Freedom CPU Project

F-CPU Design Team

Draft and Request For Comment

Patch YG 2001.1.14

FCPU MANUAL REV. 0.2.2 β

"Design and let design"

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

0.1 Copyright and distribution licence :

0.2 Foreword :

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

This document is (C) 1999-2001 The F-CPU Group Design Team and is a collaborative work. Anybody can participate to the F-CPU effort and become a team member by subscribing to the mailing lists and taking part to the discussions. You are welcome to submit your ideas and report errors. We are conscious that this document always contains errors but we are working on (or around ?) them constantly.

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

0.3 Revision history :

o Created July 8, 1999 by Whygee@f-cpu.org (Yann Guidon) with extracts from Mathias Brossard's RFC.
o July 10 : added some more.
o 11th : adapted for converstion to PDF with HTMLDOC.
o 8/2, 8/8, 8/9, 8/13 : added yet some more.
o 8/25 : reworked a bit (what-why, TTA, endianness, paging, jump station...)
o 11/5 : merged with some other non-architectural contents.
o 11/16 : revamped it all for HTML back.
o 2/27 : major revision of the instruction encoding. Imm6 disappears and most of the old errors/mistypings are corrected.
o 3/15 : Adapted the CPP macro processing (at last)
o 12/18/2000 : Olivier Jean finally published his Latex version of manual
o 12/24/2000 : YG patches. very incomplete !
o 12/30/2000 : YG patches again in Berlin.
o 1/*/2001 : beginning of a french translation, redrawing of the illustrations, major update....

A lot of comments are also given by other people, sometimes anonymouns, on the mailing lists.

0.4 Missing :

* instruction set split into separate files in a special directory.
* alphabetic-ordered instruction set map
* instruction set table, sorted by hex. opcode value
* examples in the instruction descriptions
* IRQ/traps
* SR map
* parts 8 and 9
* and a lot of other stuffs !!!

0.5 Hyperlink jump station :

HTTP :

* The F-CPU main sites : `http://www.f-cpu.org` and `http://www.f-cpu.de`
* The latest update of the F-CPU Manual : `http://www.f-cpu.seul.org`

The mailing lists :

* `http://www.eGroups.com/list/f-cpu` (main list)
* `http://www.eGroups.com/list/f-cpu_france` (french list)
* `http://www.eGroups.com/list/fcpu-ger` (german list)

Contents

List of Figures

List of Tables

Part I

The F-CPU Project, description and philosophy

Chapter 1

Description of the F-CPU project

There is no exact definition of the F-CPU project. It is not possible because of the amount of history, discussions, details that forge the specificity of this undertaking. We can however highlight some important facts and points.

The F-CPU architecture defines a SIMD, superpipelined, 64-bit RISC microprocessor. As of today, it is the only CPU of this kind which can be completely parameterized : it is not bound to 64-bit implementations and it is intended to scale up easily. Furthermore, it is the only processor of this class that is available with all the (VHDL) source code and manuals distributed with the GNU licence (GPL and GFDL). It is meant to be a totally ununcumbered design targeted at the widest range of technologies as possible.

The F-CPU project is also formed by a lot of people, discussing on mailing lists about the organisational and technical sides of the design. The mailing lists are public places where the processor is transparently designed with contradictory discussions. Everybody can come and influence the specifications if the modification respects the design and the project's goals.

The F-CPU group is one of the many projects that try to follow the example shown by the GNU/Linux project, which proved that non-commercial products can surpass expensive and proprietary products. The F-CPU group tries to apply this "recipe" to the Hardware and Computer Design world, starting with the "holy grail" of any computer architect : the microprocessor.

This utopic project was only a dream at the beginning but after two group splits and much efforts, we have come to a rather stable ground for a really scalable and clean architecture without sacrificing the performance. Let's hope that the third attempt is the good one and that a prototype will be created anytime soon.

The F-CPU project can be split into several (approximative and not exhaustive) parts or layers that provide compatibility and interoperability throughout the whole project's lifespan (from HardWare to SoftWare) :

* F-CPU Peripherals and Interfaces : bus, chipset, bridges...
* F-CPU Core Implementations : individual chips, or revisions (for example, F1, F2, F3...)
* F-CPU Cores generations, or families (for example, FC0, FC1, etc.)
* F-CPU Instruction Set and User-visible ressources
* F-CPU Application Binary Interface
* Operating System (aimed at Linux-likes)
* Drivers
* End-User Applications

Any layer depends directly or indirectly from any other. The most important part is the Instruction Set Architecture, because it can't be changed at will and it is not a material part that can evolve when the technology/cost ratio changes. On the other hand, the hardware must provide binary compatibility but the constraints are less important. That is why the instructions should run on a wide range of processor microarchitectures, or "CPU cores" that can be changed or swapped when the budget changes.

Any core family will be binary compatible with each other and execute the same applications, run under the same operating systems and deliver the same results with different instruction scheduling rules, special registers, prices and performances. Each core family can be implemented in several "flavours" like a different number of instructions executed by cycle, different memory sizes, different word sizes, but the software should directly benefit from these features without (much) changes.

This document is a study and working basis for the definition of the F-CPU architecture, aimed at prototyping and first commercial chip generation (codenamed "F1"). This document explains the architectural and technical backgrounds that led to the current state of the "FC0" core as to reduce the amount of basic discussions on the mailing list and introduce the newcomers (or those who come back from vacations) to the most recent concepts that have been discussed.

This manual describes the F-CPU family through its first implementation and core. The FC0 core is not exclusive to the F-CPU project, which can and will use other cores as the project grows and mutates. The FC0 core can also be used for almost any similar RISC architecture with some adaptations.

The document will (hopefully) evolve rapidly and incorporate more and more advanced discussions and techniques. This is not a definitive manual, it is open to any modification that the mailing list agrees to make. It is not exhaustive either, and may lag as the personal free time fluctuates. You are very encouraged to contribute to the discussion, because nobody will do it for you.

Some development rules :

* This Project is an experiment to prove it's possible to develop a processor in a bazaar-style environment. The decisions are made by discussion and consensus on the mailing list.

* There is no leading or ivory tower (this is not a "cathedral"). In fact this is a "Crtistal tower" because everything is as transparent as possible. Anyone may join the team and contribute - or even contribute without officially "joining" in any way. Even those with limited or no knowledge of CPU development can have something to contribute. A lot of motivation and free time is required, however ...

* The name of the game is Freedom, so our designs are being developed openly and will be openly distributed under the GNU Public License, so anyone will be able to (if they have the funding at least) use our designs, manufacture and sell their own F-CPU or derivative chips, but any changes will have to be made freely available again. Read the GNU Public Licence and the F-CPU charter for more details.

* We are aware of the extreme ambitiousness of this Project, but we believe it to be necessary for the continued existence of free software in a world of increasingly proprietary hardware, so we will persevere until we are successful.

* We are also fed up of being forced to use proprietary HW because we are not able to influence the platform. As users, we understand that Free Software can't blossom without Free Hardware.

* Remember, here at the Freedom CPU Project we are not anti-Intel, anti-Microsoft, or in fact anti-anything. We are only pro-Freedom!

* Never flame, never respond to flame bait, but please do make and take constructive criticism.

* "Design and let design" could sum up most of the behaviours adopted in the group. Some strong disagreements have and will appear during the discussions, but whether the subject correspond to the f-cpu goals or not, everybody has the right to play with his ideas. Do not force others to agree, but discuss constructively and explore the subject, instead of flaming other's idea down. A good architecture can come from a mutual respect, not from flame wars.

Chapter 2

Frequently Asked Questions

Collected from different sources. Last modified by Whygee, jan, 14 2000

2.1 Introduction

Q1 : What is the F-CPU ?

A : The F-CPU is a inherently SIMD, 64-bit, superpipelined microprocessor, available with VHDL source code and distributed under the terms of the GNU Public Licence. It is being developped by a community of hobbyists, students and professionals on the Internet.

Q2 : Why a 64-bit RISC CPU ? I want to make a x86 clone / a soundcard / a 32-bit embedded core ...

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

The original goal of a high performance 64-bit CPU dates back to the early days of the project when the founders wanted to counter the Merced (ia64). If you desire something else, there is a great chance that a project already exists with a goal similar to your requirements. The OpenCollector is one of the websites that list the "free" projects that you can access on the Internet. If you don't find what you want, don't hesitate to create your own project.

There are already a lot of free CPU projects available on the Internet. If you desire a 32-bit-only CPU, the MIPS/DLX and the LEON CPUs are good starting points, even though the F-CPU can be easily scaled down to 32 bits. If you require a 16-bit or 8-bit microcontroller, there are also a lot of free (in different ways) designs. You just have to pick one from the lists of the OpenCollector website. If you are sure that you want a F-CPU-like processor, be sure to read and understand this manual before you go further in your project. The goals of the project are firm and will not change because of an individual's whim.

2.2 Philosophy

Q1 : What does the F in F-CPU stand for ?

A : It stands for Freedom, which is the original name of the architecture, or Free, in the GNU/GPL sense.

The F does not stand for free in a monetary sense. "Free" doesn't mean "free as free beer" but "freely copiable and modifiable". You will have to pay for the chip, just as you have to pay nowadays for a copy of a GNU/Linux distribution on CD-ROMs. Of course, you're free to take the design and masks to your favorite fab and have a few batches manufactured for your own use.

Q2 : Why not call it an O-CPU (where O stands for Open) ?

A : There are some fundamental philosophical differences between the Open Source movement and the original Free Software movement. We abide by the latter, hence the F.

The fact that a piece of code is labeled Open Source doesn't mean that your freedom to use it, understand it and improve upon it is guaranteed. Further discussion of these matters can be found at `htpp://www.gnu.org`.

We tried to make a licence similar to the GPL (GNU Public Licence from the Free Software Foundation) (see `http://www.opencollector.org/hardlicense/`) but this effort has been abandonned because it doesn't seem necessary or useful. Today it is replaced by an external charter that strengthens the meaning of the GPL.

Specifically, there are at least three levels of freedom that must be preserved at any cost :

- Freedom to use the Intellectual Property : no restriction must exist to use the work of the F-CPU project. This means, no fee to access the data and ALL the necessary informations to recreate a chip.
- Freedom to reverse-engineer, understand and modify the Intellectual Property at will.
- Freedom to redistribute the design files.

This design is NOT public domain. The F-CPU group owns the IP that it produces. It chooses to make it freely available to anybody by any means. Every file or hardware generated from the description files and the Intellectual Property of the F-CPU team keeps the copyright of the F-CPU team. You can read more about it at `htpp://www.gnu.org`.

Q3 : How is the F-CPU design protected ?

A : The F-CPU Design Team protects his work with the copyright laws. Every file holds the copyright notice and the GPL notice. Nothing else is required.

Additional measures should ensure that no patent issue will arise in the future. Patents are well known for their inefficiency and high cost. The F-CPU design team is protected because it only describes the device, while issues appear when the design is implemented. We must publish proofs of prior art, during conferences and in the press, to avoid the remaining troubles. In the end, the design must remain totally unencumbered.

Q4 : And what if I patent a feature of the F-CPU ?

A : You will loose time and money, that's all.

First, the design is based on common techniques that are heavily studied for thirty years. You'll have a hard time explaining what is new enough to justify a patent.

Second, if the patent is accepted, noone will accept to pay the royalties on something that was stolen from the F-CPU team. Suing the implementors will lead to nothing and in the end, you will own a useless patent that gives you only worries.

Q5 : Why would my company use the F-CPU instead of another core ?

A : The technical advantages of the F-CPU are described in this manual : extreme scalability and orthogonality, ununcumbered and clean design, emphasis on performance, simplicity and retargetability to various technologies (FPGA/ASIC...)

However the non-technical side of the project might ring a bell if you want to integrate the F-CPU core in your design. The design files are available at no charge, but it's not the only meaning of the "free" in the F-CPU. It is a transparent design, it is not a "black box" obfuscated by a proprietary and closed team. If you're in troubles, (for example : the design is deprecated, abandonned by the company or it runs out of business, in short : you're left alone with the design) you don't have to reverse-engineer the "black box" to figure out what goes wrong. You simply read the source code and patch it. The F-CPU is distributed under the terms of the GPL which gives you all the rights to understand and modify (customize) the files.

Another aspect concerns the legal expenses. Just as the GPL is called a "gentleman agreement", the F-CPU is a "gentleman's CPU". We promote peaceful collaboration between the teams : more budget can be dedicated to research and design, less money is spent for the legal departments. In the end, everybody wins because the teams can be larger and spend all their time on the quality of the design and the time to market, rather than on expensive endless lawsuits. "Design and let design" : the only interesting and determining things are the reactiion time and the efficiency (cost, performance, ease of use) of the product.

