Freedom CPU Project

F-CPU Design Team
Request For Comment
created July 8, 1999 by Whygee
July 10 : added some more.
11th: adapted for converstion to PDF with HTMLDOC.
8/2: added yet some more.
8/8: added yet some more.

F1 CORE DRAFT PROPOSAL REV. 1.1

What, why :

This document is the first study and working basis for the third generation of F-CPU (F1) architecture. This document explains the architectural and technical backgrounds that led to the current state of the F1 core, as to reduce the amount of basic discussions on the mailing list and introduce the newcomers to the most recent concepts that have been discussed. The draft will (hopefully) evolve rapidly and incorporate more advanced discussions and techniques. This is not a definitive draft, it is open to any modification that the mailing list agrees to make. You are very encouraged to contribute to the discussion, because nobody will do it for you.

This document is independent from the **F-CPU Architecture Guide** which describes the instruction set and the programming rules from the high level point of view. This draft describes the way the instructions are executed, rather independently from the instruction formats, fields or conventions, because only the instruction decoder is concerned.

Since the F-CPU project is rather independent on how the instructions are executed, there is no garantee that the F1 CPU will use this core. It has therefore been agreed that the core described in this draft is called "FC0".

Please send comments to the F-CPU mailing list.

Summary :

A bit of F-CPU history :

The first generation was a "memory to memory" (*M2M*) architecture that disapeared with the original F-CPU team members. 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 ;-)

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.

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 Hennessy, the MMIX, the MISC CPUs, and other similar, simple projects. 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 specific choices and particular characteristics. This is what we will discuss here.

The main characteristics :

The core described here can be thought as a crossover between a R2000 chip 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, F0 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.

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 :-)

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 even-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 F1 microarchitecture was not clearly 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.

Register 0 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 F1 by "CMP R3,R1,R2" because CMP does 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).

The F-CPU has 64 registers, while 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 has been 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 has been based on schoolbook algorithm examples,
- memory was less constrianing than today (even though magnetic cores was slow) and memory to memory instructions were common,
- chips had less room than today (tens of thousands vs. tens of million),
- 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 register 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 registers in IA64 ???
It is clear anyway that *FAST* context switch is an issue for a lot of obvious reasons. Several techniques 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 an issue.
In the discussions of the mailing lists, it appeared that:

- most of the time is actually spent in the 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 technique which replaces the software at interleaving the backup code with the computation code.

A code like this:

```
IRQ_HANDLER:
   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
....
```

can be a common code that would be the beginning of an IRQ handler.
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:

```
IRQ_HANDLER:
   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. This technique will be described deeply later.

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 (instructions per cycle)x(pipeline depth)x3 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.

The F-CPU is a variable-size processor. This is a controversial side of the project that has been finally accepted recently. 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 on the fly the size of the data we handle. 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, KNI, AlphaPC, PPC or SPARC 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 directly proportional to the width of the data that the CPU can handle.

The next question is : how wide ?
Because fixed sizes 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 width 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 ?
We have to consider the whole process of programming, coding, making processors and enhancing them. The easiest solution is to use a lookup table, which interprets the 2 bits of the size flag defined in the **F-CPU Architecture Guide**. The flags are by default interpreted like this:

FLAGS	SIZE 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 exist in the instruction itself. The lookup table will probably 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.

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 compiler could simply use the default four sizes but more sophisticated compilers will be needed to benefit from the performance of the later chips. 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, and the old "RISC" principle of letting the software solve the problems is used once again. I hope that this side of the project will be soon included in the Architecture Guide, and that coding examples will be given, but 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.

The F-CPU has generalized registers, meaning 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 one of the set would not be used (the FP set would be unused for example during programs like a mailer or a bitmap graphics edition, while a lot of FP is needed during ray-tracing or 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 two register read ports are needed, thus there is no register set problem at the beginning.
Later chips, with more instructions issued per cycle, will probably use a technique dear to the team : 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.

The F-CPU has special registers that 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 modern operating system.

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 each 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 pop instructions cause multiple register uses in a single cycle in a superscalar processor, which requires special material.
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, and 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:
pop:

