Hardware Platform to Test New ISA Extensions

Author:
Mikel Fernández Oreja

Advisor:
Daniel Jiménez-González

September 2010

Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my advisor Daniel Jiménez-González for all the help, time and dedication he has offered me. I would also like to thank Cecilia González Álvarez for her patience and her willingness to help me.

Chapter 1

Introduction

Testing how modifying the Instruction Set Architecture (ISA) of a processor would change its performance is an interesting analysis that usually is made through software based hardware simulators. This is so because implementing the changes that would allow extending an ISA in hardware is costly in both effort and time. But in the other hand, software simulation is slow and the accuracy provided is never as good as using the hardware itself.

To address these problems an automated hardware generator has been implemented and analyzed to build an infrastructure that will allow to test ISA extensions in hardware in a simple way. For this matter, an OpenSPARC has been used as the baseline hardware architecture, and a FPGA as the platform where it will be deployed.

The goal of this work is to provide a platform to allow easy changes to the done to the OpenSPARC. The modified OpenSPARC solves the existing problems on analyzing the impact caused by ISA extensions. Instead of simulating that impact, real hardware is built to make the measurements more accurate, and to allow faster execution of tests as well as complete operating systems. Moreover, the platform should provide a way to use more than two operands by the new instructions.

A set of scripts has been developed to patch OpenSPARC's RTL[1] Verilog[2] code to make changes easily implementable. Designers do not have to implement the logic required to detect interlocks or bypasses, because the platform will automatically take care of these issues, freeing the designer from having to know the internals of the OpenSPARC.

This modified hardware will be implemented on an FPGA to make it more flexible to changes. An Ubuntu (Linux 2.6) operating system image will be loaded on the FPGA to check if the system still works after the modifications.

To execute programs on the modified OpenSPARC, their assembly code will be modified so the programs make use of the newly available instructions.

[1] Register transfer level, hardware description of how data is transformed between registers. This abstraction level is synthesizable.

[2] Verilog is a hardware description language (HDL)

1.1 Motivation

PhD. student Cecilia Gonzalez Alvarez's ongoing work on compilers in the Barcelona Supercomputing Center (BSC) analyzes programs and determines which non-implemented instructions of a target processor can be useful to have in order to boost performance. The accuracy of the analysis can be later proved using a software simulator. But this presents the following problems:

- Slow simulations for large programs or full systems.
- Differences between the hardware simulator and the real hardware.
- The implementation of new instructions in the simulator could not take into account hardware limitations that make adding a new instruction more costly than assumed in the simulation.
- Some drastic changes, such as allowing a new instruction to use more than two source operands can not be just simulated and ignore the fact that that would require several changes in the hardware.

Ideally, a new processor would be implemented with the changes required to support the new instructions. This would take a lot of time and a lot of work done would not be reusable in the future if a different ISA extension is needed to be analyzed. An easily changeable hardware would be the most desirable solution.

1.2 Specific objectives

The following are the objectives of this work:

1. Development of a platform to allow easily implementable ISA extensions in hardware.
   (a) Must be totally automated, leaving to the designer only the design of the combinational logic of the functional unit.
   (b) Allowing flexibility in the extensions.
   (c) Allowing more than two source operands to be used.
   (d) Freeing the designer of having to know the microarchitecture.
2. Analysis of the hardware generated by the platform.
   (a) Overhead analysis.
   (b) Modifiability analysis.
   (c) Tools to help the analysis.
   (d) Tools for the generation of the .word instruction word to be inserted in the assembly code.

3. Implementation of the OpenSPARC based platform support on a FPGA.

4. Implementation of samples of new instructions specialized in executing a set of algorithms.

5. Implementation of samples of programs that will use the newly available instructions.

6. Performance analysis of the original programs running on the original hardware, and the modified programs running on the modified hardware.

All the planned objectives have been accomplished. The last point, doing a performance analysis of the hardware and comparing modified versus unmodified versions, has been only partially tested in hardware. Instead, the full system has been simulated on the RTL simulator of the OpenSPARC. The reason is mainly that we realize, in the last stage of the development, that the original OpenSPARC system generated hardware emulates the floating point unit in a program running on the Microblaze softprocessor used in the system. The rest of modifications done has been successfully tested. Therefore, that objective was most desirable to be accomplished by running the tests on an FPGA, but simulation plus hardware synthesis gave good enough results. In any case, the modified hardware could be downloaded to a FPGA and a complete operating system could be booted.

1.3 Document organization

The rest of this document is divided in the following chapters:

1. State of the Art: this chapter shows the related and previous work.

2. OpenSPARC: this chapter gives a brief overview of the OpenSPARC processor, which is the baseline processor used for this work.

3. Design: in this chapter all the design alternatives existing and the reasoning under the final selection of one of them is presented.

4. Implementation: this chapter explains how the ISA Extension Platform has been developed, and how it can be used.

5. ISA Extension Platform Support Analysis: this chapter presents the experimental setup, the tests conducted, and the results.

6. Conclusions: this chapter summarizes the work done and points to possible future work.

7. Appendix A: this appendix has an example on how to generate a new hardware using the platform developed and analyzed in this work.

8. Appendix B: in this appendix a figure summarizing the module hierarchy of the OpenSPARC is shown.

Chapter 2

State of the Art

The ability to extend the ISA or having configurable architecture has been an interesting research subject for some time. Xtensa [9] is a synthesizable processor that implements a superset of the traditional RISC ISA, and also allows to configure additional instructions depending on the application the designer can choose. This processor implements a traditional 5 stage pipeline and does not allow deeper pipelines to be used. Its aim is not to modify an existing processor with a defined ISA; instead they use an ISA extension description language (TIE). This language can be used to generate multiple cycle extensions [11].

In [10], authors worked on extending the ISA of an open implementation of the Sparc v8 ISA called LEON-2. Their focus was to add multimedia SIMD instructions to make a video compressing algorithm run faster.

The MOLEN [13] is a reconfigurable processor which exposes its architecture to the programmer. It allows extending the ISA without architectural modifications. This platform has been used in [14] and [15] to test hardware extensions to speed up bioinformatics applications, using profiling information to identify candidate ISA extensions, selecting the best candidates, and finally generating a hardware description of the ISA extension and executable code which will use that extension. This selection strategy will be used to automate the process of generating ISA Extension with the platform framework presented in this work.

Trimaran [16] is a popular, research oriented, very parametrizable integrated compiler and simulation infrastructure. It allows customization of every detail of an architecture. This platform has been widely used to work on architectural modifications and compiler back end optimizations [17] [18] [19]. Trimaran can be used as a compiling infrastructure to analyze the modified applications.

Chapter 3

OpenSPARC

OpenSPARC [7] is a pipelined open source RISC processor implementing the Sparc v9 ISA and the Niagara architecture developed by Sun Microsystems (now Oracle). It has been used as the target architecture to modify to allow extensions to its ISA. The election of this processor is due to its open source nature, providing free access to its source code written in the Verilog hardware description language (HDL). It is an in-order processor.

OpenSPARC T1 can be built with 4 threads per core and up to 8 cores per chip. A crossbar is used for inter-core communication, such as accessing the L2 cache or accessing the Floating Point Unit (FPU). The FPU is shared between all the cores. Figure 3.1 shows a 8-core OpenSPARC T1, while figure 3.2 shows the OpenSPARC T1 core pipeline.

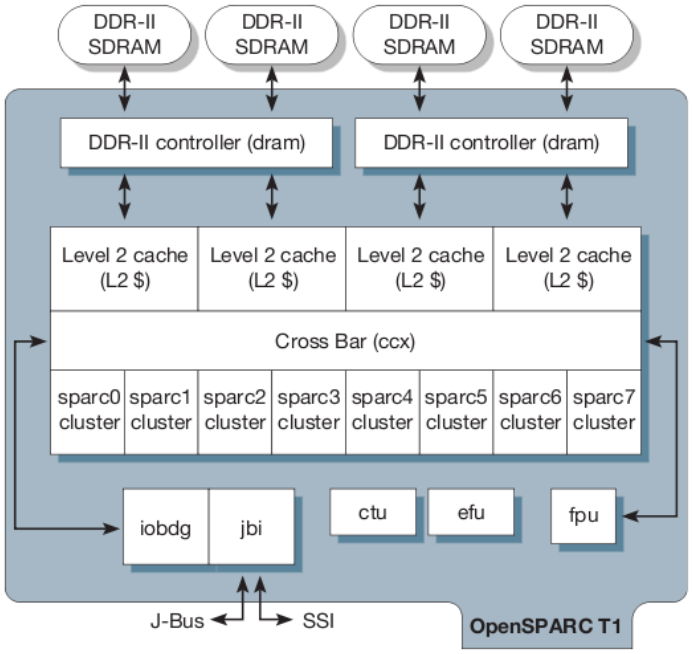


Figure 3.1: OpenSPARC T1 block diagram [3].

Threads in a core are executed in a round robin basis, sharing a Register File (RF) structure which uses a windowing system to separate each thread's registers.

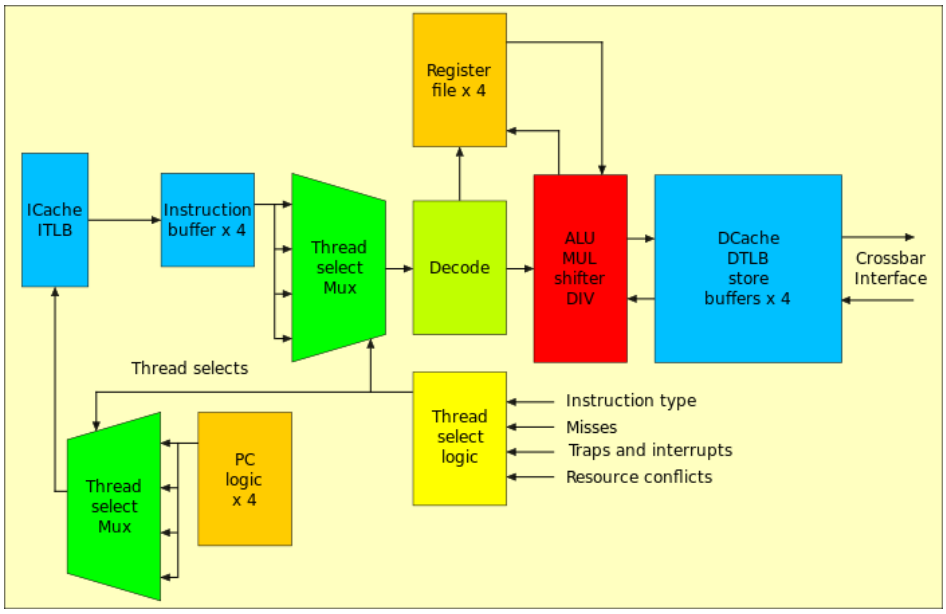


Figure 3.2: OpenSPARC T1 core pipeline [4].

The Verilog design of the processor is divided in modules. There is a module for the sparc core (which is divided in other sub-modules describing elements in the processor, such as the instruction fetch unit, the floating point front end (FFU), the load and store unit, etcetera), a module for the floating point unit (FPU), and a module for the crossbar (CCX). There are also additional modules for memories and interfaces. In the appendix B the module hierarchy of the OpenSPARC design can be found.

In this work, a single-core single-threaded OpenSPARC T1 has been used.

Chapter 4

Design Exploration

To allow ISA extensions, there may be different approaches:

- Modifying the execution unit: adding a new functional unit to the datapath. This would have been the natural way of extending the processor, as this is the place where all the other units are. However, this is not trivial to achieve, because changing control of the main pipeline has proved to be extremely costly.
- Modifying the floating point unit: similar to the previous option, but adding the new functional unit in the FPU (outside of the core). This option requires a higher latency for the operands to reach the unit. However, it is more easily modifiable because it implements input and output queues that maintain order. That allows to implement new instructions with arbitrary latency without making many changes to the control unit.
- Modifying the Cryptography unit (or Stream Processing Unit, SPU): This part of the code is implemented as a module in the core, but the source code is missing from the release of the OpenSPARC due to the United States cryptography export controls, which state that cryptography products can not be open-sourced.

Modifying the execution unit (the regular integer pipeline) would be faster than using the FPU or the SPU for two operand instructions, but does not allow any extra flexibility.

The SPU had the biggest potential, because it is in the core and allows a bigger flexibility than the execution unit, as it can work as a coprocessor and it has an interface to the L2 cache. However, after studying how to modify it, it was concluded that the SPU was not reasonably easy to modify, so the FPU was finally used.

The FPU provides a high potential for modifications, as it is very decoupled from the rest of the processor, but only allows two source operands to be used. To overcome this and add flexibility to the design, a new register file was added to the FPU to allow its contents to be used as operands in the ISA extensions. This new register file is called Temporal Register File (TRF) and can only be written by using three new instructions called Temporal Move (TMV). The values stored in the TRF can be read and used by the instructions implemented using the ISA extension platform. TMV instructions are explained in section 4.2, later in this chapter.

4.1 Floating Point Unit (FPU)

The FPU is located outside of the core. OpenSPARC uses the Load/Store Unit (LSU) in order to send the instructions through the crossbar. The Floating Point Front-End Unit (FFU) is located in the core and it manages sending the operands through the LSU and getting the results generated at the FPU from the crossbar.

The FPU has three different pipelines, one for each floating point operation family implemented: add, multiply, and division. Add and multiply data paths are totally pipelined and have a fixed number of stages, while divisions take a variable time to execute (depending on its operands) and are not pipelined. There are other simpler floating point instructions that are implemented in the core's FFU, such as register to register and memory to register movement instructions.

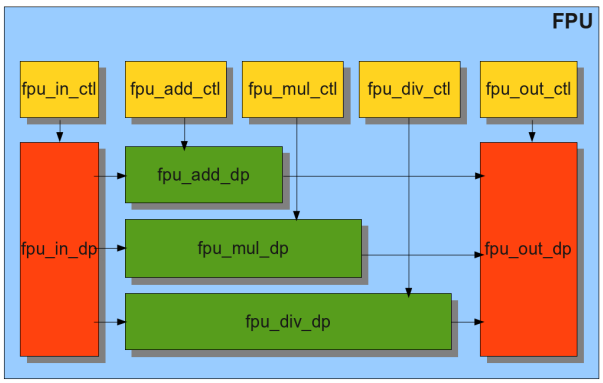


Figure 4.1: Block diagram of the FPU. Modules with a name finishing with _ctl are control units.

In figure 4.1 a block diagram of the FPU is presented. The FPU has two input FIFO queues, implemented in the *fpu_ in* module (which is divided in two submodules: *fpu_ in_ ctl* as the control unit and *fpu_ in_ dp*). One of them has priority over the other one and executes floating point divisions. The other one executes add and multiply instructions. The FPU also has two output queues implemented in the *fpu_ out* modules prioritizing divisions, but output queues are not FIFO: they dispatch the result calculated, and if there is more than one pipeline giving a result in the same cycle, the queue works as a round-robin.

Any new pipeline added to the OpenSPARC using the ISA extension platform framework will work the same way as *fpu_ add* or *fpu_ mul*, divided in two submodules (one for the control unit and another one for the data path), and may return only one value to the Sparc core.

4.1.1 Floating Point Front-End Unit (FFU)

The FFU is a module located in the Sparc core. It makes use of the LSU to send operands and instructions to the FPU. It is capable of reading registers in the Floating Point Register File (FRF). In figure 4.2 a block diagram of the FFU module is shown, where DP stands for data path, CTL for control logic, VIS is the sub-module which

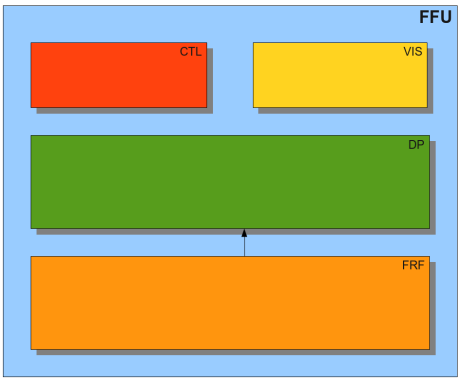


Figure 4.2: Block diagram of the FFU module.

manages a SIMD visual instructions and FRF is the floating point register file.

The FFU may send two operands per instruction to the FPU, and for this matter the LSU is used. The LSU needs two cycles to send the instructions plus the operands through the crossbar, regardless of the number of operands being sent.

The FFU stalls while waiting for the results of the FPU. The results are collected listening directly to the crossbar, so there is not any interaction with the LSU for receiving data from the FPU.