Q6: Great but what are the fine lines ? Are there drawbacks ?

A : They are contained in the GPL and the F-CPU charter. Just as the design was brought to you free, you have to keep it free and redistribute all the modifications or additions to the core. Because the F-CPU (like all GPL projects) is based on collaboration/cooperation and not competition, your enhancements will benefit others but they can also enhance on your enhancements and this will benefit you in return.

If you want to keep your design secret, don't integrate the F-CPU in your project. You will not be able to benefit from other's work and experience. You will have to reinvent the wheel and loose time and money.

2.3 Tools

Q1 : Which EDA tools will you use ?

A : There has been a lot of debate on this subject. It's mainly a war between Verilog and VHDL. We have started with VHDL'93 for convencience because it is most used in Europe (where most of the code is written) but it will certainly be translated to other file formats. Currently, the design exists in VHDL'93 only for convenience and uniformity. The other representations will be derived from it.

Now that VHDL is the main langage, VHDL tool choices are more limited. We want to promote GNU EDA SW but this branch is not yet developped or mature enough. A particular software may be difficult to install, another may be unstable, outdated or not compliant with today's standards and requirements.

The use of Alliance (`http://www-asim.lip6.fr/alliance/`) is considered but it will be useful only during the layout process. Other free tools and designs can be found from `http://www.opencollector.org`.

Today we use Simili (`http://www.symphonyeda.com`) under the Win32 platform. It is not a GNU software but has many advantages such as independence, compliance with the IEEE standard, ease of use, compacity... We expect a Unix port in the future, as well as other good GNU EDA software.

The sources have also been compiled without modification with FreeHDL and Modelsim. Other IEEE compliant compilers will certainly confirm the high portability and quality of the design.

Cadence has just proposed free licences for some of its tools. Other offers will probably follow and are welcome, as long as the "counterparts" are compatible with the F-CPU charter.

We'll probably use commercial products at one point or another because the chip makers use proprietary software. In any case, a pen, paper and a brain always help.

2.4 Architecture

Q1 : What's that memory-to-memory architecture I heard about ? Or this TTA engine ? Why not a register-to-register architecture like all other RISC processors ?

A : *M2M* was an idea that was discussed at the beginning of the F-CPU project. It had several believed advantages over register-to-register architectures, like very low context switching latency (no registers to save and restore). Today, the SRB mechanism included in the FC0 solves this problem (see the Part IV, chapter 3, "The Smooth Register Backup mechanism").

TTA is another architecture that was explored before the current design (FC0) started.

The F-CPU architecture might evolve in the future and borrow some new features of other architectures.

Q2 : You're thinking about an external FPU ?

A : No. Bandwidth and pin count problems. We can easily fit such units on a chip today.

Q3 : Why don't you support SMP ?

A : Symmetric Multi-Processing like it is implemented on low-end PCs limits the performance and scalability of the architecture. We're actively investigating other architectures, mainly Non-Uniform Memory Access through a specific bus called F-BUS. We try to avoid all the complex techniques needed by multi-CPU processing. No firm decision has been made as of today. The F-CPU core is independent from the bus interface anyway, almost any type of connexion can be implemented.

2.5 Performance

Q1 : What can we expect in terms of performance from the F1 CPU ?

A : Merced-killer. :-). No seriously, we hope to get some serious performance, though it would be impossible to make any claim before we can measure the performance of a real chip.

We think we can achieve good performance because we start from scratch (x86 is slower because it has to be compatible with older models).

LINUX and GCC are not the best garanties for performance in themselves. For example, GCC doesn't handle SIMD data. We will certainly create a compiler that is more adapted to the F-CPU and GCC will be used as a "bootstrap" for the “legacy SW” at the beginning. The ongoing work on GNL and XML-based interfaces will probably allow developpers to create better code than what GCC would ever do.

Objectively, the FC0 core family is aimed to achieve the best MOPS/MIPS ratio possible, around 1 (and maybe a bit more). The superpipeline garanties that the best clock frequency is reached for any silicon technology. The memory bandwidth can be virtually increased with different hint strategies. So we can predict that a 100MHz chip with 1 instruction decoded at each cycle can easily achieve 100 million operations per second. Which is not bad at all because you can achieve that with an "older" (cheap) silicon technology that couldn't achieve 100MOPS with a x86 architecture. Add to that the unconstrained SIMD data width, and you get a picture of the peak MOPS it can reach. If you really want screaming numbers, with a 64-bit version, SIMD operations on bytes leads to 8 operations per cycle, or 800MOPS peak.

2.6 Compatibility

Q1 : Will the F-CPU be compatible with x86 ?

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

There will be NO binary compatibility between the F-CPU and x86 processors. It could however run Windows emulators that include x86 CPU emulators such as Twin, as well as Windows itself under whole-PC emulators such as Bochs. In either case however you will need to run another operating system, such as GNU/Linux, and emulation will likely be fairly slow. But what would be the point of using Windblows when you can run GNU-Linux/xBSD instead ? ;-D

Q2 : Will I be able to plug the F-CPU in a standard Socket 7, Super 7, Slot 1, Slot 2, Slot A or any other existing motherboard ?

A : Great chances are that no version of the F-CPU will ever be available for Socket7 or any x86 mother boards.

Reason 1 : the BIOS should be rewritten, the chipsets should be analysed, and there are way too many chipsets/motherboards combinations around. It is clearly out of the scope of our project.

Reason 2 : Socket/pins/bandwidth : the x86 chips are really "memory bound", the bandwidth is too low, some pins are not useful for a non-x86 chip, and supporting all the functions of the x86 interface will make the chip (its design and debugging) too complex, more expensive and slower.

Reason 3 : we don't want to pay the fees for the use of proprietary slots.

ALPHA- or MIPS-like slots will probably be supported, we might include an EV-4 interface to the F-CPU standard. Anyway, a custom socket and interface will avoid any compatibility and misunderstanding problem. If you want to plug your F-CPU chip on something else, "just do it".

Q3 : What OS kernels will the F-CPU support?

A : Linux will probably be ported first. Other ports will follow and different kernel types are possible. But first we must have a working software development tool for the architecture, thus we must first fully define the F-CPU ...

Q4 : What programs will I be able to run on the F-CPU ?

A : We have a first prototype/preliminary port of gcc/egcs for the Freedom architecture. Basically the F-CPU will run all the software available for a standard GNU/Linux distribution, except the low-level parts such as assembly, I/O and bootstrap code.

Remember that GCC is not perfectly adapted to fifth generation CPUs. We have adapted it for the F-CPU but it was very difficult and it supports only a small subset of the capabilities of the F-CPU ISA. Don't expect good performance from the generated code, at least for the FC0.

2.7 Cost/Price/Purchasing

Q1 : Will I be able to buy a F-CPU someday ?

A : We hope so. That's all the point of the project, but be patient and take part of the discussions ! If you think it is not developped fast enough, join the team and help us. Before the F-CPU will exist in a chip, it will be available in other forms such as software or hardware emulations or simulations.

Q2 : How much will the F-CPU cost ?

A : We don't know. It depends on how many are made. There was an early slightly optimistic estimate that an F-CPU would cost approximately $100, if 10000 were made. This also depends on a lot of factors like the desired performance, the size of the cache memory, the number of pins, and most of all, the possibility to combine all these factors in the available technology. The latest estimations for a first limited version gave around $60 each for a batch of 1K ASIC. The FC0 chip looks a bigger and simplified 486, it belongs to the class of 1 million transistors chips. It is more than the LEON core or the ARM, but it is small compared to other 64 bit chips. Therefore it shouldn't be as expensive as a high-end CPU.

Chapter 3

The genesis of the F-CPU Project

A lot of things have happened since the following document was written. The motivation has not changed though, and the method is still the same. The original authors are unreachable now but we have kept on working more and more seriously on the project. At the time of writing, several questions asked in the following text have been answered, but now that the group is structuring itself, the other questions become more important because we really have to face them : it's not utopy anymore, the fiction slowly becomes reality.

Don't forget too that the technical features that are described here are NOT realistic and don't correspond to anything real. This was more a dream than a coherent analysis. Please don't flame us for other's dreams.

3.1 The Freedom CPU Architecture : A GNU/GPL'ed high-performance 64-bit microprocessor developed in an open, Web-wide collaborative environment.

Authors : Andrew D. Balsa w/ many contributions from Rafael Reilova and Richard Gooch.

5 August 1998

3.1.1 History

The idea of a GNU/GPL'ed CPU design sprang in the middle of some email exchanges between three long-time GNU/Linux users (also Linux kernel developers in their spare time) with diverse backgrounds.

We were questioning monopolies and how the dominance of an operating system (including the kernel, the Graphical User Interface and the availability of "killer-applications" as well as documentation) was intimately related to the world-wide dominance of a specific, outdated, awkward and inefficient CPU architecture. I guess we all know what I am referring to.

We also expressed our faith that GNU/Linux is well on its way to provide the basic foundation for a totally Free software environment (in the GNU/GPL sense; please get a copy of the GNU GPL license if you are reading this, or check www.gnu.org). However, this Freedom is limited or rather bound by the proprietary hardware on which it feels most at home to run : the traditional x86-based PC.

Finally, we were worried that Intel's attitude of not releasing advance information to the Free Software community about its forthcoming Merced architecture would delay the development of a compatible gcc compiler, of a custom version of the Linux kernel, and finally of the vast universe of Free Software tools. It is vaguely rumoured that Linus Torvalds may have received advance information on Merced by signing an Intel NDA, but this would be an individual exception and wouldn't really fit with the spirit of Free Software. On the whole, even though Merced will certainly be more modern that the x86 architecture, it will be a step backwards in terms of Freedom, since unlike for the x86, there will most likely never be a Merced clone chip.

In the previous days, we had been discussing the various models for Free Software development, their advantages and disadvantages. Putting these two discussions together, I quickly drafted an idea and posted it to Rafael and Richard, warning them that this would be good reading while they were compiling XFree86 or a similarly large package... and then they liked it ! Here is this crazy, utopic idea, merged with comments, criticism and further ideas from Rafael and Richard :

3.1.2 The Freedom GNU/GPL'ed architecture

We started with some questions :

- Why don't we develop a 64-bit CPU and put the design under the GNU General Public License?
- Why don't we make the development process of this new CPU completely open and transparent, so that the best brains worldwide can contribute with the best ideas (somehow using the same communication mechanisms traditionally used by the Free Software community) ?
- How can we make the CPU development process entirely democratic and truly open, whereas it is usually surrounded by paranoia and secrets?
- How can we design something that will improve in *technical* *grounds* on what will be available in 2000 from the most advanced CPU architecture team ever put together by any corporation (the Merced) ?

There are really two distinct incredible challenges here :

a) the performance and feasability of the resulting architecture, and

b) the open development process under a GNU/GPL license and the intellectual property rights questions raised by this process.

Tackling a) first (performance and feasability), we think the Freedom architecture could be more efficient under GNU/Linux compared to other architectures by making it :

1. More compatible with the gcc compiler.We have the source code to gcc, but most importantly, we have the gcc developers available to help us figure out what features they would like to see in a CPU architecture. Why gcc? Because it is the cornerstone of the entire body of Free Software. Basically, an efficient architecture for gcc will see an increase in efficiency across-the-board on *all* Free Software programs.
2. Faster in the Linux kernel. Right now, if we take for example the PC architecture, we notice that the Linux kernel has to "work around" (and some would say "work against") various idiosyncrasies of the x86/PC specifications and hardware. We also have to maintain compatibility with outdated x86 chips. And obviously, there is no possibility of implementing some of the often used Linux kernel functions in silicon. A new design, custom fitted to the Linux kernel code, would vastly improve the performance of any kernel-bound applications.

Further ideas for a possible architecture and implementation can be found in the appendices (as well as the "economics" of the project). Note that we are calling the architecture "Freedom" (for obvious reasons), and its first implementation "F1". Projected end-user cost of an F1 CPU is around $100. Everything is very utopic, we know. : -)

However, it also seems to us that at this stage, the real challenges for our project are entirely within b) : the development process and the intellectual property issues.

3.1.3 Developing the Freedom architecture : issues and challenges

The Dilbert cartoon says it all, in fact : our project *is* a whole new paradigm! What we are basically proposing is to bring together the competences and creative powers of thousands of individuals over the Web into the design process of an advanced, Free, GNU/GPL'ed 64-bit CPU architecture. And we don't even know if it's possible!

