EXTENSIBLE MICROPROCESSOR WITHOUT INTERLOCKED PIPELINE STAGES (eMIPS), THE RECONFIGURABLE MICROPROCESSOR

A Thesis

by

RICHARD NEIL PITTMAN

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2007

Major Subject: Computer Engineering

EXTENSIBLE MICROPROCESSOR WITHOUT INTERLOCKED PIPELINE STAGES (eMIPS), THE RECONFIGURABLE MICROPROCESSOR

A Thesis

by

RICHARD NEIL PITTMAN

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Approved by:

Chair of Committee,	Jyh-Charn Liu
Committee Members,	Duncan M. Walker
	Norman Guinasso
Head of Department,	Valerie E. Taylor

May 2007

Major Subject: Computer Engineering

ABSTRACT

Extensible Microprocessor without Interlocked Pipeline Stages

(eMIPS), the Reconfigurable Processor. (May 2007)

Richard Neil Pittman, B.S., Texas A&M University

Chair of Advisory Committee: Dr. Jyh-Charn Liu

In this thesis we propose to realize the performance benefits of application-specific hardware optimizations in a general-purpose, multi-user system environment using a dynamically extensible microprocessor architecture. We have called our dynamically extensible microprocessor design the Extensible Microprocessor without Interlocked Pipeline Stages, or eMIPS.

The eMIPS architecture uses the interaction of fixed and configurable logic available in modern Field Programmable Gate Array (FPGA). This interaction is used to address the limitations of current microprocessor architectures based solely on Application Specific Integrated Circuits (ASIC). These limitations include inflexibility, size, and application specific performance optimization. The eMIPS system allows multiple secure extensions to load dynamically and to plug into the stages of a pipelined central processing unit (CPU) data path, thereby extending the core instruction set of the microprocessor. Extensions can also be used to realize on-chip peripherals, and if area permits, even multiple cores. Extension instructions reduce dramatically the execution time of frequently executed instruction patterns. These new functionalities we have

developed can be exploited by patching the binaries of existing applications, without any changes to the compilers.

A FPGA based workstation prototype and a flexible simulation system implementating this design demonstrates speedups of 2x-3x on a set of applications that include video games, real-time programs and the SPEC2000 integer benchmarks. eMIPS is the first realized workstation based entirely on a dynamically extensible microprocessor that is safe for general purpose, multi-user applications. By exposing the individual stages of the data path, eMIPS allows optimizations not previously possible. This includes permitting safe and coherent accesses to memory from within an extension, optimizing multi-branched blocks, and throwing precise and restart able exceptions from within an extension.

This work describes a simplified implementation of an extensible microprocessor architecture based on the Microprocessor without Interlocked Pipeline Stages (MIPS) Reduced Instruction Set Computer (RISC) architecture. The concepts and methods contained within this thesis may be applied to other similar architectures. Given this simplified prototype we look forward to propose how this architecture will be expanded as it matures.

DEDICATION

To my wife

ACKNOWLEDGEMENTS

I would like to thank my committee chair, Dr. Jyh-Charn (Steve) Liu, for his guidance and support throughout the course of this research and my years in the graduate program. I would also like to thank my committee, Dr. Duncan M. Walker and Dr. Norman Guinasso, for their help with this work.

Thank you to my friends, colleagues, faculty and staff of the Department of Computer Science at Texas A&M University for making the last eight years of my life so memorable. I hope that I gave back at least half of what I received in my interactions with all of you.

I would also like to extend my gratitude to Alessandro Forin of Microsoft Research who provided tremendous support at every stage of this research project. The tools and previous work I had the privilege to build upon, allowed me to explore a problem that might have been overwhelming otherwise.

Finally, thank you to my family for their support through it all. Thank you to my wife for showing me that sometimes when you get stuck, you to take a few steps back. Thank you to my mother for always believing in me. Thank you to my brother and sister for supporting me as only they can. Thank you to my grandmother for always being there when I need to someone to talk to. Thank you to my in-laws for their emotional support for my wife and me.

NOMENCLATURE

ALU	Arithmetic Logic Unit
API	Application Program Interface
ASIC	Application Specific Integrated Circuit
CAM	Content Addressable Memory
CPU	Central Processing Unit
DPS	Digital Signal Processor
eMIPS	Extensible MIPS
GUI	Graphic User Interface
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
HDL	Hardware Description Language
IC	Integrated Circuit
ID	Instruction Decode
IDE	Integrated Drive Electronics
IE	Instruction Execute
IF	Instruction Fetch
ISA	Instruction Set Architecture
JTAG	Joint Test Action Group
LUT	Look-up Table
MA	Memory Access

MIC	Microsoft Invisible Computing
MIPS	Microprocessor without Interlocked Pipeline Stages
MMU	Memory Management Unit
OS	Operating System
PC	Program Counter
PIO	Parrallel Input/Output Interface
PLI	Programming Language Interface
RAM	Random Access Memory
RI	Reserved Instruction
RISC	Reduced Instruction Set Computer
SRAM	Static Random Access Memory
System ACE	System Advanced Configuration Environment
TBUF	Tristate Buffer
TISA	Trusted Instruction Set Architecture
USART	Universal Serial Synchronous Asynchronous Receiver Transmitter
VHDL	Very High Speed Integrated Circuit Hardware Description Language
WB	Writeback

TABLE OF CONTENTS

PAGE

PAGE

LIST OF FIGURES

FIGURE PAGE

1 INTRODUCTION

Most contemporary microprocessors, e.g., MIPS, ARM and PowerPC, etc., are based on a RISC architecture, in which a fixed set of assembly instructions are defined based on the analysis of software benchmarks, costs and hardware technologies. The vast majority of those microprocessors are fabricated as Application Specific Integrated Circuits, or ASIC, for best performance in terms of speed and chip area. In an ASIC microprocessor the instruction set is optimized statistically, therefore the temporal and spatial locality of a particular software application cannot be considered individually. Adding co-processing units to a microprocessor is a common approach to remedy the limitations of an instruction set architecture, but this approach faces the same problem that the coprocessor architecture cannot be easily customized for each software application.

In this thesis we propose an Extensible MIPS architecture, or eMIPS, which allows an application designer to identify heavily executed software regions, or basic block patterns, so that extension instructions and their execution units can be synthesized based on optimized execution flows in those basic blocks. The fixed instruction areas can be implemented using the ASIC technology, but the extension instructions and their execution units are based on configurable logic, so that they can be configured before program run time to optimize the computations.

This thesis follows the style of *IEEE Transactions on Computers.*

Integrated architectures that support an ASIC processor interfaced with programmable fabrics are available in the embedded marketplace, e.g., the Xilinx Vertex IV and V FX series [49][58][51]. However, the general issue of using instruction extensions in a general-purpose, multi-user application environment is an open issue being addressed in this thesis. Additionally, those architectures do not allow the programmable logic to interact with the individual stages of the pipeline and therefore are only usable for larger-grain optimizations. Unlike the typical instruction compaction techniques which would be targeted for a particular architecture, in eMIPS we aim to identify the computing needs of basic block patterns and leverage the unused instruction space together with the necessary execution units to improve the overall performance. With relatively small additions of configurable logic to existing processor, the extension instruction architecture allows optimization of individual software applications at a fine granularity that cannot be achieved otherwise.

To validate our concepts, the partial reconfiguration feature supported by modern FPGAs (Virtex series from Xilinx Inc. [49][51][58][52] and the Stratix series from Altera [3]) is used for prototype implementation, and the interactive hardware-software development processes for eMIPS are realized using an integrated simulation/development environment called Giano [37][22]. The MIPS R4000 processor is chosen as the Trusted Instruction Set Architecture, or TISA, to guarantee that at cold start the microprocessor has a secure and known instruction set and hardware state. The unused operation codes of the R4000 instruction set can be used for the extension instructions.[32] We designed the instruction decoder, data paths, and the

pipeline interlock mechanisms, and implemented them on an FPGA board to validate the eMIPS architecture. The field programmable logic fabrics are partitioned into areas for either dedicated functions, or extension logic. Through the partial reconfiguration technique, only the region that needs to be reconfigured is updated with its configuration file while the TISA operations and other extensions remain unchanged. The Xilinx ML401 Evaluation board has enough peripherals that we could create a complete eMIPS workstation and experiment with a multi-user application environment.

Real software programs are complied into their binary images, their basic block patterns identified and then encoded into Extension instructions to evaluate the performance gains. We used several software systems, ranging from an object oriented real-time operating system for embedded applications, video games, and the SPEC2000 benchmarks. Experimental results show substantial performance gains for both large and small software systems, with overall application speedup factors of 2x-3x using a very small number of extension instructions per application.

The eMIPS system makes the following specific contributions. eMIPS is the first realized workstation based entirely on a dynamically extensible processor that is safe for general purpose, multi-user applications. By exposing the individual stages of the data path, eMIPS allows optimizations not previously possible. This includes permitting safe and coherent accesses to memory from within an extension, optimizing multi-branched blocks, and throwing precise and restart able exceptions from within an extension.

The remainder of this thesis is structured as follows. Section 2 gives an overview of all relevant background information including FPGA architecture and software

profiling. The eMIPS architecture is introduced, including software support, in Section 3. Section 4 lays out the challenges and implementation of the eMIPS prototype. The testing and verification processes are discussed in Section 5. Experimental results are reported in Section 6. In Section 7, future avenues available for exploration and development are discussed. Section 8 presents related work and Section 9 concludes the thesis.

2 BACKGROUND

The following section provides information supporting the motivation of this thesis and places the claims made within into perspective. Section 2.1 makes the argument that the design paradigms based solely on ASIC technology approach their limits and that eMIPS has the potential to provide continued growth in microprocessor design. The selection of the MIPS instruction set and explanation of RISC pipelines is provided in Section 2.2. An overview of the FPGA technology features utilized in the eMIPS prototype is given in Section 2.3. Finally, Section 2.4 presents a summary of the software profiling on which the eMIPS architecture is based.

2.1 SHORTCOMINGS OF MODERN ASICS

Most of the modern available microprocessors implement the Reduced Instruction Set Computer architecture, or RISC, which is based on fixed instruction sets. Many different types of RISC microprocessors populate the market based on these different sets of instructions including, MIPS, ARM and PowerPC to name some of the more popular. These microprocessors are realized in the form of application specific integrated circuits, or ASIC, made up of logic fixed at design time that cannot be altered after the chip fabrication process is complete.

When designing instruction sets, computer engineers attempt to capture all the instructions necessary to cover the largest space of potential applications, while keeping

in mind factors such as size, cost and power. This set of instructions form the blue print for the instruction set architecture, or ISA, to be implemented on the new microprocessor. Despite all efforts, the quest for the 'optimal fixed instruction set architecture' is an impossible one. The space of applications to which the designers apply general purpose microprocessors evolves constantly. In addition, the trends that govern this evolution shift periodically in response to changes in consumer lifestyles and demands. For example, the need to process more audio and video data has led to extensions for all of the available RISC architectures. Therefore, the selection of instructions for a 'general purpose' microprocessor designed to meet the demands of today's market place may be ill equipped to handle the applications of future markets.

We can also argue that the quest for the 'optimal general purpose' microprocessor for today's market does not make sense anymore, especially in the embedded market place. In the embedded market, system designers work within the strictest constraints of size, cost and power. The systems they design apply to a specialized and significantly reduced space of potential applications. In this context, a 'general purpose' microprocessor is inefficient and under-utilized when the majority of applications never use a large subset of the capabilities it provides. For instance, many embedded systems rarely if ever use floating point operations but many 'general purpose' microprocessors include them.

For these applications, a popular solution is to use custom microprocessors with reduced instruction sets but with customized instructions added specifically for the intended application space. This requires redesigning the microprocessor architecture

and fabricating custom microprocessors for each of the desired application domains; still they suffer from the inflexibility previously discussed. Therefore, manufacturers cannot offset the cost in design and fabrication of the new custom chip if the market for the given application is not large enough. As demonstrated in this discussion, the problem of this inflexible 'general purpose' microprocessor architecture becomes a severe hindrance. What is needed in the embedded application space and potentially in all areas of microprocessor hardware design is a new technology that can provide for flexibility and customization, allowing the microprocessors to evolve with their target markets at all stages of their life cycles.

The classical microprocessors have grown exponentially in speed and complexity while maintaining relatively constrained costs, in a fashion characterized by Moore's Law. Moore's Law states that the complexity of modern integrated circuits, or IC, with respect to cost doubles every two years. One may observe this trend as the various microprocessor manufacturers; including Intel and AMD, vie for supremacy in the market by delivering increasingly faster and more complex microprocessors. These new microprocessors include new features that augment their capabilities while maintaining fairly level costs over time. As manufacturers reach the limits of the physical constraints of the materials available in terms of speed, conductivity, size and reliability, they also reach the end of this level of growth under current microprocessor design paradigms.

The current trend in the quest for additional execution speed provides additional processing cores in the CPU and to parallelize the execution of as much of the software as possible. This paradigm has potential, but its effectiveness has been limited by the

difficulties involved in parallelizing software that was written for sequential execution, resulting in cumbersome dependencies. These dependencies require changes in the software design paradigms if this approach means to reach its full potential.

The extensions of the eMIPS microprocessor depart from the current techniques for attaining microprocessor execution speedups and flexibility in that to fully realize their benefits, it does not require any changes in software design and it is hidden from developers at the software level. To utilize the extensions, only minimal changes are required at the assembly/binary level. Performing instruction compaction using extensions in eMIPS improves performance by up to three times in most cases when compared to systems of a similar architecture without extensions.

The software design techniques for multi-core systems must become more mature before applications can take full advantage of the large-grain parallelism they offer. When this happens, the eMIPS technology can be leveraged to realize even greater speedups by providing multiple customized processor cores executing in parallel. Each processor core may additionally execute the instruction blocks that it is best suited for. If the chip area devoted to the extensions is abundant we can use the eMIPS architecture to realize a more complex and more adaptable multiprocessing system by allowing extensions to include additional complete data paths, or some other custom processor design. These additional data paths can be loaded on-demand to increase overall throughput. In this way we can convert the eMIPS into a multi-core system when the system is under heavy computational demand. When the system is lightly loaded the

extension data paths are disabled to save energy or reconfigured to be used for more specialized purposes.

The eMIPS architecture also addresses the waste associated with including functions in systems where they are never used. The extensions of the eMIPS architecture do not load when the microprocessor powers up. The fixed logic system only includes the minimum functionality (system management, reconfiguration support, load, store, arithmetic and logical functions); the extensions provide any additional functions required by the applications running on the microprocessor. For instance, the floating point, the media or vector co-processors can be loaded only if and when software applications use them. When applications do not require these functions the extensions are not loaded, providing for potentially large power savings because the unused Extension slots may have their clocking resources and power disabled to reduce the power consumption. Area savings are also possible because not all extensions need to be present at all times, as is the case instead for an ASIC implementation. This waste reduction may increase by dynamically loading and unloading not only the co-processors but also the on-chip peripherals that are part of an embedded microcomputer. Rather than including all possible peripherals in the ASIC we can load them on an extensible microprocessor as extensions, again with power and area reductions.

The standard RISC architecture lacks the infrastructure to allow for the kind of flexibility and extensibility possible through the use of eMIPS. This new extensible instruction set computer architecture provides this infrastructure. The FPGA is partitioned into fixed and reconfigurable regions. The fixed logic region constitutes the

base functionality of the processor including security sensitive resources such as the system coprocessor and the systems used by the microprocessor to control its configuration. The extensions to the base processor make up the reconfigurable region of the FPGA. The architecture provides the flexibility and adaptability lacking in the RISC architecture.

2.2 MIPS INSTRUCTION SET

The Microprocessor without Interlocked Pipeline Stages, or MIPS, instruction set [32][30] provides a good example of a modern Reduced Instruction Set Computer instruction set architecture, or RISC ISA, although it is but one of many. There are other RISC ISA available such as the ARM and PowerPC that arguably could be better than the MIPS ISA. Unfortunately, these ISA fall under proprietary controls that removes them from consideration in research projects such as this. The MIPS documentation for the MIPS R16000 is widely available and by residing in the public domain where most other instruction sets are proprietary, it makes MIPS an ideal choice for research in microprocessor architecture and design. The MIPS instruction set architecture like all fixed RISC architectures has experienced growth, modification and expansion in order to keep up with the changing needs of the microprocessor market. The architecture is currently in its ninth generation with the MIPS R16000 [32]. For this reason, the MIPS R4000 was chosen as the basis of the eMIPS processor. In order to consider a device a MIPS microprocessor, all the basic instructions must be implemented. Software compiled using this base instruction set must run on the eMIPS microprocessor and

existing compilers for high-level programming languages that target the R4000 should remain usable without change.

The MIPS architecture is pipelined to provide improved throughput at higher clock frequencies. One can increase the clocking frequency by breaking the instruction execution within the processor into stages that require less time to execute. Each stage of the pipeline works on different instructions at different times and then passes the instruction to the next stage to continue execution. After the pipeline is full, the microprocessor completes execution of one instruction per clock cycle just like the non-pipelined version except at a much higher clock rate. This architecture results in a net increase in throughput despite the overhead that the architecture experiences with flow control, hazards and exceptions. The time to fill the pipeline is negligible.[30]

The classic implementation of the RISC pipeline architecture [30] includes five pipeline stages: Instruction Fetch (IF), Instruction Decode (ID), Instruction Execute (IE), Memory Access (MA) and Writeback (WB). The functions of these stages are as follows:

- Instruction Fetch (IF) – Update the program counter, or PC, and fetch the instruction located in memory at the address stored in the PC. [30]
- Instruction Decode (ID) – Using wired logic, decode the instruction passed from IF into control signals that control the remainder of the pipeline. Read any data required by the instruction from the general purpose register file. Test branch conditions and calculate the memory location of the next instruction to be executed if a branch is taken. [30]

- Instruction Execute (IE) – Using an Arithmetic Logic Unit, or ALU, and other special purpose logic perform operations on data based on the control signals passed from ID. [30]
- Memory Access (MA) – In case of a load or store instruction, the output of IE is used as the memory location to be read from or written to. Otherwise, the output of IE is passed through. [30]
- Writeback (WB) – In the event a register in the general purpose register file is modified by the instruction, the output of MA is written to the desired register. [30]

To realize greater throughput at higher frequency, some microprocessor implementations have utilized as many as eight pipeline stages. The deeper the pipeline is the greater the overhead in the event of a branch event, hazards and exceptions on execution. To further offset this overhead, microprocessor designers have developed a variety of features, including branch predictors and speculated execution. These features have highly complex implementations and exist beyond the scope of this project. For this reason, the architecture of the eMIPS microprocessor omits these features and has just the basic five pipeline stages. [30]

2.3 FPGA ARCHITECTURE

The Field Programmable Gate Array, or FPGA, is a digital semiconductor device often used for prototyping. Developers use FPGA in prototyping for their ability to configure or program their electrical interconnects to realize an expansive space of

applications from glue logic to application coprocessors. As the name indicates, developers are free to apply modifications to the design implemented on the FPGA in the field, after deployment. Developers synthesize the configurations from a Hardware Description Language (HDL), like Verilog or Very High Speed Integrated Circuit Hardware Description Language (VHDL), for the targeted FPGA device. Eventually, the configuration file downloads to the chip through an interface such as Joint Test Action Group, or JTAG. This flexibility comes with a price. The configurable logic of the FPGA experiences significantly lower performance, currently clocking at frequencies up to 500 MHz, than the modern ASIC, currently clocking at frequencies over 3 GHz.[49][51][58] Despite this limitation, FPGA technology has evolved to the point where developers can implement fixed logic microprocessors with performance levels competitive with their ASIC counterparts in the embedded market. These future microprocessors will have the advantage that they can be dynamically updated after deployment to meet new demands. This approach to microprocessor design leads to a new class of microprocessors termed the “dynamically extensible microprocessor”.

Some of the new ‘state of the art’ FPGA, which have come to market in recent years, have added features that further augment their flexibility and power. First, modern FPGA, such as the Virtex series from Xilinx Inc. [49][51][58] and the Stratix series from Altera Corp.[2][4], have included a feature called ‘dynamic partial reconfiguration’. FPGA that include this feature may have their configurations partitioned and later allow individual partitions within the design to continue operations while a new configuration downloads to another partition.[56][55][52][57] FPGA

configuration solutions allow for FPGA to be configured at runtime by other devices or by themselves. An example of one of these solutions is the System ACE Compact Flash solution. This is an IC chip set that provides five interfaces: JTAG to host workstation, JTAG to FPGA, external to a Compact Flash chip, and Control from either the microprocessor or the FPGA. Using these interfaces a system can configure one or multiple FPGA using the JTAG from a workstation or reading a configuration from the compact flash and streaming it to the FPGA on the JTAG chain.[54]

Using modern FPGAs it is possible to partition the FPGA into sections containing a standard fixed logic processor core with interconnects to blocks termed extensions. These extensions contain customized instructions. To use these extensions the processor core includes functionality that loads, modifies and enables these extensions while the fixed logic continues to function without interruption. In this way, the dynamically extensible processor, using a library of extensions from which it can draw, adapts to the changing application needs in the field. Using the configuration solution with hardware and software support allows for self extension of the system. By using these dynamically reconfigurable FPGA, an “extensible central processing unit” becomes possible. The eMIPS project described in this document argues that such a device is feasible today and proves this thesis by means of an example prototype implementation.