4.2 Semantics and syntax of new instructions

The ISA Extension Platform requires the TRF to allow flexibility in the number of operands that a new instruction may use. The platform tools add the TRF to the FPU and generate an extra pipeline called TMV which will allow the programmer to send data from the core to that register file. The new implemented instructions will be able to read data from this register file, so the new instructions will be able to read two input values (from the FRF) and up to 8 64-bit values read from the TRF. Those 10 operands should be enough for our current needs, but the TRF could be easily replaced by a bigger one.

TMV instructions allow one or two operands to be moved from the FRF to the TRF. The TMV pipeline syntax and semantics follow:

- **TMV1**:
  - Syntax:

    ```
    TMV1 %RS1, %RS2
    ```

  - Semantic description:

    ```
    {Precondition: 0 <= FRF[%RS1] <= 7, %RS1 and %RS2 are floating point registers}
    a <= FRF[%RS1]
    b <= FRF[%RS2]
    FRF[a] <= b
    ```

    1. Read RS1 and RS2 source operands from the FRF.

2. Use RS1 as an index. Store RS2 in the TRF of the FPU, at position RS1.

- **TMVR**:
  - Syntax:

    ```
    TMVR %RS1, %RS2
    ```

  - Semantic description:

    ```
    {Precondition: %RS1 and %RS2 are floating point registers}
    a = FRF[%RS1]
    b = FRF[%RS2]
    TRF[0] <= a
    TRF[1] <= b
    idx <= 2
    ```

    1. Read RS1 and RS2 source operands from the FRF.
    2. Store RS1 at position 0 of the TRF. Store RS2 at position 1 of the TRF.
    3. Set the *idx* index pointer of the TRF to 2.

- **TMV2**:
  - Syntax:

    ```
    TMV2 %RS1, %RS2
    ```

  - Semantic description:

    ```
    {Precondition: idx is set, %RS1 and %RS2 are floating point registers}
    a <= FRF[%RS1]
    b <= FRF[%RS2]
    TRF[idx]   <= a
    TRF[idx+1] <= b
    idx <= idx + 2
    ```

    1. Read RS1 and RS2 source operands from the FRF.
    2. Store RS1 in the TRF, in the position indexed by the index *idx*.
    3. Store RS2 in the TRF, in the position indexed by the index *idx+1*.
    4. Increase the *idx* index by two.

In figure 4.3 a view of all the available register files in the processor is shown. RF is the integer register file, located in the Sparc core. FRF is the floating point register file, in the FFU module in the core. TRF is the new register file, added to temporarily hold input values for the new pipeline, and located in the FPU.

A general new instruction has the following syntax and semantics

- **newInstr**:
  - Syntax:

    ```
    newInstr %RS1, %RS2, %RD
    ```

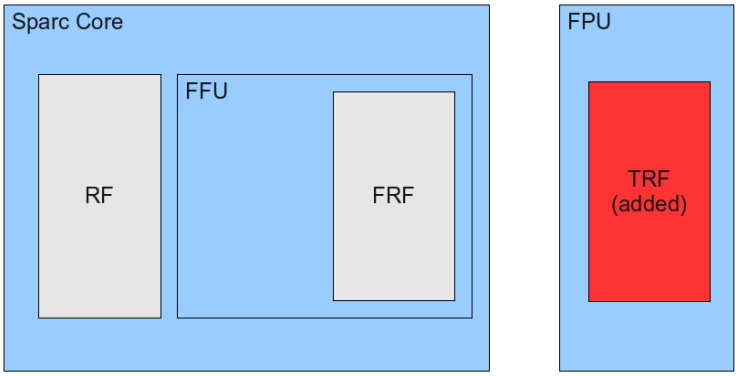


Figure 4.3: Available register files.

– Semantic description:

```
{Precondition: RS1, RS2 and RD are floating point registers}
a <= FRF[%RS1]
b <= FRF[%RS2]
result <= [The operation of the instruction has to be implemented.]
FRF[%RD] <= result
```

1. Read RS1 and RS2 source operands from the FRF.
2. The operation of the instruction has to be implemented. Up to 8 values can be read from the TRF.
3. The result is stored in register RD of the FRF.

Chapter 5

Implementation

The hardware platform to extend the Sparc ISA has been developed on the OpenSPARC T1 version 1.1.7. The main goal of the implementation work is to have an automatic simple way of generating the needed changes in the Verilog source code. In order to achieve this goal, we implement the tools so that just specifying how many instructions will be added to the ISA and their latencies a patched version of the OpenSPARC with all these changes applied will be provided.

OpenSPARC design is divided in three top-level modules in Verilog code which are synthesized separately. Those are the *sparc* core, the *fpu* and the *ccx* (crossbar). The top-level design has been left untouched.

Changes introduced consist on adding an empty pipeline with the appropriate control signals. Therefore, the specific hardware needed to execute the new instructions will be easily insertable. The platform tools also output the coding of the instruction(s) that will be used.

An interface has been defined to easily use the modified OpenSPARC to implement the new pipeline. The goals of the generated code is to provide:

- Clear separation between stages to design fully pipelined functional units.
- Easy access to input data, such as operands, at any stage.
- Easy access to the TRF.
- Storage of partial results between stages.
- Automated control flow management.
- Automated output of results.
- Separation between sequential and combinational logic.

A set of signals is generated in order to make all the advantages described available. Most of them are for internal use, but some others are intended to be used by the developer, as described in the following sections.

5.1 Platform usage

The ISA Extension Platform framework has been developed to allow very easy modifications. It is totally automated and generates all the required changes except for the functional unit itself. The platform implements the new data path and takes care of all conflicts that may exist in run time. The platform framework is distributed as a package containing:

- A script which generates new RTL code of the OpenSPARC and modifies the required RTL files of the original OpenSPARC. A list of the modified files and a summary of the changes the platform tools will apply to them is explained in section 5.2.
- New Verilog modules to enable the TRF, and TMV instructions: *fpu_tmv.v* working as a pipeline which will execute TMV instructions, and *fpu_rf.v* as the module implementing the TRF.

The user must have a bash shell in the */usr/bin* location and needs to set the *OPENSPARC_SRC* environment variable pointing to the OpenSPARC T1 installation directory. Then the script is ready to be run. It takes no parameters, the input is read from the standard input after prompting the user. The following information is asked to the user:

- Name for the new pipeline (3 letters).
- Identifier for the new pipeline (1 letter).
- Number of stages for the new pipeline (number greater than 0).
- Whether the TRF will be used or not (y/n choice).

Both a 3 letter name and a 1 letter identifier are asked to the user to follow OpenSPARC's convention of having two different ways to identify a pipeline in the FPU. After reading this information, the tools generate new code which overwrite the old one in the *OPENSPARC_SRC* directory. Flist files, which reference all source files in the design, used by the sims simulation script and the rxil synthesis script are also modified. These Flist files are located at the following paths, and should not require manual modifications:

- sims file list: design/sys/iop/fpu/rtl/Flist.fpu
- rxil file list: design/sys/iop/fpu/xst/fpu.flist

A step-by-step example on how to add a new pipeline containing a new instruction to speed-up the FIR algorithm analyzed in chapter 6 is described in the appendix A.

5.2 RTL code supporting the new ISA

The platform framework generates five new Verilog modules (each one in a separated file) for the new pipeline and will modify several original RTL files in order to make the new pipeline work. The following is a list of the RTL files of the original OpenSPARC that are modified. All directories are under *$OPENSPARC_SRC/design/sys/iop/*.

- sparc/ffu/sparc_ffu_ctl.v
- fpu/fpu.v
- fpu/fpu_in.v
- fpu/fpu_in_ctl.v
- fpu/fpu_out.v
- fpu/fpu_out_ctl.v
- fpu/fpu_out_dp.v
- fpu/fpu_rptr_groups.v

The following list shows the new files added to the design (being "new" the code name for the new pipeline):

- fpu/fpu_new.v
- fpu/fpu_new_ctl.v
- fpu/fpu_new_dp.v
- fpu/fpu_tmv.v
- fpu/fpu_rf.v

A brief description of the changes done is presented in the following subsections.

5.2.1 sparc_ffu_ctl

This is the only module modified in the Sparc core. It is part of the FFU module and is responsible for controlling FFU's data path. It is also the module which decodes every floating point instruction (which contain the floating point opcode called opf). This decoding part must be modified to allow unimplemented floating point opcodes to be used for the new instructions.

This module is also responsible of stalling the pipeline whenever an instruction which might use the TRF is ready to go to the FPU, but an older instruction which also may use the TRF is still being executed. This is to avoid more than one access to the TRF happening at the same cycle (see section 5.2.13 for more information).

5.2.2 fpu

This is the top-level module for the FPU, where all the other submodules in figure 4.1 are instantiated. Here the new pipeline plus the TRF and the TMV pipeline are instantiated. New wires are added to plug in the new modules.

5.2.3 fpu_in

This module manages the inputs of the FPU. It is divided in two sub-modules, fpu_in_ctl and fpu_in_dp. This module reads the information from the crossbar and provides information about execution requests.

5.2.4 fpu_in_ctl

This is the control unit for the fpu_in module. It has been changed to recognize the opcodes of the new instructions and output an appropriate control signal when a new instruction is found. For this matter, it needs more bits of opcode information than it used to need, and also it requires two extra control signal bits of output (one for the new pipeline and another one for the TMV pipeline). The incoming new instructions will be put to the low priority queue.

5.2.5 fpu_out

This module manages the output from the FPU. It implements a round robin queue to manage more than one FPU pipeline providing output at the same cycle. The module needs to be changed to allow the new requirements of the output data path module and the output control unit module.

5.2.6 fpu_out_ctl

This is the control unit of the fpu_out module. It has two output queues, one for divisions, which has the highest priority, and another one for all the other pipelines. A reimplementation of the low priority round robin output queue was needed to allow the new pipeline to output results.

The pipeline added by the platform framework uses the low priority queue, leaving the high priority one for the division.

5.2.7 fpu_out_dp

This is the data path of the fpu_out module. A change is needed to allow an extra bit to be output informing that the result being output comes from the new pipeline.

5.2.8 fpu_rptr_groups

This module buffers the input to the execution pipelines. New buffers are created for the new pipeline.

5.2.9 fpu_new

This is the top-level module of the new pipeline. It contains the control unit and the data path sub-modules for the new pipeline. This module is automatically generated based on the user input.

5.2.10 fpu_new_ctl

The control unit of the new pipeline has flip-flops to separate the stages. It also generates all the control signals to the *fpu_in* module inform about the state of the pipeline. It has been implemented trying to mimic the implementation of the control units of other pipelines.

This module is generated ready to use if the TRF is not required. However, to use the TRF a control signal has to be set controlling which register needs to be read in each stage.

Control Unit interface

In the control unit module (*fpu_new_ctl.v*) the designer may define the accesses to the TRF needed by the pipeline. The result is provided by the TRF on the next cycle and delivered to the data path module (*fpu_new_dp.v*).

To use the TRF, the signal *reg_r* must be set to the register number we want to read. A different register may be read at each stage, so different inputs may be multiplexed to read different registers depending on the stage the pipeline is at.

The following example in Verilog code will request access to register 0 at stage 1, register 1 at stage 2, and register 2 at stage 3. The values will be provided to the data path in the same cycle.

```
assign reg_r =          z1stg_vld ? 0 :
                        z2stg_vld ? 1 :
                        z3stg_vld ? 2 :
                        5'bzzzzz;
```

To multiplex the register number based on the current stage of the instruction, "valid" signals must be used. These signals are asserted when there is a valid instruction being executed in that stage of the pipeline. Valid signals are named as follows: *{I}{N}stg_vld*, being I the one letter identifier given to the pipeline (*z* in the example, see section 5.1 for details), and N the number of the stage (*1* to *3* in the example). *vld* signals mean that a valid instruction is in the N stage; but if the pipeline implements more than one instruction, *{I}{N}stg_{ins}* can be used, where *ins* is the identifier of the instruction. When the *{I}{N}stg_{ins}* is one, an *ins* instruction is being executed in pipeline *{I}* at stage *{N}*.

In figure 5.1 a block diagram of the control unit signals is presented.

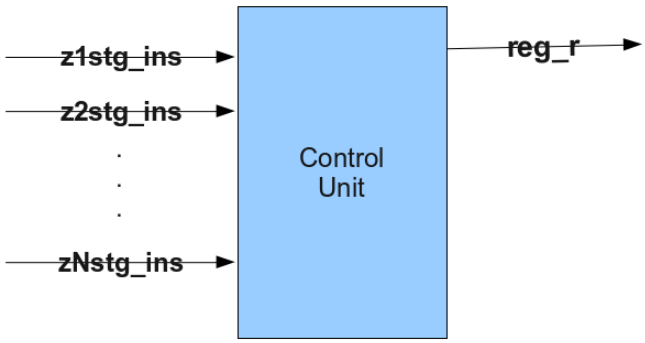


Figure 5.1: Control unit for the *zzz* pipeline identified by *z*, with N stages and implementing the *ins* instruction.

5.2.11 fpu_new_dp

The data path of the new pipeline propagates the input parameters of the module (the operands) and the partial results through the stages. All the logic required to execute the new instructions is supposed to be described in this module. The automatically generated RTL source code for this module is designed to ease the separation between stages and to allow full combinational implementation.

Data path interface

All the logic implementing the functional unit required by the new pipeline is intended to go in the data path module (fpu_new_dp.v). This module separates in stages every usable signal. The following is a list of usable signals:

- {I}{N}stg_in1: input operand 1 at the stage *N* of the pipeline with identifier *I* .
- {I}{N}stg_in2: input operand 2 at the stage *N* of the *I* pipeline.
- {I}{N}stg_partial: input partial result at the stage *N* of the *I* pipeline. Generated by the previous stage. N>1.
- {I}{N}stg_partial_out: output partial result at the stage *N* of the *I* pipeline. Generated for the next stage.
- rf_in: value read from the TRF. Requested by the control unit using the reg_r signal.
- {NME}_out: output signal where the result of the pipeline named *{NME}* is stored at the last stage.

- {I}{N}stg_{ins}: an instruction *ins* is being executed at the stage *N* of the pipeline with identifier *I* . *{NME}* represents the same pipeline as *{I}*; it is a convention in OpenSPARC to use 1 letter or 3 letter identifier depending on the length of the signal name.

In the following example Verilog code, the *zzz* pipeline makes the following calculation partitioned in 3 stages: output = 2*operand1 + operand2 + 1

```
assign z1stg_partial_out = z1stg_in1 + z1stg_in1;
assign z2stg_partial_out = z2stg_partial + z1stg_in2;
assign zzz_out = z3stg_partial + 1;
```

Figure 5.2 shows the signals available for a given *N* stage.

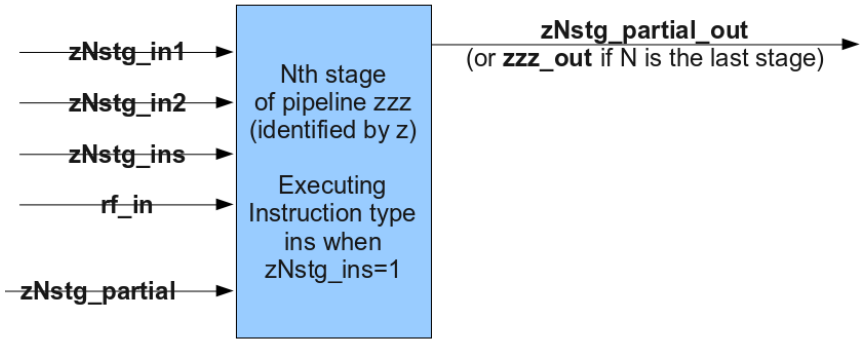


Figure 5.2: Stage N of the datapath of the *zzz* pipeline identified by *z*, implementing the *ins* instruction.

Signals {I}{N}stg_partial_out and {NME}_out are always the left part of *assigns*, while signals {I}{N}stg_in1, {I}{N}stg_in2, {I}{N}stg_partial, and rf_in can only be used as the right parts.

5.2.12 fpu_tmv

A new pipeline added in the FPU which handles TMV type instructions. These instructions read values from the FRF and write them in the TRF. There are 3 instructions of this type, which are explained in detail in section 4.2.

5.2.13 fpu_rf

This module implements the TRF. It allows two values to be written and one value to be read per cycle. As the new pipeline may use the TRF once in every stage, no more than one instruction which use the TRF is allowed to be executed at the same time.

The TRF has eight 64-bit registers, but could be easily extended (at the cost of area).

The TRF implements an index to support the TMV2 instruction. This index is internal to the TRF and is increased by 2 each time a TMV2 instruction is executed, allowing multiple TMV2 instruction to store consecutive values. The TMVR instruction resets this index.