We know two things for sure :

- In the past and present, corporations like Intel, IBM and Motorola are known for having broken down design teams, so that no close groups could be formed that would be able to recreate the entire design (and eventually quit and form their own companies). Recently, Andy Grove has given a new meaning to the word "paranoia" as a management tool. Our proposed Free, open, transparent, collaborative environment counters this trend. It is also in a large part related to some new trends in Human Resources management and Organizational theory. In fact, it is very akin to the concept of Virtual Corporations, except that in this case we are rather dealing with a Virtual Non-Profit Organization. In this respect, the Freedom project is also an experiment in Organizational theory, but it's not a gratuitous experiment. Many studies indicate that keeping people in small closed groups, bound by strict NDA and other legal constraints to public silence, and putting a relatively high amount of pressure on these groups, is not the best method to unleash creative powers. It also sometimes leads to buggy designs...

- The development of the Linux kernel, by a group of highly talented programmers/system developers is an example that an open, collaborative environment aiming for a GNU/GPL'ed piece of software with a particularly high intellectual/technological value, is possible. Moreover, it can be shown that in some areas, the Linux kernel performs better then its commercial counterparts. However, this list of certainties is rather short compared to the list of questions generated by our proposal :
  - How will new ideas be selected or discarded for inclusion in the design, amid the inevitable "noise" of Bad Ideas (tm) ? Who will be the judge of what's Good and Bad?
  - Also inevitably, mutually exclusive options/features will appear during the course of development. Again, who will decide on the direction to be chosen?
  - Who will own the final design intellectual property rights? Is the "copyleft" applicable in the case of a CPU design? What about the masks for the first silicon?
  - Will the GPL be sufficient as a legal instrument to protect the design? What changes, if any, will have to be made to the GNU/GPL to adapt it to a chip design?
  - If the design process uses commercial EDA and other tools, in what measure do these proprietary items "taint" our GNU/GPL'ed design? Is it possible to separate the GPL part from the commercial/proprietary parts?
  - What about existing patents? Will the project need any? Will it be able to "buy" any, or pay royalties?
  - Contrarily to a piece of software, partial implementations of the Freedom design will not be possible. The first implementation that will go to silicon *must* be functional and complete. All "holes" in the design must be plugged before the first mask gets drawn. How do we make volunteers accept such a rigid schedule ?

There are some questions raised as a consequence of the possible succes of the Freedom implementation :

- There are vast possibilities for a GNU/GPL'ed CPU design in the industrial, medical, aeronautical, automotive and other domains. In fact, a Free, stable, high-performance design offers possibilities never before envisioned by hardware designers in various domains. Is this the beginning of a small revolution in e.g. embedded hardware ?
- Will the design sustain itself over the years as the ideal GNU/Linux processor ?
- Can this experiment in open development have other consequences on the electronics industry? Are we really proposing a new paradigm for CPU development? Can this paradigm be applied to other VLSI designs?

3.1.4 Tools

We all know the saying : "If the only tool one has is a hammer...". We'll need "groupware" tools for the Freedom project, but the word "groupware" has a bad reputation nowadays. We prefer to use "collaborative work tools". Some of them have only come into existence and widespread use in the last decade; I am obviously talking about the Web itself, and its assortment of communication technologies : email, newsgroups, mailing lists, Web sites, SGML/PDF/HTML documentation and editing/translation software. Much of this infrastructure is/has been used to develop GNU/Linux, and is nowadays based on GNU/Linux, BTW.

But we'll also need new tools, that perhaps don't even exist yet. I think it's worth mentionning that perhaps one the greatest steps in this direction is the WELD project, developped at Berkeley. It could well become the cornerstone of the Freedom project, or conversely, the Freedom project can perhaps be thought of as the ideal, perfect test case for the WELD project.

3.1.5 Conclusion

The conclusion is simple and obvious :

- if you are a CPU architect/VLSI engineer, or
- if you have a good idea on CPU design that you have been toying with for some time and would like to test, or
- if you just like challenging intellectual propositions and brainstorming interaction :

Please join and help us turn this idea into a reality!

–

* : Richard is an Australian astrophysicist preparing his Ph.D. on astronomic visualization; Rafael is a researcher on EDA tools at the University of Cincinatti. I am an ex-Ph.D. student in Management and an ex-firmware engineer, with a special interest in Ethical problems in multi-cultural environments (I was born in Brazil and am presently living in France). None of us has any formal education in CPU architecture. Rafael comes closest, since he is in VLSI design and EDA tools development, and also developed some new code for CPU recognition in the Linux kernel. Richard developed the Pentium Pro MTRR support in the Linux 2.1.x kernels (as well as other novel kernel routines), and is also a hardware developer. I have the honour of having diagnosed the Cyrix 6x86 "Coma" bug and proposed a workaround for it under GNU/Linux (both were at first rejected by Cyrix Corp.). I am also a long time hardware and firmware developer, and have contributed in various ways to GNU/Linux development (e.g. the Linux Benchmarking HOWTO).

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

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

Andrew D. Balsa <andrebalsa@altern.org>

note : today, none of these addresses work. altern.org has even disapeared.

3.1.6 Appendix A

Ideas for a GPL'ed 64-bit high performance processor design

This is just a dream, a utopic idea of a free processor design. It's also a list of things I would like to see in a future processor.

- This project will need a sponsor if it ever wants to become a reality. Getting first silicon is not going to be free, nor easy.
- Choice of a 64-bit datapath, address space : obvious nowadays. Simplifies just about everything.
- Huffman encoded instruction set : improves cache/memory→CPU bandwidth, which is one of the main bottlenecks nowadays. Should be quite simple to add a Huffman encoder to a compiler back-end. All instructions lengths are multiple of byte.
- RISC vs. CISC vs. dataflow debate : it's over! Get the advantages of each, disadvantages of none as much as feasible.
- 1, 2 or 4 internal 7-stage pipelines.
- Speculative execution : 4 branches, 8 instructions deep each.
- 64-byte instruction prefetch queue.
- 32-byte write buffers.
- Microprogram partly in RAM. Must be able to emulate x86 instruction set (assembler source level).
- 64-bit TSC w/ multiple interrupt capabilities.
- Power saving features.
- MMX and 3DNow! emulation.
- Fully-static design (clock-stoppable).
- F1 implementation : 128 bits external data path, 40 bits external addressing capabilities.
- Performance monitoring registers "a la" Pentium.
- External FPU, memory mapped (have no idea what it should look like). FPUs can be added to work in parallel (up to 4?). Separate bus. Same bus can handle a graphics coprocessor with its dual-ported memory.

- 8KB 4-ported L1 unified cache, with independent line-locking/line-flushing capabilities. Can be thought of as a 1 KB register set.
- Separate 64KB each L2 instruction and data caches, running at CPU speed.
- Integrated intelligent DMA controller, 32 channels.
- Integrated interrupt controller : 30 maskable interrupts, 1 System Management interrupt, 1 non-maskable interrupt.
- 0 internal registers ! Yep, this is a memory-memory machine. Instruction set recognizes 32 pseudo-registers at any moment.
- Interrupts cause automatic register set switch to vectored register set : 0 (zero) context switch latency!
- No penalty for instructions that access byte, word, dword data.
- Operation in little or big-endian mode "a la" MIPS.
- Paging "a la" Intel, with 4k pages + 4M extension.
- Also VSPM "a la" Cyrix 6x86, with 1K definable pages.
- ARR registers "a la" Cyrix 6x86 (similar to MTRR on Intel PPro) : allows defining non-cacheable regions (useful for NUMA, see below).
- Internal PLL with software programmable multiplier; can switch from 1x to 2x to 3x to nx in 0.5 increments, on-the-fly.
- The MMU should also support object protection "a la" Apple Newton.
- Single-bit ECC throughout.
- Direct support of 4 1MB dual ported memory regions for NUMA-style multiprocessing (also on FPU bus).
- CPU architecture project name : "Freedom". Could also be called "Merced-killer", or "Anti-Merced", or "!Merced", but in fact we are not anti-anything with this project. We are just pro-Freedom and open; what we dislike about the Intel Merced is its proprietary design and restrictive development environment. I guess the challenge here is to determine whether a GPL'ed CPU design is feasible. Is open, collaborative development possible WRT CPU design? How does one get the funding to actually put the design on silicon, once it is ready? How can revisions be handled? Are there patents that would inherently block such a development process?

The idea also is to use gcc as the ideal development compiler for this CPU (unlike Merced). And to be able to port the Linux kernel with a minimal effort on this new processor.

3.1.7 Appendix B

Freedom-F1 die area / cost / packaging physical characteristics / external bus
Just as a reminder, the F1 CPU does not include an FPU or 3DNow! unit (but SIMD integer instructions will be included).

Recommended maximum size : 122 sq. mm. This gives us 200 dies/8-inch wafer (see an example of such a wafer on Hennessy and Patterson, page 11).

Roughly, die yield = 0.5 for our 122 mm2 5-layer 0.25 micron CPU (H AND P, page 13, updated to reflect better fabs). This allows more or less 10-11 million transistors, divided as follows : 6-7 million for the caches, 4-5 million for the rest.

Assume wafer yield = 95, final test yield = 95. Testing costs of $500/hour, 20 seconds/CPU.

Packaging costs = $25-50 (see below).

Roughly, following H and P, this gives us a unit cost of $75-100/good CPU, tested, boxed in anti-static packaging and shipped to the US, if the Taiwan foundries can keep the wafer processing cost around $3.500.

Packaging : I am going to propose something surprising, but I think we should use the same packaging as the Celeron CPU, in terms of physical dimensions and CPU placement. Like that we can also use the Celeron heatsink/fans already in the market, and the Celeron mounting hardware.

PCI set : again I am going to propose a heresy, but I think we could use 100MHz Slot 1 motherboards. First, Intel is not alone anymore manufacturing Slot 1 chipsets : VIA has just released a Slot 1 chipset with excellent performance and the latest goodies in terms of technology (we can get timing info from the VIA chipset datasheets). Second, we don't have to worry about the motherboard/PCI set issue anymore. Third, it's almost impossible to go beyond 100MHz on a standard motherboard, because of RFI issues; so basically 100-112MHz is as good as it gets. Fourth, there will be many people out there with Slot 1 motherboards, willing to upgrade their PII/Celeron CPUs (specially the Celeron). Fifth, these motherboards are nowadays quite cheap, and we get all the benefits of high-volume production. Sixth, this allows easy upgrades of the Freedom CPU to higher speed grades, larger cache versions, FPU-with versions, etc.

Now, if we accept the above, we have to put on the Celeron-style Freedom printed circuit a small EEProm that will contain the Freedom BIOS, the L2 cache and a socket for the FPU. This increases the cost of the CPU, but decreases overall costs, so I still think it's a good move.

Please check a photograph of the Celeron and tell me if I am just dreaming.

3.1.8 Appendix C

Legal issues / financial issues

August 5, 1998

We would like to have support from the Free Software Foundation for the Freedom project.

We are not proposing that the Free Software Foundation build a fab. What we are saying is : if we go to a foundry in the US or Taiwan, give them a mask, and ask them to run a batch of 0.25 micron, 5 layer 8-inch wafers for us, they'll quote approx. $3K-5K or less even, per wafer, as their price (our cost) for our batch (in the year 2000).

An approximate cost for a batch of F1 CPUs would theoretically be somewhere between $500k and $1000K, for 5000-10000 good CPUs.

Not exactly pocket money, but we could sell those CPUs on a subscription basis. Like this : people who would subscribe would get the Merced-killer for around $100 (compare that to the projected cost of $5000/unit for the Merced), on a first-come/first served basis, and any left-over CPUs after the cost of the batch would be covered, could be sold for a slightly higher price to pay for the next batch and further mask development.

We suggest putting some quotas in the system. Demand is likely to be higher than supply. ;-)

The Free Software Foundation could coordinate all the legal/financial/logistic aspects of the project (and would be adequately compensated for this work). This, of course, would depend on getting support from Mr. Stallman for this initiative.

Chapter 4

A bit of F-CPU history

(And a reflexion on the evolution of the F-CPU through a description of the different proposed architectures)

4.1 M2M

The first generation was a "memory to memory" (M2M) architecture that disapeared with the original F-CPU team members (they have written the previous text). It was believed that context switch time consumed much time, so they mapped memory regions to the register set, as to switch the registers by changing the base register. I have not tracked down the reasons why this has been abandonned, I came later in the group. Anyway, they launched the F-CPU project, with the goals that we now know, and the dream to create a "Merced Killer". Actually, i believe that we should compete with the ALPHA directly ;-)

4.2 TTA

The second generation was a "Transfer Triggered Architecture" (TTA) where the computations are triggered by transfers between the different execution units. The instructions mainly consist of the source and destination "register" numbers, which can also be the input or output ports of the execution units. As soon as the needed input ports are written to, the operation is performed and the result is readable on the output port. This architecture has been promoted by the anonymous AlphaRISC, now known as AlphaGhost. He has done a lot of work on it but he has left the list and the group lost track of the project without him.

