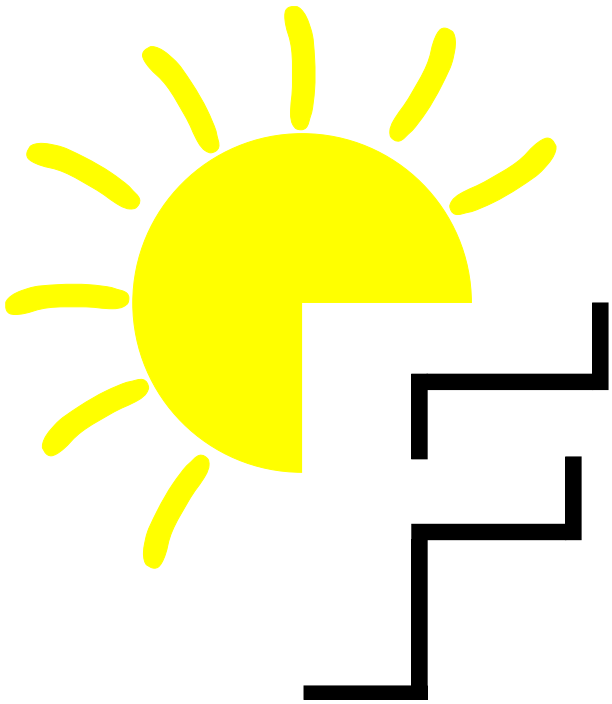
Projet Freedom CPU

Groupe des Concepteurs du F-CPU

Brouillon et Demande de Commentaires

MANUEL F-CPU REV. 0.2.7c



"Design and let design"

Please visit us at `http://www.f-cpu.org` and send comments to the F-CPU mailing list at `f-cpu@seul.org`.

0.1 Copyright and distribution licence :

This manual is distributed under the terms of the GFDL, or "GNU Free Documentation License", which text can be found on the GNU web site (`http://www.gnu.org`). A copy of this licence is included in this package (fdl.htm).

0.2 Foreword :

Although the overall specifications are getting slowly more stable, everything in this document is *furiously preliminary* and changes often without notice. Please keep in touch with the group on the mailing list and check the latest updates on the official F-CPU web site.

This document is (C) 1999-2002 The F-CPU Group Design Team and is a *work of collaboration*. Anybody can participate to the F-CPU effort and become a team member by subscribing to the mailing lists and taking part to the discussions. You are welcome to submit your ideas and report errors. We are conscious that this document always contains errors but we are working on (or around ?) them constantly.

This manual has been translated to several file formats and may additionally lack some parts or contain some errors. It is very incomplete even though it's becoming huge !

0.3 Revision history :

o Créé le 8 Juillet 1999 par Whygee@f-cpu.org (Yann Guidon) á partir d'extraits des RFC de Mathias Brossard.
o 10 Juillet : quelques ajouts.
o 11 : adapté pour la conversion vers PDF avec HTMLDOC.
o 2/8, 8/8, 9/8, 13/8 : ajout supplémentaire.
o 25/8 : retravaillé un peu (pourquoi-comment, TTA, endianness, mémoire paginée, jump station...).
o 5/11 : mélange avec d'autre contenus autres qu'architecture.
o 16/11 : retour au format HTML.
o 27/2 : version majeure pour le codage d'instruction. Imm6 disparait et la plupart des anciennes erreurs et fautes de frappe.
o 15/3 : Adapté pour le macro processing par CPP (au moins)
o 18/12/2000 : Olivier Jean a finalement publié la version Latex du manuel
o 24/12/2000 : YG l'a patché. très incomplet !
o 30/12/2000 : YG l'a repatché à Berlin.
o */01/2001 : début de la traduction française, redessinage des illustrations, mise à jour majeure...
o 02/05/2002 : starting new organisation of the manual (Cedric BAIL, bail_c@epita.fr).
o 04/19/2002 : updating the manual with Michael RIEPE recommandation.
o 04/30/2002 : adding index. (Cedric BAIL, bail_@epita.fr)
o 06/08/2002 : Add the new dshift instruction from Michael RIEPE.
o 07/19/2002 : Applying change suggested by Thomas LAVERGNE.
o 07/20/2002 : Reverse all the bits.
o 07/21/2002 : New loadcons from Michael RIEPE.
o 08/20/2002 : Add color and new convention for ISA.
o 08/27/2002 : Change the color to the recommandation from Matthieu BERNARD.
o 09/08/2002 : Add new shift and rot.
o 09/08/2002 : Remove unnecessary option.
o 09/12/2002 : Add widen.

o 09/14/2002 : Add LSU flush to serialize for stream hints problem.
o 09/14/2002 : Add madd/msub/mschg instructions.
o 09/14/2002 : Change stream hints postfix by adding a dot (reduce gperf colision).
o 09/14/2002 : Add chsift.
o 09/14/2002 : Start SR_MAP.
o 09/16/2002 : Change loadcons to the new form.
o 09/28/2002 : Change loadaddri to the new form.
o 12/10/2002 : Add new memory instruction : ss, sl, cstore, cload.
o 17/10/2002 : Change logic/logici instructions.
o 17/10/2002 : Add new combine_or, combine_and instructions.
o 18/10/2002 : Add new mux instruction.
o 19/10/2002 : Add first revision of TLB entry.
o 20/10/2002 : Update instructions rot, shift, addsub, nabs, dshifti, dbitrevi, dbitrev.
o 27/10/2002 : Update sl, ss, loadaddr.
o 10/11/2002 : Update with "Just an Illusion" recommandation.
o 12/11/2002 : Applied patch from Karl-Heinz Eischer ¡karl-heinz@eischer.net¿.
o 16/11/2002 : Correct push meaning.
o 17/11/2002 : Update dbitrev, dbitrevi, bitrev, bitrevi.

Un tas de commentaires sont aussi donnés par d'autres personnes, quelquefois anonymes, sur la liste de diffusion.

0.4 Manque :

* carte du jeu d'instruction dans l'ordre alphabétique
* table du jeu d'instruction, triée par valeur opcode en hexa.
* plus d'exemples dans les descriptions d'instructions
* IRQ/traps
* parties 8 et 9
* et plein d'autre choses!!!

0.5 Zone de sauts Hyperliens :

HTTP :

* Les principaux sites F-CPU : `http://www.f-cpu.org` et `http://www.f-cpu.de`
* La dernière mise à jour du Manuel F-CPU : `http://www.f-cpu.seul.org`
* The latest CVS-Snapshots at : `http://f-cpu.gaos.org`

Les listes de diffusion :

* `http://www.seul.org/archives/f-cpu/f-cpu` (liste principale)
* `http://lists.april.org/wws/info/f-cpu_france` (liste française)
* `http://www.eGroups.com/list/fcpu-ger` (liste allemande, obsolte)

Mailing lists ar

Table des matières

Table des figures

Première partie

Le Projet F-CPU, description et philosophie

Chapitre 1

Description du projet F-CPU

Il n'y a pas de description exacte du projet F-CPU. Cela n'est pas possible à cause de toutes les discussions des détails et de l'histoire du projet qui ont créé les spécificités de cette entreprise. Nous pouvons néanmoins souligner quelques points et faits importants.

L'architecture F-CPU définit un microprocesseur 64 bits SIMD, superpipeline. À aujourd'hui, c'est le seul CPU de ce type qui peut être complètement paramétré : il n'est pas limité aux implémentations 64 bits et il est prévu pour pouvoir s'étendre et grossir facilement. De plus, c'est le seul processeur de cette classe qui est diponible avec tous les sources (VHDL) et les manuels distribués avec la licence GNU (GPL et GFDL). Le but est de concevoir un processeur qui ne soit pas "encombré" par des brevets et qui puisse s'adapter au plus large choix de technologies possibles.

Le projet F-CPU est aussi formé par beaucoup de personnes dialoguant sur la liste de diffusion des les cotés organisationnels et techniques de la conception. Les listes de diffusion sont des endroits où le processeur est conçu de manière transparente à partir de réflexions contradictoires. Tout le monde peut y participer et influencer les spécifications si les modifications respectent les buts généraux du projet.

Le groupe F-CPU est un des nombreux projets qui tentent de suivre la voie montrée par GNU/Linux qui a prouvé que des produits non commerciaux peuvent surpasser les productions chères et propriétaires. Le groupe F-CPU tente d'appliquer les "recettes du Free Software" au monde de la conception électronique et surtout des ordinateurs en commençant avec le "saint graal" de toute architecture d'ordinateur : le microprocesseur.

Ce projet utopique était seulement un rêve au début mais après la séparation de deux groupes et beaucoup d'efforts, nous sommes parvenus à une assise assez stable pour une architecture developpable et propre sans sacrifier la performance. Nous espérons que la troisième sera la bonne et qu'un prototype sera créé bientôt.

Le projet F-CPU peut être séparé en plusieurs parties (aproximatives et non exhaustives) ou couches qui fournissent la compatibilité et l'interopérabiblité pendant la vie du projet (du Matériel vers le Logiciel) :

* bus, chipset, ponts F-CPU Périphériques et d'Interfaces...
* Implémentations individuelles des circuits Core F-CPU ou révisions (par exemple, F1, F2, F3...)
* Génération ou familles de Coeurs F-CPU (par exemple, FC0, FC1, etc.)
* Jeu d'instruction F-CPU et ressources diponibles pour l'utilisateur
* Interface Binaire F-CPU pour les Applications
* Système d'Exploitation (destiné aux clones Linux)
* Pilotes
* Applications utilisateurs

Toute couche dépend plus ou moins directement des autres. La partie capitale est l'architecture du Jeu d'Instructions car il ne peut pas être changé à volonté et ne fait pas partie du matériel qui peut évoluer lors des changements de ratios technologie/coût. D'un autre coté, le matériel peut fournir une compatibilité binaire mais les contraintes sont moins importantes. C'est la raison pour laquelle les instructions doivent tourner sur un grand panel de micro-architectures de processeurs où les "coeurs CPU" peuvent être changés ou swappés lors des changements de budgets.

Toutes les familles de coeur peuvent être compatibles binairement les unes avec les autres et exécuter les mêmes applications, faire tourner les même systèmes d'exploitations et délivrer les mêmes résultats avec diffŕentes règles d'ordonnancement d'instructions, de registres spéciaux, de prix et de performances différentes. Chaque famille de coeur peut être implémentée avec des "ingrédients" différents comme le

nombre d'instructions exécuté par cycle, des tailles de mémoires, des tailles de mots mais le logiciel doit bénéficier de ces particularités sans (beaucoup) de changements.

Ce document est une base d'étude et de travail pour la définition de l'architecture F-CPU, destiné au prototypage et la commercialisation de la première génération de circuits (nom de code "F1"). Ce document traite des bases techniques et de l'architecture menant à l'état actuel du coeur "FC0". Cela permet de réduire les discussions sur la base dans la liste de diffusion et informe les nouveaux (ou ceux qui reviennent de vacances) sur les concepts les plus récement traités.

Ce manuel décrit la famille F-CPU au travers de son premier coeur et son implémentation. Le coeur FC0 n'est pas exclusif pour le projet F-CPU, qui pourra et devra utiliser d'autres coeurs lors de sa croissance et sa mutation. Le coeur FC0 peut aussi être utilisé pour à peu près toutes les architectures RISC similaires avec quelques adaptations.

Le document évoluera rapidement (nous l'espérons) et incorporera de plus en plus de discussions et de techniques avancées. Ce n'est pas un manuel définitif et il est ouvert à toutes les modifications que la liste de diffusion valide. Il n'est pas non plus exhaustif et peut être très en retard sur l'état du projet, en fonction des fluctuations de temps libre des contributeurs. Vous êtes fortement encouragés de contribuer au débat car personne ne le fera à votre place.

Quelques règles de developpement :

* Ce Projet est une expérience pour démontrer qu'il est possible de développer un processeur par Internet et dans un espace public. Les décisions sont faites par argumentations et consensus sur la liste de diffusion.
* Il n'y a pas de meneur ou de tour d'ivoire (ce n'est pas une "cathédrale"). En fait c'est une "Tour de Cristal" car tout est aussi transparent que possible . Tout le monde peut rejoindre l'équipe et contribuer - ou même contribuer sans officiellement "rejoindre" l'équipe d'une quelconque manière. Même ceux dont la connaissance du développement d'un CPU est limitée ou nulle peuvent contribuer à leur manière (NdT : heureusement sinon vous ne liriez pas cette traduction). Beaucoup de motivation et de temps libre sont ncessaires, nanmoins...
* Le but du jeu est la Liberté ; le processeur est développé de manière ouverte et sera distribué selon les termes de la GNU Public License (GPL), pour que toute personne ait la possibilité (si elle en a au moins les finances) d'utiliser cette création, de la fabriquer et de vendre ses propres F-CPU et ses dérivés pourvu que les modifications restent libres. Lisez la GNU Public Licence et la charte F-CPU pour plus de détails.
* Nous sommes conscients de l'ambition extrême de ce Projet mais nous pensons qu'il est la nécessaire extension du mouvement du Logiciel Libre dans un monde de matériel propriétaire omniprésent. Nous perséverons donc jusqu'à la réussite.
* Nous somme aussi fatigué d'utiliser du matériel propriétaire dont nous ne pouvons pas influencer la plateforme. En tant qu'utilisateurs, nous comprenons que le Logiciel Libre ne peut s'épanouir sans Plateforme réellement Libre.
* Rappelez-vous qu'au Freedom CPU Project nous ne somme ni anti-Intel, ni anti-Microsoft, ni, en fait, anti-quelquechose. Nous ne sommes que pro-liberté !
* Dans le groupe de développement, ne massacrez pas, ne répondez pas à une tentative de massacre mais priviégiez et tenez compte des critiques constructives !
* "Design and let design" (concevez et laissez concevoir) pourrait résumer la plupart des comportements adoptés dans le groupe. Quelques désaccords puissants sont apparus et apparaîtrons pendant les échanges mais que le sujet soit à propos de F-CPU ou non, tout le monde a le droit d'exprimer ses idées. Ne forcez pas l'accord des autres mais dialoguez de manière constructive et explorez le sujet, plutôt que dénigrer les idées des autres. Une bonne architecture peut aboutir d'un respect mutuel, pas de guerres de dénigrements.

Chapitre 2

FAQ

Collecté à partir de différentes sources. Dernière modification effectuée par Whygee, le 14 janvier 2000

2.1 Introduction

Q1 : Qu'est-ce que F-CPU ?

A : F-CPU est essentiellement un microprocesseur SIMD, 64 bits, superpipeline ; disponible avec le code source VHDL'93 et distribué sous les termes de la GNU Public Licence. Il est developpé par une communauté de hobbyistes, étudiants et professionnels sur Internet.

Q2 : Pourquoi un CPU RISC 64 bits ? Je veux faire un clone x86 / une carte son / un RISC 32 bits pour l'embarqué...

A : `http://www.opencollector.org`

L'objectif de concevoir un CPU 64 bits haute performance remonte aux origines du projet lorsque les fondateurs voulaient contrer le Merced (ia64). Si vous souhaitez concevoir autre chose, il y a de grandes chances qu'un projet existe déjà avec un objectif se rapprochant du vôtre. OpenCollector est un des sites web qui répertorie les projets "libres" auquels vous pouvez avoir accès sur Internet. Si vous ne trouvez pas ce que vous voulez, n'hésitez pas à créer votre propre projet.

Il existe déjà beaucoup de projets de CPU libres disponibles sur Internet. Si vous ne désirez qu'un CPU 32 bits, MIPS/DLX et LEON sont des bons points de départ, même si F-CPU peut être facilement adapté à des mots de 32 bits seulement. Si vous désirez un microcontrôleur 16 ou 8 bits, il existe aussi beaucoup de versions libres (distribuées avec des licences variées). Vous n'avez qu'à en choisir un dans la liste du site web OpenCollector. Si vous êtes sûr que vous voulez un processeur tel que F-CPU, veuillez lire le présent manuel avant de vous investir trop vite dans le projet. Les buts généraux du projet sont fixés et ne changeront pas au gré des souhaits personnels.

2.2 Philosophie

Q1 : Que signifie le F dans F-CPU ?

A : Il signifie Freedom (liberté), qui est le premier nom de l'architecture ou Free (libre), dans le sens GNU/GPL.

Le F ne signifie pas qu'il est donné ou gratuit mais qu'il est "librement copiable et modifiable" (NdT : pourvu que l'on respecte la GPL). Vous devrez payer pour le circuit intégré, comme vous payez aujourd'hui pour une distribution GNU/Linux sur CD-ROM. Bien sûr, vous êtes libre de prendre les sources du circuit pour que votre fondeur favori vous fabrique des séries de F-CPU pour votre propre usage, en tant que tel ou intégré dans un autre produit.

Q2 : Pourquoi ne pas l'avoir appelé O-CPU (où O signifie Open) ?

A : Il existe des différences philosophiques fondamentales entre le mouvement Open Source et le mouvement Free Software. Nous aspirons aux orientations de la FSF.

Le fait qu'une partie de code soit labellée Open Source ne signifie pas que la liberté de l'utiliser, de la comprendre et de l'améliorer vous soit garantie. De plus amples détails peuvent être trouvés sur `htpp://www.gnu.org`.

Nous avons tenté de créer une licence similaire à la GPL (GNU Public Licence de la Free Software Foundation) (voir `http://www.opencollector.org/hardlicense/`) mais cet effort a été abandonné car il ne semble ni nécessaire, ni utile. Aujourd'hui, elle est remplacée par une charte externe qui renforce et précise la signification de la GPL dans le cadre du monde de l'électronique industrielle.

D'un manière spécifique, il existe au moins trois niveaux de liberté qui doivent être préservés à tout prix :

- Liberté d'utiliser la Propriété Intellectuelle : aucune restriction ne doit exister pour utiliser le travail du projet F-CPU. Ceci signifie aucun péage pour l'accès au données et TOUTES les informations pour recréer un circuit doivent être fournies.
- Liberté d'analyser, de comprendre et de modifier la Propriété Intelectuelle à volonté. On peut remarquer qu'avec les sources, cela est beaucoup plus facile et rapide que pour un circuit déjà compilé ou synthétisé.
- Liberté de redistribuer les fichiers source.

Les sources ou tous les fichiers du projet ne sont PAS versés dans le domaine public. Les participants au projet F-CPU jouissent des droits d'auteur sur leurs créations et ils choisissent de les rendre librement disponibles à tout le monde par tous les moyens, à condition de respecter certaines règles. Chaque fichier créé à partir des fichiers sources du F-CPU conserve le Copyright du groupe F-CPU. Vous pouvez en lire plus sur `htpp://www.gnu.org`.

Q3 : Comment est protégée le F-CPU ?

A : Le Groupe de Conception F-CPU protège ses travaux avec les lois sur le copyright. Chaque fichier contient la mention du copyright et de la GPL. Rien d'autre n'est nécessaire.

Des mesures additionnelles peuvent assurer qu'aucun dépot de brevet ne sera fait dans le futur. Les brevets sont connus pour leur inefficacité et leurs coûts élevés. Le Groupe de Conception F-CPU est protégé dans le sens où il ne fait que décrire le composant alors que les problèmes apparaissent lorsque le composant est fabriqué et vendu. Nous devons publier des documents pour prouver l'antériorité de nos idées, lors de conférences et dans la presse pour éviter les problèmes restants (et aussi pour nous faire mieux connaître). Le projet ne doit en aucun cas être encombré par des problèmes juridiques.

Q4 : Et que ce passerait-il si je brevetais une fonctionalité du F-CPU ?

A : Vous perdriez simplement du temps et de l'argent.

D'abord, l'architecture générale est basée sur des techniques connues et étudiées parfois depuis trente ans. Vous aurez beaucoup de mal à expliquer ce qui est suffisamment nouveau pour justifier un brevet.

Ensuite, si le brevet est accepté, personne n'acceptera de payer des royalties sur quelque chose qui a été volé au groupe F-CPU. Poursuivre ceux qui vont implémenter ces parties ne mènera à rien puisque le brevet sera contesté et, à la fin, vous possèderez un brevet inutile qui ne vous donne que des problèmes.Vous auriez mieux fait, pendant ce temps-là, de travailler à votre propre circuit.

Q5 : Pourquoi mon entreprise devrait-elle utiliser le F-CPU plutôt qu'un autre CPU ?

A : Les avantages techniques du F-CPU sont décrits dans ce manuel : possibilités d'extension et orthogonalité extrême, conception propre et libre de brevets, accent mis sur la performance et la simplicité, l'implémentation aisée avec différentes technologies (FPGA/ASIC...)

Néanmoins, vous serez certainement encore plus sensible aux cotés non-techniques du projet si vous voulez intégrer un coeur F-CPU dans votre projet. Les fichiers sources peuvent être disponibles sans frais mais cela n'est pas la seule signification de "libre" pour F-CPU. C'est une conception transparente, pas une "boite noire" embrouillée par une équipe propriétaire et fermée. Si vous avez des problèmes avec le F-CPU (par exemple, si la version est obsolète, abandonnée par l'entreprise ou qu'elle a abandonné le produit, en résumé : vous restez seul avec le produit) vous n'avez pas à faire de reverse engineering sur la "boite noire" pour trouver ce qui ne va pas. Vous lisez simplement les sources et les corrigez (ou les patchez si un correctif existe). F-CPU est distribué sous les termes de la GPL qui vous donne le droit de comprendre et de modifier (personnaliser) les fichiers. Vous pouvez en plus participer au projet dans son ensemble, interagir avec les développeurs, soumettre et obtenir des patch preque en temps réel sur Internet.

Un autre aspect concerne les frais légaux. De même que la GPL est parfois appelée un "gentleman agreement", le F-CPU est un "gentleman CPU". Nous encourageons la collaboration pacifique entre les équipes : plus d'argent peut être dédié à la recherche et la conception, moins d'argent pour les avocats. A la fin, tout le monde gagne puisque le groupe des utilisateurs/développeurs F-CPU devient plus important et passe tout son temps à améliorer la qualité des sources, ce qui accelère la mise sur le marché de nouveaux produits. "Design and let design" : les seules choses intéressantes et déterminantes sont les temps de réaction et l'efficacité (coût, performance, facilité d'utilisation) du produit et des équipes.

Q6 : Cool mais où est le truc ? Quels sont les inconvénients ?

A : Ils sont bien connus et se trouvent dans la GPL et dans la charte F-CPU. De même que les sources sont disponibles librement, vous devez les maintenir libres et redistribuer toutes les modifications et additions faites au coeur. F-CPU (comme tous les projets GPL) étant basé sur la collaboration/coopération et non la compétition, vos mises à jour vont bénéficier aux autres mais ils peuvent toujours les améliorer et vous en bénéficierez en retour.

Si vous devez garder vos travaux complétement secrets, n'intégrez pas le F-CPU dans votre projet pour le modifier. Vous ne pourrez pas bénéficier du travail et de l'expérience des autres. Vous aurez à réinventer la roue et perdrez du temps et de l'argent.

2.3 Outils

Q1 : Quel outil EDA allez-vous utiliser ?

A : Il y a déjà eu beaucoup de débats sur ce sujet. C'est principalement une guerre entre Verilog et VHDL. Nous avons démarré avec VHDL'93 par commodité car c'est le plus utilisé en Europe (où la plupart du code est écrit) mais les fichiers seront certainement traduits en d'autres formats. Actuellement, les sources n'existent en VHDL que par commodité et uniformité. Les autres représentations en seront dérivées.

Maintenant que VHDL est le langage majeur, le choix des outils logiciels est plus limité. Nous voulons promouvoir des logiciels GNU mais cette branche n'est pas encore suffisamment developpée ou mature actuellement. Un logiciel spécifique peut être difficile à installer, un autre peut être instable, trop vieux ou incompatible avec les standards et besoins actuels.

L'utilisation d'Alliance (`http://www-asim.lip6.fr/alliance/`) est envisagé mais il ne sera utile que pendant le processus du layout pour une version "full custom". Les autres outils de conception libre peuvent être trouvés sur `http://www.opencollector.org`.

En ce moment, nous utilisons Simili (`http://www.symphonyeda.com`) sur la plateforme Win32. Ce n'est pas un logiciel GNU mais il a beaucoup d'avantages que l'on ne retrouve nulle part actuellement, comme l'indépendance par rapport aux constructeurs, le respect presque total des standards IEEE, la facilité d'utilisation, la compacité... Nous espérons un portage Unix dans le futur, de même que d'autres bons logiciels EDA GNU.

Les sources ont aussi été compilées sans modifications avec FreeHDL et Modelsim. D'autres compilateurs compatibles IEEE vont certainement confirmer la haute portabilité et la qualité des sources.

Cadence a proposé des licences gratuites pour quelque uns de ses outils. D'autres offres vont probablement suivre et sont les bienvenues tant que les contreparties sont compatibles avec la charte F-CPU.

Nous allons probablement utiliser des outils commerciaux à un moment ou à un autre car les fondeurs utilisent des logiciels propriétaires mais dans tous les cas, un crayon, une feuille de papier et un cerveau sont les ingrédients les plus importants pour concevoir un circuit.

2.4 Architecture

Q1 : Quelle est cette architecture mémoire-à-mémoire dont j'ai entendu parler ? Ou ce truc TTA ? Pourquoi pas une architecture registre-à-registre comme tous les autres processeurs RISC ?

A : *M2M* était une idée défendue au début du projet F-CPU. Elle avait quelques avantages sur l'architecture registre-à-registre, comme un délai de commutation de contexte très bas (pas de registre à sauver et restaurer). Aujourd'hui, le mécanisme SRB inclu dans FC0 résoud ces problèmes (voir Partie IV, chapitre 3, "Le mécanisme de Smooth Register Backup").

TTA est une autre architecture qui a été explorée avant que la conception actuelle de (FC0) ne soit démarrée.

L'architecture de F-CPU pourra évoluer dans le futur et emprunter quelques nouvelles fonctionnalités à d'autres architectures.

Q2 : Envisagez-vous une FPU externe ?

A : Non. la bande passante et le nombre de broches pose problème. Nous pouvons actuellement intégrer de telles unités dans une puce.

Q3 : Pourquoi ne supportez-vous pas le SMP ?

A : Le Symmetric Multi-Processing tel qu'il est implémenté dans les PCs limite la performance et les possibilités d'extension de l'architecture. Nous cherchons activement d'autres architectures, principalement NUMA (Non-Uniform Memory Access) au travers d'un bus spécifique appelé F-BUS. Nous tentons d'éviter toutes les techniques complexes qui peuvent apparaitre dans un système multi-CPU. Aucune décision ferme n'a été prise pour l'instant. Le coeur F-CPU est de toute manière indépendant de l'interface, tout type de connexion pouvant être implémentée.

2.5 Performance

Q1 : Que pouvons-nous espérer, en termes de performances, du F-CPU ?

A : Merced-killer. :-). Plus sérieusement, nous espérons avoir de bonnes performances, bien qu'il soit impossible de faire une annonce avant d'avoir fait des mesures sur le circuit réel : ce serait une marque d'amateurisme et les performances dépendent largement des technologies disponibles, du budget, des contraintes et des besoins du fabricant.

Nous pensons pouvoir obtenir des bonnes performances car nous repartons de zéro avec une approche fraîche. x86 est relativement lent car il doit être compatible avec les anciens modèles. Les familles ARM, MIPS et SPARC vont avoir bientôt 20 ans, Power et Alpha/AXP approchent les 10 ans et nous pouvons tirer des leçons de leur évolution.

LINUX et GCC par eux-mêmes ne sont pas les meilleures garanties de performance. Par exemple, GCC ne traite pas les données SIMD. Nous allons certainement créer un compilateur qui est plus adapté au F-CPU et GCC sera utilisé au début comme "bootstrap" pour les logiciels existants. Le travail à venir sur les interfaces GNL et XML va probablement permettre aux développeurs de créer un meilleur code que GCC ne pourrait jamais le faire.

Objectivement, la famille de coeur FC0 est destinée à réaliser le meilleur ratio MOPS/MIPS possible. Le superpipeline garantit que la meilleur fréquence d'horloge est atteinte quelle que soit la technologie du circuit. La bande passante mémoire peut être virtuellement augmentée avec des stratégies d'"indices" explicites. Nous pouvons donc prévoir qu'un circuit à 100MHz avec 1 instruction décodée à chaque cycle peut facilement effectuer 100 millions d'opérations à la seconde. Ce qui n'est pas si mal du tout car vous pouvez l'obtenir avec une "ancienne" technologie silicium (bon marché) qui ne pourrait pas atteindre 100MOPS avec une architecture x86 pour le même prix.

Ajoutez à ceci une largeur de donnée SIMD libérée de toute contrainte et vous avez une idée de la performance crête qu'il peut atteindre. Si vous voulez des chiffres parlants, avec la version 64 bits, les opérations SIMD sur des octets donnent 8 opérations par cycle, ou 800MOPS en pointe.

2.6 Compatibilité

Q1 : F-CPU sera-t-il compatible avec les x86 ?

A : No. Ne. Nada. Niet. Nein. Non.

Il N'y aura PAS de compatibilité binaire entre F-CPU et les processeurs x86. Il pourra néanmoins faire tourner des émulateurs Windows qui incluent des émulateurs de CPU x86 comme Twin, de même que Windows lui-même sous tous les émulateur PC comme Bochs. Dans tous les cas, néanmoins, vous aurez besoin de faire tourner un autre système d'exploitation, comme GNU/Linux et l'émulation sera assez lente. Mais quel est l'intérêt d'utiliser Windblows alors que vous pouvez lancer GNU-Linux/xBSD à la place ? ;-D

Q2 : Aurais-je la possibilité de connecter un F-CPU dans un Socket 7, Super 7, Slot 1, Slot 2, Slot A standard ou sur toute autre carte mère existante ?

A : Il y a de grandes chances qu'aucune version du F-CPU ne soit jamais disponible pour le Socket7 ou toute carte mère x86.

Raison 1 : le BIOS doit être réécrit, les chipsets doivent être analysés et il existe trop de combinaisons chipsets/cartes mères. Cela est en dehors du champ du projet.

Raison 2 : Socket/broches/bande passante : les circuits x86 sont réellement "memory-bound", la bande passante avec la mémoire est trop basse, de nombreuses broches ne sont pas utiles pour des circuits non-x86 et supporter toutes les fonctions de l'interface x86 rendra le circuit (sa conception et son débuggage) trop complexe, plus cher et plus lent.

Raison 3 : nous ne voulons pas payer de licence pour l'utilisation de slots propriétaires.

Les slots ALPHA ou MIPS seront peut-être supportés. Nous pourrons inclure une interface EV-4 au F-CPU car de nombreux "chipsets" sont déjà disponibles pour le marché des cartes embarquées. Enfin, une interface personnalisé évitera tout problème de compatibilité et d'incompréhension. Si vous voulez connecter ou interfacer votre F-CPU sur quelque chose d'autre, "just do it".

Q3 : F-CPU supportera quel noyau d'OS ?

A : Linux sera sûrement supporté en premier. Les autres portages suivrons et différents types de noyaux sont possibles. Mais avant cela, nous devons avoir un outil de développement de logiciel fonctionnel pour l'architecture. Nous devons donc définir complètement F-CPU en premier... Le kernel n'en est qu'au stade des discussions.

Q4 : Quels programmes pourrais-je faire tourner sur F-CPU ?

A : Nous avons un portage prototype/préliminaire de gcc/egcs pour l'architecture Freedom. Théoriquement, le F-CPU pourra exécuter tous les logiciels disponibles pour une distribution standard GNU/Linux, mis à part les parties bas niveau tels que I/O, code bootstrap ou écrit en assembleur.

Rappelez-vous que GCC n'est pas parfaitement adapté aux processeurs de cinquième génération (et plus). Nous l'avons adapté pour F-CPU mais c'était très difficile et il n'utilise qu'une petite partie des possibilités du jeu d'instruction et des ressources du F-CPU. N'attendez pas de performances extraordinaires d'un code généré ainsi, au moins pour le FC0.

2.7 Coût/Prix/Achat

Q1 : Aurais-je la possibilité d'acheter un F-CPU un jour ?

A : Nous l'espérons. C'est l'objectif du projet mais soyez patients et prenez part aux discussions ! Si vous pensez qu'il n'est pas développé suffisamment rapidement, rejoignez le groupe et aidez nous. Avant que F-CPU n'existe en tant que circuit, il sera disponible sous d'autres formes telles que des émulations logicielles ou matérielles ou des simulations.

Q2 : Combien coûtera le F-CPU ?

A : Nous ne savons pas. Cela dépend du nombre d'unités et de nombreux autresfacteurs.

Il y a eu une estimation préliminaire optimiste qui donne aproximativement $100 par unité pour un lot de 10000 circuits. Cela dépend aussi de beaucoup d'autres facteurs comme les performances souhaitées, la taille de la mémoire cache, le nombre de broches et surtout la possibilité de combiner tous ces facteurs dans les technologies disponibles. Les dernières estimations pour une première version limitée étaient autour de $60 pièce pour un lot de 1000 ASIC. Le circuit FC0 ressemble à un 486 plus gros et simplifié. Il appartient à la classe des circuits à 1 million de transistors. C'est plus que le coeur LEON ou ARM mais c'est peu comparé aux autres circuits 64 bits haut de gamme. Il sera donc moins cher que ceux-ci.

Chapitre 3

La genèse du projet F-CPU

Ce chapitre provient essentiellement de la première période de F-CPU dont les auteurs sont mentionnés plus loin. Beaucoup de choses ont changé depuis que ce document a été écrit. La motivation n'a pourtant pas changé et la méthode est toujours la même. Les auteurs originaux sont maintenant injoignables mais nous avons continué de travailler de plus en plus sérieusement sur le projet. Au moment de l'écriture, plusieurs questions posées dans le texte suivant ont reçu une réponse mais maintenant que le groupe s'est lui-même structuré, les autres questions deviennent plus importantes car nous devons réellement y faire face : ce n'est plus une utopie, la fiction devient réalité.

N'oubliez pas que les éléments techniques qui sont décrit ici NE sont PAS réalistes et ne correspondent à rien de réel. C'est plus un rêve qu'une analyse cohérente. Please ne nous descendez pas pour les rêves des autres.

3.1 L'Architecture CPU Libre : Un microprocesseur 64 bits GNU/GPL à haute performance développé dans un environnement ouvert et collaboratif au travers du Web.

Auteurs : Andrew D. Balsa w/ plusieurs contributions de Rafael Reilova et Richard Gooch.

5 Août 1998

3.1.1 Histoire

L'idée d'une conception de CPU GNU/GPL se trouve au milieu de quelques échanges de couriels entre trois utilisateurs de longue date de GNU/Linux (aussi développeurs du noyau Linux à leurs moments perdus) avec diverse taches de fond.

Nous nous posions des questions sur les monopoles et comment la domination d'un système d'exploitation (incluant le noyau, l'Interface Graphique Utilisateur et la disponibilité de "killer-applications", de même que la documentation) était intimement reliée à la domination mondiale d'une architecture CPU spécifique, inefficace, dépassée et maladroite. Je suppose que vous devinez tous à qui je fais allusion.

Nous avons aussi exprimé notre croyance dans le fait que GNU/Linux est le mieux placé pour fournir les fondations de base pour un environnement logiciel totalement Libre (dans le sens GNU/GPL ; please cherchez une copie de licence GNU GPL si vous lisez ceci ou allez sur www.gnu.org). Néanmoins, cette Liberté est limitée ou plutôt limité par le matériel propriétaire qui tourne dans beaucoup de maisons : le traditionnel PC basé sur le x86.

Finallement, nous étions inquiets de l'attitude de Intel qui ne fournit pas d'informations préliminaires à la communauté Free Software à propos de l'architecture du Merced à venir. Ceci a pour conséquence le retard du développement du compilateur gcc compatible, d'une version personnalisée du noyau Linux et finallement au vaste univers des outils Free Software. Il existe une vague rumeur que Linus Torvalds ait reçu des informations avancées sur Merced en signant un NDA Intel mais cela reste une exception individuelle et cela ne correspond pas à l'esprit du Logiciel Libre. Avec du recul, si Merced sera sûrement plus moderne que les architectures x86, il sera un pas en arrière en termes de Liberté car contrairement aux x86, il n'y aura sûrement pas de clone Merced.

Ces précédents jours, nous avons discuté sur les différents modèles de développement Free Software, leurs avantages et inconvénients. En rassemblant ces deux discussions ensemble, j'ai rapidement fait un brouillon d'une idée et l'ai posté à Rafael et Richard, les prévenant que cela serait bon à lire pendant

qu'ils compilaient XFree86 ou un gros package... et ils ont aimé ! Ici, vous trouvez cette idée utopique, folle, mélangée avec des commentaires, des critiques et d'autres idées de Rafael et Richard :

3.1.2 L'architecture GNU/GPL Libre

Nous avons commencé avec quelques questions :

- Pourquoi ne pas développer un CPU 64 bits et mettre la conception sous GNU General Public License ?
- Pourquoi ne pas rendre le processus de développement de ce nouveau CPU complètement ouvert et transparent, de manière à ce queque les meilleurs cerveaux du monde puissent contribuer avec les meilleures idées (d'une manière ou d'une autre, en utilisant les mêmes mécanismes de communication traditionnellements utilisés par la communauté Free Software) ?
- Comment rendre le processus de développement du CPU entièrement démocratique et vraiment ouvert, là où il est habituellement entouré de paranoia et de secrets ?
- Comment pouvons nous concevoir quelque chose qui améliorera les *bases fondamentales de la technique* de ce qui sera disponible en 2000 avec le groupe à l'architecture la plus avancée jamais rassemblée par aucune industrie (le Merced) ?

Ici, on a deux incroyables challenges distincts :

a) la performance et la faisabilité de l'architecture résultante et

b) le processus de développement ouvert sous licence GNU/GPL et les questions de droits de propriété intellectuelle soulevés par ce procédé.

((((((((((Matériel a) en premier)))))))))) (performance et faisabilité), nous pensons que l'architecture libre peut être plus efficace sous GNU/Linux comparé aux autres architectures en le rendant :

1. Plus compatible avec le compilateur gcc. Nous avons le code source de gcc mais, plus important, les développeurs de gcc sont disponibles pour nous aider à trouver quels éléments ils souhaiteraient voir dans une architecture CPU. Pourquoi gcc ? Car c'est la pierre d'angle de tout le Logiciel Libre. Simplement, une architecture efficace pour gcc verra une augmentation de l'efficacité ((((((((((across-the-board)))))))))) pour *tous* les programmes Logiciels libre.
2. Plus rapide avec le noyau Linux. ((((((((((Right now)))))))))), si nous prenons pour exemple l'architecture PC, nous avons noté que le noyau Linux a besoin de "work around" (et certains diraient "travailler contre") ((((((((((various idiosyncrasies)))))))))) sur les spécifications du x86 et du matériel. Nous devons aussi préserver la compatibilité avec les circuits x86 dépassés. Et, bien évidement, il n'y a pas de possibilité d'implémenter quelques unes des fonctions du noyau les plus usitées dans le silicium. Une nouvelle conception personnalisée pour le noyau Linux accroitra énormément les performances de toutes les applications limitées par le noyau.

D'autres idées pour des architectures et des implémentations possibles peuvent être trouvées dans les annexes (de même que le coté "économique" du projet). Notez que nous appelons les architectures "Freedom" (pour des raisons évidentes), et sa première implémentation "F1". Le coût prévisionnel pour un utilisateur final d'un F1 est autour des $100. Nous savons que tout est encore très utopique. : -)

Néanmoins, il nous semble qu'a ce point, les challenges réels à ce stade de notre projet sont entièrement dans b) : le processus de développement et les versions des propriétés intellectuelles.

3.1.3 Développer l'architecture Libre : versions et challenges

Le dessin Dilbert l'a résumé : en fait, notre projet *est* un paradigme en son entier ! Ce qui nous proposons, en fait, est de rassembler les compétences et la puissance de création de milliers d'individus avec le web pour le processus de conception d'une architecture CPU GNU/GPL 64 bits avancée et Libre. Et nous ne savons même pas si c'est possible ! Nous somme simplement sûrs de deux choses :

- Dans le passé et le présent, les entreprises comme Intel, IBM et Motorola ont été connues pour avoir cassé des équipes de conception, de telle manière qu'aucun groupe (((((((((close))))))))) ne puisse se former en étant capable de recréer entièrement les sources (et même quitter la firme et former leur propre compagnie). Récemment, Andy Grove a donné une nouvelle signification au mot "paranoia" comme outil de management. La proposition de notre environnement collaboratif, Libre, ouvert et transparent va à l'encontre de cette tendance. C'est aussi relié pour une grande partie à des tendances nouvelles dans les théories sur le management des Ressources Humaines et d'Organisation. En fait, c'est très collé au concept des Corporations Virtuelles, excepté que dans ce cas, nous parlons plutôt d'une Organisation Virtuelle à but non lucratif. De ce point de vue, le projet Freedom est aussi une expérimentation dans la théorie de l'Organisation mais cela n'est pas une expérience gratuite. Plusieurs études montrent que garder des personnes dans de petits groupes fermés, limités par des NDAs stricts et autres contraintes légales menant au silence en public et

imposer un haut niveau de pression sur ces groupes n'est pas le meilleur moyen pour libérer leur pouvoir créatif. Cela peut aussi mener à des conceptions erronées...

- Le développement du noyau Linux, par un groupe de programmeurs/développeurs système hautement talentueux, est un exemple : un environnement ouvert, collaboratif, destiné à un logiciel GNU/GPL avec un contenu à haute valeur intellectuelle/technologique peut être viable. De plus, il peut être démontré que dans certains domaines, le noyau Linux est plus performant que sa contre partie commerciale. Néanmoins, cette liste de certitudes est plutôt courte comparée à la liste des questions générées par nos propositions :
  - Comment allons-nous écarter ou sélectionner les nouvelles idées pour les inclure dans la conception, parmi le "bruit" inévitable des Mauvaises Idées (tm) ? Qui sera le juge de ce qui sera bon ou pas ?
  - Aussi, inévitablement, des options/fonctions mutuellement exclusives apparaitrons au cours du développement. encore une fois, qui décidera de la direction à suivre ?
  - Qui possèdera les droits finaux de la propriété intellectuelle ? Le "copyleft" est-il applicable dans le cas de la conception de CPU ? Qu'en est-il des masques pour les premier circuits ?
  - La GPL sera-t-elle suffisante comme instrument légal pour protéger les sources ? Quels changements, s'il y en a, doivent être faits au GNU/GPL pour l'adapter à la conception de circuits ?
  - Si le processus de conception utilise des EDA commerciaux et d'autres outils, dans quelle mesure ces systèmes propriétaires "entachent" les sources GNU/GPL ? Est-il possible de séparer la partie GPL de celle commerciale/propriétaire ?
  - Qu'en est-il des brevets existants ? Le projet en aura-t-il besoin ? Aura-t-il la possibilité d'en "acheter" ou de payer des royalties ?
  - Contrairement à un logiciel, des implémentations partielles des sources Freedom ne seront pas possibles. La première implémentation dans le silicium *doit* être fonctionelle et complète. Tous les "trous" dans la conception doivent être cablés avant que le premier masque ne soit dessiné. Comment faire pour que les volontaires acceptent un planning aussi rigide ?

Il existe aussi quelques questions qui surviennent comme conséquence du possible succès de l'implémentation Freedom :

- Il y a de vastes possibilités pour des sources de CPU GNU/GPL dans l'industrie, le médical, l'aéronautique, l'automobile et autres domaines. En fait, une conception Libre, stable, haute performance offre des possibilités jamais encore imaginées par les concepteurs de matériel dans des domaines variés. Est-ce le début d'une petite révolution par exemple, dans le matériel embarqué ?
- La conception va-elle se maintenir elle-même pendant des années comme le processeur idéal GNU/Linux ?
- Cette expérimentation dans le développement ouvert aura-t-elle d'autres conséquences sur l'industrie électronique ? Sommes-nous en train de proposer un nouveau paradigme pour le développement de CPU ? Ce paradigme pourra-t-il être appliqué aux autres conceptions VLSI ?

3.1.4 Outils

Nous connaissons tous le proverbe : ((((((((((”If the only tool one has is a hammer...”)))))))))). Nous aurons besoin d'outils "groupware" pour le projet Freedom mais le terme "groupware" a une mauvaise réputation de nos jours. Nous préférons utiliser des "outils de travail collaboratifs". Certains d'entre eux n'existent et ne sont largement utilisés que depuis la dernière décénie ; je suis bien évidement en train de parler du web et son assortiment de technologies de communication : email, newsgroups, listes de diffusion, sites Web, documentation SGML/PDF/HTML et logiciels d'édition/traduction. Soit dit en passant, une grande partie de cette infrastructure a été utilisée pour développer GNU/Linux et est de nos jours basé sur GNU/Linux.

Mais nous avons aussi besoin de nouveaux outils qui n'existent probablement pas encore. Je pense qu'il faut mentionner que le premier pas dans cette direction est peut être le projet WELD, développé à Berkeley. Il pourrait très bien devenir la pierre d'angle du projet Freedom ou inversement, le projet Freedom pourrait être considéré comme le cas idéal de test pour le projet WELD.

3.1.5 Conclusion

La conclusion est simple et évidente :

- si vous êtes un ingénieur VLSI ou en architecture CPU, ou
- si vous avez une bonne idée sur la conception de CPU avec laquelle vous vous êtes déjà amusé pendant quelque temps et que vous voudriez tester, ou
- si vous aimez simplement la stimulation provenant de propositions intellectuelles et d'interactions de reflexions :

Please rejoignez nous et aidez nous à transformer cette idée en réalité !