```
load.64 R2,[R3]+8
```

push:

```
store.64 R2,[R3]-8
```

Since the addition and the memory fetch are performed in the same time, the updated pointer is available *after* the instruction.
The "Smooth Register Backup" hardware in place can be used by instructions, but none has been agreed upon yet. 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.

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).
But, as explained later, reading a register is rather "slow" in the F1 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 latches" 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.

The problem of some other properties or status flags, mainly of arithmetic order, is not yet completely solved. It is sure though that this must go through the use of the general register set, or we'll experience troubles saving them.

The F-CPU 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 an 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 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. 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 companded 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 F-CPU can be considered as superpipelined. But it is only one aspect of the project and today, several processors are also superpipelined.

The FC0 core implements an *out of order completion* pipeline to get more performance from a single-issue pipeline. This is NOT a superscalar or out-of-order *execution* scheme but the "adaptation" of a simple pipelined CPU.
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. 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 two 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 precedent cycle.
For this last reason, the register set has two write buses. 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 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") but by the number of register ports.

The F-CPU uses a scoreboard because it is the simplest way to handle the out-of-order nature of the processor. 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 checked by the scoreboard. 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.

The F-CPU 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 CPU.
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 4 "vertical" buses (see figure 2), and each four 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).

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 :

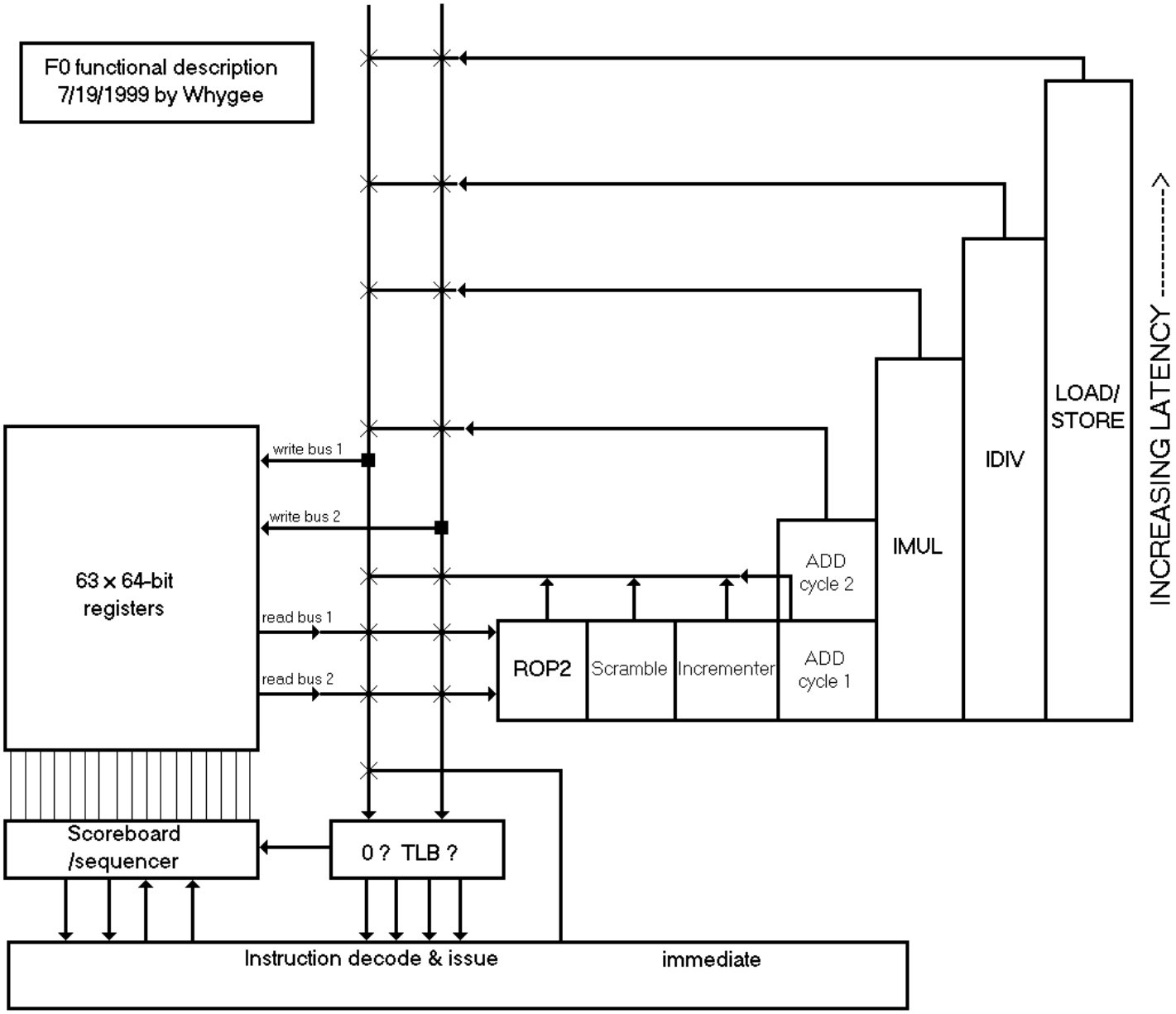

figure 1 : the pipeline is folded around the Xbar.

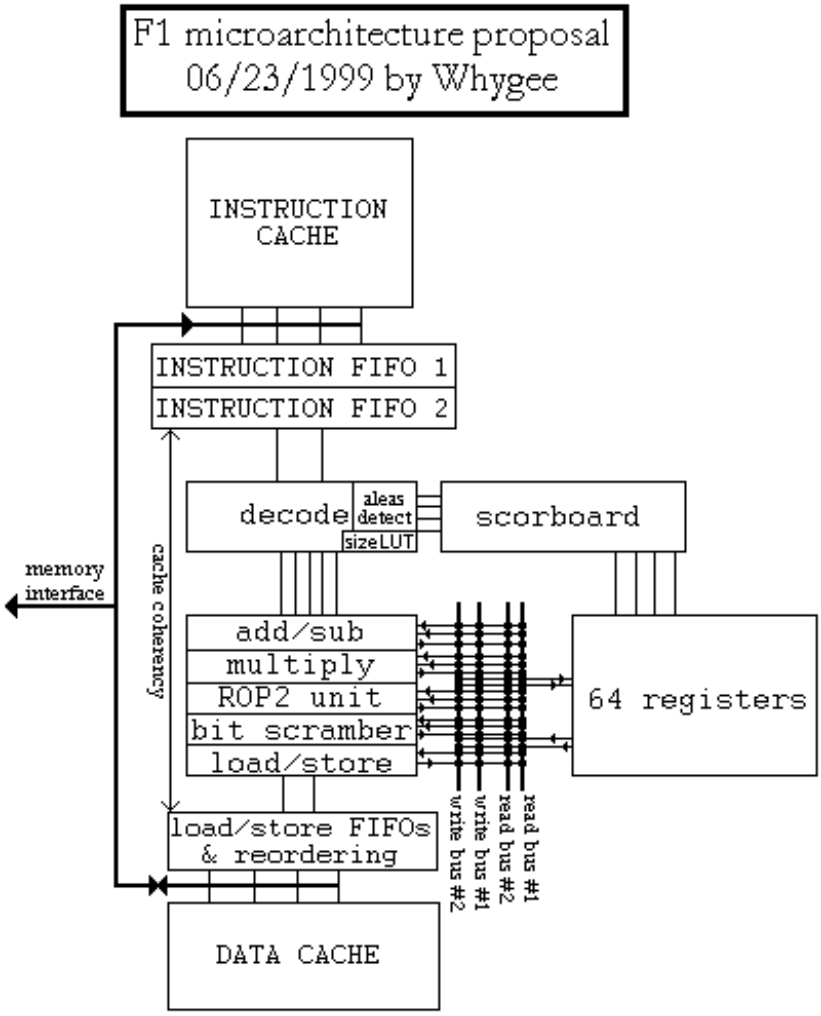

figure 2 : The first F-CPU chip proposal.

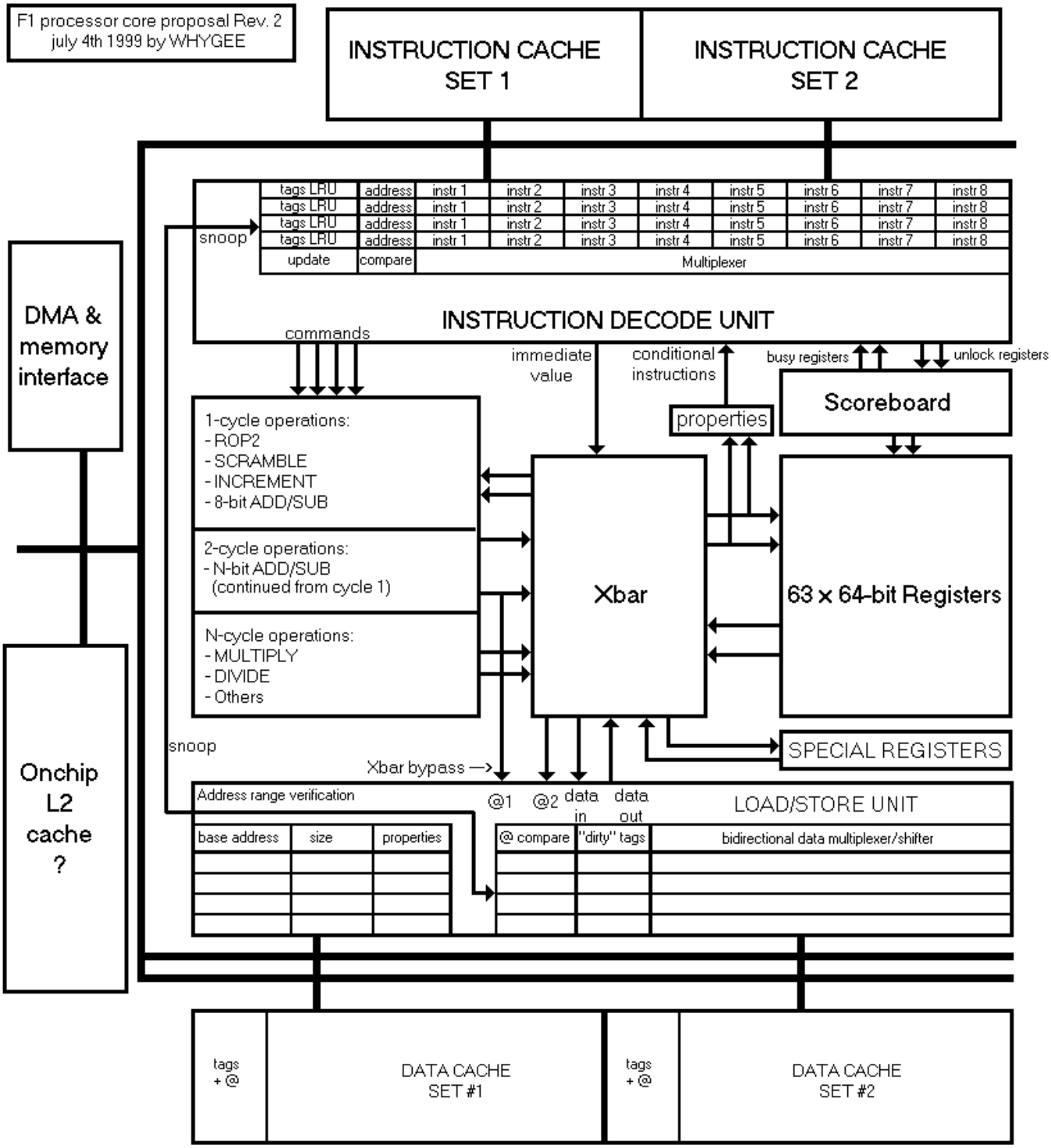

figure 3 : A more precise, first-attempt F-CPU description.

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, 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, 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.

The "ROP2" unit :

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 (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 (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 (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 can include directly 4 function bits in the opcode, but if we need room and better architectures, we can save some opcodes by using "condensed" codes. We select 8 2-operands operations, 1 1-operand operation (NOT) and 1 0-operand operation (SET to all ones). The decoder can thus avoid to read unnecessary source registers.

opcode	real code	Function
000	0001	A AND B
001	0010	A AND /B
010	0110	A XOR B
111	0111	A OR B
100	1000	A NOR B
101	1001	A XNOR B
110	1101	A OR /B