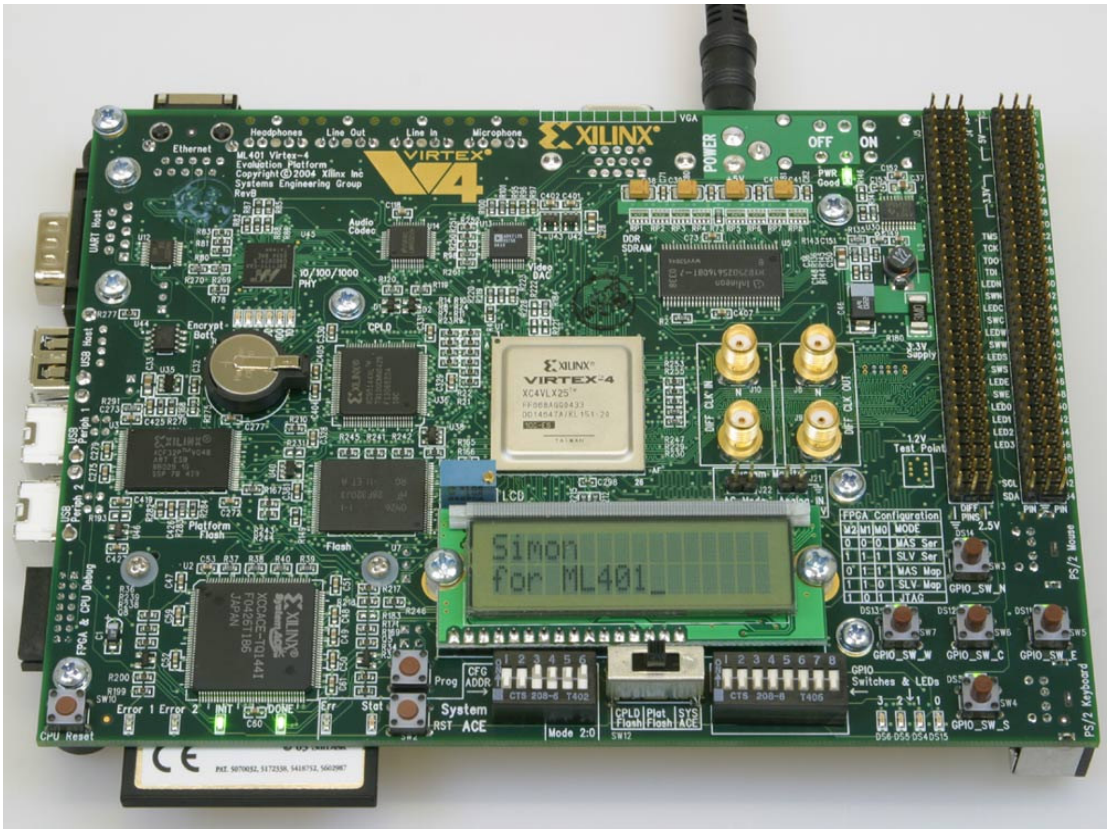

Fig. 1. Xilinx ML401 Evaluation Board with Virtex 4 LX25[51]

The FPGA selected for the development and experimentation of this project is the Xilinx Virtex IV product line. The Virtex series rates among the most powerful FPGA devices in the market in terms of density, feature set and speeds. These FPGA clock commonly at frequencies of 100 MHz but their specifications indicate they could operate at much higher frequencies. The specifications claim 500 MHz. These frequencies fall significantly short of modern ASIC frequencies approaching multiple gigahertzes but the FPGA continue to grow in speed with each new generation. The

Virtex IV FPGA comes in three distinguishable types denoted by LX, SX and FX. Each flavor includes a set of special features to allow developers to select an FPGA with the feature set that best fits their application domain. The Virtex IV LX targets logic design applications. For this reason, the Virtex IV LX provides the largest number of logical blocks for implementation. Given the floor planning requirements of partial reconfiguration, having more logic area to work with is preferred. Therefore, the implementation of the eMIPS processor targets the Xilinx ML401 Evaluation board with the Virtex LX25 shown in Fig. 1.[51][58][49][60]

2.3.1 PARTIAL RECONFIGURATION OF XILINX FPGA

Xilinx has supported partial reconfiguration since its Virtex FPGA and that feature continues in the more modern Virtex IV and Spartan III. [56][55][52] Despite the continued support of this feature on their hardware, Xilinx continues to under-emphasize this feature in the tool suite, the ISE Foundation. Xilinx recently released the latest release of the ISE, release 8.2i.[49] In the past, hardware designers performed the partial reconfiguration design flow using command line instructions to the tools in the ISE. Xilinx did not integrate the design flow into the graphical user interface, or GUI of the ISE Project Navigator. This results in a large amount of tedious repetitive steps one must perform to run the design flow. Project management and organization becomes crucial, a large set of files is required for each stage of the design flow. In the last year, Xilinx made a greater effort to provide tool support for partial reconfiguration and to make it more accessible to developers. A new tool that Xilinx has developed is called

Planahead and attempts to provide improved floor planning utilities and to integrate the partial reconfiguration design flow into a project structure managed by a GUI. Planahead currently has limited availability in an advanced access beta program. [56][55][52]

The smallest reconfigurable unit of the FPGA configuration fabric is called the 'frame'. When partitioning the FPGA into different independently reconfigurable and static regions the boundaries between these regions must coincide with the boundaries of these 'frames'. Multiple frames may be grouped together into a single rectangular region. Regions cannot be smaller than a 'frame'. In the Virtex and Virtex II architectures a frame constituted a column of logic cells called slices that spanned the height of the chip. In the Virtex IV, a column of sixteen slices makes up the 'frame'. In this way, each column of the Virtex IV contains multiple frames. In the LX25, which has 192 rows of slices, each column contains twelve frames. This architecture provides the Virtex IV the advantage of allowing for rectangular regions in the form of tiles on the FPGA configuration fabric as opposed to strictly columns as in the previous architectures.[56][55][52]

Hardware designers must also consider the routing of signals crossing the boundaries of the various regions. Only the region containing the reconfigurable module changes when reconfiguration occurs. The remaining configuration fabric remains unchanged. Therefore any inconsistency from one configuration to the next will result in unpredictable results.

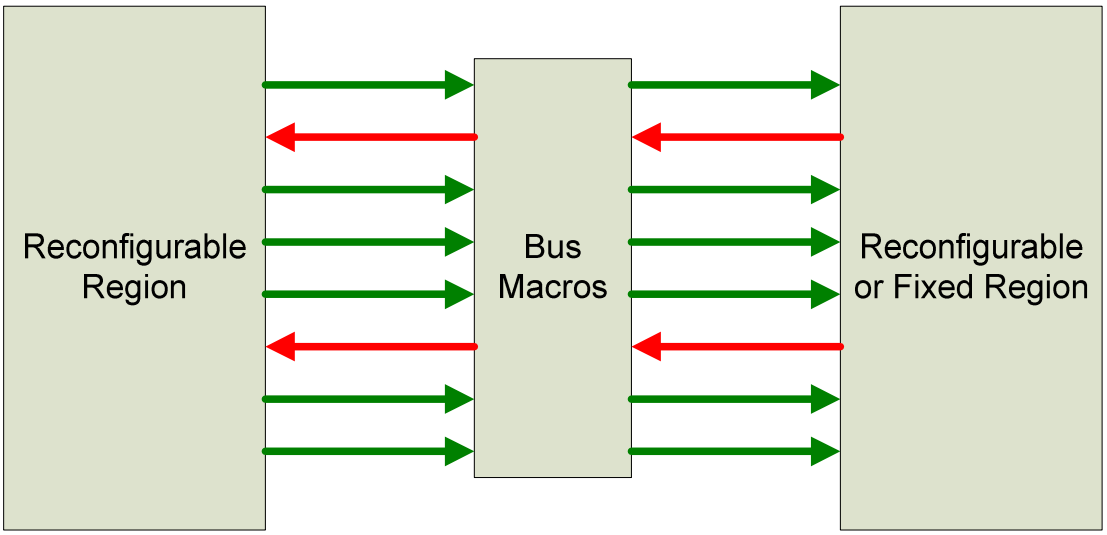

Fig. 2. Logical Connections of Signals Across Region Boundary [52]

One potential inconsistency can occur when a signal crosses the module boundaries. Consider for instance the case of a signal that crosses the boundary and in one configuration the signal routes through row four but the same signal is routed in row five in another configuration. When the system undergoes reconfiguration, the signal will not line up on the boundary where the reconfiguration occurred, therefore cutting the signal. To prevent this inconsistency we can restrict the routing of such signals to fixed locations along the region boundaries. This is done by passing the signals through a 'bus macro' or a hard pre-routed macro as seen in Fig. 2. The bus macros are positioned on the module boundary and force the routing program to route the signal through a given location in each configuration. For the eMIPS processor, bus-macros are

placed between the interfaces of the fixed instruction set logic and the dynamic extensions. [56][55][52]

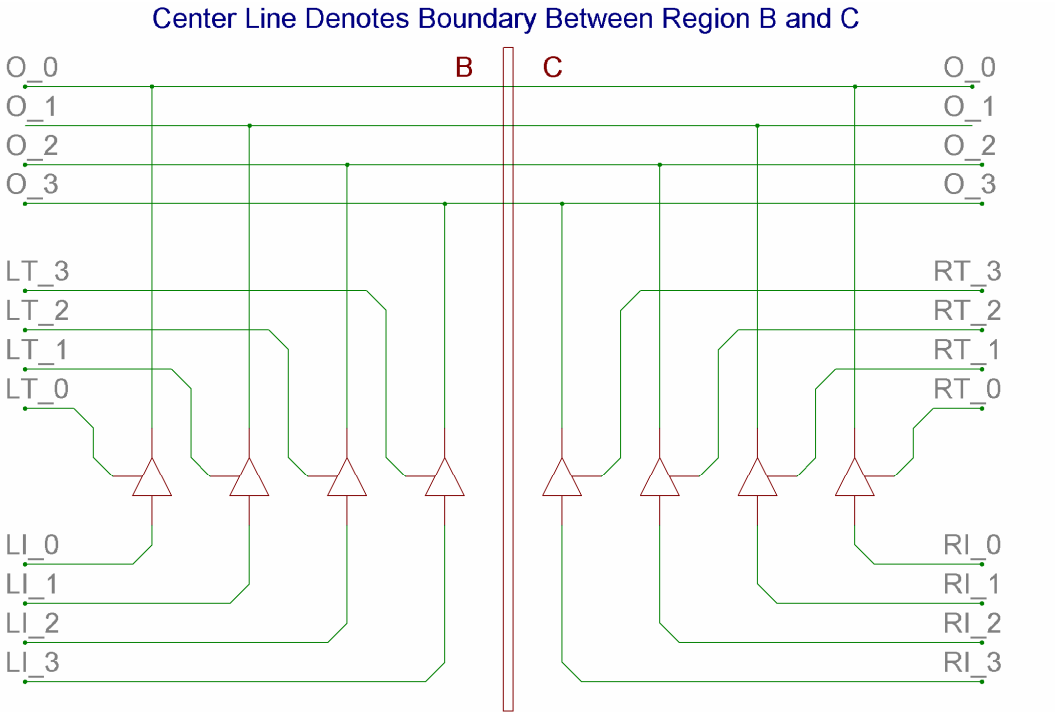

Fig. 3. TBUF Based Bus Macro [56]

In the Virtex and Virtex II architectures Xilinx provided ‘bus macros’ based on tri-state buffers, or TBUF. Fig. 3 shows an example of a TBUF-based bus macro used to interface two modules, module B and module C. TBUFs are not included in the Virtex IV and Xilinx provided no alternative ‘bus macros’ when the device was released. Researchers intending to do partial reconfiguration created ‘bus macros’ of their own

during this time using the Xilinx FPGA Editor.[53] Most of these bus macros ended up being based on look-up tables, or LUT. Researchers and developers would use the FPGA Editor tool available in the Xilinx ISE to route and generate these hard macros. Fig. 4 shows an example of a LUT-based bus macro.[56][55][52]

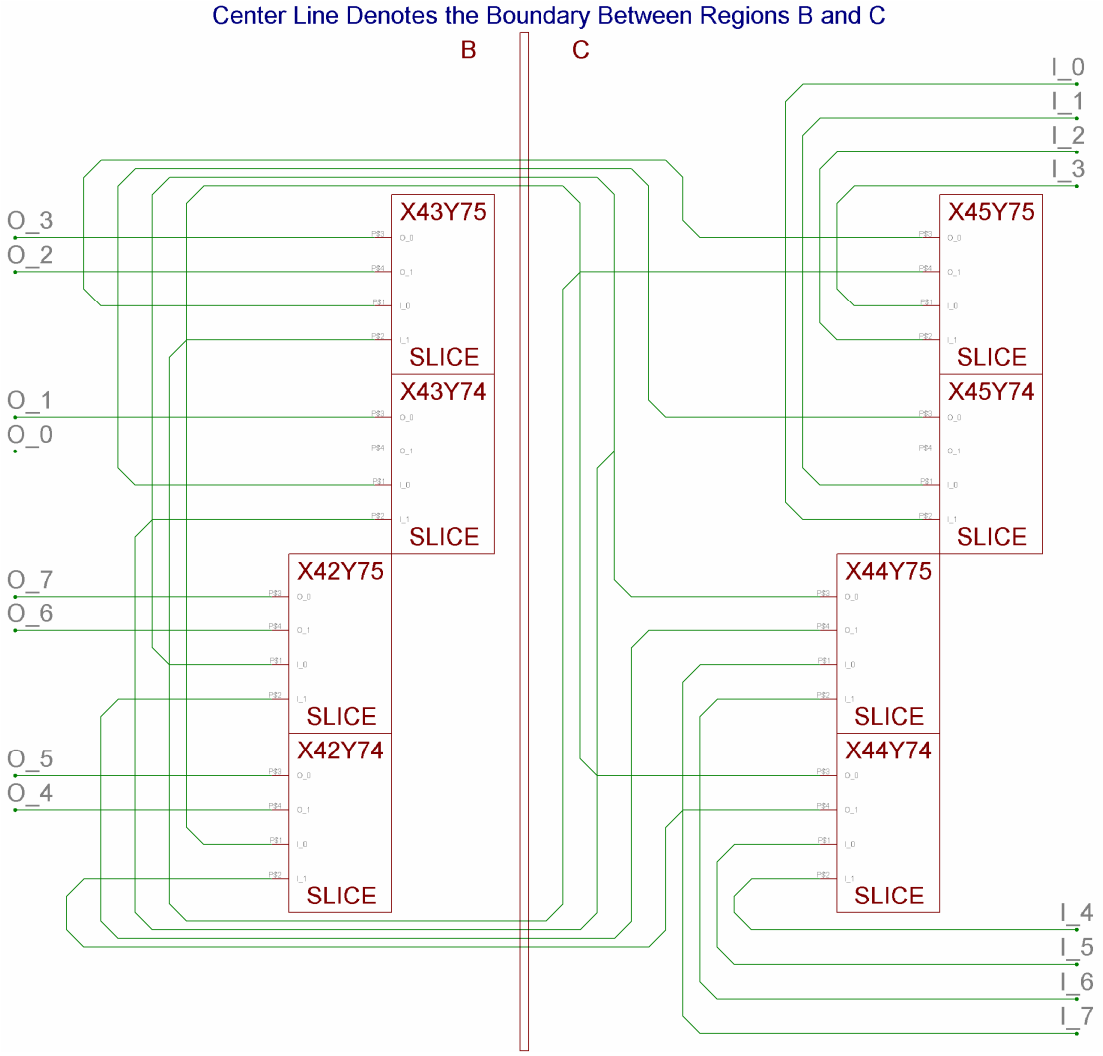

Fig. 4. LUT Based Bus Macro [52]

When Xilinx released Planahead in beta, they also released LUT based 'bus macros' for all their products including the partial reconfiguration feature. Planahead takes the required routing consistency a step further by recording the routing of all fixed logic that passes through reconfigurable regions in a routing database. Planahead incorporates these routing patterns in the place-and-route phase of compilation, so that the reconfigurable regions will maintain consistency. [56][52]

The partial reconfiguration design flow includes four phases as documented by Xilinx. These phases are Design Entry, Initial Budgeting, Active Module, and Final Assembly. Full details can be found in [52][55][56]. The following is a brief description of each phase:

- Design Entry – This phase involves setting up the project by targeting the desired FPGA device, decide on design partitioning and performing some design planning. Before Planahead, this phase also included manually setting up the project directory structure. Planahead now handles this in project setup. In large projects including multiple engineers, this phase is usually carried out by the team lead. [52][55][56]
- Initial Budgeting – In this phase the design engineers write the top level module and implementation constraint files. The constraint files include information such as pin assignments, area definitions, assignment of modules to areas and clocking constraints. The top level module defines the ports of the design and instantiates

all second level modules and defines their interfaces to each other and to the system ports. This top level should be minimal in its contents. There should be as little logic in this layer as possible and contain only the modules that will be implemented at this layer. Any top level logic that is present goes through place and route and this data is written to the routing database for future use. In most cases, a team lead also carries out this phase. [52][55][56]

- Active Module – Design engineers execute this phase of the design flow for each module instantiated in the top level in parallel. The team lead assigns hardware designers to implement the different modules using the interface outlined in the top level written in the previous phase. In the case of reconfigurable modules, hardware designers implement two or more versions of this module. In some cases, designers write module level constraints into the implementation constraint file. The Planahead tool synthesizes each module independently of the rest of the design and performs place and route within the region designated for it while taking the contents of the routing database into account. [52][55][56]
- Final Assembly – This is the final phase of the design flow. In this phase, the team lead collects the module implementation of each module from the hardware designers and uses Planahead to integrate them together. The team lead creates a floor plan of the system for each possible configuration or combination of modules. Using these floor plans Planahead completes any additional place and route required and generates configurations files for the desired default

configuration and other configuration files for the reconfigurable regions that change dynamically. [52][55][56]

We used these phases in the eMIPS project. Through Planahead the eMIPS processor was synthesized, floor planned, components placed, signals routed, and configuration files generated.

2.3.2 SYSTEM ACE COMPACT FLASH SOLUTION

The System Advanced Configuration Environment, or System ACE [54], attempts to fill a niche for pre-engineered configuration solutions of multiple FPGA systems. The system applies to the eMIPS microprocessor's need to control and modify its extensible microprocessor architecture. System ACE works through the interaction of four interfaces: JTAG to host workstation, JTAG to FPGA, compact flash and control from a microprocessor or FPGA. Using the host JTAG interface a configuration file can be downloaded manually to the system and used to configure one or multiple FPGA. This feature is excellent for debugging. It allows the developer to download test configurations and run code before including the new configuration in the system. When configuring the system from the host JTAG the System ACE reads the bits stream on the host interface and transfers it to the system JTAG chain it controls. After the configuration design completes and the system is ready for deployment, system controlled configuration can be performed via microcontroller or FPGA control. In the case of a single FPGA system, like the eMIPS microprocessor, the microprocessor interface can be integrated in the FPGA to allow it to control its own configuration. The

compact flash is a portable, permanent storage device that stores the configuration files and inserts into a reader integrated with the System ACE. Using the control interface the FPGA or microprocessor can initiate configuration of the system by selecting a configuration file stored in the compact flash that the System ACE drives on the system JTAG chain. The System ACE also provides an interface similar to Integrated Drive Electronics, or IDE, disk interface commonly found on workstations to allow the controller to read and write to the compact flash. [54]

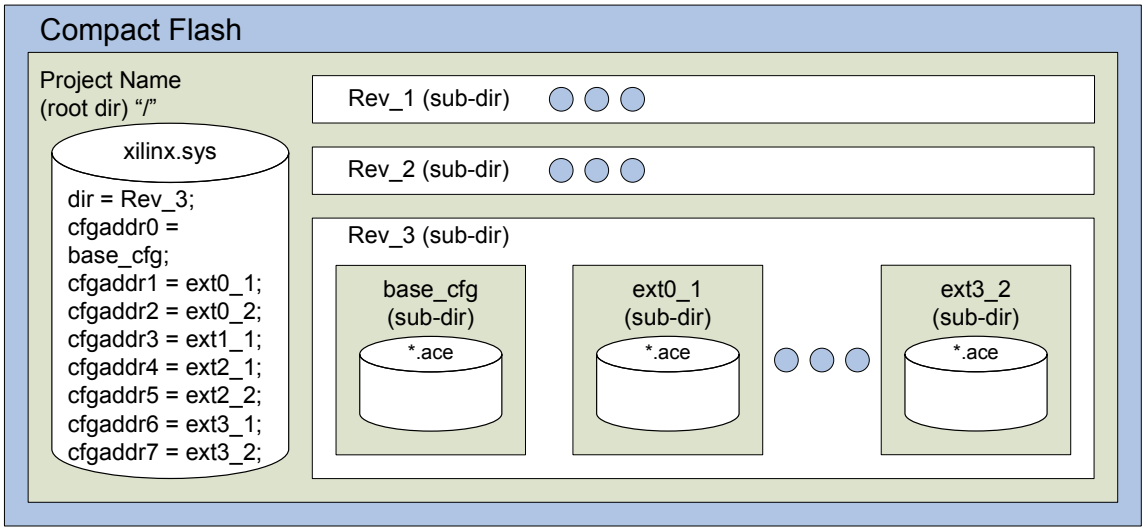

Fig. 5. System ACE File Structure [54]

The System ACE controller interface provides a three bit configuration selection input to allow the controller to select one of eight potential configurations. Note that the compact flash can store more than eight configurations, as illustrated in Fig. 5. The

configurations are grouped into sets of no more than eight and placed in directories on the compact flash. In the root directory there exists a file called ‘xilinx.sys’. This file tells System ACE which directory containing configuration files should be considered ‘active’. The controller can only use configuration files from the ‘active’ directory. The ‘xilinx.sys’ file also assigns to each file the numerical designation zero through seven for the configuration selection. To change which set of configurations is considered active, one must change the assignment in the ‘xilinx.sys’ file. System software can do this dynamically using the IDE interface to the compact flash.[54]

2.3.3 eMIPS AND THE FPGA

This implementation of the eMIPS microprocessor uses the dynamic partial reconfiguration feature of the Xilinx FPGA to implement the dynamic loading of extensions to the core microprocessor architecture. During the design phase, the core microprocessor architecture resides in the fixed logic region of the FPGA configuration. The configurable partition(s) constitute the area dedicated to the loading of extensions after the processor has begun operation. After design implementation completes, the process produces a binary file containing the default configuration. Later the process generates the binary configuration files used to alter the configuration of the extension slots. The process uses the default configuration and the hardware description of the extension as a starting point. At power up, the default configuration contains the implementation of the core microprocessor architecture and the region allocated for the extensions is empty, or actually contains the minimum logic needed to prevent the

synthesis tool from removing it during optimization. When the microprocessor starts an application that requires an extension, it loads the extension using a JTAG chain provided by the System ACE Compact Flash solution.

The System ACE Compact Flash solution provides the functionality needed for the Xilinx FPGA to control and change its configuration. The System ACE Compact Flash solution includes a JTAG configuration chain capable of dynamically configuring the extensions of the microprocessor. The microprocessor stores the default configuration and the extensions in the compact flash card interfaced to the System ACE chipset. At power up, the System ACE Compact Flash solution reads the default configuration from the compact flash card and streams it on the JTAG chain implementing it on the FPGA. The microprocessor running on the FPGA requests changes to the extension configurations by requesting the loading of a configuration from System ACE through the microprocessor control interface. In this event, System ACE reads the requested extension configuration from the compact flash card. Then like the default configuration at power up, System ACE streams the configuration on the JTAG chain. In the case of a partial reconfiguration, the signals on the JTAG chain only modify the region of the FPGA where the extension is located. The remainder of the FPGA continues to function normally. Consequently, the microprocessor may continue to execute instructions while the configuration process alters the extension configuration. In this way, the microprocessor powers up using the default configuration and modifies its own configuration using the partial extension configurations stored in the compact flash card.