-

* : Richard est un astrophysicien Australien préparant son Ph.D. sur la visualisation astronomique ; Rafael est un chercheur sur outils EDA à l'Université de Cincinatti. Je suis un étudiant ex-Ph.D. en Management et un ingénieur ex-firmware, avec un intérêt particulier pour les problèmes éthiques dans les environnements multi-culturels (je suis né au Brésil et je vis actuellement en France). Aucun de nous n'a de formation spécifique en architecture CPU. Rafael s'en rapproche le plus, étant un concepteur VLSI et développeur d'outils EDA et il a aussi développé de nouveaux programmes pour la reconnaissance de CPU dans le noyau Linux. Richard a développé le portage du Pentium Pro MTRR dans les noyaux Linux 2.1.x (de même que de nouvelles routines de noyau), et c'est aussi un développeur de matériel. J'ai l'honneur d'avoir diagnostiqué le bug de "virgule" sur le Cyrix 6x86 et lui ait proposé une solution sous GNU/Linux (les deux étaient d'abord rejetés par Cyrix Corp.). Je suis aussi depuis longtemps un développeur de matériel et de firmware et j'ai contribué de différentes manières au développement de GNU/Linux (e.g. le HOWTO Linux Benchmarking).

Richard E. Gooch <Richard.Gooch@atnf.csiro.au>

Rafael R. Reilova <rreilova@ececs.uc.edu>

Andrew D. Balsa <andrebalsa@altern.org>
note : aujourd'hui aucune de ces adresses ne fonctionne. altern.org a même disparu.

3.1.6 Annexe A

Idées pour la conception d'un processeur GPL 64 bits haute performance
C'est juste un rêve, une idée utopique de la conception d'un processeur libre. C'est aussi une liste des éléments que je souhaiterais avoir dans un futur processeur.

- Ce projet aurait besoin d'un sponsor si nous voulons qu'il devienne une réalité. Obtenir les premiers circuits ne permet ni de devenir libre, ni ne sera facile.
- Le choix d'un bus de donnée 64 bits pour l'espace d'adressage : c'est devenu évident et cela simplifie tout.
- Le jeu d'instructions codé d'Huffman : améliore la bande passante cache/mémoire→CPU, qui est un des principaux bottlenecks de nos jours. Il doit être simple d'ajouter un codeur Huffman à un compilateur ((((((((((back-end)))))))))). Toutes les longueurs d'instructions sont des multiples d'octets.
- Le débat RISC vs. CISC vs. flux de donnée est terminé ! Prenez les avantages de chaque sans leurs inconvénients si possible.
- 1, 2 ou 4 pipelines 7-stages internes.
- Exécution spéculative : 4 branches, 8 profondeurs d'instruction pour chaque.
- Queue de pré-traitement d'instructions 64 octets.
- Buffer d'écriture 32 octets.
- Microprogramme partiellement en RAM. Nous devons être capable d'émuler le jeu d'instructions x86 (au niveau du code source assembleur).
- capacité d'interruption multiple 64 bits TSC w/.
- Système d'économie d'énergie.
- Émulation MMX et 3DNow !.
- Conception entièrement statique (possibilité d'arrêter l'horloge).
- Implémentation F1 : bus de donnée externe 128 bits, capacité d'adressage externe de 40 bits.
- Registres de contrôle de performance à la Pentium.
- FPU externe, mémoire paginée (je n'ai pas idée à quoi cela peut ressembler). Les FPUs peuvent être additionnés pour le travail en parallèle (plus de 4 ?). Bus séparé. Le même bus peut traiter un coprocesseur graphique avec sa mémoire double.
- Cache L1 8KB 4-ports unifié, avec possibilités d'indépendance du line-locking/line-flushing. ((((((((((Can be thought of as a 1 KB register set)))))))))).
- Caches L2 d'instructions et de données séparés de 64KB chaque, tournant à la vitesse du CPU.
- Contrôleur DMA intégré intelligent, 32 canaux.
- Contrôleur intégré d'interruption : 30 interruptions masquables, 1 interruption de la Gestion Système, 1 interruption non-masquable.
- Aucun registre interne ! Oui, c'est une machine mémoire-mémoire. Le jeu d'instruction reconnaît 32 pseudo-registres à tout moment.
- Les interruptions commutent automatiquement le jeu de registre vers un jeu de registre vectorisé : pas de temps d'attente de commutation de contexte !

- Pas de pénalité pour les instructions qui accèdent des octets, des mots ou des mots doubles.
- Opérations dans le mode petit ou grand endian à la MIPS.
- Pagination à l'Intel, avec des pages de 4k + 4M d'extension.
- Aussi : VSPM à la Cyrix 6x86, avec des pages définissables de 1K.
- Registres ARR à la Cyrix 6x86 (similaire au MTRR sur Intel PPro) : permet la définition de zones non cachables (utile pour NUMA, voir ci-dessous).
- PLL interne avec multiplicateur programmable par logiciel ; peut commuter de 1x à 2x puis 3x puis nx avec des incréments de 0.5, à la volée.
- Le MMU doit aussi supporter la protection d'objet à la Apple Newton.
- ((((((((((((((Single-bit ECC throughout)))))))))))))).
- Support direct de portage de régions de mémoires double ((((((((((4 1MB)))))))))) pour le multi-processing de type NUMA (aussi sur le bus FPU).
- Nom de projet de l'architecture CPU : "Freedom". Peut aussi être appelé "Merced-killer" ou "Anti-Merced" ou " !Merced" mais en fait nous ne sommes contre personne dans ce projet. Nous sommes simplement pro-liberté et ouverts ; ce que nous détestons sur le Merced d'Intel, c'est sa conception propriétaire et son environnement de développement resteint. Ici, je suppose que le challenge est de déterminer comment la conception d'un CPU GPL est faisable. Est-ce qu'un développement collaboratif et ouvert d'un CPU WRT est possible ? Comment fait-on avec le fondeur pour réellement mettre les sources sur le silicium, une fois que tout est prêt ? Comment traite-t-on les révisions ? Existe-t-il des brevets qui vont bloquer un tel processus de développement ?

Aussi, l'idée est d'utiliser gcc comme compilateur idéal de développement pour ce type de CPU (contrairement au Merced). Et pour être à même de porter le noyau Linux avec un effort minimum sur ce nouveau processeur.

3.1.7 Annexe B

surface de la puce / coût / caractéristiques physiques du boitier / bus externe pour le Freedom-F1

Simplement pour rappel, le CPU F1 n'inclue pas de FPU ou d'unité 3DNow ! (mais des instructions entières SIMD seront incluses).

Taille maximum recommandée : 122 sq. mm. Ceci nous donne 200 wafer dies/8-inch (voir un exemple d'un tel wafer sur Hennessy et Patterson, page 11).

Grosso modo, die yield = 0.5 pour notre 122 mm2 5-couches 0.25 micron CPU (H AND P, page 13, mis à jour pour refléter de meilleures fabrications). Ceci permet plus ou moins 10-11 millions de transistors, divisé comme suit : 6-7 millions pour les caches, 4-5 millions pour le reste.

Supposons un wafer avec un yield = 95, et au test final : yield = 95. Coûts de tests de $500/heure, 20 secondes/CPU.

Coût du boitier = $25-50 (voir ci-dessous).

Grosso modo, suivant H et P, ceci nous donne un coût unitaire de $75-100/bon CPU, testé, conditionné dans des supports anti-statiques et transporté vers les US, si les fondeurs Taiwannais peuvent garder le processus de wafer autour des $3.500.

Boitier : Je vais proposer quelque chose de surprenant mais je pense que nous devrions utiliser le même boitier que le Celeron, en terme de dimensions physiques et de placement. De cette manière, nous pourrons utiliser les radiateurs/ventilateurs du Celeron déjà sur le marché et le matériel de montage du Celeron.

Jeu PCI : je vais encore proposer une hérésie mais je pense que nous pourrions utiliser les cartes mères Slot 1 à 100MHz. En premier, Intel n'est plus le seul à fabriquer les chipsets Slot 1 : VIA vient juste de sortir un chipset Slot 1 avec des performances excellentes et les dernières améliorations en terme de technologie (nous pouvons avoir les informations de synchronisation sur les datasheets des chipset VIA). En second, nous n'avons plus besoin de nous inquiéter sur l'avenir du jeu carte mère/PCI. En troisième, il est à peu près impossible d'aller au-delà des 100MHz sur une carte mère standard à cause des émissions RFI ; ((((((((((so basically 100-112MHz is as good as it gets)))))))))). En quatrième, il y aura beaucoup de personnes avec des cartes mères Slot 1, souhaitant mettre à jour leur CPU PII/Celeron (spéciallement le Celeron). En cinquième, ces cartes mères sont bon marché actuellement et nous avons les bénéfices des gros volumes de production. En sixième, ceci permet les mises à jour faciles des CPU Freedom vers des indices de vitesse plus importants, des versions de caches plus grandes, des versions avec FPU, etc.

Maintenant, si nous acceptons ce qui est au-dessus, nous devons mettre sur le circuit imprimé du Freedom une petite EEProm qui contiendra le BIOS Freedom, le cache L2 et un socket pour le FPU. Ceci augmente le coût du CPU mais diminue le coût global donc je pense que c'est un bon mouvement.

Please regardez une photographie du Celeron et dites moi si je suis en train de réver.

3.1.8 Annexe C

Emissions légales / financières

5 Août 1998

Nous souhaiterons avoir un support de la Free Software Foundation pour le projet Freedom.

Nous ne proposons pas que la Free Software Foundation construise une usine. Ce que nous proposons c'est que : si nous voyons des fondeurs aux US ou Taiwan, leur donnons un masque et leur demandons de faire un batch de 0.25 micron, 5 couches sur wafer 8-inch pour nous, qu'ils nous épaulent à hauteur de approximativement $3K-5K ou même moins, par wafer, sur leurs prix (notre coût) pour nos batchs (dans l'année 2000).

Un coût approximatif pour un batch de CPU F1 tournera autour de $500k et $1000K, pour 5000-10000 CPUs on.

Ce n'est pas exactement de l'agent de poche mais nous pouvons vendre ces CPU sur une base de souscription. Comme ceci : les personnes qui y souscrivent auront le Merced-killer pour à peu près $100 (comparé au coût prévu de $5000/unité pour le Merced), sur une base de premier arrivé/premier servi et tout les CPU restants après les coûts de couverture du batch pourraient être vendus à un coût légèrement supérieur pour payer les batchs suivants et d'autres développements de masques.

Nous suggérons de mettre quelques quotas dans le système. La demande est susceptible d'être supérieure à l'offre. ;-)

La Free Software Foundation pourrait coordiner tous les aspects légaux/financiers/logistiques du projet (et auront une compensation adéquate pour ce travail). Ceci, bien entendu, dépend de l'obtention du support de Mr. Stallman pour cette initiative.

Chapitre 4

Un morceau d'histoire du F-CPU

(Et une réflexion sur l'évolution du F-CPU au travers d'un description des différentes architectures proposées.)

4.1 M2M

La première génération était une architecture "mémoire à mémoire" (M2M) qui a disparu avec les membres de l'équipe originale (ils ont écrit le texte précédent). Ils pensaient que le temps de commutation de contexte prenait beaucoup de temps, ils ont donc réservé des zones de mémoire pour le jeu de registres. De cette manière, ils pouvaient changer de registres en changeant leur adresse de base. Je n'ai pas recherché les raisons pour lesquelles ceci a été abandonné car je suis arrivé plus tard dans le groupe. De toute manière, ils ont lancé le projet F-CPU, avec les buts que nous connaissons maintenant et le rêve de créer un "Merced Killer". Je pense que nous pouvons réellement nous mesurer avec l'ALPHA directement ;-)

4.2 TTA

La seconde génération était la "Transfer Triggered Architecture" (TTA) où les calculs étaient déclenchés par les transferts entre les différentes unités d'exécution. Les instructions consistent principalement en des nombres correspondant aux "registres" de source et de destination, qui peuvent être des ports d'entrée ou de sortie d'unités d'exécution. Dès qu'il est écrit quelque chose dans les ports d'entrée d'une unité, l'opération est effectuée et le résultat est lisible sur le port de sortie. Cette architecture a été proposée par AlphaRISC l'anonyme, aussi connu comme AlphaGhost. Il a fait un gros travail dessus mais il a quitté la liste de diffusion et le groupe a perdu la piste du projet avec lui.

Brian Fuhs (bkfuhs1@attglobal.net) a expliqué le TTA sur la liste de diffusion de cette manière :

```
TTA signifie Transfer-Triggered Architecture. L'id\'ee de base est de ne pas dire au CPU
ce qu'il doit faire des donn\'ees, mais seulement o\'u les mettre. Alors, en positionnant les
donn\'ees aux bons endroits, vous r\'ecup\'erez magiquement de nouvelles donn\'ees \'a d'autres
endroits qui r\'esultent d'op\'erations effectu\'ees sur les anci\'ennes donn\'ees. Alors que sur
une machine traditionelle OTA (operation-triggered architecture), vous devez dire
"ADD R3, R1, R2"; dans une TTA, vous diriez "MOV R1, add; MOV R2, add; MOV add, R3".
L'int\'er\^et du jeu d'instruction (si on peut dire, car le TTA n'a qu'une instruction : MOV)
est sur la donn\'ee elle-m\^eme, \'a l'oppos\'e des op\'erations que vous faites sur la donn\'ee. Vous
sp\'ecifiez simplement les adresses puis vous cartographiez celles-ci en des fonctions comme
ADD ou DIV.

C'est l'id\'ee de base. Je devrais commencer en sp\'ecifiant que je me concentre ici sur le
traitement global et j'ignore temporairement les choses comme les interruptions. De cette
mani\'ere, il est possible de traiter les cas r\'eels car personne ne l'a encore fait. Pour
l'instant, je me suis plus int\'erress\'e \'a la th\'eorie. Tout pipeline CPU peut \^etre d\'ecompos\'e
en trois \'etapes de base : approvisionnement et d\'ecodage, ex\'ecuter et stocker. Garbage in,
garbage processing, garbage out. Avec les OTA, tout est fait par mat\'eriel. Vous donnez
"ADD~R3,~R1,~R2" et le mat\'eriel fait le reste. Il traite les moyens de communication interne
pour obtenir les donn\'ees de R1 et R2 vers l'entr\'ee de l'additionneur, le laisse traiter
l'information et r\'ecup\'ere les donn\'ees \'a la sortie vers le fichier de registre, dans R3. Dans
les architectures modernes, il contr\^ole les anomalies, fait suivre les donn\'ees pour que le
```

reste du pipeline puisse l'utiliser plus t\^ot et truc encore plus compliqu\'e comme r\'eordonner les instructions. Le logiciel ne connait que le 32 bits ; le mat\'eriel fait le reste.

L'\'etape IF/ID d'un TTA est tr\'es diff\'erente. Toute la charge est plac\'ee sur le logiciel. L'instruction n'est pas sp\'ecifi\'ee comme ADD (quelque chose), mais comme une s\'erie de paires d'adresses SRC, DEST. Tout ce que le mat\'eriel doit faire est le contr\^ole des bus internes pour mettre les donn\'ees l\'a o\'u elles sont suppos\'ees aller. Toutes les v\'erifications de risque, d'ordre d'instructions optimal, etc doit \^etre fait par le compilateur. Ici, la cl\'e est qu'un TTA, pour r\'ealiser des mesures IPC comparables \'a un OTA, doit \^etre VLIW : vous DEVEZ pouvoir sp\'ecifie des mouvements multiples en un seul cycle, pour que vous puissiez d\'eplacer toutes vos sources de donn\'ees vers les zones appropri\'ees et encore bouger les r\'esultats vers votre fichier de registres (o\'u l\'a ou vous voulez qu'ils aillent). En r\'esum\'e, pour faire un "ADD~R3,~R1,~R2", le mat\'eriel fera ce qui suit :

TTA	OTA
MOV R1, add	ADD R3, R1, R2
Move R1→adder	*Contrôle de danger*
MOV R2, add	*Contrôle d'additionneur disponible*
Move R2→adder	*Sélectionne un bus interne et déplace la donnée*
(l'additionneur fait le traitement dans les deux cas)	
MOV add, R3	*Contrôle de danger*
Move adder→R3	*Planifie l'instruction pour la récupération*
	Sélectionne un bus interne et déplace la donnée
	Abandonne l'instruction

Le compilateur, bien s\^ur, devient beaucoup plus compliqu\'e car il doit faire tout le travail de
planification, au moment de la compilation. Mais le mat\'eriel n'a pas besoin de s'occuper de grand
chose dans un TTA... Tout ce qu'il fait dans les cas simple est d'envoyer les instructions et de g\'en\'erer les signaux de contr\^ole pour tous les bus.

L'ex\'ecution est identique entre le TTA et OTA. Crunch the bits. Period.

La compl\'etion des instructions est encore simplifi\'ee dans un TTA. Si vous voulez un comportement correct, assurez vous que le compilateur g\'en\'ere les bonnes s\'equences de d\'eplacement. C'est \'a comparer avec un OTA o\'u vous devez au moins savoir quels port d'\'ecriture vous devez utiliser, etc.

A la base, un TTA et un OTA sont fonctionellement identiques. La principale diff\'erence est qu'un TTA doit vraiment \^etre VLIW et demande plus au compilateur. N\'eanmoins, si la philosophie "smart compiler and dumb machine" est r\'eellement l\'a o\'u l'on veut aller, le TTA doit fonctionner. Il offre une part plus importante du pipeline au logiciel, r\'eduisant les besoins de mat\'eriel et laissant plus de champ au compilateur pour l'optimisation. Bien s\^ur, il y a des cas comme les embouteillages de code et la g\'en\'eration de constantes, mais ces cas sont trait\'es plus tard. Les id\'ees de base ont \'et\'e cou ici (quoique d'une mani\'ere assez d\'ecousue)... J'ai compos\'e cet email dans ma t\^ete et j'ai trouv\ quelques explications claires directement lorsque je me suis assis et j'ai commenc\'e \'a saisir). Pour plus d'informations, voyez http://www.cs.uregina.ca/~bayko/design/design.html et
http://cardit.et.tudelft.nl/MOVE. Ceux-ci ont beaucoup plus d'informations sur les d\'etails de TTA; j'esp\'ere encore que nous pourrons le mener \'a bien et je pense que cela sera bien pour la performance g\'en\'erale, de co\^ut et en simplicit\'e. En plus, c'est suffisamment r\'evolutionaire pour faire d\'e les t\^etes -et que cela puisse nous donner une plus grande base d'utilisateurs (et de d\'eveloppeurs) et rendre le projet plus stable.

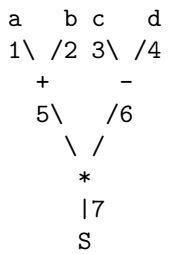
Envoyez moi les questions, je sais qu'il en a beaucoup ...

Brian

Si vous voulez comprendre le concept TTA un peu mieux, la différence est dans la philosophie, c'est comme si vous aviez des instructions pour coder le flux de données de la machine à la volée. Notez aussi le fait que moins de registres sont nécessaires : les registres sont requis pour stocker les résultats temporaires des opérations entre les instructions d'une séquence de code. Ici, les résultats sont directement stockés par les unités d'où un moindre besoin de "stockage temporaire" et moins de pression sur les registres.

Pour visionner cette différence, pensez au graphe de dépendance des données : dans un OTA, une instruction est un noeud alors que dans TTA l'instruction mov est la branche. Une fois que cela est compris, il n'y a pas beaucoup de travail à faire sur un compilateur existant (jusqu'ici simple) pour générer des instructions TTA.

Examinons : $S = (a+b)*(c-d)$ par exemple. a, b, c, d sont des "ports", des registres ou des adresses TTA connus.



Dans l'OTA, avec des instructions à 3 opérandes, il y a une instruction par "node" (+, -, *). Deux registres temporaires sont nécessaires pour stocker le résultat de l'addition et de la soustraction (branches 5 et 6). Supposons que le ((((((((((tree-flattening)))))))))) doit préserver la superscalarité (bien, les instructions ont des temps d'attente), donc nous codons :

```
ADD r5, a, b
SUB r6, c, d
MUL r7, r5, r6
```

(par exemple).

Dans TTA il y a un "port" dans chaque unité pour chaque branche entrante. Ceci signifie que ADD, ayant deux opérandes, a deux ports. Il y a un port de résultat qui utilise l'adresse d'un port mais qui est utilisé en lecture, pas en écriture. Un autre détail est que le port de lecture peut être statique : il contient le résultat jusqu'à ce qu'une autre opération ne soit déclenchée. Nous pouvons coder

```
mv ADD1,a
mv SUB1,c
mv ADD2,b    (ceci d\'eclenche l'op\'eration a+b)
mv SUB2,d    (ceci d\'eclenche l'op\'eration c-d)
mv MUL1,ADD
mv MUL2,SUB  (ceci d\'eclenche l'op\'eration *)
```

TTA n'est pas "meilleur", il n'est pas "pire", il est juste complètement différent alors que les problèmes seront toujours les mêmes. Si les instructions sont à 16 bit, il faut 96 bits, comme le ferait un exemple OTA. Dans certains cas, cela peut être meilleur comme cela a déjà été montré sur la liste. TTA a quelques propriétés intéressantes mais malheureusement, dans un futur proche, il n'est pas probable que TTA puisse rentrer dans les gros ordinateurs RISC ou CISC. Un coeur TTA peut être aussi efficace que le coeur ARM, par exemple, cela convient bien à cette taille ((((((((((scale of die size)))))))))) mais trop peu d'études ont été faites, comparé aux études existantes sur l'OTA. La solution de sa croissance n'étant pas (encore) connue, ceci mène à la controverse qui a secoué la liste de diffusion autour de décembre 1998˜ : le problème de savoir où mapper les registres, comment les ports seront remappés à la volée, etc. Lorsque des instructions additionnelles sont nécessaires, ceci met en danger l'équilibre complet du CPU et l'évolutivité est plus contraignante que pour les RISC ou OTA en général.

Les problèmes physiques des bus ont été aussi soulevés : si nous avons disons 8 bus de 64 bits, cela fait 512 fils, qui prennent autour d'un millimètre d'épaisseur avec un processus de .5u. Bien sûr, nous pouvons utiliser un ((((((((((((crossbar)))))))))))) à la place.

Comme nous en avons parlé quelques fois, il y a longtemps, à cause de ses problèmes de croissance (assignation des ports et flexibilité), TTA n'est pas le choix parfait pour une famille de CPU sur le long terme alors que le ratio performance/complexité est bon. Il est donc possible que le groupe F-CPU travaille sur un traducteur RISC → TTA devant un coeur TTA qui n'aura pas la plupart des problèmes de croissance. Il sera appelé le "FC1" (FC0 est le coeur RISC). Bien sûr, le temps montrera comment les fantômes TTA du groupe F-CPU changerons.

Mais le problème du TTA est trop spécialisé là où l'OTA peut changer son coeur et toujours utiliser les mêmes binaires. C'est un des points qui a "tué" la tentative précédente du F-CPU. Chaque implémentation TTA ne peut pas être complètement compatible avec une autre à cause du format des instructions, de l'assignation des "ports" et autres détails similaires : la notion "d'instruction" est liée à la notion de "registre".

Je ne suis pas en train de prouver les avantages d'une technique sur une autre, je tente de montrer la différence de point de vue, qui traite finalement du même problème. La croissance, qui est nécessaire pour un tel projet est plus importante que nous le pensons et le groupe s'est intérressé à une technologie plus classique lorsque AlphaRISC l'a quitté.

4.3 RISC Traditionel

La troisième génération est partie des membres de la liste de diffusion qui ont naturellement étudié les bases de l'architecture RISC, comme la première génération de processeurs MIPS ou le DLX (décrit par Patterson et Hennessy dans leur livre "QA"), le MMIX (Knuth), les CPUs MISC (tels que les machines Forth de Chick Moore ou le 4Stack de Bernd) et d'autres projets similaires simple. Ces conceptions sont expliquées et décrites dans des livres bien connus et enseignés en universités. A partir d'un simple projet RISC, le projet est devenu plus complexe et est devenu indépendant des autres architectures existantes, principalement grâce aux leçons apprises de leur histoire et des besoins spécifique du groupe, qui a mené vers des choix adaptés et des caractéristiques particulières. C'est ce dont nous parlons dans les parties suivantes de ce document.

Chapitre 5

Les contraintes de conception

Le groupe F-CPU est plutôt hétérogène mais chaque membre a le même espoir que le projet devienne une réalité car nous sommes convaincus que ce n'est pas impossible et donc faisable. Rappelez-vous la Constitution du Projet Freedom CPU :

```
"
   D\'evelopper et rendre une architecture librement disponible, ainsi que toutes les
propri\'et\'es intellectuelles n\'ecessaires pour fabriquer un ou plusieurs impl\'emen-
tations de cette architecture, avec les propri\'et\'es suivantes, dans l'ordre d\'ecroissant
d'importance~:

  1) adaptabilit\'e et utilit\'e pour le champ d'applications le plus grand possible
  2) Performance, accent mis sur le parall\'elisme au niveau de l'utilisateur et d\'eriv\'e
avec une architecture intelligente, plut\^ot que par des processus avanc\'es en electronique
  3) Architecture ((((((((((lifespan)))))))))) et compatibilit\'e directe
  4) Co\^ut, incluant des consid\'erations mon\'etaires et thermiques

"
```

Nous pourrions aussi ajouter : 5) réussir !

Ce texte résume beaucoup d'aspects du projet : c'est une "propriété intellectuelle libre", signifiant que tout le monde peut faire de l'argent avec sans être inquiété, tant que le produit respecte les règles et les standards généraux décrits dans la charte F-CPU et toutes les caractéristiques sont librement disponibles (sous la GNU Public Licence et en respectant la charte F-CPU). Tout comme le projet LINUX, les membres du groupe espèrent que la libre disponibilité de cette conception bénéficiera à tout le monde en réduisant les coûts des produits (car la plupart du travail intellectuel est déjà réalisé), en fournissant un standard ouvert et flexible que tout le monde peut influencer à volonté sans signer de NDA. C'est aussi un banc de test de nouvelles techniques et le "premier CPU" pour un grand nombre de "passionnés" qui peuvent le construire à la maison. Bien sûr, les autres résultats sont que le F-CPU sera utilisé dans tous les ordinateurs famillliaux ainsi que par tous les autres marchés spécialisés (embarqués/temps réels, ordinateurs portables/portatifs, machines parallèles pour le décodage de nombres scientifiques...).

Dans cette situation, il est clair qu'un seul circuit ne remplit pas tous les besoins. Il existe aussi des contraintes économiques qui influencent aussi les décisions technologiques et tout le monde ne peut pas accéder aux unités de fabrication des fondeurs les plus avancés. La réalité du F-CPU "pour et par tout le monde" est plus dans le royaume des FPGA reconfigurables, des ((((((((((sea-of-gates)))))))))) bon marché et des ASICS qui seront fabriqués en petits volumes. Même si le but ultime est d'utiliser des technologies entièrement personnalisées, il existe une forte limitation dûe aux prototypes et aux faibles volumes. La complexité est limitée pour les premières générations et FC0, les estimations du nombre de transistors pour les premiers circuits seront de 1 Million, incluant un peu de cache. C'est plutôt faible comparé aux CPUs actuels mais c'est énorme si on se rappelle les coeurs ARM ou les premiers CPUs RISC.

La "Propriété Intellectuelle" est disponible car les fichiers VHDL'93 (ou VERILOG) sont disponibles pour quiconque peut les lire, les compiler et les modifier. Une vue schématique est aussi souvent nécessaire

pour comprendre une fonction d'un circuit d'un premier coup d'oeil. Le processeur existe alors plus sous la forme d'un logiciel descriptif que d'un circuit matériel. Ceci permettra à la famille de processeur d'évoluer plus facilement et mieux que les versions commerciales et ce polymorphisme garantira que tout le monde pourra trouver le meilleur coeur dans toutes les situations. Et comme le développement du logiciel sera commun à toutes les puces, librement disponible avec la GPL, le portage de tous les logiciels sur toutes les plateformes sera facilité au maximum.

L'intéropérabilité du logiciel quelque soit le membre de la famille est une forte contrainte et probablement une des règles de développement du projet les plus importantes : `"AUCUNE RESSOURCE NE DOIT ETRE LIMITEE"`. Ceci mène à créer un CPU avec une largeur de donnée "indéterminée". Une puce F-CPU peut implémenter une caractéristique de largeur de donnée de toute taille au dessus de 32 bits. Le logiciel portable respectera quelques règles simples donc il pourra tourner aussi rapidement que le processeur le peut, indépendemment des considérations algorithmiques. En fait la vitesse d'un CPU est déterminé par des contraintes économiques et le concepteur construira un CPU aussi large que le permettent le budget et la technologie. De cette manière, il n'y a pas d'autre "roadmap" que les besoins des utilisateurs car c'est son propre fondeur. Le projet n'est pas limité par la technologie et est suffisamment flexible pour durer... aussi longtemps que nous le souhaitons.

Chapitre 6

Cheminement du projet

Il existe des étapes que le projet tente de suivre dans le futur. Il N'Y A PAS DE PLANNING car c'est un projet grossisant naturellement, pas un projet orienté vers le commercial ; nous sommes plus concernés par la pertinence et l'efficacité que par la mise sur le marché dans les temps ; et plusieurs "coopérateurs" peuvent changer les priorités du groupe F-CPU. Ce cheminement n'est pas définitif, il a déjà été changé et changera dans le futur. Il aide à comprendre les orientations du travail du groupe. Les étapes suivantes sont néanmoins très importantes et montrent que c'est un projet EVOLUTIF plutôt qu'une utopie sans fondement.

Génération	Prototype	Pre-séries	Classe Commerciale
Nom de Code	"POC" : Preuve du Concept	"JOUET" : dois-je en dire plus ?	F1, F2, F3 ... d'autres sobriquets seront trouvés (et trademarkés)
But	Avoir une "puce" qui peut être montrée ou avec laquelle on peut faire une démonstration dans les salons commerciaux / conférences, faire fonctionner le coeur FC0, le tester, explorer la mémoire d'interface et son impact sur la performance, fabriquer le premier circuit qui fonctionne, prouver les suppositions initiales architecturales, prouver que le concept du F-CPU est possible. *Il N'est PAS prévu d'être commercialisé car il n'aura que des fonctions limitées. D'autres F-CPUs limités devront être dérivés de conceptions plus avancées, démarrant avec la "classe commerciale".*	Fournir les premier utilisateurs avec une plateforme avancée, encore limitée pour tester le F-CPU au réel. Permettre aux gens d'écrire des logiciels réels et d'avoir de l'expérience avec le jeu d'instruction et les habitudes de programmation, de manière à modifier ultérieurement le jeu d'instructions et l'architecture pour la classe commerciale. *Ce n'est même pas une conception à partir de laquelle les autres architectures doivent être dérivées. Le but est de réaffecter la carte des opcodes et d'apprendre à concevoir des ASICs, de même que faire de la publicité / de la couverture de presse / ((((hype)))).*	Définir une plateforme matérielle à partir de laquelle les autres puces compatibles broches à broches peuvent être dérivées. La "carte mère" et les interfaces I/O doivent fournir autant d'espace libre que possible pour de futures améliorations. Les "coopérateurs" auront une base commune pour développer des puces efficaces. Le principal problème étant la bande passante mémoire, l'interface mémoire sera TRES large pour que les circuits ne l'attendent pas. *A ce moment, une première version stable de l'architecture de référence sera officielle. Cela évoluera ensuite naturellement.*
Technologie	CMP / Europractrice / ATMEL / HITACHI selon les sponsors et les opportunités et le budget disponible. Probablement autour de 0.35, 5V. Cela pourrait être le prix d'un concours de conception.	ATMEL / HITACHI selon les sponsors et les opportunités et le budget disponible. Probablement 0.35 ou 0.25, 3.3V	Selon les souhaits de chacun.

Vitesse	Une des choses à faire est de le cadencer avec une horloge à PLL externe. La mémoire étant asynchrone avec le coeur, nous aurons la possibilité de tester la capacité du coeur à des fréquences hautes et basses. Je n'ai absolument aucune idée des fréquences que nous pouvons avoir de cette manièrer.	Au moins plus que le prototype.	Aussi vite que vous pouvez...
Nombre	Une demi-douzaine	Quelques centaines ou milliers	Beaucoup plus !
Taille des mots	64	64	64 ou plus (toute puissance de 2, au-dela de 32 bits)
Champ d'adressage mémoire	logique : 64 bits physique : 20 (+5) bits (économique)	logique : 64 bits physique : 32 (+5) bits extérieurs + 4 slots SDRAM (mux[10+12](+5) = 27 bits) de mémoire privée (confortable...)	logique : 64 bits physique : 64 (+5) bits extérieurs, 4 ou 8 slots SDRAM (28 bits) (prêt pour faire des grands clusters)
taille des bus de mémoire externe	64 bits (SDRAM privée asynchrone) + 8 bits (port de débug)	128 bits + 16 ECC pour de la SDRAM privée, bus de 32 multiplexé + bursté + asynchrone "I/O" (memory-mapped)	256 bits + 32 ECC de bus mémoire externe (DDR-SDRAM ?) + 64 bits de "IO" memory-mapped (multiplexé, bursté, asynchrone)
JTAG / debug embarqué	interface multiple d'octet personnalisée	JTAG (ou similaire) + bus I / O (utilisé pour un examen rapide / port de debug)	JTAG + port I / O
Cache	Donnée embarquée + instruction, 2KB chaque.	Donnée embarquée + instruction, 4 ou 8 Kb chaque.	Donnée embarquée + instruction, 8 Kb ou plus, chaque. Cache externe : bus de donnée partagé avec la SDRAM, TAG SRAM embarquée
Instructions par cycle	1	1	1 ou plus
Coeur	FC0	FC0	FC0 et autres
Espérance de vie	Courte (mois)	Courte (pas plus de quelques années)	Beaucoup plus longue :-)
Evolutivité / Compatibilité	Aucune (proto)	Aucune	Oui
Carte mêre (Module CPU)	PCB simple ou double face, interface avec le bus ISA ou similaire	PCB haute qualité 6-couches + PCB I/O maison (simple couche)	PCB de haute qualité, haut volume de production + I/O + intercom + PCB EEPROM (des ponts PCI, AGP, IDE / SCSI seront nécessaires)
Cible / utilisateurs	Groupe F-CPU, démonstrateurs et utilisateurs avancés	Programmeurs / Développeurs / Intégrateurs Avancés	tout le monde (au dessus de 10ans)

Nous espérons que cette table répond à la pluspart de vos questions. Si ce n'est pas le cas, n'hésitez pas à demander.

Deuxième partie

Description générale du F-CPU

2.1 Les caractéristiques principales

Le CPU décrit ici peut être vu comme un croisement entre un circuit R2000 (ou un ALPHA des débuts) et un ordinateur CDC6600. Quelques contraintes sont identiques : le F-CPU doit être aussi simple et performant que possible. Du R2000, il hérite de ses principales caractéristiques RISC comme la taille fixée des instructions, le jeu de registres et la taille du circuit qui est liée par la technologie courante. Dans le CDC6600, FC0 y a trouvé le schéma d'exécution, le tableau, les unités d'exécution parallèle multiples et surtout : l'inspiration pour les techniques complexes qui facilitent à la fois la conception et la programmation.

Récement, le CPU SH5 (Hitachi/ST) a montré des éléments similaires, comme les 64 registres ou les buffers de cibles de sauts. Vous remarquerez néanmoins que F-CPU est complètement différent, particulièrement du point de vue du planificateur.

Le texte suivant est une description pas à pas du F-CPU actuellement développé. Les caractéristiques seront plus profondement décrites et rendues interdépendantes, il est donc recommandé de les lire à partir du début :-) Nous allons commencer avec la caractéristique la plus basique de F-CPU avant de parler de sujets plus critiques et dépendants du matériel dans la partie suivante.

2.2 Les instructions ont une largeur de 32 bits

C'est un héritage des processeurs RISC traditionnels et les bénéfices des tailles d'instructions fixes n'est plus rediscuté, excepté pour certaines applications spécifiques. Même le marché des micro-contrôleurs est envahi par les coeurs RISCs avec des instructions de taille fixée.

La taille des instructions peut être néanmoins discutée plus avant. Il est clair qu'un mot de 16 bits ne peut pas contenir suffisamment d'espace pour coder des instructions à 3 opérandes impliquant des dizaines de registres et d'opérations. Il existe des processeurs à instructions 24 et 48 bits mais ils sont limités à des niches de marchés (comme les DSP) et ils ne sont pas adaptés aux lignes de cache à puissance de deux. Si nous accédons à la mémoire sur une base d'octet, cela devient trop complexe. Le processeur F-CPU est principalement 64 bits, des instructions de cette taille ont été proposées où deux instructions sont inclues mais ceci est identique aux 2 instructions 32 bits qui peuvent être atomiques alors que les paires 64 bits ne peuvent être séparées. Il existe aussi le Merced (IA64) qui possède des mots d'instructions 128 bits, chacun contenant 3 opcodes et des informations de dépendances des registres. Depuis, nous utilisons un simple tableau et comme les compilateurs identiques au IA64 (VLIW) sont vraiment très difficiles à programmer, nous laissons le coeur CPU décider ou non de bloquer le pipeline lors des besoins, et donc permettre un grand champ de types de coeurs CPU pour exécuter les mêmes instructions et programmes simples.

L'architecture du F-CPU n'a pas été définie au début du projet et les instructions doivent s'exécuter sur un grand champ de types de processeurs (pipeliné, superscalaire, out-of-order, VLIW, quoi que va créer le futur). Un jeu d'instructions de taille 32 bits, à la taille fixée, semble être le meilleur choix pour la simplicité et l'augmentation de taille dans le futur. Des optimisations dépendantes du coeur peuvent être faites sur les binaires en appliquant des règles de planification spécifiques mais les applications vont toujours tourner sur d'autres membres de la famille qui ont des coeurs complètement différents.

2.3 Registre #0

C'est un "read-as-zero/unmodifiable". C'est une autre technique classique "RISC" qui est destiné à faciliter le codage et réduire le compte des opcodes. C'est valable pour les processeurs précédents mais les technologies actuelles ont besoin de trucs spécifiques sur ce que font les instructions. Il nous parait inutile, de nos jours, de coder "SUB R1,R1,R1" pour effacer R1 car il doit traiter R1, faire une soustraction 64 bits et écrire le résultat, alors que tout ce que nous voulons est d'effacer simplement R1. Ce temps d'attente a été caché dans mes premiers processeurs MIPS mais les technologies actuelles souffrent de ces techniques de codage car chaque étape contribuant à la réalisation d'une opération coûte cher. Si nous voulons accélérer ces instructions, leur décodage devient plus complexe. Donc si la convention R0=0 est gardée, il y a plus d'insistance sur les instructions spécifiques. Par exemple, "SUB R3,R1,R2" qui compare R1 et R2, en général pour savoir si c'est égal ou plus grand, peut être remplacé dans F-CPU par "CMP R3,R1,R2" car CMP peut utiliser une unité spéciale de comparaison qui possède moins de temps d'attente qu'une soustraction (après tout, nous n'avons rien à faire du résultat numérique, nous voulons simplement ses "propriétés").

"MOV R1,R0" efface R1 sans temps d'attente car la valeur de R0 est déjà connue (matériellement à zéro).

2.4 Le F-CPU possède 64 registres

Les processeurs RISC possèdent traditionnellement 32 registres. Plus qu'une guerre de religion, ce sujet prouve que les choix de conceptions sont grandement influencés par beaucoup de paramètres (ceci ressemble à un thread sur `comp.arch`).

Examinons les :

- *"Il est prouvé que 8 registres sont largement suffisants pour la plupart des algorithmes."* est un argument mortel qui apparait quelques fois. regardons pourquoi et comment la conclusion a été faite :
  - c'est une vieille étude,
  - elle est basée des exemples d'algorithmes d'écoles,
  - la mémoire était moins contraignante que maintenant (même si les coeurs magnétiques étaient lents) et les instructions mémoire à mémoire étaient courantes,
  - les circuits avaient beaucoup moins d'espace que maintenant (des dizaines de milliers contre des dizaines de millions) et un registre était une ressource matérielle chère
  - les pipelines n'étaient pas aussi profonds que maintenant
  - nous utilisons TOUJOURS des algorithmes "spéciaux" car chaque programme est une modification et une adaptation de cas commun vers des cas spéciaux, (nous vivons dans un monde réel, vous savez !)
  - quiconque a déjà programmé des processeurs x86 en langage assembleur sait comment c'est dur...

  La raison réelle pour avoir beaucoup de registres est celle de réduire le besoin de stockage et de chargement à partir de la mémoire. Nous savons que même avec plusieurs niveaux de cache mémoire, les architectures classiques sont limitées par la vitesse mémoire, donc garder plus de variables proches des unités d'exécution va réduire le temps d'attente global.
- *"s'il y a trop de registres, il n'y a plus de place pour le codage des instructions"* : c'est pourquoi la conception de processeurs est un art d'équilibre et de bon sens commun. Et nous sommes des artistes, n'est-ce pas ? Avec le renommage des registres, le nombre des registres physiques peut être virtuellement étendu aux limites physiques.
- *"Plus il y a de registres, plus c'est long pour commuter entre les tâches et prendre en compte les interruptions"* est une autre raison qui a été beaucoup discutée.

  Alors devinez pourquoi Intel a placé 128*2 registres dans IA64 ? ? ?

  Il est néanmoins clair qu'une commutation de contexte *RAPIDE* est un problème pour un tas de raisons évidentes. Quelques techniques existent et sont bien connues, comme des fenêtres de registres (à la SPARC), des commutations de bancs de registres (comme dans les DSPs) ou des architectures mémoires à mémoire (pas assez connues), mais aucune d'entre elles ne peut exister dans une conception simple d'un premier prototype où les transitors comptent et la complexité est un réel problème.

  Dans les discussions sur la liste de diffusion, il apparait que :
  - la plupart du temps est réellement passée dans le code du planificateur de tâches (si nous parlons de la vitesse de l'OS) donc le problème de restauration de registre est l'arbre qui cache la forêt,
  - le nombres de sauts dans la mémoire généré par une commutation de contexte ou une interruption gaspille beaucoup de temps lorsque la bande passante mémoire est limitée (le bon sens et les mesures de performances sur un PII feront le reste si vous n'êtes pas convaincu)
  - Un programmeur performant entrelacera le code de sauvegarde des registres avec le code de traitement des IRQs car une instruction a habituellement besoin d'une destination et de deux sources. Donc si le CPU exécute une instruction par cycle, il N'y a PAS de besoin de commuter tous le jeu de registres en un cycle. En peu de mots, pas de besoin de bancs de registres. Ces faits conduisent à concevoir le "Smooth Register Backup" (SRB), une technique matérielle qui remplace le logiciel dans l'entrelacement du code de sauvegarde et de calcul.

    Considérons un code IRQ démarrant comme ceci :

    IRQHANDLER :

```
clear   R1          ; cycle 1
load    R2,[imm]    ; cycle 2
load    R3,[imm]    ; cycle 3
OP      R1,R2,R3    ; cycle 4
OP      R2,R3,R0    ; cycle 5
store   R2,[R3]     ; cycle 6
...
```

Quelque soit le nombre de registres, nous n'avons qu'à sauvegarder R1 avant le cycle 1, R2 avant le cycle 2 et R3 avant le cycle 3.
Cela prendra 3 instructions qui seront interlacées comme ceci :

```
IRQHANDLER :
    store   R1,[imm]
    clear   R1          ; cycle 1
    store   R2,[imm]
    load    R2,[imm]    ; cycle 2
    store   R3,[imm]
    load    R3,[imm]    ; cycle 3
    OP      R1,R2,R3    ; cycle 4
    OP      R2,R3,R0    ; cycle 5
    store   R2,[R3]     ; cycle 6
    ...
```

Le "Smooth Register Backup" est un mécanisme matériel simple qui sauve automatiquement les registres à partir du thread précédent pour qu'aucun code de sauvegarde n'ait besoin d'être entrelacé. Il est basé sur une technique simple de tableau , un algorithme "find first" et nécessite un drapeau par registre (positionné lorsque le registre a été sauvé, mis à 0 sinon). C'est complètement transparent pour l'utilisateur et le programmeur d'applications. Il peut donc être modifié sur les futures générations de processeurs avec un faible impact sur l'OS. Il permet d'économiser au moins 64 instructions de sauvegarde ou 256 octets de code qui ne sont pas chargés en mémoire. Cette bande passante est libérée pour d'autres opérations nécessitées par la commutation de tâches : charger un nouveau code, lire le nouveau contexte de tâche, écrire l'ancien contexte de tâche... cette technique sera décrite plus avant plus tard dans le chapitre 4.3.

La conclusion de cette discussion est que 64 registres ne sont pas de trop. L'autre problèmes est : 64 seront-ils suffisants ?