5.3 Encoding new instructions

The platform framework modifies the floating point decode to be able to execute new instructions. Figure 5.3 shows the used opcodes for the original OpenSPARC.

		opf{3:0}							
	opf{8:4}	0	1	2	3	4	5	6	7	8–F
	00_{16}	—	FMOVs (fcc0)	FMOVd (fcc0)	FMOVq (fcc0)	—	† ‡	† ‡	† ‡	—
tmv	01_{16}	—	—	—	—	—	—	—	—	—
	02_{16}	—	—	—	—	—	FMOVRsZ ‡	FMOVRdZ ‡	FMOVRqZ ‡	—
new	03_{16}	—	—	—	—	—	—	—	—	—
	04_{16}	—	FMOVs (fcc1)	FMOVd (fcc1)	FMOVq (fcc1)	—	FMOVRsLEZ ‡	FMOVRdLEZ ‡	FMOVRqLEZ ‡	—
	05_{16}	—	FCMPs	FCMPd	FCMPq	—	FCMPEs ‡	FCMPEd ‡	FCMPEq ‡	—
	06_{16}	—	—	—	—	—	FMOVRsLZ ‡	FMOVRdLZ ‡	FMOVRqLZ ‡	—
	07_{16}	—	—	—	—	—	—	—	—	—
	08_{16}	—	FMOVs (fcc2)	FMOVd (fcc2)	FMOVq (fcc2)	—	†	†	†	—
	09_{16}	—	—	—	—	—	—	—	—	—
	$0A_{16}$	—	—	—	—	—	FMOVRsNZ ‡	FMOVRdNZ ‡	FMOVRqNZ ‡	—
	$0B_{16}$	—	—	—	—	—	—	—	—	—
	$0C_{16}$	—	FMOVs (fcc3)	FMOVd (fcc3)	FMOVq (fcc3)	—	FMOVRsGZ ‡	FMOVRdGZ ‡	FMOVRqGZ ‡	—
	$0D_{16}$	—	—	—	—	—	—	—	—	—
	$0E_{16}$	—	—	—	—	—	FMOVRsGEZ ‡	FMOVRdGEZ ‡	FMOVRqGEZ ‡	—
	$0F_{16}$	—	—	—	—	—	—	—	—	—
	10_{16}	—	FMOVs (icc)	FMOVd (icc)	FMOVq (icc)	—	—	—	—	—
	11_{16}–17_{16}	—	—	—	—	—	—	—	—	—
	18_{16}	—	FMOVs (xcc)	FMOVd (xcc)	FMOVq (xcc)	—	—	—	—	—
	19_{16}–$1F_{16}$	—	—	—	—	—	—	—	—	—

Figure 5.3: Floating point opcodes (opf) for floating point instructions (op=10b) with op3=34h [1].

The 32-bit instruction encoding follows:

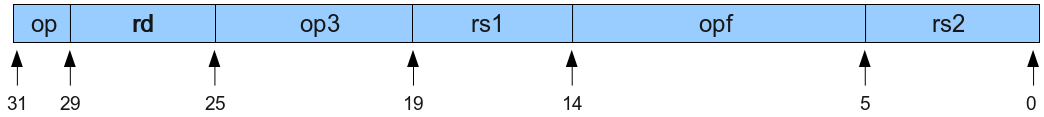


Figure 5.4: Encoding of new instructions.

The following table explains the meaning of each field in figure 5.4:

op:	This is the opcode. For floating point it is always 10.
rd:	Destination register. It is written to the FRF.
op3:	Instruction format. Must be set to 0x34.
rs1:	Source register 1, read from the FRF.
opf:	Floating point sub-opcode. Determines which FP instruction is executed.
rs2:	Source register 2, read from the FRF.

Table 5.1: Instruction encoding field meaning.

TMV instructions use the opf[8:4]=1 row, opf[3:0]=1,2,3 columns. The encoding for

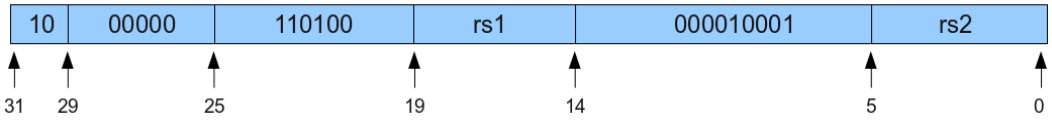


Figure 5.5: TMV1 instruction encoding.

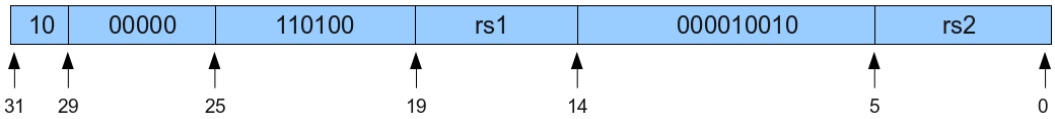


Figure 5.6: TMV2 instruction encoding.

these instructions is shown in figure 5.5, figure 5.6 and 5.7.

The instructions in the new pipeline use the opf[8:4]=3 row, and consecutive numbers for opf[3:0], starting with 1. The encoding is similar to the TMV instructions, but changing the opf from 0000100XX to 000110YYY and assigning a non-zero value to rd. The encoding of a new instruction is presented below:

In future work, the generation of code using these new instructions will be automated based on profiling information of a target program that may be integrated in a compiler.

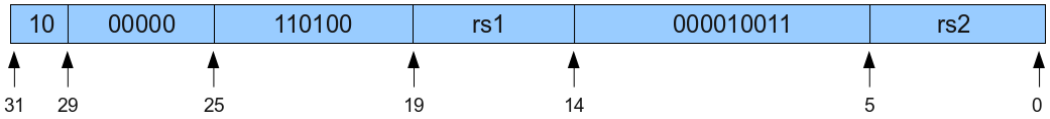


Figure 5.7: TMVR instruction encoding.

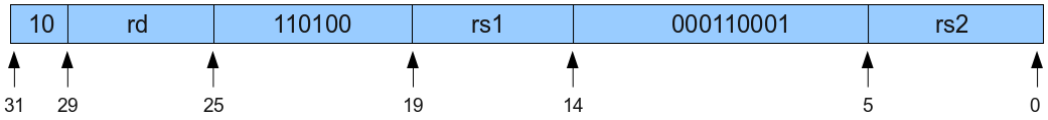


Figure 5.8: Encoding for a new instruction.

Chapter 6

ISA Extension Platform Support Analysis

The resulting modified OpenSPARC hardware has been analyzed and compared to the original OpenSPARC hardware. The original hardware has been used as the baseline in terms of area consumption, cycle time, and performance and overhead analysis.

The new designs have been simulated and then partially placed on the FPGA. The OpenSPARC placed on the FPGA has been loaded with an Ubuntu image, and booted to test correctness of the design. Then test programs have been run on the FPGA and their real execution time measured with the *time* command to compare the performance between the baseline and the new hardware; however, due to restrictions described in section 6.4, these executions have not been used to collect data for this analysis. Instead, the Verilog code of each modified OpenSPARC has been simulated to run the modified programs.

The following tests have been conducted:

- ISA Extension Platform support overhead test: a hardware has been generated with support for a functional unit of up to 50 stages to see how that affects resource utilization and cycle time. The goal of this test is to know the overhead caused by the platform. In addition we also check that the modified RTL source code is synthesizable.
- Program performance test: different algorithms have been analyzed to measure the performance impact of the overhead of the FPU coprocessor execution and the potential performance improvements. A new functional unit has been added to the hardware for each algorithm analyzed to allow the execution of a new instruction which should improve program performance. The algorithms analyzed are:
  - FIR algorithm: The algorithm is executed in the baseline hardware and in a modified hardware that extends the ISA with some new instructions.
  - EDGE algorithm: The algorithm is executed in the baseline hardware and in a modified hardware that extends the ISA with some new instructions.
  - Stencil 3D algorithm: The algorithm is executed in the baseline hardware and in a modified hardware that extends the ISA with some new instructions.

6.1 Experimental setup

Each test program has been compiled using the sparc-linux-gcc cross-compiler in a x86 machine, with options -O2 and -mcpu=v9. No optimizations have been manually applied to the resulting code. With those flags, the FPU is used by default to run floating point operations.

The synthesis tool used was the rxil tool included in the OpenSPARC package, which uses Xilinx *xst* and *netgen* applications, and the target FPGA is a Xilinx Virtex 5 (5vlx110tff1136-1). Xilinx XST version 11.1 has been used for synthesis.

The simulation platform used was *sims*, included in the OpenSPARC package. It simulates the processor described in HDL at a frequency of 1200MHz.

We define as baseline hardware the original OpenSPARC T1, as opposed to the ISA extended version. We define as baseline program or executable the original program containing only Sparc v9 instructions, as opposed to the program using the ISA extensions provided by the modified hardware.

6.2 ISA Extension Platform support overhead

The inclusion of the elements described in chapter 5 to allow new functional units to be added in the FPU has no impact in the cycle time (the critical path is still the same it was before the changes) and has very little impact on the resource utilization of the FPGA. To reach these conclusions, a test pipeline was generated with a very deep pipeline (50 stages). The design was synthesized and the results compared to the baseline unmodified OpenSPARC. The differences in the synthesis are presented below. The results have been divided in three modules, as they are synthesized separately: sparc, fpu, and ccx. For each module, results are presented divided by report. Only the differences between the baseline and the modified processor are shown. Depending on the module, the *HDL synthesis report*, the *Advanced HDL synthesis report*, the *Low Level Synthesis*, and the *Final Report* are presented. For more detail about the full report please refer to [12].

A summary of the results in the following tables is presented in subsection 6.2.5.

6.2.1 Sparc module reports

In this section the results for the sparc module are presented. Table 6.1 shows the reports of the resource utilization output by the synthesis of the baseline and the modified version. In the synthesis rows, specific resource utilization is shown, while the final report shows LUT[1] and flip-flop usage, which are BELS[2] (real resources of the FPGA). Table 6.2 shows a comparison of the FPGA slice logic utilization for the baseline and the modified OpenSPARC. Table 6.3 shows the timing constraints found for the baseline and modified OpenSPARC. Only the timing summary is shown.

[1]Look-Up Tables. The number following the acronym specifies the number of inputs of the LUT.
[2]Basic Element of Logic, pieces of functional elements that make up a component.

Report	Type or Module	Resource	Baseline	New	Diff
HDL Synthesis	sparc_ifu	1-bit register	1681	1682	+1
Advanced HDL synth.	Registers	Flip-flops	36606	36607	+1
Final Report	BELS	LUT2	898	899	+1
		LUT4	5116	5117	+1
		LUT5	15138	15137	-1
		LUT6	24490	24498	+8
		MUXF7	2536	2535	-1
	FF/Latches	FD	7247	7246	-1
		FDR	1385	1387	+2
		FDRS	126	125	-1

Table 6.1: Sparc module baseline and new resource utilization. Only the differences are shown.

Element	Baseline usage	(%)	New usage	(%)
Number of Slice Registers	31693 out of 69120	45%	no change	
Number of Slice LUTs	50594 out of 69120	73%	50603	73%
Number used as Logic	48677 out of 69120	70%	48686	70%
Number used as Memory	1917 out of 17920	10%	no change	
Number used as RAM	1384		no change	
Number used as SRL	533		no change	

Table 6.2: Slice logic utilization for the Sparc module.

	Baseline	New
Min. period	**11.882ns**	
	(Max. Freq.: 84.161MHz)	no change
Min. input arrival time before clock	7.721ns	no change
Max. output required time after clock	0.471ns	no change
Max. combinational path delay	No path found	no change

Table 6.3: Timing summary for the Sparc module.

6.2.2 FPU module reports

In this section the results for the fpu module are presented. Table 6.4 shows the reports of the resource utilization output by the synthesis of the baseline and the modified version. It is distributed the same way the table for the sparc module is, differentiating between the baseline and the modified, and the synthesis and the final report rows. Table 6.5 shows a comparison of the FPGA slice logic utilization for the baseline and the modified OpenSPARC for the fpu module. Table 6.6 shows the timing constraints found for the baseline and modified OpenSPARC. Only the timing summary is shown. The *fpu* module is faster than the *sparc*.

Report	Module	Resource	Baseline	New	Diff
HDL synthesis	fpu	64-bit adder	4	5	+1
	fpu	1-bit register	198	204	+6
	fpu	3-bit register	25	9	-16
	fpu	4-bit register	19	20	+1
	fpu	5-bit register	17	84	+67
	fpu	8-bit register	28	78	+50
	fpu	63-bit register	1	3	+2
	fpu	64-bit register	17	76	+59
Advanced HDL synth.	fpu	Flip-flops	4241	8940	+4699
Low Level synthesis	fpu(Registers)	Flip-flops	3522	4326	+804
	fpu(Shift regs)	2-bit shift reg.	43	53	+10
	fpu(Shift regs)	5-bit shift reg.	4	58	+54
	fpu(Shift regs)	6-bit shift reg.	0	172	+172
	fpu(Shift regs)	7-bit shift reg.	8	210	+202
	fpu(Shift regs)	8-bit shift reg.	0	20	+20
Final Report	fpu(BELS)	LUT2	497	587	+90
		LUT3	453	446	+13
		LUT4	763	745	-18
		LUT5	1090	1113	+23
		LUT6	2869	2964	+95
		MUXCY	708	771	+63
		MUXF7	246	252	+6
		XORCY	528	592	+64
	fpu(FF/Latches)	FDE	2135	2961	+826
		FDRE	190	625	+435
		FDS	152	153	+1
	fpu(Shift regs)	SRLC32E	0	528	+528

Table 6.4: FPU module baseline and new resource utilization. Only the differences are shown.

In table 6.5 it can be seen that the amount of LUTs used as memory has noticeably increased. This is because of the addition of the 50 stage registers.

Element	Baseline usage	(%)	New usage	(%)
Number of Slice Registers	3604 out of 69120	5%	4866	7%
Number of Slice LUTs	6148 out of 69120	8%	6789	9%
Number used as Logic	5768 out of 69120	8%	5951	8%
Number used as Memory	380 out of 17920	2%	838	4%
Number used as RAM	310		no change	
Number used as SRL	70		528	

Table 6.5: Slice logic utilization for the fpu module.

	Baseline	New
Min. period	**9.550ns**	
	(Max. Freq.: 104.712MHz)	no change
Min. input arrival time before clock	3.212ns	3.217ns
Max. output required time after clock	0.471ns	no change
Max. combinational path delay	No path found	no change

Table 6.6: Timing summary for the fpu module.

6.2.3 CCX module reports

The CCX module has been unmodified, and the synthesis results for the baseline and modified OpenSPARC are exactly the same. For completeness, the slice logic utilization (table 6.7) and the timing summary (table 6.8) are presented.

Element	Usage	(%)
Number of Slice Registers	29184 out of 69120	42%
Number of Slice LUTs	24147 out of 69120	34%
Number used as Logic	24147 out of 69120	34%

Table 6.7: Slice logic utilization for the ccx module.

6.2.4 Overhead of hardware modifications on Sparc v9-compliant programs

The original executables have been run on the modified and original versions of the OpenSPARC to check if the modification of the hardware causes any interference to the execution of original Sparc v9 instructions which do not use the modified hardware. After several simulations it could be observed that there is not any difference in terms of execution time between running the original executables in the baseline OpenSPARC and running them in the modified OpenSPARC.

6.2.5 Summary on the platform overhead

The overhead caused by the addition of the new pipeline, the TMV pipeline, the TRF and all the changes made to accommodate these additions is small in both terms of area consumption and cycle time. Even with the addition of a very deep pipeline (50 stages)

Min. period	**6.367ns (Max. Freq.: 157.060MHz)**
Min. input arrival time before clock	1.147ns
Max. output required time after clock	1.045ns
Max. combinational path delay	No path found

Table 6.8: Timing summary for the ccx module.

the only resource that started to be more used was flip-flops. That makes sense, because every stage added to a pipeline requires flip-flops to maintain partial results and to make input operands available at any stage. The changes are bigger in the fpu module (the module that contains the new pipeline), and there is not any impact on the ccx module. The sparc module has very few changes in the resource utilization.

In any case, even with the increased number of flip-flops used in the new design, the amount of used resources compared to the total amount of available resources is small, and in the other hand, this deep pipeline gives flexibility on the type of functional unit or instruction to be implemented. For instance, in table 6.5 we can see that the number of slice registers went from 3604 to 4866 (+35% usage), and the amount of slice LUTs that were used as Memory went from 380 to 838 (+120%). This might look as a dramatic increase of resource consumption, but compared to the total amount of available resources it is small: the number of available slice registers is 69120, so the baseline usage was 5% and the new usage is 7%, while the amount of slice LUTs usable as memory is 17920, so the baseline usage was 2% and the new usage is 4%.

