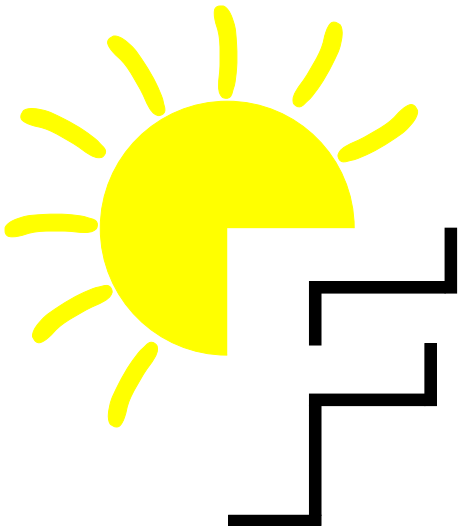
Institut Supérieur
d'Informatique de
Modélisation et de
leurs Applications

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

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

3rd year project report

Making an Instruction Set Simulator for F-CPU



Pierre Tardy
François Vieville
ISIMA teacher: WODEY Pierre
School-year 2003-2004

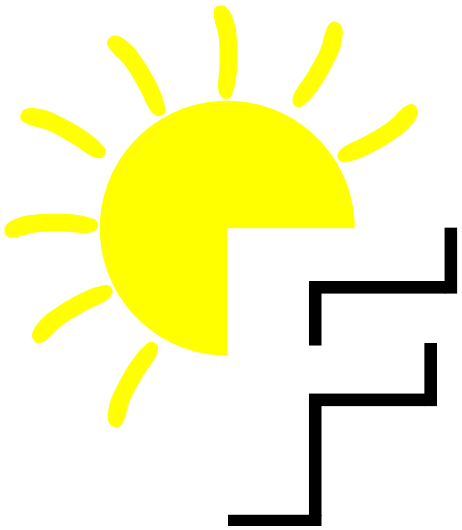
Institut Supérieur
d'Informatique de
Modélisation et de
leurs Applications

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

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

3rd year project report

Making an Instruction Set Simulator for F-CPU



Pierre Tardy
François Vieville
ISIMA teacher: WODEY Pierre
School-year 2003-2004

Résumé

Le projet F-CPU a pour but de développer un microprocesseur RISC, SIMD, superpipeline 64 bits libre. Les simulateurs de jeu d'instructions jouent un rôle central dans la cosimulation, qui a permis une nette amélioration en terme de vitesse, de coût et de qualité de développement. Nous avons donc souhaité réaliser un simulateur de jeu d'instructions pour le microprocesseur F-CPU, pour permettre à la communauté de mettre en oeuvre cette méthodologie, en vue de l'intégration du F-CPU dans un système.
L'étude détaille l'amélioration d'un simulateur « untimed fonctional » existant, en un modèle transactionnel, lui-même facilement évolutif vers un modèle « bus cycle accurate », ainsi que l'utilisation de ce simulateur dans un modèle SystemC. Ensuite est décrite une utilisation de modèle dans le cadre de l'évaluation de performance d'un logiciel simple.
Mots clés : F-CPU, simulateur de jeu d'instruction, cosimulation, SystemC.

Abstract

The F-CPU project aims at designing a Free, SIMD, superpipelined 64 bit RISC microprocessor. Instruction Set Simulators are the central part of cosimulation, which has greatly improved the design speed, cost, and quality. Thus, we've tried to design an instruction set simulator for the F-CPU microprocessor, to have its developer community use the cosimulation methodology for the system level integration of the chip.
This study reports the improvement of an existing untimed-functional simulator into a transactional level model. This model being itself easily improvable to a bus-cycle-accurate model of the F-CPU. We have then integrated this simulator into a SystemC model, and used it to profile and evaluate the performances of a simple piece of software.
Keywords: F-CPU, instruction set simulator, cosimulation, SystemC.

Thanks

We want to thanks all the persons who gave us help in this project. Michael Riepe for his experimented advices, Yann Guidon for his excellent presentation at ISIMA, and all the guys of the mailing list who feed the prefetcher troll.

A great thanks, of course, for our ISIMA teacher, Pierre Wodey, who help us keeping a critical point af view on our weird ideas.

Contents

List of Figures

List of Tables

Introduction

Nowadays, the hardware companies are building more and more complex architectures. The needs for different abstraction levels for designing is becoming obvious to keep a good sight on the System on Chip projects. Instruction Set Simulator is one of the tools needed to build System on Chip where a CPU Core is present. It allows software development on not yet available chips.

The F-CPU project is an ambitious project that consists in developing a modern, fully functional, general-purpose CPU core, with the rules of free software. Participating to such a project is very interesting for us, if you consider that the entire project is available, there is lot of things to learn here. The feeling to do a useful work for F-CPU project is also very motivating.

Our ISIMA third-year project was to make and improve an Instruction Set Simulator for F-CPU. We will describe in the report what is the F-CPU project, what we mean by ISS, what has been realized, and finally give clues about where our work is useful.

Chapter I

Context

1 Description of the F-CPU project, Project Basis

This section is mainly taken from the F-CPU manual. [1]

[...]

The F-CPU architecture defines a SIMD, super pipelined, 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 license (GPL and GFDL). It is meant to be a totally unencumbered design targeted at the widest range of technologies as possible.

The F-CPU project is also formed by many people, discussing on mailing lists about the organizational 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 could 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 utopist project was only a dream at the beginning but after two group splits and many 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 (approximate 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 resources
* 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 micro architectures, 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 "flavors" 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.

[...]

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 "Crystal 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 License 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 behaviors adopted in the group. Some strong disagreements have and will appear during the discussions, but whether the subject corresponds 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.