Brian Fuhs (bkfuhs1@attglobal.net) explained TTA on the mailing list this way :

```
TTA stands for Transfer-Triggered Architecture. The basic idea is that you don't tell
the CPU what to do with your data, you tell it where to put it. Then, by putting your
data in the right places, you magically end up with new data in other places that
consists of some operation performed on your old data. Whereas in a traditional OTA
(operation-triggered architecture) machine, you might say "ADD R3, R1, R2", in a TTA
you would say "MOV R1, add; MOV R2, add; MOV add, R3". The focus of the instruction set
(if you can call it that, since a TTA would only have one instruction : MOV) is on the
data itself, as opposed to the operations you are performing on that data. You specify
only addresses, then map addresses to functions like ADD or DIV.

That's the basic idea. I should start by specifying that I'm focusing on general
processing here, and temporarily ignoring things like interrupts. It is possible to
handle real-world cases like that, since people have already done so; for now, I'm
more interested in the theory. Any CPU pipeline can be broken down into three basic
stages : fetch and decode, execute, and store. Garbage in, garbage processing, garbage
out. With OTAs this is all done in hardware. You say "ADD~R3,~R1,~R2", and the hardware
does the rest. It handles internal communication devices to get data from R1 and R2 to
the input of the adder, lets the adder do its thing, then gets the data from the output
of the adder back into the register file, in R3. In most modern architectures, it
checks for hazards, forwards data so the rest of the pipeline can use it earlier, and
might even do more complicated things like reordering instructions. The software only
```

knows 32 bits ; the hardware does everything else.

The IF/ID stage of a TTA is very different. All of the burden is placed on\\
software. The instruction is not specified as ADD (something), but as a series of\\
SRC, DEST address pairs. All the hardware needs to do is control internal busses\\
to get the data where it is supposed to go. All verification of hazards, optimal\\
instruction order, etc should be done by the compiler. The key here is that a TTA,\\
to achieve IPC measures comparable to an OTA, must be VLIW : you MUST be able to specify\\
multiple moves in a single cycle, so that you can move all of your source data to the\\
appropriate places, and still move the results back to your register file (or wherever\\
you want them to go). In summary, to do an "ADD~R3,~R1,~R2", the hardware will do the\\
following :

TTA	OTA
MOV R1, add	ADD R3, R1, R2
Move R1$\rightarrow$adder	*Check for hazards*
MOV R2, add	*Check for available adder*
Move R2$\rightarrow$adder	*Select internal busses and move data*
(adder now does its thing in both cases)	
MOV add, R3	*Check for hazards*
Move adder$\rightarrow$R3	*Schedule instruction for retire*
	Select internal busses and move data
	Retire instruction

TTA stands for Transfer-Triggered Architecture. The basic idea is that you don't tell

The compiler, of course, becomes much more complicated, because it has to do all of the scheduling work, at compile time. But the hardware in a TTA doesn't need to worry about much of anything... About all it does in the simple cases is fetch instructions and generate control signals for all of the busses.

Execution is identical between TTA and OTA. Crunch the bits. Period.

Instruction completion is again simplified in a TTA. If you want correct behavior, make sure your compiler will generate the right sequence of moves. This is compared to a OTA, where you at least have to figure out what write ports to use, etc.

Basically, a TTA and an OTA are functionally identical. The main differences are that a TTA pretty much has to be VLIW, and requires more of the compiler. However, if the "smart compiler and dumb machine" philosophy is really the way to go, TTA should rule. It exposes more of the pipeline to software, reducing the hardware needed and giving the compiler more room to optimize. Of course, there are issues, like code bloat and constant generation, but these can be covered later. The basic ideas have been covered here (albeit in a somewhat rambling fashion... I had this email all composed in my head, and had some very clear explanations, right up until I sat down and started typing). For more information see http://www.cs.uregina.ca/~bayko/design/design.html and http://cardit.et.tudelft.nl/MOVE. These two have a lot more information on the details of TTA; I'm still hopeful that we can pull one of these off, and I think it would be good for performance, generality, cost, and simplicity. Plus, it's revolutionary enough that it might turn some heads - and that might get us more of a user (and developer) base, and make the project much more stable.

Send me questions, I know there will be plenty ...

Brian

If you want to understand further the TTA concept, the difference is in the philosophy, it's as if you had instructions to code a dataflow machine on-the-fly. Notice also the fact that less registers are needed : registers are required to store the temporary results of operations between instructions of a code sequence. Here, the results are directly stored by the units, there are less "temporary storage" needed, less register pressure.

To envision this difference, think about a data dependency graph : in OTA, an instruction is a node, while in TTA the mov instruction is the branch. Once this is understood, there's not much work to do on an existing (yet simple) compiler to generate TTA instructions.

Let's examine : $S = (a + b) * (c - d)$ for example. a, b, c, d are known "ports", registers or TTA addresses.

```
a   b c   d
1\ /2 3\ /4
  +     -
  5\   /6
    \ /
     *
     |7
     S
```

In OTA, with 3-operands instructions, there is in this case one instruction per "node" (+, -, *). Two temporary registers are needed to store the result of the addition and the substraction (branches 5 and 6). Let's assume that the tree-flattening must preserve superscalarism (well, instructions have latencies), so we code :

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

(for example).

In TTA there is one "port" in each unit for each incoming branch. This means that ADD, having two operands, has two ports. There is one result port, which uses the address of one port, but that is used as read, not write. Another detail is that this read port can be static : it holds the result until another operation is triggered. We can code

```
mv ADD1,a
mv SUB1,c
mv ADD2,b    (this triggers the a+b operation)
mv SUB2,d    (this triggers the c-d operation)
mv MUL1,ADD
mv MUL2,SUB  (this triggers the * operation)
```

TTA is not "better", it's not "worse", it's just completely different while the problem will always be the same. If the instructions are 16 bit wide, it takes 96 bits, just as the OTA example would do. In some cases, it can be better as it was shown long ago on the list. TTA has some interesting properties, but unfortunately, in the very near future, it's not probable that a TTA will enter inside a big computer as RISC or CISC do. A TTA core can be as efficient as the ARM core, for example, it suits well to this scale of die size, but too few studies have been made, compared to the existing studies on OTA. Because the solution of its scaling up are not (yet) known, this leads to the discussions that shaked the mailing list near december 1998 : the problem of where to map the registers, how would the ports be remapped on the fly, etc. When additional instructions are needed, this jeopardizes the whole balance of the CPU and evolutivity is more constraining than for RISC or OTA in general.

The physical problem of the busses has also been raised : if we have say 8 buses of 64 bits, this makes 512 wires, it takes around one millimeter of width with a .5u process. Of course, we can use a crossbar instead.

As discussed a few times long ago, because of its scalability problems (assignation of the ports and its flexibility), TTA is not the perfect choice for a very long-lasting CPU family, while its performance/complexity ratio is good. So, it would be possible that the F-CPU team makes a RISC →

TTA translator in front of a TTA core that would not have most of the scalability problems. This would be called the "FC1" (FC0 is the RISC core). Of course, time will show how the TTA ghosts of the F-CPU group will change.

But TTA's problem is probably that it is too specialized, where OTA can change its core and still use the same binaries. It's one of the points that "killed" the previous F-CPU attempt. Each TTA implementation could not be completely compatible with another, because of the instruction format, of the assignation of the "port" and other similar details : the notion of "instruction" is bound to the notion of "register".

I am not trying to prove the advantage of one technique over another, i am trying to show the difference of point of view, that finally treats the same problem. The scalability, that is necessary for such a project, is more important than we thought, and the group finally showed interest for a more classical technology when AlphaRISC left.

4.3 Traditional RISC

The third generation rose from the mailing list members who naturally studied a basic RISC architecture, like the first generation MIPS processors or the DLX (described by Patterson and Hennessy in their "QA" book), the MMIX (Knuth), the MISC CPUs (such as Chick Moore's Forth engines or Bernd's 4Stack), and other similar, simple projects. These designs are explained and described in well-known books and taught in universities. From a simple RISC project, the design grew in complexity and won independence from other existing architectures, mainly because of the lessons learnt from their history and the specific needs of the group, which led to adapted choices and particular characteristics. This is what we will discuss in the next parts of this document.

Chapter 5

The design constraints

The F-CPU group is rather heterogeneous but each member has the same hope that the project will come true, because we are convinced that it is not impossible and therefore feasible. Let's remember the Freedom CPU Project Constitution :

```
"
   To develop and make freely available an architecture, and all other intellectual
property necessary to fabricate one or more implementations of that architecture, with
the following priorities, in decreasing order of importance :

  1) Versatility and usefulness in as wide a range of applications as possible
  2) Performance, emphasizing user-level parallelism and derived through intelligent
         architecture rather than advanced silicon process
  3) Architecture lifespan and forward compatibility
  4) Cost, including monetary and thermal considerations

"
```

We could also add : 5) be successful !

This text sums up a lot of aspects of the project : this is "free intellectual property", meaning that anybody can make money with it without worrying, as long as the product complies with the general rules and standards described in the F-CPU charter, and all the characteristics are freely available (under the GNU Public Licence and respecting the F-CPU charter). Just like the LINUX OS project, the team members hope that the free availability of this design will benefit everybody by reducing the cost of the products (since most of the intellectual work is already performed), by providing an open and flexible standard that anyone can influence at will without signing a NDA. It is also the testbench of new techniques and the "first CPU" for a lot of "hobbyists" that can build it easily at home. Of course, the other expected result is that the F-CPU will be used in everybody's home computer as well as by the other specialized markets (embedded/real time, portable/wearable computers, parallel machines for scientific number crunching...).

In this situation, it is clear that one chip does not fit all needs. There are economic constraints that also influence the technologic decisions, and everybody can't access the most advanced silicon fabrication units. The reality of the F-CPU "for and by everybody" is more in the realm of the reconfigurable FPGAs, the low-cost sea-of-gates and ASICs that are fabricated in low volumes. Even though the final goal is to use full-custom technologies, there is a strong limitation for the prototyping and the low-volume quantities. The complexity is limited for the early generations and FC0, the estimated transistor count for the first chips would be 1 Million, including some cache memory. This is rather tight, compared to the current CPUs but it's huge if one remembers the ARM core or the early RISC CPUs.

The "Intellectual Property" is available as VHDL'93 (or VERILOG) files that anyone can read, compile and modify. A schematic view is also often needed to understand the function of a circuit at the first sight. The processor will therefore exist more in the form of a software description than a hardware circuit. This will help the processor families to evolve faster and better than other commercial ones, and this polymorphism will garantee that anyone finds the best core in any situation. And since the development software will be common to all the chips, freely available through the GPL, porting any software to any platform will be eased to the maximum.

The interoperability of the software on any member of the family is a very strong contraint, and probably the most important design rule of the project : `"NO RESSOURCE MUST BE BOUND"`. This led to create a CPU with an "undetermined" data width. A F-CPU chip can implement a characteristic datawidth of any size above 32 bits. Portable software will respect some simple rules so that it will run as fast as the chip can, independently from algorithmic considerations. In fact, the speed of a certain CPU is determined by the economic constraints, and the designer will build a CPU as wide as the budget and the technology allow. This way, there is no other "roadmap" than the user's needs, since he is his own funder. The project is not bound by technology and is flexible enough to last... as long as we want.

Chapter 6

The project's roadmap

Here are the steps that the project intends to follow in the future. There is NO SCHEDULE because this is a naturally growing project, not a commercially oriented product ; we are more concerned by the pertinence and efficiency of the chip than time-to-market, and several "coopetitors" can change the priorities of the F-CPU team. This roadmap is not definitive, it has already changed and it will change in the future. It helps understand the orientations of the team's work. The following milestones are very important though and show that this is an EVOLUTIVE project rather than a truely ground-breaking utopia.