IA64 possède 128 registres et les processeurs superscalaires ont besoin de plus de ports de registres ; avoir plus de registres limite l'augmentation du nombre de ports de registres. Comme règle de base, un processeur aura besoin de *au moins (instructions par cycle) x (profondeur du pipeline) x 3* registres pour éviter les attentes dûes aux registres sur une séquence de code sans dépendances de registres. Et comme la profondeur du pipeline et des instructions par cycle augmentent tous les deux pour avoir plus de performance, la taille du jeu de registres augmente. 64 registres permettrons à un CPU superscalaire à 4 sorties d'avoir un pipeline à 5 étages, ce qui parait suffisamment complexe. Des implémentations futures utiliserons probablement le renommage de registre et des techniques out-of-order pour obtenir plus de performances sur le code commun mais les 64 registres sont suffisants pour un prototype. Comme pour augmenter le nombre d'instructions exécuté pendant chaque cycle, les futurs F-CPUs auront besoin d'un renommage de registre explicite. Ceci permettra à un F-CPU d'avoir des dizaines d'unités d'exécutions sans changer le format des instructions.

2.5 Le F-CPU est un processeur à taille variable

Les buts du F-CPU spécifient la *compatibilité directe*. Il y deux principales raisons derrière ce choix :
- Lors des évolutions des processeurs et de leurs familles, la largeur des données devient trop juste. Adapter la largueur des données au cas par cas mène à la complexité du x86 ou du VAX qui sont considérés comme des exemples sur l'horreur que peut devenir une architecture.
- Nous avons souvent besoin de traiter des données de tailles différentes en même temps, comme les pointeurs, les caractères, les nombres en virgule flottante et entier (par exemple dans une fonction

virgule flottante vers ASCII). Traiter toutes les données avec la même taille n'est pas une solution optimale car nous pouvons économiser des registres si plusieurs caractères ou entiers peuvent être stockés dans un seul registre et dans lequel on fera une rotation pour accéder à toutes les sous-parties.

Nous avons besoin, *depuis le début*, d'une bonne manière pour adapter la taille des données que nous traitons "à la volée". Et nous savons que la taille des données à traiter augmentera beaucoup dans le futur car c'est à peu près la seule technique pour augmenter les performances. Nous ne pouvons pas compter sur l'augmentation régulière des performances fournie par des nouveaux processus sur le silicium car ils sont chers et nous ne savons pas s'ils vont continuer. Le meilleur exemple de cette parallélisation des données est la programmation SIMD, comme dans les récents jeux d'instructions MMX, SSE, AlphaPC, PPC Altivec ou SPARC VIS où une instruction effectue plusieurs opérations. De 64, cela a évolué vers 128 et 256 bits par instruction et rien ne limite cette augmentation qui fournit plus de performance. Bien sur, nous ne fabriquons pas un CPU casseur de PGP et les entiers 512 bits ne sont presque jamais nécessaires. La performance tient dans le parallélisme, pas dans la largeur. Par exemple, il serait très utile de comparer des caractères en parallèle comme pendant les recherches dans des sous-chaines : la performance de tels programmes sera directement proportionnelle à la largeur des données que le CPU peut traiter.

La question suivante est : *quelle largeur ?*

Les tailles fixes des entiers et des pointeurs vont faire apparaitre un jour ou l'autres des problèmes qui ne seront pas résolus par le choix d'une taille fixe. Et, comme il l'a été vu dans le cas de la recherche dans une sous-chaine, le plus large donne le meilleur résultat. La solution est donc : *de ne pas décider de la taille des données traitées avant leur exécution.*

L'idée est que le logiciel doit tourner aussi rapidement que possible sur chaque machine, quelque soit sa famille ou sa génération. Les fabricants de circuits décident de la taille lors de la fabrication mais ce choix est indépendant du modèle de programmation car il prend aussi en compte : le prix, la technologie, le besoin, la performance...

Donc, en quelques mots : nous ne connaissons pas *a priori* la taille des registres. Nous devons lancer l'application qui reconnaitra la configuration de l'ordinateur avec des instructions spéciales et calibrera alors le compteur de boucles ou modifiera la mise à jour des pointeurs. C'est à peu près le même processus que le chargement de bibliothèques dynamiques...

Une fois que le programme a reconnu les caractéristiques de la taille des données que l'ordinateur peut gérer, le programme peut tourner aussi rapidement que le pc le permet. Bien sur, si l'application utilise une taille plus grande que ce qu'il est possible, cela génére un trap que l'OS peut traiter comme une erreur ou un élément à émuler.

Alors la question est : *comment* ?

La solution la plus facile est d'utiliser une table de correspondance qui interprète les 2 bits du drapeau de taille dans les instructions, comme défini dans la Partie 5 : L'Architecture du Jeu d'Instruction F-CPU. Les drapeaux sont *par défaut* interprétés de cette manière :

TAILLE du DRAPEAU	LARGEUR en octets	LARGEUR en bits
00	1	8
01	2	16
10	4	32
11	8	64

Utiliser une table de correspondance, située dans l'unité de décodage des instruction, permettra de modifier l'interprétation de ce champ, avec un rapport de puissance de deux. De cette manière, il n'existe pas de limitations sur l'instruction elle-même. La table de correspondance pourra voir ses valeurs par défaut changées avec 4 registres spéciaux. Les instructions ayant accès à ces registres spéciaux s'assureront que la protection et les tailles des données sont cohérentes et déclencheront une exception dans les autres cas. Un cinquième registre spécial sera cablé à la plus haute valeur possible qui est seulement dépendante du processeur.

nom du Registre Spécial	valeur par défaut en octets	fonction
SR_SIZE_0	1	signification de SIZE
SR_SIZE_1	2	signification de SIZE
SR_SIZE_2	4	signification de SIZE
SR_SIZE_3	8	signification de SIZE
SR_MAX_SIZE	inconnu (cablé)	Taille maximum gérée par le CPU

Le logiciel et particulièrement le compilateur sera un peu plus compliqué à cause de ces mécanismes. Les algorithmes seront modifiés (le compte de boucle sera changé par exemple) et les quatre registres spéciaux devront être sauvés et restaurés pendant chaque commutation de tche ou interruption. Les compilateurs simples et les CPUs à moins de 128 bits pourront utiliser simplement les quatres tailles par défaut mais des compilateurs plus sophistiqués seront nécessaires pour bénéficier des performances des derniers circuits les plus larges. L'interface doit être respectée par tous les membres de la famille et si le CPU ne suppporte pas les données plus grandes que 64 bits, le code ne doit pas tenter de modifier la taille donnée dans le registre spécial (précablé) sinon le CPU se bloquera. C'est la raison pour laquelle dans les algorithmes, la taille la "plus" large doit être utilisée avec SIZEFLAG=11 pour qu'ils puissent aussi bénéficier des processeurs précablés à des tailles plus petites.

Au moins, le problème de croissance est connu, résolu dès le départ et les techniques de codage ne vont pas changer entre les générations de processeurs. Ceci garanti un futur stable pour le projet et l'architecture FCPU et le vieux principe "RISC" de laisser le logiciel régler les problèmes est appliqué une fois encore. Nous pouvons envisager de cabler le prototype F1 avec des valeurs par défaut et tenter de les modifier déclenchera une erreur. Mais plus tard, les F-CPUs 4096 bits seront capables de lancer des programmes conçus pour les F-CPUs 128 bits et vice versa.

2.6 Le F-CPU est orienté SIMD

C'est une manière facile d'augmenter le nombre des opérations effectuées pendant chaque cycle sans augmenter la logique pour le contrôle. Les registres à taille variable permettent une croissance sans fin et donc une augmentation des performances à suivre mais chaque instruction effectuant des opérations sur les données doivent avoir un drapeau SIMD, pour différencier le type d'opération.

La taille maximum pour un élément SIMD ("chunk") est défini dans un Registre Spécial supplémentaire SR_MAX_CHUNK_SIZE. Il est habituellement mis à 64 sur une implémentation 64 bits car c'est l'entier le plus grand que le coeur peut traiter. Sur une architecture 128 bits, SR_MAX_CHUNK_SIZE restera probablement égal à 64 mais il pourrait aussi être à 32 ou 128.

2.7 Le F-CPU a des registres généraux

Ceci signifie que les nombres entiers sont mixés avec des pointeurs et des nombres en virgule flottante. L'objection la plus commune est du coté du matériel car le premier effet est qu'il augmente le nombre de ports de lecture/écriture dans le jeu de registres (c'est à peu près la même chose que d'avoir deux fois plus de registres).

Le premier argument du coté du F-CPU est que le logiciel devient plus simple et qu'il existe des solutions matérielles à ce problème. Le premier problème provient des algorithmes eux-même : certains sont purement basés sur des entiers alors que d'autres ont besoin de beaucoup de valeurs en virgule flottante. Avoir une séparation entre les registres pour les entiers et les nombres à virgule flottante handicapera les deux algorithmes car les registres spécialisés ne seront pas utilisés (par exemple, le jeu de registres Virgule Flottante sera inutilisé pendant les programmes comme les mailers ou l'édition graphique de bitmap logicielle alors que beaucoup de Virgule Flottante sera nécessaire pour le ray-tracing ou des simulations physiques). Et il y en a besoin de beaucoup lorsque cela se produit. Un autre aspect logiciel est sur la compilation, où les algorithmes d'allocation de registres sont critiques pour la performance. Avoir un simple (seul) "pool" de registres simplifie les décisions.

La seconde réponse au problème matériel est dans le matériel. Le premier circuit F-CPU, le F1, sera un processeur avec un pipeline à simple étage, où seuls trois registres de ports de lecture sont nécessaires et donc il n'y a pas de problème de jeu de registres au départ.

Les futurs circuits, avec plus d'instructions effectués par cycles pourront utiliser une autre technique : chaque registre possède un bit d'attribut (ou "propriété") qui indique si le registre est utilisé comme un pointeur, un nombre à vigule flottante, etc, pour qu'il puisse être inclus dans des jeu de registres différents tout en restant unifié du point de vue de la programmation. Les attributs sont regénérés automatiquement et n'ont pas besoin d'être sauvés ou restaurés pendant les commutations de contextes.

2.8 Le F-CPU possède des registres spéciaux

Ils stockent le contexte du processeur, gérent les fonctions vitales et assurent la protection.

Ces registres spéciaux ne peuvent être accédés que par quelques instructions spéciales et déclenchent une erreur si les registres n'existent pas ou s'il n'y a pas suffisamment de permission pour y accéder dans le contexte courant. Comme à peu près tout est géré avec ces registres spéciaux, ils sont la clé de la protection du système des opérations multi-utilisateurs et multi-tâches. Ces registres spéciaux sont très important pour reconnaitre la configuration du CPU et la "carte SR (Registres Spéciaux)" évoluera beaucoup dans le futur, par l'addition de nouveaux éléments sans toucher au jeu d'instruction. La carte SR courante peut être trouvée dans les fichiers F-CPU_config.vhdl et SR.h dans le dernier paquetage. Il n'existe pas encore de carte SR standard car elle sera définie à la fin de la phase de prototypage du F1.

Les instructions qui accèdent aux registres spéciaux sont séparées des autres car elles peuvent avoir une influence potentiellement dangereuse sur le matériel. Gérer les SR au travers de la mémoire (avec des instructions charger/stocker) rendra le pipeline beaucoup plus complexe. Par exemple, les SRs gèrent la mémoire virtuelle : les unités L/S demanderont des éléments spéciaux pour reconnaitre les adresses SR et éviter tout état instable de processeur (qui sont potentiellement dangereux). Le problème est identique à la commutation du mode protégé x86 où tous les pipelines et les descripteurs de mémoire cachés doivent être changés. Les SRs sont très similaires aux MSR introduits avec le CPU Pentium et ils peuvent aider la séparation des "opérations communes" (qui doivent être pipelinées et simples) à partir des "opérations de gestion" (lent, complexe et habituellement microcodé dans les CPUs CISC). Les instructions GET et PUT (voir leur description dans les partie VI) sont atomiques et ne perturbent pas le planificateur du pipeline.

2.9 Le F-CPU n'a pas de pointeur de pile

Ou plus exactement, il n'a pas de pointeur de pile dédié. En fait, il n'a pas du tout de pile car tous les registres peuvent être utilisés pour accéder à la mémoire. Un simple pointeur de pile cablé poserait des problèmes qui sont trouvés dans les processeurs CISCs et nécessiterait des trucs spéciaux pour les traiter. Par exemple, plusieurs instructions push et pop génèrent une utilisation de multiples registres en un seul cycle dans un processeur superscalaire avec une gestion matérielle spécifique.

Dans le monde du RISC, les conventions (les ABI) sont utilisées pour décider comment communiquer entre les applications ou comment initialiser les registres à leur démarrage. Pourvu que vous sauviez les registres entre deux appels, rien n'empêche que vous ayez 60 piles au même moment si votre algorithme en a besoin.

Accéder à la pile est effectué avec une simple instruction charger/stocker qui possède (seulement) une capabilité de post-increment. Considérons une pile descendante pointée par R3, nous coderons par exemple :

pop : *load 8,r3,r2* (r2=[r3] ; r3+=8)
push : *store -8,r3,r2* (r2=[r3] ; r3-=8)

Comme le traitement de l'addition et de la mémoire sont effectués au même moment, le pointeur mis à jour est disponible après que les instructions accèdent à la mémoire.

Le "Smooth Register Backup" cablé sur place peut être utilisé par les instructions sur certaines implémentations. Il peut y avoir des instructions qui sauvent ou restaurent des parties ou tout le jeu de registres à un endroit spécifique mais c'est une fonctionalité optionnelle.

2.10 Le F-CPU n'a pas de registre de code conditionnel

Ce n'est pas parce que nous ne les aimons pas mais ils génèrent certains problèmes lorsque le processeur monte en fréquence et en instructions par cycles : gérer quelques bits devient aussi complexe que la pile décrite ci-dessus.

La solution à ce problème est la mode classique RISC : un registre est soit à zéro ou pas. Un branchement ou une opération conditionnelle est exécutée si un registre est à zéro (ou pas). C'est pourquoi plusieurs conditions peuvent être créées, sans le besoin de gérer un jeu fixe de bits (par exemple, pendant les commutations de contexte). Nous n'utilisons pas les bits de prédiction comme ceux trouvés sur certaines autres architectures : nous n'en avons pas besoin, ni de leurs instructions spécifiques. Cela permet de garder l'ISA, le compilateur et le planificateur très simple.

Mais, comme expliqué plus tard, lire un registre est plutôt "lent" dans le FC0 et le temps d'attente peut ralentir un grand nombre d'instructions usuelles. La solution est de ne pas les lire mais une copie "cache" des attributs nécessaires. Comme décrit au-dessus pour les bits "d'attribut" ou de "propriété" des registres pour l'étage virgule flottante, chaque registre possède un bit d'attribut qui est regénéré à chaque fois que le registre est écrit. Lors de l'accès au registre, la valeur qui est présente sur le bus d'écriture est contrôlée à 0 et un bit sur 63, correspondant au registre sur lequel nous écrivons est mis à 0 ou 1 en fonction du résultat. Ce jeu de porte à "verrou transparent" est positionné proche du décodeur d'instructions de manière à réduire le temps d'attente des instructions conditionnelles. Comme il est regénéré à chaque écriture, il n'y a pas de besoin de le sauver ou de le restituer pendant les commutations de contexte et ils n'y a pas d'édition de cohérence.

Il n'y a pas d'indicateur de retenue non plus. Les additions avec retenue sont effectuées à travers une forme spéciale d'instruction qui écrit la retenue dans un répertoire à usage général proche du registre de résultat. Ceci évite les problèmes de cohérence avec les commutations de contexte et permet l'utilisation de la retenue avec des instructions SIMD : cela permet la croissance et la sécurité pour le planificateur de pipeline.

2.11 Le F-CPU est "sans endian"

Ni big endian seul ou little endian seul ne satisfait tout le monde. Pour résoudre le problème, il existe un bit endian dans les instructions charger/stocker. Le processeur lui-même n'est pas spécifié pour un des deux endian (bien que par la nature SIMD du CPU, il est préférable d'utiliser little endian) et les instructions elle-même ne sont pas de sujet de ce débat. Le choix est à la volonté de l'utilisateur final. Pour de plus amples informations, lisez les discussions dans le chapitre "5.5.5 indicateur Endian" ou la FAQ Endian sur `http://www.rdrop/com/~cary/html/endian_faq.html`.

2.12 Le F-CPU utilise de la mémoire paginnée

Ceci fournit à l'utilisateur une grande mémoire virtuelle privée, linéaire pour toutes les tâches à exécuter. La protection de page est simple car c'est le logiciel qui va protéger les espaces de mémoire des tâches. Le système VM n'est pas encore complètement défini mais nous avons déjà des caractéristiques préliminaires :

- Les pages vont avoir plusieurs tailles, par exemple 4KB, 32KB, 256KB et 2048KB, de manière à réduire le nombre de descripteur de pages (pression des routines malloc !). Quelques descripteurs de pages de blocs de taille arbitraire (puissance de deux) seront aussi nécessaires pour gérer les pages plus grandes que 2MB (si vous avez 128MB de RAM dans votre ordinateur, vous aurez besoin de 64 x 2MB descripteurs, plus que le CPU ne peut en contenir embarqués). La granularité proposée pour ces blocs plus grands est de 128KB (adresse de base et taille, dans un système "protégé" et le CPU peut stocker deux descripteurs de telles pages en embarqué.
- Les pages peuvent être compressées à la volée lorsqu'elles sont envoyées vers les disques durs (spécialement pour les grandes pages). C'est un élément optionel car cela ne diminue pas le temps d'attente du disque dur mais cela peut optimiser la bande passante du bus de mémoire principal. Nous devons trouver un bon compresseur de même qu'un bon compromis SW/HW pour le système de compression.
- On pourrait réserver un peu d'espace dans la hiérarchie de la mémoire cache pour garder les pages les plus importantes. Le noyau sera responsable de ce choix.
- Les indicateurs de cache et ceux de lecture/écriture des pages seront utilisés dans les premières implémentations pour assurer la cohérence du cache dans des systèmes multi-CPU avec les fonctions de l'OS et les erreurs, plutôt que d'utiliser du matériel dédié. Donc, non seulement la mémoire paginée est utilisée pour proteger les tâches et fournir plus de mémoire visible mais elle sert aussi comme remplacement "logiciel" du protocole MESI dans une architecture de Mémoire à Accès Non Uniforme (NUMA).
- Les TLBs internes sont contrôlés par le logiciel au travers d'un jeu de Registres Spéciaux. Aucun microcode ou matériel n'est prévu pour aider la recherche de l'entrée d'une table de page en mémoire. Une exception OS est déclenchée lorsqu'une tâche émet une instruction qui accède à une zone mémoire qui n'est pas dans la Table des Pages interne (TLB). Comme il n'y aura probablement pas plus de quatre ou huit entrées de 4KB, 32KB, 256KB ou 2048KB chaque(32 descripteurs partagés pour les données et les instructions dans les premières implémentations), ((((((((((l'OS PTE miss trap handler)))))))))) doit être très précautionneusement codé. *Vous rappelez-vous de la devise ? "coder avec attention a toujours payé !"*.

Attention, ces caractéristiques sont préliminaires. Des détails vont surement bientôt évoluer.

Il apparaît clairement que la partie la plus critique du mécanisme de protection est le TLB. Il y a quelques mécanismes annexes mais le TLB est le "gardien du pont" pour les cas les plus communs. Il doit être très bien conçu et fournir des mécanismes utiles qui aident efficacement la gestion de la mémoire et de l'allocation des blocs. Par exemple, l'entrée TLB contient des champs additionnels comme le VMID (il est utilisé pour réduire le gaspillage) et des bits d'usage (8 compteurs 2 bits saturés qui mesurent l'usage de la mémoire réelle et de l'activité à l'intérieur d'une page). Les deux champs ont 16 bits de largeur et aident le noyau à améliorer l'allocation de la mémoire.

De manière à garder une bonne performance générale, le projet compte sur un OS efficace. Les clones LINUX sont vraissemblablement les systèmes les mieux placés car ils bénéficient de toutes les nouvelles recherches et avancées dans les technologies des noyaux, les planificateurs de tâches astucieux et les algorithmes de remplacement de page efficaces. Le choix d'une stratégie de remplacement logicielle de page ne garde pas seulement une complexité matérielle faible mais il permet aussi au système de bénéficier des avancées algorithmiques futures. Si ces éléments ne sont pas utilisés, il n'y aura pas de performance matérielle...

2.13 Le F-CPU stocke l'état de la tâche dans des Blocs de Mémoire de Contexte (Context MemoryBlocks (CMB))

Ce sont des structures très importantes pour l'OS car le mécanisme SRB empèche les traitements de voir les tâches interrompues pour des raisons de cohérence. L'OS dialoguera avec ces blocs de manière à positionner ou modifier les propriétés et les droits d'accès d'une tâche, lire ses registres ou interpréter un appel système. Un bloc de mémoire de contexte doit stocker toutes les données internes à une tâche de manière à les stocker et les retrouver complètement. L'endian du CMB n'est pas défini.

Le CMB contient l'état de toute tâche d'une telle manière qu'elle peut être arrêtée et redémarrée. Il est utilisé pour le débuggage de même que le multi-tâche. Chaque instruction F-CPU est *atomique* et ne peut pas être divisée donc nous ne stockons pas de résultat partiel ou d'état de pipeline temporaire dans le CMB.

Un Context Memory Block est divisé en un nombre variable de "slots" qui sont aussi large que le CPU peut le supporter (ie, 64 bits pour un CPU 64 bits). Chaque slot contient un registre individuel global ou spécial.

Les premiers 64 slots contiennent les contenus des registres normaux "généraux". Ils sont stockés et restaurés par le mécanisme Smooth Register Backup. Alors que R0 est cablé à 0, le slot correspondant (le premier) contient le pointeur d'instruction.

Le CMB contient les droits d'accès et les indicateurs de protection les plus importants. L'OS modifie les droits d'accès d'une tâche dans le CMB car il ne peut pas le faire directement dans les registres spéciaux (qui a ce moment, stockent les propriétés de l'OS...). Les indicateurs les plus importants sont stockés dans le Machine Status Register ("MSR") : les indicateurs de taille, le VMID, les indicateurs de capabilité...

Le CMB contient le pointeur de la table de pagination des tâches (lorsque la pagination est validée). Cette table de pagination peut être stockée à la fin du CMB si l'OS en décide ainsi.

Les deux dernier slots sont utilisés pour le multitâche et le débuggage, en conjonction avec le mécanisme SRB : les slots "next" et "time slice". Le slot "next" est un pointeur vers un autre CMB ; la tâche stockée dans le CMB peut commuter automatiquement vers une nouvelle tâche, dont le CMB est pointé par le champ "next". Le "time slice" stocke le nombre de cycles d'horloge qu'une tâche peut exécuter avant de commuter automatiquement vers la tâche "next".

Cette description n'est pas exhaustive et le nombre de slots CMB augmentera dans le futur, avec les besoins et l'évolution des architectures. Un certain nombre de Registres Spéciaux sont dédiés à la gestion du CMB.

2.14 Le F-CPU peut utiliser les CMBs pour les tâches à simple étape

Pour utiliser le CMB lors d'une simple étape de tâche, aucun composant spécial n'est nécessaire (excepté un cerveau) :

1. Installations des paramètres suivants dans le CMB de la tâche : "next" pointe vers le CMB propre au debugger et le "time slice" est mis à 1 (ou tout chiffre souhaité pour de multiples étapes).
2. Installer le registre spécial "next" au CMB de la tâche.

3. Exécuter une instruction RFE (retour d'une exeption).

Lorsque RFE est exécuté, le processeur commutera automatiquement vers la tâche dont le CMB est pointé par le registre spécial "next". Le processeur chargera alors le CMB du slot "next" dans le registre spécial "next", exécute les instructions et recommute (retourne) vers le debugger lorsque ce nombre expire. Le debugger peut alors analyser le contenu du CMB de la tâche, ses registres et les champs spéciaux.

Un indicateur dans le MSR est aussi dédié aux tâches à étape simple. Le CPU génère une erreur (((((((trap))))))) après avoir exécuté toute instruction lorsque l'indicateur est mis.

Pour d'autres cas que l'étape simple, le F-CPU fournira à l'utilisateur des traps sur des conditions spéciales et des évènements, comme l'implementation le permet (c'est plus dépendant de l'implementation et n'est pas encore défini).

2.15 Le F-CPU utilise un mécanisme de protection simple

Avant qu'un système plus sophistiqué ne soit développé, un schéma simple utilisateur/superviseur est une bonne manière de démarrer un CPU mais une protection basée sur les ressources (((((((refined))))))) permettra aux utilisateurs de créer un OS plus flexible, basé, par exemple sur une approche micro noyau.

Ce n'est pas "une bonne chose" d'utiliser une protection par niveaux en anneaux car quelques morceaux de logiciel, par exemple dans un OS micro noyau, sont dédiés à une certaine tâche et les anneaux n'isolent pas leurs fonctions correctement. D'un autre coté, une tâche dédiée au traitement de l'entrée de la table des pages (PTE) a raté seulement les besoins d'accéder aux Registres Spéciaux associés et le pilote du disque dur : s'il échoue, il n'y a pas de conséquences sur les autres tâches qui sont dédiées aux communications ou à la gestion de la mémoire, même s'ils sont "de confiance" : ils y a des tâches normales mais leurs indicateurs de propriété leur permet d'accéder à un certain matériel.

Nous donnons ici quelques uns des "bits de capabilité" qui sont associés à toute tâche :

* TLB_OFF positionné si les adresses ne doivent pas être validées par le TLB
* GET_CMB positionné si la tâche peut lire ou écrire son pointeur CMB
* GET_VM positionné si la tâche peut lire ou écrire son pointeur TLB manquant
* etc.

Ici, on voit comment la protection est implémentée par l'OS :

* La protection mémoire est assurée par manque de tâche TLB sur une base de page par page.
* Le traitement de manque de TLB est pointé par le SR TLB manquant qui n'est accédé que par les tâches avec les capacités correspondantes.
* Ces capabilités sont stockées dans le CMB qui réside à l'extérieur du champ visible des tâches non sécurisées et le pointeur CMB n'est pas accessible aux tâches n'ayant pas les bits de capabilité correspondants.
* *d'autres bits de capabilité apparaîtrons dans le futur.*

Une tâche utilisateur (non sécurisée) aura tous ses bits de capabilité mis à zéro, alors que le noyau les aura tous à un. Après un reset, tous les bits sont positionnés et le noyau permet à chaque tâche d'avoir plus ou moins de possibilités en effaçant ou en positionnant les bits correspondants lorsqu'il crée une tâche. Par exemple, une tâche sécurisée responsable de la gestion du VM n'aura que le bit GET_VM de positionné.

Troisième partie

Description générale du Coeur FCPU #0

Chapitre 1

A propos du coeur FC0

Ici, nous allons parler des caractéristiques spécifiques du FC0 ("F-CPU Core #0") et même si elles peuvent influencer la définition générale du F-CPU, elles peuvent être abandonnées dans le futur. C'est là où les ingénieurs matériels sont les plus impliqués.

1.1 Le FC0 est superpipeliné

Lors de la conception d'un processeur, une des premières questions est "quelle va être la granularité du pipeline ?". Ce n'est pas une étape critique pour les "processeurs pour jouer" ou pour les conceptions qui sont adaptées des processeurs existants mais le F1 n'est pas un jouet et il doit être très performant depuis le premier prototype... Pour le cas du F1, où le premier prototype sera probablement un FPGA ou un ASIC avec des millions de portes mais pas un processeur complètement personnalisé, la performance compte plus car le processus ne pourra pas se mesurer aux circuits existants. La performance compte toujours de toute manière mais dans notre cas, il y a un très fort handicap technologique. Nous avons besoin d'une technique qui atteint la même "vitesse" avec un technologie plus lente.

Donc l'équation est : *vitesse* = *technologie de siliciun* x *longueur critique des chemins*, ou *vitesse* = *vitesse d'un transitor* x *nombre de transistors*, donc avec des transistor lents, la seule voie pour tourner plus vite est de réduire le chemin critique (comme estimation approximative, car d'autres paramètres tels que la capacitance et l'infuence de la longueur des fils ne sont pas comptés). Donc maintenant, quelle est l'opération minimale que nous pouvons effectuer sans surcharger le circuit avec des flip-flops ?

La profondeur d'à peu près dix transistors est un compromis entre la fonctionnalité et l'atomicité. Nous pouvons créer des circuits qui ont autour de six portes logiques de profondeur ou ajouter des nombres huit bits. En plus de ceci, le nombre maximum d'entrées par porte est mis à 4 pour qu'il puisse être facile d'utiliser des bibliothèques existantes et des architectures FPGA. Une attention est portée pour avoir des "blocs incorporé" rapides et indépendants mais le on coté est qu'avec 6 portes logiques, nous ne pouvons pas faire des choses complexes, alors que des chemins de données plus longs donnent habituellement naissance à des problèmes complexes. Avec cette "limitation" en tête, nous limitons aussi la complexité et seules les connexions de voisin à voisin sont possibles. De plus, dès que les unités deviennent plus complexes, elles sont soit "parallélisées" (une grande table de correspondance peut être utilisée par exemple) ou "sérialisées" (en d'autres mots, pipeliné) il n'y a donc pas de besoin de ralentir le processeur ou d'utiliser des technologies asynchrones.

L'effet net de ce biais par rapport à une logique dont la granularité est extrèmement fine et les étages de pipeline est que même les additions deviennent "lentes" car elles ont besoin de plus de cycles qu'habituellement. Cette lenteur apparente est balancée par de plus hautes performances au travers du chevauchement des operations (pipelining) mais nécessite l'utilisation de techniques de codage habituellement trouvées dans les processeurs superscalaires (duplication des pointeurs, ((((((((((loop unrolling)))))))))) et l'entrelacement, etc.). Les étages étant plus courts, il y a plus d'étages de pipeline qu'habituellement, c'est pourquoi le FC0 peut être considéré comme superpipeliné. Mais c'est seulement un des aspects du projet et aujourd'hui, plusieurs processeurs sont aussi superpipelinés.

1.2 Le FC0 implémente un pipeline *Out Of Order Completion*

C'est une solution simple si nous voulons avoir plus de performance d'un pipeline à étage simple. Ce N'est PAS un schéma d'exécution superscalaire ou out-of-order (ou *étape* d'instruction OOO) mais

"l'adaptation" d'un CPU à pipeliné simple où les instructions sont *émises dans l'ordre*.

La raison fondamentale derrière ce choix est que toutes les instructions ne prennent pas réellement le même temps pour s'achever. Ce fait devient plus important dans le F-CPU car il est superpipeliné et une instruction courte sera pénalisée par des instructions plus longues qui vont alonger le pipeline. Par exemple, si nous voulons calibrer la longueur du pipeline sur une addition 64 bits, les opérations les plus longues comme les divisions, les multiplications ou les accès mémoire hors cache gèleront le pipeline complet ; d'un autre coté, les déplacements registres à registres simples ou les écritures simples d'une valeur immédiate vers un registre sera plus lent que réellement nécessaire. Ceci peut être fait sur les premiers processeurs mais pas sur les processeurs superpipelinés.

Regardons les instructions qui ont besoin d'être réalisées après l'étage de décodage :

cycles approximatifs	1	2	3	4
écrire imm to reg	écrire dest			
charger de la mémoire	lire adresse	accéder données : indeterminé	écrire dest	
écrire vers mémoire	lire adresses	donnée accède donnée		
opération logique	lire opérandes	opération	écrire résultat	
opération arithmétique	lire opérandes	opération1	opération2	écrire résultat
déplacer reg vers reg	lire source	écrire dest		

Nous pouvons aussi noter que les instructions successives peuvent être indépendantes, sans le besoin des résultats des instructions précédentes. La dernière remarque est qu'il n'ont pas tous besoin du même matériel. Nous pouvons faire quelques conclusions : toutes les instructions n'ont pas besoin de lire et écrire dans les registres ou calculer quelque chose, les instructions ne se réalisent pas à la même vitesse et quelques instructions peuvent être plus longues que les autres (par exemple, lire une zone mémoire hors cache, comparé à une simple opération logique). Nous avons besoin d'un pipeline à taille variable qui permette à plusieurs opérations d'être effectuées et qui finissent au même moment. Une manière de visualiser ceci est de considérer le pipeline comme "folded" ou "forked" comme un processeur superscalaire. Mais tout ceci consiste en trois choses successives et optionelles : lire les opérandes, les traiter et écrire le résultat.

- Lire les opérations n'est pas un problème car les trois registres peuvent avoir besoin d'être lus en un cycle . Ceci est limité par les instructions elles-même,
- Calculer est complètement pipeliné et indépendent car les unités spécialisées traitent les données,
- Ecrire les résultats est un peu plus complexe car plusieurs opérations peuvent être faites en même temps. Une opération à seul cycle (opération logique par exemple) se terminera au même moment que les opérations à deux cycles (arithmétique) qui ont été sorties pendant le cycle précédent.

Pour cette dernière raison, le jeu de registres a (au moins) deux bus d'écriture. Le FC0 émet plus d'une instruction par cycle et plusieurs instructions peuvent se terminer au même moment. Dans le cas où plus de deux valeurs peuvent être écrites en même temps, l'instruction la plus "ancienne" (la première sortie) a la priorité.

Ce type de coeur de processeur a l'avantage de ne pas ralentir ou de bloquer le programme complet lors de longues opérations si les données résultantes ne sont pas nécessaires avant que les opérations lentes soient terminées. Par exemple, une lecture mémoire peut entrainer des délais si elle n'est pas faite dans le cache mais cela n'empêchera pas les autres unités d'exécution de faire leur travail et d'écrire leur résultat dans le jeu de registres. Bien sûr, ceci met la pression sur le compilateur mais pas plus que pour les autres processeurs existants et le codage avec précaution a toujours payé de toute manière.

La différence entre la complétion OOO et l'exécution OOO est que l'exécution CPU OOO peut sortir les opérations out of order et a besoin d'une dernière unité appelée "unité de complétion" ou "unité (((((((retire))))))))" qui valide les opérations dans l'ordre du programme. Ceci nécessite un registre qui valide les opérations dans l'ordre du programme. Ceci nécessite aussi des registres "renommés" qui contiennent les résultats temporaires avant qu'ils soient validés comme bons par l'unité de complétion. Tous ces "éléments" peuvent être évités par les techniques décrites dans ce document et, contrairement aux processeurs à exécution OOO (comme les PowerPC et les coeurs P6) la performance pic est limitée par la taille de l'unité de complétion FIFO (ou le "ReOrdering Buffer", ROB) mais par le nombre de ports de registres.

1.3 Le FC0 utilise un tableau (scoreboard)

C'est la manière la plus simple de traiter la nature out-of-order du coeur. La manière dont cela fonctionne est très simple : *chaque registre possède un drapeau qui est positionné lorsque le résultat est actuellement traité et les instructions sont retardées jusqu'à ce qu'aucun drapeau ne soit positionné pour*

les registres utilisés en lecture et écriture. De cette manière, la stricte cohérence est assurée et aucune opération ne peut entrer en conflit avec une autre lors de l'étape de l'exécution : la vérification des conflits est faite à un point seulement.

Ces drapeaux ne sont pas exactement comme le bit "d'attribut" car ils ne sont pas directement accessibles par l'utilisateur mais ils ont le même comportement dynamique et ne sont ni sauvés ni restaurés. Ne se produisant pas souvent et n'étant pas critiques pour la performance, les situations d'écriture après écriture ne sont pas examinées par le planificateur. La simple règle de blocage d'une instruction à l'étage de décodage si au moins un des registres utilisé (lu ou écrit) n'est pas prêt est strictement respecté. Bien sûr, le Registre 0 qui est cablé à 0 est la seule exception et ne bloque rien.

Le tableau interragit avec le mécanisme "Smooth Register Backup" pour assurer la cohérence entre les commutations de tâches.

1.4 Le crossbar

Le FC0 utilise un crossbar entre le jeu de registres et les unités d'exécution car :
- C'est la manière la plus facile de "remplir" le pipeline,
- Il fournit un registre de bypass de bus "fourre tout" qui diminue le temps d'attente *entre* les instructions dépendantes,
- Il réduit le nombre de ports de registres.

A cause de son rôle, le crossbar (ou "Xbar" en court) est une partie centrale du FC0. Le jeu de registres est seulement écrit ou lu au travers de cet élément qui lui fournit virtuellement plus de dix ports. Il permet aux unités d'exécution de communiquer sans le besoin d'écrire et de lire les registres (dans le mode passant des registres, lorsque les opérations sont dépendantes), il fournit le registre cablé 'zéro' et les résultats sont contrôlés pour savoir s'ils sont à zéro avec deux ports additionnels.

Le Xbar étend les ports d'écriture et de lecture du jeu de registre, en créant des bus "verticaux" (voir la figure 2.1) et chaque bus vertical est connecté à un des ports des entrées et de sortie de chaque unité d'exécution avec des bus "horizontaux". Il effectue aussi quelques formatages de tailles (octet, mot, etc) pour les valeurs immédiates arrivant du décodeur d'instruction. A cause du nombre relativement haut de ports, le crossbar utilise beaucoup de surface et de transistors. Il nécessite un cycle pour lui pour laisser les données circuler sur son entière longueur et le but de dix équivalents transistors est prêt d'être atteint rapidement à cause et du compte de transistor et de la longueur des fils. C'est pourquoi, accéder un registre prend deux cycles à partir du moment où le nombre du registre a été décodé : un cycle pour le jeu de registre et un autre pour le Xbar. Mais lorsque des instructions consécutives sont dépendantes le résultat qui sera écrit dans un registre est présent dans le Xbar et peut être utilisé pendant le cycle suivant pour l'opération suivante ("bypass de registre").

Ceci peut être résumé dans le dessin suivant :

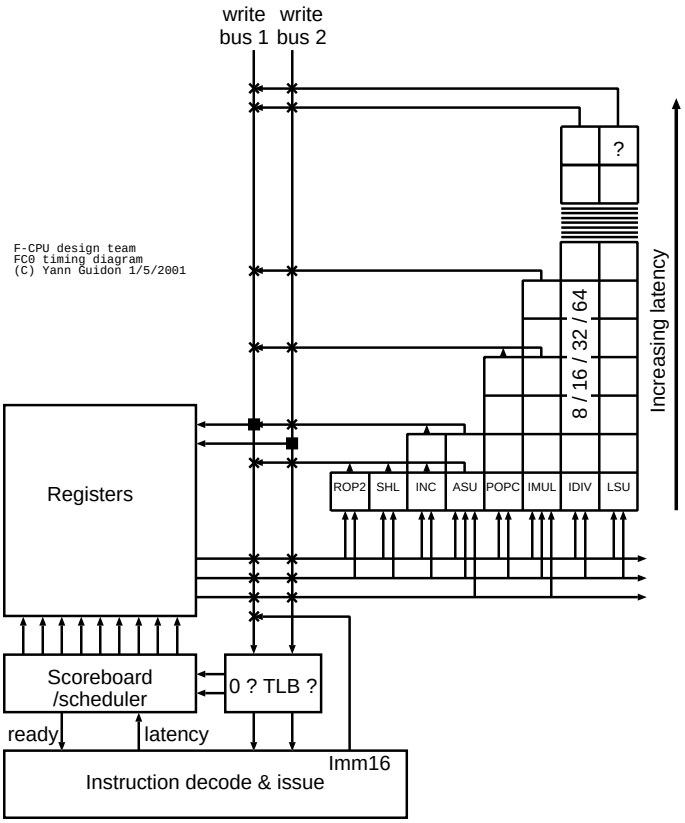


FIG. 1.1 – The pipeline is folded around the Xbar

Chapitre 2

Evolution du FC0

Discussions après discussions, le FC0 a pris une forme qui le rend unique. Etant un changement graduel et qu'il n'y a pas qu'une vue sur la structure du processeur, il y eu plusieurs dessins qui montrent l'organisation interne du circuit.

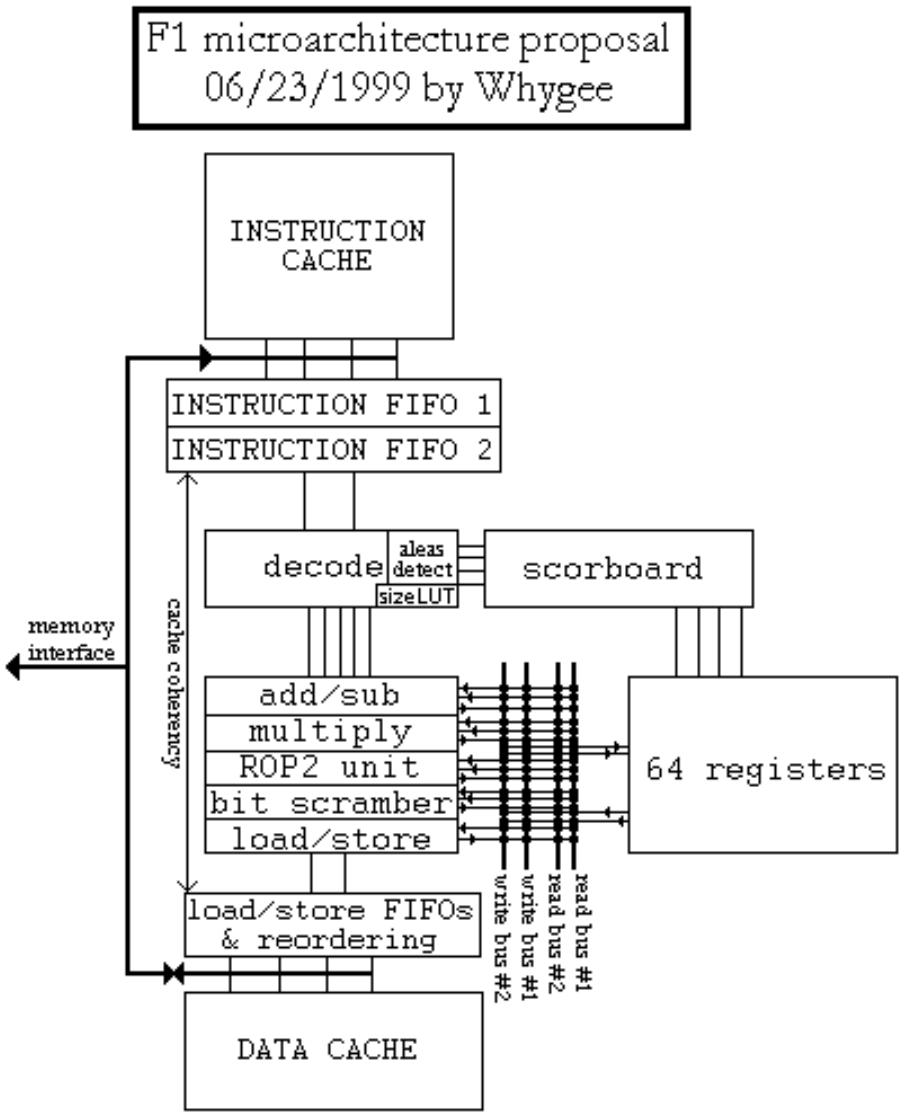


Fig. 2.1 – La première proposition de circuit F-CPU

La figure 2.1 est le premier dessin qui montre les formes générales du FC0, des points de vue du schéma, fonctionnel et de l'implémentation. A ce moment, le Xbar ne comptait pas pour un cycle d'horloge complet dans le pipeline. La hiérarchie mémoire n'était pas conue et consistait en des "unités" vides. Par contre le pipeline d'exécution était en grande partie déterminé et n'a pas beaucoup changé.