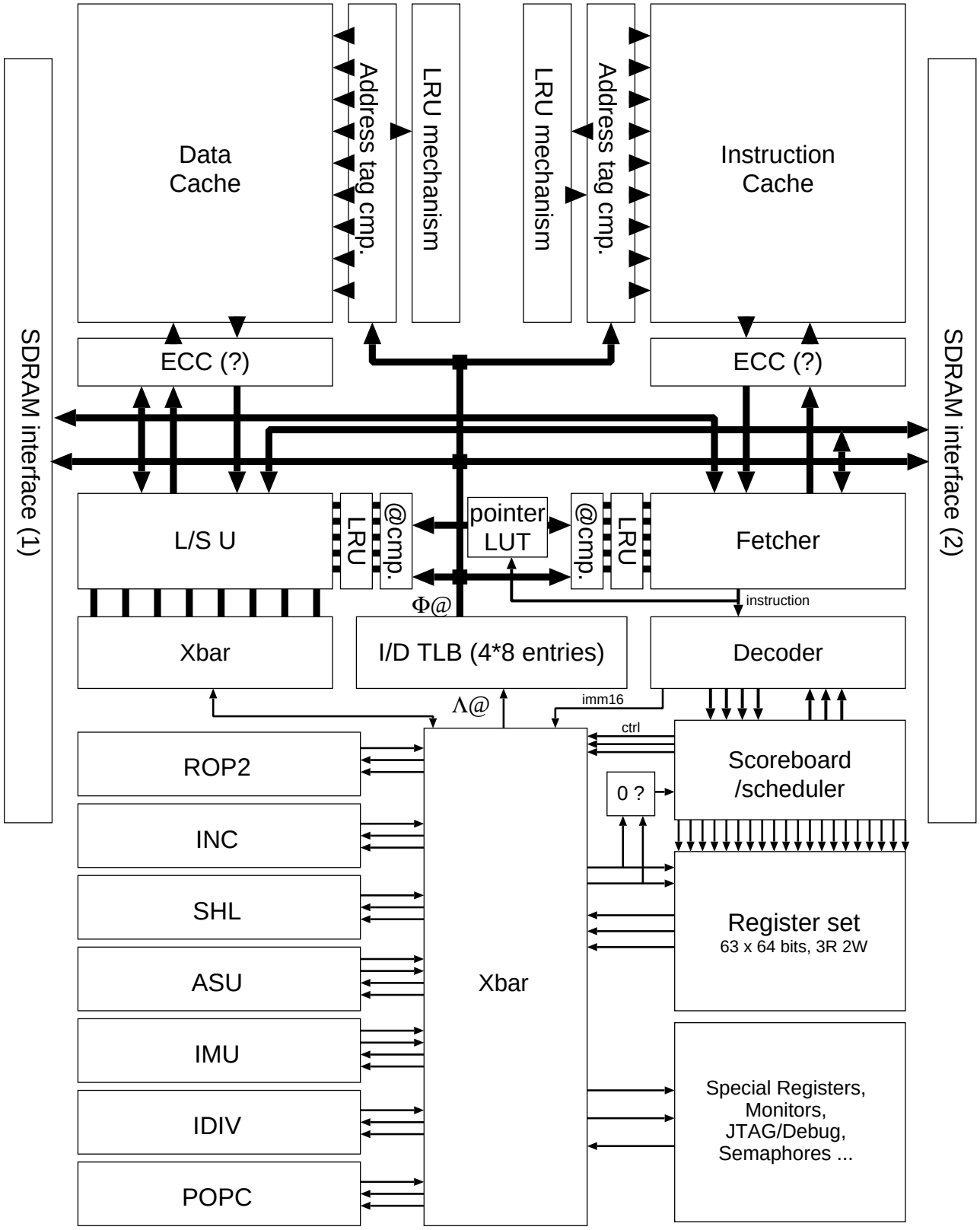


Figure I.1: the FC0 Architecture

2 FC0 Architecture

The FC0 is the first version of F-CPU. See figure 1 for a detailed synoptic schema. Its main characteristics are:

- Super pipelined processor with a critical data path of 6 logic gates. This means that all we can do in one cycle must be done with at most 6 logic gates (AND, OR, NOT, etc.). This is few, but this can made the processor very fast.
- 64 bits by default, but it can be extended to any multiple of 64.
- SIMD. That means that several data can be processed at the same time. Practically, register are cut into several chunks, where operation are done. For example, a 64-bit register contains 4×16 bits chunks.

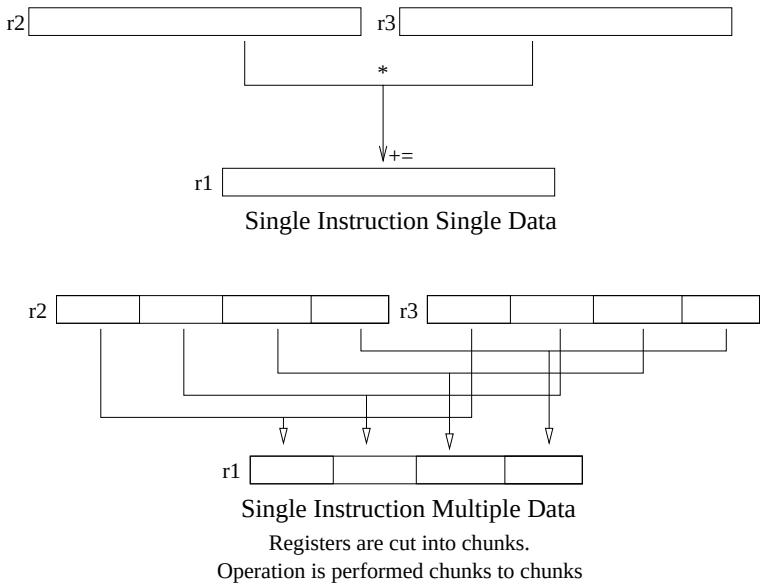


Figure I.2: SIMD in FC0

- There are special registers that are used to configure the state and the behavior of the f-cpu.
- Support for logical vs. physical addresses. The correspondence is done through the TLB, which associate each logical address to a physical address. This is useful for any modern OS that do memory mapping and protection, like Linux.
- The cache is divided into 2 cache types, the data cache and the instruction cache.
- There are L0 and L1 cache for each instruction and data, burned directly into the CPU.
- The cross bar (aka Xbar) is a complex bus architecture that links Each module to each other.
- each module is super-pipelined and commanded by the scoreboard/scheduler with out of order algorithms. This means that the result of one operation may be ready before the one of the precedent instruction. Complex mechanisms are needed to avoid data decency conflicts.

3 SystemC

SystemC is A C++ framework that helps designing complex hardware-based systems.

3.1 A new language for modelisation

Hardware coder already have VHDL and Verilog to describe their Hardware architecture, but nowadays, the systems are becoming more and more complex, pushing the developers to design and verify at higher levels of abstraction. There is also a problem with hardware and software co-design. In the first steps of design, we know that we want a DVD player; there will be an mpeg2 decoder, a subtitles rasteriser, etc. Some will be coded in hardware other in software, all will work together. However, the Software development has to be done at early times. This is called Software/Hardware co-design. We need a new language and this language will be based on C++ because:

- Verilog and VHDL are not very good at high-level design.
- Developer's existing knowledge of C/C++ can be leveraged.
- Software developments are mainly done in C/C++, integration will be easy.
- The C++ language has many tools and supports (compilers, debuggers, books, etc.)

3.2 Levels of abstraction

Basically, we have:

- UTF: UnTimed Functional. That describes a process in its behavior only. Basically, it is a C function that will get a MPEG file in input, ad that will output a YUV video stream.

  No need of systemC here.

- TLM: Transaction Level Model. We describe here the transactions needed to make our MPEG decoder communicate with other modules. Will the MPEG file transmitted byte-after-byte or frame-after-frame? What control signals do we need?

  The systemC may help here. TLM module can be connected with other TLM module to verify the global architecture.

- BCA: Bus cycle accurate. Here we define exactly what append on the bus when there is a transaction. The intern behavior of the module is still coded in behavioral, in order to be as fast as possible during a simulation.

  SystemC and co simulation are a precious help here. A module coded in BCA can be connected to other BCA modules, CA modules, RTL modules and even to real HW chips.

- CA: Cycle Accurate. In this model, we can know exactly how much time (how much clock cycles) is needed for the real chip to compute an operation.

  SystemC and VHDL/Verilog are used here; SystemC is known to be faster.

- RTL: Register Transfer Level. Here, all the details of the micro architecture are described, this model can be automatically compiled to a real chip.

  SystemC doesn't bring any real advantage here; The RTL is in the greatest part VHDL/Verilog only.

3.3 SystemC evolution

SystemC is currently developed by the Open SystemC Initiative (OSCI), a non-profit organization with many members from the Hardware companies.

- SystemC 1.0 implements C++ class needed to describe RTL, CA and BCA models. No concept is added compared to VHDL/Verilog.

- SystemC 2.0 as more general system level modeling capabilities with channels, interfaces, and events. This is the current version of SystemC, the one that is useful.
- SystemC 3.0 will focus on software and scheduler modeling.

Chapter II

Instruction Set Simulators (ISS), several approaches.

1 ISS definition

An instruction set simulator, according to its widest definition, is a software tool, which runs on a host machine, generally a workstation, to simulate the execution of a program on a target machine, allowing the user to examine more or less precisely the internal state of the target machine during the execution of each instruction. Typically, an ISS decodes an instruction-flow designed for the target CPU, and converts it into another instruction-flow which has the equivalent semantics inside the simulator.
ISS can perform both architectural (functional) or micro-architectural simulation. Architectural simulation, also called functional simulation, refers to simulation of the processor instruction set, whereas micro-architectural simulation refers to components inside the processor such as pipelines, caches, functional units, etc.

