuses contraintes difficiles à respecter, et les quelques zones d'ombres du manuel n'arrangent pas les choses...

[1] attention, cela ne veut pas dire que ces travaux sont dans le domaine public, les auteurs en restent propriétaires, mais offrent la possibilité de les distribuer librement

[2] par exemple l'architecture RISC est connue pour être très efficace depuis une vingtaine d'années, pourtant bon nombre de microprocesseurs actuels utilisent une architecture concurrente, CISC... et utilisent en interne un schéma RISC

Chapitre I

Le Projet F-CPU et La Conception matérielle

1 Particularités du projet F-CPU

Le projet F-CPU est essentiellement basé sur un développement collaboratif au travers du réseau Internet. Ces dix dernières années, avec l'apparition de ce réseau ainsi que le développement du système GNU/Linux, bon nombre de projets libres sont apparus, le plus souvent couverts par la célèbre licence libre GPL (« Gauche d'Auteur »). Ce système convient parfaitement aux logiciels car il protège le code source qui une fois compilé devient un binaire exécutable. Mais les choses sont différentes avec la conception matérielle.
Jusqu'aux années 80, les concepteurs en électronique devaient principalement travailler au niveau des transistors, travail qui devenait de plus en plus dur à mesure que les systèmes se compliquent[1].
Mais avec l'apparition des langages de description matérielle VHDL et Verilog, il est possible d'écrire le « code source » d'une puce. De nombreux projets libres concernant le matériel sont apparus ces dernières années sur Internet[2]. Le projet F-CPU fait ainsi partie de ces projets d'un type particulier. Il leur est financièrement impossible de déposer un brevet pour protéger leur création. La seule protection qui existe pour l'instant est la licence GPL pour protéger les codes sources. Mais la couverture juridique de la GPL n'est pas applicable pour le résultat de la synthèse, c'est-à-dire la carte de placement des transistor, où il y a encore beaucoup de travail à faire pour arriver à une puce terminée. C'est pourquoi le projet ne considère que la conception décrite en langage de haut niveau (VHDL), qui correspond à la définition d'un code source, protégé par la GPL.

Il a donc été nécessaire de veiller à respecter le plus strictement les droits d'auteur et les brevets. Ainsi, il est impossible d'utiliser une bibliothèque VHDL existante mais n'étant pas sous licence GPL : par exemple le DesignWare de Synopsys, qui fournit des unités de calculs complètes et optimisés.

[1] la loi de Moore prédit un doublement de la vitesse des puces tous les 18 mois et est respectée depuis une trentaine d'années

[2] voir le site http ://www.opencores.org

a) Description du microprocesseur

Le projet vise donc à développer un microprocesseur entièrement nouveau, libre de tout brevet. Le coeur du microprocesseur est de type RISC, superpipeline, fonctionnant sur des mots de 64 bits et plus grâce à l'intégration de techniques SIMD. A ce jour, il s'agit probablement du seul projet de microprocesseur qui peut être entièrement paramétré : la taille d'un mot n'est pas limitée à 64 bits ! Avec le **même** binaire, il sera possible de faire tourner un programme sur un F-CPU instancié avec une taille de registre de 64 bits ou 128 bits. En effet, même si le mot de base fera 64 bits de large, un fondeur pourra très bien instancier un F-CPU d'une taille supérieure (puissance de 2 supérieure à 64 bits). C'est là l'un des principaux concepts qui font l'originalité du projet. Il faudra donc prendre en compte cette particularité dans le développement de l'unité de calcul : la taille des opérandes n'est pas connues à priori[1].
D'autres originalités caractérisent le microprocesseur du F-CPU, il est conseillé de lire le manuel disponible sur le site Internet `http://www.f-cpu.org` pour plus de détails.
Je me concentrerai sur les caractéristiques importantes qui ont influencé le développement de l'unité d'addition en virgule flottante.

b) SIMD

Le terme SIMD désigne une organisation parallèle des calculs, où une instruction donne un ordre identique pour traiter plusieurs données à la fois :

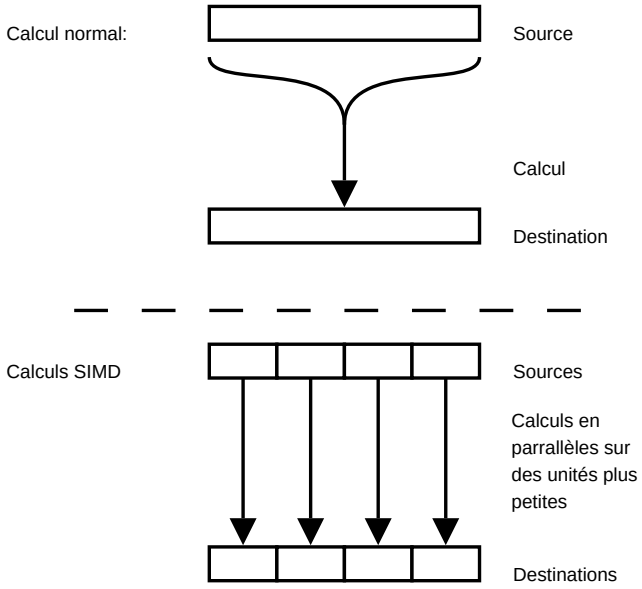


FIG. I.1 – Présentation du SIMD

Il s'agit donc de découper un long mot (qui sera, selon les spécifications du F-CPU, de taille une puissance de 2 supérieure ou égale à 64) en plusieurs mots courts, afin d'effectuer la même opération en parallèle. De plus, la taille du mot SIMD doit être modulaire, elle n'est pas connue à priori : **Il est donc nécessaire d'instancier autant d'unités que de tailles possibles**, cette taille étant paramétrable[1].

Pour l'unité de calcul flottant, comme nous le verrons par la suite, il n'y a que 2 tailles de « Chunk »[3] possibles : 32 bits (simple précision), et 64 bits (double précision). Cela a grandement influencé le développement car il faut que l'unité puisse calculer, par exemple pour des mots de 128 bits, soit 2 flottants double précision, soit 4 flottants simple précision, selon le mode SIMD choisi.

c) Superpipeline et estimation des délais

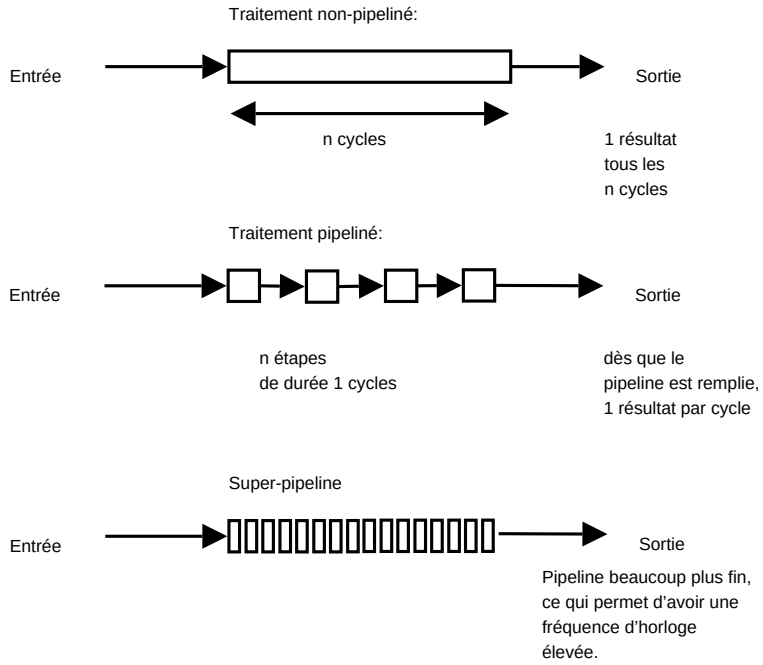


FIG. I.2 – Présentation du superpipeline

Le F-CPU est un véritable microprocesseur superpipeline, c'est-à-dire que la profondeur de chaque étage est très petite en nombre de transistors.
Le projet devra, à terme, pouvoir directement être comparable aux microprocesseurs les plus performants du marché[4].
Parce que le projet n'a pas les moyens ni les capacités techniques qu'ont les grands fondeurs tel qu'Intel ou STMicro-electronics pour effectuer des recherches sur la matière première, le silicium, le projet F-CPU doit utiliser une technique qui permette d'atteindre la même vitesse avec une technologie éventuellement moins performante.
En effet, l'équation est la suivante[1] :

$$\text{vitesse} = \text{technologie sillicium} \times \text{longueur du chemin critique}$$

ou plus précisément :

[3]un Chunk est un élément de calcul SIMD, par exemple on peut donner dans un mot de 64 bits 4 Chunks de 16 bits

[4]à l'origine, F-CPU devait être un « Merced Killer »

vitesse = vitesse d'un transistor × nombre de transistors dans le chemin critique

Remarque : Cette première analyse ne tenait pas compte des délais inhérents aux connexions, qui deviennent prépondérants dans les technologies silicium récentes.

Donc pour parvenir à une vitesse élevée, il faut minimiser le chemin critique.
Pour cela, le pipeline est utilisé pour découper une tâche longue en plusieurs tâches courtes et rapides que le processeur pourra effectuer en série. L'avantage principal est qu'une fois que le pipeline est rempli, le microprocesseur est capable de sortir un résultat tous les cycles d'horloge[5].
Le terme Superpipeline désigne en fait un découpage extrême du pipeline, c'est-à-dire réduire au maximum la granularité du pipeline. En effet, la profondeur du pipeline retenue pour le Core 0 du F-CPU est de *seulement* 10 transistors[1][6].
Cette profondeur, arbitrairement choisie, correspond à peu près à une moyenne de la profondeur de 6 portes logiques à 4 entrées, dans différentes technologies communes.

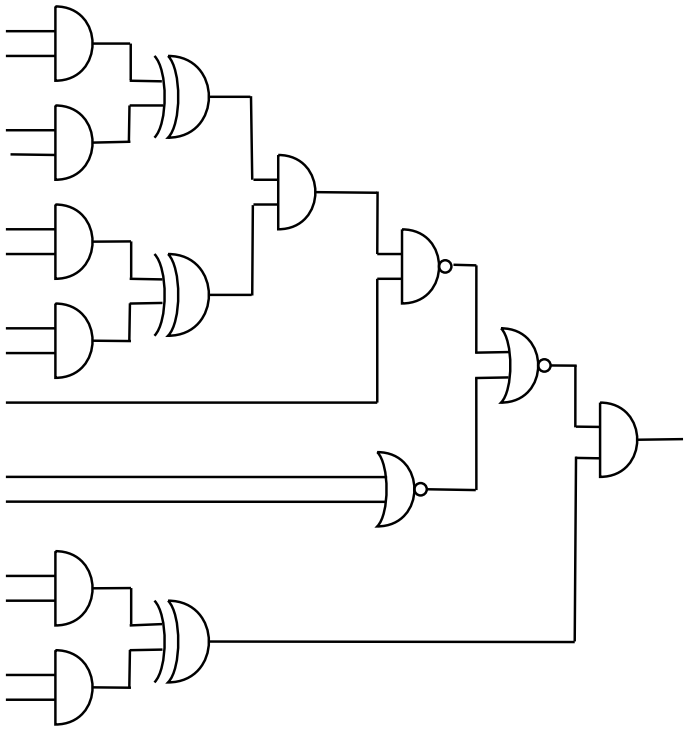


FIG. I.3 – Exemple de la profondeur du pipeline

Ici apparaît donc un important problème : il faut maîtriser la profondeur du chemin critique pour entrer dans cette granularité. Or, on ne connaît pas avec précision la technologie cible. Il faut que le code soit utilisable aussi bien sur un FPGA que en transistors sur un wafer de

[5]les problèmes inhérent à cette technique, notamment d'aléa, ne sont pas abordés, car ils dépassent le cadre du projet. En effet, ce n'est pas à une unité d'exécution comme c'est le cas pour l'addition en virgule flottante de s'occuper de ce que lui donne le séquenceur du microprocesseur

[6]en technologie PMOS, utilisant souvent des *pass transistores* Aujourd'hui les contraintes de bruit et les faibles tensions de fonctionnement rendent cette « règle » encore plus difficile à respecter.

Nom	Délai en portes (d)	Délai en transistors (t)
MUX2 :1	1	1
MUX4 :1	1	1
DEC2 (Décodeur 2 vers 1)	1	1
AND2	1	1
AND4	1	1
OR2	1	1
OR4	1	1
XOR2	1	2
XOR4	2	3
NOT	1	1

TAB. I.1 – Estimation de la profondeur d'éléments logiques communs

silicium (fondu dans une vrai puce). C'est pour cela que des portes à 4 entrées sont utilisées pour estimer la profondeur d'une tâche.
Néanmoins, certains contributeurs[7] du projet estiment que la règle des 6 portes logiques peut être violée si et seulement si celle des 10 transistors ne l'est pas.
Cette règle n'est donc pas un dogme mais une estimation de complexité relative pour que tous les étages fonctionnent à la même vitesse. Si une unité était 10% plus lente que toutes les autres, la fréquence du processeur diminuerait de 10% !

Aussi étrange que cela puisse paraître, il est estimé qu'un MUX (Multiplexeur) ne fait qu'une porte ou qu'un transistor de profondeur.