Generation	Prototype	Pre-series	Commercial class
Codename	"POC" : Proof Of Concept	"TOY" : need I say more ?	F1, F2, F3 ... other nicknames wil be found (and trademarked)
Goal	Have a "chip" that can be shown or demonstrated at trade shows / conferences, make the FC0 core work, test it, explore the memory interface and its performance impact, make a first chip that works, prove the initial architectural assumptions, prove that the F-CPU concept is possible. *It is NOT intended to be a commercial chip because of the very limited functions it provides. Other scaled-down F-CPUs should be derived from more advanced designs, starting with the "commercial class".*	Provide the first users with an advanced, yet limited platform for testing the F-CPU for real. Allow people to write real-world software and have experience with the instruction set and the programming habits, in order to further modify the instruction set and the architecture for the commercial class. *It is not either a design from which other architectures should be derived. The goal is to reassign the opcode map and learn to design ASICs, as well as to gain publicity/ press coverage / hype.*	Define a hardware platform from which other pin-compatible chips can be derived. The "motherboard" and the I/O interfaces should giver as much free space as possible for further enhancements. "Cooopetitors" will have a common ground from which to develop efficient chips. The main problem being the memory bandwidth the memory interface will be VERY wide as to keep the following chips from being memory-starved. *At the time, a first stable version of the reference architecture will be officially released. It will then evolve naturally.*
Technology	CMP / Europractrice / ATMEL / HITACHI depending on the sponsors, opportunities and available budget. Probably around 0.35, 5V. It could be a design contest prize.	ATMEL / HITACHI depending on the sponsors, opportunities and available budget. Probably 0.35 or 0.25, 3.3V	Depending on anyone's whims

Speed	One of the fun stuff to do is to clock it whith an an external PLL. Since the memory will be asynchronous to the core, we will be able to test the capacity of the core to stand very high and low working frequencies. I have absolutely no idea of the frequencies we can get this way.	At least, more than the proto.	As fast as you can...
Number	Half a dozen	a few hundreds or thousands	A lot more !
Word size	64	64	64 or more (any power of 2, above 32 bits)
Memory addressing range	logical : 64 bits physical : 20 (+5) bits (economic)	logical : 64 bits physical : 32 (+5) bits offchip + 4xSDRAM slots (mux[10+12](+5) = 27 bits) of private memory (comfortable ...)	logical : 64 bits Physical : 64 (+5) bits offchip, 4 or 8 SDRAM slots (28bits) (ready to make big cluster)
External memory bus widths	64 bits (private asynchronous SRAM) + 8 bits (debug port)	128 bits + 16 ECC for private SDRAM, 32 of multiplexed + bursted + asynchronous "I/O" bus (memory-mapped)	256 bits + 32 ECC of external memory bus (DDR-SDRAM ?) + 64 bits of memory-mapped "IO" (multiplexed, bursted, asynchronous)
JTAG / on-site debug	custom byte-wide interface	JTAG (or similar) + I/O bus (used for fast examination / debug port)	JTAG + I/O port
Cache	Onchip data + instruction, 2KB each.	Onchip data + instruction, 4 or 8 Kb each.	Onchip data + instuction, 8 Kb or more each. External cache : data bus shared with the SDRAM, onchip TAG SRAM
Instructions per cycle	1	1	1 or more
Core	FC0	FC0	FC0 and others
Lifespan	Short (months)	Short (not more than a few years)	Much longer :-)
Evolutivity / Compatibility	None (proto)	None	Yes
Motherboard (CPU Module)	Breadboard or 2-layers PCB, interface with ISA bus or similar	High quality 6-layers PCB + home-made (breadboard or 1-layer) I/O PCB	High quality, high volume production-class PCB + I/O + intercom + EEPROM PCB (A PCI, AGP, IDE/SCSI bridge will be needed)
Target / users	F-CPU team, demonstrators and advanced users	Programmers / Developpers / Advanced Integrators	Anybody (above 10yo)

We hope that this table answers most of your questions. If not, do NOT hesitate to ask.

Part II

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 <u>Part 5 : The F-CPU Instruction Set Architecture</u>. 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.

Part III

General description of the FCPU Core #0

Chapter 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 implements an *Out Of Order Completion* pipeline

It is a simple solution if we want 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 :

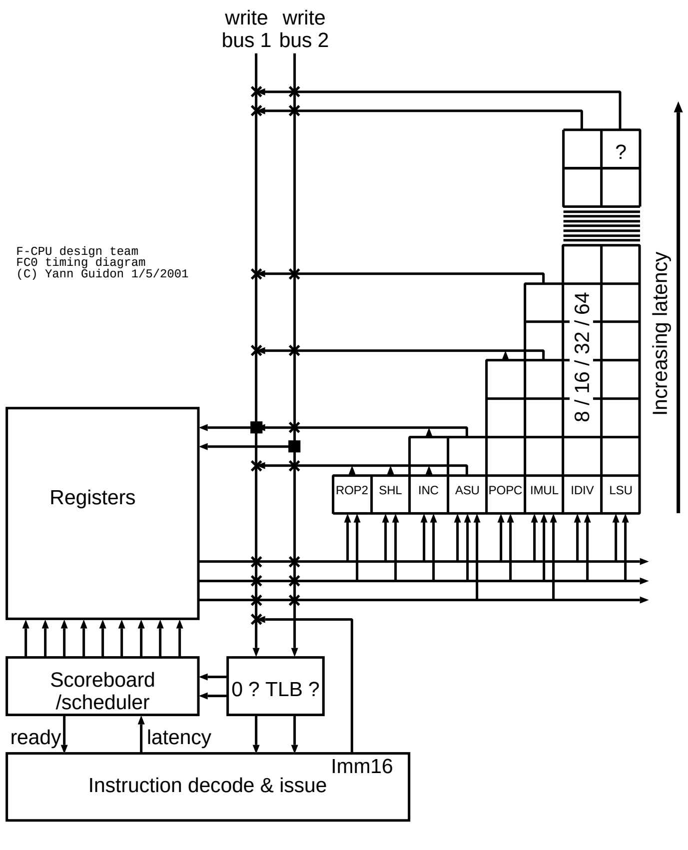

Figure 1.1: The pipeline is folded around the Xbar

Chapter 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.

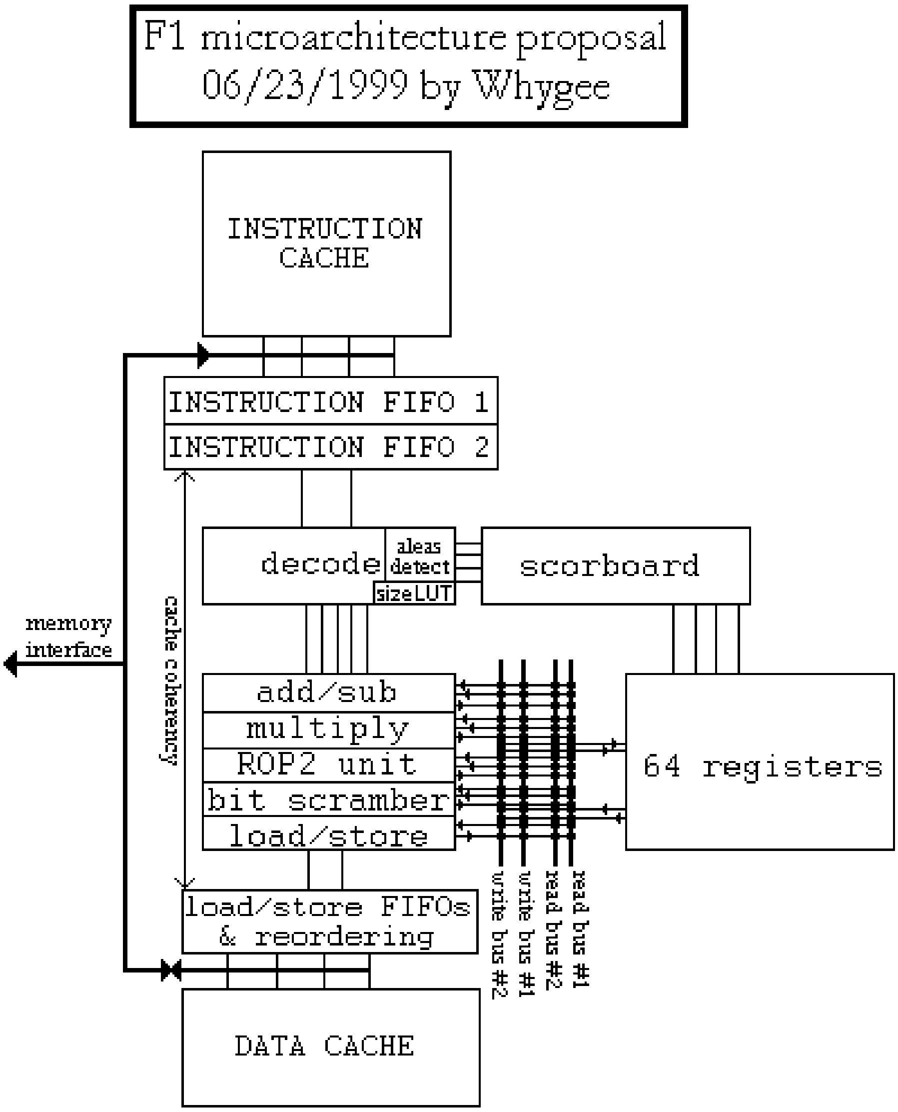

Figure 2.1: The first F-CPU chip proposal

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.

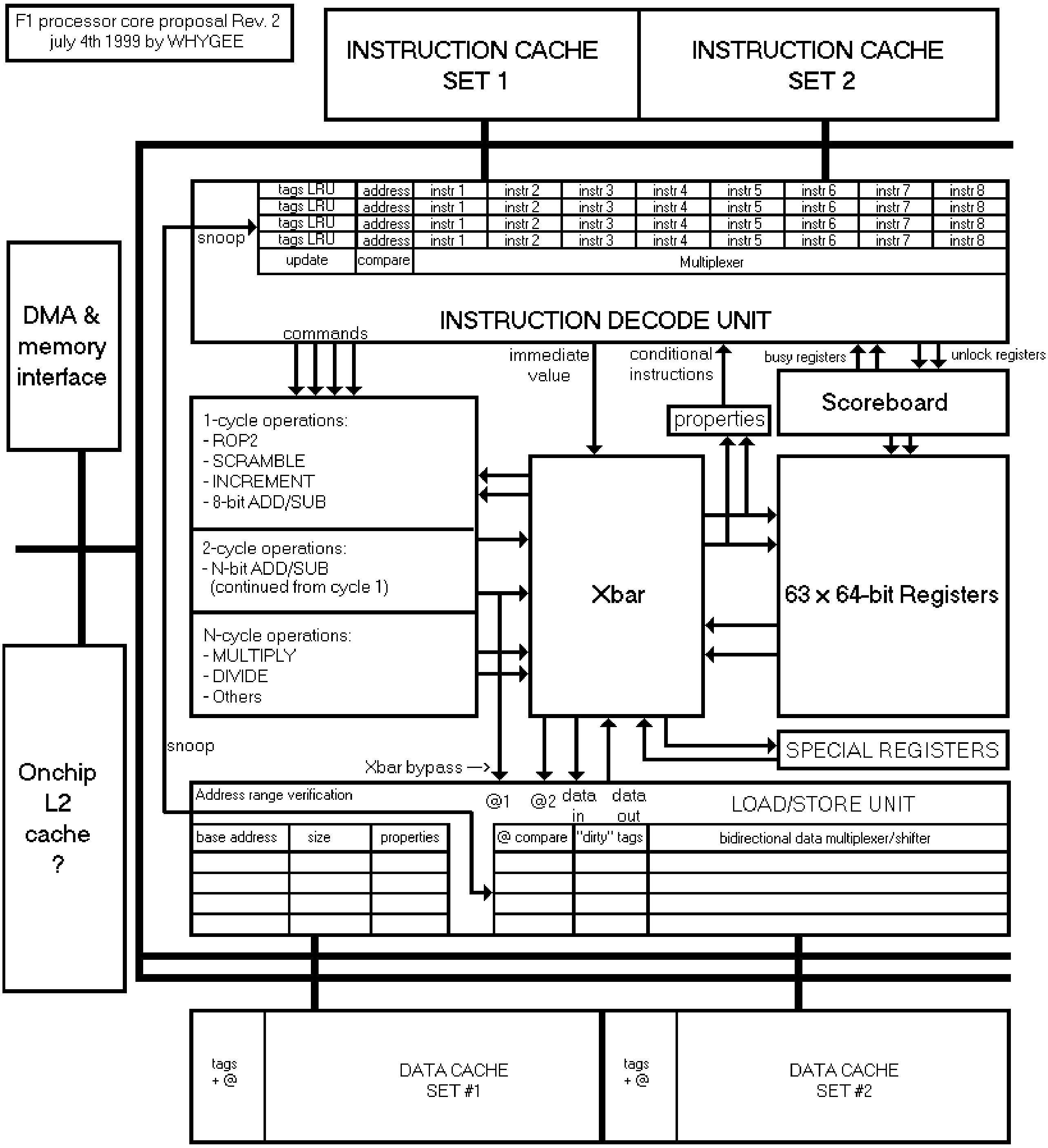

Figure 2.2: A more precise, first-attempt F-CPU description

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.

