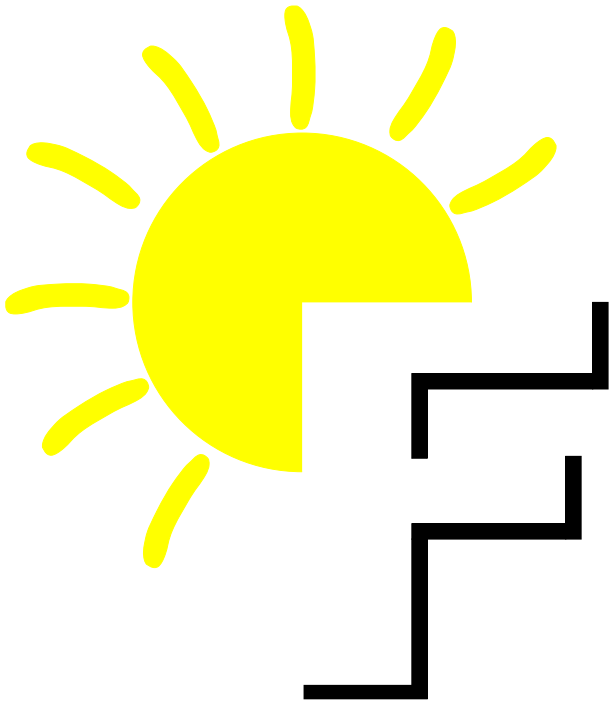
Freedom CPU Project

F-CPU Design Team

traduction du Patch YG 2001.1.14

FCPU MANUAL REV. 0.2.2 β



"Design and let design"

Please venez nous voir sur `http://www.f-cpu.org` et envoyez vos commentaires sur la liste de diffusion F-CPU
`f-cpu@egroups.com`.

0.1 Copyright et licence de distribution :

Ce manuel est distribué sous les termes de la GFDL ou "GNU Free Documentation License", dont le texte peut être trouvé sur le site web GNU (`http://www.gnu.org`). Une copie de cette licence est inclue dans ce paquetage. (fdl.htm).

0.2 Avant-propos :

Tout ce qui se trouve dans ce document est *sérieusement préliminaire* et peut changer sans préavis. Please restez en contact avec le groupe sur la liste de diffusion et contrôlez les dernières mises à jour sur le site web F-CPU officiel.

Ce document est (C) 1999-2001 The F-CPU Group Design Team et est un travail collaboratif. Tout le monde peut participer à l'effort F-CPU et devenir un membre du groupe en s'incrivant sur la liste de diffusion et en prenant par aux discussions. Vous pouvez même participer sans vous inscrire mais vous êtes fortement incités à soumettre vos idées et reporter les erreurs. Nous sommes conscients que ce document contient des erreurs mais nous travaillons dessus constamment.

Ce manuel a été transformé en plusieurs formats de fichiers mais il peut y manquer des parties ou contenir des erreurs. Il est très incomplet, même s'il commence à devenir énorme!

0.3 Historique des versions :

* Créé le 8 Juillet 1999 par `whygee@f-cpu.org` (Yann Guidon) á partir d'extraits des RFC de Mathias Brossard.
* 10 Juillet : quelques ajouts.
* 11 : adapté pour la conversion vers PDF avec HTMLDOC.
* 2/8, 8/8, 9/8, 13/8 : ajout supplémentaire.
* 25/8 : retravaillé un peu (pourquoi-comment, TTA, endianness, mémoire paginée, jump station...)
* 5/11 : mélange avec d'autre contenus autres qu'architecture.
* 16/11 : retour au format HTML.
* 27/2 : version majeure pour le codage d'instruction. Imm6 disparait et la plupart des anciennes erreurs et fautes de frappe.
* 15/3 : Adapté pour le macro processing par CPP
* 18/12/2000 : Olivier Jean a finalement publié la version Latex du manuel
* 24/12/2000 : YG l'a patché. très incomplet!
* 30/12/2000 : YG l'a repatché à Berlin.
* */01/2001 : début de la traduction française, redessinage des illustrations, mise à jour majeure...

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

0.4 Manque :

* jeu d'instruction découpé en fichiers séparés dans un répertoire spécial.
* carte du jeu d'instruction dans l'ordre alphabétique
* table du jeu d'instruction, triée par valeur opcode en hexa.
* exemples dans les descriptions d'instructions
* IRQ/traps
* carte SR
* 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`

Les listes de diffusion :

* `http://www.eGroups.com/list/f-cpu` (liste principale)
* `http://www.eGroups.com/list/f-cpu_france` (liste française)
* `http://www.eGroups.com/list/fcpu-ger` (liste allemande)

Table des matières

Table des figures

Liste des tableaux

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 gral" 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).
* 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 persévérons 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 autres facteurs.

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 mails 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. Une grande partie de cette infrastructure a été utilisée pour développer GNU/Linux et est de nos jours basé sur GNU/Linux, ((((((((((BTW)))))))))).

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'idée de base est de ne pas dire au CPU
ce qu'il doit faire des données, mais seulement où les mettre. Alors, en positionnant les
données aux bons endroits, vous récupérez magiquement de nouvelles données à d'autres
endroits qui résultent d'opérations effectuées sur les anciènnes données. 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'intérêt du jeu d'instruction (si on peut dire, car le TTA n'a qu'une instruction : MOV)
est sur la donnée elle-même, à l'opposé des opérations que vous faites sur la donnée. Vous
spécifiez simplement les adresses puis vous cartographiez celles-ci en des fonctions comme
ADD ou DIV.

C'est l'idée de base. Je devrais commencer en spécifiant que je me concentre ici sur le
traitement global et j'ignore temporairement les choses comme les interruptions. De cette
manière, il est possible de traiter les cas réels car personne ne l'a encore fait. Pour
l'instant, je me suis plus intérressé à la théorie. Tout pipeline CPU peut être décomposé
en trois étapes de base : approvisionnement et décodage, exécuter et stocker. Garbage in,
garbage processing, garbage out. Avec les OTA, tout est fait par matériel. Vous donnez
"ADD~R3,~R1,~R2" et le matériel fait le reste. Il traite les moyens de communication interne
pour obtenir les données de R1 et R2 vers l'entrée de l'additionneur, le laisse traiter
l'information et récupère les données à la sortie vers le fichier de registre, dans R3. Dans
les architectures modernes, il contrôle les anomalies, fait suivre les données pour que le
```

reste du pipeline puisse l'utiliser plus tôt et truc encore plus compliqué comme réordonner les instructions. Le logiciel ne connait que le 32 bits ; le matériel fait le reste.

L'étape IF/ID d'un TTA est très différente. Toute la charge est placée sur le logiciel. L'instruction n'est pas spécifiée comme ADD (quelque chose), mais comme une série de paires d'adresses SRC, DEST. Tout ce que le matériel doit faire est le contrôle des bus internes pour mettre les données là où elles sont supposées aller. Toutes les vérifications de risque, d'ordre d'instructions optimal, etc doit être fait par le compilateur. Ici, la clé est qu'un TTA, pour réaliser des mesures IPC comparables à un OTA, doit être VLIW : vous DEVEZ pouvoir spécifier des mouvements multiples en un seul cycle, pour que vous puissiez déplacer toutes vos sources de données vers les zones appropriées et encore bouger les résultats vers votre fichier de registres (où là ou vous voulez qu'ils aillent). En résumé, pour faire un "ADD~R3,~R1,~R2", le matériel 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 sûr, devient beaucoup plus compliqué car il doit faire tout le travail de planification, au moment de la compilation. Mais le matériel 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 générer les signaux de contrôle pour tous les bus.

L'exécution est identique entre le TTA et OTA. Crunch the bits. Period.

La complétion des instructions est encore simplifiée dans un TTA. Si vous voulez un comportement correct, assurez vous que le compilateur génère les bonnes séquences de déplacement. C'est à comparer avec un OTA où vous devez au moins savoir quels port d'écriture vous devez utiliser, etc.

A la base, un TTA et un OTA sont fonctionellement identiques. La principale différence est qu'un TTA doit vraiment être VLIW et demande plus au compilateur. Néanmoins, si la philosophie "smart compiler and dumb machine" est réellement là où l'on veut aller, le TTA doit fonctionner. Il offre une partie plus importante du pipeline au logiciel, réduisant les besoins de matériel et laissant plus de champ au compilateur pour l'optimisation. Bien sûr, il y a des cas comme les embouteillages de code et la génération de constantes, mais ces cas sont traités plus tard. Les idées de base ont été couvertes ici (quoique d'une manière assez décousue)... J'ai composé cet email dans ma tête et j'ai trouvé quelques explications claires directement lorsque je me suis assis et j'ai commencé à 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 détails de TTA; j'espère encore que nous pourrons le mener à bien et je pense que cela sera bien pour la performance, générale, de coût et en simplicité. En plus, c'est suffisamment révolutionaire pour faire détourner les têtes -et que cela puisse nous donner une plus grande base d'utilisateurs (et de développeurs) et rendre le projet plus stable.

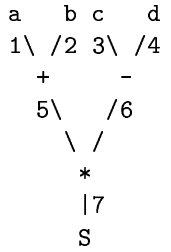
Envoyez moi les questions, je sais qu'il en a beaucoup ...

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 déclenche l'opération a+b)
mv SUB2,d    (ceci déclenche l'opération c-d)
mv MUL1,ADD
mv MUL2,SUB  (ceci déclenche l'opération *)
```

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 $\rightarrow$ 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 :