2 Why using an ISS ?

public application of ISS's is the emulation, which is in fact the simulation of a whole computing system on another. For example such emulators are available to execute i386/Windows applications on a Macintosh.
However, what is the most interesting for us in the F-CPU project is the integration of the ISS into a hardware/software simulation platform. ISS's are indeed today the central part of co simulation, which has brought a large improvement in the methodology of microelectronic design, and has allowed to design Systems on chip.
The co simulation allows for example to perform tests on the base software (compiler and OS) before a first version of the hardware is available, and thus spare lots of time and money.
ISS are also helpful in the hardware design flow. For example, they can be used to tune some performance/cost issues such as cache or FIFO sizes. They can be also used to show up the strongness (or weakness) of some design choices (cf. Chapter XX).
In short, ISS and co simulation has brought a cheap, quick and efficient new test and validation methodology, and though these can't fully take the place of classical hardware-based verification, they can be performed up-stream in the design-flow and thus drastically reduce the expensive use of tools such as hardware debuggers or hardware simulators.
Obviously, base-software verification and hardware-design performance tuning don't require the same type of ISS.

3 Different ISS types

In this section, we will first define the different simulation levels, then we will determine how to evaluate the quality of an ISS and eventually we will compare different techniques to build an ISS.

3.1 Simulation levels

There are several well-known levels to define the accuracy of a simulator. The higher the level, the less accurate the simulator. Here are these levels from the up most to the downmost.

- Untimed Functional : at this level, the execution atom is the assembly instruction. The simulator executes an instruction at a time even though in the real processor, instructions may be pipelined, and it systematically accesses the memory, even though the real processor has a cache. This level is well suited to help design and debug software
- Transactional Level: at this level, the simulator performs the same memory operations as the real processor would do, and these operations are handled by events. This simulation level is often needed by system level designers to perform system analysis.
- Bus Cycle Accurate: at this level, the memory accesses are signal based. The simulator is thus the exact copy of the real processor if seen as a black box. This level is used to perform optimizations on the micro-architecture.
- Cycle Accurate or Register Transfer Level: here the simulator equals the real processor. This level is used for synthesis and micro-architecture simulation.

3.2 ISS quality metrics

Though the most obvious evaluation criterion is its simulation-speed, other qualities are required to make a good ISS, for example:

- Compilation speed: this evaluates the time needed by the simulator to bring an application into a suitable state for the simulation. This criterion is only relevant in the case of compilation-based simulation, where the target-machine input code is translated into host instructions or into a series of virtual-machine instruction. This last technique is often used to build highly retargetable ISS.
- Traceability: evaluates both the variety of information available to the user about the state of the target machine during simulation, and how easy this information can be retrieved. This quality is in close relationship with the targeted accuracy of the simulator.
- Interoperability: capacity for the simulator to interact with other tools such as debuggers, execution profilers, simulation cores (e.g. SystemC), or CAD tools.
- Retargetability: how easy can an ISS be changed to simulate another target machine. For example, there exist ISS?s, which can simulate any kind of RISC processor, by only writing an instruction-set configuration file.

3.3 ISS tasks

In this section, we define what an ISS has to perform.

Register Mapping

The ISS has to host the registers of the target CPU on the machine it is running on.
The way the registers are hosted in the ISS memory is called register mapping. Most of the time, this is done with a simple array of words in the data segment, but in some very high-speed simulators, target registers can be mapped directly to host-CPU registers.

Memory Mapping

The memory around the target CPU can be simulated inside the ISS process itself, for example in a huge array stored in its heap. In such a case, the ISS is said 'standalone'. This very simple solution offers poor interoperability and traceability, but these are quite enough in some cases. For better interoperability and traceability, this memory should be externalized: the memory is then accessed through a read/write interface, so that the memory access can be logged. Such externalization is often performed by wrapping a standalone ISS into a SystemC model.

Instruction translation

Obviously, the primary task of the ISS is to decode an input instruction-flow designed for the target CPU, and convert it into another instruction-flow, which has the equivalent semantics inside the simulator.
The translation of each instruction should be composed of two distinguishable steps:

- Decode: the binary instruction is translated into a target-machine assembly instruction.
- Execute: the target-machine assembly instruction is mapped to one or more host-machine instruction, which have the same semantics.

According to the type of ISS, this translation is made at compile time or at load time for compilation based simulation, or at execution time for interpretation based simulation.

3.4 Compilation vs. Interpretation

Compilation based simulation

In this kind of simulation, the input instruction-flow is translated either at compile time, as in the case of static compiled simulation, or at load time, as in the case of dynamic compiled simulation.
An example of static compiled simulation is shown by Figure II.1. The input file is translated according to a translation table as shown by Table II.1. In this example, the ISS emits C code, but several ISS directly emit host machine assembly.

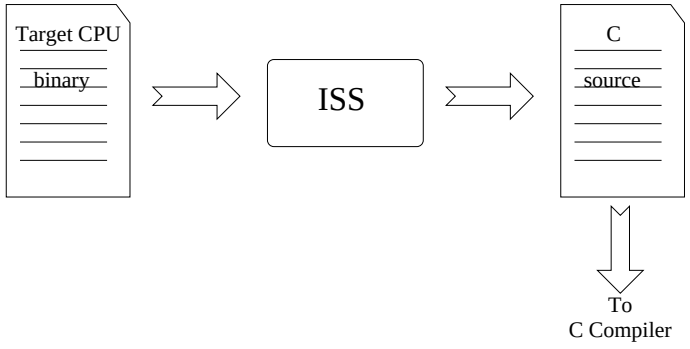


Figure II.1: Static Compiled Simulation

Target assembly instruction	C translation
`add r1,r2,r3 ;`	`r[3] = r[1] + r[2] ;`
`load r1,r2 ;`	`r[2] = memoryread(r[1]) ;`

Table II.1: Example of Translation Table

This kind of simulation offers very high speed but poor simulation accuracy. Indeed, it doesn't test the instruction fetching and can't handle self-modifying code, since the output instruction-flow is fixed before execution time.
Thus, this simulation strategy is well suitable for testing and debugging software at functional level, but insufficient for simulation of system architecture.

Interpretation based simulation

Interpretation based simulators builds in memory a data structure representing the state of the target processor, according to the register and memory mapping, and it maintains it up-to-date all through the execution of the instruction-flow. Thus, it allows a much better accuracy than the compiled approach, but this has a cost: a much lower simulation-speed.
An interpretative ISS consists of an infinite loop, which executes the sequence of actions:

- Fetch: reads an instruction from memory
- the opcode field, the flags and the registers from the instruction, in C, this is usually done with a custom `struct` data type as shown by Figure II.2
- dispatch : jumps to the appropriate code to handle the instruction
- execute: updates the processor state mapped in memory according to the semantics of the instruction

An example of such a loop is shown by Figure II.3.

```
struct instruction_s
  {
    unsigned int opcode : 8 ;
    unsigned int flags  : 6 ;
    union
     {
      struct
       {
        unsigned int r1 : 6 ;
        unsigned int r2 : 6 ;
        unsigned int r3 : 6 ;
       } 3regs ;
      struct
       {
        unsigned int r  : 6 ;
        int imm12       : 12 ;
       } 1reg1imm12 ;
       ...
     } reg ;
  } ;
```

Figure II.2: Decoding Structure

```
while(1)
  {
    instr = fetch(PC) ; /* fetch and decode */
    PC += 4 ;
    switch(instr.opcode) /* dispatch */
      {
      case OPCODE_ADD:
      /* execute the add instruction */
        add(instr.reg.3regs.r1, instr.reg.3regs.r2, instr.reg.3regs.r3) ;
        break ;
      case OPCODE_JMP:
      /* execute the jump instruction */
        jmp(instr.reg1) ;
        break ;
      ...
      }
  }
```

Figure II.3: Main Loop of an Interpretative ISS

Chapter III

Study of what exists

As said before, the f-cpu project is free, and the model is nearer form the Bazaar than form the Cathedral. The project is hosted at f-cpu.seul.org. Each project contributor has an account of seul.org, and publishes his files in his directory on fcpu. However, the most important files are uploaded to `http://fcpu.seul.org/new/`. It may also have been some contributors that have no account in seul.org, and that host their projects in their own web site. Finding all that have been written on f-cpu is currently not an easy task.