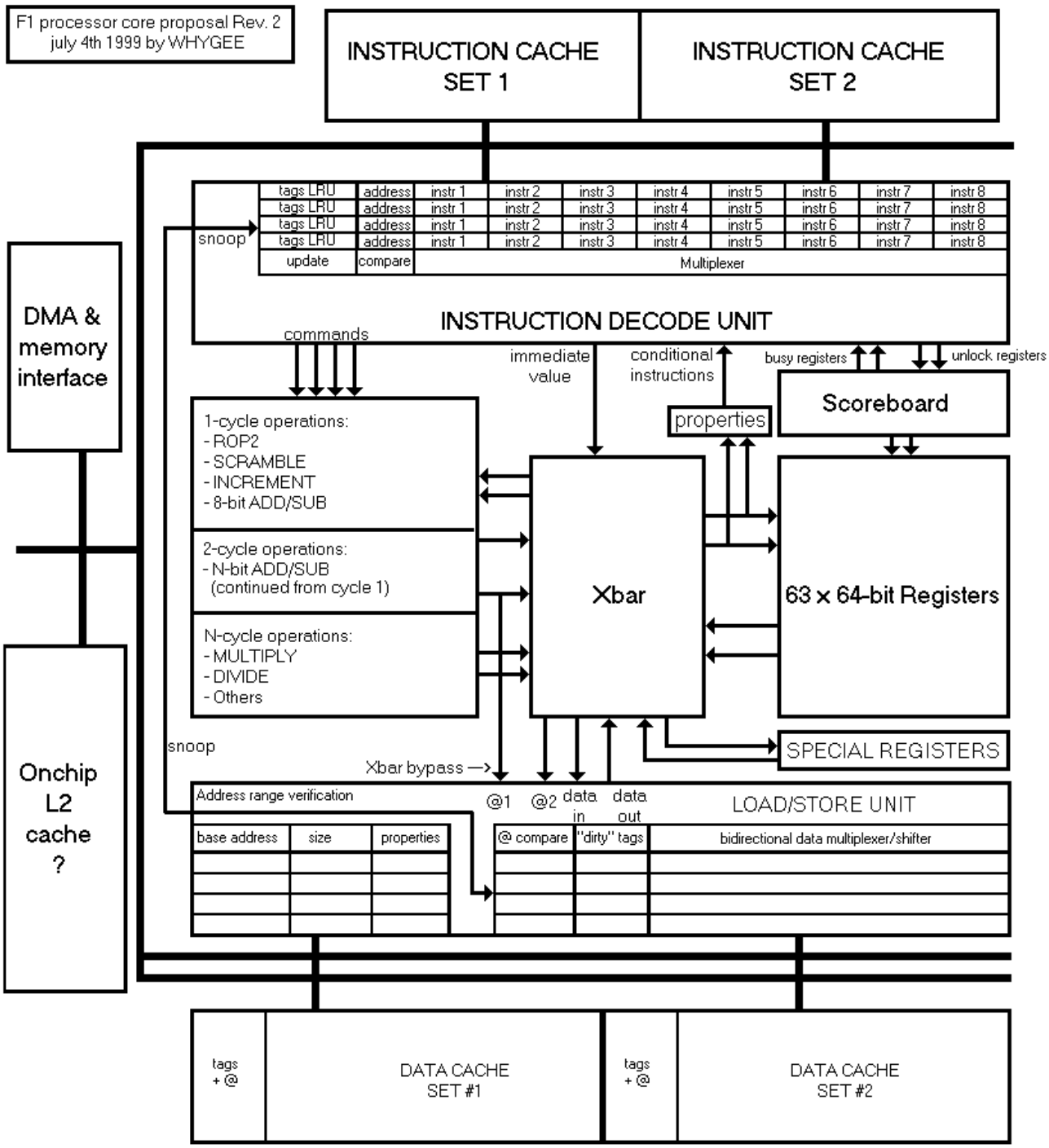


FIG. 2.2 – Une première tentative, plus précise, de la description du F-CPU

La figure 2.2 montre quelle architecture auraient les unités qui accédent à la mémoire. Elles sont encore aux extrémités du circuit et nécessitent de très long fils pour dbusquer les conflits d'accès de données/instructions. Les unités mémoire sont tout de même détaillées et consistent en plusieurs buffers cache linéaires. Un élément curieux est que les "barrières" des adresses (qui stockent les adresses de base et la taille limite des blocs qu'une tâche est autorisée à accéder) sont à l'intérieur des unités de mémoire et les TLBs sont maintenant à l'extérieur des unités. Le Xbar prend maintenant un cycle d'horloge complet et est considéré comme une unité complète, le pipeline d'exécution est amélioré. en fonction des discussions, le jeu de registres est resté à deux ports de lecture et deux d'écriture, le troisième port de lecture a été accepté plus tard.

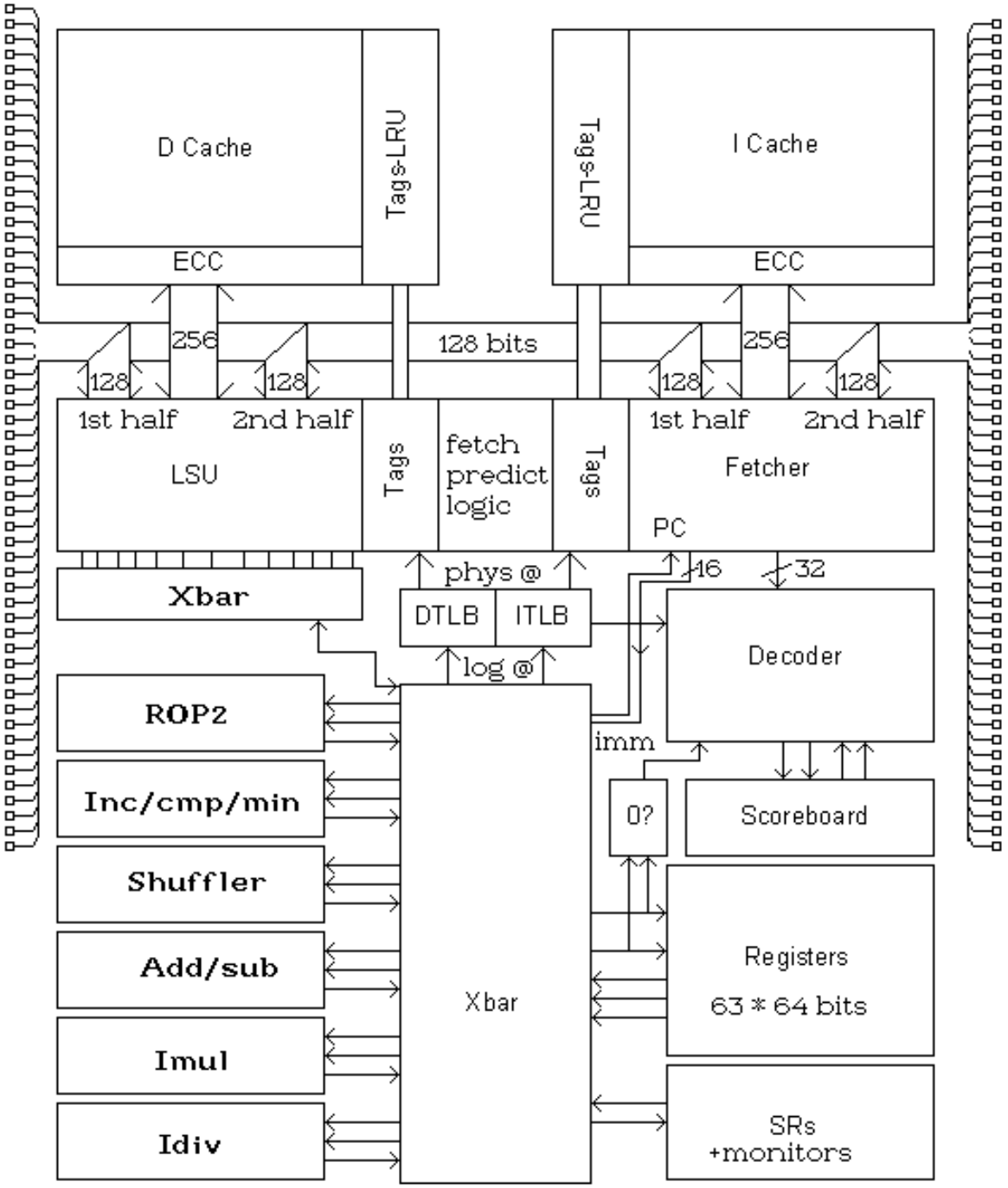


FIG. 2.3 – Une troisième description du F-CPU

La figure 2.3 montre le statut actuel du FC0 comme il est vu pour le F1. Les unités mémoire ont été rassemblées pour que les fils qui pilotent les lignes d'adresses et de données à l'extérieur du circuit aient une longueur minimale. Ils sont positionnés symétriquement pour que les marqueurs des buffers cache linéaires puissent être tous comparés dans une seule unité simple qui décide et planifie les accès mémoire. Les TLB données et instructions sont séparés des unités mémoire car ils forment une partie du pipeline et doivent être placé près de l'unité de décodage de manière à signaler un pointeur invalide dès que possible.

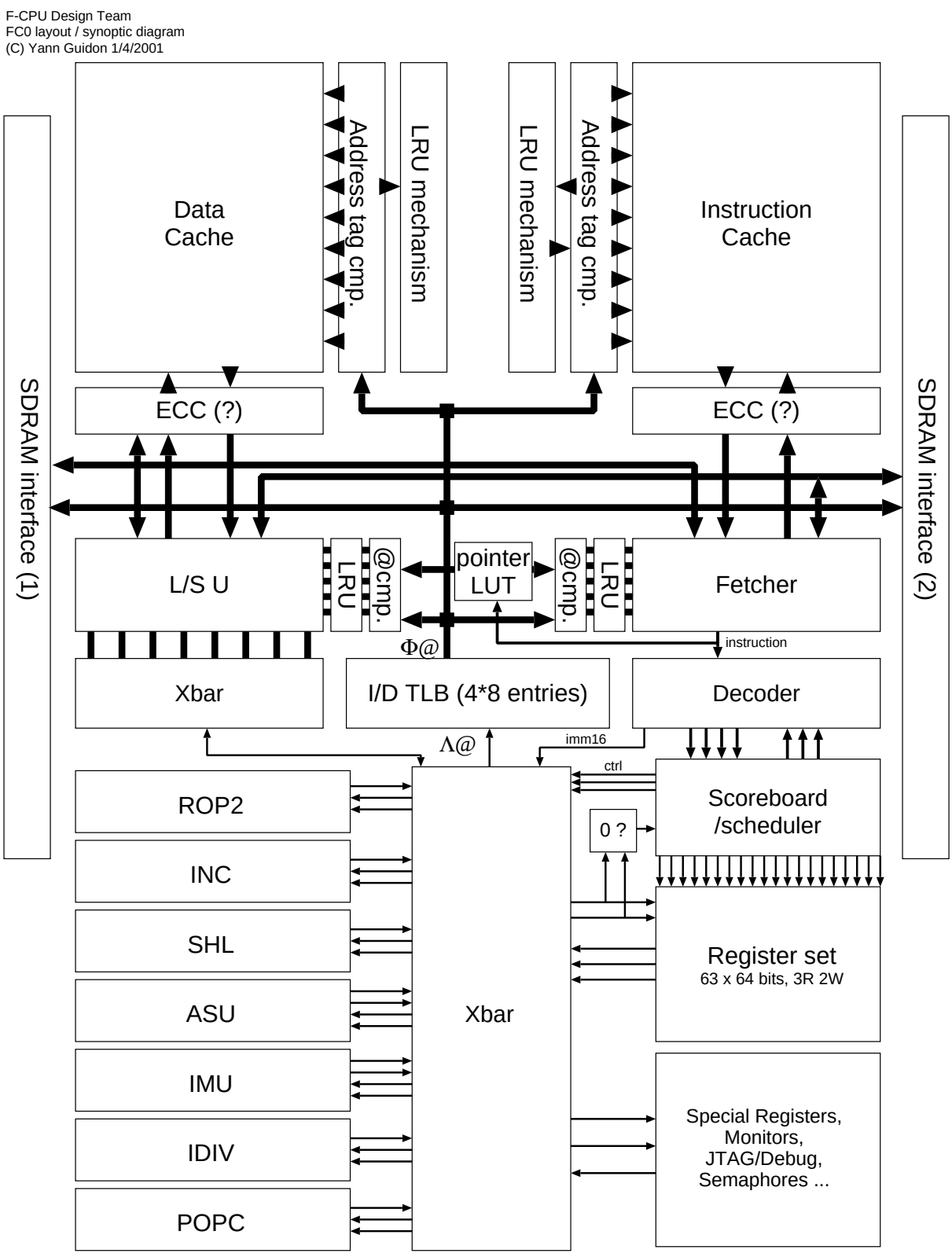


FIG. 2.4 – Le diagramme actuel du F-CPU

La figure 2.4 est la dernière mise à jour : le bus de donnée externe a été séparé en 2 bus 64 bits de SDRAM, l'unité POPCOUNT a été ajoutée et le système de mémoire (TLB/LUT/cache etc.) est encore plus précis. Nous pouvons voir que la forme générale n'a pas changée mais a été affinée.

Chapitre 3

Les Unités d'exécution du FC0

Pour des facilités de developpement et d'augmentation de taille, pour donner quelques raisons, les Unités d'Exécution (EUs) sont comme des briques de LEGO qui ajoutent des nouvelles capacités de traitement au processeur. Comme le coeur en son entier, elles sont conçues avec un processus complètement personnalisé en tête mais peuvent être implémentées avec des bibliothèques (si elles ont les fonctions correspondantes) ou en cellules FPGA ou toute technologie alien qui tomberait du ciel...

Ici sont décrites les EUs minimales nécessaires qui ont été envisagées jusqu'ici. A leur venue, plusieurs unités peuvent fournir la même fonction (comme : décalage à gauche de un est comme multiplier par deux ou ajouter un nombre à lui-même) donc l'habitude la plus sage est de contrôler quelle unité fait quoi et en combien de cycles d'horloges, de manière à prendre la meilleure instruction pour l'opération souhaitée dans chaque contexte. Le nombre de transistors n'a pas été pris sérieusement en considération car plus d'attention a été portée pour réduire le chemin critique au minimum possible.

A cause de leurs différents temps d'attente et leurs particularités, les Unités d'exécution n'ont pas été formatées dans un "ALU tout en un". Nous pouvons aussi prendre une unité et y penser sans se préoccuper des unités alentour. De cette manière, nous voyons que la conception du matériel fournit de nouvelles opérations qui peuvent être utilisées dans le jeu d'instruction. Lorsque le matériel est en place, seules quelques portes logiques additionnelles fournissent des opérations utiles qui peuvent économiser plusieurs instructions dans les logiciels d'application et accélèrent quelques algorithmes critiques sans trop de dépassement.

3.1 L'unité "logique" (ROP2)

C'est "l'unité logique" classique. son but est de calculer les opérations bit à bit. Grâce à sa simplicité, elle a un cycle de temps d'attente et est parmi les unités les plus rapides.

Maintenant, quelles opérations va-t-elle exécuter ? Avec deux entrées, il y a 2^{2^2}=16 opérations possibles, parmi lesquelles 8 sont uniques et utiles :

A :	0	0	1	1	
B :	0	1	0	1	
	00	01	10	11	Fonction
	0	0	0	0	CLEAR (mis 0) : équiv. à mov res, reg0
	0	0	0	1	A AND B
	0	0	1	0	A AND /B
	0	0	1	1	A (ne fait rien)
	0	1	0	0	/A AND B (identique à A AND /B ci-dessus)
	0	1	0	1	B (ne fait rien)
	0	1	1	0	A XOR B
	0	1	1	1	A OR B
	1	0	0	0	A NOR B (equiv. à NOT [A OR B])
	1	0	0	1	NOT (A XOR B)
	1	0	1	0	NOT B (ne fait presque rien)
	1	0	1	1	A OR /B (equiv. à NOT [/A AND B])
	1	1	0	0	NOT A (ne fait presque rien)
	1	1	0	1	/A OR B (identique à A OR /B]
	1	1	1	0	A NAND B (equiv. à NOT [A AND B])
	1	1	1	1	SET à 1 (-1)

Quelques instructions sont dupliquées (si nous ajoutons la commutativité des opérandes), d'autres ne sont pas des opérations à 2 opérandes "réelles" (il existe des opérations à 1 et 0 opérandes). Nous pouvons inclure directement 4 fonctions sur bits dans les instructions mais nous avons besoin de place pour les instructions "combinées" ; nous pouvons donc sauver un bit avec l'utilisation d'instructions "condensées". Nous sélectionnons les 8 opérations à 2 opérandes et nous créons une nouvelle table. Le décodeur peut donc éviter de lire les registres de sources non nécessaires. Pour les instructions ROP2, les trois fonctions sur bits sont décodées avec une petite table de recherche cablée dans le décodeur comme suit :

instruction	code réel	Fonction	Nom symbolique
000	0001	A AND B	AND
001	0010	A AND /B	ANDN
010	0110	A XOR B	XOR
111	0111	A OR B	OR
100	1000	A NOR B	NOR
101	1001	A XNOR B	XNOR
110	1011	A OR /B	ORN
111	1110	A NAND B	NAND

Le matériel nécessaire pour traiter cette fonction est plutôt économique :

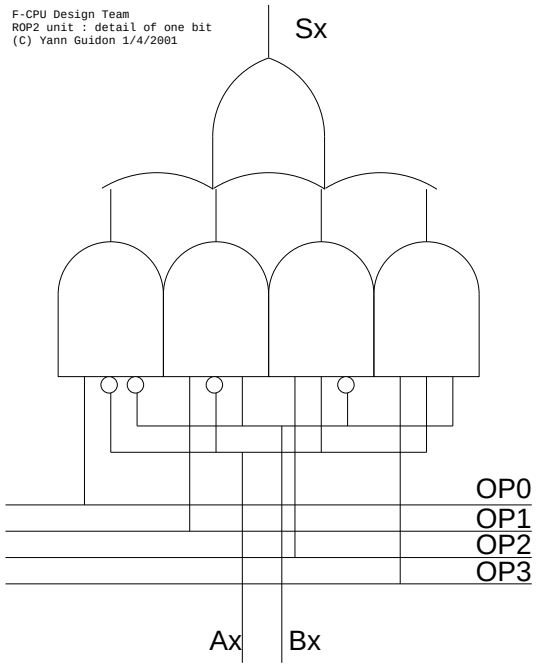


Fig. 3.1 – Détail de l'unité ROP2

Il y a probablement quelques autres détails techniques sur lesquels nous pourrions discuter mais ils sont trop dépendants de la technologie (signal "arbre" sur le bus d'opérations par exemple). C'est l'élément le plus directement stratégique du processeur.

A cause du chemin de donnée critique très court de cette unité, nous pouvons ajouter quelques fonctionnalités (simples) : appelons-là la fonction "combine". Alors que ROP2 est bit à bit, la "combine" effectue les AND ou OR logiques de chaque résultat ROP2 pour chaque paquet SIMD (de taille variable) d'un mot. Combiné avec la fonction ROP2, il est possible d'effectuer des masques complexes et des mouvements de bits avec peu d'instructions et des besoins moindre pour les décalages. Remarque : à cause du grand nombre d'entrées, seule les combines 8 bits sont actuellement implémentées.

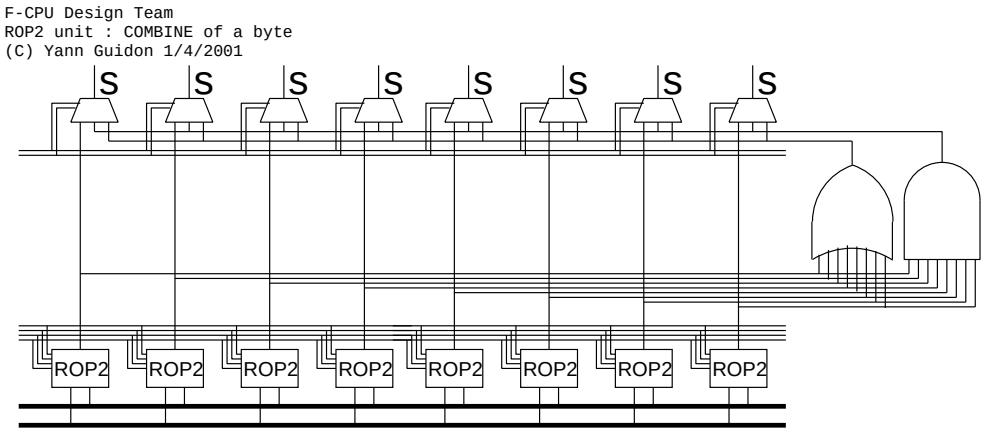


FIG. 3.2 – Description de la fonction COMBINE au début de ROP2 pour un paquet SIMD de la largeur d'un octet

VHDL : voir le répertoire /vhdl/eu_rop2 dans le paquetage F-CPU.

3.2 L'unité "bit scrambling" (SHL)

Le but est d'avoir une unité de décalage à un cycle qui peut aussi faire d'autres choses. Contrairement à l'unité ROP2, la fonction principale n'est pas de changer la valeur des bits de données présents sur l'entrée mais de changer la position de ces bits. C'est pourquoi, le décalage et la rotation sont les seuls exemples des buts intentionnels de cette unité quelquefois appelée "shuffling" : l'extraction et l'insertion de champs de bits, de même que l'inversion ou le test de bits ou d'octets sont des exemples de ce que le matériel est supposé effectuer.

Il y a néanmoins un problème : F-CPU sera une processeur 64 bits et un ((((((((((barrel shifter)))))))))) est une unité $O(\log_2(n))$, ce qui est très près de la granularité du pipeline. Un tableau de décalage (une sorte de tableau de transistors) sera nécessaire pour pour être à O(1), au détriment du nombre de transistors et probablement de plus de charge sur les transistors mais c'est la seule solution si nous voulons décaler 128, 256 ou 512 bits en un seul cycle de pipeline à 10 transistors. Lors du prototypage, nous pouvons utiliser du matériel pré-défini mais une production de masse nécessitera quelque chose comme un réseau Omega de petits shufflers.

Cette unité effectuera aussi les opérations SIMD spécifiques comme l'expansion de mot SIMD et le mélange. Une petite unité logique au bout du chemin de donnée critique peut effectuer des opérations sur les bits (test, positionnement, effacement, changement) si suffisamment de portes sont laissées dans le chemin des données critiques.

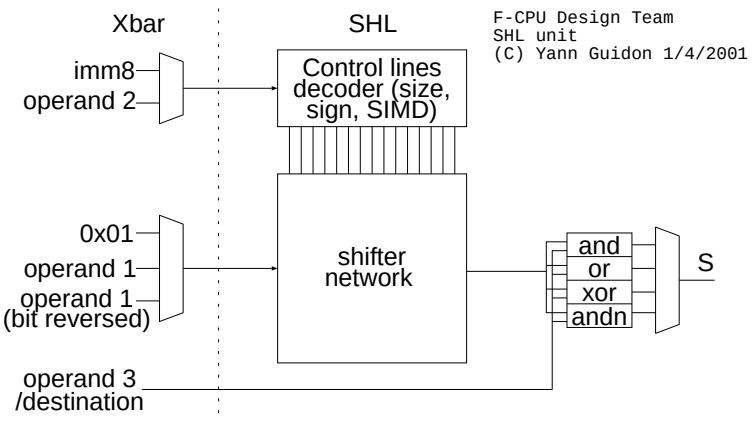


FIG. 3.3 – Vue générale de l'unité Scrambling

VHDL : voir le répertoire /vhdl/eu_shl dans le paquetage F-CPU.

3.3 L'unité "d'incrémentation"

C'est peut être l'unité la plus curieuse car elle n'est pas habituellement trouvée dans les CPUs. La raison de cette unité dédiée est simple : beaucoup d'instructions ajoutent ou soustraient 1, dans les boucles par exemple. C'est un travail non nécessaire pour les additionneurs, si la seconde opérande est 1, donc nous le cablons et cela tourne plus vite. C'était la première idée.

La méthode pour incrémenter un nombre binaire n'est pas difficile à comprendre : vous scannez le nombre en démarrant par le LSB, en inversant chaque bit jusqu'à ce que vous trouviez un 0. Vous changez alors ce 0 en 1. C'est en fait un arbre dédié de propagation de la retenue avec des portes XOR à la sortie. L'arbre fait la même chose que "trouver le premier LSB positionné". Donc, on y va, on en a deux dans le jeu d'instruction. Dans certains cas, c'est très intéressant et il n'y a pas de surchage matérielle. Cela fait deux instructions : INC et LSB1.

Donc maintenant que nous incrémentons, nous pouvons aussi décrémenter : nous devons inverser chaque bit à l'entrée et à la sortie de l'unité. Ce matériel rapporté nous permet aussi de trouver le "LSB mis à zéro". Quatre instructions (ajouter DEC et LSB0). Nous pouvons aussi ajouter un inverseur de bit à l'entrée, de même que pour trouver aussi le MSB. Six instructions (ajouter MSB1 et MSB0 au jeu d'instruction et un inverseur de bits sur le Xbar).

Allons plus loin : ajoutons un multiplexeur au bout de l'incrementeur, qui est contrôlé par le bit de signe de la valeur d'entrée. Si le bit de signe est à 1, nous mettons l'entrée à -(n+1) (il y a un bit de jonglage à faire avec les inverseurs mais c'est simplement un "détail technique"). Avec cette unité, nous pouvons calculer la valeur absolue de nombres binaires en compléments à 2. Sept instructions (ajouter ABS). Maintenant que nous avons ces multiplexeurs à l'entrée et à la sortie de 'l'incrémenteur', nous pouvons encore faire plus de choses. L'incrémenteur est arbre binaire qui "trouve le premier" et nous pouvons donc l'utiliser pour comparer deux nombres. L'idée est simple, un nombre (positif) est plus grand qu'un autre si au moins de ses MSBs est positionné alors que le bit correspondant de l'autre nombre est à zéro : $0 > 1$, $11 > 10$ ...

Donc, il faut simplement XOR les deux nombre d'entrée, trouver le premier MSB positionné et AND le résultat avec un nombre d'entrée. Si le résultat est effacé, alors ce nombre est plus bas que l'autre et vice versa. Ceci fait huit instructions. Encore mieux, nous pouvons utiliser le multiplexeur de fin pour sélectionner une des valeurs d'entrée : nous pouvons avoir les instructions min et max, de même que les dérivées comme *"si reg1 > reg2 alors reg1=reg2"* (pour les graphiques, dans le (((((((coordinates clipping))))))) ou l'arithmétique saturée...). Nous pouvons avoir plus de dix instructions utiles avec cette simple unité à 1 cycle. Quelque unes sont très utiles car elles impliquent des branches conditionnelles réelles (et des arrêts pipeline ou des mauvaises préditions de branches...).

D'un point de vue purement abstrait, trouver le premier bit positionné est fait avec un "arbre binaire", alors la profondeur de l'unité est $O(\log_2(n))$ avec des "nodes" plutôt simples. C'est à peu près un cas d'école à concevoir. De toute manière, comme pour le tableau de shifter, il y aura quelques problèmes avec la profondeur de l'étage du pipeline, principalement pour les instructions compare et clip... Au moins, INC, DEC, ABS, NEG (et leurs variantes SIMD) sont possibles en pratique avec de fortes contraintes de temps.

Dans cette unité, je n'ai pas encore réglé le problème des données SIMD. Comparer les nombres signés est direct : nous devons simplement XOR le bit de signe de chaque morceau SIMD.

L'implémentation actuelle de l'unité INC, réalisant *inc, dec, neg* et *abs*, est inclue dans le chemin des données critiques avec 6 portes de profondeur. Elle est composée d'une première ligne de XORs, un arbre AND à trois portes de profondeur, une ligne de multiplexeurs et un dernier étage de Xors.

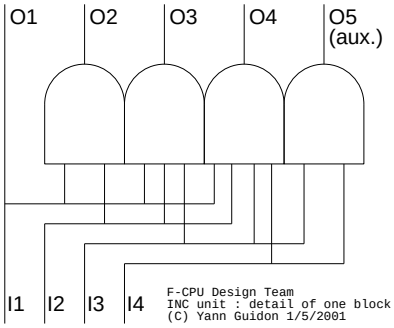


FIG. 3.4 – Description d'un bloc d'arbre AND

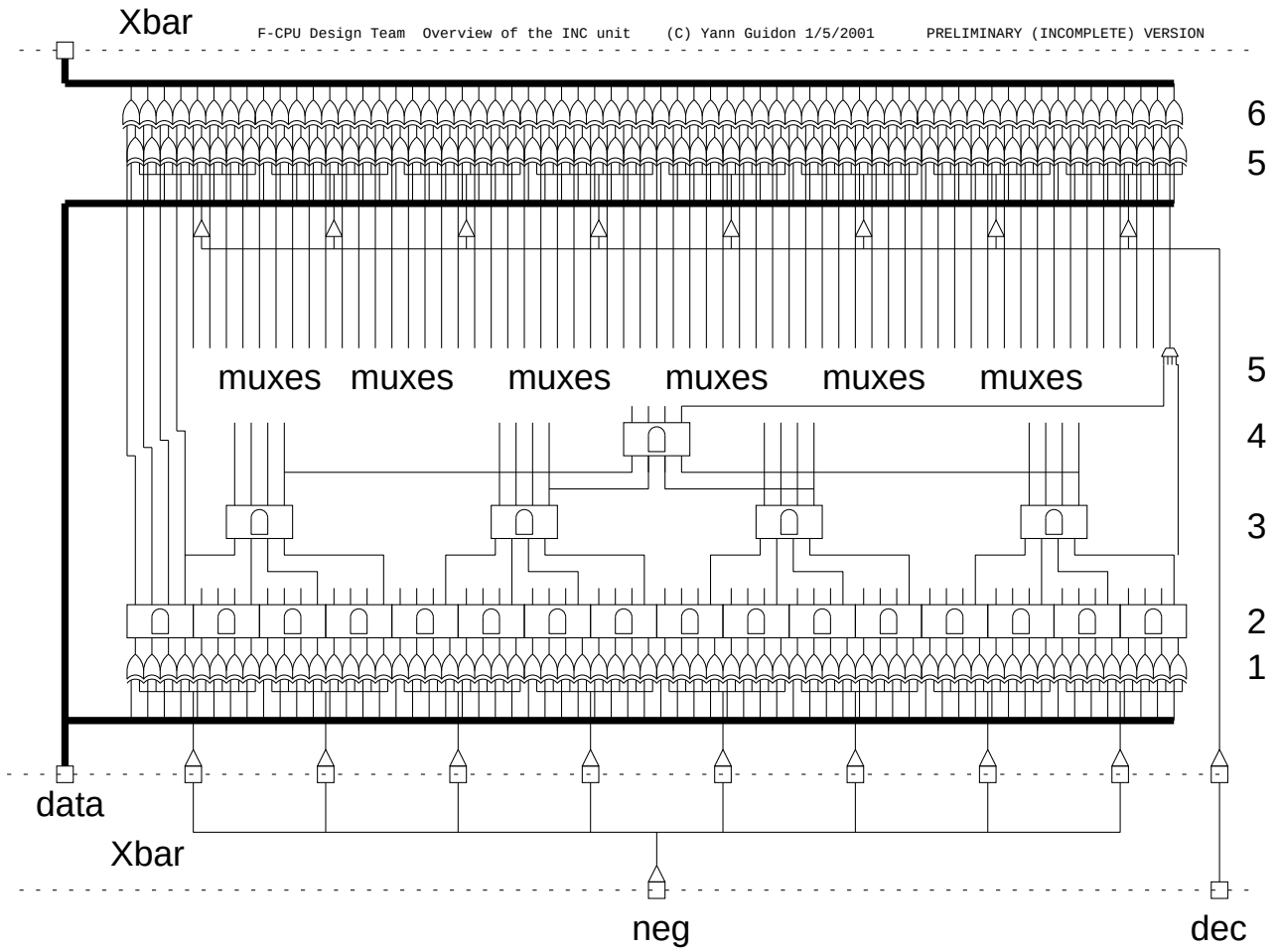


FIG. 3.5 – Vue générale de l'Unité d'Incrémentation (version préliminaire)

La sortie du dernier étage XOR pourrait remplir un autre étage pipeline qui effectuera les opérations restantes (LSBx, MSBx, min, max ...).

Vous pouvez remarquer que le cycle Xbar peut être utilisé pour amplifier un simple signal vers un grand nombre d'entrées. Le Xbar donne suffisamment de temps/porte pour compenser un aussi grand éventail de sortie.

VHDL : voir le répertoire /vhdl/eu_inc dans le paquetage F-CPU.

3.4 L'unité add/sub

Utilisant un additionneur à retenue, il a besoin d'à peu près deux cycles pour compléter une addition ou soustraction 64 bits : c'est un processus O(log_2(n)) avec quelques mécanismes un peu plus lourds que l'incrémenteur mais il calcule un add/sub 8 bits en un cycle. Par conséquent, les données 8 bits sur SIMDs rendent le traitement rapide (1 cycle au lieu de deux 2). Pour ces raisons, il sera difficile d'utiliser des éléments pré-synthétisés standard à cause de la profondeur variable et la nature SIMD de cette unité. La saturation (signée et non-signée) est souhaitée, avec un temps d'attente additionel possible d'un cycle.

VHDL : voir le répertoire /vhdl/eu_asu dans le paquetage F-CPU.

3.5 L'unité de multiplication d'entier

Ici, même remarque que pour l'additionneur. Il y a des contraintes SIMD et un pipeline à profondeur variable et granularité affinée (en fonction de la largeur de la donnée d'entrée). Il sera difficile de trouver ce type d'unité dans des bibliothèques pré-synthétisées. Les unités actuelles effectuent un MAC 64 bits en 6 cycles.

VHDL : voir le répertoire /vhdl/eu_imu dans le paquetage F-CPU.

3.6 L'unité de division d'entier

De même que pour le multiplicateur. Notez néanmoins que la division par zéro peut être interceptée au moment du décodage avec le drapeau de propriété "zéro". Nous pouvons déclencher une erreur ((((((((((trap)))))))))) sans sortir l'instruction. Une vieille unité soustraction/décalage peut être suffisante car ce n'est pas souvent utilisé. Si des divisions sont nécessaires, la méthode Newton-Raphson peut être utilisée.

VHDL : voir le répertoire /vhdl/eu_idu dans le paquetage F-CPU.

3.7 L'unité Chargement/Stockage (L/SU)

C'est un cas très spécial car aucun calcul réel n'est effectué. Le temps d'attente est complètement inconnu au moment de la compilation et il y a un problème de protection de la mémoire. Si la protection de la mémoire est effectuée par d'autres mécanismes, l'unité L/SU est simplement un grand buffer cache avec un crossbar pour effectuer la sélection de mot/endian. Notez que sa structure est identique à celle de l'unité de traitement d'instructions : c'est un mirroir avec une granularité différente.

Lorsqu'il n'y a pas de manque du cache ou de buffer à envoyer, la donnée peut être envoyée directement ou lue à partir du buffer au travers du crossbar L/SU et ensuite envoyée au Xbar principal. Dans un cas idéal, il n'y a pas de temps d'attente pour les écritures mémoires et un cycle pour les lectures. La logique du traitement de la mémoire tente de garder les buffers pleins lorsque des accès contigus sont effectués. Un double buffer (avec une paire de buffers linéaires) peut cacher le temps d'attente mémoire dans une certaine mesure.

Le buffer mémoire peut "cacher" huit lignes de cache (le nombre de lignes peut varier avec les implémentations). Il communique avec le bus de données de la mémoire externe, la mémoire de donnée cache et le Xbar principal. Ceci réduit les temps d'attente lors de manquements de la mémoire cache et simplifie l'organisation du cache mémoire car le cache L1 ne communique pas directement avec la mémoire externe : le buffer mémoire (L/SU) est utilisé pour séparer le grand cache linéaire en petits éléments qui peuvent être envoyés à l'interface de mémoire. Non seulement le L/SU stocke les données mais il joue un rôle dans la hiérarchier mémoire, dans les cycles de remplacement du cache et dans la cohérence du cache dans une interface multi-bus avec un jeu limité de buffers qui sont utilisés pour plusieurs fonctions.

VHDL : voir le répertoire /vhdl/eu_lsu dans le paquetage F-CPU.

3.8 (((((((Population count))))))) / Single Error Correction (POPC)

C'est une unité spéciale optionnelle multicycle qui traite des fonction SEC et POPC.

L'instruction POPC effectue aussi des soustractions saturées avec un résultat sur 6 bits (voyez la description de l'instruction popc dans la partie 6).

Les éléments SIMD ont une largeur de base de 64 bits mais rien n'empêche le concepteur d'avoir d'autres granularités.

VHDL : voir le répertoire /vhdl/eu_popc dans le paquetage F-CPU.

3.9 Autres unités

Les nombres en virgule flottante n'ont pas été traités car nous préférons d'abord avoir quelque chose qui fonctionne correctement dans le domaine des entiers, nous ajouterons le matériel et les instructions FP plus tard. Le cas des exceptions mathématiques sera probablement géré avec le même type de mécanisme que le drapeau de propriété "zéro", donc aucune erreur ne cassera le flux du pipeline d'exécution.

Une manière "économique" pour éviter l'utilisation de nombres en virgule flottante est l'utilisation de la base de nombre logarithmique (LNS). De récents travaux ont réussi à faire un additionneur 32 bits logarithmique avec une vitesse et une utilisation de l'espace silicium décente. Toutes les autres opérations (SQRT, SQR, multiplication, division...) peuvent être faites par du matériel existant (peut être à modifier légèrement pour le MSB). Les conversions entre les entiers et les nombres logs sera une tâche logicielle lourde, tant qu'aucun matériel n'existera. Une coopération avec d'autres équipes de recherche est encouragée.

Lorsque le matériel FP sera disponible, seules les unités add/sub et multiplications seront d'abord implémentées. Tout autre opération mathématique (incluant la division) sera calculé avec un algorithme d'approximation Newton-Raphson en logiciel. Une troisième unité fournira les ((((((((((((("seed")))))))))))))
à partir des tables cablées en ROM.

3.10 Extensions et croissance

Si vous voulez ajouter votre propre Unité d'Exécution personnalisée au F-CPU, c'est assez simple : vous devrez d'abord préparer la carte du Jeu d'Instruction et le décodeur pour que les instructions nécessaires soient faciles à décoder et qu'elles ne soient pas en conflit avec d'autres instructions. Alors, vous aurez à vous assurer que le planificateur et que les exceptions ne compromettent la structure de l'unité décodeur (voir les discussions dans les parties suivantes). Finallement, vous "pluggez" votre unité sur port nouvellement créé du Xbar.

Selon votre technologie cile, vous pouvez ajouter un nombre indéterminé de nouvelles unités : l'architecture FC0 ne limite pas le nombre physique d'unités d'exécutions. Les limites physiques sont néanmoins importantes et le Xbar ne peut pas être étendu sans fin : le but de la conception (chemin de donnée critique à 6 portes avec 4 entrée max par porte) doit aussi être respecté.

Les largeurs de données sont aussi les paramètres qui jouent en faveur du FC0 et le F-CPU en général : L'extension de la largeur des registres ou de la largeur des éléments permet aux ingénieurs d'augmenter la taille de la conception facilement. Encore une fois, les buts de la conception doivent être respectés mais c'est une autre manière simple d'étendre l'architecture sans reconcevoir tout à partir du début. Par exemple, la largeur des registres et des éléments sont découplé pour qu'ils puissent être changés indépendament.

Quatrième partie

Sujets Avancés

Un coeur CPU superpipeliné n'implique pas seulement l'utilisation de pipelines à longueur variable. Quelques caractéristiques du FC0 et du F-CPU en général seront discutés ici, ils ne sont pas seulement des "composants" mais des philosophies de conception qui sont menées par des choix donnés dans la première partie du document.

Chapitre 1

Les exceptions

Quelque soit le type du processeur (CISC, RISC ou toute autre architecture), il génère beaucoup d'exceptions, d'interruptions, (((((((((traps))))))))) et appels systèmes (ici, on ne parle pas des commutations de contexte). Chaque étage de pipeline peut générer plusieurs erreurs que l'OS doit traiter ; qui demandent que l'application doive "redémarrer" l'instruction (((((((trapped))))))) ou continuer après (((((((((the trap))))))))). Ceci implique que le contexte complet doit être sauvegardé, mais lequel ?

Le contrôle peut être transféré à l'OS, un traitement d'interruption ou un traitement de (((((((((trap))))))))) à tout moment, à tout étage du pipeline. Un pipeline RISC classique comprend (et génère) par exemple :

- IF (Traitement de l'Instruction) : erreur de page
- ID (Décodage d'Instruction) : instruction invalides, instruction (((((((trap))))))), instructions privilégiées.
- EX (EXécute) : divise par zéro, dépassement, toute erreur mathématique IEEE FP...
- MEM (accès MEMoire) : erreur de page, erreur de protection

Non seulement le processeur doit déclencher le traitement adéquat (car plusieurs erreurs peuvent survenir dans le même cycle) mais il doit aussi préserver ou vider les étages corrects du pipeline. Et comme le FC0 fait des opérations OOO, la faisabilité est trop complexe sans un tas de buffers un peu partout, de même qu'un (((((((((bookkeeping))))))))) sophistiqué, ce que nous ne pouvons pas nous permettre pour des raisons évidentes. Nous avons néanmoins besoin de garder des exceptions précises et l'aptitude de stopper le pipeline à tout moment sans perdre de données, ce qui nécessiterait qu'une partie du code soit réexécuté. Nous avons besoin d'un pipeline simple et prévisible et donc efficace qui n'est pas influencé dans son architecture par les erreurs.

L'alternative la plus simple à ce problème est dicté par le bon sens : faire que toutes les exceptions se produisent au même endroit, avant que les instructions potentiellement fautives n'entrent dans le pipeline et ne nécessitent du matériel supplémentaire. Ceci signifie : *AUCUNE INSTRUCTION N'EST EMISE SI ELLE PEUT DECLENCHER UNE EXCEPTION* ou, en d'autres termes, *TOUTES LES EXCEPTIONS DOIVENT ETRE CONTROLEES AU MOMENT DU DECODAGE POUR EVITER QU'ELLES SE PRODUISENT DANS LE PIPELINE D'EXECUTION*. Rappelez-vous ceci clairement, méditez dessus car cela influence aussi comment le jeu d'instruction est conçu.

Le bon coté de ce choix est qu'il n'y a pas de registre (((((((("trap source"))))))))) comme dans les CPUs MIPS. Toutes les exceptions sont détectées au même endroit et ((((((((((disambiguified)))))))))) et sont ordonnées implicitement. Une autre importante conséquence très bonne est qu'il n'y a pas de buffer temporaire ou de "registres renommés" comme ceux appelés dans le PowerPC. Le pipeline OOOC précédemment décrit n'est pas du tout changé et le chemin de donnée critique ne souffre pas de l'ajout de buffers supplémentaires. Il n'y a pas d'allocation de registre (((((((bookkeeping))))))), ni de logique de contrôle additionnelle.

L'autre coté, qui est à propos des contraintes est détaillé ici. Les limitations les plus évidentes ont des parades simples. Le premier problème est : pouvons-nous détecter toutes les exceptions au moment du décodage et comment ?

Première cause : erreur de page au moment du traitement de l'instruction.

Premièrement, nous ne sommes pas absolument sûr que nous allons décoder l'instruction venant à la suite car la dernière instruction de la page peut être une instruction de saut ou toute instruction iden-

tique. Donc pourquoi déclencher le ((((((((((trap)))))))))) maintenant ? Une manière facile de contourner ce problème est de "marquer" l'instruction comme fautive ou, mieux, la remplacer avec une instruction (((((((trap))))))) (ce qui demande moins de matériel). Donc si l'instruction est exécutée, elle sera ((((((((((trap)))))))))). Simple, n'est-ce pas ? Bien sûr, si un défaut de page est déclenché par l'unité de prétraitement de l'instruction, c'est une bonne pratique de prétraiter directement le code nécessaire avant qu'il soit demandé. Juste par précaution.

Seconde cause : instructions invalides, instruction privilégiées...

Pourquoi s'en faire ? (((((((((It traps))))))))). Selon le type de (((((((((trap))))))))), nous avancerons le pointeur d'instruction ou pas, traiterons le code nécessaire pour l'exécuter et démarrerons la sauvegarde des registres avec le mécanisme SRB. Les instructions précédentes n'auront pas besoin d'être enlevées du pipeline car le SRB communiquera avec le tableau pour sauvegarder les registres dans un ordre correct. Lorsque le pipeline sera "naturellement" vidé des instructions de l'ancienne application, les registres seront sauvés et l'application fautive redémarrera plus tard sans perte ou réexécution.

Troisième cause : erreur mathématique.

L'exception de saturation (ou dépassement (overflow)) (à la MIPS) n'est pas implémentée. Les instructions en virgule flottante IEEE ont un drapeau de "conformité" qui stoppe la sortie d'une instruction jusqu'à ce que le résultat soit "bon", sinon, le résultat ((((((((((sturate)))))))))) et ne déclenchera pas de ((((((((((trap)))))))))). La condition "division par zéro" est facilement détectée au niveau de l'étape du décodage avec le bit de propriété ZERO du registre de division. Au même moment, nous pouvons détecter si le résultat sera zéro et émettre une opérations "clear" plutôt qu'une division.

Quatrième cause : erreur de page, erreur d'adresse invalide.

Nous pouvons considérer que la mémoire est protégée sur la base d'une granularité de page, donc l'erreur de page génèrera un code de contrôle de protection avant de charger la page. Mais détecter une erreur de page est très simple : nous devons contrôler l'adresse avec les valeurs contenues dans la table des pages. Si l'adresse ne correspond pas aux pages disponibles, c'est une erreur de page : nous ((((((((((trap)))))))))).

Maintenant, le problème est d'avoir le statut (page présente ou non ?) au moment du décodage. C'est compliqué car les accès mémoire sont à peu près la moitié des instructions exécutées !

L'alternative est d'utiliser un mécanisme similaire aux bits de "propriété" ZERO pour chaque registre. Ceci signifie que lorsqu'une valeur est écrite dans le jeu de registres par le Xbar, quelques ports du Xbar communiquent la valeur de la page dans la table. En un cycle ou deux, la donnée est prête pour l'étape ID, c'est un contrôle spéculatif qui est transparent à l'architecture du jeu d'instruction. Au moment de ce contrôle de page, nous pouvons aussi contrôler le champ de l'adresse, vérifier si la valeur est dans le cache L1 et si oui, indiquer dans quel banc elle se trouve et prétraiter la ligne de cache, etc...

Un problème évident, néanmoins, est que nous ne pouvons pas sérieusement vérifier toutes les valeurs circulant dans le Xbar vers le jeu de registre. Non seulement cela n'est pas toujours utile, mais cela consomme de la puissance. La manière la plus simple (pour le prototype) est de contrôler le résultat des mises à jour du pointeur car ils sont susceptibles d'être réutilisés rapidement comme pointeurs.