"

```
   Développer et rendre une architecture librement disponible, ainsi que toutes les
propriétés intellectuelles nécessaires pour fabriquer un ou plusieurs implémen-
tations de cette architecture, avec les propriétés suivantes, dans l'ordre décroissant
d'importance~:

  1) adaptabilité et utilité pour le champ d'applications le plus grand possible
  2) Performance, accent mis sur le parallélisme au niveau de l'utilisateur et dérivé
avec une architecture intelligente, plutôt que par des processus avancés en electronique
  3) Architecture (((((((((lifespan))))))))) et compatibilité directe
  4) Coût, incluant des considérations monétaires 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 familliaux ainsi que par tous les autres marchés spécialisés (embarqués/temps réels, ordinateurs portables/((((((((((wearable)))))))))), 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 10 ans)

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

General description of the F-CPU

2.1 The main characteristics

The CPU described here can be thought as a crossover between a R2000 chip (or early ALPHA) and a CDC6600 computer. Some constraints are similar: the F-CPU must be as simple and performant as possible. From the R2000, it inherits from the RISC main characteristics like fixed size instructions, the register set and the size of the chip that is bound by the current technology. In the CDC6600, FC0 finds the execution scheme, the scoreboard, the multiple parallel execution units and most of all: the inspiration for smart techniques that ease both design and programming.

Recently, the SH5 (Hitachi/ST) CPU showed some similar looking features, such as the 64 registers or the jump target buffers. You will remark, however, that the F-CPU is completely different, particularly from the scheduling point of view.

The following text is a step-by-step description of the currently developped F-CPU. The features will be more deeply described and get interdependent, so it is recommended to read them from the beginning:-) We will begin with the most basic F-CPU characteristics before discussing more critical and hardware-dependent subjects in the next part.

2.2 The instructions are 32-bit wide

This is a heritage of the traditional RISC processors, and the benefits of fixed size instructions are not discussed anymore, except for certain niche applications. Even the microcontroller market is invaded by RISC cores with fixed size intructions.

The instruction size can be discussed a bit more anyway. It is clear that a 16-bit word can't contain enough space to code 3-operand instructions involving tens of registers and operation codes. There are some 24- and 48-bit instruction processors, but they are limited to niche markets (like DSP) and they don't fit in power-of-two-sized cache lines. If we access memory on a byte basis, this becomes too complex. Because the F-CPU is mainly a 64-bit processor, 64-bit instructions have been proposed, where two instructions are packed, but this is similar to 2 32-bit instructions which can be atomic, while 64-bit pairs can't be split. There is also the Merced (IA64) that has 128-bit instruction words, each containing 3 opcodes and register dependency informations. Since we use a simple scoreboard, and because IA64-like (VLIW) compilers are very tricky to program, we let the CPU core decide wether to block the pipeline or not when needed, thus allowing a wide range of CPU core types to execute the same simple instructions and programs.

Since the F-CPU microarchitecture was not defined at the beginning of the project, the instructions had to execute on a wide range of processor types (pipelined, superscalar, out-of-order, VLIW, whatever the future will create). A fixed-sized, 32-bit instruction set seems to be the best choice for simplicity and scalability in the future. Core-dependent optimisations can be made on the binaries by applying specific scheduling rules, but the application will still run on other family members that have a completely different core.

2.3 Register #0

It is "read-as-zero/unmodifiable". This is another classical "RISC" feature that is meant to ease coding and reduce the opcode count. This was valuable for earlier processors but current technologies need specific hints about what the instruction does. It is dumb today to code "SUB R1,R1,R1" to clear R1 because it needs to fetch R1, perform a 64-bit substraction and write the result, while all we wanted to do is simply clear R1. This latency was hidden on the early MIPS processors but current technologies suffer from this kind of coding technique, because every step contributing to perform the operation is costly. If we want to speedup these instructions, the instruction decoder gets more complex. So, while the R0=0 convention is kept, there is more emphasis on specific instructions. For example, "SUB R3,R1,R2" which compares R1 and R2, generaly to know if greater or equal, can be replaced in the F-CPU by "CMP R3,R1,R2" because CMP can use a special comparison unit which has less latency than a substraction (after all we don't care about the numerical result, we simply want its "property").

"MOV R1,R0" clears R1 with no latency because the value of R0 is already known (hardwired to zero).

2.4 The F-CPU has 64 registers

The RISC processors traditionally have 32 registers. More than a religion war, this subject proves that the design choices are deeply influenced by a lot of parameters (this looks like a thread on `comp.arch`).

Let's look at them :

- *"It is proved that 8 registers are plain enough for most algorithms."* is a deadbrain argument that appears sometimes. Let's see why and how this conclusion has been made :
  - it is an OLD study,
  - it was based on schoolbook algorithm examples,
  - memory was less constraining than today (even though magnetic cores were slow) and memory to memory instructions were common,
  - chips had less room than today (tens of thousands vs. tens of million) and a register was an expensive hardware ressource
  - the pipelines were not as deep as today
  - we ALWAYS use algorithms that are "special" because each program is a modification and an adaptation of common cases to special cases, (we live in a real world, didn't you know?)
  - who has ever programmed x86 processors in assembly langage knows how painful it is...

  The real reason for having a lot of registers is to reduce the need to store and load from memory. We all know that even with several cache memory levels, classical architectures are memory-starved, so keeping more variables close to the execution units reduces the overall execution latency.
- *"If there are too much registers there is no room for coding instructions"* : that is where the design of processors is an art of balance and common sense. And we are artists, aren't we? Through register renaming, the number of physical registers can be virtually extended to any physical limit.
- *"The more there are registers, the longer it takes to switch between tasks or acknowlege interrupts"* is another reason that is discussed a lot.

  Then, I wonder why Intel has put 128*2 registers in IA64???

  It is clear anyway that *FAST* context switch is an issue for a lot of obvious reasons. Several technics exist and are well known, like register windows (a la SPARC), register bank switching (like in DSPs) or memory-to-memory architectures (not much known), but none of them can be used in a simple design and a first proto, where transistor count and complexity are real issues.

  In the discussions of the mailing lists, it appeared that :

  - most of the time is actually spent in the task scheduler's code (if we're discussing about OS speed) so the register backup issue is like the tree that hides the forest,
  - the number of memory bursts caused by a context switch or an interrupt wastes most of the time when the memory bandwidth is limited (common sense and performance measurements on a P2 will do the rest if you're not convinced)
  - a smart programmer will interleave register backup code with IRQ handler code, because an instruction usually needs one destination and two sources, so if the CPU executes one instruction per cycle there is NO need to switch all the register set in one cycle. In fewer words, no need of register banks. These facts led to design the "Smooth Register Backup", a hardware technic which replaces the software at interleaving the backup code with the computation code.

    Let's consider an IRQ code starting like this :

```
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
        ...
```

    Whatever the register number is, we only have to save R1 before cycle 1, R2 before cycle 2 and R3 before cycle 3.

    This would take 3 instructions that would be interleaved like this :