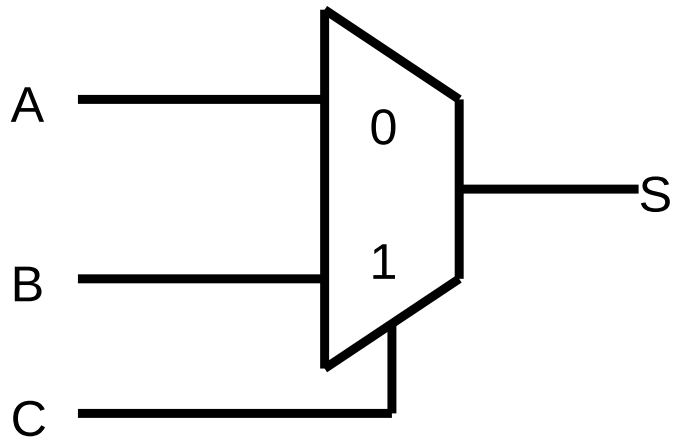


FIG. I.4 – Multiplexeur 2 vers 1

[7]dont Micheal Riepe, qui à réalisé bon nombre d'unités d'exécution, notamment les unités de calcul entier

En effet l'équation de base d'un MUX2 est :

$$S = A.C + B.\overline{C}$$

On attend donc une estimation de l'ordre de d=2/t=2 (d représentant le délai en portes, t le délai en transistors). Mais on peut facilement faire l'hypothèse que la technologie cible pourra fournir des multiplexeurs bien plus rapides que cette implémentation basique[8]. Il convient en fait de séparer les deux chemins caractéristiques du multiplexeur : le chemin de contrôle (sélecteur $\longrightarrow$ sortie) et le chemin de donnée (entrée sélectionnée $\longrightarrow$ sortie).
Le chemin de contrôle (« Controlpath ») a une plus grande latence que le chemin de donnée (« Datapath »), de l'ordre de d=2/t=2, alors que le chemin de donnée, représentant la plupart du temps le chemin caractéristique, n'a qu'un délai de d=1/t=1. Il est en effet plus long d'aiguiller les données (rôle du chemin de contrôle) que de passer au travers du MUX comme le fait le chemin de donnée.

De même, il peut sembler étrange d'estimer qu'un AND2 prenne autant de temps qu'un AND4, mais il faut se rappeler qu'il s'agit d'une estimation, et que le synthétiseur améliorera grandement les portes à 4 entrées, en utilisant par exemple des portes AOI (And/Or/Invert) en technologie CMOS[9].

Exemple :

A and B and C and D

sera optimisé par le synthétiseur par une porte AND4 de délai d=1/t=1. Mais à l'inverse,

(A and B) or (C and D)

ne le saura probablement pas, il convient donc de compter d=2/t=2.

Néanmoins, on peut considérer que toutes les portes simples (AND, OR, NAND, NOR) avec un nombre d'entrées inférieur ou égal a 4 auront un délai de d=1/t=1, et que toutes les portes avec un plus grand nombre d'entrées ne seront quant à elles qu'une combinaison de portes a 4 entrées. Les XOR étant plus compliqués, on estime qu'ils ont un délai de d=1/t=2 jusqu'à 2 entrées. C'est pour cela que M. Riepe force le synthétiseur, dans les unités qu'il a écrit, à n'utiliser que des portes à 4 entrées[10]. Faisant confiance à son expérience, j'ai essayé de suivre ce « modèle ».

Une autre difficulté consiste à maîtriser les délais lors des équations logiques pour les optimiser.
Soit l'équation suivante

sel = Cout and (not(g) or (MSB and LSB))

[8] là encore, je m'appuie sur les propos de M. Riepe

[9] Merci a Michael de me l'avoir expliqué a plusieurs reprises tard le soir...

[10] il faut remarquer que cela risque de poser quelques problèmes si ce code est synthétisé sur un FPGA ne fournissant que des cellules à 2 entrées

Si on effectue une estimation du délai, on arrive à d=3/t=3.
g est le bit de garde, ce signal « arrive » très tôt à l'endroit où il faut effectuer l'opération (après l'addition entière).
LSB est le bit de poids faible de l'opération d'addition, il sort donc relativement rapidement de l'additionneur (c'est en fait un XOR des bits de poids faibles des 2 opérandes d'entrées). Donc si g arrive a d=0/t=0, LSB arrive à d=1/t=2.
C'est différent pour le MSB (bit de poids fort du résultat) et Cout (retenue sortante) : ceux-ci vont arriver bien plus tard (délai total n de tout l'additionneur). Donc par rapport à l'arrivée de g en d=0/t=0, la sortie sel sera fixée à d=n+3/t=n+3.

On peut donc écrire l'équation différemment en utilisant un MUX : voir figure I.5.
De ce fait, les signaux aux entrées du multiplexeur seront prêts lorsque les signaux de contrôle arriveront. Le délai sera donc de d=n+2/t=n+2. Attention, ici, c'est le « control path » qui fixe la vitesse.

Il convient donc de maîtriser parfaitement les délais du chemin critique tout au long de la conception, en vue d'un découpage final en étages.

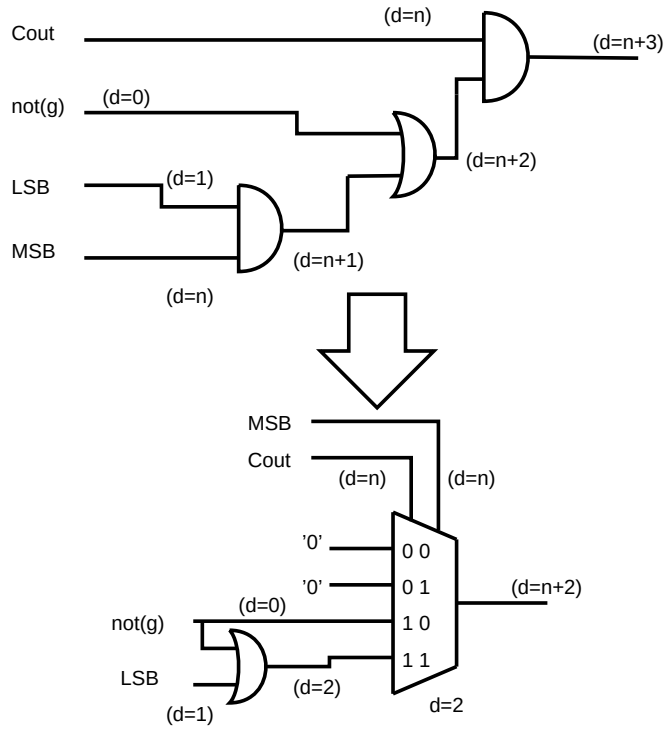


FIG. I.5 – Optimisation du délai d'une opération booléenne

d) Outils de développement

Le projet F-CPU étant un projet de la communauté du libre, on s'attend qu'il utilise des outils libres, comme GNU/Linux le fait avec la chaîne de développement libre GCC. Malheu-

reusement, les outils existant dans le domaine du libre ne sont pas facilement utilisables, voire incomplets. Par exemple, le prometteur compilateur/simulateur ghdl n'est pas suffisamment stable pour être utilisable.
Alliance (`http://www-asim.lip6.fr/alliance/`) est envisagé mais ne pourra être utilisé que lors de l'étape finale de routage car il ne dispose pas d'un front-end VHDL standard.
Comme il n'est pas envisageable d'utiliser un outil commercial cher dans le cadre d'un projet libre sans financement, il faut donc rechercher dans les version d'évaluation de ces outils commerciaux. La plupart suppriment une grande partie de leurs caractéristiques pour la version d'évaluation, ou imposent un temps limité d'utilisation.
Il s'avère néanmoins que Simili (`http://www.symphonyeda.com`) convient parfaitement à un développement centré sur la simulation, avec la synthèse envisagée. Simili n'est malheureusement pas un outil libre mais il a l'avantage d'être l'un des outils qui respecte le mieux les standards IEEE, il a aussi une interface assez conviviale ainsi que des restrictions pas trop contraignantes[11].
Une capture d'écran est disponible en annexe A.

Simili ne faisant pas de synthèse, le synthétiseur Synopsys dont l'ISIMA dispose sera utilisé pour effectuer des tests de « synthétisabilité ».

2 Représentation des nombres en virgule flottante : la norme IEEE-754-1985

Lorsque les premières unités de calcul en virgule flottante ont été implémentées dans différents microprocesseurs, chaque constructeur avait sa norme. Pour faciliter la programmation, il a rapidement été nécessaire de définir une norme. Ce fut réalisé en 1985 avec la norme IEEE-754[8]. Cette norme décrit une représentation d'un nombre à virgule flottante, ainsi que la gestion des problèmes (exceptions), mais ne décrit pas d'algorithme. Elle distingue deux principales tailles : les flottants simple précision (dits « single ») codés sur 32 bits, et les flottants double précision (dits « double ») codés sur 64 bits. D'autres format existent (notamment les flottants codés sur 80 bits, les flottants double précision étendue), mais ceux-ci sont moins utilisés. Le projet F-CPU prévoit d'implémenter les formats single et double mais se réserve la possibilité d'implémenter d'autre formats à venir.

a) Virgule flottante simple précision

Ce format est stocké sous la forme d'un mot de 32 bits organisé sous la forme « bit de signe, vecteur exposant, vecteur mantisse ».

On retrouve le nombre représenté avec la formule suivante :

$$FP = +/- \ Mantisse \times base^{Exposant}$$

Le signe est représenté par un bit. 0 signifie que le nombre est positif, 1 signifie que le nombre est négatif.

[11] pas de mode debug, pas de possibilité de visualiser plus de 10 signaux dans la vue « waveform »

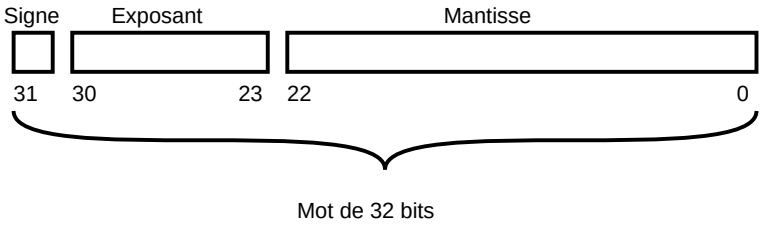


FIG. I.6 – Codage d'un flottant simple précision

L'exposant est un vecteur qui contient un nombre entier non-signé mais baisé, c'est-à-dire auquel il faut retirer une constante appelée biais pour obtenir un nombre signé. Pour la simple précision par exemple, l'exposant est stocké dans un champ de 8 bits, donc qui peut représenter un nombre de 0 à 255. Le biais est de 127, l'exposant représente ainsi un entier entre -127 et +126.
La base communément utilisée en informatique est bien entendu la base 2
La mantisse représente un nombre non-signé qui a la particularité d'être normalisé. En effet, tout nombre à virgule peut être considéré comme ayant une partie décimale suivie d'un nombre quelconque de zéro suivis d'un autre nombre différent de zéro. Il suffit de commencer la représentation par le premier chiffre différent de zéro et de faire correspondre l'exposant et on a une mantisse normalisée. En binaire, la mantisse commencera donc nécessairement par un 1 au bit de poids fort (MSB) avec l'exposant correspondant. On ne représente donc pas ce 1 dans le stockage de la mantisse, ce qui permet de gagner un bit en précision de cette mantisse. Ainsi, pour une mantisse de 23 bits, on a une précision de 24 bits.
Mais la représentation d'un nombre flottant n'est pas toujours la même, elle peut dépendre de la valeur de ces champs :

1. si $0 < Exp < 255$ alors, on obtient le nombre représenté par :

$$FP = (-1)^{sign} \times 2^{exp-127} \times 0,1.Man$$

   $1.Man$ signifiant que la mantisse est normalisée, il faut donc rajouter un 1 à gauche de celle-ci.
2. si $Exp = 0$ et $Man = 0$, alors $FP = 0$
3. si $Exp = 0$ et $Man \neq 0$, alors le nombre est représenté de façon dénormalisée[12]
4. si $Exp = 255$, le vecteur représente une valeur spéciale en fonction de la mantisse :
   - si $Man = 0 \ldots 01$, $FP = SNaN$, Signaling Not A Number.
   - si $Man \neq 0 \ldots 01$ et $Man \neq 0 \ldots 0$, $FP = QNaN$, Quiet Not a Number.
   - si $Man = 0$, $FP = \pm\infty$

   De plus amples explications seront fournies dans la suite.

On peut donc représenter des nombres en virgule flottante avec une précision relative dans l'intervalle 10^{-38} à 10^{38} pour les nombres positifs (et -10^{38} à -10^{-38} pour les nombres négatifs), avec le codage en simple précision.

[12] les nombres dénormalisés seront décrits dans la section f)

b) Virgule flottante double précision

Le codage en double précision se distingue du codage en simple précision uniquement par la taille de ses champs.

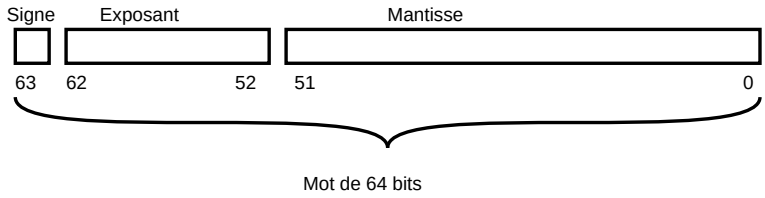


FIG. I.7 – Codage d'un flottant double précision

C'est la seule différence intrinsèque. D'autres différences en découlent, comme par exemple le biais de l'exposant qui est de 2047.
L'étendue des nombres représentables va de $\pm 10^{-308}$ à $\pm 10^{308}$.

c) Nombres spéciaux

La norme IEEE 754 définit la représentation de nombres dits spéciaux. Ce ne sont pas des nombres en tant que tels. Deux types de nombres spéciaux existent :

Not A Number (Pas un Nombre) : abrégé en NaN, cette représentation caractérise un résultat d'une opération qui n'est pas représentable dans le format demandé, ou que le résultat de l'opération n'existe pas mathématiquement. La norme IEEE demande de faire la distinction entre les NaN verbeux et silencieux :

- **Quiet NaN** (NaN silencieux) : résultat de l'opération invalide, sans aucun sens mathématique. Ne génère pas d'exception lors de la simple propagation (un QNaN est en entré) mais peut en générer une lorsque sa création. De plus, la représentation d'un QNaN autorise plusieurs types de QNaN a être définis par le programmeur, par exemple, pour identifier les opérations invalides.
- **Signaling NaN** (NaN verbeux) : signale la présence d'un SNaN uniquement en entrée de l'opération. Ne doit jamais être généré par une opération, mais peut être généré en mémoire par le programmeur.

+/- infini : La norme demande de pouvoir représenter le concept de nombre infini. Les opérations mathématiquement possibles sur les infinis doivent l'être dans l'unité :

$$\forall a \in \mathbb{R}, \infty + a = \infty$$

$$\infty + \infty = \infty$$

Les tableaux I.2 et I.3 indiquent les sorties à avoir selon les configurations des opérandes A et B.