2.4 PROFILING AND IDENTIFICATION OF BASIC BLOCKS

While CPU designers seek generality in their designs, all application programs spend most of their execution time in small sections of the code that make up the executable file image. This observation holds regardless of platform or application, from personal computers to embedded systems to entertainment consoles and devices. Analysis of software execution profiles revealed that in many cases the two to three most executed sequences of code in the applications program account for more than 80% of the total instruction execution count. Based on this observation, if we can somehow optimize the execution of these few select sequences we can attain an overall improvement of the performance of the entire application.[21]

These sequences are termed the “basic blocks” of the software application. Technically, a basic block is a sequence of instructions that ends in a (conditional) branch and is not branched-to anywhere but at the first instruction. In our work, the basic blocks are identified using the tools distributed with Microsoft Giano [37][22]. Giano has been used internally at Microsoft Corporation for system verification for some years now. The profiling tool outputs a database of basic block and basic block patterns that manifest in the (set of) application program(s). Roughly speaking, a basic block pattern is the set of all basic blocks that perform the same function but differ in their register assignments or in the embedded constants. Once the database is generated, it is updated by the simulator each time the profiled application is run. Each basic block is uniquely identified by a hash value; there are no duplicates in the database. Each entry also

contains counters for the static and dynamic repetition counts of the basic block. The static count indicates how many times the basic block is repeated in the application binary itself, or possibly across more than one binary, according to the user's preference. The dynamic count is maintained by the execution simulator and counts the times the basic block has been actually executed during one or possibly more executions of the application. It is possible to obtain the distribution of dynamic counts against time by check pointing the database with a certain frequency. This can capture the behavior of programs that exhibit "phases" during their execution.[21]

In the practice, the basic block sequences are given to hardware design engineers as specifications. The hardware engineers create a hardware module that implements the semantics of that sequence in the most optimized way possible and in a manner that conforms to the extension interface of the microprocessor. Automatic generation of the extension appears feasible [61][25] but it is beyond the scope of this project and we will assume manual generation. The engineers create the design for the extension and generate the FPGA configuration file. The configuration file integrates into the application software package. When the application starts up on a platform that supports it the extension is loaded into a free extension slot, provided one is available. If the platform does not support extensions, or if the security settings of the platform disallow it or if the configuration file is damaged the extension instruction is ignored and the basic block sequence executes normally. The extension interfaces and configuration file generation are discussed in Section 4.2.3.

The implementation of the extension, including its use in the application, is abstracted or hidden at the software level. No change to the higher level compiler is necessary. After the basic block is identified, implemented and encoded into an extension instruction, another tool independent of the compiler augments the executable image. The tool scans the assembly binary image for occurrences of the basic block pattern identified in the profile database. The tool encodes the new extension instruction to match the register assignments and constants, and inserts the new extension instruction immediately before the instance of the basic block pattern. In this way, the software developers are not aware of the hardware acceleration and need not be aware of it. [21]

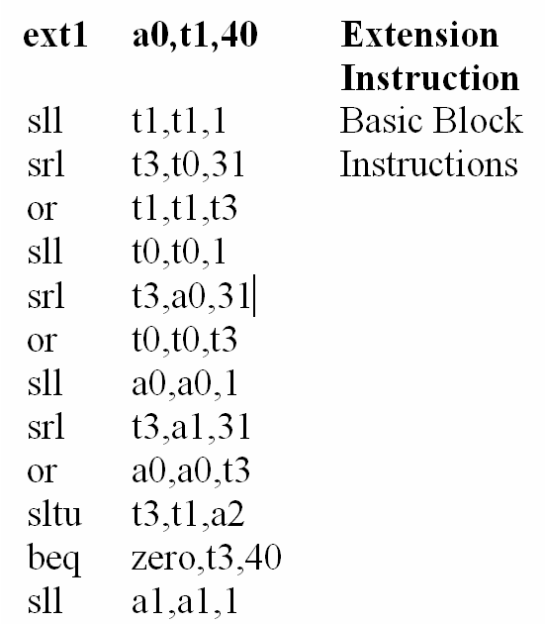

Fig. 6. A Basic Block Augmented with an Extension Instruction

Fig. 6 shows the basic block that was found to be the most frequently executed during testing of the Giano profiling and simulation tool. This basic block was the dynamically most frequent one; it was not the statically most frequent. It was only through actual execution profiling that we found it to be the best candidate for optimization. Simple inspection of the binary code base pointed to a completely different basic block. The first instruction in Fig. 6 is the extension instruction, inserted before the basic block itself. The basic block is part of a software implementation of a 64-bit division. It shifts left by one the 128-bit number contained in the register quad t0-t1-a0-a1, then makes a conditional branch depending on register t1 being greater than register a2. [21]

The extension instructions must conform (at least in part) to the instruction format restrictions of the ISA from which they are derived. In the case of the MIPS ISA, instructions consist of a 32-bit value including an op code, up to two operand registers and a destination register or data immediate. If this single instruction must replace a potentially long sequence of several instructions with as many as two operands per instruction, one destination and data, the question becomes how to encapsulate all that information into a 32 bit instruction. One way is to leverage the relationships that exist between operands, destinations and immediate values. For instance, if the register operand of one instruction is the same as the destination of a previous instruction it only needs to be encoded once. Scratch registers need not be encoded, for each instance in Fig.

6 we can skip the scratch register t3. Notice though that in the implementation of the extension care should be taken to maintain the same semantics as the original sequence and to write the new value to the register file even if nothing else uses it.

When registers differ, the relationships between register numbers can be built into the instruction decoding phase of the extension. For instance, if there are two registers used by different instructions in a sequence but the second is always one away from the first, the extension designers only require one of the register numbers to encode both of them. In Fig. 6 this is the case both for t0-t1 and for a0-a1-a2. It is up to the binary patching tools to verify that these constraints are met. Encoding of destination registers can be performed in the same way. In the case of immediate values, if the value is the same every time the block is executed, then this may be encoded directly into the extension. This is not the case in the example of Fig. 6 and the immediate field must be used, reducing the number of available slots for register numbers by one. [21]

Similar relationships can be identified amongst operand registers to reduce the number of bits (register numbers and the like) needed to encode the required data in the instruction format. If hardware designers cannot reduce the required register numbers to two operands and one destination they could violate the ISA rules and use any of the bits other-than the op code as they see fit. The only penalty is that a disassembler will not be able to provide any meaningful decoding of the extension instruction. Yet another possibility is to further break up the sequence into two or more extension instructions. More information, including the means to automatically identify patterns of instruction sequences and applying such patterns to modify executable binaries is presented in [21].

Using the eMIPS means of execution acceleration requires no change in software design or practice. Modifications are applied, after the software development process is complete, to the finished product. This is in sharp contrast to multi-core parallel systems that require parallelization of the software design at the highest levels to benefit from the hardware feature. Future business models for platform manufacturers that utilize these microprocessors include independent services to profile and augment application software binaries with an extension to fully utilize hardware acceleration. [21]

3 DYNAMICALLY EXTENSIBLE PROCESSORS

The eMIPS processor is a ‘dynamically extensible microprocessor’ because it is based on a new, extensible architecture. The architecture is extensible because it allows additional logic to interface and interact with the basic data path at all stages of the pipeline. The additional logic, which we term extensions, can be loaded on-chip dynamically during execution by the microprocessor itself. The architecture therefore possesses the unique ability to extend its own ISA at run-time. To explain this more in detail, we will first describe the ‘classical’ RISC CPU architecture in Section 3.1 and then show where the eMIPS microprocessor architecture departs from it in Section 3.2 and Section 3.3. The overall functioning of a complete system based on an eMIPS microprocessor is then described in Section 3.4.

3.1 THE ‘CLASSIC’ RISC CPU

Fig. 7 presents a block diagram of a ‘classic’ RISC CPU organization, including five pipeline stages, a general purpose register file, a memory interface that includes the interface to the peripherals and a system coprocessor. The five pipeline stages include Instruction Fetch (IF), Instruction Decode (ID), Instruction Execute (IE), Memory Access (MA) and Writeback (WB). The stages are as described in Section 2.2.

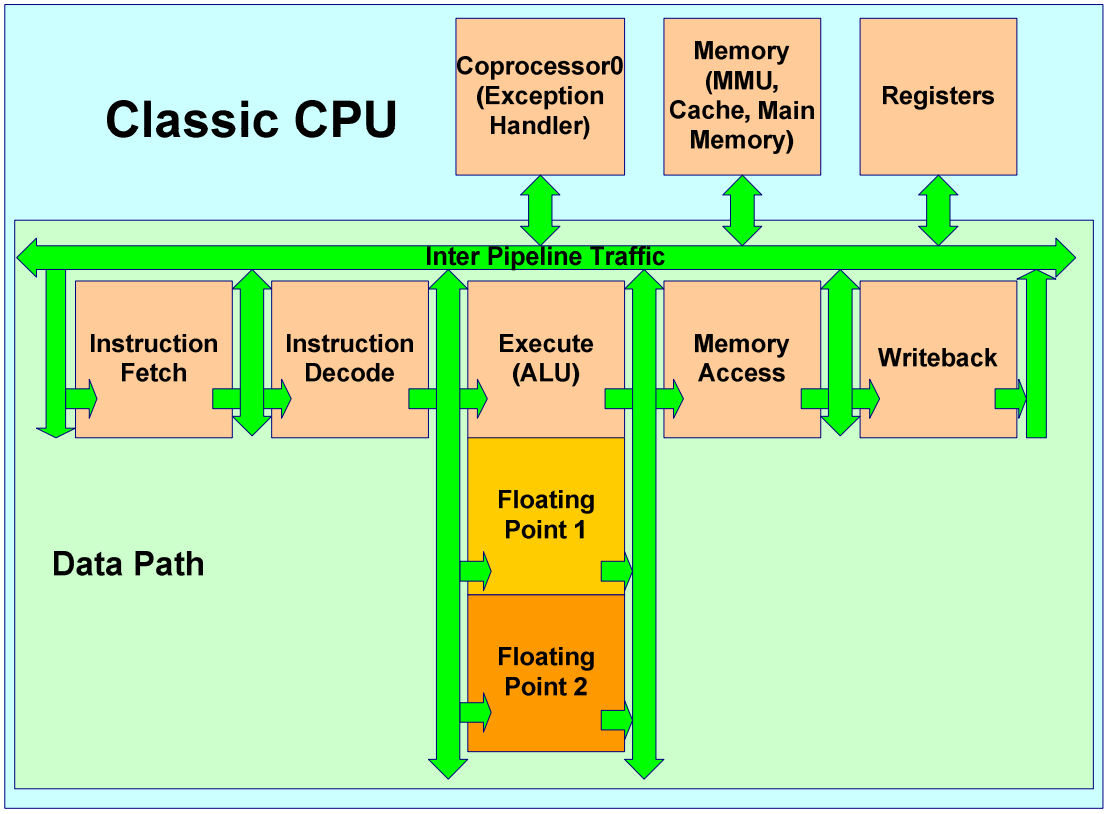

Fig. 7. Block Diagram of a Typical Pipelined CPU Architecture

It is important to note the inclusion in this organization of two Floating Point Units, or FPU, depicted as operating in parallel with the IE block. Execution of floating point arithmetic and other coprocessor operations require considerably longer execution times to complete than most integer operations performed by the Arithmetic Logic Unit, or ALU, inside of the IE block. These FPU operate on data in register files that are independent of the general purpose register file used by the rest of the CPU. Independence of the data in these units removes potential conflicts and dependencies,

and allows these functional units to execute in parallel with the rest of the CPU. Parallel execution limits the latency effects of these floating point operations to those tasks dependent on their outputs. The presence of two FPU in the diagram denotes the established practice of including multiple instances of functional units on a single CPU to achieve a higher rate of instruction throughput. Several other functional units available in modern high-performance microprocessors have been omitted from this diagram for simplicity. These functional units go beyond the scope of this project and have been omitted from the design.[30][32]

Fig. 7 depicts the FPU using different colors to signify that it differs from the other blocks in important ways. In the first place, unlike the other blocks in the diagram the CPU does not require these units to function correctly. Floating point operations could be performed in software, using the ALU, although at a significant execution time penalty. Many applications in embedded systems never use floating point operations and it is fairly common to omit these functional units from simpler microprocessors. The omission results in smaller chips, lower power and reduced costs for the embedded market. In the second place, system software has the capability of disabling access to these functional units when switching between software tasks and of restricting use of these units to particular tasks. When restricting access to the FPU the state of the FPU is preserved across context switches for the benefit of that particular task. This eliminates the swapping of the FPU register file in and out to memory and increases overall system performance. [30][32]

3.2 THE eMIPS ARCHITECTURE

Fig. 8 presents a block diagram of the eMIPS CPU organization. The pipeline stages, general purpose register file and memory interface match those depicted for the 'classic' CPU and are depicted in lighter color in the diagram. These pipeline stages constitute the Trusted ISA or TISA, the core portion of the architecture that is required for initial operation and to provide a level of trust in the functioning of the CPU. These blocks cannot be removed or disabled and must be present at startup of the system. These blocks constitute the fixed partition of the architecture and include all resources that are of a security sensitive nature, such as the system coprocessor. The TISA also includes all the facilities for self-extension, including instructions for loading, unloading, disabling and controlling the unallocated blocks in the microprocessor. At a functional level the pipeline blocks operate similarly to the 'classic' CPU, except their interconnections with respect to each other and other blocks differs. Their implementations differ as well and this will be explained later.

In place of the FPU, Fig. 8 shows two sets of blocks labeled "Extensions". These extensions distinguish the eMIPS architecture from the established RISC architecture from which it is derived. Through the extensions the eMIPS CPU overcomes two major shortcomings of the RISC architecture; inflexibility and inability to evolve with changing needs. Using the partial reconfiguration design flow described in Section 2.3.1, the processor is partitioned into fixed and reconfigurable regions. The TISA is included in the fixed region; the extensions are included in the reconfigurable regions and are interconnected with the TISA by means of the bus macros described in Section 2.3.1. By

implementing different extensions for the reconfigurable regions, it becomes possible to adapt the functionality of the CPU. The CPU may apply these adaptations after deployment, dynamically while the applications continue executing.

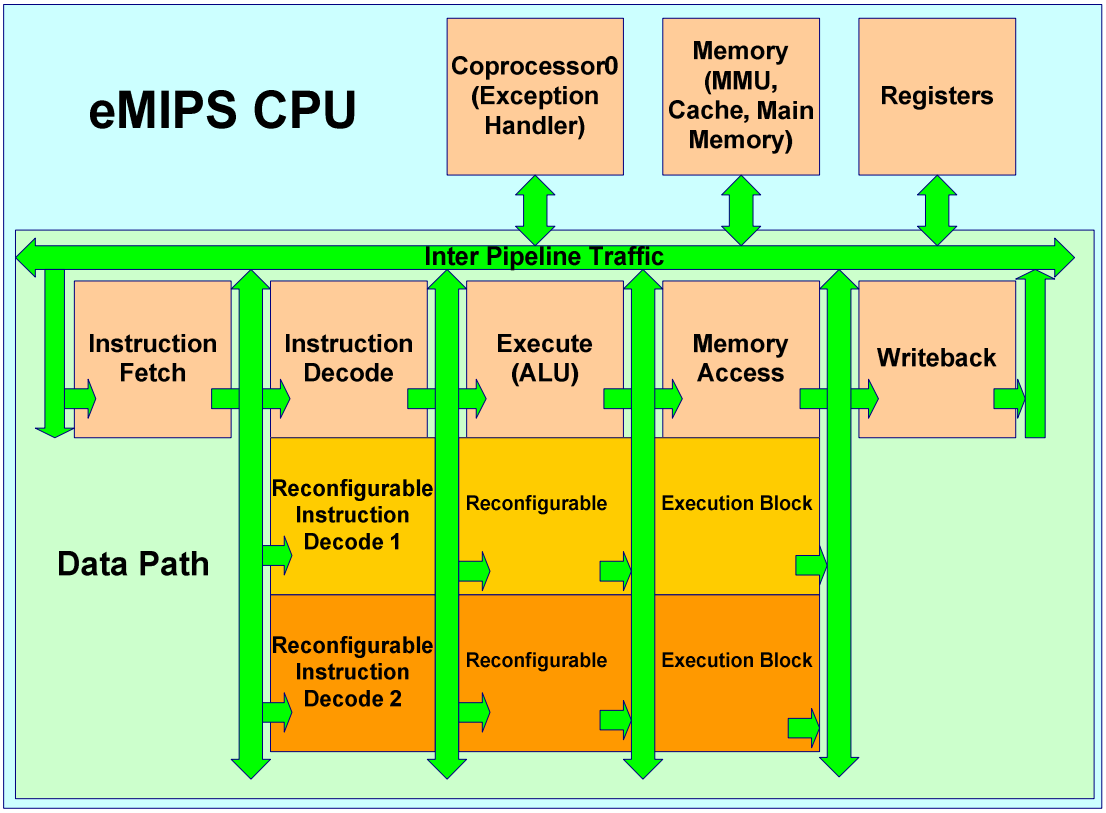

Fig. 8. Block Diagram of the eMIPS Architecture

Examples of possible extensions include but are not limited to FPU, Digital Signal Processors, or DSP, encryption coprocessors, vector processors and the

application specific instructions of Section 2.4. Using application execution profiling, engineers identify the extension instructions and implement them as hardware modules synthesized for the target device. More than one extension instruction might be included in a single extension. A successful implementation of an extension instruction runs in fewer clock cycles than the original instruction sequence it replaces. If the instruction is executed a sufficient number of times, even a single clock cycle reduction in execution could significantly improve performance.

Let us compare an extension with the FPU available in the 'classic' architecture. In the first place, in the eMIPS context a FPU is indeed implemented as an extension. The second difference between the FPU and the extension is that the extension is not available as a CPU resource at power up, because extensions are only loaded and unloaded dynamically during execution by the TISA. A third difference is that the blocks of an extension overlap with ID, IE and MA whereas the FPU only overlap with IE. The extension blocks must overlap with ID in order to recognize their instructions. The extension may not require access to memory and therefore can extend into the MA block of the pipeline as well. In this way, if an extension instruction only requires two clock cycles to complete but does not access memory, no stall is necessary and it can pass its outputs to WB to update the necessary registers without creating any pipeline bubbles.

The diagram of Fig. 8 depicts only two extension blocks but more can be included, depending on space and other limitations imposed by the physical chip. If

Moore's Law continues to hold we can project that tens and possibly even hundreds of extensions might be available in future chips.

3.3 EXECUTION DATA PATHS: MIPS VS. eMIPS

In the case of the 'classic' RISC CPU architecture, the IF block fetches the instruction indicated by the current value of the program counter, or PC. That instruction passes to the ID block that decodes the instruction into the appropriate control signals for the remainder of the pipeline. If the ID does not recognize the instruction, the ID throws a Reserved Instruction, or RI, exception to the system coprocessor. The ID also calculates the next PC based on the current PC and the instruction being decoded. In the case of a non-branching instruction, the next PC is the current PC plus four. In the case of a branch, the ID tests the branch condition. If the branch condition is true, the next PC is the current PC plus an offset, otherwise it is the same as a standard instruction. In the case of a jump, the next PC is the current PC plus an offset or the content of a register. For all instructions, the ID fetches operand data from the general purpose register file. Using the control signals and data from the ID block, the remainder of the pipeline executes the decoded instruction. In the case of an operation instruction, the IE block modifies the operands fetched by ID using the operation indicated by the instruction decoding. In the case of a load or store instruction, the IE calculates the address to be loaded or written to and passes it to the MA block. Using the address calculated by the IE, the MA loads the contents of that address from memory or modifies it using data read from a register in ID. For operation instructions, the MA block passes the result of

IE directly to WB. Finally, if the instruction modifies a register, WB writes the new value of the register to the general purpose register file.

The eMIPS follows a similar execution path for instructions included in the TISA. For instructions not in the TISA, the processor departs from this execution path in the ID phase where the ID blocks of the extensions operate in parallel with the one in the TISA. After the IF block fetches the current instruction from memory, the instruction is sent in parallel to the ID blocks of the TISA and of each extension. Each ID of the extensions attempts to decode the instruction in parallel. If the TISA recognizes the instruction, the execution path is the same as the previous example of the 'classic' CPU. In the case that none of the ID blocks recognize the instruction, a RI exception will be throw to the system coprocessor like the 'classic' CPU model.

If one of the extensions recognizes the instruction, its ID requests to take over execution of the instruction. The arbitration logic sends a NOP to the TISA pipeline stages and to those of the other extensions. The arbitration logic also passes control of the read ports of the general purpose register file to the ID of the extension that recognized the instruction. The ID of the extension that recognized the instruction finishes decoding the instruction and passes the operand data from the register file and control signals to the IE block of the extension. In general, the instruction decoding may be implemented in logic but for a more flexible design a content addressable memory, or CAM, is preferred. The IE block of the extension may span the IE and MA block of the TISA, allowing it an additional clock cycle of execution time to complete the designated operation. When operations require additional clock cycles an IE of the extension sends

a signal to the hazard detection unit of the TISA to stall the processor until the extension has completed operations. The IE of the extension completes execution of its implemented instruction and passes the results to the WB block of the TISA. Other issues exist in arbitrating between the TISA and the extensions, as well as controlling this more complex data path. These issues will be identified and addressed in Section 4.

3.4 THE eMIPS WORKSTATION

Fig. 9 illustrates the two ways in which a workstation based on the eMIPS microprocessor differs from a regular workstation. In the first place, at power up time the TISA is loaded in the FPGA if it is implemented as the default configuration. If it is implemented in fixed logic this step is not necessary. If the step is necessary, a secure component verifies the validity and integrity of the configuration file and loads it on the FPGA [9]. In the second place, when a user starts a program that uses an extension the Operating System, or OS, asks the TISA to verify and load the extension configuration file and enables it for that particular process. Other processes can later share the extension. For those extensions that are in fact peripherals the corresponding software entity is a device driver. A multi-core extension is loaded directly by the OS, automatically as appropriate.