Pour des architectures plus sophistiquées, un autre "marqueur transparent", indiquant que le registre est utilisé comme pointeur, pourra être très utile. Nous pouvons permettre, par exemple, à quelques registres de posséder ce marqueur, quelque chose comme 16 (64/4 semble raisonable) et ce drapeau sera positionné à chaque accès mémoire effectué avec ce registre. Ces drapeaux seront alloués avec un mécanisme LRU utilisant un décompteur 4 bits. De cette manière, lorsque l'ID reconnaît une instruction de lecture/écriture mémoire, il contrôle le drapeau du pointeur et s'il est positionné, il envoie les informations associées à l'unité L/S (des informations comme : dans quel banc L1 se trouve la donnée ou dans quel buffer, etc.) ou il ((((((((((traps)))))))))) si le contrôle de la table de page retourne une valeur négative. Si le drapeau du pointeur n'est pas positionné, l'ID attend un contrôle de table de page et positionne le drapeau de pointeur. Bien sûr, comme tous les drapeaux transparents, leur valeur n'est pas sauvegardée pendant les commutations de contexte et est regénérée automatiquement dès qu'ils sont utilisés. En l'absence de drapeau explicite dans les instructions, c'est un moyen assez simple de réduire la charge de contrôle de la table et les adresses peuvent être contrôlées AVANT qu'elles ne soient nécessaire. L'unité L/S est seulement chargée de stocker temporairement la donnée qui transite de/vers la mémoire et les caches. Ce dernier détail invalide le dessin de la figure 2.2 où la table des pages était stockée dans l'unité L/S.

Ici, à peu près toutes les causes d'exceptions sont couvertes et les contournements ont été expliqués. Il n'y a pas d'impact visible sur l'ISA mais les règles de codage deviennent plus strictes, comme dans les processeurs superscalaires. De toute manière, les contournements des problèmes créés par le pipeline d'exécution "sans exceptions" du FC0 sont connus et expliqués. D'autres nouvelles exceptions utiliseront probablement la même idée que pour celles existantes : utiliser un drapeau dynamique. De cette manière, programmer le FC0 ressemble plus à de la programmation d'un CPU RISC avec quelques règles de codage.

Chapitre 2

Le mécanisme Smooth Register Backup

Comme décrit dans la première section du document, dans la discussion des "64 registres", une alternative pour le fenétrage de registres, les bancs du jeu de registres ou les architectures mémoire à mémoire est d'implémenter un "Smooth Register Backup" (SRB pour faire court) pour la sauvegarde automatique des registres. Ce n'est pas un système habituel dans les microprocesseurs car il est caractérisé par une communication avec le tableau et l'utilisation d'un algorithme "drapeau du trouver d'abord". Le mécanisme complet est assez simple, comme nous le décrivons ici (même si j'ai l'impression de ((((((((((rant)))))))))), donc : lisez lentement puis relisez encore plus lentement). Note : selon son utilisation réelle et son utilité, le mécanisme SRB peut être enlevé du F-CPU avec un impact minimal sur l'architecture complète, sur le jeu d'instruction et les applicatifs. Quelques pilotes et noyaux peuvent nécessiter le code additionnel de la sauvegarde manuelle des registres. D'autres techniques similaires peuvent aussi être utilisées à la place.

Comment et quand est utilisé le SRB ? Hé bien, il est utilisé pour ce qu'il fait :

Vider le jeu de registres vers la mémoire et/ou charger un nouveau contexte.

Il peut être utilisé à tout moment car il n'interfère pas avec les autres matériels excepté l'unité L/S.

Il est principalement utilisé pour la commutation de contexte (le SRB peut être déclenché par une interruption et le reste est fait automatiquement), pour le sauvegarder et restaurer les registres après que la routine IRQ ait été complétée. Dans ces cas, il y a deux threads : "l'ancien" thread et le "nouveau" thread. Le vidage ou la lecture des registres de/vers la mémoire par des instructions reste néanmoins l'exception.

Le "nouveau" thread est défini pour démarrer aussi tôt que le signal SRB est déclenché et le SRB doit sauver ces registres avant qu'un nouveau thread les utilise pour s'assurer de la cohérence des données.

Non seulement le SRB enlève le besoin de sauver et de restaurer manuellement les registres mais il le fait plus rapidement que le logiciel (alors que l'application tourne encore) et s'adapte aux circonstances en réordonnant la séquence de sauvegarde à la volée. Il utilise un peu de matériel supplémentaire, des données du tableau (le drapeau indiquant que "la valeur du registre est en cours de calcul"), il vole les cycles d'horloge inutilisés par l'unité L/S de la mémoire pour charger et stocker les registres, il a peu de drapeaux, quelques registres de pointeurs et un peu de logique. Pour savoir comment l'utiliser, nous définissons quelques règles de comportement :

- Nous ne pouvons pas sauver le registre tant que sa valeur est traitée. Le tableau nous indique quel registre ne pas (encore) sauvegarder. Ce status change à chaque cycle, donc changer l'état du tableau rapidement est très important.
- Il n'y a pas de besoin de sauvegarder un registre qui n'a pas été modifié depuis la dernière sauvegarde. Il existe un drapeau (((((((("dirty"))))))))) pour ce propos, il est positionné lorsque l'on écrit dans le registre.
- Nous avons un drapeau spécial "pas encore sauvegardé" qui dit que le registre physique doit être sauvegardé avant qu'il soit utilisé par le nouveau thread. Au même moment, ce drapeau bloque le tableau pour qu'il puisse demander une requête "expresse". Ce drapeau est chargé à partir du "dirty bit" lorsque le signal SRB est détecté et que le dirty bit est mis à zéro pour le nouveau thread.
- Lorsqu'un nouveau thread a besoin d'utiliser (lire et écrire) un registre qui n'a pas été encore sauvegardé, il programme le séquenceur SRB pour modifier l'ordre et attend du registre qu'il soit libre. Le tableau, qui est scruté par l'unité de décodage d'instruction, "bloque" l'instruction jusqu'à ce que la valeur soit prête et que le drapeau "sauver en priorité" est positionné avant que la donnée

soit prête.

– La séquence SRB est atomique, elle ne peut pas être stoppée à moins qu'il y ait une erreur mémoire. Un nouveau signal SRB doit attendre que le signal SRB précédent émis sous complètement traité. Désactiver les IRQs lorsque le SRB fonctionne évite les cycles perdus (l'attente pour que le précédent SRB soit terminé, avant que le traitement précédent soit exécuté). Si une exception se produit pendant une séquence SRB, bonne nouvelle : nous avons déjà commencé à sauver les registres : -) Nous avons besoin que (l'ancienne) séquence se termine avant de déclencher une "nouvelle" séquence SRB et exécuter le traitement.

Bien sûr, ce grand nombre de drapeaux peut être condensé, utilisant une Machine à Etats Finis (les détails de cette implémentation sont laissés au concepteur). Mais l'algorithme suivant n'en a pas besoin : "pour chaque cycle, écrire dans la mémoire le premier registre qui : 1) demande une sauvegarde expresse, 2) n'est pas encore sauvegardé (dans la priorité décroissante), démarre à partir du registre #1". L'algorithme stoppe lorsqu'il n'y a plus de registres à sauvegarder. Lors d'une commutation de contexte, il y a deux accès mémoire, un pour sauvegarder l'ancienne valeur du registre et un pour traiter la nouvelle valeur de thread. Si une moitié des opérations du thread est chargée ou stockée, ceci prendra à peu près une centaine de cycles pour sauvegaredr un contexte. Avec un pipeline à simple étage et pas beaucoup de bande passante, cela peu prendre 200 cycles pour une commutation complète de contexte. SRB demande beaucoup de bande passante mais la sauvegarde par logiciel encore plus. Au moins, le SRB utilise tout le matériel disponible alors que les solutions logicielles nécessitent encore plus de matériel (à cause du code de sauvegarde explicite).

L'algorithme SRB est une séquence qui peut être réordonnée avec le chargement ou le stockage des 63 registres, en démarrant à partir du registre #1 : nous avons besoin d'une manière pour extraire cette séquence à partir d'une ligne de drapeaux. Une unité "trouver d'abord", similaire (mais plus simple) à l'arbre binaire utilisé dans l'unité d'incrémentation, peut le faire facilement. A l'entrée, il sélectionne la "requête expresse" s'il y en a ou la requête normale à partir des registres qui ont besoin d'être sauvegardés. Le drapeau express est positionné par le tableau et le drapeau "pas encore sauvé" appartient au mécanisme SRB. La sortie de l'arbre binaire sélectionne directement un registre (sur les 63) pour la lecture et/ou l'écriture et le reset des drapeaux de registres. Peut être qu'un simple dessin sera plus parlant : -)

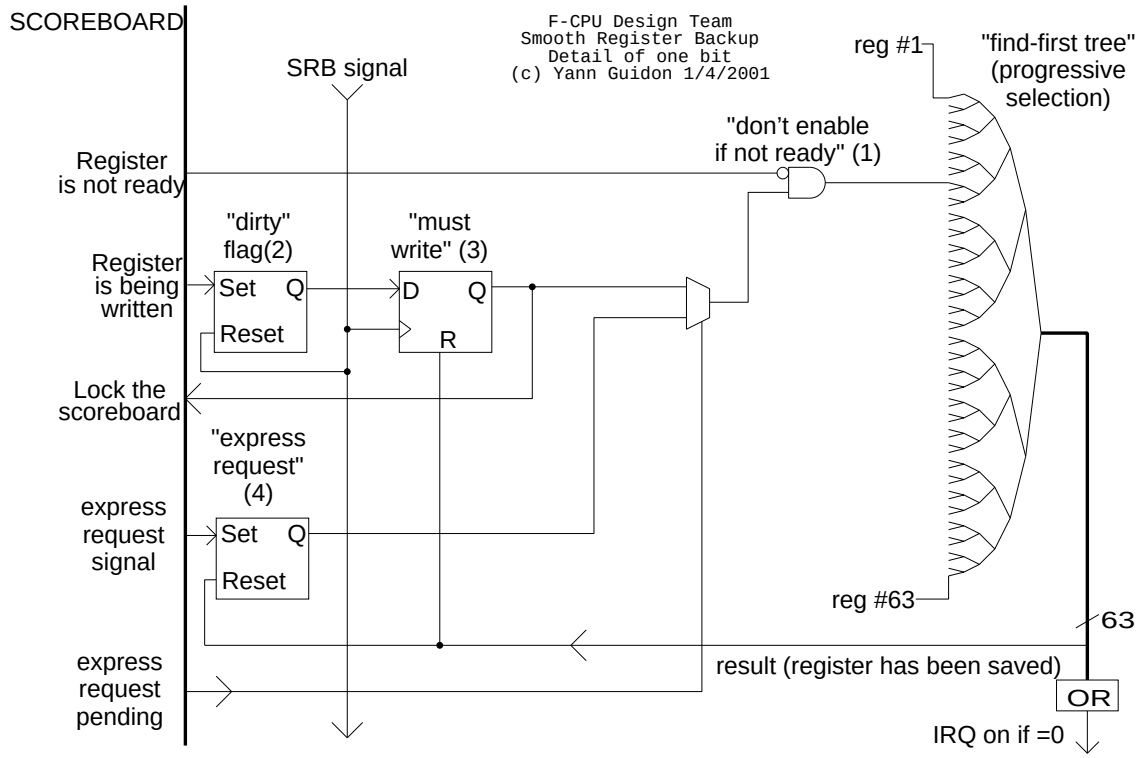


FIG. 2.1 – Détails d'un bit du drapeau SRB et du mécanisme de décision

Lorsqu'aucune requête "expresse" est faite, l'application a la priorité sur le SRB pour accéder à l'unité L/S. Sinon, le drapeau "express" signifie que l'instruction est bloquée à l'étape ID et qu'aucun accès mémoire n'est effectué (à moins d'un manque du cache vienne juste d'être résolu...). Donc, dans tous les cas, le SRB n'a jamais la priorité (ce qui simplifie les choses). Le principe du SRB peut être

étendu pour les processeurs à multiples étages sans modifications. Un F-CPU superscalaire à quatre voies pourra être capable d'utiliser cette partie, à priori sans soucis.

Comme noté plutôt ; il y a seulement deux taches considérées par le CPU : "l'ancienne" et la "nouvelle" tâche. Aucune supposition n'est faite sur l'endroit o sont transférées les données, sur les adresses des buffers de contexte, il n'y a donc pas de limitation dans le nombre de tâches. Les caches effectueront leur rôle de garder les données spatialement et temporellement proches du coeur, de telle manière que les programmes multithreadés tournent normalement. Mais l'utilisateur ne peut pas toujours spécifier l'adresse de ces buffers de contexte. Le mécanisme SRB possède deux pointeurs : SRB_old et SRB_new, qui sont utilisés pendant les opérations SRB. Après la complétion des commutations de contexte, le pointeur SRB_new est copié dans SRB_old pour que pendant la commutation de contexte suivante, seule la nouvelle tâche soit fournie au CPU. C'est à l'utilisateur de mettre en place la liste désirée. Ce "nouveau" pointeur peut aussi être stocké dans le "vieux" buffer de contexte, comme pour effectuer une opération round-robin automatiquement à chaque commutation de tâche d'IRQ. Le matériel de contrôle de cache permettra probalement de "cartographier" une certaine zone de mémoire directement dans le cache, pour qu'aucune opération LRU ne vide les données dans le cache. C'est ici que les taches importantes, telles que les tâches de noyau ou temps réel, doivent stocker leurs buffers de contexte pour une meilleure performance. De plus, si une commutation automatique de contexte est implémentée, elle voudra d'abord prétraiter tous les buffers de contexte (nouveaux et anciens) dans le cache L1 aveant de déclencher le mécanisme SRB. Ce prétraitement se produit en tâche de fond pour ne pas pénaliser les tâches prioritaires.

Notez que dans le cas où le coeur FC0 possède deux ports privés independants de mémoire, le processus de commutation de bancs de registre peut être facilité si les bancs sont cartographiés en zones de mémoire différentes. Par exemple, si le banc de registres est vidé vers un port, le nouveau banc de registres bénéficie de la lecture sur l'autre port. Il y aura beaucoup moins de cycles d'occupation de bus et le temps d'attente lors de la commutation diminuera.

Chapitre 3

Le planificateur

Gérer plusieurs unités superpipelinées qui peuvent émettre leur résultat au même moment semble difficile au premier regard. Les règles suivantes de comportement aideront à comprendre quoi faire et quand :

- Les "portes" du Xbar des deux ports d'écriture doivent être commandées pendant chaque cycle, pour que les deux ports de lecture du jeu de registres aient les données correctes venant de la bonne unité.
- Une instruction ne peut pas sortir si plus de deux ports d'écriture sont utilisés pendant le cycle lorsque l'instruction a été complétée.
- Si l'instruction peut être sortie, elle doit utiliser un port d'écriture "libre".

Rappelons aussi que les règles du tableau s'appliquent aussi. Plus spécifiquement, il n'est pas possible de sortir une instruction si les opérandes ne sont pas prêtes, dans le jeu de registres ou dans le Xbar (pendant un un cycle d'attente de registre, pour des paires d'instructions ((((((((((back-to-back))))))))))). Le planificateur doit aussi reconnaître cette situation.

Deux solutions sont possibles et ont été étudiées :

o La première possibilité est d'associer une Machine à Etats Finie à chaque registre. C'est un décompteur qui déclenche les signaux appropriés à la fin.

L'avantage est que cela est complètement indépendant du nombre réel des opérations qui peuvent être émises pendant chaque cycle d'horloge, il est préféré pour cette raison.

Malheureusement, cela crée des bus internes très très grands et la détection de risques est trop lente, particulièrement lorsque le bus d'écriture du Banc de registres doit être alloué.

o La seconde solution peut moins croître avec le nombre d'instructions émises par cycle, mais c'est un algorithme simple et déterministe qui consomme moins de ressources lorsque une ou deux instructions sont émises à un moment donné. C'est un FIFO qui est aussi profond que le pipeline et chaque ligne contient le nombre de registres qui sera écrit dans le jeu de registres. Comme il y a deux ports d'écriture, le FIFO contient des champs de 2 x 6 bits. Si le "slot" est vide, deux drapeaux additionnels sont utilisés pour indiquer cet état. Les lignes vides sont mises à zéro (les bits sont mis à zéro lorsqu'il descendent dans le FIFO) mais faire un OR sur les bits prend trop de temps (oui, un OR 6 bits prend plus de temps et de place qu'un 7ième bit par champ).

En fait, le tableau utilise la première représentation : les 63 bits qui représentent si un registre est utilisé sont étalés sur tous le jeu de registres ; ils sont mis à zéro lorsque qu'il y a une écriture sur la ligne correspondante. Cela utilise de long fils et des grands bus mais c'est assez simple. D'un autre coté, le second mode de représentation pour le tableau (beaucoup de registres contenant le nombre des registres actuellement utilisés) utilise trop de ressources et ne suit pas correctement lorsque plus d'instructions sont décodées au même moment.

Les informations du planificateur et du tableau pour le FC0 peuvent utiliser toute représentation adaptée mais elles peuvent aussi être à la fois utilisées en parallèle (lorsque cela est le cas). Avoir les deux représentations aide à avoir les informations voulues avec le moins de temps d'attente. Si un vecteur de bits est nécessaire, il sera lu dans le tableau et si un nombre est nécessaire, il sera lu dans le FIFO du planificateur.

Maintenant, il y a une caractéristique très importante, associée au FIFO du planificateur : le "slot" peut être alloué à plusieurs niveaux car les instructions peuvent avoir différents temps d'attente. Ceci

signifie qu'une instruction de multiplication "réservera" (s'il est libre) un "slot" dans le FIFO au 6ième niveau, alors qu'une addition réservera un slot au second niveau.

Le décodeur d'instruction doit donc fournir au planificateur une information précise sur le temps d'attente de l'instruction qui va sortir. Cette information est stockée dans une Table Lookup qui prend l'instruction et et le champ des drapeaux comme entrées ; sa sortie étant le nombre de cycles de temps d'attente pour l'instruction. Cette LUT est cablée mais l'implémentation supporte des registres de plus de 128 bits, une certaine partie sera reconfigurable à la volée pour supporter la taille programme des champs (voir le chapitre 2.5 à propos des tailles des variables).

Lorsque le jeu d'instruction est conu, les instructions doivent avoir un temps d'attente garanti et fixe pour que la LUT soit aussi compacte et rapide que possible. cela met un peu de pression sur la planification de deux types d'instructions : Charger/Stocker et la division. Les instructions Get/Put sont aussi à "temps d'attente indéterminé" mais elles bloquent (attente) le pipeline.

L'Unité de Division Entière du FC0 (une première implémentation "économique") est une machine à décalage/soustraction lente comme on en trouve déjà dans les vieux microprocesseurs : le temps d'attente est proportionnel au nombre de bits à diviser. Il n'est pas pipeliné et le traversement est proportionnel à cette largeur de donnée. La planification en est simplifiée car il n'est pas pipeliné : le FIFO n'a pas à contenir 64 slots pour le cas où un nombre 64 bits est divisé ; un simple décompteur est suffisant. De plus, ce temps d'attente est à 8, 16, 32 ou 64 cycles et 8 cycles est plus que le temps d'attente du multiplicateur : le compteur n'interfère pas avec le FIFO, il est au-dessus et est initialisé très facilement avec le drapeau de la taille du mot d'instruction.

Le cas des instructions Charger/Stocker est plus difficile car elles ne sont pas déterministes. La situation est simple lorsque la donnée est déjà contenue dans le buffer L/SU, sinon, c'est vraiment un sac d'embrouilles.

Lorsque la donnée est contenue dans le buffer L/S, le temps d'attente est deterministe : il prend un cycle dans le buffer, un cycle dans l'octet du shuffler (qui sélectionne et ordonne les octets en mot), un cycle dans le Xbar et un cycle dans le jeu de registres. C'est la situation qui *doit* être privilégiée lorsque cela est possible. C'est ce qui est proposé dans les premières sorties des adresses (le pointeur doit être connu dès que possible pour que la donnée chargée puisse être traitée de la mémoire par avance) et l'utilisation avisée du flux et des bits de cache (((((((((hint))))))))).

Lorsque la donnée n'est pas présente dans le buffer L/S, le planificateur doit se préparer pour les évènements asynchrones et il n'y a pas de garanties que des slots libres seront disponibles. En moyenne, il est probable que les deux ports d'écriture du jeu de registres soient utilisés 70% du temps ; l'écriture réelle de la mémoire se produisant quelques cycles après que la donnée soit réellement disponible. Il n'y a pas de tels problèmes, néanmoins, lorsque la donnée chargée est demandée pendant le cycle suivant l'instruction de chargement : le pipeline s'arrêtera et laissera un peu de place à l'unité L/S pour remplir le Xbar avec les données désirées.

dessin : je dois insérer un diagramme du planificateur FIFO et du tableau

Chapitre 4

L'unité de mémoire (Traitement et L/SU)

bientôt écrit

dessin : Je dois insérer un diagramme du (((((((LUT))))))) registre, du décodeur, du traitement et du LSU

Cinquième partie

Architecture du Jeu d'Instructions F-CPU

Chapitre 1

Concevoir un jeu d'instructions

Une fois que l'on a été d'accord sur les fonctionalités et caractéristiques les plus fondamentales du CPU, il a été nécessaire de définir le jeu d'instruction.

Pour F-CPU, ce n'est pas tout à fait direct, même si grce à l'architecture, cela reste assez simple et qu'elle n'inclut pas de grandes innovations. Le problème réel réside dans la manière itérative dont les choses sont décidées et intégrées dans le CPU. L'Architecture du Jeu d'Instructions (ISA) fait face à beaucoup de contraintes et son évolutivité reste très grande. L'ISA détermine de nombreuses caractéristiques pour le futur car nous ne pourons pas le changer comme un CPU sur un connecteur. Comme de nombreuses caractéristiques déterminent la vie de toute l'architecture et du projet, toutes les informations apportées ici doivent être considérées comme temporaires et susceptibles de changer sans préavis. L'ISA sera réellement défini lentement, après chaque cycle de simulation o l'on pourra tirer des conclusions sur l'utilité et la nécessité des opcodew ou des drapeaux particuliers.

Le jeu instructions changera donc souvent et évoluera beaucoup avant qu'il ne soit complètement défini par le groupe. Quelques changements pourront même avoir lieu après les premiers prototypes ou lors de l'élaboration des puces. C'est la raison pour laquelle l'ISA actuelle n'est pas considérée comme complètement défini au moment de l'écriture et plusieurs techniques sont utilisées pour faciliter son développement.

En premier lieu, le mot d'instruction lui-même qui fait 32 bits de large, doit être utilisé d'une manière flexible. Les instructions exécutées par le F-CPU nécessitent un nombre variable d'opérandes et de drapeaux. Ils sont rassemblés au milieu du mot pour faciliter l'allocation du champ de bit. L'opcode (un champ 8 bits qui définit l'instruction) est situé à un bout du mot (dans le MSB) et le registre de destination est à l'autre bout (dans le LSB). La largeur de donnée immédiate peut être de 8 ou 16 bits et nous pouvons inclure un ou deux opérandes de registres. La place restante est remplie par les drapeaux qui peuvent être mélangés avec l'opcode de l'instruction lorsqu'il n'y a pas suffisament de place ou le champ de donnée immédiat peut être restreint. Lorsqu'il reste un peu de place, nous pouvons étendre le champ de la donnée immédiate (même si les drapeaux tentent d'utiliser autant d'espace que possible).

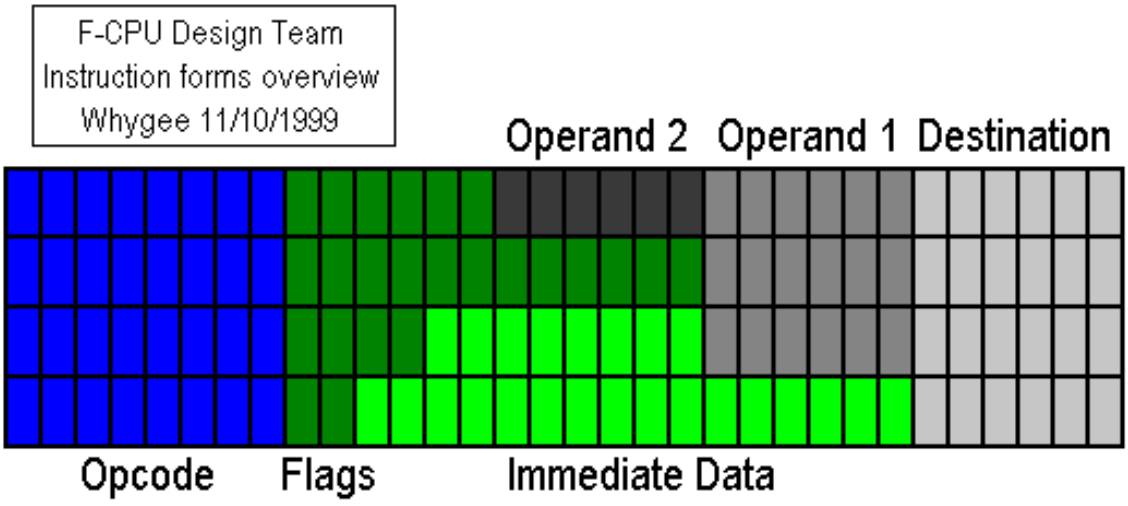


Fig. 1.1 – Preliminary overview of the instruction forms

Nous concevons le jeu d'instruction avec un panorama toutes les instructions nécessaires et les formes qu'ils utilisent. La largeur du champ immédiat n'est pas définie mais elle est gardée pour la synthèse finale. Lorsque nous avons fait la somme de toutes les formes d'instruction, nous allouerons les champs. Ils seront placés en focntion de leur fonctions et toutes les fonctions similaires seront groupées. C'est très simple pour les champs de registres mais c'est moins facile lorsque nous allouont les drapeaux. La taille des champs de données immédiates sera déterminée lorsque tous les autres champs seront alloués.

Le second truc optimise la carte des opcodes. Il y aura, bien sr, beaucoup de place dedans pour les opcodes futurs. Mais si le nombre d'opcode sera connu à un moment, leur nombre pourra être redéfini lors de la réalisation du prototype final. Cela signifie que lors de la sortie du F1, la compatibilité binaire n'est pas certaine mais les opcodes seront définis avec les fichiers include dans les simulateurs et les émulateurs. Ceci laisse le champ libre pour "allouer " des valeurs d'opcode au dernier moment et les optimiser pour simplifier la logique de décodage d'instruction. Mais la compatibilité est préservée à tout moment au niveau du source dans les fichiers de langage assemblé. Seul le codage peut changer pendant le développement.

Cette méthodologie permet au groupe de travailler avec des implémentation précoces de la puce et de synthétiser le jeu d'instructions avant de la sortir. Aucune décision arbitraire n'est faite car toutes les fonctionnalités seront analysée et discutées par le groupe.

Chapitre 2

Format des instructions

Le F-CPU est un processeur RISC avec des instructions 32 bits. Le champ opcode est de 8 bits, chaque registre nécessite un champ de 6 bits et l'espace restant est utilisé pour les valeurs immédiates et les drapeaux. Les tables (préliminaires) suivantes montrent comment elles sont organisées.

Notez que le champ de l'opcode est dans le Least Significant Bits mais l'opérande de registre la plus utilisée est dans le Most Significant Bits. Donc, par convention, la syntaxe d'assemblage du langage (pour des raisons de cohérences) suit les structures des instructions et écrit les opérandes dans cet ordre : L'opcode en premier, suivi des drapeaux, des valeurs immédiates et des opérandes de source, pour terminer par le registre de destination. Par exemple :

add.b r1,r2,r3 ; ajoute les octets dans la partie basse de r1 et r2, le résultat dans r3.

Les formats d'instruction sont :

size	8	6	6	6	6
bits	31 24	23 18	17 12	11 6	5 0
function	Opcode	Flags	Reg 3	Reg 2	Reg 1
size	8	12	6	6	
bits	31 24	23 19	20 6	5 0	
function	Opcode	Flags	Reg 2	Reg 1	
size	8	4	8	6	6
bits	31 24	23 20	19 19	11 6	5 0
function	Opcode	Flags	Imm8	Reg 2	Reg 1
size	8	2	16	6	
bits	31 24	23 22	21 6	5 0	
function	Opcode	Flags	Imm16	Reg 1	

Il est très tentant d'utiliser un préfixe d'opcode à 2 bits pour identifier les formats d'instruction mais cette idée doit être abandonnée pour une compilation d'opcode ultérieure.

Chapitre 3

La modularité ISA

Le jeu d'instruction du F-CPU est modulaire et contient un groupe "core" et plusieurs groupes "optional" d'instructions qui nécessiteront plusieurs instructions core pour compléter les opérations. La présence de ces instructions optionnelles peut être détectée au lancement avec les indications contenues dans un jeu de Registres Spéciaux cablés en dur.

Il doit être compris que le jeu d'instruction "core" est créé pour fournir une compatibilité binaire minimale parmi les différentes implémentations. Toute puce peut intégrer une ou plusieurs instructions "optional" indépendamment des autres considérations. Ceci dépend des performances nécessaires, de l'application de destination, de la technologie disponible et des algorithmes.

Qu'est ce qui est core et qu'est ce qui est optionel ? Comme règle d'or, les instructions optionnelles incluent des "fonctionnalités" qui sont habituellement possible avec du matériel ou des circuits plus complexes. Par exemple, la capacité SIMD est recommandée mais n'est pas obligatoire car une unité arithmétique SIMD est plus complexe qu'une unité scalaire. Les instructions d'incrémentation, en virgule flottante, logarithmique et gestion de SRB sont validées lorsque l'Unité d'Exécution correspondante ou fonctionnalité est implémentée. Il est possible d'implémenter un F-CPU réellement minimal et de l'étendre en ajoutant les instructions souhaitées avec des Unités d'Exécution, abandonnant les opcodes inutilisés lorsqu'il n'y a pas suffisament de transistors.

D'un autre coté, il est recommandé que la plupart des instructions d'entiers et des fonctions SIMD soient implémentées car elles fournissent les fonctionalités les plus importantes pour le futur.

Chapitre 4

Le format 2r1w et ses extensions

Le F-CPU augmente le ratio MOPS/MIPS de son architecture en cassant la règle d'or de la limitation de deux registres de lecture et d'un registre d'écriture des architectures RISC classiques. Plusieurs instructions du F-CPU ont besoin de plus d'un registre pour la réécriture du jeu de registres, d'autres ont besoin d'opérandes à trois registres pour la lecture. Ces instructions "non-RISC" sont marquées comme 3r1w ou 2r2w dans ce document car pouvant influencer les règles de codage des futures implémentations du F-CPU. Ils nécessitent probablement un bit spécial dans l'opcode pour simplifier le décodage. Leur support est optionnel (non-core), encore nécessaire pour les instructions load et store avec une mise à jour de pointeur.

Chapitre 5

Drapeaux

Les instructions partagent un certain nombre de propriétés qui sont mises dans les “drapeaux” à l'extérieur du champ d'opcode. Lorsque leur position peut changer dans le futur, leur signification restera globalement la même dans les futures générations de processeurs.

Les drapeaux n'altèrent pas beaucoup la syntaxe des instructions. Ils ajoutent une lettre par drapeau aux mnémoniques existants afin qu'ils puissent toujours reconnaitre le instruction. Cela évite la prolifération de mnémoniques obscurs et la nécessité de tous se les rapeller. D'un autre coté, la taille des mnémoniques est variable et peut aller de deux (ou) à neuf (sshiftrai) lettres et les mnémoniques seront probablement réoganisés plus tard pour réduire la taille des plus long. Habituellement, les les lettres des drapeaux sont ajoutés dans l'ordre dans lequel ils apparaisent dans le mot d'instruction.

5.1 Taille des Drapeaux

Dans certains opcodes, les drapeaux peuvent contenir un paramètre “size” qui définit la taille des opérandes dans lequelles les opérations prennent place. Ce drapeau est décodé par défaut selon la table suivante :

Flags	Size (byte)	Suffix	Name
00	1	B	Byte
01	2	D	Double-Byte
10	4	Q	Quad-Byte
11	8	(none)	Octa-Byte (Word)

Dans le langage assembleur du F-CPU, la taille du drapeau est notée par un postfix sur l'opcode, soit “.b”, “.d”, “.q” ou un nombre complet lorsque les initialisations courantes ne fournissent pas la taille nécessaire. Avec l'absence d'une taille de postfix, le drapeau est initialisé à “11”. Si le CPU est seulement une version 32 bits, le code “11” est cartographié à “10” (32 bits) de telle manière que se soit toujours le mot le plus large supporté par la machine.

Lorsque la largeur de la donnée du CPU augmente, le processeur peut changer l'interprétation de ce drapeau avec un jeu de registres spéciaux. Ceci permet à la plate-forme F-CPU de traiter toute largeur de donnée ayant une puissance de deux, au-dessus de 32 bits. Les mots SIMD et les algorithmes croitront de manière très directe à 128 bits, 256 bits, 512 bits, 1024 bits, etc.

5.2 Drapeau SIMD

Le F-CPU est un processeur orienté SIMD. La plupart des instructions pour les opérations sur les données peuvent spécifier si ces données sont traitées comme un tout ou en morceaux individuels. Le drapeau SIMD, en plus de sa taille, spécifie comment son traitées les données.

Lorsque le drapeau SIMD n'est pas initialisé, le CPU se comporte comme un processeur normal, traitant chaque registre en fonction de leur taille de drapeau. Le registre complet, ou seulement la partie la plus basse, est traitée.

Lorsque le drapeau SIMD est initialisé, le CPU traite tout le registre dans sa taille complète et le drapeau de taille définit la largeur des morceaux individuels dans la largeur du mot.

Dans la syntaxe du langage assembleur du F-CPU, le drapeau SIMD est repréré par un prefixe “s” sur l'opcode, d'une manière similaire au “f” des opérations en virgule flottante.

5.3 Drapeau IEEE

Pour les opérations en virgule flottantes, le F-CPU définit un “drapeau de compatibilité IEEE754”. Ce drapeau altère le standard IEEE pour les opérations en virgule flottante de deux manières : lorsqu'une condition d'erreur est détectée, elle ne (((((((trap))))))) pas le processeur et les valeurs résultantes sont saturées ou décalées. Ce drapeau est destiné à faciliter la conception du pipeline de la famille coeur du FC0 o aucune instruction potentiellement erronée ne doit entrer dans le pipeline. Sur d'autres coeurs de familles, ce comportement doit être préservé. Ce drapeau est utilisé lorsque la vitesse est plus importante que la précision, de telle manière que, en fonction de l'implémentation, on désactive l'utilisation de nombre denormal IEEE par exemple.

5.4 Saturation/drapeau retenue

Ce champ est utilisé par les instructions d'addition, de soustraction et de multiplication d'entiers o le résultat o le résultat ne rentre pas complètement dans un registre. Il existe trois possibilités :

- ignorer la partie haute (et “wrap around”),
- saturer (“clip”) ou
- écrire la partie haute dans un autre registre dont le nombre de destination sera destination+1 (voisin le plus proche).

Déclencher une exception sur la retenue est hors de question car cela ralentirait le CPU dans les boucles critiques. L'écriture de la retenue dans un registre de retenue spécial créerait quelques problèmes architecturaux et l'écriture de la retenue dans un des opérandes source effacera le bénéfice du format d'instructions à trois opérandes.

Notez que la retenue est effectuée avec le registre #63 comme destination, elle n'est écrite nulle part car le registe ”suivant” est le registre #0 qui est cablé à 0.

Ce drapeau nécessite deux bits qui peuvent être remis à zéro (par défaut : wrap around) ou l'un des deux est initialisé (soit (((((((((clip))))))) ou écritre dans le voisin du registre résultant). En fonction du type d'opération, la paire de drapeaux est appelée “floor” ou “saturate”.

Les comportements de retenue ou de saturation sont écrit dans un langage assembleur avec un postfixe “c” et “s” respectivement. Le comportement par défaut (wrap) est noté par l'absence de postfixe.

La combinaison interdite (initialisation et de c et de s) doit être utilisé plus tard comme une saturation “signée” o les valeurs plancher et de plafond sont 0x8000 et 0x7FFF au lieu de 0x0000 et 0xFFFF.

De manière à mélanger le résultat et la retenue, les instructions mixhi et mixlo sont fournies. Par exemple, les valeurs 16 bits SIMD d'une soustraction 8 bits peuvent être générées en trois instructions :

- **ssubb.b r1, r2, r3** ; r3=result, r4=borrow
- **mixl.b r3, r4, r5** ; takes the two lower halves from r3 and r4 and mix them into r5
- **mixh.b r3, r4, r6** ; takes the two higher halves from r3 and r4 and mix them into r6

Notez que la retenue (ou drapeaux ”borrow” [sub] ou ”high” [mul] ou ”modulo” [div]) peut influencer les règles de décodage des instructions dans l'implémentation du futur F-CPU. Ce n'est pas un problème pour le FC0 mais il doit changer avec les conceptions superscalaire, des aux limitations de la taille du jeu des registres.

5.5 Endian flag

Les instructions Load/Store et les unités dédiées peuvent spécifier le modèle endian des opérations d'accès à la mémoire qui sont effectuées. C'est optionnel pour les systèmes minimaux et embarqués car le matériel nécessaire peut ne pas être justifié, dans quel cas l'endian est recommandé comme petit. Pour des applications à objectif général, le support des deux endians est recommandé car l'OS peut être écrit pour l'une et l'application pour l'autre.

5.6 Stream Hint flag

Les instructions Load/Store peuvent spécifier auquel des sept ”**streams**” appartient le pointeur. Dans le F-CPU, un ”stream” est similaire en signification à celui d'un CRAY T3E mais avec un mécanisme différent. Cela peut être implémenté comme plusieurs Unités L/S (le nombre de streams référence un LSU individuel), comme support des différentes banques DRAM, strides, canaux ou jeux de cache visibles pour l'utilisateur ou comme toute combinaison. Comme leur nom l'indique, ceci devrait aider la séparation des streams de données indépendantes du CPU, en évitant la congestion du chemin de donnéeet le gaspillage de cache, pour finir par augmenter la largeur de bande sans avoir à ajouter de matériel complexe.

Ce champ peut être silencieusement ignoré par le CPU si l'implémentation ne peut pas supporter cette fonctionnalité.

5.7 Autres drapeaux / champs réservés

Pour le moment, tous les bits n'ont pas été alloués. Il existe des bits de champs qui ne sont pas encore utilisés et qui devraient être mis à (0), pour préserver la compatibilité suivante de l'architecture. C'est valable pour tous les champs marqués réservés, ignorés, inutilisés ou vides. Ces bits peuvent être utilisés pour tout à tout moment sans prévenir. Le groupe implémentera peut-être un F-CPU avec le support pour les logarithmes et/ou les nombres fractionnels et le bit #11 qui est réservé dans la plupart des instructions sera très utile.

Sixième partie

Brouillon du Jeu d'Instruction F-CPU

Chapitre 1

Opérations Arithmétiques

1.1 Opérations Arithmétiques de base

Instructions concernées par cette base :

Opcodes	Mnemonic
OP_ADD	add
	adds
	addc
	addcs
	sadd
	sadds
	saddc
	saddcs
OP_ADDI	addi
	saddi
OP_SUB	sub
	subb
	subf
	subbf
	ssub
	ssubb
	ssubf
	ssubbf
OP_SUBI	subi
	ssubi
OP_ADDSUB	addsub
	addsubs
	saddsub
	saddsubs
OP_MUL	mul
	mulh
	muls
	mulhs
	smul
	smulh
	smuls
	smulh
	smulhs
OP_MULI	muli
	smuli

OP_MAC	mac
	macl
	mach
	macs
	macls
	machs
	smac
	smacl
	smach
	smacs
	smacls
	smachs
OP_DIV	div
	divr
	divrem
	divs
	divrs
	divrems
	sdiv
	sdivr
	sdivrem
	sdivs
	sdivrs
	sdivrems
OP_DIVI	divi
	divri
	divremi
	sdivi
	sdivri
	sdivremi
OP_REM	rem
	rems
	srem
	srems
OP_REMI	remi
	sremi
OP_POPC	popcount
	spopcount
OP_POPCI	popcounti
	popcounti
OP_INC	inc
	sinc
OP_DEC	dec
	sdec
OP_NEG	neg
	sneg
OP_ABS	abs
	sabs
OP_NABS	nabs
	snabs
OP_SCAN	scan
	scann
	scanr
	scannr
	lsb1
	lsb0
	msb1
	msb0
	slsb1
	slsb0

	smsb1
	smsb0
OP_CMPL	cmpl
	scmpl
OP_CMPLE	cmple
	scmple
OP_CMPLI	cmpli
	scmpli
OP_CMPLEI	cmplei
	scmplei
OP_MAX	max
	smax
OP_MIN	min
	smin
OP_MAXI	maxi
	smaxi
OP_MINI	mini
	smini
OP_SORT	sort
	ssort

1.1.1 add

ADDition

add[c][s] r3, r2, r1
sadd[c][s] r3, r2, r1

Calcul r1 = r2 + r3

add effectue une addition entière des deux opérandes sources (r3 + r2) et met le résultat dans l'opérande de destination (r1).

- Le drapeau **size** indique que **add** effectue l'addition sur les opérandes complètes ou seulement sur une partie.
- Le drapeau **SIMD** indique que **add** effectue des opérations d'addition multiples sur les parties de l'opérande (la taille de ces parties est définie dans le drapeau **size**).
- Le drapeau **saturate** indique que **add** ne dépasse pas" si le résultat est plus grand que la taille des opérandes.
- Le drapeau **carry** indique que la valeur de la retenue éventuelle est écrite au numéro de registre (r1^1). Si aucune retenue a été générée, le registre voisin est remis zéro (0x00), sinon, le LSB a été mis (0x01).

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_ADD	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Définit la taille du paramètre
21	s- prefix	1 if set	Définit si l'opération est SIMD
20	(none yet)	0	Reserved
19	-s postfix	1 if set	drapeau de Saturation
18	-c postfix	1 if set	Carry flag (**2r2w**)

Exemples :

Scalar :

R1 contient 0xF8 (nous avons seulement considéré l'octet bas dans les registres)
R2 contient 0x0F

add.b r1,r2,r3 : `r3 = 0x07 (comportement par défaut)`
adds.b r1,r2,r3 : `r3 = 0xFF (saturation)`
addc.b r1,r2,r3 : `r3 = 0x07, r4= 0x01 (retenue)`

SIMD :

R1 contains 0x000000F800000001 (dans un système 64 bits)
R2 contains 0x0000000F00000002

sadd.b r1,r2,r3 : `r3 = 0x0000000700000003 (comportement par défaut)`
sadds.b r1,r2,r3 : `r3 = 0x000000FF00000003 (saturation)`
saddc.b r1,r2,r3 : `r3 = 0x0000000700000003 , r4= 0x0000000100000000 (carry)`

Performance (FC0 seulement) :

Execution Unit : Add/Sub Unit
Latency : 1 cycle pour les données 8 bits, 2 cycles pour les données de 16 64 bits
Throughput : 1 opération par cycle par ASU.

1.1.2 addi

ADDition Immediate

addi Imm8, r2, r1
saddi Imm8, r2, r1

Calcule r1 = r2 + Imm8.

Cette instruction est similaire l'instruction add mais elle prend une des opérandes source de l'opcode (sans extension de signe). Il possède moins d'espace pour les options et les drapeaux ; c'est la raison pour laquelle l'usage du bit réservé est encore en discussion.

Remark : avec des opérandes larges, la latence peut tre plus élevée que prévu car l'additionneur utilisera le pipeline complet. De manière ajouter ou soustraire 1 d'un grand nombre (plus de 8 bits), il est recommané d'utiliser les instructions inc/dec (lorsqu'elles sont disponibles) car elles utilisent l'unité d'incrémentation qui possède moins de latence.

size :	8	4	8	6	6
bits :	31 24	23 20	19 12	11 6	5 0
function :	OP_ADDI	Flags	Imm8	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Définit la taille du paramètre
21	s- prefix	1 if set	Définit si l'opérande est SIMD
20	(none yet)	0	Reserved

Exemples :

R2 contient 0x00F80F00F045FF82 (dans un système 64 bits)

addi.b 0x87,r2,r3 : `r3 = 0x00F80F00F045FF09`
addi.d 0x87,r2,r3 : `r3 = 0x00F80F00F0450009`
saddi.b 0x87,r2,r3 : `r3 = 0x877F968777CC8609`
saddi.d 0x87,r2,r3 : `r3 = 0x017F0F87F0CC0009`

Performance (FC0 seulement) :

Execution Unit : Add/Sub Unit
Latency : 1 cycle pour les données 8 bits, 2 cycles pour les données de 16 64 bits
Throughput : 1 operation par cycle par ASU.

Updated (08/19/2002)

1.1.3 sub

SUBstraction

sub[b][f] r3, r2, r1
ssub[b][f] r3, r2, r1

Computes r1 = r2 - r3

sub performs an integer substraction of the two source operands (r3 - r2) and puts the result in destination operand (r1).