Moreover, the bazaar model implies that there are many born-dead subprojects.

Index of /new

Name	Last modified	Size
Parent Directory	25-Jun-2002 21:46	-
micropipelines.pdf	07-Jan-2001 15:06	2.2M
SRB.OBJ	07-Jan-2001 20:53	28k
SRB.EPS	07-Jan-2001 20:53	31k
ex5.tar.gz	14-Feb-2001 12:42	3k
GNL.6.tar.gz	14-Feb-2001 12:45	22k
gdups.c	20-Mar-2001 05:53	27k
update.zip	09-Jul-2001 23:14	2.9M
snapshot_yg_8_23_200..>	22-Aug-2001 19:26	93k
snapshot_yg_9-1-2001..>	01-Sep-2001 13:55	120k
snapshot_yg_9-3-2001..>	03-Sep-2001 16:27	127k
fcpu-mr-20010905.tar.gz	04-Sep-2001 20:07	22k
snapshot_yg_9-7-2001..>	07-Sep-2001 18:19	139k
mr-as-0.0.tar.gz	07-Sep-2001 18:47	109k
snapshot_yg_13_9_200..>	12-Sep-2001 18:56	191k
snapshot_yg_9_20_200..>	06-Oct-2001 22:57	193k
fcpu-shl-mr-20010927..>	06-Oct-2001 22:58	11k
FF.gif	25-Nov-2001 10:16	30k
fcpu-mr-20011127.tar.gz	26-Nov-2001 18:24	27k
ROP2-YG-2001201.tgz	01-Dec-2001 00:52	39k
lm02.tgz	09-Dec-2001 20:38	50k
gifs.zip	11-Dec-2001 11:23	30k
EPS.TGZ	25-Dec-2001 00:15	59k
snapshot_yg_12_25_20..>	25-Dec-2001 13:01	283k
stable_yg_12_31_2001..>	31-Dec-2001 04:45	294k
manuel_arith.pdf	29-Jan-2002 18:31	800k
F-CPU_manual-0.2.4-e..>	17-Feb-2002 17:27	1.1M
fromfs.yg01.tgz	14-Mar-2002 19:01	8k
michael98nonblocking..>	19-Mar-2002 20:52	183k
snapshot_yg_3-2002.t..>	18-Apr-2002 16:17	288k
fcpu-mr-20020513.tar.gz	13-May-2002 17:50	27k
dct_fc0.tgz	24-May-2002 21:02	133k
yglfs_scripts.tgz	27-May-2002 21:15	136k
banner.jpg	28-May-2002 21:15	34k
banner2.jpg	29-May-2002 22:18	43k
conf_parinux.zip	13-Jun-2002 02:13	445k
snapshot_yg_6_24_200..>	24-Jun-2002 05:05	274k
snapshot_yg_6_26_200..>	25-Jun-2002 20:35	287k
vhdl-tools.tar.gz	25-Jun-2002 20:44	6k
LICENSE.txt	25-Jun-2002 21:58	15k
mr_error.txt	26-Jun-2002 18:48	1k
generic_adder.vhdl	26-Jun-2002 18:49	14k
fcpu-mr-20020628.tar.gz	28-Jun-2002 18:23	28k
FC0-02_07_2002.gif	03-Jul-2002 01:41	15k
snapshot_yg_04_07_20..>	04-Jul-2002 12:07	298k
fcpu-mr-20020706.tar.gz	06-Jul-2002 12:02	42k
snapshot_yg_07_07_20..>	07-Jul-2002 20:10	310k
snapshot_jws_16_07_2..>	16-Jul-2002 07:49	30k
Snapshot_jws_19_07_2..>	19-Jul-2002 07:50	24k
snapshot_jws_21_07_2..>	20-Jul-2002 22:49	28k
Snapshot_jws_22_07_2..>	22-Jul-2002 17:08	31k
fcpusim_23_july_2002..>	22-Jul-2002 22:26	26k
snapshot_jws_25_07_2..>	24-Jul-2002 19:07	53k
snapshot_yg_25_07_20..>	24-Jul-2002 20:32	333k
snapshot_yg_25_07_20..>	25-Jul-2002 07:05	357k
snapshot_jws_26_07_2..>	26-Jul-2002 16:45	58k
snapshot_yg_27_07_20..>	27-Jul-2002 13:55	365k
snapshot_jws_27_07_2..>	27-Jul-2002 15:07	58k
snapshot_yg_29_07_20..>	28-Jul-2002 19:12	366k
VHDL-HOWTO.f-cpu	28-Jul-2002 19:15	61k
snapshot_jws_29_07_2..>	28-Jul-2002 20:37	59k
snapshot_jws_30_07_2..>	30-Jul-2002 10:38	121k
std_logic_misc.vhd	31-Jul-2002 04:28	33k
20010906.PDF	31-Jul-2002 04:31	185k
scheduler.png	01-Aug-2002 19:27	29k
snapshot_jws_03_08_2..>	02-Aug-2002 19:00	122k
xbar_jws_4_aug_2002.png	03-Aug-2002 19:23	27k
f-cpu_logo.png	18-Sep-2002 17:25	1k
yglfs_scripts_prelim..>	29-Sep-2002 20:27	78k
F-CPU_boot.txt	12-Oct-2002 02:32	17k
gccfcpu.20021203.tgz	03-Dec-2002 06:13	23k
gcc32fcpu_20021229.tgz	31-Dec-2002 05:35	127k
fctools-0.1.tar.gz	01-Jan-2003 18:50	226k
fctools-0.1-0.1.1.diff	02-Jan-2003 18:01	1k
19c3-presentation.pdf	08-Jan-2003 05:37	1.7M
gcc32fcpu_20030110.tgz	10-Jan-2003 13:12	133k
fctools-0.2.tar.gz	11-Jan-2003 21:43	231k
gcc32fcpu_20030113.tgz	13-Jan-2003 18:14	81k
fctools-0.3.tar.gz	08-Feb-2003 19:44	256k
fcpu-mr-20030327.tar.gz	27-Mar-2003 18:29	49k
fcpu-mr-20030402.tar.gz	02-Apr-2003 18:53	49k
fcpu-mr-20030407.tar.gz	07-Apr-2003 18:34	57k
fcpu-mr-20030418.tar.gz	18-Apr-2003 18:55	63k
fcpu-mr-regfile-2003..>	22-Jun-2003 18:27	6k
assign.tgz	18-Aug-2003 22:07	65k
f-cpu_logo.eps	03-Feb-2004 23:12	24k
fcpu-mr-20040215.tar.gz	15-Feb-2004 18:39	73k
fcpu-fpu-adder-annex..>	12-Mar-2004 16:59	441k
fcpu-fpu-adder-rappo..>	12-Mar-2004 16:59	475k
fasu-presentation.pdf	19-Mar-2004 07:21	1.4M

Figure III.1: listing of the /new directory. Everything's not so new..

The initial project subject was to implement an ISS for f-cpu. After having known what is an ISS, we have searched in the f-cpu files if there where some project that could match the definition.

1 The C implementation of fc0 by Yann Guidon

The first project that may match the definition of an ISS is the project of rewriting some parts of fc0 in C. The project was born to make an alternative to the slow and very precise VHDL implementation of fc0. The two models were nearly equivalents, i.e. the C implementation followed the VHDL. This C implementation was Cycle Accurate even RTL.