Addition $A+B$		B				
		F	$-\infty$	$+\infty$	QNaN	SNaN
A	F	F	$-\infty$	$+\infty$	QNaN	SNaN
	$-\infty$	$-\infty$	$-\infty$	QNaN	QNaN	SNaN
	$+\infty$	$+\infty$	QNaN	$+\infty$	QNaN	SNaN
	QNaN	QNaN	QNaN	QNaN	QNaN	SNaN
	SNaN	SNaN	SNaN	SNaN	SNaN	SNaN

TAB. I.2 – Sorties spéciales : L'Addition

Soustraction $A-B$		B				
		F	$-\infty$	$+\infty$	QNaN	SNaN
A	F	F	$+\infty$	$-\infty$	QNaN	SNaN
	$-\infty$	$-\infty$	QNaN	$-\infty$	QNaN	SNaN
	$+\infty$	$+\infty$	$+\infty$	QNaN	QNaN	SNaN
	QNaN	QNaN	QNaN	QNaN	QNaN	SNaN
	SNaN	SNaN	SNaN	SNaN	SNaN	SNaN

TAB. I.3 – Sorties spéciales : La Soustraction

d) Exceptions IEEE

La norme IEEE spécifie la notion d'exception[9] comme étant un évènement extraordinaire entrainant l'impossibilité d'effectuer le calcul correctement.
La norme définit 5 exceptions :

Invalid Operation (Opération invalide) : Doit arriver quand l'un des opérandes est NaN, ou ou que le résultat ne peut être défini mathématiquement : $(+\infty)-(+\infty)$, $(-\infty)-(-\infty)$, $0*\infty$, $0.0/0.0$, ∞/∞.
Attention : $\infty+\infty=\infty$ et $(-\infty)-(-\infty)=-\infty$ sont des opérations correctes.

Divide By Zero (Division par zéro) : Exception levée lors d'une division par zéro. Cette exception est gardée principalement pour des raisons historiques [9]. Il s'agit aussi d'une des exceptions qui arrive le plus souvent. Ne concerne pas l'addition en virgule flottante.

Exponent Overflow (Dépassement de capacité de codage de l'exposant par le haut) : le résultat de l'opération est fini mais ne peut pas être stocké dans le format de nombre flottant demandé. La norme demande alors de retourner comme résultat l'infini (∞) qui convient.

Exponent Underflow (Dépassement de capacité de codage de l'exposant par le bas) : même raison que pour l'Overflow, mais le nombre est trop petit pour être représentable par un nombre normalisé. Un nombre dénormalisé pourra être utilisé si possible, ou 0.0 dans le cas contraire.

Inexact (Résultat inexacte) : Le résultat ne tient pas dans le format de destination et doit être arrondi, ce qui fausse le résultat. La plupart du temps, cette exception est ignorée logiciellement[9].

e) Arrondis IEEE

Lorsque le résultat d'une opération a une plus grande précision que le format n'accepte, la norme IEEE 754 définit 4 méthodes permettant d'éviter une simple troncature source d'imprécision[4].
Note : LSB signifie Least Significant Bit (bit le moins significatif) de la mantisse résultat. D'autres bits situés à droite du LSB (donc qui seront tronqués) pilotent les algorithmes : g, r et s. Ceux-ci sont définis en détail page 20.

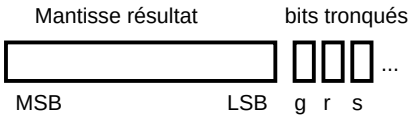


FIG. I.8 – Présentation de la mantisse et des bits pilotant l'arrondi

1. Round To Nearest (Arrondi au plus proche) :

```
si (g=1) ET ((LSB=1) OU (r=1 OU s=1)) alors
        Ajouter 1 au resultat et tronquer au niveau du LSB
sinon
        Tronquer au niveau du LSB
finsi
```

2. Round Toward $+\infty$:

```
si signe=positif ET ((g=1) OU (r=1) OU (s=1)) alors
        Ajouter 1 au resultat
sinon
        Tronquer au niveau du LSB
finsi
```

3. Round Toward $-\infty$:

```
si signe=negatif ET ((g=1) OU (r=1) OU (s=1)) alors
        Ajouter 1 au resultat
sinon
        Tronquer au niveau du LSB
finsi
```

4. Round Toward Zero (Arrondi à Zéro) :

```
Tronquer au niveau du LSB
```

f) Nombres dénormalisés

Comme on l'a vu dans la section a), les nombres très proches de zéro ($< 10^{-38}$) ne peuvent être représentés avec une mantisse normalisée. Pour garantir une certaine continuité dans la représentation autour de 0, l'utilisation de nombres dénormalisés est introduite.

L'exposant est nul, et la mantisse représente alors le nombre directement (pas de normalisation), et on retrouve le nombre FP par la formule :

$$FP = (-1)^{sign} \times Man \times 2^{1-B-P}$$

où B est le biais de l'exposant ($B = 127$ pour les single, $B = 2047$ pour les double), P est la résolution de la mantisse ($P = 23$ pour un single, $P = 52$ pour les double)[8, 10].
Le 1 implicite a donc disparu pour atteindre les plus petits nombres en mettant à zéro les bits de poids fort de la mantisse. On remarque une perte de précision à mesure que l'on se rapproche de 0.
Le principal défaut la gestion de ces nombres dénomalisés dans les processeurs actuels est que cela entraîne un nombre très important d'opération supplémentaires. Par exemple, dans un microprocesseur AMD Thunderbird, une opération avec un nombre dénormalisé prend 30 fois plus de temps qu'avec un nombre normalisé.[10].

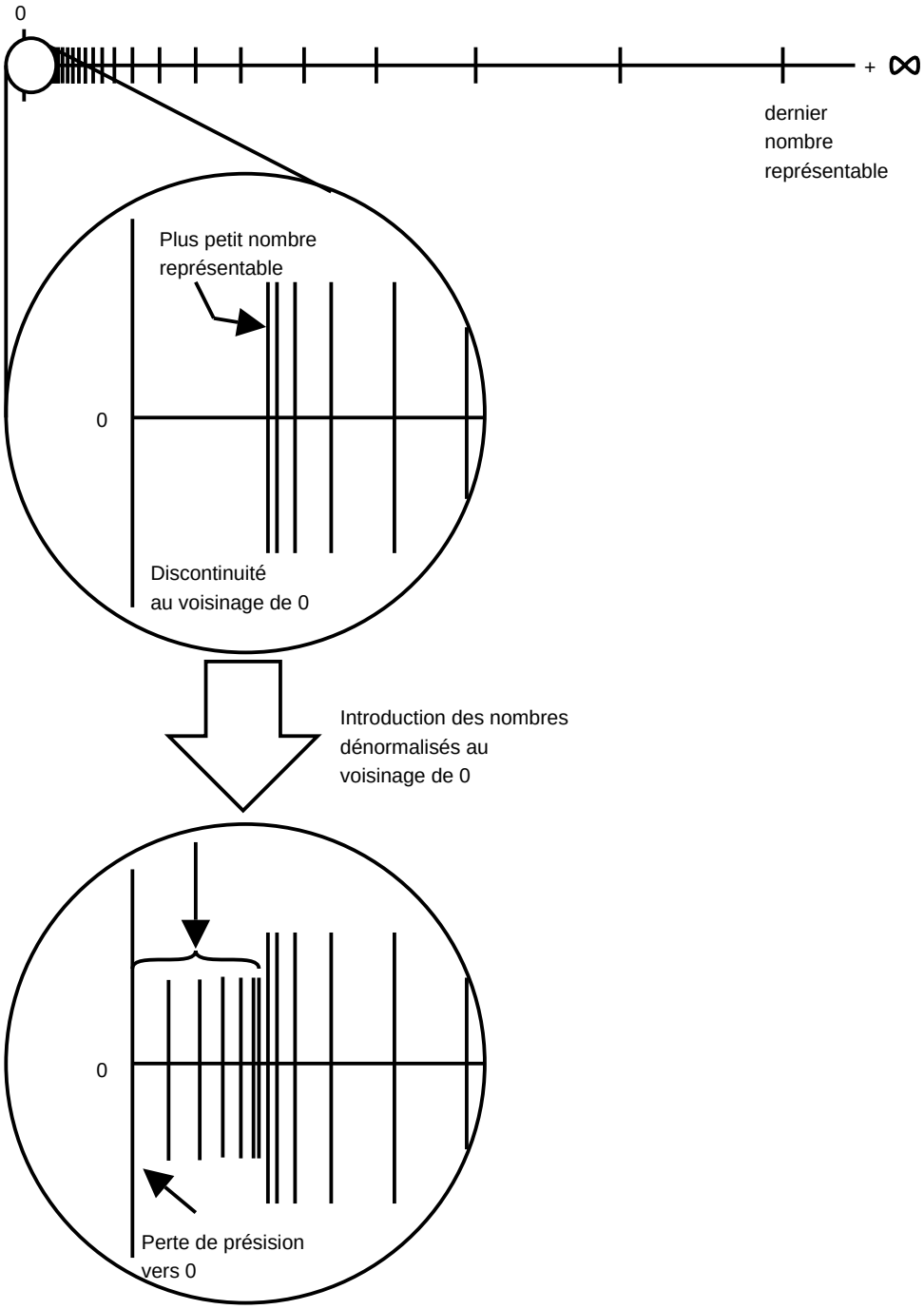


FIG. I.9 – Étendue des nombres représentables en virgule flottante

Chapitre II

Addition en virgule flottante

L'addition est certainement l'opération en virgule flottante la plus utilisée dans les programmes de calcul scientifique. Mais au delà de sa simplicité apparente, l'opération est l'une des plus complexes à réaliser le plus rapidement possible de par sa nature hautement séquentielle, et très difficilement parallélisable. Une addition en virgule flottante prend sur les unités de calcul actuels quasiment autant de temps qu'une multiplication[2]. La principale difficulté pour réduire la latence est de réduire le nombres d'additions entières portant sur les mantisses, qui peuvent être au nombre de 3 lors d'une addition.
Plusieurs algorithmes ont été étudiés pour réduire cette latence et profiter au maximum de l'aspect « superpipeline » du microprocesseur F-CPU. Il a notamment été étudié un modèle réduisant de manière significative le nombre d'opérations à effectuer en série. Ces travaux sont décrits précisément dans [2], [4] et [5], et c'est une description rapide qui sera dressée dans cette partie. Le lecteur pourra s'y référer pour de plus amples détails, détails dépassant les cadres de ce papier.

1 Additionneur trivial

Les étapes de l'addition/soustraction flottante triviale sont donc les suivantes[1][2, 4, 5, 6] :

1. Soustraction des exposants : L'exposant Ea du premier opérande est soustrait à l'exposant Eb du second opérande : $DE = Ea - Eb$
2. Alignement : décalage bit à bit vers la droite de la mantisse du plus petit opérande de DE bits.
3. Addition des mantisses : Addition (ou soustraction) des mantisses suivant leur signe et l'opération d'addition ou de soustraction à effectuer.
4. Conversion : si le résultat de l'addition/soustraction des mantisses est négatif, il faut effectuer un complément à 2 de la mantisse résultat.
5. Détection du Premier 1 (Leading One Detection) : déterminer combien de décalages vers la droite ou la gauche il faut effectuer pour avoir le bit de poids fort de la mantisse à 1 (en vue de la normalisation)
6. Normalisation : Effectuer ce décalage

[1] ce sont ces étapes séries que l'on retrouve notamment dans les algorithmes logiciels de calcul flottant

7. Arrondi IEEE : ajouter 1 au résultat si nécessaire.

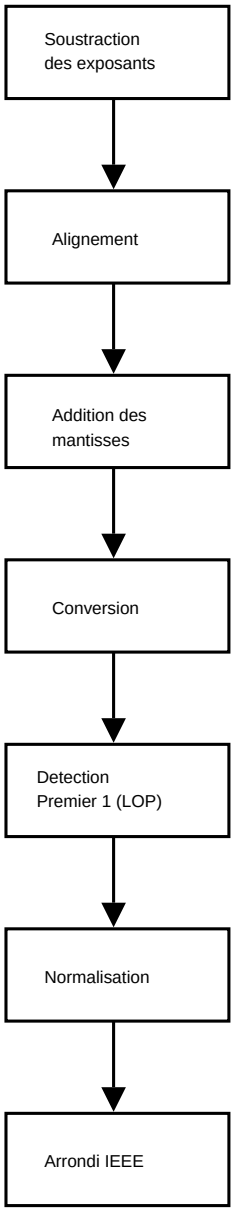


FIG. II.1 – Schema de principe de l'additionneur

Comme chaque étape dépend de la précédente, aucune ne peut vraiment être effectuée en parallèle d'une autre. C'est ici que se situe le principal problème de l'additionneur.
Néanmoins quelques observations permettent de tirer quelques conclusions :

1. l'étape de conversion est uniquement nécessaire quand le résultat est négatif. En effectuant un échange des mantisses en fonction du signe de la différence de leur exposant (DE), on saura alors laquelle des mantisses sera plus grande que l'autre et le signe du résultat de l'addition des mantisses sera connu quand $DE \neq 0$. Le problème se pose vrai-

ment quand les exposants sont égaux ($DE = 0$), alors on ne peut savoir laquelle des deux mantisses est la plus grande. Le résultat de l'addition pourra donc encore être négatif et nécessiter une conversion. Mais dans ce cas, l'absence de grand décalage (entraînant la perte des bits de poids faibles) rendra inutile l'étape d'arrondi. L'étape d'arrondi et conversion deviennent alors mutuellement exclusives[2].

2. Il existe plusieurs méthodes permettant d'effectuer une LOD en parallèle avec l'addition des mantisses, c'est-à-dire prédire à quelle position sera le 1 le plus à gauche dans le résultat. On effectue alors une LOP (Leading One Prediction), décrite dans le document [5].
3. Enfin, l'utilisation d'un additionneur entier spécial appelé Compound Adder qui permet de sortir à la fois le résultat et le résultat incrémenté permet de décaler l'étape d'arrondi en mettant sa détermination en parallèle avec l'addition et de n'avoir en sortie de cette addition qu'une étape de sélection entre le résultat et le résultat incrémenté[2][4].

Il est donc possible de réduire de manière significative le chemin critique en parallélisant certaines opérations mais en ajoutant une complexité qui n'apparaissait pas aux premiers abords lors de l'étude de l'addition flottante.
Le résultat est donc l'addition en virgule flottante appelée « Double Datapath ».