```
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
...
```

The "Smooth Register Backup" is a simple hardware mechanism that automatically saves registers from the previous thread so no backup code need being interleaved. It is based on a simple scoreboard technique, a "find first" algorithm and needs a flag per register (set when the register has been saved, reset if not). It is completely transparent to the user and the application programer, so it can be changed in future processor generations with few impact on the OS. It saves at least 64 backup instructions, or 256 bytes of code, that are not loaded from memory. This bandwidth is freed for the other operations required by a task switch: loading the new code, reading the new task's context, writing the old task's context... This technique will be described deeply later in the chapter 4.3.

The conclusion of these discussions is that 64 registers are not too much. The other problem is: is 64 enough?

Since the IA64 has 128 registers, and superscalar processors need more register ports, having more registers keeps the register port number from increasing. As a rule of thumb, a processor would need *at least (instructions per cycle) x (pipeline depth) x 3* registers to avoid register stalls on a code sequence without register dependencies. And since the pipeline depth and the instructions per cycle both increase to get more performance, the register set's size increases. 64 registers would allow a 4-issue superscalar CPU to have 5 pipeline stages, which looks complex enough. Later implementation will probably use register renaming and out-of-order techniques to get more performance out of common code, but 64 registers are yet enough for a prototype. As to increase the number of instructions executed during each cycle, the future F-CPUs will need explicit register renaming. This will allow a F-CPU computer to have tens of execution units without changing the instruction format.

2.5 The F-CPU is a variable-size processor

The F-CPU goals specify *forward compatibility*. There are mainly two reasons behind this choice:

- As processors and families evolve, the data width becomes too tight. Adapting the data width on a case-by-case basis led to the complexities of the x86 or the VAX which are considered as good examples of how awful an architecture can become.
- We often need to process data of different sizes in the same time, such as pointers, characters, floating point and integer numbers (for example in a floating-point to ASCII function). Treating every data with the same big size is not an optimal solution because we will spare registers if several characters or integers can be packed into one register which would be rotated to access each subpart.

We need *from the beginning* a good way to adapt the size of the data we handle "on the fly". And we know that the width of the data to process will increase a lot in the future, because it's almost the only way to increase performance. We can't count on the regular performance increase provided by the new silicon processes because they are expensive and we don't know if it will continue. The best example of this data parallelism is SIMD programming, like in the recent MMX, SSE, AlphaPC, PPC Altivec or SPARC VIS instruction sets where one instruction performs several operations. From 64, it evolves to 128 and 256 bits per instruction, and nothing keeps this width from increasing, while this increase gives more performance. Of course, we are not building a PGP-breaker CPU, and 512-bit integers are almost never needed. The performance lies in the parallelism, not the width. For example, it would be very useful to parallely compare characters, like during substring search: the performance of such a program would be almost directly proportional to the width of the data that the CPU can handle.

The next question is : *how wide ?*

Because fixed-size ints and pointers give rise to problems at one time or another, deciding of an arbitrarily big size is not a good solution. And, as seen in the example of substring search, the wider the better, so the solution is : *not deciding the width of the data we process before execution.*

The idea is that software should run as fast as possible on every machine, whatever the family or generation is. The chip maker decides of the width it can fund, but this choice is independent from the programming model, because it can also take into account : the price, the technology, the need, the performance...

So in few words : we don't know *a priori* the size of the registers. We have to run the application, which will recognize the computer configuration with special instructions, and then calibrate the loop counts or modify the pointer updates. This is almost the same process as loading a dynamic library...

Once the program has recognized the characteristic widths of the data the computer can manage, the program can run as fast as the computer allows. Of course, if the application uses a size wider than possible, this generates a trap that the OS can handle as a fault or a feature to emulate.

Then the question is : *how* ?

The easiest solution is to use a lookup table, which interprets the 2 bits of the size flag in the instructions, as defined in Part 5 : The F-CPU Instruction Set Architecture. The flags are *by default* interpreted this way :

FLAGS SIZE	WIDTH in bytes	WIDTH in bits
00	1	8
01	2	16
10	4	32
11	8	64

Using a lookup table that would be located in the instruction decoding unit, one could modify the interpretation of this field to any power of two. This way, no limitation exists in the instruction itself. The lookup table will be changed from the default value through 4 special registers. The instructions accessing the special registers will ensure that protection and data sizes are coherent, triggering an exception otherwise. A fifth special register will be hardwired to the highest possible value, which is dependent only from the processor.

Special Register name	default value in bytes	function
SR_SIZE_0	1	meaning of SIZE
SR_SIZE_1	2	meaning of SIZE
SR_SIZE_2	4	meaning of SIZE
SR_SIZE_3	8	meaning of SIZE
SR_MAX_SIZE	unknown (hardwired)	Maximum width managed by the CPU

The software, and particularly the compiler will be a bit more complex because of these mechanisms. The algorithms will be modified (loop counts will be changed for example) and the four special registers must be saved and restored during each task switch or interrupt. Simple compilers and less-than-128-bit CPUs could simply use the default four sizes but more sophisticated compilers will be needed to benefit from the performance of the later, wider chips. The interface must be respected by all family members, and if the CPU does not support data wider than 64 bit, the code should not attempt to modify the (hardwired) size special registers (or the CPU will trap). Therefore, in the algorithms, the "widest" size should be used with SIZEFLAG=11 so it will also benefit to hardired, downsized processors.

At least, the scalability problem is known and solved since the beginning, and the coding techniques won't change between processor generations. This garantees the stable future of the FCPU project and architecture, and the old "RISC" principle of letting the software solve the problems is applied once again. We can consider that prototype F1s will be hardwired to the default values, and attempting to modify them will trigger a fault. But later, 4096-bit F-CPUs will be able to run programs designed on 128-bit F-CPUs and vice versa.

2.6 The F-CPU is SIMD-oriented

It's one easy way to increase the number of operations performed during each cycle without increasing the control logic. The variable sized registers allow endless scalability and thus endless performance increase, but each instruction performing operations on data must have a SIMD flag, as to differentiate the type of operation.

The maximum size for a SIMD element ("chunk") is defined in an additional Special Register called SR_MAX_CHUNK_SIZE. It is usually set to 64 on a 64-bit implementation, because it's the largest integer that the core can handle. On a 128-bit architecture, SR_MAX_CHUNK_SIZE will probably remain equal to 64 but it could be equal to 32 or 128 as well.

2.7 The F-CPU has generalized registers

This means that integer numbers are mixed with pointers and floating-point numbers. The most common objection is from the hardware side, because a first effect is that it increases the number of read/write ports in the register set (this is almost similar to having twice more registers).