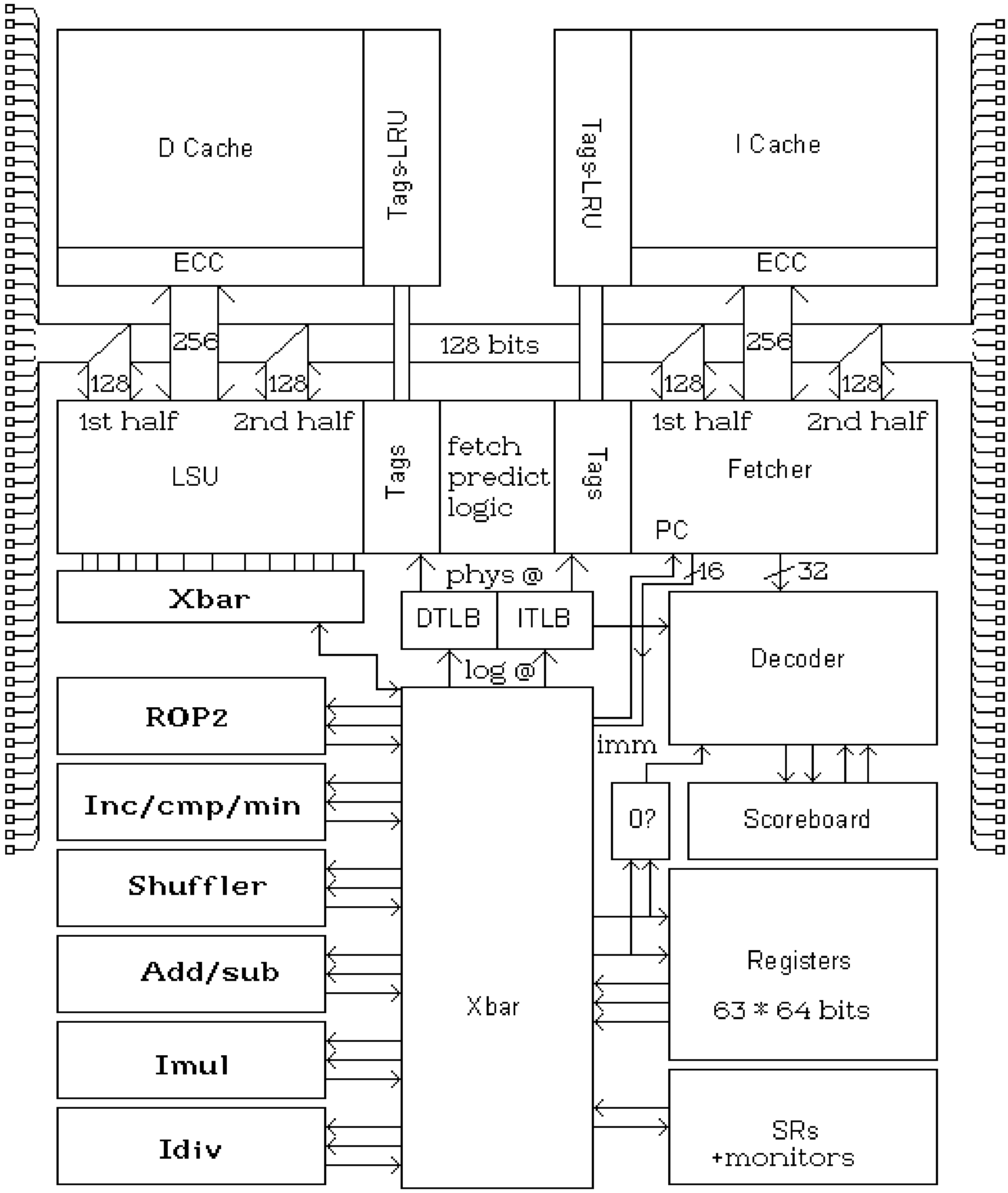

Figure 2.3: A third F-CPU description

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.

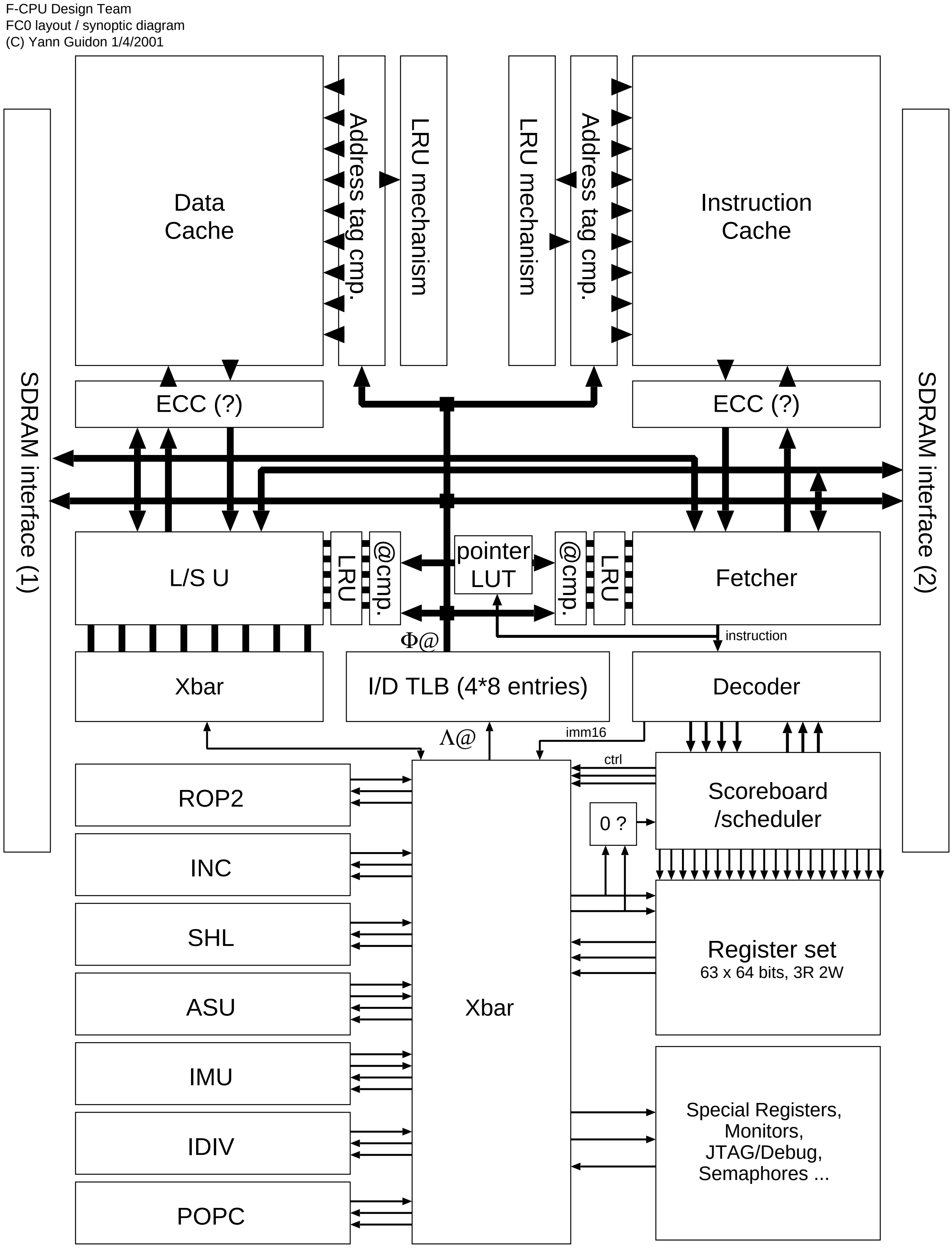

Figure 2.4: The current F-CPU diagram

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. We can see that the overall shape doesn't change much but is refined.

Chapter 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 :

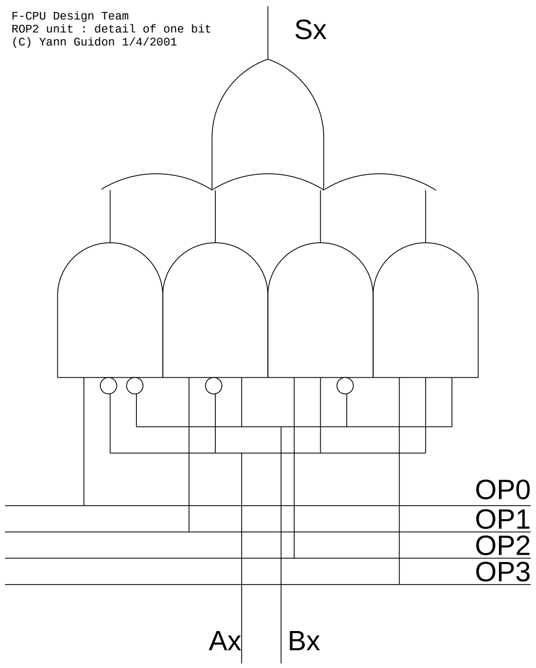

Figure 3.1: Detail of the ROP2 unit

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.

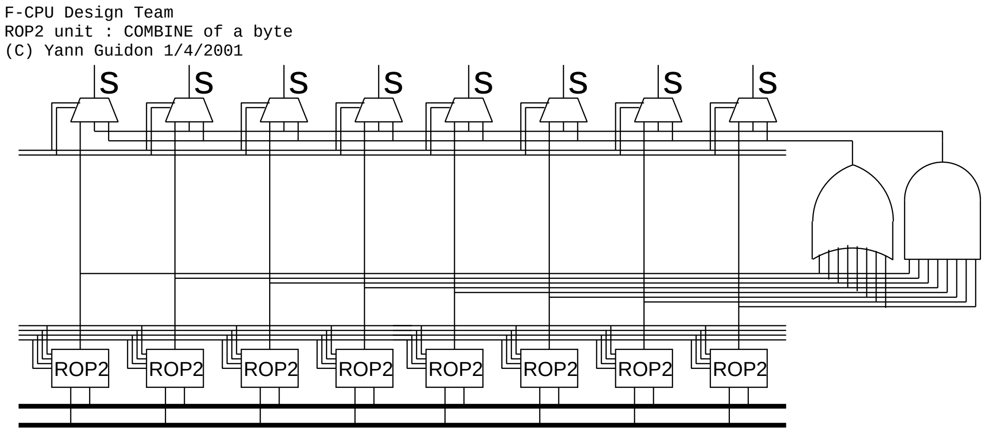

Figure 3.2: Description of the COMBINE function on top of ROP2 for a byte-wide SIMD packet

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.

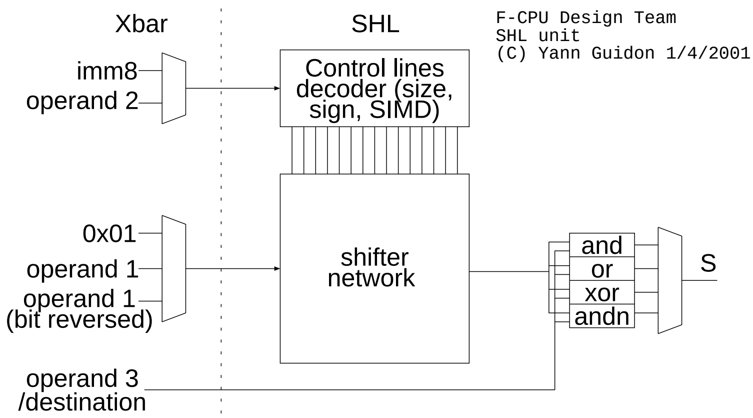

Figure 3.3: Overview of the Scrambling unit

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.