The project is not officially stopped, but the C code is out of date since a while.
The problem is that:

- We need to maintain each implementation up to date with the other.
- Finally, the simulation time gains are not so important.

2 Fctools

Fctools is a set of useful tools for building programs for the f-cpu. You can found ELF [1] tools, an assembler, a disassembler, a linker and two "instruction-level emulators":

- emu is an program that takes a raw memory dump, and start the emulation at address 0. This emulator integrate a small command line debugger, that allows to execute instructions step-by-step, see the content of registers, memory, etc.
- elfemu take a "real" elf binary, statically linked.

2.1 Fctools global architecture

There are two main parts:

- The first is a port of standard binary tools based on libelf.
- The second is the fcpu instruction set specific tools; the assembler, disassembler and emulators.

Only the second part is interesting for our project. The instruction set is still (less and less) moving. The part that encodes, decodes the instructions is factorized into the `fcpu_opcodes` "library".

2.2 emu.c

fctools' emu contains two emulators, but a great part of the emulation core is shared. `emu.c` contains the instruction dispatcher. This is a big file with the implementation of each instruction behavior. It contains lot of C tricks to make the writing of the code, more efficient, while still execution efficiency. The dispatch is made with a sorted table of flagged opcodes (this means that the `load` opcode will not execute the same function if the instruction has the flag `LS_BIG_ENDIAN`). The correct function to call is then found in the table by the classic dichotomist algorithm.

2.3 Memory structure

In the two emulators of fctools0.3, the memory is directly accessed, after a call to `memmap()`. This function gets an address, and return the pointer to the memory associated to this address. This allows the implementation of a simple TLB (the module that manages the access right of the different memory segments)

2.4 Register structure

The registers of the f-cpu are made of multiple variable-size chunks, for SIMD. The emulator provides a big union to implement that.

[1] ELF is a binary format, that provides a description of the various part of the binary (data segment, code segment, dynamic link infos, etc.)

```
/* a single register */
union reg {
        U8              b[CHUNKS(b)];
        U16             d[CHUNKS(d)];
        U32             q[CHUNKS(q)];
        U64             o[CHUNKS(o)];
        I8              sb[CHUNKS(b)];
        I16             sd[CHUNKS(d)];
        I32             sq[CHUNKS(q)];
        I64             so[CHUNKS(o)];
        float   F[CHUNKS(F)];
        double  D[CHUNKS(D)];
};

/* the register set */
extern struct regs {
        union reg       r[64];
        union reg       r_pc;
} regs;
```

Figure III.2: the register data structure

2.5 Why choosing this emulator as a basis for our project?

Many arguments made us chose to base on this code, instead of doing our own ISS, which was the original plan:

- The code of Michael Riepe is of a big quality, far better than we may have produced. It is no use remaking another architecture, instead of studying a good one, and improve its functionalities.
- fctools is an alive project, Michael is still developing it and has the will of making it the more modular possible.

3 Other emulators

We have also heard about other old f-cpu emulator cores, but these were not very advanced and reusable.

We also have thinking about using generic library such as microlib[4] or SimIt [3], but fctools was a better start.

Chapter IV

Realization

1 TLB, L0 Cache and systemC

As said above, fctools provide a very good base for an ISS. With fctools, we have a behavior accurate simulator. The next step is to build a bus cycle accurate simulator.

The SystemC is here to give us a clock. We first have to make a box for fctools's emu, making a C++/system C wrapper to it. Then we will be able to connect it to other systemC modules such as a motherboard with memory, PCI bus, etc.

The other thing needed to put 'emu' in a systemC system is Cache simulation. We have decided to implement the L0 caches of the fcpu (Instruction cache, aka prefetcher and Data Cache) which is very specific to f-cpu (lot of interactions with the core). The L1 and L2 caches then are more classical and are very easily implementable as systemC modules.

2 L0 Cache specification

The L0 cache is a very important part of the f-cpu. It will determine a great part of the real amount of instruction processed.

As you can see in figure **??** page **??**, there are two L0 Cache. The data cache and the instruction cache. Each cache is very small and fast, so it needs to be managed carefully. This management is done by 2 units.

- The prefetcher is dedicated to the instruction cache.
- The LSU is dedicated to the data cache.

As far as we know, the prefetcher has been much more thought than the LSU, the specification is clear and we manage to implement it efficiently.

2.1 L0-I Cache

Here is the description of the prefetcher behavior, given by Michael Riepe, one of the main developers of the f-cpu project [5]:

Let one fetcher line be the "current" line. The next instruction will be fetched from this line. Every line shall have an associated address, which is the address of the first instruction contained in the line (that is, the address is a multiple of 32). If all instructions from the current line have been fetched, the fetcher will switch to the "next" line (which should have been prefetcher while the current line was executed). That is, the "next" line becomes the new current line.

Any fetcher line can be in one of at least three different states:

- *1 - The line is invalid*

- *2 - The line is being prefetched but not yet valid*
- *3 - The line is valid*

In case the current line is not valid, let the CPU stall until it is. If the line is in 'invalid' state, start prefetching and proceed to state 2.

Whenever the current line is "switched" as outlined above, let the fetcher take the associated address of the new current line, add 32 to it (that's not really an add but a "shifted" increment operation) and start prefetching the (new) next line at the calculated address. If there were only these two lines, they would work just like double or "tandem" buffers – read from one of them while the other is filled in the background.

When "loadaddr[i]" is executed, take the target address, mask off the least significant 5 bits, and start prefetching at the resulting address (if the corresponding line isn't already being prefetched or even valid). In either case, associate the register number with the corresponding fetcher line.

When a jump instruction is executed (and the jump is taken), instructions from the target address may reside in the current line or another (or none at all, which will cause a stall). In the second and third case, switch to the new current line and start prefetching the new "next" line as outlined above. In the first case, simply continue.

If the return address is stored in a register (3-operand form), associate the register number with the line the next instruction would have been fetched from if the jump had not been taken. This will be either the old current line (which is already loaded) or the old next line (which should already be prefetched), so there is no need to start another prefetch operation if the CPU (or the emulator) is in a sane state. If the return address is not stored, and the target address does not reside in the current or next line, the fetcher may (but need not) stop prefetching the old "next" line and/or invalidate it.

If the jump is NOT taken, there is no need to do anything.

Whenever a register is overwritten (note: this applies to ALL instructions!), break the association between the register number and the corresponding fetcher line. If the line is no longer associated with any register afterwards, it may (but need not) be invalidated. Note that an instruction may modify more than one register, so it may be necessary to invalidate several associations at once. On the other hand, it is impossible that any register is associated with more than a single fetcher line (because it can hold only one address at any time).

From a virtual point of view, the current line is always associated with the instruction pointer (PC), and the "next" line is associated with some nameless register inside the prefetcher. These lines must never be invalidated.

There are also special events to consider, e.g. instructions like 'jump r1,r1' must be correctly handled. Another question is whether it makes sense to start another prefetch if a constant is added to or subtracted from an "associated" pointer. It may speed up "calculated jumps", but it seems to be pretty useless in other cases.

If the fetcher is "full" – that is, all lines are in use –, invalidate and overwrite the least recently used (LRU) line that is NOT associated with the instruction pointer (current line) or the prefetcher.

You can see in figure 2.1 page 18 the instruction cache as we have implemented it.