- The **size** flag indicates that **sub** performs the substraction on the whole operands or only on a part of the operands.
- The **SIMD** flag indicates that **sub** performs multiple substraction on parts of the operand (the size of these parts is defined by the **size** flags).
- The **floor** flag indicates that **sub** does not wrap" if the second operand is bigger than the first one.
- The **borrow** flag (same as carry) indicates that the borrow value is written to register number (r1^1). If no borrow has been generated, the neighbour register is cleared, otherwise the neighbour register is set to 1.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_SUB	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	(none yet)	0	Reserved
19	-f postfix	1 if set	Floor flag
18	-b postfix	1 if set	Borrow flag (**2r2w**)

Examples :

Scalar :

R1 contains 0x07 (we only consider the lower byte in the registers)
R2 contains 0x05

sub.b r1,r2,r3 : `r3 = 0xFE (default behaviour)`
subf.b r1,r2,r3 : `r3 = 0x00 (floor)`
subb.b r1,r2,r3 : `r3 = 0xFE, r4= 0xFF (borrow)`

SIMD :

R1 contains 0x0000000500000003 (in a 64-bit system)
R2 contains 0x0000000700000001

ssub.b r1,r2,r3 : `r3 = 0x0000000700000003 (default behaviour)`
ssubf.b r1,r2,r3 : `r3 = 0x0000000000000002 (floor)`
ssubb.b r1,r2,r3 : `r3 = 0x000000FE00000002, r4= 0x000000FF00000000 (borrow)`

Performance (FC0 only) :

Execution Unit : Add/Sub Unit
Latency : 1 cycle for 8-bit data, 2 cycles for 16-bit to 64-bit data
Throughput : 1 operation per cycle per ASU.

Revised (08/19/2002)

1.1.4 subi

SUBstraction Immediate

subi Imm8 , r2, r1
ssubi Imm8, r2, r1

Computes r1 = r2 - Imm8.

This instruction is similar to the sub" instruction but it takes one of the source operands from the opcode (Imm8) (without sign extension, use addi instead). It has less room for the options and flags, so the usage of the reserved bit is still being discussed.

Remark : with wide operands, the latency may be higher than expected because the adder would use the full pipeline. In order to add or substract 1 from a large number (more than 8 bits) it is recommended to use the inc/dec instructions (when available) because they use the increment unit which has a lower latency.

size :	8	4	8	6	6
bits :	31 24	23 20	19 12	11 6	5 0
function :	OP_SUBI	Flags	Imm8	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	(none yet)	0	Reserved

Performance (FC0 only) :

Execution Unit : Add/Sub Unit
Latency : 1 cycle for 8-bit data, 2 cycles for 16-bit to 64-bit data
Throughput : 1 operation per cycle per ASU.

Revised (08/19/2002)

1.1.5 mul

MULtiplication

mul[h][s] r3, r2, r1
smul[h][s] r3, r2, r1

Computes r1 = r2 x r3

mul performs an integer multiplication of the two source operands (r3 x r2) and puts the result in the destination operand (r1). The size flags indicate the size of the source operands.

- The **size** flag indicates that **mul** performs the multiplication on the whole operands or only on a part of the operands. It only puts the lower part of the result in the destination.
- The **SIMD** flag indicates that **mul** performs multiple multiplications on parts of the operand (the size of these parts is defined by the **size** flags).
- The **sign** flag indicates that **mul** will consider the operands as signed by extending the MSB.
- The **high** flag indicates that **mul** will also stores the higher part of the result in r1^1. It works in a similar fashion to the carry flag of the addition.

Remark : the multiplication computation is slow and heavy, try to use powers-of-two multipliers as to simply shift the source operand, which takes only a cycle to perform in the FC0.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_MUL	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	(none yet)	0	Reserved
19	-s postfix	1 if set	Sign flag
18	-h postfix	1 if set	High flag (**2r2w**)

Examples :

Scalar :

R1 contains 0x23 (we only consider the lower byte in the registers)
R2 contains 0x36

mul.b r1,r2,r3 : `r3 = 0x62 (default)`
mulh.b r1,r2,r3 : `r3 = 0x62 , r4 = 0x07 (High flag)`

SIMD :

R1 contains 0x00 00 00 00 00 00 00 00 (in a 64-bit system)
R2 contains 0x00 00 00 00 00 00 00 00

smul.b r1,r2,r3 : `r3 = 0x00 00 00 00 00 00 00 00`
smulh.b r1,r2,r3 : `r3 = 0x00 00 00 00 00 00 00 00 , r4 = 0x00 00 00 00 00 00 00 00`

Performance (FC0 only) :

Execution Unit : Integer Multiply Unit
Latency : unknown ATM, depends on the size of the operands.
Throughput : unknown ATM, probably 1 operation per cycle per IMU (pipelined multiplier).

Revised (08/19/2002)

1.1.6 muli

MULtiplication Immediate

muli imm8, r2, r1
smuli Imm8, r2, r1

Computes r1 = r2 x imm8.

This instruction is similar to the mul" instruction but it takes one of the source operands from the opcode (Imm8) and sign-extends it. It has less room for the options and flags, so the usage of the reserved bit is still being discussed.

Remark : the multiply computation is slow and heavy, try to use powers-of-two multipliers as to simply shift the source operand, which takes only a cycle to perform in the FC0.

size :	8		4		8		6		6	
bits :	31	24	23	20	19	12	11	6	5	0
function :	OP_MULI		Flags		Imm8		Reg 2		Reg 1	

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	(none yet)	0	Reserved

Performance (FC0 only) :

Execution Unit : Integer Multiply Unit
Latency : unknown ATM, depends on the size of the operands.
Throughput : unknown ATM, probably 1 operation per cycle per IMU (pipelined multiplier).

Revised (08/19/2002)

1.1.7 div

DIVision

div[r/rem][s] r3, r2, r1
sdiv[r/rem][s] r3, r2, r1

Computes r1 = r2 / r3

div performs an integer division of the two source operands (r3 / r2) and puts the result in destination operand (r1). The size defined by the size flags corresponds to the size of the source operands.

- The **size** flag indicates that **div** performs the division on the whole operands or only on a part of the operands.
- The **SIMD** flag indicates that **div** performs multiple division on parts of the operand (the size of these parts is defined by the **size** flags).
- The **Sign** flag determines if the operands should be treated as unsigned or signed values.
- The **Remainder** flag specifies that the remainder of the division is written to the register (r1^1).

This instruction triggers a math fault if the Reg2 operand is cleared (=0). This behaviour could be avoided with saturated arithmetics.

Remark : the division computation is slow and heavy, try to use powers-of-two divisors as to simply shift the source operand, which takes only a cycle to perform in the FC0.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_DIV	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	-r postfix (or -rem postfix)	1 if set	Remainder flag (**2r2w**)
19	-s postfix	1 if set	Sign flag
18	(none yet)	0	Reserved

Examples :

Scalar :

R1 contains 0x10 (we only consider the lower byte in the registers)
R2 contains 0x05

div.b r1,r2,r3 : `r3 = 0x03`
divr.b r1,r2,r3 : `r3 = 0x03 , r4 = 0x01`

Performance (FC0 only) :

Execution Unit : Integer Divide Unit
Latency : unknown ATM, depends on the size of the operands.
Throughput : unknown ATM, probably equal to the latency (not pipelined).

Revised (08/19/2002)

1.1.8 divi

DIVision Immediate

div[r/rem]i Imm8, r2, r1
sdiv[r/rem]i Imm8, r2, r1

Computes r1 = r2 / Imm8.

This instruction is similar to div" but the second operand is the sign-extended value of imm8. This will trigger a math trap if Imm8 is cleared (=0).

Remark : the division computation is slow and heavy, try to use powers-of-two divisors as to simply shift the source operand, which takes only a cycle to perform in the FC0.

size :	8		4		8		6		6	
bits :	31	24	23	20	19	12	11	6	5	0
function :	OP_DIVI		Flags		Imm8		Reg 2		Reg 1	

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	-r postfix (or -rem postfix)	1 if set	Remainder flag (**2r2w**)

Performance (FC0 only) :

Execution Unit : Integer Divide Unit
Latency : unknown ATM, depends on the size of the operands.
Throughput : unknown ATM, probably equal to the latency (not pipelined).

Revised (08/19/2002)

1.1.9 rem

REMainder

rem[s] r3, r2, r1
srem[s] r3, r2, r1

Computes r1 = r3 % r2

rem performs an integer remainder of the two source operands (r3 % r2) and puts the result in destination operand (r1).

- The **size** flag indicates that **rem** performs the remainder on the whole operands or only on a part of the operands.
- The **SIMD** flag indicates that **rem** performs multiple remainder on parts of the operand (the size of these parts is defined by the **size** flags).
- The **Sign** flag determines if the operands should be treated as unsigned or signed values.

This instruction triggers a math fault if the Reg2 operand is cleared (=0). This behaviour could be avoided with saturated arithmetics.

Remark : the remainder computation is slow and heavy, try to use powers-of-two remainder as to simply mask the MSB of the source operand, which takes only a cycle to perform in the FC0.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_REM	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	(none yet)	0	Reserved
19	-s postfix	1 if set	Signed flag
18	(none yet)	0	Reserved

Performance (FC0 only) :

Execution Unit : Integer Divide Unit
Latency : unknown ATM, depends on the size of the operands.
Throughput : unknown ATM, probably equal to the latency (not pipelined).

Revised (08/19/2002)

1.1.10 remi

REMainder Immediate

remi Imm8, r2, r1
sremi Imm8, r2, r1

Computes r1 = r2 % Imm8

remi performs an integer remainder of the two source operands (r2 % Imm8) and puts the result in destination operand (r1). Imm8 is sign extended (?).

- The **size** flag indicates that **rem** performs the division on the whole operands or only on a part of the operands.
- The **SIMD** flag indicates that **rem** performs multiple remainder on parts of the operand (the size of these parts is defined by the **size** flags).

This instruction triggers a math fault if the Imm8 operand is cleared (=0). This behaviour could be avoided with saturated arithmetics.

Remark : the remainder computation is slow and heavy, try to use powers-of-two remainders as to simply mask the MSB of the source operand, which takes only a cycle to perform in the FC0.

size :	8	4	8	6	6
bits :	31 24	23 20	19 12	11 6	5 0
function :	OP_REMI	Flags	Imm8	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	(none yet)	0	Reserved

Performance (FC0 only) :

Execution Unit : Integer Divide Unit
Latency : unknown ATM, depends on the size of the operands.
Throughput : unknown ATM, probably equal to the latency (not pipelined).

Revised (08/19/2002)

1.1.11 mac

Multiply and ACcumulate

mac[l/h][s] r3, r2, r1
smac[l/h][s] r3, r2, r1

Computes r1 = r1 + (r2 x r3)

mac performs an integer multiplication of the two source operands (r3 x r2) and adds the result to the destination operand (r1). The size flags indicate the size of the source operands, the granularity" of the destination operand is twice this size if the hardware can do it.

- The **size** flag indicates that **mac** performs the operation on the whole operands or only on a part of the operands.
- The **SIMD** flag indicates that **mac** performs multiple operations on parts of the operand (the size of these parts is defined by the **size** flags).
- The **sign** flag indicates that **mac** will consider the operands as signed by extending the MSB.
- The **high** flag indicates that **mac** will only operate on the high halfes of the operands (in SIMD mode).

Remark : This instruction is mostly used in computation kernels that involve some kind of convolution or frequency analysis. It will be extended later as the needs get clearer. The behaviour of the accumulation when the data overflow is still undefined so calibrate the input values so that the dynamic range is not exceeded in the computation loop. There is no sticky saturation" either.

Remark 2 : this instruction reads three operands and therefore is a **3r1w** operation that is not in the core. Its implementation depends on architectural parameters.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_MAC	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	(none yet)	0	Reserved
19	-s postfix	1 if set	Sign flag
18	-h postfix -l postfix	1 if set 0 if set	High flag Low flag

Example :

Scalar :

R1 contains 0x23 (we only consider the lower byte in the registers)
R2 contains 0x36
R3 contains 0x0136

mac.b r1,r2,r3 : `r3 = 0x0868`

Performance (FC0 only) :

Execution Unit : Integer Multiply Unit then Add/Sub Unit
Latency : unknown ATM, depends on the size of the operands.
Throughput : unknown ATM, probably 1 operation per cycle per IMU+ASU (pipelined multiplier and adder).

Updated (10/20/2002)

1.1.12 addsub

ADDition and SUBstraction

addsub[s] r3, r2, r1
saddsub[s] r3, r2, r1

Computes r1 = r3 + r2 and r1^1 = r3 - r2

- The **size** flag indicates that it performs the operation on the whole operands or only on a part of the operands.
- The **SIMD** flag indicates that it performs multiple operations on parts of the operand (the size of these parts is defined by the **size** flags).

Remark : This instruction is mostly used in computation kernels like FFT. It is included in the F-CPU because it allows the **2r2w** operation forms. Its implementation depends on architectural parameters, like the possibility to perform both addition and substraction at the same time.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_ADDSUB	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	(none yet)	0	Reserved
19	-s suffix	1 if set	Saturation flag
18	(none yet)	0	Reserved

Example :

R1 contains 0x36 (we only consider the lower byte in the registers)
R2 contains 0x23

addsub.b r1,r2,r3 : `r3 = 0x59 , r4 = 0xED`

Performance (FC0 only) :

Execution Unit : Add/Sub Unit
Latency : 1 cycle for 8-bit data, 2 cycles for 16-bit to 64-bit data
Throughput : 1 operation per cycle per ASU.

Revised (08/19/2002)

1.1.13 popcount

POPulation COUNT

popcount (r3,) r2, r1
spopcount (r3,) r2, r1

Computes r1 = nb_bits(r2)-r3 (with saturation)

popcount counts the number of set bits in r2. When the r3 field is not zeroed, the contents of the register r3 is substracted to the sum with saturation (the result doesn't wrap around if R3 is above the bit sum). The result is written to the destination operand (r1). The size flags indicate the size of the source operands.

- The **size** flag indicates that **popcount** performs the operation on the whole operands or only on a part of the operands.
- The **SIMD** flag indicates that **popcount** performs multiple operations on parts of the operand (the size of these parts is defined by the **size** flags).

Remark : This instruction is not going to be supported by the first F-CPU chips because it requires a specialized unit that is not yet designed and integrated in the FC0. It requires a separate Execution Unit that is a crossover between the Inc Unit and the Add/Sub Unit, but it does not provide enough useful instructions (as the Inc Unit does) to justify the high transistor count in FC0. Anyway, it is going to be implemented at one time or another and a lot of algorithms benefit from this instruction so the opcode is reserved for the future.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_POPC	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20-18	(none yet)	0	Reserved

Example :

R1 contains 0x0123456789ABCDEF

popcount r1,r2 : `r2 = 0x0000000000000020`

Performance (FC0 only) :

Execution Unit : Unknown
Latency : unknown, but it's O(log2(size)) if you wanted to know (just in case you're not a spook).
Throughput : unknown.

Revised (08/19/2002)

1.1.14 popcounti

POPulation COUNT with Immediate substract

popcounti (Imm8,) r2, r1
spopcounti (Imm8,) r2, r1

Computes r1 = nb_bits(r2)-Imm8 (with saturation)

popcounti counts the number of set bits in r2. When the Imm8 field is not zeroed, the value is substracted to the sum with saturation (the result doesn't wrap around if Imm8 is above the bit sum). The result is written to the destination operand (r1). The size flags indicate the size of the source operands.

- The **size** flag indicates that **popcount** performs the operation on the whole operands or only on a part of the operands.
- The **SIMD** flag indicates that **popcount** performs multiple operations on parts of the operand (the size of these parts is defined by the **size** flags).

Remark : This instruction is not going to be supported by the first F-CPU chips because it requires a specialized unit that is not yet designed and integrated in the FC0. It requires a separate Execution Unit that is a crossover between the Inc Unit and the Add/Sub Unit, but it does not provide enough useful instructions (as the Inc Unit does) to justify the high transistor count in FC0. Anyway, it is going to be implemented at one time or another and a lot of algorithms benefit from this instruction so the opcode is reserved for the future.

size :	8	4	8	6	6
bits :	31 24	23 20	19 12	11 6	5 0
function :	OP_POPCI	Flags	Imm8	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	(none yet)	0	Reserved

Example :

R1 contains 0x0123456789ABCDEF

popcounti 0, r1, r2 : `r2 = 0x0000000000000020`

Performance (FC0 only) :

Execution Unit : Unknown
Latency : unknown, but it's O(log2(size)) if you wanted to know (just in case you're not a spook).
Throughput : unknown.

Revised (08/19/2002)

1.1.15 inc

INCrement

inc r2, r1
sinc r2, r1

Computes r1 = r2 + 1

This instruction increments the source operand in a special unit that is designed for low latency when large data are processed. The value wraps around when reaching the maximum value.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_INC	Flags	0	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20-18	(none yet)	0	Reserved

Example :

R1 contains 0xFF05891213450100 (in a 64-bit system)

sinc.b r1,r2 : `r2 = 0x00068A1314460201`

Performance (FC0 only) :

Execution Unit : Increment Unit
Latency : 1 cycle
Throughput : 1 per cycle per IU.

Revised (08/19/2002)

1.1.16 dec

DECrement

dec r2, r1
sdec r2, r1

Computes r1 = r2 - 1

This instruction decrements the source operand in a special unit that is designed for low latency when large data are processed. The value wraps around when reaching the minimum value.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_DEC	Flags	0	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20-18	(none yet)	0	Reserved

Example :

R1 contains 0xFF05891213450100 (in a 64-bit system)

sinc.b r1,r2 : `r2 = 0xFE048811124400FF`

Performance (FC0 only) :

Execution Unit : Increment Unit
Latency : 1 cycle
Throughput : 1 per cycle per IU.

Revised (08/19/2002)

1.1.17 neg

NEGation

neg r2, r1
sneg r2, r1

Computes r1 = not(r2) + 1

This instruction negates the source operand in a special unit that is designed for low latency when large data are processed.

This instruction is designed to work in the 2s-complement numbering sytem (signed integer numbers) and is not subject to saturation/overflow problems. Notice though that negating -128" (if bytes are treated) will nicely fail to give you +128, without trapping. You've been warned.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_NEG	Flags	0	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20-18	(none yet)	0	Reserved

Example :

R1 contains 0xFF05891213450100 (in a 64-bit system)

sneg.b r1,r2 : `r2 = 0x01FB77EEEDBBFF00`

Performance (FC0 only) :

Execution Unit : Increment Unit
Latency : 1 cycle
Throughput : 1 per cycle per IU.

1.1.18 scan

SCAN

scan[n][r] r2, r1
sscan[n][r] r2, r1
lsb[0/1] r2, r1
msb[0/1] r2, r1
slsb[0/1] r2, r1
smsb[0/1] r2, r1

Computes r1 = scan_for_lsb(r2)

This instruction scans the source operand (r2) for the first set bit, starting from the LSB, and writes the position of this bit to the destination register (r1). If the source is cleared, the result is zero, otherwise the bit #0 is counted as position 1.

This instruction has options that bit reverse the source and/or complement the bits, so it can search for the last bit reset for example.

lsb1 is an alias for **scan**

lsb0 is an alias for **scann**
This instruction scans the source operand (r2) for the first reset bit, starting from the LSB, and writes the position of this bit to the destination register (r1). If the source is set (all ones), the result is zero, otherwise the bit #0 is counted as position 1.

msb1 is an alias for **scanr**
This instruction scans the source operand (r2) for the first set bit, starting from the MSB, and writes the position of this bit to the destination register (r1). If the source is cleared, the result is zero, otherwise the bit #0 is counted as position 1.

msb0 is an alias for **scannr**
This instruction scans the source operand (r2) for the first reset bit, starting from the MSB, and writes the position of this bit to the destination register (r1). If the source is set (all ones), the result is zero, otherwise the bit #0 is counted as position 1.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_SCAN	Flags	0	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	(none yet)	0	Reserved
19	-n postfix	1 if set	Negate the input
18	-r postfix	1 if set	Bit-Reverse the input

Examples :

R1 contains 0xFF05891213450100 (in a 64-bit system)

lsb1 r1, r2 : `r2 = 0x9`
lsb0 r1, r2 : `r2 = 0x1`
msb1 r1, r2 : `r2 = 0x40`
msb0 r1, r2 : `r2 = 0x38`

Performance (FC0 only) :

Execution Unit : Increment Unit
Latency : 1 cycle
Throughput : 1 per cycle per IU.

Revised (08/19/2002)

1.1.19 cmpl

CoMPare for Lower

cmpl r3, r2, r1
scmpl r3, r2, r1

Computes r1 = r2 < r3 ? -1 : 0

Compare the two source operands and sets or clear the destination register according to the result. This operation is performed in the Increment unit so no substraction is required and it is performed faster for large data. In order to compare for greater, simply swap the source operands or negate the result of CMPLE. The comparison is valid only for unsigned values (yet)

Remark : this instruction can't be used for IEEE floating point data (the comparison is not signed).

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_CMPL	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20-18	(none yet)	0	Reserved

Examples :

R1 contains 0x0000000500000003 (in a 64-bit system)
R2 contains 0x0000000700000001

scmpl.b r1,r2,r3 : `r3 = 0x00000000000000FF`
scmpl.b r2,r1,r3 : `r3 = 0x000000FF00000000`
cmpl r1,r2,r3 : `r3 = 0x0000000000000000`

Performance (FC0 only) :

Execution Unit : Increment Unit
Latency : 1 cycle
Throughput : 1 per cycle per IU.

1.1.20 cmple

CoMPare for Lower or Equal

cmple r3, r2, r1
scmple r3, r2, r1

Computes r1 = r2 <= r3 ? -1 : 0

Compare the two source operands and sets or clear the destination register according to the result. This operation is performed in the Increment unit so no substraction is required and it is performed faster for large data. In order to compare for greater or equal, simply swap the source operands or negate the result of CMPL. The comparison is valid only for unsigned values (yet)

Remark : this instruction can't be used for IEEE floating point data (the comparison is not signed).

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_CMPLE	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20-18	(none yet)	0	Reserved

Examples :

R1 contains 0x0000000500000003 (in a 64-bit system)
R2 contains 0x0000000700000001

scmpl.b r1,r2,r3 : `r3 = 0xFFFFFF00FFFFFFFF`
scmpl.b r2,r1,r3 : `r3 = 0xFFFFFFFFFFFFFF00`
cmpl r1,r2,r3 : `r3 = 0x0000000000000000`

Performance (FC0 only) :

Execution Unit : Increment Unit
Latency : 1 cycle
Throughput : 1 per cycle per IU.

Revised (08/19/2002)

1.1.21 cmpli

CoMPare for Lower with Immediate

cmpli Imm8, r2, r1
scmpli Imm8, r2, r1

Computes r1 = r2 < Imm8 ? -1 : 0

In SIMD mode each chunk is compared to the immediate value.

Similarly to CMPL, with an immediate operand (that is not sign-extended), compare the two source operands and sets or clear the destination register according to the result. The comparison is valid only for unsigned values (yet)

Remark : this instruction can't be used for IEEE floating point data (the comparison is not signed).

size :	8	4	8	6	6
bits :	31 24	23 20	19 12	11 6	5 0
function :	OP_CMPLI	Flags	Imm8	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	(none yet)	0	Reserved

Examples :

R1 contains 0x0000000500000003 (in a 64-bit system)

scmpli.b 0x04,r1,r2 : `r2 = 0x00000000000000FF`
cmpli 0x04,r1,r2 : `r2 = 0x0000000000000000`

Performance (FC0 only) :

Execution Unit : Increment Unit
Latency : 1 cycle
Throughput : 1 per cycle per IU.

1.1.22 cmplei

CoMPare for Lower or Equal with Immediate

cmplei Imm8, r2, r1
scmplei Imm8, r2, r1

Computes r1 = r2 <= Imm8 ? -1 : 0

In SIMD mode each chunk is compared to the immediate value.

Similarly to CMPLE, with an immediate operand (that is not sign-extended), compare the two source operands and sets or clear the destination register according to the result. The comparison is valid only for unsigned values (yet)

Remark : this instruction can't be used for IEEE floating point data (the comparison is not signed).

size :	8	4	8	6	6
bits :	31 24	23 20	19 12	11 6	5 0
function :	OP_CMPLEI	Flags	Imm8	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	(none yet)	0	Reserved

Examples :

R1 contains 0x0000000500000003 (in a 64-bit system)

scmplei.b 0x04,r1,r2 : `r2 = 0xFFFFFF00FFFFFFFF`
cmplei 0x04,r1,r2 : `r2 = 0x0000000000000000`

Performance (FC0 only) :

Execution Unit : Increment Unit
Latency : 1 cycle
Throughput : 1 per cycle per IU.

Revised (08/19/2002)

1.1.23 abs

ABSolute value

abs r2, r1
sabs r2, r1

Computes r1 = (not(r2) + 1) if MSB(r2) == 1

This instruction negates the source operand in a special unit that is designed for low latency when large data are processed. If the sign bit (MSB) of the source is set (the number is negative) then the value is written back to the register set, or else (it is already positive) the result is cancelled (that's how it works in scalar mode, not in SIMD mode...).

This instruction is designed to work in the 2s-complement number sytem (signed integer numbers) and is not subject to saturation/overflow problems. Notice though that negating -128" (if bytes are treated) will nicely fail to give you +128, without trapping. You've been warned.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_ABS	Flags	0	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20-18	(none yet)	0	Reserved

Example :

R1 contains 0xFF05891213450100 (in a 64-bit system)

sabs.b r1,r2 : `r2 = 0x0105771213450100`

Performance (FC0 only) :

Execution Unit : Increment Unit
Latency : 1 cycle
Throughput : 1 per cycle per IU.

Updated (10/20/2002)

1.1.24 nabs

Negative ABSolute value

nabs r2, r1
snabs r2, r1

Computes r1 = (not(r2) - 1) if MSB(r2) == 0

to avoid the 'sign surprise' when the argument is -2**(chunksize-1)

This instruction negates the source operand in a special unit that is designed for low latency when large data are processed. If the sign bit (MSB) of the source is set (the number is negative) then the value is written back to the register set, or else (it is already positive) the result is cancelled (that's how it works in scalar mode, not in SIMD mode...).

This instruction is designed to work in the 2s-complement number sytem (signed integer numbers) and is not subject to saturation/overflow problems. Notice though that negating -128" (if bytes are treated) will nicely fail to give you +128, without trapping. You've been warned.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_NABS	Flags	0	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20-18	(none yet)	0	Reserved

Example :

R1 contains 0xFF05891213450100 (in a 64-bit system)

nabs r1, r2 : `r2 = 0xFF05891213450100`

Performance (FC0 only) :

Execution Unit : Increment Unit
Latency : 1 cycle
Throughput : 1 per cycle per IU.

1.1.25 max

MAXimum

max r3, r2, r1
smax r3, r2, r1

Computes r1 = r3 if (r2 < r3) else r1 = r2

Compare the two source operands and writes the maximum of the two values to the destination register. The comparison is valid only for unsigned values (yet) so this instruction can't be used for IEEE floating point data.

size :	8		6		6		6		6	
bits :	31	24	23	18	17	12	11	6	5	0
function :	OP_MAX		Flags		Reg 3		Reg 2		Reg 1	

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20-18	(none yet)	0	Reserved

Examples :

R1 contains 0x0000000500000003 (in a 64-bit system)
R2 contains 0x0000000700000001

smax.b r1,r2,r3 : `r3 = 0x0000000700000003`
max r1,r2,r3 : `r3 = 0x0000000700000003`

Performance (FC0 only) :

Execution Unit : Increment Unit
Latency : 1 cycle
Throughput : 1 per cycle per IU.

1.1.26 min

MINimum

min r3, r2, r1
smin r3, r2, r1

Computes r1 = r3 if (r2 > r3) else r1 = r2

Compare the two source operands and writes the minimum of the two values to the destination register. The comparison is valid only for unsigned values (yet) so this instruction can't be used for IEEE floating point data.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_MIN	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20-18	(none yet)	0	Reserved

Examples :

R1 contains 0x0000000500000003 (in a 64-bit system)
R2 contains 0x0000000700000001

smin.b r1,r2,r3 : `r3 = 0x0000000500000001`
min r1,r2,r3 : `r3 = 0x0000000500000003`

Performance (FC0 only) :

Execution Unit : Increment Unit
Latency : 1 cycle
Throughput : 1 per cycle per IU.

Revised (08/19/2002)

1.1.27 maxi

MAXimum Immediate

maxi Imm8, r2, r1
smaxi Imm8, r2, r1

Computes r1 = Imm8 if (r2 < Imm8) else r1 = r2

Compare the two source operands and writes the maximum of the two values to the destination register. The comparison is valid only for unsigned values (yet) so this instruction can't be used for IEEE floating point data.

size :	8		4		8		6		6	
bits :	31	24	23	20	19	12	11	6	5	0
function :	OP_MAXI		Flags		Imm8		Reg 2		Reg 1	

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	(none yet)	0	Reserved

Examples :

R2 contains 0x0000000500000003 (in a 64-bit system)
smaxi.b 0x04,r2,r3 : `r3 = 0x0000000500000004`
maxi 0x04,r2,r3 : `r3 = 0x0000000500000003`

Performance (FC0 only) :

Execution Unit : Increment Unit
Latency : 1 cycle
Throughput : 1 per cycle per IU.

1.1.28 mini

MINimum Immediate

mini r3, r2, r1
smini Imm8, r2, r1

Computes r1 = Imm8 if (r2 > Imm8) else r1 = r2

Compare the two source operands and writes the minimum of the two values to the destination register. The comparison is valid only for unsigned values (yet) so this instruction can't be used for IEEE floating point data.

size :	8	4	8	6	6
bits :	31 24	23 20	19 12	11 6	5 0
function :	OP_MINI	Flags	Imm8	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	(none yet)	0	Reserved

Examples :

R2 contains 0x0000000500000003 (in a 64-bit system)
smini.b 0x04,r2,r3 : `r3 = 0x0000000400000003`
mini 0x04,r2,r3 : `r3 = 0x0000000000000004`

Performance (FC0 only) :

Execution Unit : Increment Unit
Latency : 1 cycle
Throughput : 1 per cycle per IU.

Revised (08/19/2002)

1.1.29 sort

SORT

sort r3, r2, r1
ssort r3, r2, r1

Computes { r1 = r3 , r1^1 = r2 } if (r2 > r3) else { r1 = r2 , r1^1 = r3 }

Compare the two source operands and writes the minimum of the two values to the destination register and the maximum to destination register+1. The comparison is valid only for unsigned values (yet) so this instruction can't be used for IEEE floating point data. This instruction is of the **2r2w** form.

size :	8		6		6		6		6	
bits :	31	24	23	18	17	12	11	6	5	0
function :	OP_SORT		Flags		Reg 3		Reg 2		Reg 1	

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20-18	(none yet)	0	Reserved

Examples :

R1 contains 0x0000000500000003 (in a 64-bit system)
R2 contains 0x0000000700000001

ssort.b r1,r2,r3 : `r3 = 0x0000000500000001 , r4 = 0x0000000700000003`
sort r1,r2,r3 : `r3 = 0x0000000500000003 , r4 = 0x0000000700000001`

Performance (FC0 only) :

Execution Unit : Increment Unit
Latency : 1 cycle
Throughput : 1 per cycle per IU.

1.2 Opérations optionnelles sur les systèmes de nombres logarithmiques

Ces opcodes dont réservés pour un support éventuel du Système de Nombres Logarithmiques (*LNS*) dans des versions du F-CPU qui sont trop petites pour supporter les opérations en virgule flottante. Notez que les nombres LNS ne peut actuellement pas excéder 32 bits mais cela pourra changer dans le futur. Le bit #11 qui est réservé dans la plupart des instructions sur les entiers peut tre utilisé pour spécifier que les données sont dans les formats LNS ou fractionnels mais cela n'a pas encore été décidé.

Instructions concernés par cette base :

Opcodes	Mnemonic
OP_LADD	ladd
	sladd
OP_LSUB	lsub
	slsub
OP_L2INT	l2int
	l2intr
	l2intt
	l2intf
	l2intc
	sl2int
	sl2intr
	sl2intt
	sl2intf
	sl2intc
OP_INT2L	int2l
	int2lr
	int2lt
	int2lf
	int2l
	sint2l
	sint2lr
	sint2lt
	sint2lf
	sint2lc

Subject to change (08/19/2002)

1.2.1 ladd

Lns ADDition

ladd r3, r2, r1
sladd r3, r2, r1

Computes r1 = r2 + r3 in the LNS format.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_LADD	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20-18	(none yet)	0	Reserved

Performance (FC0 only) :

Execution Unit : LNS Unit (not implemented)
Latency : unknown
Throughput : unknown.

1.2.2 lsub

Lns SUBstract

lsub r3, r2, r1
slsub r3, r2, r1

Computes r1 = r2 - r3 in the LNS format.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_LSUB	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20-18	(none yet)	0	Reserved

Performance (FC0 only) :

Execution Unit : LNS Unit (not implemented)
Latency : unknown
Throughput : unknown.

Subject to change (08/19/2002)

1.2.3 l2int

Lns to INT conversion

l2int[r/t/f/c] r2, r1
sl2int[r/t/f/c] r2, r1

Computes the equivalence of r2 in the LNS format to the integer format.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_L2INT	Flags	0	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	(none yet)	0	Reserved
19-18		see below	rounding mode

Rounding modes :

Value	Syntax	Rounding mode
00	-r	Nearest (default)
01	-t	Towards 0
10	-f	Towards -infinity
11	-c	Towards +infinity

Performance (FC0 only) :

Execution Unit : LNS Unit (not implemented)
Latency : unknown
Throughput : unknown.

Subject to change (08/19/2002)

1.2.4 int2l

INT to Lns conversion

int2l[r/t/f/c] r2, r1
sint2l[r/t/f/c] r2, r1

Computes the equivalence of r2 in the integer format to the LNS format.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_INT2L	Flags	0	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	(none yet)	0	Reserved
19-18		see below	rounding mode

Rounding modes :

Value	Syntax	Rounding mode
00	-r	Nearest (default)
01	-t	Towards 0
10	-f	Towards -infinity
11	-c	Towards +infinity

Performance (FC0 only) :

Execution Unit : LNS Unit (not implemented)
Latency : unknown
Throughput : unknown.

Chapitre 2

Opérations basées sur le Bit Shuffling

2.1 Opérations de Décalage et de Rotation de Base

Instructions concernées par cette base :

Opcodes	Mnemonic
OP_SHIFT	shiftl
	sshiftl
	sshiftlh
	shiftr
	sshiftr
	sshiftrh
	shiftdl
	sshiftdl
	sshiftdlh
	shiftdr
	sshiftdr
	sshiftdrh
OP_SHIFTI	shiftli
	sshiftli
	shiftri
	sshiftri
OP_SHIFTRA	shiftra
	sshiftra
	sshiftrah
OP_SHIFTRAI	shiftrai
	sshiftrai
OP_DSHIFTLI	dshiftli
	sdshiftli
OP_DSHIFTRI	dshiftri
	sdshiftri
OP_DSHIFTRA	dshiftra
	sdshiftra
OP_DSHIFTRAI	dshiftrai
	sdshiftrai
OP_ROT	rotl
	srotl
	srotlh
	rotr
	srotr
	srotrh

OP_ROTLI	rotli
	srotli
	rotri
	srotri
OP_BITOP	bitop
	bitopx
	bitops
	bitopc
	bitopt
	sbitop
	sbitopx
	sbitops
	sbitopc
	sbitopt
	bchg
	bset
	bclr
	btst
	sbchg
	sbset
	sbclr
	sbtst
OP_BITOPI	bitopi
	bitopix
	bitopis
	bitopic
	bitopit
	sbitopi
	sbitopix
	sbitopis
	sbitopic
	sbitopit
	bchgi
	bseti
	bclri
	btsti
	sbchgi
	sbseti
	sbclri
	sbtsti
OP_MIX	mixl
	mixh
OP_EXPAND	expandl
	expandh
OP_CSHIFT	cshiftl
	cshiftr
OP_SDUP	sdup

Updated (10/20/2002)

2.1.1 shift

logical SHIFT

shift[l/r][d][h] r3, r2, r1
sshift[l/r][d][h] r3, r2, r1

Postfix	Function	Compute
-l	left logical shift	r1 = r2 << r3
-r	right logical shift	r1 = r2 >> r3
-dl	put left logical shift into r1 put lost bits into r1^1	r1 = r2 << r3 r1^1 = lost_bits(r2)
-dr	put right logical shift into r1 and put lost bits into r1^1	r1 = r2 >> r3 r1^1 = lost_bits(r2)

The value of r3 is truncated to the number of bits needed by the bit shuffler unit.

If -h is specified in SIMD, that means r3 = sdup (r3)

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_SHIFT	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	-l or -r postfix	0 for -l 1 for -r	
19	-d postfix	1 if set	
18	-h postfix	1 if set	Defines if the operation is Half-SIMD (had effect only with SIMD operation)

Performance (FC0 only) :

Execution Unit : Bit Shuffling Unit
Latency : 1 cycle
Throughput : 1 per cycle per BSU.

2.1.2 shifti

SHIFT Immediate logic

shifti[r/l] Imm8, r2, r1
sshifti[r/l] Imm8, r2, r1

Postfix	Function	Compute
-l	left logical shift	r1 = r2 << Imm8
-r	right logical shift	r1 = r2 >> Imm8

The value of Imm8 is truncated to the number of bits needed by the bit shuffler unit.

size :	8		4		8		6		6	
bits :	31	24	23	20	19	12	11	6	5	0
function :	OP_SHIFTI		Flags		Imm8		Reg 2		Reg 1	

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	-l or -r postfix	0 for -l 1 for -r	

Performance (FC0 only) :

Execution Unit : Bit Shuffling Unit
Latency : 1 cycle
Throughput : 1 per cycle per BSU.

Revised (08/19/2002)

2.1.3 shiftra

SHIFT Right Arithmetic

shiftra r3, r2, r1
sshiftra[h] r3, r2, r1

Computes r1 = r2 >> r3 and preserve the sign.

The value of r3 is truncated to the number of bits needed by the bit shuffler unit.

If -h is specified in SIMD, that means r3 sdup (r3)

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_SHIFTRA	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20-19	(none yet)	0	Reserved
18	-h postfix	1 if set	Defines if the operation is Half-SIMD (had effect only with SIMD operation)

Performance (FC0 only) :

Execution Unit : Bit Shuffling Unit
Latency : 1 cycle
Throughput : 1 per cycle per BSU.

Revised (08/20/2002)

2.1.4 shiftrai

SHIFT Right Arithmetic Immediate

shiftrai Imm8, r2, r1
sshiftrai Imm8, r2, r1

Computes r1 = r2 >> Imm8 and preserve the sign

The value of Imm8 is truncated to the number of bits needed by the bit shuffler unit.

size :	8		4		8		6		6	
bits :	31	24	23	20	19	12	11	6	5	0
function :	OP_SHIFTRAI		Flags		Imm8		Reg 2		Reg 1	

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	(none yet)	0	Reserved

Performance (FC0 only) :

Execution Unit : Bit Shuffling Unit
Latency : 1 cycle
Throughput : 1 per cycle per BSU.

Updated (10/20/2002)

2.1.5 rot

logical ROTation

rot[r/l] r3, r2, r1
srot[r/l][h] r3, r2, r1

Postfix	Function	Compute
-l	left logical rot	r1 = r2 <@ r3
-r	right logical rot	r1 = r2 @> r3

The value of r3 is truncated to the number of bits needed by the bit shuffler unit.

If -h is specified in SIMD, that means r3 = sdup (r3)

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_ROT	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	-l or -r postfix	0 for -l 1 for -r	
19	(none yet)	0	Reserved
18	-h postfix	1 if set	Defines if the operation is Half-SIMD (had effect only with SIMD operation)

Performance (FC0 only) :

Execution Unit : Bit Shuffling Unit
Latency : 1 cycle
Throughput : 1 per cycle per BSU.

Revised (09/12/2002)

2.1.6 roti

ROTate Immediate

roti[l/r] Imm8, r2, r1
sroti[l/r] Imm8, r2, r1

Postfix	Function	Compute
-l	left logical rotation	r1 = r2 <@ Imm8
-r	right logical rotation	r1 = r2 @> Imm8

The value of Imm8 is truncated to the number of bits needed by the bit shuffler unit.

size :	8	4	8	6	6
bits :	31 24	23 20	19 12	11 6	5 0
function :	OP_ROTI	Flags	Imm8	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	-l or -r postfix	0 for -l 1 for -r	

Performance (FC0 only) :

*Execution Unit :*Bit Shuffling Unit
Latency : 1 cycle
Throughput : 1 per cycle per BSU.

2.1.7 bitop

single BIT OPeration

bitop[x/s/c/t] r3, r2, r1
sbitop[x/s/c/t] r3, r2, r1
bchg r3, r2, r1
bset r3, r2, r1
bclr r3, r2, r1
btst r3, r2, r1
sbchg r3, r2, r1
sbset r3, r2, r1
sbclr r3, r2, r1
sbtst r3, r2, r1

Computes r1 = F(function, r2, 1)

In the shifter, a 1 is shifted left r3 times and combined with the second operand (r2) according to the function F defined below :

Function number :	Logical function :	Operation :	Opcode :
00	AND	Bit Mask	btst or bitopt
01	ANDN	Bit Clear	bclr or bitopc
10	XOR	Bit Change	bchg or bitopx
11	OR	Bit Set	bset or bitops

The value of r3 is truncated to the number of bits needed by the bit shuffler unit.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_BITOP	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	(none yet)	0	Reserved
19-18	x, c, t or s	00-11	F

Example :

R1 contains 0x08
R2 contains 0xFF05891213450100 (in a 64-bit system)

bchg r1, r2, r3 : `r3 = 0xFF05891213450000`
bset r1, r2, r3 : `r3 = 0xFF05891213450100`
bclr r1, r2, r3 : `r3 = 0xFF05891213450000`
btst r1, r2, r3 : `r3 = 0x0000000000000100`

Performance (FC0 only) :

Execution Unit : Bit Shuffling Unit
Latency : 1 cycle
Throughput : 1 per cycle per BSU.

Revised (08/20/2002)

2.1.8 bitopi

single BIT OPeration Immediate

bitopi[x/s/c/t] Imm6, r2, r1
sbitopi[x/s/c/t] Imm6, r2, r1
bchgi Imm6, r2, r1
bseti Imm6, r2, r1
bclri Imm6, r2, r1
btsti Imm6, r2, r1
sbchgi Imm6, r2, r1
sbseti Imm6, r2, r1
sbclri Imm6, r2, r1
sbtsti Imm6, r2, r1

Computes r1 = F(function, r2, 1)

In the shifter, a 1 is shifted left Imm6 times and combined with the second operand (r2) according to the function F defined below :

F :	Logical function :	Operation :	Opcode :
00	AND	Bit Mask	btsti or bitopti
01	ANDN	Bit Clear	bclri or bitopci
10	XOR	Bit Change	bchgi or bitopxi
11	OR	Bit Set	bseti or bitopsi

The value of Imm6 is truncated to the number of bits needed by the bit shuffler unit.

size :	8	4	2	6	6	6
bits :	31 24	23 20	19 18	17 12	11 6	5 0
function :	OP_BITOPI	Flags	F	Imm6	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	(none yet)	0	Reserved
19-18	x, c, t or s	00-11	F

Example :

R2 contains 0xFF05891213450100 (in a 64-bit system)

bchgi 0x08,r2,r3 : `r3 = 0xFF05891213450000`
bseti 0x08,r2,r3 : `r3 = 0xFF05891213450100`
bclri 0x08,r2,r3 : `r3 = 0xFF05891213450000`
btsti 0x08,r2,r3 : `r3 = 0x0000000000000100`

Performance (FC0 only) :

Execution Unit : Bit Shuffling Unit
Latency : 1 cycle
Throughput : 1 per cycle per BSU.

Revised (08/20/2002)

2.1.9 dshiftra

Double SHIFT Right Arithmetic

sdhiftra r3, r2, r1
sdshiftra r3, r2, r1

Computes :
r1 = r2 >> r3 and preserve the sign,
r1^1 = lost_bit(r2).

The value of r3 is truncated to the number of bits needed by the bit shuffler unit.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_DSHIFTRA	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20-18	(none yet)	0	Reserved

Performance (FC0 only) :

Execution Unit : Bit Shuffling Unit
Latency : 1 cycle
Throughput : 1 per cycle per BSU.

Revised (08/20/2002)

2.1.10 dshiftrai

Double SHIFT Right Arithmetic Immediate

dshiftrai Imm8, r2, r1
sdshiftrai Imm8, r2, r1

Computes :
r1 = r2 >> Imm8 and preserve the sign,
r1^1 = lost_bit(r2).

The value of Imm8 is truncated to the number of bits needed by the bit shuffler unit.

size :	8		4		8		6		6	
bits :	31	24	23	20	19	12	11	6	5	0
function :	OP_DSHIFTRAI		Flags		Imm8		Reg 2		Reg 1	

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	(none yet)	0	Reserved

Performance (FC0 only) :

Execution Unit : Bit Shuffling Unit
Latency : 1 cycle
Throughput : 1 per cycle per BSU.

2.1.11 mix

MIX

mix[l/h] r3, r2, r1

Mix two halves of r3 and r2 and puts the result into r1.