2 Amélioration de l'additionneur : Présentation de l'addition « Double Datapath »

Cette section décrit plus en détail l'additionneur « Double Datapath ». Je garderai le terme anglais pour des raisons de conformité et de clarté[3].
Il convient donc de séparer 2 cas :

1. lorsqu'on a une addition effective[4] ou quand la différence entre les exposants (DE) est supérieure strictement à 1

2. et lorsque l'on est en soustraction effective et que cette différence est est égale à 0 ou 1

Dans la littérature, ces datapath sont appelés « CLOSE » et « FAR ». Je garderai les termes anglosaxons pour faciliter la compréhension.
Nous en arrivons donc au schéma explicatif figure II.2.

Dans le « CLOSE Datapath », la première étape consiste donc à effectuer la différence des exposants pour obtenir le décalage des mantisses à faire. Puis, selon le signe de cette différence, les mantisses sont échangées (« SWAP »). Puis le décalage est effectué (« Right Shift »). Enfin, les mantisses sont ajoutées.
Dans le « FAR Datapath », la première étape consiste à commencer la prédiction des zéros en tête (concrètement, nous verrons qu'il s'agit d'effectuer la différence sur les mantisses). Puis, les mantisses sont ajoutées (récupérées depuis l'échange (SWAP) du « FAR Datapath ».

[2] malheureusement, ce ne sera pas directement le cas pour les tous les modes d'arrondi

[3] c'est surtout parce que dans la suite je suis amené à définir un autre chemin de données, donc je pourrai utiliser le terme anglais pour le « Datapath » de la théorie, et le terme français pour désigner les « chemins de données » que j'ai été amené à concevoir pour la gestion des SIMD

[4] définition de l'addition/soustraction effective page 20

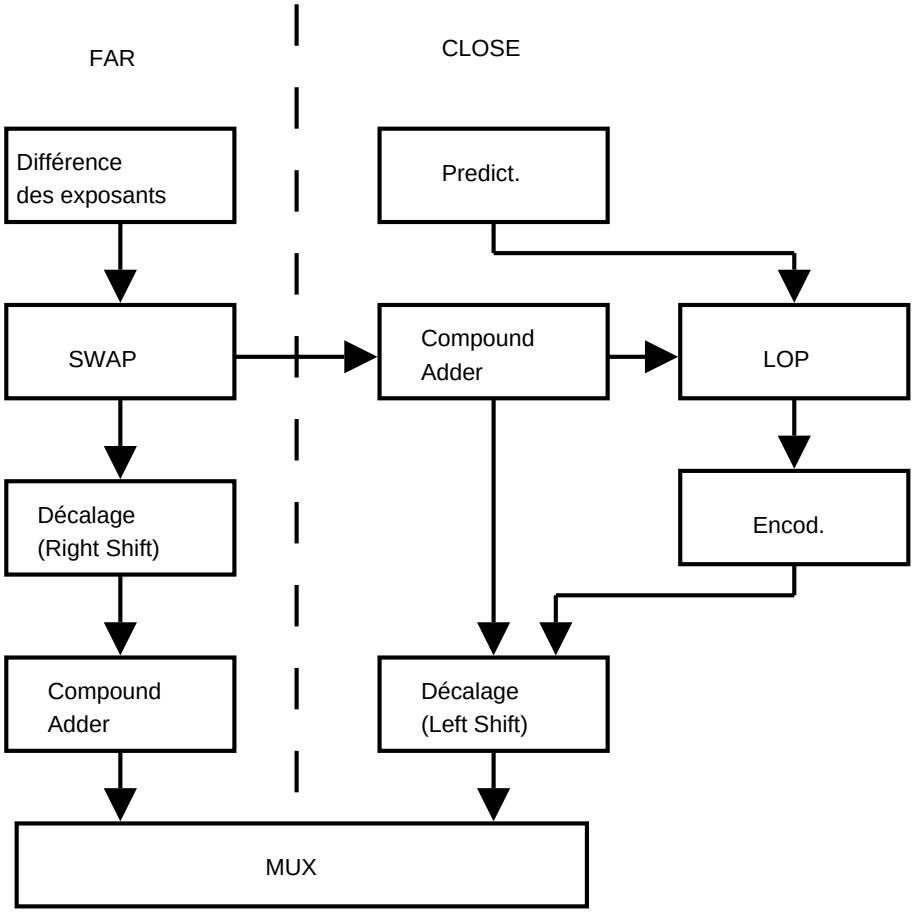


FIG. II.2 – Schéma de principe de l'additionneur Double Datapath

En parallèle, la prédiction de zéro en tête (LOP) est lancée. Toujours pendant l'addition des mantisses, le résultat de la prédiction est encodé et corrigé pour piloter le décaleur. Enfin, le décaleur effectue le décalage du résultat.
La gestion des modes d'arrondi n'est plus montrée sur ce schéma car il s'agit de logique pure pilotant principalement des multiplexeurs (notamment celui à la sortie des Compound Adder).

L'exécution d'étapes en parallèle réduit le chemin critique, mais augmente aussi de manière significative le nombre de transistors (nous avons maintenant 2 étapes d'addition des mantisses). Cette technique est largement utilisée dans les FPU actuels, sous une forme ou sous une autre[2, 12].
La clé maîtresse de cet algorithme est l'utilisation d'un Compound Adder, un additionneur entier capable de sortir la sortie normale ainsi que la sortie incrémentée. Ce Compound Adder sera utilisé lors de la soustraction des exposants pour obtenir le signe de cette opération ainsi que la valeur absolue du résultat[5]. Deux Compound Adder sont aussi utilisés pour effectuer les additions sur les mantisses (un dans chacun des datapath).

[5] moyennant un inverseur, comme nous le verrons par la suite

Avant de passer à une explication de l'algorithme en détail, je vais décrire les unités logiques utilisées indépendamment de leur contexte, pour bien comprendre leur fonctionnement, avant de les montrer assemblées. Je commencerai par la définition d'un certain nombre de termes récurrent dans la suite.

a) Définitions

Je définirai ici les termes que je vais utiliser par la suite et qu'il est indispensable de connaître pour comprendre l'algorithme « Double Datapath ».

Le terme **soustraction effective** désigne le fait de devoir soustraire les mantisses au lieu de les ajouter. On aura une soustraction effective selon les signes des opérandes ainsi que l'opération addition/soustraction qu'il faut effectuer :

$$\text{EffSub} = \text{SignA xor SignB xor Sub}$$

où SignA et SignB représentent les signes des opérandes A et B, et Sub désigne s'il faut faire l'opération d'addition ou de soustraction.
Par là même, l'addition effective est définie lorsqu'on n'a pas la soustraction effective.

L'utilisation d'un Compound Adder pour effectuer l'étape d'arrondi en même temps que l'addition des mantisses engendre un problème : dans le cas de flottant double précision, par exemple, la mantisse fait 53 bits. A cause du décalage de ces mantisses, pour avoir un résultat exact, il faudrait utiliser un additionneur 105 bits[6]. Mais comme nous ne sommes intéressés que par les bits de poids fort, un additionneur 53 bits est utilisé pour éviter d'ajouter trop de transistors[2]. Pour l'étape d'arrondi, il convient donc d'ajouter des informations sur les bits

[6]le décalage peut aller jusqu'à 52 bits

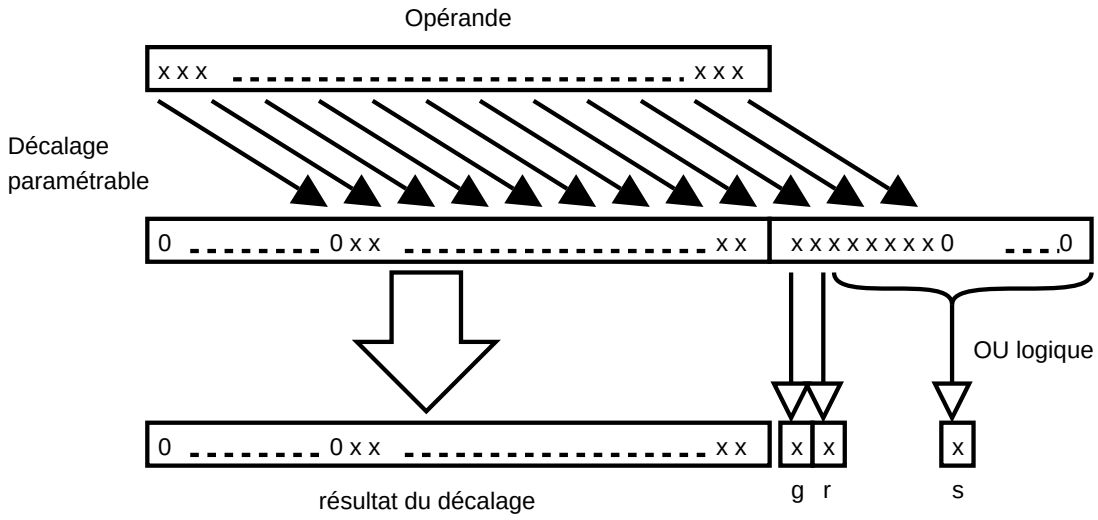


FIG. II.3 – Bits de garde, d'arrondi et collant

qui se sont « perdus » lors du décalage. On défini donc 3 bits : le **bit de Garde** g, le **bit d'Arrondi** r ainsi que le **bit Collant** s (sticky). Le bit de garde est le bit de poids fort du vecteur représentant les bits éjectés lors du décalage. Le bit d'arrondi est quant à lui le second bit de poids fort. Le bit collant est calculé en effectuant un OU logique sur tous les autres bits sortis lors du décalage.

Addition entière utilisée

Lorsqu'il faut effectuer une addition entière, l'unité utilise directement la fonction generic_adder écrit par Michael Riepe qui est un additionneur type Compound Adder à retenue anticipée. De plus la taille des opérandes n'est pas fixe pour cet additionneur (« carry-select adder »), on peut donc instancier des additionneurs entiers de n'importe quelle taille. Ce type d'addition a été préféré à des types plus complexes car il peut facilement être découpé entre plusieurs étages du pipeline, et surtout parce que le compound adder était déjà écrit.

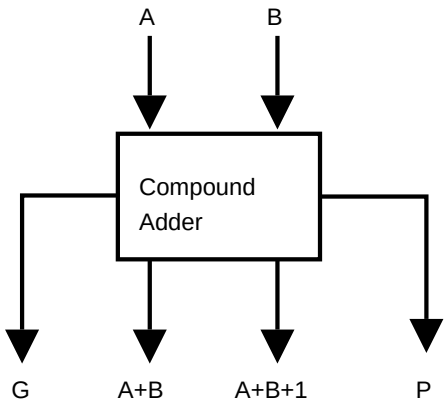


FIG. II.4 – L'additionneur « Compound Adder »

En plus du résultat et du résultat incrémenté ($A + B$ et $A + B + 1$), l'addition entière sort le bit de retenue sortante G (« Carry Out »), ainsi qu'un bit P signalant qu'une retenue entrante peut se propager jusqu'à la retenue sortante (« Carry In may propagate »).

b) La soustraction des exposants

Le soustracteur d'exposant prend les exposants en entrée et en donne la différence en valeur absolue ainsi que le signe de cette différence (et non pas une notation binaire signée classique).

Principe

On utilise pour cela la formule caractéristique de la soustraction binaires en représentation en complément à 2 (pour A et B deux entrées) :

$$A - B = A + \overline{B} + 1$$

Grâce au Compound Adder, on a $A + B + 1$. Il suffit de faire une négation de B pour obtenir $A - B$. De plus, en regardant le bit de poids fort, on obtient le signe de l'opération.

Pour obtenir la valeur absolue de la différence, il suffit de remarquer que la première sortie du Compound Adder, $A + B$ est en fait égale à $\overline{B - A}$. En effet, $B - A = B + \overline{A} + 1$ mais aussi $B - A = \overline{\overline{B} + A}$.
En ajoutant un inverseur à la sortie du Compound Adder, on obtient aussi l'opposé. Il suffit ensuite de sélectionner la valeur positive selon le signe.

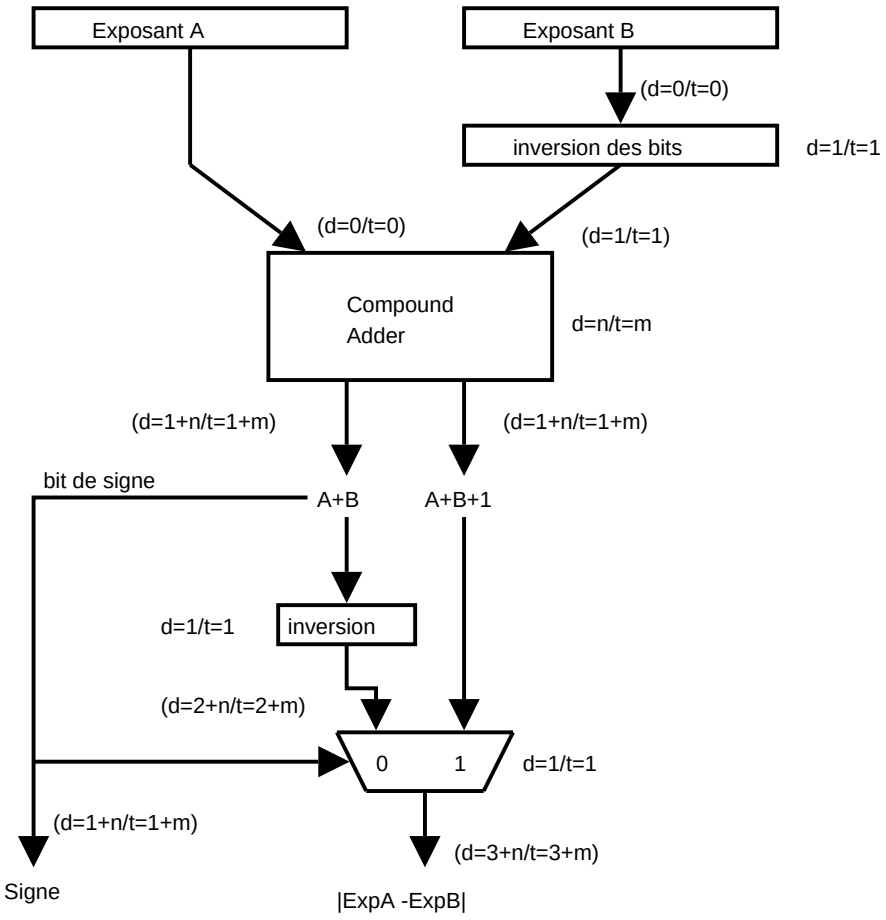


FIG. II.5 – Soustracteur d'exposants

Estimation des délais

Un inverseur coûte d=1/t=1 en délai, le multiplexeur est estimé à d=1/t=1 aussi. L'unité Compound Adder est calquée sur le generic_adder de Michael Riepe dont les délais (estimé par ses soins) dépendent de la taille des opérandes. Ces délais sont précisés dans le tableau II.1.