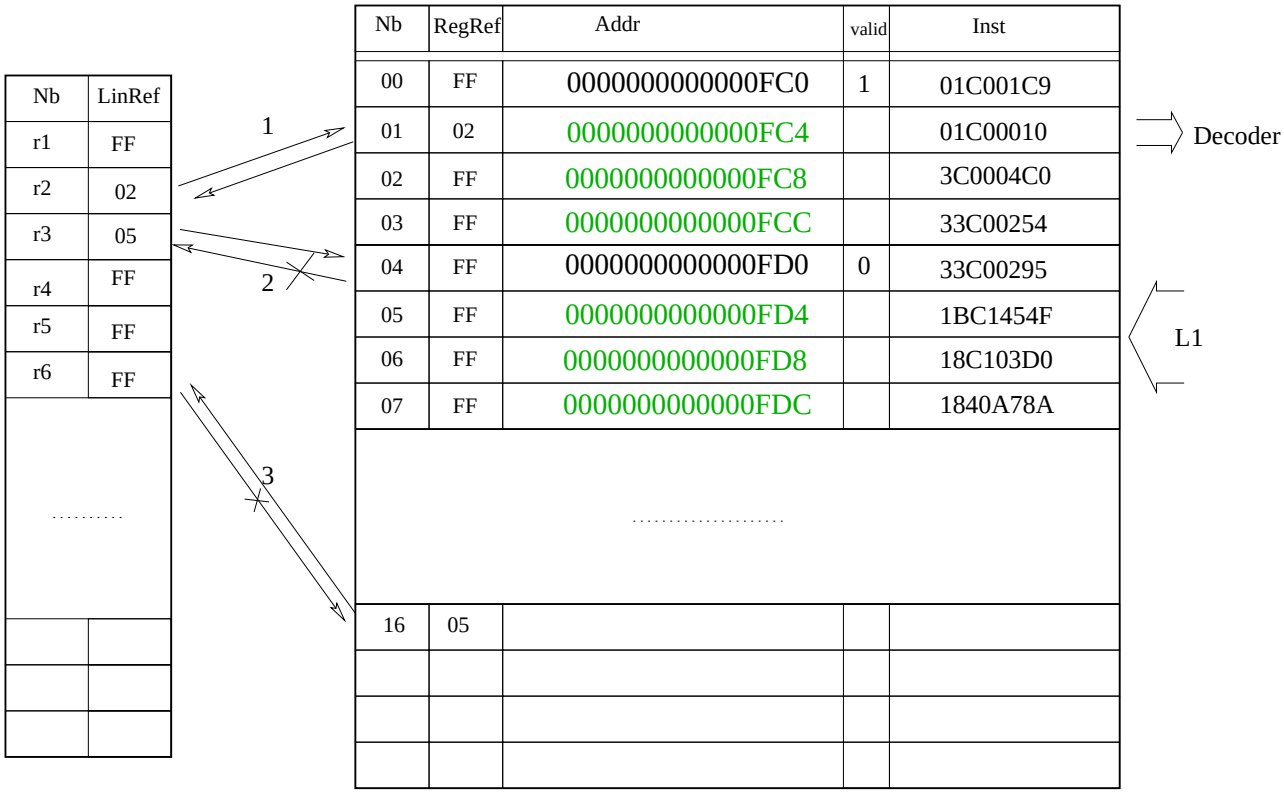


Figure IV.1: the icache structure

The "current" line, the 5th, is being prefetched (L1 data coming in), and the 2nd line is being sent to the instruction decoder.

As you can see, registers may be associated with a line. This must be a bidirectional association to quickly invalidate it if needed. For example:

- The 1st association is valid, if a "jmp r2" occurs, the CPU will not stall.
- The 2nd association is not valid. 5th line is being prefetched, all the associations in 4th to 7th have been invalidated.
- The 3rd association is not valid. The r6 register may have changed recently.

Differences between Michael's description and our implementation.

There are some minor differences between our model and Michael's one.

- The instructions are fetched 4 by 4, and not 8 by 8. This is of course reconfigurable easily by modifying the `IL1_GRANULARITY` constant. More studies about the effect of such a choice can be read in a later chapter.
- When a register whose content is present in the cache is incremented, the reference to the cache line is just forgotten. No complicated optimizations are made.

We have implemented the minimal behavior of the fc0 prefetcher. We can't implement the complicated optimizations, before the RTL is concretized.

Cycle accurate issues

We have implemented some indicators of CPU cycle lost to prefetching. This is important to explore the validity of our model. Although, the values are not just indicators, and are not strictly cycle accurate. We don't have enough experience in RTL to make any decision on the optimizations, or even on the micro architecture.

2.2 L0-D Cache

In the specifications, the Load-Store Unit is defined as following: the Load-Store Unit is just like the Prefetcher, with a granularity of one byte.

However, one can notice other differences:

- The Load-Store Unit namely acts as a read/write cache, whereas the Prefetcher is read only. The write method to L1 cache memory $write-through, write-back$ hasn't been explicitly defined, even though some words in the manual make think about a write-back method.
- The data-flow, handled by the Load-Store Unit, has a variable throughput, ranging from 0 to 64 bits (and more) per cycle, whereas the Prefetcher handles a fixed, 32-bit-per-cycle throughput instruction-flow. Thus, the always-fetch strategy used by the Prefetcher may be unadapted to the L/S-U, as it may cause some useful lines to be swept away from the buffer.
- The association between cache-line and register implies much more complex logics, for example in handling copy of pointer. Indeed, if the source register is a pointer and its pointed memory area is already in the cache, then the destination register should be associated to the same line. Thus, this needs a multiple register association, which is unreasonably complex for a high-speed seeking data cache. Figure IV.2 shows up such a code that can cause the Load/Store Unit to become messy if no verification logic is used to track copies of pointers.

```
loadaddrid %myData, r1 ;      // r1 <- @[MyData]
load       r1,      r3 ;      // r3 <- Mem[r1]
inc        r3,      r3 ;      // r3 <- r3 + 1
store      r1,      r3 ;      // Mem[r1] <- r3
move       r1,      r2 ;      // r2 <- r1 (Copy of pointer)
load       r2,      r3 ;      // r3 <- Mem[r2]
bseti      %2,      r4 ;      // r4 <- 4
add        r3,      r4,  r3 ; // r3 <- r3 + 4
store      r2,      r3 ;      // Mem[r2] <- r3

...

myData:
.long     4 ;
```

Figure IV.2: Example of code which uses copy of pointers

In Table IV.3, we see that the same memory-area is mapped twice in the data-cache, thus the strange results(Table IV.2).

Thus, we have implemented the L/S-U as a classical set-associative write-back data-cache, with Least Recently Fetched(LRF) replacement policy. In addition, we don't use the always-fetch strategy of the Prefetcher. Instead, the data fetching is triggered either by a cache-miss or by a `loadaddr[i]d` instruction. Nevermind, the real L/S-U should functionally and approximately act the same way as our data-cache.

Reg	Data
r1	@[myData]
r2	@[myData]
r3	9
r4	4
..	...

Table IV.1: Expected Register State after execution

Reg	Data
r1	@[myData]
r2	@[myData]
r3	**8**
r4	4
..	...

Table IV.2: Resulting Register State

Reg	Valid	Dirty	Data
r1	true	true	000000000000005
r2	true	true	000000000000008
..	..	..	.....

Table IV.3: Resulting State of the Load-Store Unit

3 Integrating fctools's emu into a SystemC simulator

We finaly have implemented a basic systemc simulator based on fctools' emu, and our modifications on the cache. The goal is to model a complete system, with regard to the cache system.

3.1 Global architecture

Here is the architecture of our systemC model.

- The F-CPU module is a wrapper to the C functions of fctools' emu. Because the L0 caches are very dependant from f-cpu, they are intagrated in this module.
- The L1D cache, is the level 1 data cache, connected to the f-cpu core and to L2 cache
- The L1I cache, is the level 1 instruction cache, connected to the f-cpu core and to L2 cache
- The L2 cache is the last wall before the RAM..