We can also see that new resources are used when adding our modules. For instance, in table 6.4 we can see some new shift registers being used. But again, compared to the amount of available resources, this is a small increase. In table 6.5 we see that these shift registers are using Slice LUTs used as Memory and used as SRL. This resource went from 70 to 528 (+654%), but it is still a very small portion of the total available resources (17920): it was a 0.39% for the baseline, and a 2.95% for the 50 pipeline processor. This big use of shift registers is because there is not any logic between stage registers, so each stage the value of the stage registers is in fact shifted. As we will present later in this work, when logic is added between stage registers, no additional SRLs are needed.

As for the timing, every module maintained the cycle time it had in the baseline, as can be seen in the first row of table 6.3, table 6.6, and table 6.8. This is because the added pipeline is empty, so there is not any logic that could make the cycle time slower. The only exception is the TRF, which implements an eight-way multiplexer among other logic; but it has not caused the cycle time to increase. The *sparc* module is the slowest one, so it constrains the maximum frequency for the global design.

It is safe to assume that the platform causes a small overhead in terms of area consumption and no overhead at all in terms of timing constraints.

6.3 Application performance analysis

Three algorithms are analyzed in this section: FIR, EDGE and Stencil3D. For each one of them, a baseline executable is generated without any modification, and it is simulated

on the baseline hardware. Then, using the implemented platform support explained in chapter 5, a new instruction is added to the ISA and the assembly code of the original program is altered to use the new instruction. The execution of both programs is simulated using the sims tool provided in the OpenSPARC package, and the execution time of the programs is compared. Different hardware variations were tested when possible.

The simulator adds some overhead to the execution time of the analyzed algorithms. This is because a portion of code intended to boot the processor is simulated before starting the relevant simulation. This overhead as been measured and determined to be of 13212 cycles, and every result presented in this chapter has been accordingly corrected.

6.3.1 Application versions

FIR

The FIR algorithm is shown in figure 6.1. In figure 6.2 the assembly code for the loop body generated by the cross compiler is shown. As can be seen, the compiler detected data reuse for one element, so only 4 loads are made per iteration (instead of 5). Code 6.3 shows the modified assembly code to use the new instructions.

The original code generated by the compiler is taken as the base line, and no further optimizations are made, neither in the code using the standard Sparc ISA, nor in the extended ISA.

```
int A[1000];
int B[1000];                                                                    2
int main ()
{
        int i;
        for(i = 0 ; i < 996 ; i = i + 1)
        {                                                                       7
                B[i] = A[i]* 3 +
                       A[i+1]* 5 +
                       A[i+3]* 7 +
                       A[i+2]* 9 +
                       A[i+4]* 11;                                              12

        }
}
```

Figure 6.1: Original fir.c code.

- **4 stage FIR**

  The new instruction has been implemented in a 4 stage pipeline which uses the TRF.

  The most notorious change is the number of executed instructions per loop iteration. In the original code, 22 instructions are executed while in the new adapted code only 12 are needed to do the same work. The new instruction takes care of all multiplies and adds, so only load/store and TMV instructions are needed in addition to the FIR instruction. Code 6.4 shows the changes added to the data path file of the new pipeline (fpu_fir_dp.v), and figure 6.5 shows the changes to the control unit file (fpu_fir_ctl.v). Blank rows separate signals belonging to different stages.

  The addition of the hardware needed to run the new instruction does not modify the cycle time of the processor, according to the synthesis reports.

```
ld      [%o2+4], %g2
ld      [%o2+8], %o4
add     %o3, %o3, %o5
ld      [%o2+12], %g4
add     %o5, %o3, %o5
sll     %g2, 3, %g3
ld      [%o2], %o3
add     %g3, %g2, %g3
smul    %g4, 11, %g4
sll     %o3, 2, %g1
sll     %o4, 3, %g2
add     %g1, %o3, %g1
sub     %g2, %o4, %g2
add     %g3, %g1, %g3
add     %g3, %g2, %g3
add     %g3, %g4, %g3
add     %g3, %o5, %g3
st      %g3, [%o1+%o0]
add     %o1, 4, %o1
cmp     %o1, 3984
bne,pt  %icc, .LL2
 add    %o2, 4, %o2
```

Figure 6.2: Compiler generated assembly code for the FIR algorithm.

```
        ld      [%o2+4], %f2
        ld      [%o2+8], %f3
        .word   0x81A04262      /*tmvr  %f1, %f2*/
        ld      [%o2+12], %f4
        .word   0x81A0C243      /*tmv2  %f3, %f3*/
        ld      [%o2], %f1
        .word   0x8BA04624      /*ffir  %f1, %f4, %f5*/
        st      %f5, [%o1+%o0]
        add     %o1, 4, %o1
        cmp     %o1, 3984
        bne,pt  %icc, .LL2
         add    %o2, 4, %o2
```

Figure 6.3: Modified assembly code for the FIR algorithm.

```
assign f1stg_partial_out[63:32] = (f1stg_in1 * 5) + (f1stg_in2 * 11);
assign f1stg_partial_out[31:0]  = rf_in * 3;

assign f2stg_partial_out[63:32] = f2stg_partial[63:32] + f2stg_partial[31:0];
assign f2stg_partial_out[31:0]  = rf_in * 9;

assign f3stg_partial_out[63:32] = f3stg_partial[63:32] + f3stg_partial[31:0];
assign f3stg_partial_out[31:0]  = rf_in * 7;

assign fir_out = f4stg_partial[63:32] + f4stg_partial[31:0];
```

Figure 6.4: 4-stage FIR pipeline data path modifications. The rest of the module was automatically generated.

```
assign reg_r =  f1stg_vld ? 0 :
                f2stg_vld ? 1 :
                f3stg_vld ? 2 :
                5'bzzzzz;
```

Figure 6.5: FIR pipeline control unit modifications. The rest of the module was automatically generated.

- **3 stage FIR**

  A modification of the hardware where the new FIR pipeline was reduced to 3 stages (from 4) was also tested. Only the datapath was modified, the control unit remained as shown in figure 6.5. The new datapath is presented in figure 6.6. The pipeline can not be reduced any further because three registers need to be read from the TRF.

```
assign f1stg_partial_out[63:32] = (f1stg_in1 * 5) + (f1stg_in2 * 11);
assign f1stg_partial_out[31:0]  = rf_in * 3;

assign f2stg_partial_out[63:32] = f2stg_partial[63:32] + f2stg_partial[31:0];
assign f2stg_partial_out[31:0]  = rf_in * 9;

assign fir_out  = f3stg_partial[63:32] + f3stg_partial[31:0] + (rf_in * 7);
```

Figure 6.6: 3-stage FIR pipeline data path modifications.

EDGE

The same methodology followed to test the FIR algorithm (section 6.3.1) has been applied to the Edge algorithm. Its original source code is shown in figure 6.7, and the assembly code generated by the compiler is shown in figure 6.8.

```
int P[30][30];
int B[30][30];
void main() {
        int i,j,MASKv,MASKh;

        for(i = 1; i <= 28; i = i + 1) {
                for(j = 1; j <= 28 ; j = j + 1) {
                        MASKv =
                                (P[i-1][j+1] - P[i-1][j-1]) +
                                (P[i][j+1] - P[i][j-1]) +
                                (P[i+1][j+1] - P[i+1][j-1]);
                        MASKh =
                                (P[i+1][j-1] - P[i-1][j-1]) +
                                (P[i+1][j] - P[i-1][j]) +
                                (P[i+1][j+1] - P[i-1][j+1]);
                        B[i][j] = (MASKv*MASKv + MASKh*MASKh);
                }
        }
}
```

Figure 6.7: Original edge.c code.

- **10 stage EDGE**

  The modified assembly code is shown in figure 6.9. It has 20 instructions in the inner loop, versus the 29 instructions that the original code had.

  Code 6.10 and 6.11 show the hardware description of the new pipeline of a 10 stage EDGE. It is much more complex than the one needed for the FIR algorithm, since a lot of data is reused. Because of that reuse, a lot of partial calculations must be saved to be used later in the pipeline, so new flip-flops have to be created to hold the values until they are used again. New wire declarations and new flip-flops holding the values of these new wires have been omitted from this code. The complete code is shown in appendix A. New wires have 32 bit width.

```
.LL2:                                                                   1
        smul    %g3, 2000, %g2
        add     %g2, 4, %g1
        add     %g1, %o1, %o5
        add     %g2, -1992, %g1
        add     %g3, 1, %o4                                             6
        add     %g1, %o2, %i0
        smul    %o4, 2000, %g1
        add     %g1, 8, %g1
        add     %g2, 8, %g2
        add     %g1, %o2, %i2                                           11
        add     %g2, %o2, %i1
        ld      [%o3], %g3
        ld      [%o3-4000], %g1
        ba,pt   %xcc, .LL4
         mov    1, %o7                                                  16
.LL3:
        mov     %i4, %g3
        mov     %i3, %g1
.LL4:
        ld      [%i0], %i3                                              21
        ld      [%i2], %i4
        ld      [%i0-8], %g4
        ld      [%i2-8], %i5
        add     %i5, %i4, %g2
        sub     %g2, %i3, %g2                                           26
        sub     %g2, %g4, %g2
        add     %g2, %g3, %g2
        ld      [%i1-8], %g3
        sub     %g2, %g1, %g2
        ld      [%i1], %g1                                              31
        smul    %g2, %g2, %g2
        add     %i3, %g1, %g1
        add     %g1, %i4, %g1
        sub     %g1, %g4, %g1
        sub     %g1, %g3, %g1                                           36
        sub     %g1, %i5, %g1
        smul    %g1, %g1, %g1
        add     %g2, %g1, %g2
        st      %g2, [%o5]
        add     %o7, 1, %o7                                             41
        add     %i0, 4, %i0
        add     %i1, 4, %i1
        add     %i2, 4, %i2
        cmp     %o7, 499
        bne,pt  %icc, .LL3                                              46
         add    %o5, 4, %o5
        cmp     %o4, 499
        be,pn   %icc, .LL10
         add    %o3, 2000, %o3
        ba,pt   %xcc, .LL2                                              51
```

Figure 6.8: Compiler generated assembly code for the EDGE algorithm. Loop body.

```
.LL2:
        smul    %g3, 2000, %g2
        add     %g2, 4, %g1
        add     %g1, %o1, %o5                                                    4
        add     %g2, -1992, %g1
        add     %g3, 1, %o4
        add     %g1, %o2, %i0
        smul    %o4, 2000, %g1
        add     %g1, 8, %g1                                                      9
        add     %g2, 8, %g2
        add     %g1, %o2, %i2
        add     %g2, %o2, %i1
        ld      [%o3], %f3
        ld      [%o3-4000], %f1                                                 14
        ba,pt   %xcc, .LL4
         mov    1, %o7
.LL3:
        fmovs   %f7, %f3
        fmovs   %f6, %f1                                                        19
.LL4:
        ld      [%i0], %f6
        ld      [%i2], %f7
        ld      [%i0-8], %f4
        ld      [%i2-8], %f5                                                    24
        .word   0x81A14264      /*tmvr  %f5, %f4*/
        .word   0x81A0C241      /*tmv2  %f3, %f1*/
        .word   0x81A18247      /*tmv2  %f7, %f6*/
        ld      [%i1-8], %f3
        ld      [%i1], %f1                                                      29
        .word   0x85A04623      /*edge  %f1, %f3, %f2*/
        st      %f2, [%o5]
        add     %o7, 1, %o7
        add     %i0, 4, %i0
        add     %i1, 4, %i1                                                     34
        add     %i2, 4, %i2
        cmp     %o7, 499
        bne,pt  %icc, .LL3
         add    %o5, 4, %o5
        cmp     %o4, 499                                                        39
        be,pn   %icc, .LL10
         add    %o3, 2000, %o3
        ba,pt   %xcc, .LL2
         mov    %o4, %g3
```

Figure 6.9: Modified assembly code for the EDGE algorithm.

- **8 stage EDGE**

  A modification of the hardware where the new Edge pipeline was reduced to 8 stages (from 10) was also tested. This modification makes use of the data from the TRF in the same cycle they are provided. Only the datapath was modified, the control unit remained as presented in figure 6.11. The new datapath is presented in figure 6.12.

  New wire declarations and new flip-flops holding the new wires' values have been omitted. For the full code of the 8 stage EDGE hardware please see figure A.5 in appendix A.

Stencil 3D

For this algorithm, a larger TRF has been used to allow all the additions done in an iteration to be done in a single instruction. The modified TRF for Stencil has 16 registers.

The original source code of the Stencil 3D algorithm is shown in the figure 6.13. The compiler generated code of the inner loop is shown in figure 6.14. The modified assembly code of the inner loop is shown in figure 6.15.

The new code has 27 instructions in its inner loop, while the original one had 32 instructions.

The Stencil 3D algorithm sums a series of 12 values. This has been implemented

```
/* Note: new wire declarations and new flip-flop instantiation is
not shown in this code. See figure's footnote */

assign e1stg_partial_out[63:32] = e1stg_in1 - e1stg_in2;
assign e1stg_partial_out[31:0] = rf_in;

assign e2stg_partial_out[63:32] = e2stg_partial[63:32] + e2stg_partial[31:0];
assign e2stg_partial_out[31:0] = rf_in;
assign e2stg_pm_out[31:0] = e2stg_pm[31:0];

assign e3stg_partial_out[63:32] = e3stg_partial[63:32] - e3stg_partial[31:0];
assign e3stg_partial_out[31:0] = rf_in;
assign e3stg_mm_out[31:0] = e3stg_mm[31:0];
assign e3stg_pm_out[31:0] = e3stg_pm[31:0];

assign e4stg_partial_out[63:32] = e4stg_partial[63:32] + e4stg_partial[31:0];
assign e4stg_partial_out[31:0] = rf_in;
assign e4stg_mm_out[31:0] = e4stg_mm[31:0];
assign e4stg_pp_out[31:0] = e4stg_pp[31:0];
assign e4stg_pm_out[31:0] = e4stg_pm[31:0];

assign e5stg_partial_out[63:32] = e5stg_partial[63:32] - e5stg_partial[31:0];
assign e5stg_partial_out[31:0] = rf_in;
assign e5stg_P1_out[31:0] = e5stg_partial[31:0] - e5stg_mm[31:0];
assign e5stg_pp_out[31:0] = e5stg_pp[31:0];
assign e5stg_pm_out[31:0] = e5stg_pm[31:0];

assign e6stg_partial_out[63:32] = e6stg_partial[63:32] * e6stg_partial[63:32];
assign e6stg_partial_out[31:0] = rf_in;
assign e6stg_P1_out[31:0] = e6stg_P1[31:0] + e6stg_partial[31:0];
assign e6stg_pp_out[31:0] = e6stg_pp[31:0];
assign e6stg_pm_out[31:0] = e6stg_pm[31:0];

assign e7stg_partial_out[63:32] = e7stg_partial[63:32];
assign e7stg_partial_out[31:0] = e7stg_P1 - e7stg_partial[31:0];
assign e7stg_P3_out[31:0] = e7stg_pp[31:0] - e7stg_pm[31:0];

assign e8stg_partial_out[63:32] = e8stg_partial[63:32];
assign e8stg_partial_out[31:0] = e8stg_partial[31:0] + e8stg_P3;

assign e9stg_partial_out[63:32] = e9stg_partial[63:32];
assign e9stg_partial_out[31:0] = e9stg_partial[31:0] * e9stg_partial[31:0];

assign edg_out[63:0] = e10stg_partial[63:32] + e10stg_partial[31:0];
```

Figure 6.10: 10-stage EDGE pipeline data path modifications. Full implementation can be found on page 55 in the appendix A.

```
assign reg_r =  e1stg_vld ? 5 :
                e2stg_vld ? 1 :
                e3stg_vld ? 4 :
                e4stg_vld ? 0 :
                e5stg_vld ? 2 :
                e6stg_vld ? 3 :
                5'bzzzzz;
```