On en déduit l'estimation des délais dans le cas simple précision, et double précision : voir le tableau II.2.

Remarque : la limite des 6 portes de profondeur est dépassée, il conviendra donc de découper cette unité entre 2 étages.

Taille	Délai en portes (d)	Délai en transistors (t)
0..4	4	6
5..8	5	7
9..16	6	7
17..32	7	8
33..64	8	9
65..128	9	10
129..256	10	11
257..512	11	12
513..1024	12	13

TAB. II.1 – Delai du generic adder

Taille	Delai en portes (d)	Delai en transistors (t)
Simple précision (32 bits)	8	10
Double précision (64 bits)	9	11

TAB. II.2 – Estimation du délai du soustracteur d'exposant

c) L'étape d'inversion des mantisses

Après avoir récupéré le signe de la différence des exposants, on doit effectuer une inversion (un « swap ») des mantisses. Ceci est effectué par de simples MUX. Voir figure II.6.

Remarque : on voit sur la figure II.5 que le signe de la différence est prêt bien avant que la valeur absolue ne sorte, on peut donc effectuer l'étape de « swap » pendant que cette valeur

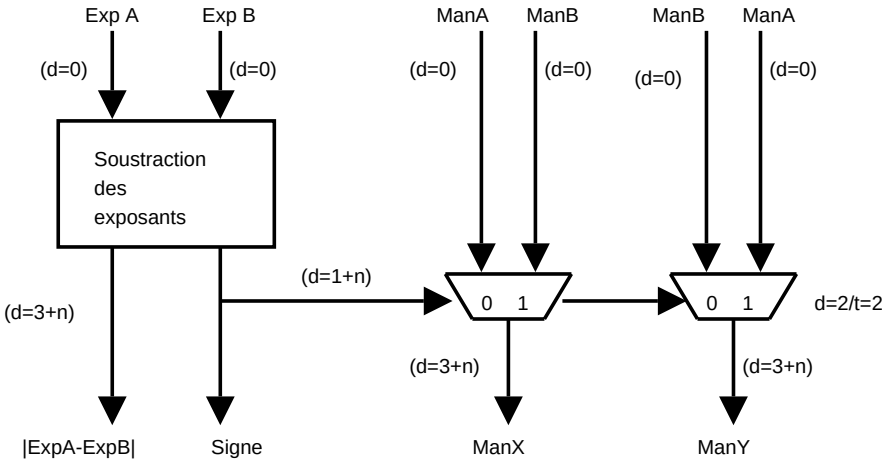


FIG. II.6 – Inversion des mantisses

absolue ne sorte. De ce fait, le chemin critique de ces deux opérations est en fait le chemin critique du soustracteur d'exposant. Néanmoins, comme expliqué page 7, c'est le signal de contrôle des multiplexeurs qui arrive en dernier, c'est donc le « Controlpath » qui fixe le délai de ces multiplexeurs, donc on estime ce délai à d=2/t=2.

d) Le décaleur vers la droite paramétrable

Principe

La soustraction des exposants fournit le nombre de bits dont il faut décaler la plus petite mantisse pour l'aligner sur la plus grande. Ce nombre est codé en binaire naturel. Pour cela un décaleur vers la droite paramétrable[13] (right shifter) est utilisé :

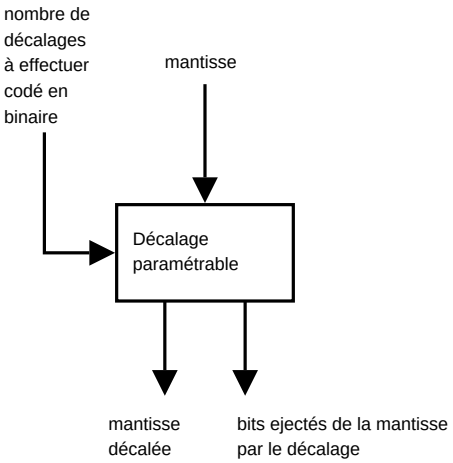


FIG. II.7 – Decaleur paramétrable

Le décaleur prend en entrée la mantisse à décaler, ainsi qu'un autre vecteur représentant en binaire l'amplitude du décalage vers la droite à effectuer. En sortie, on récupère la mantisse décalée ainsi qu'un vecteur contenant les bits éjectés lors de ce décalage qui seront utilisés pour obtenir les bits de garde, d'arrondi et collant. Les bits sont alignés à droite à l'intérieur de ce dernier vecteur, de sorte que le bit de garde soit directement le bit de poids fort.
Cette solution a été préférée à celle proposée dans [13], à savoir effectuer les OU à chaque étage du décalage pour sortir directement le bit collant (sticky bit). En effet, cela augmente inutilement le délai et rend un découpage du décaleur difficile.

Estimation du delai

A chaque étage du décaleur, on ne retrouve qu'une rangé de multiplexeurs, une ligne par bit du mot représentant le nombre de décalages à effectuer : voir figure II.8.

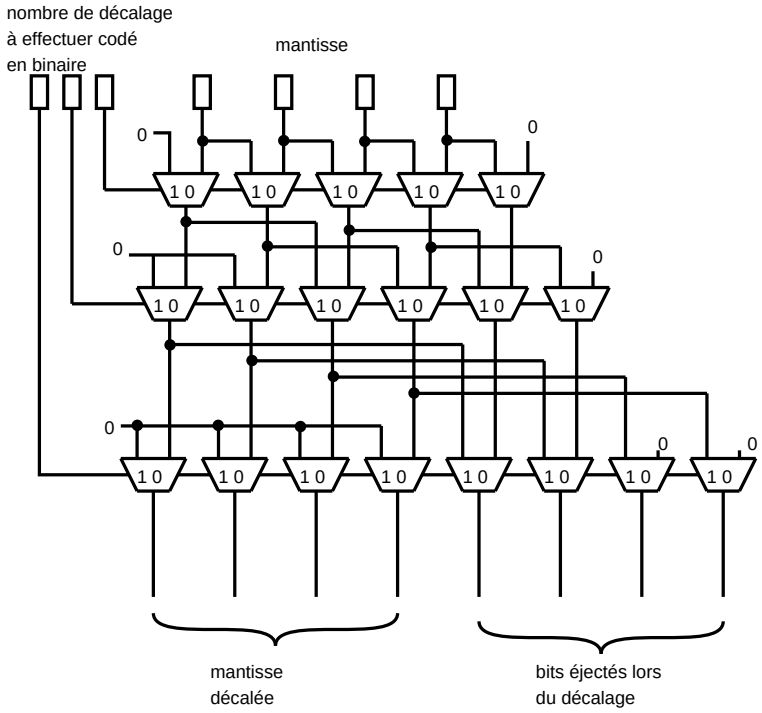


FIG. II.8 – Decaleur vers la droite parametrable 4 bits

Le délai est donc de $d{=}n$, avec n le nombre de bit du mot indiquant le nombre de bit qu'il faut décaler.
Ainsi, pour les flottants simple précision, l'exposant est stocké sur 8 bits. Mais la mantisse n'est pas stockée dans un mot de 2^8 bits, mais seulement 23 bits, donc il ne sert à rien de s'occuper des bits de poids fort au delà du $5^{\text{ème}}$ bit ($2^5 = 32 > 24$). La présence d'un bit à 1 pour les poids plus forts mettra le résultat à zéro. De même pour les flottants double précision dont la mantisse fait 53 bits, on n'utilisera que les 6 bits de poids faible.

Taille	Délai en portes (d)	Délai en transistors (t)
Simple précision (32 bits)	5	5
Double précision (64 bits)	6	6

TAB. II.3 – Estimation du délai du Décaleur vers la droite paramétrable

Si j'ai délibérément choisi de ne pas effectuer les opérations OU sur les bits éjectés pour calculer le bit collant entre chaque étage, c'est qu'à ce stade de la conception, je ne peux savoir exactement comment sera découpé l'additionneur en étage du pipeline. N'ayant qu'une estimation, et surtout n'ayant pas encore pris en compte ni le SIMD, ni les nombres dénormalisés, il convient donc de prendre des précautions. Et mettre de grands OU logiques empêcherait de facilement couper le décaleur, par exemple, entre le $3^{\text{ème}}$ et le $4^{\text{ème}}$ étage. Les OU seront donc effectués en parallèles de l'étape d'addition des mantisses.

e) Comparateur

Un comparateur est utilisé pour comparer les mantisses. Le but est de savoir si la mantisse B est plus petite strictement que la mantisse A. Donc le résultat pourra n'être qu'un seul bit (et non pas 2 bits comme on pourrait s'y attendre).

Le principe est d'utiliser une ligne de cellule primaire puis un arbre de cellules secondaires, comme le montre la figure II.9.

Les cellules primaires codent la différence entre a_i et b_i sur deux bits p_i et v_i : v_i code le fait d'avoir ces deux bits égaux, et dans le cas où ils sont différents, p_i a un sens et représente le fait que $a_i > b_i$:

a_i	b_i	p_i	v_i	Signification
0	0	0	0	Les deux entrées sont égales
0	1	0	1	$a_i \neq b_i$, $a_i < b_i$
1	0	1	1	$a_i \neq b_i$, $a_i > b_i$
1	1	1	0	Les deux entrées sont égales

TAB. II.4 – Table de vérité de la cellule primaire du comparateur binaire

La cellule secondaire quant à elle permet de propager intelligemment le résultat des cellules primaires.

Enfin, au dernier étage ne restent que 2 signaux p et v, s'ils sont tous les deux à 1, cela signifie que $ManB > ManA$. Il suffit donc d'ajouter un ET logique.
Le délai estimé de ce comparateur :

Taille	Délai en portes (d)	Délai en transistors (t)
Simple précision (32 bits)	8	9
Double précision (64 bits)	9	10

TAB. II.5 – Estimation du délai du comparateur binaire

Amélioration

En effet, page 7, la complexité des multiplexeur a été abordée, et après amples discutions avec Micheal Riepe, il semblerait que l'utilisation de MUX2 :1 impose une contrainte importante sur le délai des lignes de Cellules Secondaires (car les signaux de contrôle arrivent en même temps que les signaux de données), c'est pour cela que les délais sont augmentés de 1. Les autres étages de multiplexeurs ne sont pas affectés car le signal de contrôle arrive tôt

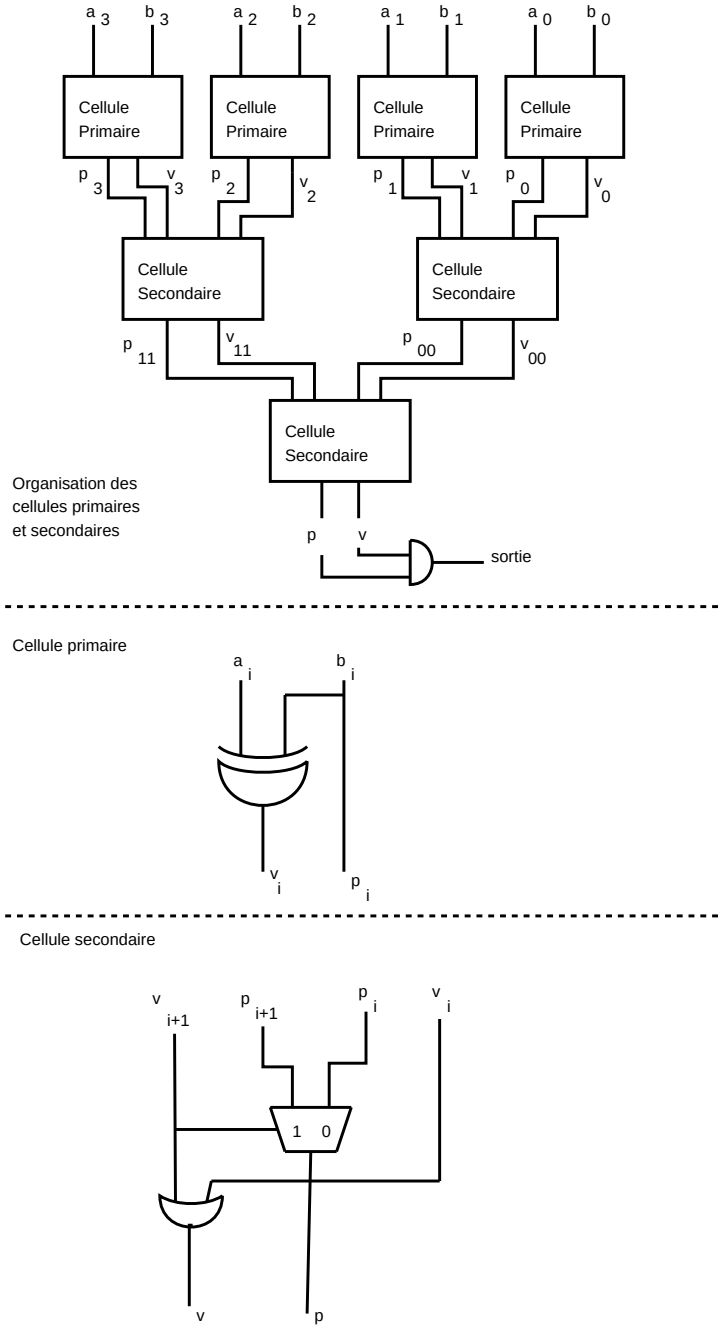


FIG. II.9 – Comparateur de mantisse

(avant les données).
La nouvelle version du comparateur utilise donc des MUX4 :1.

f) Prédicteur du nombre de zéros en tête (Leading One Predictor)

Principe

Comme nous l'avons vu page 18, il est possible de prédire combien il y aura de zéros en tête du résultat à la sortie de l'additionneur de mantisse (significant adder). Cela permet d'éviter l'étape de comptage de l'algorithme trivial. C'est ce que l'on appelle un Prédicteur de zéros en tête, que j'appellerai désormais LOP.
Il est important de noter que l'on a besoin d'un LOP seulement dans le « CLOSE Datapath », c'est-à-dire en cas de soustraction effective des mantisses ave des exposants proches ($De = 0, 1$).
L'algorithme décrit dans [5] est très intéressant, c'est cette méthode qui sera adaptée pour cette additionneur. L'algorithme suppose de connaître exactement quelle est la mantisse la plus grande. Ma sera la mantisse la plus grande, et Mb la mantisse la plus petite.