The first argument from the F-CPU side is that software gets simpler, and that there are hardware solutions to that problem. The first problem comes from the algorithms themselves : some are purely integer-based, while other need a lot of floating point values. Having a split register set for integer and floating point numbers would handicap both algorithms, because specialized registers would not be used (the FP register set would be unused for example during programs like a mailer or a bitmap graphics edition SW, while a lot of FP is needed during ray-tracing or physical simulations). And a lot of them is needed when it happens. Another software aspect is about compilation, where register allocation algorithms are critical for performance. Having a simple (single) register "pool" eases the decisions.

The second answer to the hardware problem is in the hardware. The first F-CPU chip, the F1, will be a single-issue pipelined processor, where only three register read ports are needed, thus there is no register set problem at the beginning.

Later chips, with more instructions issued per cycle, could use another technique : each register has attribute (or "property") bits that indicate if the register is used as a pointer, a floating point number, etc, so they can be mapped to different physical register sets while still being unified from the programming point of view. The attributes are regenerated automaticaly and don't need to be saved or restored during context switches.

2.8 The F-CPU has special registers

They store the context of the processor, manage the vital functions and ensure protection.

These special registers can be accessed only through a few special instructions and can trigger a trap if the register does not exist or is not allowed for access in the current running context. Since almost everything is managed through these special registers, they are the key for protection in a multi-user, multi-task operating system. These special registers are very important to recognize the CPU's configuration and the "SR map" will evolve a lot in the future, adding more features without touching the instruction set. The current SR map can be found in the files F-CPU_config.vhdl and SR.h in the latest package. No standard SR map exists yet, it will be defined at the end of the prototyping phase of the F1.

The instructions that access the special registers are separated from the others because of their potentially dangerous influence on the hardware. Managing the SR through the memory (with load/store instructions) would make pipelining much more complex. For example, the SRs manage the virtual memory : the L/S units would require special features to recognize the SR addresses and avoid any unstable processor states (which are potentially dangerous). The problem is similar to the x86 protected mode switch, where all the pipelines and all the hidden memory descriptors must be changed. The SR are very similar to the MSR introduced with the Pentium CPU and they help separate "common operations" (wich must be pipelined and simple) from the "management operations" (slow, complex and usually microcoded in the CISC CPUs). The GET and PUT instructions (see their description in Part VI) are atomic and don't disturb the pipeline scheduler.

2.9 The F-CPU has no stack pointer

Or more exactly, it has no dedicated stack pointer. It has no stack at all, in fact, because any register can be used to access memory. One single hardwired stack pointer would cause problems that are found in

CISC processors and require special tricks to handle them. For example, several push et pop instructions cause multiple register uses in a single cycle in a superscalar processor, which requires some special management HW.

In the RISC world, conventions (the ABI) are used to decide how to communicate between applications or how to initialize the registers at their beginning. Provided you save the registers between two calls, nothing keeps you from having 60 stacks at once if your algorithm requires it.

Accessing the stack is performed with the single load/store instruction which has post-increment (only) capability. Considering an expand-down stack pointed to by R3, we will code for example :

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

Since the addition and the memory fetch are performed at the same time, the updated pointer is available after the instruction accesses memory.

The "Smooth Register Backup" hardware in place could be used by instructions on some implementations. There may be an instruction that saves or restores parts or all the register set to a specified location but this is only an optional feature.

2.10 The F-CPU has no condition code register

It is not because we don't like them but they cause some troubles when the processor scales up in frequency and instructions per cycle : managing a few bits becomes as complex as the above described stack.

The solution to this problem is the classical RISC fashion : a register is either zero or not. A branch or a conditional operation is executed if a register is zero (or not). Therefore, several conditions can be setup, without the need to manage a fixed set of bits (for example during context switches). We don't use predication bits as found on some other architectures : we don't need them, and their specific instructions as well. It keeps the ISA, the compiler and the scheduling very simple.

But, as explained later, reading a register is rather "slow" in the FC0 and the latency may slow down a large number of usual instructions. The solution is not to read them, but a "cache" copy of the needed attribute. Like described above for the "attribute" or "property" bits of the registers for the floating point issue, each register has an attribute bit which is regenerated each time the register is written. While the register is being acccessed, the value that is present on the write bus is checked for 0 and one bit out of 63, corresponding to the register we write, is set or reset depending on the result. This set of "transparent latch" gates is situated close to the instruction decoder in order to reduce the latency of conditional instructions. Since they are regenerated at each write, there is no need to save or restore them during context switches, and there are no coherency issues.

There is no carry flag either. Addition with carry is performed through a special form of the intruction that writes the carry to a general purpose register next to the result register. This avoids any coherency trouble with the context switches and allows to use a carry with SIMD instructions : this is completely scalable and secure for the pipeline scheduler.

2.11 The F-CPU is "endianless"

Either only big endian or little endian does not satisfy everybody. To solve this problem, there is an endian bit in the load/store instructions. The processor itself is not much biased towards one endianness (well, due to the SIMD nature of the CPU, it is preferred to use little endianness) and the instructions themselves are not subject to this debate. The choice is up to the end user. For further informations, read the discussions in the chapter "5.5.5 Endian flag" or the Endian FAQ at `http://www.rdrop/com/~cary/html/endian_faq.html`.

2.12 The F-CPU uses paged memory

This provides the user with a large private, linear, virtual memory to all executing tasks. Page-based protection is also a simple, software way to protect the tasks' memory spaces from eachother. The VM system is not completely defined but here are the preliminary characteristics :

- The pages will have several sizes, for example 4KB, 32KB, 256KB and 2048KB, in order to reduce the number of page descriptors (pressure on the malloc routines !). A few page descriptors of arbitrary

sized blocks (powers of two) would also be necessary to manage pages larger than 2MB (if you have 128MB of RAM in your computer you will need 64 x 2MB descriptors, more descriptors than the CPU can hold onchip). Proposed granularity for these large blocks is 128KB (base address and size, in a "fence" system) and the CPU could store two such page descriptors onchip.

- The pages could be compressed on the fly when flushed to hard disks (especially for the huge pages). This is an optional feature though because it doesn't decrease the latency of the hard disk, but can optimize the bandwidth on the main memory bus. We have to find a good compressor as well as a good SW/HW compromise for the compression engine.
- One could reserve some space in the cache memory hierarchy to hold the most important pages. The kernel will be responsible of this choice.
- The cachability flags and the read/write flags of the pages will be used for the early implementations to ensure cache coherency in multi-CPU systems with the OS functions and traps, instead of using dedicated hardware. So, not only paged memory is used to protect the tasks and provide more visible memory, but it also serves as a "software" replacement of the MESI protocol in a Non-Uniform Access Memory architecture.