Figure 6.11: Control unit modifications for the Edge algorithm.

```
/* Note: new wire declarations and new flip-flop instantiation is
not shown in this code. See figure's footnote */

assign e1stg_partial_out[63:32] = (e1stg_in1 - e1stg_in2) + rf_in;
assign e1stg_partial_out[31:0] = rf_in;

assign e2stg_partial_out[63:32] = e2stg_partial[63:32] - rf_in;
assign e2stg_partial_out[31:0] = rf_in;
assign e2stg_pm_out[31:0] = e2stg_pm[31:0];

assign e3stg_partial_out[63:32] = e3stg_partial[63:32] + rf_in;
assign e3stg_partial_out[31:0] = rf_in;
assign e3stg_mm_out[31:0] = e3stg_mm[31:0];
assign e3stg_pm_out[31:0] = e3stg_pm[31:0];

assign e4stg_partial_out[63:32] = e4stg_partial[63:32] - rf_in;
assign e4stg_partial_out[31:0] = rf_in;
assign e4stg_mm_out[31:0] = e4stg_mm[31:0];
assign e4stg_pp_out[31:0] = e4stg_pp[31:0];
assign e4stg_pm_out[31:0] = e4stg_pm[31:0];

assign e5stg_partial_out[63:32] = e5stg_partial[63:32] * e5stg_partial[63:32];
assign e5stg_partial_out[31:0] = rf_in;
assign e5stg_P1_out[31:0] = rf_in + e5stg_partial[31:0] - e5stg_mm[31:0];
assign e5stg_pp_out[31:0] = e5stg_pp[31:0];
assign e5stg_pm_out[31:0] = e5stg_pm[31:0];

assign e6stg_partial_out[63:32] = e6stg_partial[63:32];
assign e6stg_partial_out[31:0] = rf_in;
assign e6stg_P1_out[31:0] = e6stg_P1[31:0] - rf_in + (e6stg_pp[31:0] - e6stg_pm[31:0]);

assign e7stg_partial_out[63:32] = e7stg_partial[63:32];
assign e7stg_partial_out[31:0] = e7stg_P1[31:0] * e7stg_P1[31:0];

assign edg_out[63:0] = e8stg_partial[63:32] + e8stg_partial[31:0];
```

Figure 6.12: 8-stage EDGE pipeline data path modifications. Full implementation can be found on page 55 in the appendix A.

```
int A[10][10][10];
int B[10][10][10];
void main()
{
        int i,j,k;

        for(i = 0 ; i < 9 ; i=i+1)
        {
                for(j = 0 ; j < 9 ; j=j+1)
                {
                        for(k = 0 ; k < 6 ; k=k+1)
                        {
                                B[i][j][k] = A[i][j][k] + A[i][j][k+1] + A[i][j][k+4] +
                                        A[i][j+1][k] + A[i][j+1][k+1] + A[i][j+1][k+4] +
                                        A[i+1][j][k] + A[i+1][j][k+1] + A[i+1][j][k+4] +
                                        A[i+1][j+1][k] + A[i+1][j+1][k+1] + A[i+1][j+1][k+4];
                        }
                }
        }
}
```

Figure 6.13: Original stencil3d.c code.

```
.LL3:
        mov     %o5, %o2
        mov     %o7, %o1
        mov     %i0, %o3
        mov     %i1, %o4
.LL4:
        ld      [%g3+12], %g2
        ld      [%i4], %o5
        ld      [%g3], %o7
        ld      [%i4+12], %g1
        add     %o5, %g1, %g1
        add     %g1, %o7, %g1
        add     %g1, %g2, %g1
        ld      [%i3+12], %g2
        ld      [%i3], %i0
        ld      [%g4], %i1
        add     %g1, %i0, %g1
        add     %g1, %g2, %g1
        ld      [%g4+12], %g2
        add     %g1, %i1, %g1
        add     %g1, %g2, %g1
        add     %g1, %o2, %g1
        add     %g1, %o4, %g1
        add     %g1, %o3, %g1
        add     %g1, %o1, %g1
        st      %g1, [%i2]
        add     %i5, 1, %i5
        add     %i4, 4, %i4
        add     %g3, 4, %g3
        add     %i3, 4, %i3
        add     %g4, 4, %g4
        cmp     %i5, 6
        bne,pt  %icc, .LL3
         add    %i2, 4, %i2
```

Figure 6.14: Compiler generated assembly code for the Stencil 3D algorithm.

```
.LL3:
        fmovs   %f1, %f7
        fmovs   %f3, %f10
        fmovs   %f5, %f9
        fmovs   %f6, %f8
.LL4:
        ld      [%g3+12], %f4
        ld      [%i4], %f1
        ld      [%g3], %f3
        ld      [%i4+12], %f2
        .word   0x81A04262      /*tmvr  %f1, %f2*/
        .word   0x81A0C244      /*tmv2  %f3, %f4*/
        ld      [%i3+12], %f4
        ld      [%i3], %f5
        ld      [%g4], %f6
        .word   0x81A10245      /*tmv2  %f4, %f5*/
        ld      [%g4+12], %f4
        .word   0x81A10246      /*tmv2  %f4, %f6*/
        .word   0x81A1C248      /*tmv2  %f7, %f8*/
        .word   0x85A2462A      /*sten  %f9, %f10, %f2*/
        st      %f2, [%i2]
        add     %i5, 1, %i5
        add     %i4, 4, %i4
        add     %g3, 4, %g3
        add     %i3, 4, %i3
        add     %g4, 4, %g4
        cmp     %i5, 96
        bne,pt  %icc, .LL3
         add    %i2, 4, %i2
```

Figure 6.15: Modified assembly code for the Stencil 3D algorithm.

into hardware to allow all the sums in one iteration to be done with a single instruction. Figures 6.16 and 6.17 show the hardware implementation of the instruction. 5 TMV instructions are needed per iteration to move 10 operands; the two extra operands are brought to the FPU by the *sten* instruction itself.

```
assign s1stg_partial_out = s1stg_in1 + s1stg_in2 + rf_in;
assign s2stg_partial_out = s2stg_partial + rf_in;
assign s3stg_partial_out = s3stg_partial + rf_in;
assign s4stg_partial_out = s4stg_partial + rf_in;
assign s5stg_partial_out = s5stg_partial + rf_in;
assign s6stg_partial_out = s6stg_partial + rf_in;
assign s7stg_partial_out = s7stg_partial + rf_in;
assign s8stg_partial_out = s8stg_partial + rf_in;
assign s9stg_partial_out = s9stg_partial + rf_in;
assign ste_out = s10stg_partial + rf_in;
```

Figure 6.16: Stencil 3d pipeline data path modifications. The rest of the module was automatically generated.

```
assign reg_r =  s1stg_vld ? 0 :
                s2stg_vld ? 1 :
                s3stg_vld ? 2 :
                s4stg_vld ? 3 :
                s5stg_vld ? 4 :
                s6stg_vld ? 5 :
                s7stg_vld ? 6 :
                s8stg_vld ? 7 :
                s9stg_vld ? 8 :
                s10stg_vld ? 9 :
                5'bzzzzz;
```

Figure 6.17: Stencil 3D pipeline control unit modifications. The rest of the module was automatically generated.

6.3.2 Application specific synthesis reports

The ISA extended hardware used to run 4-stage FIR and 10-stage EDGE was synthesized and compared to the synthesis of the baseline hardware and the 50-stage hardware analyzed in section 6.2. The results are presented following:

- The cycle time did not change, except for the 3-stage FIR.
- The reports for the ccx module are the same for every modification and for the baseline, presented in table 6.7.
- The reports for the sparc module are the same for every modification compared to the 50-stage pipeline presented in table 6.2.
- The reports for the fpu module are slightly different from the baseline and the 50-stage pipeline. The results are presented in table 6.9.

The amount of registers used in FIR and EDGE hardware is half-way between the baseline and the 50 stage pipeline. It seems to be a direct proportion between the amount of slice registers used and the amount of stages. The number of Slice LUTs used is bigger for the FIR and EDGE hardware because they implement new functional units; the 50-stage pipeline is just an empty pipeline.

Element	Baseline	50stg	FIR	EDGE	Stencil3D
# of Slice Registers	3604/69120	4866	4167	4315	5560
# of Slice LUTs	6148/69120	6789	6858	6919	8352
# used as Logic	5768/69120	5951	6478	6507	7909
# used as Memory	380/17920	838	380	412	443
# used as RAM	310	310	310	310	310
# used as SRL	70	528	70	102	133

Table 6.9: Slice logic utilization for the fpu module.

The Stencil3D uses significantly more slice LUTs and slice registers than FIR and EDGE extensions. That is because Stencil3D has a 16 entry TRF, compared to the 8 entry TRF used by FIR, EDGE, and the 50 stage pipeline. The impact of adding bigger TRFs is very big.

The number of slice LUTs used as memory has increased for EDGE ans Stencil3D compared to the baseline, but it has not been increased for FIR. Still, the amount of slice LUTs used as memory on EDGE and Stencil3D is it is for the 50 stage pipeline. It is now clear that the big amount of shift registers used for the 50 stage pipeline is because of the empty pipeline, which make the stage registers act as shift registers.

6.3.3 Results

Figure 6.18 shows the speed-up for the analyzed codes. Extension 1 refers to the results for the 4 stage FIR, the 10 stage EDGE, and the 10 stage Stencil 3D. Extension 2 refers to the results for the 3 stage FIR and the 8 stage EDGE. Results show that the original code is twice faster than the modified code running on the modified OpenSPARC. In addition, the 3 stage FIR is even worse because according to the synthesis reports it increases the cycle time of the whole processor (assuming the sparc core and the FPU share a clock signal, which is in fact what happens in the unmodified OpenSPARC), making the fpu the module with the critical path instead of the sparc module. In fact, if we take into account the new cycle time, 3 stage FIR is 42% slower than the 4 stage FIR, because they take the same amount of cycles to execute and the 3 stage FIR cycle time is larger. This is not shown in speed-up graphs because we are assuming a fixed frequency.

The increase in the cycle time in the 3 stage FIR is caused by a concatenation of sums in the last stage. In table 6.10 the timing summary for the synthesis is shown. These numbers can be compared to the timing report of the FPU containing the empty 50 stage pipeline (table 6.6).

	FPU with the changes in figure 6.6
Min. period	**16.868ns (Max. Frequency: 59.285MHz)**
Min. input arrival time before clock	3.212ns
Max. output required time after clock	0.471ns
Max. combinational path delay	No path found

Table 6.10: Timing summary for the fpu module.

As mentioned, the increased cycle time is caused by the last stage of the FIR pipeline,

as shown below:

```
=========================================================================
Timing constraint: Default period analysis for Clock 'gclk'
  Clock period: 16.868ns (frequency: 59.285MHz)
  Total number of paths / destination ports: 89287028461 / 6009
-------------------------------------------------------------------------
Delay:               16.868ns (Levels of Logic = 35)
  Source:            fpu_fir/fpu_fir_ctl/i_f3stg_op/q_7 (FF)
  Destination:       fpu_out/fpu_out_dp/i_fp_cpx_data_ca_76_0/q_63 (FF)
  Source Clock:      gclk rising
  Destination Clock: gclk rising
```

Unlike the 3 stage FIR, the reduction of stages in EDGE (from 10 to 8) did not increase the cycle time of the FPU..

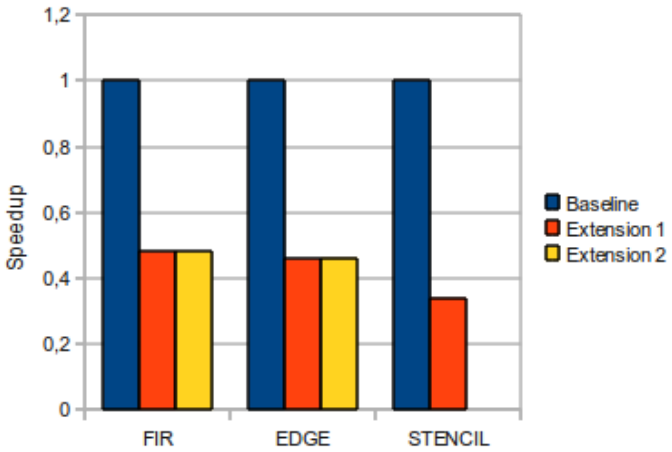


Figure 6.18: Speedup for the analyzed codes.

Knowing the simulation frequency is 1200MHz, the number of cycles needed by the programs to execute has been obtained. Using this information combined with the amount of instructions executed in each test, a CPI measurement has been calculated. Table 6.11 shows the cycles needed by each program, the number of instructions executed, and the CPI.

Program	Cycles	Instructions	CPI
Original FIR	54896	21923	2.50
4 stage modified FIR	114158	11963	9.54
3 stage modified FIR	114102	11963	9.54
Original EDGE	62029	23360	2.66
10 stage modified EDGE	135403	16305	8.30
8 stage modified EDGE	134136	16305	8.23
Original Stencil 3D	41785	17802	2.35
10 stage modified Stencil 3D	122901	15373	7.99

Table 6.11: Cycles used for each program, number of executed instructions, and CPI.

Every program using new instructions has a CPI 3 to 4 times higher than the original ones. As it was observed that the modifications do not alter the execution time for original Sparc v9 instructions, it is safe to assume that this increase in the CPI is caused by the inclusion of ISA extensions. The high CPI is caused by a high latency combined by the fact that the new instructions are executed not pipelined. Additional observation

of simulation waveforms confirm this. An extra 12 cycles latency is observed for TMV instructions, and an extra 21 cycles latency is observed for result-returning instructions. Table 6.12 shows the latencies for the analyzed ISA extensions.

Instruction	Functional unit latency	Extra latency	Total latency
All TMV instructions	1	12	13
4 stage FIR	4	21	25
3 stage FIR	3	21	24
10 stage EDGE	10	21	31
8 stage EDGE	8	21	29
10 stage Stencil 3D	10	21	31

Table 6.12: Latencies of each ISA extension tested.

Let us assume that new pipelines were inserted in the sparc core instead of the FPU to avoid the overhead of using the crossbar. In table 6.13 it is assumed that the extra latency is 0 (last column). In this case, the speed-up resulting from the assumption that the extra latency caused by accessing the FPU is 0 was calculated and is shown in figure 6.19.

Program	Sparcv9 inst.	TMV inst.	New inst.	Cycles	Cycles - latency
Original FIR	21923/21923	0	0	54896	**54896**
4 stage FIR	8975/11963	1992	996	114158	**69338**
3 stage FIR	8975/11963	1992	996	114102	**69282**
Original EDGE	23360/23360	0	0	62029	**62029**
10 stage EDGE	13169/16305	2352	784	135403	**90715**
8 stage EDGE	13169/16305	2352	784	134136	**89448**
Original Stencil 3D	17802/17802	0	0	41785	**41785**
10 stage Stencil 3D	12457/15373	2430	486	122901	**83535**

Table 6.13: Cycles of execution with FPU access latency and without latency (in bold).

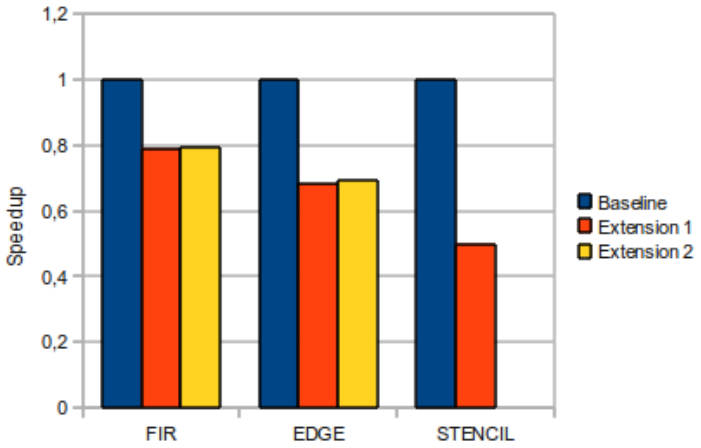


Figure 6.19: Speedup for the analyzed codes assuming no extra latency.

After removing the overhead caused by accessing the FPU, the speedup is still lower than one. Analyzing the execution with the simulator, we can see that loading and storing

values from or to a floating point register has an extremely high cost compared to loading or storing the same value to a integer register. Loading a value to a floating point register has a 9 cycle latency, while an integer load takes only 3 cycles. This is even worse for store instructions, which require 8 cycles to store a value to a floating point register, and only 1 to store it in an integer register.

Using the FPU to extend the ISA requires the source and destination operands to be in floating point registers, even if they are not floating point numbers. This causes an additional overhead which explains why the speed up in shown in figure 6.19 is lower than 1 even after removing the overhead caused by the FPU access.

To equalize the results and compare them to the baseline assuming the same cost for load and store instructions, a new set of results have been calculated. In table 6.14 the results for the execution of the analyzed ISA-extended programs are shown assuming no FPU accessing latencies and reduced latency for load (3 instead of 9 cycles) and store (1 instead of 8 cycles) instructions. The results in the table are also shown in figure 6.20.

Program	FP. Stores	FP. Load	Reduced latency cycles
Original FIR	0/21923	0	**54896**
4 stage FIR	996/11973	3986	**38450**
3 stage FIR	996/11973	3986	**38394**
Original EDGE	0/23360	0	**62029**
10 stage EDGE	784/16305	4761	**56661**
8 stage EDGE	784/16305	4761	**55394**
Original Stencil 3D	0/17802	0	**41785**
10 stage Stencil 3D	486/15373	4213	**54855**

Table 6.14: Cycles of execution without FPU access latency and assuming integer load-/store latency.