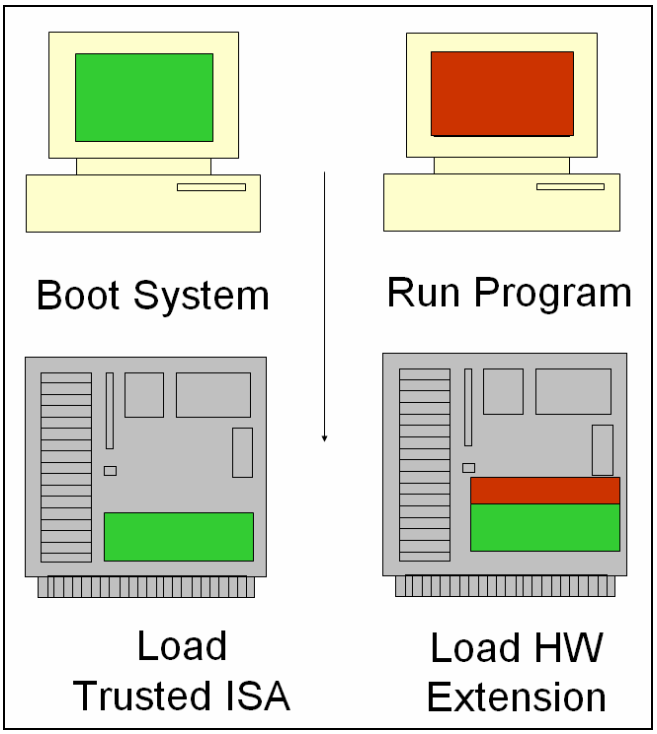

Fig. 9. The eMIPS Workstation, Concept

When the execution comes to a basic block accelerated by an extension, the extension will execute its optimized extension instruction and skip the block that the instruction replaced. If for any reason the extension is not available the extension instruction is ignored and the software executes normally. To accomplish this evolution of the eMIPS microprocessor within a system some software support is required. Some of this software support was described in Section 2.4, namely the software profiling tools used to identify the basic blocks to be implemented as extensions. Additional changes to the OS software are required to control the loading of extensions and to activate them when they are available.

The OS is notified of the extension requirements at application loading time, either explicitly by invoking some Application Program Interface (API), or implicitly by information contained in the executable file image. The OS keeps track of the extension information on a per-process and per-processor basis, as part of its protected state. Fig. 10 depicts the additional state that is required for a dual-multiprocessor using the extensible processor of Fig. 8. Notice that it is necessary to keep track of which extension is loaded in which slot of each available processor, as is depicted in Fig. 10. This knowledge is necessary to load extensions from different application programs at the same time.

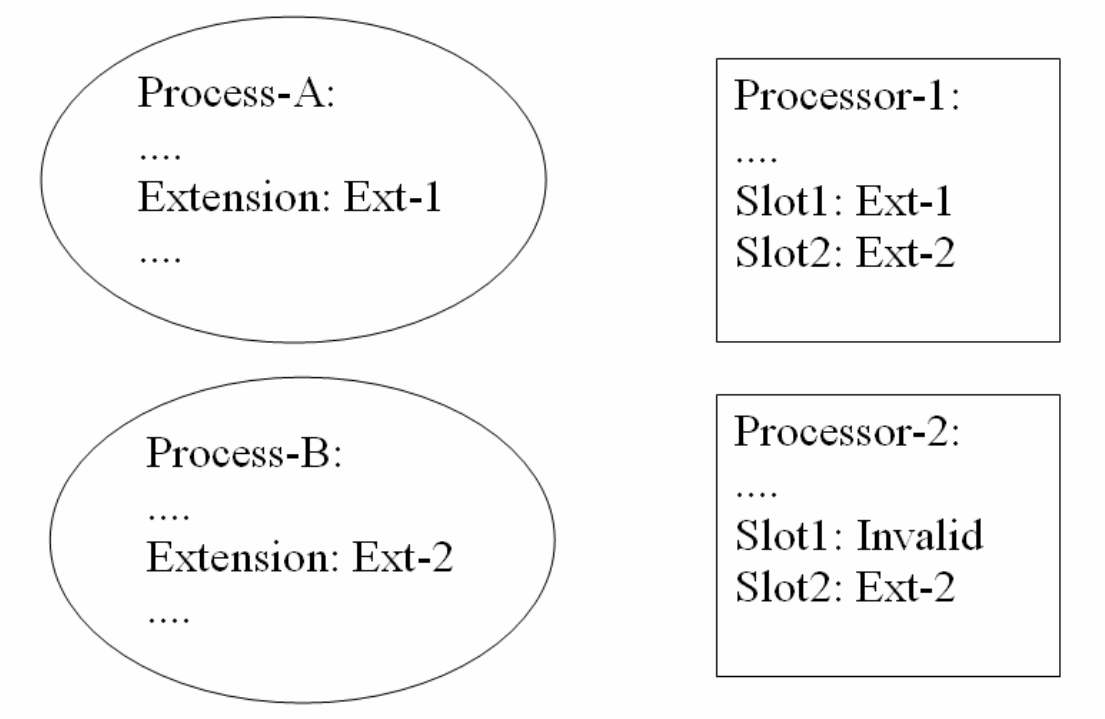

Fig. 10. Required Additions to the OS-Managed Protected State, Processes and Processors

The OS loader is the module responsible for providing the extension data to the OS, at application launch time. The loader can be used to help sharing extensions among applications. Every extension instruction is defined by the binary patching tool using op codes that all start at some value and progress in the same manner. This leads with certainty to collisions between op codes among independently developed extensions. What the loader can do instead is to modify the op codes in a newly loaded image such that they minimize the conflict with other previously loaded images. Fig. 10 shows the effects of this optimization. Because Process-A and Process-B use different op codes they could both be scheduled on Processor-1 without incurring any management trap, and Process-B can be scheduled on both processors. If we try to schedule Process-1 on Processor-2 we will instead incur a trap.

An extensible CPU has only a limited number of slots for currently-loaded extensions. The actual number could be as small as depicted in Fig. 8 or much larger, but it will always be finite. The OS is the arbiter that manages this limited resource on behalf of the application program(s), sharing it in an efficient manner. This management problem is similar to the problem of handling a FPU and the state left in it by multiple application programs. Well known algorithms can be applied here, with one crucial difference. In the FPU case execution cannot proceed unless the coprocessor is made available because there is state left in it and only the FPU is capable of executing the floating-point instructions. In the case of extensions, we are subject to neither of these

constraints. In the first place, the application state is held in the general purpose registers and not in a special unit, unless the extension provides extra register state, which is a special case similar to the floating-point case. Notice that the extension configuration file is not changeable and does not need to be preserved across context switches. Secondly, the code of the original basic block is still available; therefore the OS has the option of skipping the extension instructions and simply falling-through to the original code. This is the reason why we require the extensible microprocessor to leave it to software to decide whether to trap or not on an extension instruction.

Having the option to continue execution “without the coprocessor” opens the door to new and more elaborate software management algorithms, some ideas are as follows. The operating system could exclusively assign the resource to the application that is observed to make the most use of it or is selected by a human user; and/or disable all extensions on interrupts, assuming that interrupt service routines will not make use of them, or to guarantee predictable response times; and/or load as many extensions as there are available slots and fall-back to the non optimized basic blocks otherwise; and/or use a least-recently-used algorithm to handle what is effectively a cache of extension data.

Executable images that use extensions can potentially pose a security threat to the guest OS. A certification authority, such as the OS vendor must sign such images to attest to their safety. If the extension uses the technique described in Fig. 6 there is no security threat because the semantic of the extension instruction is the same as the block

of instructions it replaces. Nonetheless, certification is still required to prove that the extension configuration file does indeed match the intended semantics.

4 IMPLEMENTATION

One way to realize the eMIPS processor is to start from an existing data path implementation and to modify it as indicated in Section 3.3. Xilinx and other FPGA manufacturers provide examples of so called “soft-core” microprocessors, which are also easily retarget able to different devices. Unfortunately, implementing the interconnections between the individual pipeline stages of the eMIPS microprocessor data path and the extensions requires access to the inputs and outputs of each pipeline stage. Due to the proprietary nature of soft-core microprocessors used in FPGA system designs this is not readily available. For this reason, it becomes necessary to implement a full MIPS data path from scratch to provide these needed connections for the TISA.

The pipeline has five stages like the classic CPU and processor models previously discussed in Section 3.1. The pipeline stages include Instruction Fetch (IF), Instruction Decode (ID), Instruction Execution (IE), Memory Access (MA) and Writeback (WB). The descriptions of these pipeline stages have been outlined in Section 2.2.

To realize a complete microprocessor capable of executing test applications the system requires some peripherals integrated on chip in addition to the data path. The minimum set of required peripherals includes the Universal Serial Synchronous Asynchronous Receiver Transmitter, or USART, the interrupt controller, timers, the Static Random Access Memory (SRAM) interface, the flash interface, the parallel

input/output interface, or PIO, and the block Random Access Memory (RAM) interface. The data path accesses these peripherals through the memory interface. The microprocessor's peripherals map to the memory locations in accordance to a memory map stored in the block RAM. The block RAM is a pre-initialized memory element internal to the FPGA that may be used to store data or implement other functions. In this case, the block RAM is interfaced to the microprocessor as a small internal memory that stores the memory map and the boot loader for the system.

4.1 CHALLENGES

The implementation of the eMIPS microprocessor system prototype posed some nontrivial challenges that required consideration as work progressed. The solutions presented may be later improved upon in future works in this area.

4.1.1 SCALING

Many area and complexity issues rise out of the need to scale up some of the components of the data path for the extensions to work efficiently. Since some extensions require more than the standard two operand registers allowed in the standard RISC architecture the register file must be scaled up to allow for additional read ports so that the extension may gather all its operand data in parallel and prevent the delay resulting from multiple register accesses. A similar problem exists for the write ports. Most extensions will modify more than one register in WB so additional write ports are

needed. In the prototype the register file had been scaled by four (eight read ports and four write ports) and had grown in size by a factor of six. This disproportionate increase in size versus port numbers is a direct consequence of the modular architecture of the FPGA.

The FPGA slices can be easily combined and cascaded to build larger components, but there is an overhead generated by the interconnected blocks. Consider building five to one multiplexers using only four to one multiplexers as the building blocks. It is necessary to route two inputs into one multiplexer and ground the others. The output of the multiplexer is input to a second multiplexer, along with the remaining inputs. In this way it takes two four to one multiplexers to realize a five to one multiplexer, three for an eight to one multiplexer and four for a twelve to one multiplexer. Additional logic is needed to control the switching of these multiplexers. It is plain to see how this poor scaling results in such growth. In addition to the register file read and write ports, the hazard detection and data-forwarding units must also be scaled to meet the increased data throughput and they create similar scaling issues.

This scaling issue must be managed properly, it is a constraint imposed by the FPGA technology and cannot be avoided. The parallelism of operand fetch and writeback can be scaled up or down as area allows in each implementation. As the parallelism changes, the overhead for gathering data and writeback varies inversely. Therefore, the eMIPS scaling is a two parameters optimization problem with respect to area and execution overhead. The original register file attempted to remove all execution overhead by providing sufficient ports for the potential extensions identified. However,

such a module required two thirds of the FPGA alone. Such an increase in area requirements for this module makes this approach impractical.

Instead, we accepted a small overhead by adding two read ports and one write ports dedicated to the extensions in addition to those utilized by the rest of pipeline. The overhead involved results from taking multiple clock cycles to gather more data than the number of ports will allow in parallel and then writing back data with the same limitation. In this way, we experience a maximum overhead of four clock cycles over the larger register file while less significantly increasing the area requirements. This acknowledgement of allowing for multiple register transactions also leaves us with a more scalable architecture that will be able to handle any size of operands and return data.

4.1.2 AREA CHALLENGES

The limited physical resources available on the Virtex IV LX25 FPGA impose considerable constraints on the design and implementation of the eMIPS microprocessor. In the first place, the FPGA contains only 10,752 slices for realizing the logic of the design. [59][58] As the microprocessor grows in complexity additional logic is required to realize it. The minimum components to realize the eMIPS microprocessor include the baseline data path (including five pipeline stages, registers, exception handler and pipeline registers), memory interface (including memory mapped peripherals) and room for the Extensions. To realize a functional device, peripherals such as the SRAM memory interface, USART, Timers, and interrupt controller are required. An interface to

the System ACE chipset for reconfiguration is also required. These components in some cases are fairly complex requiring a large number of slices. The reconfigurable feature of the design further constrains use of the physical resources area-wise by requiring the use of bus macros, or pre-routed macros. Bus macros maintain the connections between fixed and reconfigurable logic by forcing all signals that cross the boundary to route a certain way in every configuration. These bus macros are fixed and cannot be optimized away, and their placement is important. For this reason, the bus macros have the potential to create considerable overhead in the design.

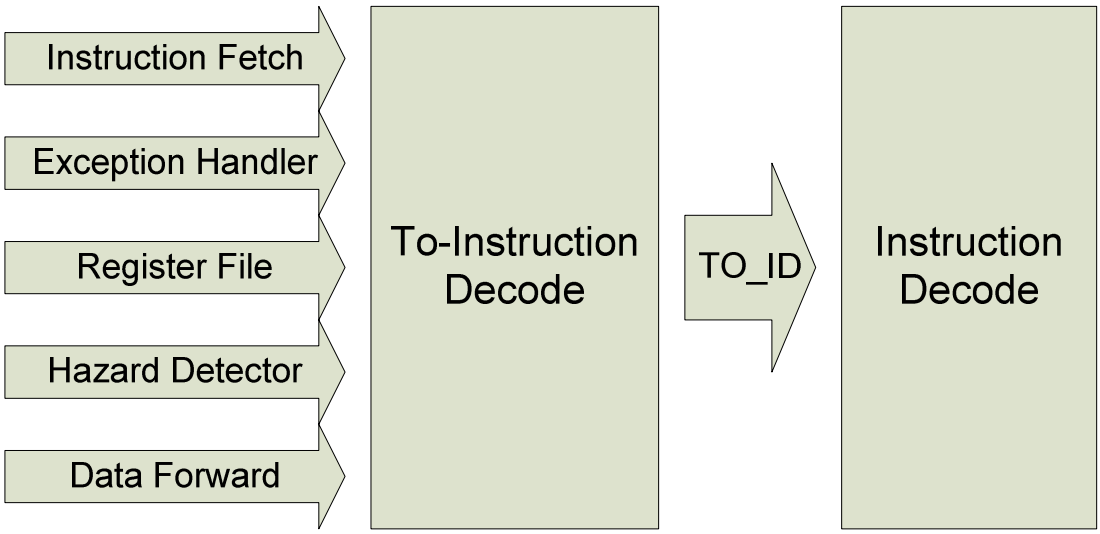

Fig. 11. Input Coupling Bus Module

To minimize the impact of the bus macros on the design and to reduce area overall, we strive in the implementation of each module to optimize each component as it is designed. Sometimes this requires revisiting a component to squeeze out a few more slices. A method, illustrated in Fig. 11, of feed-forward input-coupling/output-decoupling helps reduce the bus macro overhead. Each component has input and outputs that must be routed to multiple other components. In an effort to minimize the number of signals that cross from one component to the next, the inputs of each component are coupled together into a bus-like module that collects all inputs to a component together and then passes them to the component. Inside this bus-like module, signals can be consolidated and optimized to reduce the number of signals that must be passed to the component. For instance, if you have two control signals from different sources that through a simple logical combination such as an AND gate, you can take that logic out of the functional unit and place it in the bus module. By moving this simple logic we reduce two signals that must cross the module boundary to one and other reductions are possible. In the case of a component that is reconfigurable this reduces the number of signals that must pass through a bus macro and thus reduces the number of bus macros required.

4.1.3 PIPELINE ISSUES

For the eMIPS it is necessary to address issues that occur in all RISC architectures but that here take on additional requirements. These include pipeline control, exceptions, branches and hazards.[30] The pipeline control is assigned to the

ID of the extension that takes control of the pipeline and these controls propagate through the pipeline by the pipeline registers as in the standard RISC architecture. The eMIPS diverges from this by adding multiplexers ahead of the EX, MA, and WB stages. These are needed for path exit and reentry when an extension is used.

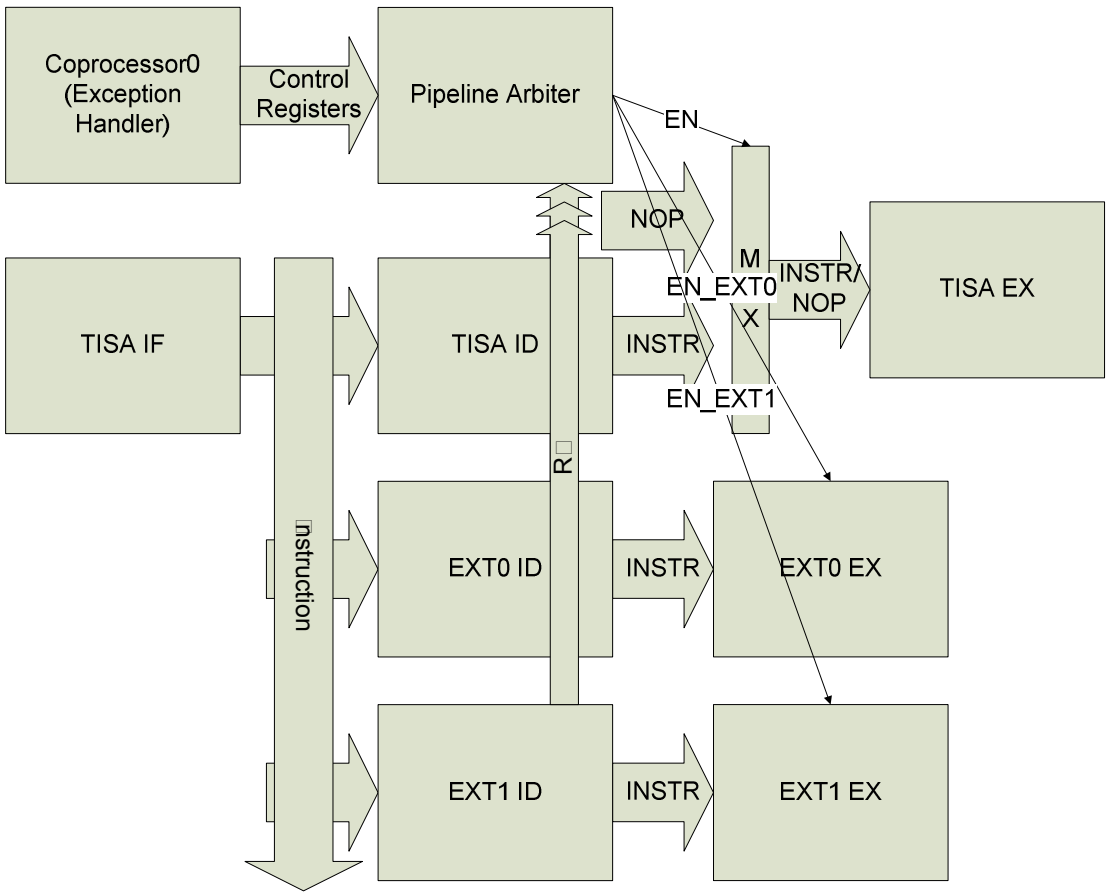

Fig. 12. Pipeline Arbitration Hardware

The eMIPS suffers from the same hazards as the ‘classic CPU’ which are addressed as follows. To minimize the number of instructions in the pipeline that are fetched before a branch is tested, RISC processors perform all branch tests and jumps from the ID phase. This explains the strange behavior of branch instructions forcing the execution of the instruction that immediately follow the branch (delay slot) whether the branch is taken or not. A problem arises when data from EX is needed for ID. The data is not guaranteed to be ready in time for it be used in ID so the pipeline must be stalled until the end of EX and forwarded in MA. When the hazard detection unit detects this, it signals a stall to the pipeline. There are two ways to handle data hazards, forward the data and stall. The eMIPS processor utilizes both. If data for a register being written to is in MA or WB, the data is forwarded to ID or EX as appropriate.

The eMIPS pipeline is more complex than the standard RISC pipeline that most readers are familiar with. In addition to the standard issues of the RISC architecture the eMIPS adds the issue of arbitration between the TISA and the extensions. A pipeline arbiter, in Fig. 12, is used as the gateway to the pipeline. The details of this arbiter are given in Section 4.2.4. In general, the pipeline arbiter is configured using the registers of the exception handler. Registers previously unassigned in the MIPS architecture within the exception handler become enable, disable and priority controls for the arbiter. Using the configuration set in the exception handler the pipeline arbiter accepts acknowledgements regarding recognized instructions and grants control of the pipe line accordingly.

In addition to these arbitration issues, the pipeline must include a way for the data control signals traveling down the extension path to return to the baseline path at some point to complete execution of the instruction by either performing a memory operation or writing back to the register file. The extension ID determines where the extension path reenters the TISA data path based on the instruction encoding. To reenter the baseline path at either MA or WB the extension uses a large multiplexer that outputs to the proper pipeline stage where it will reenter the pipeline, as seen in Fig. 13. These are two to one multiplexers, with the extension and the previous pipeline stage as inputs.

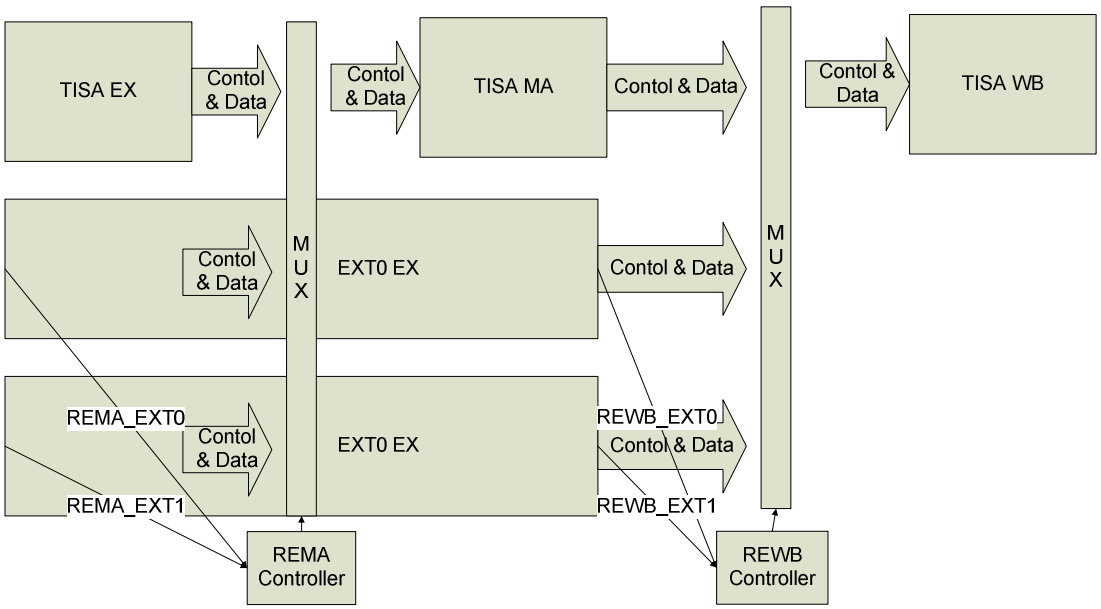

Fig. 13. Data Path Reentry Hardware

In the case of the a simple extension, one that requires less than two clock cycles with no memory access, no change in the pipeline clocking pattern is required. The instruction can execute in step with the pipeline and reenter at the appropriate point. However, in the case of an extension requiring three or more clock cycles to complete the pipeline will have to stall until the execution is complete and the result is written back. Stalling the pipeline is necessary due to potential dependencies. Any performance penalties this creates could be alleviated by further complicating the pipeline controls. The hazard detection could check for dependencies and only stall if one exists. The pipeline could continue in parallel while the extension is executing and then the extension would only have to stall the pipeline long enough to insert the results to be written back to the register file if the a dependency has not already caused the pipeline to stall. This method however, involves considerably more complexity and in this case the commit of instructions would no longer be in order.