- The internal TLBs are software-controlled through a set of Special Registers. No microcode or hardware mechanism is foreseen that will help search a page table entry in memory. An OS exception is triggered whenever a task issues an instruction that access a memory location that is not in the internal Page Table (TLB). Since there will probably be only four or eight entries of 4KB, 32KB, 256KB or 2048KB each (32 descriptors shared for data and instructions in the first implementations), the OS PTE miss trap handler must be very carefully coded. *Remember this motto? "coding carefully has always paid!".*

Warning: these characteristics are preliminary. Some details will certainly evolve soon.

It appears clearly that the most critical part of the protection mechanism is the TLB. There are some other annex mechanisms but the TLB is the "gatekeeper" for the most common cases. It must be very well designed and provide some useful mechanisms that help efficiently manage the memory and the block allocation. For example, the TLB entries contain additional fields such as the VMID (it is used to reduce the thrashing) and the usage bits (8 2-bit saturated counters that measure the actual memory usage and activity within a page). Both fields are 16 bit wide and help the kernel to enhance the memory allocation.

In order to keep a good overall performance, the project counts on an efficient OS. The LINUX-likes are likely to be the best suited systems because they benefit from all the most recent researches and advances in kernel technology, smart task schedulers and efficient page replacement algorithms. The choice of a software page replacement strategy not only keeps the HW complexity low, but also allows the system to benefit from the future algorithmic advances. If the features are not used, there is no dangling hardware...

2.13 The F-CPU stores the state of a task in Context Memory-Blocks (CMB)

These are very important structures for the OS because the SRB mechanism keeps the handlers from seeing the interrupted tasks for coherency reasons. The OS will deal with these blocks in order to set or modify the properties and access rights of a task, read its registers, or interpret a system call. A context memory block must store all the data that are private to a task in order to fully store and restore it. The endianness of the CMB is not defined.

The CMB holds the state of any task in such a way that it can be stopped and restarted. It is used for debugging as well as multi-tasking. Every F-CPU instruction is *atomic* and can't be split, so we don't store any partial result or temporary pipeline state into the CMB.

A Context Memory Block is divided into a variable number of "slots" that are as wide as the CPU can support (ie, 64 bits for a 64-bit CPU). Each slot contains an individual global or special register.

The first 64 slots hold the contents of the normal "general" registers. They are stored and restored by the Smooth Register Backup mechanism. Since R0 is hardwired to 0, the corresponding slot (the first one) contains the instruction pointer.

The CMB holds the access rights and the most important protection flags. The OS modifies the access rights of a task in the CMB because it can't do it directly in the special registers (which at this time store the OS's properties...). The most important flags are stored in the Machine Status Register ("MSR"): the size flags, the VMID, the capability flags...

The CMB holds the pointer to the task's page table (when paging is enabled). This page table can be stored at the end of the CMB if the OS decides to do so.

Two last slots are used for multitasking and debugging, in conjunction with the SRB mechanism : the "next" and "time slice" slots. The "next" slot is a pointer to another CMB ; the task stored in the CMB can switch automatically to a new task, whose CMB is pointed to by the "next" field. The "time slice" stores the number of clock cycles that the task can execute before automatically switching to the "next" task.

This description is not exhaustive and the number of CMB slots will increase in the future, as the needs and the architectures evolve. A certain number of Special Registers are dedicated to the CMB management.

2.14 The F-CPU can use the CMBs to single-step tasks

To use the CMB when single-stepping a task, no special device is required (except a brain) :

1. Setup the task's CMB to the following parameters : "next" points to the debugger's own CMB, and "time slice" is set to 1 (or any desired number for multiple stepping).
2. Set the "next" special register to the task's CMB.
3. Execute a RFE instruction (return from exception).

When RFE is executed, the processor will automatically switch to the task whose CMB is pointed to by the "next" special register. The processor will then load the CMB's "next" slot into the "next" special register, execute instructions, and switch (back) to the debugger when this number expired. The debugger can then analyze the contents of the task's CMB, its registers and special fields.

A flag in the MSR is also dedicated to single-stepping tasks. The CPU generates a trap after executing any instruction when this flag is set.

Other than single-stepping, the F-CPU will provide the user with traps on special conditions and events, as the implementations allow (this is more implementation-dependent and is not defined yet).

2.15 The F-CPU uses a simple protection mechanism

Before a more sophisticated one is developped, a simple user/supervisor scheme is a good way to start a CPU but a more refined ressource-based protection will enable users to create a more flexible OS, for example based on a micro-kernel approach.

It is not "a good thing" to use protection level rings because some pieces of software, for example in a microkernel OS, are dedicated to a certain task and the rings don't isolate their function properly. OTOH, a task that is dedicated to handle page table entry (PTE) misses only needs to access the associated Special Registers and the hard disk drive : if it fails, there is no consequence on other tasks that are dedicated to communications or memory management, even though they are "trusted" : they are normal tasks but their property flags allow them to access a certain hardware.

Here are some of the "capability bits" that are associated to any task :

* TLB_OFF set if the addresses must not be validated by the TLB
* GET_CMB set if the task can read or write its CMB pointer
* GET_VM set if the task can read or write its TLB miss handler pointer
* etc.

Here is how the protection is implemented by the OS :

* Memory protection is ensured by the TLB miss handler on a page per page basis.
* The TLB miss handler is pointed to by the TLB miss SR, which is only accessed by the tasks with the corresponding capabilities.
* These capabilities are stored in the CMB that reside outside of the visible scope of the untrusted tasks, and the CMB pointer is not accessible to the tasks that don't possess the corresponding capability bit.
* *other capability bits will appear in the future.*

A user task (untrusted) will have all its capability bits cleared, while the kernel will have all the capability bits set. After a reset, all the bits are set and the kernel allows each task to have more or less capabilities by clearing or setting the corresponding bits when it creates a task. For example, a trusted task responsible for the VM management will have the GET_VM bit set only.

Troisième partie

General description of the FCPU Core #0

Chapitre 1

About the FC0 core

Here, we speak about characteristics that are specific to the FC0 ("F-CPU Core #0"), and even though they influence the general definition of the F-CPU, they may be abandonned in the future. This is where the hardware engineer is getting more involved.

1.1 The FC0 is superpipelined

When designing a microprocessor, one of the first question is "what is the granularity of the pipeline?". This is not a critical issue for "toy processors" or designs that are adapted from existing processors, but the F1 is not a toy and it must be very performant since the first prototype... For the F1 case, where the first prototype will probably be a FPGA or a sea-of-gates ASIC but not a full-custon chip, performance matters more because the process will not be able to compete with existing chips. Performance always matters anyway, but in our case there is a strong technological handicap. We need a technique that reaches the same "speed" with slower technology.