La structure du LOP peut se diviser en 2 parties : L'encodage et la correction[5].

L'encodage se divise en deux modules : pré-encodage qui permet de mettre les entrées sous une forme exploitable et l'arbre d'encodage qui permet de générer une représentation binaire du nombre de zéros en tête. Mais dans certains cas, des erreurs peuvent se glisser dans cette représentation, c'est pourquoi une étape de correction est effectuée après.
La correction se divise en trois modules : pré-encodage, arbre de détection pour détecter les cas pathétiques où il faudra corriger le résultat donné par l'encodage, et le module de correction qui effectue cette correction[5].

Comme on peut le voir sur la figure II.10, ces opérations peuvent être effectuées en parallèles.

Je passerai rapidement sur la description de chacun des modules, ceux-ci sont en effet très bien détaillés dans le document [5]. De plus, les notations mathématiques utilisées seront les notations usuelles : $+$ est l'opération OU logique, $.$ est l'opération ET logique, $\otimes$ est le XOR,...

Encodage de la position

Le module de pré-encodage a pour but de fournir une représentation caractéristique du résultat à partir des opérandes Ma et Mb, qui sera fourni à l'arbre d'encodage. Cette représentation doit être de la forme 0^k1, c'est-à-dire, avoir un nombre éventuellement nul de zéros vers les bits de poids forts, suivi d'un 1 indiquant la position du premier 1 dans le résultat de la soustraction des mantisses (sans l'effectuer, bien sûr). C'est pour cela que l'algorithme demande de savoir quelle mantisse sera la plus grande. En effet, on aura $Ma \geq Mb$, donc $Ma - Mb$ sera bien de la forme 0^k1 et non pas $0^k(-1)$. C'est pour lever cette indétermination que l'on fixe $Ma \geq Mb$.
Selon la configuration des bits suivants, on peut en déduire quelle sera la forme des bits de poids forts du résultat. De plus, on peut détecter les configurations où des problèmes peuvent éventuellement se poser, et qui seront corrigés dans le module de correction.

Ensuite, l'arbre d'encodage permet de transformer cette représentation en une représentation binaire. Le principe utilisé est un arbre similaire à celui utilisé dans le comparateur.

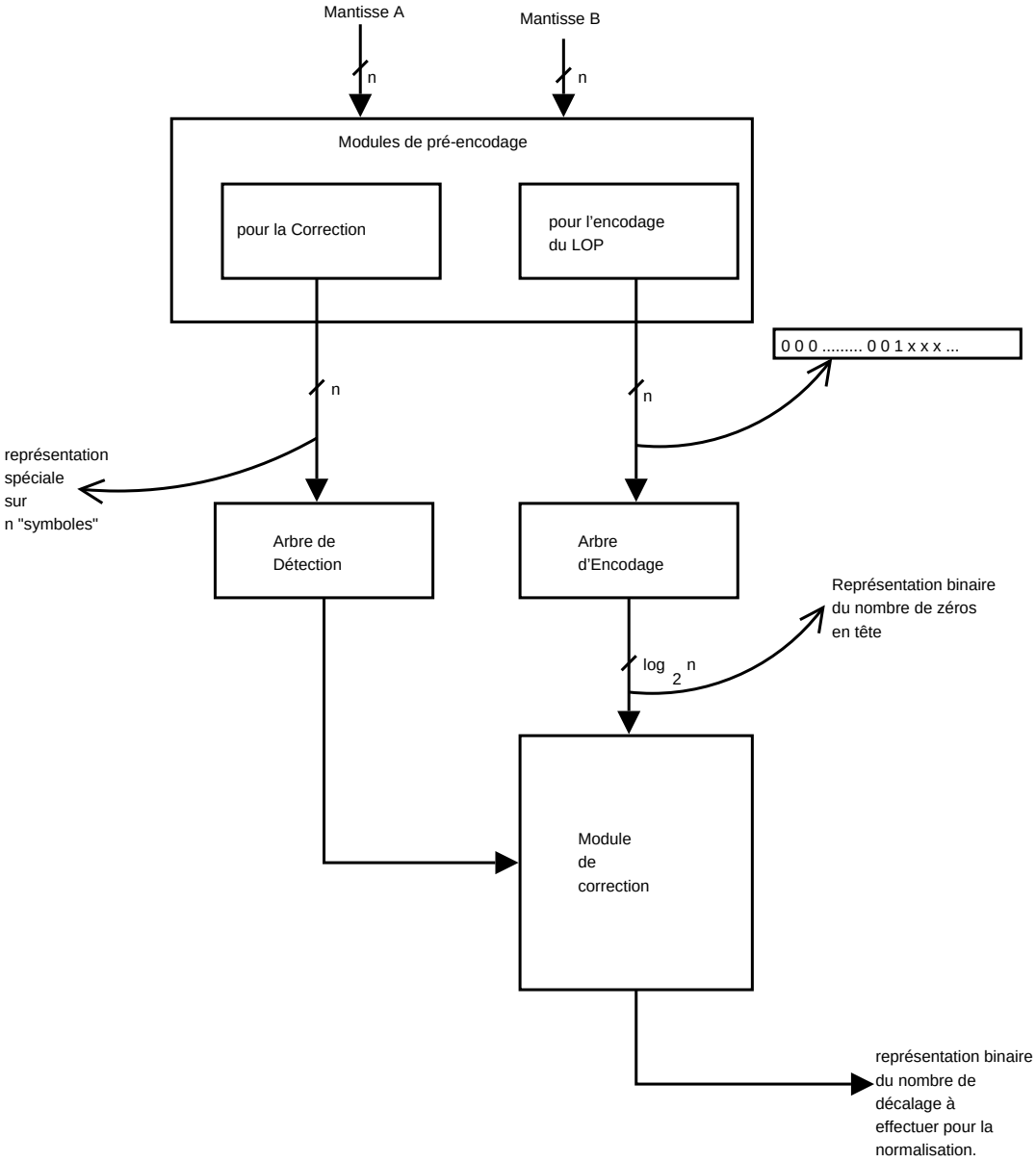


FIG. II.10 – Organisation des modules du LOP

Pour plus d'informations, se reporter directement au document [5], où le schéma de l'arbre est donné page 10.

Correction de position

Il y a deux configurations où il peut y avoir une erreur de position, il convient donc de les corriger le cas échéant. L'erreur est d'une position au maximum.

Le module de pré-encodage se charge de donner à l'étape suivante une représentation du

résultat de manière à identifier les cas litigieux.

L'arbre de détection transforme cette représentation et identifie s'il s'agit d'un cas pathétique ou non, et dans ce cas, essaye de voir s'il faut corriger ou pas.

Le module de correction est quant à lui un simple traducteur permettant de piloter la normalisation (décalage vers la gauche de la mantisse).

g) Arrondi

La gestion des arrondis s'effectue en parallèle des autres unités, et ne correspondent pas en tant que telle à des unités d'exécution. Il s'agit principalement de toute la logique nécessaire pour contrôler les sorties des Compound Adder en vue de sélectionner le résultat normal ou le résultat incrémentée (voir page 13 pour plus d'explication sur les modes d'arrondi IEEE).
Les équations booléennes sont données dans la suite. Se reporter à [4] pour leur démonstration.

L'utilisation normale est donnée par la figure II.11.

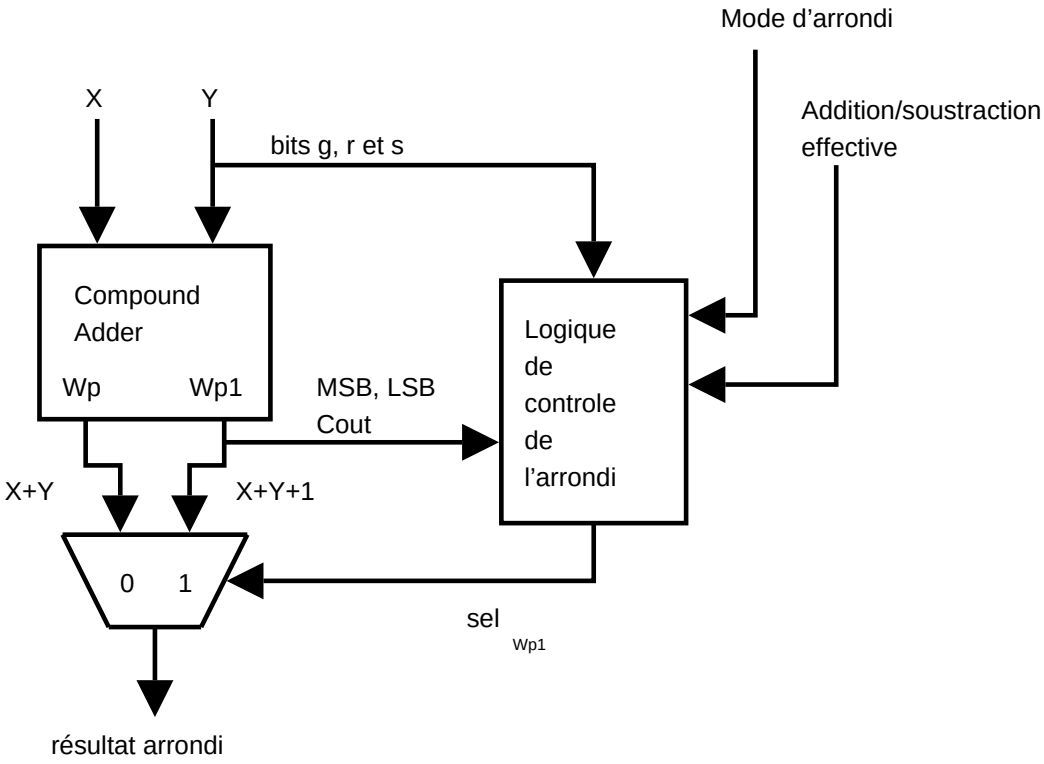


FIG. II.11 – Organisation de la logique de contrôle d'arrondi avec les Compound Adder

Il s'agit donc pour les unités de contrôle d'arrondi de sélectionner la sortie des Compound Adder : soit Wp la sortie normale, ou $Wp1$ la sortie incrémentée. Néanmoins, un problème se pose dans le cas du mode d'arrondi à l'infini[4]. Dans ce cas, il est nécessaire d'obtenir la sortie

$Wp2$ incrémentée deux fois $(X+Y+2)$ dans le cas d'une addition affective avec dépassement de capacité (Overflow)[4]. Mais il suffit d'ajouter une ligne de Half-Adder avant l'additionneur que l'on pilotera grâce à un signal indiquant si on est dans un mode d'arrondi à l'infini : *up*. Cette ligne de Half adder va nous donner deux vecteurs : le vecteur somme S et le vecteur retenue (Carry) C. En remplaçant le bit de poids faible L du vecteur de retenu par 0, on pourra obtenir avec le compound adder [4] :

1. $A+B+1$ et $A+B+2$ si $L=0$. On pourra obtenir $A+B$ en remplaçant par 0 le bit de poids faible de $A+B+1$ qui est obligatoirement 1.
2. $A+B$ et $A+B+1$ si $L=1$. On obtiendra $A+B+2$ en changeant le bit de poids faible de $A+B+1$ par 1.

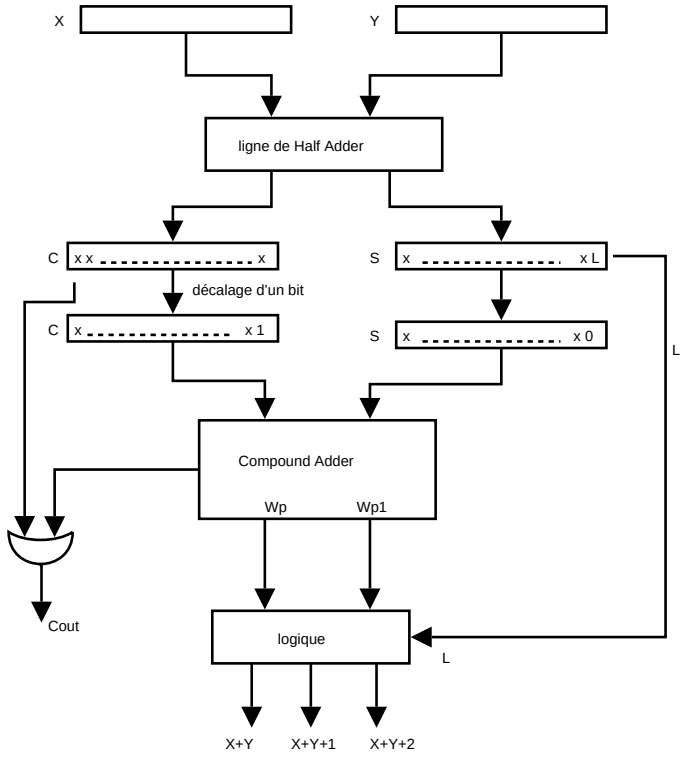


FIG. II.12 – Utilisation de half adder pour obtenir 3 sorties avec le Compound Adder

Les signaux contrôlés par la logique d'arrondi sont définis par la suite, pour obtenir une démonstration et de plus amples précisions, se reporter à [4].
Il est à noter que le délai de la logique de contrôle de l'arrondi ne devrait pas influencer le chemin critique, car effectué en parallèle de l'addition des mantisses.