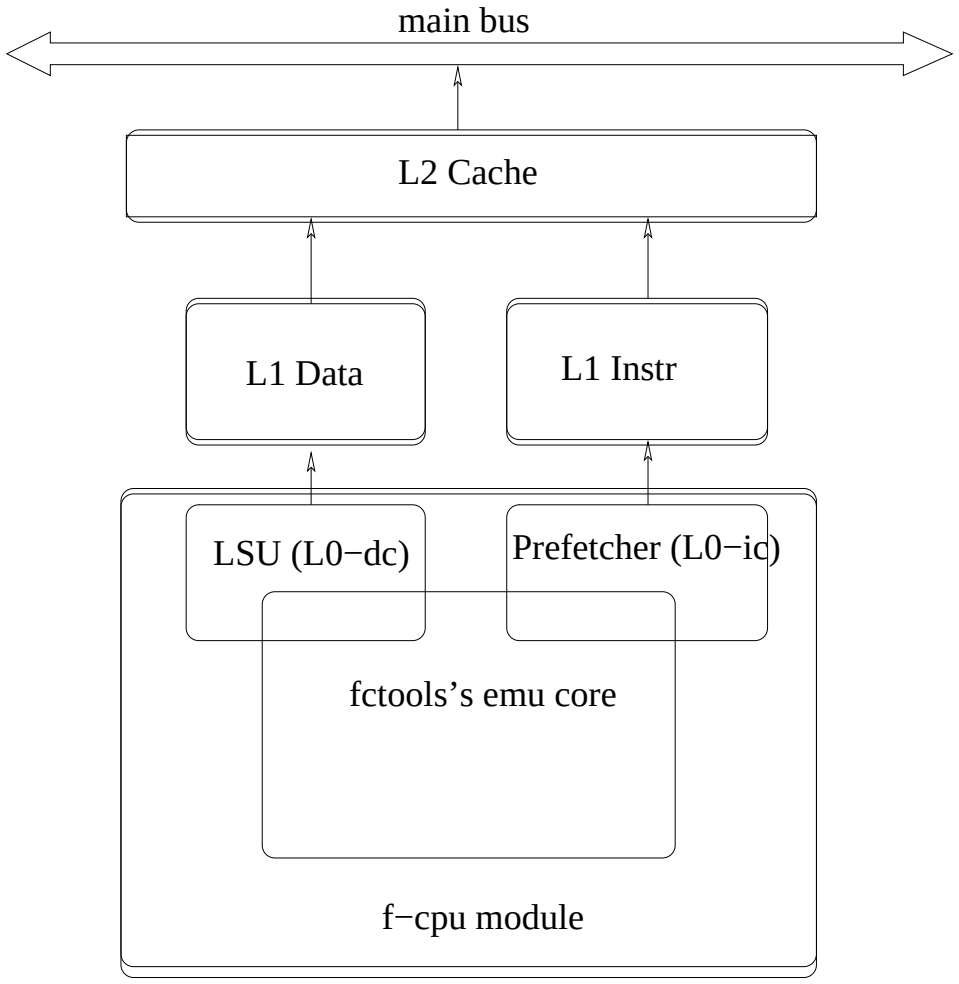


Figure IV.3: the fcpu systemC model

3.2 Limits of our implementation

As you may see by reading the code, our implementation of the theoric model is very simple (for time reasons).

- Our L1 caches are of infinite size, it is implemented as a STL map that remember all the access. If the address as already been accessed, we wait 'hit' cycles, else we wait 'miss' cycles.
- There is no L2 cache. The L1 caches access directly the memory.

The result is a little simulator, which should share the emulator core with emu and elfemu.

Chapter V

Results: It works! Optimization issues

Our output simulator, though its differences with the specifications (some optimizations are not performed by the prefetcher, and the definition of the Load/Store Unit is incomplete), is close enough to the real F-CPU to study some optimization issues at software level (mostly at compiler level).

1 The benchmark programs

To track these possible optimization tricks, we have written two assembly programs:

- `matmul`, a matrix of 64-bit integers multiplication program to examine the behavior of the prefetcher regarding three embraced loops and multiple `loadaddri` instructions.
- `bbsort`, a simple bubble-sort program. This program operates on an 8-bit data vector and highly (excessively) stimulates the Load-Store Unit, as it uses many read-after-write.

The first execution of `matmul` has shown up the problem in the specifications (or at least in our understanding of the specifications) of the Load/Store Unit.
Then we were able to measure the speed-ups of both the prefetcher and the load-store unit.
For that, we have implemented some reports into the ISS. At the end of the simulation, the profile of the program, i.e. the total amount of clock cycle used for each instruction of the program. Before each asm instruction of the program, a line is displayed with

- The total amount of cycle used here
- The detailed profile, i.e. the number of cycle used each time the instruction was executed. This is useful to know why this instruction is so used. If there are a lot of 1, this means that this instruction is very often used, and that the cache is very useful (each time the instruction takes only one cycle). If you have a bigger value, this means that the CPU has stalled during some cycles to fetch or save a data/ an instruction.
- The values in parenthesis are for cycles used in data cache. Remember that when there is a data cache access, the prefetcher is not loosing its time; it is still prefetching the next block of instructions.

```
19   =            19
    0000   bseti          $0x14, r0, r62
01   =            1
    0004   loadaddrid     $0xc4, r1
01   =            1
    0008   loadaddrid     $0xe0, r2
01   =            1
    000c   loadaddrid     $0xfc, r3
--
15   =            15
    0010   bseti          $0x2, r0, r24
01   =            1
    0014   bseti          $0x3, r0, r28
01   =            1
    0018   move           r0, r4
01   =            1
    001c   move           r0, r5
--
15   =            15
    0020   move           r0, r6
01   =            1
    0024   loadcons.0     $0x2, r4
01   =            1
    0028   loadcons.0     $0x2, r5
01   =            1
    002c   loadcons.0     $0x2, r6
--
15   =            15
    0030   move           r1, r7
01   =            1
    0034   move           r2, r8
01   =            1
    0038   mul.d          r28, r6, r30
01   =            1
    003c   mul.d          r28, r5, r31
--
15   =            15
    0040   move           r0, r11
01   =            1
    0044   move           r1, r7
01   =            1
    0048   loadaddri      $0xc, r17
01   =            1
    004c   loadaddri      $0x14, r18
--
15   =            15
    0050   loadaddri      $0x24, r19
01   =            1
    0054   jmp            r17
08   =            1+7
    0058   move           r0, r12
02   =            1+1
    005c   move           r2, r8
--
24   =            19+5
    0060   jmp            r18
10   =            1+7+1+1
    0064   move           r0, r13
04   =            1+1+1+1
    0068   move           r8, r10
04   =            1+1+1+1
    006c   move           r7, r9
--
20   =            15+1+3+1
    0070   move           r0, r16
04   =            1+1+1+1
    0074   jmp            r19
14   =            1+1+1+1+1+1+1+7
    0078   load           r9, r20
46   =            (19)+1+(19)+1+1+1+1+1+1+1
    007c   load           r10, r21
--
67   =            (19)+1+(19)+1+1+(19)+1+3+1+1+1
    0080   mul            r20, r21, r15
08   =            1+1+1+1+1+1+1+1
    0084   add            r16, r15, r16
08   =            1+1+1+1+1+1+1+1
    0088   add.d          r10, r30, r10
08   =            1+1+1+1+1+1+1+1
    008c   add.d          r9, r28, r9
--
10   =            1+1+1+1+3+1+1+1
    0090   inc            r13, r13
08   =            1+1+1+1+1+1+1+1
    0094   cmpg           r13, r5, r14
08   =            1+1+1+1+1+1+1+1
    0098   jmpnz          r14, r19
04   =            1+1+1+1
    009c   store          r3, r16
--
61   =            (19)+1+(19)+1+1+(19)+1
    00a0   add            r3, r28, r3
04   =            1+1+1+1
    00a4   inc            r12, r12
04   =            1+1+1+1
    00a8   add            r8, r28, r8
04   =            1+1+1+1
    00ac   cmpg           r12, r6, r14
--
04   =            1+1+1+1
    00b0   jmpnz          r14, r18
02   =            1+1
    00b4   inc            r11, r11
02   =            1+1
    00b8   add.d          r31, r7, r7
02   =            1+1
    00bc   cmpg           r11, r4, r14
--
02   =            1+1
    00c0   jmpnz          r14, r17
01   =            1
    00c4   halt
453 cycles in total
```

Figure V.1: The profile of the matmul program

2 Software level optimizations for fcpu's caches.

The fact is that compilators sometimes add nop into programs. The main reason given is that it is to avoid pipeline faults (for example, data dependency error, when an instruction need a value that is not yet totally computed). This is mind; we have wondered if there would be some optimization to avoid cache misses.