So the equation is : *speed = silicon technology* x *critical datapath length*, or *speed = speed of one transistor* x *number of transistors*, so with slow transistors the only way to run fast is to reduce the critical datapath (as an approximative estimation, because other parameters, such as capacitance and wire lengths influence this). So now, what is the minimal operation we can perform without overloading the chip with flip-flops?

The depth of around ten transistors is a compromise between functionality and atomicity. We can create circuits that have around six logical gates of depth or add eight-bit numbers. On top of that, the maximum number of input per gate is set to 4, so it can be easily mapped to existing libraries and FPGA architectures. Care is taken to have simple and fast "building blocks", but the good side is that with 6 logic gates we can't make complex things, while longer datapaths usually give birth to complex problems. With this "limitation" in mind, we also limit complexity and only neighbour-to-neighbour connexions between units are possible. Furthermore, as soon as a unit becomes too complex, it becomes either "parallelized" (a large lookup table can be used for example) or "serialized" (in another word, pipelined) so there is no need to slow down the processor or use asynchronous technology.

The net effect of this bias toward extremely fine grained logic and pipeline stages is that even an addition becomes "slow" because it needs more cycles than usual. This apparent slowness is balanced by higher performance through overlapping of the operations (pipelining) but requires the use of coding techniques usually found in superscalar processors (pointer duplication, loop unrolling and interleaving etc.). Because the stages are shorter, there are more pipeline stages than usual, that's why the FC0 can be considered as superpipelined. But it is only one aspect of the project and today, several processors are also superpipelined.

1.2 The FC0 core

It implements an *out of order completion pipeline* to get more performance from a single-issue pipeline. This is NOT a superscalar or out-of-order execution (or OOO instruction issue) scheme but the "adaptation" of a simple pipelined CPU where instructions are issued in order.

The fundamental reason behind this choice is that not all instructions really take the same time to complete. This fact becomes more important in the F-CPU because it is superpipelined, and one short instruction will be penalized by longer instructions which would lengthen the pipeline. For example,

if we want to calibrate the pipeline length on a 64-bit addition, then longer operations like division, multiplication or memory access with cache miss will freeze the whole pipeline ; on the other side, simple register-to-register moves or simply writing an immediate value to a register will be much slower than actually needed. This can be done on an early MIPS processor but not on a superpipelined processor.

Let's look at the instructions that need to be completed, after the decoding stage :

approximative cycles	1	2	3	4
write imm to reg	write dest			
load from memory	read address	access data: undetermined	write dest	
write to memory	read address	data access data		
logic operation	read operands	operation	write result	
arithmetic op	read operands	operation1	operation2	write result
move reg to reg	read source	write dest		

We can also notice that successive instructions may be independent, not needing the result of the precedent instructions. The last remark is that they don't all need the same hardware. We can come to some conclusions : not all instructions need to read and write registers or compute something, not all instructions complete at the same speed, and some instructions may be much longer than others (for example, reading a memory location with a cache miss, compared to a simple logic operation). We need a variable sized pipeline that allows several instructions to be performed and finish at the same time. One way to envision this is to consider the pipeline as "folded", or "forked" like in a superscalar processor. But this all consists to three successive and optional things : reading operands, processing them and writing the result.

- Reading the operands is not a problem since at most three registers can need to be read in one cycle. this is limited by the instructions themselves,
- Computing is fully pipelined and independent because specialized units process the data,
- Writing the results is a bit more complex because several operations can complete at the same time. A one cycle operation (logical operation for example) will complete at the same time as a two cycle (arithmetic) operation that has been issued during the preceding cycle.

For this last reason, the register set has (at least) two write buses. The FC0 emits up to one instruction per cycle and several instructions can end at the same time. In case more than two values must be written at the same time, the "oldest" instruction (earliest issued) has priority.

This kind of processor core has the advantage that long operations don't slow down or block the whole program if the result data are not needed before the slow operation is finished. For example, a memory read can cause cache miss delays but this won't keep the other execution units to do their job and write their result to the register set. Of course, this puts some pressure on the compiler but not more than for other existing processors, and careful coding has always paid anyway.

The difference between OOO completion and OOO execution is that OOO execution CPUs can issue the operations out of order and need a last unit called "completion unit" or "retire unit" that validates the operations in the program order. This also requires "renamed" registers that hold the temporary results before they are validated for good by the completion unit. All these "features" can be avoided by the techniques described in this document and, unlike OOO execution processors (like PowerPC and P6 cores) the peak performance is not limited by the size of the completion unit's FIFO (or the "ReOrdering Buffer", ROB) but by the number of register ports.

1.3 The FC0 uses a scoreboard

It is the simplest way to handle the out-of-order nature of the core. The way it works is very simple : *each register has a flag that is set when the result is currently being computed, and the instructions are delayed until no flag is set for the registers it uses for read and write.* This way, strict coherency is ensured and no operation can conflict with another at the execution stage : verification of conflicts is done at only one point.

These flags are not exactly like the "attribute" bits because they are not directly accessible by the user but they have the same dynamic behaviour and are not saved or restored. Because they don't occur often and are not critical for performance, write-after-write situations are not examined by the scheduler. The simple rule of blocking an instruction at the decode stage if at least one of the used (read or written) register is not ready is strictly enforced. Of course, the Register 0 which is hardwired to 0 is the only exception and does not block anything.

The scoreboard interacts with the "Smooth Register Backup" mechanism to ensure coherency between the switching tasks.

1.4 The crossbar

The FC0 uses a crossbar between the register set and the execution units because :

- It is the easiest way to "fold" the pipeline,
- It provides a "one fits all" register bypass bus that shortens the latency *between* dependent instruction,
- It reduces the number of register ports.

Because of its role, the crossbar (or "Xbar" for short) is a central part of the FC0. The register set is only written or read through this device which virtually provides it with more than ten ports. It allows the execution units to communicate without the need to write and read registers (in register bypass mode, when operations are dependent) it provides the hardwired register 'zero' and the results are checked for zero with two additional ports.

The Xbar extends the register set's read and write ports, making "vertical" buses (see figure 2.1), and each vertical bus is connected to one of the input and output ports of each execution unit with "horizontal" buses. It also performs some width formatting (byte, word, etc) for the immediate values coming from the instruction decoder. Because of the relatively high number of ports, the crossbar uses a lot of surface and transistors. It requires a cycle of his own to let the data flow through its whole length, and the goal of ten equivalent transistors is likely to be reached fast, because of both transistor count and wire lengths. Therefore, accessing a register takes two cycles from the time the register number has been decoded : one cycle for the register set and another for the Xbar. But when consecutive instructions are dependent, the result that will be written to a register is present on the Xbar and can be used during the next cycle for the next operation ("register bypass").