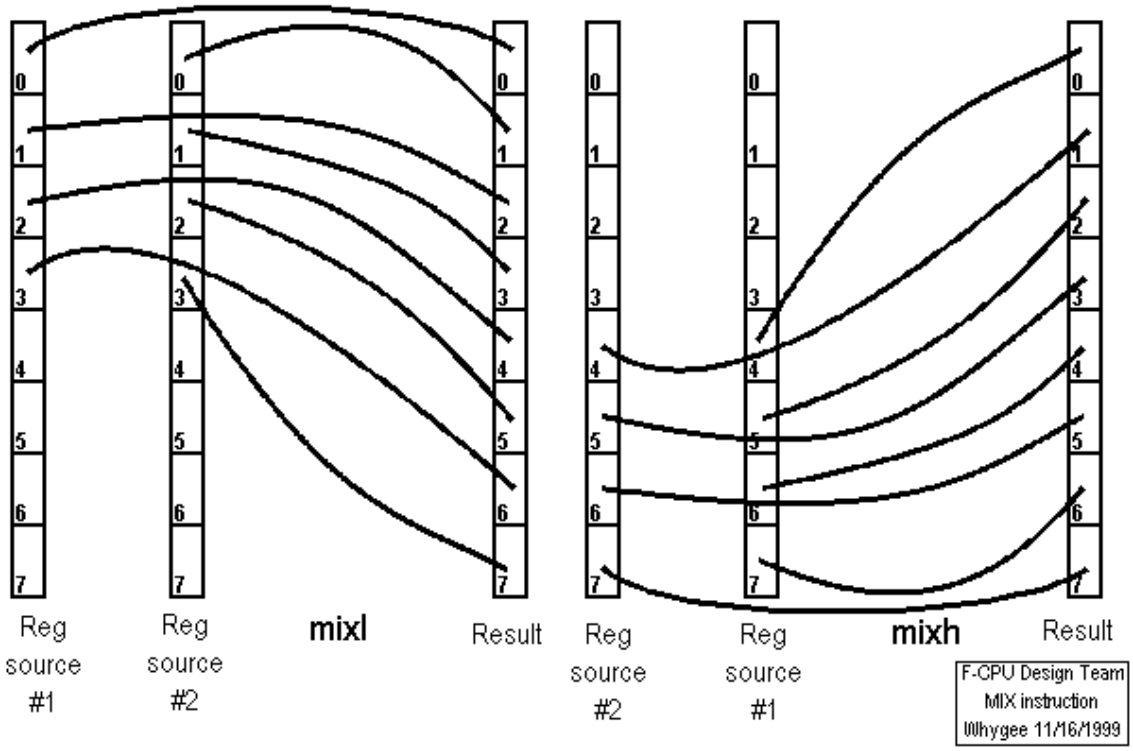


FIG. 2.1 – Description of the mix instruction

Depending on the **h** flag, the lower or higher part of r3 and r2 are interleaved. The size of the source chunks is determined by the size flags. This instruction is useful to interleave words in a butterfly" fashion or reverse a little matrix. Or simply it can be used to create an extended form of the result of an addition with carry.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_MIX	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix		Size flag
21-20	(none yet)	0	Reserved
19	-l or -h postfix	0 for -l 1 for -h	High flag
18	(none yet)	0	Reserved

Examples :

R1 contains 0x0001020304050607 (in a 64-bit system)
R2 contains 0x08090A0B0C0D0E0F

mixl.d r1,r2,r3 : `r3 = 0x04050C0D06070E0F`
mixh.d r1,r2,r4 : `r4 = 0x0001080902030A0B`

Performance (FC0 only) :

Execution Unit : Bit Shuffling Unit
Latency : 1 cycle
Throughput : 1 per cycle per BSU.

2.1.12 expand

EXPAND

expand[l/h] r3, r2, r1

Mix chunks of r3 and r2 and puts the result into two halves of r1.

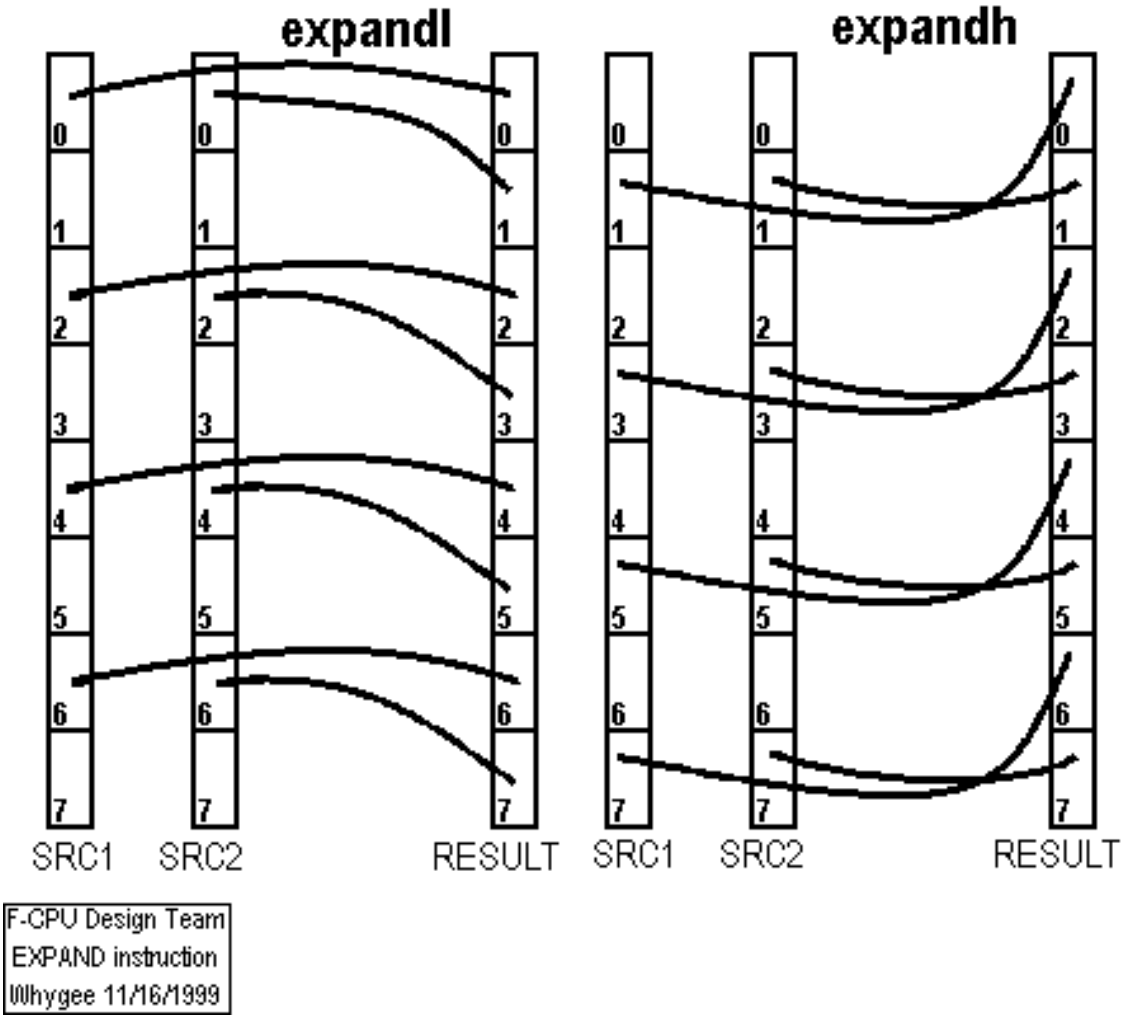


FIG. 2.2 – Description of the expand instruction

This is the reverse operation of the **mix** instruction. Depending on the **h** flag, the lower or higher part of r3 and r2 are interleaved. The size of the source chunks is determined by the size flags.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_EXPAND	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix		Size flag
21-20	(none yet)	0	Reserved
19	-l or -h postfix	0 for -l 1 for -h	High flag
18	(none yet)	0	Reserved

Examples :

R1 contains 0x0001020304050607 (in a 64-bit system)
R2 contains 0x08090A0B0C0D0E0F

expandl.b r1,r2,r3 : `r3 = 0x09010B030D050F07`
expandh.b r1,r2,r4 : `r4 = 0x08000A020C040E06`

Performance (FC0 only) :

Execution Unit : Bit Shuffling Unit
Latency : 1 cycle
Throughput : 1 per cycle per BSU.

Updated (09/08/2002)

2.1.13 cshift

Constant SHIFT

cshift[l/r] r2, r1

Postfix	Function	Compute
-l	constant left shift	r1 = r2 << 64
-r	constant right shift	r1 = r2 >> 64

This instruction must be used to pack/unpack over the 64 bits.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_CSHIFT	Flags	0	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-21	(none yet)	0	Reserved
20	-l or -r postfix	0 for -l 1 for -r	
19-18	(none yet)	0	Reserved

Example :

First pack :

mixl.q r3, r2, r1
mixl.q r4, r5, r6
cshiftl r1, r6

On a 128 bits F-CPU, r6 = (r4.q << 96) | (r5.q << 64) | (r3.q << 32) | r2.q

Then unpack :

expandl.q r6, r0, r2
expandh.q r6, r0, r3
cshiftr r6, r6
expandl.q r6, r0, r5
expandh.q r6, r0, r4

Performance (FC0 only) :

Execution Unit : Bit Shuffling Unit
Latency : 1 cycle
Throughput : 1 per cycle per BSU.

Revised (08/20/2002)

2.1.14 sdup

Simd DUPlication

sdup r2, r1

Duplicates the lower part of r2 and put the result in r1. The size of the destination SIMD chunks is determined by the size flags.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_SDUP	Flags	0	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix		Size flag
21-18	(none yet)	0	Reserved

Examples :

R1 contains 0x0001020304050607 (in a 64-bit system)

sdup.b r1,r2 : `r2 = 0x0707070707070707`
sdup.d r1,r3 : `r3 = 0x0607060706070607`
sdup.q r1,r4 : `r4 = 0x0405060704050607`

Performance (FC0 only) :

Execution Unit : Bit Shuffling Unit
Latency : 1 cycle
Throughput : 1 per cycle per BSU.

2.2 Opérations de Bit Shuffling optionnelles

L'Unité Bit Shuffling du F-CPU peut aussi effectuer d'autres opérations que la simple rotation ou le décalage de bits. Son objectif est de changer la position des bits, ce qui inclut aussi l'inversion de bit et d'octet et les opérations de mise en paquet SIMD.

Instructions concernées par cette base :

Opcodes	Mnemonic
OP_BITREV	bitrev
	bitrevo
	sbitrev
	sbitrevo
OP_BITREVI	bitrevi
	bitrevio
	sbitrevi
	sbitrevio
OP_BYTEREV	byterev
	sbyterev
OP_DBITREV	dbitrev
	sdbitrev
OP_DBITREVI	dbitrevi
	sdbitrevi

Updated (10/20/2002)

2.2.1 bitrev

BIT REVerse

bitrev[o] (r3,) r2, r1
sbitrev[o] (r3,) r2, r1

Computes :
r1 = bit_reverse(r2) >> (size - 1 - r3 % size)
r1^1 = r1 | (bit_reverse(r2) >> (size - 1 - r3 % size))

by default r3 == r0

R2 is first bit-reversed then shifted right size - r3 times.

If the -o flag is set, the result is combined by a OR with the content of r1 before it is written back to r1^1. This instruction is used to compute pointer updates in butterfly data structures, where r3 is the log2 of the size of the structure, r2 is the current index in the structure (always inferior to 2^r3) and r1 is the base pointer. It is a **3r1w** instruction form and therefore optional.

The value of r3 is truncated to the number of bits needed by the bit shuffler unit. Because it is aimed at pointer manipulation, the SIMD flag is not used. When a base address is used in conjunction with the -o flag, take care to align the base address to a boundray at least equivalent to the size of the data structure (just in case you weren't aware). In case the butterfly buffer is not aligned, an addition must be performed and the **bitrev** instruction must be used instead. The alignment of the final data is ensured by limiting the index : for example, a 256-byte buffer with 32-bit words requires that the index is between 0 and 63, so the final 2 LSB are always cleared.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_BITREV	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	-o postfix	1 if set	OR the result with the destination
19-18	(none yet)	0	Reserved

Example :

R1 contains 0x08 (a 256-byte buffer)
R2 contains 0x48 (the current index)
R3 contains 0xFF05891213450100 (the buffer base address)

bitrev r1, r2, r3 : `r3 = 0x0C`
bitrevo r1, r2, r3 : `r4 = 0xFF0589121345010C`

Performance (FC0 only) :

Execution Unit : Bit Shuffling Unit
Latency : 1 cycle
Throughput : 1 per cycle per BSU.

Updated (10/20/2002)

2.2.2 bitrevi

BIT REVerse Immediate

bitrevi[o] (Imm8,) r2, r1
sbitrevi[o] (Imm8,) r2, r1

Computes :
r1 = bit_reverse(r2) >> (size - 1 - Imm8 % size)
r1^1 = r1 | (bit_reverse(r2) >> (size - 1 - Imm8 % size))

The value of Imm8 is truncated to the number of bits needed by the bit shuffler unit.

size :	8		4		8		6		6	
bits :	31	24	23	20	19	12	11	6	5	0
function :	OP_BITREVI		Flags		Imm8		Reg 2		Reg 1	

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	-o postfix	1 if set	OR the result with the destination

Example :

R2 contains 0x48 (the current index)
R3 contains 0xFF05891213450100 (the buffer base address)

bitrevi 0x08,r2,r3 : `r3 = 0x0C`
bitrevio 0x08,r2,r3 : `r4 = 0xFF0589121345010C`

Performance (FC0 only) :

Execution Unit : Bit Shuffling Unit
Latency : 1 cycle
Throughput : 1 per cycle per BSU.

Revised (08/20/2002)

2.2.3 byterev

BYTe REVerse

byterev r2, r1
sbyterev r2, r1

Changes the endianness of r2 and puts the result into r1.

All the versions of the F-CPU may not support dual-endianness in the Load/Store unit, or simply the software may require internal operations of this kind. This is optional for the minimal systems, but yet useful in communication software. Remark, byterev.b has no use :-)

size :	8		6		6		6		6	
bits :	31	24	23	18	17	12	11	6	5	0
function :	OP_BYTEREV		Flags		0		Reg 2		Reg 1	

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix		Size flag
21	s- prefix	1 if set	Defines if the operation is SIMD

Examples :

R2 contains 0xFF05891213450100 (in a 64-bit system)
byterev.d r2,r3 : `r3 = 0xFF05891213450001`
byterev.q r2,r4 : `r4 = 0xFF05891200014513`
sbyterev.d r2,r3 : `r3 = 0x05FF128945130001`
sbyterev.q r2,r4 : `r4 = 0x128905FF00014513`

Performance (FC0 only) :

Execution Unit : Bit Shuffling Unit
Latency : 1 cycle
Throughput : 1 per cycle per BSU.

Revised (08/20/2002)

2.2.4 dbitrev

Double BIT REVerse

dbitrev (r3,) r2, r1
sdbitrev (r3,) r2, r1

Computes :
r1 = bit_reverse(r2) >> (size - r3 % size - 1),
r1^1 = bit_reverse(r2) >> (r3 % size + 1)

by default r3 == r0

R2 is first bit-reversed then shifted right r̃3 % size times.

The value of r3 is truncated to the number of bits needed by the bit shuffler unit. Because it is aimed at pointer manipulation, the SIMD flag is not used.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_DBITREV	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20-18	(none yet)	0	Reserved

Example :

R1 contains 0x08 (a 256-byte buffer)
R2 contains 0x48 (the current index)
R3 contains 0xFF05891213450100 (the buffer base address)

bitrev r1, r2, r3 : `r3 = 0x0C`
bitrevo r1, r2, r3 : `r4 = 0xFF0589121345010C`

Performance (FC0 only) :

Execution Unit : Bit Shuffling Unit
Latency : 1 cycle
Throughput : 1 per cycle per BSU.

Revised (08/20/2002)

2.2.5 dbitrevi

Double BIT REVerse Immediate

dbitrevi (Imm8,) r2, r1
dsbitrevi (Imm8,) r2, r1

Computes :
r1 = bit_reverse(r2) >> (size - Imm8 % size - 1),
r1^1 = bit_reverse(r2) >> (Imm8 % size + 1)

The value of Imm8 is truncated to the number of bits needed by the bit shuffler unit.

size :	8		4		8		6		6	
bits :	31	24	23	20	19	12	11	6	5	0
function :	OP_DBITREVI		Flags		Imm8		Reg 2		Reg 1	

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	(none yet)	0	Reserved

Example :

R2 contains 0x48 (the current index)
R3 contains 0xFF05891213450100 (the buffer base address)

bitrevi 0x08,r2,r3 : `r3 = 0x0C`
bitrevio 0x08,r2,r3 : `r4 = 0xFF0589121345010C`

Performance (FC0 only) :

Execution Unit : Bit Shuffling Unit
Latency : 1 cycle
Throughput : 1 per cycle per BSU.

Chapitre 3

Opérations logiques

3.1 Opération Logiques de Base

Instructions concernées par cette base :

Opcodes	Mnemonic
OP_LOGIC	or
	orn
	and
	andn
	xor
	nxor
	not
	nor
	nand
OP_LOGICI	ori
	andi
	andni
	xori
OP_CAND	cand.or
	cand.orn
	cand.and
	cand.andn
	cand.xor
	cand.nxor
	cand.not
	cand.nor
	cand.nand
OP_COR	cor.or
	cor.orn
	cor.and
	cor.andn
	cor.xor
	cor.nxor
	cor.not
	cor.nor
	cor.nand
OP_MUX	mux

Subject to change (10/17/2002)

3.1.1 logic

bitwise LOGIC

logic.[and/andn/xor/or/nor/xnor/orn/nand] r3, r2, r1
and r3, r2, r1
andn r3, r2, r1
xor r3, r2, r1
or r3, r2, r1
nor r3, r2, r1
xnor r3, r2, r1
orn r3, r2, r1
nand r3, r2, r1

Computes r1 = f(r2, r3) where f is a logic function whose truth table is defined in the flags.

size :	8		6		6		6		6	
bits :	31	24	23	18	17	12	11	6	5	0
function :	OP_LOGIC		Flags		Reg 3		Reg 2		Reg 1	

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21-19	and, andn, or, xor, nor, xnor, orn, nand or postfix	000-111	Define the operation
18	(none yet)	0	Reserved

Values	Function
000	and
001	andn
010	xor
011	or
100	nor
101	xnor
110	orn
111	nand

Performance (FC0 only) :

Execution Unit : ROP2 Unit
Latency : 1 cycle
Throughput : 1 per cycle per ROP2.

3.1.2 logici

bitwise LOGIC Immediate

logici.[and/andn/or/xor] Imm8, r2, r1
andi Imm8, r2, r1
andni Imm8, r2, r1
ori Imm8, r2, r1
xori Imm8, r2, r1

Computes r1 = f(Imm8, r2) where f is a logic function whose truth table is defined in the flags.

Because there is less room than in the register form of the instruction, the logic functions are reduced to 4. I have chosen to use the same logic functions as in the bitop instructions. Yet, the SIMD flag is cruelly missing. The function could maybe be included in the opcode.

size :	8	4	8	6	6
bits :	31 24	23 20	19 12	11 6	5 0
function :	OP_LOGICI	Flags	Imm8	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix		Size flags
21-20	and, andn, or or xor postfix	00-11	logic function

Values	Function
00	and
01	andn
10	xor
11	or

Performance (FC0 only) :

Execution Unit : ROP2 Unit
Latency : 1 cycle
Throughput : 1 result per cycle per ROP2.

3.1.3 cand

logic Combine AND

cand.[and/andn/xor/or/nor/xnor/orn/nand] r3, r2, r1
scand.[and/andn/xor/or/nor/xnor/orn/nand] r3, r2, r1

The **combine and** perform a **and** after the logic operation on all the bit of a word (depend on the size parameter). So you can test if the result is either 0 or not. This instruction is very usefull to make SIMD test for example.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_CAND	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20-18	and, andn, or, xor, nor, xnor, orn, nand postfix	000-111	Define the operation

Values	Function
000	and
001	andn
010	xor
011	or
100	nor
101	xnor
110	orn
111	nand

Example :

In this example we will replace all 'B' with 'b' in r2.

sdupi.b 'B', r3
sdupi.b 'b', r4

cand.xnor.b r2, r3, r5
mux.b r4, r5, r2
move r3, r2

Performance (FC0 only) :

Execution Unit : ROP2 Unit
Latency : 1 cycle
Throughput : 1 per cycle per ROP2.

3.1.4 cor

logic Combine OR

cor.[and/andn/xor/or/nor/xnor/orn/nand] r3, r2, r1
scor.[and/andn/xor/or/nor/xnor/orn/nand] r3, r2, r1

The combine or perform a or after the logic operation on all the bit of a word (depend on the size parameter). So you can test if the result is either 0 or not. This instruction is very usefull to make SIMD test for example.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_COR	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20-18	and, andn, or, xor, nor, xnor, orn, nand or postfix	000-111	Define the operation

Value	Syntax
000	and
001	andn
010	xor
011	or
100	nor
101	xnor
110	orn
111	nand

Example :

In this example we will do this bits movement :
0 -> 3
1 -> 2
2 -> 4
3 -> 7
4 -> 5
5 -> 1
6 -> 0
7 -> 6
for a purely random example.

The first step is to prepare the masks :
r3 = mask1 = 0x8040201008040201 ; // Original position
r5 = maks2 = 0x4001028020100408 ; // permuted mask
To understand this mask divide them by 8bits, and when a bit is set
it's mean bit came from or go to this position.

The byte to move are in r1 and the result will go in r7.

sdup.b r1, r2

cor.and.b r2, r3, r4
and r4, r5, r6

shri 32, r6, r7
or r6, r7, r7
shri 16, r6, r7
or r6, r7, r7
shri 8, r6, r7
or r6, r7, r7

Performance (FC0 only) :

Execution Unit : ROP2 Unit
Latency : 1 cycle
Throughput : 1 per cycle per ROP2.

Revised (08/20/2002)

3.1.5 mux

Mux

mux r3, r2, r1 smux r3, r2, r1

For each word (depending on the size parameter), select in r3, if r2 corresponding word contain all his bit are set, if not it take the word from r1 and the result is put in r1^1.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_MUX	Flags	0	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix		Size flag
21-18	(none yet)	0	Reserved

Examples :

In this example we will replace all 'B' with 'b' in r2.

sdupi.b 'B', r3
sdupi.b 'b', r4

cand.xnor.b r2, r3, r5
mux.b r4, r5, r2
move r3, r2

Performance (FC0 only) :

Execution Unit : ROP2 Unit
Latency : 1 cycle
Throughput : 1 per cycle per ROP2.

Chapitre 4

Opérations en Virgule Flottante

Il peut tre possible d'implementer un F-CPU avec des différents degrés de support d'opérations en virgule flottante ou de niveaux , en fonction des besoins et des technologies. Le niveau 0 de FP (Flotting Point ; Virgule Flottante) est l'abscence de matériel en Virgule Flottante et le niveau augmente lorsque le matériel offre plus de fonctionnalités.

Instruction \ Level	0	1	2	3
fadd		*	*	*
fsub		*	*	*
fmul		*	*	*
int2f/f2int		*	*	*
fiaprx, fsqrtiaprx		*	*	*
fcmple, fcmpl		*	*	*
fdiv, fsqrt			*	*
flog				*
fexp				*
fmac				*
faddsub				*

Le niveau FP d'un CPU doit tre lu dans le Registre Spécial associé avant de tenter d'exécuter les instructions en FP.

4.1 Niveau 1 des Opérations en Virgule Flottante

Instructions concernées par cette base :

Opcodes	Mnemonic
OP_FADD	fadd
	faddx
	sfadd
	sfaddx
OP_FSUB	fsub
	fsubx
	sfsub
	sfsubx
OP_FMUL	fmul
	fmulx
	sfmul
	sfmulx

OP_FIAPRX	fiaprx
	fiaprxx
	sfiaprx
	sfiaprxx
OP_FSQRTIAPRX	fsqrtiaprx
	fsqrtiaprxx
	sfsqrtiaprx
	sfsqrtiaprxx
OP_F2INT	f2int
	f2int
	f2int
	f2int
	f2int
	f2intx
	f2intrx
	f2inttx
	f2intfx
	f2intcx
	sf2int
	sf2intr
	sf2intt
	sf2intf
	sf2intc
	sf2intx
	sf2intrx
	sf2inttx
	sf2intfx
	sf2intcx
OP_INT2F	int2f
	int2fr
	int2ft
	int2ff
	int2fc
	int2fx
	int2frx
	int2ftx
	int2ffx
	int2fcx
	sint2f
	sint2fr
	sint2ft
	sint2ff
	sint2fc
	sint2fx
	sint2frx
	sint2ftx
	sint2ffx
	sint2fcx
OP_FCMPLE	fcmple
	fcmplex
	sfcmple
	sfcmplex
OP_FCMPL	fcmpl
	fcmplx
	sfcmpl
	sfcmplx

4.1.1 fadd

Floating point ADDition

fadd[x] **r3, r2, r1**
sfadd[x] **r3, r2, r1**

Computes r1 = r2 + r3 in IEEE-754 compliant format.

fadd performs a floating addition of the two source operands (r1 + r2) and puts the result in destination operand (r3). The operation should be compliant with the IEEE-754 format.

- The **size** flag indicates that **fadd** performs the addition on the whole operands or only on a part of the operands. This size flags is different from the integer size flag : only two values are currently assigned (01) for 64 bits and (00) for 32 bits.
- The **SIMD** flag indicates that **fadd** performs multiple addition on parts of the operand (the size of these parts is defined by the **size** flags).
- The **Exception** flag indicates if **fadd** should generate exceptions (when needed) in accordance to the IEEE-754 standard. When this flag is set, no exception is generated and the result is biased in an implementation-dependent way. The absence of this flag (by default) stops the pipeline until the FP execution unit confirms that an exception should be triggered, because the F-CPU doesn't implement imprecise exceptions.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_FADD	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.[??] postfix	00 : 32-bit FP 01 : 64-bit FP	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	-x postfix	1 to skip the tests	IEEE compliance flag
19-18	(none yet)	0	Reserved

4.1.2 fsub

Floating point SUBstraction

fsub[x] r3, r2, r1
sfsub[x] r3, r2, r1

Computes r1 = r2 - r3 in IEEE-754 compliant format.

fsub performs a floating substraction of the two source operands (r1 - r2) and puts the result in destination operand (r3). The operation should be compliant with the IEEE-754 format.

- The **size** flag indicates that **fsub** performs the operation on the whole operands or only on a part of the operands. This size flags is different from the integer size flag : only two values are currently assigned (01) for 64 bits and (00) for 32 bits.
- The **SIMD** flag indicates that **fsub** performs multiple substraction on parts of the operand (the size of these parts is defined by the **size** flags).
- The **Exception** flag indicates if **fsub** should generate exceptions (when needed) in accordance to the IEEE-754 standard. When this flag is set, no exception is generated and the result is biased in an implementation-dependent way. The absence of this flag (by default) stops the pipeline until the FP execution unit confirms that an exception should be triggered, because the F-CPU doesn't implement imprecise exceptions.

size :	8		6		6		6		6	
bits :	31	24	23	18	17	12	11	6	5	0
function :	OP_FSUB		Flags		Reg 3		Reg 2		Reg 1	

Flags	Syntax	Values	Function
23-22	.[??] postfix	00 : 32-bit FP 01 : 64-bit FP	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	-x postfix	1 to skip the tests	IEEE compliance flag
19-18	(none yet)	0	Reserved

Subject to change (08/20/2002)

4.1.3 fmul

Floating point MULtiplication

fmul[x] r3, r2, r1
sfmul[x] r3, r2, r1

Computes r1 = r2 x r3 in IEEE-754 compliant format.

fmul performs a floating multiplication of the two source operands (r1 x r2) and puts the result in destination operand (r3). The operation should be compliant with the IEEE-754 format.

- The **size** flag indicates that **fmul** performs the operation on the whole operands or only on a part of the operands. This size flags is different from the integer size flag : only two values are currently assigned (01) for 64 bits and (00) for 32 bits.
- The **SIMD** flag indicates that **fmul** performs multiple multiplication on parts of the operand (the size of these parts is defined by the **size** flags).
- The **Exception** flag indicates if **fmul** should generate exceptions (when needed) in accordance to the IEEE-754 standard. When this flag is set, no exception is generated and the result is biased in an implementation-dependent way. The absence of this flag (by default) stops the pipeline until the FP execution unit confirms that an exception should be triggered, because the F-CPU doesn't implement imprecise exceptions.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_FMUL	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.[??] postfix	00 : 32-bit FP 01 : 64-bit FP	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	-x postfix	1 to skip the tests	IEEE compliance flag
19-18	(none yet)	0	Reserved

4.1.4 f2int

Floating point to INTeger conversion

f2int[r/t/f/c][x] r2, r1
sf2int[r/t/f/c][x] r2, r1

f2int" converts a floating point number in register r2 into an integer number, according to the mode flags, and put it in register r1.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_F2INT	Flags	0	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.[??] postfix	00 : 32-bit FP 01 : 64-bit FP	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	-x postfix	1 to skip the tests	IEEE compliance flag
19-18		see below	Rounding modes

Rounding modes :

Value	Syntax	Rounding mode
00	-r	Nearest (default)
01	-t	Towards 0
10	-f	Towards -infinity
11	-c	Towards +infinity

4.1.5 int2f

INTeger to Floating point conversion

int2f[r/t/f/c][x] r2, r1
sint2f[r/t/f/c][x] r2, r1

int2f" converts an integer number in register r2 into a floating point number and put it in register r1.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_INT2F	Flags	0	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.[??] postfix	00 : 32-bit FP 01 : 64-bit FP	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	-x postfix	1 to skip the tests	IEEE compliance flag
19-18		see below	rounding mode

Rounding modes :

Value	Syntax	Rounding mode
00	-r	Nearest (default)
01	-t	Towards 0
10	-f	Towards -infinity
11	-c	Towards +infinity

4.1.6 fiaprx

Floating point Inverse APpRoXimation

fiaprx[x] **r2, r1**
sfiaprx[x] **r2, r1**

fiaprx approximates the inverse of r2 (*1/r2*) with the help of a hardwired lookup table and puts the result into r1. This operation is used at the beginning of a Newton-Raphson algorithm to compute a division. The accuracy of the lookup table depends on the application, and the number of NR iteration also depends on the desired accuracy and the size of the FP number.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_FIAPRX	Flags	0	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.[??] postfix	00 : 32-bit FP 01 : 64-bit FP	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	-x postfix	1 to skip the tests	IEEE compliance flag
19-18	(none yet)	0	Reserved

4.1.7 fsqrtiaprx

Floating point SQuare RooT Inverse APpRoXimation

fsqrtiaprx[x] **r2, r1**
sfsqrtiaprx[x] **r2, r1**

fsqrtiaprx approximates the inverse of the square root of r2 (*1/r2*) with the help of a hardwired lookup table and puts the result into r1. This operation is used at the beginning of a Newton-Raphson algorithm to compute a square root. The accuracy of the lookup table depends on the application, and the number of NR iteration also depends on the desired accuracy and the size of the FP number.

size :	8		6		6		6		6	
bits :	31	24	23	18	17	12	11	6	5	0
function :	OP_FSQRTIAPRX		Flags		0		Reg 2		Reg 1	

Flags	Syntax	Values	Function
23-22	.[??] postfix	00 : 32-bit FP 01 : 64-bit FP	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	-x postfix	1 to skip the tests	IEEE compliance flag
19-18	(none yet)	0	Reserved

4.1.8 fcmple

Float CoMPare for Lower or Equal

fcmple[x] r3, r2, r1
sfcmple[x] r3, r2, r1

Compare the two source operands and sets or clear the destination register according to the result. This operation is performed in the Increment unit so no substraction is required and it is performed faster for large data. In order to compare for greater or equal, simply swap the source operands or negate the result of FCMPL.

Remark : this instruction must be used for IEEE floating point data.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_FCMPLE	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.[??] postfix	00 : 32-bit FP 01 : 64-bit FP	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	-x postfix	1 to skip the tests	IEEE compliance flag
19-18	(none yet)	0	Reserved

4.1.9 fcmpl

Float CoMPare for Lower

fcmpl[x] r3, r2, r1
sfcmpl[x] r3, r2, r1

Compare the two source operands and sets or clear the destination register according to the result.

Remark : this instruction must be used for IEEE floating point data.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_FCMPL	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.[??] postfix	00 : 32-bit FP 01 : 64-bit FP	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	-x postfix	1 to skip the tests	IEEE compliance flag
19-18	(none yet)	0	Reserved

4.2 Niveau 2 des Opérations en Virgule Flottante

Instructions concernées par cette base :

Opcodes	Mnemonic
OP_FDIV	fdiv
	fdivx
	sfdiv
	sfdivx
OP_FSQRT	fsqrt
	fsqrtx
	sfsqrt
	sfsqrtx

4.2.1 fdiv

Floating point Division

fdiv[x] r3, r2, r1
sfdiv[x] r3, r2, r1

fdiv performs a floating division of the two source operands (r2 / r3) and puts the result in destination operand (r1). The operation should be IEEE-754 compliant.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_FDIV	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.[??] postfix	00 : 32-bit FP 01 : 64-bit FP	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	-x postfix	1 to skip the tests	IEEE compliance flag
19-18	(none yet)	0	Reserved

4.2.2 fsqrt

Floating point SQuare RooT

fsqrt[x] **r2, r1**
sfsqrt[x] **r2, r1**

fsqrt performs a floating point square root of the source operand (*r2*) and puts the result in destination operand (r1). The operation should be IEEE-754 compliant.

size :	8		6		6		6		6	
bits :	31	24	23	18	17	12	11	6	5	0
function :	OP_FSQRT		Flags		0		Reg 2		Reg 1	

Flags	Syntax	Values	Function
23-22	.[??] postfix	00 : 32-bit FP 01 : 64-bit FP	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	-x postfix	1 to skip the tests	IEEE compliance flag
19-18	(none yet)	0	Reserved

4.3 Niveau 3 des Opérations en Virgule Flottante

Instructions concernées par cette base :

Opcodes	Mnemonic
OP_FLOG	flog
	flogx
	sflog
	sflogx
OP_FEXP	fexp
	fexpx
	sfexp
	sfexpx
OP_FMAC	fmac
	fmacx
	sfmac
	sfmacx
OP_FADDSUB	faddsub
	faddsubx
	sfaddsub
	sfaddsubx

4.3.1 flog

Floating point LOGarithm

flog[x] (r3,) r2, r1
sflog[x] (r3,) r2, r1

Computes r1 = log2(r2) / r3

In the two operand form, r3 is set to 1.0.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_FLOG	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.[??] postfix	00 : 32-bit FP 01 : 64-bit FP	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	-x postfix	1 to skip the tests	IEEE compliance flag
19-18	(none yet)	0	Reserved

4.3.2 fexp

Floating point EXPonential

fexp[x] r3, r2, r1
sfexp[x] r3, r2, r1
fexp[x] r2, r1
sfexp[x] r2, r1

Computes r1 = exp2(r2 * r3)

In the 2 operand form, r3 is set to 1.0.

size :	8		6		6		6		6	
bits :	31	24	23	18	17	12	11	6	5	0
function :	OP_FEXP		Flags		Reg 3		Reg 2		Reg 1	

Flags	Syntax	Values	Function
23-22	.[??] postfix	00 : 32-bit FP 01 : 64-bit FP	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	-x postfix	1 to skip the tests	IEEE compliance flag
19-18	(none yet)	0	Reserved

4.3.3 fmac

Floating point Multiply and ACcumulate

fmac[x] r3, r2, r1
sfmac[x] r3, r2, r1

Computes r1 = r1 + (r2 x r3)

fmac performs a floating multiplication of the two source operands (r2 x r3) and adds the result to destination operand (r3). The operation should be IEEE-754 compliant.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_FMAC	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.[??] postfix	00 : 32-bit FP 01 : 64-bit FP	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	-x postfix	1 to skip the tests	IEEE compliance flag
19-18	(none yet)	0	Reserved

4.3.4 faddsub

Floating point ADDition and SUBstraction

faddsub[x] **r3, r2, r1**
sfaddsub[x] **r3, r2, r1**

Computes r1 = r2 + r3 and r1^1 = r2 - r3

faddsub is a **2r2w** instruction that performs both floating point addition and substraction of the two operands in IEEE-754 format.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_FADDSUB	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.[??] postfix	00 : 32-bit FP 01 : 64-bit FP	Defines the size parameter
21	s- prefix	1 if set	Defines if the operation is SIMD
20	-x postfix	1 to skip the tests	IEEE compliance flag
19-18	(none yet)	0	Reserved

Chapitre 5

Opérations d'Accès en Mémoire

5.1 Opérations d'Accès en Mémoire de base

Instructions concernées par cette base :

Opcodes	Mnemonic
OP_LOAD	load
	loade
OP_STORE	store
	storee
OP_LOADI	loadi
	loadie
OP_STOREI	storei
	storeie
OP_LOADF	loadf
	loadfe
OP_STOREF	storef
	storefe
OP_LOADIF	loadif
	loadife
OP_STOREIF	storeif
	storeife
OP_MADD	madd
OP_MSUB	msub
OP_MSHCHG	mshchg
OP_CSTORE	cstorez
	cstorem
	cstorel
	cstorenz
	cstorenm
	cstorenl
OP_CLOAD	cloadz
	cloadm
	cloadl
	cloadnz
	cloadnm
	cloadnl
OP_SS	ss
OP_SL	sl

Revised (08/20/2002)

5.1.1 load

LOAD a memory item into a register and adjust the Endianness

load[e][.0-.7] r2, r1

Performs r1 = endian(e,mem[r2]).

LOAD fetches the memory item pointed to by r2, changes the endianness according to the endian flag, and puts the result of the specified size into r1.

This instruction can trigger two exceptions (in order of decreasing priority) :

- **Alignment fault** : One or more LSB of the pointer are set. The number of significant LSB varies with the size flag. The F-CPU does not allow unaligned memory accesses.
- **Page fault** : The location referenced by r2 is not mapped in the internal TLB, and the OS kernel must update it, after checking for address range validity and access rights.

Prefetch :
In the case where the destination register is r0 (the NULL register), none of these exceptions are raised. This instruction form serves as a **prefetch** instruction that is issued several cycles before the actual reference is performed. The prefetch form prepares the memory hierarchy, the protection mechanisms and all the internal hidden flags for an eventual exception. The CPU can use the time between the prefetch and the actual fetch to prepare the page fault handler and the memory hierarchy so that the actual fetch will have almost no latency, whenever there is a fault or not.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_LOAD	Flags	0	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix		Size Flag
21	-e postfix	0 : little endian 1 : big endian	Endian Flag
20-18	-.0 .. -.7 postfix	000 .. 111	Reserved for the Stream Hint bits

Performance (FC0 only) :

Execution Unit : Load/Store Unit
Latency : 2 cycles if the item is already in the memory buffer, undetermined (but more) otherwise.
Throughput : 1 operation per cycle per LSU (peak).

5.1.2 store

adjust the Endianness and STORE the result in memory

store[e][.0-.7] r2, r1

Performs mem[r2] = endian(e,r1).

STORE adjusts the endianness of r1 according to the Endian flag and stores the item of the defined size to memory, at the location pointed to by r2.

This instruction can trigger two exceptions (in order of decreasing priority) :

- **Alignment fault** : One or more LSB of the pointer are set. The number of significant LSB varies with the size flag. The F-CPU does not allow unaligned memory accesses.
- **Page fault** : The location referenced by r2 is not mapped in the internal TLB, and the OS kernel must update it, after checking for address range validity and access rights.

The L/S Unit of the FC0 can perform the store operation with no latency for the entire pipeline when there is a free line in the memory bufer. If there are too much pending memory access requests, the pipeline must wait at the decoding stage for a memory buffer line to be freed.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_STORE	Flags	0	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix		Size Flag
21	-e postfix	0 : little endian 1 : big endian	Endian Flag
20-18	-.0 .. -.7 postfix	000 .. 111	Reserved for the Stream Hint bits

Performance (FC0 only) :

Execution Unit : Load/Store Unit
Latency : none if the memory buffer has a free slot, undetermined (but more) otherwise.
Throughput : 1 operation per cycle per LSU (peak).

Updated (08/20/2002)

5.1.3 loadi

LOAD a memory item into a register, adjust the Endianness and update the pointer with an Immediate number

loadi[e] Imm8, r2, r1

Performs :
r1 = endian(e,mem[r2])
r2 = r2 + Imm8

LOAD fetches the memory item pointed to by r2, changes the endianness according to the endian flag, puts the result of the specified size into r1. Curiously, this is a **1r2w** instruction. The Imm8 data is sign-extended with a ninth bit in the instruction word which also serves to predict in which direction the pointer moves.

This instruction can trigger two exceptions (in order of decreasing priority) :

- **Alignment fault** : One or more LSB of the pointer are set. The number of significant LSB varies with the size flag. The F-CPU does not allow unaligned memory accesses.
- **Page fault** : The location referenced by r2 is not mapped in the internal TLB, and the OS kernel must update it, after checking for address range validity and access rights.

Prefetch :
In the case where the destination register is r0 (the NULL register), none of these exceptions are raised. This instruction form serves as a **prefetch** instruction that is issued several cycles before the actual reference is performed. The prefetch form prepares the memory hierarchy, the protection mechanisms and all the internal hidden flags for an eventual exception. The CPU can use the time between the prefetch and the actual fetch to prepare the page fault handler and the memory hierarchy so that the actual fetch will have almost no latency, whenever there is a fault or not.

The behaviour of the pointer update obeys to the simplest arithmetics rules. No saturation is performed and the pointer will wrap around in memory.

After the addition is performed, the result will be submitted to the DTLB (Data virtual address Translation Lookaside Buffer) to check for the pointer validity in advance. As soon as the physical address is known, the processor can also prefetch the data if necessary, issuing a fetch command to the cache or the external memory. In the same time, the processor uses the sign bit of Imm8 in order to predict in which direction the pointer advances and prepare the memory buffer.

Because of the width of the immediate data, there is no room to specify the stream hint bits. It is therefore assumed that the processor will associate" the stream number with the pointer register thanks to a hidden status flag.

size :	8	4	8	6	6
bits :	31 24	23 20	19 12	11 6	5 0
function :	OP_LOADI	Flags	Imm8	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix		Size Flag
21	-e postfix	0 : little endian 1 : big endian	Endian Flag
20			Sign bit of Imm8

Performance (FC0 only) :

Execution Unit : Load/Store Unit and Add/Sub Unit.
Latency : 2 cycles if the item is already in the memory buffer, undetermined (but more) otherwise. The pointer update takes three cycles (2 ASU + 1 DTLB).
Throughput : 1 operation per cycle per LSU (peak).

5.1.4 storei

adjust the Endianness, STORE the result in memory and update the pointer with an Immediate number

storei[e] Imm8, r2, r1

Performs :
mem[r2] = endian(e,r1)
r2 = r2 + Imm8.

STORE adjusts the endianness of r1 according to the Endian flag and stores the item of the defined size to memory, at the location pointed to by r2 then adds Imm8 to the pointer. This is a **2r1w** instruction. The Imm8 data is sign-extended with a ninth bit in the instruction word which also serves to predict in which direction the pointer moves.

This instruction can trigger two exceptions (in order of decreasing priority) :

- **Alignment fault** : One or more LSB of the pointer are set. The number of significant LSB varies with the size flag. The F-CPU does not allow unaligned memory accesses.
- **Page fault** : The location referenced by r2 is not mapped in the internal TLB, and the OS kernel must update it, after checking for address range validity and access rights.

The L/S Unit of the FC0 can perform the store operation with no latency for the entire pipeline when there is a free line in the memory bufer. If there are too much pending memory access requests, the pipeline must wait at the decoding stage for a memory buffer line to be freed.

The behaviour of the pointer update obeys to the simplest arithmetics rules. No saturation is performed and the pointer will wrap around in memory.

After the addition is performed, the result will be submitted to the DTLB (Data virtual address Translation Lookaside Buffer) to check for the pointer validity in advance. In the same time, the processor can check the sign bit of Imm8 in order to predict in which direction the pointer advances and prepare the memory buffer.

size :	8	4	8	6	6
bits :	31 24	23 20	19 12	11 6	5 0
function :	OP_STOREI	Flags	Imm8	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix		Size Flag
21	-e postfix	0 : little endian 1 : big endian	Endian Flag
20			Sign bit of Imm8

Performance (FC0 only) :

Execution Unit : Load/Store Unit and Add/Sub Unit.
Latency : 2 cycles if the item is already in the memory buffer, undetermined (but more) otherwise. The pointer update takes three cycles (2 ASU + 1 DTLB).
Throughput : 1 operation per cycle per LSU (peak).

Revised (08/20/2002)

5.1.5 loadf, storef, loadif, storeif

loadf[e] **(r3,) r2, r1**
storef[e] **(r3,) r2, r1**
loadif[e] **(Imm8,) r2, r1**
storeif[e] **(Imm8,) r2, r1**

These instructions only differ from the normal opcodes by one flag which does not fit in the flag field (not enough room). This **F** flag is a hint for the onchip memory system, it influences the caching strategy. **F** means **Flush**, the data that is currently being processed (read or written) is not needed anymore, the CPU doesn't need to keep a copy onchip. This flag is meant to reduce the cache line thrashing whenever possible and increase the effective memory bandwidth.