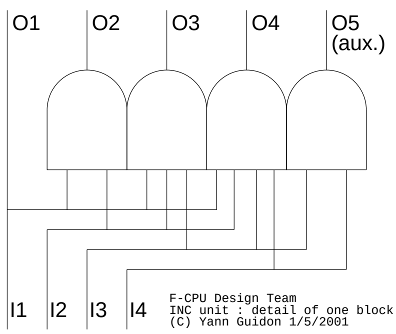

Figure 3.4: Description of one block of the AND tree

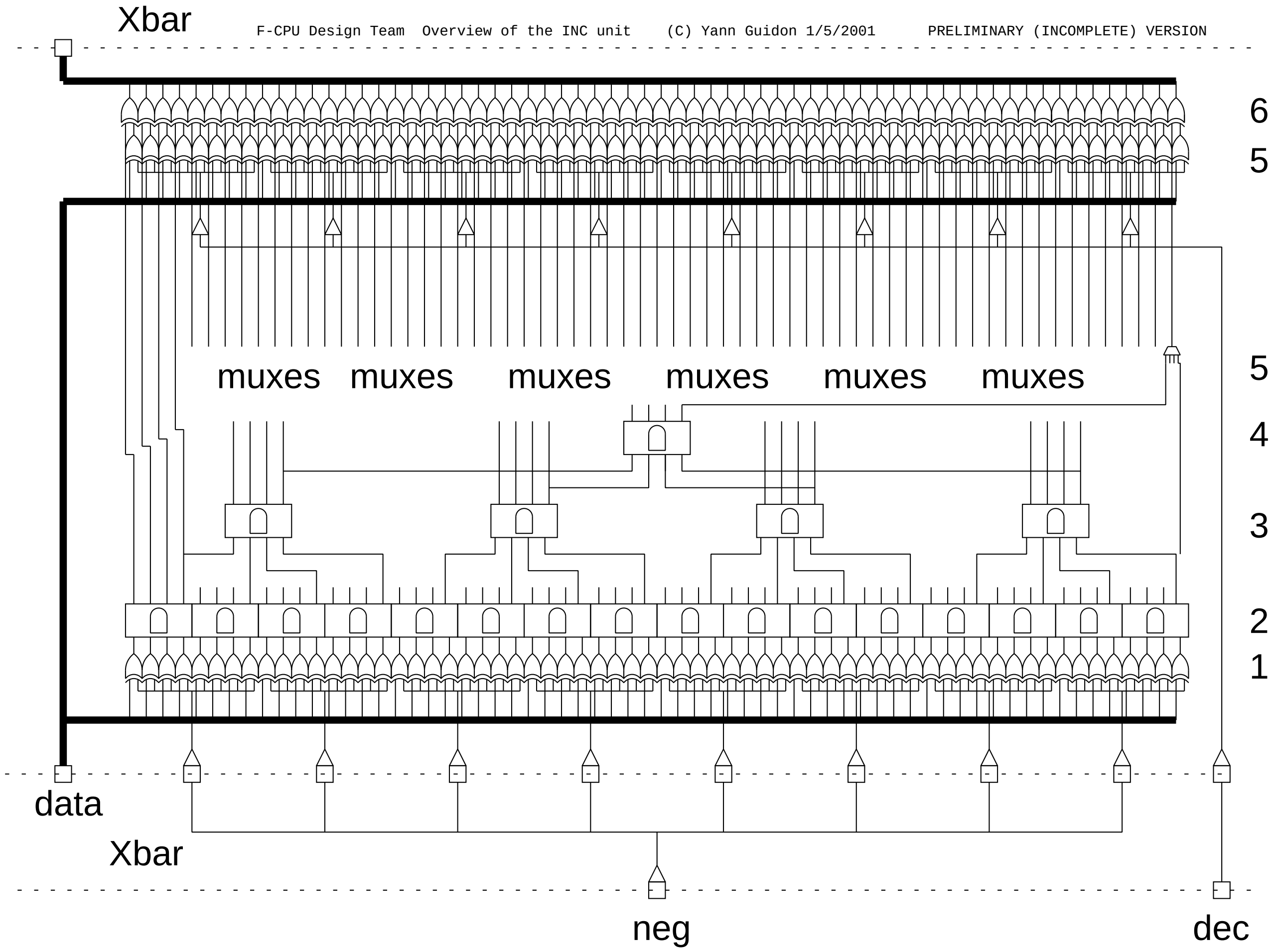

Figure 3.5: Overview of the Incrementer Unit (preliminary version)

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 (LNS). 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. A cooperation with other research teams is encouraged.

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.

3.10 Extensions and scalability

If you want to add your own custom Execution Unit to the F-CPU, it is rather simple : you first have to prepare the Instruction Set map and the decoder, so the necesary instructions are easy to decode and they don't conflict with other instructions. Then, you have to ensure that the scheduling and the exceptions don't jeopardize the decoder unit's structure (see the discussions in the following parts). Finally, you "plug" your unit on a newly created port of the Xbar.

Depending on your target technology, you can add an undetermined number of new units : the FC0 architecture does not limit the number of physical execution units. The physical limits are however important and the Xbar can't be extended endlessly : the design goal (6-gate critical datapath with 4 input max. per gate) must also be respected.

The widths of the data are also the parameters that play in favor of the FC0 and the F-CPU in general : the extension of the register width or the chunk width allow the engineer to scale the design up easily. Again, the design goals must be respected but this is another simple way to extend the architecture without redesigning everything from scratch. For example, the register width and the chunk width are decoupled, so they can be changed independently.

Part IV

Advanced topics

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.

Chapter 1

The exceptions

A processor of any kind (CISC, RISC or any other architecture) generates a lot of exceptions, interrupts, traps and system calls (here, context switches are not the point). Each pipeline stage can generate several errors that the OS must handle, which requires that the application must "restart" the trapped instruction or continue after the trap. This implies that the whole context must be saved, but which ?

Control can be transfered to the OS, an interrupt handler or a trap handler at anytime, at any stage of the pipeline. A classic RISC pipeline comprises (and generates) for example :

- IF (Instruction Fetch) : page fault
- ID (Instruction Decode) : invalid instruction, trap instruction, privileged instruction.
- EX (EXecute) : divide by zero, overflow, any IEEE FP math error...
- MEM (MEMory access) : page fault, protection error

Not only should the processor trigger the correct handler (because several errors can occur in the same cycle) but it must also preserve or flush the correct stages of the pipeline. And since FC0 completees the operations OOO, it is too complex to do without a lot of buffers everywhere as well as sophisticated bookkeeping, which we can't afford for obvious reasons. We need to keep precise exception anyway, and the ability to stop the pipeline at any time without losing data that would require some code to be reexecuted. We need a simple and predicatable yet efficient pipeline that is not influenced in its architecture by faults.

The simplest alternative to this problem is dictated by good sense : make all the exception occur at one place, before the potentially faultive instructions enter the pipeline and require additional hardware. This means : *NO INSTRUCTION IS ISSUED IF IT CAN TRIGGER AN EXCEPTION* or, in other words, *ALL EXCEPTIONS MUST BE CHECKED AT DECODE TIME AS TO PREVENT THEM FROM OCCURING IN THE EXECUTION PIPELINE.* Remember this clearly, meditate about this, since it influences how the instruction set is designed too.

The good side of this choice is that there is no "trap source" register as in the MIPS CPUs. All exceptions are caught at the same place and are disambiguified and ordered implicitely. Another important good consequence is that there is no temporary buffer or "renamed registers" as called in the PowerPC. The previously described OOOC pipeline is not changed at all and the critical datapath does not suffer from additional buffers. There is no register allocation bookkeeping, nor added control logic.

The other side, which is about the constraints, is discussed here. Most obvious limitations have simple turnarounds. The first problem is : can we detect all the exceptions at decode time and how ?

<u>First cause :</u> page fault at instruction fetch time.

First, we are not absolutely sure that we will even decode the next coming instruction, since the last instruction of a page could be a jump, or any similar instruction. So why trigger the trap now ? The easy turnaround to this problem is to "tag" the instruction as faultive or, better, replace it with a trap instruction (which requires less hardware). So, if the instruction is executed, it will trap. Simple, isn't it ? Of course, if a page fault is triggered by the instruction prefetch unit, it is a good practice to directly prefetch the necessary code before it is needed. Just by precaution.

Second cause : invalid instruction, privileged instruction...

Why bother ? It traps. Depending on the type of trap, we will advance the instruction pointer or not, fetch the needed code to execute it, and begin to backup of the registers with the SRB mechanism. The precedent instructions don't need to be flushed from the pipeline, because the SRB will communicate with the scoreboard to backup the registers in a correct order. When the pipeline will be "naturally" flushed from the old application's instructions, the registers will be saved and the faultive application will restart later without any loss or reexecution.

Third cause : math fault.

The saturation (or overflow) exception (a la MIPS) is not implemented. The IEEE Floating Point instructions have a "compliance" flag that stops the instruction issue until the result is "safe", otherwise the result will sturate and not trigger any trap. The "division by zero" condition is easily detected at decode stage with the ZERO property bit of the dividing register. At the same time, we can detect if the result will be zero and issue a "clear" operation instead of the divide operation.

Fourth cause : page fault, invalid address fault.

We can consider that the memory is protected on a page granularity basis, so the page fault will trigger a protection checking code before loading the page. But detecting a page fault is very simple : we have to check the address with the values contained in a page table. If the address does not correspond to the available pages, it is a page fault : we trap.

Now, the problem is to have the status (page present or not ?) at decode time. Let's be smart, because memory accesses are almost half of the executed instructions !

The alternative is to use a similar mechanism to the ZERO "property" bits of each registers. This means that when a value is written to the register set through the Xbar, some ports of the Xbar communicate the value to the page table. In one cycle or two, the data is ready for the ID stage, this is a speculative check that is transparent to the instruction set architecture. In this page check time, we can also check for the address range, verify if the value is in L1 cache , and if yes, indicate in which bank it is and prefetch the cache line, etc...

An obvious problem though is that we can't seriously check all the values flowing through the Xbar to the reg set. Not only this is not always useful but it also consumes power. The simplest way (for the prototype) is to check the result of the pointer updates since they are most likely to be reused soon as pointer.

For more sophisticated architectures, another "transparent tag", saying that the register is used as a pointer, can be very useful. We can allow for example only a few registers to hold this tag, something like 16 (64/4 sounds reasonable) and this flag would be set each time a memory access is performed with this register. The flags would be allocated with a LRU mechanism using a 4 bit down counter. This way, when the ID recognizes a memory read/write instruction, it checks the pointer flag and if set, sends the associated informations to the L/S unit (informations like : in which L1 bank the data is, or in which buffer, etc.) or it traps if the page table lookup returned a negative value. If the pointer flag is not set, the ID pauses for a page table lookup and sets the pointer flag. Of course, like all transparent flags, their value is not saved during context switches and is regenerated automatically as soon as they are used. In the absence of explicit flags in the instructions, this is a rather simple way to reduce the table lookup overhead, and the address can be checked BEFORE it is needed. The L/S unit is only in charge of buffering the data that flows to/from memory and caches. This last detail invalidates the drawing of the figure 2.2 where the page table was stored in the L/S Unit.

There, almost all exception causes are covered and the turnarounds have been explained. There is no visible impact to the ISA but coding rules are getting tighter, like in a superscalar processor. Anyway the turnarounds of the problems caused by the "exception-less" execution pipeline of the FC0 are known and explained. Other new exceptions will probably use the same idea of the existing exceptions : using a dynamic flag. This way, programming the FC0 looks almost like programming a normal RISC CPU with some additional coding rules.

Chapter 2

The Smooth Register backup mechanism

As described in the first section of the document, in the "64 registers" discussion, one alternative to register windowing, banked register sets or memory-to-memory architectures is to implement a "Smooth Register Backup" (SRB for short) for automatic register saving. It is not an usual feature in a microprocessor, because it is characterized by the communication with the scoreboard and the use of a "find first flag" algorithm. The whole mechanism is rather simple, as we will describe it here (even though i seem to rant too much, thus : read slowly then reread more slowly). Note : Depending on its actual use and usefulness, the SRB mechanism may be removed from the F-CPU with minimal impact on the overall architecture, instruction set and application software. Some drivers and kernels may need the additional, manual register backup code. Other similar techniques can also be used instead.

How and when is the SRB used ? Well, it is used for what it does :
Flush the register set to memory and/or load a new context.
It could be used at any time, since it does not interfere with other hardware except the L/S unit.

It is mainly used for context switch (the SRB can be triggered by an interrupt and the rest is done automatically), to save a context when an interrupt is triggered, and to restore the registers after the IRQ routine has completed. In these cases, there are two threads : the "old" thread and the "new" thread. The flushing or reading of registers to/from memory by a load-many or store-many instruction is an exception, though.

The "new" thread is defined to start as soon as the SRB signal is triggered, and the SRB must save these registers before the new thread uses them as to ensure data coherency.