This can be summarized in the following drawing :

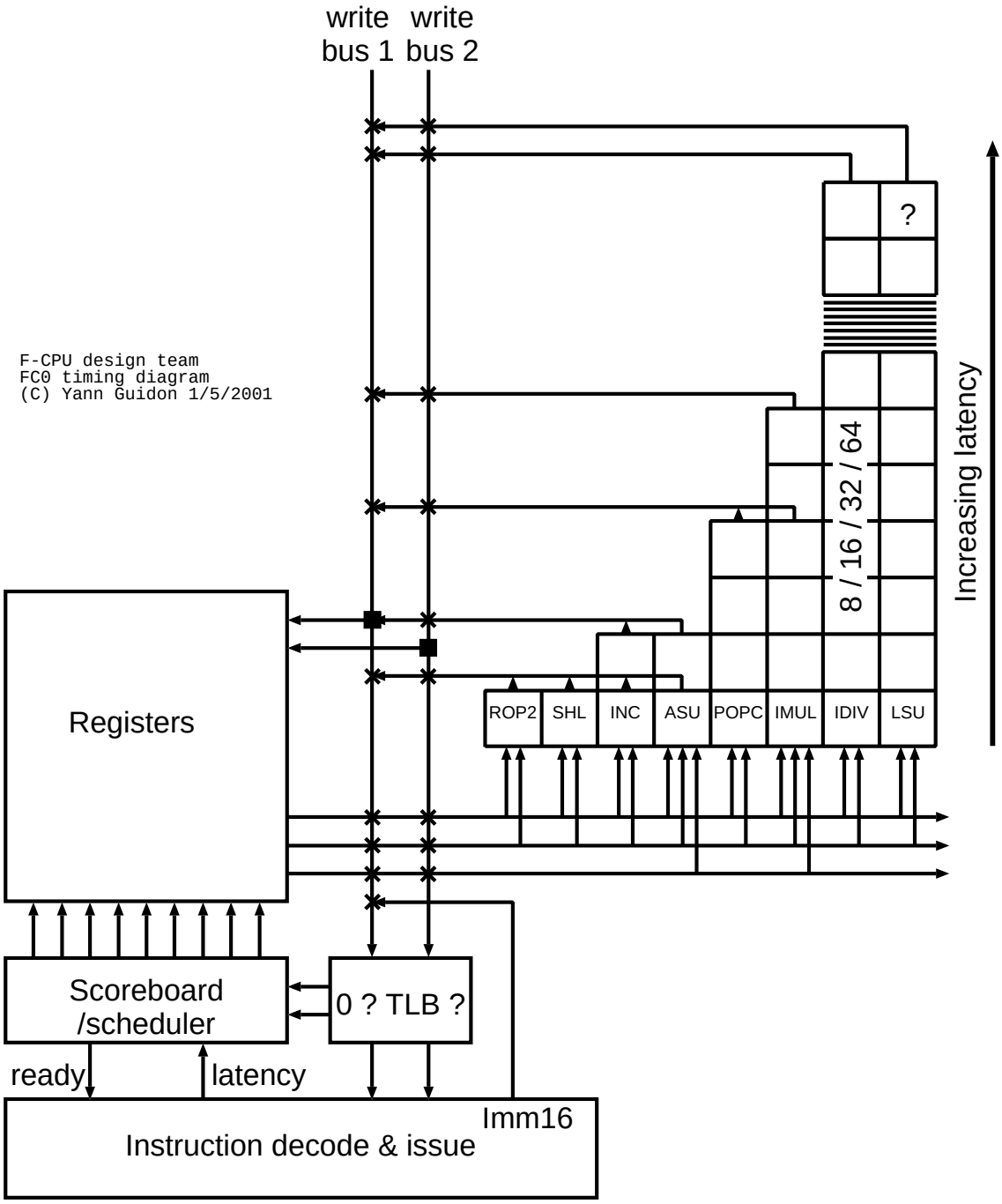


FIG. 1.1 – *The pipeline is folded around the Xbar*

Chapitre 2

Evolution of the FC0

Discussion after discussion, the FC0 has taken a shape that makes it unique. Because it is a gradual change, and because there is not only one view of the processor structure, there have been several drawings that show the internal organization of the chip.

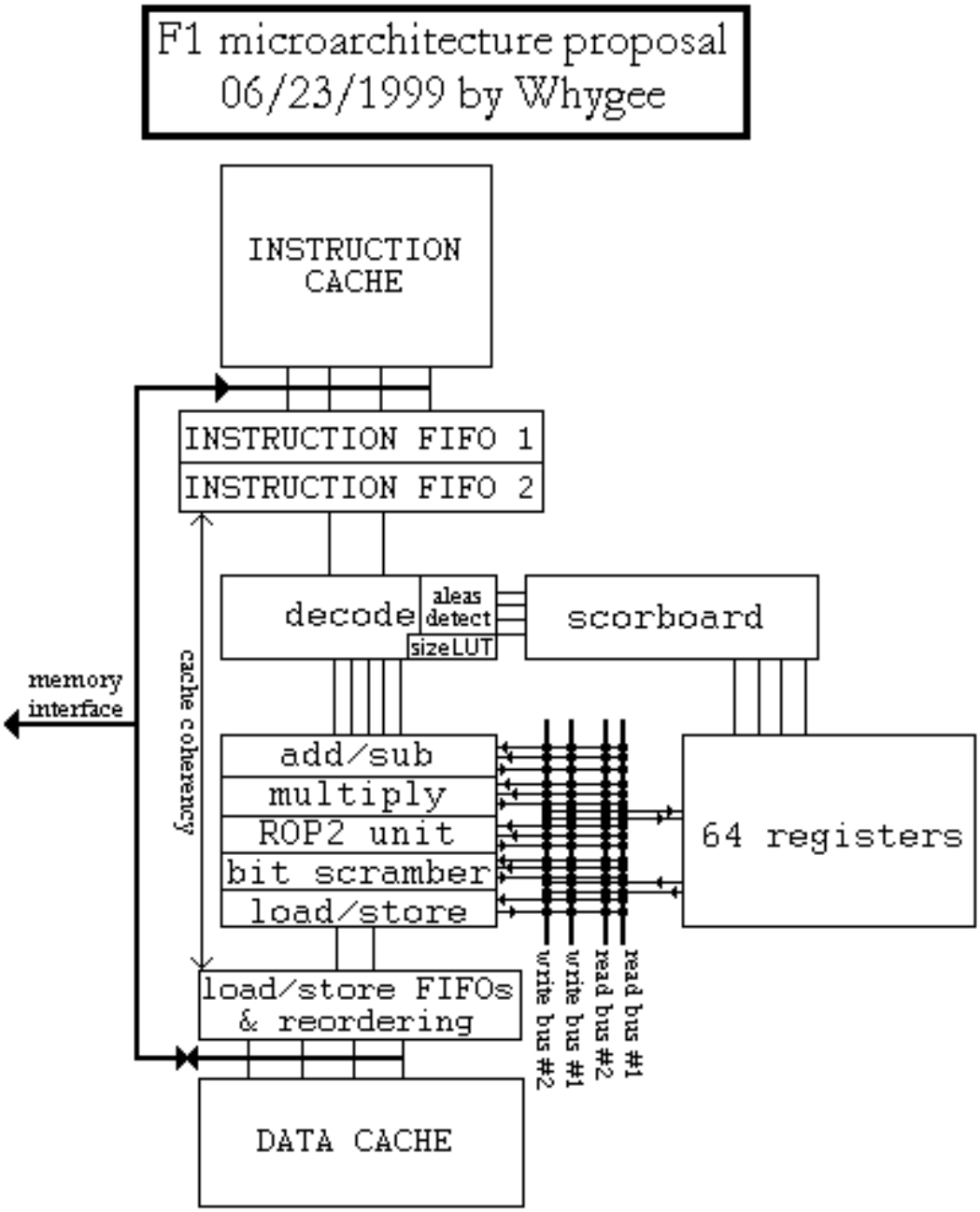


FIG. 2.1 – *The first F-CPU chip proposal*

The figure 2.1 is the first drawing that shows the general shapes of the FC0, from the schematic, functional and implementation points of view. At that time, the Xbar did not count for a full clock cycle in the pipeline. The memory hierarchy was not designed and consisted of empty "units". The execution pipeline though was almost determined and did not change much.

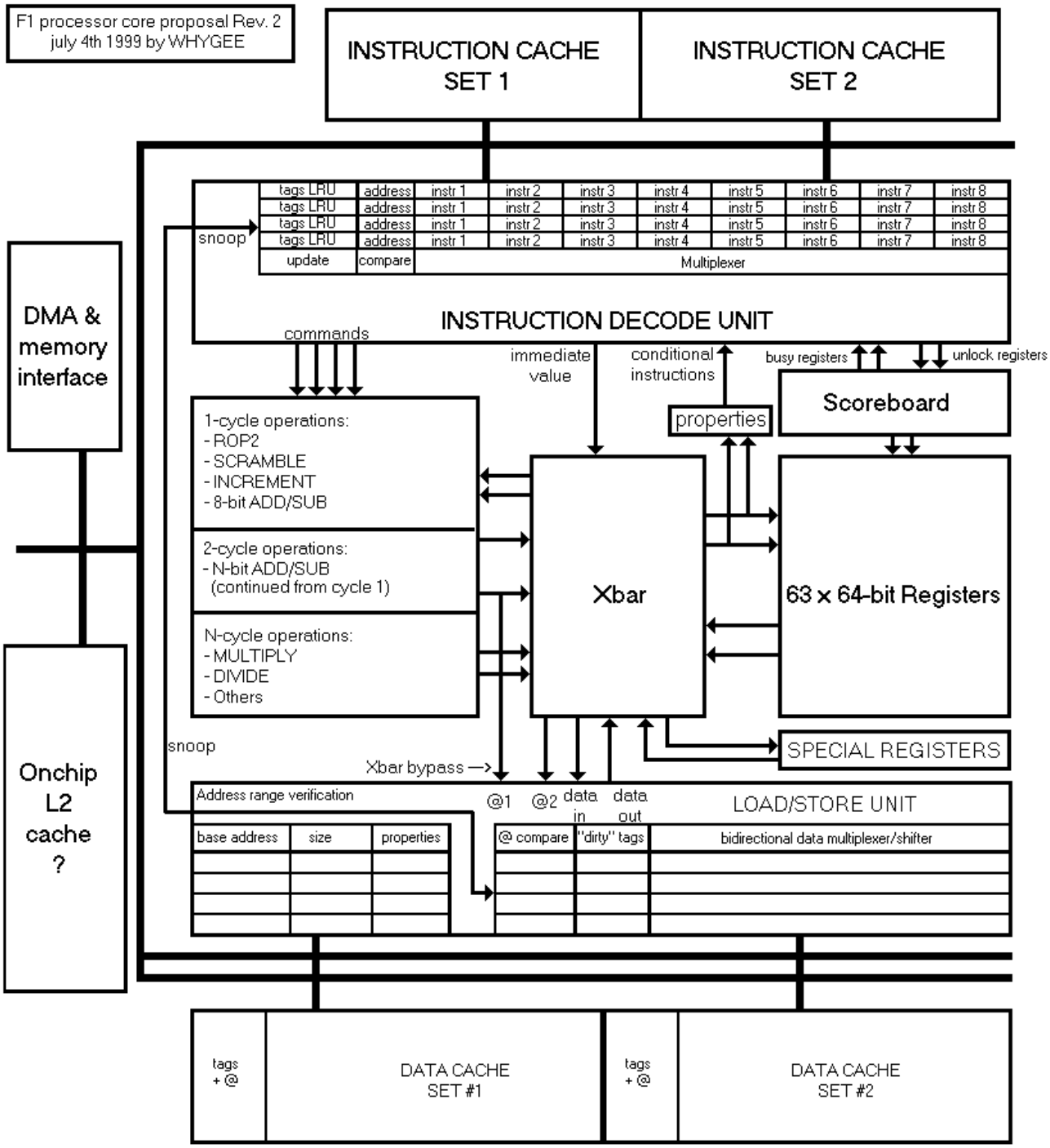


FIG. 2.2 – *A more precise, first-attempt F-CPU description*

The figure 2.2 shows how the units that access the memory would be architected. These are still at both extremities of the chip and require very long wires to snoop for data/instruction access conflicts. The memory units are explicited though, and consist of several cache line buffers. A curious feature is that the address "fences" (that store the base address and limit size of the blocks that a task is allowed to access) are inside the memory units, the TLBs are now outside of the units. The Xbar now takes a full clock cycle and is considered as a full unit, the execution pipeline is refined. Due to the ongoing discussions, the register set had only two read and two write ports, the third read port was accepted later.

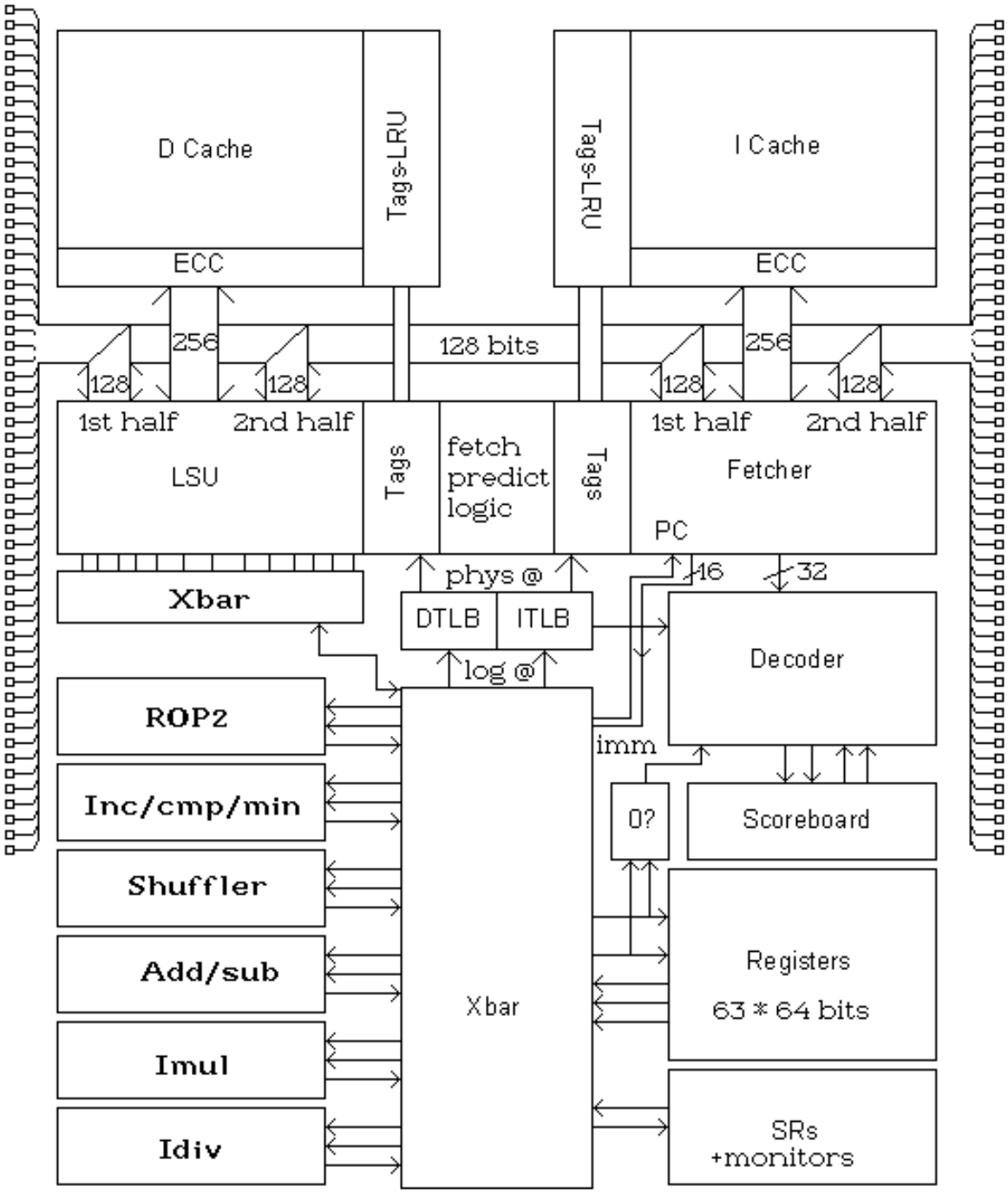


FIG. 2.3 – *A third F-CPU description*

The figure 2.3 shows the current status of the FC0 as it is envisionned for the F1. The memory units have been gathered so the wires that drive the address and data lines outside of the chip have a minimal length. They are symmetrically positionned so the tags of the cache line buffers can all be compared in one simple unit that decides and schedules the memory accesses. The data and instruction TLBs are separated from the memory units because they are parts of the pipeline, and should be placed close to the decoding unit in order to signal an invalid pointer as soon as possible.

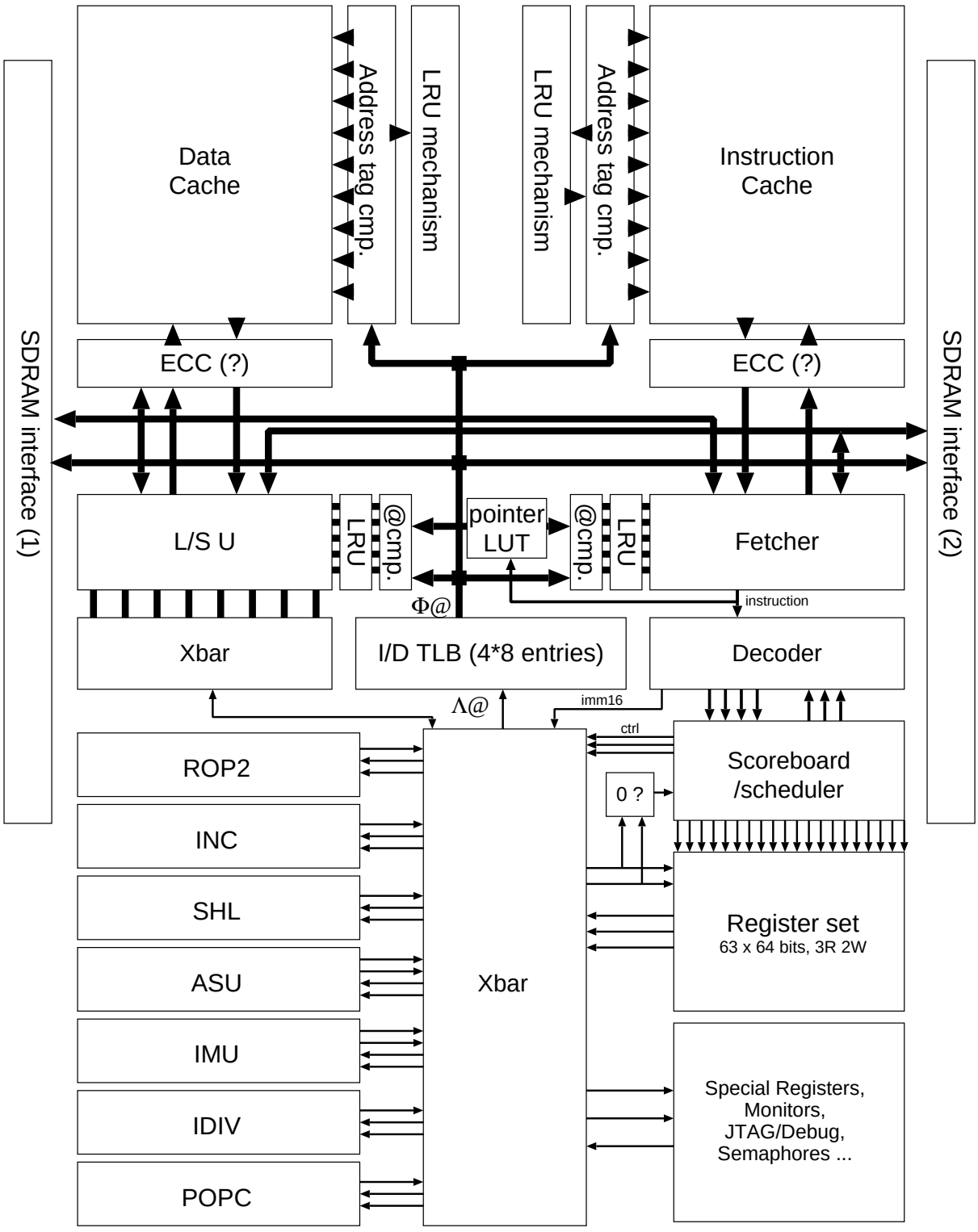


FIG. 2.4 – *The current F-CPU diagram*

The figure 2.4 is the latest update : the external data bus has been split into 2 x 64-bit SDRAM buses, the POPCOUNT unit is added and the memory system (TLB/LUT/cache etc.) is even more precise.

Chapitre 3

The FC0 Execution Units

For ease of development and scalability, to name a few reasons, the Execution Units (EUs) are like LEGO bricks that add new computational capabilities to the processor. Like the whole core, they are designed with a full-custom process in mind but can be implemented with libraries (if they have the corresponding functions) or in FPGA cells or whatever alien technology falls from the sky ...

Here are described the minimal necessary EUs that have been considered until today. As they come, several units can provide the same function (like : shifting left by one is like multiplying by two or adding the number to itself) so the wisest habit is to check which unit does what and in how many cycles with wich throughput, in order to pick the best opcode for the desired operation in each context. Transistor count saving has not been a serious consideration, more care has been taken to reduce the critical datapath to the minimum possible.

Because of their different latencies and particularities, the EUs have not been packed into one "one-fits-all ALU". We can also pick one unit and think about it without caring of the surrounding units. This way, we see that the hardware being designed provides new unexpected operations that can be used in the instruction set. When the hardware is in place, only a few additional logic gates provide useful operations that can spare several instructions in application software, and speedup some critical algorithms with almost no overhead.

3.1 The "logic" unit (ROP2)

This is the classical "logic unit". Its purpose is to compute bit-to-bit operations. Due to its simplicity, it has one cycle of latency and is among the fastest units.

Now, what operations will it execute? With two inputs, there are 2^ 2^ 2=16 possible operations, from which 8 are unique and useful :

A:	0	0	1	1	
B:	0	1	0	1	
	00	01	10	11	Function
	0	0	0	0	CLEAR (set to 0) : equiv. to mov res, reg0
	0	0	0	1	A AND B
	0	0	1	0	A AND /B
	0	0	1	1	A (do nothing)
	0	1	0	0	/A AND B (similar to A AND /B above)
	0	1	0	1	B (do nothing)
	0	1	1	0	A XOR B
	0	1	1	1	A OR B
	1	0	0	0	A NOR B (equiv. to NOT [A OR B])
	1	0	0	1	NOT (A XOR B)
	1	0	1	0	NOT B (do almost nothing)
	1	0	1	1	A OR /B (equiv. to NOT [/A AND B])
	1	1	0	0	NOT A (do almost nothing)
	1	1	0	1	/A OR B (similar to A OR /B]
	1	1	1	0	A NAND B (equiv. to NOT [A AND B])
	1	1	1	1	SET to 1 (-1)

Some opcodes are duplicated (if we include operands commutativity), others are not "real" 2-operands operations (there are 1-op and 0-op operations). We could include directly 4 function bits in the opcode, but we need some room for the "combine" instructions, so we can save one bit with the use of "condensed" codes. We select the 8 2-operands operations and create a new table. The decoder can thus avoid to read unnecessary source registers. For the ROP2 instructions, the 3 function bits are decoded by a tiny hardwired lookup table in the decoder as follows :

opcode	real code	Function	Symbolic name
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

The necessary hardware for computing this function is rather inexpensive :

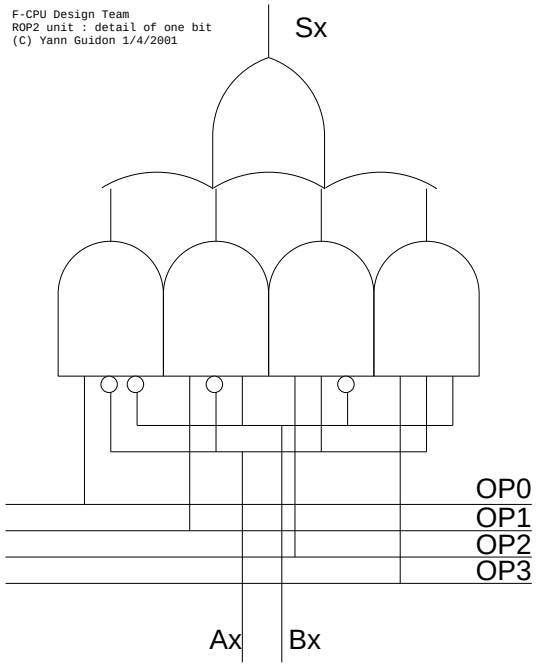


FIG. 3.1 – *Detail of the ROP2 unit*

There are probably a few other technical details to discuss about, but they are too technology dependent (signal "tree" of the operation bus, for example). This is the most straight-forward element of the processor.

Because the critical datapath of this unit is so short, we can add some (simple) functionality : let's call it the "combine" function. While ROP2 is bit-to-bit, the "combination" performs the logical AND or OR of each ROP2 result in every SIMD packet (variable size) of a word. Combined with the ROP2 function, it is possible to perform complex masks and bit moves with few instructions and less need of shifts. Remark : due to the large number of inputs, only 8-bit combines are currently implemented.

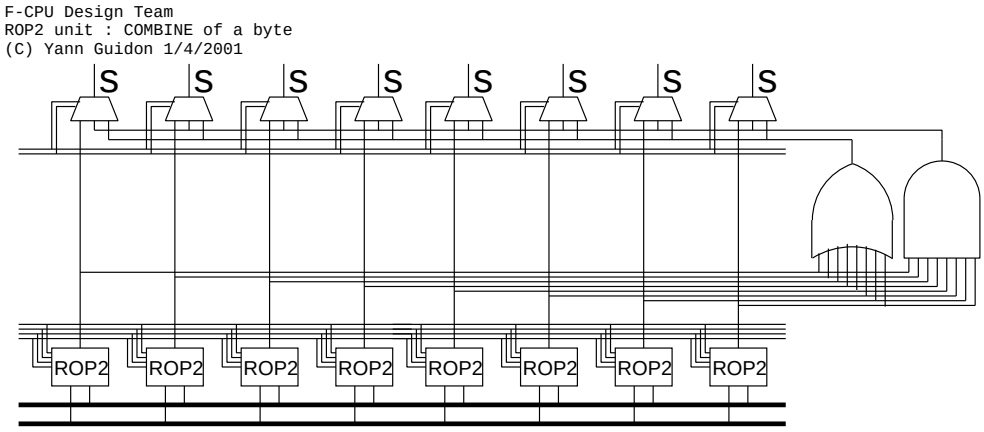


FIG. 3.2 – *Description of the COMBINE function on top of ROP2 for a byte-wide SIMD packet*

VHDL : see the /vhdl/eu_rop2 directory in the F-CPU package.

3.2 The "bit scrambling" unit (SHL)

The goal is to have a one-cycle shifting unit that can do other things as well. As opposed to the ROP2 unit, the principal function is not change the value of the input data bits but to changes the position of the bits. Therefore, shifting and rotating are only examples of the intended purposes of this sometimes called "shuffling" unit : bit field extraction and insertion, as well as bit and byte reversing and bit testing are examples of what this hardware is meant to perform.

There is a problem, though : F-CPU will be a 64-bit processor and a classical barrel shifter is a O (log2 (n)) unit, which is fairly close to the pipeline granularity. A shifting array (a kind of transistor array) will be necessary to get to O (1), at the price of more transistors and probably more transistor load, but it is the only solution if we want to shift 128, 256 or 512 bits in one 10-transistor pipeline cycle. During prototyping, we can use pre-synthetized hardware but a production-class CPU will require something looking like an Omega network of small shufflers.

This unit will also perform SIMD specific operations like SIMD word expansion and mixing. A little logic unit at the end of the critical datapath could perform bit operations (test, set, clear, change) if enough gates are left in the critical datapath.

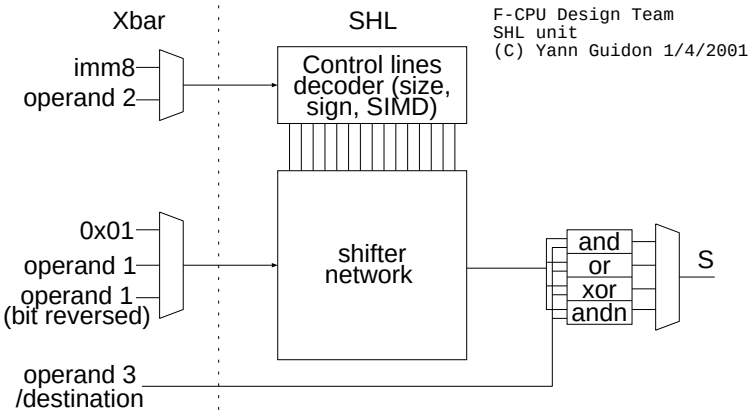


FIG. 3.3 – *Overview of the Scrambling unit*

VHDL : see the /vhdl/eu_shl directory in the F-CPU package.

3.3 The "increment" unit

This is maybe the most curious unit, because it is not usually found in normal CPUs. The reason for this dedicated unit is simple : a lot of code adds or substracts one, in loops for example. This is