4.1.4 EXCEPTION PROCESSING

eMIPS uses an exception handling coprocessor to control the state of the microprocessor and to store information about exceptions for later software processing. Replacing an arbitrary sequence of instructions with a single extension instruction can create exceptions (such as TLB misses) at a number of points during execution of the extension instruction. Consider the case of a load instruction that is the third instruction in the original sequence. Should the effective address of the load fail to translate an exception must be reported, according to the MIPS ISA, for a PC at the fourth address in

the sequence. The address that failed translation must also be made available to software. If any register was modified by the first or second instructions they must be written back to the register file.

In some cases it is possible to implement this type of extensions in a transactional style. All resources and address translations are gathered before the instruction starts and any failure is reported at the starting address. No write-backs are needed and this simplifies exception reporting. Once the instruction starts it is guaranteed to complete and to reach the Writeback stages without incidents.

In more complicated cases this scheme is not feasible, for instance if an effective address is the result of a preceding load instruction in the same sequence. In these cases the extension can maintain a virtual PC register that follows the progress of execution, mimicking the progress of the PC in the original sequence. The instruction will proceed to the Writeback regardless, using the partial results. When an exception is reported execution will restart from within the original basic block. This is the main reason why the preferred means of patching binaries for eMIPS is to insert the extension instruction, without damaging the original basic block.

4.2 eMIPS SYSTEM COMPONENTS

This section provides detailed descriptions of the key components that make up the eMIPS microprocessor architecture. These include the eMIPS pipeline itself, the extensions, pipeline arbiter and memory map.

4.2.1 PIPELINE DATA PATH

The outputs of the IF block routes to the ID stage of the TISA, the ID stages of each extension slot and to a CAM. The input to the CAM is the op code field of the instruction; the output is a set of enable lines, one per extension slot. Each ID attempts to decode the instruction in parallel using combinatorial logic. The output of the IDs and of the CAM route to an arbitration module that determines which ID has recognized the instruction and who should control execution from that point on. The CAM output is used to arbitrate conflicts between the IDs. Additional logic from the system coprocessor can disable individual IDs.

The general purpose register file connects to the IDs through the arbitration module to allow the winning ID to gather operand data for executing the instruction. The number of read ports on the general purpose register file increases to cope with cases where the extension instruction requires more than two operands and reduce execution overhead. The top two read port outputs route to the TISA IE block and each port routes to each of the extension blocks. The remaining pipeline stages in the TISA remain as they would be for the classic CPU, except the connections at the EX-MA and MA-WB pipeline intersections are multiplexed with connections to the extensions. This allows execution in the extensions to re-enter the normal pipeline at any point. The data forward and hazard modules of the pipeline are scaled to incorporate connections to the extension slots. The write ports of the general purpose register file more than the standard one. This supports extensions that produce more than one result.

The extensions have ports that interface to the IF, MA and WB stages of the TISA pipeline. These interfaces conform to a standard bus macro that must be applied to all extensions in order for them to be applicable to the microprocessor's hardware. These interfaces include control signals from the arbitration unit derived from the outputs of the ID blocks and the status enable/disable bits of the extension control registers in the system coprocessor. The extensions use the interfaces to the other pipeline stages to pass data to those stages in order to reenter the TISA pipeline and continue normal execution. Extensions requiring more than two clock cycles to complete may stall the pipeline through the hazard detection unit. The status registers in the system coprocessor controls the clocks to the extensions to reduce power consumption from a disabled extension.

4.2.2 BOOTLOADER

Like other systems the eMIPS prototype requires some initialization or setup before it can begin to execute application software. This is performed by a simple program that we have called the bootloader. The bootloader initializes the peripherals that are available and determines where to jump to begin operation. In our prototype system, after initializing the peripherals the bootloader checks the PIO for the status of a button. If the button is asserted, the bootloader attempts to download the application from the serial line and to write it to the SRAM. After download completes the bootloader jumps to the base address of the application it received to begin execution. If the button is not asserted, the bootloader jumps to the flash and begins executing there.

The bootloader must run each time the system powers up after the default configuration is loaded and each time the system is reset. The bootloader is stored in side a block RAM inside the FPGA. A block RAM is memory module that is part of the FPGA configurable fabric can be used to implement small fast internal memories or memory related components such as FIFO and CAM. The block RAM is integrated in to memory controller at a known fixed location that corresponds to the reset vector of the eMIPS microprocessor. This ensures that the system is initialized each time it powers up or is reset.

4.2.3 EXTENSIONS

The eMIPS extensions provide flexibility in microprocessor design of such a fine granularity that has not been previously presented. By integrating hardware modules directly into the pipeline stages the overhead for communication between specialized hardware and the data path is minimized and in some cases eliminated. This makes the performance gains made through the hardware optimization much more valuable. Also, since the extensions are reconfigurable, modules may be loaded and unloaded on demand when necessary to allow the capabilities of the system to evolve as needs change. The extensions address the area waste that originates from requiring microprocessors to include hardware features in designs just in case they are needed. The eMIPS microprocessor can omit them and add them later if-and-when they are indeed needed, thereby reducing size and power.

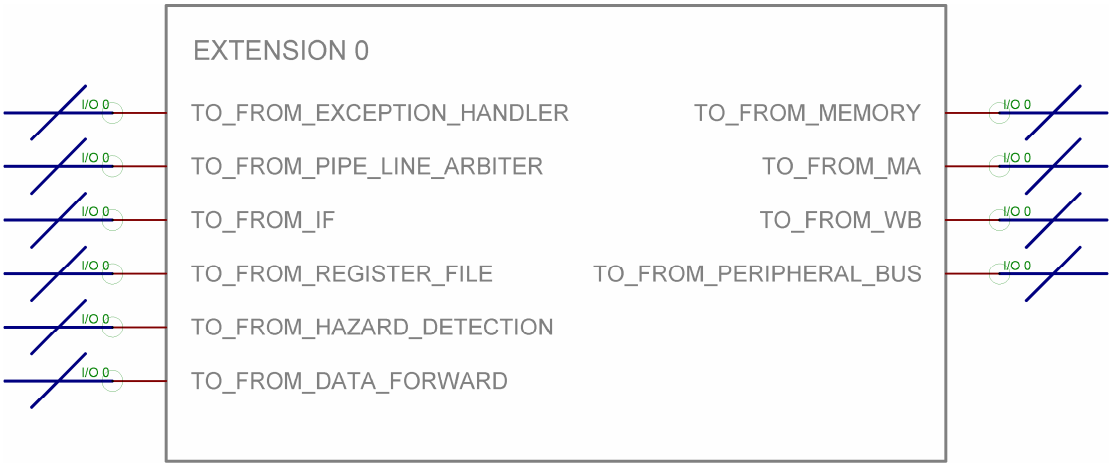

Fig. 14. eMIPS Extension Interfaces

The extensions have ports that interface to the IF, MA and WB stages of the TISA pipeline. These interfaces conform to a standard bus macro that must be applied to all extensions in order for them to be applicable to the microprocessor's hardware. The interface described herein is implementation specific. Other interface may perform better under different circumstances. The following interfaces depicted in Fig. 14 were considered:

- To/From IF – This interface is how the extension receives the instruction read from memory that must be decoded. The extension also takes the program counter, or PC, from this interface for calculating the next instruction to be fetched in the event the extension wins arbitration and executes the instruction.

The extension passes this new PC to the IF to fetch that instruction if it wins arbitration.

- To/From Pipeline Arbiter – The interface allows the extension to request control of the pipeline when it thinks that the instruction fetched belongs to it. After arbitration is complete the pipeline arbiter uses this interface to grant or deny the extensions request for execution.
- To/From Register File – The pipeline arbiter controls access on this interface by only allowing access to the register file to the extension that won the arbitration so it can gather operand data and writeback results.
- To/From Exception Handler – The extension requires this interface for reporting exceptions so that they can be handled by software. This interface requires a port for reporting the virtual PC of the instructions being replaced in the case of instruction compacting extensions. This way the exception may be handled as it would be in the original block and reenter within that block without re-executing the entire extension.
- To/From Data Forward and Hazard Detection – These are two different modules designed to handle the dependencies between consecutive sequential instructions that generate hazards in a pipeline environment. The interfaces include ports to report the operand and destination registers being used by the extensions and ports for retrieved data forwarded from later pipeline stages that have been modified but not yet written back to registers. This interface also includes signals

to the hazard detection unit to assert a stall to the pipeline if the extension requires additional time to complete its task.

- To/From Memory Controller – This is an additional read/write port to memory similar to that used in the MA stage of the pipeline. Like in the original memory ports, access to memory would be managed by a Memory Management Unit, or MMU, to prevent illegal memory accesses. This interface can be thought of as optional if the implementation does not wish to support multiple memory fetches in extensions. Single memory fetches may be implemented by reentering the pipeline at MA by passing the address through the multiplexer with the appropriate control signals.
- To MA – This interface allows the extension to reenter the TISA execution data path after it has completed its task. This interface resembles the outputs of the EX pipeline stage in the TISA, with the addition of control signals switching the multiplexer and allowing for the extension passing its signals to the MA pipeline stage. From here the extension may assert the signals to perform a memory operation or simply pass along the data to WB for writeback to the register file.
- To WB – This interface allows the extension to reenter the TISA execution data path after it has completed its task. This interface resembles the outputs of the MA pipeline stage in the TISA, with the addition of control signals to switch the multiplexer to allow the extension to pass its signals to the WB pipeline stage. From here the extension may assert signals to writeback values to the register file.

- To/From Memory Bus – This is an additional port to the microprocessor's peripheral memory bus that is used to interface the on-chip peripherals through the memory interface. This is another optional interface. If the eMIPS implementation allows for extensions to be on-chip peripherals, the extension may be connected using memory interface and allotted a partition in the memory map. This way the extension peripheral may be treated just like any other on-chip peripheral.

Other interfaces or variations may be adapted given the needs of the individual implementation. The architecture is modular in that anything may be developed as an extension but it must conform to the interface provided by the extensible microprocessor design that is meant to use it. For instance, this means that the datasheet of a hypothetical commercial microprocessor using the extensible microprocessor architecture should include the descriptions of its interfaces and their conventions. This information is required for the hardware designers who will derive the hardware modules. In addition to information about the interfaces, information about the how instructions decode and pipeline arbitration is important for instruction decoding in the extension. With the information from the data sheet, the task of creating a new extension can be broken down into the following steps:

1. Implement the logic to execute the semantic of the desired function or basic block in the most optimal way possible.
2. Implement the instruction decode per the data sheet.

3. Implement top level extension model with interfaces that conform to the conventions expressed in the data sheet.
4. Integrate the logic and instruction decode inside the top level extension model with the interfaces to the pipeline.
5. Open the desired development environment with the model of the targeted extensible microprocessor and associate the extension with the desired extension slot. When using Planahead from Xilinx on a partially reconfigurable implementation, open a project with the base extensible microprocessor assembled. The extension being implemented will have to be named to the same name as the extension slot for which it is being targeted in the top level module of the microprocessor so it can be associated with that slot.
6. Execute the design flow for the targeted architecture. In the case of Planahead we execute the partial reconfiguration design flow for Xilinx FPGA using the dialog windows provided.
7. Generate configuration files for the targeted extensible microprocessor. For the Xilinx FPGA we used in this implementation, after Planahead finishes the partial reconfiguration design flow, users have the option to generate base configuration files (*.bit) for the base configuration including the new extension or a partial configuration file that will configure only the region partitioned for the extension and leave the remainder of the FPGA as it is. The partial configuration file is the one that would be included with binary image of an accelerated program to be loaded to the targeted microprocessor's architecture.

8. If the microprocessor system includes the System ACE configuration solution to allow for the configurability of the FPGA, the configuration file (*.bit) will need to be converted to a System ACE configuration file (*.ace).

The extensions are meant to provide additional hardware support as needed to accelerate the performance of software applications, or to provide new features or on-chip peripherals. Regardless of the intended purpose, all extension must include certain functionalities. All extensions must be able to recognize their own instructions and request control from the pipeline arbiter. The extension must be able to calculate the PC of the next instruction to be fetched if the instruction being decoded is for this extension and it wins arbitration. This calculation of the new PC will take into account the size of the basic block being replaced by the extension so that those instructions may be skipped and the microprocessor resume execution of non-extension instructions after the basic block. The extensions must also provide all the controls to allow the extensions data and controls to reenter the TISA pipeline data path to finish execution.

Other functionality described herein is recommended to deal with known issues that the extensible architecture can create. These features are not just possible ways of addressing the problems; it is the position of this thesis that they are the best methods known to date. We present the previous work in the area of reconfigurable processors and explain where we depart from them in Section 8.

4.2.4 PIPELINE ARBITER

The pipeline arbiter acts as the gateway to the controls of the remainder of the eMIPS pipeline. The functionality of the system coprocessor increases to let software configure the pipeline arbiter and manage the microprocessor's extension slots. Additional register numbers, not previously allocated, in the system coprocessor or CP0 are defined to control the state of the extensions. Bits in these registers may enable or disable a given extension slot, and define the behavior when an extension instruction is recognized by an extension that is currently disabled. Two alternatives are to treat the instruction as a NOP or to generate a RI exception. Some registers are defined to control access to the op code CAM, in ways similar to the MMU interface. Yet other registers are used to set the priority of the extension slots.

When an instruction is decoded the arbiter decides whether the TISA data path or the extensions will take control. The arbiter receives acknowledgements from the instruction decoders of the TISA data path and the extensions whether or not an instruction has been recognized. Using the op code CAM, different op codes are assigned to either the TISA or different extensions. An instruction is considered recognized if the extension acknowledges the instruction and the op code of the instruction is associated with that extension in the op code CAM. If only one instruction decoder recognizes the instruction the path that is associated with that decoder will normally take control, but there are some special cases that must be addressed.

If the instruction is only recognized by a disabled extension or if the op code CAM entry does not match, the arbiter must not allow that extension to take control and

prevent the TISA data path from throwing a RI exception. When this occurs the microprocessor must interpret the instruction as a NOP. In this case the arbiter uses control registers in the exception handler and the output of the op code CAM to verify which extensions are enabled and which are not for a given op code. It is also necessary to prevent generation of a false RI exception by TISA data path, and the arbiter is therefore the one responsible for throwing the exception when none of the extensions recognizes the instruction.

A similar solution is implemented in the case of multiple instruction decoders recognizing the same instruction. In this case the extension control registers assign a priority to each extension and to the TISA data path. By default, a daisy chain priority is assigned starting from the TISA data path. When conflicts do occur, the extension or the TISA data path with the highest priority wins control of the pipeline. In this way, if the extension has higher priority than the TISA, the extension may mask an instruction in the TISA. For instance, consider the case of multiplication. If an application requires a lot of multiplications to the point that developers want a more optimized multiplier than what is available in the TISA, an extension could be developed that includes a faster multiplier that perhaps uses a larger area. Similarly, this may also be accomplished by altering the entry in the op code CAM to only associate the op code with a particular extension. However, this solution would result in a stall if the extension were already in use rather than fall back on the original multiply.

Finally, if none of the instruction decoders recognize the instruction, the pipeline arbiter should throw a RI exception, except when the application running on the

microprocessor uses an extension and for whatever reason that extension is not available in the microprocessor. There are two potential solutions for this in hardware and software. In software, the exception handling routine could check a table of extension op codes and if the op code matches one that is in use, ignore the exception. Otherwise, handle it normally. In hardware a similar look up table could be implemented and checked by the pipeline arbiter before throwing the exception.

4.2.5 MEMORY MAP AND PERIPHERALS

Like most modern microprocessors, the eMIPS prototype includes a few hardware devices with the microprocessor on the same chip. These devices range from specialized hardware modules to communications interfaces and memory. Peripherals that might be useful in an eMIPS microprocessor include but are not limited to USART, SRAM controller, FLASH controller, timers, interrupt controller and others. The microprocessor interfaces to these devices through memory operations such as load and store. The devices are allocated to ranges of memory addresses within the microprocessors memory space using a memory controller or combinatorial logic. The description of these ranges and the function of each of the addresses are called the microprocessor system's memory map. In most microprocessors, this memory map is largely static with limited configurability and only for external devices.

32'hfffdffff 32'hfffd0000	SRAM
32'hfffcffff 32'hfffc0000	DDRAM
32'hfffbffff 32'hfffb0000	FLASH
32'hfffaffff 32'hfffa0000	INTERRUPT CONTROLLER
32'hfff9ffff 32'hfff90000	USART
32'hfff8ffff 32'hfff80000	TIMER
32'hfff7ffff 32'hfff70000	WATCHDOG
32'hfff6ffff 32'hfff60000	GPIO
32'hfff5ffff 32'hfff50000	IDE
32'hfff4ffff 32'hfff40000	LCD
32'hfff3ffff 32'hfff30000	PS2
32'hfff2ffff 32'hfff20000	VGA
32'hfff1ffff 32'hfff10000	ETHERNET
32'hfff0ffff 32'hfff00000	POWER MANAGEMENT
32'hffefffff 32'hffef0000	AUDIO CODEC
32'hffeeffff 32'hffee0000	BLOCK RAM (BOOTLOADER)

Fig. 15. Example Memory Map for an eMIPS Microprocessor System

The strength of the extensible microprocessor architecture is its flexibility and ability to evolve. This feature continues in the design and implementation of the eMIPS memory map. The eMIPS memory map is partitioned into 64K ranges of addresses, as

depicted in Fig. 15, that may be used for interfacing devices. Since the eMIPS microprocessor is an extensible microprocessor, the number, type, and location of peripherals may not be known or could change. For this reason, the bootloader which is run at start up plays an important role in determining what peripherals are available and where they are.

Each peripheral designed for integration into the eMIPS memory map includes a read-only tag register to identify the purpose and functionality of the peripheral. The tag register is sixteen bits and is broken into two eight bit fields. One field is the peripheral identification and the other is the instance identification. The peripheral identification denotes the type of peripheral the device is and its function. Each type of peripheral is assigned a unique peripheral identification to allow software to determine what the device is and how to use it. With eight bits it is possible to uniquely identify 255 types of devices. The tag values used to identify peripherals in eMIPS are given in Fig. 16. The instance identification allows systems to interface multiple instances of the same device type. Each instance must have unique instance identification. In this way each device interfaced to the eMIPS memory map has a unique identifier and its purpose can be determined.

The eMIPS microprocessor is a 32-bit based processor and performs operations on 32-bit data. For this reason the sixteen bit tag register is coupled with a sixteen bit base register for memory controller peripherals. In these memory controller peripherals the base register gives the upper sixteen bits of the base address of the memory being

interfaced by the controller. The value of this register may be changed by software to configure the location of memory.

Peripheral Type	Tag
Block RAM	0
Pulse Width Modulation	1
SRAM Controller	2
DDRAM Controller	3
FLASH Controller	4
Interrupt Controller	5
USART	6
Timer	7
Watchdog	8
General Purpose IO	9
System ACE	10
LCD	11
PS2	12
VGA	13
Ethernet	14
Audio Codec	15
Power Management	16

Fig. 16. Peripheral Types and Associated Tags

This base-tag register is located at the base address of the memory map partition it is assigned. In this way, the bootloader may determine what peripherals are available and where they are by checking the base address of each 64k partition allocated for peripherals. The bootloader reads these address locations. If a recognizable tag is read from the least significant sixteen bits of the base-tag register, the bootloader can

initialize it and configure the OS accordingly to make this resource available. In most cases, the number of address locations required to control and interface to peripherals are small compared to the 64k address spaces allotted for them. For this reason, the eMIPS memory map allows for peripheral of similar type to be grouped into the same 64k partition. After the base-tag register is read and recognized, the number of addresses used by this peripheral is known because they conform to the interface assigned to this type. The bootloader after reading and configuring the peripheral at the base location, it skips the appropriate number of address locations utilized by this peripheral to check for a second one. If this address location contains a valid tag that matches the type of the previous, this will also be initialized and configure. The bootloader continues to skip the address spaces and checking tags until it finds an invalid tag or the end of 64k partition.

After the bootloader initializes the static peripherals in this way, a similar method can be used to initialize extension peripherals. Since peripherals may be implemented as extensions, the memory map must be dynamically maintainable. Also, the memory map partition it is assigned to and position in that partition should be configurable and protected after configuration. Since these extensions will not be used to execute instructions, the unused op codes may instead be used for special platform specific functions for this extension to configure its position in the memory map. After the extension peripheral is loaded by the OS, the extension is enabled using the exception handler registers that control the pipeline arbitration. Using these special instructions in privileged mode, the OS configures the peripheral extension to reside in an unoccupied location. When this is done, the extension is disabled in the exception handler and the

base location of the peripheral extension is assigned and checked. If the tag is recognized and matches the type expected, the OS will initialize and configure the extension peripheral like the bootloader initialized and configured the static peripherals. By only enabling the extension for execution long enough for configuration and disabling it after, we protect the configuration of the peripheral extension from tampering. This protection is strengthened if the constraint is made that these configuration instructions or the change of extension state may only be made from privileged mode.

5 TESTING AND VERIFICATION

The testing and verification process of the eMIPS system required extensive interactions between simulations and HDL coding of the microprocessor architecture, using purely software-based functional simulations and software-simulation integrated with FPGA prototypes and their debugging tools. Specifically, Mentor Graphics' ModelSim 6 Xilinx edition [36] was used for HDL functional simulations, Microsoft Giano [37][22] was used for processor and HDL co-simulation and for profiling, Xilinx ISE tools were used for FPGA programming and Xilinx ChipScope Pro for FPGA debugging.[50] With Giano a user can create a complete computer system with typical resources such as processor, memory, busses, and peripherals for register-level functional simulation. Giano interfaces to ModelSim through the Verilog Programming Language Interface, or PLI, [46] so that the behaviors of microprocessors and FPGA chips can be simulated at the same time.