Not only does the SRB remove the need to manually save and restore registers, but it does it faster than software (while the application still runs) and adapts itself to the circumstances by reordering the backup sequence on the fly. It uses a few simple additional hardware, data from the scoreboard (the "register's value is being computed" flag), it steals unused clock cycles from the memory L/S unit to load and store the registers, it has a few flags, some pointer registers and some logic. To know how to use this, let's define some behaviour rules :

- We can't save a register as long as its value is being computed. The scoreboard tells us what register not to backup (yet). This status changes at every cycle, so knowing the state of the scoreboard quickly is very important.
- There's no need to save a register that has not been modified since the last backup. There is a "dirty" flag for this purpose, that is set whenever the register is written to.
- We have a special "not yet saved" flag that says that the physical register must be saved before it is ready for use by the new thread. In the same time, this flag blocks the scoreboard so that it can query an "express" request. This flag is loaded from the "dirty bit" when the SRB signal is detected, and the dirty bit is cleared for the new thread.
- When the new thread needs to use (read and write) a register that has not been saved yet, it instructs the SRB sequencer to modify the order and waits for the register to be free. The scoreboard, that is queried by the instruction decoding unit, "blocks" the instruction until the value is ready, and the "save in priority" flag is set until the data is ready.
- The SRB sequence is atomic, it can't be stopped unless there's a memory fault. A new SRB signal must wait for the previous SRB signal issued to be completely processed. Turning off the IRQs while

SRB is running avoids lost cycles (waiting for the previous SRB to complete, while the previous handler is being executed). If an exception occurs during the SRB sequence, good news : we had already begun to save the registers : -) We need to wait for the (old) sequence to complete, before triggering a "new" SRB and executing the handler.

Of course, this high number of flag bits can be condensed, using a Finite State Machine (this implementation detail is left to the designer). But the following algorithm doesn't need one : "for each cycle, write to memory the first register that 1) asks for express backup 2) is not yet saved (in decreasing priority), starting from register #1". The algorithm stops when no more register needs to be saved. When a context switch occurs, there are two memory accesses, one for saving the old register value and one for fetching the new thread's value. If one half of the thread's operations are loads or stores, this would use about onehundred cycles to save a context. With a single-issue pipeline and not much bandwidth, it can take about 200 cycles to perform a full context switch. SRB is bandwidth-hungry, but software backup would be too. At least, the SRB uses the whole available hardware, while a SW solution requires yet more bandwidth (because of the explicit backup code).

The SRB algorithm is a sequence that can be reordered with 63 register load and/or stores, starting from register #1 : we need a way to extract this sequence from a line of flags. A "find first" unit, similar (but simpler) than the binary tree used in the increment unit, can do this easily. At the input, it selects the "express requests" if any, or the normal request from the registers that need to be saved. The express flag is set by the scoreboard, and the "not yet saved" flags belong to the SRB mechanism. The output of the binary tree directly selects one register (out of 63) for reading and/or writing and resets the register's flags. Maybe a simple drawing speaks more : -)

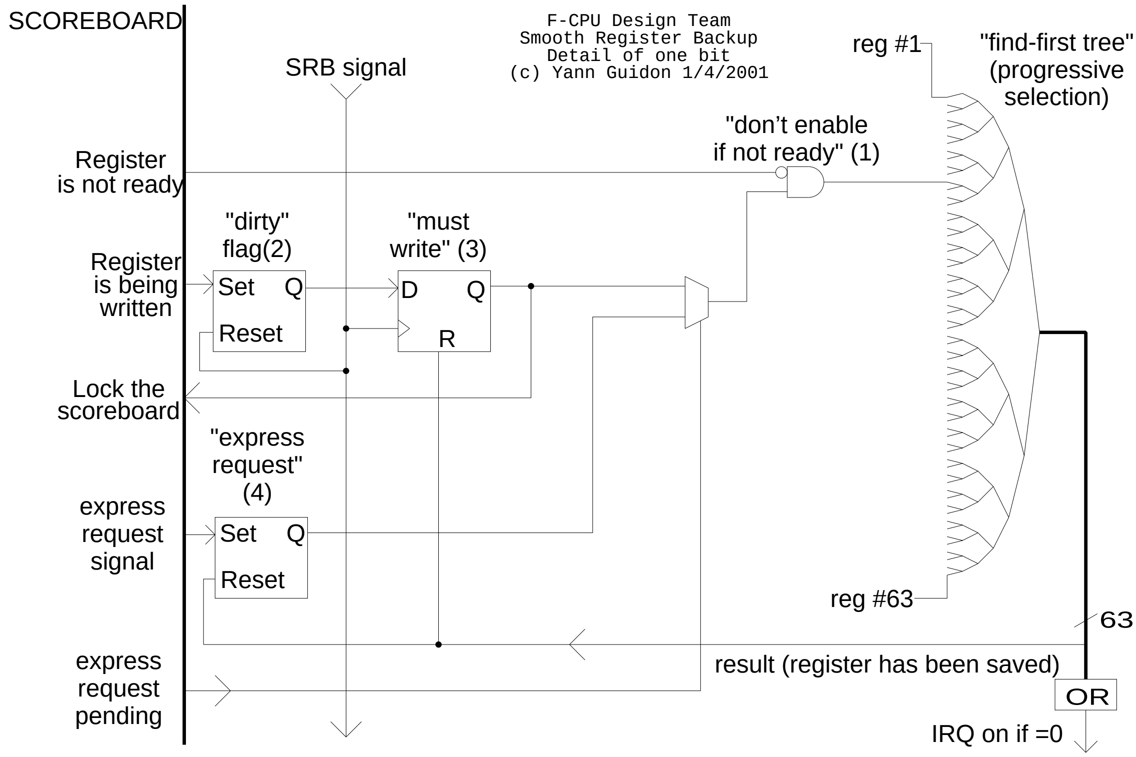

Figure 2.1: Detail of one bit of the SRB flags and decision mechanism

When no "express" request is made, the application has priority over the SRB for accessing the L/S unit. Otherwise, the "express" flag means that the instruction is blocked at ID stage and that no memory access is performed (unless a cache miss had just been resolved...). So, in any case, the SRB has never priority (which simplifies things). The SRB principle can be extended for multiple-issue processors without modification. A four-way superscalar F-CPU would be able to reuse this part a priori with no worries.

As noted before, there are only two tasks that the CPU considers : the "old" and the "new" task. No assumption is made about where the data is transferred, about the address of the context buffers, therefore there is no limitation in the number of tasks. The caches will perform their roles of keeping data with time and space locality close to the core, so multithreaded programs will run normally. But the user can't always specify the address of these context buffers. The SRB mechanism has two pointers : SRB_old and SRB_new, that are used during SRB operation. After context switches completion, the SRB_new pointer is copied into SRB_old so that during the next context switch, only the new task need to be provided to the CPU. It's up to the user to setup the desired list. This "new" pointer could also be stored in the "old"'s context buffer, as to perform round-robin operation automatically at each task switch IRQ. Cache control hardware will probably allow to "map" a certain memory area directly to the cache, so that no LRU operation will flush the data from cache. This is where important tasks such as the kernel and real-time tasks should store their context buffers for better performance. Furthermore, if automatic context switch is implemented, it would first prefetch the whole context buffers (old and new) into L1 cache before triggering the SRB mechanism. This prefetch would occur in background as to not penalize the foreground tasks.

Note that in the case where the FC0 core has 2 independent private memory ports, the register bank switch process can be eased if the banks are mapped to different memory regions. For example, if the register bank is flushed to one port, the new register bank benefits from being read from the second port. There will be much less bus turnaround cycles and the switch latency will decrease.

Chapter 3

The scheduler

Managing several superpipelined units which can issue their result at the same time looks tricky at first. The following behavioural rules will help understand what to do and when :

- The Xbar "gates" of the 2 write ports must be commanded during every cycle, so the 2 read ports of the register set have the correct data coming from the correct unit.
- One instruction can not be issued if more than two write ports are used during the cycle when the instruction will complete.
- If the instruction can be issued, it must use a "free" write port.

Let's remember too that the scoreboard rules apply. More specifically, it is not possible to issue an instruction if the operands are not ready, in the register set or on the Xbar (during a register bypass cycle, for back-to-back dependent instruction pairs). The scheduler must also recognize this situation.

Two solutions are possible and were investigated :

o The first possibility is to associate a Finite State Machine to every register. It is a countdown machine that triggers the apropriate signals as it elapses.

The advantage is that this is completely independent from the actual number of operations that can be issued during every clock cycle, it is preferred for this reason.

Unfortunately, it creates very large internal buses and the detection of the hazards is too slow, particularly when the Register Bank's write bus must be allocated.

o The second solution is less scalable with the number of instructions issued per cycle, but is a simple and deterministic algorithm that consumes much less ressources when only one or two instructions are issued at a time. It is a FIFO that is as deep as the pipeline, and each line contains the number of the register which will be written to the register set. Since there are two write ports, the FIFO contains 2 x 6-bits fields. If the "slot" is empty, two additional bit flags are used to indicate this state. The empty lines are zeroed (the bits are cleared when they shift down the FIFO) but ORing the bits takes too much time (yes, a 6-bit OR takes more time and room than a 7th bit per field).

In fact, the scoreboard uses the first representation : the 63 bits that represent if a register is being used are spread along the register set, they are cleared when the corresponding line gets written. This uses long wires and large buses, but it is rather simple. On the other hand, the second mode of representation for the scoreboard (a lot of registers containing the numbers of the currently used registers) takes too much ressources and it doesn't scale well when more instructions are decoded at the same time.

The scheduling and scoreboarding informations for the FC0 can use any suitable representation for the informations, but they can be both used in parallel (as it is the case). Having both representation helps get the wanted information with the least latency. If a bit vector is needed, it will be read in the scoreboard, and if a number is required, it is read from the scheduler's FIFO.

Now, there is a very important characteristic associated to the scheduling FIFO : the "slot" can be allocated at several levels, because the instructions can have different latencies. This means that a multiply instruction will "reserve" (if it is free) a "slot" in the FIFO at the 6th level, while an addition will reserve a slot at the second level.

The instruction decoder must therefore provide the scheduler with a precise information about the latency of the instruction it will issue. This information is stored in a Lookup Table that takes the opcode and the flag fields as inputs, it outputs the number of cycles of latency for the instruction. This LUT is hardwired but if the implementation supports 128+ bit registers, a certain part will be reconfigurable on-the-fly to support the programmable size field (see chapter 2.5 about the variable sizes).

When the instruction set is designed, the instructions must be garanteed to be fixed-latency so the LUT can be as compact and fast as possible. This puts some pressure for the scheduling of two types of instructions : Load/Store and division. The Get/Put instructions are also "undetermined-latency instructions" but they block (stall) the pipeline.

The Integer Division Unit of the FC0 (a first "cheap" implementation) is a slow shift-substract machine like it is found on older microprocessor : the latency is proportional to the number of bits to divide. It is not pipelined and the throughput is also proportional to this data width. The scheduling is therefore simplified because it is not pipelined : the FIFO doesn't have to contain 64 slots for the case where a 64-bit number is divided ; a simple downcounter is enough. Furthermore, this latency is either 8, 16, 32 or 64 cycles, and 8-cycles is more than the latency of the multiplier : the counter does not interfer with the FIFO, it sits on top of it and it is initialised very easily with the size flag of the instruction word.

The case of the Load/Store instructions is more difficult because it is not deterministic. The situation is simple when the data is already contained in the L/SU buffer, otherwise it's a real stinking can of worms.

When the data is contained in the L/S buffer, the latency is deterministic : it takes one cycle in the buffer, one cycle in the byte shuffler (that selects and orders the bytes in a word), one cycle in the Xbar and one cycle in the Register Set. This is the situation that *must* be privileged whenever possible. This is promoted with the early issue of the address (the pointer must be known as soon as possible so the loaded data can be fetched from memory in advance) and the wise use of the stream and cache hint bits.

When the data is not present in the L/S buffer, the scheduler must prepare for an asynchronous event and there is no garantee that a free slot will be available. On average, it is probable that the 2 write ports of the register set are used 70the data is actually available. There is no such problem, though, when the loaded data is needed during the cycle following the load instruction : the pipeline will stall and leave some room for the L/S U to feed the Xbar with the desired data.

drawing : I must insert a diagram of the scheduler FIFO and the scoreboard

Chapter 4

The memory units (Fetcher and L/SU)

to be written soon

drawing : I must insert a diagram of the register LUT, the decoder, the Fetcher and the LSU