unnecessary work for an adder, if the second operand is one, so let's hardwire it and run it faster. That was the first idea.

The method to increment a binary number is not complex to understand : you scan the number starting from the LSB, inverting every bit until you find a 0. Then, you turn this 0 into 1. It is in fact a dedicated carry propagation tree with XOR gates at the output. The tree does the same thing as "find the first LSB set". So, let's go, let's have it too in the instruction set. In some cases, it is very valuable, and there's no hardware overhead. This makes two instructions : INC and LSB1.

So now, we can increment, we can also decrement : we have to invert each bit at the input and the output of the unit. This added hardware lets us also find the "LSB cleared". Four instructions (add DEC and LSB0). We can also add a bit reverser at the input, as to find the MSB too. Six instructions (add MSB1 and MSB0 to the I7, and a bit reverser on the Xbar).

Let's go further : let's put a multiplexer at the end of the incrementer, wich is controlled by the sign bit of the input value. If the bit sign is set, we set the output to - (n+1) (there is a bit of juggling to do with inverters but it's just a "technical detail"). With this unit, we can compute the absolute value of a 2s-complement binary number. Seven instructions (add ABS). Now that we have these multiplexers at the input and the output of the 'incrementer', we can do yet more things. Since the incrementer is a "find first bit" binary tree, we can use it to compare two numbers. The idea is simple, a (positive) number is greater than another if at least one of its MSBs is set while the corresponding bits of the other number is cleared : $0 > 1$, $11 > 10$ ...

So, just XOR the two input numbers, find the first MSB set, and AND the result with one input number. If the result is cleared, then this number is lower then the other, and vice versa. This makes eight instructions. Still better, we can use the ending multiplexer to select one of the input values : we can have the min and max instructions, as well as the derivated like *"if reg1 > reg2 then reg1=reg2"* (for graphics, in coordinates clipping, or saturated arithmetics...). We can have more than ten useful instructions with this simple single-cycle unit ! Some are very useful because they usually involve conditional branches (and pipeline stalls or branch mispredictions...).

From a purely abstract point of view, finding the first set bit is done with a "binary tree", so the depth of the unit is O (log2 (n)) with rather simple "nodes". This is almost a schoolbook case to design. Anyway, like for the shifter array, there are be some problems to fit it in the pipeline's stage depth, mainly for the compare and clip instructions... At least, the INC, DEC, ABS, NEG (and their SIMD variants) are possible in practice with a strong timing constraint.

In this unit, I have not yet addressed the problem of the SIMD data. Comparing signed numbers is straight-forward though : we just have to XOR the sign bit of each SIMD chunk.

The current implementation of the INC unit, doing *inc, dec, neg and abs*, fits in the 6 gates depth of critical datapath. It is composed of a first line of XORs, a 3-gates deep AND tree, a line of multiplexors and a last stage of Xors.

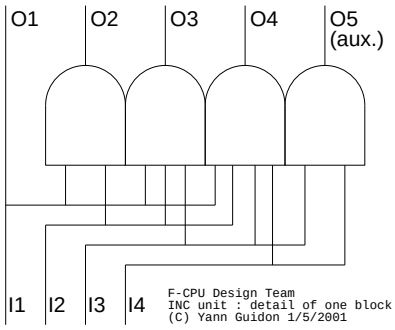


FIG. 3.4 – *Description of one block of the AND tree*

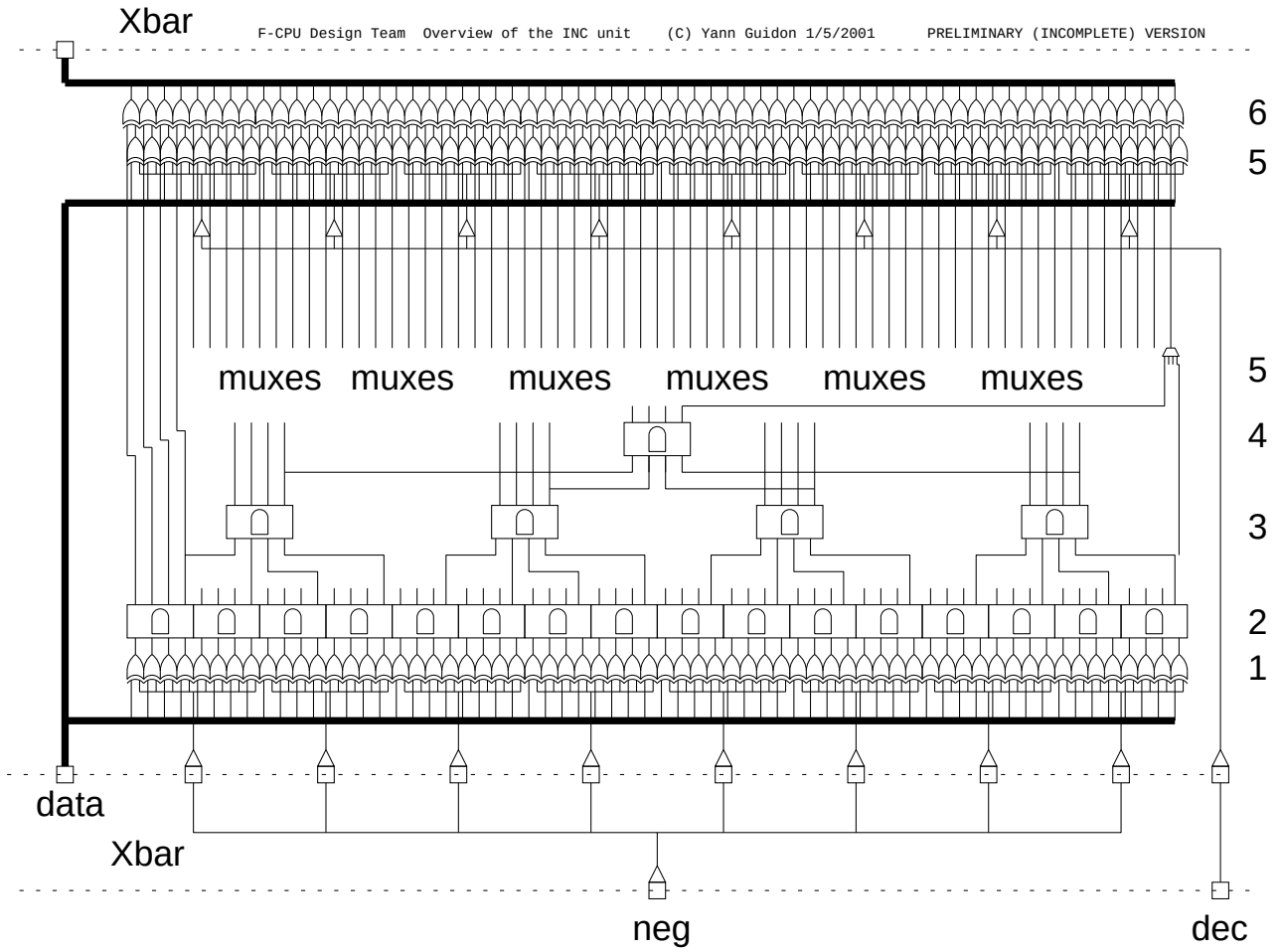


FIG. 3.5 – *Overview of the Incrementer Unit (preliminary version)*

The output of the last XOR stage can be fed to another pipeline stage that will perform the remaining operations (LSBx, MSBx, min, max ...).

You can remark that the Xbar cycle can be used to amplify a single signal to a large number of inputs. The Xbar gives enough time/gates to compensate such a large fanout.

VHDL : see the /vhdl/eu_inc directory in the F-CPU package.

3.4 The add/sub unit

Using a carry-lookahead adder, it needs around two cycles to complete a 64-bit addition or substraction : it is a O (log2 (n)) process with some more heavy mechanisms than the incrementer, but it computes a 8-bit add/sub in one cycle. Therefore, SIMD with 8-bit data is fast (1 cycle instead of 2). For these reasons, it would is difficult to use standard pre-synthetized elements because of the variable-depth and SIMD nature of this unit. Saturation (signed and unsigned) is desired, with a possible additional latency of one cycle.

VHDL : see the /vhdl/eu_asu directory in the F-CPU package.

3.5 The integer multiply unit

Here, same remarks as for the adder. There are SIMD constraints and a variable-depth, fine-grained pipeline (depending on the width of the input data). It will be difficult to find this kind of unit in pre-synthetized libraries. Today's unit does a 64-bit MAC in 6 cycles.

VHDL : see the /vhdl/eu_imu directory in the F-CPU package.

3.6 The integer divide unit

Same as the multiplier. Notice, though, that a divide by zero can be caught at decode time with the "zero" property flags. We can trigger a trap without issuing the instruction. An old substract-shift unit can be enough because it is not used often. If faster divisions are required, the Newton-Raphson method can be used.

VHDL : see the /vhdl/eu_idu directory in the F-CPU package.

3.7 The Load/Store unit

This is a very special case because no actual computation is performed. The latency is completely unknown at compile time, and there is the problem of the memory protection. If the memory protection is ensured by other mechanisms, the L/SU is simply a big cache buffer with a crossbar to perform the word/endian selection. Notice that its structure is similar to the instruction fetcher unit : it is mirored with a different granularity.

When there is no cache miss or buffer to flush, the data can be directly sent or read from the buffer through the L/S crossbar then sent to the main Xbar. In the ideal case, there is no latency for memory writes and 1 cycle for memory reads. The memory fetch logic tries to keep the buffers full when contiguous accesses are performed. A double-buffer (with a pair of line buffers) can hide the memory latency to a certain extent.

The memory buffer can "cache" eight cache lines (the number of lines may vary with implementations). It communicates with the external memory data bus, the data cache memory and the main Xbar. This reduces the latency when recovering from cache misses, and simplifies the cache memory organisation because the L1 cache does not communicate directly with the external memory : the memory buffer (L/SU) is used to split the large cache line into smaller chuncks that can be sent to the memory interface. Not only the LSU stores data but it plays a major role in the memory hierarchy, in the cache replacement cycles and the cache coherency in a multi-bus interface with a limited set of buffers that are used for several functions.

VHDL : see the /vhdl/eu_lsu directory in the F-CPU package.

3.8 Population count / Single Error Correction (POPC)

This is an optional special multicycle unit wich performs SEC and POPC functions.
The POPC instruction also performs saturated substraction with the 6-bit result (see the popc instruction description in part 6).
The SIMD chuncks are basicly 64-bit wide, but nothing keeps the designer to support other granularities.

VHDL : see the /vhdl/eu_popc directory in the F-CPU package.

3.9 other units

The floating point numbers have not been discussed, because we better have something that works correctly in the integer domain first, we'll add FP hardware and instructions later. The case of the math exceptions will be probably managed with the same kind of mechanism as the "zero" property flag, so no error will break the execution pipeline flow.
One "cheap" way to avoid the use of floating point numbers is by using the logarithmic number base. Recent works succeeded in making a 32-bit logarithmic adder with descent speed and die space use. Any other operation (SQRT, SQR, multiply, divide...) can be performed by existing hardware (maybe slightly modified for the MSB). The conversion between integers and log numbers will be a rather heavy software task, as long as no hardware exists.
When FP hardware will become available, only add/sub and multiply units will be implemented at first. Any other mathematical operation (including division) will be computed with a Newton-Raphson approximation algorithm in software. A third unit will provide the "seed" from hardwired ROM tables.

A superpipelined CPU core does not only implies the use of variable length pipelines. Some characteristics of the FC0 and the F-CPU in general will be discussed here, they are not only "features" but design philosophies that are lead by the choices as discussed in the first part of the document.