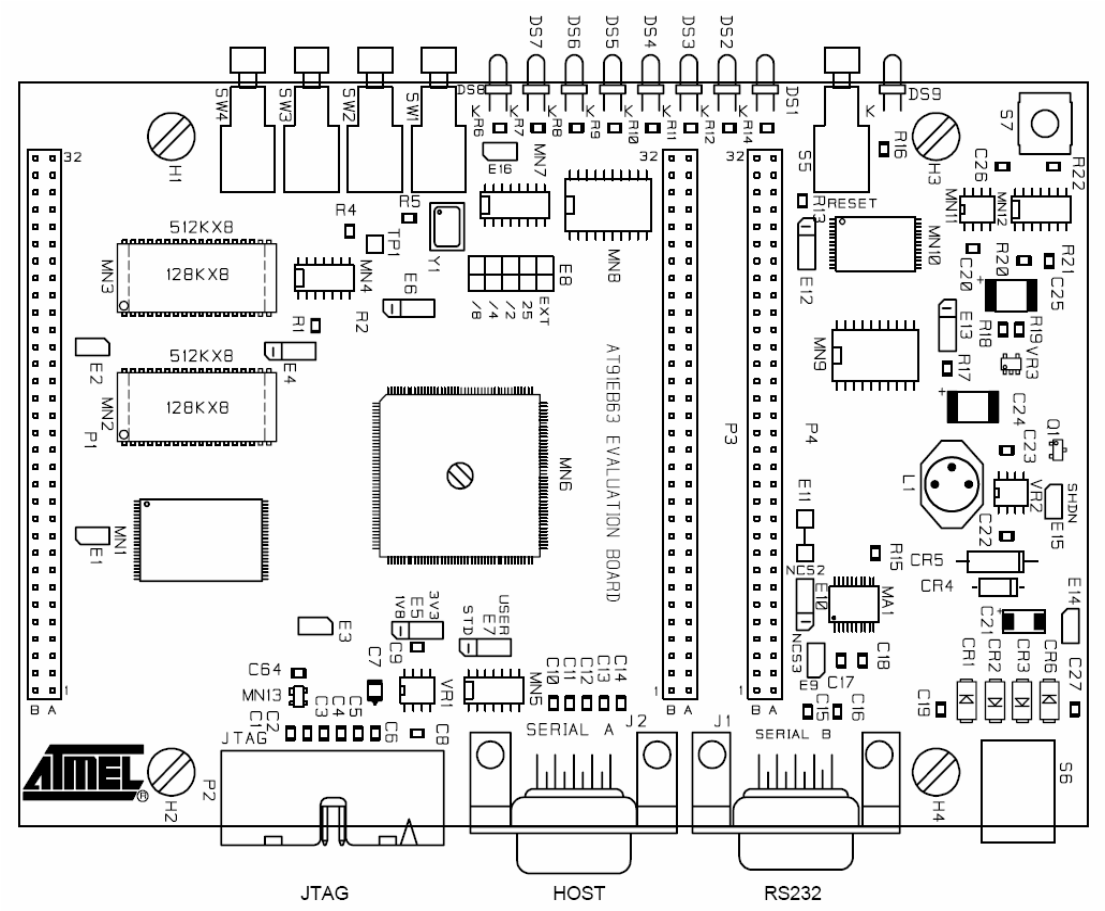

Fig. 17. Atmel EB63 Evaluation Board [6]

Giano simulates a full microprocessor system, initially the Atmel EB63 Evaluation Board depicted in Fig. 17. The simulation of this board provided the initial operating environment for the CPU. Both Microsoft Research and the Microprocessor Design course at Texas A&M University (CPSC 462) have used this platform for research and educational purposes for some time, since before 2004. Software for this board was already publicly available and easy to modify [29] and this well-known

platform therefore is a good baseline for comparisons. Later on in the project the CPU module and the software have been adapted and simulated in a configuration that more closely resembles the target ML401 board from Xilinx.[51] This configuration is shown in Fig. 18.

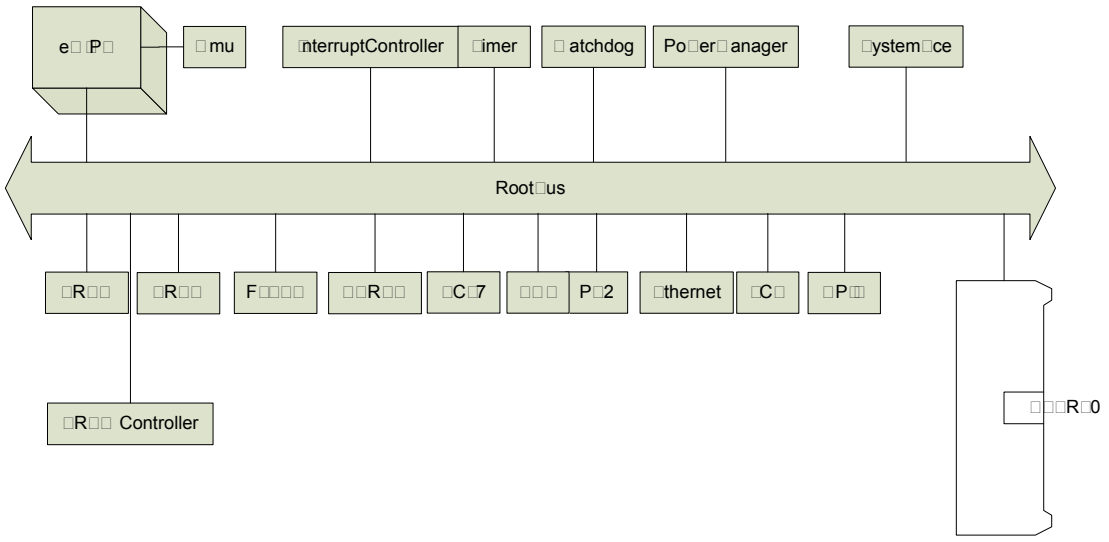

Fig. 18. Peripherals for the ML401 Board

In the test environment depicted in Fig. 19, two instances of the Giano simulation run in parallel. Both instances run a slightly modified model of the EB63 board or other desired platform. One instance (shown at right) runs with a verified functional model of a MIPS data path in place of the ARM core used in the real evaluation board. This instance records its execution history and sends it to the eMIPS ModelSim simulation. It

acts as an ‘Oracle’, providing the correct execution stream of the software application for comparison with the system being tested. The second instance (left) replaces the ARM core with an interface to the ModelSim hardware simulator (center).

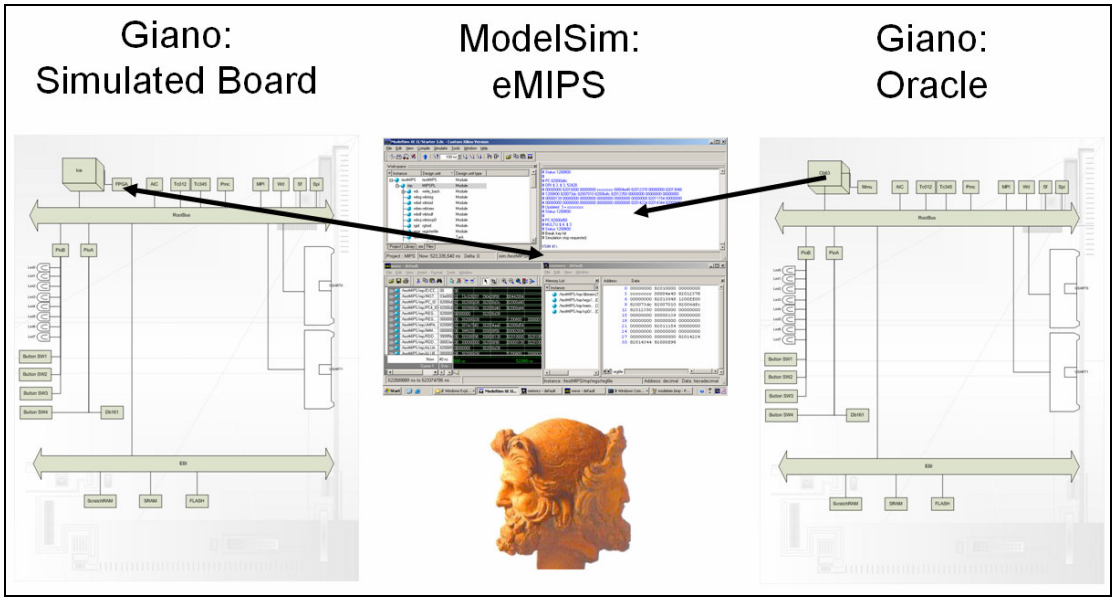

Fig. 19. Testing eMIPS with Giano

Inside of ModelSim, a Verilog implementation of the eMIPS data path being tested is running within a wrapper module providing a simulated clock. The data path simulation interacts with Giano through the Verilog PLI [46] using memory requests. The data path fetches instructions from the memory simulated in Giano and interacts with the peripherals according to the EB63 memory map. These peripherals are

simulated by Giano to better isolate errors in the data path itself. Each time the data path running in ModelSim commits an instruction it checks its internal state against the state reported by the 'Oracle' instance of Giano. This allows the Verilog implementation of the data path to verify if it is behaving correctly. When a discrepancy is found the simulation can stop immediately, and the history trace most likely contains all necessary data to find the cause of the error. In this configuration we can execute arbitrary long sequences of instructions, including the bootstrapping of an entire OS and the loading of an application program.

In addition to the EB63 Oracle environment, another environment was assembled for more exhaustive testing. This environment is called the TestGenerator environment and is shown in Fig. 20. This simulation includes the same ModelSim interface (labeled "FPGA" in the picture) and the simulated MIPS processor ("MipsOracle") used in the previous setup except they are now present in the same Giano instance. They both connect to the same test pattern generator ("TestGenerator") that acts as a memory bus interface to both simulated cores. The test pattern generator feeds the simulated cores with the same sequence of instructions and data, and captures the addresses and values of any data written to memory. As the Verilog implementation running in ModelSim commits instructions it compares its state to that of the 'Oracle' like in the previous example. The TestGenerator scans the entire core instruction set of the microprocessor (the TISA) and for each instruction generates a sufficient number of tests to guarantee coverage both with respect to the instruction encoding and with respect to the functional results given a set of test values in registers.

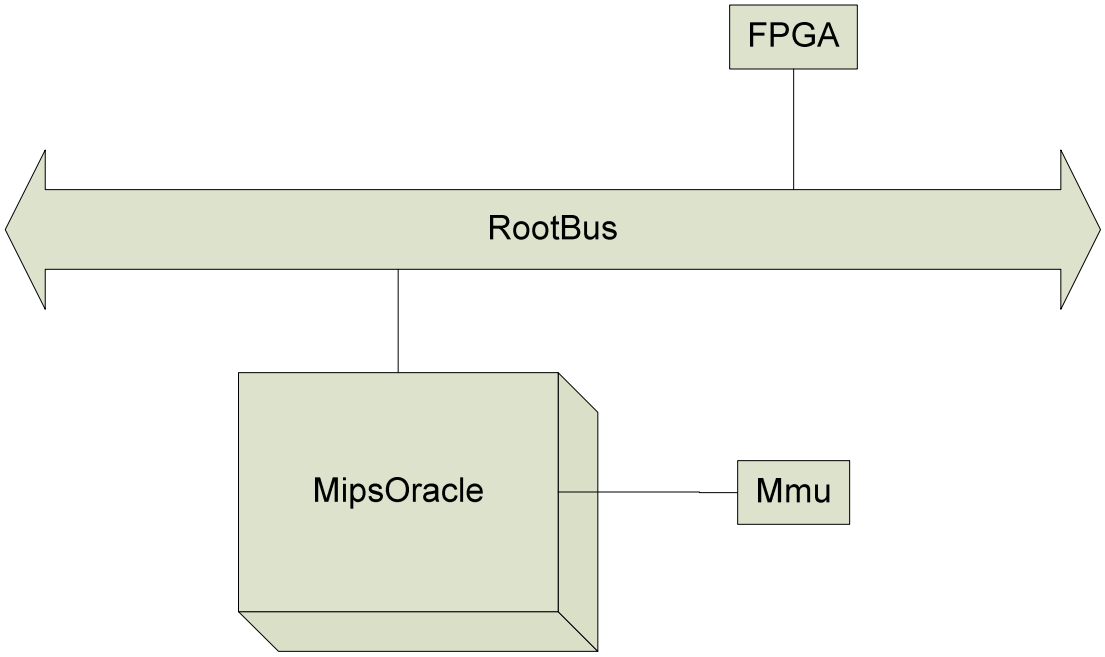

Fig. 20. The TestGenerator Environment

The data path can be verified using these two simulation environments. Application testing performed by the EB63 and ML401 environments provides confidence that the microprocessor executes the application software like the real MIPS microprocessor. The exhaustive functional tests performed by the Test Generator ensure that all corner cases operate within specification.

Setting up this test environment presented some interesting challenges. The eMIPS data path being tested in both these environments were implemented using a five stage pipeline. This architecture requires five clock cycles from IF to WB to complete

execution of an individual instruction assuming no hazards. The MIPS model acting as the oracle however is a simple behavioral simulator that executes an entire instruction each clock cycle like a non-pipelined MIPS microprocessor. As a result, the timing of these two simulations can be out of synchrony and this makes the testing of timing related features such as interrupts difficult to test. As a result some modifications were made to the MIPS simulation and the PLI interface to synchronize the two environments. First, the PLI interface was changed to allow the eMIPS Verilog simulation running in ModelSim to signal to the board simulation in Giano that it had committed an instruction. Then the simulation in Giano of the EB63 was modified to only clock the board when it receives the signal that an instruction had been committed. In this way, the timing of the board relative to the eMIPS data path resembled that of the behavioral MIPS simulator running in parallel. The timing of both simulations was synchronized giving both simulations the same outputs from the timers and other timing related systems. Using this modification, the timer interrupts occur at the same points in execution allowing testing of those interrupts in simulation with no modification of the trusted MIPS model being used for reference.

As the data path is tested and verified for correctness, implementation of the on-chip peripherals begins. These peripherals have the potential of becoming complex systems on their own. Unit testing must be performed on some of these peripherals using ModelSim. After the peripherals have been verified in simulation, debugging continues on the FPGAs using test benches based on the simulation. Xilinx ChipScope Pro [50], an on-chip debugging and verification tool, allows us to monitor the internal signals of

the FPGA for this purpose. Using ChipScope, integration testing follows unit testing as these peripherals become interconnected with the data path through the memory map. Eventually, a full system is assembled on the ML401 board using the same software binaries used in the ML401 simulation with Giano.

The process of taking the Verilog code from a behavioral simulation environment to the FPGA required some tweaking as well. The behavioral simulations in ModelSim and Giano were good for confirming the logic implemented by the code was indeed correct but these simulations do not take into account the transient characteristics of logic implemented on an FPGA. Due to how the synthesis tool implemented the logic in some cases, the path delay has been considerable so logic that worked in a simulation environment that experience little to no delay reveals timing errors and bugs in an FPGA. As these issues were discovered, we observed their characteristics using ChipScope to identify the timing bottlenecks and tweak the logic to correct for them or eliminate them. Then we would take these tweaked versions of the data path back to the simulation environment used to validate the original design to make sure the logic continues to hold with the new timing.

6 RESULTS

This first prototype implementation, while limited in its capabilities, has allowed us to explore the different aspects of this problem in detail. This experience has made it possible for us to propose innovative solutions to the issues and challenges this new microprocessor architecture presents. The following presents the results of our experiments with the new architecture.

Using the TestGenerator environment explained in Section 5, we verified the behavioral correctness of the eMIPS TISA data path across the span of the MIPS R4000 instruction set. On a server utilizing a four core Intel processor, the test spanning the breadth of the instruction set required a little more than a day to complete. The simulation executes over a million instructions in the microprocessor during the course of the verification.

Using the profiling technique described in Section 2.4 we identified potential basic blocks from a range of applications and determined the effects of the optimization on execution timing performance. The results of these software profiles lend credence to the claims of this thesis. Fig. 21 shows the distribution of the top 100 basic blocks in the XQuake video game on the Xbox360 gaming platform. On the Y axis is the dynamic execution count of the individual basic blocks, numerically indexed on the X axis. Other profiling data demonstrate the same two basic traits shown in Fig. 21: the graphs drop exponentially and have a rather long tail. In the XQuake case for instance, there are more

than 12,500 basic blocks that are executed at least once, against a total population of more than 38,500 individual basic blocks.

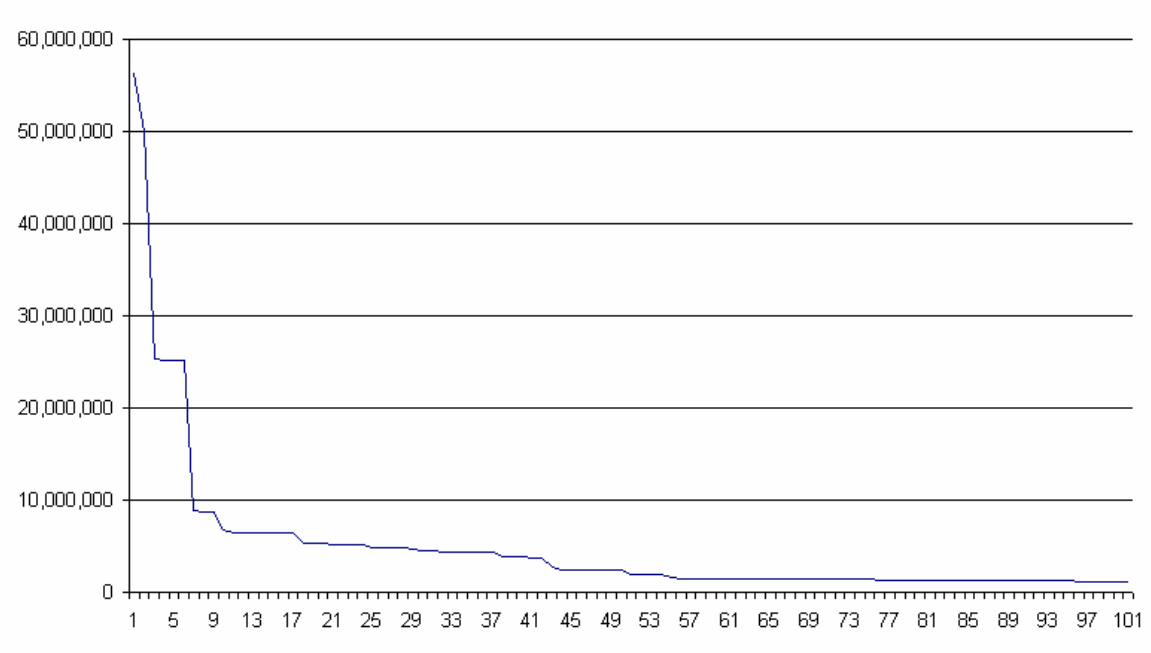

Fig. 21. Execution Counts of Individual Basic Blocks in XQuake, on the Xbox360

To demonstrate the potential speedups that optimizing these basic blocks would deliver extension instructions with the same semantics as the basic blocks are added to the simulation of the CPU. Then the profiled application program is run again, this time with the optimized instructions for the basic blocks inserted into the binary of the software at locations immediately preceding the basic blocks. When the CPU reaches these instructions, it executes the optimized instructions and skips the following block.

The idea is that these optimized instructions in the simulation will then become the basis for hardware extensions of the eMIPS microprocessor. As shown in Fig. 22, in the XQuake case this results in a three-fold improvement in the game's frame rate. Similar results are obtained with the Doom video game, this time using eMIPS and system software from the Microsoft Invisible Computing, or MIC [29].

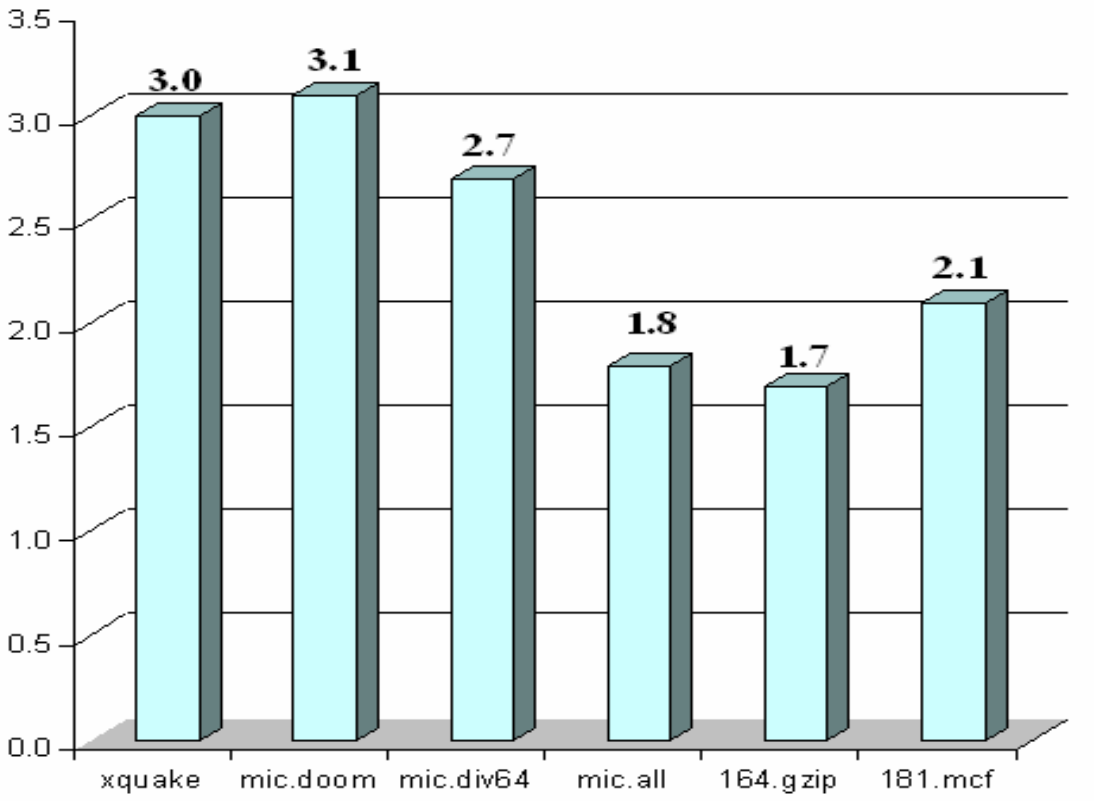

Fig. 22. Speedups from Extension Instructions

Fig. 22 then shows the results from the execution of a series of over 40 tests programs that are part of the MIC system. Only one basic block was optimized in these tests, the same in all cases and shown in Fig. 6. This optimization is also applicable to the Doom case. The third column shows the speedup for the individual test that benefits the most from this optimization. The fourth column shows the cumulative speedup across all tests.