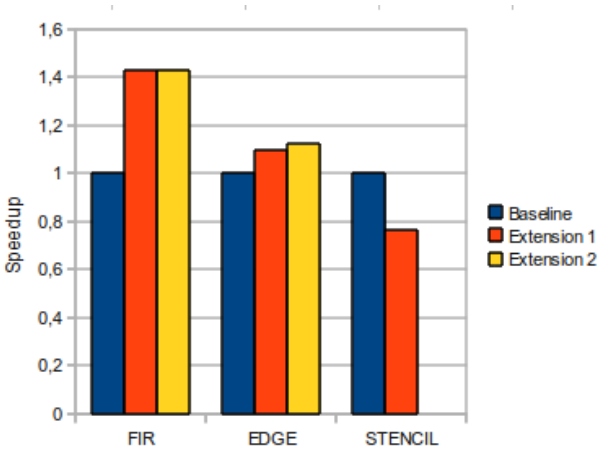


Figure 6.20: Speedup for the analyzed codes assuming integer latency for load and store and no extra latency for accessing the FPU.

With the suppression of the floating point load/store overhead, the speedups go from less than one to up to 1.43x, depending on the algorithm. The best speed-up is for the

FIR algorithm, because the loop body instruction count is greatly decreased (22 down to 12) and also the *fir* instruction itself has a very low latency (4 cycles). In the other hand, Stencil 3D shows the worst performance compared to the baseline (0.76x) because it does not allow to reduce as many instructions from the innermost loop (32 down to 27), and also the *sten* instruction has a bigger latency (10 cycles). EDGE has a big latency too, but the new instruction allows a bigger reduction to the loop body (29 down to 20). This makes possible a speedup of 1.09x for the 10 stage EDGE, and a 1.12x speedup for the 8 stage EDGE.

6.4 FPGA implementation

The modified OpenSPARC works on an FPGA, as it can boot an Ubuntu Linux operating system. All the hardware placed on the FPGA functioned normally. However, we were unable to run programs which use the ISA extensions because the FPGA design provided in the OpenSPARC package emulates the FPU using the Microblaze firmware instead of placing the FPU in the FPGA.

This partial FPGA implementation has helped anyways to check that the modified OpenSPARC works, even without being able to verify FPU function in real hardware.

In figure 6.21 we can see a block diagram of the FPGA implementation.

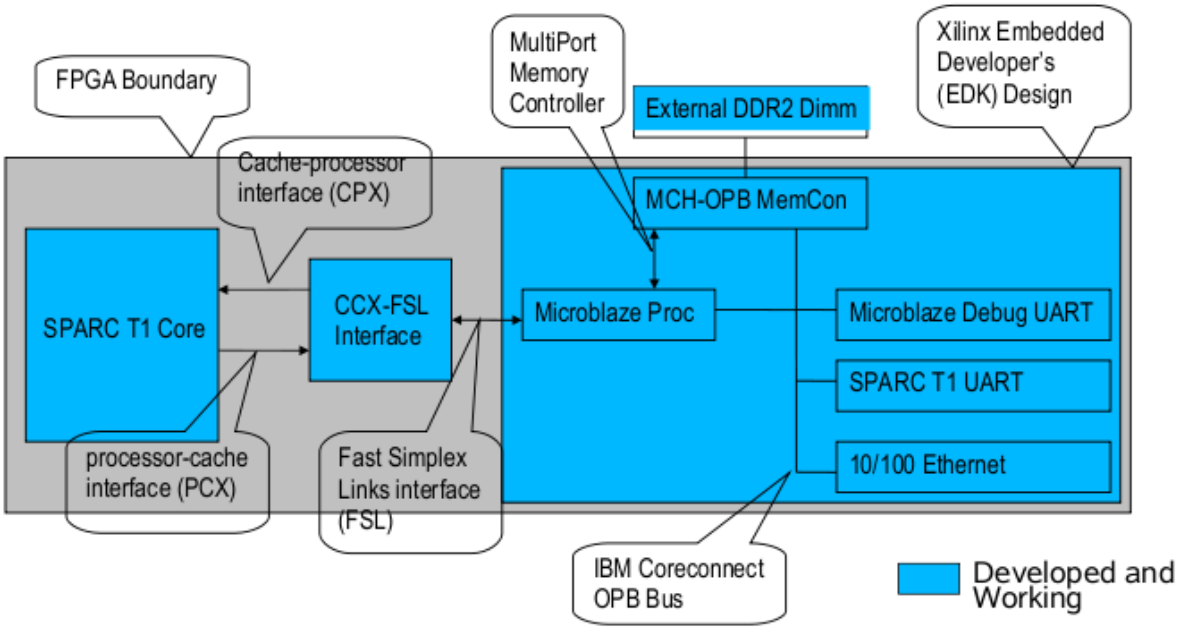


Figure 6.21: OpenSPARC T1 FPGA implementation block diagram [5].

Chapter 7

Conclusions

In this work a platform framework to allow ISA extensions on an OpenSPARC has been developed and analyzed. The main achievements of this work are:

- Automation: the platform framework generates a new hardware after the user answers a few questions. This allows the researcher to test a lot more possibilities without needing a lot of knowledge about the microarchitecture of the OpenSPARC and its source code organization.
- Flexibility: the generated pipelines have an arbitrary length, and new instructions extending the ISA may use up to 18 source operands, allowing a wider number of possibilities for the researcher.
- Reliability: the work done does not rely a hand made software simulator emulating the hardware.
- Exactitude: the generated hardware is RTL Verilog; it can be simulated with a lot of precision.
- Synthesizable: the platform framework generates code that can be synthesized. Information such as estimations of the cycle time and area cost of the modified OpenSPARC can be obtained from the synthesis.
- Integration: functional units can be plugged-in in a very standardized way, defining a clear interface to the user. This allows this platform framework to be easily integrable with a profiling-based ISA extension generator.
- Tested and testable: some programs were tested to prove functionality using the RTL simulation tools included in the OpenSPARC package. The modified hardware has been also synthesized and downloaded to an FPGA, where it booted an Ubuntu Linux operating system, proving its correctness. However, the ISA extended generated hardware has been only partially tested in real hardware. The modified code in the OpenSPARC core has been tested in a FPGA, but the included FPGA design of OpenSPARC emulates FPU operations on the Microblaze processor. New FPU operations have been tested by emulating them reprogramming the Microblaze firmware, using the current OpenSPARC mechanism.

An small howto is given to test the hardware modifications in appendix A.

The provided platform framework implements the ISA extensions using the Floating Point Unit. That mechanism makes including new hardware easier, but also introduces a significant overhead on the execution of the new ISA.

Fortunately, the analysis work done provides exact measurements of this overhead and tools to automatically measure it from simulation log files. This makes the platform useful or researching purposes and could also be reworked in the future to avoid the overheading elements, which were accessing the FPU, and using floating point load and store instructions. Without the detected overhead, speedups of up to 1.43x have been measured.

7.1 Future work

In the future, this work could be extended to allow the following enhancements:

1. Modifying the OpenSPARC package to execute the FPU hardware rather than emulating it on the Microblaze softprocessor.
2. Returning more than one result: it may be desirable for some applications that a new instruction provides more than one result. It would be relatively easy to return a result to the core and store other results in the TRF. Returning more than one result to the Sparc core would require more work, as it implies the crossbar protocol.
3. Using larger FPGAs or other compilers that allow to synthesize and place and route the OpenSPARC system at higher clock frequencies.
4. Vectorization of the TRF registers to allow SIMD operations. This would require reworking the TRF.
5. Data reuse in the TRF using shift registers: this would increase the performance for algorithms which make a lot of data reuse, such as FIR.
6. Publication of a paper describing the ISA Extension Platform.

Appendix A

ISA Extension Platform Framework How To

This appendix shows a step-by-step explanation on how to use the ISA Extension Platform to generate a modified OpenSPARC. The algorithm analyzed to determine a new ISA extension is the FIR algorithm.

A.1 Algorithm analysis

First of all, the original source code must be analyzed to look for fragments that could be executed together by a single instruction. It is intended that this work will be automated in the future.

We may have a C program such as the following one:

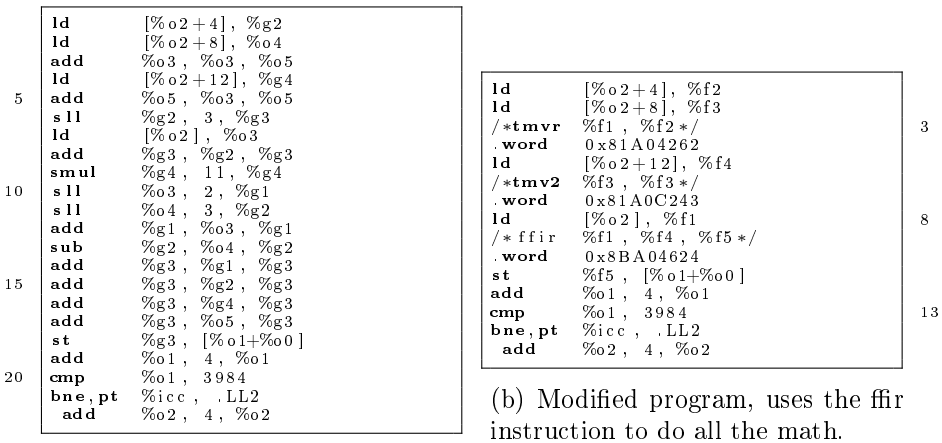
```
int A[1000];
int B[1000];
int main ()
{
        int i;
        for(i = 0 ; i < 996 ; i = i + 1)
        {
                B[i] = A[i]* 3 +
                        A[i+1]* 5 +
                        A[i+3]* 7 +
                        A[i+2]* 9 +
                        A[i+4]* 11;

        }
}
```

Figure A.1: Original fir.c code.

The user should compile it with gcc for sparc with the -O2, -S, and -mcpu=v9. We obtain the assembly code shown in figure A.2(a). After analyzing it and determining the critical part, we decide to transform it to the code in figure A.2(b) supposing *ffir* instruction makes all the computation required by the algorithm.

We can see that the modified assembly code needs TMV instructions. That is because the fir algorithm requires 5 operands to calculate a result for each iteration, but the OpenSPARC only allows two source operands per instruction. To solve this problem,



```
ld      [%o2+4], %g2
ld      [%o2+8], %o4
add     %o3, %o3, %o5
ld      [%o2+12], %g4
add     %o5, %o3, %o5
sll     %g2, 3, %g3
ld      [%o2], %o3
add     %g3, %g2, %g3
smul    %g4, 11, %g4
sll     %o3, 2, %g1
sll     %o4, 3, %g2
add     %g1, %o3, %g1
sub     %g2, %o4, %g2
add     %g3, %g1, %g3
add     %g3, %g2, %g3
add     %g3, %g4, %g3
add     %g3, %o5, %g3
st      %g3, [%o1+%o0]
add     %o1, 4, %o1
cmp     %o1, 3984
bne,pt  %icc, .LL2
 add    %o2, 4, %o2
```

(a) Original program, generated by gcc

```
ld      [%o2+4], %f2
ld      [%o2+8], %f3
/*tmvr  %f1, %f2*/
.word   0x81A04262
ld      [%o2+12], %f4
/*tmv2  %f3, %f3*/
.word   0x81A0C243
ld      [%o2], %f1
/*ffir  %f1, %f4, %f5*/
.word   0x8BA04624
st      %f5, [%o1+%o0]
add     %o1, 4, %o1
cmp     %o1, 3984
bne,pt  %icc, .LL2
 add    %o2, 4, %o2
```

(b) Modified program, uses the ffir instruction to do all the math.

Figure A.2: Compiler generated assembly code for the FIR algorithm.

TMV instructions send the 3 extra source operands to the TRF, so they can be used by the hardware to complete the calculation.

A.2 Design an appropriate pipeline

The user of this platform framework must be aware of the limitations of the platform and the hardware where the modified design is going to be implemented. In this case, the user must know that the TRF only provides one operand per cycle, and the *fir* instruction is going to need 3 parameters provided by the TRF (in addition to its two source operands). This is because the required computation for the fir instruction is:

$$f = \alpha_1 * 3 + \alpha_2 * 5 + \alpha_3 * 9 + \alpha_4 * 7 + \alpha_5 * 11$$

This gives the user the minimum number of cycles required to complete the *fir* instruction; three in this case. Then, the hardware complexity that will go to each stage has to be analyzed; a hardware too complex may increase the processor's cycle time. In this case, the required calculations are:

The hardware needs to do 5 multiplications by a constant value and then add the partial results. The multiplications are not specially complex, since they are by a small constant; but the addition must be split in 3 or 4 smaller additions, or the carry propagation would make the cycle time too high. The figure bellow shows a possible solution to this calculation split in small parts:

$$r_1 = \alpha_1 * 3 + \alpha_2 * 5$$
$$r_2 = \alpha_3 * 9 + \alpha_4 * 7$$
$$r_3 = \alpha_5 * 11 + r_1$$
$$r_T = r_2 + r_3$$

Figure A.3: Separation of FIR calculation in 4 parts. r_T has the final result.

In this approach, two multiplication are made in each part, and their result is then added. The resulting latency will be the latency of the multiplication by a constant plus the latency of the add, because both multiplications can be done in parallel.

We can see that this solution uses two operands at the first and the second parts and another operand on the third part, which can be converted to hardware. We can ask the TRF for the 3 values we need on stages 1, 2 and 3. So we can conclude that 4 stages should be enough to solve this problem.

A.3 Automatic hardware generation

Now we can generate a modified OpenSPARC with an extra 4 stage pipeline which will execute the *fir* instruction, and with support for TMV instructions. To do so, we first need to set the *OPENSPARC_SRC* environment variable to the installation directory of OpenSPARC T1 1.1.7. Then we just need to run the generation script:

```
mikel@home:~$ ./create_pipeline.sh
```

The script will then ask for the following information:

```
Name of the new pipeline (3 letters): fir
Give the pipeline an identifier (1 letter): f
How many stages will the new pipeline have?: 4
Will the new pipeline use the TRF? [y/n]: y
How many instructions will the fir pipeline handle?: 1
Write an instruction identifier for each one of the 1 instructions, separated
 by spaces (3 letters):
Example: aaa bbb ccc ddd for 4 instructions
fir
```

In this example above we have answered *fir*, *f*, *4*, *y*, *1*, and *fir*. The script will then output the following information:

```
Info: Encoding for ffir: opf=0x31 iword=10 5'rd 110100 5'rs1 9'HEX(31) 5'rs2
Patching /home/mikel/OpenSPARC
patching file design/sys/iop/fpu/xst/fpu.flist
patching file design/sys/iop/fpu/rtl/Flist.fpu
patching file design/sys/iop/fpu/rtl/fpu_in_ctl.v
patching file design/sys/iop/fpu/rtl/fpu_in.v
patching file design/sys/iop/fpu/rtl/fpu_out_ctl.v
patching file design/sys/iop/fpu/rtl/fpu_out_dp.v
patching file design/sys/iop/fpu/rtl/fpu_out.v
patching file design/sys/iop/fpu/rtl/fpu_rptr_groups.v
patching file design/sys/iop/fpu/rtl/fpu.v
patching file design/sys/iop/sparc/ffu/rtl/sparc_ffu_ctl.v
Copying new files: output/fpu_fir_ctl.v output/fpu_fir_dp.v output/fpu_fir.v
 output/fpu_rf.v output/fpu_tmv.v
done.
```

Once this is done, the source code in *OPENSPARC_SRC* is modified to fit the new *fir* pipeline, which can execute the *ffir* instruction. Flist files are also modified so the code can be simulated and synthesized. The scripts also output the encoding we will need to use to execute the instruction.

A.4 Functional unit implementation

After the automatic generation, we have a modified OpenSPARC with an extra 4 stage pipeline which is empty at the moment. The user has to write in Verilog the combinational logic that will calculate the result of an iteration of *fir*.