111	1110	A NAND B

So if we run out of opcodes, we can use this table to translate the opcode field into the real computation code without really lengthening the critical datapath.

The necessary hardware for computing this function is rather low, maybe twenty (?) transistors per bit :

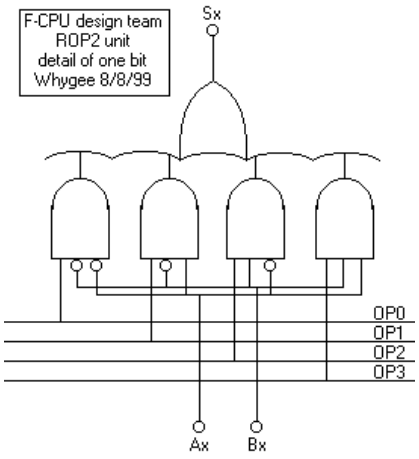

figure 4 : Detail of the ROP2 unit.

There are probably a few 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.

The "bit scrambling" unit :

The aim is to have a one-cycle shifting unit that can do other things as well. As opposed to the ROP2 unit, it does not change the value of the input data bits but it changes the positions of the bits. Therefore, shifting and rotating are only examples of the intended purposes of this unit : bit fileds extraction and insertion, as well as bit 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 anyway.

In this unit, i have not yet addressed the problem of the SIMD data.

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 1. then, you turn this 1 into 0. This is 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.

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. We can also add a bit reverser at the input, as to find the MSB too. Six instructions.

let's go further : let's put a multiplexer at the end of the incrementer, wich is commanded 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.

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**, 1**1** > 1**0**...

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, i presume there will be some problems to fit it in the pipeline's stage depth...

In this unit, i have not yet addressed the problem of the SIMD data.