Column five and six in Fig. 22 report the speedups for two benchmarks from the SPEC2000 suite run on eMIPS with MIC. Porting of other benchmarks is in progress. Other experiments reported in the literature [15][16][61][10][62] similarly show that replacing basic block sequences with optimized instructions results in a speedup from factors of two to factors of five and in some cases over a factor of ten.

Module	Slices	%Total	LUTs	%Total	Clocks	%Total
Available on Chip	10752		21504		32	
Data Path	5392	50.15	9614	44.71	2	6.25
Peripherals	2558	23.79	4714	21.92	4	12.50
Base Total	7950	73.94	14328	66.63	6	18.75
Extensions	983	9.14	1608	7.48	2	6.25
Total	8933	83.08	15936	74.11	8	25.00

Fig. 23. eMIPS on the Virtex IV XC4LX25 FPGA Device

It took slightly over 100 modules, and about 10k lines of HDL codes to implement eMIPS. Fig. 23 provides all the area figures for the key components. The

TISA data path requires a little more than a third of the area and on-chip peripherals one quarter of the area. A 64 bit signed/unsigned divisor is implemented as an extension, which takes 64 clock cycles per instruction at the cost about 10% of the area.

7 FUTURE OF eMIPS

The eMIPS extensible microprocessor is still in its infancy compared to other more advanced architectures but show great potential. Throughout this thesis we made references to various potential points of expansion of this microprocessor model. In this section, some of the strongest potential areas of expansion of the architecture are discussed in further detail. These ideas have not been implemented only due to the limited time at our disposal.

7.1 CAM BASED DECODING

A more flexible implementation of the instruction decoding would include a CAM to hold the allowed op codes and control signals stored within. The CAM would be loaded with all the instructions supported by the TISA data path. Support for extension instruction is implemented by loading the extension into the FPGA and then writing a new entry into the CAM. The output of the CAM when presented with an instruction will indicate to the pipeline arbiter if the instruction is recognized and if so, whether the baseline data path or one of the extensions will carry out execution. This alternative instruction decoding and arbitration design is shown in Fig. 24. The same data and control signals are inputted to each of the execution blocks and only the one that is enabled by the pipeline arbiter executes. The baseline data path is given a NOP when it loses the arbitration like before.

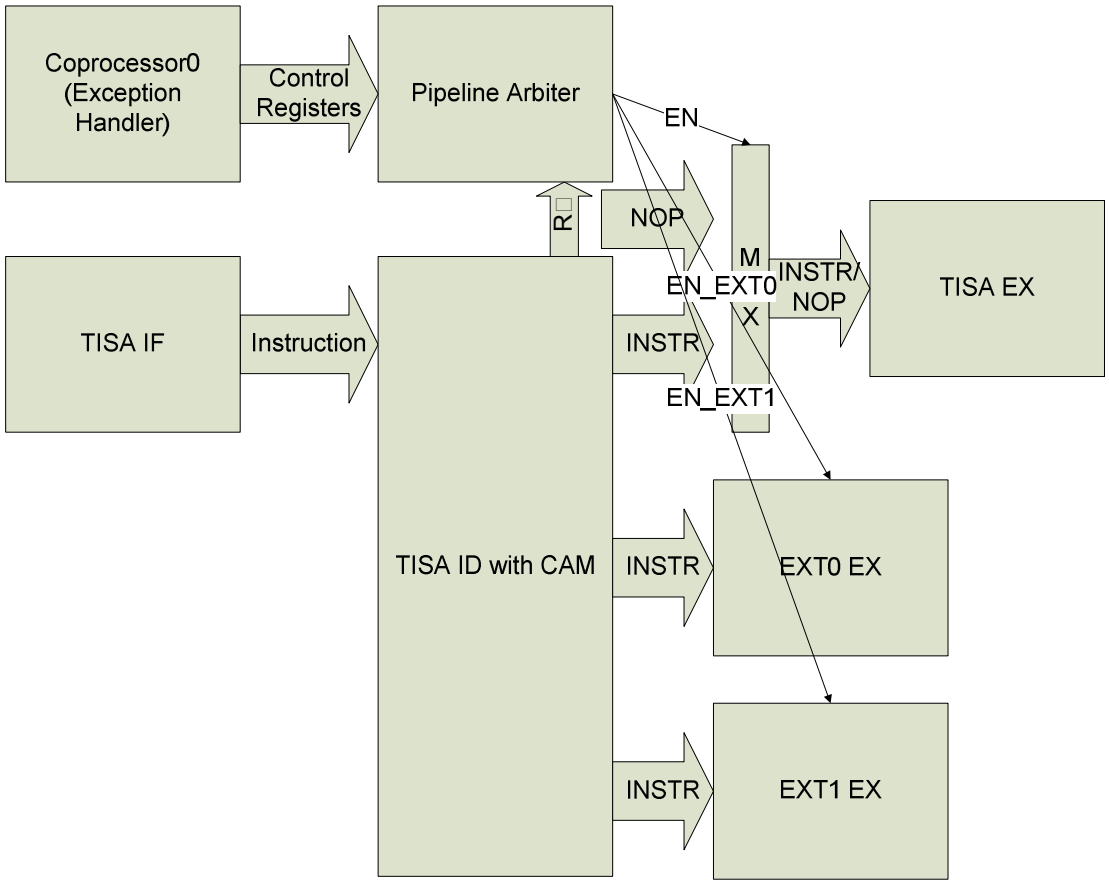

Fig. 24. Instruction Decoding with a CAM

Use of a CAM solves the problem of the limited number of op codes available to the extensions and the resulting conflicts as illustrated in Section 3.4. The pipeline arbitration is simplified by this change because the CAM will output the path for which the instruction is assigned and then the pipeline must only check to see which if any of the eligible extensions are enabled. Then the pipeline arbiter must resolve conflicts if

any exist. This approach also eliminates the need for another lookup table of loaded extension op codes because this CAM can provide that function as well. Also, we eliminate the combinatorial logic required in both the TISA and the extensions required for instruction decoding and reducing area requirements. This approach is slightly more complex given that it combines functions of several components currently implemented. The major draw backs to this approach would be the latency of the CAM and the routing overhead to use the memory modules within the FPGA to implement them. The latency of the CAM can be up to two clock cycles to read from and then process the output. The routing overhead should be small but must be considered due to the floor planning requirements of partial reconfiguration.

7.2 MULTIPLE INSTRUCTION DISPATCH

The eMIPS provides parallel execution paths within the same microprocessor used by difference classes of instructions: the base instruction set executed on the TISA and the extension instructions executed on their respective hardware modules. As an instruction is fetched and decoded, it is steered to the appropriate path. At each stage of the pipeline only a single path in the microprocessor is active as each instruction passed through. If you have an eMIPS implementation with sufficient density to contain a large number of extensions, the amount of idle time for some modules could be considerable.

Consider the situation of a TISA instruction and one or more extension instructions in a sequence. Normally we would allow them to be fetched and propagated sequentially. However, if each of these instructions use a different hardware module, it is

possible that these different instructions could be executed in parallel. A more advanced IF unit could fetch multiple instructions and dispatch as many as it could to parallel paths at the same time. This has potential to increase throughput by maximizing the hardware utilization.

A bottleneck or collision point does exist in this scenario when the parallel Extensions attempt to reenter the TISA pipeline to complete execution. For this, the hazard detection module must be expanded to arbitrate the instructions reentry, commit and completion. The hazard detection module most be able determine the original order of the instruction and work with data forwarding to maintain data dependencies and in-order completion. The hazard detection unit must be able to individually stall and grant reentry to each parallel path individually. If there are no data dependencies and if in order completion is not required, the hazard detection module may allow the parallel paths to commit as they complete allowing faster modules to release and be reused without waiting on the preceding instruction that may be slow or stalled waiting for data.

Architecture of this type, while potentially higher in performance if realized, is more complex that even the original eMIPS. The exception handling in particular would pose a challenge to deal with all the issues involved in detection, reporting, and CPU state.

7.3 MULTI-CORE eMIPS MICROPROCESSORS

In Section 2.1, we presented multi-core microprocessor designs as a competing design paradigm to our extensible microprocessor architecture. Microprocessors with

multiple cores attempt to increase throughput by exploiting parallelization of the software being executed. Unfortunately, this is not the natural way most software is written. Most high level programming languages are designed for sequential execution of the code and results in dependencies that make parallelization difficult. To fully realize the potential performance gains of a multi-core system, software development for such systems must be revisited. The extensible microprocessor architecture of eMIPS attempts to improve performance using optimized hardware after the software is implemented. In this way, no change is required at the software level and is invisible from the perspective of the software engineers.

Despite their current limitations, multi-core systems are growing in popularity and new programming languages and techniques exist that can better utilize their parallelization. As the trend continues, the extensible architecture of eMIPS and software parallelization will come into position to complement each other in an extensible multi-core microprocessor. Given the flexibility extensions provide to the extensible microprocessor architecture, two flavors of extensible multi-core microprocessors may be considered. One is a multi-core processor using extension that includes multiple instances of the extensible microprocessor data path, each with its own extension slots. The other is a combination of static and dynamic processor data paths. The static data paths would resemble the TISA and interact like other multi-core systems. The dynamic data paths are implemented as extensions that may be loaded or disabled as the load on the microprocessor requires it to take on heavier loads or reduce power on lighter loads. These extension data paths may resemble the TISA or be custom data paths optimized

for a given application. Both flavors make it possible to realize parallel data paths optimized for different applications all running on the systems. To further optimize execution of high parallelized applications the OS may configure a sufficient number of available core slots with the optimized data paths. Then the OS would schedule the threads to run on the core that best fits the needs of that thread.

7.4 FROM eMIPS TO eARM, ePOWERPC, ETC

The eMIPS prototype of the extensible microprocessor architecture implements the MIPS R4000 as its TISA. In Section 2.2, we presented our rational for selecting this ISA as the basis for our prototype. This decision was largely based on the availability of documentation on the MIPS ISA and the lack of proprietary restrictions on its use. In the embedded microprocessor market, microprocessors based on other ISAs are increasingly popular. The most popular of these ISAs include ARM and PowerPC. Both these ISAs are argued by many to be superior architectures to MIPS. For the extensible microprocessor to gain wider acceptance, it will require an implementation of the architecture using one or both of these popular ISAs.

In addition to other ISAs, there are other microprocessor features omitted from this eMIPS prototype in order to keep the implementation simple. Some of these features are highly complex in their implementation and were beyond the scope this thesis. To develop an eMIPS prototype more quickly, it was decided that omitting such features was best. Some features such as branch prediction and speculative execution have been shown to improve performance of established RISC microprocessor architectures.

Including these features in an extensible microprocessor and addressing the issues raised by these features in an extensible platform would provide new avenues for research.

8 RELATED WORK

The concept of using reconfigurable logic to improve application performance is certainly not new [19] but to date not enough progress has been made towards an actual implementation of this and related concepts in a complete, usable and safe multi-user system.

One difficult point is addressing the security risk posed by the potentially tamper able FPGA execution engine [28]. Bossuet et al. [9] looked at FPGA security in the sense of securing the reconfiguration configuration file and protecting the IP contained therein. This is a good contribution, but only solves one aspect of the problem. There are a number of FPGA-based "accelerator" products [43][38][47] that restrict the use of the accelerator to a single process. This conservative approach still fails to secure other users from a virus injected into the one process that uses the accelerator. Dales [17] simulates a system that can leverage the FPGA acceleration in a general purpose workstation environment, switching the device among many applications. The FPGA is interfaced as a co-processor; security issues are not really addressed. Many other projects have simulated similar systems [27][39][48][13][34], ours is the first attempt to actually build a FPGA-based extensible microprocessor and a workstation that is safe for general, multi-user application loads.

One way to classify the various designs is in the way they interface the configurable logic to the main processor. Some use a memory mapped interface or FIFO,

most likely over an I/O bus [43][38][47][58], some use separate co-processors and explicit communication [58][33][39][27][13][35][26] others implicitly communicate using hints in regular (branch) instructions [14]. In eMIPS the programmable logic plugs directly into the pipeline and is accessed by explicit, per-application instructions.

Razdan and Smith [39] designed and simulated the PRISC system, the first system to use the idea of augmenting the pipeline of a MIPS processor with special instructions that are actually executed (on a per-process basis) in a reconfigurable logic array, the Programmable Function Unit, or PFU. They did not consider letting the PFU stall the pipeline, or access memory. They envisioned using the compiler to leverage the new, generic instructions but actually just patched binary objects in their experiments. The required system support was not addressed and PRISC was never physically realized. Garp [27] was also not realized; it improved on the PRISC design by considering system support, albeit in the expensive form of swapping the content of the entire logic array in and out to main memory at context switch time. The logic array was controlled using a clock counter to enable/disable its clock and to synchronize with the main processor's instruction stream. This results in a heterogeneous multiprocessor of sorts that requires both sophisticated compiler support and parallel programming expertise. The security threat of a direct path to memory was not considered but it does permit (physically addressed!) load/store operations that most other designs cannot handle. Borgatti et al. [8] have recently realized a system similar to Garp, using a mixture of ASIC and embedded FPGAs. This system is reminiscent of the eMIPS if we

map the ASIC component to the TISA and the FPGA to the extension slots. Unlike eMIPS though, the interface between FPGA and data path is limited to stopping the clock to the ASIC module when the slower FPGA needs more time. There is no access to the register file, memory accesses are only to a local buffer, and there is no MMU and no consideration to multi-user safety. Borgatti's work does study the practical problems that arise from integrating the slower and larger FPGAs into a 90 nm ASIC process, but the actual prototype chip only runs at about 100 MHz like our ML401 board.

Lysecky et al. [34] imagine a processor that is capable of auto-optimizing itself. Much like eMIPS, the Warp processor would optimize the frequently executed blocks of code by keeping frequency counts to detect the most frequently executed ones, automatically generate custom logic for their direct execution through micro-CAD on-chip tools, and patch the binaries of the executing programs to execute faster. While certainly ambitious and rather impractical, this approach does not address issues that are important in a practical system, such as security, multiprogramming and virtual memory support.

Clark et al. [14] propose using a reconfigurable dataflow engine as a co-processor that semi-transparently augments a regular RISC processor. This approach uses a hardware accelerator that identifies execution patterns based on execution tree compression and some compiler help. Using this pattern recognition, the accelerator controller configures the reconfigurable dataflow engine to realize the pattern in a fixed number of cycles rather than in the data path. This approach falls short of this work in three respects. In the first place, it requires considerable modification of the software

compiler to recognize the candidate code fragments and generate basic blocks that are recognizable by the runtime engine. The eMIPS does not require any change in the software tools and processes. In the second place, the dataflow engine has limited depth and applicability and this limits the performance benefits achievable with this approach. With eMIPS the pipelined can be stalled and all blocks are accessible, including memory accesses. In the third place, the approach was only tested using a modified SimpleScalar simulator [12] and did not result in a practically usable prototype. We intend for our prototype to be freely available to the research community for full evaluation and modification, thereby allowing the practical testing of this and other approaches.

Athanas and Silverman [7] did produce a prototype, the PRISM I. This approach also focuses attention on the compiler, specifically a C compiler that produces both software and hardware binaries targeted to the platform. According to [7] there are several limitations in the implementation that do not apply to this work. These include the inability of the PRISM I to support global variables in its hardware Extensions, exit conditions for for-loops must be determined in advance, not all of the C functions have been implemented and floating point operations are not supported. These limitations are now largely addressed by more recent systems that use a similar architecture and a similar software approach. For instance, Altera Corp. C2H compiler [33] targets an FPGA with a hard-core CPU and lets the user easily partition a C application in functions running on one or the other engine. The eMIPS processor provides transparency to users at the software level and uses a deeper coupling between custom logic and the data path. Any MIPS compiler can be used for the eMIPS, for any

programming language. Similar considerations apply to the many other C-to-gates flows [24] and, at least in part, to Anderson's HThreads hybrid system [5].

Sawitzki et al. [41] realized CoMPARE, a simple and cheap 8-bit extensible processor similar to Davidson's RM [18]. The limitations of these practical systems illustrate very well the gap between simulation and reality in this field of research.

Chow et al. [48][13][31] introduce OneChip, motivating the need for a close coupling of the data path and reconfigurable logic. While the basic idea is similar to those explored in eMIPS, the three different implementations of OneChip differ each in its own way. OneChip-96 [48] is a small subset that does not provide reconfiguration other than at boot time; processor and Extensions are literally compiled together. Interestingly, one of the two examples provided is for a peripheral, a USART. OneChip-98 [31] uses a dedicated path to memory like Garp and suffers from the same memory coherency and security problems. The instruction encoding is now fixed and based on the notion of copying data in and out of the FPGA array; similar to the PRISM I based systems. One Extension op code is reserved for FPGA use; four additional bits select the specific Extension. The actual implementation is very constrained and does not provide dynamic reconfiguration or memory coherence. There is no system software, or interrupts of any type. Two test programs demonstrate 10x-32x speedups. OneChip-01 [13] does away with an actual implementation and is simulated using SimpleScalar.

The Xtensa architecture [40][44] has similarities with eMIPS and two important differences. In the first place, Xtensa processors are realized as ASIC based on the customer's application requirements. They are statically extensible processors and

are therefore subject to the limitations previously illustrated for a classic RISC processor. In the second place, the suggested approach is to identify via profiling and simulation new additional instructions that are described (as GCC's RTL patterns) to the automated compiler generation system. The new compiler is then used to recompile the application program. We favor instead leveraging the predictable nature of the compiler's working, which manifests itself in repeated patterns of instructions (basic blocks). We optimize the basic blocks at the binary level, on a per application binary basis. This does not preclude leveraging the compiler but it does not mandate it, either.

In this paper we pessimistically assume that Extensions are manually generated and we consider automatic synthesis an orthogonal problem. Some complementary efforts are nonetheless worth mentioning.

Yehia [61] describes a semi-automated approach for realizing sequences of dependent instructions using combinatorial logic directly rather than some form of dataflow graph. These are then added to a superscalar processor and evaluated by simulation against a set of benchmarks. The rePLay tool [20] automatically generates the logic. The best result is a speedup of 40% over baseline in the Spec2000 benchmarks. This approach cannot handle load/store instructions, which limits the size of the blocks optimized. Faruque [1] looks at the problem of automatically recognizing patterns of instructions that can benefit the application performance if realized directly as ASIP instructions. Bracy et al. [10] look at the problem of generating mini-graphs, small coupling of instructions that can be tightly integrated into the micro-architecture of a super-scalar microprocessor. Mini-graphs are limited to two register inputs, one output,

one memory operation and one branch. Mini-graphs are automatically generated from application profiling. Over a large set of simulated benchmarks this approach leads to a peak gain of 40% over a baseline processor. Sun [45] attacks the problem of automatically generating a complete multiprocessor system, built out of ASIPs, that optimally executes a fixed set of applications.

Brisk et al. [11] describe a technique for collapsing multiple specialized instructions into a single data path, thereby reducing the area requirement by as much as 83%. Hauck et al. with Chimaera [26] and Totem [25] look at the possibility of designing the reconfigurable logic array that is attached to the main processor in ways that are more amenable to realizing domain-specific operations, in a fully automated way.

Extensible processors are not to be confused with micro-programming, WLIW processors or regular co-processors. A micro-programmed processor uses a fixed set of components (ALUs, memory busses, registers etc) and interconnects them directly with the micro-instructions. The eMIPS can use arbitrary logic in its Extensions, down to a single AND gate. WLIW processors are a restricted form of micro-programming, in a way, and therefore dissimilar to the eMIPS approach. The co-processor approach differs because it is implemented in fixed logic, it requires compiler support, and cannot plug into the data path but operates entirely separately from it.

The eMIPS approach is evolutionary, not revolutionary and differs from attempts to fundamentally redefine the core execution engine. For instance, Goldstein et al. [23] designed and Schmit et al. implemented [42] PipeRench, a new reconfigurable

architecture that uses a (virtual) reconfigurable pipeline to accelerate computationally intensive kernels.

9 CONCLUSION

We propose to use a dynamically extensible microprocessor architecture to address the inflexibility, sub-optimality, lack of performance growth and waste of area and power of a traditional, fixed RISC architecture. We have designed and implemented a prototype of the proposed architecture, the eMIPS microprocessor. eMIPS augments a core MIPS data path with dynamically loaded extensions that can plug into the individual stages of the pipeline. We have realized both a flexible simulation system and an FPGA implementation of eMIPS. We have demonstrated the use of extensions to transparently improve the performance of a set of applications by identifying candidate patterns of instructions (basic blocks), realizing an equivalent extension directly in fixed logic and automatically patching the binaries of the applications to use the extension instructions.

Despite the limitations of the first prototype we have shown that the approach is indeed flexible. The core data path only needs to implement the set of instructions that provides for self-extension and to manage the security sensitive resources. Anything else can be an extension, including multiplication and division, floating point and other co-processor based instruction sets, on-chip peripherals and eventually even multiple cores. For closed systems, a microprocessor can be fully optimized to include only the resources actually needed by the applications. This includes the instruction set, peripherals and area that are required and nothing else. Further extensions can still be

added later when/if the application requirements change, even after deployment in the field. A number of applications demonstrate speedups of 2x-3x simply by optimizing the top-three basic block patterns. Our tests have used video games, real-time programs, and the SPEC2000 integer benchmarks. This proves that a dynamically extensible microprocessor can easily outperform a traditional one that implements the same ISA.

Indications point to the available on-chip area as the limiting factor for this approach, not the clock. The basic data path can be implemented in as little as 50% of the resources of a Xilinx XC4LX25 device, leaving the majority of the device free for the extensions. An extension can take as little as 10% of the area and still provide a factor of 2x-3x speedup in a video game and in an embedded OS test set. On the negative side, the set of on-chip peripherals required to realize a complete eMIPS Workstation using the Xilinx ML401 development board is rather large. Even the minimal usable set requires 25% of the area resources, leaving only 25% of the chip for all other extensions. In this area we can conceivably place one additional core, but certainly not the “tens or hundreds” that have been speculated elsewhere.