2.1 L0-I Cache.

We have made a script that add 0 to 5 nops in many places of our matmul program, and then find the best combination, that needs the least number of cycles to compute the matrix multiplication. The result is actually difficult to interpret. There are in fact alignment issues that may affect the execution time of one program, It is clear that adding nop may correct the alignment of the beginning of a loop, placing it at an divisible-by-16 address. However, placing too much nop means loosing the time needed to execute the nop...

For high performance, a program should always jump to a well-aligned address

2.2 L0-D Cache.

We haven't enough time to run tests on optimizing the D-Cache access at compiler level.

2.3 Further analysis.

As we said before, the results are not easy to interpret. Compilers usualy use heuritic to optimise such issues. We are sure that it is possible to find other optimisation clues that simply say *please jmp at aligned address*, however, this work need time and probably a more important bibliography, and may be achieved in another study.

```
   15   =            15
        0050   loadaddri       $0x24, r19
   01   =            1
        0054   jmp             r17
   08   =            1+7
r17->   0058   move            r0, r12
   02   =            1+1
        005c   move            r2, r8
   --
   24   =            19+5
        0060   jmp             r18
   10   =            1+7+1+1
r18->   0064   move            r0, r13
   04   =            1+1+1+1
        0068   move            r8, r10
   04   =            1+1+1+1
        006c   move            r7, r9
   --
   20   =            15+1+3+1
        0070   move            r0, r16
   04   =            1+1+1+1
        0074   jmp             r19
   14   =            1+1+1+1+1+1+1+7
r19->   0078   load            r9, r20
   46   =            (19)+1+(19)+1+1+1+1+1+1+1+1
        007c   load            r10, r21
   --
   67   =            (19)+1+(19)+1+1+(19)+1+3+1+1+1
        0080   mul             r20, r21, r15
   08   =            1+1+1+1+1+1+1+1
        0084   add             r16, r15, r16
   08   =            1+1+1+1+1+1+1+1
        0088   add.d           r10, r30, r10
   08   =            1+1+1+1+1+1+1+1
        008c   add.d           r9, r28, r9
   --
   10   =            1+1+1+1+3+1+1+1
        0090   inc             r13, r13
   08   =            1+1+1+1+1+1+1+1
        0094   cmpg            r13, r5, r14
   08   =            1+1+1+1+1+1+1+1
        0098   jmpnz           r14, r19
   04   =            1+1+1+1
        009c   store           r3, r16
   --
   61   =            (19)+1+(19)+1+1+(19)+1
        00a0   add             r3, r28, r3
   04   =            1+1+1+1
        00a4   inc             r12, r12
   04   =            1+1+1+1
        00a8   add             r8, r28, r8
   04   =            1+1+1+1
        00ac   cmpg            r12, r6, r14
   --
   04   =            1+1+1+1
        00b0   jmpnz           r14, r18
   02   =            1+1
        00b4   inc             r11, r11
   02   =            1+1
        00b8   add.d           r31, r7, r7
   02   =            1+1
        00bc   cmpg            r11, r4, r14
   --
   02   =            1+1
        00c0   jmpnz           r14, r17
   01   =            1
        00c4   halt
   453 cycles in total
```

```
   15   =            15
        0050   nop
   01   =            1
        0054   loadaddri       $0x24, r19
   01   =            1
        0058   jmp             r17
   08   =            1+7
r17->   005c   move            r0, r12
   --
   25   =            19+6
        0060   move            r2, r8
   02   =            1+1
        0064   jmp             r18
   10   =            1+1+1+7
r18->   0068   move            r0, r13
   04   =            1+1+1+1
        006c   move            r8, r10
   --
   24   =            15+1+3+5
        0070   move            r7, r9
   04   =            1+1+1+1
        0074   move            r0, r16
   04   =            1+1+1+1
        0078   jmp             r19
   08   =            1+1+1+1+1+1+1+1
r18->   007c   load            r9, r20
   --
   48   =            (19)+1+1+1+1+(19)+1+1+3+1
        0080   load            r10, r21
   46   =            (19)+1+(19)+1+1+1+1+1+1+1
        0084   mul             r20, r21, r15
   08   =            1+1+1+1+1+1+1+1
        0088   add             r16, r15, r16
   08   =            1+1+1+1+1+1+1+1
        008c   add.d           r10, r30, r10
   --
   10   =            1+1+1+1+1+1+3+1
        0090   add.d           r9, r28, r9
   08   =            1+1+1+1+1+1+1+1
        0094   inc             r13, r13
   08   =            1+1+1+1+1+1+1+1
        0098   cmpg            r13, r5, r14
   08   =            1+1+1+1+1+1+1+1
        009c   jmpnz           r14, r19
   --
   04   =            1+1+1+1
        00a0   store           r3, r16
   42   =            (19)+1+1+(19)+1+1
        00a4   add             r3, r28, r3
   04   =            1+1+1+1
        00a8   inc             r12, r12
   04   =            1+1+1+1
        00ac   add             r8, r28, r8
   --
   04   =            1+1+1+1
        00b0   cmpg            r12, r6, r14
   04   =            1+1+1+1
        00b4   jmpnz           r14, r18
   02   =            1+1
        00b8   inc             r11, r11
   02   =            1+1
        00bc   add.d           r31, r7, r7
   --
   02   =            1+1
        00c0   cmpg            r11, r4, r14
   02   =            1+1
        00c4   jmpnz           r14, r17
   01   =            1
        00c8   halt
   415 cycles in total
```

Figure V.2: Profile of the naive program version , and a “nop-optimised” version. 10 % of gain with 1 well placed nop.

Conclusion

The main result of this project is for us a far better understanding of what is an ISS, and why it is very usefull in System on Chip development. Then, we have the satisfaction to have made the f-cpu project advance, by giving the first implementation of the prefetcher, long time discussed, but never coded. We hope that this will give motivation to the RTL coders of the f-cpu team to resolve the micro-architecture issues of such a module..

Our results about data cache fill us a bit less with enthusiasm, as it became a simple LRF cache in order to work and answer all the cases.

The next process for fctools is beeing though by Michael Riepe, as he said recently in the mailing list. He is thinking on some changes that will make his simulator cycle accurate. This implies that the simulator gives all informations about cycles lost in pipeline and in cache. For the last thing, we hope that our work will fit the needs, and will be useful for him.

Bibliography

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

[2] Yann Guidon and F-CPU Design Team, *Présentation ISIMA*
`http://fcpu.seul.org/waigee/ISIMA`

[3] W. Qin, S. Malik, *Arm SimIt, CA and BCA ISS for ARM*
`http://www.ee.princeton.edu/~wqin/armsim.htm`

[4] G. Mouchard, *MicroLib*
`http://www.microlib.org/`

[5] Michael Riepe *Prefetcher description posted on the mailing list*
`http://archives.seul.org/f-cpu/f-cpu/Jan-2004/msg00019.html`