More precisely, the semantic behind this flag is : the data is needed once". This is achieved inside the CPU by modifying the **caching strategy** with a cache line granularity. By default, when the **F** flag is omitted, the strategy is :
- keep the current line in the memory buffer
- when the line expires, flush it to the internal cache
- when the line expires in cache, flush it to the external memory

When the F flag is used in a load operation, the whole cache line is retrieved from external memory to the memory buffer. If possible, the succeeding memory location (it can be the precedent or next memory locations, depending on the sign of the pointer update) is retrieved. When the content of this second fetch begins to be used, this frees the first line, which is then used to fetch the third location. The two memory buffer lines continue this ping-pong as long as the stream goes on. The cache line is clearly flushed" but is not written back in memory because it is not modified.
With the store instruction, the operation doesn't necessarily need to begin with a fetch from memory. The F flag says that the line is flushed directly to the external memory instead of going to the internal cache memory.

The behaviour when loading to r0 with the F flag set is undetermined. The semantics don't go together, it would be prefetch something that will not be used after"... That's what i'd call waste time". So stay tunned.

Performance (FC0 only) :

Execution Unit : Load/Store Unit
Latency : undetermined
Throughput : 1 operation per cycle per LSU (peak).

5.1.6 madd

Memory ADDition

madd[.0-.7] r3, r2, r1

Computes r1 = r2 + r3

madd performs an pointer addition of an index (r3) plus a pointer (r2) and puts the result in the destination operand (r1).

If r2 didn't have the same **stream hints** as the one specified previously, the result is unknow.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_MADD	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-21	(none yet)	0	Reserved
20-18	-.0 .. -.7 postfix	000 .. 111	Stream Hints bits

Performance (FC0 only) :

Execution Unit : Add/Sub Unit + TLB
Latency : unknow.
Throughput : unknow.

5.1.7 msub

Memory SUBstraction

msub[.0-.7] r3, r2, r1

Computes r1 = r2 - r3

msub performs an pointer substraction of an index (r3) plus a pointer (r2) and puts the result in the destination operand (r1).

If r2 didn't have the same **stream hints** as the one specified previously, the result is unknow.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_MSUB	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-21	(none yet)	0	Reserved
20-18	-.0 .. -.7 postfix	000 .. 111	Stream Hints bits

Performance (FC0 only) :

Execution Unit : Add/Sub Unit + TLB
Latency : unknow.
Throughput : unknow.

Subject to change (09/14/2002)

5.1.8 mshchg

Memory Stream Hints CHange

mshchg[.0-.7] r1

Change the stream hints of a pointer without side effect.

It will perhaps lost LSU cache information, but you have garanty that your stream are correct. You must use it, if you want to change the stream hints of a pointer.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_MSHCHG	Flags	0	Reg 1	0

Flags	Syntax	Values	Function
23-21	(none yet)	0	Reserved
20-18	-.0 .. -.7 postfix	000 .. 111	Stream Hints bits

Performance (FC0 only) :

Execution Unit : Add/Sub Unit + TLB
Latency : unknow.
Throughput : unknow.

5.1.9 cstore

adjust the Endianness and Conditionnally STORE the result in memory

cstore[e][z/m/l/nz/nm/nl] r3, r2, r1

IF (condition(r3)) then mem[r2] = endian(e,r1).

CSTORE adjust the endianness of r1 according to the Endian flag and store the item of the defined size to memory, only if the condition is true, at the location pointed to by r2.

This instruction can trigger two exceptions only if the condition is true (in order of decreasiong priority) :

- **Alignment fault** : One or more LSB of the pointer are set. The number of significant LSB varies with the size flag. The F-CPU does not allow unaligned memory accesses.
- **Page fault** : The location referenced by r2 is not mapped in the internal TLB, and the OS kernel must update it, after checking for address range validity and access rights.

The L/S Unit of the FC0 can perform the store operation with no latency for the entire pipeline when there is a free line in the memory bufer. If there are too much pending memory access requests, the pipeline must wait at the decoding stage for a memory buffer line to be freed.

Because of the condition, there is no room to specify the stream hint bits. It is therefore assumed that the processor will associate" the stream number with the pointer register thanks to a hidden status flag.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_CSTORE	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix		Size Flag
21	-n	1 if negated	Negation of the condition
20-19	-z, -m, -l, -nz, -nm, -nl postfix	(see below)	Condition
18	-e postfix	0 : little endian 1 : big endian	Endian Flag

Performance (FC0 only) :

Execution Unit : Load/Store Unit
Latency : none if the memory buffer has a free slot, undetermined (but more) otherwise.
Throughput : 1 operation per cycle per LSU (peak).

Subject to change (10/12/2002)

5.1.10 cload

Conditionnally LOAD a memory item into a register, adjust the endianness

cload[e][z/m/l/nz/nm/nl] r3, r2, r1

IF (condition(r3)) then r1 = endian(e,mem[r2])

CLOAD fetches the memory item pointed to by r2 if the condition is true, changes the endianness according to the endian flag, puts the result of the specified size into r1.

This instruction can trigger two exceptions only if the condition is true (in order of decreasiong priority) :

- **Alignment fault** : One or more LSB of the pointer are set. The number of significant LSB varies with the size flag. The F-CPU does not allow unaligned memory accesses.
- **Page fault** : The location referenced by r2 is not mapped in the internal TLB, and the OS kernel must update it, after checking for address range validity and access rights.

The L/S Unit of the FC0 can perform the store operation with no latency for the entire pipeline when there is a free line in the memory bufer. If there are too much pending memory access requests, the pipeline must wait at the decoding stage for a memory buffer line to be freed.

Because of the condition, there is no room to specify the stream hint bits. It is therefore assumed that the processor will associate" the stream number with the pointer register thanks to a hidden status flag.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_CLOAD	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix		Size Flag
21	-n	1 if negated	Negation of the condition
20-19	-z, -m, -l, -nz, -nm, -nl postfix	(see below)	Condition
18	-e postfix	0 : little endian 1 : big endian	Endian Flag

Performance (FC0 only) :

Execution Unit : Load/Store Unit
Latency : none if the memory buffer has a free slot, undetermined (but more) otherwise.
Throughput : 1 operation per cycle per LSU (peak).

5.1.11 ss

Safe Store

ss r3, r2, r1

Chekin the value of r3 at address r2 and put the status in r1.

The system look to the flag on the item pointed by r2, and if some change appear on since we load it, ss will fail and notice it in r1. With this mecanism you can implement a CAS or CAS2 primitiv.

If the memory pointed by r2 is accessed without safe store and safe load, the result is unknow.

This instruction can trigger two exceptions (in order of decreasing priority) :

- **Alignment fault** : One or more LSB of the pointer are set. The number of significant LSB varies with the size flag. The F-CPU does not allow unaligned memory accesses.
- **Page fault** : The location referenced by r2 is not mapped in the internal TLB, and the OS kernel must update it, after checking for address range validity and access rights.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_SS	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix		Size Flag
21-18	(none yet)	0	Reserved

Example :

A "dumb" 'safe add' would look like :

loopentry r1
sl [r3], r4
add r5, r4, r4
ss r4, [r3], r2
loop r1, r2

Performance (FC0 only) :

Execution Unit : Safe Load/Store Unit + TLB
Latency : unknow.
Throughput : unknow.

5.1.12 sl

Safe Load

sl r2, r1

Checkout at address r2 then put the result in r1.

The system put a flag on the item pointed by r2 so that if some change appear on it, a safe store will fail. With this mecanism you can implement a CAS or CAS2 primitiv.

If the memory pointed by r2 is accessed without safe store and safe load, the result is unknow.

This instruction can trigger two exceptions (in order of decreasing priority) :

- **Alignment fault** : One or more LSB of the pointer are set. The number of significant LSB varies with the size flag. The F-CPU does not allow unaligned memory accesses.
- **Page fault** : The location referenced by r2 is not mapped in the internal TLB, and the OS kernel must update it, after checking for address range validity and access rights.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_SL	Flags	0	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix		Size Flag
21-18	(none yet)	0	Reserved

Example :

A "dumb" 'safe add' would look like :

loopentry r1
sl [r3], r4
add r5, r4, r4
ss r4, [r3], r2
loop r1, r2

Performance (FC0 only) :

Execution Unit : Safe Load/Store Unit + TLB
Latency : unknow.
Throughput : unknow.

5.2 Opérations Optionnelles d'Accès la Mémoire

Instructions concernées par cette base :

Opcodes	Mnemonic
OP_LOAD	load
	loade
OP_STORE	store
	storee
OP_CACHEMM	cachemm
	cachemmf
	cachemmp
	cachemml
	cachemmc
	cachemmfl
	cachemmfc
	cachemmpl
	cachemmpc
	cachemmflc
	cachemmlc

5.2.1 load

LOAD a memory item into a register, adjust the Endianness and update the pointer

load[e][.0-.7] r3, r2, r1

Performs :
r1 = endian(e,mem[r2])
r2 = r2 + r3

LOAD fetches the memory item pointed to by r2, changes the endianness according to the endian flag, puts the result of the specified size into r1. This version uses the same opcode as the core version but differs by the r3 parameter which makes it a **2r2w** instruction. In addition to the core version, the r3 parameter is used to update the r2 pointer by adding them in parallel with the memory operation. Note that if r3 contains 0, the core version is executed : the CPU checks the zero flags, instead of checking the register number.

This instruction can trigger two exceptions (in order of decreasing priority) :

- **Alignment fault** : One or more LSB of the pointer are set. The number of significant LSB varies with the size flag. The F-CPU does not allow unaligned memory accesses.
- **Page fault** : The location referenced by r2 is not mapped in the internal TLB, and the OS kernel must update it, after checking for address range validity and access rights.

Prefetch :
In the case where the destination register is r0 (the NULL register), none of these exceptions are raised. This instruction form serves as a **prefetch** instruction that is issued several cycles before the actual reference is performed. The prefetch form prepares the memory hierarchy, the protection mechanisms and all the internal hidden flags for an eventual exception. The CPU can use the time between the prefetch and the actual fetch to prepare the page fault handler and the memory hierarchy so that the actual fetch will have almost no latency, whenever there is a fault or not.

The behaviour of the pointer update obeys to the simplest arithmetics rules. No saturation is performed and the pointer will wrap around in memory.

After the addition is performed, the result will be submitted to the DTLB (Data virtual address Translation Lookaside Buffer) to check for the pointer validity in advance. As soon as the physical address is known, the processor can also prefetch the data if necessary, issuing a fetch command to the cache or the external memory. In the same time, the processor can check the sign of r3 in order to predict in which direction the pointer advances and prepare the memory buffer.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_LOAD	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix		Size Flag
21	-e postfix	0 : little endian 1 : big endian	Endian Flag
20-18	-0 .. -7 postfix	000 .. 111	Reserved for the Stream Hint bits

Performance (FC0 only) :

Execution Unit : Load/Store Unit and Add/Sub Unit.
Latency : 2 cycles if the item is already in the memory buffer, undetermined (but more) otherwise. The pointer update takes three cycles (2 ASU + 1 DTLB).
Throughput : 1 operation per cycle per LSU (peak).

5.2.2 store

adjust the Endianness, STORE the result in memory and update the pointer

store[e][.0-.7] r3, r2, r1

Performs :
mem[r2] = endian(e,r1)
r2 = r2 + r3.

STORE adjusts the endianness of r1 according to the Endian flag and stores the item of the defined size to memory, at the location pointed to by r2. This version uses the same opcode as the core version but differs by the r3 parameter which makes it a **3r1w** instruction. In addition to the core version, the r3 parameter is used to update the r2 pointer by adding them in parallel with the memory operation. Note that if r3 contains 0, the core version is executed : the CPU checks the zero flags, instead of checking the register number.

This instruction can trigger two exceptions (in order of decreasing priority) :

- **Alignment fault** : One or more LSB of the pointer are set. The number of significant LSB varies with the size flag. The F-CPU does not allow unaligned memory accesses.
- **Page fault** : The location referenced by r2 is not mapped in the internal TLB, and the OS kernel must update it, after checking for address range validity and access rights.

The L/S Unit of the FC0 can perform the store operation with no latency for the entire pipeline when there is a free line in the memory bufer. If there are too much pending memory access requests, the pipeline must wait at the decoding stage for a memory buffer line to be freed.

The behaviour of the pointer update obeys to the simplest arithmetics rules. No saturation is performed and the pointer will wrap around in memory.

After the addition is performed, the result will be submitted to the DTLB (Data virtual address Translation Lookaside Buffer) to check for the pointer validity in advance. In the same time, the processor can check the sign of r3 in order to predict in which direction the pointer advances and prepare the memory buffer.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_STORE	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix		Size Flag
21	-e postfix	0 : little endian 1 : big endian	Endian Flag
20-18	-.0 .. -.7 postfix	000 .. 111	Reserved for the Stream Hint bits

Performance (FC0 only) :

Execution Unit : Load/Store Unit and Add/Sub Unit.
Latency : 2 cycles if the item is already in the memory buffer, undetermined (but more) otherwise. The pointer update takes three cycles (2 ASU + 1 DTLB).
Throughput : 1 operation per cycle per LSU (peak).

Deprecated (08/20/2002)

5.2.3 cachemm

CACHE Memory Management

cachemm[f/p][l][c] r2, r1

Controls where a block of data or instructions is cached in the memory hierarchy. The block begins at the location pointed to by r2 and the size of the block is determined by r1.

This instruction should provide an universal way to control the caching mechanism of the FCPU accross all the variants that may appear. The instruction may operate on a page or cache line granularity, in an implementation dependent way. This instruction is purely a hint for the CPU that may or may not transfer data between different memory levels (that physically exist or not).

The instruction can act in either of these two directions :
- Flush : all the data present in the levels between the CPU and the parameter are flushed to at most this level. No data in the defined block is left in the above levels.
- Prefetch : loads the data belonging to the block in at least the level defined as parameter.

In addition, the L flag is used to influence the LRU tags in order to define the importance and the use of the block. L means Lock" and its absence unlocks the data from the level.

The C flag, when supported, tries to compress the block when it is flushed, or decompress it when it is loaded, with a dedicated hardware.

The status of this instruction could be read from a Special Register. This instruction is very important for memory management, and should be used when performing SMC or DMA for memory coherency. The OS can also lock the main TLB tables and the critical codes so that TLB replacement doesn't thrash the cache.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_CACHEMM	Flags	0	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix		Size Flag
21	-f postfix -p postfix	0 : Flush 1 : Prefetch	Direction flag
20		[l]	Lock. The data will be used a lot
19		[c]	De/Compress data on the fly
18-16		[0-7]	Memory level (see table below)

D	000	onchip Data L1 cache
I	001	onchip Instructions L1 cache
C	010	onchip unified Cache
	011	[unused]
U	100	offchip Unified cache
L	101	offchip Local memory
G	110	offchip Global memory
V	111	Virtual memory (hard disk)

Examples :

cachemmfg ra,rb flushes rb bytes starting at address ra from every memory level until global memory. Any cache (L1, L2, local...) containing data that belong to the block is updated in main memory and the corresponding cache spaces are freed (available for future use). this should be executed everytime the programer knows that he won't use a block of data until a certain moment, and the cache level is a hint for performance.

cachemmpu ra,rb copies the data block at address ra and size rb that is present in lower memory levels (virtual, global, local) to the unified offchip memory (at least", which means that some parts may be present closer to the processor).

Performance (FC0 only) :

Execution Unit : Load/Store Unit (?).
Latency : unknown, context dependent.
Throughput : one instruction at a time. And it's slow.

Chapitre 6

Opérations de déplacement de données

Ces instructions n'utilisent typiquement pas les Unités d'Execution.

6.1 Opérations de déplacement de données de base

Instructions concernées par cette base :

Opcodes	Mnemonic
OP_MOVE	move
	moves
	movez
	movem
	movel
	movenz
	movenm
	movenl
	movesz
	movesm
	movesl
	movesnz
	movesnm
	movesnl
OP_LOADCONS	loadcons
	loadconsp
	loadconss
OP_LOADADDR	loadaddr
	loopentry
OP_LOADADDRI	loadaddri
OP_GET	get
OP_PUT	put
OP_GETI	geti
OP_PUTI	puti

Updated (08/19/2002)

6.1.1 move

conditionally MOVE a register into another

move[z/m/l/nz/nm/nl] (r3,) r2, r1

IF (condition(r3)) then r1 = r2

The value of r3 is checked with the specified condition. If the condition is right, r2 is copied to r3 according to the size parameter. The condition is tested on the full register, and only the move uses the size flag.

Moving to r0 has no effect.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_MOVE	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22	.q, .d or .b postfix	*	Defines the size parameter
21	-n	1 if negated	Negation of the condition
20-19	-z, -m, -l, -nz, -nm, -nl postfix	(see below)	Condition
18	(none yet)	0	Reserved

Value	Syntax	Meaning
000	-z	zero
001		unassigned
010	-m	msb == 1
011	-l	lsb == 0
100	-nz	not zero
101		unassigned
110	-nm	msb == 0
111	-nl	lsb == 1

Examples :

r1 contains 0x0124356789ABCDEF
r2 contains 0xFEDCBA9876543210

move.b r1,r2 ; `r2 = 0xFEDCBA98765432FE`
if LSB r1 move.b r1,r2 ; `r2 = 0xFEDCBA98765432FE`
if MSB r1 move.b r1,r2 ; `r2 = 0xFEDCBA9876543210 (do nothing)`
if r1==0 move.b r1,r2 ; `r2 = 0xFEDCBA9876543210 (do nothing)`

Performance (FC0 only) :

Execution Unit : none
Latency : 2 cycle (Xbar)
Throughput : 1 per cycle per instruction.
Scheduling :

Cycle	1	2	3	4
Stage	Fetch	Decode/ Register Read	[Xbar]	[Register write]

6.1.2 loadcons

LOAD a CONSTant into a register

loadcons[.0-.3] Imm16, r1 loadcons Imm64, r1

r1(n) = Imm16

This instruction virtually shifts Imm16 by **n** multiples of 16 before writing the value to r1, leaving the other parts unchanged. The constant is not sign-exended (see **widen**). This instruction is used in groups as to create a large constant in a register.

If the developper didn't specify the multiples **n**, the assembler will divide the Imm64 in 4 equivalents parts for **loadcons.n**. It means that loadcons is an alias for 4 **loadcons.n**.

The architecture should ensure that a burst of LOADCONS does not stall the CPU. It is pipelinable in the FC0 so that a 64-bit constant only takes four cycles to complete.

size :	8		2		16		6	
bits :	31	24	23	22	21	6	5	0
function :	OP_LOADCONSP		Flags		Imm16		Reg 1	

Flags	Syntax	Values	Function
23-22	-.0 .. -.3 postfix	00 .. 11	Say where to put the Imm16.

Examples :

r1 contains 0x0124356789ABCDEF, the following instructions load 0xFEDCBA9876543210

loadcons.0 0x3210, r1 ; `r1 = 0x0123456789AB3210`
loadcons.1 0x7654, r1 ; `r1 = 0x0123456776543210`
loadcons.2 0xBA98, r1 ; `r1 = 0x0123BA9876543210`
loadcons.3 0xFEDC, r1 ; `r1 = 0xFEDCBA9876543210`

Performance (FC0 only) :

Execution Unit : none
Latency : 1 cycle (Xbar)
Throughput : 1 per cycle per instruction.

Revised (08/20/2002)

6.1.3 loadaddr

LOAD a relative ADDRess to a register

loadaddr[d] r2, r1

r1 = PC + 4 + r2, check the result in the D/I TLB and eventually prefetch the data.

If the **D**ata flag is set (1), the Data TLB is used instead of the Instruction TLB to check the pointer validity and the register is associated" to either the L/S Unit or the Fetcher unit on success. Eventually, the CPU can prefetch the pointed data or prefetch the TLB miss code.

The Size flag is not used, all registers are used in full length.

size :	8	1	10	6	6
bits :	31 24	23	22 12	11 6	5 0
function :	OP_LOADADDR	D	0	Reg 2	Reg 1

Performance (FC0 only) :

Execution Unit : Add/Sub Unit
Latency : 2 cycles if r2 != 0.
Throughput : 1 per cycle.

Updated (08/20/2002)

6.1.4 loadaddri

LOAD a relative ADDRess to a register with an Immediate offset

loadaddri[d][s] Imm16, r1
loadaddril[d] Imm64, r1

r1 = PC + 4 + Imm16, check the result in the D/I TLB and eventually prefetch the data.

If the **D**ata flag is set (1), the Data TLB is used instead of the Instruction TLB to check the pointer validity and the register is associated" to either the L/S Unit or the Fetcher unit on success. Eventually, the CPU can prefetch the pointed data or prefetch the TLB miss code.

This instruction is similar to loadaddr but uses an immediate offset. The **S** flag sign-extends the Imm16 data.

If the immediate is a relocation, it became a PC relative relocation.

If Imm64 is specified, the assembler will generate a serie of loadcons plus a loadaddr using always r1 and the s flag will be ignored.

The Size flag is not used, all registers are used in full length.

size :	8	1	1	16	6
bits :	31 24	23	22	21 6	5 0
function :	OP_LOADADDRI	D	S	Imm16	Reg 1

Performance (FC0 only) :

Execution Unit : Add/Sub Unit
Latency : 2 cycles.
Throughput : 1 per cycle.

6.1.5 get

GET the value of a special register and write it to a register.

get r2, r1

r1 = SPR(r2)

Get the Special Register at index r2 and put its content in register r1. The whole register gets dumped, there is no size flag.

Since protection is enforced through this kind of instruction, it may raise different exceptions if the access rights are not respected or if the SR number is not valid (supervisor or unimplemented). This is highly implementation dependent but a common and flexible definition will appear soon. Please refer to the manual.

Get and Put are atomic serializing" instructions that block the pipeline at the decoding stage until it is finished or the completion is safe. They are used to configure the CPU and the programming environment during the program start for example. The values of r2 are not yet defined and symbolic names are used instead (like the opcodes).

size :	8		12		6		6	
bits :	31	24	23	12	11	6	5	0
function :	OP_GET		0		Reg 2		Reg 1	

Performance (FC0 only) :

Execution Unit : none
Latency : unknown
Throughput : unknown (usually several cycles)

Revised (08/20/2002)

6.1.6 put

PUT the value of a register into a special register.

put r2, r1

SPR(r2) = r1

Read r1 and puts its value in the Special Register defined by r2. The whole register is used, there is no size flag.

Since protection is enforced through this kind of instruction, it may raise different exceptions if the access rights are not respected, if the SR number is not valid (supervisor or unimplemented) or if the put value does not correspond to the required format. This is highly implementation dependent but a common and flexible definition will appear soon. Please refer to the manual.

Get and Put are atomic serializing" instructions that block the pipeline at the decoding stage until it is finished or the completion is safe. They are used to configure the CPU and the programming environment during the program start for example. The values of r2 are not yet defined and symbolic names are used instead (like the opcodes).

size :	8	12	6	6
bits :	31 24	23 12	11 6	5 0
function :	OP_PUT	0	Reg 2	Reg 1

Performance (FC0 only) :

Execution Unit : none
Latency : unknown
Throughput : unknown (usually several cycles)

6.1.7 geti

GET the value of a special register defined by an Immediate value and write it to a register.

geti Imm16, r1

r1 = SPR(Imm16)

Get the Special Register at index Imm16 and put its content in register r1. The whole register gets dumped, there is no size flag.

Since protection is enforced through this kind of instruction, it may raise different exceptions if the access rights are not respected or if the SR number is not valid (supervisor or unimplemented). This is highly implementation dependent but a common and flexible definition will appear soon. Please refer to the manual.

Get(i) and Put(i) are atomic serializing" instructions that block the pipeline at the decoding stage until it is finished or the completion is safe. They are used to configure the CPU and the programming environment during the program start for example. The values of Imm16 are not yet defined and symbolic names are used instead (like the opcodes).

This version of GET is a shorthand for the instruction that limits the addressable range to the first 65536 Special Registers. The Core version (get) can address virtually ANY number of Special Registers through the use of a general register.

size :	8	2	16	6
bits :	31 24	23 22	21 6	5 0
function :	OP_GETI	0	Imm16	Reg 1

Performance (FC0 only) :

Execution Unit : none
Latency : unknown
Throughput : unknown (usually several cycles)

6.1.8 puti

PUT the value of a register to the special register Imm16.

puti Imm16, r1

SPR(Imm16) = r1

read r1 and puts its value in the Special Register defined by Imm16. The whole register is read, there is no size flag.

Since protection is enforced through this kind of instruction, it may raise different exceptions if the access rights are not respected or if the SR number is not valid (supervisor or unimplemented). This is highly implementation dependent but a common and flexible definition will appear soon. Please refer to the manual.

Get(i) and Put(i) are atomic serializing" instructions that block the pipeline at the decoding stage until it is finished or the completion is safe. They are used to configure the CPU and the programming environment during the program start for example. The values of Imm16 are not yet defined and symbolic names are used instead (like the opcodes).

This version of PUT is a shorthand for the instruction that limits the addressable range to the first 65536 Special Registers. The Core version (put) can address virtually ANY number of Special Registers through the use of a general register.

size :	8	2	16	6
bits :	31 24	23 22	21 6	5 0
function :	OP_PUTI	0	Imm16	Reg 1

Performance (FC0 only) :

Execution Unit : none
Latency : unknown
Throughput : unknown (usually several cycles)

6.2 Opérations de déplacement de données optionnelles

Instructions concernées par cette base :

Opcodes	Mnemonic
OP_LOADM	loadm
OP_STOREM	storem

6.2.1 loadm

LOAD Multiple registers from memory

loadm r3, r2, r1

load registers starting from r3 to r2 from the location in memory pointed by r1.

This instruction uses the SRB mechanism to load multiple contiguous registers from memory. This can be used during function epilogs where the classical RISC approach loads one register at a time.

The endianness of the operation is the endianness of the machine and the registers are full-length because it uses the SRB machinery verbatim. It benefits from the SRB reordering mechanism so when a value is needed but is not yet loaded, the SRB modifies the loading order. The operation is also performed in the background with few overhead for the application. Unlike the natural use of the SRB, this instruction can raise exceptions like all load operation.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	13 6	5 0
function :	OP_LOADM	0	Reg 3	Reg 2	Reg 1

Performance (FC0 only) :

Execution Unit : L/S Unit
Latency : unknown
Throughput : unknown

6.2.2 storem

STORE Multiple registers to memory

storem r3, r2, r1

store registers starting from r3 to r2 from the location pointed in memory by r1.

This instruction uses the SRB mechanism to store multiple contiguous registers to memory. This can be used during function prologs where the classical RISC approach stores one register at a time.

The endianness of the operation is the endianness of the machine and the registers are full-length because it uses the SRB machinery verbatim. It benefits from the SRB reordering mechanism so when a value is needed but is not yet loaded, the SRB modifies the loading order. The operation is also performed in the background with few overhead for the application. Unlike the natural use of the SRB, this instruction can raise exceptions like all load operation.

size :	8		6		6		6		6	
bits :	31	24	23	18	17	12	13	6	5	0
function :	OP_STOREM		0		Reg 3		Reg 2		Reg 1	

Performance (FC0 only) :

Execution Unit : L/S Unit
Latency : unknown
Throughput : unknown

Chapitre 7

Instructions de Contrôle de Flux d'Instructions

7.1 Instructions de Contrôle de Flux d'Instructions de base

Instructions concernées par cette base :

Opcodes	Mnemonic
OP_NOP	nop
OP_JMP	jmp
OP_LOOP	loop
OP_SYSCALL	syscall
OP_HALT	halt
OP_RFE	rfe

Revised (08/20/2002)

7.1.1 nop

No OPeration

Used for alignement purpose.

The reserved field might be used later for specifing the number of consecutive nops.

size :	8	24
bits :	31 24	23 0
function :	OP_NOP	0

7.1.2 jmp

JuMP

jmp[z/m/l/nz/nm/nl] r3, r2 [, r1]
jmp r2[, r1]

IF (condition(r3)) THEN
r1 = PC + 4
PC = r2

If no condition are specified, the jump will always be done. If the condition is verified for r3, the content of the Program Counter is saved

If the condition is verified for r3, the content of the Program Counter is saved to r1 and branches to the address pointed by r2. This instruction works like a mix between MOVE and LOAD.

If r1 is not cleared (written to register #0 which is hardwired to 0) the instruction is assimilated to a function call. The user is responsible of the stack frame". Otherwise (r1=0) the value of PC is lost and the instruction is a normal jump.

For several reasons, it is highly recommended that the destination of the jump is already associated to the register that contains the address, for example through a loadaddr instruction or by preserving r1 (overwriting it would cancel the association, for example when the stack" in the register set is flushed then loaded from memory). When association" is not certain or too early, it is recommended to prefetch the destination location a few tens of cycles in advance, otherwise it will result in a processor stall.

The Size flag is not used, all registers are used in full length.

This instruction can trigger two exceptions (in order of decreasing priority) :

- **Alignment fault** : One or 2 LSB of r2 are set, the address is not 4-byte aligned. The F-CPU does not allow unaligned memory instructions.
- **Page fault** : The location referenced by r2 is not mapped in the internal ITLB, and the OS kernel must update it, after checking for address range validity and access rights.

size :	8	6	6	6	6
bits :	31 24	23 18	17 12	11 6	5 0
function :	OP_JMP	Flags	Reg 3	Reg 2	Reg 1

Flags	Syntax	Values	Function
23-22		0	[undefined] branch probability hint
21	-n postfix	1 if negated	Negation of the condition
20-19	-z, -m, -l, -nz, -nm, -nl postfix	(see below)	Condition
18		0	(reserved)

Value	Syntax	Meaning
000	-z	zero
001		unassigned
010	-m	msb == 1
011	-l	lsb == 0
100	-nz	not zero
101		unassigned
110	-nm	msb == 0
111	-nl	lsb == 1

Performance (FC0 only) :

Execution Unit : none
Latency : 1 or 2 cycles if the destination is already in the memory buffer, undetermined (but much more) otherwise.
Throughput : unknown ATM.

Revised (08/20/2002)

7.1.3 loopentry

LOOP ENTRY point

loopentry r1

r1 = PC + 4 then check the result in the ITLB.

This instruction copies the address of the next instruction in r1 as to mark the entry point of a loop. A jmpa instruction will then use r1 instead of recalculating a relative offset at each loop iteration.

This instruction is a special case of the LOADADDR instruction with no D flag and no offset (r2 = 0).

The Size flag is not used, all registers are used in full length.

size :	8	18	6
bits :	31 24	23 6	5 0
function :	OP_LOADADDR	0	Reg 1

Performance (FC0 only) :

Execution Unit : none (theoretically)
Latency : none.
Throughput : 1 per cycle.

Revised (loop)

7.1.4 loop

LOOP to r2 if r1 has not expired.

loop r2, r1

r1 = r1 - 1
IF r1 != 0 THEN PC = r2

LOOP parallelly decrements r1 and checks the old value for nullity. If this old value was not zero, the CPU branches to [r2]. This is the simplest and fastest way to loop, the latency is typically 1 cycle and the operations overlap.

The couple LOOPENTRY/LOOP can code a WHILE or DO/WHILE loop where the loop count is known in advance. An initial value of r1 yields r1+1 iteration in a DO/WHILE loop, and the final value is -1.

The Size flag is not used, all registers are used in full length.

size :	8		12		6		6	
bits :	31	24	23	12	11	6	5	0
function :	OP_LOOP		0		Reg 2		Reg 1	

Performance (FC0 only) :

Execution Unit : Inc Unit (or Add/Sub Unit when unavailable)
Latency : 1 cycle
Throughput : 1 per cycle.

7.1.5 syscall

operating SYStem CALL

syscall Imm16, r1
trap Imm16, r1

jump in supervisor mode and execute the service # Imm16.

If the **T**rap flag is set, the user-mode application gives up his current time slice and requests a critical service (the SRB mechanism is triggered).

The r1 operand is not (yet) used, it is cleared. The argument is ignored by the hardware and may be used to encode information for system software. To retrieve the argument system software must load the instruction word from memory.

Typically, the service's entry point address is computed with the immediate value (shifted left by 6, as it appears in the instruction, as to have 16-instruction entry points) and added to a supervisor-mode Special Register. In the same time, the immediate value is compared with another Special Register which specifies the maximum number of implemented services, and a trap is triggered if there is an overflow.

size :	8	1	1	16	6
bits :	31 24	23	22	21 6	5 0
function :	OP_SYSCALL	T	0	Imm16	Reg 1

Performance (FC0 only) :

Execution Unit : none
Latency : unknown.
Throughput : unknown.

Revised (halt)

7.1.6 halt

HALT the CPU

halt

Goes idle until an exception occurs.

If in user mode, the application gives up his current time slice and the SRB mechanism is triggered to switch to the next task.

size :	8	24
bits :	31 24	23 0
function :	OP_HALT	0

Revised (rfe)

7.1.7 rfe

Return From Exception

rfe

Restore the precedent task.

At the end of an Interrupt Service Routine, an exception handler or a Supervisor service, this instruction flushes the current task and restores the precedent one with the SRB mechanism.

size :	8	24
bits :	31 24	23 0
function :	OP_RFE	0

7.2 Instructions de Contrôle de Flux d'Instructions Optionnelles

Instructions concernées par cette base :

Opcodes	Mnemonic
OP_SRB_SAVE	srb_save
OP_SRB_RESTORE	srb_restore
OP_SERIALIZE	serialize
	serializem
	serializes
	serializex
	serializems
	serializemx
	serializesx
	serializemsx

7.2.1 srb_save

use the SRB to SAVE the current task's context.

srb_save

Begins to save the current task in its dedicated CMB.

In prevision of a system call or in real-time sensitive conditions where the CPU is about to trigger the SRB and switch to another routine, it is recommended to execute srb_save in advance to speed the switch up.

size :	8	24
bits :	31 24	23 0
function :	OP_SRB_SAVE	0

7.2.2 srb_restore

use the SRB to RESTORE the last task's context.

srb_restore

Begins to restore the last task from its dedicated CMB.

In prevision of a return from exception or in prevision of a task switch involving SRB use, it is recommended to execute this instruction in advance so the CPU can prefetch the necessary data and reduce the switch latency.

size :	8	24
bits :	31 24	23 0
function :	OP_SRB_RESTORE	0

Performance (FC0 only) :

Execution Unit : none
Latency : Unknow
Throughput : none

7.2.3 serialize

stop the CPU while it is not flushed.

serialize[m][s][x][l]

Don't execute the next instruction before the internal state of the CPU has not reached the specified condition.

This instruction ensures that the specified units have completed processing any previously issued instruction. The current flags consider three conditions :
- Memory operations (all transactions are finished and there are free LSU lines)
- Executions units (there is no operation pending, the scoreboard is clear)
- SRB ready (the scoreboard has no SRB, or smooth context switch pending, so a loadm or storem instruction can be issued).

The condition is the logical product" (AND) of all the individual conditions : execution continues when all individual conditions are met.

size :	8	24
bits :	31 24	23 0
function :	OP_SERIALIZE	condition

Flags	Syntax	Values	Function
23	-m postfix	1 if used	Memory operations pending
22	-x postfix	1 if used	Execution Units busy
21	-s postfix	1 if used	SRB pending
20	-l postfix	1 if used	Flush all LSU L0 cache
19-0	(none yet)	0	Reserved

Performance (FC0 only) :

Execution Unit : none
Latency : depend on the pipeline state
Throughput : none

Septième partie

Programming the F-CPU

Chapitre 1

Introduction

As written before, programming the F-CPU has a different "taste" or "feeling" because of the particular processor structure and the design philosophy. Not only scheduling the individual instructions is important, but scheduling the use of each unit and the memory accesses is yet more important than ever before. Here, the key to performance, architectural simplicity and security in the FC0 is the use of many "speculative flags" that are not accessible to the user, but that influence the behaviour of the whole CPU. The F-CPU goes even further by allowing the user to explicitely indicate some "hints" like the "stream flags". An individual F-CPU can ignore these flags but their use will dramatically enhance the performance of the application if a few simple rules are respected, whatever the CPU type or core is used.

Chapitre 2

Call convention

Because of the large register bank (63 registers), we must use a good call convention to use them correctly. This call convention must be used by all the F-CPU librairies and by exported function in object and callback function.

Name	Number	Usage
zero	0	Constant 0
rv	1	Return value
a0 - a13	2 - 15	Argument 1 to argument 14
t0 - t16	16 - 31	Temporary 1 to temporary 17 (not preserved across call)
s0 - s26	32 - 58	Saved temporary (preserved across call)
plt	59	Pointer to the Procedure Linkage Table
got	60	Pointer to the Global Offset Table
fp	61	Frame pointer
sp	62	Stack pointer (used by function call)
ra	63	Return address (used by function call)

Chapitre 3

Memory convention

Because F-CPU is endian-less and have stream hints, we must specify some memory convention so that every programs, librairies and kernel. First by default all the memory access are done in little endian (That will ease porting software on F-CPU).

For stream hints, we currently didn't decide anything, so when you exchange memory pointer use memory stream hints 0.

Chapitre 4

Special Register Map

Special Register that define F-CPU core familly :

Name	Right	Value	Usage
FAMILY	Read only	0xFC0 for FC0	Give the name of the family of the core.
STEPPING	Read only	0x0A for alpha, version 0 0x0B for beta, version 0 0x01 for first release, version 0	Specify the version of the core.
URL_BASE	Read	0 - 2^64	Number of the SR where URL start
URL_SIZE	Read	0 - 2^64	Size of the URL

Special Register that define the core capacity :

Name	Right	Value	Usage
MAX_SIZE	Read only		unknow
SIZE_0	Read only		unknow
SIZE_1	Read only		unknow
SIZE_2	Read only		unknow
SIZE_3	Read only		unknow
MAX_CHUNK_SIZE	Read only		unknow

Special Register that offer counter capacity :

Name	Right	Value	Usage
CYCLE	Read only	0 - 2^64	Number of cycle since the boot
TIME_SLICE	Read/Write	0 - 2^64	Number of cycle that the task are alowed before trap (will certainly be used by kernel)
USER_SLICE	Read/Write	0 - 2^64	Same usage as TIME_SLICE (saved/restored during a SRB)
ALWAYS_SLICE	Read/Write	0 - 2^64	Same usage as TIME_SLICE (not saved/restored during a SRB)
NOTE_SLICE	Read/Write	0 - 2^64	Same usage as TIME_SLICE (set NOTE when equal 0 and when NOTE_SLICE is set, not saved/restored during a SRB)
NOTE	Read/Write	1 if set 2 if NOTE_SLICE has been set	Set by NOTE_SLICE
USER_NOTE_SLICE	Read/Write	0 - 2^64	Exactly same as NOTE_SLICE, but saved/restored during a SRB
USER_NOTE	Read/Write	0 - 2^64	Same as NOTE, but saved/restored during a SRB

Special Register that define where to find IRQ/TRAP/SYS handler :

Name	Right	Value	Usage
IRQ_BASE	Read/Write	pointer address	Where IRQ jump code are (16 instructions per IRQ before next)
IRQ_SIZE	Read	0 - 2^64	Number of supported IRQ
IRQ_STACK_BASE	Read/Write	pointer address	Need a stack to do a SRB if an IRQ occur during a TRAP/IRQ/SYSCALL
IRQ_STACK_SIZE	Read/Write	0 - 2^64	Stack size
TRAP_BASE	Read/Write	pointer address	Where TRAP jump code are (16 instructions per TRAP before next)
TRAP_SIZE	Read	0 - 2^64	Number of supported TRAP
TRAP_STACK_BASE	Read/Write	pointer address	Need a stack to do a SRB if a TRAP occur during a TRAP/IRQ/SYSCALL
TRAP_STACK_SIZE	Read/Write	0 - 2^64	Stack size
SYSCALL_BASE	Read/Write	pointer address	Where SYSCALL jump code are (16 instructions per SYSCALL before next)
SYSCALL_SIZE	Read/Write	0 - 2^64	Number of supported SYSCALL
SYSCALL_STACK_BASE	Read/Write	pointer address	Need a stack to do a SRB if a SYSCALL occur during a TRAP/IRQ/SYSCALL
SYSCALL_STACK_SIZE	Read/Write	0 - 2^64	Stack size

Special Register that manage access right to the system :

Name	Right	Value	Usage
TLB_OFF	Read and Write	1 if activate	Activate/Unactivate TLB.
IRQ_OFF	Read and Write	1 if activate	Activate/Unactivate IRQ.
READ_ONLY	Read/Write	1 if activate	Activate TRAP on access to read only special register
READ_WRITE	Read/Write	1 if activate	Activate TRAP on access to read/write special register
ASI	Read/Write	8 bits	Task identifier in the TLB
USER	Read/Write	1 if super-user	Say if we are in user/super-user mode.
CMB	Read/Write	0 - 2^64	pointer to the CMB structure of the current task
CMB_NEXT	Read/Write	0 - 2^64	pointer to the CMB structure of the task to switch to when TIME_SLICE equal 0
GET_CMB			unknow
GET_VM			unknow

Chapitre 5

Memory management

Subject to change (10/19/2002)

In the F-CPU the virtual memory management is done via the TLB mecanism in hardware. The F-CPU has around one hundred TLB entry, that give the correspondance from virtual to physical address.

When a address is accessed a trap append and the kernel must handle it. First he must look in his table to verify if the address exist, if it exist, he must replace an entry in the TLB by the needed one. If it didn't exist, a segfault append. When finished it must give the hand back to the program so that he can retry to access memory.

Current definition of the TLB entry :

Name	Size (bits)	Usage
TLB_ENTRY_VIRTUAL_ADDRESS	64	The beginning of the virtual address (Some lasts bit are forgotten corresponding to the page size)
TLB_ENTRY_PHYSICAL_ADDRESS	64	Beginning of the virtual address (The same last bits as the virtual address are forgotten)
TLB_ENTRY_ASI	8	Address space identifier (see SR_ASI)
TLB_SUPER_USER_RIGHT	5	Read/Write/Execution/Safe read/Safe store
TLB_USER_RIGHT	5	Read/Write/Execution/Safe read/Safe store
TLB_VALID	1	If set this entry is valid
TLB_DIRTY	1	Set if that page has been written to
TLB_USED	1	Set if that page has been accessed
TLB_PAGE_SIZE	6	The size of this page = 4Ko << TLB_PAGE_SIZE

Chapitre 6

Pseudo-superscalar

The FC0 uses a crossbar (”Xbar”) in order to reduce the register port number and provide a fast and universal register bypass mechanism. This central part of the FC0 is not complex but spans on a large part of the CPU. Each port has a relative high fanout and drives long wires, which justifies by itself the fact that the Xbar has its own cycle in the pipeline, when the operands are brought to the Execution Units and when the results are written back to the register set. This last part is used when ”bypassing” the register, with the help of the scoreboard that keeps trace of the use of the different Xbar channels.

In practice, the Xbar adds a 1-cycle latency to any normal computation instruction. This means that at least another independent instruction must be interleaved between two dependent instructions. From this point of view, programming a single-issue FC0 is similar to programming a 2- or 3-way superscalar processor, because of the very short pipeline stages. While this applies for the computational instructions, this doesn't apply to other data movement instructions that typically use the Xbar only once : they can be pipelined and don't suffer from instruction pairing restrictions as in superscalar CPUs.

The scoreboard checks the data dependencies and prevents multiple-cycle-latency instructions from giving wrong results. It is therefore very interesting to unroll loops at least twice, and if possible ”dephase” the different copies, as to get the most of the FC0 architecture. On the other hand, this reduces the number of available registers and a loop unrolling might not yield a good win with more than 4 copies.

Curiously, loop unrolling also applies to the pointers. Each new address value must be valid before entering the execution pipeline. One must duplicate the pointer registers because the [register+immediate offset] addressing mode is potentially dangerous. The ” pointer duplication ” technique must be used when a high memory bandwidth must be sustained because it benefits from the fully pipelined pointer update and checking mechanism. Again, at least 4 pointers are necessary to achieve the peak instant bandwidth. The problem is that only two registers can point to the same cache line at a time, the four register must reference two different streams.

Because of the previously explained mechanisms (speculative and background checking of the pointers in order to catch faultive instructions at decode stage) only post-increment addressing and direct register jump [/call] are supported, because the address is known before the instruction is executed. One must prefetch the locations from memory before use, by ”associating” a pointer register to a memory location. When this prefetch is scheduled enough in advance, this give the CPU time to check the pointer in the TLB, prefetch the necessary data from the memory hierarchy or prefetch the TLB replacement code if the pointer is invalid.

In ”vector loops” where linear arrays of data are processed, the prefetch mechanism is helped by the ” stream hint ” which help the CPU determine (following the architecture) which L/S Unit contains the data and/or which memory stride (or SDRAM bank) must be used. The ”cache hint”ed L/S instructions further reduce the cache memory thrashing by specifying which data should reside on-chip, which data can be flushed after use and which data must bypass the cache and go to the main memory.

It is also recommended to use the L/S post-incremented instructions in order to prefetch data that are not accessed linearly. For example, a program that reads non-contiguous operands in memory with only one pointer register (r2) can do the following :

loadi (operand2-operand1) ,r2,r3 .. (several instructions here) ..

loadi (operand3-operand2) ,r2,r3 .. (several instructions here) ..

loadi (operand4-operand3) ,r2,r3 .. (several instructions here) ..

loadi (operand5-operand4) ,r2,r3

Of course, several conditions must be present : the difference between the addresses must be known and fit in the immediate field. If the difference is below $2\hat{1}6$, one can use a loadconsx inside a stall cycle. Ultimately, the data addresses or the access order can be changed.

[to be continued ! yg.]

Huitième partie

Index

Index