REFERENCES

[1] M. Al Faruque. *A Fine Grained Application Profiling for Guiding Application Specific Instruction Set Processor (ASIPs) Design.* Master's Thesis. Aachen, Germany: Aachen University, 2004.

[2] Altera Corporation, *Home*, http://www.altera.com/index.jsp. 2006

[3] Altera Corporation, *Excalibur Embedded Processor Solutions*, 2005. Available at http://www.altera.com/products/devices/excalibur/excindex.html,

[4] Altera Corporation, *Stratix III Device Family*, 2006. Available At http://www.altera.com/products/devices/stratix3/ st3-index.jsp?WT.mc_id=s0_sm_go_xx_tx_1_491&WT.srch=1

[5] E. Anderson, J. Agron, W. Peck, J. Stevens, F. Baijot, E. Komp, R. Sass and D. Andrews, "Enabling a Uniform Programming Model across the Software/Hardware Boundary," *Proc. 14th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM) '06,* pp. 89-98. 2006.

[6] Atmel Corporation, *Atmel AT91EB63 Evaluation Board User Guide*, 2001. Available at http://www.atmel.com/dyn/resources/prod_documents/DOC1359.PDF#search=%22AT91EB63%20Evaluation%20Board%20User%20Guide%22

[7] P. Athanas and H. Silverman, "Processor Reconfiguration Through Instruction-Set Metamorphosis," *Computer,* vol. 26, no. 3, pp. 11-18, 1993.

[8] M. Borgatti, F Lertora, B. Foret and L. Cali, "A Reconfigurable System Featuring Dynamically Extensible Embedded Microprocessor, FPGA, and Customizable I/O," *IEEE Journal of Solid-State Circuits*, vol. 38, no. 3, pp. 521-529, 2003.

[9] L. Bossuet, G. Gogniat and W. Burleson, "Dynamically Configurable Security for SRAM FPGA Bitstreams," *Proc. 18th International Parallel and Distributed Processing Symposium,* pp. 146-154, 2004.

[10] A. Bracy, P. Prahlad and A. Roth, "Dataflow Mini-Graphs: Amplifying Superscalar Capacity and Bandwidth," *Proc. 37th IEEE/ACM International Symposium on Microarchitecture (MICRO) '04*, pp. 18-29, 2004.

[11] P. Brisk, A. Kaplan and M. Sarrafzadeh, "Area-Efficient Instruction Set Synthesis for Reconfigurable System-on-Chip Designs," *Proc. 41st Design Automation Conference (DAC) '04*, pp. 395-400, 2004.

[12] D. Burger and T. M. Austin, "The SimpleScalar Tool Set, version 2.0," *SIGARCH Computer Architecture News*, vol. 25, no. 3, pp. 13-25, 1997.

[13] J. E. Carrillo and P. Chow, "The Effect of Reconfigurable Units in Superscalar Processors." *Proc. ACM/SIGDA 9th International Symposium on Field Programmable Gate Arrays,* pp. 141-150, Feb. 2001.

[14] N. Clark, J. Blome, M. Chu, S. Mahlke, S. Biles and K. Flautner, "An Architecture Framework for Transparent Instruction Set Customization in Embedded Processors," *Proc. 32nd International Symposium on Computer Architecture (ISCA) '05*, pp. 272-283, 2005.

[15] N. Clark, H. Zhong and S. Mahlke, "Processor Acceleration Through Automated Instruction Set Customization," *Proc. 36th IEEE/ACM International Symposium on Microarchitecture (MICRO) '03*, pp. 129, 2003.

[16] N. Clark, M Kudlur, H. Park, S. Mahlke and K. Flautner, "Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set

Customization," *Proc. 37th IEEE/ACM International Symposium on Microarchitecture (MICRO) '04,* pp. 30-40, 2004.

[17] M. Dales, "Managing a Reconfigurable Processor in a General Purpose Workstation Environment," *Proc ACM/IEEE Design Automation and Test Europe (DATE) '03,* vol. 1, pp. 10980, 2003.

[18] J. Davidson, "FPGA Implementation of a Reconfigurable Microprocessor," *Proc. IEEE Custom Integrated Circuits Conference (CICC) '93,* pp. 3.2.1-3.2.4, 1993.

[19] G. Estrin, "Organization of Computer Systems: The Fixed Plus Variable Structure Computer," *Proc. Western Joint Computer Conference,* pp. 33-40, 1960.

[20] B. Fahs, S. Bose, M. Crum, B. Slechta, F. Spadini, T. Tung, S. Patel and S. Lumetta¸"Performance Characterization of a Hardware Framework for Dynamic Optimization," *Proc. 34th ACM/IEEE International Symposium on Microarchitecture (MICRO) `01,* pp. 16-27, 2001.

[21] A. Forin, N. L. Lynch and R. N. Pittman, *Software Support for Dynamically Extensible Processors*, Microsoft Research Technical Report MSR-TR-2006-147, Redmond WA, 2006.

[22] A. Forin, B. Neekzad and N. L. Lynch, *Giano: The Two-Headed Simulator*, Microsoft Research Technical Report MSR-TR-2006-130, Redmond WA, 2006.

[23] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe and R. R. Taylor, "PipeRench: A Reconfigurable Architecture and Compiler," *IEEE Computer,* vol. 33, no 4, pp. 70-77, 2000.

[24] S. Hauck and A. Agarwal, *Software Technologies for Reconfigurable Systems*. Northwestern University Technical Report, Evanston IL, 1996.

[25] S. Hauck, K. Compton, K. Eguro, M. Holland, S. Phillips and A. Sharma, "Totem: Domain-Specific Reconfigurable Logic," submitted to *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* 2006. Available at http://www.ee.washington.edu/faculty/hauck/publications/TotemSummaryJ.pdf

[26] S. Hauck, T. W. Fry, M. M. Hosler and J. P. Kao, "The Chimaera Reconfigurable Functional Unit," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems '04*, vol. 12, no. 2, pp. 206-217, 2004.

[27] J. R. Hauser and J. Wawrzynek, "Garp: A MIPS Processor with a Reconfigurable Coprocessor," *Proc. 5th IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 12-21, 1997.

[28] I. Hadžić, S. Udani and J. M. Smith, "FPGA Viruses," *Proc. 9th International Workshop on Field-Programmable Logic and Applications*, pp. 291-300, 1999.

[29] J. Helander and A. Forin. "MMLite: A Highly Componentized System Architecture," *Proc. 8th ACM SIGOPS European Workshop on Support of Composing Distributed Applications,* pp. 96-103, 1998.
Download software at http://research.microsoft.com/invisible/

[30] J. L. Hennessy and D. A. Patterson, *Computer Organization and Design: The Hardware/Software Interface*, San Francisco, CA.: Morgan Kaufmann Publishers, 1998.

[31] J. A. Jacob and P. Chow, "Memory Interfacing and Instruction Specification for Reconfigurable Processors," *Proc. ACM/SIGDA 7th International Symposium on Field Programmable Gate Arrays*, pp. 145-154, 1999.

[32] G. Kane and J. Heinrich, *MIPS RISC Architecture*, Upper Saddle River, NJ.: Prentice Hall, 1992.

[33] D. Lau, O. Pritchard and P. Molson, "Automated Generation of Hardware Accelerators with Direct Memory Access from ANSI/ISO Standard C Functions," *Proc. 14th IEEE Symposium on FPGAs for Custom Computing Machines (FCCM) '06*, pp. 45-54, 2006.

[34] R. Lysecky, G. Stitt and F. Vahid, "Warp Processors," *Proc. 41st Design Automation Conference (DAC) '06*, pp. 659-681, 2006.

[35] R. Lysecky and F. Vahi, "A Configurable Logic Architecture for Dynamic Hardware/Software Partitioning," *Proc. ACM/IEEE Design Automation and Test Europe (DATE) '04*, vol. 1, pp. 10480, 2004.

[36] Mentor Graphics Corporation, *Mentor Graphics ModelSim*.
Available At http://www.mentor.com/products/fpga_pld/simulation/index.cfm

[37] Microsoft Corporation, *Microsoft Giano*.
Available At http://research.microsoft.com/downloads/ and
http://www.ece.umd.edu/~behnam/giano.html

[38] Mitrionics, Incorporated, *Home*, http://www.mitrionics.com 2001.

[39] R. Razdan and M. D. Smith, "High-Performance Microarchitectures with Hardware-Programmable Functional Units," *Proc. 27th ACM/IEEE International Symposium on Microarchitecture (MICRO) `94,* pp. 172-180, 1994.

[40] C. Rowen and D. Maydan, "Automated Processor Generation for System-on-Chip," *Proc. 27th European Solid-State Circuits Conference (ESSCIRC) '01*, pp. 464-496, 2001.

[41] S. Sawitzki, S. Köhler and R. Spallek, "Prototyping Framework for Reconfigurable Processors," *Proc. 11th International Conference on Field Programmable Logic and Applications (FPL) '01*, pp. 6-16, 2001

[42] H. Schmit, D. Whelihan, A. Tsai, M. Moe, B. Levine and R. Taylor, "PipeRench: A Virtualized Programmable Data path in 0.18 Micron Technology," *Proc. IEEE Custom Integrated Circuits Conference (CICC) '02*, pp.63-66, 2002.

[43] SRC Computers Incorporated, *Home*, http://www.srccomp.com. 1996.

[44] Stretch, Incorporated, *Home*, http://www.stretchinc.com. 2006.

[45] F. Sun, S. Ravi, A. Raghunathan and N.K. Jha, "Synthesis of Custom Processors Based on Extensible Platforms," *IEEE/ACM International Conference on Computer Aided Design (ICCAD) '02*, pp. 641-648, 2002.

[46] S. Sutherland, *The Verilog PLI Handbook*, 2nd ed, Norwell, MA.:Kluwer Academic Publishers, 2002.

[47] Tarari, Incorporated, *Home*, http://www.tarari.com. 2002.

[48] R. D. Wittig and P. Chow, "OneChip: An FPGA Processor with Reconfigurable Logic," *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 126-135, 1996.

[49] Xilinx, Incorporated, *Home*, http://www.xilinx.com. 2005

[50] Xilinx, Incorporated, *Xilinx Chipscope Pro Software and Cores User Guide*, October 2005. Available At http://www.xilinx.com/ise/verification/ chipscope_pro_sw_cores_8_1i_ug029.pdf

[51] Xilinx, Incorporated, *Xilinx Virtex-4 Development Boards*, 2005. Available At http://www.xilinx.com/products/silicon_solutions/fpgas/virtex/ virtex4/index.htm

[52] Xilinx, Incorporated, “Chapter 5, Partial Reconfiguration,” *Xilinx Development System Reference Guide,* pp. 113-140, 2005.
Available At http://toolbox.xilinx.com/docsan/xilinx8/books/docs/dev/dev.pdf

[53] Xilinx, Incorporated, *Xilinx FPGA Editor Guide*, 1999.
Available At http://www.xilinx.com/support/sw_manuals/2_1i/download/fpedit.pdf

[54] Xilinx, Incorporated, *Xilinx System ACE Compact Flash Solution*, 2002.
Available At http://www.xilinx.com/bvdocs/publications/ds080.pdf

[55] Xilinx, Incorporated, *Xilinx Two Flows for Partial Reconfiguration: Module Based or Difference Based*, 2003.
Available At http://www.xilinx.com/bvdocs/appnotes/xapp290.pdf

[56] Xilinx, Incorporated, Xilinx *Using Partial Reconfiguration to Time Share Device Resources in Virtex II and Virtex II Pro*, 2005.

[57] Xilinx, Incorporated, *Xilinx Virtex 4 Configuration Guide*, 2006.
Available At http://direct.xilinx.com/bvdocs/userguides/ug071.pdf

[58] Xilinx, Incorporated, *Xilinx Virtex 4 Family Overview*, 2005. Available At http://direct.xilinx.com/bvdocs/publications/ds112.pdf

[59] Xilinx, Incorporated, *Xilinx Virtex 4 Packaging and Pin out Specification*, 2005. Available At http://direct.xilinx.com/bvdocs/userguides/ug075.pdf

[60] Xilinx, Incorporated, *Xilinx Virtex 4 User Guide*, 2005. Available At http://direct.xilinx.com/bvdocs/userguides/ug070.pdf

[61] S. Yehia and O. Teman, "From Sequences of Dependent Instructions to Functions: An Approach for Improving Performance without ILP or Speculation," *Proc. 31st International Symposium on Computer Architecture (ISCA) '04*, pp. 238-249, 2004.

[62] P. Yu and T. Mitra, "Characterizing Embedded Applications for Instruction-Set Extensible Processors," *Proc. 41st Design Automation Conference (DAC) '04*, pp. 723-728, 2004.

OTHER CONSULTED SOURCES

A. A. Aggarwal and D. M. Lewis, "Routing Architectures for Hierarchical Field Programmable Gate Arrays," *Proc. IEEE International Conference on Computer Design (ICCD) '94*, pp. 475 –478, 1994.

Atmel Corporation, *Atmel ARM Thumb Microcontrollers: AT91M63200*. 1999. Available at http://www.atmel.com/dyn/resources/prod_documents/DOC1028.PDF

Atmel Corporation, *FPSLIC (AVR with FPGA)*, 2005. Available at: http://www.atmel.com/products/FPSLIC/.

N. G. Bartzoudis, A. G. Fragkiadakis, D. J. Parish, J. L. Nunez and J. M. Sandford, "Reconfigurable Computing and Active Networks," *Proc. International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA) '03*, pp. 27-33, 2003.

V. Betz, *Architecture and CAD for the Speed and Area Optimization of FPGAs*, Ph.D. Dissertation, Toronto:University of Toronto, 1998.

J. Becker and M. Glesner, "A Parallel Dynamically Reconfigurable Architecture Designed for Flexible Application-Tailored Hardware/Software Systems in Future Mobile Communication," *The Journal of Supercomputing*, vol. 19, no. 1, pp. 105-127, 2001.

Berkeley Design Technology, Inc., *Home*, 2004.
Available at: http://www.bdti.com/articles/info_eet0207 fpga.htm#DSPEnhanced%20FPGAs.

P. Biswas, S. Banerjee, N. Dutt, P. Ienne and L. Pozzi, "Performance and Energy Benefits of Instruction Set Extensions in an FPGA Soft Core," *Proc. of the 19th International Conference on VLSI Design (VLSID) '06*, pp. 651-656, 2006.

W. Böhm, J. Hammes, B. Draper, M. Chawathe, C. Ross, R. Rinker and W. Najjar, "Mapping a Single Assignment Programming Language to Reconfigurable Systems," *The Journal of Supercomputing*, vol. 21, no. 2, pp. 117-130, 2002.

L. Cali, F. Lertora, C. Tazzina, M. Besana and M. Borgatti, "Platform IC with Embedded Via Programmable Logic for Fast Customization," *Proc. IEEE Custom Integrated Circuits Conference (CICC) '04*, pp. 419-422, 2004.

W. Chen, P. Kosmas, M. Leeser and C. Rappaport, "An FPGA Implementation of the Two-Dimensional Finite-Difference Time-Domain (FDTD) Algorithm," *Proc. ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, pp. 213-222, 2004.

J. Cong, Y. Fan, G. Han, A. Jagannathan, G. Reinman and Z. Zhang, "Instruction Set Extension with Shadow Registers for Configurable Processors," *Proc. ACM/SIGDA 13th International Symposium on Field Programmable Gate Arrays*, pp. 99-106, 2005.

J. Cong, Y. Fan, G. Han and Z. Zhang, "Application-Specific Instruction Generation for Configurable Processor Architectures," *Proc. ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, pp. 183-189, 2004.

Critical Blue, *Home*, http://www.criticalblue.com, 2005.

J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl and G. Chrysos, "ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors," *Proc. 30th IEEE/ACM International Symposium on Microarchitecture (MICRO) '97*, pp. 292-302, 1997.

A. DeHon, “DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century,” *Proc. IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 31-39, 1994.

R. Ernst, J. Henkel and T. Benner, “Hardware-Software Co synthesis for Microcontrollers,” *IEEE Design & Test of Computers*, vol. 10, no. 4, pp. 64-75, 1993.

W. Fu and K. Compton, “An Execution Environment for Reconfigurable Computing,” *Proc. 13th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM) ’05*, pp. 149-158, 2005.

M. Gokhale and J. Stone, “NAPA C: Compiling for Hybrid RISC/FPGA Architectures,” *Proc. IEEE Symposium on FPGAs for Custom Computing Machines*, pp.126, 1998.

A. Gordon-Ross and F. Vahid, “Frequent Loop Detection Using Efficient Non-Intrusive On-Chip Hardware,” *Proc. International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES) ’03*, pp. 117-124, 2003.

S. L. Graham, P. B. Kessler and M. K. McKusick, “gprof: a Call Graph Execution Profiler,” *Proc. SIGPLAN Symposium on Compiler Construction*, pp. 120-126, 1982.

Z. Guo, B. Buyukkurt, W. Najjar and K. Vissers, “Optimized Generation of Data-Path from C Codes,” *Proc. ACM/IEEE Design Automation and Test Europe (DATE) ’05*, vol. 1, pp. 112-117, 2005.

Z. Guo, W. Najjar, F. Vahid and K. Vissers, “A Quantitative Analysis of the Speedup Factors of FPGAs Over Processors,” *Proc. ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, pp. 162-170, 2004.

S. Hauck, “The Roles of FPGAs in Reprogrammable Systems,” *Proc. of the IEEE*, vol. 86, no. 4, pp. 615-638, 1998.

J. Keane, C. Bradley and C. Ebeling, “A Compiled Accelerator for Biological Cell Signaling Simulations,” *Proc. ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, pp. 233-241, 2004.

Y. Lai and P. Wang, "Hierarchical Interconnection Structures for Field Programmable Gate Arrays," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 5, no. 2, pp. 186 –196, 1997.

F. Lertora and M. Borgatti, "Handling Different Computational Granularity by a Reconfigurable IS Featuring Embedded FPGAs and a Network-On-Chip," *Proc. 13th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM) '05*, pp. 45-54, 2005.

W. Li and D. K. Banerji, "Routability Prediction for Hierarchical FPGAs," *Proc. 9th Great Lakes Symposium on VLSI*, pp. 256-259, 1999.

M. Nelson, "Fast String Searching With Suffix Trees," *Dr. Dobb's Journal*. August, 1996.

K. Sarrigeorgidis and J. M. Rabaey, "Massively Parallel Wireless Reconfigurable Processor Architecture and Programming," *Proc. International Parallel and Distributed Processing Symposium,* 2003. CD-ROM

M. Simat, S. Cotofana, J. T. J. van Eijndhoven, S. Vassiliadis and K. Vissers, "An 8x8 IDCT Implementation on an FPGA-Augmented TriMedia," *Proc.9th*

IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM) '01, pp. 160-169, 2001.

M. Simat, S. Cotofana, S. Vassiliadis, J. T. J. van Eijndhoven and K. Vissers, "MPEG-compliant entropy decoding on FPGA-augmented TriMedia/CPU64," *Proc. 10th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM) '02*, pp. 261- 270, 2002.

G. Stitt, F. Vahid, G. McGregor and B. Einloth, "Hardware/Software Partitioning of Software Binaries: A Case Study of H.264 Decode," *Proc. 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Co design and System Synthesis (CODES+ISSS)*, pp. 285-290, 2005.

Tensilica, Incorporated, *Home*, http://www.tensilica.com. 2006.

Triscend Corporation, *Home*, http://www.triscend.com, 2003.

G. Venkataramani, W. Najjar, F. Kurdahi, N. Bagherzadeh and W. Bohm, "A Compiler Framework for Mapping Applications to a Coarse-grained Reconfigurable Computer Architecture," *Proc. International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES) `01*, pp. 116-125, 2001.

M. Wan, H. Zhang, V. George, M. Benes A. Abnous V. Prabhu and J. M. Rabaey, "Design Methodology of a Low-Energy Reconfigurable Single-Chip DSP System," *Journal of VLSI Signal Processing Systems*, no. 28, pp. 47-61, 2001.

Xilinx, Incorporated, "Chapter 4, Modular Design," *Xilinx Development System Reference Guide*, pp. 75-112, 2005. Available At http://toolbox.xilinx.com/docsan/xilinx8/books/docs/dev/dev.pdf

Xilinx, Incorporated, *Xilinx Virtex 4 Datasheet: DC and Switching Characteristics*. 2006. Available At http://direct.xilinx.com/bvdocs/publications/ds302.pdf

S. Yehia, N. Clark, S. Mahlke and K. Flautner, "Exploring the Design Space of LUT-based Transparent Accelerators," *Proc. International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES) `05*, pp. 11-21, 2005.

M. Zagha, B. Larson, S. Turner and M. Itzkowitz, "Performance Analysis Using the MIPS R10000 Performance Counters," *Proc. ACM/IEEE Conference on Supercomputing*, article no. 6, Nov. 1996. CD-ROM

H. Zhang, M. Wan, V. George and J. Rabaey, "Interconnect Architecture Exploration for Low-Energy Reconfigurable Single-Chip DSPs," *Proc. IEEE Computer Society Workshop on VLSI '99*, pp. 2-8, 1999.

H. Zhang, V. Prabhu, V. George, M. Wan, M. Benes, A. Abnous and J. M. Rabaey, "A 1-V Heterogeneous Reconfigurable DSP IC for Wireless Baseband Digital Signal Processing," *IEEE Journal of Solid-State Circuits*, vol. 35, no. 11, pp. 1697-1704, 2000.

X. Zhang, Z. Wang, N. Gloy, J. B. Chen and M. D. Smith, "System Support for Automatic Profiling and Optimization," *Proc. 16th ACM Symposium on Operating Systems Principles*, pp. 15-26, 1997.

C. B. Zilles and G.S. Sohi, "A Programmable Co-processor for Profiling," *Proc. 7th International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 241-252, 2001.

VITA

Richard Neil Pittman is a member of the class of 2003. He received his Bachelor of Science degree in computer engineering from Texas A&M University in College Station, TX in 2004. He continued his education there after applying and being accepted to the Master of Science program. There he studied and conducted research in the area of computer engineering focusing on subjects including computer architectures for embedded systems and configurable computing. He has accepted a position as a Research Hardware Engineer at Microsoft Research in Redmond, WA. He plans to continue working in these subjects and others after his arrival.

Mr. Pittman may be reached at Microsoft Research on the Microsoft Campus at One Microsoft Way, Redmond, WA 98052-6399. His email is a-ricpit@microsoft.com.