To do so, the user has to edit two files: design/sys/iop/fpu/rtl/fpu_fir_dp.v and design/sys/iop/fpu/rtl/fpu_fir_ctl.v

A.4.1 fpu_fir_ctl.v

When we go to the last lines of this file, we find a comment on how to modify the module:

```
/*************************************************************************
| Add your control logic here. If you do not want to use the TRF you
| can leave this commented. If you want to use it you have to specify
| what value you need to read 1 cycle before you use it.
|                                                                          5
| In the following example, R0 will be available to the instruction
| in stage 2, R1 to the instruction in stage 3, and R2 to the instruction
| in stage 4.
|
| assign reg_r =          f1stg_vld ? 0 :                                  10
|                         f2stg_vld ? 1 :
|                         f3stg_vld ? 2 :
|                         5'bzzzzz;
|
|*************************************************************************/ 15
```

In our example, we need to uncomment the assignment of *reg_r* because we want to use the TRF. We will read registers 0, 1 and 2 in stages 1, 2 and 3, like in the example, leaving the code like this:

```
assign reg_r =          f1stg_vld ? 0 :
                        f2stg_vld ? 1 :
                        f3stg_vld ? 2 :
                        5'bzzzzz;
```

This code will provide 3 values to our datapath, we will need to store the correct values in these registers. We will take care of that later.

A.4.2 fpu_fir_dp.v

The last lines of this files provide some help to the user too. The following comment is located before the *endmodule* reserved word:

```
/*************************************************************************  1
| Add your datapath logic here. You need to assign a value to the
| fir_out signal at the end, so the pipeline returns a valid value.
| If you need to propagate partial results you have to use the
| fNstg_partial_out signals (being N the stage number).
| If you need to use a value from the TRF you have to use the rf_in        6
| signal.
|
| --Listg_partial_out-->|--L(i+1)_partial--> ~~~~~ --L(i+1)_partial_out-->|-->L(i+2)_partial-->
|
```

```
/* In the following example, f1stg_in1 and f1stg_in2 are
 * added in the first stage, then propagated through the pipeline
 * until the result is written in fir_out in the last stage.
 *
 ************************************************************************/

assign f1stg_partial_out = f1stg_in1 + f1stg_in2;
assign f2stg_partial_out = f2stg_partial;
assign f3stg_partial_out = f3stg_partial;
assign fir_out = f4stg_partial;
```

As we can see, unlike in the control unit, the hardware is already doing something: an add in the first stage. This is only for illustrative purposes, and this code can be removed (but not disconnected; disconnecting the pipeline will cause an undefined result or a high impedance to be output). Partial and partial_out signals are 64 bit of width.

This last piece of code shows us how the pipelining works: we assign values to the *f1stg_partial_out signal*, and the next cycle these values will be in the *f2stg_partial* signal. And if we then assign that to the *f2stg_partial_out* signal, the next cycle these values will be in the *f3stg_partial* signal.

In figure A.3 we had the algorithm separated in 4 parts. Those four parts will be the four stages we have now. We will read α_3, α_4, and α_5 from the TRF, and we will read α_1 and α_2 from the fir instruction operands. The following could be the resulting Verilog code:

```
// First stage: r1 = alpha1 * 3 + alpha2 * 5
// We also read alpha3, to use it next cycle
assign f1stg_partial_out[63:32] =       f1stg_in1 * 3 + f1stg_in2 * 5;
assign f1stg_partial_out[31:0] =        rf_in;

// Second stage: r2 = alpha3 * 9 + alpha4 * 7
// We also propagate r1 to the third stage
assign f2stg_partial_out[63:32] =       f2stg_partial[63:32];
assign f2stg_partial_out[31:0] =        f2stg_partial[31:0] * 9 + rf_in * 7;

// Third stage: r3 = alpha5 * 11 + r1
// We also propagate r2 to the fourth stage
assign f3stg_partial_out[63:32] =       rf_in * 11 + f3stg_partial[63:32];
assign f3stg_partial_out[31:0] =        f3stg_partial[31:0];

// Fourth and last stage: rT = r2 + r3
assign fir_out =                        f4stg_partial[63:32] + f4stg_partial[31:0];
```

As we can see, we are using each signal to store 2 values. This is because we are operating with 32 bit integers and we have 64-bit registers available. This last code exactly mimics the behavior of the algorithm in figure A.3. The first stage, we multiply and add α_1 and α_2. We also save the incoming value from the TRF, corresponding to register 0. The second cycle, we operate with α_3 and α_4, both coming from the TRF (one of the values came this cycle, the other one came at the first stage and was saved). At the third stage, we add the result of the first stage (r_1) (which was propagated to the second stage through the high bits of the *partial* signal) to the result of multiplying $\alpha_5 * 11$. The last cycle, we just add $r_2 + r_3$.

This is a possible solution, however this is not the best one. We have to keep in mind that the TRF provides the value to the datapath the same cycle it is requested. The requested value comes with a certain delay, which makes it not recommended to do a lot of calculations with that value until the next cycle. If we do not do so, we will fail on keeping the cycle time small. In the example above, the *rf_in* signal is multiplied and then added in stage 2 and 3. To avoid this, we can change the way we separated the stages in figure A.3 to the way proposed in figure A.4.

$$r_1 = \alpha_1 * 3 + \alpha_2 * 5$$
$$r'_1 = \alpha_3 * 9$$
$$r_2 = r_1 + r'_1$$
$$r'_2 = \alpha_4 * 7$$
$$r_3 = r_2 + r'_2$$
$$r'_3 = \alpha_5 * 11$$
$$r_T = r_3 + r'_3$$

Figure A.4: Separation of FIR calculation in 4 parts with some changes to make it less likely to increase the cycle time. r_T has the final result.

Again, α_1 and α_2 are the values carried by the *fir* instruction, and the rest of *alphas* will be read from the TRF. We can see we use the inherent capability of parallelism of hardware to multiply the value read from the TRF the same cycle it is requested, but we save the add for the next cycle, decreasing the probability of increasing the cycle time. We do this at every stage. The resulting verilog code would be the following one:

```
assign f1stg_partial_out[63:32] = (f1stg_in1 * 11) + (f1stg_in2 * 9);
assign f1stg_partial_out[31:0]  = rf_in * 7;

assign f2stg_partial_out[63:32] = f2stg_partial[63:32] + f2stg_partial[31:0];
assign f2stg_partial_out[31:0]  = rf_in * 5;

assign f3stg_partial_out[63:32] = f3stg_partial[63:32] + f3stg_partial[31:0];
assign f3stg_partial_out[31:0]  = rf_in * 3;

assign fir_out = f4stg_partial[63:32] + f4stg_partial[31:0];
```

It is important to apply this kind of optimizations, because that helps reduce the cycle time (increase the frequency). In the 3 stage FIR analyzed in the 6.3.1 subsection in page 31 in this work, an implementation focusing on decreasing the number of stages of the pipeline failed to maintain the cycle time low.

A.5 Implementing bigger functional units

It is possible to implement bigger designs, however, the designer may need more signals than the provided *partial*, *in1*, and *in2* signals. To add new signals, new wires have to be defined. If we require that these new signals are propagated through the pipeline to use them in future stages, we also have to instantiate new flip-flops. The following example shows how the hardware for the 8 stage EDGE is implemented. It is also commented so it is more easily understandable:

```
/* This is the loop body of EDGE:
*                           MASKv =
*                                   (P[i-1][j+1] - P[i-1][j-1]) +
*                                   (P[i][j+1] - P[i][j-1]) +
*                                   (P[i+1][j+1] - P[i+1][j-1]);
*                           MASKh =
*                                   (P[i+1][j-1] - P[i-1][j-1]) +
*                                   (P[i+1][j] - P[i-1][j]) +
*                                   (P[i+1][j+1] - P[i-1][j+1]);
*                           B[i][j] = (MASKv*MASKv + MASKh*MASKh);
*
* We define:
*   P1 = MASKv
*   P2 = MASKh
*   ++, pp = P[i+1][j+1]
*   +-, pm = P[i+1][j-1]
*   -+, mp = P[i-1][j+1]
*   --, mm = P[i-1][j-1]
*   0+ = P[i][j+1]
*   0- = P[i][j-1]
*   +0 = P[i+1][j]
```

```
*  -0 = P[i-1][j]
*  P11 = (P[i-1][j+1] - P[i-1][j-1])
*  P12 = (P[i][j+1] - P[i][j-1])
*  P13 = (P[i+1][j+1] - P[i+1][j-1])
*  P21 = (P[i+1][j-1] - P[i-1][j-1])
*  P22 = (P[i+1][j] - P[i-1][j])
*  P23 = (P[i+1][j+1] - P[i-1][j+1])
*/

wire [31:0]     e2stg_pm_out;
wire [31:0]     e2stg_pm;
wire [31:0]     e3stg_pm_out;
wire [31:0]     e3stg_pm;
wire [31:0]     e4stg_pm_out;
wire [31:0]     e4stg_pm;
wire [31:0]     e5stg_pm_out;
wire [31:0]     e5stg_pm;
wire [31:0]     e6stg_pm;
wire [31:0]     e4stg_pp_out;
wire [31:0]     e4stg_pp;
wire [31:0]     e5stg_pp_out;
wire [31:0]     e5stg_pp;
wire [31:0]     e6stg_pp;
wire [31:0]     e3stg_mm_out;
wire [31:0]     e3stg_mm;
wire [31:0]     e4stg_mm_out;
wire [31:0]     e4stg_mm;
wire [31:0]     e5stg_mm;
wire [31:0]     e5stg_mp;
wire [31:0]     e5stg_P1_out;
wire [31:0]     e6stg_P1;
wire [31:0]     e6stg_P1_out;
wire [31:0]     e7stg_P1;

// P22 + half P21
assign e1stg_partial_out[63:32] = (e1stg_in1 - e1stg_in2) + rf_in;
// R5, +-
assign e1stg_partial_out[31:0] = rf_in;

// P22 + P21
assign e2stg_partial_out[63:32] = e2stg_partial[63:32] - rf_in;
// R1, --
assign e2stg_partial_out[31:0] = rf_in;
// other
assign e2stg_pm_out[31:0] = e2stg_pm[31:0];

// P21 + P22 + half P23
assign e3stg_partial_out[63:32] = e3stg_partial[63:32] + rf_in;
// R4, ++
assign e3stg_partial_out[31:0] = rf_in;
// other
assign e3stg_mm_out[31:0] = e3stg_mm[31:0];
assign e3stg_pm_out[31:0] = e3stg_pm[31:0];

// P21 + P22 + P23
assign e4stg_partial_out[63:32] = e4stg_partial[63:32] - rf_in;
// R0, -+
assign e4stg_partial_out[31:0] = rf_in;
// other
assign e4stg_mm_out[31:0] = e4stg_mm[31:0];
assign e4stg_pp_out[31:0] = e4stg_pp[31:0];
assign e4stg_pm_out[31:0] = e4stg_pm[31:0];

// P2 * P2
assign e5stg_partial_out[63:32] = e5stg_partial[63:32] * e5stg_partial[63:32];
// R2, 0+
assign e5stg_partial_out[31:0] = rf_in;
// P11 + half P12
assign e5stg_P1_out[31:0] = rf_in + e5stg_partial[31:0] - e5stg_mm[31:0];
// other
assign e5stg_pp_out[31:0] = e5stg_pp[31:0];
assign e5stg_pm_out[31:0] = e5stg_pm[31:0];

// P2 done
assign e6stg_partial_out[63:32] = e6stg_partial[63:32];
//R3, 0-
assign e6stg_partial_out[31:0] = rf_in;
// P11 + P12 + P13
assign e6stg_P1_out[31:0] = e6stg_P1[31:0] - rf_in + (e6stg_pp[31:0] - e6stg_pm[31:0]);

// P2 done
assign e7stg_partial_out[63:32] = e7stg_partial[63:32];
// P1*P1
assign e7stg_partial_out[31:0] = e7stg_P1[31:0] * e7stg_P1[31:0];

// P1 done P2 done
assign edg_out[63:0] = e8stg_partial[63:32] + e8stg_partial[31:0];

dffe_s #(32) i_e6stg_P1 (
        .din    (e5stg_P1_out),
        .en     (e8stg_step),
        .clk    (rclk),

        .q      (e6stg_P1),
```

```
          .se       (se),
          .si       (),
          .so       ()
);
dffe_s #(32) i_e7stg_P1 (
          .din      (e6stg_P1_out),
          .en       (e8stg_step),
          .clk      (rclk),

          .q        (e7stg_P1),

          .se       (se),
          .si       (),
          .so       ()
);

dffe_s #(32) i_e5stg_mp (
          .din      (e4stg_partial_out[31:0]),
          .en       (e8stg_step),
          .clk      (rclk),

          .q        (e5stg_mp),

          .se       (se),
          .si       (),
          .so       ()
);

dffe_s #(32) i_e5stg_mm (
          .din      (e4stg_mm_out),
          .en       (e8stg_step),
          .clk      (rclk),

          .q        (e5stg_mm),

          .se       (se),
          .si       (),
          .so       ()
);
dffe_s #(32) i_e4stg_mm (
          .din      (e3stg_mm_out),
          .en       (e8stg_step),
          .clk      (rclk),

          .q        (e4stg_mm),

          .se       (se),
          .si       (),
          .so       ()
);
dffe_s #(32) i_e3stg_mm (
          .din      (e2stg_partial_out[31:0]),
          .en       (e8stg_step),
          .clk      (rclk),

          .q        (e3stg_mm),

          .se       (se),
          .si       (),
          .so       ()
);

dffe_s #(32) i_e6stg_pp (
          .din      (e5stg_pp_out),
          .en       (e8stg_step),
          .clk      (rclk),

          .q        (e6stg_pp),

          .se       (se),
          .si       (),
          .so       ()
);
dffe_s #(32) i_e5stg_pp (
          .din      (e4stg_pp_out),
          .en       (e8stg_step),
          .clk      (rclk),

          .q        (e5stg_pp),

          .se       (se),
          .si       (),
          .so       ()
);
dffe_s #(32) i_e4stg_pp (
          .din      (e3stg_partial_out[31:0]),
          .en       (e8stg_step),
          .clk      (rclk),

          .q        (e4stg_pp),

          .se       (se),
          .si       (),
          .so       ()
);
```

```
dffe_s #(32) i_e6stg_pm (
        .din    (e5stg_pm_out),
        .en     (e8stg_step),
        .clk    (rclk),

        .q      (e6stg_pm),

        .se     (se),
        .si     (),
        .so     ()
);
dffe_s #(32) i_e5stg_pm (
        .din    (e4stg_pm_out),
        .en     (e8stg_step),
        .clk    (rclk),

        .q      (e5stg_pm),

        .se     (se),
        .si     (),
        .so     ()
);
dffe_s #(32) i_e4stg_pm (
        .din    (e3stg_pm_out),
        .en     (e8stg_step),
        .clk    (rclk),

        .q      (e4stg_pm),

        .se     (se),
        .si     (),
        .so     ()
);
dffe_s #(32) i_e3stg_pm (
        .din    (e2stg_pm_out),
        .en     (e8stg_step),
        .clk    (rclk),

        .q      (e3stg_pm),

        .se     (se),
        .si     (),
        .so     ()
);
dffe_s #(32) i_e2stg_pm (
        .din    (e1stg_partial_out[31:0]),
        .en     (e8stg_step),
        .clk    (rclk),

        .q      (e2stg_pm),

        .se     (se),
        .si     (),
        .so     ()
);
```

Figure A.5: EDGE pipeline's data path modifications.

In this code we can see we are saving several partial results that will be used in the next stages. This requires some additional signals to be declared and some new flip-flops to save the data until it reaches the stage where it is used. For example, signal *e2stg_pm* holds the $P[i+1],[j-1]$ value. It is then propagated through the pipeline assigning it to the *e2stg_pm_out* signal, which is then saved in flip-flop *i3stg_pm*, giving to stage 3 the *e3stg_pm* signal. This is repeated with every new signal declared. This was not needed in the FIR example because *partial*, *partial_out*, *in1* and *in2* signals are already declared and flip-flopped.

A.6 Simulation of the hardware using sims

The OpenSPARC package provides a series of tools to simulate the Verilog source code. This section tries to summarize OpenSPARC's help, which is available on chapter 1.3 of [3]. To run the simulations, one of Vera, Debussy, VCS, or NCVerilog programs are

required.

It is important before proceeding to modify the OpenSPARCT1.bash script to fit to our needs and to source it:

```
mikel@home:~/OpenSPARC$ source OpenSPARCT1.bash
```

Additionally, this can be inserted to our .bashrc file to avoid having to source the script every time we open a new console session.