CLOSE datapath

$sel_{close}^{nearest}$ est le signal qui sélectionnera la sortie incrémentée du Compound Adder dans le « CLOSE Datapath », pour le mode d'arrondi au plus proche (Rounding To Nearest). sel_{close}^{∞} correspond à cette même sélection mais pour les modes d'arrondi à l'infini (+ et -). Dans le cas

d'arrondi à zéro, la sortie sélectionnée sera toujours la sortie normale et elle sera simplement tronquée. $BSHIN$ correspond à la valeur du bit à insérer à droite lors du décalage vers la gauche suivant.

$$sel_{close}^{nearest} = C_{out}(\overline{g} + MSB \cdot L)$$
$$sel_{close}^{\infty} = C_{out}(\overline{g} + up \cdot MSB)$$
$$BSHIN = C_{out} \cdot g$$

FAR datapath

Nous retrouvons des variables qui jouent les mêmes rôles que dans le « CLOSE Datapath », mais la logique est plus complexe car il faut aussi corriger le bit de poids faible du résultat dans les modes d'arrondi à l'infini.

Variables d'entrées :
path fixe quel chemin choisir entre le « FAR » et le « CLOSE » Datapath.

$$path = sub \cdot (d(n-1) + d(n-2) + \cdots + d(1))$$

d est la représentation de la différence des exposants. Il suffit donc de faire un arbre de OU logique sur le vecteur d (sauf le bit de poids faible).
sign est la valeur du signe du résultat.

$$sign = sign(opA) \cdot \overline{EffSub} + (\overline{sign(d)} \cdot (sign(OpA) \oplus sign(A-B)) + sign(d) \cdot \overline{sign(OpA)})$$

$sign(OpA)$ et $sign(OpB)$ désignent les bits de signe des opérandes A et B respectivement. $sign(d)$ est le bit de signe de la différence des exposants, $sign(A-B)$ est le signe de l'opération $ManA - ManB$, tiré du bit de retenue sortante du Compound Adder du « CLOSE Datapath ». up est un signal indiquant si on est dans un mode d'arrondi à l'infini et que le résultat doit être arrondi[4] :

$$up_{+\infty} = \overline{sign(OpA)} \cdot \overline{EffSub} + (\overline{sign(d) \oplus sign(OpA)}) \cdot EffSub$$
$$up_{-\infty} = sign(OpA) \cdot \overline{EffSub} + (sign(d) \oplus sign(OpA)) \cdot EffSub$$
$$up = up_{+\infty} \cdot round_{+\infty} + up_{-\infty} \cdot round_{-\infty}$$

Variables intermédiaires :
$sel_{far.Sp1}^{\infty}$ définit s'il faut choisir la sortie incrémentée une fois $(A+B+1)$ de l'ensemble Half-Adder + Compound Adder pour les modes d'arrondi à l'infini, $sel_{far.Sp2}^{\infty}$ s'il faut choisir la sortie incrémentée deux fois $(A+B+2)$.
$sel_{far}^{nearest}$ désigne quelle sortie du Compound Adder il faut choisir pour le mode d'arrondi au plus proche (pas besoin du résultat incrémenté deux fois dans ce cas).
sel_{Sp1}, sel_{Sp2} et sel_S font le point de cette sélection quelque soit le mode d'arrondi choisi.
$shiftright$ et $shiftleft$ définissent s'il faut faire un décalage respectivement à droite ou à gauche pour l'étape de normalisation.

$$sel_{far.Sp1}^{\infty} = \begin{cases} up \cdot \overline{C_{out}}(g+r+s) & si\ addition\ effective \\ C_{out} \cdot (\overline{grs} + up \cdot (g \cdot (r+s) + MSB))) & si\ soustraction\ effective \end{cases}$$

$$sel^{\infty}_{far.Sp2} = add \cdot up \cdot C_{out} \cdot (L + r + r + s)$$

$$sel^{nearest}_{far} = \begin{cases} \overline{C_{out}} \cdot g \cdot (L + r + s) + C_{out} \cdot L \cdot ([LSB - 1] + g + r + s) & si\ addition\ effective \\ C_{out} \cdot (\overline{grs} + gr + MSB \cdot g \cdot (L + s)) & si\ soustraction\ effective \end{cases}$$

$$sel_{Sp1} = sel^{nearest}_{far} \cdot round_{nearest} + sel^{\infty}_{Far.Sp1} \cdot round_{\infty}$$

$$sel_{Sp2} = sel^{\infty}_{far.Sp2} \cdot round_{\infty}$$

$$sel_S = \overline{sel_{Sp1} + sel_{Sp2}}$$

$$shiftright = C_{out} \cdot \overline{sub}$$

$$shiftleft = \overline{MSB} \cdot sub$$

Variables de sorties :
sel_{Wp1} désigne quelle sortie du Compound Adder il faut effectivement choisir.
$BSHIN$ est la valeur du bit à introduire lors du décalage de l'étape de normalisation.
$shiftenable$ et $left/rightshift$ piloteront l'étape de normalisation (décalage vers la droite d'un bit ou vers la gauche de n bit).
$LSB_{corrige}$ désigne la valeur du bit de poids faible à corriger (dans tous les cas d'arrondi).

$$sel_{Wp1} = sel_{Sp2} + sel_{Sp1} \cdot L$$

$$BSHIN = \begin{cases} C_{out} \cdot (\overline{g}rs + g\overline{r}) & si\ round\ to\ nearest \\ C_{out} \cdot (up \cdot (g \oplus (r + s)) + \overline{up} \cdot g) & si\ round\ to \pm\infty \end{cases}$$

$$shiftenable = shiftright + shiftleft$$

$$left/rightshift = shiftright$$

$$LSB_{corrige} = sel_{Sp1} \oplus L$$

h) Pour plus de détails

Pour une explication interactive des éléments, notamment le décalage paramétrable ainsi que le LOP, le lecteur pourra se référer à [13].

3 Implémentation de l'additionneur « Double Datapath »

Le fonctionnement de addition en virgule flottante est décrit sur le schéma II.13. On y retrouve tous les éléments décrits individuellement précédemment. Il est à mettre en parallèle avec celui de la page 19.
Il y a bien au début la soustraction des exposants et la comparaison des mantisses. Puis on retrouve les étapes d'échange (SWAP).

Dans le « FAR Datapath », le décaleur vers la droite est piloté par la valeur absolue de la différence des exposants (étape d'alignement). Ce décalage sort le vecteur « aligné » ainsi qu'un vecteur contenant tous les bits qui ont été éjectés, qui sont alignés à gauche dans ce

vecteur. Ces deux vecteurs alimentent une ligne d'inverseurs sur les bits de la mantisses B en cas de soustraction effective, pour effectuer $A - B = A + \overline{B} + 1$.
Puis suit la ligne de Half Adder qui est activée quand le mode d'arrondi est $\pm\infty(up = 1)$.
Puis le Compound Adder calcule la somme et la somme incrémentée une fois ; en parallèle, la logique d'arrondi sort le signal Sel_{Wp1} et la correction du bit de poids faible pour obtenir le bon résultat arrondi.
Enfin, la dernière étape consiste à un décalage d'un seul bit à droite ou à gauche, selon l'opération effective.

Dans le « CLOSE Datapath », pendant la soustraction des exposants, les mantisses sont comparées, en préparation du LOP. Puis, lorsque les mantisses arrivent (X et Y), un décalage d'un bit vers la droite sert d'alignement, mais comme la différence entre les exposants dans ce « Datapath » ne sera prise en compte que si $d = 0, 1$, il suffit de piloter ce décalage par la valeur du bit de poids faible de d.
Puis Y passe dans une ligne d'inverseurs.
Le Compound Adder effectue la somme et la somme incrémentée, pendant que la logique d'arrondi sélectionne la bonne sortie, et que le LOP prévoit la position du premier 1 dans le résultat.
Ensuite, une rangée d'inverseurs est activée en cas de résultat négatif (chose qui ne peut arriver dans le « FAR Datapath »)[7].
Enfin, la normalisation finale est en fait un décaleur vers la gauche paramétrable, piloté par le résultat du LOP.

La gestion du SIMD n'est pas montrée sur le schéma. En effet, cela rendrait celui-ci incompréhensible. Un double chemin de données est utilisé pour éviter d'avoir des parties qui seront inutilisées dans l'additionneur.
Du fait qu'il y ait deux tailles distinctes d'opérande possibles, on doit mettre autant de chemins de données que de tailles : un pour les opérandes en double précision, deux pour les simple précision, par « Chunk » de 64 bits. Voir la figure II.14. Mais cele pose un problème d'efficacité : selon le mode SIMD (double ou simple), la moitié de la puce n'est pas utilisée ! Pour éviter cela, on ne définit que deux chemins de données :

1. « Double Précision OU Plus Haut Simple Precision » : ce chemin de données est emprunté par les opérandes en tant que Double si SIMD est mis sur Double, ou par l'opérande Single placée le plus haut dans le « Chunk » de 64 bits, moyennant de judicieux alignements dans les vecteurs.
2. « Plus bas Simple Précision » : est emprunté dans tous les cas par l'opérande Single placée le plus bas dans le « Chunk »

On le voit, cela permet d'éviter que toute une partie de la puce ne soit pas utilisée quand on est dans un mode SIMD ou autre, mais cela pose quelques contraintes : avant chaque élément, il faut effectuer des alignements : par exemple avant la soustraction des exposants, il faut aligner à gauche l'exposant simple précision (8 bits), dans un vecteur de 11 bits si on est en mode SIMD Single, et compléter les bits de poids faible par des 1 pour que la retenue se propage.
Cela implique un certain nombre de logique de sélection tous au long de l'additionneur, ce qui ajoute un certain délai. Voir la figure II.15.

[7] qui pourra éventuellement être supprimée grâce à la présence de la comparaison des mantisses

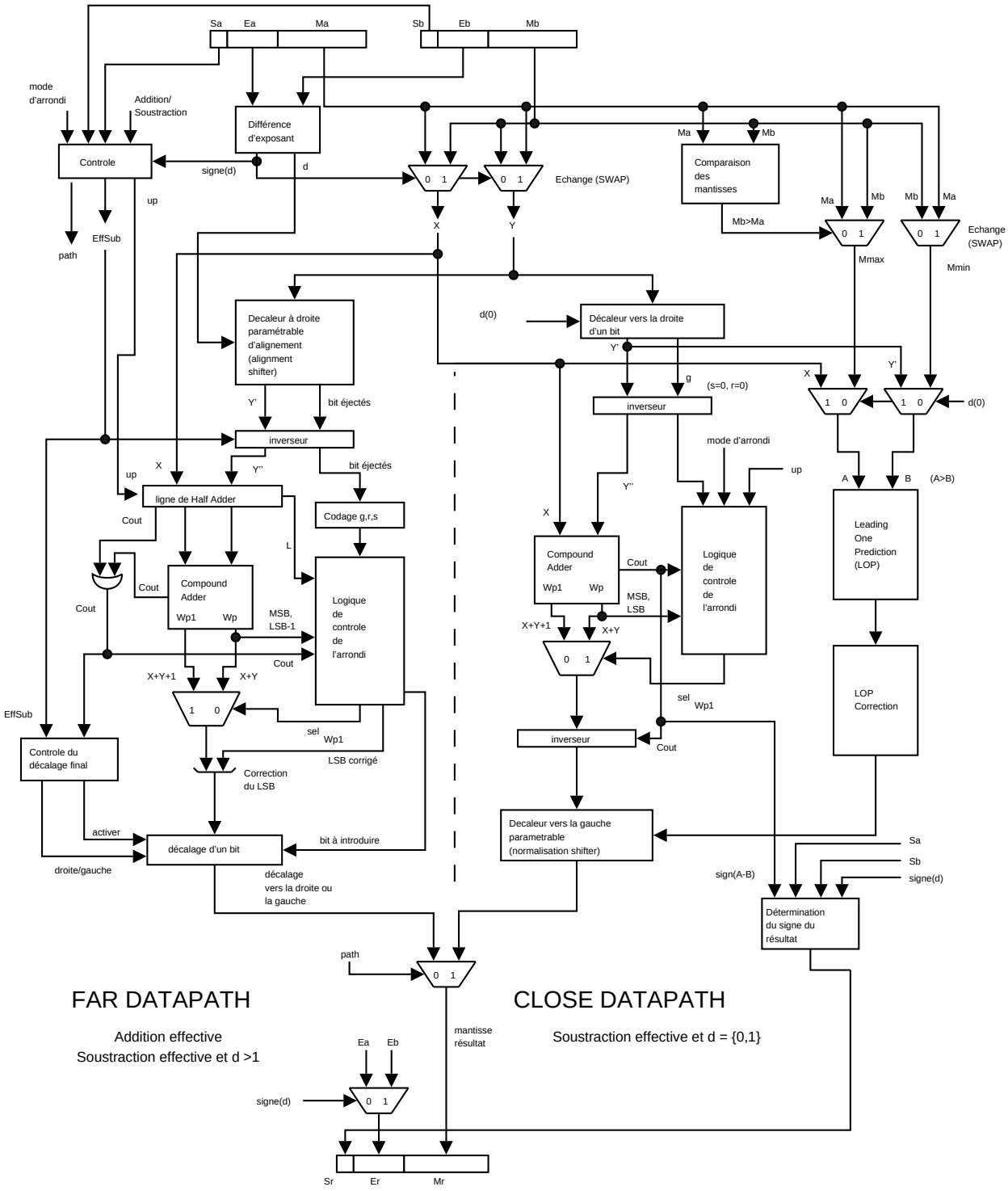


FIG. II.13 – Additionneur en virgule flottante « Double-Datapath »