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 would compute a 8-bit add/sub in one cycle. Therefore, SIMD with 8-bit data would be fast (1 cycle instead of 2). For these reasons, it would be difficult to use standard library pre-synthetized elements because of the variable-depth and SIMD nature of this unit.

The 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.

The divide unit :

Same as the multiplier.

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. This will be the subject of a next part.

Other units :

The floating point numbers have not been discussed, because we better have something work in the integer domain first, we'll add FP hardware and instructions later. The case of the math exceptions will be probably managed with the same kind of mechanism as the "zero" property flag, so no error will break the execution pipeline flow.

One "cheap" way to avoid the use of floating point numbers is by using the logarithmic number base. Recent works succeeded in making a 32-bit logarithmic adder with descent speed and die space use. Any other operation (SQRT, SQR, multiply, divide...) can be performed by existing hardware (maybe slightly modified for the MSB). The conversion between integers and log numbers will be a rather heavy software task, as long as no hardware exists.

When FP hardware will be available, only add/sub and multiply units will be implemented. 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. In some cases, it can be faster to compute a division in floating point than in integer with the Newton-Rapshon method.

Some possible enhancements of the overal architecture include more read ports from the register set as to provide ROP3 operations (3-input logic), MAC (multiply two numbers and add the result to a third operand), bitfield extraction/insertion from a bitstream, address bit-reversing, and store to memory with a register as index increment.