After the environment has been set up, we can run simulations of the full 8-core OpenSPARC or of a subset of it. The script then will compile test programs and run them on the simulated hardware. The amount of test programs that will be run is determined by the regression group name, which can be one of the following:

- core1_mini
- core1_full
- chip8_mini
- chip8_full
- thread1_mini
- thread1_full

As it can be observed, the first word defines which subset of hardware will be simulated, and the second word defines the amount of test programs that will be built. For the tests in this work, core1_mini has been used. The following example shows an example use of the *sims* tool:

```
mikel@home:~/OpenSPARC$ sims -max_cycle=500000 -rtl_timeout=100000 -sim_type=vcs
 -group=core1_mini
```

The list of programs to be run can be modified by the user so new tests can be done. To do this, we need to edit the file verif/diag/cmp_basic.diaglist, where we can enclose our new programs between a <core1_mini> and </core1_mini> so they are executed. The syntax for adding new programs to this list is "testname program.s". For example "FIR-4stg fir-4stg.s". Then, we need to place our program in the verif/diag/assembly/arch/fp directory so it can be found by the *sims* tool. The assembly file with our new program has to be modified so it can be executed by the simulator. For example, the following code shows the assembly code for the modified FIR. Note the inclusion of a boot.s file.

```
#include "boot.s"

.global sam_fast_immu_miss
.global sam_fast_dmmu_miss

.text
.global main
main:
        wr      %g0, 0x7, %fprs         /* make sure fef is 1 */
        sethi   %hi(A), %g1
        sethi   %hi(B), %g2
        ld      [%g1+%lo(A)], %o3
```

```
        or      %g2, %lo(B), %o0
        or      %g1, %lo(A), %g1
        mov     0, %o1
        add     %g1, 4, %o2
.LL2:                                                                   17
        ld      [%o2+4], %g2
        ld      [%o2+8], %o4
        add     %o3, %o3, %o5
        ld      [%o2+12], %g4
        add     %o5, %o3, %o5                                           22
        sll     %g2, 3, %g3
        ld      [%o2], %o3
        add     %g3, %g2, %g3
        smul    %g4, 11, %g4
        sll     %o3, 2, %g1                                             27
        sll     %o4, 3, %g2
        add     %g1, %o3, %g1
        sub     %g2, %o4, %g2
        add     %g3, %g1, %g3
        add     %g3, %g2, %g3                                           32
        add     %g3, %g4, %g3
        add     %g3, %o5, %g3
        st      %g3, [%o1+%o0]
        add     %o1, 4, %o1
        cmp     %o1, 3984                                               37
        bne,pt  %icc, .LL2
         add    %o2, 4, %o2
        ta              T_GOOD_TRAP
.data
A:      .space 4000                                                     42
B:      .space 4000
```

It is also important to note the inclusion of a *ta* instruction generating a GOOD_TRAP interruption. The simulator captures this trap and determines that the simulation has been good.

After the simulations are concluded, a directory named after the current date is created. This directory contains all the information output by the simulator. There is a subdirectory for each test defined in the cmp_basic.diaglist, and in every subdirectory there is a sim.log file which has the information about the execution.

A.6.1 Removing simulation overhead and overhead caused by accessing the FPU and loading/storing to floating point registers

A tool has been developed to remove the overhead caused by accessing the FPU to execute new instructions (including TMV instructions) and the overhead caused by using floating point load and store instructions instead of integer load/store. This tool also removes the simulation overhead, caused by the inclussion of the boot.s file. The tool requires a sim.log file generated as an output of *sims*. The following listing is a small fraction of the *sim.log* log file:

```
(105147579)Info-perm thread(00) pc(0000000020000040) npc(0000000020000044) opcode(92026004)
105148411:L2_DRAM_MON[0] -> Read Data  :: ReqId = 0, Addr = 1170001f00, Data =   2
     00000000000000000000000000000000
105148411:pc-updated -> spc(0) thread(0) window(0) val = 000020000044
105148411:npc-updated -> spc(0) thread(0) window(0) val = 000020000048
105148411:ccr_reg-updated -> spc(0) thread(0) window(0) val = 99
(105148411)Info-perm thread(00) pc(0000000020000044) npc(0000000020000048) opcode(80a26f90)
105149247:L2_DRAM_MON[0] -> Read Data  :: ReqId = 0, Addr = 1170001f00, Data =   7
     00000000000000000000000000000000
105149247:pc-updated -> spc(0) thread(0) window(0) val = 000020000048
105149247:npc-updated -> spc(0) thread(0) window(0) val = 00002000004c
(105149247)Info-perm thread(00) pc(0000000020000048) npc(000000002000004c) opcode(124ffff6)
```

The following example shows the tool being used to measure the overhead on the 4 stage FIR extension.

```
mikel@home:~$ export VERBOSE=0; ./overhead.sh ~/OpenSPARC/OpenSPARC_model/2010_08_10_0/fir-mod\:
     model_core1\:core1_mini\:0/sim.log
```

```
Total cycles                    127370 cycles
Simulation overhead             13212 cycles
Instructions to be analyzed:    11963. Please wait...

FPU Overhead:                   75708 cycles

 TMV instructions found:              1992
 New instructions found:              996
 Load to floating point found:        3986
 Store to floating point found:       996

Time-Overhead:                  38450 cycles
```

Setting the VERBOSE environment variable to 1 prints for each instruction that has been executed in the simulation a line, informing on the PC of the instruction (normalized so the first instruction has PC=0 and the step between instructions is 1 instead of 4) and the overhead caused by that instruction. The following is an example of the verbose output. We can see that a loop is being executed:

```
PC=8,   Overhead=6
PC=9,   Overhead=6
PC=10,  Overhead=12
PC=11,  Overhead=6
PC=12,  Overhead=12
PC=13,  Overhead=6
PC=14,  Overhead=21
PC=15,  Overhead=7
PC=16,  Overhead=0
PC=17,  Overhead=0
PC=18,  Overhead=0
PC=19,  Overhead=0
PC=8,   Overhead=6
PC=9,   Overhead=6
PC=10,  Overhead=12
```

A.6.2 Adding simulation traces

The simulator can be modified to generate traces about the execution. We can generate traces of all the processor, but it is better to limit the amount of traces to the submodules we are interested in, because generating a lot of traces can slow down the simulation and output files of several gigabytes of size.

To output traces we should edit the verif/env/cmp/cmp_top.sh file and add the following non-RTL Verilog code:

```
  initial begin
        $dumpfile("trace.vcd");
        //Uncomment one of the following to output traces

        //$dumpvars(0, cmp_top.iop);
        //$dumpvars(0, cmp_top.iop.sparc0.ffu);
        //$dumpvars(0, cmp_top.iop.fpu);
        //$dumpvars(3, cmp_top.iop.sparc0);
  end
```

The dumpvars function outputs the trace information to the trace.vcd file. The first parameter is the levels of depth that will be explored (0 means no limit of depth), and the second parameter is the top level module where the collection of traces will start. For example, if we use $dumpvars(0,cmp_top.iop.sparc0.ffu), we will not be able to access the traces generated by the sparc0 module or any of its submodules (except for ffu). If we use $dumpvars(1,cmp_top.iop.sparc0), all the signals of sparc0 will be output, but as the first parameter is 1, there will not be information about any submodule (such as ffu, lsu, exu, ifu...). Once the trace.vcd file is generated, we can use the *gtkwave* software to view the traces. In figure A.6 we can see and example of a trace, where we have selected some signals relevant to the execution of the *fir* pipeline.

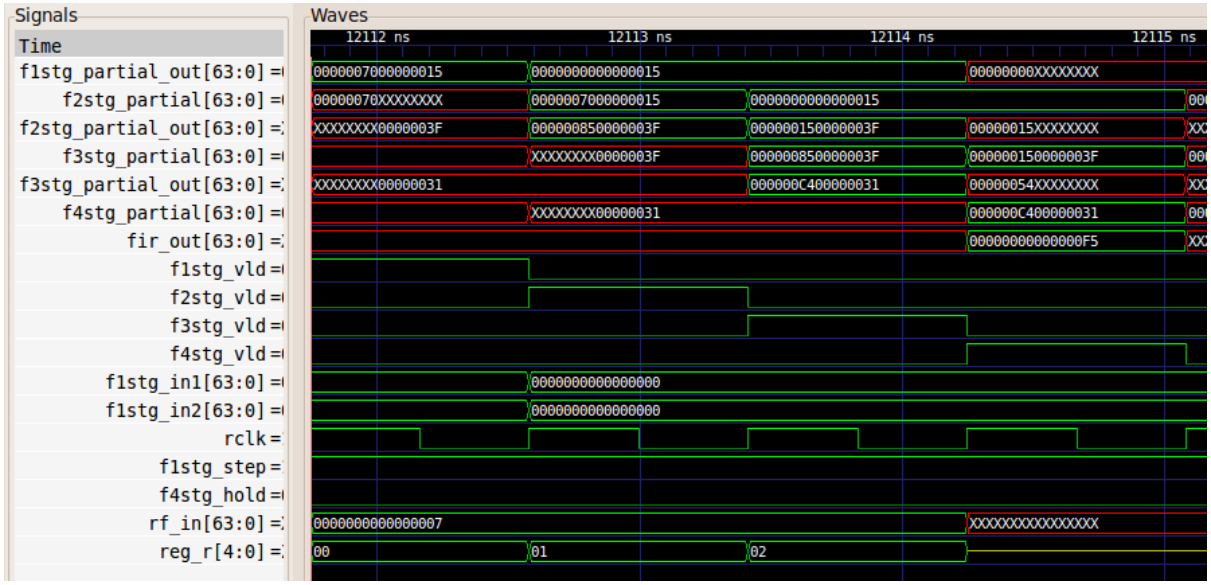


Figure A.6: A view of a part of a trace using the gtkwave open source software.

Looking at figure A.6, we can see we are executing a *ffir* instruction where both operators (*in1* and *in2*) are 7, and the values coming from the TRF (*rf_in*) are also 7. We can see each stage doing its partial calculations, and finally the *fir_out* signal taking the hexadecimal value F5, which is correct for the parameters used.

A.7 Implementing OpenSPARC on an FPGA and running Ubuntu

.

There has been some work done [6] on how to automate the generation of hardware to implement the OpenSPARC on an FPGA and to run the Ubuntu Linux operating system on it. This is also explained in [3], section 6.6. Unfortunately, the FPGA design does not implement the FPU, instead it is emulated by the Microblaze.

To change the FPU emulation on the FPGA, the file sys/edk/ccx-firmware/src/mbfw.c can be modified.

The following code is an example on how to change the Microblaze firmware source code to allow TMV and FIR instructions.

```
#define FP_OPCODE_TMV1      0x11                                    1
#define FP_OPCODE_TMVR      0x12
#define FP_OPCODE_TMV2      0x13
#define FP_OPCODE_FIR       0x31

[...]                                                               6

static uint_t TRF[8] = {0,0,0,0,0,0,0,0};
static uint_t IDX = 0;

[...]                                                               11

    switch (fp_opcode) {

    case FP_OPCODE_TMVR:
    TRF[0] = *((uint_t*)&(pcx_pkt->data0));                         16
    TRF[1] = *((uint_t*)&(pcx_pkt->data1));
    IDX = 2;
```

```
        break;

    case FP_OPCODE_TMV2:
    TRF[IDX] = *((uint_t*)&(pcx_pkt->data0));
    TRF[IDX+1] = *((uint_t*)&(pcx_pkt->data1));
    IDX += 2;
        break;

    case FP_OPCODE_FIR:
    fp_out = (*((uint_t*)(&(pcx_pkt->data0))))*5 +
     (*((uint_t*)(&(pcx_pkt->data1))))*11
     + TRF[0]*3 + TRF[1]*9 + TRF[2]*7;
    fp_rd = *((float32*) &fp_out);
        break;

[...]
```

Appendix B

Module hierarchy

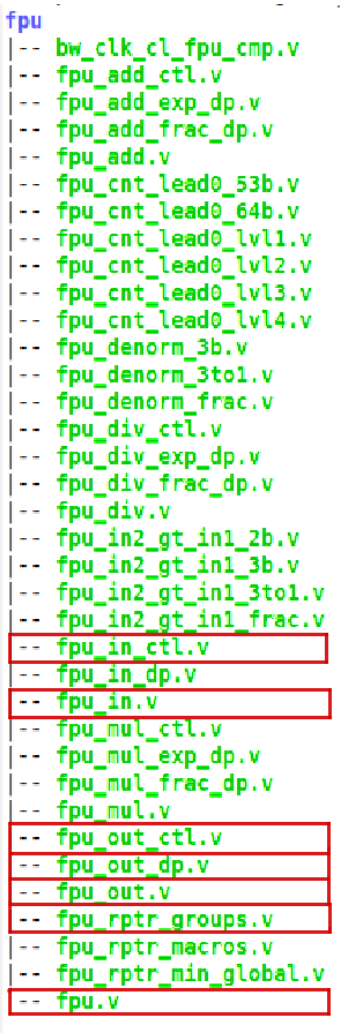
The OpenSPARC T1 code is organized in modules. The top level module is called *iop* and instantiates the core, the fpu, the crossbar, and other required structures such as memories.

The only modules shown in figure B.1 are *iop*, *iop/sparc*, *iop/fpu*, and *iop/sparc/ffu* because these are the most relevant ones in this project. In the figure, the modules modified in this work are shown squared. Module FPU includes also five new modules: *fpu_rf.v*, *fpu_tmv.v*, *fpu_new.v*, *fpu_new_dp.v*, and *fpu_new_ctl.v*.

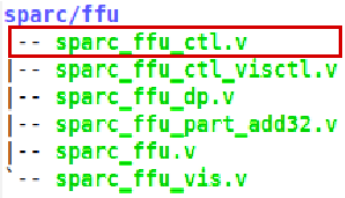
For a more detailed list of the module, hierarchy, the OpenSPARC source code can be downloaded; the module hierarchy matches the directory structure of the rtl code.

(a) Top module IOP.

(b) Sparc module.



(c) FPU module.



(d) FFU submodule.

Figure B.1: OpenSPARC T1 module hierarchy.

Bibliography

[1] Sun Microsystems, *UltraSPARC Architecture 2005*, Revision Draft D0.9.2, 2008.

[2] Sun Microsystems, *OpenSPARC T1 Microarchitecture Specification*, Revision A, 2006.

[3] Sun Microsystems, *OpenSPARC T1 Processor Design and Verification User's Guide*, Revision E, 2009.

[4] Sun Microsystems, *OpenSPARC T1 Supplement to the UltraSPARC Architecture 2005*, Draft D2.0, 2006.

[5] Sun Microsystems, *OpenSPARC T1 FPGA Implementation*, Release 1.6

[6] Desarrollo de una plataforma de trabajo para la investigación, *Victor Blazquez Francisco*, Undergraduate final project, 2010.

[7] OpenSPARC web site, *http://www.opensparc.net*

[8] David L. Weaver, Tom Germond, *The Sparc Architecture Manual, Version 9.*

[9] Ricardo E. Gonzalez, *Xtensa: a configurable and extensible processor*, Micro 2000.

[10] P. Guironnet de Massas, P. Amblard, F. Petrot, *On SPARC LEON-2 ISA Extensions Experiments for MPEG Encoding Acceleration*, VLSI Design, 2007, Issue 1.

[11] A. Wang, E. Killian, D. Maydan, C. Rowen, *Hardware/Software Instruction Set Configurability for System-on-Chip Processors*, 2001.

[12] Xilinx, *Xilinx Synthesis Technology (XST) User Guide.*

[13] S. Vassiliadis, S. Wong, G. Gaydadjiev, K. Bertels, G. Kuzmanov, E. Moscu Panainte, *The MOLEN Polymorphic Processor*, IEEE transactions on computers, Volume 53, Issue 11, 2004.

[14] C. Gonzalez, D. Jimenez-Gonzalez, X. Martorell, C. Alvarez, G. Gaydadjiev, *Preliminary Work on a Mechanism for Testing a Customized Architecture*, Advanced Computer Architecture and Computation for Embedded Systems (ACACES 2009), Poster Session, pp. 147-150

[15] C. Gonzalez, D. Jimenez-Gonzalez, X. Martorell, C. Alvarez, G. Gaydadjiev, *Metodologia para la generacion y evaluacion automatica de hardware especifico*, XX Jornadas de Paralelismo (JP 2009), pp. 295-300.

[16] *Trimaran: An infrastructure for research in back-end compilation and architecture exploration*, http://www.trimaran.org

[17] N. Clark, H Zhong, *Automated custom instruction generation for domain-specific processor acceleration*, IEEE Trans. Comput., vol.54, no. 10, pp. 1258-1270, 2005.

[18] B. Middha, V. Raj, A. Gangwar, A. Kumar, M. Balakrishnan, P. Ienne, A Trimaran Based Framework for Exploring the Design Space of VLIW ASIPs with Coarse Grain Functional Units,in Proceedings of the 15th International Symposium on System Synthesis, 2002, pp. 2-7.

[19] D. Jain, A. Kumar, L. Pozzi, P. Ienne, *Automatically Customising VLIW Architectures with Coarse Grained Application-specific Functional Units*, in Proceedings of the 8th International Workshop on Software and Compilers for Embedded Systems, 2004