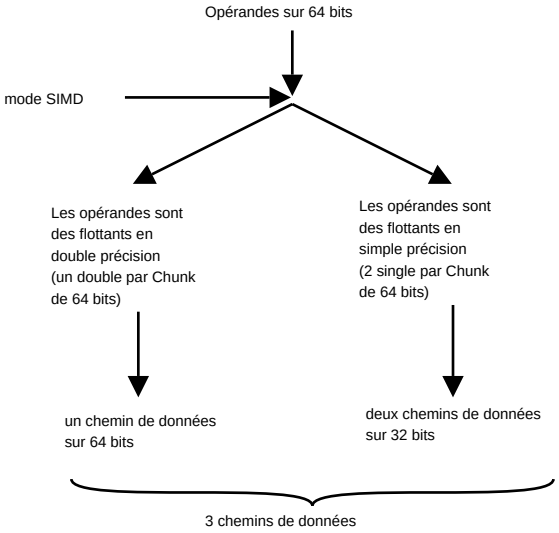


FIG. II.14 – Triple Chemin de données

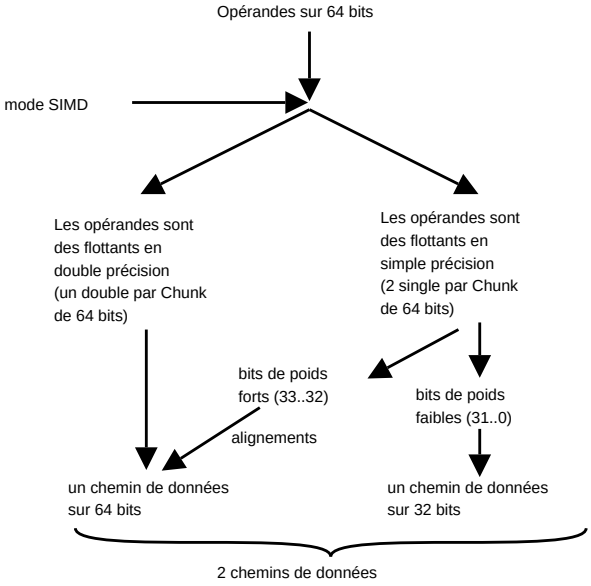


FIG. II.15 – Double Chemin de données

Chapitre III

Résultats et avenir

1 Test de l'unité

Le test d'une unité de calcul matérielle s'avère être extrêmement problématique : il y a une quantité astronomique de cas possibles ce qui rend impossible un test exhaustif. Pour des opérandes sur 64 bits, il y a $2^{128} \approx 10^{38}$ possibilités, c'est plus qu'il ne faut pour occuper un simulateur pendant plusieurs centaines de siècles. A cela doit s'ajouter un autre problème : à quoi comparer le résultat ? Il n'est possible de comparer qu'avec une autre unité existante. Mais a-t-on un « Golden Code », c'est-à-dire un code dont on est sûr de la validité de ses résultats ?. Comment savoir que cette unité n'implémente pas l'algorithme de manière incorrecte ? Voir le bug sur la division des premiers microprocesseurs Intel Pentium. De plus, l'utilisation de nombres aléatoires pour générer un calcul pose aussi un autre problème : l'étendue des tests effectués ne couvrira pas tous les cas possibles (une erreur peut se cacher n'importe où).
La seule méthode qui reste est la méthode systématique : il faut délibérément choisir un jeu de test extrêmement pertinent quitte à oublier un ensemble de cas équivalent.

Chaque unité utilisée (Addition avec le Compound Adder des mantisses, soustraction des exposants, décaleur paramétrable,...) a été testée et validée individuellement en simulation (avec donc Simili).
Mais par manque de temps, le code n'a pas été terminé, donc la dernière version n'a pas été validée entièrement. En fait, seul le premier étage l'est.

2 Résultat de synthèse

Chaque unité validée (Compound Adder, LOP, décaleur paramétrable,...) est synthétisable avec Synopsys. De plus, le premier étage est lui aussi pleinement synthétisable dans la version actuelle.

3 Déroulement du développement

Il y a eu plusieurs versions du code VHDL de l'additionneur. L'environnement Simili permet de faire des simulations tout au long du développement.

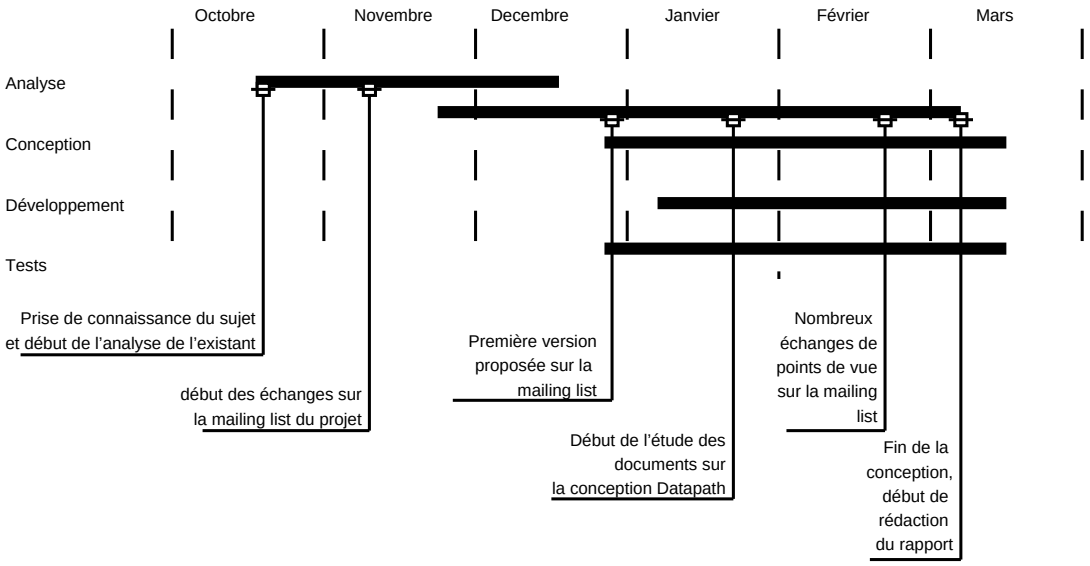


FIG. III.1 – Planification du développement

La première était un additionneur trivial écrit pour bien comprendre les subtilités du codage matériel et surtout des particularités du F-CPU.
Cette version a été mise sur la Mailing List le 30 Décembre 2003, et le mois de Janvier a été consacré à l'étude de toutes les réactions suscitées.
Le coeur du projet est véritablement la Mailing List, où l'on parle librement des idées que l'on aimerait mettre en oeuvre. Les réactions ne se font attendre, et de vifs débats peuvent s'y dérouler. Ces personnes m'ont beaucoup aidé pour la conception matérielle. De plus, même si le manuel du F-CPU est très complet, de nombreuses zones d'ombre subsistent et ne manquent de provoquer quelques échanges constructifs sur cette Mailing List.
Au milieu du mois de Janvier, la découverte des documents sur [4], [2], [6] et [5] sur l'additionneur Double Datapath a véritablement changé la conception de l'additionneur. De simple mais peu efficace, l'additionneur s'est transformé en un additionneur très optimisé. Mais de nombreux problèmes de conception sont venus s'ajouter : difficulté de prendre en compte les délais de chaque transistor[1], la conception assez avancée des unités de prédiction et de logique d'arrondi, puis des problèmes inhérents à la conception VHDL pur. Tout ceci a retardé beaucoup l'avancement du codage de l'additionneur. C'est pour cela que j'ai préféré terminer complètement la conception générale, et coder chaque élément indépendamment du système pour qu'ils soient complètement synthétisables.
Au mois de Février ont eu lieu pas mal d'échange de messages sur la Mailing List concernant la synthèse du code VHDL. Enfin, le début du mois de Mars a été consacré à boucler la conception. Le code en est pour l'instant à sa quatrième version. La première n'étant qu'un additionneur trivial, elle ne serait en fait pas à prendre en compte. La seconde version était une tentative de réalisation d'un additionneur simplifié Double Datapath sans LOP. La troisième version est proche de la dernière en date exceptée qu'elle est découpée en étages. M. Riepe m'a alors conseillé d'effectuer le découpage en étage qu'au dernier moment, ce qui facilite les

[1] ce qui a provoqué de longs messages de M. Riepe pour qu'il m'explique comment il faisait

modifications[2]. En effet, il est arrivé relativement souvent que certains propos des acteurs du projet remettent grandement en cause ma vision de la conception, il a donc fallu refaire entièrement certaines parties car un membre a proposé une idée intéressante.
La dernière version consiste donc en une réécriture de ce code sans étage. C'est cette version qui est disponible dans les annexes, mais seulement le premier étage a pu être réécrit, par manque de temps.

4 Améliorations et conclusion du projet

L'additionneur n'est pas terminé à l'heure actuelle. La conception est terminée pour la première version de l'additionneur, il ne reste que le codage et les tests à effectuer.
Mais pour la prochaine version, il faudra prendre en compte les nombres dénormalisés, ce qui entraînera un certain nombres de traitements spéciaux à effectuer (transformer ces nombres dénormalisés en une représentation interne « compatible » avec les algorithmes développés pour les nombres normalisés), donc rajouter quelques étages qu'emprunteront uniquement les nombres dénormalisés. Ceci transformera l'unité en lui donnant une latence variable.

De plus le comparateur de mantisse pourrait bien servir à supprimer la seconde ligne d'inverseurs dans le « CLOSE Datapath », celle qui n'est activée qu'en cas de dépassement de capacité (Overflow) du Compound Adder (donc en cas de résultat négatif), ce qui ne peut arriver que dans le cas où $d = 0$. Mais pas manque de temps, une étude complète n'a pu être effectuée, c'est une optimisation qui est néanmoins à envisager.

De plus, si la « synthétisabilité » est prouvée, la synthèse en tant que telle n'a pas été effectuée en détail, et notamment, il subsiste certains points d'interrogations quant aux chemins critiques (car il n'a été fait que des estimations a priori). De même, des études approfondies sur la surface occupée, le nombre de transistors, la consommation, etc, pourront être effectués une fois l'unité entièrement écrite.

[2] mais cela ne simplifie pas le debug car Simili ne permet pas d'observer les variables internes à chaque étage (processus), mais seulement des signaux externes

Conclusion

Le projet F-CPU concerne la création d'un utopique microprocesseur, et c'est cela qui en constitue l'intérêt principal. Il a été demandé de concevoir une unité de calcul flottante, sujet qui a été réduit par la suite à l'additionneur. Le but n'est pas de concevoir une unité fonctionnelle pour un usage immédiat, son inclusion n'est pas prévue pour le premier coeur de l'architecture (Core 0), mais pour la prochaine. En effet, il n'y a aucun impératif commercial ou industriel dans un futur proche. C'est pour cela qu'il a été décidé d'effectuer la conception la plus complète possible pour un additionneur le plus optimisé et adapté aux spécifications du F-CPU, quitte à ne pas pouvoir terminer avec une unité entièrement fonctionnelle. Le but est de faire avancer ce projet gigantesque aux moyens très réduits.
Les principales difficultés rencontrées concernent la conception matérielle avancée ainsi que l'étude et la compréhension d'algorithmes et de logiques pointues. Un des principaux freins au développement a été l'imprécision des estimations des délais de chaque unité.
Il reste donc à terminer le code VHDL commencé ainsi qu'écrire un code de validation le plus complet possible. Le présent rapport ainsi que sa bibliographie pourront servir de base pour une éventuelle future version de l'additionneur.
Si la possibilité m'est offerte pendant le stage, je terminerai le code VHDL de l'unité pendant mon temps libre, et je resterai impliqué dans ce projet prometteur.

Bibliographie

[1] F-CPU Design Team, *F-CPU Manual rev. 0.2.7c*
`http://www.f-cpu.org/`

[2] N.T. Quach et M.J. Flynn, *An Improved Algorithm for High-Speed Floating-Point Addition*, Technical Report CSL-TR-9-442
`http://citeseer.nj.nec.com/quach90improved.html`

[3] Santanu Roy, *AN701 - SP Floating Point Math With XA*, Philips Semiconductors Application Notes, 1995 Jul 28
`http://www.semiconductors.philips.com/acrobat/applicationnotes/AN701.pdf`

[4] J.D. Bruguera, T. Lang, *Rounding in Floating Point Addition Using a Compound Adder*, Internal Report, Dept. of Electronic and Computer Engineering, University of Santiago de Compostela, Spain Jul. 2000
`http://citeseer.nj.nec.com/bruguera98rounding.html`

[5] J.D. Bruguera, T. Lang, *Leading-One Prediction With Concurrent Position Correction For Floating Point Addition*, Internal Report, Dept. of Electronic and Computer Engineering, University of Santiago de Compostela, Spain 1998
`http://citeseer.nj.nec.com/bruguera98leadingone.html`

[6] S. F. Oberman, M. J. Flynn, *A Variable Latency Pipelined Floating Point Adder*, Technical Report : CSL-TR-96-689, 1996
`http://citeseer.nj.nec.com/oberman96variable.html`

[7] H. Suzuki, H. Morinaka, H. Makino, Y. Nakase, K. Mashiko, T. Sumi, *Leading-Zero Anticipatory Logic for High-Speed Floating Point Addition*, IEEE Journal of Solid State Circuit, Vol 31, No 8, August 1996
`http://mpu.yonsei.ac.kr/paper_scan/`
`LeadingZeroAnticipatoryLogicForHighSpeedFloatingPointAddition.pdf`

[8] IEEE. 1985, *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std 754-1985, Institute of Electrical and Electronics Engineers New York
`http://citeseer.nj.nec.com/bruguera98leadingone.html`

[9] Prof. W. Kahan, *Lecture Notes on the Status of IEEE Standard 754 for Binary Floating Point Arithmetic*, Lectures notes on the Status of IEEE 754 Oct 1997
`http://www.cs.berkeley.edu/~wkahan/ieee754status/ieee754.ps`

[10] Laurent de Soras, *Denormal numbers in floating point signal processing applications*,
`http://citeseer.nj.nec.com/desoras02denormal.html`

[11] N. Shirazi, A. Walters, P. Athanas, *Quantitative Analysis of Floating Point Arithmetic on FPGA Based Custom Computing Machines*, Virginia Polytechnic

Institute and State University
http://csdl.computer.org/comp/proceedings/fccm/1995/7086/00/70860155abs.htm

[12] A. Beaumont-Smith, N. Burgess, S. Lefrere, C.C. Lim, *Reduced Latency IEEE Floating-Point Standard Adder Architectures*, CHiPTec, Department of Electrical and Electronic Engineering, The University of Adelaide
http://www.stanford.edu/class/ee486/doc/smith1999.pdf

[13] Alain Guyot, *Cours Architecture des Equipements, Opérateurs arithmétiques : Addition En Virgule Flottante*,
http://tima-cmp.imag.fr/~guyot/Cours/Oparithm/francais/Flottan.htm